

# *Implementação de Linguagens de Programação*

# *Implementação de Linguagens de Programação*

**Tomasz Kowalcowski**

*Instituto de Matemática, Estatística e  
Ciência da Computação  
Universidade Estadual de Campinas*

GUANABARA  
DOIS

CIP-Brasil. Catalogação-na-fonte  
Sindicato Nacional dos Editores de Livros, RJ.

K89i Kowaltowski, Tomasz.  
Implementação de linguagens de  
programação/Tomasz Kowaltowski.—  
Rio de Janeiro: Ed. Guanabara Dois, 1983.

Apêndices.  
Bibliografia.  
ISBN 85-7030-009-3

1. Linguagem de programação  
2. PASCAL (Linguagem de programação  
para computadores) I. Título.

82-0788 CDD — 001.642  
001.6424  
CDU — 800.92  
92PASCAL

*Aos meus pais.  
A Doris, Alicia e André.*

Direitos exclusivos para a língua portuguesa  
Copyright © by  
**EDITORIA GUANABARA DOIS S.A.**  
Rio de Janeiro — RJ

1983 — 10 9 8 7 6 5 4 3 2 1

Reservados todos os direitos. É proibida a duplicação  
ou reprodução deste volume, ou de partes do mesmo,  
sob quaisquer formas ou por quaisquer meios  
(eletrônico, mecânico, gravação, fotocópia, ou outros),  
sem permissão expressa da Editora.

Fotocomposição da Editora Guanabara Koogan S.A.

# Prefácio

Este livro foi escrito para ser seguido num curso dedicado à construção de compiladores. O seu nível corresponde ao fim de um programa de graduação, ou então, ao início de um programa de pós-graduação em ciência da computação. Os pré-requisitos exigidos são: experiência de programação de computadores em vários níveis (linguagens de máquina, de montagem e de alto nível) e conhecimento de técnicas de programação (manipulação de pilhas, árvores, tabelas, uso de recursão). É útil uma exposição prévia à programação de sistemas, como por exemplo a construção de um montador.

O material contido no texto pode ser coberto num curso de um semestre, especialmente se os alunos já estão familiarizados com métodos formais para descrição da sintaxe de linguagens e com noções básicas de análise sintática. No caso de um curso de pós-graduação, o texto deve ser complementado com leitura de outros livros e artigos. Os objetivos do curso não serão atingidos se ele não for concluído com a implementação de um compilador para a versão simplificada, mas bastante representativa, do Pascal, sugerida no texto. A implementação deverá ser efetuada numa linguagem de alto nível, como Pascal, Algol 60 ou PL/I.

O texto cobre, basicamente, os aspectos sintático e semântico da implementação de linguagens. Na parte sintática, é apresentado o material tradicionalmente incluído nos textos de compilação: uma introdução às linguagens livres de contexto e aos métodos de análise sintática correspondentes. Uma ênfase especial é dada ao aspecto semântico, que tem merecido menos atenção na literatura, mas que é uma parte fundamental da implementação. Os dois aspectos são integrados, levando à construção de um compilador de porte razoável, mas ainda viável para ser realizada durante um curso.

O texto não tem a pretensão de ser um livro de referência, deixando de tratar vários assuntos relevantes. O leitor interessado em aprofundar-se mais nesta área deverá consultar a bibliografia sugerida no fim de cada capítulo.

Um problema enfrentado ao elaborar o livro foi o da terminologia. Tentou-se, sempre que possível, utilizar termos que já ganharam aceitação em outros textos escritos em português. Foi necessário, entretanto, introduzir vários termos novos, equivalentes aos utilizados em inglês. Procuraram-se, nestes casos, traduções adequadas, e não necessariamente literais.

Este livro resultou da experiência de ensino acumulada pelo autor durante vários anos, ao ministrar cursos na Universidade Estadual de Campinas (UNICAMP), Universidade de São Paulo (USP) e Universidade da Califórnia em Santa Bárbara (UCSB). A versão original foi publicada pela Primeira Escola de Computação, realizada em janeiro de 1979, em São Paulo. Essa versão original estava orientada para a linguagem Algol 60. A nova versão aqui apresentada resultou de uma revisão completa, incluindo a mudança para a linguagem Pascal.

Diversas instituições contribuíram, direta ou indiretamente, para a elaboração deste livro. Além das três universidades já citadas, o autor gostaria de agradecer o apoio da Universidade de Karlsruhe (Alemanha), onde foi redigida uma parte do texto original, e do Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) e da Gesellschaft für Mathematik und Datenverarbeitung (GMD), que possibilitaram a estada nesta última universidade. O CNPq patrocinou, parcialmente, a estada do autor na UCSB.

Agradecimentos pessoais são dirigidos, em primeiro lugar, aos muitos alunos que tiveram o trabalho de “depurar” as várias versões iniciais deste texto, bem como aos membros da Comissão Organizadora da Primeira Escola de Computação: Imre Simon, Paulo Augusto Veloso e Istvan Simon, que convenceram o autor a empreender a tarefa de escrever o livro. Finalmente, o autor agradece aos Srs. João Baptista Esteves de Oliveira, da USP, e Raul dos Santos, da UNICAMP, que pacientemente datilografaram os dois manuscritos que levaram ao texto aqui apresentado.

Campinas, SP  
Janeiro de 1983

Tomasz Kowaltowski

# Índice

## 1 Introdução, 1

- 1.1 Generalidades, 1
- 1.2 Aspectos Básicos de Compilação, 2
- 1.3 Notação, 5

## 2 Especificação de Linguagens de Programação, 6

- 2.1 Generalidades, 6
- 2.2 Definições Indutivas e Notação de Backus, 6
- 2.3 Gramáticas e Linguagens, 8
- 2.4 Árvores de Derivação. Ambigüidade. Derivações Canônicas, 11
- 2.5 Algumas Relações Úteis. Gramáticas Reduzidas, 16
- 2.6 Análise Sintática, 19
- Exercícios, 20
- Notas Bibliográficas, 21

## 3 Análise Sintática Ascendente, 22

- 3.1 Generalidades, 22
- 3.2 Análise de Precedência Simples, 25
- 3.3 Análise de Precedência de Operadores, 34
- 3.4 Análise  $LR(k)$ , 39
- Exercícios, 48
- Notas Bibliográficas, 50

## 4 Análise Sintática Descendente, 51

- 4.1 Generalidades, 51
- 4.2 Análise Descendente com Retrocesso, 51
- 4.3 Eliminação de Retrocessos e da Recursão Esquerda, 54

- 4.4 Implementação da Análise Descendente. Analisador Descendente Recursivo, 62
- 4.5 Diagramas Sintáticos, 66
- Exercícios, 69
- Notas Bibliográficas, 70

## 5 Descrição do Pascal Simplificado, 71

- 5.1 Generalidades, 71
- 5.2 Programas e Blocos, 72
- 5.3 Declarações, 72
- 5.4 Comandos, 73
- 5.5 Expressões, 73
- 5.6 Números e Identificadores, 74
- 5.7 Comentários, 74
- Exercícios, 74
- Notas Bibliográficas, 75

## 6 Análise Léxica, 76

- 6.1 Generalidades, 76
- 6.2 Implementação de Analisadores Léxicos, 78
- Exercícios, 80
- Notas Bibliográficas, 80

## 7 Diagramas de Execução, 81

- 7.1 Generalidades, 81
- 7.2 Estrutura de Programas, 81
- 7.3 Simulação da Execução, 82
- Exercícios, 92
- Notas Bibliográficas, 92

## 8 Sistema de Execução Básico para Pascal, 93

- 8.1 Generalidades, 93
- 8.2 Características Gerais da MEPA, 94
- 8.3 Avaliação de Expressões, 94
- 8.4 Comandos de Atribuição, 96
- 8.5 Comandos Condicionais e Iterativos, 98
- 8.6 Comandos de Entrada e de Saída, 102
- 8.7 Programas, 103
- 8.8 Procedimentos sem Parâmetros, 105
- 8.9 Passagem de Parâmetros por Valor, 114
- 8.10 Passagem de Parâmetros por Referência, 117

# Introdução

- 8.11 Funções, 125
- 8.12 Rótulos e Comandos de Desvio, 128
- Exercícios, 135
- Notas Bibliográficas, 137

## 9 Sistema de Execução: Extensões e Implementação, 139

- 9.1 Generalidades, 139
- 9.2 Passagem de Procedimentos e Funções como Parâmetros, 139
- 9.3 Passagem de Parâmetros por Nome, 150
- 9.4 Blocos com Declarações Locais, 153
- 9.5 Matrizes, 154
- 9.6 Implementação da MEPA, 157
- Exercícios, 159
- Notas Bibliográficas, 160

## 10 Organização do Compilador, 161

- 10.1 Generalidades, 161
- 10.2 Tabela de Símbolos, 161
- 10.3 Análise Sintática e Geração de Código, 165
- 10.4 Tratamento de Erros, 170
- Exercícios, 172
- Notas Bibliográficas, 172

*Apêndice 1 Sintaxe do Pascal Simplificado, 173*

*Apêndice 2 Diagramas Sintáticos do Pascal Simplificado, 176*

*Apêndice 3 Descrição da MEPA, 180*

*Bibliografia, 183*

*Índice Alfabético, 187*

### 1.1 GENERALIDADES

O advento de computadores eletrônicos tornou evidente, desde muito cedo, a existência de uma barreira de comunicação entre o homem e a máquina. Os computadores operam, em geral, num nível muito atômico, manipulando dígitos binários, registradores, posições de memória etc., enquanto pessoas preferem expressar-se usando línguas naturais, auxiliadas, quando necessário, pelo emprego de notação matemática. Desde muito cedo tentou-se superar essa barreira de comunicação com a introdução de linguagens de programação.

As primeiras linguagens introduzidas foram as chamadas *linguagens de montagem*,<sup>1</sup> que, apesar de facilitar a tarefa de programação, ainda eram muito próximas das linguagens de máquina. Um passo decisivo foi a introdução, na segunda metade da década de 1950, de *linguagens de alto nível*, tais como FORTRAN e Algol 60.<sup>2</sup> Desde então surgiram algumas centenas de linguagens de alto nível, algumas das quais tiveram utilização muito larga.

Com a introdução de linguagens de alto nível, surgiu a necessidade de programas *tradutores*, isto é, sistemas que traduzissem programas escritos por programadores — *programas-fonte*, para programas em linguagem de máquina — *programas-objeto*. Por motivos históricos, os tradutores para linguagens de montagem são chamados de *montadores*,<sup>3</sup> e os para linguagens de alto nível, de *compiladores*. É este último tipo de tradutores que será estudado neste texto.

Deve-se notar que existem outras soluções para a implementação de linguagens de alto nível. Uma possibilidade é o uso de *interpretadores*. Estes programas, em vez de traduzir de uma vez o programa-fonte para então executá-lo, decodificam unidades básicas do programa (por exemplo, comandos) para executá-los imediatamente. Este processo acarreta, em geral, a decodificação repetida de várias partes do programa-fonte e pode ser muito ineficiente. Na prática, uma parte da decodificação é feita antes de iniciar-se a interpretação, tornando o processo mais eficiente. Dependendo das características da linguagem, a interpretação pode ser a melhor maneira de implementá-la.

Na realidade, não existe uma linha divisória muito clara entre a interpretação e a compilação. Mesmo numa linguagem compilada, certas partes do programa podem ser interpretadas durante a execução. Como exemplos, temos às vezes as

<sup>1</sup>Em inglês: *assembly languages*.

<sup>2</sup>A primeira versão da linguagem Algol foi apresentada em 1958.

<sup>3</sup>Em inglês: *assemblers*.

operações de entrada e saída e operações que não têm correspondência direta em linguagem de máquina.

## 1.2 ASPECTOS BÁSICOS DE COMPILAÇÃO

Existem três aspectos fundamentais na implementação de linguagens de programação. Os programas-fonte são representados, em geral, por seqüências de caracteres que indicam composição de várias construções permitidas pela linguagem. A identificação correta destas construções corresponde ao aspecto *sintático* da implementação. A cada construção da linguagem está associado um significado que deve estar refletido na tradução desta construção no programa-objeto; este é o aspecto *semântico* da implementação. Finalmente, há um aspecto que poderíamos chamar de *pragmático*, que corresponde ao problema de integração do compilador num sistema de computação hospedeiro.

O aspecto sintático da implementação de linguagens é certamente o que foi mais estudado e descrito. A razão principal para isto é a existência de uma formalização conveniente para descrever sintaxe de linguagens por meio de gramáticas livres de contexto e o consequente surgimento de uma teoria de análise sintática. Podemos afirmar, inclusive, que na prática os problemas sintáticos foram totalmente resolvidos.

Quanto ao aspecto semântico, existem algumas tentativas de formalização, mas que trouxeram poucas contribuições para a solução dos problemas práticos. Sob o ponto de vista semântico, há uma grande variação entre as linguagens de programação. Os problemas são simples e fáceis de resolver no caso de uma linguagem como FORTRAN, mas tornam-se complexos e com muitas soluções possíveis no caso de uma linguagem como SNOBOL.

Finalmente, o aspecto pragmático é o que apresenta mais variabilidade entre os diversos sistemas de computação. Há grandes diferenças na arquitetura básica dos sistemas, com linguagens de máquina totalmente incompatíveis. Os sistemas operacionais podem ter características muito distintas, exigindo soluções diferentes, especialmente no que se refere à manipulação de arquivos, tratamento de exceções, administração de memória. Os sistemas podem diferir quanto ao modo de operação — compartilhado,<sup>1</sup> contínuo,<sup>2</sup> em tempo real — impondo soluções distintas em cada caso. Por um lado, o aspecto pragmático é muito importante, pois pode absorver a maior parte do trabalho de implementação de uma linguagem. Por outro lado, os problemas conceituais envolvidos não são, em geral, muito complexos e estão mais relacionados com o conhecimento do sistema hospedeiro do que com a linguagem.

Neste texto, procuraremos dar uma ênfase relativamente maior ao aspecto semântico. O aspecto sintático será tratado de maneira mais superficial do que em outros textos, pois dispõe de uma literatura muito vasta, e existem soluções simples e bem compreendidas. O aspecto pragmático terá apenas um tratamento mínimo, necessário para atingir o objetivo final, que é levar à construção de um compilador para uma linguagem de alto nível razoavelmente complexa.

Contrariamente ao aspecto sintático, o aspecto semântico é mais difícil de ser tratado com generalidade, devido à inexistência de modelos adequados. Decidimos, portanto, escolher uma linguagem-exemplo na qual estará baseada a nossa discussão, adotando para tal um subconjunto da linguagem Pascal. Esta linguagem, definida por Wirth (1971a), ganhou uma grande aceitação não apenas como uma linguagem para ensino de programação, mas também para aplicações mais gerais. Além disto, ela teve impacto muito grande sobre o projeto de linguagens de programação posteriores. A linguagem apresenta características comumente en-

contradas em outras linguagens, sem apresentar, entretanto, muitas idiossincrasias. Serão discutidas, também, algumas extensões baseadas em Algol 60.

O funcionamento básico de um compilador está indicado de maneira esquemática na Figura 1.1. Os retângulos menores representam as várias *fases* do compilador, e as flechas indicam o fluxo de informação. Dependendo da implementação, certas fases podem ser executadas seqüencialmente, ou então várias fases podem ter a sua execução entrelaçada. Certas fases podem ser omitidas, como é o caso frequente das fases de otimização. Todas as fases consultam, em geral, tabelas do compilador. Algumas destas tabelas têm conteúdo fixo, como por exemplo a tabela de palavras reservadas e símbolos da linguagem. Outras, como a tabela de símbolos, são criadas a partir do próprio programa-fonte e utilizadas por várias fases.

As fases podem ser classificadas em dois grupos: as que participam da *análise* do programa-fonte e as que participam da *síntese* do programa-objeto.

A fase de *análise léxica* tem por objetivo transformar seqüências de caracteres usados na representação externa de programas em códigos internos; o Capítulo 6 do livro discute esta análise. A fase de *análise sintática* tem por finalidade revelar a estrutura do programa-fonte em termos das construções permitidas pela linguagem. Este assunto é coberto pelos Capítulos 2 a 4. A fase de *análise semântica* extrai da estrutura sintática as informações que serão necessárias para as fases de síntese. Na fase de *otimização global*, procura-se introduzir melhorias de programa que, em geral, independem da linguagem de máquina. Podemos citar, como exemplos, a eliminação de cálculos repetidos dentro de malhas de programa e a eliminação de subexpressões iguais. *Otimização local*<sup>1</sup> refere-se, em geral, a melhorias de código-objeto que aproveitam o repertório de instruções de um dado computador. Os problemas de otimização não serão tratados neste texto. Os Capítulos 7 a 10 cobrem os conceitos correspondentes às fases de análise semântica e de *geração de código*. Durante a análise do programa podem ser encontrados vários erros de programação. O tratamento destes erros é uma parte importante do compilador e é comentado no Capítulo 10.

Como já foi mencionado, as várias fases do compilador podem ser executadas em seqüência, ou então ter a sua execução combinada, induzindo os termos: compilação em *vários passos* e compilação em *um passo*. Numa compilação em vários passos, a execução de algumas fases termina antes de iniciar-se a execução das fases seguintes. Assim, um compilador de dois passos poderia combinar num primeiro passo as análises léxica e sintática, e num segundo passo, as outras fases. Neste caso, haveria necessidade de se transmitir os resultados do primeiro ao segundo passo, sob a forma de uma estrutura de dados conveniente, e que poderia ser a árvore sintática (veja Capítulo 2) ou outra estrutura semelhante. No caso de programas-fonte muito grandes, ou de sistemas de pouca capacidade de memória, esta *representação intermediária* poderia ser gravada num meio externo, aumentando, entretanto, o tempo de compilação.

Numa compilação de um passo, o programa-objeto é produzido à medida que o programa-fonte é processado, dispensando a representação intermediária. Normalmente, a compilação de um passo é mais eficiente, mas isto pode não ser verdadeiro em certos casos. Por exemplo, se o sistema disponível tem pouca memória, poderá ser necessário dividir o compilador, que é um programa bastante extenso, e usar a superposição de código,<sup>2</sup> que acarreta ineficiência. É mais difícil, também, introduzir as fases de otimização num compilador de um passo. Finalmente, certas linguagens possuem características que tornam inevitável a compilação em dois ou mais passos.

Suporemos, neste texto, que se deseja construir um compilador de um passo,

<sup>1</sup>Em inglês: *time-sharing*.

<sup>2</sup>Em inglês: *batch*.

<sup>1</sup>Em inglês: *peephole optimization*.

<sup>2</sup>Em inglês: *overlays*.

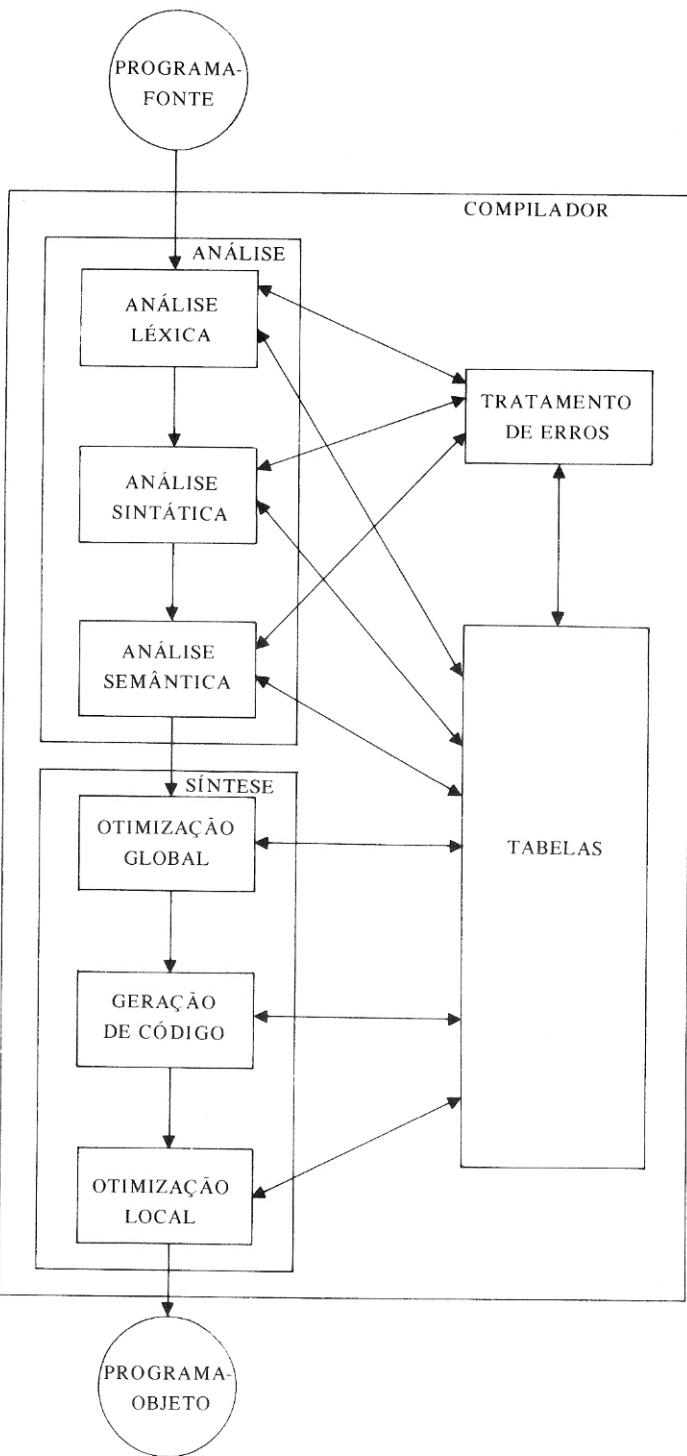


Fig. 1.1

com o código a ser gerado numa linguagem de montagem. Assim, haverá um passo adicional a ser executado pelo montador.

### 1.3 NOTAÇÃO

O material coberto nos Capítulos 2 a 4 do livro é de natureza mais formal e exige o uso de alguma notação matemática. Procuramos utilizar, sempre, a notação que é comumente empregada. Assim, o símbolo  $\in$  denota “pertence” a um conjunto; o símbolo  $\subseteq$  denota subconjuntos; os símbolos  $\cup$ ,  $\cap$ ,  $\cup\cup$  e  $\cap\cap$  denotam a união, a interseção, a união múltipla e a interseção múltipla de conjuntos, respectivamente; o símbolo  $\times$  denota o produto cartesiano, e  $-$  indica a diferença entre conjuntos. Os símbolos utilizados para denotar valores lógicos (booleanos) são *verdadeiro* e *falso*;  $\vee$ ,  $\wedge$ ,  $\vee\vee$  e  $\wedge\wedge$  indicam as operações de disjunção (“ou”), conjunção (“e”), disjunção múltipla e conjunção múltipla, respectivamente. Outras notações serão introduzidas no texto.

Os algoritmos serão representados neste livro de duas maneiras. Alguns serão apresentados informalmente, como seqüências de passos descritos em português, aumentados com notação matemática. Outros, mais semelhantes a verdadeiros programas, aparecem numa versão aportuguesada de uma linguagem de alto nível, semelhante ao Pascal, e que será facilmente compreendida pelo leitor.

Os exemplos de programas em Pascal seguirão a definição original, com seus símbolos escritos em inglês.

# Especificação de Linguagens de Programação

## 2.1 GENERALIDADES

Uma linguagem de programação deve ter uma definição precisa antes de ser implementada. Em geral, esta definição consiste de duas partes. Em primeiro lugar são especificadas todas as seqüências de símbolos que constituem programas válidos numa dada linguagem. São estes, e somente estes, os programas que deveriam ser aceitos e traduzidos pelo compilador. Esta parte da definição, que especifica a forma dos programas válidos, é chamada de especificação da *sintaxe* da linguagem. A segunda parte da definição consiste na especificação do significado associado a cada construção da linguagem, e é também chamada de especificação da *semântica* da linguagem. O código-objeto gerado pelo compilador para cada construção da linguagem está baseado nesta especificação.

Existem métodos formais para especificação completa tanto da sintaxe como da semântica de linguagens de programação. Entretanto, as descrições que usam os métodos atualmente existentes tendem, em geral, a ser muito complexas. Deve-se notar, porém, que seria muito útil a existência de métodos formais convenientes para descrever linguagens de programação, pois isto permitiria automatizar uma grande parte do desenvolvimento de um compilador. Na prática, usam-se métodos formais para especificar a maior parte dos aspectos sintáticos da linguagem, sendo que algumas restrições sintáticas adicionais, bem como a semântica, são especificadas por meios informais. Como veremos mais adiante, é mais fácil desenvolver as partes do compilador correspondentes aos aspectos da linguagem que são especificados formalmente. Este é o caso da análise sintática a ser estudada nos Capítulos 3 e 4. O restante deste capítulo será dedicado ao estudo de um método de especificação da sintaxe de linguagens.

## 2.2 DEFINIÇÕES INDUTIVAS E NOTAÇÃO DE BACKUS

Suponhamos que se deseja especificar a forma de todas as expressões aritméticas que usam os símbolos de variáveis  $a$  e  $b$ , os símbolos de operação  $+$  e  $*$ , e os parênteses ( $$  e  $)$ . Uma maneira de definir tais expressões seria através de uma definição indutiva da seguinte forma:

1.  $a$  e  $b$  são expressões;
2. se  $\alpha$  e  $\beta$  são seqüências de símbolos que são expressões, então as seqüências de símbolos  $\alpha+\beta$  e  $\alpha*\beta$  são expressões (neste contexto, a notação  $\alpha+\beta$  indica a justaposição dos símbolos de  $\alpha$  com o símbolo  $+$  e com os símbolos de  $\beta$ );
3. se  $\alpha$  é uma seqüência de símbolos que é uma expressão, então a seqüência  $(\alpha)$  é uma expressão;

4. expressões são todas e somente as seqüências de símbolos obtidas por uma aplicação das regras 1 a 3, um número finito de vezes.

Podemos tornar mais concisa esta definição, denotando com  $E$  o conjunto das seqüências de símbolos que são expressões:

1.  $a$  e  $b$  estão em  $E$ ;
2. se  $\alpha$  e  $\beta$  estão em  $E$ , então  $\alpha+\beta$  e  $\alpha*\beta$  estão em  $E$ ;
3. se  $\alpha$  está em  $E$ , então  $(\alpha)$  está em  $E$ .

A regra 4 fica subentendida. Note-se que para mostrar que a definição acima realmente caracteriza todas e somente as expressões válidas, teríamos que ter uma outra definição formal para expressões e demonstrar que as duas definições determinam o mesmo conjunto de seqüências de símbolos. Parece, porém, que qualquer outra definição seria mais complicada e menos intuitiva do que a dada acima.

Essa definição pode ser usada de duas maneiras. Em primeiro lugar, ela indica como construir seqüências de símbolos que são expressões e, em segundo lugar, ela indica como verificar se uma dada seqüência é uma expressão. Assim, temos a seguir os passos de construção da seqüência  $(a+b)*a$ , indicando entre chaves os números das regras aplicadas e das linhas anteriores que justificam a aplicação da regra.

- |              |                           |
|--------------|---------------------------|
| 1. $a$       | { regra 1}                |
| 2. $b$       | { regra 1}                |
| 3. $a+b$     | { regra 2, linhas 1 e 2 } |
| 4. $(a+b)$   | { regra 3, linha 3 }      |
| 5. $(a+b)*a$ | { regra 2, linhas 4 e 1 } |

Note-se que esta ordem não é a única possível, pois poderíamos trocar, por exemplo, a ordem dos passos 1 e 2.

As definições indutivas constituem o método mais comumente usado para a especificação da sintaxe. Devido a este fato existe uma notação especial, chamada *Forma Normal de Backus* ou *FNB*.<sup>1</sup> Nesta notação, a definição das expressões ficaria:

$$\begin{aligned} E &::= a \\ E &::= b \\ E &::= E+E \\ E &::= E*E \\ E &::= (E) \end{aligned}$$

A quarta linha, por exemplo, indica que uma expressão pode ser constituída de uma expressão, seguida do símbolo  $*$ , e seguida de outra expressão. Os símbolos  $a$ ,  $b$ ,  $+$ ,  $*$ ,  $($  e  $)$  são os únicos que podem entrar na formação de uma expressão;  $E$  é um símbolo auxiliar que facilita a especificação concisa das expressões. É comum agruparem-se as várias alternativas da definição, separando-as com barras verticais:

$$E ::= a \mid b \mid E+E \mid E*E \mid (E)$$

<sup>1</sup>Em inglês: *BNF* ou *Backus Normal Form*

## Exemplo 2.1

Seja a seguinte definição em FNB:

$$\begin{aligned} C &::= I \leftarrow E \mid C; C \\ E &::= I \mid EOE \mid (E) \\ O &::= + \mid * \\ I &::= a \mid b \end{aligned}$$

Nesta definição, o conjunto  $C$  especifica seqüências de comandos de atribuição, separados pelo símbolo ;. Note-se a utilização dos símbolos auxiliares  $I$  (identificadores),  $E$  (expressões),  $O$  (operadores). Um exemplo de seqüência especificada por esta definição é  $b \leftarrow a+b; a \leftarrow a*(b+a)$ .

A notação FNB é usada para representar uma classe de definições que em Teoria de Linguagens Formais são chamadas de gramáticas livres de contexto. Um tratamento mais extenso dessa teoria está fora do escopo deste texto, mas as seções seguintes introduzem alguns conceitos e propriedades que serão necessários para a compreensão dos capítulos subsequentes.

## 2.3 GRAMÁTICAS E LINGUAGENS

Chamaremos de *alfabeto* (ou *vocabulário*) a um conjunto finito e não-vazio de símbolos. Neste texto, os símbolos poderão ser simples, como letras, dígitos, caracteres especiais, ou então compostos, como **begin**, **if**, **:=**. Suporemos apenas que, dada uma seqüência de símbolos, é possível separá-los de maneira única, isto é, **begin** ou **if** ou **:=** serão tratados, em geral, como símbolos indivisíveis.

Seja  $\Sigma$  um alfabeto. Uma *cadeia* (ou uma *palavra*) sobre  $\Sigma$  é uma seqüência finita de símbolos de  $\Sigma$ . As cadeias serão indicadas justapondo-se os seus símbolos. Assim, as cadeias de um único símbolo serão identificadas com os próprios símbolos. A *cadeia nula*, constituída pela seqüência de zero símbolos, será denotada por  $\lambda$ . O *comprimento*  $|\alpha|$  de uma cadeia  $\alpha = X_1 X_2 \dots X_n$  com  $n \geq 0$  e  $X_i \in \Sigma$  é dado por  $|\alpha|=n$ ; consequentemente  $|\lambda|=0$ . A *concatenação* (ou o *produto*)  $\alpha\beta$  de duas cadeias com  $\alpha = X_1 \dots X_n$  e  $\beta = Y_1 \dots Y_m$  ( $n, m \geq 0, X_i, Y_i \in \Sigma$ ) é dada pela seqüência  $\alpha\beta = X_1 \dots X_n Y_1 \dots Y_m$ . Obviamente,  $|\alpha\beta| = |\alpha| + |\beta|$  e  $\lambda\alpha = \alpha\lambda = \alpha$  para todo  $\alpha, \beta \in \Sigma^*$ . É fácil verificar que a operação de concatenação é associativa, isto é que  $\alpha\beta\gamma$  está bem definido para todo  $\alpha, \beta, \gamma \in \Sigma^*$ . É conveniente usar-se a notação  $\alpha^n$ ,  $n \geq 0$ , para indicar:

$$\begin{aligned} \alpha^0 &= \lambda \\ \alpha^n &= \alpha^{n-1}\alpha \quad \text{para } n > 0. \end{aligned}$$

O conjunto de todas as cadeias sobre  $\Sigma$  é indicado por  $\Sigma^*$ ; o conjunto  $\Sigma^* - \{\lambda\}$  será denotado por  $\Sigma^+$ . É fácil ver que  $\Sigma^*$  e  $\Sigma^+$  são sempre conjuntos infinitos de cadeias.

Uma *linguagem* sobre um alfabeto  $\Sigma$  é um subconjunto de  $\Sigma^*$ .

## Exemplo 2.2

São exemplos de linguagens sobre  $\{a, b\}$ :

$$\begin{aligned} L_1 &= \emptyset \\ L_2 &= \{\lambda\} \\ L_3 &= \{\alpha \in \{a, b\}^* \mid |\alpha| \leq 2\} \end{aligned}$$

$$L_4 = \{a^n \mid n \geq 0\}$$

$$L_5 = \{a^n b^n \mid n \geq 1\}$$

$$L_6 = \{a, b\}^+$$

$$L_7 = \{a, b\}^*$$

Note-se que  $L_1$ ,  $L_2$  e  $L_3$  são linguagens finitas, com 0, 1 e 7 elementos, respectivamente;  $L_4$ ,  $L_5$ ,  $L_6$  e  $L_7$  são infinitas.

Sejam  $L_1$  e  $L_2$  duas linguagens. Então a *concatenação* (ou o *produto*)  $L_1 L_2$  das duas é dado por  $L_1 L_2 = \{\alpha\beta \mid \alpha \in L_1 \text{ e } \beta \in L_2\}$ . É fácil verificar que esta operação também é associativa. Se  $L$  é uma linguagem, então  $L^n$ ,  $n \geq 0$ , indica:

$$\begin{aligned} L^0 &= \{\lambda\} \\ L^n &= L^{n-1}L \quad \text{para } n > 0. \end{aligned}$$

O *fecho*  $L^*$  de  $L$  é dado por  $L^* = \bigcup_{n \geq 0} L^n$ , e o *fecho positivo*  $L^+$ , por  $L^+ = \bigcup_{n > 0} L^n$ . Em outras palavras, a cadeia  $\alpha$  está em  $L^*$  (ou em  $L^+$ ) se, e somente se, existem as cadeias  $\alpha_1, \alpha_2, \dots, \alpha_n$ , com  $\alpha_i \in L$  e  $n \geq 0$  (ou  $n > 0$ ) tais que  $\alpha = \alpha_1 \alpha_2 \dots \alpha_n$ . Note-se que  $L^* = \{\lambda\} \cup L^+$  e  $L^+ = LL^* = L^*L$ . Note-se, também, que as definições de  $\Sigma^*$  e  $\Sigma^+$  dadas anteriormente são consistentes com esta notação, tratando os símbolos de  $\Sigma$  como cadeias de comprimento unitário. Se  $\alpha$  é uma cadeia, indicaremos por  $\alpha^*$  e  $\alpha^+$  as linguagens  $\{\alpha\}^*$  e  $\{\alpha\}^+$ .

Utilizaremos freqüentemente o conceito de *relações* sobre linguagens. Se  $L_1$  e  $L_2$  são linguagens, então  $R \subseteq L_1 \times L_2$  é uma *relação de  $L_1$  para  $L_2$* . Escreveremos  $\alpha R \beta$  quando  $(\alpha, \beta) \in R$ . Quando  $R \subseteq L \times L$ , dizemos que  $R$  é uma *relação sobre  $L$* . O *produto*  $RQ$  de duas relações  $R \subseteq L_1 \times L_2$  e  $Q \subseteq L_2 \times L_3$  é dado por:

$$RQ = \{(\alpha, \gamma) \mid (\alpha, \beta) \in R \text{ e } (\beta, \gamma) \in Q \text{ para algum } \beta \in L_2\}.$$

Escreveremos  $\alpha R \beta Q \gamma$  para indicar que  $\alpha R \beta$  e  $\beta Q \gamma$ . Note-se que  $RQ \subseteq L_1 \times L_3$ . Novamente, é fácil mostrar que este produto é associativo. Se  $R$  é uma relação sobre  $L$ , então  $R^n$ ,  $n \geq 0$ , denota:

$$\begin{aligned} R^0 &= \{(\alpha, \alpha) \mid \alpha \in L\} \quad (\text{relação identidade sobre } L) \\ R^n &= R^{n-1}R \quad \text{para } n > 0. \end{aligned}$$

Dada uma relação  $R$  sobre  $L$ , o seu *fecho transitivo reflexivo*  $R^*$  é dado por  $R^* = \bigcup_{n \geq 0} R^n$ , e o seu *fecho transitivo* por  $R^+ = \bigcup_{n > 0} R^n$ . (Note-se que  $R^* = R^0 \cup R^+$ .)

Em outras palavras,  $\alpha R^* \beta$  (ou  $\alpha R^+ \beta$ ) se, e somente se, existem as cadeias  $\alpha_0, \alpha_1, \dots, \alpha_n$ , com  $\alpha_i \in L$  e  $n \geq 0$  (ou  $n > 0$ ) tais que  $\alpha = \alpha_0$ ,  $\alpha_n = \beta$  e  $\alpha_i R \alpha_{i+1}$  para  $i = 0, 1, \dots, n-1$ . Se  $R$  é uma relação de  $L_1$  para  $L_2$ , então  $R^T$  denotará a *relação inversa* de  $L_2$  para  $L_1$  dada por  $R^T = \{(\beta, \alpha) \mid \beta R \alpha\}$ .

Passaremos agora a definir o nosso conceito básico que formaliza a notação de definições indutivas introduzida na seção anterior.

Uma *gramática livre de contexto*<sup>1</sup> é uma quádrupla  $G = (T, N, P, S)$  onde:

1.  $T$  é um alfabeto, denominado *vocabulário terminal* de  $G$ ;
2.  $N$  é um alfabeto, denominado *vocabulário não-terminal*<sup>2</sup> de  $G$ , disjunto de  $T$ ;

<sup>1</sup>A designação *livre de contexto*, muito difundida no nosso meio, é uma tradução literal do termo inglês *context-free*. Parece-nos que um termo como *independente do contexto* seria mais apropriado.

<sup>2</sup>Novamente, temos uma tradução literal de *non terminal*; um adjetivo como *intermediário* seria mais adaptado ao nosso idioma.

3.  $\Sigma = T \cup N$  é o *vocabulário* de  $G$ ;
4.  $P$  é uma relação finita de  $N$  para  $\Sigma^*$ ;
5.  $S$  é um elemento de  $N$  (*raiz* ou *símbolo inicial* de  $G$ ).

Os elementos de  $T$  são os *símbolos terminais*, e os de  $N$  são os *símbolos não-terminais* de  $G$ . Os elementos de  $P$  são chamados de *produções* ou *regras de transcrição* de  $G$ .

A qualificação *livre de contexto* vem do fato de termos definido uma classe restrita de gramáticas, em que a regra de transcrição para um símbolo não-terminal independe do contexto em que o símbolo se encontra. Neste texto, trabalharemos apenas com gramáticas livres de contexto, passando a denominá-las simplesmente *gramáticas*.

### Exemplo 2.3

Seja  $G = (\{a,b,+,*,(,),\leftarrow,:,\}, \{C,E,I,O\}, P, C)$  onde  
 $P = \{(C, I \leftarrow E), (C, C; C), (E, I), (E, EOE),$   
 $(E, (E)), (O, +), (O, *), (I, a), (I, b)\}$

Claramente, esta gramática formaliza a definição dada no Exemplo 2.1. ●

Na prática, utiliza-se a notação FNB para indicar as produções. Assim, se os pares  $(A, \alpha_1), (A, \alpha_2), \dots, (A, \alpha_n), n \geq 1$ , estão em  $P$ , então escreveremos  $A ::= \alpha_1 | \alpha_2 | \dots | \alpha_n$ . A fim de simplificar a exposição, adotaremos mais algumas convenções. Os símbolos terminais serão, em geral, letras minúsculas iniciais do alfabeto latino:  $a, b, c, \dots$ , algarismos  $0$  a  $9$ , símbolos especiais e de operação, como  $\#, +, *, (,), \dots$ , símbolos compostos impressos em negrito, como **begin**, **if**, **...**. Os símbolos não-terminais serão indicados por letras maiúsculas iniciais do alfabeto latino:  $A, B, C, \dots$ , ou então por nomes mnemônicos colocados entre parênteses angulares  $< \dots >$ , como  $<\text{expressão}>, <\text{comando}>, \dots$ . As letras maiúsculas finais do alfabeto latino  $U, V, W, X, \dots$  poderão denotar símbolos terminais e não-terminais. Cadeias de símbolos terminais ou não-terminais serão denotadas por letras minúsculas gregas:  $\alpha, \beta, \gamma, \dots$ . Cadeias de símbolos terminais serão denotadas por letras minúsculas latinas:  $u, v, w, x, \dots$ . Finalmente, os alfabetos (vocabulários) serão denotados por letras gregas maiúsculas.<sup>1</sup>

Passaremos a definir agora como uma gramática específica uma linguagem. Seja uma gramática  $G = (T, N, P, S)$ . A relação  $\stackrel{G}{\Rightarrow}$  (ou simplesmente  $\Rightarrow$ , signifi-

cando *deriva diretamente*) sobre  $\Sigma^*$  é definida por: se  $\alpha A \beta \in \Sigma^+$  e  $A ::= \gamma$ , então  $\alpha A \beta \stackrel{G}{\Rightarrow} \alpha \gamma \beta$ . Em outras palavras, se numa cadeia de símbolos ocorre o não-terminal  $A$ , podemos derivar uma nova cadeia substituindo-se esta ocorrência de  $A$  por  $\gamma$ , contanto que exista a produção  $A ::= \gamma$ . Para indicar derivações em vários passos,

utilizaremos os fechos  $\stackrel{*}{\Rightarrow}$  (*deriva*) e  $\stackrel{+}{\Rightarrow}$  (*deriva não trivialmente*). Assim  $\alpha \stackrel{*}{\Rightarrow} \beta$  (ou  $\alpha \stackrel{+}{\Rightarrow} \beta$ ) indica que existem  $\gamma_0, \gamma_1, \dots, \gamma_n$ , com  $n \geq 0$  (ou  $n > 0$ ), tais que  $\alpha = \gamma_0 \gamma_n = \beta$  e  $\gamma_i \stackrel{G}{\Rightarrow} \gamma_{i+1}$  para  $i = 0, 1, \dots, n-1$ . A sequência  $\gamma_0, \gamma_1, \dots, \gamma_n$  é chamada *derivação de  $\beta$  a partir de  $\alpha$* , e o *comprimento da derivação* é definido como sendo

$n$ . Uma cadeia  $\alpha$  que pode ser derivada a partir da raiz  $S$  de  $G$ , isto é  $S \stackrel{*}{\Rightarrow} \alpha$ , é chamada *forma sentencial* de  $G$ ; se  $\alpha \in T^*$ , então  $\alpha$  é uma *sentença*. Finalmente, a

<sup>1</sup>Note-se que as letras  $T$  e  $N$ , que denotam vocabulários terminal e não-terminal, são as letras gregas maiúsculas correspondentes às minúsculas  $\tau$  e  $\nu$ .

linguagem  $L(G)$  definida (ou gerada) por uma gramática  $G$  é o conjunto de suas sentenças, isto é

$$L(G) = \{\alpha \in T^* \mid S \stackrel{*}{\Rightarrow} \alpha\}$$

### Exemplo 2.4

Consideremos a gramática

$$G = (\{a, b\}, \{S\}, \{S ::= ab \mid aSb\}, S)$$

Algumas das derivações possíveis são:

$$\begin{aligned} S &\stackrel{*}{\Rightarrow} ab \\ S &\stackrel{*}{\Rightarrow} aSb \stackrel{*}{\Rightarrow} aabb \\ S &\stackrel{*}{\Rightarrow} aSb \stackrel{*}{\Rightarrow} aaSbb \stackrel{*}{\Rightarrow} aaabbb \end{aligned}$$

Não é difícil mostrar, por indução, que

$$L(G) = \{a^n b^n \mid n \geq 1\}.$$

Note-se que, por exemplo,  $aaSb \stackrel{*}{\Rightarrow} aaabb$  mas  $aaabb \notin L(G)$ . ●

Uma pergunta natural que surge ao estudar as gramáticas livres de contexto é sobre o seu “poder” de definição. O assunto está além do escopo deste texto, mas devemos observar que não é difícil dar exemplos de linguagens que não podem ser definidas por nenhuma gramática desse tipo, ou seja, que não são linguagens livres de contexto. Um exemplo muito concreto é o Pascal. Certos aspectos dessa linguagem tornam impossível a especificação completa de sua sintaxe por meio de uma gramática livre de contexto. Em princípio, poder-se-ia utilizar um tipo diferente de gramática, com maior poder de definição, para esta especificação. Entretanto, esta gramática seria muito complexa e não forneceria diretamente uma maneira eficiente de realizar a análise sintática a ser estudada nos capítulos subsequentes. Na prática, utiliza-se, então, uma gramática livre de contexto, complementada com restrições adicionais especificadas informalmente. Um exemplo desse tipo de restrição é a exigência de que todo identificador seja declarado antes de ser utilizado.

No restante deste texto abreviaremos a notação para gramáticas, indicando apenas o conjunto das produções. Por convenção, serão considerados não-terminais todos os símbolos que aparecem do lado esquerdo de alguma produção; todos os outros símbolos serão considerados terminais. O não-terminal que aparece do lado esquerdo da primeira produção será o símbolo inicial da gramática.

## 2.4 ÁRVORES DE DERIVAÇÃO. AMBIGÜIDADE. DERIVAÇÕES CANÔNICAS

Consideremos novamente a gramática de expressões que denotaremos por  $G_1$ :

$$E ::= a \mid b \mid E+E \mid E*E \mid (E)$$

A sentença  $(a+b)*a$  dessa gramática pode ser obtida por meio de várias derivações distintas, como por exemplo:

$$\begin{aligned} E &\stackrel{*}{\Rightarrow} E*E \stackrel{*}{\Rightarrow} (E)*E \stackrel{*}{\Rightarrow} (E+E)*E \stackrel{*}{\Rightarrow} (a+E)*E \stackrel{*}{\Rightarrow} (a+b)*a \\ E &\stackrel{*}{\Rightarrow} E*E \stackrel{*}{\Rightarrow} E*a \stackrel{*}{\Rightarrow} (E)*a \stackrel{*}{\Rightarrow} (E+E)*a \stackrel{*}{\Rightarrow} (E+b)*a \stackrel{*}{\Rightarrow} (a+b)*a \\ E &\stackrel{*}{\Rightarrow} E*E \stackrel{*}{\Rightarrow} (E)*E \stackrel{*}{\Rightarrow} (E+E)*E \stackrel{*}{\Rightarrow} (E+E)*a \stackrel{*}{\Rightarrow} (a+E)*a \stackrel{*}{\Rightarrow} (a+b)*a \end{aligned}$$

Na realidade existem oito derivações distintas para a sentença  $(a+b)*a$ . Por outro lado, uma comparação cuidadosa de todas estas derivações mostra que o que muda é apenas a ordem em que as produções são aplicadas, sem mudar as alternativas que foram escolhidas para cada ocorrência dos símbolos não-terminais. Num certo sentido, todas estas derivações são equivalentes, ao indicar a mesma estrutura da cadeia  $(a+b)*a$ . Um meio comumente usado para representar todas as derivações que indicam a mesma estrutura são as árvores de derivação (ou árvores sintáticas).

Seja  $G = (T, N, P, S)$  uma gramática e  $X$  um símbolo terminal ou não-terminal. Seja  $\alpha$  uma cadeia derivável de  $X$  e seja  $\beta_0, \beta_1, \dots, \beta_n$ ,  $n \geq 0$ , uma derivação de  $\alpha$  a partir de  $X$ , isto é,  $X = \beta_0 \Rightarrow \beta_1 \Rightarrow \dots \Rightarrow \beta_n = \alpha$ . Uma árvore de derivação de  $\alpha$  a partir de  $X$  correspondente a esta derivação é uma árvore ordenada rótulada  $D$  obtida pela seguinte construção:

1. A árvore inicial  $D_0$  é uma folha com rótulo  $X$ .
2. Seja  $\beta_{i-1} = \gamma A \delta$  e  $\beta_i = \gamma X_1 \dots X_m \delta$ ,  $m \geq 0$ ,  $X_i \in \Sigma$ , com  $A ::= X_1 \dots X_m$  (caso  $m = 0$ , então  $A ::= \lambda$ ). A árvore  $D_i$  é obtida substituindo-se na árvore  $D_{i-1}$  a folha de rótulo  $A$  correspondente a esta ocorrência do não-terminal  $A$  em  $\beta_{i-1}$  por uma subárvore cuja raiz tem rótulo  $A$  e cujas subárvore diretas são folhas de rótulos  $X_1, \dots, X_m$  (caso  $m = 0$ , então haverá uma única folha de rótulo  $\lambda$ ).
3.  $D = D_n$ .

Note-se que faz sentido falar da folha de rótulo  $A$  correspondente a uma ocorrência de  $A$  em  $\beta_i$ , pois prova-se facilmente, por indução, que a concatenação dos símbolos que são rótulos das folhas de  $D_i$  (isto é, aparecem na sua fronteira), da esquerda para a direita, é igual à cadeia  $\beta_i$ . As três derivações da sentença  $(a+b)*a$  correspondem à mesma árvore da Figura 2.1.

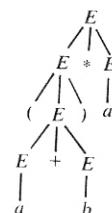


Fig. 2.1

As árvores de derivação não indicam necessariamente apenas derivações de sentenças. A Figura 2.2 indica outros exemplos de árvores de derivação a partir de  $E$ . Note-se que, sempre que o rótulo da raiz da árvore for o símbolo inicial da gramática, a fronteira da árvore fornece uma forma sentencial.

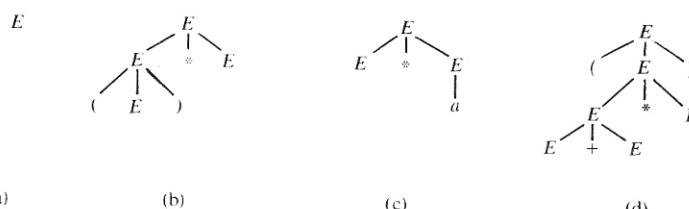


Fig. 2.2

Consideremos agora a cadeia  $a+b*a$ . Duas derivações possíveis são:

$$\begin{aligned} E &\Rightarrow E*E \Rightarrow E+E*E \Rightarrow a+E*E \Rightarrow a+b*E \Rightarrow a+b*a \\ E &\Rightarrow E+E \Rightarrow E+E*E \Rightarrow a+E*E \Rightarrow a+b*E \Rightarrow a+b*a \end{aligned}$$

A Figura 2.3 indica as árvores de derivação correspondentes às duas derivações. Obviamente, as duas árvores são distintas, indicando duas derivações que diferem de maneira mais profunda. A primeira derivação parece indicar, intuitivamente, que a expressão dada é um produto de duas subexpressões, sendo a primeira uma soma. A segunda derivação, por sua vez, indica que a expressão dada é uma soma de duas subexpressões, sendo a segunda um produto. As duas alternativas da Figura 2.3 representam estas duas interpretações.

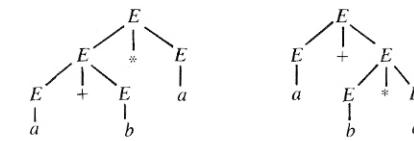


Fig. 2.3

(a) (b)

Uma gramática  $G$  é dita *ambígua* se a linguagem  $L(G)$  contém uma sentença para a qual existe mais do que uma árvore de derivação, usando a gramática  $G$ . Deveria ser óbvio que ambigüidade é uma propriedade indesejável quando se trata de uma linguagem de programação. Como veremos mais adiante, um programa é uma sentença para uma gramática, e o seu significado depende da estrutura revelada pela sua árvore de derivação. Se há mais do que uma árvore, poderá haver dúvida quanto ao significado a ser associado ao programa. Seria interessante ter-se um algoritmo que, para uma dada gramática  $G$ , pudesse decidir se ela é ambígua ou não. Entretanto, usando as técnicas da Teoria da Computabilidade, prova-se que a ambigüidade de uma gramática é indecidível. Em outras palavras, não se pode escrever um programa que aceite como entrada uma descrição de qualquer gramática  $G$ , que sempre pare e que responda se a gramática  $G$  é ou não ambígua. Isto não quer dizer que não se possa provar este fato para uma gramática particular. Assim, os exemplos acima mostram que a nossa gramática de expressões  $G_1$  é ambígua.

Consideremos agora a seguinte derivação da sentença  $a+a+a$ :

$$E \Rightarrow E+E \Rightarrow E+E+E \Rightarrow a+E+E \Rightarrow a+a+E \Rightarrow a+a+a$$

Devido à maneira como foi definido o conceito de derivação, não está claro no passo  $E+E \Rightarrow E+E+E$  qual das duas ocorrências de  $E$  foi substituída. Esta derivação corresponde a duas árvores distintas, conforme os dois casos, indicadas na Figura 2.4. Este é, portanto, mais um exemplo de ambigüidade para a nossa gramática de expressões.

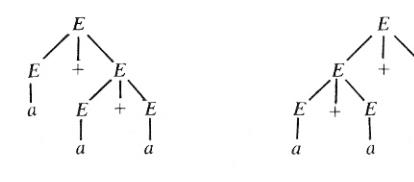


Fig. 2.4

(a) (b)

A causa da ambigüidade está no fato de ela não indicar as precedências entre os operadores  $+$  e  $*$ . É costume atribuir-se a  $*$  precedência sobre  $+$  e associar cada operador à esquerda (isto é,  $+$  e  $*$  têm precedência sobre eles mesmos). As alternativas das Figs. 2.3b e 2.4b são as que correspondem a esta convenção.

Consideremos agora a seguinte gramática  $G_2$ :

$$\begin{aligned} E &::= E+T \mid T \\ T &::= T*F \mid F \\ F &::= a \mid b \mid (E) \end{aligned}$$

Pode-se demonstrar que  $L(G_1) = L(G_2)$ , isto é, que as duas gramáticas definem a mesma linguagem, e que a gramática  $G_2$  não é ambígua. Por exemplo, no caso da sentença  $a+b*a$  ainda temos várias derivações, como:

$$\begin{aligned} E &\Rightarrow E+T \Rightarrow T+T \Rightarrow F+T \Rightarrow a+T \Rightarrow a+T*F \Rightarrow a+F*F \Rightarrow a+b*F \Rightarrow a+b*a \\ E &\Rightarrow E+T \Rightarrow E+T*F \Rightarrow E+T*a \Rightarrow E+F*a \Rightarrow E+b*a \Rightarrow T+b*a \Rightarrow F+b*a \Rightarrow a+b*a \end{aligned}$$

Há, porém, uma única árvore de derivação, indicada na Fig. 2.5a. Para a sentença  $a+a+a$  temos, entre outras, a derivação:

$$E \Rightarrow E+T \Rightarrow E+T+T \Rightarrow T+T+T \Rightarrow F+T+T \Rightarrow a+T+T \Rightarrow a+F+T \Rightarrow a+a+T \Rightarrow a+a+F \Rightarrow a+a+a$$

A Figura 2.5b mostra a única árvore de derivação para esta sentença. Note-se que a gramática  $G_2$  parece representar corretamente as convenções usuais sobre precedência, mas por outro lado as derivações obtidas são mais compridas. Este fato surge da aplicação das produções  $E ::= T$  e  $T ::= F$ .

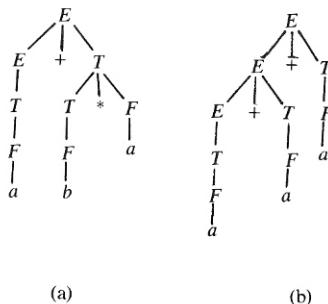


Fig. 2.5

Seja  $G = (T, N, P, S)$  uma gramática. A relação  $\stackrel{*}{\Rightarrow}_e$  (*deriva diretamente à esquerda*) sobre  $\Sigma^*$  fica definida por: se  $\alpha \in T^*$  e  $\beta \in \Sigma^*$  e a produção  $A ::= \gamma$  está em  $P$ , então  $\alpha A \stackrel{*}{\Rightarrow}_e \alpha \gamma \beta$ . A relação  $\stackrel{*}{\Rightarrow}_d$  (*deriva diretamente à direita*) fica definida de maneira análoga, impondo-se  $\alpha \in \Sigma^*$  e  $\beta \in T^*$ . Utilizaremos também os fechos  $\stackrel{*}{\Rightarrow}_e^+$ ,  $\stackrel{*}{\Rightarrow}_d^+$  e  $\stackrel{*}{\Rightarrow}_d^+$  das duas relações. Se  $\alpha = \gamma_0 \stackrel{*}{\Rightarrow}_e \gamma_1 \stackrel{*}{\Rightarrow}_e \dots \stackrel{*}{\Rightarrow}_e \gamma_n = \beta$ , então a seqüência  $\gamma_0, \gamma_1, \dots, \gamma_n$  é uma *derivação esquerda* de  $\beta$  a partir de  $\alpha$ . Define-se de maneira análoga uma *derivação direita*. As derivações esquerdas e direitas são também chamadas de *derivações canônicas*.<sup>1</sup>

Uma propriedade importante das derivações canônicas é que toda sentença de uma gramática  $G$  tem pelo menos uma derivação esquerda e pelo menos uma derivação direita. Esta propriedade não é necessariamente verdadeira para formas sentenciais. Assim  $(E+b)*E$  é uma forma sentencial de  $G_1$ , isto é,  $S \stackrel{*}{\Rightarrow}_e (E+b)*E$ , mas não existem derivações esquerda ou direita de  $G_1$  para esta forma.

Uma outra propriedade importante é que uma gramática  $G$  não é ambígua se, e somente se, toda sentença de  $G$  tem uma única derivação esquerda e uma única derivação direita.

### Exemplo 2.5

Um exemplo famoso de ambigüidade é o chamado “else pendente”. Consideremos a gramática:

$$C ::= a \mid \text{if } b \text{ then } C \text{ else } C \mid \text{if } b \text{ then } C$$

Esta gramática é ambígua, como o demonstra a Figura 2.6, onde temos duas árvores de derivação para a sentença  $\text{if } b \text{ then if } b \text{ then } a \text{ else } a$ .



Fig. 2.6

(a)

(b)

Uma outra maneira de mostrar a ambigüidade é exibir duas derivações esquerdas (ou direitas) para a sentença:

$$\begin{aligned} C &\stackrel{*}{\Rightarrow}_e \text{if } b \text{ then } C \text{ else } C \stackrel{*}{\Rightarrow}_e \text{if } b \text{ then if } b \text{ then } C \text{ else } C \\ &\stackrel{*}{\Rightarrow}_e \text{if } b \text{ then if } b \text{ then } a \text{ else } C \\ &\stackrel{*}{\Rightarrow}_e \text{if } b \text{ then if } b \text{ then } a \text{ else } a \end{aligned}$$

<sup>1</sup>Alguns autores chamam de canônicas apenas as derivações direitas.

A causa da ambigüidade está no fato de ela não indicar as precedências entre os operadores + e \*. É costume atribuir-se a \* precedência sobre + e associar cada operador à esquerda (isto é, + e \* têm precedência sobre eles mesmos). As alternativas das Figs. 2.3b e 2.4b são as que correspondem a esta convenção.

Consideremos agora a seguinte gramática  $G_2$ :

$$\begin{aligned} E &::= E+T \mid T \\ T &::= T*F \mid F \\ F &::= a \mid b \mid (E) \end{aligned}$$

Pode-se demonstrar que  $L(G_1) = L(G_2)$ , isto é, que as duas gramáticas definem a mesma linguagem, e que a gramática  $G_2$  não é ambígua. Por exemplo, no caso da sentença  $a+b*a$  ainda temos várias derivações, como:

$$\begin{aligned} E &\Rightarrow E+T \Rightarrow T+T \Rightarrow F+T \Rightarrow a+T \Rightarrow a+T*T \Rightarrow a+F*F \Rightarrow a+b*F \Rightarrow a+b*a \\ E &\Rightarrow E+T \Rightarrow E+T*T \Rightarrow E+T*a \Rightarrow E+F*a \Rightarrow E+b*a \Rightarrow T+b*a \Rightarrow \\ &\quad F+b*a \Rightarrow a+b*a \end{aligned}$$

Há, porém, uma única árvore de derivação, indicada na Fig. 2.5a. Para a sentença  $a+a+a$  temos, entre outras, a derivação:

$$E \Rightarrow E+T \Rightarrow E+T+T \Rightarrow T+T+T \Rightarrow F+T+T \Rightarrow a+T+T \Rightarrow a+F+T \Rightarrow a+a+T \Rightarrow a+a+F \Rightarrow a+a+a$$

A Figura 2.5b mostra a única árvore de derivação para esta sentença. Note-se que a gramática  $G_2$  parece representar corretamente as convenções usuais sobre precedência, mas por outro lado as derivações obtidas são mais compridas. Este fato surge da aplicação das produções  $E ::= T$  e  $T ::= F$ .

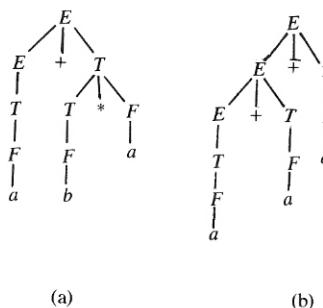


Fig. 2.5

Deve-se notar que nem sempre é possível eliminar a ambigüidade. Uma linguagem  $L$  é *inherentemente ambígua* se toda gramática  $G$  que gera  $L$  é ambígua. Existem exemplos bastante simples de linguagens inherentemente ambíguas (veja Exercício 2.11).

Como vimos acima, mesmo no caso de uma gramática não-ambígua como  $G_2$ , podemos ter mais de uma derivação para uma mesma sentença da linguagem. Isto se deve ao fato de, dada uma forma sentencial  $\alpha$ , podermos escolher para substituição qualquer um dos não-terminais que ocorrem em  $\alpha$ . Como veremos mais adiante, existem duas maneiras naturais de escolher o não-terminal a ser substituído: o mais à esquerda ou o mais à direita.

Seja  $G = (T, N, P, S)$  uma gramática. A relação  $\xrightarrow[e]{\cdot}$  (*deriva diretamente à esquerda*) sobre  $\Sigma^*$  fica definida por: se  $\alpha \in T^*$  e  $\beta \in \Sigma^*$  e a produção  $A ::= \gamma$  está em  $P$ , então  $\alpha A \beta \xrightarrow[e]{} \alpha \gamma \beta$ . A relação  $\xrightarrow[d]{\cdot}$  (*deriva diretamente à direita*) fica definida de maneira análoga, impondo-se  $\alpha \in \Sigma^*$  e  $\beta \in T^*$ . Utilizaremos também os fechos  $\xrightarrow[e]^*$ ,  $\xrightarrow[d]^*$  e  $\xrightarrow[e,d]^*$  das duas relações. Se  $\alpha = \gamma_0 \xrightarrow[e]{} \gamma_1 \xrightarrow[e]{} \dots \xrightarrow[e]{} \gamma_n = \beta$ , então a sequência  $\gamma_0, \gamma_1, \dots, \gamma_n$  é uma *derivação esquerda* de  $\beta$  a partir de  $\alpha$ . Define-se de maneira análoga uma *derivação direita*. As derivações esquerdas e direitas são também chamadas de *derivações canônicas*.<sup>1</sup>

Uma propriedade importante das derivações canônicas é que toda sentença de uma gramática  $G$  tem pelo menos uma derivação esquerda e pelo menos uma derivação direita. Esta propriedade não é necessariamente verdadeira para formas sentenciais. Assim  $(E+b)*E$  é uma forma sentencial de  $G_1$ , isto é,  $S \xrightarrow[*]{\cdot} (E+b)*E$ , mas não existem derivações esquerda ou direita de  $G_1$  para esta forma.

Uma outra propriedade importante é que uma gramática  $G$  não é ambígua se, e somente se, toda sentença de  $G$  tem uma única derivação esquerda e uma única derivação direita.

### Exemplo 2.5

Um exemplo famoso de ambigüidade é o chamado “**else pendente**”. Consideremos a gramática:

$$C ::= a \mid \text{if } b \text{ then } C \text{ else } C \mid \text{if } b \text{ then } C$$

Esta gramática é ambígua, como o demonstra a Figura 2.6, onde temos duas árvores de derivação para a sentença **if b then if b then a else a**.



Fig. 2.6

Uma outra maneira de mostrar a ambigüidade é exibir duas derivações esquerdas (ou direitas) para a sentença:

$$\begin{aligned} C &\xrightarrow[e]{} \text{if } b \text{ then } C \text{ else } C \xrightarrow[e]{} \text{if } b \text{ then if } b \text{ then } C \text{ else } C \\ &\qquad\qquad\qquad \xrightarrow[e]{} \text{if } b \text{ then if } b \text{ then a else } C \\ &\qquad\qquad\qquad \xrightarrow[e]{} \text{if } b \text{ then if } b \text{ then a else a} \end{aligned}$$

<sup>1</sup>Alguns autores chamam de canônicas apenas as derivações direitas.

$$\begin{aligned}
 C &\stackrel{e}{\Rightarrow} \text{if } b \text{ then } C \stackrel{e}{\Rightarrow} \text{if } b \text{ then if } b \text{ then } C \text{ else } C \\
 &\stackrel{e}{\Rightarrow} \text{if } b \text{ then if } b \text{ then } a \text{ else } C \\
 &\stackrel{e}{\Rightarrow} \text{if } b \text{ then if } b \text{ then } a \text{ else } a
 \end{aligned}$$

Neste caso, a ambigüidade vem do fato de a parte **else a** poder ser associada tanto com o primeiro como com o segundo **if**. Ainda neste caso, a ambigüidade pode ser eliminada, se adotarmos a seguinte gramática que gera a mesma linguagem:

$$\begin{aligned}
 C &::= a \mid \text{if } b \text{ then } D \text{ else } C \mid \text{if } b \text{ then } C \\
 D &::= a \mid \text{if } b \text{ then } D \text{ else } D
 \end{aligned}$$

A única árvore de derivação da sentença usada acima está indicada na Figura 2.7. A sua derivação esquerda é:

$$\begin{aligned}
 C &\stackrel{e}{\Rightarrow} \text{if } b \text{ then } C \stackrel{e}{\Rightarrow} \text{if } b \text{ then if } b \text{ then } D \text{ else } C \\
 &\stackrel{e}{\Rightarrow} \text{if } b \text{ then if } b \text{ then } a \text{ else } C \\
 &\stackrel{e}{\Rightarrow} \text{if } b \text{ then if } b \text{ then } a \text{ else } a
 \end{aligned}$$

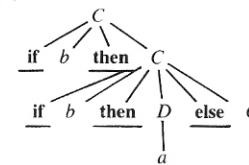


Fig. 2.7

Note-se que esta nova gramática sempre associa o **else** com o **if** mais próximo possível.

## 2.5 ALGUMAS RELAÇÕES ÚTEIS. GRAMÁTICAS REDUZIDAS

Como veremos nos capítulos subsequentes, será útil definir algumas relações derivadas de gramáticas e mostrar como tais relações podem ser calculadas na prática. Trataremos apenas de relações sobre conjuntos finitos — consequentemente, todas as relações também serão finitas.

Se  $H$  é um conjunto finito, podemos sempre enumerar os seus elementos em alguma ordem arbitrária  $H = \{h_1, h_2, \dots, h_n\}$ ,  $n \geq 0$ . Uma relação  $R$  sobre  $H$  pode ser representada, então, por uma matriz quadrada  $\hat{R}$  de ordem  $n$ , cujos elementos são valores booleanos *verdadeiro* ou *falso* tais que  $\hat{R}_{ij} = \text{verdadeiro}$  se e somente se  $h_i R h_j$ . É fácil ver que a matriz  $\hat{R}^t$  da relação inversa será dada pela transposta da matriz  $\hat{R}$ . O produto de relações corresponde ao produto de matrizes definido convenientemente. Sejam  $M$  e  $N$  duas matrizes booleanas de ordem  $n$ . Então o seu produto  $K = MN$  será dado por:

$$K_{ij} = (M_{i1} \wedge N_{1j}) \vee (M_{i2} \wedge N_{2j}) \vee \dots \vee (M_{in} \wedge N_{nj}) \quad i, j = 1, 2, \dots, n.$$

Usando esta operação, pode-se mostrar facilmente que  $\hat{R}\hat{Q} = \hat{R}\hat{Q}$ .

Uma propriedade importante de relações sobre conjuntos finitos é que o seu

fecho pode ser calculado de maneira finita. Assim, se  $R$  é uma relação sobre  $H = \{h_1, h_2, \dots, h_n\}$ ,  $n \geq 0$ , então:

$$R^+ = R \cup R^2 \cup \dots \cup R^n$$

e

$$R^* = R^0 \cup R \cup R^2 \cup \dots \cup R^n$$

Conseqüentemente

$$\widehat{R}^+ = \hat{R} \vee \hat{R}^2 \vee \dots \vee \hat{R}^n$$

e

$$\widehat{R}^* = \hat{R}^0 \vee \hat{R} \vee \hat{R}^2 \dots \vee \hat{R}^n$$

onde  $\vee$  denota a operação  $\vee$  elemento por elemento, e  $\hat{R}^0$  é a matriz cujos elementos são todos *falso*, exceto os da diagonal principal. Usando estas igualdades, o fecho de  $R$  pode ser determinado num número de operações aproximadamente proporcional a  $n^4$ . Existe um algoritmo mais eficiente, que realiza este trabalho com um número de operações da ordem de  $n^3$  (veja Exercício 2.18).

Veremos, mais adiante, que dada uma gramática  $G$ , estaremos interessados, por exemplo, no conjunto de todos os símbolos que podem aparecer no início de uma cadeia derivável de um não-terminal  $A$ , ou seja, no conjunto  $\{X \mid A \xrightarrow{*} X\alpha\}$ . O conjunto  $\{\beta \mid A \xrightarrow{*} \beta\}$  das cadeias deriváveis de  $A$  pode ser infinito, e será difícil utilizá-lo para determinar o conjunto desejado, que por sua vez é finito, por ser um subconjunto de  $\Sigma$ . Suponhamos que a gramática  $G$  não contém produções da forma  $B ::= \lambda$ , e consideremos a relação  $\psi_p$  (primeiro símbolo) sobre  $\Sigma$  definida por:  $X\psi_p Y$  se e somente se  $X ::= Y\alpha$ . Em outras palavras, a relação  $\psi_p$  indica todos os símbolos que aparecem no início de alguma cadeia diretamente derivável de  $X$ . É fácil ver, agora, que o conjunto desejado é dado também por  $\{X \mid A\psi_p^* X\}$ . Obviamente,  $\psi_p$  é uma relação finita que pode ser obtida por uma inspeção direta da gramática.

### Exemplo 2.6

Consideremos novamente a gramática  $G_2$  da seção anterior:

$$\begin{aligned}
 E &::= E + T \mid T \\
 T &::= T * F \mid F \\
 F &::= a \mid b \mid (E)
 \end{aligned}$$

O vocabulário da gramática é  $\Sigma = \{E, T, F, a, b, +, *, (\cdot)\}$ . A matriz que representa a relação  $\psi_p$  está indicada na Figura 2.8a. Para maior legibilidade, as entradas iguais a *verdadeiro* foram representadas por *v*, e as iguais a *falso* foram deixadas em branco; ao invés dos índices das linhas e colunas, colocamos os próprios símbolos. Temos, por exemplo,  $E\psi_p T$  e  $F\psi_p()$ . A Figura 2.8b indica a matriz que representa o fecho  $\psi_p^+$ .

Os exemplos deste texto terão, em geral, poucas produções, e os cálculos poderão ser simplificados. Note-se que a relação  $\psi_p$  é, na realidade, de  $N$  para  $\Sigma$ , e os cálculos podem ser organizados numa tabela como a da Figura 2.9. A coluna sob  $\psi_p$  é obtida, como antes, por inspeção da gramática. A coluna sob  $\psi_p^+$  foi obtida a partir da anterior, acrescentando-se os símbolos obtidos por transitividade. Por exemplo, o símbolo  $F$  foi acrescentado à linha  $E$ , pois já temos  $E\psi_p^* T$ , e  $T\psi_p F$  pela linha  $T$ . Uma vez que  $F$  aparece na linha  $E$ , acrescentam-se, pela mesma razão, os símbolos  $a$ ,  $b$ , e  $(\cdot)$ . O processo pára quando não há mais possibilidade de se acrescentar um símbolo novo a nenhuma das linhas.

	$E$	$T$	$F$	$a$	$b$	$+$	$*$	( )
$E$	$v$	$v$						
$T$		$v$	$v$					
$F$			$v$	$v$			$v$	
$a$								
$b$								
$+$								
$*$								
(								
)								

(a)

	$E$	$T$	$F$	$a$	$b$	$+$	$*$	( )
$E$	$v$	$v$	$v$	$v$	$v$			$v$
$T$		$v$	$v$	$v$	$v$			$v$
$F$			$v$	$v$			$v$	
$a$								
$b$								
$+$								
$*$								
(								
)								

(b)

Fig. 2.8

	$\psi_p$	$\psi_p^+$
$E$	$E \quad T$	$E \quad T \quad F \quad a \quad b \quad ($
$T$	$T \quad F$	$T \quad F \quad a \quad b \quad ($
$F$	$a \quad b \quad ($	$a \quad b \quad ($

Fig. 2.9

Definiremos outras relações que serão úteis:  $\psi_U$  (último símbolo) e  $\psi_I$  (símbolo interno):

$$X\psi_U Y \text{ se e somente se } X ::= \alpha Y$$

$$X\psi_I Y \text{ se e somente se } X ::= \alpha Y\beta$$

Mais relações serão definidas quando necessárias.

Será conveniente impor algumas restrições sobre as gramáticas que serão usadas neste texto. Algumas dessas restrições eliminarão o uso de certas gramáticas que de qualquer maneira não seriam encontradas na prática.

Uma produção da forma  $A ::= A$  é obviamente inútil, podendo introduzir ambigüidade, e a sua eliminação não altera a linguagem gerada pela gramática. De uma maneira mais geral, imporemos que a gramática não deve permitir derivações da forma  $A \xrightarrow{+} A$ . Esta propriedade pode ser verificada facilmente, definindo-se uma relação conveniente.

Notemos, também, que todo símbolo de uma gramática deve ser “útil”, isto é, deve aparecer em alguma forma sentencial, e deve ser possível derivar dele

uma cadeia terminal. Em outras palavras, para qualquer símbolo terminal ou não-terminal  $X$ , devemos ter:

$$\begin{aligned} \text{e} \quad S &\xrightarrow{*} \alpha X \beta \text{ para algum } \alpha \text{ e } \beta \\ &X \xrightarrow{*} w \text{ para algum } w \in T^* \end{aligned}$$

Deveria ser claro que uma produção na qual aparece um símbolo  $X$  que não satisfaz a alguma das duas propriedades pode ser eliminada sem mudar a linguagem gerada. A primeira condição pode ser facilmente verificada, pois equivale a  $S\psi_I^* X$ . Um algoritmo para verificar a segunda condição também não é muito complicado, e será deixado para o leitor (veja Exercício 2.17).

Uma gramática que satisfaz as restrições citadas acima é chamada de *gramática reduzida*, e é apenas esse tipo de gramática que será considerado nos capítulos subseqüentes.

Finalmente, imporemos a restrição de que a gramática não deve conter produções da forma  $A ::= \lambda$ . Essa é uma restrição muito forte, pois o uso desse tipo de produções é bastante comum em linguagens de programação. Entretanto, essa restrição simplifica a discussão da análise sintática a ser abordada nos capítulos seguintes. Os eventuais inconvenientes serão minimizados com a introdução, no Cap. 4, da notação FNB estendida.

## 2.6 ANÁLISE SINTÁTICA

Vimos nas seções anteriores que a cada sentença de uma gramática não-ambígua pode-se associar uma única árvore de derivação ou, equivalentemente, uma única derivação canônica de cada tipo (esquerda e direita). O objetivo dos dois capítulos seguintes é o estudo de alguns métodos para decidir se uma dada cadeia pertence ou não à linguagem definida por uma dada gramática. No caso afirmativo estaremos interessados em obter a estrutura sintática da cadeia, revelada pela sua árvore de derivação ou por uma das suas derivações canônicas. O conhecimento desta estrutura sintática é indispensável para o processo de tradução. Deve-se notar que existem métodos de *análise sintática*<sup>1</sup> que se aplicam também a gramáticas ambíguas, mas na prática estaremos interessados apenas em gramáticas não-ambíguas.

A análise sintática tem sido objeto de pesquisa muito intensa, e existem muitos métodos que podem ser usados. Alguns são aplicáveis a quaisquer gramáticas livres de contexto, outros só se aplicam a certas classes mais restritas de gramáticas. Os métodos variam bastante quanto à sua eficiência, simplicidade de implementação, aplicabilidade a linguagens de programação específicas. Nos capítulos seguintes veremos apenas uns poucos métodos existentes. A escolha baseou-se na popularidade do método (como é o caso da análise de precedência de operadores, da análise descendente recursiva e de algumas formas da análise LR), ou na importância do método para a compreensão de outros métodos derivados ou relacionados (análise de precedência simples).

Suporemos que o objetivo da análise sintática é a construção da árvore de derivação ou, então, apenas a decisão se a cadeia dada é ou não uma sentença. Como veremos mais adiante, o analisador sintático de um compilador não precisa construir explicitamente a árvore, mas as suas ações equivalem conceitualmente a realizar esta tarefa.

Os métodos de análise sintática mais comumente usados podem ser classificados em *ascendentes* ou *descendentes*.<sup>2</sup> Estes termos indicam, de maneira intuitiva,

<sup>1</sup>Em inglês utiliza-se o termo *parsing*, de origem latina, que indica decomposição em partes constituintes.

<sup>2</sup>Em inglês: *bottom-up* e *top-down*.

como é realizada a construção da árvore de derivação: começando pelas folhas ou então pela raiz.

### Exercícios

1. Mostre que as operações de produto para cadeias, linguagens e relações são associativas.
2. Dado um conjunto  $L$ , indique se e quando  $L^*$  e  $L^+$  podem ser:
  - vazios
  - unitários
  - finitos com mais de um elemento
  - infinitos.
3. Prove a veracidade ou a falsidade das afirmações:
  - $L^* = L^+ \cup \{\lambda\}$  para qualquer linguagem  $L$ ;
  - $L^+ = L^* - \{\lambda\}$  para qualquer linguagem  $L$ .
4. Mostre que qualquer conjunto finito de cadeias pode ser gerado por alguma gramática livre de contexto.
5. Mostre que a gramática do Exemplo 2.4 gera a linguagem  $\{a^n b^n \mid n \geq 1\}$ .
6. Escreva gramáticas para gerar cada uma das linguagens abaixo:
  - $\{a^m b^n \mid m \geq n \geq 1\}$
  - $\{a^m b^n a^n b^m \mid m \geq 1 \text{ e } n \geq 0\}$
  - $\{x \in \{a,b\}^* \mid |x|_a = |x|_b\}$

onde  $|w|_a$  denota o número de ocorrências do símbolo  $a$  na palavra  $w$ . Prove que as suas gramáticas geram as linguagens correspondentes.
7. Enumere todas as derivações distintas da cadeia  $b*a+a$  usando as gramáticas  $G_1$  e  $G_2$ . Indique as árvores de derivação correspondentes.
8. Suponha que temos uma gramática tal que  $A \stackrel{+}{\Rightarrow} A$ . Esta gramática é necessariamente ambígua? Justifique.
9. Mostre que as gramáticas  $G_1$  e  $G_2$  geram a mesma linguagem.
10. Mostre que a gramática  $G_2$  não é ambígua.
11. Escreva uma gramática para a linguagem  $\{a^i b^j c^k \mid i, j, k \geq 1, \text{ e } i=j \text{ ou } j=k\}$ . Ache exemplos de sentenças que demonstram que a sua gramática é ambígua. (A linguagem dada é inherentemente ambígua e, portanto, toda gramática que a gera é ambígua.)
12. Dê uma condição necessária e suficiente para que as derivações esquerda e direita de cada sentença de  $L(G)$  sejam idênticas.
13. É possível que uma sentença tenha duas derivações esquerdas e uma única derivação direita, dada a gramática  $G$ ? Justifique.
14. Escreva uma gramática para definir expressões formadas com variáveis  $x$ ,  $y$  e  $z$ , operações unitárias  $+ e -$ , operações binárias  $+, -, *, /, \uparrow$  e parênteses. Os operadores unitários devem ter precedência sobre os binários. Os operadores binários  $+ e -$  têm a mesma precedência e são associados à esquerda. Os operadores  $* e /$  têm a mesma precedência, são associados à esquerda e precedem  $+ e -$  binários. O operador  $\uparrow$  precede todos os outros operadores binários e é associado à direita.
15. Calcule as relações  $\psi_P, \psi_P^\dagger, \psi_U, \psi_U^\dagger, \psi_I$  e  $\psi_I^\dagger$  para a gramática do Exercício 2.14.
16. Defina a relação  $\psi_x$  tal que  $A \psi_x B$  se e somente se  $A \stackrel{+}{\Rightarrow} B$ .
17. Descreva um algoritmo que verifica se todos os símbolos de uma gramática satisfazem a condição  $X \stackrel{+}{\Rightarrow} w$  para algum  $x \in T^*$ , discutida na Seção 2.5.
18. Suponha que  $M$  é uma matriz booleana de ordem  $n$  que representa uma

relação  $R$  (veja Seção 2.5). Prove que a execução do seguinte comando faz com que  $M$  represente a relação  $R^+$ :

```
para i:=1 até n faça
  para j:=1 até n faça
    se  $M_{ji}$  então
      para k:=1 até n faça  $M_{jk} := M_{jk} \vee M_{ik}$ 
```

19. Seja  $\Delta(X)$  o conjunto de todos os símbolos que podem seguir  $X$  em alguma forma sentencial, isto é:

$$\Delta(X) = \{ Y \in \Sigma \mid S \stackrel{+}{\Rightarrow} \alpha X Y \beta, \quad \alpha, \beta \in \Sigma^*\}$$

Indique uma maneira finita de calcular a função  $\Delta(X)$ ,  $X \in \Sigma$ .

20. Descreva um algoritmo que verifica se para um dado não-terminal  $A$  tem-se  $A \stackrel{+}{\Rightarrow} \lambda$  (suponha que são permitidas produções com  $\lambda$  do lado direito).

### NOTAS BIBLIOGRÁFICAS

O conceito de linguagem livre de contexto foi introduzido por Chomsky (1956 e 1959). Independentemente, a notação FNB foi usada pela primeira vez em Naur (1963) para definir a linguagem Algol 60. Existem vários textos que incluem tratamento profundo da teoria de linguagens livres de contexto, sendo os mais conhecidos Ginsburg (1966), Salomaa (1973), Hopcroft e Ullman (1969) e Harrison (1978); em português existe Simon (1981). Backhouse (1979) é um texto recente e muito acessível para leitor principiante. Praticamente todos os textos dedicados à compilação, e citados nos dois capítulos seguintes, trazem uma introdução sobre este assunto. O algoritmo apresentado no Exercício 2.18 vem de Warshall (1962).

# Análise Sintática Ascendente

## 3.1 GENERALIDADES

Nos algoritmos de análise sintática ascendente, a construção da árvore de derivação para uma dada cadeia começa pelas folhas da árvore e procede na direção da sua raiz. Caso seja obtida uma árvore cuja raiz tem por rótulo o símbolo inicial da gramática, e a qual a sequência dos rótulos das folhas forma a cadeia dada, então a cadeia é uma sentença da linguagem, e a árvore obtida é a sua árvore de derivação.

O funcionamento básico de um algoritmo ascendente pode ser descrito, informalmente, da seguinte maneira:

1. Adota-se como o valor inicial de  $\alpha$  a cadeia dada.
2. Procura-se decompor  $\alpha$  de tal maneira que  $\alpha = \beta X_1 X_2 \dots X_n \gamma$ , e exista uma produção da forma  $X ::= X_1 X_2 \dots X_n$ . Caso isto seja possível, adota-se a nova cadeia  $\alpha' = \beta \gamma$ , associando-se com esta ocorrência do não-terminal  $X$  uma árvore cuja raiz tem rótulo  $X$  e cujas subárvore diretas são as árvores que estavam associadas com as ocorrências de  $X_1, X_2, \dots, X_n$  que foram substituídas. Se  $X_i$  é um terminal, então a árvore associada será uma folha de rótulo  $X_i$ . (Note-se que este processo é o inverso de uma derivação, e é chamado de *redução*.)
3. O passo 2 é repetido até que o valor de  $\alpha$  seja o símbolo inicial da gramática.

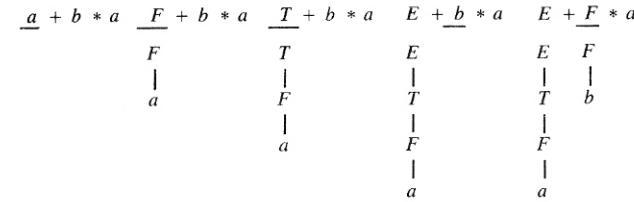
### Exemplo 3.1

Consideremos novamente a gramática de expressões:

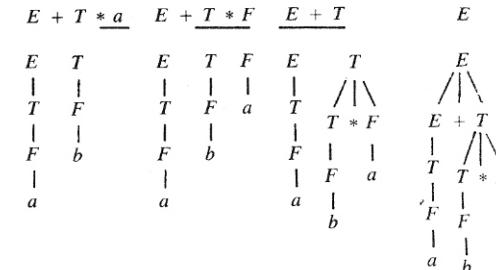
$$\begin{aligned} E &::= E+T \mid T \\ T &::= T*F \mid F \\ F &::= a \mid b \mid (E) \end{aligned}$$

e a sentença  $a+b*a$ . As reduções sucessivas que poderiam ser realizadas por um algoritmo típico estão indicadas na Figura 3.1. Sublinhamos a parte da cadeia a ser reduzida em cada passo, e indicamos as árvores associadas aos não-terminais das cadeias intermediárias.

Note-se que se as cadeias que aparecem nos passos da Figura 3.1 forem lidas em ordem inversa, obter-se-á a sequência  $E, E+T, E+T*F, E+T*a, E+F*a, E+b*a, T+b*a, F+b*a, a+b*a$ , que é a derivação direita da cadeia  $a+b*a$ . Este fato é uma consequência de termos procurado, nesse exemplo, as reduções aceitáveis mais à esquerda possíveis. Como o processo de redução é o inverso da derivação, obtém-se desta maneira a derivação direita. Esta ordem de redução é muito natural se



(a) (b) (c) (d) (e)



(f) (g) (h) (i)

Fig. 3.1

considerarmos que programas são geralmente lidos da esquerda para a direita. Na realidade, o analisador sintático de um compilador não precisa, em geral, conhecer a cadeia  $\alpha$  para decidir qual é a próxima redução. Como veremos mais adiante, é suficiente conhecer uma parte inicial  $\alpha'$  de  $\alpha$  que é decomposta em  $\alpha' = \beta X_1 X_2 \dots X_n \gamma'$ . À medida que for necessário, novos símbolos serão buscados pelo analisador — por exemplo, lidos — estendendo a cadeia  $\alpha'$ .

Uma outra observação importante é que nem todas as reduções aparentemente possíveis devem ser realizadas. Assim, se após o passo (f) da Figura 3.1 reduzíssimo  $T$  para  $E$  — como fizemos no passo (c) — então chegaríamos eventualmente a uma cadeia  $\alpha$  para a qual não há mais reduções possíveis, e que não é a cadeia final desejada. Seria possível, em princípio, usar um algoritmo que explora todas as alternativas, retrocedendo sempre que não for possível continuar. Entretanto, este tipo de análise com *retrocesso*<sup>1</sup> poderia ser muito ineficiente. Estaremos interessados, portanto, em algoritmos em que nunca são escolhidas reduções erradas. Estes serão mais eficientes, mas o preço a ser pago é o fato de serem aplicáveis a uma classe mais restrita de gramáticas.

Podemos concluir que os problemas básicos da análise ascendente são: (1) identificação da parte da cadeia que deve ser reduzida e (2) identificação da produção que deve ser usada na redução. Note-se que se a gramática possui duas produções da forma  $A ::= \delta$  e  $B ::= \delta$ , e se a cadeia a ser reduzida é  $\delta$ , então deve haver uma maneira de decidir qual das duas produções deve ser aplicada. Alguns algoritmos de análise ascendente usam um mecanismo de redução um pouco dife-

<sup>1</sup>Em inglês *backtracking*.

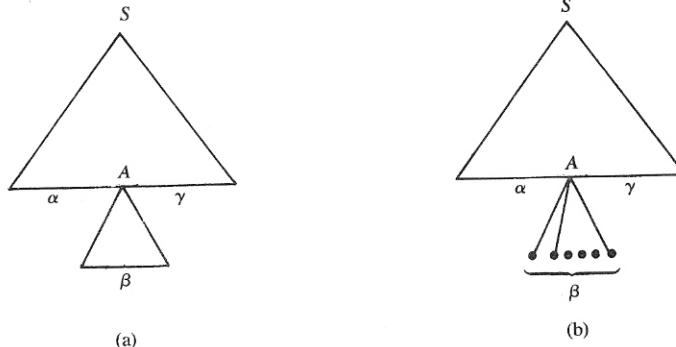


Fig. 3.2

rente do descrito acima, como é o caso da análise de precedência de operadores, a ser vista na Seção 3.3.

Antes de passar a descrever métodos específicos, introduziremos alguns conceitos adicionais. Seja  $G$  uma gramática, e suponhamos que  $S \xrightarrow{*} \alpha A \gamma$  e  $A \xrightarrow{*} \beta$ . Dizemos então que  $\beta$  é uma *frase* da forma sentencial  $\alpha \beta \gamma$  para o não-terminal  $A$  (é fácil ver que  $S \xrightarrow{*} \alpha \beta \gamma$ ). Em termos da árvore de derivação para a forma sentencial  $\alpha \beta \gamma$ ,  $\beta$  é a fronteira de uma subárvore cuja raiz tem o rótulo  $A$  e que não é uma folha. A Figura 3.2a indica esta situação de maneira esquemática. Caso se tenha  $A \xrightarrow{*} \beta$  (derivação direta), então  $\beta$  é chamada de *frase simples* (veja Figura 3.2b).

### Exemplo 3.2

Consideremos a mesma gramática do Exemplo 3.1. As frases da forma sentencial  $a+b*a$  são:

- a (frase para  $F, T$ , e  $E$ ; frase simples para  $F$ )
- b (frase para  $F$  e  $T$ ; frase simples para  $F$ )
- $a$  (frase para  $F$ ; frase simples para  $F$ )
- $b*a$  (frase para  $T$ )
- $a+b*a$  (frase para  $E$ )

As frases da forma sentencial  $a+T*T$  são:

- a (frase para  $F, T$  e  $E$ ; frase simples para  $F$ )
- $T*T$  (frase para  $T$ ; frase simples para  $T$ )
- $a+T*T$  (frase para  $E$ )

A definição a seguir usa o conceito mais restrito de derivação direita, uma vez que ela está associada de maneira natural com a análise ascendente. Se  $S \xrightarrow{d} \alpha A w$  e  $A \xrightarrow{*} \beta$ , então  $\beta$  é um *redutendo*<sup>1</sup> da forma sentencial direita  $\alpha \beta w$  para o não-terminal  $A$  ( $S \xrightarrow{d} \alpha \beta w$ , pois  $w \in T^*$ ). Note-se que o redutendo também é uma

<sup>1</sup>Ao invés de utilizar uma tradução literal do termo inglês *handle* (maçaneta, punho, cabo, asa, etc.), decidimos introduzir este termo que, apesar de não constar dos dicionários da língua portuguesa, parece designar de maneira adequada o conceito envolvido.

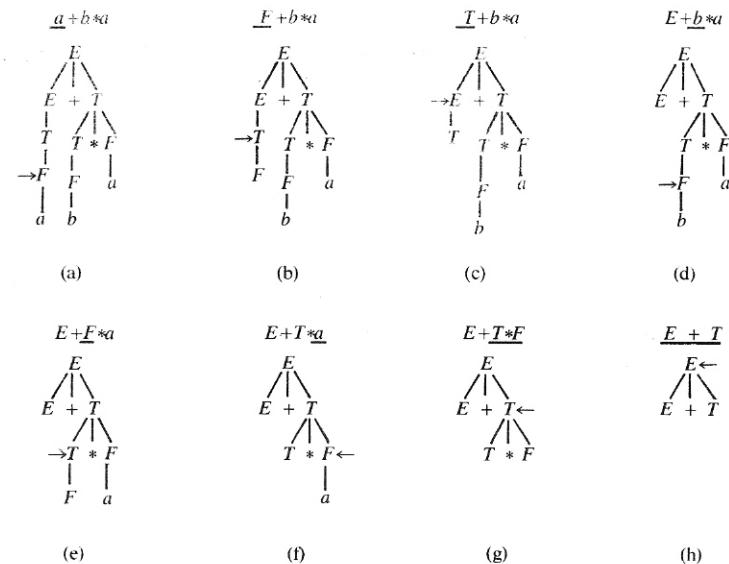


Fig. 3.3

frase simples. Pode-se demonstrar que, para gramáticas não-ambíguas, o redutendo de uma forma sentencial direita é único. Neste caso, o redutendo será a frase simples mais à esquerda da forma sentencial. Conclui-se, portanto, que dada uma forma sentencial direita, o algoritmo de análise deverá aplicar a redução ao seu redutendo. Note-se que o resultado desta redução será uma nova forma sentencial direita.

### Exemplo 3.3

Consideremos novamente a gramática do Exemplo 3.1 e a cadeia  $a+b*a$ . A Figura 3.3 indica o redutendo em cada passo da análise e a sua interpretação em termos de árvore de derivação. Os redutendos estão sublinhados, e as flechas indicam as raízes das árvores cujas fronteiras formam esses redutendos. ●

## 3.2 ANÁLISE DE PRECEDÊNCIA SIMPLES

O primeiro passo deste método é determinar, para uma dada gramática  $G$ , três relações auxiliares sobre  $\Sigma$ , indicadas por  $<$ ,  $\doteq$  e  $>$ . Estas relações devem permitir a identificação do redutendo de  $\alpha$ . Assim, se  $\alpha = Y_1 Y_2 \dots Y_n$ , e o redutendo de  $\alpha$  é  $Y_k Y_{k+1} \dots Y_m$ ,  $1 \leq k \leq m \leq n$ , então devemos ter:

$$\begin{array}{ll} Y_i < Y_{i+1} \text{ ou } Y_i \doteq Y_{i+1} & \text{para } i=1, \dots, k-2, \\ Y_{k-1} < Y_k & (\text{se } k > 1), \\ Y_i \doteq Y_{i+1} & \text{para } i=k, \dots, m-1 \end{array}$$

e

$$Y_m > Y_{m+1} \quad (\text{se } m < n).$$

Em outras palavras, a parte que deve ser reduzida é a parte  $Y_k \dots Y_m$  que está mais

à esquerda com

$$Y_{k-1} \lessdot Y_k \doteq Y_{k+1} \doteq \dots \doteq Y_m \succ Y_{m+1}.$$

Caso  $k = 1$  ou  $m = n$ , então  $Y_{k-1}$  ou  $Y_{m+1}$  não existem.

Note-se que para que o método funcione, as relações  $\lessdot$ ,  $\doteq$  e  $\succ$  devem ser disjuntas. Deve-se, também, chamar a atenção para o fato de que estas relações não têm nenhuma ligação com as relações  $<$ ,  $=$  e  $>$  definidas sobre números, e que elas não têm as mesmas propriedades.

Uma definição precisa das relações  $\lessdot$ ,  $\doteq$  e  $\succ$  é dada por:

1. Dizemos que  $X \lessdot Y$  se existe uma forma sentencial direita  $\alpha = \beta X Y \gamma w$  tal que  $Y \gamma$  é um redutendo de  $\alpha$ .
2. Dizemos que  $X = Y$  se existe uma forma sentencial direita  $\alpha = \beta \gamma X Y \delta w$  tal que  $\gamma X Y \delta$  é um redutendo de  $\alpha$ .
3. Dizemos que  $X \succ Y$  se existe uma forma sentencial direita  $\alpha = \beta \gamma X Y w$  tal que  $\gamma X$  é um redutendo de  $\alpha$ . Note-se que neste caso  $Y$  é um símbolo terminal.

(Deve-se lembrar que, de acordo com a convenção adotada na Seção 2.3, temos nesta definição  $\beta, \gamma, \delta \in \Sigma^*$  e  $w \in T^*$ .) A definição dada indica que se  $\alpha = Y_1 Y_2 \dots Y_n$  é uma forma sentencial direita, com o redutendo  $Y_1 Y_{k+1} \dots Y_m$ , então ela induz os pares  $Y_{k-1} \lessdot Y_k$ ,  $Y_i = Y_{i+1}$  para  $i = k, \dots, m-1$  e  $Y_m \succ Y_{m+1}$ . Se  $k = 1$  ou  $m = n$ , então não é induzido nenhum par para as relações  $\lessdot$  ou  $\succ$ .

Uma gramática  $G$  é chamada *gramática de precedência simples* se:

1. As relações  $\lessdot$ ,  $\doteq$  e  $\succ$  são disjuntas, isto é, não existe nenhum par de símbolos que pertence a mais de uma destas relações.
2.  $G$  não possui duas produções da forma  $A ::= \gamma$  e  $B ::= \gamma$ .

É importante notar que a definição das relações de precedência não depende do fato de a gramática ser ambígua ou não. Pode-se demonstrar, porém, que toda gramática de precedência simples não é ambígua. A recíproca não é verdadeira, pois uma gramática pode ser não-ambígua sem ser de precedência simples. Uma outra proposição, que justifica a utilização das relações de precedência, é que o redutendo de uma forma sentencial direita de uma gramática de precedência simples é dado pela parte mais esquerda da forma, tal que os seus símbolos consecutivos estão em  $\doteq$ , e é delimitado por pares em  $\lessdot$  e  $\succ$  (exceto nas extremidades da forma).

As relações de precedência são definidas sobre o conjunto finito  $\Sigma$ , e são portanto finitas. Entretanto, a definição dessas relações, dada acima, não pode ser usada diretamente para a sua determinação, pois parte do conjunto de todas as formas sentenciais direitas que, em geral, é infinito. A Figura 3.4 indica, de maneira esquemática, as implicações da definição das relações de precedência simples. Nesta figura, o nó com rótulo  $A$  é sempre o antecessor comum mais baixo dos nós  $X$  e  $Y$  ( $A$  pode, eventualmente, ser a própria raiz  $S$  da árvore). Na Figura 3.4a ( $X \lessdot Y$ ), temos  $\beta = \beta_1 \beta_2$  e  $w = w_1 w_2 w_3$ ; os nós rotulados com  $V$  e  $Z$  podem coincidir. Na Figura 3.4c ( $X \succ Y$ ), temos  $\beta = \beta_1 \beta_2 \beta_3$  e  $w = w_1 w_2 w_3$ ; o nó  $V$  pode coincidir com  $Z$ , e o nó  $Y$  com  $W$ . Usando as idéias sugeridas pela Figura 3.4, pode-se demonstrar então a seguinte proposição ( $\psi_P$  e  $\psi_U$  são as relações definidas na Seção 2.5>):

1.  $X \lessdot Y$  se e somente se  $X(\doteq) \psi_P^+ Y$ .
2.  $X = Y$  se e somente se existe uma produção da forma  $A ::= \alpha X Y \beta$ .
3.  $X \succ Y$  se e somente se  $Y \in T$  e  $X(\psi_U^+)^T (\doteq) \psi_P^* Y$ .

A proposição indica que a relação  $\doteq$  pode ser determinada por uma inspeção direta das produções; as relações  $\lessdot$  e  $\succ$  podem ser determinadas usando-se fechos

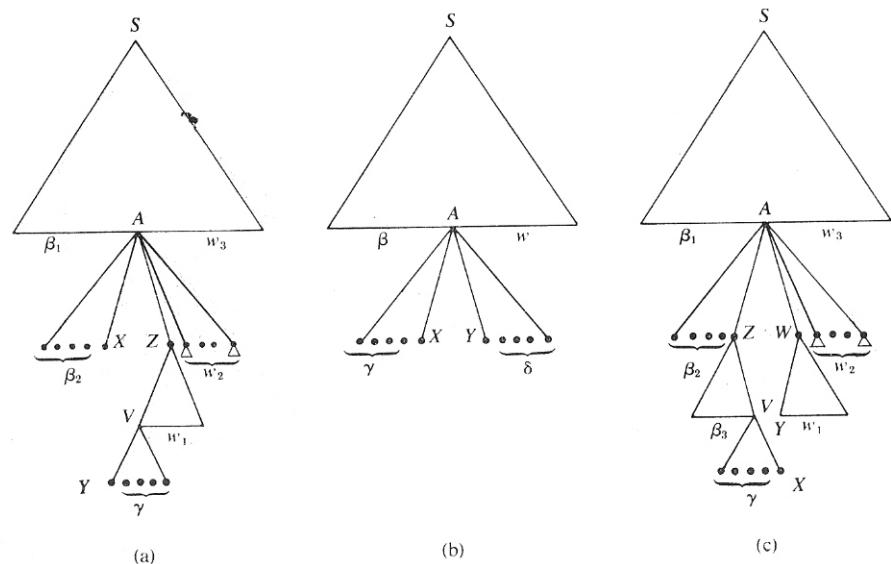


Fig. 3.4

e produtos de relações finitas, como foi indicado na Seção 2.5. Note-se que, dada uma gramática  $G$ , este cálculo precisa ser feito uma única vez. Não é difícil automatizar este processo, escrevendo-se um programa de computador que aceita como entrada a descrição de uma gramática e produz como saída a tabela de relações de precedência.

#### Exemplo 3.4

Consideremos a gramática:

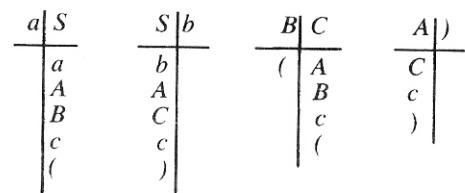
$$\begin{aligned} S &::= aSb \mid A \\ A &::= BC \mid c \\ B &::= ( \\ C &::= A \end{aligned}$$

Calculando as relações  $\psi_P^+$  e  $\psi_U^+$ , teremos a tabela da Figura 3.5a. A Figura 3.5b indica uma maneira prática de se calcular as relações  $\lessdot$  e  $\succ$  para gramáticas de poucas produções. No topo de cada tabela de duas colunas aparece um par  $(X, Y)$  tal que  $X \doteq Y$ . Embaixo de  $X$  colocam-se todos os símbolos  $V$  tais que  $V(\psi_U^+)^T X$  (isto é,  $X(\psi_U^+)^T V$ ), e embaixo de  $Y$  todos os símbolos  $W$  tais que  $Y\psi_P^+ W$ . A relação  $\lessdot$  é obtida tomando-se todos os  $(X, V)$  onde  $V$  está na segunda coluna. A relação  $\succ$  é obtida tomando-se todos os  $(V, W)$  onde  $V$  está na primeira coluna e  $W$  está na segunda coluna (incluindo o próprio símbolo  $Y$ ), sempre que  $W$  for terminal. A tabela da Figura 3.5c indica as três relações. Como todas as relações são disjuntas e a gramática não tem produções com lados direitos iguais, conclui-se que ela é de precedência simples.

Consideremos agora a cadeia  $aa(c)bb$ . A Figura 3.6a indica todas as reduções baseadas nas relações calculadas na Figura 3.5. A Figura 3.6b indica as árvores obtidas em cada passo.

	$\psi_P$	$\psi_P^+$	$\psi_U$	$\psi_U^+$
$S$	$aA$	$aABC($	$bA$	$bACC)$
$A$	$Bc$	$Bc($	$Cc$	$Cc)$
$B$	$($	$($	$($	$($
$C$	$A$	$ABC($	$)$	$)$

(a)



(b)

	$S$	$A$	$B$	$C$	$a$	$b$	$c$	$($	$)$
$S$						$\doteq$			
$A$						$>$			$\doteq$
$B$		$<$	$<$	$\doteq$			$<$	$<$	
$C$						$>$			$>$
$a$	$\doteq$	$<$	$<$		$<$		$<$	$<$	
$b$						$>$			
$c$						$>$			$>$
$($						$>$	$>$		
$)$						$>$			$>$

(c)

Fig. 3.5

PASSO	FORMA SENTENCIAL	REDUTENDO	REDUÇÃO PARA
1	$a \triangleleft a \triangleleft (\triangleright c) b b$	$($	$B$
2	$a \triangleleft a \triangleleft B \triangleleft c \triangleright) b b$	$c$	$A$
3	$a \triangleleft a \triangleleft B \triangleleft A \doteq) \triangleright b b$	$A )$	$C$
4	$a \triangleleft a \triangleleft B \doteq C \triangleright b b$	$BC$	$A$
5	$a \triangleleft a \doteq A \doteq b b$	$A$	$S$
6	$a \triangleleft a \doteq S \doteq b b$	$aSb$	$S$
7	$a \doteq S \doteq b$	$aSb$	$S$
8	$S$		

(a)

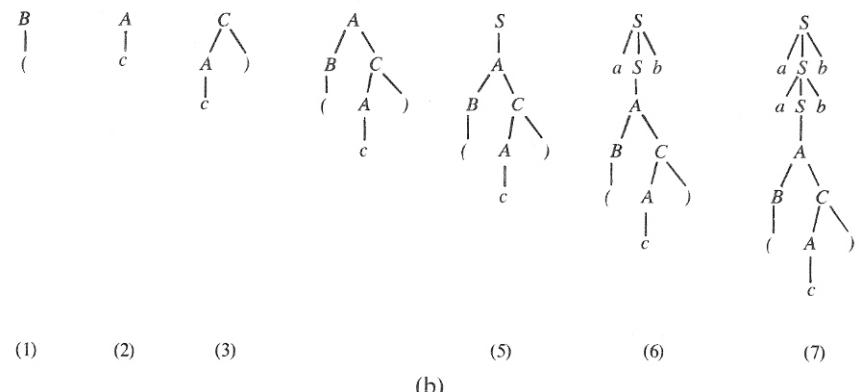


Fig. 3.6

Note-se que várias entradas da tabela da Figura 3.5c ficaram vazias. Estas indicam que os pares correspondentes não pertencem a nenhuma das três relações de precedência. É fácil mostrar que tais pares não podem aparecer consecutivos numa forma sentencial direita. A sua ocorrência indica que a cadeia inicial dada não é uma sentença da linguagem. Na realidade, o particular par de símbolos encontrado ajuda na emissão de um diagnóstico mais preciso sobre a natureza do erro.

### Exemplo 3.5

Consideremos novamente a gramática de expressões:

$$\begin{aligned} E &::= E + T \mid T \\ T &::= T * F \mid F \\ F &::= a \mid b \mid (E) \end{aligned}$$

A Figura 3.7 indica o cálculo das relações de precedência correspondentes. Esta gramática não é de precedência simples, pois há conflitos entre as relações (por

	$\psi_P$	$\psi_P^+$	$\psi_U$	$\psi_U^+$
$E$	$TE$	$TEFab($	$T$	$TFab)$
$T$	$FT$	$FTab($	$F$	$Fab)$
$F$	$ab($	$ab($	$ab)$	$ab)$

(a)

$E$	$+$	$+T$	$T*$	$*F$	$( E $	$ E )$
$T$		$F$	$F$		$T$	$T$
$F$		$T$	$a$	$b$	$E$	$F$
$a$		$a$	$b$		$F$	$a$
$b$		$b$	)		$a$	$b$
)		(			$b$	)
				(		

(b)

	$E$	$T$	$F$	$a$	$b$	$+$	$*$	$($	$)$
$E$						$\doteq$			$\doteq$
$T$						$>$	$\doteq$		$>$
$F$						$>$	$>$		$>$
$a$						$>$	$>$		$>$
$b$						$>$	$>$		$>$
$+$		$\doteq$	$<$	$<$	$<$			$<$	
$*$			$\doteq$	$<$	$<$			$<$	
$($	$\doteq$	$<$	$<$	$<$	$<$			$<$	
$)$						$>$	$>$		$>$

(c)

Fig. 3.7

exemplo,  $( \leq E \text{ e } ( \doteq E )$ .

Este último exemplo ilustra uma dificuldade que surge quando a gramática contém produções com recursão esquerda, ou seja, da forma  $A ::= A\alpha$ . Se houver uma outra produção da forma  $B ::= \beta X A \gamma$ , então teremos  $X \doteq A$  e  $X \leq A$ . Uma maneira de evitar este conflito é modificar a gramática sem modificar a linguagem gerada, substituindo-se a produção  $B ::= \beta X A \gamma$  pelas produções  $B ::= \beta X D \gamma$

e  $D ::= A$ , onde  $D$  é um novo símbolo não-terminal. Obtém-se assim  $X \doteq D$  e  $X \leq A$ . Este processo, que é chamado *estratificação* ou *separação*, pode ser aplicado também no caso da recursão direita que apresenta problemas análogos. Esta modificação, entretanto, nem sempre elimina o problema, podendo introduzir outros conflitos.

### Exemplo 3.6

Consideremos a gramática abaixo, obtida pela estratificação das produções do Exemplo 3.5:

$$\begin{aligned} E &::= E + T' \mid T' \\ T' &::= T \\ T &::= T * F \mid F \\ F &::= a \mid b \mid (E') \\ E' &::= E \end{aligned}$$

Note-se que  $E ::= T$  foi modificado para  $E ::= T'$  a fim de evitar duas produções com lados direitos iguais. A Figura 3.8 indica o cálculo das relações de precedência para esta gramática modificada, e a Figura 3.9 indica os passos de redução para a sentença  $a+b*a$ .

A árvore de derivação final obtida no Exemplo 3.6 é semelhante àquela obtida no Exemplo 3.1, mas inclui alguns nós internos que indicam apenas uma mudança no nome do não-terminal associado com a frase, sem mudar essencialmente a estrutura. Uma consequência óbvia da estratificação é que ela aumenta o número de reduções realizadas para analisar uma cadeia. Uma outra consequência importante é que a introdução de novos símbolos não-terminais aumenta o tamanho das tabelas que são necessárias para guardar as relações de precedência.

Na prática, acrescenta-se à gramática um delimitador especial (por exemplo  $\#$ ) que não pertence ao seu vocabulário, e coloca-se  $\# \leq X$  para todo  $X$  tal que  $S\psi_P^+X$ , e  $X \geq \#$  para todo  $X$  tal que  $S\psi_U^+X$ . Isto quase que equivale a acrescentar à gramática uma nova produção da forma  $S' ::= \# S \#$ , e tomar  $S'$  como o símbolo inicial. Com esta modificação, se  $Y_1\dots Y_n$  é o redutendo da forma sentencial  $Y_1\dots Y_n$  desta gramática modificada, então  $k > 1$  e  $m < n$ , pois  $Y_1 = Y_n = \#$ . A análise deve proceder até que se chegue à cadeia  $\# S \#$ .

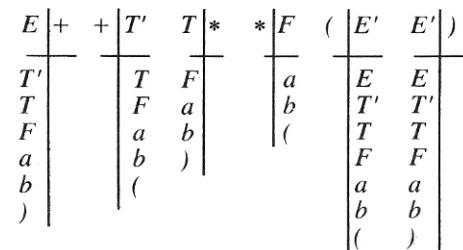
A Figura 3.10 apresenta o algoritmo de análise de precedência simples.  $P$  é um vetor que contém uma pilha de símbolos que estão esperando redução;  $M$  é uma tabela de precedência, que para cada par de símbolos tem o valor  $\leq, \doteq, >$  ou  $?$  (indicando uma entrada vazia). *PRÓXIMO* é um procedimento que coloca na variável global *símbolo* o próximo símbolo da cadeia de entrada (quando terminar a cadeia será devolvido  $\#$ ); *ERRO* é um procedimento que deverá tratar de erros. Caso a chamada do procedimento não acuse nenhum erro, a cadeia é uma sentença da linguagem.

Não seria difícil transformar o programa da Figura 3.10 num programa que constrói a árvore de derivação numa representação adequada. Poderíamos empilhar juntamente com cada símbolo a sua árvore correspondente (uma folha no caso de terminais). Numa redução, as árvores correspondentes aos símbolos que constituem o redutendo seriam combinadas, tornando-se as subárvore diretas da raiz da nova árvore associada ao não-terminal para o qual a redução foi realizada.

A análise de precedência simples é pouco usada na prática, apesar de ser muito simples de implementar. Em primeiro lugar, ela requer uma tabela para guardar as relações de precedência que para uma linguagem de programação real pode ser muito grande. Como já vimos, este problema pode ser agravado pela aplicação da estratificação, pois o uso da recursão nas produções é muito comum. Pode-se mostrar que o tempo de execução do algoritmo é essencialmente proporcional ao

	$\psi_P$	$\psi_P^+$	$\psi_U$	$\psi_U^+$
$E$	$T'E$	$T'ETFab($	$T'$	$T'TFab)$
$E'$	$E$	$ET'TFab($	$E$	$ET'TFab)$
$T$	$FT$	$FTab($	$F$	$Fab)$
$T'$	$T$	$TFab($	$T$	$TFab)$
$F$	$ab($	$ab($	$ab)$	$ab)$

(a)



(b)

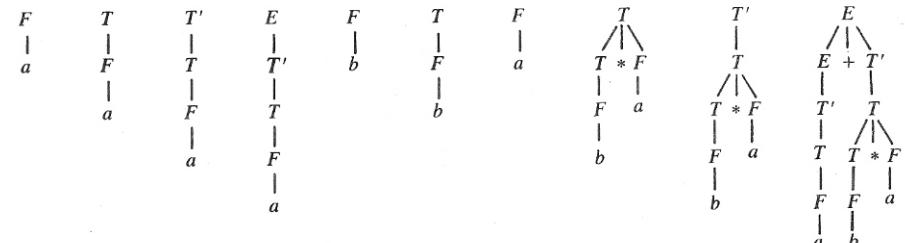
$E$	$E'$	$T$	$T'$	$F$	$a$	$b$	$+$	$*$	$($	$)$
$E$							$\doteq$			$>$
$E'$										$\doteq$
$T$							$>$	$\doteq$		$>$
$T'$							$>$			$>$
$F$							$>$	$>$		$>$
$a$							$>$	$>$		$>$
$b$							$>$	$>$		$>$
$+$					$\lessdot$	$\doteq$	$\lessdot$	$\lessdot$	$\lessdot$	
$*$							$\doteq$	$\lessdot$		
$($									$\lessdot$	
$)$										$>$

(c)

Fig. 3.8

PASSO	FORMA SENTENCIAL	REDUTENDO	REDUÇÃO PARA
1	$a \geq + b * a$	$a$	$F$
2	$F \geq + b * a$	$F$	$T$
3	$T \geq + b * a$	$T$	$T'$
4	$T' \geq + b * a$	$T'$	$E$
5	$E \doteq + \langle b \rangle * a$	$b$	$F$
6	$E \doteq + \langle F \rangle * a$	$F$	$T$
7	$E \doteq + \langle T \doteq * \rangle a$	$a$	$F$
8	$E \doteq + \langle T \doteq * \rangle \doteq F$	$T * F$	$T$
9	$E \doteq + \langle T \doteq * \rangle$	$T$	$T'$
10	$E \doteq + \doteq T'$	$E + T'$	$E$
11	$E$		

(a)



(1) (2) (3) (4) (5) (6) (7) (8) (9) (10)

(b)

Fig. 3.9

comprimento da cadeia de entrada, mas a constante de proporcionalidade pode ser bastante alta em comparação com outros métodos. Finalmente, existem muitas linguagens de programação para as quais é difícil ou impossível obter-se uma gramática de precedência simples.

Existem vários métodos derivados do descrito nesta seção, e indicaremos alguns deles sem entrar nos pormenores. Um método, chamado de *análise de precedência fraca*, permite que se tenham símbolos  $X$  e  $Y$  tais que  $X \lessdot Y$  e  $X \doteq Y$  simultaneamente. A extremidade direita de um redutendo ainda é localizada através da relação  $\geq$ . Em seguida, procura-se uma produção cujo lado direito coincida com os símbolos no topo da pilha. Caso haja mais de uma, escolhe-se a mais comprida entre as aplicáveis. Pode-se demonstrar que, para uma certa classe de

### procedimento ANÁLISE DE PRECEDÊNCIA SIMPLES;

início

$P[1]:='\#'; i:=1; \text{término}:=\text{falso};$

PRÓXIMO;

repita

enquanto  $M[P[i],\text{símbolo}] \in \{\langle, \rangle\}$

faz { $i:=i+1; P[i]:=\text{símbolo}; \text{PRÓXIMO};$

se  $M[P[i],\text{símbolo}] \neq \rangle$  então ERRO;

$j:=i;$

enquanto  $M[P[j-1],P[j]] = \langle \rangle$

faz { $j:=j-1;$

se "existe uma produção da forma  $A ::= P[j] \dots P[i]$ "

então { $i:=j; P[j]:=A;$

senão  $\text{término}:=\text{verdadeiro}$

até  $\text{término};$

se  $P[1]P[2]P[3] \neq \#\#S\#$  então ERRO

figm

Fig. 3.10

gramáticas, este procedimento sempre erra, já que é o redutendo da forma sentencial direita. Este é, por exemplo, o caso da gramática do Exemplo 3.5. Obviamente, esta classe contém a classe de gramáticas de precedência simples.

Um outro método que tem sido usado é a chamada *análise de precedência estendida*. Neste método, as relações  $<$  e  $>$  são definidas sobre conjuntos de cadeias de um certo comprimento máximo, tipicamente 2. Assim podemos ter, por exemplo,  $XY < Z$  ou  $WX > YZ$ . Com estas definições, eliminam-se alguns conflitos, pois o algoritmo usa mais informação sobre a cadeia dada. Um problema óbvio desse método é o crescimento maior ainda das tabelas que representam as relações. Este método pode ser combinado com os anteriores, e usado apenas para resolver os conflitos.

### 3.3 ANÁLISE DE PRECEDÊNCIA DE OPERADORES

A *análise de precedência de operadores* é um método que generaliza a técnica de se atribuir níveis de precedência aos operadores em expressões (por exemplo, a atribuição ao símbolo  $*$  de um nível mais alto do que ao símbolo  $+$ ). O método funciona para uma classe de linguagens mais restrita do que o de precedência simples, mas é muito eficiente e simples de implementar. Tem sido usado principalmente para a análise sintática de expressões, sendo combinado com outros métodos para tratar das construções restantes de linguagem.

Este método também utiliza três relações de precedência, que indicaremos por  $\triangleleft$ ,  $\triangleq$  e  $\triangleright$ , mas definidas sobre o conjunto de símbolos terminais, chamados também, neste contexto, de *operadores*. Com isto haverá uma redução significativa no tamanho das tabelas usadas para guardar as relações. A parte da forma sentencial a ser reduzida será determinada verificando-se a relação entre terminais consecutivos e ignorando-se os não-terminais intermediários. Conseqüentemente, este método não pode determinar as reduções de redutendos constituídos por não-terminais apenas. Como veremos mais adiante, isto é uma vantagem sob ponto de vista prático.

Necessitaremos de algumas definições adicionais para descrever de maneira precisa o método.  $G$  é uma *gramática de operadores* se ela não possui produções com dois não-terminais consecutivos do lado direito, ou seja, da forma  $A ::= \alpha BC\beta$ . Demonstra-se facilmente que qualquer forma sentencial de uma gramática de operadores não contém dois não-terminais consecutivos. Se  $\alpha$  é uma forma sentencial, então  $\beta$  é uma *frase prima* de  $\alpha$  se  $\beta$  é uma frase de  $\alpha$  na qual aparece pelo

menos um símbolo terminal, e se  $\beta$  não contém nenhuma outra frase prima. Como veremos mais adiante, o algoritmo de análise de precedência de operadores sempre reduz a frase prima mais à esquerda da forma sentencial. Pode-se demonstrar que para gramáticas não-ambíguas esta frase é sempre única.

#### Exemplo 3.7

Consideremos ainda a mesma gramática de expressões

$$E ::= E + T \mid T$$

$$T ::= T * F \mid F$$

$$F ::= a \mid b \mid (E)$$

As frases primas da forma sentencial  $a + b * a$  são:  $a$ ,  $b$  e  $a$ . Para a forma  $T + T * F + a$ , temos as frases primas:  $T * F$  e  $a$ . Note-se que  $T$  não é uma frase prima, apesar de ser uma frase simples.

Definiremos agora as relações de precedência  $\triangleleft$ ,  $\triangleq$  e  $\triangleright$  sobre  $T$ :

1. Dizemos que  $X \triangleleft Y$  se existe uma forma sentencial direita  $\alpha = \beta XY\gamma w$  (ou  $\alpha = \beta XBY\gamma w$ ), tal que  $Y\gamma$  (ou  $B\gamma$ ) é uma frase prima mais à esquerda de  $\alpha$ .
2. Dizemos que  $X \triangleq Y$  se existe uma forma sentencial direita  $\alpha = \beta\gamma XY\delta w$  (ou  $\alpha = \beta\gamma XBY\delta w$ ), tal que  $\gamma XY\delta$  (ou  $\gamma XBY\delta$ ) é uma frase prima mais à esquerda de  $\alpha$ .
3. Dizemos que  $X \triangleright Y$  se existe uma forma sentencial direita  $\alpha = \beta\gamma XYw$  (ou  $\alpha = \beta\gamma XBYw$ ), tal que  $\gamma X$  (ou  $\gamma XB$ ) é uma frase prima mais à esquerda de  $\alpha$ .

(Aqui  $X, Y \in T, B \in N, \alpha, \beta, \gamma, \delta \in \Sigma^*$  e  $w \in T^*$ .) Uma gramática de operadores  $G$  é uma gramática de precedência de operadores se as relações  $\triangleleft$ ,  $\triangleq$ ,  $\triangleright$  correspondentes são disjuntas.

A Figura 3.11 indica de maneira esquemática as implicações da definição acima. Nesta figura, o nó com rótulo  $A$  (que pode ser a própria raiz  $S$ ) é o antepassado

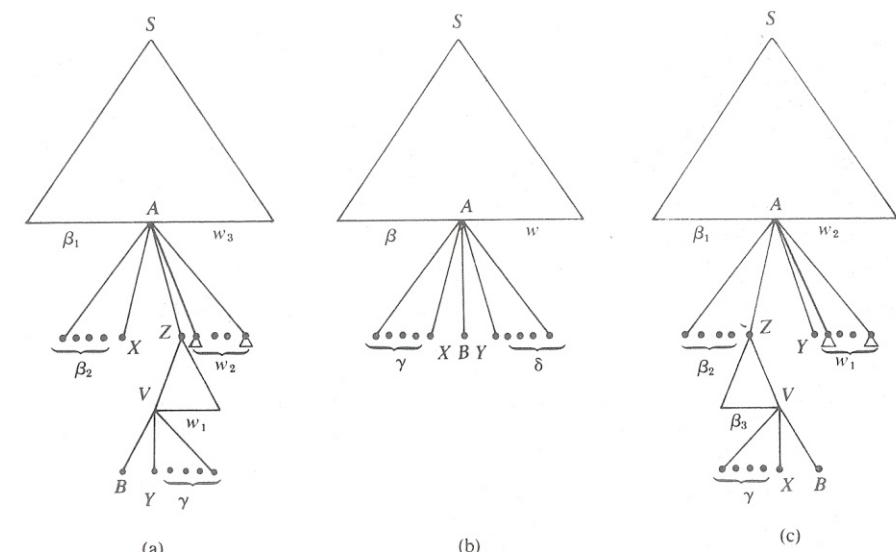


Fig. 3.11

comum mais baixo dos nós de rótulos  $X$  e  $Y$ ; todas as ocorrências do não-terminal  $B$  podem não existir. Na Figura 3.11a ( $X \triangleleft Y$ ), temos  $\beta = \beta_1\beta_2$  e  $w = w_1w_2w_3$ ; os nós com rótulos  $V$  e  $Z$  podem coincidir. Na Figura 3.11c ( $X \triangleright Y$ ),  $\beta = \beta_1\beta_2\beta_3$  e  $w = w_1w_2$ ; os nós  $V$  e  $Z$  também podem coincidir.

A Figura 3.11 indica que as relações de precedência de operadores também podem ser calculadas num número finito de operações. Para expressar este fato, definiremos mais duas relações auxiliares de  $N$  para  $T$  ( $A, B \in N$  e  $X \in T$ ):

1.  $A\theta_p X$  (primeiro terminal) se e somente se existe uma produção da forma  $A ::= X\alpha$  ou  $A ::= BX\alpha$ .
2.  $A\theta_v X$  (último terminal) se e somente se existe uma produção da forma  $A ::= \alpha X$  ou  $A ::= \alpha XB$ .

As relações  $\theta_p$  e  $\theta_v$  indicam o primeiro e o último terminal que aparecem numa cadeia que pode ser derivada diretamente de um não-terminal.

Utilizando-se estas relações, e as definidas anteriormente, podemos demonstrar então a proposição:

1.  $X \triangleleft Y$  se e somente se  $X(\doteq)\psi_p^*\theta_p Y$ .
2.  $X \triangleq Y$  se e somente se existe uma produção da forma  $A ::= \alpha XY\beta$  ou  $A ::= \alpha XBY\beta$ .
3.  $X \triangleright Y$  se e somente se  $X(\psi_v^*\theta_v)^T(\doteq)Y$ .

Note-se que a relação  $\doteq$  usada em 1 e 3 é a definida para a precedência simples na Seção 3.2, e que  $(\psi_i^*\theta_i)^T = \theta_i^T\psi_i^{*T}$ .

Pode-se demonstrar que as relações  $\triangleleft$ ,  $\triangleq$  e  $\triangleright$  identificam corretamente a frase prima mais à esquerda de uma forma sentencial, ou seja, que dada uma forma sentencial direita  $\alpha = \Lambda_0 Y_1 \Lambda_1 Y_2 \Lambda_2 \dots Y_{n-1} \Lambda_{n-1} Y_n \Lambda_n$ , com  $Y_i \in T$  e  $\Lambda_i = \lambda$  ou  $\Lambda_i \in N$ , cuja frase prima mais à esquerda é  $\Lambda_{k-1} Y_k \Lambda_k \dots \Lambda_{m-1} Y_m \Lambda_m$ ,  $1 \leq k \leq m \leq n$ , tem-se:

$$\begin{aligned} Y_i \triangleleft Y_{i+1} \text{ ou } Y_i &\stackrel{\triangleq}{=} Y_{i+1} & \text{para } i = 1, \dots, k-2 \\ Y_{k-1} \triangleleft Y_k & & \text{se } k > 1 \\ Y_i &\stackrel{\triangleq}{=} Y_{i+1} & \text{para } i = k, \dots, m-1 \\ Y_m \triangleright Y_{m+1} & & \text{se } m < n \end{aligned}$$

### Exemplo 3.8

Consideremos a nossa gramática de expressões do Exemplo 3.7. Calculando as relações  $\psi_p^*$ ,  $\psi_v^*$ ,  $\theta_p$  e  $\theta_v$ , e os produtos  $\psi_p^*\theta_p$  e  $\psi_v^*\theta_v$ , temos as tabelas da Figura 3.12a. A Figura 3.12b indica uma maneira prática de calcular as relações  $\triangleleft$  e  $\triangleright$ . No topo de cada tabela colocam-se os símbolos  $XAY$ ,  $X, Y \in T$  e  $A \in N$ , se a sequência  $XAY$  ocorre do lado direito de alguma produção. Um dos símbolos  $X$  ou  $Y$  pode não existir, se a sequência ocorre no início ou no fim do lado direito da produção. Na primeira coluna colocam-se todos os símbolos  $V$  tais que  $A\psi_v^*\theta_v V$ ; não é necessário colocar estes símbolos caso  $Y$  não exista. Na segunda coluna colocam-se todos os símbolos  $V$  tais que  $A\psi_p^*\theta_p V$ ; não é necessário colocar estes símbolos caso  $X$  não exista. A relação  $\triangleleft$  é dada por todos os pares  $(X, V)$  tais que  $V$  ocorra na segunda coluna; a relação  $\triangleright$  é dada por todos os pares  $(V, Y)$  tais que  $V$  ocorra na primeira coluna. A Figura 3.12c apresenta a tabela de precedências obtida. Conclui-se que a gramática é de precedência de operadores, apesar de não ser de precedência simples.

Consideremos agora a sentença  $a + (b * a)$ . A Figura 3.13 indica os passos de redução usando as precedências da Figura 3.12c. Foi incluído o delimitador  $\#$  com precedências definidas convenientemente.

	$\psi_p$	$\psi_p^*$	$\theta_p$	$\psi_p^*\theta_p$
E	TE	$TEFab($	+	$*ab($
T	FT	$FTab($	*	$ab(*)$
F	ab(	$ab(F$	$ab($	$ab($

	$\psi_u$	$\psi_u^*$	$\theta_u$	$\psi_u^*\theta_u$
E	T	$TFab)E$	+	$*ab)+$
T	F	$Fab)T$	*	$ab)*$
F	ab)	$ab)F$	$ab)$	$ab)$

(a)

$E +$	$+ T$	$T *$	$* F$	$( E )$
*	a	a	a	*
a	b	b	b	a+
b	(	)	(	b a
)	*	*	*	) b
+				+

(b)

$a$	$b$	$+$	$*$	$($	$)$
		$\triangleleft$	$\triangleleft$	$\triangleleft$	$\triangleleft$
		$\triangleleft$	$\triangleleft$	$\triangleleft$	$\triangleleft$
$\triangleleft$	$\triangleleft$	$\triangleleft$	$\triangleleft$	$\triangleleft$	$\triangleleft$
$\triangleleft$	$\triangleleft$	$\triangleleft$	$\triangleleft$	$\triangleleft$	$\triangleleft$
$\triangleleft$	$\triangleleft$	$\triangleleft$	$\triangleleft$	$\triangleleft$	$\triangleleft$
$\triangleleft$	$\triangleleft$	$\triangleleft$	$\triangleleft$	$\triangleleft$	$\triangleleft$

(c)

Fig. 3.12

O exemplo acima ilustra vários aspectos importantes desse método. Em primeiro lugar, a identificação da produção a ser aplicada em cada redução é baseada apenas nos terminais que aparecem na frase prima. Os não-terminais indicam os lugares onde deve aparecer algum não-terminal, não necessariamente o mesmo indicado pela produção. Conseqüentemente, a árvore não é realmente uma árvore

PASSO	FORMA SENTENCIAL	FRASE PRIMA	REDUÇÃO PARA
1	# $\triangleleft a \triangleright + (b * a) \#$	$a$	$F$
2	# $\triangleleft F + \triangleleft (\triangleleft b \triangleright * a) \#$	$b$	$F$
3	# $\triangleleft F + \triangleleft (\triangleleft F * \triangleleft a \triangleright) \#$	$a$	$F$
4	# $\triangleleft F + \triangleleft (\triangleleft F * F \triangleright) \#$	$F * F$	$T$
5	# $\triangleleft F + \triangleleft (\triangleleft T \triangleright) \#$	$(T)$	$F$
6	# $\triangleleft F + F \triangleright \#$	$F + F$	$E$
7	# $E \#$		

(a)

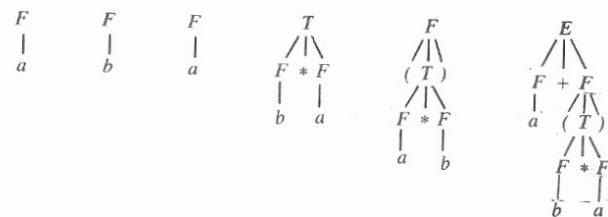
(1) (2) (3) (4) (5) (6)  
(b)

Fig. 3.13

PASSO	FORMA SENTENCIAL	FRASE PRIMA	REDUÇÃO PARA
1	# $\triangleleft a \triangleright + b * a \#$	$a$	$E$
2	# $\triangleleft E + \triangleleft b \triangleright * a \#$	$b$	$E$
3	# $\triangleleft E + \triangleleft E * \triangleleft a \triangleright \#$	$a$	$E$
4	# $\triangleleft E + \triangleleft E * E \triangleright \#$	$E * E$	$E$
5	# $\triangleleft E + E \triangleright \#$	$E + E$	$E$
6	# $E \#$		

(a)

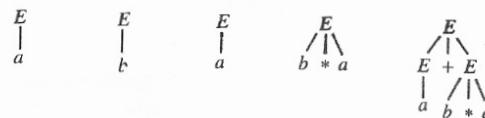
(1) (2) (3) (4) (5)  
(b)

Fig. 3.14

de derivação de acordo com a gramática dada. Entretanto, ela tem uma estrutura semelhante, tendo sido eliminadas as derivações diretas baseadas em  $E ::= T$  e  $T ::= F$ , que apenas mudam o nome do não-terminal. Acontece que na prática este tipo de reduções não corresponde a nenhuma ação de outras partes do compilador. Consequentemente, ficam eliminadas várias reduções supérfluas, aumentando a eficiência da análise.

Note-se que a gramática do último exemplo foi introduzida justamente para associar precedências aos operadores. Uma vez que as relações de precedência foram calculadas, podemos voltar à gramática ambígua dada no Capítulo 2:

$$E ::= a \mid b \mid E+E \mid E*E \mid (E)$$

A Figura 3.14 indica a análise da sentença  $a + b * a$ , usando a gramática acima e as relações da Figura 3.12c.

O programa que implementa a análise de precedência de operadores é semelhante ao dado para precedência simples, e será deixado como exercício. Deve-se notar que um algoritmo de análise para gramáticas de precedência de operadores, e que funciona da maneira indicada no Exemplo 3.8, pode, no caso de certas gramáticas, aceitar cadeias que não pertencem à linguagem (veja Exercício 3.15).

### 3.4 ANÁLISE LR( $k$ )

Os algoritmos de precedência simples e de precedência de operadores discutidos nas seções anteriores são exemplos de uma classe de algoritmos denominados analisadores de deslocamento ou redução<sup>1</sup>. Nestes analisadores, os símbolos são deslocados para a pilha até que seja encontrado um redutendo (ou uma frase prima), quando então ocorre a redução. O processo continua até que seja encontrado um erro, ou então até que a cadeia de entrada seja aceita.

Um estudo do algoritmo de precedência simples na Figura 3.10 revela que ele faz muito pouco uso da informação que poderia ser obtida a partir dos símbolos já processados e armazenados na pilha. Nesta seção estudaremos um método mais geral de análise e que aproveita melhor as informações obtidas a partir dos símbolos já processados. A idéia básica do método a ser discutido é que, ao invés de deslocar para a pilha símbolos da linguagem (terminais e não-terminais), serão empilhados outros símbolos, chamados *estados*. A cada passo do algoritmo, o último estado empilhado será utilizado para tomar decisões sobre um eventual deslocamento ou redução. Além disso, o algoritmo poderá consultar um número  $k$  preestabelecido ( $k \geq 0$ ) de símbolos seguintes da cadeia de entrada para tomar essas decisões. Este tipo de algoritmo será denominado analisador  $LR(k)$ , indicando uma análise da esquerda para a direita, resultando a derivação direita,<sup>2</sup> e consultando  $k$  símbolos seguintes de entrada.<sup>3</sup> Neste texto discutiremos apenas os casos  $k=0$  e  $k=1$ .

Na implementação usual da análise  $LR(k)$  determina-se, a partir da gramática, uma tabela de análise que será usada pelo algoritmo; o programa analisador propriamente dito independe da gramática. A tabela de análise é uma matriz retangular cujas linhas são indexadas pelos estados, e as colunas pelos símbolos do vocabulário da gramática. Os elementos da matriz indicam as ações a serem tomadas pelo algoritmo, e que podem ser:

<sup>1</sup>Em inglês: *shift-reduce parsers*.

<sup>2</sup>Em inglês: *Left-to-right parsing producing Rightmost derivation*.

<sup>3</sup>Em geral, podem ser incluídas nessa classe gramáticas com produções da forma  $A ::= \lambda$ .

```

procedimento ANÁLISE LR;
início
  P[1]:='e0'; i:=1; término:=falso; reduzido:=falso;
  PRÓXIMO;
  repita
    se reduzido
      então s:=símbolo reduzido
      senão s:=símbolo;
      caso M[P[i],s] de
        empilhar ('ei'):
          {i:=i+1; P[i]:='ej';
          se reduzido então reduzido:=falso
          senão PRÓXIMO};
        reduzir ('A ::= α'):
          {i:=i-1|'α'|; reduzido:=verdadeiro;
          símbolo reduzido:='A'};
        aceitar: término:=verdadeiro;
        rejeitar: ERRO
      fim do caso
      até término
    fim

```

Fig. 3.15

1. Empilhar um estado  $e_i$ .
2. Reduzir usando uma produção  $A ::= \alpha$ .
3. Aceitar.
4. Rejeitar.

A Figura 3.15 mostra um programa que implementa a análise LR(0) e LR(1) usando uma tabela  $M$  como descrito acima. A diferença entre LR(0) e LR(1) está na maneira de construir a tabela. Note-se que o próximo símbolo consultado pode ser o próximo símbolo de entrada (terminal), ou então o não-terminal para o qual houve redução no passo anterior. Tudo se passa, portanto, como se esse não-terminal fosse acrescentado ao início da cadeia de entrada. Os procedimentos *PRÓXIMO* e *ERRO* seguem as mesmas convenções anteriores;  $e_0$  denota o estado inicial.

#### Exemplo 3.9

Consideremos a gramática abaixo cujas produções foram numeradas para fins de referência:

1.  $E ::= +EE$
2.  $E ::= *EE$
3.  $E ::= a$
4.  $E ::= b$

A Figura 3.16 indica uma tabelade análise para esta gramática (veremos mais adiante como ela foi obtida). Supusemos que o símbolo  $\#$  é usado para marcar o fim da cadeia. As ações são indicadas por  $e_i$  (empilhar o estado  $e_i$ ),  $r_i$  (reduzir usando a  $i$ -ésima produção) e  $a$  (aceitar); as entradas em branco indicam rejeição.

A Figura 3.17 indica os passos de análise que seriam executados pelo algoritmo de análise LR usando esta tabela, quando aplicado à cadeia  $+*a+baa$ . A fim de tornar o processo de análise mais claro, os estados empilhados são indicados por  $X_i$ , significando que o empilhamento do estado  $e_i$  foi causado pelo símbolo  $X$  do vocabulário.

$E$	+	*	$a$	$b$	$\#$
$e_0$	$e_1$	$e_2$	$e_3$	$e_4$	$e_5$
$e_1$					$a$
$e_2$	$e_6$	$e_2$	$e_3$	$e_4$	$e_5$
$e_3$	$e_7$	$e_2$	$e_3$	$e_4$	$e_5$
$e_4$		$r_3$	$r_3$	$r_3$	$r_3$
$e_5$		$r_4$	$r_4$	$r_4$	$r_4$
$e_6$	$e_8$	$e_2$	$e_3$	$e_4$	$e_5$
$e_7$	$e_9$	$e_2$	$e_3$	$e_4$	$e_5$
$e_8$		$r_1$	$r_1$	$r_1$	$r_1$
$e_9$		$r_2$	$r_2$	$r_2$	$r_2$

Fig. 3.16

A seqüênciados números usados nas reduções (em ordem inversa) produz a derivação direita da cadeia dada ( $\overset{i}{\Rightarrow}$  indica a aplicação da  $i$ -ésima produção):

$$\begin{aligned}
E &\overset{1}{\Rightarrow} +EE \overset{3}{\Rightarrow} +Ea \overset{2}{\Rightarrow} +*EEa \overset{1}{\Rightarrow} +*E+EEa \overset{3}{\Rightarrow} +*E+Eaa \overset{4}{\Rightarrow} \\
&\quad +*E+baa \overset{3}{\Rightarrow} +*a+baa
\end{aligned}$$

Podemos observar neste último exemplo que cada estado contém informação sobre o símbolo do vocabulário que causou o deslocamento para a pilha, pois note-se que nenhum estado  $e_i$  ocorre em mais de uma coluna da tabela. Por outro lado, os estados contêm mais informação, pois ao mesmo símbolo  $E$  correspondem os estados  $e_1$ ,  $e_6$ ,  $e_7$ ,  $e_8$  e  $e_9$ . Um estudo da Figura 3.17 dá uma idéia intuitiva do significado desses estados. Assim, por exemplo, a presença do estado  $e_8$  no topo da pilha parece indicar que foi encontrado um redutendo da forma  $+EE$ . O estado  $e_7$  indica, por sua vez, que foi encontrada a parte  $*E$  do redutendo  $*EE$ . Em geral, cada estado parece indicar uma parte inicial, ou então todo o redutendo já encontrado.

Note-se que a tabela da Figura 3.16 indica que se trata de uma gramática do tipo LR(0), pois quando o estado no topo da pilha é  $e_4$ ,  $e_5$ ,  $e_8$  ou  $e_9$ , pode-se efetuar uma redução sem consultar o símbolo seguinte. (Veja o Exercício 3.18 para entender por que as entradas da tabela correspondentes a esses estados e à coluna  $E$  podem ficar em branco.)

PASSO	PILHA	SÍMBOLO REDUZIDO	CADEIA DE ENTRADA	AÇÃO
0	$e_0$		$+*a+baa\#$	$e_2$
1	$e_0 +_2$		$*a+baa\#$	$e_3$
2	$e_0 +_2*_3$		$a+baa\#$	$e_4$
3	$e_0 +_2*_3a_4$		$+baa\#$	$r_3$
4	$e_0 +_2*_3$	$E$	$+baa\#$	$e_7$
5	$e_0 +_2*_3E_7$		$+baa\#$	$e_2$
6	$e_0 +_2*_3E_7 +_2$		$baa\#$	$e_5$
7	$e_0 +_2*_3E_7 +_2b_5$		$aa\#$	$r_4$
8	$e_0 +_2*_3E_7 +_2$	$E$	$aa\#$	$e_6$
9	$e_0 +_2*_3E_7 +_2E_6$		$aa\#$	$e_4$
10	$e_0 +_2*_3E_7 +_2E_6a_4$		$a\#$	$r_3$
11	$e_0 +_2*_3E_7 +_2E_6$	$E$	$a\#$	$e_8$
12	$e_0 +_2*_3E_7 +_2E_6E_8$		$a\#$	$r_1$
13	$e_0 +_2*_3E_7$	$E$	$a\#$	$e_9$
14	$e_0 +_2*_3E_7E_9$		$a\#$	$r_2$
15	$e_0 +_2$	$E$	$a\#$	$e_6$
16	$e_0 +_2E_6$		$a\#$	$e_4$
17	$e_0 +_2E_6a_4$		$\#\#$	$r_3$
18	$e_0 +_2E_6$	$E$	$\#\#$	$e_8$
19	$e_0 +_2E_6E_8$		$\#\#$	$r_1$
20	$e_0$	$E$	$\#\#$	$e_1$
21	$e_0E_1$		$\#\#$	$a$

Fig. 3.17

Tentaremos agora tornar precisas estas idéias intuitivas para podermos construir tabelas de análise do tipo LR(0). Veremos posteriormente como obter tabelas do tipo LR(1).

Introduziremos inicialmente algumas definições. Um *item* é uma produção na qual foi marcada uma posição na cadeia do lado direito; essa posição será indicada por meio do símbolo  $\bullet$ .

### Exemplo 3.10

O conjunto de itens derivados da gramática do Exemplo 3.9 é dado por (continuamos a utilizar a barra vertical para abreviar a notação):

$$\begin{aligned} E ::= & \bullet +EE \mid +\bullet EE \mid +E\bullet E \mid +EE\bullet \mid \\ & \bullet *EE \mid *\bullet EE \mid *E\bullet E \mid *EE\bullet \mid \\ & \bullet a \mid a\bullet \mid \bullet b \mid b\bullet \end{aligned}$$

Note-se que o conjunto de itens de uma gramática é sempre finito. Os itens serão usados para construir estados. A presença, no topo da pilha, de um estado contendo um item da forma  $A ::= \alpha\bullet\beta$  indica que já foi processada e deslocada para a pilha a parte inicial  $\alpha$  do redutendo  $\alpha\beta$ . O símbolo  $\bullet$  no fim de um item, isto é,  $\beta = \lambda$ , indica um redutendo completo; neste caso diremos que o item é *completo*.

Suponhamos agora que um estado contém um item da forma  $A ::= \alpha\bullet B\beta$ , com  $A, B \in N$  e  $\alpha, \beta \in \Sigma^*$ . A presença desse estado no topo da pilha, indicará que já foi empilhada a cadeia  $\alpha$ , sendo esperada a seguir a cadeia  $B\beta$ . Isto significa que a cadeia de entrada corrente (incluindo o eventual resultado da redução anterior) poderá ser da forma  $\gamma w$ , com  $\gamma \in \Sigma^*$ ,  $w \in T^*$  e  $B\beta \stackrel{*}{\not\rightarrow} \gamma$ ; desta maneira as reduções subsequentes poderão transformar  $\gamma$  em  $B\beta$ . Conseqüentemente, a presença do estado contendo  $A ::= \alpha\bullet B\beta$  no topo da pilha implica que o algoritmo deverá aceitar a seguir qualquer cadeia derivável de  $B$ , ou em outras palavras, esse estado deverá conter também todos os itens da forma  $B ::= \bullet\xi$ .

Diremos que um conjunto  $K$  de itens é *fechado* se para todo item de  $K$  da forma  $A ::= \alpha\bullet B\beta$  todos os itens da forma  $B ::= \bullet\xi$  estão em  $K$ . Denotaremos por  $fecho(K)$  o menor conjunto fechado que contém  $K$ . Dado um conjunto  $K$ ,  $fecho(K)$  pode ser calculado de uma maneira muito simples:

1. Adota-se o conjunto dado  $K$  como valor inicial de  $H$ .
2. Se existe um item  $A ::= \alpha\bullet B\beta$  de  $H$  e uma produção  $B ::= \xi$  tal que  $B ::= \bullet\xi$  não está em  $H$ , acrescenta-se  $B ::= \bullet\xi$  a  $H$ .
3. O passo 2 é repetido até que não se possam acrescentar mais itens a  $H$ .
4.  $H = fecho(K)$ .

### Exemplo 3.11

Consideremos os seguintes conjuntos de itens da gramática do Exemplo 3.9:

$$\begin{aligned} K_1 &= \{E ::= +\bullet EE\} \\ K_2 &= \{E ::= +E\bullet E \mid *\bullet EE \mid \bullet a\} \\ K_3 &= \{E ::= \bullet b\} \end{aligned}$$

Então os seus fechos são

$$\begin{aligned} fecho(K_1) &= \{E ::= +\bullet EE \mid \bullet +EE \mid \bullet *EE \mid \bullet a \mid \bullet b\} \\ fecho(K_2) &= \{E ::= +E\bullet E \mid *\bullet EE \mid *E\bullet E \mid *EE\bullet \mid \bullet a \mid \bullet b\} \\ fecho(K_3) &= \{E ::= \bullet b\}. \end{aligned}$$

A discussão anterior implica que um estado deve ser um conjunto fechado de itens. Cada item desse estado representa uma das possibilidades para continuar a análise.

Examinemos agora o problema de determinar as entradas da forma  $e_i$  da tabela de análise. Suponhamos então que um estado  $e_i$  contenha um item incompleto da forma  $A ::= \alpha \bullet X\beta$ ,  $X \in \Sigma$ . A presença desse estado  $e_i$  no topo da pilha indica que se o próximo símbolo a ser consultado for  $X$ , então terá sido processada a parte  $\alpha X$  do redutendo, devendo ser empilhado, portanto, um estado que contém o item  $A ::= \alpha X \bullet \beta$ . Definiremos então a função  $transfere(K, X)$  como sendo o fecho do conjunto de todos os itens da forma  $A ::= \alpha X \bullet \beta$  tais que o item  $A ::= \alpha X\beta$  está em  $K$ .

### Exemplo 3.12

Consideremos a gramática do Exemplo 3.9, e os conjuntos:

$$\begin{aligned} K_1 &= \{E ::= * \bullet EE\} \\ K_2 &= fecho(K_1) \\ K_3 &= \{E ::= \bullet b\} \end{aligned}$$

Tem-se, então:

$$\begin{aligned} transfere(K_1, *) &= \emptyset \\ transfere(K_1, E) &= \{E ::= *E \bullet E | \bullet +EE | \bullet *EE | \bullet a | \bullet b\} \\ transfere(K_2, +) &= \{E ::= + \bullet EE | \bullet +EE | \bullet *EE | \bullet a | \bullet b\} \\ transfere(K_2, a) &= \{E ::= a \bullet\} \\ transfere(K_3, E) &= \emptyset \\ transfere(K_3, b) &= \{E ::= b \bullet\} \end{aligned}$$

As funções *fecho* e *transfere* permitem a construção dos estados que serão identificados com conjuntos de itens. A construção partirá de um estado inicial, e aplicando as funções *fecho* e *transfere* calculará novos estados, até que não seja possível prosseguir na construção. Uma vez que o número de itens é finito, o número de estados distintos também é. A fim de introduzir o estado inicial  $e_0$ , convencionaremos que a toda gramática de raiz  $S$  acrescenta-se a produção  $S' ::= S \#$ , onde o símbolo  $\#$  não está em  $T$ ; o símbolo  $S'$  passa a ser a nova raiz. Convencionaremos, também, que toda cadeia de entrada será seguida do símbolo  $\#$ . Fica claro, dessa maneira, que o estado  $e_0$  deve conter o item  $S' ::= \bullet S \#$ . Por outro lado, se um estado que contém  $S' ::= S \bullet \#$  estiver no topo da pilha quando o símbolo seguinte a ser consultado é  $\#$ , então a cadeia de entrada deverá ser aceita.

A discussão acima leva à seguinte formulação do algoritmo para determinar os estados de uma gramática ( $C$  é uma coleção de estados, ou seja, um conjunto de conjuntos de itens):

1. Adota-se o estado  $e_0 = \{fecho(\{S' ::= \bullet S \#\})\}$  como o valor inicial da coleção  $C$ .
2. Se existe um estado  $e$  de  $C$ , e um símbolo  $X \in \Sigma$  tais que  $e' = transfere(e, X)$  não é vazio, e  $e'$  não está em  $C$ , então acrescenta-se  $e'$  à coleção  $C$ .
3. O passo 2 é repetido até que não se possam acrescentar mais estados à coleção  $C$ .
4.  $C$  é a coleção de estados tipo LR(0) da gramática.

### Exemplo 3.13

Consideremos a gramática:

0.  $E' ::= E \#$
1.  $E ::= +EE$
2.  $E ::= *EE$
3.  $E ::= a$
4.  $E ::= b$

Aplicando-se a construção acima, obtém-se os seguintes estados (omitimos as chaves que indicam conjuntos):

$$\begin{aligned} e_0: E' &::= \bullet E \# \\ E &::= \bullet +EE | \bullet *EE | \bullet a | \bullet b \\ e_1: E' &::= E \bullet \# \\ e_2: E &::= + \bullet EE | \bullet +EE | \bullet *EE | \bullet a | \bullet b \\ e_3: E &::= * \bullet EE | \bullet +EE | \bullet *EE | \bullet a | \bullet b \\ e_4: E &::= a \bullet \\ e_5: E &::= b \bullet \\ e_6: E &::= +E \bullet E | \bullet +EE | \bullet *EE | \bullet a | \bullet b \\ e_7: E &::= *E \bullet E | \bullet +EE | \bullet *EE | \bullet a | \bullet b \\ e_8: E &::= +EE \bullet \\ e_9: E &::= *EE \bullet \end{aligned}$$

Não foi incluído o cálculo de  $transfere(e_1, \#)$ , pois, como foi dito anteriormente, esta situação corresponde à aceitação da cadeia de entrada. Ao calcularrem-se os estados, foi determinada também a função *transfere*; os valores dessa função fornecem as entradas da forma  $e_i$  da tabela da Figura 3.16.

Os estados calculados no Exemplo 3.13 contêm apenas um item completo ( $e_4$ ,  $e_5$ ,  $e_8$  e  $e_9$ ), ou apenas itens incompletos ( $e_0$ ,  $e_1$ ,  $e_2$ ,  $e_3$ ,  $e_6$  e  $e_7$ ). Estes últimos, como já vimos, indicam que um estado correspondente ao próximo símbolo consultado deve ser deslocado para a pilha. Os estados constituídos de itens completos indicam que os últimos estados empilhados correspondem a um redutendo, e que portanto deve haver redução, independentemente do próximo símbolo. É essa propriedade que faz com que a gramática dada pertença à classe LR(0). Se o estado que está no topo da pilha contém um único item da forma  $A ::= \alpha \bullet$ , então deve-se aplicar uma redução utilizando a produção  $A ::= \alpha$ . As entradas  $r_i$  da tabela da Figura 3.16 foram obtidas dessa maneira.

Veremos a seguir um exemplo de gramática que não é do tipo LR(0).

### Exemplo 3.14

Consideremos a gramática de expressões:

0.  $E' ::= E \#$
1.  $E ::= E + T$
2.  $E ::= T$
3.  $T ::= T * F$
4.  $T ::= F$
5.  $F ::= (E)$
6.  $F ::= a$

(A produção  $F ::= b$  foi omitida a fim de diminuir o tamanho da tabela de análise.) Calculando-se os estados correspondentes à construção dada para o tipo LR(0), obtém-se:

Existem outros métodos para construir as tabelas de análise LR, e cuja descrição pode ser encontrada na literatura citada no fim do capítulo. Um desses métodos fornece as chamadas *tabelas canônicas* para análise LR, e corresponde ao caso mais geral de gramáticas do tipo LR. Entretanto, na prática o método não é viável, pois resulta num número muito grande de estados. Um outro método, chamado *LALR*,<sup>1</sup> é menos geral do que LR, porém mais geral do que SLR, e é o método preferido para construir as tabelas.

Na prática, a determinação das tabelas de análise é bastante trabalhosa, e deve ser automatizada, usando-se um dos métodos indicados. Existem várias técnicas para diminuir a quantidade de memória necessária para armazenar as tabelas, que em geral são bastante esparsas (maior parte das entradas corresponde a casos impossíveis ou então à rejeição). Os pormenores podem ser encontrados na literatura citada.

Uma propriedade importante das gramáticas do tipo LR( $k$ ) é que elas são não-ambíguas. Uma outra propriedade é que dada uma tabela de análise LR(1) sem conflitos, o programa da Figura 3.15 aceita a linguagem gerada pela gramática da qual foi derivada a tabela.

### Exercícios

1. Indique todas as frases simples das formas sentenciais  $a*(b+a)+(a+b)*b$  e  $T*(E)+(T+F)+b$  usando a gramática do Exemplo 3.1.
2. Explique a assimetria existente nas definições de  $X \triangleleft Y$  e  $X \triangleright Y$ , isto é, por que na segunda definição deve-se ter  $Y \in T$ .
3. Demonstre as expressões dadas para as relações de precedência simples  $\triangleleft$  e  $\triangleright$  em termos de  $\triangleq$ ,  $\psi_P$  e  $\psi_V$ .
4. Suponha que, para uma gramática  $G$ , as relações de precedência simples  $\triangleleft$ ,  $\triangleq$  e  $\triangleright$  são disjuntas mas existem duas produções da forma  $A ::= \gamma$  e  $B ::= \gamma$ . Seja  $G'$  a gramática obtida a partir de  $G$ , substituindo-se  $B$  por  $A$  em todas as produções.
  - a)  $G'$  é de precedência simples?
  - b)  $G'$  gera a mesma linguagem que  $G$ ?

Justifique.

5. Determine as relações de precedência simples para a gramática:

$$\begin{aligned} C &::= a \mid \text{if } b \text{ then } D \text{ else } C \mid \text{if } b \text{ then } C \\ D &::= a \mid \text{if } b \text{ then } D \text{ else } D \end{aligned}$$

Esta gramática é de precedência simples ou então pode ser transformada numa, gerando a mesma linguagem?

6. Determine as relações de precedência simples para a gramática:

$$\begin{aligned} S &::= AB) \mid () \\ A &::= ( \\ B &::= aS \mid b \mid S \end{aligned}$$

Se necessário modifique a gramática de maneira que ela defina a mesma linguagem mas seja de precedência simples. Utilizando as relações obtidas, construa a árvore de derivação para a sentença  $(a((b)))$ .

7. Seja  $G$  uma gramática, e suponha que foram calculadas as relações de precedência simples para  $G$ . Mostre que dois símbolos quaisquer  $X$  e  $Y$  do

seu vocabulário podem aparecer consecutivos numa forma sentencial direita se e somente se o par  $(X, Y)$  pertence a uma ou mais das três relações. A afirmação é verdadeira para formas sentenciais que não sejam direitas? Justifique.

8. Indique todas as frases primas das formas sentenciais do Exercício 3.1.
9. Mostre que uma forma sentencial de uma gramática de operadores nunca contém dois não-terminais consecutivos.
10. Justifique por que as frases primas sempre incluem os não-terminais, se estes ocorrem nas suas extremidades.
11. Demonstre as expressões dadas para as relações de precedência de operadores  $\triangleleft$  e  $\triangleright$  em termos de  $\triangleq$ ,  $\psi_P$ ,  $\psi_V$ ,  $\theta_P$  e  $\theta_V$ .
12. Determine as relações de precedência de operadores para a gramática do Exercício 3.5.
13. Determine as relações de precedência de operadores para a gramática:

$$\begin{aligned} S &::= \uparrow A \rightarrow S \mid a \\ A &::= b \mid b;A \end{aligned}$$

Utilizando estas relações construa a árvore de derivação para a sentença  $\uparrow b; b; b \rightarrow \uparrow b \rightarrow a$ .

14. Escreva um programa que implementa o algoritmo de análise de precedência de operadores.
15. Considere a gramática  $G$ :

$$\begin{aligned} S &::= aA \mid bB \\ A &::= Ac \mid c \\ B &::= Bcd \mid cd \end{aligned}$$

- a) Calcule as relações de precedência de operadores para  $G$ .
- b) Considere agora a gramática  $G'$ , obtida de  $G$ , substituindo-se todos os não-terminais por  $S$ :

$$S ::= aS \mid bS \mid Sc \mid Scd \mid c \mid cd$$

Ache uma cadeia  $\alpha$  tal que  $\alpha \notin L(G)$ ,  $\alpha \in L(G')$ , e que seria aceita pelo algoritmo de análise de operadores usando as relações da parte (a).

- c) Sugira um mecanismo para evitar o fenômeno verificado na parte (b). (*Sugestão*: Inclua a verificação dos não-terminais que entram nas reduções.)

16. Uma gramática de precedência de operadores pode ser ambígua? Justifique.
17. Dada uma gramática  $G$ , seja  $G'$  a gramática obtida invertendo-se as cadeias que aparecem dos lados direitos das produções. Prove ou desprove as seguintes afirmações:
  - a) Se  $G$  é de precedência simples, então  $G'$  também o é;
  - b) Se  $G$  é de precedência de operadores, então  $G'$  também o é.
18. Justifique por que, numa tabela de análise LR, as ações que indicam reduções não podem ocorrer nas colunas correspondentes aos não-terminais.
19. Construa a tabela de análise LR para a gramática:

$$\begin{aligned} S &::= A^* \mid B\$ \\ A &::= a \\ B &::= a \end{aligned}$$

<sup>1</sup>Em inglês *LookAhead LR*.

$e_0: E' ::= \bullet E \#$
$E ::= \bullet E + T \mid \bullet T$
$T ::= \bullet T * F \mid \bullet F$
$F ::= \bullet (E) \mid \bullet a$
$e_1: E' ::= E \bullet \#$
$E ::= E \bullet + T$
$e_2: E ::= T \bullet \mid T \bullet * F$
$e_3: T ::= F \bullet$
$e_4: F ::= (\bullet E)$
$E ::= \bullet E + T \mid \bullet T$
$T ::= \bullet T * F \mid \bullet F$
$F ::= \bullet (E) \mid \bullet a$

$e_5: F ::= \bullet a$
$e_6: E ::= E + \bullet T$
$T ::= \bullet T * F \mid \bullet F$
$F ::= \bullet (E) \mid \bullet a$
$e_7: T ::= T * \bullet F$
$F ::= \bullet (E) \mid \bullet a$
$e_8: F ::= (E \bullet)$
$E ::= E \bullet + T$
$e_9: E ::= E + T \bullet$
$e_{10}: T ::= T \bullet * F$
$e_{11}: F ::= (E) \bullet$

Os valores da função *transfere* para esta gramática são indicados pelas entradas da forma  $e_j$  da tabela da Figura 3.18. Note-se que, agora, existem estados ( $e_2$  e  $e_9$ ) que contêm tanto itens completos como incompletos, mostrando que a gramática não é do tipo LR(0). (Um outro tipo de conflito que faz com que a gramática não seja do tipo LR(0) é ilustrado pelo Exercício 3.19.)

Uma gramática será do tipo LR(1) se a decisão quanto a uma redução ou um deslocamento puder ser baseada no próximo símbolo de entrada. Existem várias técnicas para construir tabelas de análise LR(1). Veremos a seguir uma técnica muito simples, mas que é aplicável a uma classe mais restrita de gramáticas do que outras técnicas. Explicaremos o uso dessa técnica através da sua aplicação à gramática do Exemplo 3.14. Tomemos, por exemplo, o estado  $e_2$ . A sua presença no topo da pilha indica que foi empilhado anteriormente um estado correspondente ao símbolo  $T$ , e que este símbolo pode constituir um redutendo, ou então pode ser a

	$E$	$T$	$F$	$a$	$+$	$*$	$($	$)$	$\#$
$e_0$	$e_1$	$e_2$	$e_3$	$e_5$			$e_4$		
$e_1$					$e_6$				$a$
$e_2$					$r_2$	$e_7$		$r_2$	$r_2$
$e_3$					$r_4$	$r_4$		$r_4$	$r_4$
$e_4$	$e_8$	$e_2$	$e_3$	$e_5$			$e_4$		
$e_5$					$r_6$	$r_6$		$r_6$	$r_6$
$e_6$		$e_9$	$e_3$	$e_5$			$e_4$		
$e_7$			$e_{10}$	$e_5$			$e_4$		
$e_8$				$e_6$			$e_{11}$		
$e_9$					$r_1$	$e_7$		$r_1$	$r_1$
$e_{10}$					$r_3$	$r_3$		$r_3$	$r_3$
$e_{11}$					$r_5$	$r_5$		$r_5$	$r_5$

Fig. 3.18

parte inicial do redutendo  $T * F$  (se o próximo símbolo for  $*$ ). Note-se, entretanto, que para esta gramática  $\Delta(E) = \{+, \#, \$\}$  (veja o Exercício 2.19, restringindo a função  $\Delta$  aos conjuntos de terminais). Em outras palavras, numa forma sentencial, o símbolo  $E$  só pode ser seguido pelos símbolos  $+, \#$  ou  $\$$ . Conseqüentemente, a redução  $E ::= T$  só pode ser aplicada se o próximo símbolo de entrada for  $+$ ,  $\#$  ou  $\$$ . Esta conclusão permite a colocação das entradas  $r_2$  na linha  $e_2$  da tabela da Figura 3.18, apenas nas colunas correspondentes aos símbolos  $+$ ,  $\#$  e  $\$$ . Não haverá conflitos, pois o símbolo  $*$  não pertence a  $\Delta(E)$ . As entradas  $r_1$ ,  $r_3$ ,  $r_4$ ,  $r_5$  e  $r_6$  foram obtidas de maneira análoga, verificando-se que  $\Delta(T) = \Delta(F) = \{+, *, \#, \$\}$ .

Uma gramática para a qual esta técnica fornece uma tabela de análise sem conflitos pertence à classe *SLR(1)*.<sup>1</sup> Veremos a seguir o exemplo de uma gramática que não é do tipo SLR(1).

### Exemplo 3.15

Consideremos a gramática:

0.  $S' ::= S \#$
1.  $S ::= A \$ B$
2.  $S ::= B$
3.  $A ::= @ B$
4.  $A ::= a$
5.  $B ::= A$

O cálculo dos estados do tipo LR(0) fornece:

$e_0: S' ::= \bullet S \#$	$e_5: A ::= a \bullet$
$S ::= \bullet A \$ B \mid \bullet B$	$e_6: S ::= A \$ \bullet B$
$A ::= \bullet @ B \mid \bullet a$	$B ::= \bullet A$
$B ::= \bullet A$	$e_7: A ::= @ B \bullet$
$e_1: S' ::= S \bullet \#$	$e_8: B ::= A \bullet$
$e_2: S ::= A \bullet \$ B$	$B ::= A \bullet$
$B ::= A \bullet$	$e_9: S ::= A \$ B \bullet$
$e_3: S ::= B \bullet$	
$e_4: A ::= @ B$	
$B ::= \bullet A$	
$A ::= \bullet @ B \mid \bullet a$	

Considerando o estado  $e_2$ , temos  $\Delta(B) = \{\#, \$\}$ ; consequentemente, se  $e_2$  está no topo da pilha e  $\$$  é o próximo símbolo, temos duas possibilidades: empilhar o estado  $e_6$  (pois  $\text{transfere}(e_2, \$) = e_6$ ), ou aplicar a redução correspondente à produção  $B ::= A$ . Conclui-se, portanto, que a gramática não é do tipo SLR(1).

O conflito do Exemplo 3.15 é causado pelo fato de utilizarmos o conjunto  $\Delta(B)$  que indica os símbolos que podem seguir  $B$  em alguma forma sentencial, independentemente da sua posição. Entretanto, uma análise daquela gramática revela que a redução indicada pelo item  $B ::= A \bullet$  do estado  $e_2$  só deve ser aplicada se inicialmente foi escolhida a alternativa  $S ::= B$  para o não-terminal  $S$ . Mas neste caso, o não-terminal  $B$  só pode ser seguido pelo símbolo  $\#$ . Entretanto,  $\Delta(B)$  inclui  $\$$ , pois  $\$$  pode seguir  $B$  em outras formas sentenciais, como por exemplo  $@B\$a\#$ . Conclui-se, então, que mesmo neste caso podemos resolver o conflito, apesar da construção do tipo SLR(1) não revelar este fato.

<sup>1</sup>Em inglês *Simple LR(1)*.

- A gramática é do tipo LR(0) ou SLR(1), ou nenhum dos dois? (Note que a linguagem descrita pela gramática é  $\{a^*, a\$ \}$ .)
20. Considere a gramática

$$\begin{aligned} E &::= TE^+ \mid TE^* \\ T &::= \$E \mid a \end{aligned}$$

- a) Construa a tabela de análise LR(0).
- b) Construa a tabela de análise SLR(1).
- c) Verifique os passos das análises da cadeia incorreta  $a^+$ , usando as duas tabelas.
- d) O que se pode concluir sobre a diferença entre análises LR(0) e LR(1) quando as duas são possíveis?

## NOTAS BIBLIOGRÁFICAS

Os analisadores de precedência de operadores foram sugeridos de maneira intuitiva por vários autores. Os primeiros tratamentos rigorosos podem ser encontrados em Paul (1962) e Floyd (1963). A precedência simples apareceu como uma generalização da precedência de operadores, sendo introduzida em Wirth e Weber (1966). A classe LR( $k$ ) de gramáticas foi definida e estudada por Knuth (1965), que introduziu o método de construção para tabelas canônicas. DeRemer (1971) introduziu a construção SLR, e LALR foi proposto por Anderson, Eve e Horning (1973). Praticamente todos os textos dedicados à compilação tratam alguns ou todos esses métodos. Em particular, recomendamos Gries (1971), Aho e Ullman (1977) e Barrett e Couch (1979). Um tratamento mais geral e completo dos métodos ascendentes pode ser encontrado em Aho e Ullman (1972), e num nível mais elementar, em Backhouse (1979). Em português, existe Moura (1982).

4

# Análise Sintática Descendente

## 4.1 GENERALIDADES

Nos algoritmos de análise sintática descendente, a construção da árvore de derivação começa pela sua raiz e procede na direção das folhas. Quando todas as folhas têm rótulos que são terminais, a fronteira da árvore deve coincidir com a cadeia dada. É claro que a construção da árvore deveria ser feita de maneira sistemática, usando sempre que possível a cadeia dada para auxiliar nas decisões do algoritmo.

Na seção seguinte descreveremos como funciona um método bastante geral de análise descendente, mas que pode ser extremamente ineficiente. Nas seções subsequentes descreveremos como o método pode ser implementado de maneira eficiente para uma classe mais restrita de gramáticas.

## 4.2 ANÁLISE DESCENDENTE COM RETROCESSO

Este método, apesar de potencialmente muito ineficiente, foi usado em algumas implementações pioneiras de sistemas automáticos para gerar analisadores sintáticos e ainda é usado para algumas aplicações especiais. O funcionamento básico do método pode ser descrito da seguinte maneira:

1. Adota-se a cadeia dada como o valor inicial de  $\alpha$ , e uma folha cujo rótulo é o símbolo inicial  $S$  da gramática, como o valor da árvore  $D$ . Esta folha é adotada também como a folha corrente.
2. Seja  $X$  o rótulo da folha corrente. Se  $X$  não é um terminal, então escolhe-se uma produção da forma  $X ::= X_1 X_2 \dots X_n$  e substitui-se, na árvore  $D$ , a folha corrente por uma árvore cuja raiz tem rótulo  $X$  e cujos descendentes diretos são folhas de rótulos  $X_1, X_2, \dots, X_n$ . A folha de rótulo  $X_1$  torna-se a nova folha corrente, e o passo 2 é repetido. Se  $X$  é um símbolo terminal, e  $\alpha = X\beta$  para alguma cadeia  $\beta$ , então adota-se  $\beta$  como o novo valor de  $\alpha$ ; a folha que segue a folha corrente em  $D$ , quando  $D$  é percorrida da esquerda para a direita, é adotada como a nova folha corrente, e o passo 2 é repetido. Se  $X$  é um terminal, e o primeiro símbolo de  $\alpha$  não é  $X$  (ou  $\alpha = \lambda$ ), então deve-se retroceder (isto é, restaurar os valores de  $\alpha$ ,  $D$  e da folha corrente) à última configuração em que foi feita a escolha de uma produção, adotando-se uma outra alternativa para o não-terminal da folha corrente. Caso não haja mais alternativas, repete-se o retrocesso até que uma seja encontrada, e volta-se a repetir o passo 2. Finalmente, se o algoritmo avança além da última folha de  $D$ , mas  $\alpha \neq \lambda$ , então deve-se retroceder como no caso anterior.
3. A análise termina quando o algoritmo avança além da última folha de  $D$  e

$\alpha = \lambda$ . Neste caso, a cadeia dada é uma sentença da linguagem. Se o algoritmo é forçado a retroceder à configuração inicial do passo 1 depois de esgotar todas as alternativas para o símbolo inicial da gramática, então a cadeia dada não pertence à linguagem.

#### Exemplo 4.1

Consideremos a gramática:

$$\begin{aligned} E &::= T+E \mid T \\ T &::= F*T \mid F \\ F &::= a \mid b \mid (E) \end{aligned}$$

Note-se que esta gramática é ligeiramente diferente da gramática usada no capítulo anterior. Consideremos também a sentença  $a*b$ . A Figura 4.1 indica os passos de análise descendente com retrocesso, como foi descrito acima. As flechas verticais indicam a folha corrente; a notação

$$p_n \Leftarrow p_{n-1} \Leftarrow \dots \Leftarrow p_1 \Leftarrow (n \geq 1)$$

indica que foi necessário retroceder através das configurações  $p_1, p_2, \dots, p_{n-1}$ , cujas alternativas já foram esgotadas, para chegar à configuração  $p_n$ , a fim de escolher uma nova alternativa.

Note-se que na Figura 4.1 os passos (1), (24), (25), (26), (34) e (36) correspondem a escolhas corretas entre as alternativas aplicáveis. Enumerando-se as fronteiras das árvores correspondentes a estes passos, obtém-se a seqüência:  $E, T, F*T, a*T, a*F, a*b$ , que é a derivação esquerda da cadeia  $a*b$ . Este fato não é surpreendente, pois o algoritmo sempre procura substituir o não-terminal mais à esquerda na árvore parcial já construída.

O Exemplo 4.1 ilustra algumas das dificuldades associadas com o método exposto. Em primeiro lugar, a sua implementação pode ser bastante complicada. Em cada passo será necessário conhecer a cadeia e a árvore (ou pelo menos a sua fronteira) correntes, e além disso guardar todas as configurações anteriores em que foram tomadas decisões sobre a escolha de alternativas, para poder executar o retrocesso. Isto requer uma estrutura de dados bastante elaborada. Um problema mais importante ainda é que o método pode ser extremamente ineficiente devido à possibilidade de haver retrocessos. O número de operações para analisar uma cadeia pode chegar a ser uma função exponencial do seu comprimento (veja Exercício 4.3). Uma outra dificuldade de caráter prático é que as derivações diretas, ou seja, no caso, as decisões sobre as alternativas entre as produções, muitas vezes acarretam certas ações de outras partes do compilador, como por exemplo geração de instruções de código-objeto, ou modificação das tabelas internas. O efeito dessas ações terá que ser anulado caso haja retrocesso, o que é, em geral, uma tarefa complicada. Os problemas que acabamos de discutir fazem com que a análise descendente seja usada quase que exclusivamente quando se podem eliminar os retrocessos.

Note-se que a gramática do Exemplo 4.1 gera a mesma linguagem de expressões que a gramática utilizada nos capítulos anteriores. A modificação consistiu em eliminar a recursão esquerda, mas introduzindo a recursão direita. Conseqüentemente, esta gramática corresponde a associar-se à direita os operadores  $+$  e  $*$ , ao contrário do caso anterior. Essa modificação foi introduzida para evitar que o algoritmo entrasse numa repetição infinita. Logo de início, a folha corrente teria o rótulo  $E$ , e seria substituída pela subárvore correspondente à derivação  $E \Rightarrow E+T$ , fazendo com que a nova folha corrente também tenha o rótulo  $E$ , e assim por diante (veja também Exercício 4.2). O problema pode surgir tanto por causa da recursão esquerda direta, ou seja produções da forma  $A ::= A\alpha$ , como de maneira mais

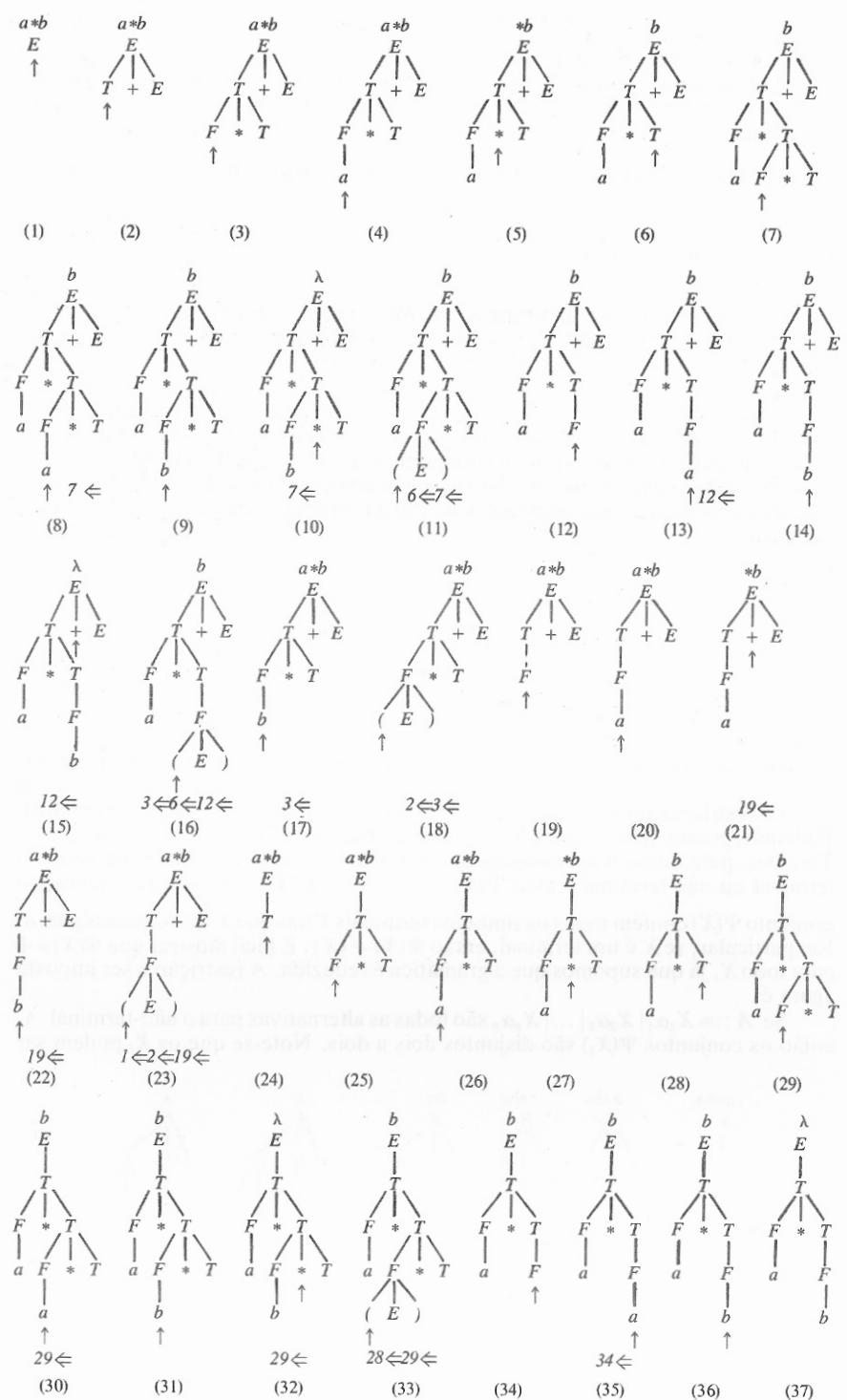


Fig. 4.1

geral, de derivações da forma  $A \xrightarrow{*} A\alpha$ . A solução dada ao problema no Exemplo 4.1 nem sempre é conveniente, pois ela muda o significado atribuído às expressões. Na seção seguinte veremos algumas maneiras práticas de resolver os problemas da análise descendente, tornando o método eficiente e aplicável a muitas linguagens de programação.

### 4.3 ELIMINAÇÃO DE RETROCESSOS E DA RECURSÃO ESQUERDA

A maneira mais simples de evitar retrocessos é fazer com que o algoritmo sempre tome a decisão correta quanto à produção a ser aplicada. Uma classe muito simples de gramáticas para as quais isto pode ser feito é obtida impondo-se as restrições:

1. Toda produção é da forma  $A ::= X\alpha$ , onde  $X$  é um terminal.
2. Se  $A ::= X_1\alpha_1|X_2\alpha_2|\dots|X_n\alpha_n$  são todas as alternativas para o não-terminal  $A$ , então os terminais  $X_i$  são todos distintos entre si.

É suficiente agora modificar ligeiramente o algoritmo descrito na seção anterior, fazendo com que a escolha da produção seja baseada no primeiro símbolo da cadeia  $\alpha$ . É óbvio que poderá haver no máximo uma alternativa que deverá ser escolhida. Caso não haja nenhuma escolha, isto é,  $\alpha$  não comece com algum dos símbolos  $X_i$  que correspondem ao não-terminal  $A$  da folha corrente, então a cadeia não é uma sentença.

#### Exemplo 4.2

Consideremos a gramática

$$E ::= a \mid b \mid +EE \mid *EE$$

Esta gramática satisfaz as restrições discutidas acima. A Figura 4.2 indica os passos da análise da sentença  $+a*ba$ . Foram omitidos os passos correspondentes à comparação do primeiro terminal de cada alternativa com o primeiro símbolo da cadeia corrente, uma vez que esta comparação já foi feita para escolher a própria alternativa.

As restrições impostas acima são muito severas para serem viáveis na prática. Podemos, porém, generalizar a idéia obtendo uma classe mais ampla de gramáticas. Teremos, para tanto, a necessidade de uma definição auxiliar. Seja  $X$  um símbolo terminal ou não-terminal; então  $\Psi(X) = \{Y \in T \mid X\psi_p^* Y\}$ . Em outras palavras, o conjunto  $\Psi(X)$  contém todos os símbolos terminais  $Y$  tais que  $X \xrightarrow{*} Y\alpha$  para algum  $\alpha$ . Em particular, se  $X$  é um terminal, então  $\Psi(X) = \{X\}$ . É fácil mostrar que  $\Psi(X) \neq \emptyset$  para todo  $X$ , já que supomos que a gramática é reduzida. A restrição a ser imposta agora é:

Se  $A ::= X_1\alpha_1|X_2\alpha_2|\dots|X_n\alpha_n$  são todas as alternativas para o não-terminal  $A$ , então os conjuntos  $\Psi(X_i)$  são disjuntos dois a dois. Note-se que os  $X_i$  podem ser

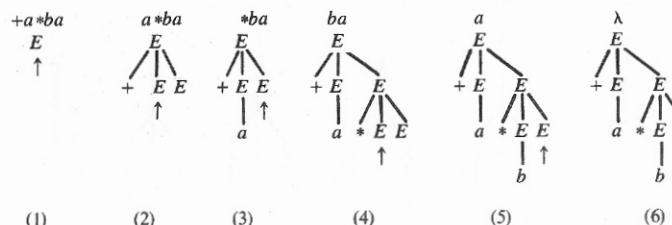


Fig. 4.2

tanto terminais como não-terminais.

O algoritmo de análise continua sendo muito simples, só que ele deve verificar a qual dos conjuntos  $\Psi(X_i)$  pertence o primeiro símbolo da cadeia  $\alpha$ . Como os conjuntos devem ser disjuntos, haverá no máximo uma alternativa possível.

#### Exemplo 4.3

Seja a gramática:

$$\begin{aligned} S &::= AS \mid BA \\ A &::= aB \mid C \\ B &::= bA \mid d \\ C &::= c \end{aligned}$$

	$\psi_p$	$\psi_p^*$	$\Psi$
$S$	$AB$	$ABaCbdcS$	$abdc$
$A$	$aC$	$aCcA$	$ac$
$B$	$bd$	$bdB$	$bd$
$C$	$c$	$cC$	$c$

(a)

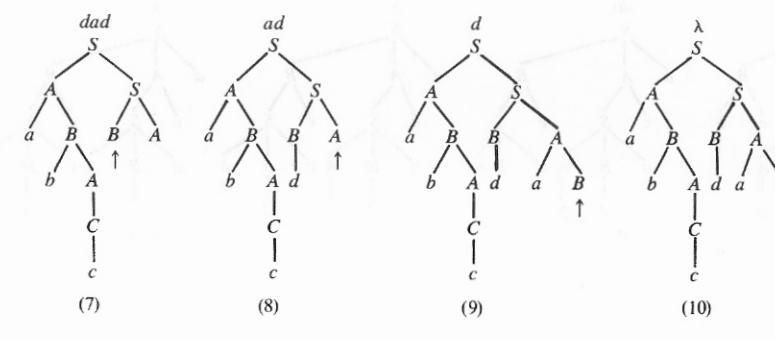
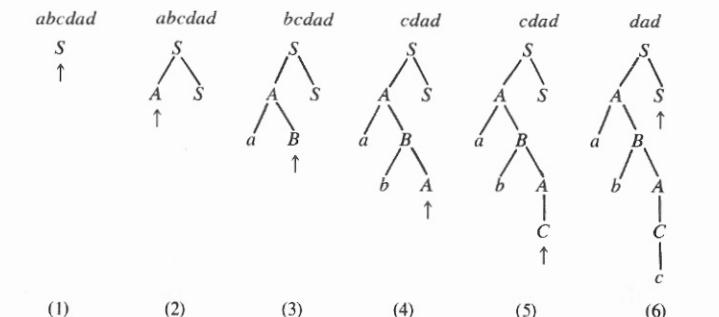


Fig. 4.3

A Figura 4.3 indica o cálculo dos conjuntos  $\Psi$  para os símbolos não-terminais, e os passos da análise para a sentença  $abcad$ .

A classe de gramáticas que satisfazem esta última restrição é denominada  $LL(1)$ . O nome vem do fato de se poder analisar uma cadeia da esquerda para a direita, produzindo uma derivação esquerda,<sup>1</sup> verificando apenas um símbolo da cadeia de entrada para decidir qual é a produção a ser aplicada. A definição pode ser generalizada para  $LL(k)$ ,  $k \geq 0$ . Para tais gramáticas podem-se obter analisadores descendentes sem retrocesso, se a escolha da produção a ser aplicada for baseada nos  $k$  primeiros símbolos (se existirem) da cadeia corrente.<sup>2</sup> Ao invés de definir formalmente estas classes de gramáticas, daremos um exemplo de gramática que não é do tipo  $LL(1)$ , mas para a qual a análise sem retrocessos é possível, se forem consultados ocasionalmente os dois primeiros símbolos da cadeia corrente.

#### Exemplo 4.4

Seja a gramática:

$$\begin{aligned} S &::= aAB \mid aBA \\ A &::= b \mid cS \\ B &::= d \mid eS \end{aligned}$$

Claramente, a primeira produção faz com que a gramática não seja do tipo  $LL(1)$ . Entretanto, como é fácil verificar, uma cadeia derivada de  $A$  começa sempre com os símbolos  $b$  ou  $c$ , e uma cadeia derivada de  $B$ , com os símbolos  $d$  ou  $e$ . Conseqüentemente, a escolha das produções para o não-terminal  $S$  pode-se basear no segundo símbolo da cadeia corrente. A Figura 4.4 indica a análise da cadeia  $acadbeadb$ .

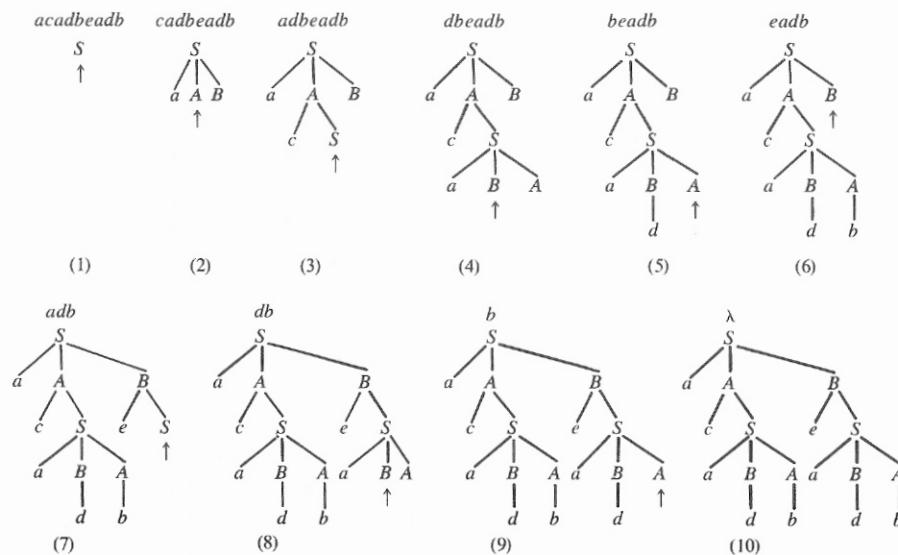


Fig. 4.4

<sup>1</sup>Em inglês: *Left-to-right parsing producing Leftmost derivation*.

<sup>2</sup>Em geral, podem ser incluídas nessa classe gramáticas com produções da forma  $A ::= \lambda$ .

Uma propriedade importante é que toda gramática que pertence à classe  $LL(1)$  é não-ambígua (veja Exercício 4.4).

Mesmo a classe  $LL(k)$  não inclui certas gramáticas de interesse. Consideremos a gramática do Exemplo 4.1:

$$\begin{aligned} E &::= T+E \mid T \\ T &::= F*T \mid F \\ F &::= a \mid b \mid (E) \end{aligned}$$

É fácil verificar que a partir do não-terminal  $T$  podemos derivar cadeias terminais arbitrariamente compridas. Conseqüentemente, nenhum número  $k$  finito de símbolos iniciais da cadeia corrente será suficiente para decidir, em geral, qual das duas alternativas para o não-terminal  $E$  (e analogamente para  $T$ ) deve ser usada. Uma maneira possível de se resolver o problema, no caso, é transformar a gramática em:

$$\begin{aligned} E &::= TE' \\ E' &::= +E \mid \lambda \\ T &::= FT' \\ T' &::= *T \mid \lambda \\ F &::= a \mid b \mid (E) \end{aligned}$$

Com esta transformação acabamos de introduzir produções com lados direitos nulos, tornando difícil a decisão quanto à escolha das alternativas para  $E'$  e  $T'$ . Convencionaremos, entretanto, que a escolha nula só pode ser adotada se forem excluídas todas as outras alternativas. Não é difícil ver, no caso, que para que esta convenção dé resultados corretos devemos ter certeza de que, numa forma sentencial esquerda da gramática, o símbolo  $E'$  nunca pode ser seguido do símbolo  $+$ , nem o símbolo  $T'$  de  $*$ . Esta verificação pode ser efetuada, em geral, utilizando-se as técnicas do Capítulo 2 (veja Exercício 2.19). A Figura 4.5 indica a análise da cadeia  $a+b^*a$  segundo a gramática modificada. Um problema óbvio que surge com este tipo de modificação é o aumento no comprimento das derivações, que será refletido num número maior de operações para realizar a análise. Este problema pode ser aliviado adotando-se uma notação estendida para gramáticas e modificando convenientemente o algoritmo de análise. Suponhamos que algumas alternativas para o não-terminal  $A$  têm a forma

$$A ::= \beta\gamma_1 \mid \beta\gamma_2 \mid \dots \mid \beta\gamma_n$$

com  $\beta \neq \lambda$ . Podemos “fatorar” estas produções escrevendo<sup>1</sup>

$$A ::= \beta(\gamma_1 \mid \gamma_2 \mid \dots \mid \gamma_n).$$

Caso  $\gamma_i = \lambda$  para algum  $i$ , colocaremos esta alternativa em último lugar, isto é,  $\gamma_n = \lambda$ . Assim,  $E ::= T+E \mid T$  pode ser reescrito como  $E ::= T(+E \mid \lambda)$ . Esta notação pode ser generalizada de maneira óbvia, adotando-se níveis arbitrários de encaixamento de parênteses, concatenação, e assim por diante. Usando esta notação, o algoritmo poderá adiar a decisão sobre a escolha de alternativa até que seja reconhecida a parte comum derivável de  $\beta$ . O exemplo seguinte ilustra esta técnica.

<sup>1</sup>Note-se que os parênteses (bem como chaves introduzidas mais adiante) fazem parte da notação, e não são símbolos terminais da gramática.

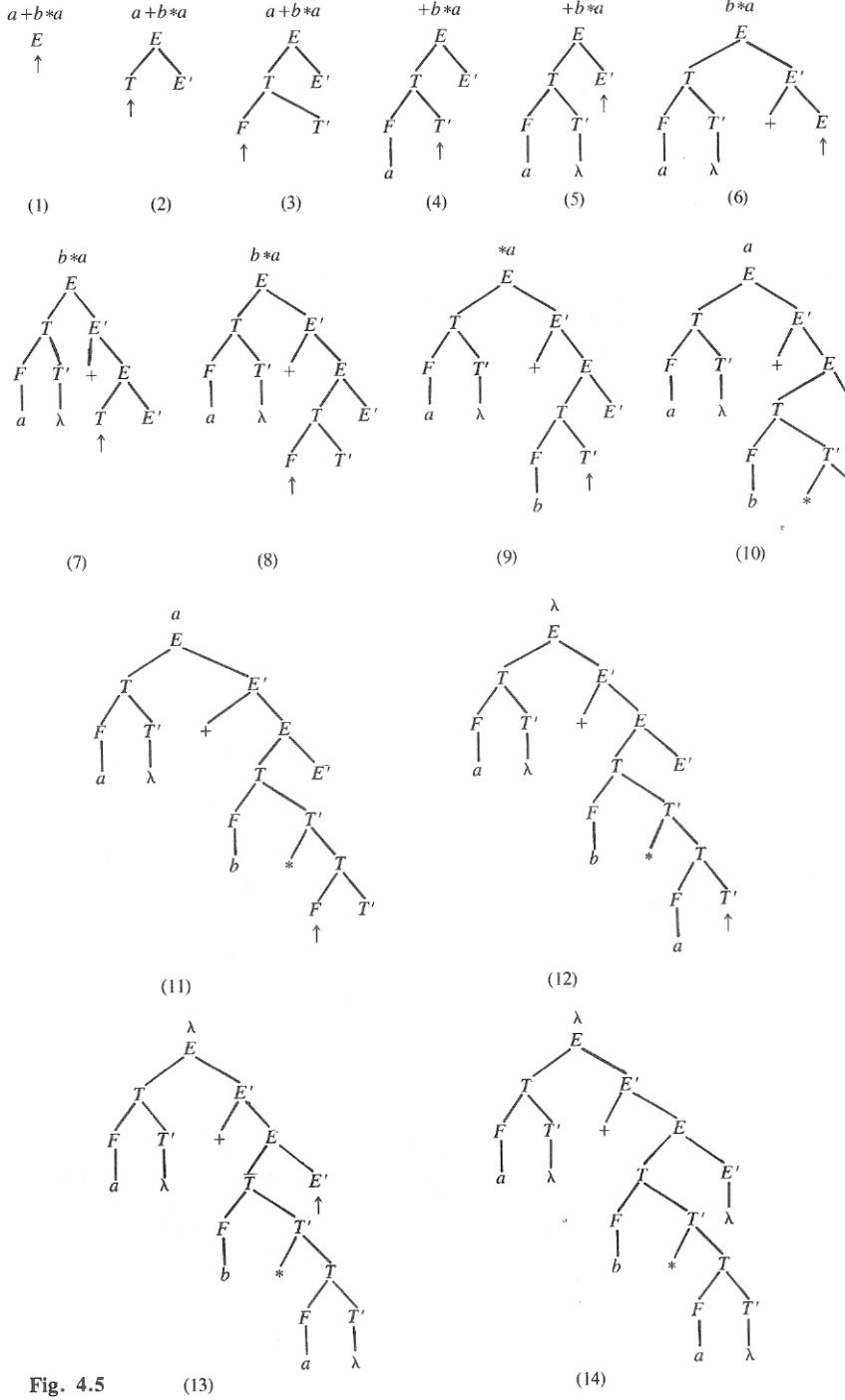


Fig. 4.5

**Exemplo 4.5**  
Consideremos a gramática:

$$\begin{aligned} E &::= T+E \mid T-E \mid T \\ T &::= F*T \mid F/T \mid F \\ F &::= a \mid b \mid (E) \end{aligned}$$

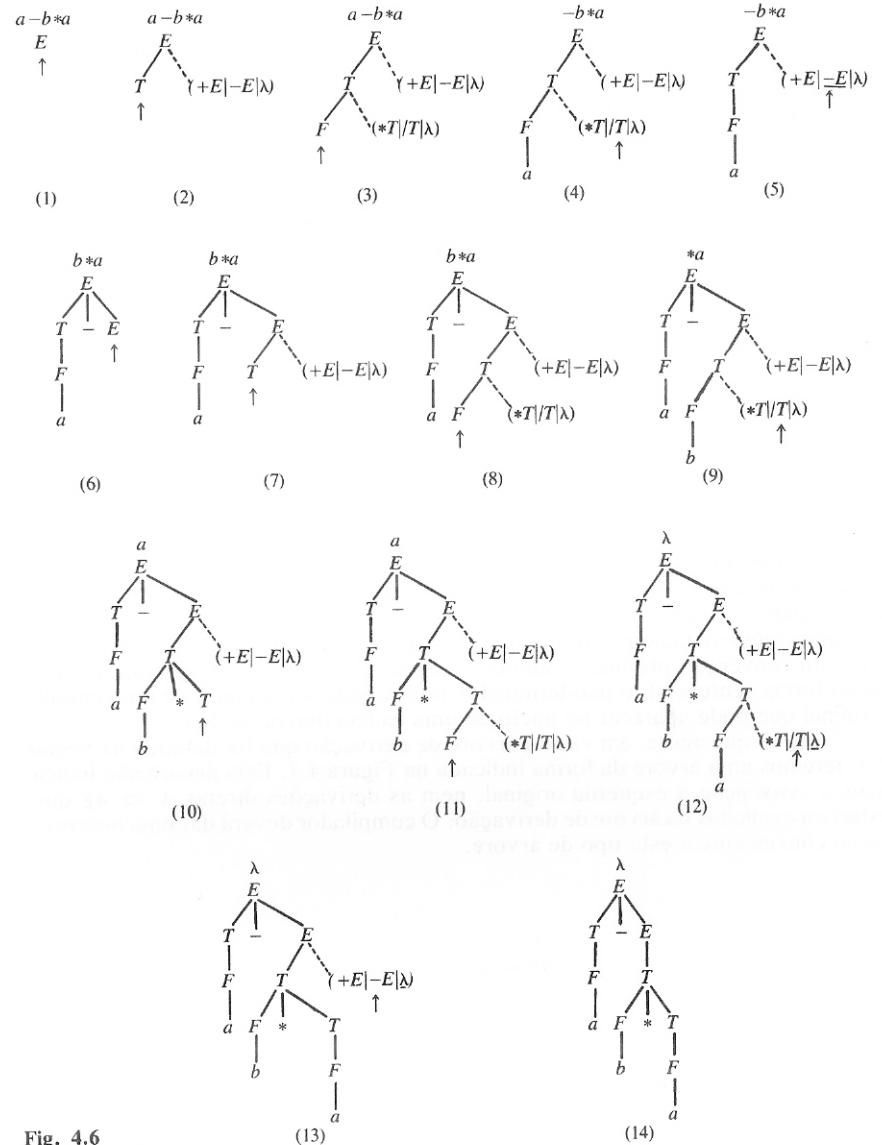


Fig. 4.6

que pode ser reescrita como

$$\begin{aligned} E &::= T(+E \mid -E \mid \lambda) \\ T &::= F(*T \mid /T \mid \lambda) \\ F &::= a \mid b \mid (E) \end{aligned}$$

A Figura 4.6 indica a análise da cadeia  $a-b*a$ . A flecha vertical aponta agora para a folha corrente ou então para um conjunto de alternativas. Estão sublinhadas as alternativas escolhidas.

O problema da recursão esquerda tem uma solução bastante simples quando se trata da recursão direta da forma  $A ::= A\alpha$ . Tomemos, por exemplo, as produções  $E ::= E+T \mid T$ . Note-se que com estas produções podemos ter as derivações da forma  $E \Rightarrow^* T, E \Rightarrow^* T+T, E \Rightarrow^* T+T+T, \dots, E \Rightarrow^* T+T+\dots+T$ , isto é, podemos derivar um número arbitrário de símbolos  $T$ , separados pelos símbolos  $+$ . Este fato sugere a substituição da notação recursiva por uma notação iterativa. Assim, a ocorrência da construção  $\{\alpha\}$  numa produção denotará a repetição da cadeia  $\alpha$  zero ou mais vezes, ou seja, um elemento de  $\alpha^*$ . No caso da produção citada, teremos  $E ::= T\{+T\}$ . Nos casos mais complicados esta notação poderá ser combinada com a fatoração. Assim,  $E ::= E+T \mid E-T \mid T$  poderia ser reescrito como  $E ::= E(+T \mid -T) \mid T$  e, em seguida, como  $E ::= T\{+T \mid -T\}$ . De maneira mais geral, se existirem produções da forma:

$$A ::= \beta\gamma_1 \mid \beta\gamma_2 \mid \dots \mid \beta\gamma_n, \quad \beta \neq \lambda,$$

deve-se substituí-las por  $A ::= \beta(\gamma_1 \mid \gamma_2 \mid \dots \mid \gamma_n)$ . Desta maneira poderá existir apenas uma única alternativa com recursão esquerda. Se todas as alternativas para o não-terminal  $A$  são dadas por  $A ::= \delta_1 \mid \delta_2 \mid \dots \mid \delta_m \mid A\xi$ , então reescrevemos:  $A ::= (\delta_1 \mid \delta_2 \mid \dots \mid \delta_m)\{\xi\}$ . Esta notação indica que se o rótulo da folha corrente é  $A$ , então o algoritmo deve reconhecer uma parte inicial de  $\alpha$  derivável de algum dos  $\delta_i$ , e depois deve procurar zero ou mais ocorrências de cadeias deriváveis de  $\xi$ . Para que não haja retrocessos, adotaremos a convenção de procurar o número máximo possível de ocorrências de cadeias deriváveis de  $\xi$ . Para que esta convenção produza resultados corretos, devemos ter a certeza de que, numa forma sentencial, o não-terminal  $A$  nunca pode ser seguido de um símbolo terminal que pode aparecer no início de uma cadeia derivável de  $\xi$ .

Note-se que agora, em vez da árvore de derivação que foi definida na Seção 2.4, teremos uma árvore da forma indicada na Figura 4.7. Esta árvore não indica mais a associação à esquerda original, nem as derivações diretas  $A \Rightarrow A\xi$  que estariam explícitas na árvore de derivação. O compilador deverá dar uma interpretação conveniente a este tipo de árvore.

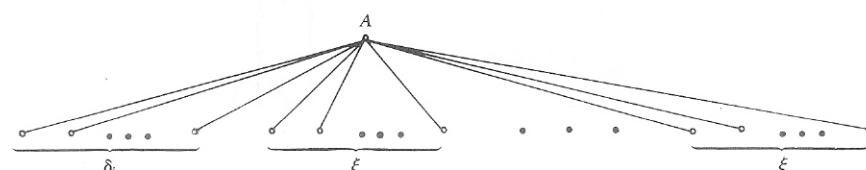


Fig. 4.7

#### Exemplo 4.6

Consideremos a gramática:

$$\begin{aligned} E &::= E+T \mid E-T \mid +T \mid -T \mid T \\ T &::= F(*T \mid /T \mid \lambda) \\ F &::= a \mid b \mid (E) \end{aligned}$$

Reescrevendo de acordo com a notação introduzida nesta seção, teremos:

$$\begin{aligned} E &::= (+T \mid -T \mid T) \{+T \mid -T\} \\ T &::= F\{*T \mid /T\} \\ F &::= a \mid b \mid (E) \end{aligned}$$

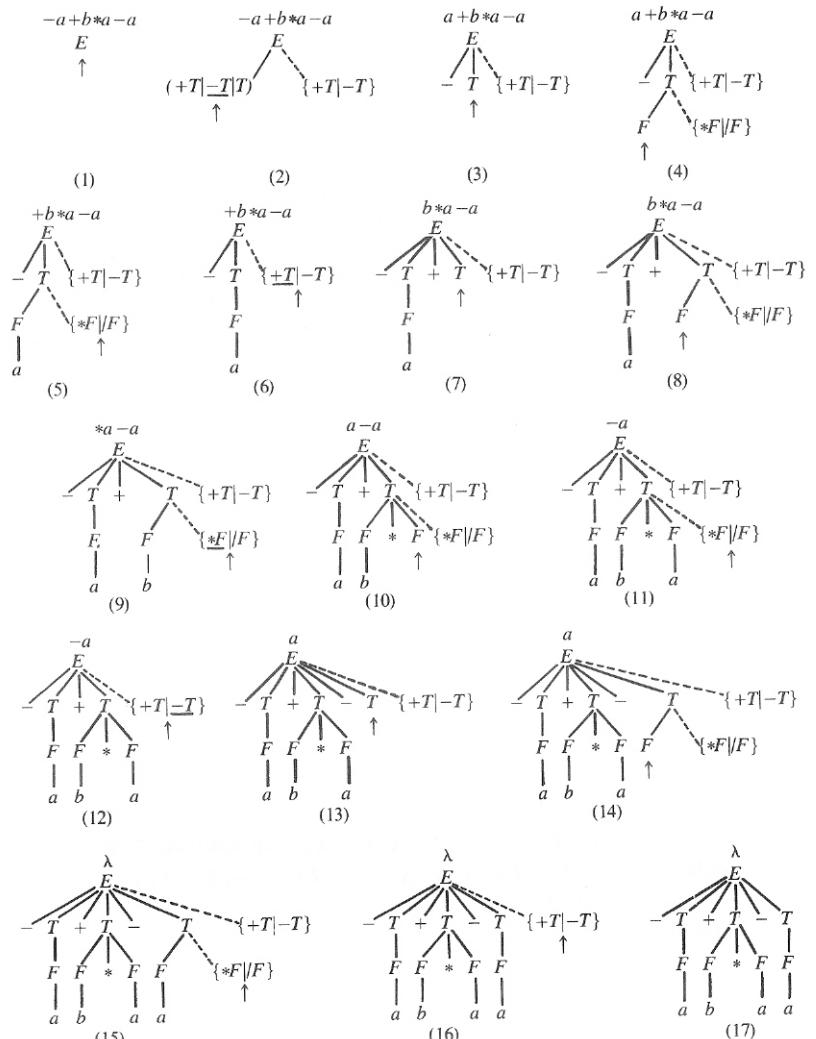


Fig. 4.8

Note-se que  $\Psi(T) = \{a, b, \{\}\}$ , tornando simples a escolha entre as alternativas  $+T$ ,  $-T$  e  $T$ . A Figura 4.8 apresenta os passos da análise da sentença  $-a+b*a-a$ . Note-se que a notação  $\{\xi\}$  inclui uma alternativa implícita  $\lambda$  correspondente a zero ocorrências de  $\xi$ .

Como já mencionamos antes, a árvore final da Figura 4.8 não é exatamente uma árvore de derivação. O compilador deverá encarregar-se de associar corretamente os operadores  $+, -, *, /$ . A ausência explícita de algumas derivações da forma  $E \Rightarrow T$  e  $T \Rightarrow F$  é, como já mencionamos na Seção 3.3, uma vantagem nesse caso, pois o compilador não executa nenhuma ação associada com estas derivações.

#### 4.4 IMPLEMENTAÇÃO DA ANÁLISE DESCENDENTE. ANALISADOR DESCENDENTE RECURSIVO

A implementação da análise descendente sem retrocesso pode ser feita de maneira bastante simples. A fim de facilitar a exposição, consideremos o caso de uma gramática do tipo LL(1), em que não foi necessário usar a notação estendida. Um estudo do algoritmo geral de análise descrito na Seção 4.2 mostra que em cada passo é suficiente conhecer a cadeia dos rótulos das folhas da árvore corrente  $D$ , começando pela folha corrente. Esta cadeia pode ser mantida numa pilha. O elemento no topo da pilha será o rótulo da folha corrente. Quando o topo da pilha contiver um terminal, ele será comparado com o primeiro símbolo da cadeia corrente, e se estes símbolos forem iguais, serão ambos removidos. Caso o símbolo do topo da pilha for um não-terminal, ele será substituído na pilha pela cadeia de símbolos correspondente à alternativa escolhida. O primeiro símbolo desta cadeia deverá ficar no topo da pilha. A escolha da alternativa será baseada nos conjuntos  $\Psi$  definidos anteriormente. A Figura 4.9 indica esta implementação. O vetor  $P$  contém os elementos da pilha, e os procedimentos *PRÓXIMO* e *ERRO* têm a mesma função descrita na Seção 3.2. Caso o procedimento retorne sem chamar *ERRO*, a

```

procedimento ANÁLISE DESCENDENTE;
início
  P[I]:=‘S’; i:=I; término:=falso;
  PRÓXIMO;
  repita
    x:=P[i];
    se “x é terminal”
      então se x=símbolo
        então {PRÓXIMO; i:=i-1;
              se i=0 então término:=verdadeiro}
        senão ERRO
      senão
      se “existe uma produção x ::= X1X2...Xn com símbolo ∈ Ψ(Xj)”
        então {P[i]:=‘Xn’; P[i+1]:=‘Xn-1’; ...; P[i+n-1]:=‘X1’;
               i:=i+n-1}
        senão ERRO
      até término;
      se símbolo ≠ ‘#’ então ERRO
  fim

```

Fig. 4.9

cadeia dada é uma sentença da gramática. Numa implementação real, as produções da gramática teriam que ser representadas por meio de uma estrutura de dados conveniente. Não é difícil modificar o procedimento para que ele construa também a árvore de derivação, nem para que funcione para gramáticas usando a notação estendida introduzida na seção anterior.

#### Exemplo 4.7

Consideremos novamente a gramática do Exemplo 4.3:

$$\begin{aligned} S &::= AS \mid BA \\ A &::= aB \mid C \\ B &::= bA \mid d \\ C &::= c \end{aligned}$$

e a sentença  $abcdad$ . A Figura 4.10 indica os passos de execução do procedimento da Figura 4.9 neste caso. A variável *símbolo* contém sempre o primeiro símbolo da cadeia corrente.

Uma outra maneira simples e comumente usada de se implementar a análise descendente é através de um conjunto de procedimentos mutuamente recursivos. Cada procedimento corresponde a um não-terminal da gramática, e a sua função é analisar a parte inicial da cadeia corrente  $\alpha$  que pode ser derivada a partir deste não-terminal.

#### Exemplo 4.8

Consideremos a gramática do Exemplo 4.7. A Figura 4.11 apresenta um analisador descendente recursivo para esta gramática.

Note-se que nos procedimentos da Figura 4.11, para cada ocorrência de um não-terminal do lado direito da produção, tem-se uma chamada do procedimento

<i>i</i>	<i>P</i>	<i>x</i>	CADEIA CORRENTE
1	<i>S</i>	<i>S</i>	<i>abcdad</i> #
2	<i>SA</i>	<i>A</i>	<i>abcdad</i> #
3	<i>SBa</i>	<i>a</i>	<i>abcdad</i> #
2	<i>SB</i>	<i>B</i>	<i>bcdad</i> #
3	<i>SAb</i>	<i>b</i>	<i>bcdad</i> #
2	<i>SA</i>	<i>A</i>	<i>cdad</i> #
2	<i>SC</i>	<i>C</i>	<i>cdad</i> #
2	<i>Sc</i>	<i>c</i>	<i>cdad</i> #
1	<i>S</i>	<i>S</i>	<i>dad</i> #
2	<i>AB</i>	<i>B</i>	<i>dad</i> #
2	<i>Ad</i>	<i>d</i>	<i>dad</i> #
1	<i>A</i>	<i>A</i>	<i>ad</i> #
2	<i>Ba</i>	<i>a</i>	<i>ad</i> #
1	<i>B</i>	<i>B</i>	<i>d</i> #
1	<i>d</i>	<i>d</i>	<i>d</i> #
0			#

Fig. 4.10

procedimento EXEMPLO 4.8;

```
procedimento S;
início
  caso símbolo de
    'a','c': {A;S};
    'b','d': {B;A};
    outros: ERRO
  fim do caso
fim de S;

procedimento A;
início
  caso símbolo de
    'a': {PRÓXIMO; B};
    'c': C;
    outros: ERRO
  fim do caso
fim de A;

procedimento B;
início
  caso símbolo de
    'b': {PRÓXIMO; A};
    'd': PRÓXIMO;
    outros: ERRO
  fim do caso
fim de B;

procedimento C;
início
  caso símbolo de
    'c': PRÓXIMO;
    outros: ERRO
  fim do caso
fim de C;

início
  PRÓXIMO; S;
  se símbolo ≠ '#' então ERRO
fim
```

Esta maneira de implementar a análise descendente é facilmente adaptada a gramáticas descritas em notação estendida da Seção 4.3.

#### Exemplo 4.9

Consideremos a mesma gramática do Exemplo 4.6:

$$\begin{aligned} E &::= (+T \mid -T \mid T) \{+T \mid -T\} \\ T &::= F \{ *F \mid /F \} \\ F &::= a \mid b \mid (E) \end{aligned}$$

A Figura 4.12 traz o analisador descendente recursivo para esta gramática. ●

Deve-se notar que não seria difícil escrever um programa cuja entrada fosse a descrição de uma gramática e cuja saída fosse o programa de análise descendente

procedimento EXEMPLO 4.9;

```
procedimento E;
início
  caso símbolo de
    '+'','-': {PRÓXIMO; T};
    outros: T
  fim do caso;
  enquanto símbolo ∈ {'+', '-'}
    faça {PRÓXIMO; T};
  fim de E;

procedimento T;
início
  F;
  enquanto símbolo ∈ {'*', '/'}
    faça {PRÓXIMO; F};
  fim de T;

procedimento F;
início
  caso símbolo de
    'a','b': PRÓXIMO;
    '(': {PRÓXIMO; E;
      se símbolo = ')' então PRÓXIMO
      senão ERRO};
    outros: ERRO
  fim do caso
fim de F;

início
  PRÓXIMO; E;
  se símbolo ≠ '#' então ERRO
fim
```

Fig. 4.12

Fig. 4.11

correspondente. Para cada terminal, temos uma verificação se o mesmo ocorre como o primeiro símbolo da cadeia corrente. No caso afirmativo, avança-se na cadeia dada chamando o procedimento PRÓXIMO; caso contrário, a cadeia dada não era uma sentença.

recursiva correspondente.

Uma grande vantagem deste tipo de analisador sintático é a sua flexibilidade. Como veremos mais adiante, é conveniente muitas vezes que o compilador execute certas ações em pontos intermediários que não correspondem à aplicação de produções. Isto pode ser conseguido facilmente inserindo-se comandos apropriados nos pontos correspondentes dos procedimentos recursivos. A implementação recursiva pode ser bastante eficiente se usada num sistema onde as chamadas de procedimentos causam pouca sobrecarga. Podem-se, por outro lado, usar as técnicas padronizadas para substituir o uso de recursão por uma pilha explícita, introduzindo algumas otimizações. Este tipo de implementação terá a vantagem adicional de se ter acesso à pilha, facilitando, eventualmente, a recuperação de erros. Na prática, os analisadores descendentes recursivos são bastante utilizados, conjugados às vezes com outros métodos. Assim, para linguagens como Pascal, uma solução muito satisfatória é usar a análise de precedência de operadores da Seção 3.3 para reconhecer as expressões, e o método recursivo descendente para as outras construções. Elimina-se, assim, uma grande parte de chamadas recursivas, além de aproveitar-se a eficiência do método de precedência de operadores.

#### 4.5 DIAGRAMAS SINTÁTICOS

Introduziremos nesta seção uma maneira gráfica comumente usada para apresentar gramáticas de linguagens de programação através dos chamados *diagramas sintáticos* ou *cartas sintáticas*. A notação é introduzida neste capítulo, pois o seu uso mais comum está ligado ao uso dos métodos descendentes de análise sintática.

Nesta representação, a cada não-terminal da gramática corresponde um grafo orientado cujos nós são rotulados com os símbolos terminais e não-terminais da gramática. Este grafo é obtido através de um conjunto de regras enumeradas a seguir, considerando-se todas as alternativas para o não-terminal. As regras de construção são:

1. Um símbolo terminal  $a$  é representado por um grafo da forma indicada na Figura 4.13a.
2. Um símbolo não-terminal  $A$  é representado por um grafo da forma indicada na Figura 4.13b.
3. Uma construção  $\alpha_1 | \alpha_2 | \dots | \alpha_n$  é representada por um grafo da forma indicada na Figura 4.13c, sendo que  $\tilde{\alpha}_i$  indica o grafo obtido para  $\alpha_i$ .
4. Uma construção  $\alpha_1\alpha_2\dots\alpha_n$  é representada por um grafo da forma indicada na Figura 4.13d, sendo que  $\tilde{\alpha}_i$  indica o grafo obtido para  $\alpha_i$ . A cadeia nula  $\lambda$  é indicada por uma simples aresta.
5. Uma construção repetitiva  $\{\alpha\}$  é representada por um grafo da forma indicada na Figura 4.13e, sendo  $\tilde{\alpha}$  o grafo obtido para  $\alpha$ .

Note-se que cada grafo tem uma aresta de entrada e uma de saída, devendo-se fazer as ligações apropriadas quando os grafos são compostos durante a construção. A derivação de uma cadeia a partir de um não-terminal da gramática corresponde a percorrer um dos caminhos possíveis do grafo correspondente, começando pela aresta de entrada e terminando pela aresta de saída. Se for encontrado no caminho um símbolo terminal, ele é enumerado como o próximo símbolo da cadeia gerada. No caso de um símbolo não-terminal, deve-se usar o grafo correspondente como uma sub-rotina, eventualmente recursiva.

#### Exemplo 4.10

Consideremos novamente a gramática:

$$\begin{aligned} E &::= (+T \mid -T \mid T) \{+T \mid -T\} \\ T &::= F \{ *F \mid /F\} \\ F &::= a \mid b \mid (E) \end{aligned}$$

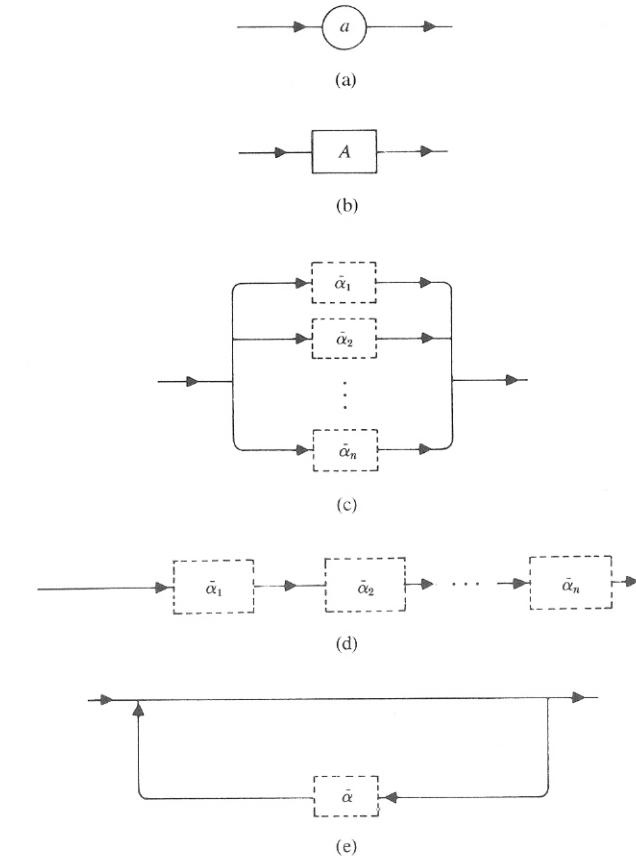


Fig. 4.13

A Figura 4.14 indica o diagrama sintático correspondente. Podemos fatorar ainda mais esta gramática, obtendo:

$$\begin{aligned} E &::= (+ \mid - \mid \lambda) T \{ (+ \mid -) T\} \\ T &::= F \{ (* \mid /) F\} \\ F &::= a \mid b \mid (E) \end{aligned}$$

O diagrama correspondente está indicado na Figura 4.15.

É comum reduzir-se o número de grafos para uma gramática, substituindo-se as ocorrências de um não-terminal pelo grafo correspondente. Esta técnica reduz o número de não-terminais na gramática, tornando mais eficiente o algoritmo de análise descendente correspondente, como veremos mais adiante.

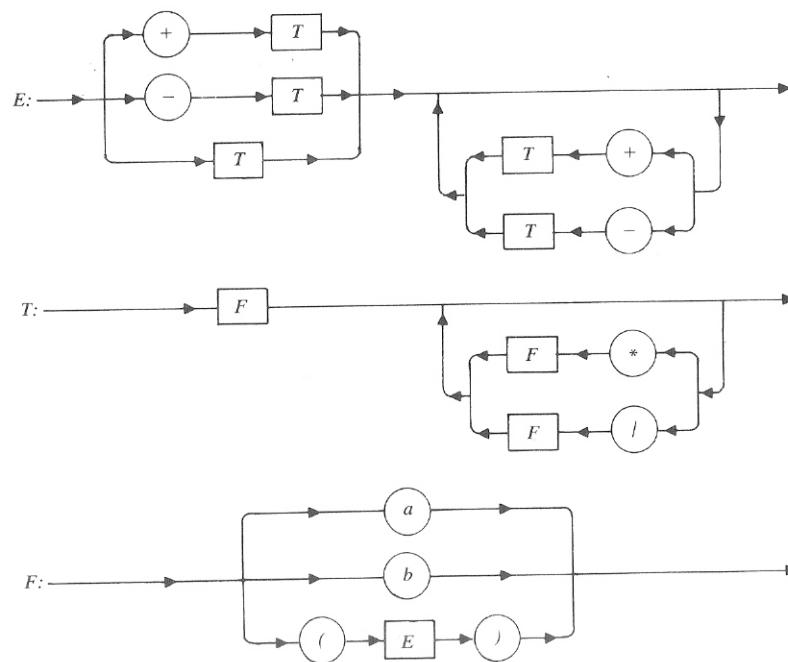


Fig. 4.14

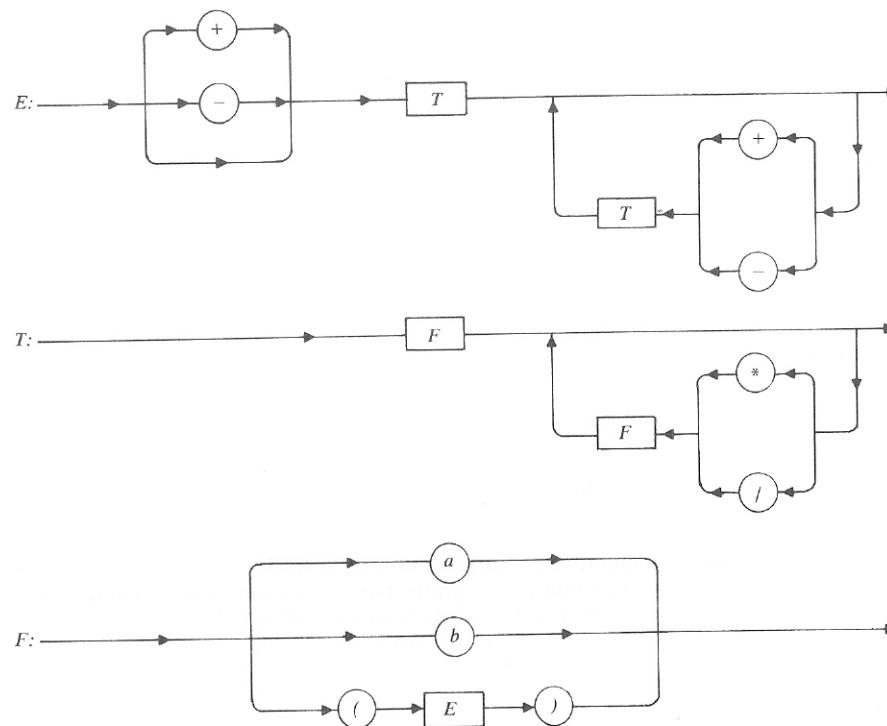


Fig. 4.15

### Exercícios

- Indique os passos da análise sintática da sentença  $a+b*a$ , usando o algoritmo com retrocesso descrito na Seção 4.2, seguindo a gramática do Exemplo 4.1.
- Considere a gramática de expressões

$$\begin{aligned} E &::= E+T \mid T \\ T &::= T*T \mid F \\ F &::= a \mid b \mid (E) \end{aligned}$$

Mostre que qualquer que seja a ordem prefixada para a escolha das alternativas dos não-terminais, haverá cadeias para as quais o analisador descendente com retrocesso entrará numa repetição infinita.

- Considere a gramática

$$S ::= aSb \mid aSc \mid ab \mid ac$$

Mostre que, para uma cadeia de entrada da forma  $a^n c^n$  ( $n \geq 1$ ), o analisador descendente com retrocesso, que escolhe as alternativas na ordem dada, faz  $t(n)$  escolhas antes de aceitar a cadeia, onde  $t(n) = 7 \cdot 2^n - 2$ .

(Sugestão: Mostre, em primeiro lugar, que:

$$\begin{aligned} t(1) &= 12 \\ \text{e } t(k) &= 2 \cdot t(k-1) + 2 \quad \text{para } k > 1. \end{aligned}$$

- Mostre que as gramáticas do tipo LL(1) são não-ambíguas. (Sugestão: Mostre que toda sentença tem uma única derivação esquerda.)
- Considere a gramática:

$$\begin{aligned} A &::= B \rightarrow A \mid B \\ B &::= B \vee C \mid C \\ C &::= C \wedge D \mid D \\ D &::= E \mid \sim E \\ E &::= a \mid (A) \end{aligned}$$

- Reescreva as produções em notação estendida conveniente para análise descendente e construa os diagramas sintáticos correspondentes.
- Escreva o analisador descendente recursivo para esta gramática.
- Indique os passos de análise para a sentença:

$$a \vee \sim a \rightarrow a \rightarrow \sim a.$$

- (Projeto) Escreva um programa que implementa a análise descendente com retrocesso descrita na Seção 4.2. O programa aceita como dados uma gramática em notação FNB e uma cadeia a ser analisada. Caso a cadeia pertença à linguagem gerada pela gramática, o programa deve produzir uma das suas árvores de derivação.
- Modifique o programa do exercício anterior para que ele produza todas as árvores de derivação de uma sentença.
- (Projeto) Escreva um programa que, a partir da gramática em notação FNB estendida, produz um analisador descendente recursivo correspondente.

9. Transforme o programa da Figura 4.12 num programa que substitui o uso da recursão por uma pilha explícita.
10. Suponha que as produções de uma gramática são representadas por uma estrutura de dados sugerida pela forma dos diagramas sintáticos. Escreva um programa genérico que usa tal estrutura para realizar análise descendente sem retrocesso. O programa poderá usar recursão ou então uma pilha explícita.

#### NOTAS BIBLIOGRÁFICAS

Analisadores descendentes foram utilizados em muitos sistemas, e a sua descrição pode ser encontrada praticamente em todos os textos dedicados à compilação. Citamos em particular Lewis, Rosenkrantz e Stearns (1976), Aho e Ullman (1977), Barrett e Couch (1979), Gries (1971). Os textos Wirth (1976) e Setzer e Melo (1981) descrevem de maneira pormenorizada a implementação da análise descendente para classes de gramáticas restritas. O conceito de gramáticas LL( $k$ ) foi introduzido em Lewis e Stearns (1968) e desenvolvido em Rosenkrantz e Stearns (1970). Uma discussão mais completa da teoria de análise descendente pode ser encontrada em Aho e Ullman (1972).

# 5

## Descrição do Pascal Simplificado

### 5.1 GENERALIDADES

Por razões expostas no Capítulo 1, discutiremos neste texto a implementação da linguagem de programação Pascal. Esta linguagem foi projetada por Niklaus Wirth (1971a) e teve um impacto muito grande sobre linguagens desenvolvidas posteriormente. Trataremos aqui apenas da implementação dos mecanismos mais representativos do Pascal, definindo uma versão simplificada da linguagem. A definição será compatível com a definição geral, isto é, um programa válido em Pascal simplificado será também um programa válido em Pascal, e terá o mesmo significado. No Capítulo 9, discutiremos algumas extensões que não fazem parte do Pascal, mas que são interessantes para o estudo de implementação de linguagens.

Os aspectos importantes do Pascal que não serão incluídos na nossa versão são: tipos *real* e *char*, declarações de constantes, tipos escalares definidos pelo usuário, registros, apontadores, declaração de arquivos, algumas formas de comandos. Outros aspectos que foram incluídos na nossa definição serão considerados opcionais, e a sua discussão será mais superficial. Entre eles estão matrizes e declarações de tipos.

A nossa descrição será muito sumária na parte de comentários informais que descrevem algumas restrições sintáticas e a semântica das construções. Definições mais completas poderão ser encontradas em Wirth (1971a) ou Jensen e Wirth (1974).

A sintaxe da nossa versão do Pascal será dada em FNB estendida, usando a notação da Seção 4.3. Por conveniência, introduziremos mais uma extensão: a notação  $[\alpha]$  será equivalente a  $\alpha|\lambda$ , ou seja, indicará que a cadeia  $\alpha$  é opcional. A gramática diferirá bastante da gramática oficial do Pascal. As modificações visaram à eliminação dos mecanismos não incluídos na nossa versão, à simplificação devida ao uso da FNB estendida e à conveniência de implementação de um analisador sintático.

Os não-terminais da nossa gramática são nomes mnemônicos colocados entre parênteses angulares  $<\cdot>$ , como por exemplo  $<\text{expressão}>$  ou  $<\text{declaração de procedimento}>$ . A referência no texto a um nome como “expressão” indica qualquer cadeia terminal derivável do símbolo  $<\text{expressão}>$ . As produções correspondentes aos mecanismos opcionais estão marcadas com (\*). A divisão em seções tem apenas finalidade didática, e as produções estão numeradas para facilitar referências.

O Apêndice I apresenta a nossa gramática excluindo os comentários informais, e o Apêndice II traz os diagramas sintáticos correspondentes.

## 5.2 PROGRAMAS E BLOCOS

1. <programa> ::= **program** <identificador> (<lista de identificadores>); <bloco>.
  2. <bloco> ::= [<parte de declarações de rótulos>]  
[<parte de definições de tipos>]  
[<parte de declarações de variáveis>]  
[<parte de declarações de sub-rotinas>]  
<comando composto>
  - (\*)
- 
- ## 5.3 DECLARAÇÕES
3. <parte de declarações de rótulos> ::= **label** <número> {, <número>};
  4. <parte de definições de tipos> ::= **type** <definição de tipo> {; <definição de tipo>};
  - (\*)
  5. <definição de tipo> ::= <identificador> = <tipo>
  - (\*)
  6. <tipo> ::= <identificador>|  
**array** [<índice> {, <índice>} ] **of** <tipo>
  - (\*)
  7. <índice> ::= <número> .. <número>
  - (\*)
  8. <parte de declarações de variáveis> ::= **var** <declaração de variáveis>  
{; <declaração de variáveis>};
  9. <declaração de variáveis> ::= <lista de identificadores> : <tipo>
  10. <lista de identificadores> ::= <identificador> {, <identificador>}
  11. <parte de declarações de sub-rotinas> ::= { <declaração de procedimento>; |  
<declaração de função> ; }
  12. <declaração de procedimento> ::= **procedure** <identificador>  
[ <parâmetros formais>]; <bloco>
  13. <declaração de função> ::= **function** <identificador>  
[ <parâmetros formais>]: <identificador>; <bloco>
  14. <parâmetros formais> ::= (<seção de parâmetros formais>  
{: <seção de parâmetros formais>})
  15. <seção de parâmetros formais> ::= [**var**]<lista de identificadores>: <identificador>  
| **function** <lista de identificadores>:<identificador>  
(\*)  
| **procedure** <lista de identificadores>  
(\*)

Todos os identificadores e rótulos devem ser declarados antes de serem utilizados. Isto implica a impossibilidade de se utilizar recursão mútua entre sub-ro-

tinas quando elas não estão declaradas uma dentro da outra. São considerados identificadores pré-declarados: *input*, *output*, *integer*, *boolean*, *read*, *write*, *true* e *false*.

## 5.4 COMANDOS

16. <comando composto> ::= **begin** <comando> {;<comando>} **end**
17. <comando> ::= [<número> :] <comando sem rótulo>
18. <comando sem rótulo> ::= <atribuição>  
| <chamada de procedimento>  
| <desvio>  
| <comando composto>  
| <comando condicional>  
| <comando repetitivo>
19. <atribuição> ::= <variável> := <expressão>
20. <chamada de procedimento> ::= <identificador> [(<lista de expressões>)]
21. <desvio> ::= **goto** <número>
22. <comando condicional> ::= **if** <expressão> **then** <comando sem rótulo>  
[ **else** <comando sem rótulo> ]
23. <comando repetitivo> ::= **while** <expressão> **do** <comando sem rótulo>

Os comandos de entrada e saída estão incluídos entre chamadas de procedimentos. Suporemos que existe um arquivo padrão de entrada contendo apenas números inteiros lidos pelo comando *read*( $v_1, \dots, v_n$ ), onde  $v_1, \dots, v_n$  são variáveis inteiros. Os pormenores sobre o formato do arquivo de entrada serão ignorados. Analogamente, o comando *write* ( $e_1, \dots, e_n$ ) imprimirá os valores das expressões inteiros  $e_1, \dots, e_n$  num arquivo padrão de saída.

Note-se que a produção de número 22 faz com que a gramática seja ambígua. Não seria difícil eliminar esta ambigüidade (veja Exemplo 2.5), mas a gramática resultante não seria conveniente para uma análise sintática descendente. A ambigüidade será resolvida então pelo próprio compilador, que deverá associar toda parte **else**... com **if**... **then**... mais próximo (veja também Exercício 5.2).

Note-se também que o comando iterativo tem uma só forma, e que foi eliminado o comando nulo.

## 5.5 EXPRESSÕES

24. <lista de expressões> ::= <expressão> {, <expressão>}
25. <expressão> ::= <expressão simples>[<relação><expressão simples>]
26. <relação> ::= = | <> | <|=|=|> = | >
27. <expressão simples> ::= [+|-] <termo> {(+|-|or) <termo>}

28.  $\langle \text{termo} \rangle ::=$   
 $\quad \langle \text{fator} \rangle \{ (*|\text{div}|\text{and}) \langle \text{fator} \rangle \}$
29.  $\langle \text{fator} \rangle ::=$   
 $\quad \langle \text{variável} \rangle$   
 $\quad | \langle \text{número} \rangle$   
 $\quad | \langle \text{chamada de função} \rangle$   
 $\quad | (\langle \text{expressão} \rangle)$   
 $\quad | \text{not } \langle \text{fator} \rangle$
30.  $\langle \text{variável} \rangle ::=$   
 $\quad \langle \text{identificador} \rangle$   
 $\quad | \langle \text{identificador} \rangle [ \langle \text{lista de expressões} \rangle ]$
31.  $\langle \text{chamada de função} \rangle ::=$   
 $\quad \langle \text{identificador} \rangle [ ( \langle \text{lista de expressões} \rangle ) ]$

As expressões podem ser inteiras ou booleanas, e a distinção deverá ser feita pelo compilador. Note-se que as constantes booleanas *true* e *false* são consideradas identificadores pré-declarados.

## 5.6 NÚMEROS E IDENTIFICADORES

32.  $\langle \text{número} \rangle ::=$   
 $\quad \langle \text{dígito} \rangle \{ \langle \text{dígito} \rangle \}$
33.  $\langle \text{dígito} \rangle ::=$   
 $\quad 0|1|2|3|4|5|6|7|8|9$
34.  $\langle \text{identificador} \rangle ::=$   
 $\quad \langle \text{letra} \rangle \{ \langle \text{letra} \rangle | \langle \text{dígito} \rangle \}$
35.  $\langle \text{letra} \rangle ::=$   
 $\quad a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z$

As produções 32 a 35 estão incluídas para tornar completa a gramática. Como veremos mais adiante, será mais conveniente considerar números e identificadores como símbolos terminais da linguagem. Será conveniente, também, impor para uma dada implementação o número máximo de caracteres que pode entrar na formação dos números e identificadores.

## 5.7 COMENTÁRIOS

A inclusão de comentários não modifica o significado do programa. Os comentários serão delimitados pelos símbolos compostos (\* e \*). Dentro de um comentário pode ocorrer qualquer sequência de caracteres, exceto o símbolo composto \*).

### Exercícios

1. Usando a gramática deste capítulo, indique as árvores de derivação para as cadeias:
  - (a)  $(x*(y+z))>t$  (a partir de  $\langle \text{expressão} \rangle$ );
  - (b) **if** *b* **then** *x:=y*  
**else if** *c* **then** *p* **else** *q(x)* (a partir de  $\langle \text{comando} \rangle$ ).
2. Considere a cadeia:  
 $\text{if } b \text{ then while } c \text{ do}$   
 $\quad \text{if } d \text{ then } p \text{ else } q$ 
  - (a) Indique todas as árvores de derivação dessa cadeia a partir do não-terminal  $\langle \text{comando} \rangle$ .

- (b) Quais são as consequências do resultado da parte (a), e qual é a solução a ser adotada?
3. Quantos símbolos de entrada deveriam ser consultados por um analisador descendente sem retrocesso que seguisse a gramática deste capítulo? Justifique.
4. A gramática deste capítulo não distingue entre expressões booleanas e inteiras. Reescreva a gramática de maneira a introduzir esta distinção. Quais são os problemas acarretados por esta gramática? (Sugestão: Considere o que acontece se for usado um analisador descendente.)
5. (Projeto) Escreva um analisador descendente recursivo para a gramática deste capítulo. (Suponha que já existe um analisador léxico do tipo descrito no capítulo seguinte.)

### NOTAS BIBLIOGRÁFICAS

A descrição da linguagem Pascal apareceu em Wirth (1971a). A mesma descrição, bem como o manual do usuário, apareceram em Jensen e Wirth (1974). Existem também muitos textos didáticos para o ensino de Pascal.

# Análise Léxica

## 6.1 GENERALIDADES

De acordo com a descrição dada no Capítulo 5, um programa em Pascal é uma sentença de uma gramática livre de contexto, cujo vocabulário terminal inclui vários tipos de símbolos:

- letras: *a,b,c,...,z*
- dígitos: *0,1,2,...,9*
- símbolos especiais: *.,;,(,),:=,<,>,+, -,\*,[],*
- símbolos especiais compostos: *:=,...,(\*,\*)*
- símbolos compostos em negrito: **program, label, type, array, of, var, procedure, function, begin, end, if, then, else, while, do, or, and, div, not.**

Na prática, porém, somos obrigados a utilizar um vocabulário diferente, devido a várias razões:

- Muitos sistemas não dispõem de letras minúsculas, sendo estas substituídas pelas maiúsculas.
- Muitos símbolos especiais não são encontrados em certos sistemas, sendo substituídos por outros. Exemplos comuns de substituição: *:* por *,,* [e] por *(/ e /)*.
- Sistemas comuns não dispõem de caracteres em negrito, sendo necessário algum tipo de representação. O mais frequente é adotar-se caracteres comuns, resultando representações como *begin* e *end* (ou *BEGIN* e *END*). As sequências de caracteres que representam símbolos compostos em negrito são chamadas de *palavras-chaves*.

A substituição dos símbolos em negrito sugerida acima pode acarretar certos problemas, dificultando a distinção, por exemplo, entre a palavra-chave **begin** e o identificador *begin*. Na prática costuma-se resolver o problema colocando-se entre apóstrofos as palavras-chaves, obtendo-se assim '*begin*' e '*end*' (ou '*BEGIN*' e '*END*'). Uma outra solução conveniente, adotada em muitas implementações, é tratar as palavras-chaves como palavras reservadas que não podem ser usadas como identificadores. Para que não haja confusão entre palavras reservadas, identificadores e números, deverá haver espaços em branco separando os mesmos. Assim, *if a* e *while 35* são identificadores, enquanto *if a* e *while 35* representam *if a* e *while 35*. Neste capítulo suporemos o uso de palavras reservadas.

Deve-se notar que algumas linguagens de programação permitem o uso de palavras-chaves como identificadores, como por exemplo FORTRAN e PL/I. Isto

dificulta a análise de programas, como mostra o seguinte exemplo de uma construção em PL/I:

*DECLARE (X<sub>1</sub>,X<sub>2</sub>,...,X<sub>n</sub>)*

onde *X<sub>i</sub>* são identificadores, *n* ≥ 1. Esta construção pode indicar uma declaração, ou então um comando de chamada de um procedimento denominado *DECLARE*. A distinção e, portanto, a compilação desta construção só podem ser determinadas depois de verificar o primeiro símbolo que segue o parêntese direito. Como o valor de *n* é arbitrário, o compilador deverá ser capaz de verificar um número ilimitado de símbolos para frente, se bem que toda implementação real imporia algum limite para o valor de *n*.

Os problemas discutidos acima referem-se apenas à maneira como os programas são representados fisicamente devido às limitações dos sistemas utilizados e, algumas vezes, à conveniência do leitor humano. É comum que esta representação varie de um sistema para outro ou então, num mesmo sistema, de um dispositivo de comunicação para outro. Cada representação distinta poderia ser refletida na descrição da linguagem, modificando-se convenientemente algumas produções e mudando-se o vocabulário terminal. Entretanto, isto acarretaria a construção de analisadores sintáticos diferentes para cada representação, quando na realidade as modificações na gramática são muito pequenas. Uma outra desvantagem desta solução é que um analisador sintático assim implementado trataria uma palavra-chave *begin* como uma sequência dos símbolos terminais *b, e, g, i e n*, acarretando ineficiências. Na realidade, esta estrutura da palavra-chave *begin* não tem nenhuma relação com o seu significado, e a única informação que interessa ao analisador sintático é que esta é uma representação do símbolo terminal **begin**.

Os identificadores apresentam um problema semelhante ao das palavras-chaves, pois o significado de um identificador como *índice*<sup>1</sup> não tem nenhuma relação com o fato de ele ser constituído pelos particulares símbolos *i,n,d,i,c* e *e*.<sup>2</sup> Este já não é o caso, por exemplo, de constantes numéricas, cuja representação indica o valor correspondente. É comum porém, por motivo de eficiência, tratar números de maneira semelhante a identificadores.

Todas estas razões levam, na prática, a uma separação da análise de programas em duas partes:

- análise léxica, que se encarrega de isolar as palavras-chaves, símbolos especiais, símbolos especiais compostos, identificadores e constantes, transformando-os em códigos convenientes;
- análise sintática propriamente dita, que utiliza os resultados da análise léxica toda vez que necessita de mais um símbolo terminal da cadeia de entrada.

Uma sequência de caracteres que representa um símbolo terminal da linguagem é chamada de átomo.<sup>3</sup> Exemplos de átomos para o nosso Pascal seriam: *begin, +, :, =, ;, :, =, índice, 308*. (Note-se que a nossa sintaxe não permite que o símbolo terminal *:* seja seguido do símbolo terminal *=*, facilitando assim a identificação do símbolo composto *:=*; observação análoga vale para o símbolo composto *... .*) Em geral, a cada símbolo da linguagem associa-se um código distinto (por exemplo, um número inteiro). O analisador léxico<sup>4</sup> processa o programa-fonte,

<sup>1</sup>Nos identificadores que são palavras da língua portuguesa utilizaremos os acentos, til, trema e cedilha. Suporemos que estes sinais são ignorados na representação física de programas.

<sup>2</sup>Isto poderia não ser verdade em FORTRAN ou PL/I, onde o fato de o identificador *ÍNDICE* começar pela letra *I* poderia ter significado.

<sup>3</sup>Em inglês utiliza-se também o termo *token*.

<sup>4</sup>Em inglês utiliza-se também o termo *scanner*.

devolvendo os códigos dos átomos encontrados. No caso de átomos que são identificadores ou números, deve ser devolvida também a própria sequência de caracteres, ou outra informação equivalente, como a posição na tabela de símbolos, ou o valor do número em representação interna.

A separação das análises léxica e sintática traz várias vantagens:

- Toda parte referente à representação dos terminais está concentrada numa única rotina do compilador, tornando mais simples as modificações da representação.
- As regras de formação dos átomos são mais simples do que o resto da sintaxe, permitindo o uso de técnicas de análise mais simples e mais eficientes.
- Certos problemas criados pelas idiossincrasias das linguagens de programação, como o uso de palavras-chaves como identificadores, podem ser resolvidos pelo analisador léxico, mantendo a clareza conceitual do analisador sintático.
- Uma parte apreciável do tempo de compilação corresponde à análise léxica. A sua implementação como uma rotina separada do compilador facilita a introdução de certas otimizações, ou o uso de linguagem de montagem, quando o resto do compilador está escrito numa linguagem de alto nível. Muitos sistemas possuem instruções de máquina especialmente apropriadas para realizar análise léxica. Neste caso, a parte do compilador que depende do sistema utilizado poderá estar concentrada numa única rotina.

Normalmente o analisador léxico executa várias tarefas adicionais, tais como:

- Leitura e impressão do programa-fonte, incluindo o processamento de fins de linha, fins de arquivo, etc.
- Eventual reedição do programa num formato mais legível, usando indentação.
- Eliminação de comentários.
- Eventual manipulação das tabelas de símbolos.
- Diagnóstico e tratamento de alguns erros.

Não são muitos os erros que podem ser detetados pelo analisador léxico. Exemplos típicos são símbolo especial desconhecido, ou identificador ou número mal formados. O tratamento de erros será discutido, de uma maneira mais geral, no Capítulo 10.

## 6.2 IMPLEMENTAÇÃO DE ANALISADORES LÉXICOS

A discussão da seção anterior indica dois aspectos da análise léxica. Por um lado, ela tem um papel importante no processo de compilação, e pode influir de maneira significativa sobre a sua eficiência. Por outro lado, a análise léxica parece ser bastante simples de implementar se comparada com outras partes do compilador. Podemos distinguir duas abordagens comuns à implementação de analisadores léxicos: simulação de autômatos finitos ou então construção *ad hoc*. Um estudo da Teoria dos Autômatos Finitos está fora do escopo deste texto, e na realidade muito pouco desta teoria seria usado na implementação de um analisador léxico. A sua aplicação baseia-se no fato de que a estrutura dos átomos de uma linguagem como Pascal é, em geral, muito simples. Usando-se técnicas semelhantes às da análise sintática, mas particularizadas para átomos, chega-se a analisadores muito simples. Não discutiremos estas técnicas, mas indicamos na Figura 6.1 um analisador típico obtido desta maneira para o nosso Pascal. O procedimento *PRÓXIMO* é usado agora para colocar na variável *próximo* o próximo caráter da cadeia de entrada. No

caso de fim de registro deve ser devolvido um espaço em branco (indicado por  $\square$ ), e *PRÓXIMO* deve ler o próximo registro. No caso de fim de arquivo deve ser devolvido o símbolo especial  $\#$ . A tabela *CÓDIGO* contém os códigos correspondentes aos símbolos especiais e palavras-chaves. O procedimento de análise deve na variável global *símbolo* o código do átomo encontrado. No caso de identificadores ou números será devolvido também o próprio átomo, na variável global *átomo*. A função das constantes como *letras* e *código-de-identificador* é evidente. O símbolo  $\&$  denota a operação de concatenação de cadeias de caracteres. O procedimento de análise léxica deteta caracteres especiais não-pertencentes ao vocabulário, bem como cadeias de dígitos e letras que não correspondem nem a números e nem a identificadores. Os espaços em branco são ignorados exceto quanto à sua função de separar átomos não constituídos de símbolos especiais. Não está incluído o tratamento de comentários.

Note-se que o procedimento da Figura 6.1 segue muito naturalmente da

```

procedimento ANALISADOR_LÉXICO;
início
    átomo:=cadeia_vazia;
    enquanto próximo='□' faça PRÓXIMO;
    se próximo ∈ símbolos_especiais
        então {s:=próximo; PRÓXIMO;
            caso s de
                ':' : se próximo='='
                    então {s:='='; PRÓXIMO};
                '.' : se próximo='.'
                    então {s:='.'; PRÓXIMO};
                outros: nada
            fim do caso;
            símbolo:=CÓDIGO(s)
            senão
                se próximo ∈ letras
                    então {repita
                            átomo:=átomo & próximo; PRÓXIMO
                            até próximo ∉ letras_e_dígitos;
                            se átomo ∈ palavras_chave
                                então símbolo:=CÓDIGO(átomo)
                            senão símbolo:=código_de_identificador }
                senão
                    se próximo ∈ dígitos
                        então {repita
                            átomo:=átomo & próximo; PRÓXIMO
                            até próximo ∉ dígitos;
                            se próximo ∈ letras então ERRO;
                            símbolo:=código_de_número }
                senão ERRO
            fim
    fim

```

Fig. 6.1

discussão da seção anterior, sem usar o conceito de autômato finito, se bem que este está implícito na construção. Existem sistemas para a construção automática de analisadores léxicos, partindo, em geral, de uma descrição formal dos átomos por meio de gramáticas simplificadas. O uso destes sistemas tem a vantagem óbvia de simplificar o trabalho de programação e verificação. Entretanto, devido à generalidade de tais sistemas, dificilmente podem-se obter programas eficientes e bem adaptados à linguagem em questão.

Nos compiladores reais, o analisador léxico costuma ser programado à mão, aproveitando-se, quando conveniente, os recursos da linguagem de programação e do computador utilizados. Alguns computadores possuem instruções de busca que permitem localizar caracteres pertencentes a conjuntos especificados, em cadeias de caracteres. Estas instruções são, em geral, muito convenientes para a implementação do analisador léxico, eliminando muitas vezes a análise caráter por caráter.

Devemos observar, finalmente, que o analisador léxico indicado na Figura 6.1 poderia ser usado por um analisador sintático que verifica apenas um símbolo terminal para frente, a fim de tomar as suas decisões. Este é o caso da análise de precedência simples e de operadores, e da análise para gramáticas do tipo LL(1) e LR(1). Se o analisador sintático deve consultar mais símbolos, pode-se implementar um analisador léxico que usa, por exemplo, as variáveis globais *símbolo1*, *átomo1*, *símbolo2* e *átomo2* para deixar as informações correspondentes a dois átomos consecutivos.

#### Exercícios

1. Estabeleça uma representação para os símbolos do Pascal conveniente para o sistema de computação que está à sua disposição.
2. (Projeto) Escreva um analisador léxico consistente com o exercício anterior, e que indica o próximo símbolo na cadeia de entrada.
3. (Projeto) Reescreva o programa do exercício anterior, para que o analisador indique os dois próximos símbolos na cadeia de entrada. O analisador deve avançar de um só símbolo a cada chamada.

---

#### NOTAS BIBLIOGRÁFICAS

Todos os textos dedicados à compilação contêm uma discussão da análise léxica, incluindo, em geral, uma introdução à teoria dos autômatos finitos.

---

# 7

## Diagramas de Execução

### 7.1 GENERALIDADES

Já vimos, no Capítulo 2, que a descrição de uma linguagem de programação compõe-se de duas partes bastante distintas, isto é, a sintaxe e a semântica. Nos Capítulos 2 a 6 foram discutidos apenas os aspectos sintáticos. No Capítulo 5 vimos a descrição de uma versão simplificada da linguagem Pascal, cuja implementação será discutida nos capítulos subsequentes. Quanto à semântica dessa versão, indicamos apenas que ela coincide com a semântica do Pascal oficial, quando aplicável.

O objetivo deste capítulo é introduzir uma notação que permitirá simular a execução de programas em Pascal de maneira sistemática, indicando claramente a relação entre o programa-fonte, que é um objeto estático, e a sua execução dinâmica. Esta notação, que chamaremos de *diagramas de execução*, tem por objetivo tornar mais clara a semântica do Pascal, especialmente no que se refere às suas regras de escopo e mecanismos de chamada de procedimentos e de passagem de parâmetros. Esta notação será utilizada nos capítulos seguintes para justificar o desenvolvimento de um sistema de execução para o Pascal.

Uma vez que o nosso uso destes diagramas será apenas para melhorar a compreensão intuitiva dos conceitos envolvidos, não chegaremos a definir rigorosamente os diagramas de execução. A notação será introduzida principalmente através de exemplos de nível crescente de complexidade. Deve-se notar que é possível desenvolver um formalismo adequado para esta notação, e relacioná-lo com a sintaxe, obtendo-se assim um método para especificar precisamente a semântica do Pascal. As simulações por meio de diagramas de execução poderiam ser programadas para o computador, constituindo uma maneira muito inefficiente de implementar a linguagem.

### 7.2 ESTRUTURA DE PROGRAMAS

Um programa em Pascal é um objeto que possui certa estrutura revelada pela sua árvore de derivação. Esta estrutura resulta de várias construções sintáticas permitidas em Pascal, sendo de particular interesse para nós as construções que determinam escopos. O *escopo* de uma definição é a região do programa onde esta definição se aplica. Em Pascal, os escopos das declarações estão associados com procedimentos.<sup>1</sup> A fim de realçar este fato, colocaremos grandes colchetes do lado esquerdo das declarações de procedimentos, como será visto nos exemplos a seguir; o programa todo também será tratado como um procedimento.

---

<sup>1</sup>A não ser que haja observação em contrário, o termo *procedimento* incluirá procedimentos e funções.

### 7.3 SIMULAÇÃO DA EXECUÇÃO

A simulação da execução de um programa será feita através de diagramas compostos por retângulos encaixados, sendo que cada retângulo corresponde à ativação de um procedimento. Para indicar o início da ativação de um procedimento  $p$ , abriremos a parte superior de um retângulo, que será denominado *registro de ativação de  $p$* , e será rotulado com o mesmo nome  $p$ . O retângulo só será completado ao encerrar-se a simulação da execução desta ativação de  $p$ . Na parte superior do retângulo serão indicados os valores dos parâmetros e das variáveis locais. Os valores sucessivos indicados para cada parâmetro e variável serão os resultados da execução de comandos de atribuição e de entrada de dados. Se um procedimento  $q$  for chamado dentro de um procedimento  $p$ , então o registro de ativação correspondente de  $q$  estará encaixado no de  $p$ . Dentro de um mesmo registro de ativação, os registros encaixados seguirão a ordem cronológica das chamadas. Convenções adicionais serão indicadas nos exemplos que seguem.

#### Exemplo 7.1

A Figura 7.1 apresenta um programa muito simples sem procedimentos, e a Figura 7.2 indica o diagrama de execução correspondente, supondo que os valores lidos são 5 e 8.

```
program exemplo1(input,output);
var m,n,s: integer;
begin
  read(m,n);
  s:=0;
  while m<=n
    do begin
      s:=s+m*m;
      write(m,s);
      m:=m+1
    end
  end.
```

Fig. 7.1

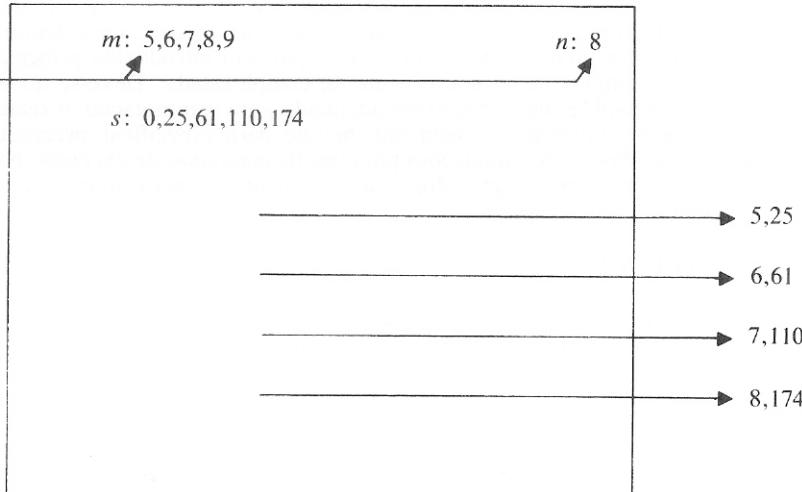


Fig. 7.2

#### Exemplo 7.2

A Figura 7.3 apresenta um programa que contém um procedimento sem parâmetros; o diagrama de execução correspondente está na Figura 7.4, supondo que foi lido o valor 3.

```
program exemplo2(input,output);
var n,s,i: integer;
procedure soma;
var q: integer;
begin
  q:=i * i;
  if (i div 2)*2 = i
    then s:=s+q
    else s:=s-q
end (* soma *);
begin
  read(n);
  s:=0; i:=0;
  while i<=n
    do begin
      soma; write(s); i:=i+1
    end
  end.
```

Fig. 7.3

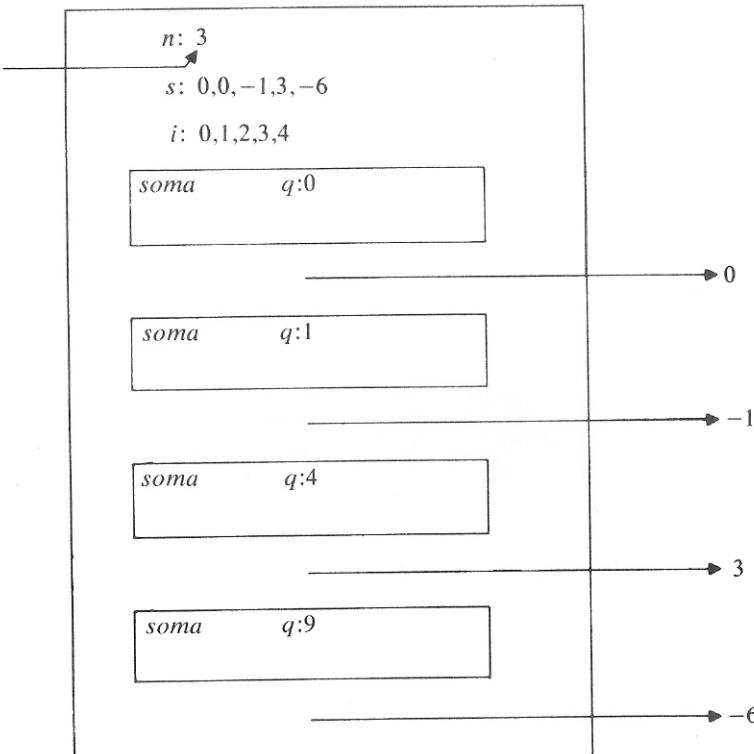


Fig. 7.4

### Exemplo 7.3

A Figura 7.5 mostra um programa com um procedimento que tem um parâmetro passado *por valor*. Como o procedimento é chamado em vários lugares do programa-fonte, introduzimos a convenção adicional de numerar essas chamadas. Esses números são utilizados no diagrama de execução da Figura 7.6, permitindo a identificação do ponto do programa-fonte ao qual deve-se retornar após a execução do procedimento.

Note-se que o programa usa o mesmo identificador *t* para duas variáveis distintas. No caso desse exemplo, é suficiente utilizar a variável do registro de ativação mais interno que contém o ponto de execução corrente. ●

```
program exemplo3(input,output);
var z,t: integer;
procedure g(t: integer);
  var x: integer;
begin
  t:=2*t; x:=2*t; z:=x+1
end (* g *);
begin
  z:=3; t:=4;
  g(t)①; write(z,t);
  g(z)②; write(z,t);
  g(t+z)③; write(z,t);
  g(7)④; write(z,t)
end.
```

Fig. 7.5

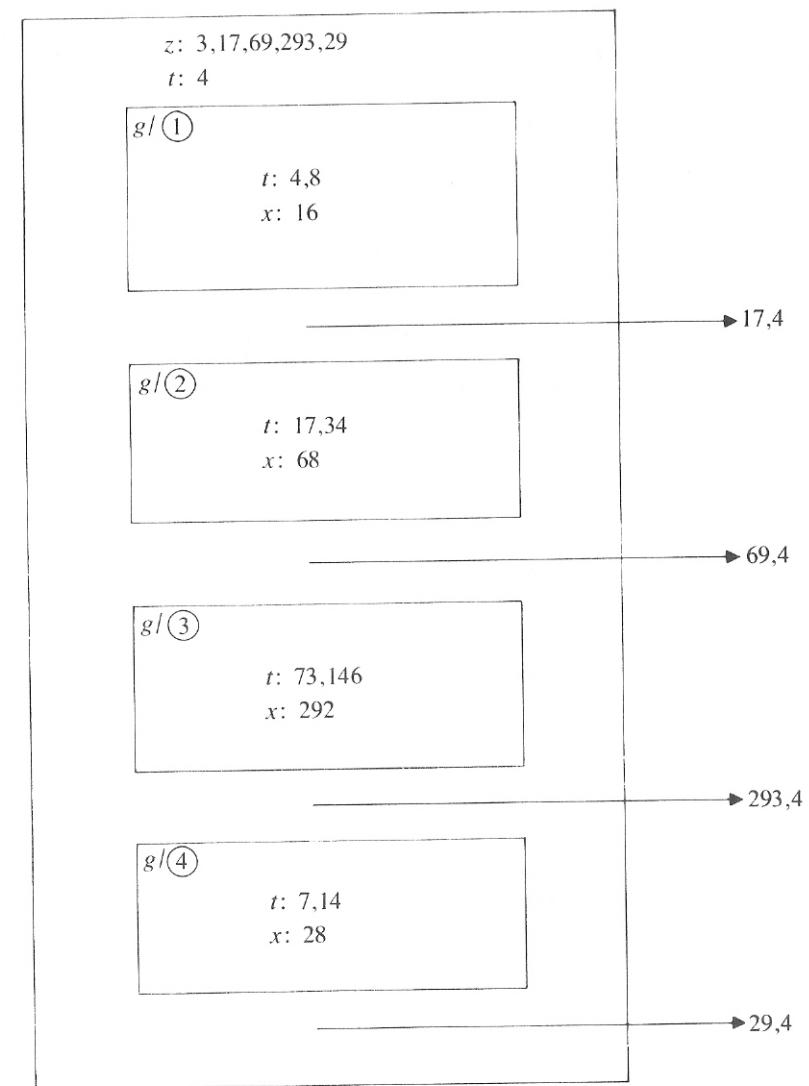


Fig. 7.6

#### Exemplo 7.4

A Figura 7.7 apresenta um programa com vários procedimentos. Segundo a definição do Pascal, os escopos são determinados de maneira estática, de acordo com o programa-fonte. Assim, a Figura 7.8 apresenta o diagrama de execução correspondente. Note-se que nos pontos marcados com símbolos \$ e §, correspondendo à execução do comando  $z:=z+x+y$  no corpo do procedimento  $g$ , a variável  $x$  utilizada é a do programa principal, apesar de existir nestes pontos de execução a variável  $x$  do procedimento  $h$ . Isto deve-se ao fato de que, estaticamente, o procedimento  $g$  está encaixado no programa principal, mas não no procedimento  $h$ . A fim de interpretar corretamente as regras de escopo do Pascal, introduzimos então uma convenção adicional. O registro de ativação de um procedimento estará ligado por meio de uma flecha ao registro de ativação do procedimento dentro do qual o procedimento chamado foi declarado. Ao procurar o valor de uma variável que não é local ao procedimento correto, deveremos seguir a cadeia formada pelas flechas, até encontrar um registro de ativação que contenha uma variável com este nome. Esta cadeia de flechas é chamada às vezes de *cadeia estática*.

```

program exemplo4(input,output);
var y: integer;
procedure g(t: integer);
var y: integer;
begin
y:=t*t; z:=z+x+y;
write(z);
end (* g *);
procedure h(y: integer);
var x: integer;
procedure f(y: integer);
var t: integer;
begin
t:=z+x+y; g(t)① ;
z:=t
end (* f *);
begin
x:=y+1;
f(x)② ;
g(z+x)③ ;
end (* h *);
begin
z:=1; x:=3; h(x)④ ;
g(x)⑤ ;
write(x,z)
end.

```

Fig. 7.7

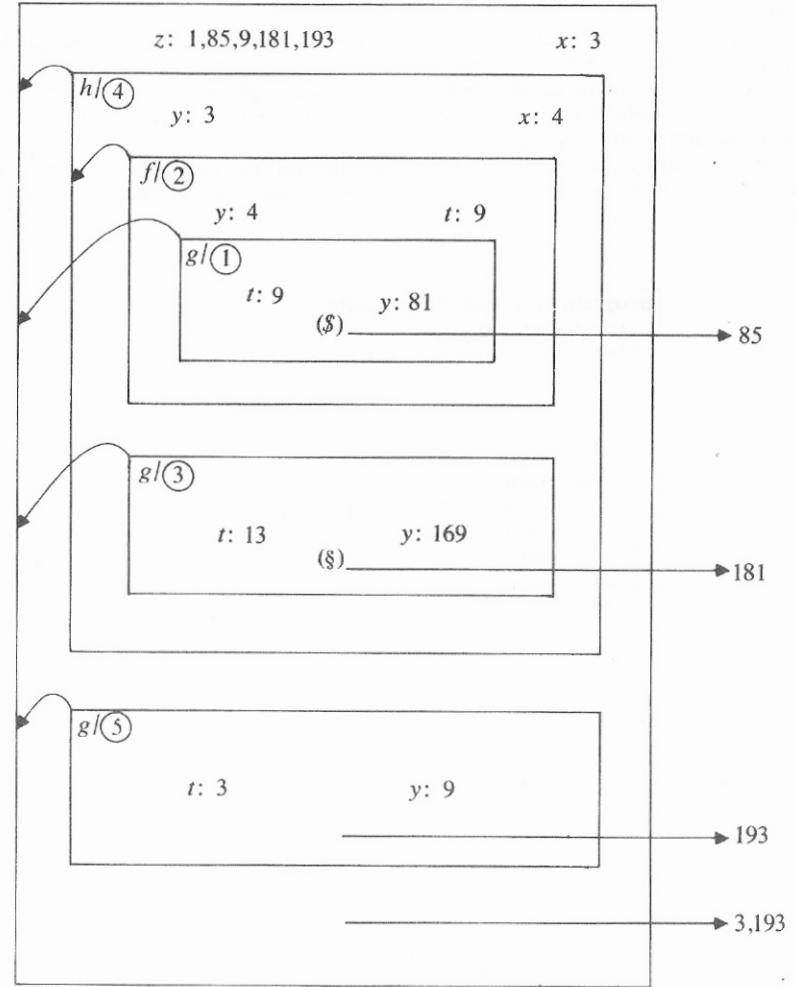


Fig. 7.8

### Exemplo 7.5

Apresentamos na Figura 7.9 um programa com uma função recursiva que tem um parâmetro passado *por referência* (ou *por variável*). No caso destes parâmetros, precisamos indicar de que registro de ativação provém a variável passada como *parâmetro efetivo*.<sup>1</sup> Utilizamos para tal uma flecha ao registro de ativação correspondente, como indicado na Figura 7.10.

Uma vez que  $f$  é uma função, acrescentamos ao seu registro de ativação uma variável local, também denominada  $f$ . O último valor dessa variável, antes de encerrar o registro de ativação da função, é o valor devolvido.

```
program exemplo5(input,output);
var m: integer;
function f(n: integer; var k: integer): integer;
var p,q: integer;
begin
  if n<2
    then begin f:=n; k:=0 end
    else begin
      f:=f(n-1,p)①+f(n-2,q)②;
      k:=p+q+1
    end;
  write(n,k)
end (*f *);
begin
  write(f(3,m)③ ,m)
end.
```

Fig. 7.9

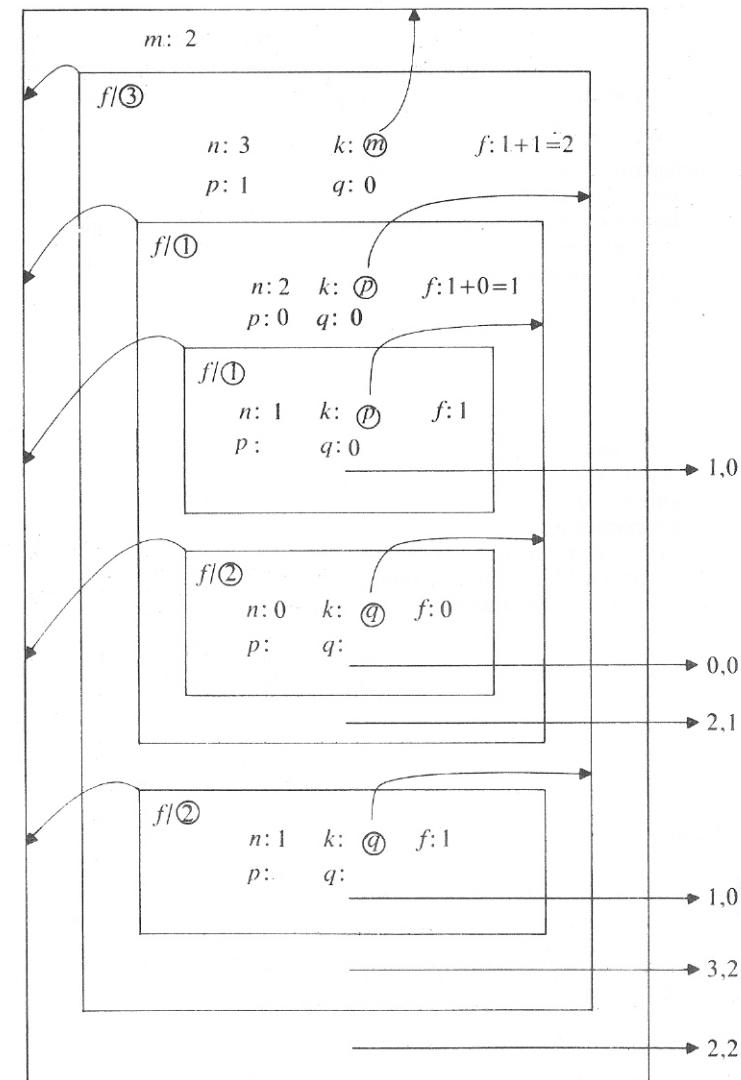


Fig. 7.10

<sup>1</sup> Preferimos esta tradução para o termo inglês *actual parameter*, ao invés de *parâmetro atual*, que interpreta erroneamente o significado da palavra *actual*.

### Exemplo 7.6

As Figuras 7.11 e 7.12 ilustram o uso de procedimentos (no caso, funções) como parâmetros. Como no exemplo anterior, utilizamos uma flecha para indicar o registro de ativação do procedimento em que o parâmetro foi declarado.

```
program exemplo6(input,output);
var a: integer;
function f(m,n: integer; function h: integer; var x: integer):
    integer;
begin
  var s,i: integer;
  begin
    i:=m; s:=0;
    while i<=n
      do begin
        s:=s+h(i)①;
        i:=i+1; a:=a+1
      end;
    f:=s; x:=a
  end (*f *);
procedure g;
var a,b: integer;
function k1(x: integer): integer;
begin k1:=x*x end (* k1 *);
function k2(x: integer): integer;
begin k2:=k1(k1(x)②)③ end (* k2 *);
begin
  write(f(1,3,k1,a)④,a);
  write(f(1,2,k2,b)⑤,b);
end (* g *);
begin
  a:=0; g⑥
end.
```

Fig. 7.11

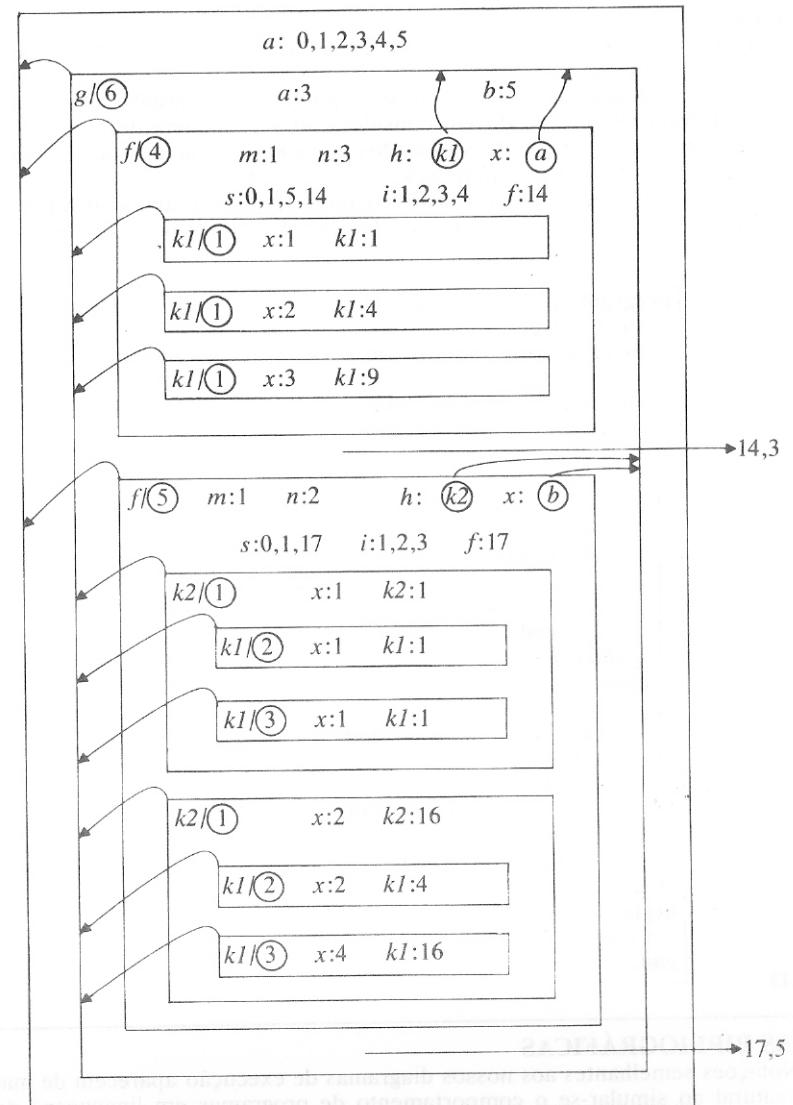


Fig. 7.12

## Exercícios

1. Simule a execução do programa apresentado na Figura 7.13. (Os valores impressos deverão ser 21 e 12.)
2. Adapte a notação desenvolvida neste capítulo para programas em Algol 60 (veja Naur (63)), levando em consideração a existência de blocos com variáveis locais, parâmetros passados por nome, variáveis locais permanentes,<sup>1</sup> rótulos como parâmetros.
3. Adapte a notação desenvolvida neste capítulo para programas em Ada (veja Wegner (1979)), levando em consideração a existência de tarefas concorrentes.

```
program exercício1(input,output);
var t: integer;
function g(n,r: integer; var k: integer): integer;
var t: integer;
function f(i: integer): integer;
var s,t: integer;
begin
  s:=i*i;
  if r<s
    then begin k:=k+1; f:=0 end
  else begin
    f:=g(n-1,r-s,t)+f(i+1);
    k:=k+t
  end
end (*f *);
begin
  if n=0
    then begin k:=1; g:=1 end
  else begin
    k:=0;
    g:=g(n-1,r,t)+2*f(1);
    k:=k+t
  end
end (* g *);
begin
  writeln(g(2,5,t),t)
end.
```

Fig. 7.13

## NOTAS BIBLIOGRÁFICAS

Notações semelhantes aos nossos diagramas de execução aparecem de maneira natural ao simular-se o comportamento de programas em linguagens de programação como Pascal, especialmente quando se trata de procedimentos recursivos. Os diagramas de execução utilizados neste texto são semelhantes ao chamado *modelo de contorno* introduzido em Johnston (1971). Entretanto, preferimos utilizar encaixamento dinâmico ao invés do estático para os registros de ativação de procedimentos. Desta maneira fica mais aparente a correspondência com a pilha de execução a ser introduzida nos capítulos subsequentes. Um exemplo do uso do modelo de contorno para descrever processos computacionais pode ser encontrado também em Organick (1973). Setzer e Melo (1981) utilizam diagramas equivalentes aos nossos, com algumas convenções diferentes.

<sup>1</sup>Em inglês: *own variables*.

# 8

## Sistema de Execução Básico para Pascal

### 8.1 GENERALIDADES

O nosso objetivo final é construir um compilador para Pascal, isto é, construir um programa que traduz um programa-fonte em Pascal para um programa-objeto em linguagem de máquina de um dado computador. Antes de empreender a tarefa de escrever um compilador, devemos estabelecer precisamente a tradução que corresponde a cada construção do Pascal. Entretanto, traduzir para a linguagem de máquina de um computador real é, em geral, uma tarefa muito trabalhosa. As linguagens de máquina têm muitas características com as quais teríamos que nos preocupar, perdendo de vista a correspondência entre as construções do programa-fonte e a sua tradução. Ao invés de traduzir, então, os programas em Pascal para a linguagem de máquina de um computador real, definiremos uma *máquina hipotética*, mais conveniente para as nossas finalidades. Esta abordagem é muito comum na implementação de linguagens de programação, e a máquina hipotética assim definida é chamada de *sistema de execução* ou *sistema de tempo-objeto*<sup>1</sup> para a linguagem. A implementação de um sistema de execução pode ser estudada separadamente, e será vista no Capítulo 9. Suporemos, por enquanto, que esta máquina hipotética é programada numa linguagem de montagem, facilitando a tradução.

Os próximos capítulos são dedicados, então, ao desenvolvimento de um computador hipotético que chamaremos de *MEPA — Máquina de Execução para Pascal*. A mesma sigla será usada para denotar a linguagem de montagem desta máquina. O desenvolvimento será gradual, acompanhando a discussão de vários mecanismos do Pascal. Freqüentemente, seremos obrigados a rever as decisões já tomadas sobre o funcionamento da MEPA, a fim de acomodar um mecanismo. Desta maneira ficarão mais claras as razões para as várias características da MEPA, e a sua relação com as construções do Pascal. Para justificar as nossas decisões, utilizaremos freqüentemente os diagramas de execução introduzidos no capítulo anterior, procurando com que o sistema de execução seja, no fundo, uma implementação eficiente destes diagramas.

Será coberta neste capítulo a versão mínima da linguagem descrita no Capítulo 5; os mecanismos opcionais, cuja sintaxe está marcada com asteriscos, bem como algumas extensões, serão discutidos no capítulo seguinte.

<sup>1</sup>Em inglês: *run-time system*.

## 8.2 CARACTERÍSTICAS GERAIS DA MEPA

Descreveremos nesta seção a estrutura básica da MEPA, que é bastante simples. Como veremos nas seções subsequentes, é o repertório de instruções que é complexo.

Uma decisão importante que tomaremos agora é que a MEPA deverá ser uma máquina a pilha. Isto é muito natural, pois o Pascal permite o uso de procedimentos recursivos, e a recursão está intimamente ligada com o uso da pilha. As razões para esta decisão ficarão mais claras, portanto, quando tratarmos de procedimentos recursivos.

A nossa máquina terá uma memória composta de três regiões:

- A *região de programa P* que conterá as instruções da MEPA. O formato exato de cada instrução é irrelevante para a nossa discussão.
- A *região da pilha de dados M* que conterá os valores manipulados pelas instruções da MEPA. Suporemos que esta região compõe-se de palavras que podem conter valores inteiros ou então indefinidos.
- O *vetor dos registradores<sup>1</sup> de base D* que conterá apontadores (endereços) para a região M, ou então valores indefinidos.

Suporemos que cada uma das três regiões têm palavras numeradas com 0, 1, 2, ..., e não nos preocuparemos com as limitações de tamanho de cada região, nem de cada palavra.

A MEPA terá dois registradores especiais que serão usados para descrever o efeito das instruções:

- O registrador de programa  $i$  conterá o endereço da próxima instrução a ser executada, que será, portanto,  $P[i]$ .
- O registrador  $s$  indicará o elemento no topo da pilha cujo valor será dado, portanto, por  $M[s]$ .

O uso da região D será introduzido mais adiante neste texto.

Uma vez que o programa da MEPA está carregado na região  $P$ , e os registradores têm os seus valores iniciais, o funcionamento da máquina é muito simples. As instruções indicadas pelo registrador  $i$  são executadas até que seja encontrada a instrução de parada, ou ocorra algum erro. A execução de cada instrução incrementa de um o valor de  $i$ , exceto as instruções que envolvem desvios.

Passaremos a desenvolver nas seções seguintes o repertório de instruções da MEPA motivado pelas várias construções do Pascal.

## 8.3 AVALIAÇÃO DE EXPRESSÕES

Suponhamos que  $E$  é uma expressão em Pascal da forma  $E = E_1 \square E_2$ , onde  $E_1$  e  $E_2$  são duas expressões mais simples, e  $\square$  é um operador binário como  $+, -, *, \wedge, <, =$ , etc. A expressão  $E$  deve ser avaliada calculando-se em primeiro lugar os valores de  $E_1$  e  $E_2$  e aplicando-se, em seguida, a operação correspondente a  $\square$ . Este cálculo pode ser implementado de várias maneiras, guardando-se os valores intermediários de  $E_1$  e  $E_2$  em localizações de memória especiais. Numa máquina como a MEPA, uma maneira conveniente é guardar estes valores intermediários na própria pilha  $M$ . Assim, supondo que os valores  $v_1$  e  $v_2$  das expressões  $E_1$  e  $E_2$  são calculados de maneira semelhante, a pilha terá as configurações sucessivas indicadas na Figura 8.1. Denotamos por  $m$  o endereço do topo da pilha antes de iniciar a avaliação da expressão  $E$ , e por  $v$ , o valor final calculado. O símbolo  $?$  denota valores irrelevantes que já estavam na pilha.

Note-se que estamos resolvendo o problema de avaliar expressões de maneira indutiva, supondo em primeiro lugar que sabemos resolvê-lo para expressões mais

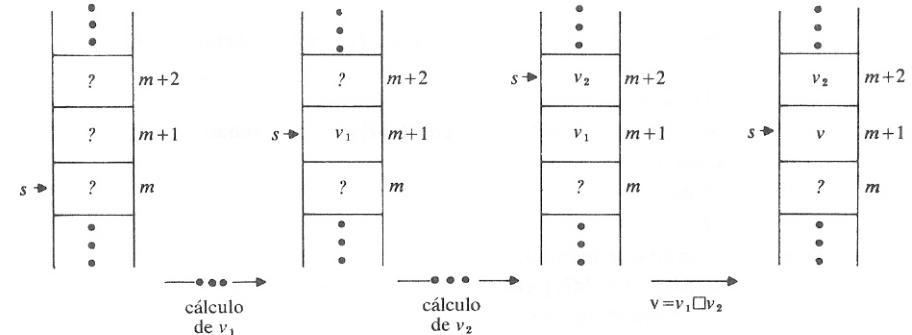


Fig. 8.1

simples, isto é, mais curtas, como  $E_1$  e  $E_2$ . Assim, supusemos na Figura 8.1 que os valores  $v_1$  e  $v_2$  são calculados possivelmente em vários passos, ficando no fim o valor correspondente no topo da pilha, uma posição acima da posição inicial. A base desta indução corresponde às expressões o mais simples possível, isto é, às constantes e variáveis. Para estas, basta colocar os seus valores respectivos no topo da pilha. Os operadores unários serão tratados de maneira análoga. O caso de chamadas de funções que devolvem resultados será tratado mais adiante, mas é importante notar desde já que qualquer que seja a implementação destas chamadas, o seu efeito deverá ser sempre o de deixar o resultado final no topo da pilha, uma posição acima da inicial.

Podemos concluir da discussão acima que a MEPA deve possuir instruções que carregam na pilha valores de constantes e de variáveis, e outras que executam operações correspondentes aos operadores do Pascal. Definiremos, portanto, uma série de instruções para a MEPA, mas devemos notar que algumas dessas definições são provisórias, e serão modificadas mais adiante. O efeito de cada instrução está descrito numa notação semelhante à do Pascal, indicando as modificações no estado dos registradores e da memória da MEPA. Omitimos nesta descrição a operação  $i := i + 1$  que está implícita em todas as instruções, exceto quando há desvio. Adotaremos, também, a convenção de representar os valores booleanos por inteiros: true por 1 e false por 0.

**CRCT k** (Carregar constante):

$s := s + 1; M[s] := k$

**CRVL n** (Carregar valor):

$s := s + 1; M[s] := M[n]$

**SOMA** (Somar):

$M[s - 1] := M[s - 1] + M[s]; s := s - 1$

**SUBT** (Subtrair):

$M[s - 1] := M[s - 1] - M[s]; s := s - 1$

**MULT** (Multiplicar):

$M[s - 1] := M[s - 1] * M[s]; s := s - 1$

**DIVI** (Dividir):

$M[s - 1] := M[s - 1] \text{div } M[s]; s := s - 1$

**INVR** (Inverter sinal):

$M[s] := -M[s]$

<sup>1</sup>A palavra *registrator* será usada como tradução do termo inglês *register*, ficando a palavra *registro* reservada como equivalente do termo *record*.

<i>CONJ</i>	(Conjunção): se $M[s-1]=1$ e $M[s]=1$ então $M[s-1]:=1$ senão $M[s-1]:=0$ ; $s:=s-1$
<i>DISJ</i>	(Disjunção): se $M[s-1]=1$ ou $M[s]=1$ então $M[s-1]:=1$ senão $M[s-1]:=0$ ; $s:=s-1$
<i>NEGA</i>	(Negação): $M[s]:=1-M[s]$
<i>CMME</i>	(Comparar menor): se $M[s-1]<M[s]$ então $M[s-1]:=1$ senão $M[s-1]:=0$ ; $s:=s-1$
<i>CMMA</i>	(Comparar maior): se $M[s-1]>M[s]$ então $M[s-1]:=1$ senão $M[s-1]:=0$ ; $s:=s-1$
<i>CMIG</i>	(Comparar igual): se $M[s-1]=M[s]$ então $M[s-1]:=1$ senão $M[s-1]:=0$ ; $s:=s-1$
<i>CMDG</i>	(Comparar desigual): se $M[s-1]\neq M[s]$ então $M[s-1]:=1$ senão $M[s-1]:=0$ ; $s:=s-1$
<i>CMEG</i>	(Comparar menor ou igual): se $M[s-1]\leq M[s]$ então $M[s-1]:=1$ senão $M[s-1]:=0$ ; $s:=s-1$
<i>C MAG</i>	(Comparar maior ou igual): se $M[s-1]\geq M[s]$ então $M[s-1]:=1$ senão $M[s-1]:=0$ ; $s:=s-1$

### Exemplo 8.1

Consideremos a expressão  $a+(b \text{ div } 9-3) * c$ , e suponhamos que os endereços atribuídos pelo compilador às variáveis  $a$ ,  $b$  e  $c$  são, respectivamente, 100, 102, e 99. Então o trecho do programa-objeto correspondente à tradução desta expressão seria:

```

CRVL 100
CRVL 102
CRCT 9
DIVI
CRCT 3
SUBT
CRVL 99
MULT
SOMA

```

Suponhamos que os valores armazenados nas posições 99, 100 e 102 da pilha são  $-2$ ,  $10$  e  $100$ , respectivamente, e que o registrador  $s$  contém o valor 104. As configurações sucessivas da pilha ao executar as instruções acima são dadas na Figura 8.2. Os valores sucessivos de  $s$  estão indicados pelas flechas.

Uma observação interessante é que o código da MEPA gerado para expressões está diretamente ligado com a *notação polonesa posfixa*, que no caso da expressão do exemplo acima seria  $ab9\text{ div}3-c*+$ . Esta seqüência de símbolos deve ser comparada com a seqüência de instruções da MEPA desse exemplo.

### 8.4 COMANDOS DE ATRIBUIÇÃO

Consideraremos, por enquanto, apenas as atribuições a variáveis simples da forma  $V:=E$ , onde  $V$  é o nome de uma variável e  $E$  é uma expressão que já sabemos

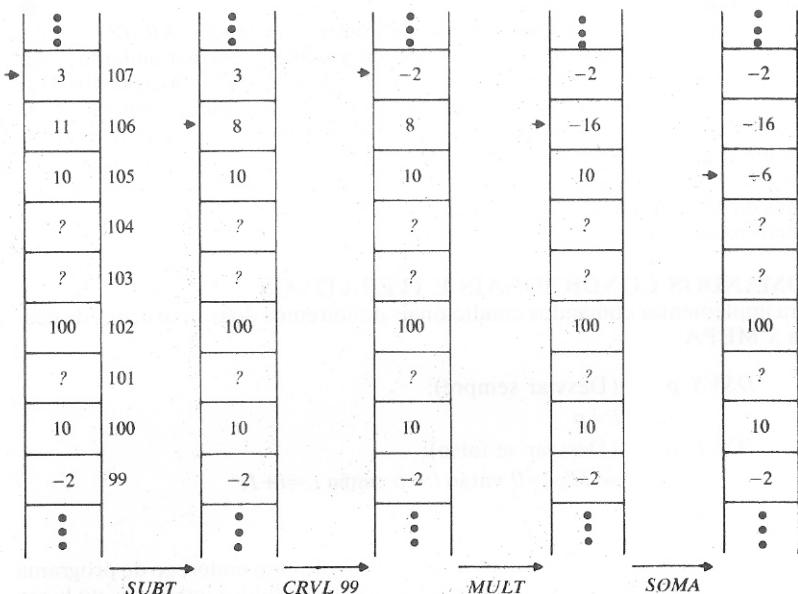
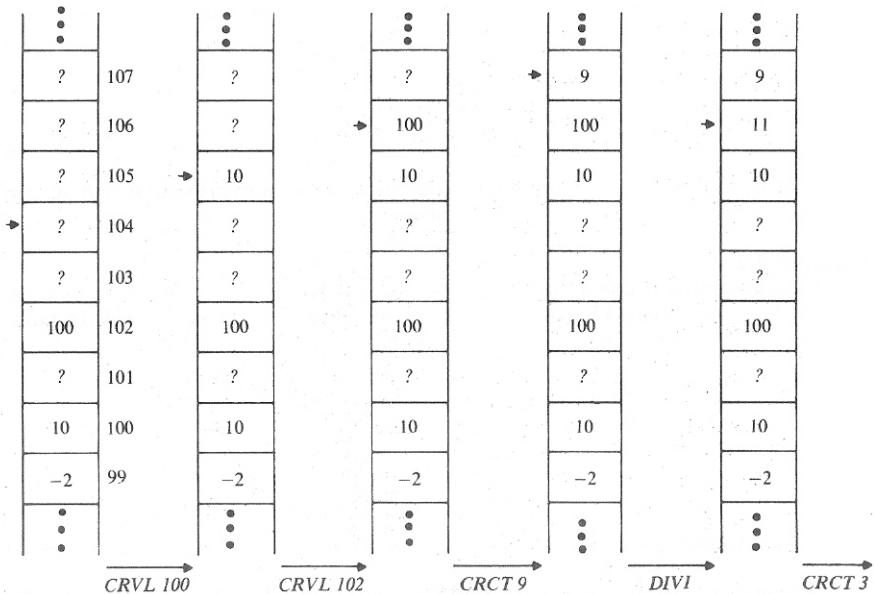


Fig. 8.2

traduzir. Uma maneira simples de implementar este comando é dada por uma instrução de armazenamento:

**ARMZ  $n$**  (Armazenar valor):

$M[n]:=M[s]; s:=s-1$

Esta instrução seguirá o código-objeto que corresponde à expressão  $E$ , e  $n$  será o endereço atribuído pelo compilador à variável  $V$ .

#### Exemplo 8.2

Consideremos o comando  $a:=a+b*c$  e suponhamos que os endereços e os valores destas variáveis são os mesmos do Exemplo 8.1. O código da MEPA para este comando será:

```
CRVL 100
CRVL 102
CRVL 99
MULT
SOMA
ARMZ 100
```

A Figura 8.3 apresenta as configurações sucessivas da pilha ao ser executado este código-objeto.

Note-se que, devido à maneira como definimos a instrução **ARMZ**, o valor final do registrador  $s$ , após a execução do código-objeto correspondente a um comando de atribuição, será igual ao seu valor inicial. Esta é uma propriedade importante que será verdadeira para qualquer comando em Pascal, como verificaremos mais tarde. As únicas exceções são os comandos de desvio, e de chamada de procedimentos e funções quando estes não retornam por causa de desvios.

A fim de simplificar a apresentação, usaremos nos exemplos subsequentes de programas da MEPA nomes simbólicos para variáveis, trocando letras minúsculas pelas maiúsculas.

## 8.5 COMANDOS CONDICIONAIS E ITERATIVOS

Para implementar comandos condicionais definiremos duas instruções de desvio para a MEPA:

**DSVS  $p$**  (Desviar sempre):

$i:=p$

**DSVF  $p$**  (Desviar se falso):

**se**  $M[s]=0$  **então**  $i:=p$  **senão**  $i:=i+1;$

$s:=s-1$

Nestas instruções,  $p$  é um número inteiro que indica um endereço de programa da MEPA. Nos exemplos que se seguem utilizaremos rótulos simbólicos no lugar de números. Note-se que, haja ou não desvio, a instrução **DSVF** elimina o valor que foi testado, e que está no topo da pilha.

Introduziremos, por conveniência, mais uma instrução que não é estritamente necessária, mas que simplifica o processo de tradução e que não tem nenhum efeito sobre a execução:

**NADA** (Nada):

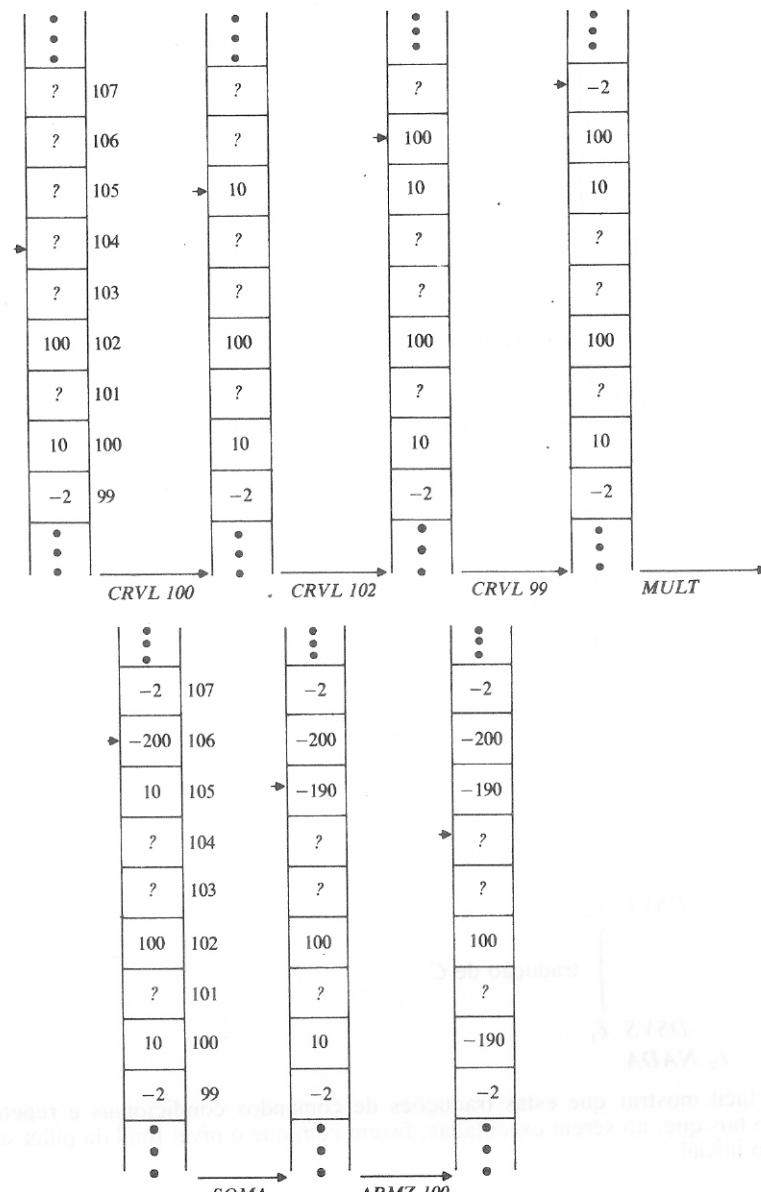


Fig. 8.3

Um comando condicional da forma **if**  $E$  **then**  $C_1$  **else**  $C_2$ , onde  $E$  é uma expressão e  $C_1$  e  $C_2$  são comandos, será traduzido por:

$\left. \begin{array}{l} \text{DSVF } \ell_1 \\ \text{DSVS } \ell_2 \end{array} \right\}$  tradução de  $E$   
 $\left. \begin{array}{l} \text{DSVF } \ell_1 \\ \text{DSVS } \ell_2 \end{array} \right\}$  tradução de  $C_1$   
 $\left. \begin{array}{l} \text{DSVS } \ell_2 \\ \ell_1 \text{ NADA} \end{array} \right\}$  tradução de  $C_2$   
 $\ell_2 \text{ NADA}$

Caso o comando tenha a forma **if**  $E$  **then**  $C$ , podemos traduzi-lo por:

$\left. \begin{array}{l} \text{DSVF } \ell \\ \text{DSVS } \ell \end{array} \right\}$  tradução de  $E$   
 $\left. \begin{array}{l} \text{DSVF } \ell \\ \text{DSVS } \ell \end{array} \right\}$  tradução de  $C$   
 $\ell \text{ NADA}$

As mesmas instruções de desvio podem ser usadas para implementar comandos repetitivos. No caso do comando **while**  $E$  **do**  $C$  teremos a tradução:

$\left. \begin{array}{l} \ell_1 \text{ NADA} \\ \text{DSVF } \ell_2 \end{array} \right\}$  tradução de  $E$   
 $\left. \begin{array}{l} \text{DSVF } \ell_2 \\ \text{DSVS } \ell_1 \end{array} \right\}$  tradução de  $C$   
 $\ell_2 \text{ NADA}$

É fácil mostrar que estas traduções de comandos condicionais e repetitivos são tais que, ao serem executadas, fazem com que o nível final da pilha seja igual ao inicial.

### Exemplo 8.3

Indicaremos a seguir as traduções correspondentes a três comandos em Pascal:

1. **if**  $q$  **then**  $a:=1$  **else**  $a:=2$

$\text{CRVL } Q$   
 $\text{DSVF } L1$   
 $\text{CRCT } 1$

$\text{ARMZ } A$   
 $\text{DSVS } L2$   
 $L1 \text{ NADA}$   
 $\text{CRCT } 2$   
 $\text{ARMZ } A$   
 $L2 \text{ NADA}$

2. **if**  $a > b$  **then**  $q:=p$  **and**  $q:=true$   
**else if**  $a < 2*b$  **then**  $p:=true$   
**else**  $q:=false$

$\text{CRVL } A$   
 $\text{CRVL } B$   
 $\text{CMMA}$   
 $\text{DSVF } L3$   
 $\text{CRVL } P$   
 $\text{CRVL } Q$   
 $\text{CONJ}$   
 $\text{ARMZ } Q$   
 $\text{DSVS } L4$   
 $L3 \text{ NADA}$   
 $\text{CRVL } A$   
 $\text{CRCT } 2$   
 $\text{CRVL } B$   
 $\text{MULT}$   
 $\text{CMME}$   
 $\text{DSVF } L5$   
 $\text{CRCT } 1$   
 $\text{ARMZ } P$   
 $\text{DSVS } L6$

$L5 \text{ NADA}$   
 $\text{CRCT } 0$   
 $\text{ARMZ } Q$   
 $L6 \text{ NADA}$   
 $L4 \text{ NADA}$

3. **while**  $s \leq n$  **do**  $s:=s+3 * s$

$L7 \text{ NADA}$   
 $\text{CRVL } S$   
 $\text{CRVL } N$   
 $\text{CMEG}$   
 $\text{DSVF } L8$   
 $\text{CRVL } S$   
 $\text{CRCT } 3$   
 $\text{CRVL } S$   
 $\text{MULT}$

SOMA  
 ARMZ S  
 DSVS L7  
 L8 NADA

## 8.6 COMANDOS DE ENTRADA E DE SAÍDA

Já indicamos, na Seção 5.4, que a nossa versão do Pascal possui comandos muito simples para entrada e saída de dados, e que tomam forma de chamadas de procedimentos especiais. Um comando da forma  $read(V_1, \dots, V_n)$  deve ler os próximos  $n$  valores inteiros do arquivo de entrada e atribuí-los a variáveis inteiros,  $V_1, \dots, V_n$ , nesta ordem; suporemos por enquanto que  $V_1, \dots, V_n$  são variáveis simples. A fim de implementar o comando de leitura definiremos a seguinte instrução para a MEPA.

**LEIT** (Leitura):

$s := s + 1; M[s] :=$  “próximo valor no arquivo de entrada”

Usando esta instrução, podemos traduzir  $read(v_1, v_2, \dots, v_n)$  por:

LEIT  
 ARMZ  $V_1$   
 LEIT  
 ARMZ  $V_2$   
 .  
 .

LEIT  
 ARMZ  $V_n$

onde  $V_1, \dots, V_n$  são os endereços das variáveis  $v_1, \dots, v_n$ .

O comando de saída do Pascal tem a forma  $write(E_1, E_2, \dots, E_n)$ , indicando que os valores das expressões inteiros  $E_1, E_2, \dots, E_n$  devem ser impressos no arquivo de saída. Definiremos então a instrução:

**IMPR** (Impressão):

“Imprimir  $M[s]$ ;  $s := s - 1$

Assim, o comando indicado acima pode ser traduzido por:

.  
 .  
 .  
 IMPR  
 .  
 .  
 .  
 tradução de  $E_1$   
 tradução de  $E_2$

**IMPR**

.  
 .  
 .  
 } tradução de  $E_n$   
**IMPR**

### Exemplo 8.4

1.  $read(a, b)$

LEIT  
 ARMZ A  
 LEIT  
 ARMZ B

2.  $write(x, x * y)$

CRVL X  
 IMPR  
 CRVL X  
 CRVL Y  
 MULT  
 IMPR

É fácil verificar que, ainda neste caso, a execução das instruções correspondentes à tradução de um comando de entrada ou saída faz com que o valor final do registrador  $s$  seja igual ao inicial.

## 8.7 PROGRAMAS

A Figura 8.4 mostra um programa muito simples, sem procedimentos. Deveria ser claro que, neste caso, o programa-objeto deveria reservar as cinco posições iniciais da pilha para as variáveis, e em seguida começar a executar as instruções

```

program exemplo5(input,output);
var n,k: integer;
    f1 f2 f3: integer;
begin
  read(n);
  f1:=0; f2:=1; k:=1;
  while k<=n
    do begin
      f3:=f1+f2;
      f1:=f2; f2:=f3;
      k:=k+1
    end;
  write(n,f1)
end.
  
```

Fig. 8.4

que correspondem ao bloco de comandos. Definiremos então duas instruções para a MEPA.

**INPP** (Iniciar programa principal):

$s := -1$

**AMEM m** (Alocar memória):

$s := s + m$

A instrução **INPP** será sempre a primeira instrução de um programa-fonte. Uma declaração da forma  $v_1, v_2, \dots, v_m : tipo$  será traduzida por **AMEM m**. Note-se que o compilador pode calcular facilmente os endereços das variáveis  $v_1, v_2, \dots, v_m$  que deverão ser  $0, 1, \dots, m-1$ , respectivamente (quando esta é a primeira declaração de variáveis).

Para completar a tradução de programas, definiremos uma instrução de término de execução:

**PARA** (Parar):

“Pára a execução da MEPA”

#### Exemplo 8.5

Indicamos a seguir o programa-objeto que resulta da tradução do programa da Figura 8.4. Fragmentos do programa-fonte são usados como comentários a fim de tornar mais clara a tradução:

```

INPP      program
AMEM 2   var n,k
AMEM 3   f1,f2,f3
LEIT
ARMZ 0   read(n)
CRCT 0
ARMZ 2   f1:=0
CRCT 1
ARMZ 3   f2:=1
CRCT 1
ARMZ 1   k:=1
LI NADA  while
CRVL 1
CRVL 0
CMEG    k<=n
DSVF L2  do
CRVL 2
CRVL 3
SOMA
ARMZ 4   f3:=f1+f2
CRVL 3
ARMZ 2   f1:=f2
CRVL 4
ARMZ 3   f2:=f3
CRVL 1

```

```

CRCT 1
SOMA
ARMZ 1   k:=k+1
DSVS  LI
L2 NADA
CRVL 0
IMPR
CRVL 2
IMPR      write(n,f1)
PARA      end.

```

Note-se que a tradução de um comando composto delimitado pelos símbolos **begin** e **end** é obtida pela justaposição das traduções dos comandos componentes.

Uma observação importante sobre o nosso sistema de execução é que ele é muito complicado para o caso de programas sem procedimentos. É fácil perceber que, neste caso, o compilador pode determinar os endereços de todas as variáveis e de todas as posições intermediárias na pilha. Poderíamos ter definido, portanto, instruções mais simples para MEPA que fizessem acesso a localizações de memória sem a manipulação explícita da pilha. Esta observação mantém-se também verdadeira para programas que têm procedimentos mas que não são recursivos.

#### 8.8 PROCEDIMENTOS SEM PARÂMETROS

Consideremos o programa indicado na Figura 8.5, em que  $p$  é um procedimento recursivo sem parâmetros. Supondo que o valor lido pelo comando de entrada seja 4, obtém-se o diagrama de execução indicado na Figura 8.6. O número de ativações do procedimento  $p$  dependerá, em geral, do valor de  $n$  que foi lido, não sendo possível determinar-se um limite. Conseqüentemente, num certo instante de execução, poderão existir várias “encarnações” da variável local  $z$  de  $p$ . Por outro lado, as instruções da MEPA que definimos até agora usam endereços fixos de memória, como por exemplo **CRVL 3** ou **ARMZ 7**. Isto significa que, ao traduzir expressões que envolvem a variável  $z$ , o compilador deverá usar um endereço fixo para esta variável. Entretanto, as várias encarnações não podem ocupar a mesma

```

program exemplo6(input,output);
var x,y: integer;
procedure p;
var z: integer;
begin
  z:=x; x:=x-1;
  if z>1 then p①
  else y:=1;
  y:=y*z
end (* p *);
begin
  read(x);
  p②;
  write(x,y)
end.

```

Fig. 8.5

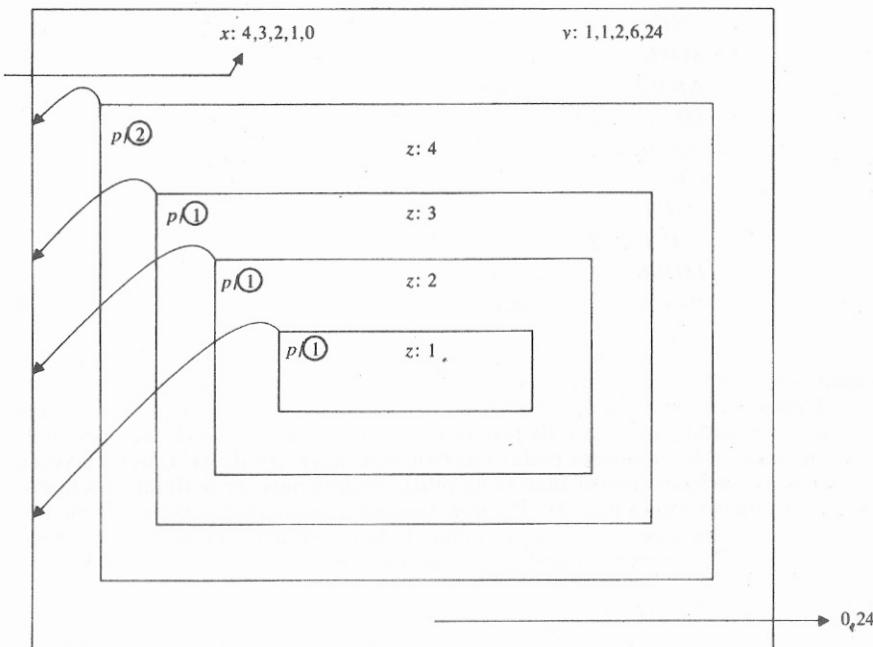


Fig. 8.6

posição de memória. O problema pode ser resolvido notando-se que a criação e a destruição das encarnações de  $z$  seguem uma disciplina de pilha, isto é, a última encarnação a ser alocada é a primeira a desaparecer. Um outro fato, que podemos concluir observando as flechas estáticas da Figura 8.6, é que num dado instante o programa só tem acesso à encarnação de  $z$  criada por último. O problema de endereçamento será resolvido, então, usando-se a própria pilha. Sempre que o procedimento  $p$  for chamado, o valor anterior de  $z$  será temporariamente guardado no topo da pilha, liberando a posição fixa, atribuída pelo compilador à variável  $z$ , para a próxima encarnação. O valor anterior de  $z$  será restaurado antes de executar o retorno. A Figura 8.7 indica as configurações sucessivas da pilha para cada chamada do procedimento  $p$ . Após cada retorno, a configuração prévia será restaurada. Os símbolos  $z_1, z_2, z_3$  e  $z_4$  denotam os valores das encarnações sucessivas de  $z$ .

Não é difícil ver que esta solução aplica-se no caso geral de procedimentos sem parâmetros. Cada procedimento alocará a sua memória, salvando no topo da pilha o conteúdo prévio das posições de memória correspondentes às suas variáveis locais. Estes valores serão restaurados antes de executarem-se os retornos. A fim de tornar o processo de tradução uniforme, o programa inteiro também será tratado como um procedimento. Redefiniremos, então, a instrução  $AMEM$ , e introduziremos uma nova instrução  $DMEM$ :

```
AMEM  $m,n$  (Alocar memória):
para  $k:=0$  até  $n-1$  faça
     $\{s:=s+1; M[s]:=M[m+k]\}$ 
```

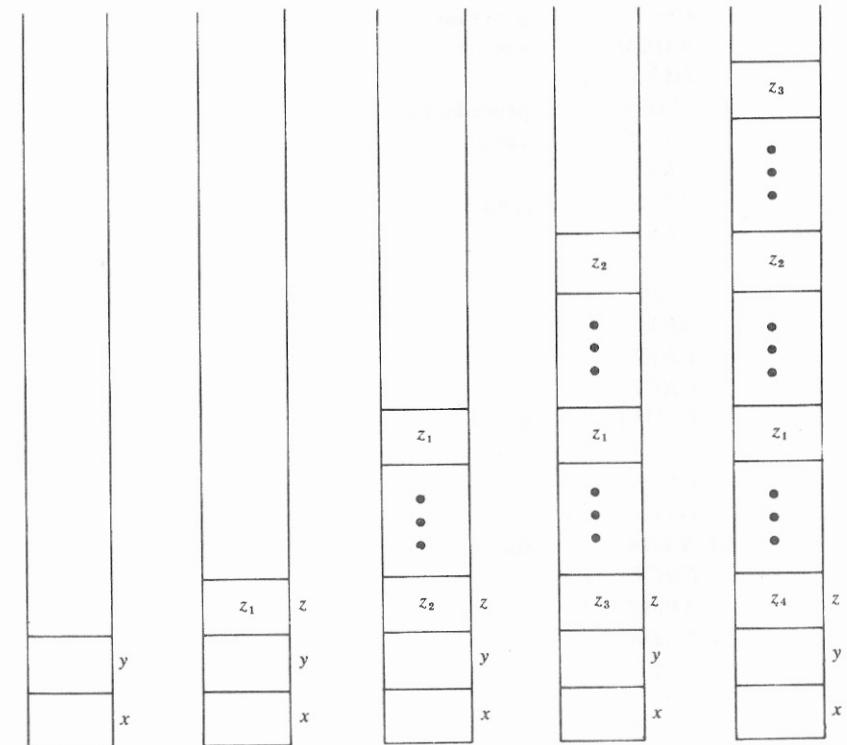


Fig. 8.7

```
DMEM  $m,n$  (Desalocar memória):
para  $k:=n-1$  até  $0$  faça
     $\{M[m+k]:=M[s]; s:=s-1\}$ 
```

Um outro problema a ser resolvido é o próprio mecanismo de chamada de procedimentos. A única informação necessária para executar o retorno, neste estágio da MEPA, é o endereço da instrução à qual a execução deve retornar. O lugar natural para guardar esta informação é a própria pilha. Definiremos, portanto, as instruções:

```
CHPR  $p$  (Chamar procedimento):
 $s:=s+1; M[s]:=i+1; i:=p$ 
RTPR (Retornar de procedimento):
 $i:=M[s]; s:=s-1$ 
```

Note-se que a instrução  $RTPR$  sempre encontra a informação correta no topo da pilha (veja Exercício 8.19).

#### Exemplo 8.6

O programa da Figura 8.5 produz a seguinte tradução:

```

INPP      program
AMEM    0,2  var x,y
DSVS    L1
L2 NADA   procedure p
AMEM    2,1  var z
CRVL    0
ARMZ    2  z:=x
CRVL    0
CRCT    1
SUBT
ARMZ    0  x:=x-1
CRVL    2
CRCT    1
CMMA    if z>1
DSVF    L3  then
CHPR    L2  p
DSVS    L4
L3 NADA   else
CRCT    1
ARMZ    1  y:=1
L4 NADA
CRVL    1
CRVL    2
MULT
ARMZ    1  y:=y*z
DMEM    2,1  end
RTPR
L1 NADA
LEIT
ARMZ    0  read(x)
CHPR    L2  p
CRVL    0
IMPR
CRVL    1
IMPR    write(x,y)
DMEM    0,2  end.
PARA

```

Note-se o uso da instrução *DSVS L1* para pular o código do procedimento *p* ao iniciar-se a execução. Desta maneira, a tradução segue a mesma ordem do programa-fonte. Deve-se notar, também, a maneira como o compilador atribui endereços às variáveis *x*, *y* e *z*, que são, no caso, 0, 1 e 2, respectivamente.

A Figura 8.8 mostra o resultado da simulação da execução do programa-objeto acima. Nesta simulação, cada linha indica uma posição da pilha, e contém os

11		1	1	1	
10			1	1	1 1 0 1 ?
9		1	1	2	2
8			2 2 1 2 ?	1 1	1 2
7		1	1	3	3
6			3 3 2 3 ?	1 1	2 6
5		1	1	4	4
4			4 4 3 4 ?	1 1	6 24
3			3 1		
2		4 3 1 4	3	2	1
1				2	3  4  3 1 0 24
0		? 4	3	2	1
					0

Saída: 0,24

Fig. 8.8

valores consecutivos na pilha. Uma barra vertical depois de um valor indica que este valor foi eliminado pela gravação de um novo valor, após a execução de uma instrução como *SOMA* ou *ARMZ*, ou então que este valor torna-se irrelevante por estar acima do topo da pilha, como no caso de *DMEM*, *SOMA*, *ARMZ*, etc. O símbolo ? indica um valor indefinido numa posição de memória reservada por *AMEM*. Com estas convenções, o nível da pilha em cada instante corresponde à posição mais alta que contém algum valor ou o símbolo ? não seguidos da barra. Para facilitar a compreensão da figura, os endereços de retorno carregados por *CHPR* estão indicados em negrito, e os valores booleanos por *t* (*true*) e *f* (*false*). Deve-se lembrar que as instruções da MEPA começam no endereço zero. ●

#### Exemplo 8.7

A Figura 8.9 mostra um outro programa com dois procedimentos recursivos sem parâmetros. O diagrama de execução correspondente está apresentado na Figura 8.10, supondo que o valor lido foi 4. Uma vez que os procedimentos *p* e *q* não estão encaixados um dentro do outro, não pode haver acesso simultâneo às variáveis locais dos dois; este fato é indicado também pelas flechas estáticas da Figura 8.10. Conseqüentemente, as mesmas posições fixas de memória podem ser atribuídas às variáveis dos dois procedimentos. Como resultado, as variáveis receberão os seguintes endereços:

x: 0	s: 2
y: 1	t: 3.
z: 2	

```

program exemplo7(input,output);
var x,y: integer;
procedure p;
var z: integer;
begin
z:=x; x:=x-1;
if z>1 then p① else y:=1;
y:=y*z;
end (* p *);
procedure q;
var s,t: integer;
begin
s:=x; t:=x-1; x:=t;
if s=0 then y:=1
else
if ((s div 2)*2)=s then q②
else p③;
y:=y*s;
end (* q *);
begin
read(x);
q④;
write(y)
end.

```

Fig. 8.9

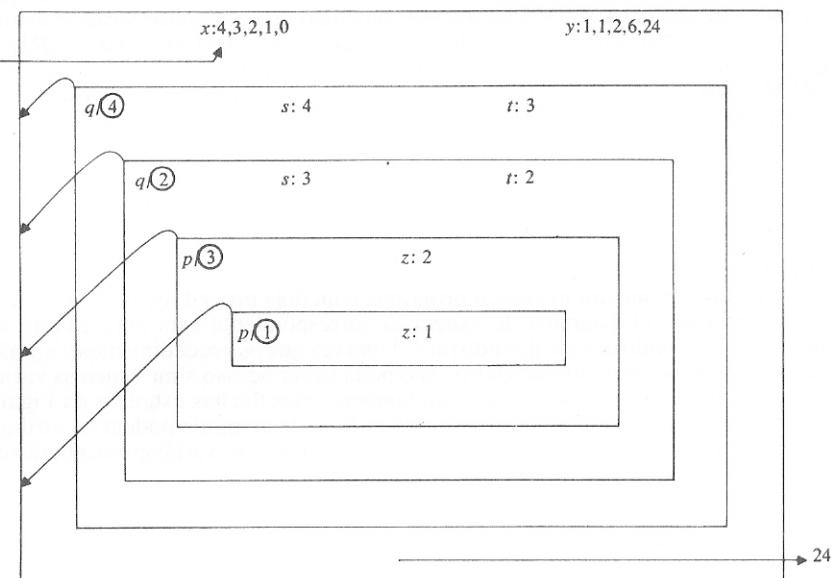


Fig. 8.10

Segue a tradução do programa para MEPA:

<i>INPP</i>	<i>program</i>
<i>AMEM</i> 0,2	<i>var x,y</i>
<i>DSVS</i> L1	
<i>L2 NADA</i>	<i>procedure p</i>
<i>AMEM</i> 2,1	<i>var z</i>
<i>CRVL</i> 0	
<i>ARMZ</i> 2	<i>z:=x</i>
<i>CRVL</i> 0	
<i>CRCT</i> 1	
<i>SUBT</i>	
<i>ARMZ</i> 0	<i>x:=x-1</i>
<i>CRVL</i> 2	
<i>CRCT</i> 1	
<i>CMMA</i>	<i>if z&gt;1</i>
<i>DSVF</i> L3	<i>then</i>
<i>CHPR</i> L2	<i>p</i>
<i>DSVS</i> L4	
<i>L3 NADA</i>	<i>else</i>
<i>CRCT</i> 1	
<i>ARMZ</i> 1	<i>y:=l</i>
<i>L4 NADA</i>	
<i>CRVL</i> 1	
<i>CRVL</i> 2	
<i>MULT</i>	
<i>ARMZ</i> 1	<i>y:=y*z</i>
<i>DMEM</i> 2,1	<i>end</i>
<i>RTPR</i>	
<i>L5 NADA</i>	<i>procedure q</i>
<i>AMEM</i> 2,2	<i>var s,t</i>
<i>CRVL</i> 0	
<i>ARMZ</i> 2	<i>s:=x</i>
<i>CRVL</i> 0	
<i>CRCT</i> 1	
<i>SUBT</i>	
<i>ARMZ</i> 3	<i>t:=x-1</i>
<i>CRVL</i> 3	
<i>ARMZ</i> 0	<i>x:=t</i>
<i>CRVL</i> 2	
<i>CRCT</i> 0	
<i>CMIG</i>	<i>if s=0</i>
<i>DSVF</i> L6	<i>then</i>
<i>CRCT</i> 1	
<i>ARMZ</i> 1	<i>y:=l</i>

<i>DSVS</i>	<i>L7</i>
<i>L6 NADA</i>	
<i>else</i>	
<i>CRVL</i> 2	
<i>CRCT</i> 2	
<i>DIVI</i>	
<i>CRCT</i> 2	
<i>MULT</i>	
<i>CRVL</i> 2	
<i>CMIG</i> <i>if ((s div 2)*2)=s</i>	
<i>DSVF</i> <i>L8</i> <i>then</i>	
<i>CHPR</i> <i>L5</i> <i>q</i>	
<i>DSVS</i> <i>L9</i>	
<i>L8 NADA</i> <i>else</i>	
<i>CHPR</i> <i>L2</i> <i>p</i>	
<i>L9 NADA</i>	
<i>L7 NADA</i>	
<i>CRVL</i> 1	
<i>CRVL</i> 2	
<i>MULT</i>	
<i>ARMZ</i> 1 <i>y:=y*s</i>	
<i>DMEM</i> 2,2 <i>end</i>	
<i>RTPR</i>	
<i>L1 NADA</i>	
<i>LEIT</i>	
<i>ARMZ</i> 0 <i>read(x)</i>	
<i>CHPR</i> <i>L5</i> <i>q</i>	
<i>CRVL</i> 1	
<i>IMPR</i> <i>write(y)</i>	
<i>DMEM</i> 0,2 <i>end.</i>	
<i>PARA</i>	

#### Exemplo 8.8

Apresentamos nas Figuras 8.11 e 8.12 um programa semelhante ao do exemplo anterior, e o seu diagrama de execução. Neste caso, entretanto, o procedimento *q* está encaixado dentro do procedimento *p*, e faz acesso à sua variável local *z*. Consequentemente, as variáveis *s* e *z* não podem ocupar a mesma posição de memória, resultando:

<i>x:</i> 0	<i>z:</i> 2
<i>y:</i> 1	<i>s:</i> 3

Deixamos como exercício a tradução do programa da Figura 8.11 para MEPA. ●

Os exemplos ilustram uma regra geral que pode ser adotada para atribuir endereços a variáveis, por enquanto na ausência de parâmetros: se um procedimento *q*, que tem *k* variáveis locais, está diretamente encaixado dentro de um procedimento *p* cuja última variável local recebeu endereço *m*, então as variáveis de *q* recebem os endereços *m+1, m+2,...,m+k*. Suporemos que o programa

```

program exemplo8(input,output);
  var x,y: integer;
  procedure p;
    var z: integer;
  procedure q;
    var s: integer;
  begin
    s:=z-1; x:=x-1;
    if s>1 then p① else y:=1;
    y:=y*s
  end (* q *);
  begin
    z:=x; x:=x-1;
    if z>1 then q② else y:=1;
    y:=y*z
  end (* p *);
  begin
    read(x);
    p③;
    write(y)
  end.

```

Fig. 8.11

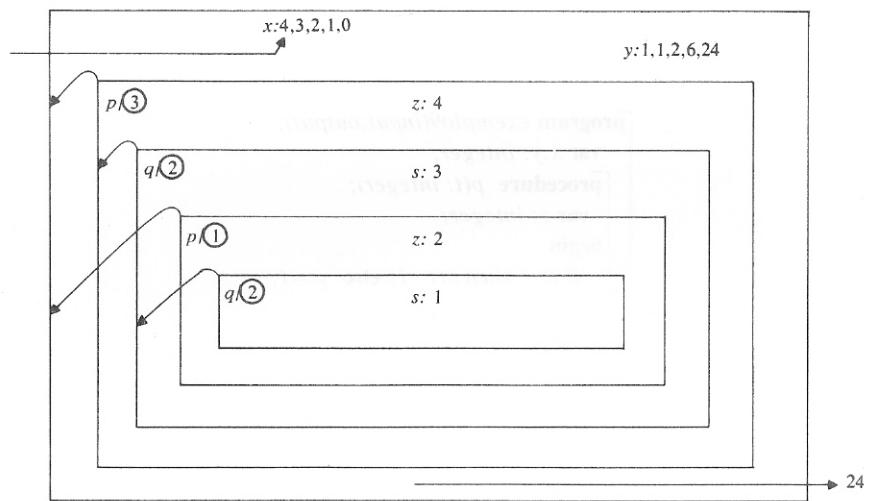


Fig. 8.12

principal é um procedimento encaixado num outro tal que  $m = -1$ ; consequentemente, as  $k$  variáveis do programa principal têm endereços  $0, 1, \dots, k-1$ .

É importante notar que, também no caso de comandos que são chamadas de procedimentos, vale a propriedade enunciada anteriormente, ou seja, de que a execução das instruções do programa-objeto correspondentes mantém o nível da pilha final (após o retorno) igual ao inicial (antes da chamada).

## 8.9 PASSAGEM DE PARÂMETROS POR VALOR

Como próximo passo no desenvolvimento do sistema de execução, consideremos procedimentos com parâmetros passados por valor, como exemplificado na Figura 8.13. De acordo com a definição do Pascal, parâmetros formais passados por valor comportam-se como variáveis locais cujos valores são inicializados com os valores dos parâmetros efetivos. Para que os parâmetros formais se comportem como variáveis locais, é conveniente que o compilador lhes atribua endereços fixos de pilha. Por outro lado, os parâmetros efetivos devem ser avaliados no lugar da chamada, e passados ao procedimento. Os parâmetros efetivos são expressões, e é natural que os seus valores sejam avaliados e deixados no topo da pilha. Dentro do procedimento, estes valores devem ser usados para inicializar as posições fixas de memória destinadas aos parâmetros formais. Como os valores prévios dessas posições fixas terão que ser restaurados após o término da execução do procedimento, podemos simplesmente intercambiar os valores prévios com os efetivos no início da execução e fazer a restauração no fim.

A Figura 8.14 indica uma situação típica quando é executada uma chamada de procedimento da forma  $p(E_1, \dots, E_n)$ ;  $x_1, \dots, x_n$  são os parâmetros formais do procedimento,  $v_1, \dots, v_n$  são os valores dos parâmetros efetivos  $E_1, \dots, E_n$ , e  $a_1, \dots, a_n$  são os valores prévios das posições correspondentes a  $x_1, \dots, x_n$ . O endereço de retorno é indicado por  $r$ . A configuração (a) corresponde ao instante anterior à chamada mas após a avaliação dos parâmetros efetivos; a configuração (b) indica o estado da pilha após a inicialização dos parâmetros formais.

```
program exemplo9(input,output);
  var x,y: integer;
  procedure p(t: integer);
    var z: integer;
    begin
      if t>1 then p(t-1) else y:=1;
      z:=y;
      y:=z*t;
    end (* p *);
  begin
    read(x);
    p(x);
    write(x,y)
  end.
```

Fig. 8.13

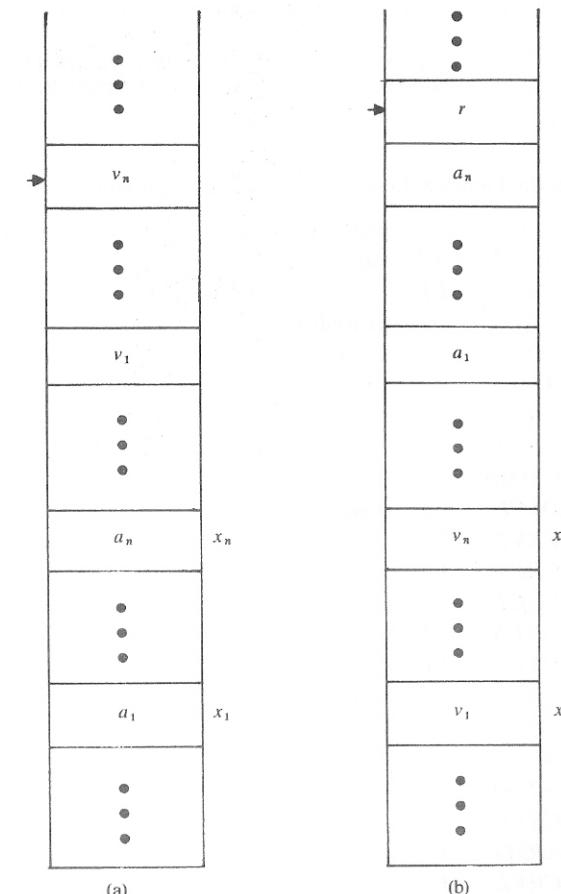


Fig. 8.14

A fim de implementar este mecanismo, definiremos mais uma instrução para a MEPA:

**IPVL m,n** (Inicializar parâmetros por valor):  
 para  $k := 0$  até  $n-1$  faça  
 {temp :=  $M[m+k]$ ;  
 $M[m+k] := M[s-n+k]$ ;  
 $M[s-n+k] := temp$ }

A restauração dos valores salvos pela instrução **IPVL** poderá ser feita pela própria instrução **RTPR** convenientemente redefinida:

**RTPR m,n** (Retornar de procedimento):  
 $i := M[s]; s := s-1$ ;  
 para  $k := n-1$  até  $0$  faça  
 { $M[m+k] := M[s]$ ;  
 $s := s-1$ }

Deve-se notar a semelhança entre essas duas instruções e as instruções *AMEM* e *DMEM*.

A atribuição de endereços às variáveis do programa seguirá a mesma regra da seção anterior, mas incluindo-se agora os parâmetros formais entre as variáveis locais.

#### Exemplo 8.9

O programa da Figura 8.13 produz a seguinte tradução:

```

INPP      program
AMEM    0,2  var x,y
DSVS    L1
L2 NADA   procedure p
IPVL    2,1  (t: integer)
AMEM    3,1  var z
CRVL    2
CRCT    1
CMMA    if t>1
DSVF    L3  then
CRVL    2
CRCT    1
SUBT
CHPR    L2  p(t-1)
DSVS    L4
L3 NADA   else
CRCT    1
ARMZ    1  y:=1
L4 NADA
CRVL    1
ARMZ    3  z:=y
CRVL    3
CRVL    2
MULT
ARMZ    1  y:=z*t
DMEM    3,1  end
RTPR    2,1
L1 NADA
LEIT
ARMZ    0  read(x)
CRVL    0
CHPR    L2  p(x)
CRVL    0
IMPR
CRVL    1
IMPR    write(x,y)
DMEM    0,2  end.
PARA

```

Note-se que os endereços atribuídos às variáveis *x*, *y*, *t* e *z* são 0, 1, 2 e 3, respectivamente.

#### 8.10 PASSAGEM DE PARÂMETROS POR REFERÊNCIA

Consideremos agora procedimentos que têm parâmetros passados por referência (isto é, por variável). A Figura 8.15 mostra um exemplo de tal procedimento, e a Figura 8.16 indica o diagrama de execução correspondente. Uma vez que o procedimento *p* é recursivo, podemos ter várias encarnações da variável local *z*. Entretanto, durante a execução do corpo do procedimento *p*, os comandos podem fazer acesso a duas encarnações: a corrente e a anterior (através do parâmetro formal *s*). Desta maneira, a implementação sugerida na seção anterior não pode mais ser usada. Este exemplo ilustra o problema geral que surge ao introduzirem-se parâmetros passados por referência.

Em princípio poderíamos tentar estender a solução da seção anterior de modo a aplicar-se, também, ao caso de parâmetros passados por referência. O sistema de execução teria que manter a informação sobre variáveis cujo valor foi guardado na pilha, e as suas localizações correntes. A complicação e as ineficiências de um tal esquema seriam proibitivos na prática. Adotaremos, então, uma outra solução, eliminando a causa original do problema, isto é, a atribuição de endereços fixos às variáveis do programa. O endereçamento fixo será substituído pelo endereçamento relativo dentro de cada procedimento (incluindo o programa principal). O compilador associará com cada procedimento um registrador de base, e às variáveis locais de cada procedimento será atribuído um deslocamento fixo, relativo ao conteúdo desse registrador. O endereço atribuído a uma variável será, então, um par constituído do número do registrador de base e do deslocamento, determinados pelo compilador e fixos. Este tipo de endereço é chamado de *endereço léxico* ou *endereço textual*. Os conteúdos dos registradores de base variarão durante a execução do programa, garantindo acesso correto às variáveis.

A fim de estudar melhor o problema geral, consideremos um esboço de programa como apresentado na Fig. 8.17. O seu diagrama de execução poderia ter a

```

program exemplo(input,output);
var x,y: integer;
procedure p(var s: integer);
var z: integer;
begin
  if s=1 then y:=1
  else begin
    z:=s-1;
    p(z)①;
    y:=y*s
  end
end (* p *);
begin
  x:=4;
  p(x)②;
  write(x,y)
end.

```

Fig. 8.15

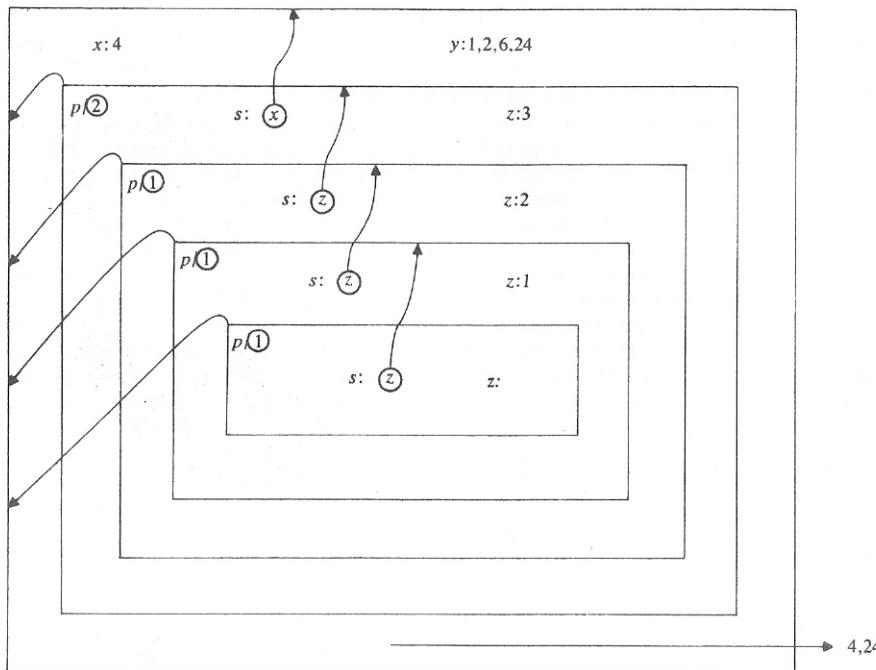


Fig. 8.16

forma indicada na Figura 8.18. A fim de simplificar a discussão, suporemos, provisoriamente, que os procedimentos da Figura 8.17 não têm parâmetros; eles serão reintroduzidos mais adiante. Denotemos por  $d_p$ ,  $d_q$  e  $d_r$  os registradores de base atribuídos pelo compilador aos procedimentos  $p$ ,  $q$  e  $r$ , respectivamente. Em princípio, seria possível atribuir um registrador diferente a cada procedimento. Por outro lado, deve-se minimizar o número de registradores de base utilizados. A razão para isto é que, num computador real, os registradores de base deverão corresponder aos registradores rápidos, tornando eficiente o acesso às variáveis. O número destes registradores rápidos é, em geral, reduzido, sendo tipicamente da ordem de 8, 16 ou 32. Observando a Figura 8.18, podemos notar que os registradores  $d_p$  e  $d_q$  têm que ser distintos, pois dentro do registro de ativação de  $q$  podemos ter acesso às variáveis locais de  $p$ . Por outro lado, dentro de um registro de ativação de  $p$ , e consequentemente de  $q$ , não podemos ter acesso às variáveis de  $r$ , como implicam as flechas estáticas da figura. Obviamente, a diferença está no fato de o procedimento  $q$  ter sido declarado dentro do procedimento  $p$ . Em termos de escopos, podemos afirmar que os escopos das variáveis de  $p$  e de  $r$  são disjuntos, enquanto que o escopo das variáveis de  $p$  inclui o procedimento  $q$ . Estas considerações indicam a possibilidade de se associar com os procedimentos  $p$  e  $r$  o mesmo registrador, por exemplo  $d_1$ , e com o procedimento  $q$ , o registrador  $d_2$ . Não é difícil ver que, em geral, esta numeração de registradores de base para os procedimentos pode ser baseada no encaixamento estático dos procedimentos no programa-fonte, seguindo uma disciplina de pilha. A fim de tornar uniforme o esquema de endereçamento, atribuiremos ao programa principal um registrador de base  $d_0$ .

Podemos adotar, então, as seguintes regras para atribuir números de registradores, também chamados de *níveis léxicos* ou *níveis textuais*, aos procedimentos:

1. O programa é um procedimento de nível 0.
2. Se um procedimento  $p$  tem nível  $m$ , todos os procedimentos encaixados diretamente em  $p$  têm nível  $m+1$ .

O compilador pode determinar facilmente o nível léxico de cada procedimento, e o número de registradores de base necessários corresponderá ao nível de encaixamento máximo existente no programa, e que normalmente é reduzido.

A região  $D$  da MEPA (veja Seção 8.2) será utilizada para formar o vetor de registradores de base,<sup>1</sup> com  $D[k]$  correspondendo a  $d_k$ . Durante a execução de um procedimento, o registrador de base correspondente apontará para a região da pilha  $M$  onde foram alocadas as variáveis locais do procedimento, formando o seu registro de ativação. Como veremos adiante, não será mais necessário guardar e restaurar valores de variáveis. A alocação e a desalocação dos registros segue obviamente uma disciplina de pilha, tornando muito natural a sua implementação. A Figura 8.19 mostra as configurações da pilha e dos registradores de base correspondentes aos instantes marcados com os símbolos \*, #, \$ e & na Figura 8.18. Por convenção, um registrador de base aponta para a primeira variável local do registro de ativação correspondente. Consequentemente, o deslocamento atribuído à primeira variável deve ser zero. Note-se que, se num dado instante está sendo

```

program exemplo(input,output);
var a: integer;
procedure p;
var b: integer;
procedure q;
var c: integer;
begin ... p(1); ... q(2); ... end (* q *);
begin ... q(3); ... end (* p *);
procedure r;
var d: integer;
begin ... p(4); ... end (* r *);
begin
:
p(5);
:
r(6);
:
end.

```

Fig. 8.17

<sup>1</sup>Em inglês, o vetor de registradores de base é chamado de *display*.

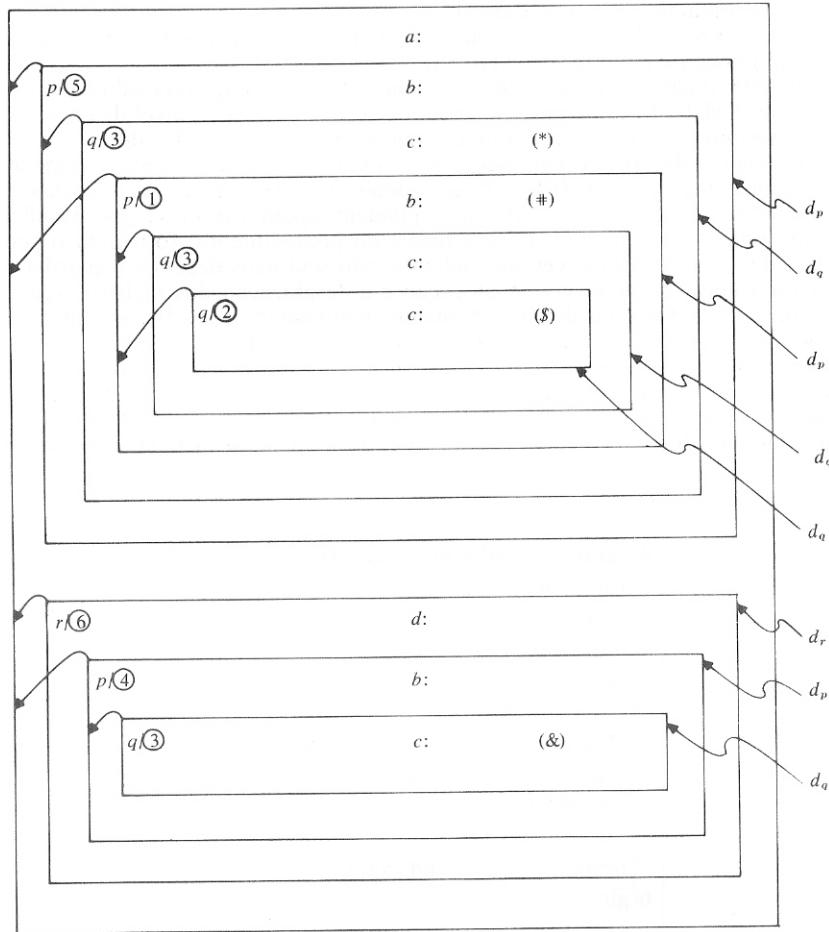


Fig. 8.18

executado um procedimento  $p$  de nível  $m$ , então o programa só pode ter acesso a variáveis locais de  $p$  e de procedimentos em que  $p$  está encaixado, cujos níveis são mais baixos. Na Figura 8.19, o nível corrente está indicado pelas flechas. Na configuração marcada com  $\#$ , o registrador  $D[2]$  aponta para um registro de ativação temporariamente inacessível do procedimento  $q$ .

Várias instruções da MEPA terão que ser redefinidas para implementar o esquema discutido e, em particular, todas as instruções que se referem a variáveis (devemos lembrar que temporariamente excluímos parâmetros dessa discussão):

*CRVL m,n* (Carregar valor):

$s := s + 1; M[s] := M[D[m] + n]$

*ARMZ m,n* (Armazenar valor):

$M[D[m] + n] := M[s]; s := s - 1$

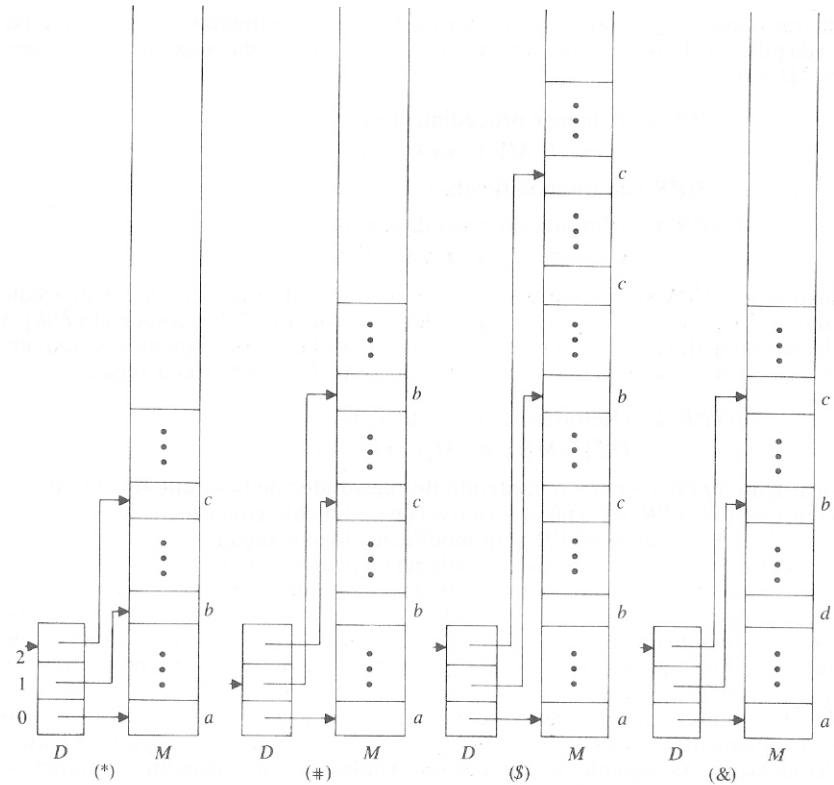


Fig. 8.19

A alocação de memória para variáveis torna-se muito simples, bastando reservar o número conveniente de posições no topo da pilha (é claro que os registradores deverão ter valores apropriados, como veremos a seguir). A desalocação de memória tem o efeito contrário:

*AMEM n* (Alocar memória):

$s := s + n$

*DMEM n* (Desalocar memória):

$s := s - n$

Examinemos agora o problema da chamada de procedimentos. Observando as Figuras 8.18 e 8.19, notamos, por exemplo, que durante a execução do procedimento  $r$ ,  $D[1]$  aponta para o seu registro de ativação. Entretanto, o conteúdo de  $D[1]$  deve ser modificado quando for chamado o procedimento  $p$  (chamada ④), pois  $p$  e  $r$  compartilham o mesmo registrador de base. Ao voltar do procedimento  $p$ , o conteúdo anterior de  $D[1]$  deverá ser restaurado para que as referências a variáveis locais de  $r$  funcionem corretamente. Este problema é semelhante ao que encontramos ao implementar variáveis locais com endereços fixos na Seção 8.8, e a sua solução também será semelhante. Cada procedimento se encarregará de salvar e

restaurar o conteúdo do seu registrador de base. Este conteúdo será guardado no topo da pilha  $M$ . Este esquema será implementado através das seguintes instruções para MEPA:

*CHPR p* (Chamar procedimento):

$s := s + 1; M[s] := i + 1; i := p$

(A instrução *CHPR* não foi modificada.)

*ENPR k* (Entrada no procedimento):

$s := s + 1; M[s] := D[k]; D[k] := s + 1;$

A instrução *ENPR* será sempre a primeira instrução da tradução de um procedimento;  $k$  é o seu nível léxico. Note-se que a instrução *ENPR* coloca em  $D[k]$  o endereço onde deverá estar a primeira variável local do procedimento, se houver; este espaço seria reservado por uma instrução *AMEM* que viria a seguir.

*RTPR k* (Retornar de procedimento):

$D[k] := M[s]; i := M[s - 1]; s := s - 2$

A instrução *RTPR* restaura o conteúdo do registrador de base que foi empilhado pela instrução *ENPR*;  $k$  é o nível léxico do procedimento cuja execução está sendo encerrada. (A instrução *RTPR* será modificada logo a seguir.)

Retornaremos agora ao nosso problema original, supondo que os procedimentos têm parâmetros passados ou por valor ou por referência. Como já vimos, o lugar natural para guardar os parâmetros efetivos é no topo da pilha. No caso dos parâmetros passados por valor, é suficiente, então, avaliar as expressões correspondentes. Os parâmetros passados por referência podem ser implementados através do endereçamento indireto, sendo suficiente então empilhar o endereço da variável passada como parâmetro efetivo. Note-se que não poderíamos ter feito isto com a implementação da seção anterior, pois o valor de uma variável poderia mudar de localização. As seguintes instruções serão utilizadas para manipular parâmetros passados por referência:

*CREN m,n* (Carregar endereço):

$s := s + 1; M[s] := D[m] + n$

*CRVI m,n* (Carregar valor indiretamente):

$s := s + 1; M[s] := M[D[m] + n]$

*ARMI m,n* (Armazenar indiretamente):

$M[D[m] + n] := M[s]; s := s - 1$

A instrução *CREN* será utilizada para empilhar o valor de um parâmetro efetivo passado por referência, ou seja, no nosso caso, o endereço de uma variável simples. Note-se que este endereço depende agora do conteúdo de um registrador de base. As instruções *CRVI* e *ARMI* serão usadas dentro do corpo do procedimento para fazer acessos aos parâmetros formais passados por referência. Parâmetros passados por valor serão tratados como variáveis locais.

Consideremos agora uma chamada de procedimento da forma  $p(E_1, \dots, E_n)$ , e sejam  $v_1, \dots, v_n$  os valores dos parâmetros efetivos  $E_1, \dots, E_n$  calculados ao tempo da chamada; os parâmetros podem ser passados por valor ou por referência. A Figura 8.20 indica, esquematicamente, a configuração da pilha e dos registradores de base após a avaliação dos parâmetros  $E_1, \dots, E_n$ , e após a execução das instruções *CHPR* e *ENPR*. Supusemos que o nível léxico de  $p$  é  $k$ ,  $r$  é o endereço de retorno armazenado por *CHPR*, e  $d$  é o valor prévio de  $D[k]$  salvo por *ENPR*. Observando a figura, fica aparente que dentro do corpo do procedimento  $p$  podemos ter acesso aos seus parâmetros formais usando o mesmo registrador de base  $D[k]$  e desloca-

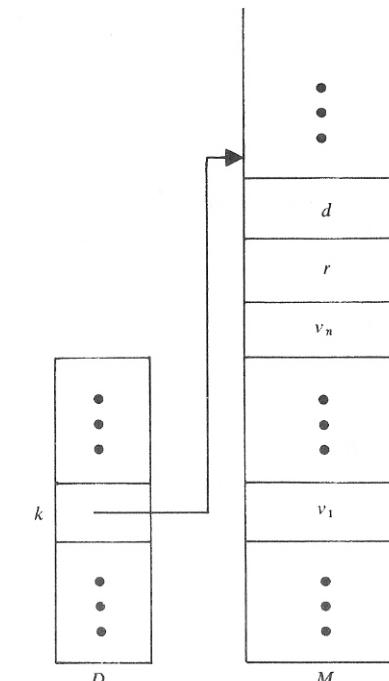


Fig. 8.20

mentos negativos, que podem ser calculados pelo compilador. Se o procedimento tem  $n$  parâmetros, então o  $i$ -ésimo parâmetro terá deslocamento  $-(n+3-i)$ . Teremos que modificar agora a instrução *RTPR* para que ela remova os parâmetros da pilha ao retornar:

*RTPR k,n* (Retornar de procedimento):

$D[k] := M[s]; i := M[s - 1];$

$s := s - (n + 2)$

Uma vez que tratamos o programa inteiro como um procedimento de nível zero, temos que modificar também a instrução *INPP*:

*INPP* (Iniciar programa principal):

$s := -1; D[0] := 0$

### Exemplo 8.10

Consideremos o programa da Figura 8.21. A sua tradução é:

<i>INPP</i>	<b>program</b>
<i>AMEM</i> $I$	<b>var</b> $k$
<i>DSVS</i> $L1$	
<i>L2 ENPR</i> $I$	<b>procedure</b> $p$
<i>AMEM</i> $I$	<b>var</b> $h$
<i>CRVL</i> $I, -4$	

```

CRCT 2
CMME      if n<2
DSVF   L3    then
CRVI  1,-3
CRVL  1,-4
SOMA
ARMI  1,-3  g:=g+n
DSVS   L4
L3 NADA  else
CRVI  1,-3
ARMZ  1,0   h:=g
CRVL  1,-4
CRCT  1
SUBT
CREN  1,0
CHPR  L2    p(n-1,h)
CRVL  1,0
ARMI  1,-3  g:=h
CRVL  1,-4
CRCT  2
SUBT
CRVL  1,-3
CHPR  L2    p(n-2,g)
L4 NADA
CRVL  1,-4
IMPR
CRVI  1,-3
IMPR      write(n,g)
DMEM  1   end
RTPR  1,2
L1 NADA
CRCT  0
ARMZ  0,0   k:=0
CRCT  3
CREN  0,0
CHPR  L2    p(3,k)
DMEM  1   end.
PARA

```

Note-se que os endereços textuais atribuídos às variáveis e aos parâmetros formais são:

k:	0,0
n:	1,-4
g:	1,-3
h:	1,0

```

program exemplo10(input,output);
var k: integer;
procedure p(n: integer; var g: integer);
var h: integer;
begin
  if n<2 then g:=g+n
  else begin
    h:=g;
    p(n-1,h);
    g:=h;
    p(n-2,g)
  end;
  write(n,g)
end (* p *);
begin
  k:=0; p(3,k)
end.

```

Fig. 8.21

Deve-se notar, também, a maneira como é traduzida a chamada  $p(n-2,g)$  no corpo de  $p$ . Para empilhar o parâmetro efetivo  $g$  a ser passado por referência, é usada a instrução  $CRVL$ , pois  $g$  por sua vez já é um parâmetro formal passado por referência, ou seja, o seu valor é um endereço.

Na Figura 8.22 está indicada a simulação da execução do programa-objeto. O símbolo  $\Delta$  indica aqui o conteúdo irrelevante de uma posição da pilha reservada pela instrução  $AMEM$ . Os endereços da pilha estão impressos em itálico para maior clareza, e as outras convenções seguem o Exemplo 8.6.

Deve-se notar, agora, que a propriedade dos comandos, que foi mencionada nas seções anteriores, deve ser modificada para incluir a existência dos registradores de base: os valores dos registradores de base e do registrador  $s$  são os mesmos antes e depois da execução das instruções que correspondem à tradução de um comando. A verificação desta propriedade ficará a cargo do leitor (veja Exercício 8.20).

## 8.11 FUNÇÕES

A implementação de funções é muito semelhante à dos procedimentos. A única diferença é que o nome da função corresponde a uma variável local adicional; o valor final dessa variável deverá estar no topo da pilha, após o retorno da função, para ficar coerente com a implementação das expressões da Seção 8.3.

Uma maneira conveniente de conseguir o efeito descrito, sem modificar a MEPA, é fazer com que o programa-objeto reserve uma posição no topo da pilha, antes de avaliar os parâmetros efetivos da chamada da função (usando, por exemplo, a instrução  $AMEM\ 1$ ). Dentro do corpo da função, a posição assim reservada será usada como a variável local correspondente ao nome da função. Uma vez que esta variável precede os parâmetros, o seu deslocamento será também negativo, valendo  $-(n+3)$ , onde  $n$  é o número de parâmetros. (Podemos pensar que esta variável é o zero-ésimo parâmetro.) A Figura 8.23 ilustra a configuração do sistema de execução após a chamada de função da forma  $f(E_1, \dots, E_n)$ , e adotando-se as mesmas hipóteses da Figura 8.20. Note-se que, com esta implementação, o valor devolvido pela função ficará automaticamente no topo da pilha após a execução da instrução de retorno  $RTPR$ .

17	2  1	2  0		
16	1 0 1 1 1	0 1 1 0 1		
15	Δ	Δ		
14	10	10		
13	<u>22 </u>	<u>29 </u>		
12	2  1 10	2  5	2  1	
11	2 10 2  1	1 2  0	2 1 1 2 1 2 1 2	
10	Δ  0	1	Δ	
9	5		5	
8	<u>22 </u>	<u>29 </u>		
7	2  1  5		2p	
6	3 10 3  2		1 3 1	3 2
5	Δ  0	1	1	
4	?			
3	<u>42 </u>			
2	0			
1	0  3			
0	Δ  0		1  2	

1	?	5 10 15 10 15 10 5 10 5 ?
0	0	

Saída: 1, 1, 0, 1, 2, 1, 1, 2, 3, 2

Fig. 8.22

### Exemplo 8.11

Damos a seguir a tradução do programa apresentado na Figura 7.9, no capítulo anterior.

```

INPP      program
AMEM    I      var m
DSVS    LI
L2 ENPR   I      function f
AMEM    2      var p,q
CRVL    I, -4
CRCT    2

```

```

CMME      if n<2
DSVF    L3      then
CRVL    I, -4
ARMZ    I, -5      f:=n
CRCT    0
ARMI    I, -3      k:=0
DSVS    L4
L3 NADA
AMEM    I
CRVL    I, -4
CRCT    I
SUBT
CREN    I,0
CHPR    L2
AMEM    I
CRVL    I, -4
CRCT    2
SUBT
CREN    I,1
CHPR    L2
SOMA
ARMZ    I, -5      f:=f(n-1,p)+f(n-2,q)
CRVL    I,0
CRVL    I,1
SOMA
CRCT    I
SOMA
ARMI    I, -3      k:=p+q+l
L4 NADA
CRVL    I, -4
IMPR
CRVI    I, -3
IMPR
DMEM    2      end
RTPR    I,2
L1 NADA
AMEM    I
CRCT    3
CREN    0,0
CHPR    L2
IMPR
CRVL    0,0
IMPR
DMEM    1      write(f(3,m),m)
PARA

```

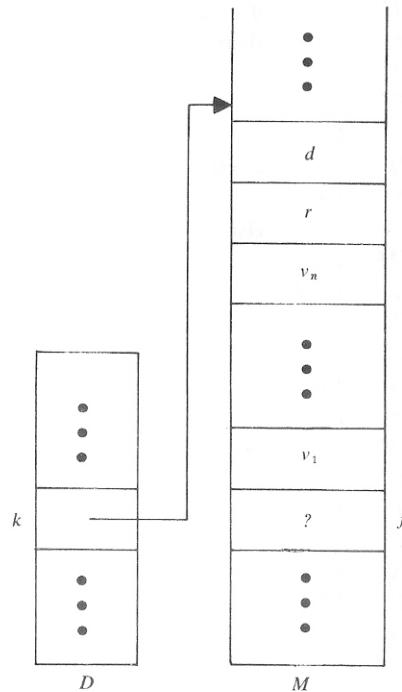


Fig. 8.23

## 8.12 RÓTULOS E COMANDOS DE DESVIO

Consideremos um programa como o esboçado na Figura 8.24. A Figura 8.25 ilustra um possível diagrama de execução para esse programa, em que supusemos que o comando `goto 100` foi executado no instante marcado como símbolo  $\&$ . A Figura 8.26 mostra as configurações da MEPA correspondentes aos instantes marcados com  $*$ ,  $\#$  e  $\&$ . Deveria ser óbvio, pela Figura 8.25, que a configuração no instante  $\$$  deve ser a mesma que no instante  $\#$ , e no instante  $\%$  a mesma que em  $*$ . (Note-se que quando dizemos que duas configurações são iguais, isto implica apenas a igualdade de todos os registradores de base e da altura da pilha  $M$ ; os valores das variáveis obviamente podem diferir.)

Na nossa implementação, ao ser executado o retorno de  $p$  (de  $\$$  para  $\%$ ), apenas o valor de  $D[1]$  seria restaurado para apontar ao registro de ativação de  $r$  (isto é, variável  $d$ ). O valor de  $D[2]$ , que deveria apontar agora para o registro de  $s$  (isto é, variável  $e$ ), teria sido restaurado normalmente ao serem executados retornos do procedimento  $q$ . Conseqüentemente, o comando `goto` deveria ser implementado de maneira a executar as mesmas ações que teriam sido executadas pelas sucessivas instruções de retorno, se não houvesse desvio. Essas ações incluem a restauração de possivelmente vários valores dos registradores de base e o ajuste da altura da pilha, ou seja, do registrador  $s$ , para o valor correto. Para restaurar os conteúdos dos registradores de base, precisamos da informação sobre quais são os registradores a serem restaurados. Num retorno normal, o parâmetro  $k$  da instrução `RTPR k,n` dá essa informação. No caso de um desvio, o número de registradores de

```
program exemplo(input,output);
var a: integer;
procedure p;
label 100;
var b: integer;
procedure q;
var c: integer;
begin ... q①; ... goto 100; ... end (* q *);
begin ... q②; ... 100; ... end (* p *);
procedure r;
var d: integer;
procedure s;
var e: integer;
begin ... p③; ... end (* s *);
begin ... s④; ... end (* r *);
begin
...
r⑤;
...
end.
```

Fig. 8.24

base a serem restaurados é variável. A outra informação necessária são os conteúdos a serem restaurados, mas estes já estão disponíveis na pilha, fazendo parte de cada registro de ativação. A nossa solução consistiu, então, em incluir em cada registro de ativação mais um valor, que é o nível léxico do procedimento que chamou o procedimento dado. A Figura 8.27 ilustra esquematicamente o registro de ativação de um procedimento cujo nível léxico é  $k$  e que foi chamado de um outro procedimento de nível  $m$ ; denotamos por  $r$  o endereço de retorno, por  $d$  o valor prévio de  $D[k]$  salvo na pilha, e por  $v_1, \dots, v_n$  os valores dos parâmetros efetivos.

Redefiniremos, portanto, algumas instruções da MEPA, levando em consideração a modificação sugerida. A instrução `CHPR` deverá empilhar o nível léxico  $m$  do procedimento do qual originou-se a chamada:

`CHPR p,m` (Chamar procedimento):

$M[s+1]:=i+1; M[s+2]:=m;$   
 $s:=s+2; i:=p$

A instrução de retorno deve levar em consideração a existência de um campo adicional num registro de ativação, se bem que ela não o utilizará:

`RTPR k,n` (Retornar de procedimento):

$D[k]:=M[s]; i:=M[s-2];$   
 $s:=s-(n+3)$

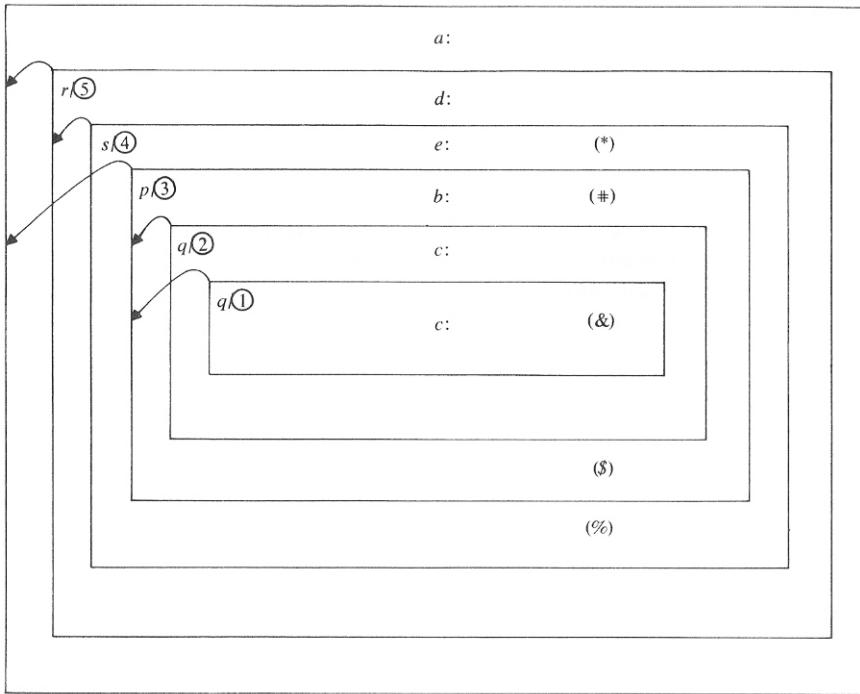


Fig. 8.25

A nova instrução *DSVR* utilizará as informações empilhadas para restaurar os registradores de base e desviar, e será sempre usada como a tradução do comando **goto**:

*DSVR p,j,k* (Desviar para rótulo):  
 $temp1 := k;$   
**enquanto**  $temp1 \neq j$  **faz**  
{ $temp2 := M[D[temp1]-2];$   
 $D[temp1] := M[D[temp1]-1];$   
 $temp1 := temp2$ };  
*i:=p*

Nesta instrução,  $p$  é o endereço para o qual deve ser executado o desvio,  $j$  é o nível léxico do procedimento dentro do qual está o comando rotulado, e  $k$  é o nível do procedimento dentro do qual ocorre o comando **goto**. Em cada repetição, a instrução restaura o registrador de base que seria restaurado pela instrução *RTPR* num retorno normal.

Deve-se notar que a instrução *DSVR p, j, k* funciona corretamente quando  $j=k$ , isto é, quando há um desvio local dentro do mesmo procedimento; seu efeito neste caso é equivalente à instrução *DSVS p*.

A instrução *DSVR* restaura corretamente os registradores de base, faltando entretanto o ajuste da altura da pilha  $M$ . Suponhamos que a instrução *DSVR* já executou o desvio para uma instrução que corresponde a um comando rotulado de um procedimento de nível  $j$ , ou seja,  $D[j]$  aponta para o registro de ativação do

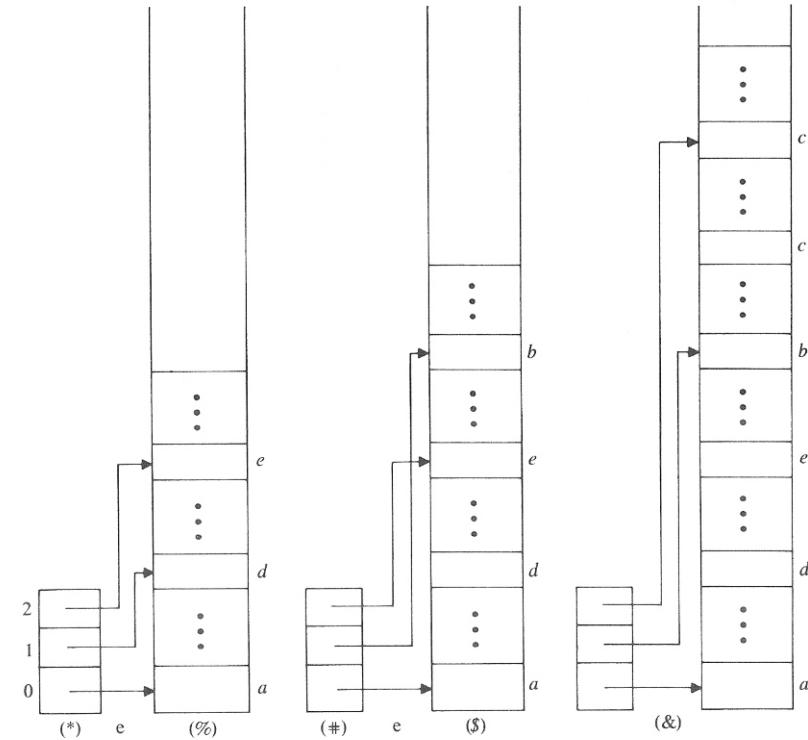


Fig. 8.26

procedimento corrente. Em consequência da propriedade de comandos enunciada no fim da Seção 8.10, podemos concluir que o nível da pilha deve ser o mesmo que existiria antes de executar as instruções correspondentes a qualquer comando do procedimento corrente de nível  $j$ , ou seja, o mesmo nível da última variável local desse procedimento. Se o procedimento tem  $n$  variáveis locais, o valor do registrador  $s$  deve ser, então,  $D[j]+n-1$ . Definiremos, portanto, uma nova instrução para MEPA que realizará este ajuste:

*ENRT j,n* (Entrar no rótulo):  
 $s := D[j]+n-1$

Esta instrução será usada para traduzir um rótulo que ocorre num procedimento de nível  $j$  e que tenha  $n$  variáveis locais. É para esta instrução que deverá desviar a instrução *DSVR* correspondente.

Observamos finalmente que, devido à inclusão de mais um campo no registro de ativação de um procedimento, a fórmula para calcular o deslocamento do  $i$ -ésimo parâmetro formal torna-se  $-(n+4-i)$ , como é fácil ver pela Figura 8.27. Analogamente, no caso de uma função, a variável local que corresponde ao seu nome tem o deslocamento  $-(n+4)$ .

#### Exemplo 8.12

Consideremos o programa da Figura 8.28, cuja tradução segue abaixo. Note-se que um compilador “inteligente” poderia ter eliminado as instruções marcadas

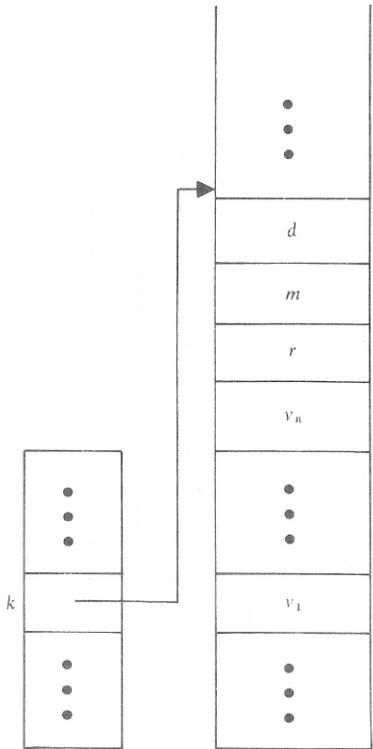


Fig. 8.27

com o símbolo \* e usado a instrução *DSVS L3* no lugar de *DSVR L3,1,1*, e *L3 NADA* no lugar de *L3 ENRT 1,1*. Observe-se, também, que atribuímos rótulos simbólicos *L3* e *L4* aos rótulos do programa-fonte *100* e *200*, respectivamente, supondo que o programa foi traduzido da esquerda para a direita, e ao encontrar a declaração **label** *100, 200*. Os endereços atribuídos às variáveis e parâmetros formais são:

*x*: 0,0

*t*: 1,-4

*s*: 1,0

*z*: 2,-4

*y*: 2,0

O programa-objeto obtido é:

<i>INPP</i>	program
<i>AMEM I</i>	<i>var x</i>
<i>DSVS LI</i>	
<i>L2 ENPR I</i>	procedure <i>p</i>
<i>AMEM I</i>	<i>var s</i>

<i>DSVS</i>	<i>L5</i>	
<i>L6 ENPR</i>	2	function <i>f</i>
<i>CRVL</i>	2,-4	
<i>CRCT</i>	0	
<i>CMME</i>		if <i>z</i> <0
<i>DSVF</i>	<i>L7</i>	then
<i>DSVR</i>	<i>L4,1,2</i>	goto 200
<i>DSVS</i>	<i>L8</i>	
<i>L7 NADA</i>		else
<i>CRVL</i>	2,-4	
<i>CRCT</i>	0	
<i>CMIG</i>		if <i>z</i> =0
<i>DSVF</i>	<i>L9</i>	then
<i>CRCT</i>	2	
<i>ARMZ</i>	2,-5	<i>f</i> :=2
<i>DSVS</i>	<i>L10</i>	
<i>L9 NADA</i>		else
<i>AMEM</i>	1	
<i>CRVL</i>	2,-4	
<i>CRCT</i>	2	
<i>SUBT</i>		
<i>CHPR</i>	<i>L6,2</i>	
<i>CRVL</i>	2,-4	
<i>MULT</i>		
<i>CRCT</i>	1	
<i>SOMA</i>		
<i>ARMZ</i>	2,-5	<i>f</i> := <i>f</i> ( <i>z</i> -2)* <i>z</i> +1
<i>L10 NADA</i>		
<i>L8 NADA</i>		
<i>RTPR</i>	2,1	end
<i>L5 NADA</i>		
(*) <i>L3 ENRT</i>	1,1	100:
<i>AMEM</i>	1	
<i>CRVI</i>	1,-4	
<i>CHPR</i>	<i>L6,1</i>	
<i>ARMZ</i>	1,0	<i>s</i> := <i>f</i> ( <i>i</i> )
<i>CRVL</i>	1,0	
<i>ARMI</i>	1,-4	<i>t</i> := <i>s</i>
<i>CRVI</i>	1,-4	
<i>CRVL</i>	0,0	
<i>CMME</i>		if <i>t</i> < <i>x</i>
<i>DSVF</i>	<i>L11</i>	then
(*) <i>DSVR</i>	<i>L3,1,1</i>	goto 100
<i>L11 NADA</i>		
<i>L4 ENRT</i>	1,1	200:

```

CRVL 0,0
CRCT 1
SUBT
ARMZ 0,0      x:=x-1
DMEM 1       end
RTPR 1,1
L12 ENPR 1       procedure r
DSVS L13
L14 ENPR 2       procedure q
AMEM 1       var y
LEIT
ARMZ 2,0      read(y)
CREN 2,0
CHPR L2,2      p(y)
CRVL 2,0
IMPR          write(y)
DMEM 1       end
RTPR 2,0
L13 NADA
CHPR L14,1      q
RTPR 1,0       end
L1 NADA
begin
LEIT
ARMZ 0,0      read(x)
CHPR L12,0      r
DMEM 1       end.
PARA

```

```

program exemplo12(input,output);
var x: integer;
procedure p(var t: integer);
label 100,200;
var s: integer;
function f(z: integer): integer;
begin
  if z < 0 then goto 200
  else
    if z = 0 then f := 2
    else f := f(z-2)*z+1
end (* f *);
begin
  100: s := f(t); t := s;
  if t < x then goto 100;
  200: x := x - 1
end (* p *);

procedure r;
procedure q;
var y: integer;
begin
  read(y); p(y); write(y)
end (* q *);
begin q end (* r *);

begin
  read(x);
  r
end.

```

Fig. 8.28

### Exercícios

- Suponha que incluímos na nossa versão do Pascal os comandos de seleção e iterativo, incluindo para tal as seguintes produções na gramática do Capítulo 5:

```

<comando de seleção> ::=
  case <expressão> of
    <caso> {; <caso>}
  end
<caso> ::=
  (<número> | <identificador>): <comando sem rótulo>
<comando iterativo>::=
  for <identificador>:= <expressão> to <expressão>
    do <comando sem rótulo>

```

Além disso, os não-terminais <comando de seleção> e <comando iterativo> seriam incluídos como alternativas do lado direito da produção 18 (Seção 5.4).

Indique como traduzir para MEPA comandos da forma:

```
case E of
    A1 : C1;
    :
    An-1 : Cn-1;
    An : Cn
end               (n≥0)
e      for v:=E1 to E2 do C
```

onde  $E$ ,  $E_1$  e  $E_2$  são expressões,  $A_1, \dots, A_n$  são constantes (números ou identificadores *true* ou *false*),  $C, C_1, \dots, C_n$  são comandos, e  $v$  é um identificador.

2. Simule a execução do programa objeto em MEPA do Exemplo 8.7, supondo que o valor lido é 4. Use a versão da MEPA definida até então.
3. Simule a execução do programa-objeto em MEPA do Exemplo 8.9, supondo que o valor lido é 5. Use a versão da MEPA definida até então.
4. Indique como poderiam ser implementadas funções para a versão da MEPA definida na Seção 8.9.
5. Indique como poderiam ser implementados rótulos e comandos de desvio para a versão da MEPA definida na Seção 8.9. (*Sugestão*: Considere a possibilidade de empilhar informações adicionais.)
6. Traduza para a versão da MEPA da Seção 8.9 o programa da Figura 7.7.
7. Traduza para a versão da MEPA da Seção 8.10 o programa da Figura 8.15.
8. Simule a execução do programa-objeto do Exemplo 8.11, usando a versão da MEPA definida até então.
9. Traduza para MEPA o programa da Figura 7.13 e simule a execução do programa-objeto. Use a versão da MEPA da Seção 8.11.
10. Foi mencionada na Seção 8.10 a possibilidade de se associar um registrador de base distinto a cada procedimento e função do programa. Responda e justifique se ainda neste caso o conteúdo dos registradores de base teria que variar durante a execução do programa-objeto.

Os exercícios seguintes referem-se à versão da MEPA definida na Seção 8.12.

11. As ações executadas pelas instruções *CHPR* e *ENPR* poderiam ser combinadas numa única instrução de chamada de procedimento. Defina esta nova instrução. Qual é a vantagem de se utilizarem as duas instruções originais?
12. Aplique o Exercício 8.11 às instruções *DSVR* e *ENTR*.
13. Em geral, a instrução *AMEM* segue uma instrução *ENPR*, e a instrução *DMEM* precede uma instrução *RTPR*. Este fato sugere a possibilidade de se combinar *ENPR* com *AMEM* e *DMEM* com *RTPR*. Defina as novas instruções que resultariam dessa combinação.
14. Que outras instruções poderiam ser utilizadas no lugar de *AMEM 1* que serve para reservar lugar para o resultado de uma função?
15. A instrução *DSVR* contém uma repetição. É possível, em geral, examinando-se o programa-fonte, prever-se um limite para o número de vezes que o corpo da repetição será executado, para uma dada instrução *DSVR*? Justifique.

16. O programa principal poderia deixar de ser tratado como procedimento de nível 0, eliminando-se assim o uso de  $D[0]$  e tornando mais eficiente o acesso às variáveis globais do programa, através de endereços fixos. Redefina a MEPA de maneira a implementar essa sugestão.

17. Algumas instruções da MEPA necessitam de um parâmetro que é o nível léxico do procedimento corrente (*CHPR*, *RTPR*, *DSVR*, *ENRT*). O projeto da MEPA poderia ser modificado, introduzindo-se um novo registrador que sempre contivesse o nível correto e eliminando-se a necessidade desse parâmetro. Redefina as instruções da MEPA para implementar esta idéia.
18. A maioria das implementações do Pascal permite a referência antecipada a procedimentos e funções mediante uma declaração em que o bloco é substituído pelo identificador pré-declarado *forward*, a fim de permitir recursão mútua. Quais seriam as consequências, sobre o projeto do nosso sistema de execução, se esta declaração fosse considerada?
19. Explique por que a instrução *RTPR* sempre encontra a informação correta no topo da pilha.
20. Suponha que um procedimento  $p$  tem o corpo da forma

```
begin C1; ...; Cn end (n≥1)
```

onde  $C_1, \dots, C_n$  são comandos quaisquer, eventualmente precedidos de rótulos. Suponha também que no início de uma ativação de  $p$ , imediatamente antes de começar a executar as instruções resultantes da tradução de  $C_1$ , a MEPA está numa certa configuração. Mostre que antes de começar ou depois de terminar a execução de instruções correspondentes a qualquer comando  $C_i$ , nesta mesma ativação, a MEPA está na mesma configuração inicial (isto é, todos os registradores de base e o registrador  $s$  terão o mesmo valor). Lembre-se que um  $C_i$  pode ser, por exemplo, uma chamada de procedimento que não retorna mas executa desvio para um  $C_j$ , nesta mesma ativação de  $p$ . (*Sugestão*: Proceda por indução sobre a estrutura dos comandos considerados até agora e verifique cada caso.)

21. Em algumas linguagens, a passagem por referência é substituída pela passagem por cópia, também chamada de passagem por *valor-resultado*. Neste caso, o parâmetro formal é uma variável local cujo valor inicial é o valor da variável passada como parâmetro efetivo. Ao fim da execução do procedimento, o valor final do parâmetro formal é atribuído de volta à variável passada como parâmetro.
  - a) Estabeleça as condições em que a passagem por referência é equivalente à passagem por cópia.
  - b) Modifique a versão final deste capítulo da MEPA para substituir a passagem por referência pela passagem por cópia.
  - c) Verifique que, uma vez que houve esta substituição, voltamos a ter a propriedade de que um procedimento só pode ter acesso a variáveis e parâmetros declarados nos escopos em que ele está contido. Consequentemente, pode-se eliminar o endereçamento relativo e estender a versão da MEPA da Seção 8.9. Defina esta nova versão da MEPA. Discuta as vantagens e desvantagens em relação à versão da parte (b) acima.

## NOTAS BIBLIOGRÁFICAS

A idéia de se definir uma máquina hipotética para descrever a implementação de uma linguagem de programação é bastante antiga. O texto de Randell e Russell (1964) é um clássico, e descreve de maneira muito completa e pormenorizada a implementação do Algol 60. As notas de aula de Morris (1970) (utilizadas também

Além disso, os não-terminais <comando de seleção> e <comando iterativo> seriam incluídos como alternativas do lado direito da produção 18 (Seção 5.4).

Indique como traduzir para MEPA comandos da forma:

```
case E of
    A1 : C1;
    :
    An-1 : Cn-1;
    An : Cn
end           (n≥0)
e   for v:=E1 to E2 do C
```

onde  $E$ ,  $E_1$  e  $E_2$  são expressões,  $A_1, \dots, A_n$  são constantes (números ou identificadores *true* ou *false*),  $C, C_1, \dots, C_n$  são comandos, e  $v$  é um identificador.

2. Simule a execução do programa objeto em MEPA do Exemplo 8.7, supondo que o valor lido é 4. Use a versão da MEPA definida até então.
3. Simule a execução do programa-objeto em MEPA do Exemplo 8.9, supondo que o valor lido é 5. Use a versão da MEPA definida até então.
4. Indique como poderiam ser implementadas funções para a versão da MEPA definida na Seção 8.9.
5. Indique como poderiam ser implementados rótulos e comandos de desvio para a versão da MEPA definida na Seção 8.9. (*Sugestão:* Considere a possibilidade de empilhar informações adicionais.)
6. Traduza para a versão da MEPA da Seção 8.9 o programa da Figura 7.7.
7. Traduza para a versão da MEPA da Seção 8.10 o programa da Figura 8.15.
8. Simule a execução do programa-objeto do Exemplo 8.11, usando a versão da MEPA definida até então.
9. Traduza para MEPA o programa da Figura 7.13 e simule a execução do programa-objeto. Use a versão da MEPA da Seção 8.11.
10. Foi mencionada na Seção 8.10 a possibilidade de se associar um registrador de base distinto a cada procedimento e função do programa. Responda e justifique se ainda neste caso o conteúdo dos registradores de base teria que variar durante a execução do programa-objeto.

Os exercícios seguintes referem-se à versão da MEPA definida na Seção 8.12.

11. As ações executadas pelas instruções *CHPR* e *ENPR* poderiam ser combinadas numa única instrução de chamada de procedimento. Defina esta nova instrução. Qual é a vantagem de se utilizarem as duas instruções originais?
12. Aplique o Exercício 8.11 às instruções *DSVR* e *ENTR*.
13. Em geral, a instrução *AMEM* segue uma instrução *ENPR*, e a instrução *DMEM* precede uma instrução *RTPR*. Este fato sugere a possibilidade de se combinar *ENPR* com *AMEM* e *DMEM* com *RTPR*. Defina as novas instruções que resultariam dessa combinação.
14. Que outras instruções poderiam ser utilizadas no lugar de *AMEM 1* que serve para reservar lugar para o resultado de uma função?
15. A instrução *DSVR* contém uma repetição. É possível, em geral, examinando-se o programa-fonte, prever-se um limite para o número de vezes que o corpo da repetição será executado, para uma dada instrução *DSVR*? Justifique.

16. O programa principal poderia deixar de ser tratado como procedimento de nível 0, eliminando-se assim o uso de  $D[0]$  e tornando mais eficiente o acesso às variáveis globais do programa, através de endereços fixos. Redefina a MEPA de maneira a implementar essa sugestão.

17. Algumas instruções da MEPA necessitam de um parâmetro que é o nível léxico do procedimento corrente (*CHPR*, *RTPR*, *DSVR*, *ENRT*). O projeto da MEPA poderia ser modificado, introduzindo-se um novo registrador que sempre contivesse o nível corrente e eliminando-se a necessidade desse parâmetro. Redefina as instruções da MEPA para implementar esta idéia.
18. A maioria das implementações do Pascal permite a referência antecipada a procedimentos e funções mediante uma declaração em que o bloco é substituído pelo identificador pré-declarado *forward*, a fim de permitir recursão mútua. Quais seriam as consequências, sobre o projeto do nosso sistema de execução, se esta declaração fosse considerada?
19. Explique por que a instrução *RTPR* sempre encontra a informação correta no topo da pilha.
20. Suponha que um procedimento  $p$  tem o corpo da forma

```
begin C1; ...; Cn end (n≥1)
```

onde  $C_1, \dots, C_n$  são comandos quaisquer, eventualmente precedidos de rótulos. Suponha também que no início de uma ativação de  $p$ , imediatamente antes de começar a executar as instruções resultantes da tradução de  $C_1$ , a MEPA está numa certa configuração. Mostre que antes de começar ou depois de terminar a execução de instruções correspondentes a qualquer comando  $C_i$ , nesta mesma ativação, a MEPA está na mesma configuração inicial (isto é, todos os registradores de base e o registrador  $s$  terão o mesmo valor). Lembre-se que um  $C_i$  pode ser, por exemplo, uma chamada de procedimento que não retorna mas executa desvio para um  $C_j$ , nesta mesma ativação de  $p$ . (*Sugestão:* Proceda por indução sobre a estrutura dos comandos considerados até agora e verifique cada caso.)

21. Em algumas linguagens, a passagem por referência é substituída pela passagem por *cópia*, também chamada de passagem por *valor-resultado*. Neste caso, o parâmetro formal é uma variável local cujo valor inicial é o valor da variável passada como parâmetro efetivo. Ao fim da execução do procedimento, o valor final do parâmetro formal é atribuído de volta à variável passada como parâmetro.
  - a) Estabeleça as condições em que a passagem por referência é equivalente à passagem por cópia.
  - b) Modifique a versão final deste capítulo da MEPA para substituir a passagem por referência pela passagem por cópia.
  - c) Verifique que, uma vez que houve esta substituição, voltamos a ter a propriedade de que um procedimento só pode ter acesso a variáveis e parâmetros declarados nos escopos em que ele está contido. Consequentemente, pode-se eliminar o endereçamento relativo e estender a versão da MEPA da Seção 8.9. Defina esta nova versão da MEPA. Discuta as vantagens e desvantagens em relação à versão da parte (b) acima.

## NOTAS BIBLIOGRÁFICAS

A idéia de se definir uma máquina hipotética para descrever a implementação de uma linguagem de programação é bastante antiga. O texto de Randell e Russell (1964) é um clássico, e descreve de maneira muito completa e pormenorizada a implementação do Algol 60. As notas de aula de Morris (1970) (utilizadas também

por Barrett e Couch (1979)) influenciaram bastante a nossa abordagem gradual. A utilização dos diagramas de execução para justificar o desenvolvimento do sistema de execução parece ser original. Vários textos sobre compilação e implementação de linguagens tratam, com maior ou menor profundidade, dos problemas tratados neste capítulo, como já citados Barrett e Couch (1979), Pratt (1975), Gries (1971). Como leitura complementar recomendamos Pasko (1973), Bobrow e Wegbreit (1973), Amman (1978) e Sanches (1979). Sistemas de execução específicos para Pascal foram descritos por Nori, Amman, Jensen e Nägeli (1975), Wirth (1976), Berry (1978) e Joy (1979). A solução do Exercício 8.21 pode ser encontrada em Kowaltowski (1981). Existem descrições de sistemas de execução para linguagens de características bem distintas do Pascal; um exemplo interessante é Griswold (1972).

# 9

## Sistema de Execução: Extensões e Implementação

### 9.1 GENERALIDADES

Foi desenvolvido, no Capítulo 8, um sistema de execução correspondente à parte básica do nosso Pascal simplificado descrito no Capítulo 5. O objetivo do presente capítulo é discutir a implementação de alguns mecanismos do Pascal que foram considerados opcionais, tais como procedimentos que são parâmetros, matrizes, bem como alguns mecanismos que não são encontrados em Pascal mas que são de interesse geral. Entre estes últimos incluímos passagem de parâmetros por nome, matrizes dinâmicas e blocos como em Algol 60. A discussão de alguns desses mecanismos será mais superficial, ficando a cargo do leitor a elaboração mais completa. Na seção final deste capítulo serão discutidas as várias maneiras de se implementar um sistema de execução.

### 9.2 PASSAGEM DE PROCEDIMENTOS E FUNÇÕES COMO PARÂMETROS

A definição do Pascal inclui a passagem de procedimentos (e funções) como parâmetros, contanto que os parâmetros destes últimos sejam, por sua vez, todos passados por valor. Consideremos, então, o programa esboçado na Figura 9.1, cujo diagrama de execução poderia ser como o apresentado na Figura 9.2. Suponhamos, provisoriamente, que foram eliminados os comandos de desvio, voltando assim à versão da MEPA definida até a Seção 8.11.

As configurações da MEPA correspondentes aos instantes marcados com os símbolos \*, # e & deveriam ser as indicadas na Figura 9.3. Note-se que, em cada instante, os apontadores contidos nos registradores de base são consistentes com as flechas de encaixamento estático da Figura 9.2. Podemos observar que, ao passar da configuração # para &, isto é, ao ser chamado o procedimento *u* dentro do corpo de *q*, vários registradores de base devem ser modificados a fim de refletir o encaixamento estático de *u* dentro de *t*, de *t* dentro de *s*, e de *s* dentro de *r*. (É apenas neste caso particular que o valor de *D[3]* já está correto; se dentro do corpo de *q* tivesse havido uma chamada de procedimento de nível 3, este valor também teria sido alterado.)

A implementação atual da MEPA não prevê essa situação, pois ela pressupõe que todos os registradores de base necessários para fazer acesso a variáveis não-locais ao procedimento chamado já estão corretos. Este fato era verdadeiro até agora, pois o procedimento chamado era sempre um procedimento cujo nome era conhecido no escopo corrente da chamada; isto não é verdade no caso da chamada de *u* dentro de *q*. A fim de implementar corretamente este novo mecanismo, devemos prever que, na chamada de procedimento passado como parâmetro, todos

```

program exemplo(input,output);
var a: integer;
procedure p (procedure h);
var b: integer;
procedure q;
var c: integer;
begin ... h①; ... end (* q *);
begin ... q②; ... end (* p *);
procedure r;
var d: integer;
procedure s;
var e: integer;
procedure t;
var f: integer;
procedure u;
var g: integer;
begin ... end (* u *);
begin ... p(u)③; ... end (* t *);
begin ... t④; ... end (* s *);
begin ... s⑤; ... end (* r *);
begin
.
.
.
r⑥;
.
.
.
end.

```

Fig. 9.1

os registradores de base poderão necessitar de ajuste. A única exceção é  $D[0]$ , cujo conteúdo não muda durante a execução do programa. Este efeito só pode ser conseguido se o sistema tiver acesso à informação necessária. Uma maneira de obter esta informação seria passar o vetor de registradores de base corrente com cada parâmetro que é um procedimento. A Figura 9.2 sugere, entretanto, um método mais simples, que consistirá em manter uma cadeia de apontadores análogos às flechas estáticas. Para tal, em cada registro de ativação será incluído um campo adicional — *apontador estático* — apontando para o registro de ativação do procedimento em que o procedimento dado está encaixado. Com esta modificação, a Figura 9.3 passará a ter o aspecto indicado na Figura 9.4, onde o apontador estático precede a primeira variável local do procedimento. Para cada registro de ativação temos uma cadeia de apontadores estáticos formando a chamada *cadeia estática*, resultante do encaixamento dos procedimentos no programa-fonte. Podemos verificar, nos instantes marcados, uma propriedade que valerá em qualquer instante de execução: se num dado instante está sendo executado um procedimento de nível  $k$ , então os conteúdos de  $D[k-1], D[k-2], \dots, D[0]$  são iguais aos valores dos apontadores que constituem a cadeia estática corrente.

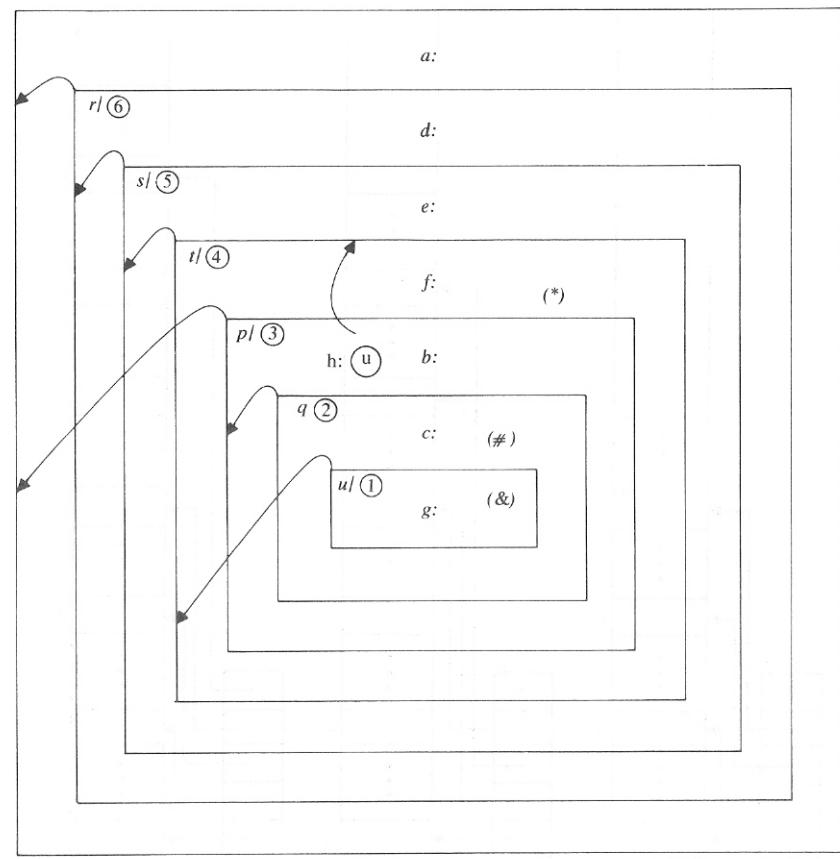


Fig. 9.2

Analisaremos a seguir as várias ações que deverão ser executadas a fim de implementar os mecanismos de chamada e de retorno de procedimentos. Devemos considerar várias situações:

- Chamada de um procedimento que não é parâmetro: neste caso todos os registradores de base já contêm valores convenientes, exceto o que corresponde ao nível  $k$  do procedimento chamado. As ações a serem executadas são, portanto: o empilhamento do endereço de retorno (a ser analisado adiante), a criação de um novo registro de ativação para o procedimento chamado (incluindo a criação do apontador estático correto) e a atribuição do valor conveniente a  $D[k]$ . Note-se que o valor do novo apontador estático é dado por  $D[k-1]$ , devido à propriedade mencionada acima. Como verificaremos mais adiante, não será mais necessário empilhar o conteúdo prévio de  $D[k]$ .
- Chamada de um procedimento passado como parâmetro: após o empilhamento do endereço de retorno, devem ser ajustados, no pior caso, os valores de  $D[k-1], D[k-2], \dots, D[1]$  (supondo que o nível do procedimento chamado é  $k$ ). Uma vez executadas estas ações, deve-se proceder como em (a) acima: criar um novo registro de ativação e fazer com que  $D[k]$  aponte para este

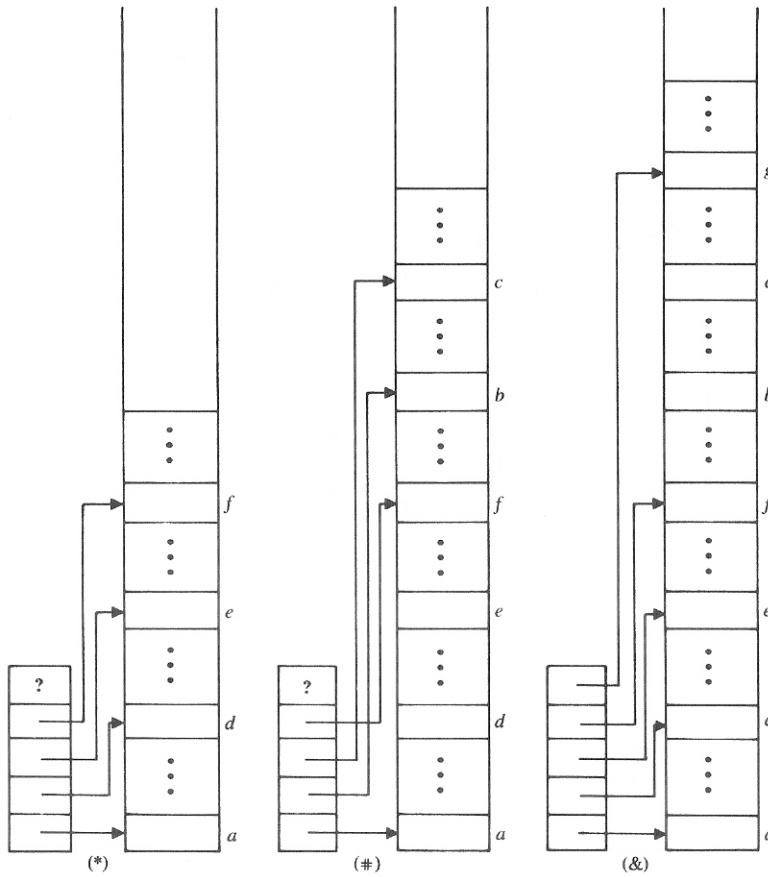


Fig. 9.3

registro. Os valores de  $D[k-1], D[k-2], \dots, D[1]$  podem ser obtidos seguindo-se a cadeia estática apropriada. O início desta cadeia, juntamente com o nível  $k$ , deve ser passado como parte do parâmetro efetivo. Além desses dois valores, será necessário passar o endereço do corpo do procedimento que é passado como parâmetro. Este terno de informação é chamado de *endereço generalizado*, e ocupará três posições da pilha.

- (c) Retorno de um procedimento: uma vez que o mesmo procedimento pode ser chamado diretamente ou então como um parâmetro, o mecanismo de retorno deverá ser uniforme para incluir ambos os casos. Ao retornar do procedimento  $u$  (veja Figura 9.4), a configuração deve mudar de & novamente para #. Consequentemente, será necessária uma nova modificação de vários, eventualmente todos exceto  $D[0]$ , registradores de base. A informação necessária para realizar esta modificação é novamente um endereço generalizado de retorno que incluirá o nível  $k$  do procedimento ao qual se realiza o retorno, o início da cadeia estática, ou seja, o valor de  $D[k]$  a ser restaurado, e o endereço da próxima instrução a ser executada. Conclui-se, portanto, que ambas as instruções que chamam procedimentos devem empilhar um endereço generalizado de retorno.

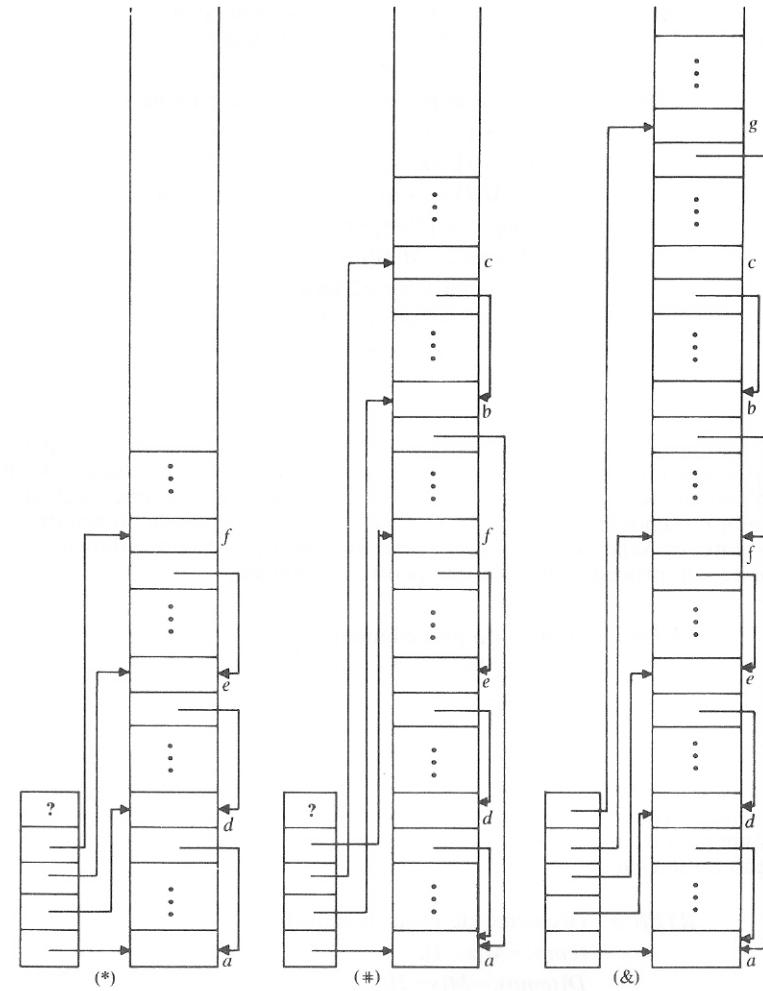


Fig. 9.4

Note-se que um endereço generalizado traz a informação completa sobre a próxima instrução a ser executada, bem como sobre o *contexto de execução* correspondente, ou seja, indica através da cadeia estática todas as variáveis acessíveis naquele ponto.

A discussão acima leva à redefinição de algumas e à introdução de novas instruções para a MEPA:

**CHPR  $p,k$**  (Chamar procedimento):

$M[s+1]:=i+1; M[s+2]:=D[k];$   
 $M[s+3]:=k; s:=s+3; i:=p$

A instrução CHPR será utilizada para chamar procedimentos que não são parâmetros. Ela empilha um endereço generalizado que será utilizado posteriormente pela

instrução de retorno e desvia para a primeira instrução do procedimento cujo endereço é  $p$ ;  $k$  é o nível léxico do procedimento em que ocorreu a chamada.

**CHPP**  $m,n,k$  (Chamar procedimento que é parâmetro):

```

 $M[s+1]:=i+1; M[s+2]:=D[k];$ 
 $M[s+3]:=k; s:=s+3;$ 
 $i:=M[D[m]+n];$ 
 $temp:=M[D[m]+n+2];$ 
 $D[temp]:=M[D[m]+n+1];$ 
enquanto  $temp \geq 2$  faz
     $\{D[temp-1]:=M[D[temp]-1];$ 
     $temp:=temp-1\}$ 
```

A instrução **CHPP** será utilizada para chamar procedimento passado como parâmetro. Após empilhar o endereço generalizado de retorno, a instrução modifica convenientemente os registradores de base e desvia para o procedimento chamado. O par  $m,n$  é o endereço textual desse parâmetro e corresponde ao seu primeiro componente, ou seja, ao endereço da primeira instrução do procedimento;  $k$  é o nível léxico do procedimento em que ocorre a chamada.

**ENPR**  $k$  (Entrar no procedimento):

```

 $s:=s+1; M[s]:=D[k-1];$ 
 $D[k]:=s+1$ 
```

Como antes, a instrução **ENPR** será a primeira instrução de um procedimento. A sua execução cria um novo apontador estático e modifica o valor de  $D[k]$  para que ele aponte para o registro de ativação corrente;  $k$  é o nível do procedimento cuja execução foi iniciada.

**RTPR**  $n$  (Retornar de procedimento):

```

 $temp:=M[s-1];$ 
 $D[temp]:=M[s-2];$ 
 $i:=M[s-3]; s:=s-(n+4);$ 
enquanto  $temp \geq 2$  faz
     $\{D[temp-1]:=M[D[temp]-1];$ 
     $temp:=temp-1\}$ 
```

A instrução **RTPR** usa o endereço generalizado presente no topo da pilha (logo abaixo do apontador estático), para executar o retorno, reconstruindo para tal o vetor de registradores de base;  $n$  é o número de posições ocupadas pelos parâmetros.

O segundo componente do endereço generalizado de retorno, ou seja, o apontador para o registro de ativação do procedimento que executou a chamada, é denominado *apontador dinâmico*. A seqüência desses apontadores forma a chamada *cadeia dinâmica*, que liga todos os registros de ativação a partir do topo até o fundo da pilha. Num diagrama de execução, esta cadeia corresponde ao encaixamento dinâmico dos retângulos que representam registros de ativação.

Uma vez que o registro de ativação do programa principal não necessita do apontador estático, a definição da instrução **INPP** permanece inalterada:

**INPP** (Iniciar programa principal):

```

 $s:=-1; D[0]:=0$ 
```

O empilhamento de parâmetros efetivos que são procedimentos será realizado pela instrução:

**CREG**  $p,k$  (Carregar endereço generalizado):

```

 $M[s+1]:=p; M[s+2]:=D[k];$ 
 $M[s+3]:=k; s:=s+3$ 
```

onde  $k$  é o nível léxico dentro do qual o procedimento que está sendo passado foi declarado;  $p$  é o endereço da primeira instrução desse procedimento. O empilhamento de outros tipos de parâmetros é feito como anteriormente.

Note-se que é necessário agora rever a fórmula usada para calcular o deslocamento atribuído aos parâmetros formais. A Figura 9.5 ilustra a configuração da MEPA após a entrada num procedimento de nível  $k$ , com valores  $v_1, \dots, v_n$  dos parâmetros efetivos;  $r_1, r_2, r_3$  formam o endereço generalizado de retorno, e  $d$  é o apontador estático. Conclui-se que o deslocamento do  $i$ -ésimo parâmetro formal será dado por  $-(n+5-i)$ . Note-se, também, que um parâmetro que é um procedimento é considerado, na realidade, como constituído de três parâmetros.

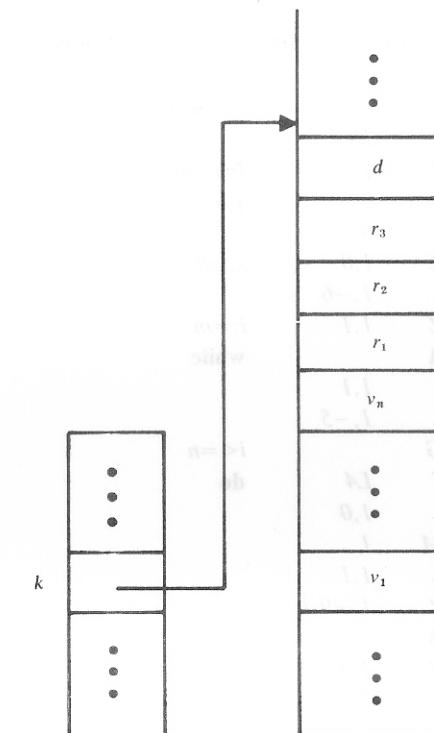


Fig. 9.5

```

program exemplo1(input,output);
function f(function g: integer; m,n: integer): integer;
var s,i: integer;
begin
  s:=0; i:=m;
  while i<=n
    do begin
      s:=s+g(i);
      i:=i+1
    end;
  f:=s
end (*f *);
function h1(k: integer): integer;
begin h1:=(2*k+1)*k end (*h1 *);
function h2(j: integer): integer;
begin h2:=j*j end (*h2 *);
begin
  write(f(h1,1,f(h2,5,10)))
end.

```

Fig. 9.6

### Exemplo 9.1

Consideremos o programa da Figura 9.6. A sua tradução é:

<i>INPP</i>	program
<i>DSVS</i>	<i>L1</i>
<i>L2 ENPR</i>	function <i>f</i>
<i>AMEM</i>	2
<i>CRCT</i>	0
<i>ARMZ</i>	1,0
<i>CRVL</i>	1,-6
<i>ARMZ</i>	1,1
<i>L3 NADA</i>	while
<i>CRVL</i>	1,1
<i>CRVL</i>	1,-5
<i>CMEG</i>	<i>i&lt;=n</i>
<i>DSVF</i>	<i>L4</i>
<i>CRVL</i>	1,0
<i>AMEM</i>	1
<i>CRVL</i>	1,1
<i>CHPP</i>	1,-9,1
<i>SOMA</i>	
<i>ARMZ</i>	1,0
<i>CRVL</i>	1,1
<i>CRCT</i>	1

<i>SOMA</i>		
<i>ARMZ</i>	1,1	<i>i:=i+1</i>
<i>DSVS</i>	<i>L3</i>	
<i>L4 NADA</i>		
<i>CRVL</i>	1,0	
<i>ARMZ</i>	1,-10	<i>f:=s</i>
<i>DMEM</i>	2	<i>end</i>
<i>RTPR</i>	5	
<i>L5 ENPR</i>	1	<b>function <i>h1</i></b>
<i>CRCT</i>	2	
<i>CRVL</i>	1,-5	
<i>MULT</i>		
<i>CRCT</i>	1	
<i>SOMA</i>		
<i>CRVL</i>	1,-5	
<i>MULT</i>		
<i>ARMZ</i>	1,-6	<i>h1:=(2*k+1)*k</i>
<i>RTPR</i>	1	<i>end</i>
<i>L6 ENPR</i>	1	<b>function <i>h2</i></b>
<i>CRVL</i>	1,-5	
<i>CRVL</i>	1,-5	
<i>MULT</i>		
<i>ARMZ</i>	1,-6	<i>h2:=j*j</i>
<i>RTPR</i>	1	
<i>L1 NADA</i>		
<i>AMEM</i>	1	
<i>CREG</i>	<i>L5,0</i>	
<i>CRCT</i>	1	
<i>AMEM</i>	1	
<i>CREG</i>	<i>L6,0</i>	
<i>CRCT</i>	5	
<i>CRCT</i>	10	
<i>CHPR</i>	<i>L2,0</i>	
<i>CHPR</i>	<i>L2,0</i>	
<i>IMPR</i>		<i>write(...)</i>
<i>PARA</i>		<i>end.</i>

Note-se que os endereços textuais usados nesta tradução são:

<i>f: 1,-10</i>	<i>i: 1,1</i>
<i>g: 1,-9</i>	<i>h1: 1,-6</i>
<i>m: 1,-6</i>	<i>k: 1,-5</i>
<i>n: 1,-5</i>	<i>h2: 1,-6</i>
<i>s: 1,0</i>	<i>j: 1,-5</i>

Deve-se observar que, com esta nova versão da MEPA, o vetor de registradores de base tornou-se redundante, pois a mesma informação pode ser encontrada percorrendo-se a cadeia estática que começa no registro de ativação corrente. A única informação que falta é o apontador para o próprio registro corrente. Este último poderia ser mantido num registrador especial da MEPA, cujo valor fosse modificado pelas instruções, sempre que necessário. Deixaremos ao leitor a elaboração da MEPA que siga esta sugestão (veja Exercício 9.6). Deve-se notar que, em geral, o vetor de registradores de base é mantido a fim de se obter acesso eficiente às variáveis não-locais de um procedimento. Por outro lado, a necessidade da sua atualização torna menos eficientes as chamadas e retornos de procedimentos.

A inclusão de rótulos e de comandos de desvio torna-se muito simples nesta nova versão da MEPA. Uma vez que o comando de desvio tem que estar contido dentro do escopo do rótulo de destino, conclui-se que os registradores de base relevantes já contêm os valores corretos, sendo apenas necessário ajustar a altura da pilha. Deve-se observar que na versão anterior da MEPA, isto é, definida na Seção 8.12, foi necessário restaurar vários registradores de base, a fim de simular as ações que seriam executadas pelas várias instruções de retorno. Nessa nova versão, não há mais necessidade de fazê-lo, pois cada retorno restaura todos os registradores de base relevantes. Conclui-se, então, que o comando de desvio poderá ser traduzido pela instrução *DSVS*. O ajuste da altura da pilha continuará sendo feito pela instrução *ENRT* já definida anteriormente:

*ENRT j,n* (Entrar no rótulo):

$$s := D[j] + n - 1$$

onde *j* é o nível léxico do procedimento em que ocorre o rótulo, e *n* é o número de posições de pilha ocupadas pelas suas variáveis locais.

### Exemplo 9.2

Com a nova versão da MEPA, teremos a seguinte tradução para o programa da Figura 8.28:

	<i>INPP</i>	<b>program</b>
	<i>AMEM</i>	<i>l</i>
	<i>DSVS</i>	<i>L1</i>
<i>L2</i>	<i>ENPR</i>	<i>l</i>
	<i>AMEM</i>	<i>l</i>
	<i>DSVS</i>	<i>L5</i>
<i>L6</i>	<i>ENPR</i>	<i>2</i>
	<i>CRVL</i>	<i>2,-5</i>
	<i>CRCT</i>	<i>0</i>
	<i>CMME</i>	<b>if</b> <i>z&lt;0</i>
	<i>DSVF</i>	<i>L7</i>
	<i>DSVS</i>	<i>L4</i>
	<i>DSVS</i>	<i>L8</i>
<i>L7</i>	<i>NADA</i>	<b>else</b>
	<i>CRVL</i>	<i>2,-5</i>
	<i>CRCT</i>	<i>0</i>
	<i>CMIG</i>	<b>if</b> <i>z=0</i>
	<i>DSVF</i>	<i>L9</i>

<i>CRCT</i>	<i>2</i>	
<i>ARMZ</i>	<i>2,-6</i>	<i>f:=2</i>
<i>DSVS</i>	<i>L10</i>	
<i>L9</i>	<i>NADA</i>	<b>else</b>
	<i>AMEM</i>	<i>l</i>
	<i>CRVL</i>	<i>2,-5</i>
	<i>CRCT</i>	<i>2</i>
	<i>SUBT</i>	
	<i>CHPR</i>	<i>L6,2</i>
	<i>CRVL</i>	<i>2,-5</i>
	<i>MULT</i>	
	<i>CRCT</i>	<i>l</i>
	<i>SOMA</i>	
	<i>ARMZ</i>	<i>2,-6</i>
<i>L10</i>	<i>NADA</i>	
<i>L8</i>	<i>NADA</i>	
	<i>RTPR</i>	<i>1</i>
		<b>end</b>
<i>L5</i>	<i>NADA</i>	
<i>L3</i>	<i>ENRT</i>	<i>1,1</i>
	<i>AMEM</i>	<i>l</i>
	<i>CRVI</i>	<i>1,-5</i>
	<i>CHPR</i>	<i>L6,1</i>
	<i>ARMZ</i>	<i>1,0</i>
	<i>CRVL</i>	<i>1,0</i>
	<i>ARMI</i>	<i>1,-5</i>
	<i>CRVI</i>	<i>1,-5</i>
	<i>CRVL</i>	<i>0,0</i>
	<i>CMME</i>	
	<i>DSVF</i>	<i>L11</i>
	<i>DSVS</i>	<i>L3</i>
<i>L11</i>	<i>NADA</i>	
<i>L4</i>	<i>ENRT</i>	<i>1,1</i>
	<i>CRVL</i>	<i>0,0</i>
	<i>CRCT</i>	<i>l</i>
	<i>SUBT</i>	
	<i>ARMZ</i>	<i>0,0</i>
	<i>DMEM</i>	<i>l</i>
	<i>RTPR</i>	<i>1</i>
<i>L12</i>	<i>ENPR</i>	<i>l</i>
	<i>DSVS</i>	<i>L13</i>
<i>L14</i>	<i>ENPR</i>	<i>2</i>
	<i>AMEM</i>	<i>l</i>
	<i>LEIT</i>	
	<i>ARMZ</i>	<i>2,0</i>
	<i>CREN</i>	<i>2,0</i>

<i>CHPR</i>	<i>L2,2</i>	$p(y)$
<i>CRVL</i>	<i>2,0</i>	
<i>IMPR</i>		<i>write(y)</i>
<i>DMEM</i>	<i>1</i>	<b>end</b>
<i>RTPR</i>	<i>0</i>	
<i>L13 NADA</i>		
<i>CHPR</i>	<i>L14,1</i>	<i>q</i>
<i>RTPR</i>	<i>0</i>	<b>end</b>
<i>L1 NADA</i>		
<i>LEIT</i>		
<i>ARMZ</i>	<i>0,0</i>	<i>read(x)</i>
<i>CHPR</i>	<i>L12,0</i>	<i>r</i>
<i>DMEM</i>	<i>1</i>	<b>end.</b>
<i>PARA</i>		

### 9.3 PASSAGEM DE PARÂMETROS POR NOME

Consideremos o programa da Figura 9.7, escrito em pseudo-Pascal, em que a função *f* tem um parâmetro formal que deve ser passado *por nome*. Se adotarmos a semântica semelhante à do Algol 60, isto significará que o valor do parâmetro efetivo correspondente a *e* deverá ser recalculado no seu contexto original toda vez que ele for usado dentro do corpo de *f*. Os valores obtidos em cada avaliação poderão ser diferentes se o parâmetro efetivo depender, por exemplo, de alguma variável que é modificada dentro de *f*. É fácil ver que um efeito equivalente à passagem por nome pode ser obtido substituindo-se, no programa-fonte, esse parâmetro por uma função sem parâmetros. A Figura 9.8 ilustra esta substituição para o caso do programa da Figura 9.7. Chamaremos de *fictícias* as funções introduzidas por esta substituição,<sup>1</sup> que ficará a cargo do compilador.

```
program exemplo(input,output);
var j,k: integer;
function f(name e: integer; var i: integer; m,n: integer): integer;
var s: integer;
begin
  s:=0; i:=m;
  while i<=n
    do begin
      s:=s+e;
      i:=i+1
    end;
  f:=s
end (*f *);
begin
  write(f((2*k+l)*k,k,l,f(j*j,j,5,10)))
end.
```

Fig. 9.7

```
program exemplo(input,output);
var j,k: integer;
function f(function e: integer; var i: integer;
           m,n: integer): integer;
var s: integer;
begin
  s:=0; i:=m;
  while i<=n
    do begin
      s:=s+e;
      i:=i+1
    end;
  f:=s
end (*f *);
function g1: integer;
begin g1:=(2*k+l)*k end (*g1 *);
function g2: integer;
begin g2:=j*j end (*g2 *);
begin
  write(f(g1,k,l,f(g2,j,5,10)))
end.
```

Fig. 9.8

### Exemplo 9.3

O programa da Figura 9.7 produz a seguinte tradução:

<i>INPP</i>		<b>program</b>
<i>AMEM</i>	<i>2</i>	<i>var j,k</i>
<i>DSVS</i>	<i>L1</i>	
<i>L2 ENPR</i>	<i>1</i>	<b>function</b> <i>f</i>
<i>AMEM</i>	<i>1</i>	<i>var s</i>
<i>CRCT</i>	<i>0</i>	
<i>ARMZ</i>	<i>1,0</i>	<i>s:=0</i>
<i>CRVL</i>	<i>1,-6</i>	
<i>ARMI</i>	<i>1,-7</i>	<i>i:=m</i>
<i>L3 NADA</i>		<b>while</b>
<i>CRVI</i>	<i>1,-7</i>	
<i>CRVL</i>	<i>1,-5</i>	
<i>CMEG</i>		<i>i&lt;=n</i>
<i>DSVF</i>	<i>L4</i>	<b>do</b>
<i>CRVL</i>	<i>1,0</i>	
<i>AMEM</i>	<i>1</i>	
<i>CHPP</i>	<i>1,-10,1</i>	
<i>SOMA</i>		
<i>ARMZ</i>	<i>1,0</i>	<i>s:=s+e</i>

<sup>1</sup>Em inglês: *thunks*.

	<i>CRVI</i>	1,-7	
	<i>CRCT</i>	1	
	<i>SOMA</i>		
	<i>ARMI</i>	1,-7	<i>i:=i+1</i>
	<i>DVS</i>	<i>L3</i>	
<i>L4</i>	<i>NADA</i>		
	<i>CRVL</i>	1,0	
	<i>ARMZ</i>	1,-11	<i>f:=s</i>
	<i>DMEM</i>	1	<b>end</b>
	<i>RTPR</i>	6	
<i>L1</i>	<i>NADA</i>		
	<i>AMEM</i>	1	
	<i>DVS</i>	<i>L5</i>	
<i>L6</i>	<i>ENPR</i>	1	função fictícia
	<i>CRCT</i>	2	
	<i>CRVL</i>	0,1	
	<i>MULT</i>		
	<i>CRCT</i>	1	
	<i>SOMA</i>		
	<i>CRVL</i>	0,1	
	<i>MULT</i>		
	<i>ARMZ</i>	1,-5	$(2*k+1)*k$
	<i>RTPR</i>	0	
<i>L5</i>	<i>NADA</i>		
	<i>CREG</i>	<i>L6,0</i>	
	<i>CREN</i>	0,1	
	<i>CRCT</i>	1	
	<i>AMEM</i>	1	
	<i>DVS</i>	<i>L7</i>	
<i>L8</i>	<i>ENPR</i>	1	função fictícia
	<i>CRVL</i>	0,0	
	<i>CRVL</i>	0,0	
	<i>MULT</i>		
	<i>ARMZ</i>	1,-5	<i>j*j</i>
	<i>RTPR</i>	0	
<i>L7</i>	<i>NADA</i>		
	<i>CREG</i>	<i>L8,0</i>	
	<i>CREN</i>	0,0	
	<i>CRCT</i>	5	
	<i>CRCT</i>	10	
	<i>CHPR</i>	<i>L2,0</i>	
	<i>CHPR</i>	<i>L2,0</i>	
	<i>IMPR</i>		<i>write(...)</i>
	<i>DMEM</i>	2	<b>end.</b>
	<i>PARA</i>		

Deve-se mencionar aqui que a implementação da passagem por nome em Algol 60 é ainda mais complicada, pois este mecanismo substitui também a chamada por referência. Assim, um parâmetro efetivo da forma  $a[E_1, \dots, E_n]$ , onde  $a$  é o nome de uma matriz e  $E_i$  são expressões, deve ser reavaliado toda vez que o parâmetro formal correspondente for utilizado. Conseqüentemente, a função fictícia deverá devolver sempre um endereço, e o procedimento que chamar a função fictícia deverá decidir se necessita o endereço, ou então o seu conteúdo. Considere-se, por exemplo, o comando  $x := x + 1$ , em que  $x$  é um parâmetro formal passado por nome (veja, também, o Exercício 9.10).

Uma outra fonte de complicação ao se implementar o Algol 60 advém do fato de que nessa linguagem não há a restrição de que os procedimentos passados como parâmetros têm que ter, por sua vez, os seus parâmetros passados por valor (veja, também, o Exercício 9.11).

#### 9.4 BLOCOS COM DECLARAÇÕES LOCAIS

Em Pascal, as declarações somente podem ser introduzidas ao nível de procedimentos e funções. Outras linguagens permitem o uso de *blocos*, que são comandos com declarações locais. Em Algol 60, um bloco é delimitado pelos símbolos **begin** e **end** e constituído por uma seqüência de declarações seguida de uma seqüência de comandos. A semântica do Algol 60 indica que a execução de um comando que é um bloco equivale à chamada de um *procedimento anônimo* cujo corpo é igual ao bloco em questão, e que está declarado no procedimento em que aparece o bloco. Essa regra deve ser aplicada recursivamente, pois blocos podem

```

procedure p ...;
  var x: integer;
  .
  .
  .
begin
  .
  .
  .
block
  var y: integer;
  .
  .
  .
begin
  .
  .
  .
end (* block *);
  .
  .
  .
end (* p *);

procedure p ...;
  var x: integer;
  .
  .
  .
procedure pa;
  var y: integer;
  .
  .
  .
begin
  .
  .
  .
end (* pa *);
  .
  .
  .
begin
  .
  .
  .
pa;
  .
  .
  .
end (* p *);
  .
  .
  .
end (* p *);

```

Fig. 9.9

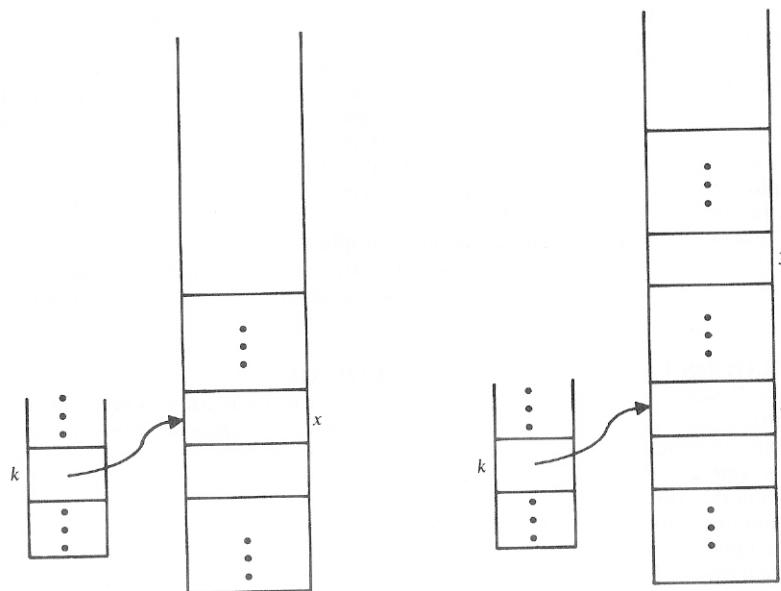


Fig. 9.10

aparecer dentro de outros blocos. A Figura 9.9 ilustra esta regra, para um programa em pseudo-Pascal, em que introduzimos blocos. Devido à sintaxe distinta do Pascal, indicamos um bloco por meio da construção **block ... begin ... end**, em que as declarações ficam entre **block** e **begin**, e os comandos entre **begin** e **end**. O procedimento anônimo foi indicado por *pa*. Essa discussão mostra uma maneira simples de implementar blocos. É suficiente que o compilador, ao traduzir o programa-fonte, gere código-objeto equivalente à aplicação da regra.

Uma outra maneira de implementar blocos consiste em considerar os seus registros de ativação como sendo parte do registro de ativação do procedimento em que estão diretamente encaixados. A alocação e a desalocação de memória para variáveis locais do bloco seriam feitas pelas mesmas instruções **AMEM** e **DGMEM**, sem a necessidade de usar o complicado mecanismo de chamada e retorno de procedimentos. A Figura 9.10 ilustra esquematicamente as configurações da MEPA imediatamente antes e depois da entrada no bloco da Figura 9.9a. Supusemos que o nível léxico do procedimento *p* é *k*. Note-se que o endereço textual da variável *y* do bloco seria nesta implementação da forma *k,n*, com deslocamento *n* conveniente. Deixaremos ao leitor a elaboração mais completa dessa implementação (Exercício 9.12).

## 9.5 MATRIZES

A linguagem Pascal impõe que os limites de matrizes sejam constantes determinadas em tempo de compilação, resultando numa implementação bastante simples. Neste caso, o método usual de alocação da memória é linearizar a matriz e reservar um bloco de posições de memória consecutivas no topo da pilha *M*. A mesma instrução **AMEM** pode ser usada, pois o número de posições alocadas é fixo, e conhecido em tempo de compilação. O compilador dispõe, também, de todos os dados necessários para calcular o endereço textual de qualquer elemento da matriz, e em particular do seu primeiro elemento, que pode ser usado como endereço de referência para algumas instruções.

A implementação das matrizes exige definição de novas instruções para a MEPA, e será deixada para o leitor (veja Exercício 9.13). Os seguintes aspectos devem ser considerados nessa implementação: (a) elementos de matrizes como expressões; (b) elementos de matrizes como variáveis; (c) matrizes como parâmetros passados por referência; (d) matrizes como parâmetros passados por valor. Note-se que, devido aos limites fixos das matrizes, não será necessário modificar as instruções já definidas que dependem do tamanho da memória ocupada pelas variáveis, tais como **AMEM**, **DGMEM**, **RTPR** e **ENRT**.

A implementação de matrizes torna-se mais complicada quando os seus limites podem ser calculados durante a execução do programa, a exemplo do Algol 60. Consideremos o programa em pseudo-Pascal esboçado na Figura 9.11. O esquema de alocação sugerido acima reservaria espaço para as variáveis locais do procedimento *p* da maneira indicada na Fig. 9.12a. Entretanto, uma vez que o compilador não pode calcular o tamanho das matrizes, ele não poderia, tampouco, determinar o deslocamento do primeiro elemento da matriz *b* (a variável simples *i* poderia ter espaço alocado abaixo de *a* e *b*). Uma solução conveniente é utilizar um apontador para a área de memória onde ficam os elementos da matriz. Cada apontador para matriz (chamado, às vezes, de *descritor da matriz*) é tratado como uma variável simples e recebe um endereço textual através do qual se fazem referências à matriz. A área para armazenar os elementos das matrizes pode ser alocada imediatamente acima das variáveis locais do procedimento. A Figura 9.12b ilustra esta alocação no caso do programa da Figura 9.11.

A fim de calcular o endereço de um elemento de matriz como *a[i<sub>1</sub>,...,i<sub>n</sub>]*, o sistema de execução necessitará de dados adicionais, como por exemplo os limites

```

program exemplo(input,output);
var m,n: integer;
procedure p;
var x,y: integer;
a,b: array[m..n] of integer;
i: integer;
begin
.
.
.
end (* p *);
begin
.
.
.
read(m,n);
p;
.
.
.
end.

```

Fig. 9.11

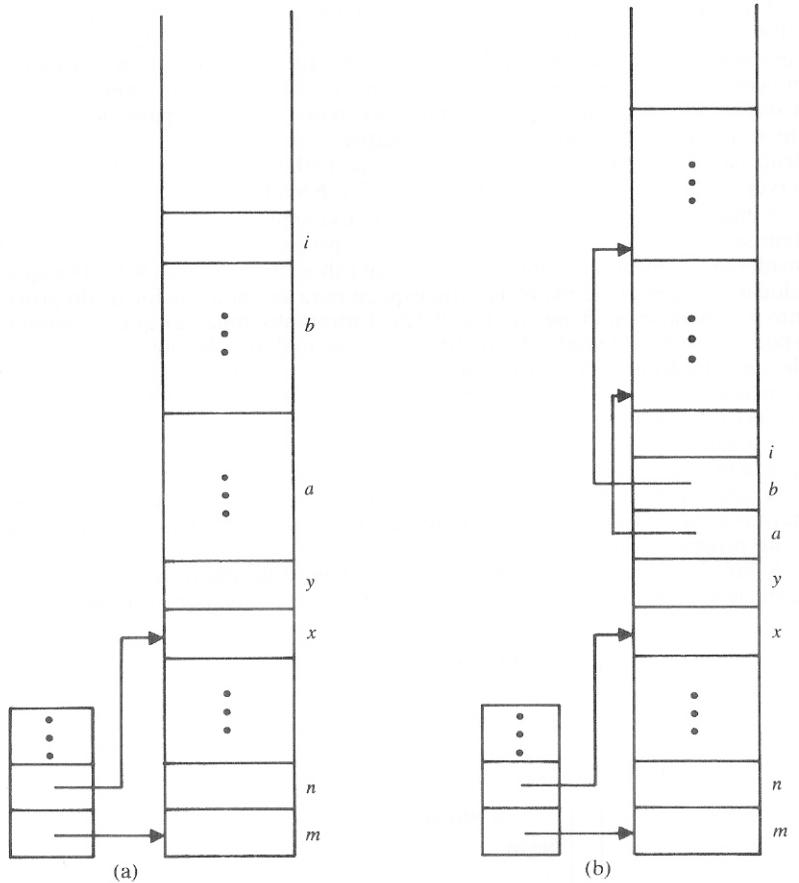


Fig. 9.12

da matriz, que não são conhecidos em tempo de compilação. Estes dados, que devem ser calculados quando são executadas as instruções da MEPA que traduzem a declaração da matriz, formam o chamado *vetor de informação*.<sup>1</sup> O vetor de informação, que ocupa um espaço cujo tamanho pode ser calculado em tempo de compilação, pode ser guardado juntamente com o apontador da matriz, ou então no início da mesma área onde estão os elementos da matriz.

Deve-se notar, finalmente, que, no caso destes tipos de matrizes com limites dinâmicos, haverá necessidade de modificar-se algumas das instruções da MEPA. Um exemplo é a instrução *ENRT* que pressupõe que a área alocada para as variáveis locais de um procedimento tem tamanho conhecido em tempo de compilação. Uma solução conveniente é criar, em cada registro de ativação de procedimento, uma variável local fictícia, denominada *apontador de trabalho*, que deverá conter o endereço da última posição da pilha reservada para as variáveis locais, inclusive matrizes. A instrução *ENRT* usaria esse apontador de trabalho para atualizar o registrador *s*. A implementação completa de matrizes dinâmicas será deixada, novamente, para o leitor (veja Exercício 9.14).

<sup>1</sup>Em inglês: *dope vector*.

## 9.6 IMPLEMENTAÇÃO DA MEPA

Até este momento preocupamo-nos apenas com a definição de uma máquina hipotética conveniente para a implementação do Pascal. É claro que, na prática, o compilador deverá gerar código-objeto em linguagem de máquina, ou pelo menos em linguagem de montagem de um computador real. Não é difícil ver que, para cada instrução da MEPA, é possível estabelecer uma tradução para a linguagem de máquina (ou montagem) de um dado computador. A tradução estará baseada na própria definição da instrução da MEPA, consistindo, em geral, em várias instruções em linguagem de máquina.

A maneira mais usada para implementar esta tradução é embuti-la dentro do próprio compilador. Assim, toda vez que o compilador decidir gerar uma instrução da MEPA, ele deverá chamar uma rotina especial que produzirá as instruções de máquina (ou de montagem) correspondentes. Caso o código seja produzido em linguagem de máquina, a rotina deverá encarregar-se, também, dos problemas encontrados normalmente em montadores, tais como manutenção de uma tabela de símbolos para os rótulos *L<sub>i</sub>* gerados pelo compilador e tratamento de referências a rótulos ainda não definidos. No caso da geração de código em linguagem de montagem, o próprio montador tratará desses problemas; o preço a ser pago é a necessidade de um processamento adicional após a compilação, que é a montagem.

Uma outra maneira de se conseguir o mesmo efeito é utilizar, quando disponível, um macromontador. Neste caso, cada instrução da MEPA será tratada como uma macroinstrução, e a implementação da MEPA consistirá na definição das macros correspondentes. Esta maneira de implementar a MEPA também exige uma macromontagem adicional, mas é muito cômoda quando se trata de um compilador experimental ou didático. Note-se que no caso de o próprio compilador gerar o código-objeto para a máquina, a rotina de geração será, no fundo, um macromontador especializado.

Não entraremos nos pormenores da implementação da MEPA, pois esses dependem muito do particular computador utilizado. Ao invés disso, indicamos na Figura 9.13 como poderiam ser as definições de algumas instruções da MEPA (versão da Seção 9.2) para um macromontador típico. As mesmas traduções poderiam ser geradas pela rotina já mencionada do compilador. Supusemos tratar-se de um computador bastante convencional, com oito registradores numerados de 0 a 7, e ocupando as posições 0 a 7 da memória. O registrador 0 é usado como acumulador, e os registradores 1 a 7, como registradores de indexação. Escolhemos o registrador 7 para representar o registrador *s* da MEPA; o registrador *i* não aparece explicitamente, sendo o próprio contador de instruções da máquina. Os registradores 1 a 5 serão usados para implementar os registradores de base *D[0]* a *D[4]*. O nível léxico máximo permitido será, portanto, quatro. O registrador 6 será usado como registrador auxiliar. Cada instrução desse computador possui um campo de endereço composto de deslocamento e do número do registrador utilizado como indexador, ou então como um dos operandos. O símbolo # precede quantidades literais, @ indica o endereçamento indireto e \* como operando é o endereço da instrução corrente. Os significados das instruções utilizadas são:

*INC (DEC)*: incrementar (decrementar) o valor do registrador indicado, com o valor do deslocamento;

*CRG*: carregar um valor no acumulador (registrador 0);

*ARM*: armazenar o valor do acumulador;

*CMP*: comparar o valor do acumulador;

*DSV*: desviar;

*DME*: desviar se a última comparação indicou “menor”;

*SOM*: somar um valor ao acumulador.

As pseudo-instruções *MACRO* e *MFIM* delimitam a definição da macro, e a pseudo-instrução *EQU* atribui um valor a um símbolo, em tempo de montagem.

<i>CRCT</i>	<i>MACRO</i>	<i>K</i>	<i>NADA</i>	<i>MACRO</i>	*
	<i>INC</i>	<i>I(7)</i>		<i>EQU</i>	
	<i>CRG</i>	<i>#K</i>		<i>MFIM</i>	
	<i>ARM</i>	<i>0(7)</i>			
	<i>MFIM</i>				
<i>CRVL</i>	<i>MACRO</i>	<i>M,N</i>	<i>RTPR</i>	<i>MACRO</i>	<i>N</i>
	<i>INC</i>	<i>I(7)</i>		<i>CRG</i>	<i>-I(7)</i>
	<i>CRG</i>	<i>N(M+1)</i>		<i>ARM</i>	<i>6</i>
	<i>ARM</i>	<i>0(7)</i>		<i>CRG</i>	<i>-2(7)</i>
	<i>MFIM</i>			<i>ARM</i>	<i>0(6)</i>
<i>ARMZ</i>	<i>MACRO</i>	<i>M,N</i>		<i>DEC</i>	<i>N+4(7)</i>
	<i>CRG</i>	<i>0(7)</i>		<i>CRG</i>	<i>6</i>
	<i>ARM</i>	<i>N(M+1)</i>		<i>CMP</i>	<i>#3</i>
	<i>DEC</i>	<i>I(7)</i>		<i>DME</i>	<i>*+7</i>
	<i>MFIM</i>			<i>CRG</i>	<i>0(6)</i>
<i>SOMA</i>	<i>MACRO</i>			<i>DEC</i>	<i>I(0)</i>
	<i>CRG</i>	<i>0(7)</i>		<i>CRG</i>	<i>@0</i>
	<i>SOM</i>	<i>-I(7)</i>		<i>DEC</i>	<i>I(6)</i>
	<i>ARM</i>	<i>-I(7)</i>		<i>ARM</i>	<i>0(6)</i>
	<i>DEC</i>	<i>I(7)</i>		<i>DSV</i>	<i>*-8</i>
	<i>MFIM</i>			<i>DSV</i>	<i>@(N+1) (7)</i>

Fig. 9.13

Numa implementação real poder-se-iam introduzir várias otimizações no código gerado, através da macromontagem condicional. Um exemplo típico seria a sequência de instruções da MEPA:

```
SOMA
CRVL 3,5
```

que de acordo com a Figura 9.13 produziria:

```
CRG 0(7)
SOM -I(7)
ARM -I(7)
* DEC I(7)
* INC I(7)
CRG 5(4)
ARM 0(7)
```

As instruções marcadas com asteriscos poderiam ser eliminadas sem alterar o resultado. Outras otimizações são possíveis, eliminando-se quase que completamente a manipulação do registrador 7 (isto é, do registrador *s* da MEPA).

Devemos notar, finalmente, que existem computadores cujas linguagens de máquina são semelhantes à MEPA e que são orientados, portanto, para a implementação de certas linguagens de programação. Um exemplo clássico é a linha de

computadores Burroughs 6700/7700, orientados para a implementação de uma versão estendida do Algol 60.

### Exercícios

1. Foi sugerido, na Seção 9.2, que ao invés de manter a cadeia estática como parte de um parâmetro efetivo que é procedimento, poder-se-ia passar o vetor de registradores de base corrente, que é, em geral, muito curto. Reproje a MEPA de maneira a implementar esta idéia. Quais são as vantagens e as desvantagens desta solução?
2. A instrução *RTPR* tem o mesmo funcionamento para retornar de procedimentos que foram passados como parâmetros e para retornar de procedimentos que foram chamados diretamente. Uma outra solução seria passar, na chamada do procedimento, uma marca especial indicando o tipo de chamada. A instrução *RTPR* usaria esta marca para decidir o tipo de retorno necessário. Reproje a MEPA implementando esta solução.
3. Em Algol 60, rótulos podem ser passados como parâmetros. Acrescente este mecanismo ao nosso Pascal e indique a sua implementação. (*Sugestão:* a implementação é quase idêntica aos procedimentos passados como parâmetros.)
4. Refaça o Exercício 8.17 para a versão da MEPA definida até a Seção 9.2.
5. Nem sempre é necessário que a instrução *RTPR*, executando um retorno para nível *k*, restaure todos os registradores *D*[*k*], ..., *D*[1]. Verifique em que condições o número de registradores a serem restaurados pode ser reduzido, e implemente esta idéia.
6. Foi mencionado, na Seção 9.2, que o vetor de registradores de base tornou-se redundante. Reproje a MEPA de maneira a eliminá-lo. Discuta as vantagens e as desvantagens desta nova implementação.
7. Na instrução *CREG p,k*, o parâmetro *k* é o nível léxico do procedimento dentro do qual o procedimento que está sendo passado como parâmetro foi declarado. Poderia-se usar, ao invés desse, o nível do procedimento corrente? Justifique.
8. Suponha que se deseja incluir na MEPA mecanismos que facilitem a depuração de programas. Uma maneira seria fornecer, ao ocorrer um erro de execução, o número do comando corrente, o nome do procedimento corrente e a sequência das chamadas que levaram à ativação corrente. Quais são as informações que teriam que ser incluídas nos registros de ativação de procedimentos? Modifique o projeto da MEPA, incluindo este mecanismo.
9. Adapte a propriedade enunciada no Exercício 8.20 à nova versão da MEPA, e mostre que ela é válida.
10. Foi mencionado na Seção 9.3 que a implementação da passagem de parâmetros por nome, como em Algol 60, torna-se complicada pelo fato de ela substituir, também, a passagem por referência. Suponha que este é, também, o caso do nosso Pascal, e reproje a MEPA de acordo. (*Sugestão:* toda função fictícia deverá devolver um endereço.)
11. Em Algol 60, os procedimentos passados como parâmetros não precisam ter, por sua vez, os seus parâmetros passados por valor. Suponha que seja este, também, o caso do nosso Pascal, e indique uma solução. (*Sugestão:* todos os procedimentos devem ter os seus parâmetros passados por referência (ou por nome), e o efeito de passagem por valor deve ser simulado.)
12. Complete a implementação de blocos sugerida na Seção 9.4.
13. Implemente matrizes com limites estáticos (veja Seção 9.5).
14. Implemente matrizes com limites dinâmicos (veja Seção 9.5).
15. Suponha que o nosso Pascal tem blocos com declarações locais e matrizes dinâmicas. Discuta as consequências deste fato e soluções possíveis.

16. Discuta a implementação de registros e apontadores do Pascal, e que não foram incluídos na nossa versão simplificada.
17. Discuta a implementação de co-rotinas, inclusive recursivas.
18. Discuta a implementação de processos paralelos.
19. Discuta as otimizações que um macromontador condicional poderia introduzir no nosso projeto da MEPA.
20. (Projeto) Implemente a MEPA no sistema de computação que está à sua disposição, usando de preferência um macromontador.

## NOTAS BIBLIOGRÁFICAS

As mesmas referências do Capítulo 8 aplicam-se a este capítulo. Conceitos básicos sobre co-rotinas (veja Exercício 9.17) podem ser encontrados em Pratt (1975). Mello (1980) discute a implementação de processos paralelos em Ada. Uma descrição dos computadores da série Burroughs 6700/7700 pode ser encontrada em Organick (1973).

10

# Organização do Compilador

## 10.1 GENERALIDADES

Nos capítulos anteriores deste texto houve preocupação com dois aspectos básicos da nossa linguagem. Os Capítulos 2 a 4 trataram o problema de analisar sintaticamente os programas, ou seja, de descobrir a sua estrutura e revelar todas as construções utilizadas. Os Capítulos 7 a 9 trataram a implementação das várias construções da linguagem de maneira a satisfazer as definições semânticas.

O objetivo deste capítulo é indicar como estes dois aspectos — sintático e semântico — podem ser integrados num compilador, e quais são os mecanismos necessários. A fim de tornar a discussão mais específica e mais didática, suporemos que o método de análise sintática utilizado é o método descendente com a implementação recursiva. Suporemos, também, que o nosso objetivo é conseguir um compilador de um passo que gera o programa-objeto à medida que o programa-fonte está sendo examinado da esquerda para a direita.

Obviamente, esta não é a única maneira de implementar um compilador. Existem muitas implementações diferentes, que usam outros métodos de análise sintática, ou funcionam em vários passos, gerando certas representações intermediárias. A escolha do método de implementação depende de vários fatores, tais como: da particular linguagem a ser implementada, da linguagem usada para escrever o compilador, da capacidade do sistema de computação disponível, da eficiência de compilação desejada, da necessidade ou não de se introduzir otimização de código, dos prazos prefixados para implementar o compilador e, finalmente, das preferências pessoais dos implementadores. Deve-se notar, entretanto, que o método a ser discutido neste capítulo é muito satisfatório, e foi utilizado na maioria das implementações do Pascal.

## 10.2 TABELA DE SÍMBOLOS

A medida que o compilador processa o programa-fonte, são encontrados identificadores que aparecem em declarações de variáveis e de procedimentos com seus parâmetros, bem como em referências a todos estes identificadores. Para cada referência, o compilador terá a necessidade de conhecer os atributos correspondentes. Por exemplo, no caso de uma referência a uma variável simples, os atributos seriam a sua categoria (isto é, o fato de ser uma variável simples), o seu tipo (inteiro ou booleano) e o seu endereço textual. Estas informações são normalmente associadas com o identificador quando é processada a declaração correspondente, e deverão ser guardadas para serem usadas enquanto for processado o trecho do programa que é o escopo desta declaração.

Em geral, estas informações são organizadas numa *tabela de símbolos*, que associa com cada identificador um certo número de atributos. A maneira de implementar a tabela de símbolos depende da linguagem a ser compilada, do sistema no qual o compilador será implementado e das características de eficiência desejadas.

Consideremos o programa esboçado na Figura 10.1. De acordo com as regras

```

program exemplo(input,output);
var a,b: integer;
.

procedure p(x);
var b,c : integer;
begin
.
.
①
end (* p *);

procedure q;
var c,d: integer;
.

procedure r(y);
var e,f: integer;
begin
.
.
②
end (* r *);
begin
.
.
③
end (* q *);
begin
.
.
④
end.

```

de escopos do Pascal, os conjuntos de identificadores aos quais pode haver referências nos vários pontos do programa-fonte são distintos. Assim, nos corpos dos procedimentos marcados na figura, temos os seguintes conjuntos de identificadores acessíveis:

- (1)  $a, x, b, c, p$  ( $b$  e  $c$  de  $p$ );
- (2)  $a, b, c, d, y, e, f, p, q, r$  ( $b$  global,  $c$  de  $q$ );
- (3)  $a, b, c, d, p, q, r$  ( $b$  global,  $c$  de  $q$ );
- (4)  $a, b, p, q$  ( $b$  global).

Percebe-se, facilmente, que temos novamente um fenômeno de pilha. À medida que o programa é processado, da esquerda para a direita, os identificadores introduzidos por último através de suas declarações serão os primeiros a terem os seus escopos encerrados pelo símbolo `end` correspondente. (Note-se que um identificador que corresponde a um procedimento é declarado no procedimento que o envolve.) Este fato sugere que, qualquer que seja a implementação física da tabela de símbolos, o seu funcionamento deve ser equivalente ao de uma pilha. A Figura 10.2 indica, de maneira esquemática, as configurações dessa pilha ao processar os corpos dos procedimentos marcados na Figura 10.1. Os campos vazios de cada entrada correspondem aos atributos a serem discutidos mais adiante. A cada referência a um identificador, os atributos correspondentes poderão ser obtidos da tabela, iniciando-se a busca desse identificador pelo topo da pilha. Note-se que dessa maneira fica resolvido, automaticamente, o problema da redeclaração de identificadores, como por exemplo o identificador  $b$  do procedimento  $p$ .

Uma implementação que seguisse literalmente a organização lógica indicada acima poderia ser muito ineficiente, pois o método de busca linear de identificadores poderia chegar a consumir 20 a 30% do tempo de compilação. Na prática, utilizam-se implementações mais eficientes, como a organização em árvore ou a técnica de

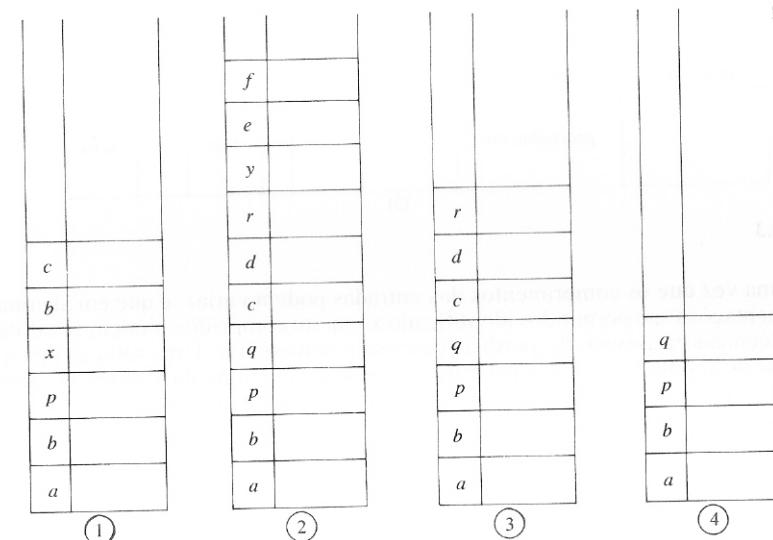


Fig. 10.2

espalhamento.<sup>1</sup> Não discutiremos aqui estas técnicas, mas suporemos apenas a existência de algumas rotinas para a manipulação e acesso às tabelas de símbolos. A rotina *INSERE(e)* colocará no topo da tabela a entrada *e*, que será em geral um registro composto de um identificador e os seus atributos. A rotina *BUSCA(i)* deverá devolver o índice na tabela de símbolos correspondente à última ocorrência do identificador *i*. Será devolvido um valor especial — por exemplo zero — se o identificador não ocorre na tabela. A rotina *ELIMINA(n)* deverá retirar do topo da tabela as *n* últimas entradas que corresponderão aos *n* identificadores declarados num escopo. Outras rotinas poderão ser definidas se necessário.

Os atributos associados com cada identificador dependerão da sua categoria. Alguns exemplos são:

1. variáveis simples: categoria, tipo e endereço textual;
2. parâmetros formais simples (isto é, que não são procedimentos, rótulos ou matrizes): categoria, tipo, endereço textual, mecanismo de passagem (por valor ou por variável);
3. procedimentos: categoria, rótulo interno (no programa-objeto), nível em que foi declarado, número de parâmetros formais, tipo e mecanismo de passagem de cada parâmetro.

A Figura 10.3 ilustra esquematicamente as três entradas exemplificadas acima, usando convenções óbvias.

identificador	categoria	nível	tipo	deslocamento
	variável simples			
identificador	categoria	nível	tipo	deslocamento
	parâmetro formal			passagem

identificador	categoria	nível	rótulo	n	t <sub>1</sub> , p <sub>1</sub>	...	t <sub>n</sub> , p <sub>n</sub>
	procedimento						

Fig. 10.3

Uma vez que os comprimentos das entradas podem variar, e que em algumas implementações são permitidos identificadores muito compridos, podem-se adotar várias técnicas para evitar desperdício de espaço de memória. Uma solução típica é a adoção de apontadores para a parte da informação cujo tamanho é variável, como indicado na Figura 10.4. Desta maneira, as entradas na tabela terão comprimentos fixos, facilitando a sua manipulação. As informações de tamanho variável poderão ser armazenadas numa área especial, cujo comportamento, como é fácil ver, seguirá também a disciplina de pilha.

No caso particular do Pascal, poderão ser guardados na tabela de símbolos,

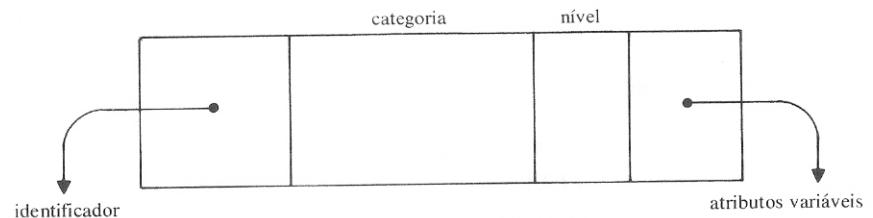


Fig. 10.4

além dos identificadores, os rótulos introduzidos pela declaração **label**; uma outra solução é usar uma tabela separada.

### 10.3 ANÁLISE SINTÁTICA E GERAÇÃO DE CÓDIGO

Como já foi mencionado na Seção 4.4, uma das vantagens do analisador descendente recursivo é a simplicidade de sua implementação e a sua flexibilidade. Uma boa parte do nosso compilador será constituída por rotinas análogas às do Exemplo 4.9, completadas com comandos que manipulam a tabela de símbolos, geram código-objeto e verificam a validade de algumas construções.

Discutiremos a seguir alguns exemplos típicos de rotinas do compilador. Suparemos que existe uma rotina de análise léxica chamada *ANALEX*, análoga à descrita na Seção 6.2, mas que permite a consulta de dois símbolos iniciais da cadeia de entrada, usando para isto as variáveis globais *símbolo1*, *símbolo2*, *átomo1* e *átomo2*. Note-se, entretanto, que *ANALEX* avança de um símbolo a cada chamada. Na realidade, não é estritamente necessário que *ANALEX* funcione assim, mas essa maneira simplifica as rotinas sintáticas. Suporemos, também, que existe uma rotina *GERA(rótulo, código, par1, par2, par3)* que gera a instrução da MEPA correspondente, gravando-a em geral num dispositivo como disco ou fita. Por conveniência, poderão ser omitidos os últimos parâmetros de *GERA*.

O processamento de expressões é relativamente simples. Como a nossa gramática do Pascal não faz distinção entre expressões inteiras e booleanas, esta deve

```

procedimento TERMO(var t);
início
  FATOR(t);
  enquanto símbolo1 ∈ código_de_mult_div_conj
    faça {s:=símbolo1;
           ANALEX;
           FATOR(t1)};
    caso s de
      código_de_multiplicação: {GERA (branco, "MULT");
      t2:=inteiro};
      código_de_divisão: {GERA (branco, "DIV")};
      t2:=inteiro}
      código_de_conjunção: {GERA (branco, "CONJ");
      t2:=booleano}
    fim do caso;
    se t ≠ t1 ou t ≠ t2 então ERRO}
  fim

```

Fig. 10.5

<sup>1</sup>Em inglês “hashing”.

```

procedimento FATOR(var t);
início
  caso símbolo1 de
    código_de_número:
      {GERA(branco, "CRCT", átomol);
      t:=inteiro; ANALEX};
    código_de_identificador:
      {k:=BUSCA(átomol); se k=0 então ERRO;
      atr:=tabela_de_símbolos [k];
      caso atr.category de
        variável_simples:
          {GERA(branco,"CRVL",atr.nível,atr.deslocamento);
          t:=atr.tipo; ANALEX};
        parâmetro_formal:
          {caso atr.passagem de
            valor: GERA(branco,"CRVL",atr.nível,atr.deslocamento);
            variável: GERA(branco,"CRVI",atr.nível,atr.deslocamento);
            fim do caso;
            t:=atr.tipo; ANALEX};
        constante:
          {t:=booleano;
          caso átomol de
            "true": GERA(branco,"CRCT",1);
            "false": GERA(branco,"CRCT",0)
            fim do caso;
            ANALEX};
        função: "tratamento de chamadas de funções";
        outros: ERRO
        fim do caso};
      código_de_abre_parêntese:
        {ANALEX; EXPRESSÃO(t);
        se símbolo1 ≠ código_de_fecha_parêntese então ERRO;
        ANALEX};
      código_de_negação:
        {ANALEX; FATOR(t);
        set ≠ booleano então ERRO;
        GERA(branco,"NEGA")};
      outros: ERRO
    fim do caso
  fim

```

Fig. 10.6

ser feita pelo compilador a fim de excluir, por exemplo, construções como  $25 + (x > y)$ . Convencionaremos, então, que as rotinas que processam expressões devolverão um resultado indicando o tipo da expressão. As Figuras 10.5 e 10.6 apresentam duas destas rotinas, correspondentes aos não-terminais  $\langle\text{termo}\rangle$  e

$\langle\text{fator}\rangle$ , respectivamente, como foram definidos na Seção 5.5. A segunda rotina está incompleta, faltando incluir o tratamento de chamadas de funções.

Algumas rotinas que tratam comandos poderiam ter a estrutura indicada nas Figuras 10.7, 10.8 e 10.9. Supusemos a existência da rotina *PRÓXIMO\_*

```

procedimento COMANDO_SEM_RÓTULO;
início
  caso símbolo1 de
    código_de_identificador:
      caso símbolo2 de
        código_de_atribuição: ATRIBUIÇÃO;
        outros: CHAMADA_DE_PROCEDIMENTO
        fim do caso;
        código_de_goto: DESVIO;
        código_de_begin: COMANDO_COMPOSTO;
        código_de_if: COMANDO_CONDICIONAL;
        código_de_while: COMANDO REPETITIVO;
        outros: ERRO
      fim do caso
    fim

```

Fig. 10.7

```

procedimento ATRIBUIÇÃO;
início
  k:=BUSCA(átomol); se k=0 então ERRO;
  atr:=tabela_de_símbolos [k];
  ANALEX; ANALEX;
  (* já foi verificada a presença de := *)
  EXPRESSÃO(t);
  se t ≠ atr.tipo então ERRO;
  caso atr.category de
    variável_simples:
      GERA(branco,"ARMZ",atr.nível,atr.deslocamento);
    parâmetro_formal:
      caso atr.passagem de
        valor: GERA(branco,"ARMZ",atr.nível,atr.deslocamento);
        variável: GERA(branco,"ARMI",atr.nível,atr.deslocamento)
        fim do caso;
    função:
      se "dentro do corpo da função"
      então GERA(branco,"ARMZ",atr.nível+1,atr.deslocamento)
      senão ERRO
    fim do caso
  fim

```

Fig. 10.8

```

procedimento COMANDO_CONDICIONAL;
início
  ANALEX; (* já foi verificada a presença de if *)
  PRÓXIMO_RÓTULO(ℓ1);
  EXPRESSÃO(t); se  $t \neq$  booleano então ERRO;
  GERA(branco, "DSVF", ℓ1);
  se símbolo $l \neq$  código_de_then então ERRO;
  ANALEX;
  COMANDO_SEM_RÓTULO;
  caso símbolo $l$  de
    código_de_else
      {PRÓXIMO_RÓTULO(ℓ2); GERA(branco, "DSVS", ℓ2);
       GERA(ℓ1, "NADA"); ANALEX;
       COMANDO_SEM_RÓTULO;
       GERA(ℓ2, "NADA");}
    outros: GERA(ℓ1, "NADA")
  fim do caso
fim

```

Fig. 10.9

$RÓTULO(\ell)$  que coloca na variável  $\ell$  o próximo rótulo consecutivo da forma  $Li$ , onde  $i$  é um inteiro positivo. Como todas estas rotinas são recursivas, deve-se tomar o cuidado de declarar como locais muitas das variáveis utilizadas, como por exemplo  $\ell1$  e  $\ell2$  na rotina *COMANDO\_CONDICIONAL*.

As Figuras 10.10, 10.11 e 10.12 ilustram alguns aspectos da implementação de declaração de procedimentos. Deve-se notar a maneira como é verificada a redeclaração de identificadores num mesmo procedimento ( $nível\_corrente$  é uma variável global do compilador). Na realidade, as rotinas indicadas são incompletas. Assim, por exemplo, a rotina *DECLARAÇÕES\_DE\_VARIÁVEIS* deveria devolver à rotina *BLOCO* o número de entradas que ela criou na tabela de símbolos, para que elas possam ser eliminadas após a compilação do bloco. Este mesmo número, ou no caso geral, o número de posições de memória ocupadas pelas variáveis locais, deve ser passado pela rotina *BLOCO* à rotina *COMANDO\_COMPOSTO*, que por sua vez deve passá-lo à rotina *COMANDO*. Esta última necessitará desse número para gerar corretamente a instrução *ENRT* no caso de um comando rotulado. A própria rotina *BLOCO* necessita desse número para gerar a instrução *DMEM*. A elaboração completa das rotinas será deixada a cargo do leitor. Deve-se notar que, declarando-se certas rotinas dentro de outras, pode-se evitar a passagem de alguns parâmetros, usando-se em seu lugar variáveis locais.

Uma observação final é que o procedimento *DECLARAÇÃO\_DE\_PROCEDIMENTO* pode ser modificado para incluir o caso de funções e programas principais cuja compilação é muito semelhante. Assim, a rotina *BLOCO* poderá ser eliminada, e as suas ações poderão ser executadas pela própria rotina *DECLARAÇÃO\_DE\_PROCEDIMENTO*.

```

procedimento DECLARAÇÃO_DE_PROCEDIMENTO;
início
  ANALEX; (* já foi verificada a presença de procedure *)
  se símbolo $l \neq$  código_de_identificador então ERRO;
   $k :=$  BUSCA(átomo $l$ );
  se ( $k > 0$ ) e (tabela_de_símbolos[k].nível = nível_corrente)
    então ERRO;
  PRÓXIMO_RÓTULO(ℓ1); PRÓXIMO_RÓTULO(ℓ2);
  GERA(branco, "DSVS", ℓ1);
  INSERE(átomo $l$ , procedimento, nível_corrente, ℓ2, 0, vetor_indefinido);
  ANALEX;
  nível_corrente := nível_corrente + 1;
  GERA(ℓ2, "ENPR", nível_corrente);
  "tratamento de parâmetros formais; atualização do vetor que foi deixado
  indefinido na tabela de símbolos";
  BLOCO;
  GERA(branco, "RTPR", número_de_posições_para_parâmetros);
  GERA(ℓ1, "NADA");
  ELIMINA(número_de_parâmetros);
  nível_corrente := nível_corrente - 1
fim

```

Fig. 10.10

```

procedimento BLOCO;
início
  se símbolo $l =$  código_de_label
    então DECLARAÇÃO_DE_RÓTUOS;
  se símbolo $l =$  código_de_var
    então DECLARAÇÕES_DE_VARIÁVEIS
  enquanto símbolo $l \in \{código_de_procedure, código_de_function\}$ 
    faça caso símbolo $l$  de
      código_de_procedure:
        DECLARAÇÃO_DE_PROCEDIMENTO;
      código_de_function:
        DECLARAÇÃO_DE_FUNÇÃO
    fim do caso;
  COMANDO_COMPOSTO;
  "elimina da tabela_de_símbolos todos os identificadores
  declarados neste bloco";
  "gera instrução DMEM conveniente"
fim

```

Fig. 10.11

```

procedimento DECLARAÇÕES_DE_VARIÁVEIS;
início
  ANALEX; (* já foi verificada a presença de var *)
  desl:=0;
  repita
    n:=0; término:=falso;
    repita
      se símbolo1 ≠ código_de_identificador então ERRO;
      k:=BUSCA(átomol);
      se (k>0) e (tabela_de_símbolos[k].nível=nível_corrente)
        então ERRO;
      INSERE([átomol,variável_simple,nível_corrente,tipo_indefinido,desl]);
      desl:=desl+1; n:=n+1;
    ANALEX;
    caso símbolo1 de
      código_de_vírgula: nada;
      código_de_dois_pontos: término:=verdadeiro;
      outros: ERRO
    fim do caso;
    ANALEX
  até término;
  GERA(branco,"AMEM",n)
  caso átomol de
    "integer": t:=inteiro;
    "boolean": t:=booleano;
    outros: ERRO
  fim do caso;
  ATUALIZA TIPO(t,n); ANALEX;
  se símbolo1 ≠ código_de_ponto_e_vírgula então ERRO;
  ANALEX
até que símbolo1 ≠ código_de_identificador
fim

```

Fig. 10.12

É aconselhável seguir, ao elaborar as rotinas do compilador, as cartas sintáticas do Apêndice II, em que foram eliminados vários não-terminais, de acordo com a observação da Seção 4.5.

#### 10.4 TRATAMENTO DE ERROS

Até este momento não nos preocupamos com o tratamento de erros. As rotinas do compilador indicadas na Seção 10.3 foram escritas como se o processo de tradução devesse ser encerrado ao encontrar o primeiro erro do programa-fonte. Entretanto, um compilador real deveria detetar, na medida do possível, todos os erros a fim de reduzir o número de compilações. A deteção de erros é bastante simples, e já foi indicada nas rotinas dadas. A parte mais complicada é a recuperação do compilador após a deteção de um erro, para que o processo de compilação possa ser continuado de maneira a detetar outros erros. Normalmente, a geração de código-objeto pode ser suspensa após a ocorrência do primeiro erro.

Além dos poucos erros detetados pelo analisador léxico, há dois tipos de erros que devem ser detetados nas rotinas do compilador. O primeiro tipo, denominado às vezes *erro sintático*, é a violação das regras da gramática da linguagem, dada em FNB. O segundo tipo, denominado às vezes (e impropriamente) *erro semântico*, é a violação das regras informais que acompanham a gramática da linguagem.

```

procedimento TESTE(s1,s2,mens);
início
  se símbolo1 ∈ s1
  então {MENSAGEM(mens)};
  s:=s1 ∪ s2;
  enquanto símbolo1 ∈ s faça ANALEX}
fim

```

Fig. 10.13

Em geral, a recuperação do compilador é mais simples no caso de um erro semântico. Assim, por exemplo, ao detetar uma expressão inteira quando era esperada uma expressão booleana, é suficiente produzir a mensagem correspondente e continuar o processo de tradução. Um outro exemplo é o uso de uma variável que não foi declarada ainda. Para que o compilador não emita a mesma mensagem toda vez que for encontrada esta mesma variável, pode-se colocar, após a primeira ocorrência, o identificador correspondente na tabela de símbolos marcado como sendo de um tipo indefinido. Referências a identificadores de tipo indefinido não devem produzir mensagem de erro. Outros erros, como redeclaração de identificadores, uso indevido de procedimentos, e assim por diante, têm tratamento semelhante.

O tratamento de erros sintáticos é mais complicado e depende do método de análise sintática utilizado. Um erro sintático deve ser detetado quando o próximo símbolo da cadeia de entrada não é um dos símbolos esperados. Não está clara, neste caso, a ação que o compilador deve tomar. Pode ser que o próximo símbolo esteja apenas incorretamente representado, como por exemplo, **ten** no lugar de

```

procedimento TERMO(var t;z);
início
  TESTE(códigos_que_iniciam_fator,z,"termo errado");
  se símbolo1 ∈ códigos_que_iniciam_fator
  então
    {FATOR(t,códigos_de_mult_div_conj ∪ z);
    enquanto símbolo1 ∈ código_de_mult_div_conj
    faça {s:=símbolo1;
    ANALEX;
    FATOR(t1,código_de_mult_div_conj ∪ z);
    caso s de
      código_de_multiplicação: {GERA(branco,"MULT");
      t2:=inteiro};
      código_de_divisão: {GERA(branco,"DIVI");
      t2:=inteiro};
      código_de_conjunção: {GERA(branco,"CONJ");
      t2:=booleano};
    fim do caso;
    se t≠t1 ou t≠t2 então MENSAGEM("tipo errado")} }
fim

```

Fig. 10.14

**then**, ou que esteja faltando um símbolo, ou então que esteja sobrando um símbolo. Pode acontecer que haja mais do que uma maneira de se corrigir o programa, dependendo da intenção do programador. Indicaremos a seguir um método bastante simples, mas representativo, para tratar erros sintáticos quando se utilizam analisadores descendentes recursivos.

Suponhamos que a rotina  $P$  do compilador (correspondente ao não-terminal  $\langle p \rangle$  da gramática) chama a rotina  $Q$  (correspondente a  $\langle q \rangle$ ). Seja  $s_1$  o conjunto de símbolos terminais que podem aparecer no início de uma cadeia derivada de  $\langle q \rangle$ , e seja  $s_2$  o conjunto de símbolos terminais que podem seguir uma cadeia derivada de  $\langle q \rangle$ , numa cadeia derivada de  $\langle p \rangle$ . Os conjuntos  $s_1$  e  $s_2$  podem ser determinados utilizando-se as técnicas da Seção 2.5. Por conveniência, utilizaremos também a rotina *TESTE* indicada na Figura 10.13. A rotina  $P$ , ao chamar a rotina  $Q$ , deve passar-lhe, como parâmetro, o conjunto  $s_2$  aumentado com os símbolos que foram passados à própria  $P$ . A rotina  $Q$  deve começar pela chamada *TESTE*( $s_1, s_2, mens$ ) onde *mens* é a mensagem que deve ser emitida caso não seja encontrada a construção derivada de  $\langle q \rangle$ . Se após o retorno de *TESTE*, o próximo símbolo pertence a  $s_1$ , então  $Q$  prossegue a compilação. Caso contrário o controle volta à rotina  $P$ , que pode prosseguir se o próximo símbolo pertencer a  $s_2$ , ou então pode voltar, por sua vez, à rotina que a chamou. A Figura 10.14 indica a rotina *TERMO* reescrita utilizando a idéia exposta acima. (*TERMO* corresponde à rotina  $Q$ .)

Alguns refinamentos desse método, bem como outras técnicas, podem ser encontrados na bibliografia indicada no fim do capítulo.

#### Exercícios

1. Discuta a organização de um compilador que utiliza um método ascendente de análise sintática (por exemplo, LR(1)).
2. Discuta a implementação de um compilador de dois passos. Estabeleça uma representação conveniente para o programa após o primeiro passo (por exemplo, uma árvore). Quais são as vantagens e desvantagens deste tipo de implementação. (Sugestão: Considere aspectos tais como as características da linguagem compilada, o método de análise sintática, a capacidade do sistema de computação utilizado, etc.)
3. Discuta os problemas de compilação separada de procedimentos. Note que o compilador deve continuar a verificação da compatibilidade entre a declaração e a chamada de procedimentos.
4. Em Algol 60, os rótulos não são declarados, e os identificadores podem ser usados antes de serem declarados. Discuta os problemas que isto acarreta e as possíveis soluções.
5. (Projeto) Implemente a versão básica do nosso Pascal simplificado, isto é, a discutida no Capítulo 8. Inclua, opcionalmente, os mecanismos discutidos no Capítulo 9: procedimentos como parâmetros e matrizes com limites estáticos.

#### NOTAS BIBLIOGRÁFICAS

A maioria dos textos dedicados à compilação cobre o material deste capítulo. Várias técnicas para implementar tabelas de símbolos podem ser encontradas em textos sobre estruturas de dados. Recomendamos, em particular, Knuth (1968 e 1973), Wirth (1976), Standish (1980), Horowitz e Sahni (1977). O tratamento de erros tem uma bibliografia bastante extensa; Silva (1981) é um estudo comparativo de vários métodos de recuperação para analisadores descendentes, além de conter uma lista de referências. O tratamento exposto na Seção 10.4 segue Wirth (1976).

# Sintaxe do Pascal Simplificado

1.  $\langle \text{programa} \rangle ::=$   
 $\text{program } \langle \text{identificador} \rangle \langle \text{lista de identificadores} \rangle;$   
 $\quad \langle \text{bloco} \rangle.$
2.  $\langle \text{bloco} \rangle ::=$   
 $\quad [\langle \text{parte de declarações de rótulos} \rangle]$   
 $\quad [\langle \text{parte de definições de tipos} \rangle]$   
 $\quad [\langle \text{parte de declarações de variáveis} \rangle]$   
 $\quad [\langle \text{parte de declarações de sub-rotinas} \rangle]$   
 $\quad \langle \text{comando composto} \rangle$
3.  $\langle \text{parte de declarações de rótulos} \rangle ::=$   
 $\quad \text{label } \langle \text{número} \rangle \{, \langle \text{número} \rangle\};$
4.  $\langle \text{parte de definições de tipos} \rangle ::=$   
 $\quad \text{type } \langle \text{definição de tipo} \rangle \{; \langle \text{definição de tipo} \rangle\};$
5.  $\langle \text{definição de tipo} \rangle ::=$   
 $\quad \langle \text{identificador} \rangle = \langle \text{tipo} \rangle$
6.  $\langle \text{tipo} \rangle ::=$   
 $\quad \langle \text{identificador} \rangle |$   
 $\quad \text{array } [\langle \text{índice} \rangle \{, \langle \text{índice} \rangle\}] \text{ of } \langle \text{tipo} \rangle$
7.  $\langle \text{índice} \rangle ::= \langle \text{número} \rangle .. \langle \text{número} \rangle$
8.  $\langle \text{parte de declarações de variáveis} \rangle ::=$   
 $\quad \text{var } \langle \text{declaração de variáveis} \rangle$   
 $\quad \quad \{; \langle \text{declaração de variáveis} \rangle\};$
9.  $\langle \text{declaração de variáveis} \rangle ::=$   
 $\quad \langle \text{lista de identificadores} \rangle : \langle \text{tipo} \rangle$
10.  $\langle \text{lista de identificadores} \rangle ::=$   
 $\quad \langle \text{identificador} \rangle \{, \langle \text{identificador} \rangle\}$
11.  $\langle \text{parte de declarações de sub-rotinas} \rangle ::=$   
 $\quad \{ \langle \text{declaração de procedimento} \rangle ; |$   
 $\quad \langle \text{declaração de função} \rangle ; \}$

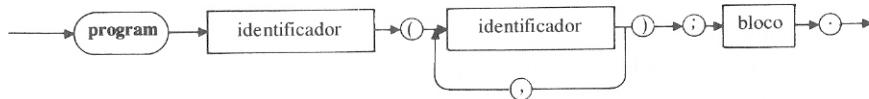
12. <declaração de procedimento> ::=  
     **procedure** <identificador>  
         [<parâmetros formais>]; <bloco>  
 13. <declaração de função> ::=  
     **function** <identificador>  
         [<parâmetros formais>]: <identificador>;  
         <bloco>  
 14. <parâmetros formais> ::=  
     (<seção de parâmetros formais>  
         {; <seção de parâmetros formais>})  
 15. <seção de parâmetros formais> ::=  
     [**var**] <lista de identificadores>; <identificador>  
     | **function** <lista de identificadores>; <identificador> (\*)  
     | **procedure** <lista de identificadores> (\*)  
 16. <comando composto> ::=  
     **begin** <comando>{;<comando>} **end**  
 17. <comando> ::=  
     [<número>:] <comando sem rótulo>  
 18. <comando sem rótulo> ::=  
     <atribuição>  
     | <chamada de procedimento>  
     | <desvio>  
     | <comando composto>  
     | <comando condicional>  
     | <comando repetitivo>  
 19. <atribuição> ::=  
     <variável> := <expressão>  
 20. <chamada de procedimento> ::=  
     <identificador> [<lista de expressões>]  
 21. <desvios> ::= **goto** <número>  
 22. <comando condicional> ::=  
     **if** <expressão> **then** <comando sem rótulo>  
         [**else** <comando sem rótulo>]  
 23. <comando repetitivo> ::=  
     **while** <expressão> **do** <comando sem rótulo>  
 24. <lista de expressões> ::=  
     <expressão> {, <expressão>}  
 25. <expressão> ::=  
     <expressão simples>[<relação><expressão simples>]  
 26. <relação> ::= = | <> | <|<=|=|>=|>  
 27. <expressão simples> ::=  
     [+|-] <termo>{(+|-|or) <termo>}  
 28. <termo> ::=  
     <fator> {(\*|div|and) <fator>}

29. <fator> ::=  
     <variável>  
     | <número>  
     | <chamada de função>  
     | (<expressão>)  
     | **not** <fator>  
 30. <variável> ::=  
     <identificador>  
     | <identificador> [<lista de expressões>] (\*)  
 31. <chamada de função> ::=  
     <identificador> [<lista de expressões>]  
 32. <número> ::=  
     <dígito>{<dígito>}  
 33. <dígito> ::=  
     0|1|2|3|4|5|6|7|8|9  
 34. <identificador> ::=  
     <letra> { <letra>|<dígito> }  
 35. <letra> ::=  
     a | b | c | d | e | f | g | h | i | j | k | l | m |  
     n | o | p | q | r | s | t | u | v | w | x | y | z

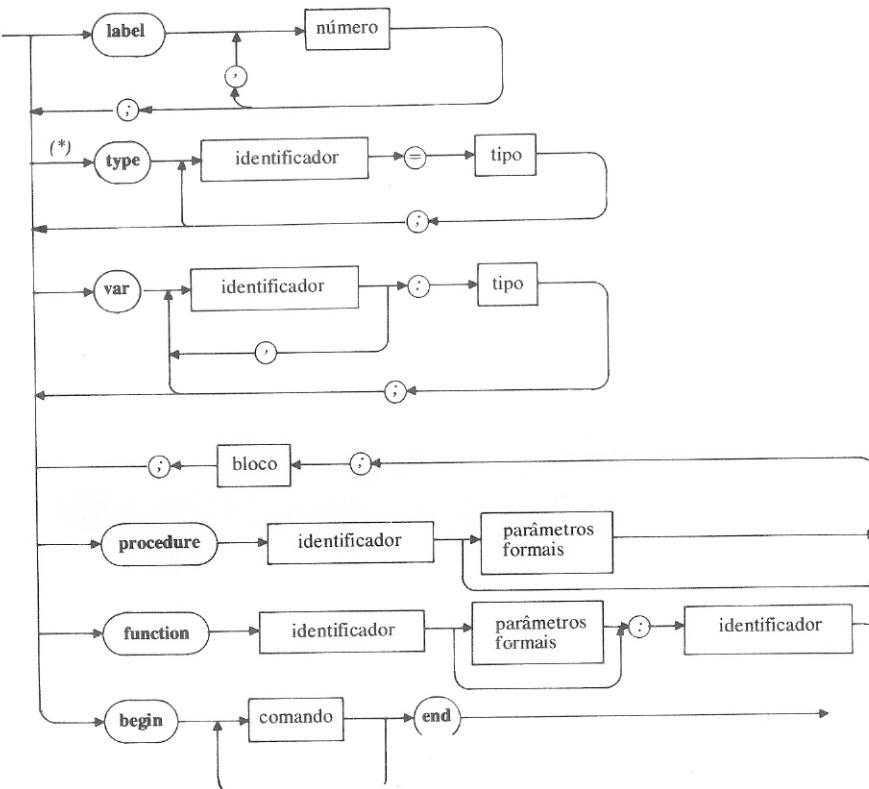
**APÊNDICE  
2**

## Diagramas Sintáticos do Pascal Simplificado

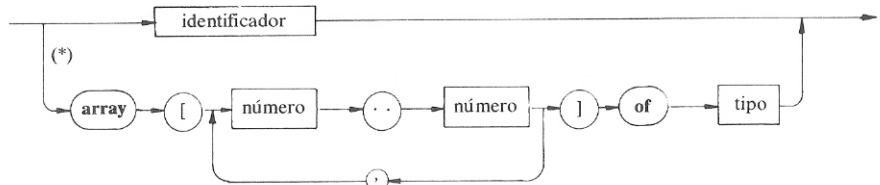
programa:



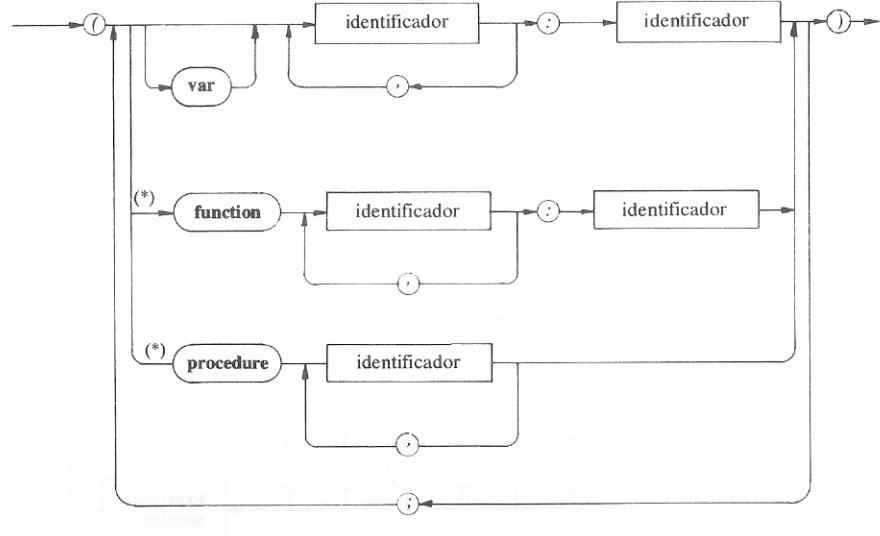
bloco:



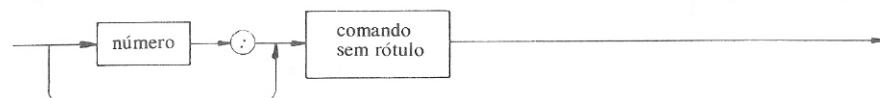
tipo:



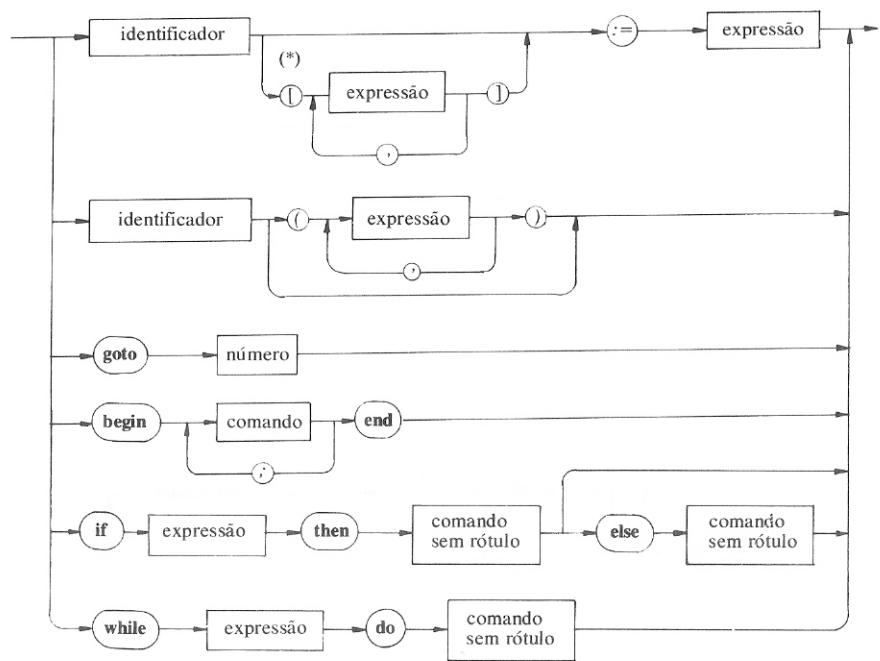
parâmetros formais:



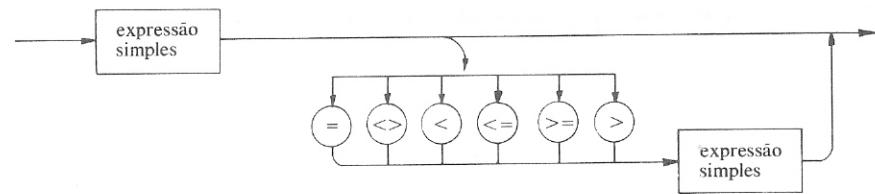
comando:



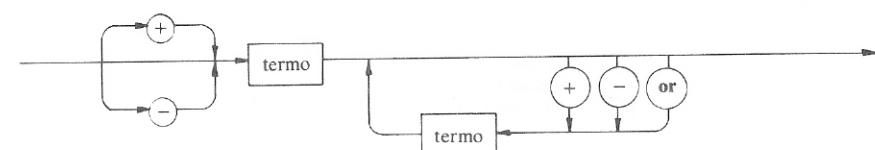
comando sem rótulo:



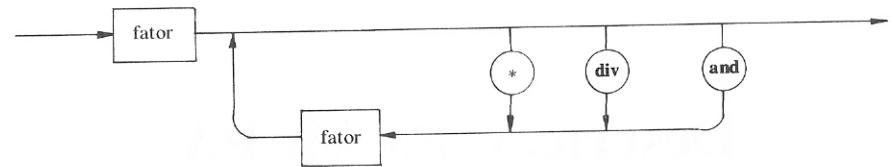
expressão:



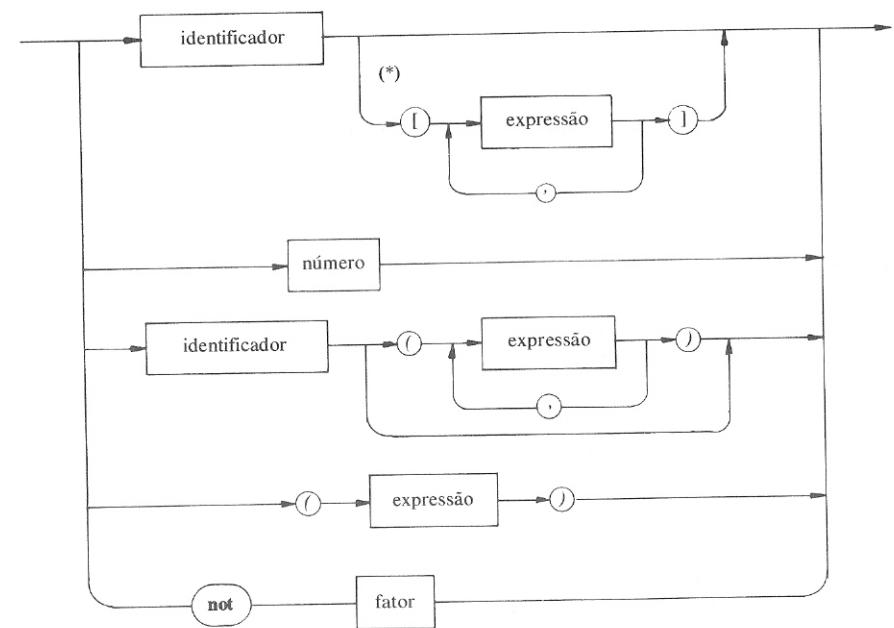
expressão simples:



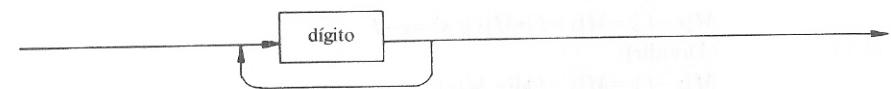
termo:



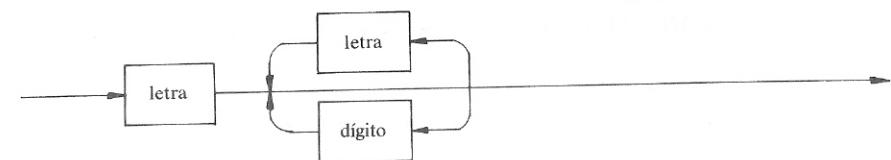
fator:



número:



identificador,



# Descrição da MEPA

A. Instruções comuns às versões básica e estendida

<i>CRCT</i> <i>k</i>	(Carregar constante): $s := s + 1; M[s] := k$
<i>CRVL</i> <i>m,n</i>	(Carregar valor): $s := s + 1; M[s] := M[D[m] + n]$
<i>CREN</i> <i>m,n</i>	(Carregar endereço): $s := s + 1; M[s] := D[m] + n$
<i>ARMZ</i> <i>m,n</i>	(Armazenar valor): $M[D[m] + n] := M[s]; s := s - 1$
<i>CRVI</i> <i>m,n</i>	(Carregar valor indiretamente): $s := s + 1; M[s] := M[M[D[m] + n]]$
<i>ARMI</i> <i>m,n</i>	(Armazenar indiretamente): $M[M[D[m] + n]] := M[s]; s := s - 1$
<i>SOMA</i>	(Somar): $M[s - 1] := M[s - 1] + M[s]; s := s - 1$
<i>SUBT</i>	(Subtrair): $M[s - 1] := M[s - 1] - M[s]; s := s - 1$
<i>MULT</i>	(Multiplicar): $M[s - 1] := M[s - 1] * M[s]; s := s - 1$
<i>DIVI</i>	(Dividir): $M[s - 1] := M[s - 1] \text{div } M[s]; s := s - 1$
<i>INVR</i>	(Inverter sinal): $M[s] := -M[s]$
<i>CONJ</i>	(Conjunção): $\text{se } M[s - 1] = 1 \text{ e } M[s] = 1 \text{ então } M[s - 1] := 1 \text{ senão } M[s - 1] := 0;$ $s := s - 1$
<i>DISJ</i>	(Disjunção): $\text{se } M[s - 1] = 1 \text{ ou } M[s] = 1 \text{ então } M[s - 1] := 1 \text{ senão } M[s - 1] := 0;$ $s := s - 1$

<i>NEGA</i>	(Negação): $M[s] := 1 - M[s]$
<i>CMME</i>	(Comparar menor): $\text{se } M[s - 1] < M[s] \text{ então } M[s - 1] := 1 \text{ senão } M[s - 1] := 0;$ $s := s - 1$
<i>CMMA</i>	(Comparar maior): $\text{se } M[s - 1] > M[s] \text{ então } M[s - 1] := 1 \text{ senão } M[s - 1] := 0;$ $s := s - 1$
<i>CMIG</i>	(Comparar igual): $\text{se } M[s - 1] = M[s] \text{ então } M[s - 1] := 1 \text{ senão } M[s - 1] := 0;$ $s := s - 1$
<i>CMDG</i>	(Comparar desigual): $\text{se } M[s - 1] \neq M[s] \text{ então } M[s - 1] := 1 \text{ senão } M[s - 1] := 0;$ $s := s - 1$
<i>CMEG</i>	(Comparar menor ou igual): $\text{se } M[s - 1] \leq M[s] \text{ então } M[s - 1] := 1 \text{ senão } M[s - 1] := 0;$ $s := s - 1$
<i>CMAG</i>	(Comparar maior ou igual): $\text{se } M[s - 1] \geq M[s] \text{ então } M[s - 1] := 1 \text{ senão } M[s - 1] := 0;$ $s := s - 1$
<i>DSVS</i> <i>p</i>	(Desviar sempre): $i := p$
<i>DSVF</i> <i>p</i>	(Desviar se falso): $\text{se } M[s] = 0 \text{ então } i := p \text{ senão } i := i + 1;$ $s := s - 1$
<i>NADA</i>	(Nada):
<i>PARA</i>	(Parar): “Pára a execução da MEPA”
<i>LEIT</i>	(Leitura): $s := s + 1; M[s] := \text{“próximo valor no arquivo de entrada”}$
<i>IMPR</i>	(Impressão): “Imprimir $M[s]$ ”; $s := s - 1$
<i>AMEM</i> <i>n</i>	(Alocar memória): $s := s + n$
<i>DMEM</i> <i>n</i>	(Desalocar memória): $s := s - n$
<i>INPP</i>	(Iniciar programa principal): $s := -1; D[0] := 0$
<i>ENRT</i> <i>j,n</i>	(Entrar no rótulo): $s := D[j] + n - 1$

B. Instruções da versão básica (veja Capítulo 8)

<i>CHPR</i> <i>p,m</i>	(Chamar procedimento): $M[s + 1] := i + 1; M[s + 2] := m;$ $s := s + 2; i := p$
------------------------	---

<i>ENPR</i> <i>k</i>	(Entrada no procedimento): <i>s</i> := <i>s</i> + 1; <i>M</i> [ <i>s</i> ] := <i>D</i> [ <i>k</i> ]; <i>D</i> [ <i>k</i> ] := <i>s</i> + 1;
<i>RTPR</i> <i>k,n</i>	(Retornar de procedimento): <i>D</i> [ <i>k</i> ] := <i>M</i> [ <i>s</i> ]; <i>i</i> := <i>M</i> [ <i>s</i> - 2]; <i>s</i> := <i>s</i> - ( <i>n</i> + 3)
<i>DSVR</i> <i>p,j,k</i>	(Desviar para rótulo): <i>temp1</i> := <i>k</i> ; <b>enquanto</b> <i>temp1</i> ≠ <i>j</i> <b>faça</b> { <i>temp2</i> := <i>M</i> [ <i>D</i> [ <i>temp1</i> ] - 2]; <i>D</i> [ <i>temp1</i> ] := <i>M</i> [ <i>D</i> [ <i>temp1</i> ] - 1]; <i>temp1</i> := <i>temp2</i> }; <i>i</i> := <i>p</i>
C. Instruções da versão estendida (veja Capítulo 9)	
<i>CREG</i> <i>p,k</i>	(Carregar endereço generalizado): <i>M</i> [ <i>s</i> + 1] := <i>p</i> ; <i>M</i> [ <i>s</i> + 2] := <i>D</i> [ <i>k</i> ]; <i>M</i> [ <i>s</i> + 3] := <i>k</i> ; <i>s</i> := <i>s</i> + 3
<i>CHPR</i> <i>p,k</i>	(Chamar procedimento): <i>M</i> [ <i>s</i> + 1] := <i>i</i> + 1; <i>M</i> [ <i>s</i> + 2] := <i>D</i> [ <i>k</i> ]; <i>M</i> [ <i>s</i> + 3] := <i>k</i> ; <i>s</i> := <i>s</i> + 3; <i>i</i> := <i>p</i>
<i>CHPP</i> <i>m,n,k</i>	(Chamar procedimento que é parâmetro): <i>M</i> [ <i>s</i> + 1] := <i>i</i> + 1; <i>M</i> [ <i>s</i> + 2] := <i>D</i> [ <i>k</i> ]; <i>M</i> [ <i>s</i> + 3] := <i>k</i> ; <i>s</i> := <i>s</i> + 3; <i>i</i> := <i>M</i> [ <i>D</i> [ <i>m</i> ] + <i>n</i> ]; <i>temp</i> := <i>M</i> [ <i>D</i> [ <i>m</i> ] + <i>n</i> + 2]; <i>D</i> [ <i>temp</i> ] := <i>M</i> [ <i>D</i> [ <i>m</i> ] + <i>n</i> + 1]; <b>enquanto</b> <i>temp</i> ≥ 2 <b>faça</b> { <i>D</i> [ <i>temp</i> - 1] := <i>M</i> [ <i>D</i> [ <i>temp</i> ] - 1]; <i>temp</i> := <i>temp</i> - 1}
<i>ENPR</i> <i>k</i>	(Entrar no procedimento): <i>s</i> := <i>s</i> + 1; <i>M</i> [ <i>s</i> ] := <i>D</i> [ <i>k</i> - 1]; <i>D</i> [ <i>k</i> ] := <i>s</i> + 1
<i>RTPR</i> <i>n</i>	(Retornar de procedimento): <i>temp</i> := <i>M</i> [ <i>s</i> - 1]; <i>D</i> [ <i>temp</i> ] := <i>M</i> [ <i>s</i> - 2]; <i>i</i> := <i>M</i> [ <i>s</i> - 3]; <i>s</i> := <i>s</i> - ( <i>n</i> + 4); <b>enquanto</b> <i>temp</i> ≥ 2 <b>faça</b> { <i>D</i> [ <i>temp</i> - 1] := <i>M</i> [ <i>D</i> [ <i>temp</i> ] - 1]; <i>temp</i> := <i>temp</i> - 1}

## Bibliografia

Algumas das referências que constam desta lista não foram citadas no texto.

- Abramson (1973) H. Abramson, *Theory and Application of a Bottom-up Syntax-directed Translator*, Academic Press.
- Aho e Ullman (1972) A. V. Aho e J. D. Ullman, *The Theory of Parsing, Translation and Compiling, vol. 1: Parsing*, Prentice-Hall.
- Aho e Ullman (1973) A. V. Aho e J. D. Ullman, *The Theory of Parsing, Translation and Compiling, vol. 2: Compiling*, Prentice-Hall.
- Aho e Ullman (1977) A. V. Aho e J. D. Ullman, *Principles of Compiler Design*, Addison-Wesley.
- Amman (1978) U. Amman, *Run Time Storage Organization*, Berichte des Instituts für Informatik Nr. 25, ETH Zürich.
- Anderson, Eve e Horning (1973) T. Anderson, J. Eve e J. J. Horning, Efficient LR(1) Parsers, *Acta Informatica* 2 (12-39).
- Backhouse (1979) R. C. Backhouse, *Syntax of Programming Languages: Theory and Practice*, Prentice-Hall.
- Barrett e Couch (1979) W. Barrett e J. D. Couch, *Compiler Construction: Theory and Practice*, Science Research Associates.
- Bauer e Eickel (1974) F. L. Bauer e J. Eickel (editores), *Compiler Construction: An Advanced Course*, Springer Verlag.
- Berry (1978) R. E. Berry, Experience with the Pascal P-Compiler, *Software: Practice and Experience* 8 (617-627).
- Bobrow e Wegbreit (1973) D. G. Bobrow e B. Wegbreit, A Model and Stack Implementation of Multiple Environments, *Communications of the ACM* 16 (591-603).
- Cheatham (1967) T. E. Cheatham Jr., *The Theory and Construction of Compilers*, Computer Associates.

- Chomsky (1956)  
 N. Chomsky, Three Models for the Description of Language, *IEEE Transactions on Information Theory* 2 (113-124).
- Chomsky (1959)  
 N. Chomsky, On Certain Formal Properties of Grammars, *Information and Control* 2 (137-167).
- DeRemer (1971)  
 F. L. DeRemer, Simple LR(k) Grammars, *Communications of the ACM* 14 (453-460).
- Floyd (1963)  
 R. Floyd, Syntactic Analysis and Operator Precedence, *Journal of the ACM* 10 (316-333).
- Ginsburg (1966)  
 S. Ginsburg, *The Mathematical Theory of Context-free Languages*, McGraw-Hill.
- Grau, Hill e Langmaack (1967)  
 A. A. Grau, U. Hill e H. Langmaack, *Translation of Algol 60*, Springer Verlag.
- Gries (1971)  
 D. Gries, *Compiler Construction for Digital Computers*, J. Wiley & Sons.
- Griswold (1972)  
 R. E. Griswold, *The Macro Implementation of SNOBOL4*, W. H. Freeman.
- Harrison (1978)  
 M. A. Harrison, *Introduction to Formal Language Theory*, Addison-Wesley.
- Hopcroft e Ullman (1969)  
 J. E. Hopcroft e J. D. Ullman, *Formal Languages and Their Relation to Automata*, Addison-Wesley.
- Hopgood (1969)  
 F. R. A. Hopgood, *Compiling Techniques*, American Elsevier.
- Horowitz e Sahni (1977)  
 E. Horowitz e S. Sahni, *Fundamentals of Data Structures*, Computer Science Press.
- Ingerman (1966)  
 P. Z. Ingerman, *A Syntax Oriented Translator*, Academic Press.
- Jensen e Wirth (1974)  
 K. Jensen e N. Wirth, *Pascal: User Manual and Report*, Springer Verlag.
- Johnston (1971)  
 J. B. Johnston, The Contour Model of Block Structured Processes, Proceedings of a Symposium on Data Structures in Programming Languages, *SIGPLAN Notices* 6, 2 (55-82).
- Joy (1979)  
 W. N. Joy, *Berkeley Pascal PX Implementation Notes*, Computer Science Division, University of California at Berkeley.
- Knuth (1965)  
 D. E. Knuth, On the Translation of Languages from Left to Right, *Information and Control* 8 (607-639).
- Knuth (1968)  
 D. E. Knuth, *The Art of Computer Programming, vol. 1: Fundamental Algorithms*, Addison-Wesley.
- Knuth (1973)  
 D. E. Knuth, *The Art of Computer Programming, vol. 3: Sorting and Searching*, Addison-Wesley.
- Kowaltowski (1981)  
 T. Kowaltowski, Parameter Passing Mechanisms and Run Time Data Structures, *Software: Practice and Experience* 11 (757-765).
- Lee (1974)  
 J. A. N. Lee, *Anatomy of a Compiler*, Van Nostrand.
- Lewis, Rosenkrantz e Stearns (1976)  
 P. M. Lewis II, D. J. Rosenkrantz e R. E. Stearns, *Compiler Design Theory*, Addison-Wesley.
- Lewis e Stearns (1968)  
 P. M. Lewis II e R. E. Stearns, Syntax Directed Transductions, *Journal of the ACM* 15 (465-488).
- McKeeman, Horning e Wortman (1970)  
 W. M. McKeeman, J. J. Horning e D. B. Wortman, *A Compiler Generator*, Prentice-Hall.
- Mello (1980)  
 R. D. B. P. Mello Filho, *Proposta de um Sistema de Execução para a Linguagem de Programação Ada*, IMECC-UNICAMP (dissertação de mestrado).
- Morris (1970)  
 J. H. Morris Jr., *Design of a Run Time System for Algol 60*, Department of Computer Science, University of Califórnia at Berkeley.
- Moura (1982)  
 A. V. Moura, *Uma Introdução a Técnicas de Análise Sintática*, Terceira Escola de Computação, Rio de Janeiro.
- Naur (1963)  
 P. Naur (editor), Revised Report on the Algorithmic Language Algol 60, *Communications of the ACM* 6 (1-17).
- Nori, Amman, Jensen e Nägeli (1975)  
 K. V. Nori, U. Amman, K. Jensen e H. H. Nägeli, *The Pascal <P> Compiler: Implementation Notes*, Berichte des Instituts für Informatik Nr. 10, ETH Zürich.
- Organick (1973)  
 E. I. Organick, *Computer System Organization: The B5700/B6700 Series*, Academic Press.
- Pasko (1973)  
 H. J. Pasko, *A Pseudo Machine for Code Generation*, Department of Computer Science, University of Toronto (dissertação de mestrado).
- Paul (1962)  
 M. Paul, *Zur Struktur Formaler Sprachen*, Universität Mainz (dissertação D77).
- Pollack (1972)  
 B. W. Pollack (editor), *Compiler Techniques*, Auerbach.
- Pratt (1975)  
 T. W. Pratt, *Programming Languages: Design and Implementation*, Prentice-Hall.
- Randell e Russell (1964)  
 B. Randell e L. J. Russell, *Algol 60 Implementation*, Academic Press.
- Rosenkrantz e Stearns (1970)  
 D. J. Rosenkrantz e R. E. Stearns, Properties of Deterministic Top-Down Grammars, *Information and Control* 17 (226-256).
- Salomaa (1973)  
 A. Salomaa, *Formal Languages*, Academic Press.
- Sanches (1979)  
 M. M. Sanches, *Portabilidade de Compiladores*, IME-USP (dissertação de mestrado).
- Setzer e Melo (1981)  
 V. W. Setzer e I. S. H. de Melo, *A Construção de um Compilador*, Segunda

Escola de Computação, Campinas.

Silva (1981)

H. V. R. C. Silva, *Recuperação de Erros em Analisadores Sintáticos Descendentes*, IMECC-UNICAMP (dissertação de mestrado).

Simon (1981)

Imre Simon, *Linguagens Formais e Autômatos*, Segunda Escola de Computação, Campinas.

Standish (1980)

T. A. Standish, *Data Structure Techniques*, Addison-Wesley.

Warshall (1962)

S. Warshall, A Theorem on Boolean Matrices, *Journal of the ACM* 9 (11-12).

Wegner (1980)

P. Wegner, *Programming with Ada: An Introduction by Means of Graduated Examples*, Prentice-Hall.

Wirth (1971a)

N. Wirth, The Programming Language Pascal, *Acta Informatica* 1 (35-63).

Wirth (1971b)

N. Wirth, The Design of a Pascal Compiler, *Software: Practice and Experience* 1 (309-333).

Wirth (1976)

N. Wirth, *Algorithms + Data Structures = Programs*, Prentice-Hall.

Wirth e Weber (1966)

N. Wirth e H. Weber, Euler: A Generalization of Algol and Its Formal Definition, *Communications of the ACM* 9 (13-25, 89-99).

## Índice Alfabético

Os números em **negrito** referem-se a locais onde o assunto é abordado mais extensamente. Os números em *íntico* referem-se a localizações fora do texto (figuras). No caso de termos muito utilizados, aparecem apenas as referências mais importantes.

### A

Acumulador, 257  
Ada, 92, 160  
Alfabeto, 8  
Algol, 60, 1, 3, 21, 92, 137, 139, 150, 153, 155, 159  
Ambigüidade, 11-16, 18, 19, 26, 35, 48, 49, 69, 73  
AMEM, 104, 106, 109, 116, 121, 122, 125, 136, 154, 155, 170, 181  
ANALEX, 165, 165, 166-171  
Analizador de deslocamento ou redução, 39  
ANALISADOR LÉXICO, 79  
Análise, 3, 4  
- léxica, 3, 4, 75, **76-80**, 165, 170  
- semântica, 3, 4  
- sintática, 2, 3, 4, 6, 11, **19-20**, 22-70, 77, 80, 161, **165-170**, 172  
- - ascendente, 19, **22-50**  
- - descendente, 19, **51-70**, 73, 75  
- - - recursiva, 19, **62-66**, 69, 75, 161, 165, 172  
ANÁLISE DE PRECEDÊNCIA SIMPLES, 34  
ANÁLISE DESCENDENTE, 62  
ANALISE LR, 40  
Apontador, 71, 160  
- da matriz, 155  
- de trabalho, 156  
- dinâmico, 144  
- estático, 140, 144  
ARMI, 122, 167, 180  
ARMZ, 98, 105, 109, 120, 158, 167, 180  
Arquivo  
- de entrada, 73, 102  
- de saída, 73, 102  
Árvore  
- de derivação, 11-16, 19, 20, 22, 24, 25, 27, 31, 39, 51, 52, 60, 62  
- sintática, 3, 12  
Aspecto  
- pragmático, 2  
- semântico, 2, 161

- sintático, 2, 161  
Átomo, **77-80**  
ATRIBUIÇÃO, 167  
Atributo, 161-164, 165  
ATUALIZA TIPO, 170  
Autômato finito, 78, 80

### B

Bloco, 72, 92, 137, 153-154, 159, 168, 169  
BLOCO, 168, 169  
BUSCA, 164, 166-170

### C

Cadeia, 8  
- dinâmica, 144  
- estática, 86, 140, 142, 148, 159  
- nula, 8  
Carta sintática, **66-68**, 170  
Categoria, 161, 164, 164, 165  
Chamada  
- de função, 95, 98, **125-128**, 166, 167  
- de procedimento, 73, **81-92**, 98, **105-125**, 136, **139-153**, 159

CHAMADA DE PROCEDIMENTO, 167

CHPP, 144, 182  
CHPR, 107, 109, 114, 122, 136, 137, 181, 182  
CMAG, 96, 181  
CMDG, 96, 181  
CMEG, 96, 181  
CMIG, 96, 181  
CMMA, 96, 181  
CMME, 96, 181  
Código-objeto, 6  
Comando, 73

- composto, 105  
- condicional, **98-102**  
- de atribuição, **96-98**  
- de desvio, 98, **128-134**, 136, 139, 148

- de entrada, **102-103**  
- de saída, **102-103**  
- iterativo ou repetitivo, 73, **98-102**, 135

- nulo, 73  
Descriptor da matriz, 155

- seleção, 135  
**COMANDO**, 168  
**COMANDO COMPOSTO**, 168, 169  
**COMANDO CONDICIONAL**, 167, 168, 168  
**COMANDO REPETITIVO**, 167  
**COMANDO SEM RÓTU-LO**, 167, 168  
Comentário, 74, 78, 79

Compilador, 1, 4  
Comprimento de cadeia, 8  
Concatenação, 8, 9

**CONJ**, 96, 180  
Conjunto fechado, 43, 44  
Contexto de execução, 143

Co-rotina, 160  
**CRCT**, 95, 158, 166, 180  
**CREG**, 145, 159, 182  
**CREN**, 122, 180  
**CRVI**, 122, 166, 180  
**CRVL**, 95, 105, 120, 125, 158, 180

### D

Declaração, 72  
- de arquivo, 71  
- de constante, 71  
- de procedimento, 81, 161  
- de rótulo, 172  
- de variável, 104, 161, 168  
- local, **153-154**

**DECLARAÇÃO DE FUN-ÇÃO**, 169  
**DÉCLARAÇÃO DE PROCE-DIMENTO**, 168, 169  
**DECLARAÇÃO DE RÓTU-LOS**, 169

**DECLARAÇÕES DE VARIÁ-VEIS**, 168, 169, 170  
Definições inductivas, 6-8

Depuração, 159  
Derivação, **10-16**  
- canônica, 11-15, 19

- direita, 15, 20, 22, 24  
- direta, 10  
- esquerda, 15, 20, 52, 56  
- não trivial, 10

Descriptor da matriz, 155

Deslocamento, 117, 119, 123, 125, 131, 145, 155, 157, 164  
**DESVIO**, 167  
Diagrama  
- de execução, 81-92, 93, 138  
- sintático, 66-68, 70, 71, 176  
**DISJ**, 96, 180  
**DIVI**, 95, 165, 171, 180  
**DMEM**, 107, 109, 116, 121, 136, 154, 155, 168, 169, 181  
**DSVF**, 98, 168, 181  
**DSVR**, 130, 131, 136, 137, 182  
**DSVS**, 98, 108, 130, 132, 148, 168, 169, 181

## E

**ELIMINA**, 164, 169  
else pendente, 15  
Endereço, 94, 112, 116, 117, 122, 153  
- de retorno, 107, 114, 129, 141  
- fixo, 105, 114, 117, 121, 137  
- generalizado, 142-145  
- - de retorno, 142, 144, 145  
- indireto, 122, 157  
- léxico ou textual, 117, 144, 154, 155, 161, 164  
- relativo, 117, 137  
**ENPR**, 122, 136, 144, 169, 182  
**ENRT**, 131, 132, 136, 137, 148, 155, 156, 168, 181  
Erro  
- semântico, 170, 171  
- sintático, 170-172  
**ERRO**, 31, 34, 40, 40, 62, 62, 64, 65, 79, 165-170  
Escopo, 81, 86, 118, 137, 139, 148, 161, 164  
Espalhamento, 164  
Estados, 39-48  
Estratificação, 31  
Estrutura de programa, 81  
Expressão, 73, 74, 94-96, 125, 153, 155, 165  
**EXPRESSÃO**, 166, 167, 168

## F

**FATOR**, 165, 166, 171  
Fecho, 9, 17  
- positivo, 9  
- transitivo, 9  
- transitivo reflexivo, 9  
fecho, 43, 44  
Flecha estática, 86, 106, 109, 118  
Forma Normal de Backus ou **FNB**, 6-8, 10, 19, 21, 69, 71, 170  
Forma sentencial, 10, 12, 24, 25, 49  
**FORTAN**, 1, 2, 76  
**forward**, 137  
Frase, 24, 34  
- prima, 34, 35, 39, 49  
- simples, 24, 48  
Função, 125-128, 131, 136, 168  
- como parâmetro, 139-150  
- fictícia, 150, 153, 159

## G

**GERA**, 165, 165-171  
Geração de código, 3, 4, 157, 165-170  
Gramática, 8-19  
- de operadores, 34, 35, 49  
- de precedência simples, 26  
- livre de contexto, 2, 8-11, 19, 20, 76  
- LL, 56  
- reduzida, 16, 18-19, 54

## I

Identificador, 11, 72, 74, 76-79, 161-164, 164, 165, 171, 172  
**IMPR**, 102, 181  
Indecidível, 13  
Indentação, 78  
Indexador, 157  
**INPP**, 104, 123, 145, 181  
**INSERE**, 164, 169, 170  
Interpretador, 1  
**INVR**, 95, 180  
**IPVL**, 115  
Item, 43-47  
- completo, 43, 45

## L

**LARL**, 48, 50  
**LEIT**, 102, 181  
Linguagem, 8-11  
- de alto nível, 1-5  
- de máquina, 1, 23, 93, 157, 158  
- de montagem, 1, 5, 93, 157  
- inerentemente ambígua, 14, 20  
Literal, 157  
**LL**, 56-57, 62, 69, 80  
**LR**, 19, 39-48, 49, 50, 80, 172

**M**

Macroinstrução, 157  
Macromontador, 157, 160  
Máquina hipotética, 93, 137, 157  
Matriz, 71, 139, 153, 154-156, 159, 164, 172  
- dinâmica, 139, 155, 159  
**MENSAGEM**, 171  
MEPA, 93-160, 165, 180  
Modelo de contorno, 92  
Montador, 1, 5, 157  
**MULTE**, 95, 165, 171, 180

## N

**NADA**, 98, 132, 158, 168, 169, 181  
**NEGA**, 96, 166, 181  
Nível léxico ou textual, 118-119, 122, 123, 129, 144, 145, 148, 154, 159, 164, 165  
Número, 74, 76-79

## O

Operador, 14, 34, 39  
- binário, 94

- unário, 95  
Otimização, 3, 78, 158, 160  
- global, 3, 4  
- local, 3, 4

## P

Palavra, 8  
- chave, 76-78  
- reservada, 76  
**PARA**, 104, 181  
Parâmetro, 81-92, 114-125, 137, 139-153, 155, 159  
- efetivo, 88, 114, 122, 125, 129, 137, 142, 145, 150, 153, 159  
- formal, 114, 117, 122, 124, 132, 137, 145, 150, 153, 164, 164, 169  
Pascal, 2, 5, 11, 66, 71-75, 76, 77, 80, 81-92, 93-138, 139-160, 161, 165, 172, 173-175, 176-179  
Passagem de parâmetros  
- cópia ou valor-resultado, 137  
- nome, 92, 139, 150-153, 159  
- referência ou variável, 88, 117-125, 137, 153, 155, 159, 164  
- valor, 84, 114-117, 122, 139, 153, 155, 159, 164  
Passos de compilação, 3, 5, 161, 172  
Pilha, 92, 94, 95, 98, 100, 103, 106, 107, 114, 117, 119, 121-123, 125, 128-130, 137, 144, 148, 154, 163  
- disciplina de, 106, 118, 119, 165  
- fenômeno de, 163  
**PL/I**, 76, 77  
Precedência, 14, 20, 34  
- de operadores, 19, 24, 34-39, 49, 50, 66  
- estendida, 34  
- fraca, 33  
- simples, 19, 25-34, 34, 48-50  
Procedimento, 81-92, 105-137, 139-153, 161, 163, 164  
- como parâmetro, 90, 139-150, 153, 159, 172  
- sem parâmetros, 83, 105-114  
Processo paralelo, 160  
Produção, 10-19, 77, 136  
Produto, 8, 9, 16, 20  
Programa, 72, 103-105  
- fonte, 1, 4, 81, 84, 93, 161, 170  
- objeto, 1, 4, 93, 103, 125, 132, 161  
- principal, 114, 117, 137, 145, 168  
**PRÓXIMO**, 31, 34, 40, 40, 62, 62, 64, 64, 65, 78, 79, 79  
**PRÓXIMO RÓTULO**, 168, 168, 169

## R

Raiz, 10  
Recursão, 88, 92, 94, 105, 109, 117, 137  
- direita, 31, 52

- esquerda, 30, 52, 54, 60  
Redeclaração, 163, 168, 171  
Redução, 22, 23, 34, 39, 40, 41, 45  
Redutendo, 24-26, 34, 39, 45  
Registrador  
- de base, 94, 117-125, 128, 130, 136, 137, 139-149, 157, 159

- de programa, 163, 165, 256  
- de topo da pilha, 94, 96, 98, 103, 125, 128, 131, 137, 157, 158  
Registro, 71, 160  
- de ativação, 82-92, 105-138, 139-156

Regra de transcrição, 10  
Relação, 9, 16-18  
- inversa, 9, 16  
Representação intermediária, 3  
Retrocesso, 23, 51-56, 69  
Rótulo, 72, 92, 128-135, 136, 137, 148, 159, 164, 172  
**RTPR**, 107, 115, 122, 123, 125, 128-130, 136, 137, 144, 155, 158, 159, 169, 182

## S

Semântica, 6, 81, 150, 153  
Sentença, 10  
Separação, 31  
Símbolo, 8, 76-77

- especial, 76-79  
- - composto, 76, 77  
- inicial, 10, 11, 12, 22  
- não-terminal, 10  
- terminal, 10

Simulação da execução, 81, 82-92, 125  
Sintaxe, 2, 6-21, 81, 154, 173  
Síntese, 3, 4  
Sistema  
- de execução, 81, 93-138, 139-160

- - implementação de, 93, 139, 157-159  
- de tempo-objeto, 93  
- hospedeiro, 2  
SLR, 47, 50  
SNOBOL, 2  
**SOMA**, 95, 109, 158, 158, 180

**SUBT**, 95, 180  
Superposição de código, 3

## T

Tabela  
- de símbolos, 3, 78, 157, 161-165, 169, 171, 172  
- em árvore, 163  
Tarefa concorrente, 92  
**TERMO**, 165, 171, 172  
**TESTE**, 171, 172  
Tipo, 71, 161, 164, 164  
*transfere*, 44-46

Tratamento de erros, 3, 4, 29, 31, 66, 78, 170-172

## V

Variável local permanente, 92  
Vetor de informação, 156  
Vocabulário, 8, 10, 76  
- não-terminal, 9  
- terminal, 9, 76