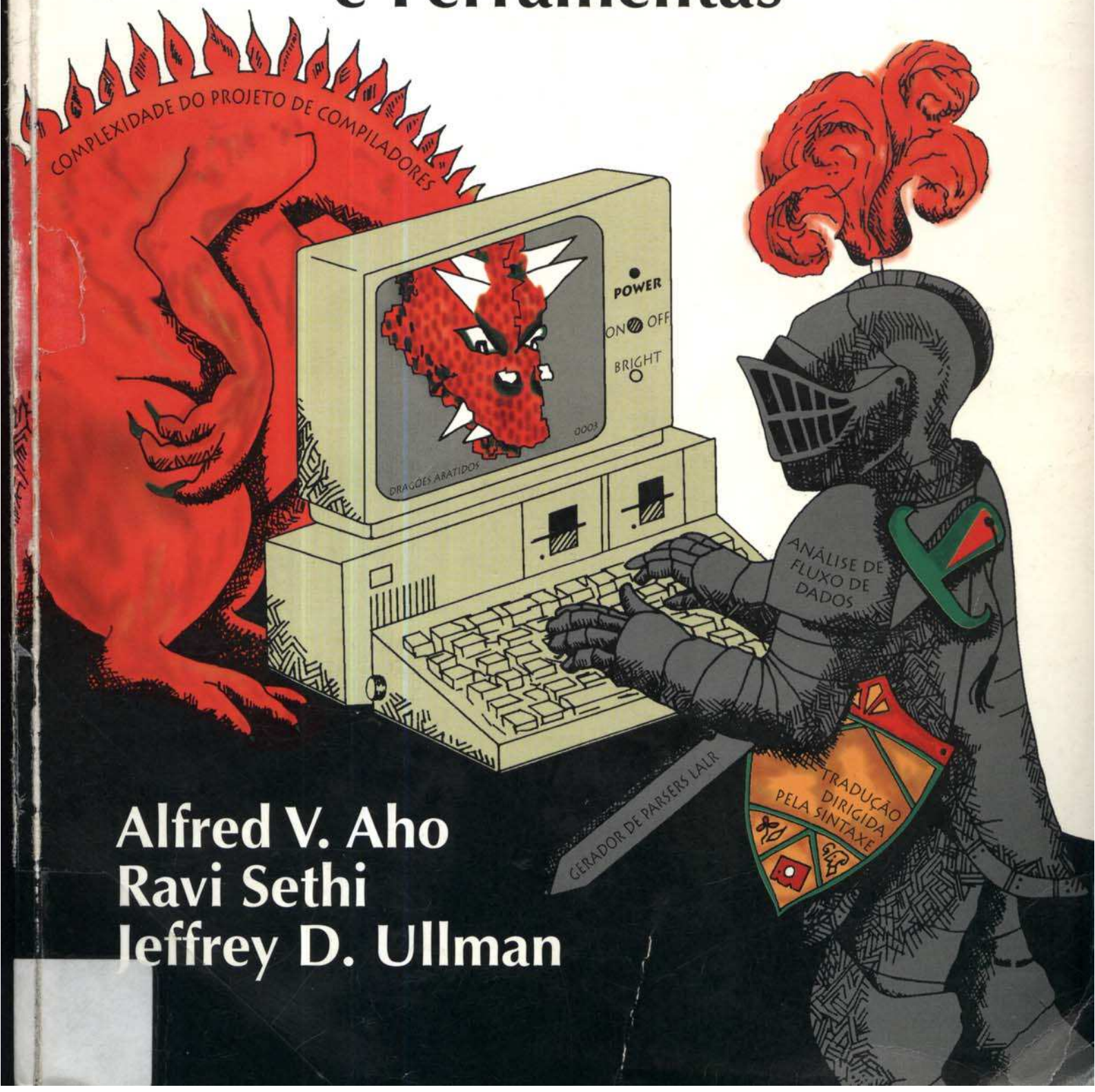
Compiladores

Princípios, Técnicas e Ferramentas



Alfred V. Aho
Ravi Sethi
Jeffrey D. Ullman

Cód 79848
60125330

COMPILADORES
Princípios, Técnicas e Ferramentas

ALFRED V. AHO
AT & T Bell Laboratories
Murray Hill, New Jersey

RAVI SETHI
AT & T Bell Laboratories
Murray Hill, New Jersey

JEFFREY D. ULLMAN
Stanford University
Stanford, California

Tradução

DANIEL DE ARIOSTO PINTO

Professor Auxiliar no Departamento de Informática e Ciência da Computação
do Instituto de Matemática e Estatística da UERJ, nas disciplinas: Compilação,
Linguagem Assembler e Estrutura de Linguagens de Programação
Tecnólogo em Processamento de Dados pela
Faculdade de Administração da Guanabara (FAG)
Coordenador do Suporte de Sistemas do Banco de Dados
e Comunicação de Dados do IBGE
Bacharel em Administração de Empresas pela FAG

LTC
EDITORA

198.267

005.453
A896c
c1995
v.2
ex.1

Compilers, Principles, Techniques and Tools
Copyright © 1986 by Bell Telephone Laboratories, Incorporated.
Publicado originalmente nos Estados Unidos por Addison-Wesley
Publishing Company, Reading, Massachusetts, 01867 USA

Direitos exclusivos para a língua portuguesa
Copyright © 1995 by
LTC — Livros Técnicos e Científicos Editora S.A.
Travessa do Ouvidor, 11
Rio de Janeiro, RJ — CEP 20040-040
Tel.: (21) 221-9621
Fax: (21) 221-3202

Reservados todos os direitos. É proibida a duplicação ou
reprodução deste volume, no todo ou em parte,
sob quaisquer formas ou por quaisquer meios
(eletrônico, mecânico, gravação, fotocópia ou outros),
sem permissão expressa da Editora.

Prefácio

Este livro descende de outro, *Principles of Compiler Design*, de autoria de Alfred V. Aho e Jeffrey D. Ullman. Como seu antecessor, está destinado a ser um livro-texto para um primeiro curso em projeto de compiladores. A ênfase está na solução de problemas universalmente encontrados no projeto de um tradutor de linguagem, independentemente da máquina-fonte ou alvo.

Embora poucas pessoas tenham a oportunidade de construir ou mesmo manter um compilador de uma grande linguagem de programação, o leitor pode aplicar com proveito as idéias e técnicas discutidas neste livro ao projeto geral de programas. Por exemplo, as técnicas de reconhecimento de cadeias de caracteres para a construção de analisadores léxicos têm sido usadas em editores de texto, sistemas de recuperação de informações e programas de reconhecimento de padrões. As linguagens livres de contexto e as definições dirigidas pela sintaxe têm sido utilizadas na construção de muitas pequenas linguagens, tais como a de composição tipográfica e desenho de figuras, que produziram o original em inglês deste livro. Já as técnicas de otimização têm sido aproveitadas em verificadores de programas e em programas que produzem programas "estruturados" a partir dos não-estruturados.

Uso do Livro

Os tópicos mais importantes no projeto de compiladores são abordados em profundidade. O primeiro capítulo introduz a estrutura básica de um compilador e é essencial para o restante do livro.

O Capítulo 2 apresenta um tradutor de expressões infixas para a forma posfixa, construído a partir do uso de algumas das técnicas básicas descritas neste livro. Muitos dos capítulos restantes abordam em maiores detalhes os assuntos tratados no Capítulo 2.

O Capítulo 3 aborda a análise léxica, expressões regulares, máquinas finitas de estado e ferramentas de geração de *scanners*. As informações contidas nesse capítulo são amplamente aplicáveis ao processamento de textos.

O Capítulo 4 trata, de forma abrangente, das técnicas mais importantes de análise sintática, indo dos métodos de descendência recursiva, que são adequados à implementação manual, às técnicas LR, mais completas do ponto de vista computacional, e que têm sido usadas nos geradores de *parsers*.

O Capítulo 5 introduz as idéias principais da tradução dirigida pela sintaxe. Esse capítulo é usado ao longo do livro, tanto para especificar quanto implementar traduções.

O Capítulo 6 apresenta as idéias principais para a realização da verificação estática e semântica de tipos. A verificação e a unificação de tipos são discutidas em detalhes.

O Capítulo 7 discute as organizações de memória usadas para dar suporte ao ambiente em tempo de execução de um programa.

O Capítulo 8 começa com uma discussão das linguagens intermediárias e mostra, então, como as construções comuns das linguagens de programação podem ser traduzidas em código intermediário.

O Capítulo 9 trata da geração do código-alvo. Estão incluídos os métodos básicos de geração de código "em vôo", bem como métodos ótimos para a geração de código para expressões. A otimização *peephole* e os geradores de geradores de código também são abordados neste capítulo.

O Capítulo 10 traz uma abordagem abrangente da otimização de código. Os métodos de análise de fluxo de dados aparecem bem detalhados, assim como os principais métodos de otimização global.

O Capítulo 11 discute alguns temas pragmáticos que emergem na implementação de um compilador. A engenharia e o teste de software são particularmente importantes na construção de compiladores.

O Capítulo 12 contém estudos de casos de compiladores construídos a partir de algumas das técnicas apresentadas neste livro.

O Apêndice A descreve uma linguagem simples, um "subconjunto" de Pascal, que pode ser usada como base de um projeto de implementação.

Utilizando o conteúdo deste livro, os autores ministraram cursos introdutórios e avançados, em níveis de graduação e pós-graduação, nos AT & T Bell Laboratories, e nas Universidades de Columbia, Princeton e Stanford.

Um curso introdutório em compiladores poderia compreender as seguintes seções deste livro:

introdução	Capítulo 1 e Seções 2.1-2.5
análise léxica	2.6, 3.1-3.4
tabelas de símbolos	2.7, 7.6
análise sintática	2.4, 4.1-4.4
tradução dirigida pela sintaxe	2.5, 5.1-5.5
verificação de tipos	6.1-6.2
organização em tempo de execução	7.1-7.3
geração de código intermediário	8.1-8.3
geração de código	9.1-9.4
otimização de código	10.1-10.2

As informações necessárias a um projeto de programação, como o do Apêndice A, são introduzidas no Capítulo 2.

Um curso enfatizando as ferramentas na construção de compiladores deveria incluir a discussão dos geradores de analisadores léxicos da Seção 3.5; dos geradores de analisadores sintáticos das Seções 4.8 e 4.9; dos geradores de geradores de código da Seção 9.12; e o material sobre as técnicas de construção de compiladores encontrado no Capítulo 11.

Um curso avançado deveria salientar os algoritmos usados nos geradores de analisadores léxicos e sintáticos discutidos nos Capítulos 3 e 4; a informação sobre a equivalência de tipos, sobrecarga, polimorfismo e unificação do Capítulo 6; o texto sobre organização de memória em tempo de execução do Capítulo 7; os métodos de geração de código dirigidos por padrões discutidos no Capítulo 9; e o material sobre otimização de código contido no Capítulo 10.

Exercícios

Como antes, classificamos os exercícios com estrelas. Os exercícios sem estrelas testam a compreensão das definições; aqueles com uma estrela dirigem-se aos cursos mais avançados; e os exercícios com duas estrelas estimulam o raciocínio.

Agradecimentos

Nos diversos estágios de elaboração deste livro, várias pessoas fizeram comentários valiosos sobre o texto. Nesse sentido, temos um débito de gratidão com Bill Appelbe, Nelson Beebe, Jon Bentley, Lois Bogess, Rodney Farrow, Stu Feldman, Charles Fischer, Chris Fraser, Art Gittelman, Eric Grosse, Dave Hanson, Fritz Henglein, Robert Henry, Gerard Holzmann, Steve Johnson, Brian Kernighan, Ken Kubota, Daniel Lehmann, Dave MacQueen, Dianne Maki, Alan Martin, Doug McIlroy, Charles McLaughlin, John Mitchell, Elliott Organick, Robert Paige, Phil Pfeiffer, Rob Pike, Kari-Jouko Räihä, Dennis Ritchie, Sriram Sankar, Paul Stoecker, Bjarne Stroustrup, Tom Szymanski, Kim Tracy, Peter Weinberger, Jennifer Widom e Reinhard Wilhelm.

Este livro foi composto pelos autores usando o excelente software disponível no sistema UNIX. O comando `pic` de composição de tipos

```
pic files | tbl | eqn | troff -ms
```

é a linguagem que Brian Kernighan desenvolveu para composição de figuras. Agradecemos especialmente a ele pela ajuda no desenho de figuras.

Destacamos ainda a linguagem `tbl`, de Mike Lesk, para dispor tabelas; a `eqn`, de Brian Kernighan e Lorinda Cherry, para a composição de tipos matemáticos; o programa `troff`, de Joe Ossana, para a formatação de textos para um fotocompositor de tipos, que em nosso caso foi um Mergenthaler Linotron 202/N. O pacote `ms` de macros `troff` foi escrito por Mike Lesk. Além disso, gerenciamos o texto usando `make`, de autoria de Stu Feldman. As referências cruzadas no texto foram mantidas usando `awk`, criado por Al Aho, Brian Kernighan e Peter Weinberger, e `sed`, criado por Lee McMahon.

Os autores gostariam de agradecer particularmente a Patricia Solomon pelo preparo do texto para a fotocomposição. Sua motivação e competência no trabalho de digitação foram grandemente apreciadas. J. D. Ullman foi patrocinado pela Irmandade Einsteiniana da Academia Israelense de Artes e Ciências durante parte do tempo de elaboração do livro. Por fim, os autores desejariam agradecer aos AT & T Bell Laboratories por seu apoio durante a preparação do texto.

A. V. A., R. S., J. D. U.

Sumário

Capítulo 1 Introdução à Compilação, 1

1.1 Compiladores, 1
1.2 Análise do programa-fonte, 2
1.3 As fases de um compilador, 5
1.4 Os primos do compilador, 8
1.5 O agrupamento das fases, 9
1.6 Ferramentas para a construção de compiladores, 10
Notas bibliográficas, 11

Capítulo 2 Um Compilador Simples de Uma Passagem, 12

2.1 Visão geral, 12
2.2 Definição da sintaxe, 12
2.3 Tradução dirigida pela sintaxe, 15
2.4 A análise gramatical, 18
2.5 Um tradutor para expressões simples, 21
2.6 Análise léxica, 25
2.7 Incorporando uma tabela de símbolos, 27
2.8 Máquinas de pilha abstratas, 28
2.9 Juntando as técnicas, 30
Exercícios, 36
Exercícios de programação, 36
Notas bibliográficas, 37

Capítulo 3 A Análise Léxica, 38

3.1 O papel do analisador léxico, 38
3.2 Buferização da entrada, 40
3.3 Especificação dos *tokens*, 42
3.4 O reconhecimento de *tokens*, 45
3.5 Uma linguagem para especificação de analisadores léxicos, 48
3.6 Autômatos finitos, 51
3.7 De uma expressão regular para um AFN, 54
3.8 O projeto de um gerador de analisadores léxicos, 58
3.9 Otimização de reconhecedores de padrões baseados em AFDs, 60
Exercícios, 66
Exercícios de programação, 70
Notas bibliográficas, 70

Capítulo 4 Análise Sintática, 72

4.1 O papel do analisador sintático, 72
4.2 Gramáticas livres de contexto, 74
4.3 Escrevendo uma gramática, 77
4.4 Análise sintática *top-down*, 81
4.5 Análise sintática *bottom-up*, 86
4.6 Análise sintática de precedência de operadores, 90
4.7 Analisadores sintáticos LR, 94
4.8 Usando gramáticas ambíguas, 107
4.9 Geradores de analisadores sintáticos, 111
Exercícios, 115
Notas bibliográficas, 118

Capítulo 5 Tradução Dirigida pela Sintaxe, 120

5.1 Definições dirigidas pela sintaxe, 120
5.2 Construção de árvores sintáticas, 123
5.3 Avaliação *bottom-up* de definições s-atribuídas, 126
5.4 Definições l-atribuídas, 127
5.5 Tradução *top-down*, 129
5.6 Avaliação *bottom-up* dos atributos herdados, 132
5.7 Avaliadores recursivos, 136
5.8 Espaço para os valores de atributos em tempo de compilação, 137
5.9 Reserva de espaço em tempo de construção do compilador, 138
5.10 Análise de definições dirigidas pela sintaxe, 141
Exercícios, 144
Notas bibliográficas, 145

Capítulo 6 Verificação de Tipos, 147

6.1 Sistemas de tipos, 147
6.2 Especificação de um verificador simples de tipos, 149
6.3 Equivalência das expressões de tipo, 151
6.4 Conversões de tipo, 153
6.5 Sobrecarga de funções e operadores, 154
6.6 Funções polimórficas, 156
6.7 Um algoritmo para a unificação, 160
Exercícios, 163
Notas bibliográficas, 165

Capítulo 7 Ambientes em Tempo de Execução, 167

7.1 Temas da linguagem-fonte, 167

7.2 Organização de memória, 170
7.3 Estratégias para a alocação de memória, 172
7.4 Acesso aos nomes não locais, 177
7.5 Transmissão de parâmetros, 183
7.6 Tabelas de símbolos, 185
7.7 Facilidades de linguagem para a alocação dinâmica de memória, 190
7.8 Técnicas de alocação dinâmica de memória, 191
7.9 Alocação de memória em Fortran, 192
Exercícios, 196
Notas bibliográficas, 198

Capítulo 8 Geração de Código Intermediário, 200

8.1 Linguagens intermediárias, 200
8.2 Declarações, 204
8.3 Enunciados de atribuição, 207
8.4 Expressões booleanas, 211
8.5 Enunciados de desvio múltiplo, 215
8.6 Retrocorreção, 216
8.7 Chamadas de procedimentos, 219
Exercícios, 220
Notas bibliográficas, 221

Capítulo 9 Geração de Código, 222

9.1 Temas no projeto de um gerador de código, 222
9.2 A máquina-alvo, 224
9.3 Gerenciamento de memória em tempo de execução, 226
9.4 Blocos básicos e grafos de fluxo, 229
9.5 Informações de uso subseqüente, 231
9.6 Um gerador de código simples, 232
9.7 Alocação e atribuição de registradores, 235
9.8 A representação de blocos básicos sob a forma de GDAs, 237
9.9 Otimização peephole, 240
9.10 A geração de código a partir de GDAs, 242
9.11 Algoritmo de programação dinâmica para a geração de código, 246
9.12 Geradores de geradores de código, 248
Exercícios, 252
Notas bibliográficas, 253

Capítulo 10 Otimização de Código, 254

10.1 Introdução, 254
10.2 As principais fontes de otimização, 257
10.3 Otimização dos blocos básicos, 261
10.4 Laços em grafos de fluxo, 262
10.5 Introdução à análise global de fluxo de dados, 264
10.6 Solução iterativa para as equações de fluxo de dados, 271
10.7 Transformações de melhoria de código, 276
10.8 Lidando com pseudônimos, 283
10.9 Análise de fluxo de dados de grafos de fluxo estruturado, 288
10.10 Algoritmos de fluxo de dados eficientes, 293
10.11 Uma ferramenta para a análise de fluxo de dados, 297
10.12 Estimativas de tipos, 303
10.13 Depuração simbólica de código otimizado, 307
Exercícios, 311
Notas bibliográficas, 313

Capítulo 11 Deseja Escrever um Compilador?, 316

11.1 Planejando um compilador, 316
11.2 Abordagens para o desenvolvimento dos compiladores, 317
11.3 O ambiente de desenvolvimento de compiladores, 318
11.4 Teste e manutenção, 319

Capítulo 12 Um Rápido Exame de Alguns Compiladores, 321

12.1 EQN, um pré-processador para composição de tipos matemáticos, 321
12.2 Compiladores para Pascal, 321
12.3 Os compiladores C, 322
12.4 Os compiladores Fortran H, 322
12.5 O compilador Bliss/11, 324
12.6 O compilador otimizante Modula-2, 325

Apêndice A Um Projeto de Programação, 326

A.1 Introdução, 326
A.2 Estrutura de programa, 326
A.3 Sintaxe de um subconjunto de Pascal, 326
A.4 Convenções léxicas, 327
A.5 Exercícios sugeridos, 327
A.6 Evolução do interpretador, 328
A.7 Extensões, 328

Bibliografia, 329
Índice Alfabético, 339

CAPÍTULO 1

INTRODUÇÃO À COMPILAÇÃO

Os princípios e técnicas da construção de compiladores são tão penetrantes, que as idéias encontradas neste livro serão usadas muitas vezes na carreira de um cientista da computação. A construção de compiladores se estende através dos temas de linguagens de programação, arquitetura de máquina, teoria das linguagens, algoritmos e engenharia de *software*. Afortunadamente, umas poucas técnicas básicas de construção de compiladores podem ser usadas para construir tradutores para uma ampla variedade de linguagens e máquinas. Neste capítulo, introduzimos o tema da compilação através da descrição dos componentes de um compilador, do ambiente no qual realiza seu trabalho e de algumas ferramentas de *software* que facilitam a sua construção.

1.1 COMPILADORES

Posto de forma simples, um compilador é um programa que lê um programa escrito numa linguagem — a linguagem *fonte* — e o traduz num programa equivalente numa outra linguagem — a linguagem *alvo* (ver a Fig. 1.1). Como importante parte desse processo de tradução, o compilador relata a seu usuário a presença de erros no programa fonte.

À primeira vista, a variedade de compiladores pode parecer assustadora. Existem milhares de linguagens fonte, que vão das linguagens de programação tradicionais, tais como Fortran e Pascal, às linguagens especializadas que emergiram virtualmente em quase todas as áreas de aplicação de computadores. As linguagens alvo são igualmente variadas; uma linguagem alvo pode ser uma outra linguagem de programação ou a linguagem de máquina de qualquer coisa entre um microprocessador e um supercomputador. Os compiladores são algumas vezes classificados como de uma passagem, de passagens múltiplas, de carregar e executar, depuradores ou otimizantes, dependendo de como tenham sido construídos ou que função se suponha que devam realizar. A despeito dessa aparente complexidade, as tarefas básicas que qualquer compilador precisa realizar são essencialmente as mesmas. Pela compreensão delas, podemos construir compiladores para uma ampla variedade de linguagens fonte e máquinas alvo, usando as mesmas técnicas básicas.

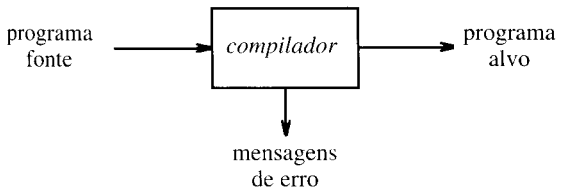


Fig. 1.1 Um compilador.

Nosso conhecimento sobre como organizar e escrever compiladores aumentou bastante desde que os primeiros começaram a surgir ao início dos anos 50. É difícil fornecer uma data exata para o primeiro compilador, porque inicialmente uma grande quantidade de experimentos e implementações foi realizada independentemente por diversos grupos. Muitos dos trabalhos iniciais em compilação lidavam com a tradução de fórmulas aritméticas em código de máquina.

Ao longo dos anos 50, os compiladores foram considerados programas notoriamente difíceis de se escrever. O primeiro compilador Fortran, por exemplo, consumiu 18 homens-ano para implementar (Backus *et al.* [1957]). Descobrimos, desde então, técnicas sistemáticas para o tratamento de muitas das mais importantes tarefas que ocorrem durante a compilação. Igualmente, foram desenvolvidas boas linguagens de implementação, ambientes de programação e ferramentas de *software*. Com esses avanços, até mesmo um compilador substancial pode ser escrito num projeto de estudantes, num curso de construção de compiladores com duração de um semestre.

O Modelo de Compilação de Análise e Síntese

Existem duas partes na compilação: a análise e a síntese. A parte de análise divide o programa fonte nas partes constituintes e cria uma representação intermediária do mesmo. A de síntese constrói o programa alvo desejado, a partir da representação intermediária. Das duas, a síntese requer as técnicas mais especializadas. Iremos considerar informalmente a análise na Secção 1.2 e esboçar, na Secção 1.3, a forma na qual o código alvo é sintetizado por um compilador padrão.

Durante a análise, as operações implicadas pelo programa fonte são determinadas e registradas numa estrutura hierárquica, chamada de árvore. Freqüentemente, é utilizado um tipo especial de árvore, chamado árvore sintática, na qual cada nó representa uma operação e o filho de um nó representa o argumento da operação. Por exemplo, a árvore sintática para um enunciado de atribuição é mostrada na Fig. 1.2.

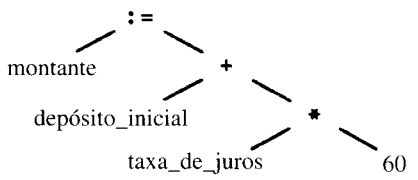


Fig. 1.2 Árvore sintática para `montante := depósito_inicial + taxa_de_juros * 60`.

Muitas ferramentas de *software* que manipulam programas fonte realizam primeiro algum tipo de análise. Alguns exemplos de tais ferramentas incluem:

1. *Editores de estruturas.* Um editor de estruturas toma como entrada um conjunto de comandos para construir um programa fonte. Realiza não só as funções de um editor de textos ordinário, tais como criação e a modificação dos mesmos, mas também analisa o conteúdo de um programa fonte, estabelecendo-lhe uma estrutura hierárquica apropriada. O editor de estruturas pode, também, realizar tarefas adicionais que são úteis ao preparo de programas. Por exemplo, pode verificar se a entrada está corretamente formada, fornecer palavras-chave automaticamente (por exemplo, quando o usuário digita `while`, o editor fornece o `do` correspondente e lembra-o que um enunciado condicional deve figurar entre ambos) e saltar de um `begin`, ou parênteses à esquerda, para o seu `end`, ou parênteses à direita, correspondente, respectivamente. Adicionalmente, a saída de um tal editor é freqüentemente similar àquela da fase de análise de um compilador.

2. *Pretty printers*.* Um *pretty printer* analisa um programa e o imprime numa forma em que a sua estrutura se torne claramente visível. Por exemplo, os comentários podem figurar numa fonte especial e os enunciados podem aparecer com uma indentação proporcional à profundidade do seu nível de aninhamento, na organização hierárquica dos comandos.

3. *Verificadores estáticos.* Um verificador estático lê um programa, analisa-o e tenta descobrir erros potenciais, sem executá-lo. A parte de análise é freqüentemente similar àquela encontrada nos compiladores otimizantes, do tipo discutido no Capítulo 10. Por exemplo, um verificador estático pode detectar quais as partes do programa fonte que não poderão nunca ser executadas, ou que uma certa variável poderia ser usada antes de ter sido definida. Adicionalmente, pode localizar erros lógicos, tais como usar uma variável real como um apontador ou empregar as técnicas de verificação de tipos discutidas no Capítulo 6.

4. *Interpretadores.* Em lugar de produzir um programa alvo como resultado da tradução, um interpretador realiza as operações especificadas pelo programa fonte. Para um enunciado de atribuição, por exemplo, poderia construir uma árvore, como a da Fig. 1.2, e subseqüentemente levar a termo as operações indicadas nos nós, à medida que a percorresse. Na raiz da árvore, descobriria a necessidade de realizar uma atribuição, chamaria uma rotina para avaliar a expressão à direita e armazenaria o valor resultante na localização associada ao identificador `montante`. No filho à esquerda da raiz, descobriria ter de computar a soma de duas expressões. Chamaria recursivamente a si mesma para computar o valor da expressão `taxa_de_juros * 60`. Adicionaria, então, aquele valor ao da variável `depósito_inicial`.

   Os interpretadores são freqüentemente usados para executar linguagens de comandos, dado que cada operador numa tal linguagem é usualmente uma invocação de uma rotina complexa, como um editor ou compilador. Similarmente, algumas linguagens de "nível muito alto", como APL, são normalmente interpretadas, pois existem muitos atributos de dados que não podem ser determinados em tempo de compilação.

Tradicionalmente, pensamos num compilador como um programa que transforma uma linguagem fonte, como Fortran, numa linguagem de montagem ou na linguagem de máquina de algum computador. No entanto, existem algumas áreas, visivelmente irrelacionadas, onde a tecnologia de compiladores é regularmente utilizada. A parte de análise, em cada um dos exemplos seguintes, é similar àquela de um compilador convencional.

1. *Formatadores de texto.* Um formatador de texto toma por entrada um fluxo de caracteres, a maior parte do mesmo como texto a ser composto tipograficamente, mas com alguma parte incluindo comandos, a fim de indicar parágrafos, figuras ou estruturas matemáticas, tais como subscritos e sobrescritos. Mencionamos algumas das análises realizadas por formatadores de texto na próxima seção.

2. *Compiladores de silício.* Um compilador de silício possui uma linguagem fonte que é similar ou idêntica à de uma linguagem de programação convencional. Entretanto, as variáveis da mesma não representam localizações de memória, mas sinais lógicos (0 ou 1) ou grupos de sinais de um circuito de chaveamento. A saída é um projeto de circuito, numa linguagem apropriada. Ver Johnson [1983], Ullman [1984] ou Trickey [1985] para uma discussão da compilação de silício.

3. *Interpretadores de queries**.* Um interpretador de *queries* traduz um predicado, contendo operadores booleanos ou relacionais, em comandos, para percorrer um banco de dados, de forma a satisfazer ao predicado.

O Contexto de um Compilador

Adicionalmente ao compilador, vários outros programas podem ser necessários para se criar um programa alvo executável. Um programa fonte pode ser dividido em módulos armazenados em arquivos separados. A tarefa de coletar esses módulos é, algumas vezes, confiada a um programa distinto, chamado de pré-processador. O pré-processador pode, também, expandir formas curtas, chamadas de macros, em enunciados da linguagem fonte.

A Fig. 1.3 mostra uma "compilação" típica. O programa alvo criado pelo compilador pode exigir processamento posterior antes que possa ser executado. O compilador da Fig. 1.3 cria um código de montagem que é traduzido no de máquina por um montador e, então, ligado a algumas rotinas de biblioteca, formando o código que é efetivamente executado em máquina.

Iremos considerar os componentes de um compilador nas próximas duas seções; os programas restantes na Fig. 1.3 são discutidos na Seção 1.4.

1.2 ANÁLISE DO PROGRAMA FONTE

Nesta seção, introduzimos a análise e ilustramos seu uso em algumas linguagens de formatação de texto. O assunto é tratado em mais detalhes nos Capítulos 2-4 e 6. Na compilação, a análise consiste em três fases:

1. *Análise linear,* na qual um fluxo de caracteres constituindo um programa é lido da esquerda para a direita e agrupado em *tokens*, que são seqüências de caracteres tendo um significado coletivo.

2. *Análise hierárquica,* na qual os caracteres ou *tokens* são agrupados hierarquicamente em coleções aninhadas com significado coletivo.

3. *Análise semântica,* na qual certas verificações são realizadas a fim de se assegurar que os componentes de um programa se combinam de forma significativa.

*N. do T. Manteremos o original em inglês por absoluta falta de correspondência na língua portuguesa de um termo que sequer se aproxime da idéia expressa na linguagem original.

*N. do T. A questão aqui é que não há termo, em português, que satisfaça o conceito de *query*. Optamos, então, por não traduzi-lo.

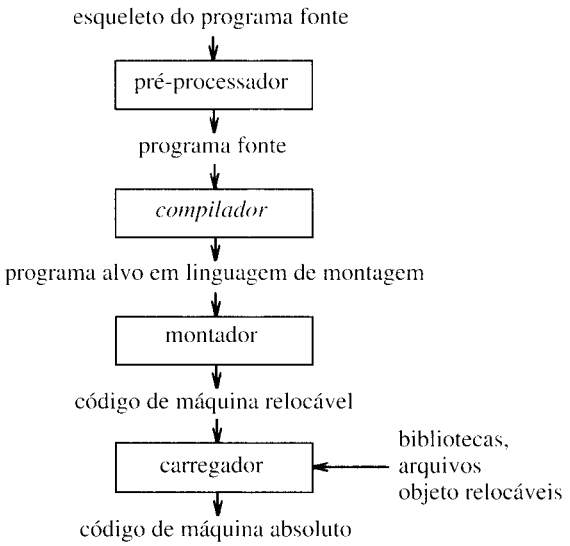


Fig. 1.3 Um sistema de processamento de linguagem.

Análise Léxica

Num compilador, a análise linear é chamada de *análise léxica* ou *esquadrinhamento (scanning)*. Por exemplo, na análise léxica, os caracteres no enunciado de atribuição

`montante := depósito_inicial + taxa_de_juros * 60`

poderiam ser agrupados nos seguintes *tokens:*

1. O identificador `montante`.
2. O símbolo de atribuição `:=`
3. O identificador `depósito_inicial`.
4. O sinal de adição.
5. O identificador `taxa_de_juros`.
6. O sinal de multiplicação.
7. O número `60`.

Os espaços que separam os caracteres desses *tokens* seriam normalmente eliminados durante a análise léxica.

Análise Sintática

A análise hierárquica é chamada de *análise gramatical* ou *análise sintática*. Envolve o agrupamento dos *tokens* do programa fonte em frases gramaticais, que são usadas pelo compilador, a fim de sintetizar a saída. Usualmente, as frases gramaticais do programa fonte são representadas por uma árvore gramatical, tal como a mostrada na Fig. 1.4.

Na expressão `depósito_inicial + taxa_de_juros * 60`, a frase `taxa_de_juros * 60` é uma unidade lógica, porque as convenções usuais das expressões aritméticas nos dizem que a multiplicação é realizada antes da adição. Como a expressão `depósito_inicial + taxa_de_juros` é seguida por um *, não é agrupada numa única frase, na Fig. 1.4.

A estrutura hierárquica de um programa é usualmente expressa por regras recursivas. Por exemplo, poderíamos ter as seguintes regras como parte da definição de expressões:

1. Qualquer *identificador* é uma expressão.
2. Qualquer *número* é uma expressão.
3. Se $expressão_1$ e $expressão_2$ são expressões, então também o são

$$expressão_1 + expressão_2$$
$$expressão_1 * expressão_2$$
$$(expressão_1)$$

As regras (1) e (2) são as regras base (não recursivas), enquanto que (3) define expressões em termos dos operadores aplicados às demais expressões. Então, pela regra (1) `depósito_inicial` e `taxa_de_juros` são expressões. Pela regra (2), `60` é uma expressão, enquanto que, pela regra (3), podemos primeiro inferir que `taxa_de_juros * 60` é uma expressão e, finalmente, que `depósito_inicial + taxa_de_juros * 60` também o é.

Similarmente, muitas linguagens definem recursivamente enunciados tais como:

1. Se $identificador_1$ é um identificador e $expressão_2$ uma expressão, então

$$identificador_1 := expressão_2$$

é um enunciado.

2. Se $expressão_1$ é uma expressão e $comando_2$ é um enunciado, então

$$\textbf{while } (expressão_1) \textbf{ do } comando_2$$
$$\textbf{if } (expressão_1) \textbf{ then } comando_2$$

são enunciados.

A divisão entre a análise léxica e a sintática é um tanto arbitrária. Usualmente, escolhemos uma que simplifique a tarefa global de

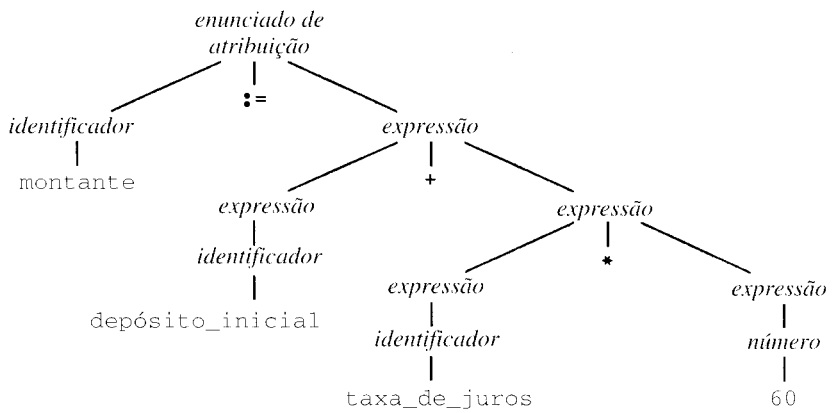


Fig. 1.4 Árvore gramatical para `montante := depósito_inicial + taxa_de_juros * 60`.

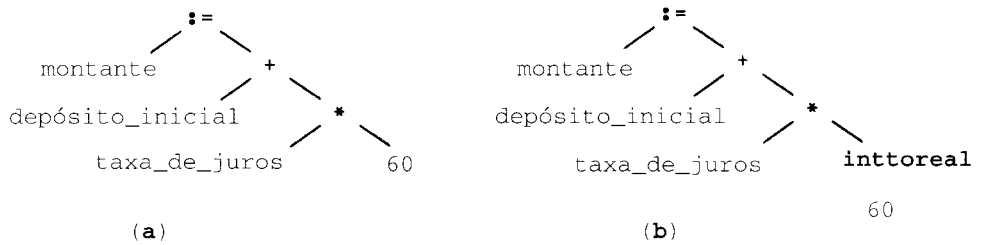


Fig. 1.5 A análise semântica insere uma conversão de inteiro para real.

análise. Um fator determinante na divisão é o de uma construção da linguagem fonte ser inerentemente recursiva ou não. As construções léxicas não requerem recursão, enquanto que as sintáticas freqüentemente a exigem. As gramáticas livres de contexto são uma formalização das regras recursivas que podem ser usadas para guiar a análise sintática. São introduzidas no Capítulo 2 e extensivamente estudadas no Capítulo 4.

Por exemplo, a recursão não é requerida para reconhecer identificadores, que são tipicamente cadeias de letras e dígitos, começando por uma letra. Reconheceríamos normalmente os identificadores por um simples esquadrinhamento do fluxo de entrada, aguardando até que um caractere, que não uma letra ou um dígito, fosse encontrado, e agrupando, num *token* tipo identificador, todas as letras e dígitos coletados até aquele ponto. Os caracteres, dessa forma agrupados, seriam registrados numa tabela, chamada tabela de símbolos, e removidos da entrada, de tal forma que o processamento do próximo *token* pudesse se iniciar.

Por outro lado, esse tipo de esquadrinhamento linear não é poderoso o suficiente para analisar expressões ou enunciados. Por exemplo, não podemos fazer corresponder apropriadamente os parênteses nas expressões, ou o `begin` e o `end` nos enunciados, sem criar algum tipo de estrutura hierárquica ou aninhamento na entrada.

A árvore gramatical da Fig. 1.4 descreve a estrutura sintática da entrada. Uma representação interna mais comum dessa estrutura sintática é dada pela árvore sintática na Fig. 1.5(a). Uma árvore sintática é uma representação condensada da árvore gramatical, na qual os operadores figuram como nós interiores e os operandos de um operador são os filhos do nó daquele operador. A construção de árvores, tais como aquela da Fig. 1.5(a), é discutida na Seção 5.2. Iremos examinar no Capítulo 2, e em mais detalhes no Capítulo 5, o tema da *tradução dirigida pela sintaxe,* na qual o compilador usa a estrutura hierárquica da entrada a fim de auxiliar a geração da saída.

Análise Semântica

A fase de análise semântica verifica os erros semânticos no programa fonte e captura as informações de tipo para a fase subseqüente de geração de código. Utiliza a estrutura hierárquica determinada pela fase de análise sintática, a fim de identificar os operadores e operandos das expressões e enunciados.

Um importante componente da análise semântica é a verificação de tipos. Nela o compilador checa se cada operador recebe os operandos que são permitidos pela especificação da linguagem fonte. Por exemplo, muitas definições nas linguagens de programação requerem que o compilador relate um erro a cada vez que um número real seja usado para indexar um *array.* No entanto, a especificação da linguagem pode permitir algumas coerções de operandos, como, por exemplo, quando um operando aritmético binário é aplicado a um inteiro e a um real. Nesse caso, o compilador pode precisar converter o inteiro para real. A verificação de tipos e a análise semântica são discutidas no Capítulo 6.

Exemplo 1.1. Dentro da máquina, um padrão de *bits* representando um inteiro é geralmente diferente do padrão de *bits* para um real, mesmo que o número inteiro e o número real tenham o mesmo valor. Suponha, por exemplo, que todos os identificadores na Fig. 1.5 tenham sido declarados como reais e assuma que 60, por si só, seja um inteiro. A verificação de tipos da Fig. 1.5(a) revela que o `*` está aplicado a um real, `taxa_ de_ juros`, e a um inteiro, 60. O enfoque geral é o de converter o inteiro em real. Isso foi conseguido na Fig. 1.5(b) pela criação de um nó extra para o operador **inttoreal**, que converte explicitamente um inteiro num real. Alternativamente, como o operando de **inttoreal** é uma constante, o compilador pode, em lugar, substituir a constante inteira por uma constante real equivalente. □

A Análise nos Formatadores de Texto

É útil considerar a entrada para um formatador de texto como especificando uma hierarquia de *compartimentos,* que são regiões retangulares a serem preenchidas por algum padrão de *bits,* representando pontos claros e escuros a serem impressos no dispositivo de saída.

Por exemplo, o sistema TEX (Knuth [1984a]) vê sua entrada dessa forma. Cada caractere, que não seja parte de um comando, representa um compartimento contendo o padrão de *bits* para aquele caractere, na fonte e tamanho apropriados. Os caracteres consecutivos não separados por "espaços em branco" (espaços ou caracteres de avanço de linha) são agrupados em palavras, consistindo em uma seqüência de compartimentos horizontalmente arranjados, mostrados esquematicamente na Fig. 1.6. O agrupamento de caracteres em palavras (ou comandos) é o aspecto linear ou léxico da análise do formatador de texto.

Os compartimentos, em TEX, podem ser construídos a partir de outros menores, através de combinações arbitrárias, horizontais e verticais. Por exemplo,

```
\hbox{ <lista de boxes> }
```

agrupa a lista de compartimentos pela justaposição dos mesmos horizontalmente, enquanto que operador `\vbox` pode agrupar uma lista de compartimentos por justaposição vertical. Dessa forma, se escrevermos em TEX

```
\hbox {\vbox{! 1} \vbox{@ 2}}
```

obteremos o arranjo de compartimentos mostrado na Fig. 1.7. A determinação do arranjo hierárquico de compartimentos estabelecido pela entrada é parte da análise sintática em TEX.

Como outro exemplo, o pré-processador EQN para a matemática (Kernighan e Cherry [1975]) ou o processador matemático em TEX constroem expressões matemáticas a partir de operadores como `sub` e

Fig. 1.6 Agrupamento de caracteres e palavras em *compartimentos.*

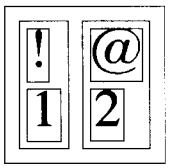


Fig. 1.7 Hierarquia de compartimentos em TEX.

sup, significando subscritos e sobrescritos. Se EQN encontra um texto de entrada da forma

BOX `sub` *box*

comprime o tamanho de *box* e o atrela a *BOX*, próximo ao canto inferior direito, como ilustrado na Fig. 1.8. O operador *sup*, similarmente, atrela *box* ao canto superior direito.

Esses operadores podem ser aplicados recursivamente, de tal forma que o texto EQN

```
a sub {i sup 2}
```

resulta em a_{i^2}.* O agrupamento dos operadores `sub` e `sup` em tokens é parte da análise léxica do texto EQN. No entanto, é necessária a sua estrutura sintática para determinar o tamanho e a localização de um compartimento.

1.3 AS FASES DE UM COMPILADOR

Conceitualmente, um compilador opera em *fases*, cada uma das quais transforma o programa fonte de uma representação para outra. Uma decomposição típica de um compilador é mostrada na Fig. 1.9. Na prática, algumas das fases podem ser agrupadas e a representação intermediária entre as mesmas não precisa ser explicitamente construída.

As três primeiras fases, formando o núcleo da parte de análise do compilador, foram introduzidas na última secção. Duas outras atividades, o gerenciamento da tabela de símbolos e a manipulação de erros, são mostradas interagindo com as seis fases de análise léxica, análise sintática, análise semântica, geração de código intermediário, otimização e geração de código. Informalmente, também chamaremos de "fases" o gerenciador da tabela de símbolos e o manipulador de erros.

Gerenciamento da Tabela de Símbolos

Uma função essencial do compilador é registrar os identificadores usados no programa fonte e coletar as informações sobre os seus diversos atributos. Esses atributos podem providenciar informações sobre a memória reservada para o identificador, seu tipo, escopo (onde é válido no programa) e, no caso de nomes de procedimentos, coisas tais como o número e os tipos de seus argumentos, o método de transmissão de cada um (por exemplo, por referência) e o tipo retornado, se algum.

Uma *tabela de símbolos* é uma estrutura de dados contendo um registro para cada identificador, com os campos contendo os atributos do identificador. A estrutura de dados nos permite encontrar rapidamente cada registro e, igualmente, armazenar ou recuperar dados do mesmo. As tabelas de símbolos são discutidas nos Capítulos 2 e 7.

Quando, no programa fonte, o analisador léxico detecta um identificador, instala-o na tabela de símbolos. No entanto, os atributos

*N. do T. operador *sub* se aplica a toda a expressão entre chaves; o operador *sup* faz com que o 2 figure como o sobrescrito de *i* e não de *a*.

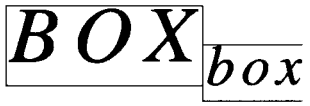


Fig. 1.8 Construindo uma estrutura de subscritos num texto matemático.

do identificador não podem ser normalmente determinados durante a análise léxica. Por exemplo, em Pascal, numa declaração como

```
var montante, depósito_inicial,
taxa_de_juros: real;
```

o tipo real ainda não será conhecido quando `montante`, `depósito_inicial` e `taxa_de_juros` forem enxergados pelo analisador léxico.

As fases remanescentes colocam informações sobre os identificadores na tabela de símbolos e em seguida as usam de várias maneiras. Por exemplo, ao realizar a análise semântica e a geração de código intermediário, precisamos conhecer de que tipos os identificadores são, a fim de verificar se o programa fonte os usa de forma válida e, por conseguinte, gerar as operações apropriadas sobre os mesmos. O gerador de código tipicamente instala e usa as informações detalhadas a respeito da memória atribuída aos identificadores.

Detecção de Erros e Geração de Relatórios

Cada fase pode encontrar erros. Entretanto, após encontrá-los, precisa lidar de alguma forma com os mesmos, de tal forma que a compilação possa continuar, permitindo que sejam detectados erros posteriores no programa fonte. Um compilador que pare ao encontrar o primeiro erro não é tão prestativo quanto poderia sê-lo.

As fases de análise sintática e semântica tratam usualmente de uma ampla fatia dos erros detectáveis pelo compilador. A fase de análise léxica pode detectá-los quando os caracteres remanescentes na entrada não formem qualquer *token* da linguagem. Os erros, onde o fluxo de *tokens* viole as regras estruturais (sintaxe) da linguagem, são determinados pela fase de análise sintática. Durante a análise semântica, o

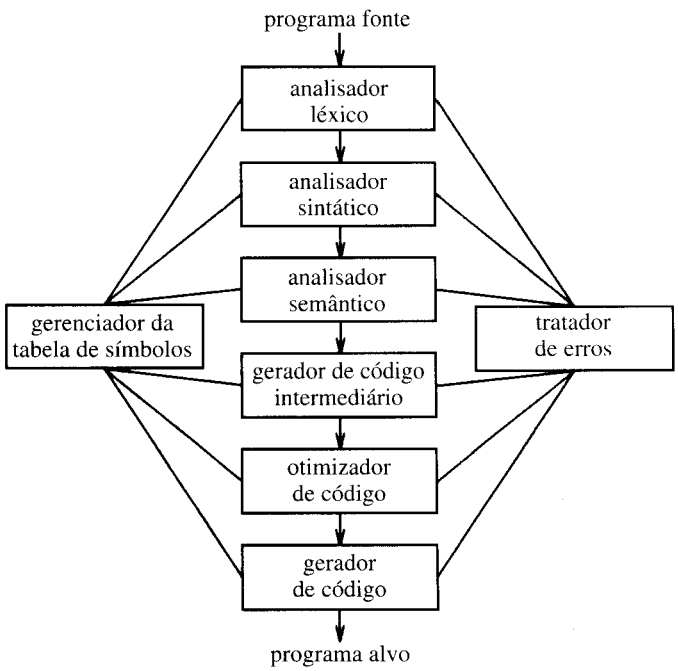


Fig. 1.9 Fases de um compilador.

compilador tenta detectar as construções que possuam a estrutura sintática correta, sem nenhuma preocupação com o significado da operação envolvida, como, por exemplo, ao tentarmos adicionar dois identificadores, um dos quais seja o nome de uma *array* e o outro o nome de um procedimento. Neste livro discutiremos o tratamento dos erros de fase ao examinarmos especificamente cada fase.

As Fases de Análise

À medida que a tradução progride, a representação interna do compilador para o programa fonte muda. Ilustramos essas representações considerando a tradução do enunciado

```
montante := depósito_inicial + taxa_de_juros
* 60                                          (1.1)
```

A Fig. 1.10 mostra a representação desse enunciado após cada fase.

A fase de análise léxica lê os caracteres de um programa fonte e os agrupa num fluxo de *tokens*, no qual cada *token* representa uma seqüência de caracteres logicamente coesiva, como, por exemplo, um identificador, uma palavra-chave (`if`, `while` etc.), um caractere de pontuação ou um operador composto por vários caracteres, como `:=`. A seqüência dos caracteres que formam um *token* é chamada o *lexema* para aquele *token*.

Certos *tokens* serão enriquecidos por um "valor léxico". Por exemplo, quando um identificador, como `taxa_de_juros`, é encontrado, o analisador léxico não somente gera um *token*, digamos **id**, mas, também, instala o lexema `taxa_de_juros` na tabela de símbolos, se já não estiver lá. O valor léxico associado a essa ocorrência de **id** aponta para a entrada de taxa de juros na tabela de símbolos.

Nesta secção, usaremos $\mathbf{id}_1$, $\mathbf{id}_2$, e $\mathbf{id}_3$ para `montante`, `depósito_inicial`, e `taxa_de_juros`, a fim de enfatizar que a

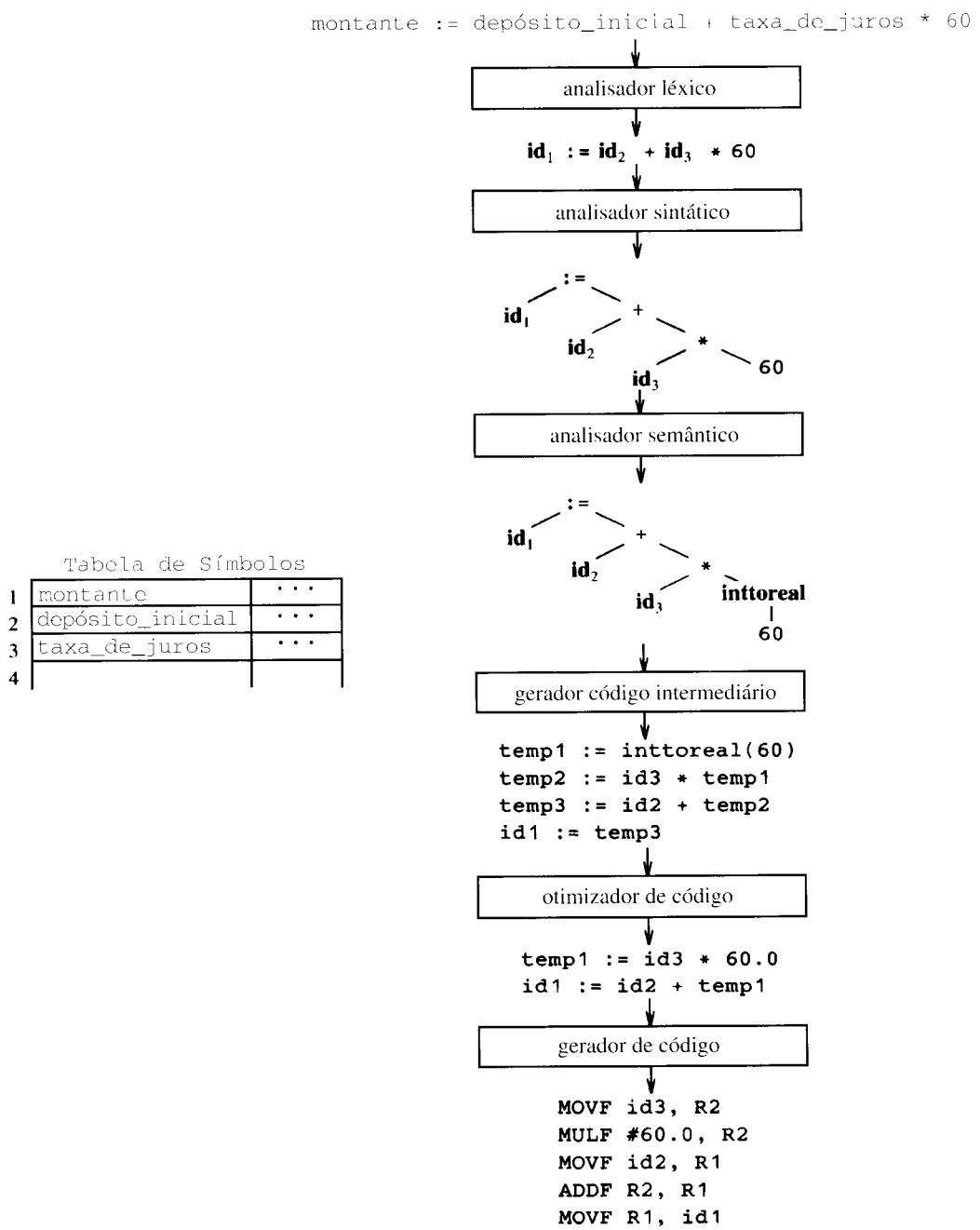


Fig. 1.10 Tradução de um enunciado.

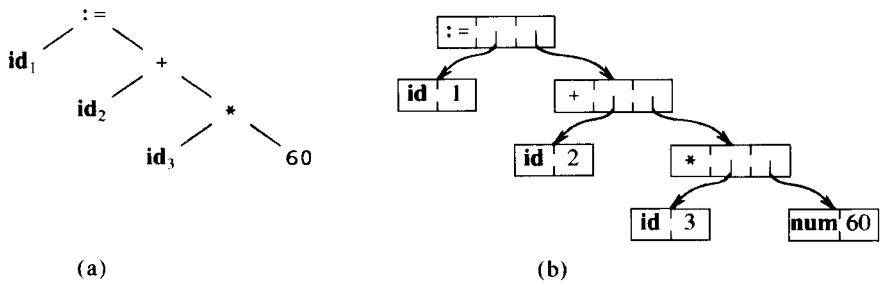


Fig. 1.11 A estrutura de dados em (b) para a árvore em (a).

representação interna de um identificador é diferente da seqüência de caracteres que formam o identificador. A representação de (1.1) após a análise léxica é, por conseguinte, sugerida por:

$$\mathbf{id}_1 := \mathbf{id}_2 + \mathbf{id}_3 * 60 \qquad (1.2)$$

Deveríamos criar, também, *tokens* para o operador multicaractere := e para o número `60`, a fim de refletir suas representações internas, mas tal será postergado até o Capítulo 2. A análise léxica é coberta em detalhes no Capítulo 3.

A segunda e a terceira fase, de análise sintática e semântica, respectivamente, já foram também introduzidas na Seção 1.2. A análise sintática impõe uma estrutura hierárquica ao fluxo de *tokens*, a qual iremos retratar através de árvores sintáticas, como na Fig. 1.11(a). Uma estrutura de dados típica para a árvore é mostrada na Fig. 1.11(b), na qual um nó interior é um registro com um campo para o operador e dois campos contendo apontadores para os registros dos filhos à esquerda e à direita. Uma folha é um registro com dois ou mais campos, um para identificar o *token* que está à folha, e os outros para registrar informações sobre o *token*. Informações adicionais sobre as construções da linguagem podem ser mantidas através da adição de novos campos aos registros dos nós. Discutimos a análise sintática e a análise semântica nos Capítulos 4 e 6, respectivamente.

Geração de Código Intermediário

Após as análises sintática e semântica, alguns compiladores geram uma representação intermediária explícita do programa fonte. Podemos pensar dessa representação intermediária como um programa para uma máquina abstrata. Essa representação intermediária deveria possuir duas propriedades importantes: ser fácil de produzir e fácil de traduzir no programa alvo.

A representação intermediária pode ter uma variedade de formas. No Capítulo 8, consideramos uma forma intermediária chamada "código de três endereços", que é como uma linguagem de montagem para uma máquina, na qual cada localização de memória possa atuar como um registrador. O código de três endereços consiste em uma seqüência de instruções, cada uma delas possuindo no máximo três operandos. O programa fonte em (1.1) poderia ser expresso no código de três endereços como

```
temp1 := inttoreal (60)
temp2 := id3 * temp1
temp3 := id2 + temp2                    (1.3)
id1 := temp3
```

Esta forma intermediária possui várias propriedades. Primeiro, cada instrução de três endereços possui, no máximo, um operador, além do de atribuição. Então, ao gerar essas instruções, o compilador precisa decidir sobre a ordem em que as mesmas devam ser realizadas; a multiplicação precede a adição no programa fonte de (1.1). Segundo, o compilador precisa gerar um nome temporário para receber o valor computado em cada instrução. Terceiro, algumas instruções de três endereços possuem menos do que três operandos, por exemplo, a primeira e a última instruções em (1.3).

No Capítulo 8, cobrimos as principais representações intermediárias usadas nos compiladores. Em geral, essas representações precisam fazer mais do que computar expressões; precisam também tratar construções do fluxo de controle e chamadas de procedimentos. Os Capítulos 5 e 8 apresentam algoritmos para a geração do código intermediário das construções típicas das linguagens de programação.

Otimização de Código

A fase de otimização tenta melhorar o código intermediário, de tal forma que venha resultar um código de máquina mais rápido em tempo de execução. Algumas otimizações são triviais. Por exemplo, um algoritmo natural gera o código intermediário (1.3) após a análise semântica, usando uma instrução para cada operador na representação em árvore, ainda que exista uma maneira melhor de se realizar a mesma computação, usando-se as duas instruções

```
temp1 := id3 * 60.0
id1 := id2 + temp1                      (1.4)
```

Não existe nada de errado com esse algoritmo simples, dado que o problema pode ser corrigido durante a fase de otimização de código. Ou seja, o compilador pode deduzir que a conversão de `60`, da representação de inteiro para a de real, pode ser realizada uma vez, e para todo o sempre, em tempo de compilação, e, por conseguinte, a operação `inttoreal` pode ser eliminada. Além do mais, `temp3` é usada para transmitir seu valor a `id1` uma única vez. Torna-se seguro, então, substituir `temp3` por `id1` e isso feito torna o último enunciado de (1.3) desnecessário, resultando no código de (1.4).

Existe uma grande variação na quantidade de otimizações de código que cada compilador executa. Naqueles que mais a realizam, chamados de "compiladores otimizantes", uma porção significativa de seus tempos é gasta nessa fase. Entretanto, existem otimizações simples que melhoram significativamente o tempo de execução do programa alvo, sem alongar o tempo de compilação. Muitas delas são discutidas no Capítulo 9, enquanto que o Capítulo 10 fornece a tecnologia usada pelos compiladores otimizantes mais poderosos.

Geração de Código

A fase final do compilador é a geração do código alvo, consistindo normalmente de código de máquina relocável ou código de montagem. As localizações de memória são selecionadas para cada uma das variáveis usadas pelo programa. Então, as instruções intermediárias são, cada uma, traduzidas numa seqüência de instruções de máquina que realizam a mesma tarefa. Um aspecto crucial é a atribuição das variáveis aos registradores.

Por exemplo, usando-se os registradores 1 e 2, a tradução do código de (1.4) poderia se tornar

```
MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1                    (1.5)
ADDF R2, R1
MOVF R1, id1
```

O primeiro e o segundo operandos de cada instrução especificam o emissor e o receptor, respectivamente. O F em cada instrução nos informa que as instruções lidam com números em ponto flutuante. Esse código copia o conteúdo do endereço[1] `id3` no registrador 2 e, então, multiplica-o pela constante 60.0 . O # significa que `60.0` deve ser tratado como uma constante. A terceira instrução copia o conteúdo de `id2` no registrador 1 e o adiciona ao valor previamente computado no registrador 2. Finalmente, o valor no registrador 1 é copiado para o endereço de `id1`, de tal forma que o código implementa a atribuição na Fig. 1.10. O Capítulo 9 cobre a geração de código.

1.4 OS PRIMOS DO COMPILADOR

Como vimos na Fig. 1.3, a entrada para o compilador pode ser produzida por um ou mais pré-processadores e pode ser necessário processamento posterior da saída do compilador, antes do código de máquina ser obtido. Nesta seção, discutimos o contexto no qual um compilador tipicamente opera.

Pré-processadores

Os pré-processadores produzem entrada para compiladores. Podem realizar as seguintes funções:

1. *Processamento de macros.* Um pré-processador pode permitir que um usuário defina macros que sejam abreviações para construções mais longas.

2. *Inclusão de arquivos.* Um pré-processador pode incluir arquivos no papel de cabeçalhos do texto do programa. Por exemplo, o pré-processador C faz com que o conteúdo do arquivo `<global.h>` substitua o enunciado `#include <global.h>` ao processar um arquivo contendo tal enunciado.

3. *Pré-processadores "racionais".* Tais pré-processadores expandem as linguagens mais antigas com facilidades modernas de controle de fluxo e estruturação de dados. Por exemplo, tal pré-processador poderia providenciar, ao usuário, macros embutidas para construções, tais como comandos *while* ou *if*, quando os mesmos não existissem na linguagem de programação em si.

4. *Extensores de linguagens.* Tentam conferir maior poder às linguagens, através de macros embutidas. Por exemplo, Equel (Stonebraker *et al.* [1976]) é uma linguagem de interrogação de banco de dados embutida em C. Os enunciados começando por `##` são considerados pelo pré-processador como comandos de acesso a banco de dados, irrelacionados com a linguagem C, e traduzidos em chamadas de procedimentos para rotinas que realizam tal acesso.

Os processadores de macros lidam com dois tipos de enunciados: definição e uso de macros. As definições são normalmente indicadas por algum caractere único ou palavra-chave, como `define` ou `macro`. Consistem em um nome para a macro sendo definida e em um *corpo*, formando a definição. Freqüentemente, os processadores de macros permitem *parâmetros formais* em suas definições, ou seja, símbolos a serem substituídos por valores (um "valor" é uma cadeia de caracteres, nesse contexto). O uso de uma macro consiste na designação de uma macro, através de seu nome, e no fornecimento dos *parâmetros atuais*, isto é, valores para seus parâmetros formais. O processador de macros substitui os parâmetros formais pelos atuais no corpo da macro; por conseguinte, o corpo substituído substitui a macro em si.

Exemplo 1.2. O sistema de composição tipográfica TEX, mencionado na Seção 1.2, contém uma facilidade generalizada para macros. As definições de macros tomam a forma

```
\define <nome da macro> <gabarito> {<corpo>}
```

Um nome de macro é qualquer cadeia de letras precedida por uma barra invertida. O gabarito é qualquer cadeia de caracteres, com as cadeias de forma `#1, #2, ..., #9` consideradas como parâmetros formais. Esses símbolos podem também figurar no corpo, qualquer número de vezes. Por exemplo, a macro seguinte define uma citação para o *Journal of the ACM.*

```
\define\JACM #1; #2; #3.
  { {\sl J. ACM} {\bf #1} : #2, pp. #3.}
```

O nome da macro é `\JACM` e o gabarito é `"#1;#2;#3."`; os pontos-e-vírgulas separam os parâmetros e o último é seguido por um ponto. Um uso dessa macro precisa tomar a forma do gabarito, exceto que cadeias arbitrárias devem substituir os parâmetros formais.[2] Podemos, então, escrever

```
\JACM 17;4;715-728.
```

e esperar ver

J. *ACM* **17**:4 , pp. 715-728

A parte do corpo `{\sl J. ACM}` chama por um "*J. ACM*" em itálico ("inclinado"). A expressão `{\bf #1}` informa que o primeiro parâmetro atual deve ficar em negrito; esse parâmetro está destinado a ser o número do volume.

TEX permite que qualquer pontuação ou cadeia de texto separe o volume, número e numeração de página na definição da macro `\JACM`. Poderíamos mesmo não ter usado pontuação alguma, caso em que TEX tomaria cada parâmetro atual como sendo constituído por um único caractere ou uma cadeia envolvida por { }. □

Montadores

Alguns compiladores produzem um código de montagem, como em (1.5), que é passado a um montador para processamento posterior. Outros compiladores realizam a tarefa do montador, produzindo um código de máquina relocável, que pode ser passado diretamente para um carregador/editor de ligações. Assumimos que o leitor tenha alguma familiaridade com o que uma linguagem de montagem se pareça e o que um montador faça. Aqui, iremos rever o relacionamento entre o código de montagem e o código de máquina.

O *código de montagem* é uma versão mnemônica do código de máquina, na qual são usados nomes em lugar do código binário para as

[1] Deixamos de lado o importante tema da reserva de memória para os identificadores no programa fonte. Como veremos no Capítulo 7, a organização de memória em tempo de execução depende da linguagem sendo compilada. As decisões sobre a reserva de memória ou são tomadas durante a geração do código intermediário ou durante a geração de código.

[2] Bem, cadeias quase arbitrárias, pois no uso da macro se dá um esquadrinhamento simples da esquerda para a direita e tão logo no texto seja encontrado um símbolo que se iguale ao que vem em seguida a um símbolo `#i` no gabarito, considera-se a cadeia precedente emparelhada com `#i`. Então, se tentássemos substituir `#1 por ab; cd` encontraríamos que somente `ab` ter-se-ia emparelhado com `#1`, tendo `cd` sido emparelhada com `#2`.

operações e fornecidos nomes aos endereços de memória. Uma seqüência típica de instruções de montagem seria

```
MOV a, R1
ADD #2, R1                    (1.6)
MOV R1, b
```

Este código copia o conteúdo do endereço `a` no registrador 1, adiciona a constante 2 ao mesmo, tratando o conteúdo do registrador 1 como um número em ponto fixo, e, finalmente, armazena o resultado na localização denominada `b`. Computa, então, `b := a + 2`.

É usual que as linguagens de montagem possuam facilidades de macros, que são similares àquelas dos pré-processadores de macros discutidas acima.

Montagem em Duas Passagens

A forma mais simples do montador realiza duas passagens sobre a sua entrada, onde uma *passagem* consiste na leitura do arquivo de entrada uma única vez. Na primeira, todos os identificadores que denotam localizações de memória são localizados e armazenados numa tabela de símbolos (separada do compilador). São associadas localizações de memória aos identificadores à medida que os mesmos sejam encontrados pela primeira vez, de tal forma que após ler (1.6), por exemplo, a tabela de símbolos poderia conter as entradas mostradas na Fig. 1.12. Nela, assumimos que seja reservada uma palavra, consistindo em quatro *bytes* para cada identificador e que os endereços sejam atribuídos começando-se pelo *byte* 0.

Na segunda passagem, o montador esquadrinha a entrada de novo. Desta vez, traduz tanto cada operação em seqüências de *bits*, representando àquela operação em linguagem de máquina, quanto cada identificador, representando uma localização no endereço atribuído ao mesmo na tabela de símbolos.

A saída da segunda passagem é usualmente um código de máquina *relocável*, significando que pode ser carregado começando em qualquer localização L na memória, isto é, se L for adicionado a todos os endereços no código, então todas as referências estarão corretas. A saída do montador precisa distinguir, então, aquelas partes das instruções que se refiram a endereços que possam ser relocados.

Exemplo 1.3. O que se segue é o código de uma máquina hipotética, no qual poderiam ser traduzidas as instruções de montagem (1.6).

```
0001 01 00 00000000 *
0011 01 10 00000010                (1.7)
0010 01 00 00000100 *
```

Examinamos aqui uma pequenina palavra de instrução, na qual os primeiros quatro *bits* são o código de instrução, com `0001`, `0010` e `0011` significando carregar *(load)*, armazenar *(store)* e adicionar *(add)*, respectivamente. Por carregar e armazenar significamos cópias a partir da memória para um registrador e vice-versa. Os dois *bits* seguintes designam um registrador, e `01` designa o de número 1, em cada uma das três instruções acima. Os dois *bits* após representam um "descritor", com `00` significando modo de endereçamento ordinário, e os últimos oito *bits* se referem a endereços de memória. O descritor `10` significa modo imediato, onde os últimos oito *bits* são tomados literalmente como um operando. Esse modo figura na segunda instrução de (1.7).

IDENTIFICADOR	ENDEREÇO
a	0
b	4

FIg. 1.12 Uma tabela de símbolos de um montador com identificadores de (1.6).

Observamos, também, em (1.7), um * associado à primeira e terceira instruções. Aquele asterisco representa o *bit de relocação* que está associado a cada operação no código relocável de máquina. Suponhamos que o espaço de endereçamento contendo os dados deva ser carregado iniciando-se pela localização L. A presença do * significa que L precisa ser adicionado ao endereço na instrução. Então, se L = `00001111`, isto é, 15, então `a` e `b` estariam nas localizações 15 e 19, respectivamente, e as instruções de (1.7) apareceriam como

```
0001 01 00 00001111
0011 01 10 00000010                (1.8)
0010 01 00 00010011
```

sob a forma de um código de máquina *absoluto* ou irrelocável. Note que, como não há * associado à segunda instrução em (1.7), L não foi adicionado ao seu endereço em (1.8), o que está exatamente correto, porque seus *bits* representam a constante 2, não a localização 2.

Carregadores e Editores de Ligação

Usualmente, um programa chamado *carregador* realiza as duas funções de carga e de edição de ligações. O processo de carga consiste em se tomar um código relocável de máquina, alterar os endereços relocáveis, como discutido no Exemplo 1.3, e em se colocar as instruções alteradas e os dados na memória nas localizações apropriadas.

O editor de ligações nos permite criar um único programa a partir de diversos arquivos de código relocável de máquina. Esses arquivos podem ter sido o resultado de diversas compilações diferentes e um ou mais deles podem ser arquivos de bibliotecas de rotinas, providenciadas pelo sistema, e disponíveis a qualquer programa que delas necessite.

Se os arquivos estão destinados a serem usados juntos numa forma útil, devem existir algumas *referências externas*, nas quais o código de um arquivo se refira a uma localização em outro arquivo. A referência pode ser para uma localização de dados definida em um arquivo, e usada num outro, ou pode ser um ponto de entrada de um procedimento que figure no código para um arquivo, e seja chamado a partir de um outro arquivo. O código relocável de máquina precisa reter a informação da tabela de símbolos para cada localização de dados ou rótulo de instrução que seja referido externamente. Se não soubermos antecipadamente o que será referido, precisaremos incluir, com efeito, toda a tabela de símbolos de montagem como parte do código relocável de máquina.

Por exemplo, o código de (1.7) seria precedido por

```
a     0
b     4
```

Se um arquivo, carregado com o conteúdo de (1.7) se referisse a `b`, então tal referência seria substituída por 4 mais o deslocamento pelo qual as localizações no arquivo (1.7) fossem relocadas.

1.5 O AGRUPAMENTO DAS FASES

A discussão das fases na Seção 1.3 lida com a organização lógica do compilador. Numa implementação, as atividades de mais de uma fase são freqüentemente agrupadas.

Interfaces de Vanguarda e Retaguarda

Freqüentemente, as fases são coletadas numa interface de *vanguarda* ou de *retaguarda*. A interface de vanguarda consiste naquelas fases, ou partes de fases, que dependem primariamente da linguagem fonte e são amplamente independentes da máquina alvo. Dentre essas fases são normalmente incluídas a análise léxica e a sintática, a criação da tabela

de símbolos, a análise semântica, e a geração do código intermediário. Uma certa quantidade de otimizações de código pode ser feita igualmente pela interface de vanguarda. A interface de vanguarda também inclui o tratamento de erros que está associado a essas fases.

A interface de retaguarda inclui aquelas partes do compilador que dependem da máquina alvo e que, geralmente, não dependem da linguagem fonte, tão-só da linguagem intermediária. Na interface de retaguarda encontramos alguns aspectos das fases de otimização e de geração de código, juntamente com as operações de tratamento de erro e manipulação da tabela de símbolos necessárias.

Tem se tornado uma praxe tomar a interface de vanguarda de um compilador e refazer sua interface de retaguarda associada, de forma a produzir um compilador para a mesma linguagem fonte numa máquina diferente. Se a interface de retaguarda tiver sido projetada cuidadosamente, pode nem ser mesmo necessário reprojetar muito de sua interface de retaguarda; esse assunto é discutido no Capítulo 9. É também atraente recompilar várias diferentes linguagens na mesma linguagem intermediária e usar uma interface de retaguarda comum para as diferentes interfaces de vanguarda, obtendo vários compiladores para a mesma máquina. No entanto, em decorrência das diferenças sutis de enfoques nas diferentes linguagens, tem havido um sucesso apenas limitado nessa direção.

Passagens

Várias fases da compilação são usualmente implementadas numa única *passagem,* consistindo na leitura de um arquivo de entrada e da escrita de um arquivo de saída. Na prática, existe grande variação na forma em que as fases de um compilador são agrupadas em passagens e, por conseguinte, preferimos organizar nossa discussão da compilação em torno das fases, ao invés das passagens. O Capítulo 12 discute alguns compiladores representativos e menciona a forma com que foram estruturadas as fases em passagens.

Como mencionamos, é comum que várias fases sejam agrupadas numa única passagem e que as atividades dessas fases estejam entremeadas durante a mesma. Por exemplo, a análise léxica, a análise sintática, a análise semântica e a geração de código intermediário poderiam ser agrupadas numa passagem. Se assim o forem, o fluxo de *tokens* após a análise léxica pode ser traduzido diretamente em código intermediário. Mais detalhadamente, podemos pensar no analisador sintático como sendo "a chefia". O analisador sintático tenta descobrir a estrutura gramatical nos *tokens* que enxerga; obtém os *tokens,* à medida que deles necessita, através de chamadas ao analisador léxico, a fim de que encontre o próximo *token.* À medida que a estrutura gramatical é descoberta, o analisador sintático chama o gerador de código intermediário para realizar a análise semântica e gerar uma parte do código. Um compilador organizado dessa forma é apresentado no Capítulo 2.

Reduzindo o Número de Passagens

É desejável se ter relativamente poucas passagens, dado que toma tempo ler e gravar arquivos intermediários. Por outro lado, se agrupamos várias fases numa única passagem, podemos ser forçados a manter todo o programa na memória, porque uma fase pode precisar de informações numa ordem diferente da que a fase anterior produziu. A forma interna do programa pode ser consideravelmente maior do que o programa fonte e também do que o programa alvo e, dessa forma, esse espaço não deve ser considerado um assunto trivial.

Para algumas fases, o agrupamento em uma passagem apresenta uns poucos problemas. Por exemplo, como mencionado acima, a interface entre o analisador léxico e o sintático pode ser freqüentemente limitada a um único *token.* Por outro lado, é freqüentemente muito difícil realizar a geração de código antes que a representação intermediária tenha sido completamente gerada. Por exemplo, linguagens como PL/I e Algol 68 permitem que as variáveis sejam usadas antes de serem declaradas. Não podemos gerar o código alvo para uma construção se não conhecemos os tipos das variáveis envolvidas na mesma. Similarmente, a maioria das linguagens permite desvios que saltem para adiante no código. Não podemos determinar o endereço alvo de tais saltos até que tenhamos visto o código fonte interveniente e o código alvo gerado para o mesmo.

Em alguns casos, é possível deixar um espaço vazio para a informação ausente e preenchê-lo quando a mesma se tornar disponível. Em particular, a geração do código intermediário e do código alvo podem ser freqüentemente combinadas numa única passagem usando-se uma técnica chamada de "retrocorreção". Conquanto não possamos explicar todos os detalhes até que tenhamos visto a geração de código intermediário no Capítulo 8, podemos ilustrar a retrocorreção em termos de um montador. Relembremos que, na secção anterior, discutimos um montador de duas passagens, onde na primeira eram determinados todos os identificadores que representavam localizações de memória e se deduziam seus endereços à medida que fossem descobertos. Uma segunda passagem substituía, então, os identificadores pelos endereços.

Podemos combinar a ação das passagens como se segue. Ao se encontrar um enunciado de montagem que seja uma referência posterior, digamos

```
GOTO alvo
```

geramos o esqueleto de uma instrução, com a operação de máquina para o `GOTO` e espaços para o endereço. Todas as instruções com espaços em branco para os endereços de `alvo` são mantidas numa lista associada à entrada da tabela de símbolos para `alvo`. Os espaços em branco serão preenchidos quando finalmente encontrarmos uma instrução tal como

```
alvo: MOV valor, R1
```

e determinarmos o valor de `alvo`; é o endereço da instrução corrente. Retrocorrigimos, então, percorrendo a lista para `alvo`, para todas as instruções que necessitem de seu endereço, substituindo pelo endereço de `alvo` os espaços em branco dos campos de endereço daquelas instruções. Essa abordagem é fácil de implementar se as instruções puderem ser mantidas na memória até que todos os endereços alvo sejam determinados.

Esse enfoque é razoável para um montador que possa manter toda a sua saída na memória. Como as representações finais de código para um montador são grosseiramente as mesmas e certamente de tamanho aproximadamente igual, a retrocorreção sobre o tamanho de todo o programa de montagem não é inviável. Entretanto, num compilador com um código intermediário consumidor de espaço, podemos precisar nos precaver quanto à distância sobre a qual a retrocorreção atua.

1.6 FERRAMENTAS PARA A CONSTRUÇÃO DE COMPILADORES

O escritor de um compilador, como qualquer outro programador, pode usar, vantajosamente, ferramentas de *software,* tais como depuradores, gerenciadores de versões, customizadores e assim por diante. No Capítulo 11, veremos como algumas delas podem ser usadas para implementar um compilador. São mencionadas apenas brevemente nesta secção; são cobertos, em detalhe, nos capítulos apropriados.

Logo após a escrita dos primeiros compiladores, surgiram os sistemas para auxiliar esse processo. Foram freqüentemente referidos como *compiladores de compiladores, geradores de compiladores e sistemas de escrita de tradutores.* São amplamente orientados em torno de um modelo particular de linguagem e mais adequados para gerar compiladores de linguagens similares ao modelo.

Por exemplo, é tentador assumir que os analisadores léxicos sejam essencialmente os mesmos para todas as linguagens, exceto para

as palavras-chave e símbolos reconhecidos. Muitos compiladores de compiladores produzem realmente rotinas de análise léxica fixas para uso no compilador gerado. Essas rotinas diferem somente na lista de palavras-chave reconhecida e essa lista precisa ser fornecida pelo usuário. O enfoque é válido, mas pode se tornar inviável se for requerido reconhecer *tokens* não padrão, tais como identificadores que possam incluir certos caracteres além de letras e dígitos.

Algumas ferramentas gerais foram criadas para o projeto automático de componentes específicos do compilador. Essas ferramentas usam linguagens especializadas para especificar e implementar o componente e muitas usam algoritmos um tanto sofisticados. As ferramentas de maior sucesso são aquelas que escondem os detalhes do algoritmo de geração e produzem componentes que podem ser facilmente integrados à parte restante do compilador. O que se segue é uma lista de algumas ferramentas úteis para a construção de compiladores:

1. *Geradores de analisadores gramaticais.* Produzem analisadores sintáticos, normalmente a partir de entrada baseada numa gramática livre de contexto. Nos primeiros compiladores, a análise sintática consumia uma grande parte não só do tempo de execução de um compilador mas, também, do esforço intelectual para se escrevê-lo. Essa fase é agora considerada uma das mais fáceis de se implementar. Muitas das "pequenas linguagens" usadas para composição de tipos deste livro, tais como PIC (Kernigham [1982]) e EQN foram implementadas nuns poucos dias usando-se o gerador de *parsers* descrito na Seção 4.7. Muitos geradores de *parsers* usam algoritmos de análise gramatical que são muito complexos para serem realizados a mão.

2. *Geradores de analisadores léxicos.* Geram automaticamente analisadores léxicos, normalmente a partir de uma especificação baseada em expressões regulares, discutidas no Capítulo 3. A organização básica do analisador léxico resultante é, com efeito, um autômato finito. Um gerador de *scanners* típico e sua implementação são discutidos nas Seções 3.5 e 3.8.

3. *Dispositivos de tradução dirigida pela sintaxe.* Produzem coleções de rotinas que percorrem uma árvore gramatical, tal como a da Fig. 1.4, gerando código intermediário. A idéia básica é que uma ou mais "traduções" sejam associadas a cada nó da árvore gramatical e que cada tradução seja definida em termos das traduções de seus nós vizinhos na árvore. Tais dispositivos são discutidos no Capítulo 5.

4. *Geradores automáticos de código.* Tal ferramenta toma uma coleção de regras que definem a tradução de cada operação da linguagem intermediária para a linguagem de máquina da máquina alvo. Tais regras precisam incluir detalhamento suficiente para que possamos lidar com os diferentes métodos de acesso possíveis para os dados; por exemplo, as variáveis podem estar em registradores, numa localização fixa (estática) de memória ou podem ser reservadas numa posição de uma pilha. A técnica básica é a de "correspondência de gabaritos". Os enunciados do código intermediário são substituídos por "gabaritos" que representam as seqüências das instruções de máquina, de uma forma tal que as suposições sobre o armazenamento das variáveis se correspondem de gabarito a gabarito. Como existem usualmente muitas opções relacionadas a onde as variáveis devam ser colocadas (por exemplo, em um dentre vários registradores ou na memória), existem diversas maneiras de se "moldar" o código intermediário com um dado conjunto de gabaritos, e é necessário selecionar uma boa moldagem sem uma explosão combinatória no tempo de execução do compilador. As ferramentas dessa natureza são cobertas no Capítulo 9.

5. *Dispositivos de fluxo de dados.* Muito da informação necessitada para se realizar uma boa otimização de código envolve a "análise do fluxo de dados", capturando-se a informação sobre como os valores são transmitidos de uma parte do programa para outra. As diferentes tarefas dessa natureza podem ser realizadas essencialmente pela mesma rotina, com o usuário fornecendo os detalhes do relacionamento entre os enunciados do código intermediário e a informação sendo capturada. Uma ferramenta dessa natureza é descrita na Secção 10.11.

NOTAS BIBLIOGRÁFICAS

Ao escrever, em 1962, sobre a história da escrita dos compiladores, Knuth [1962] observou que: "Nesse campo tem havido uma quantidade de inusitada de descobertas paralelas da mesma técnica, por pessoas trabalhando independentemente." Continuou, observando que vários indivíduos isolados haviam, de fato, descoberto "vários aspectos de uma técnica e que a mesma foi polida através dos anos num algoritmo muito agradável, que nenhum dos originadores concretizou totalmente". Reivindicar a autoria intelectual de técnicas permanece uma tarefa perigosa; as notas bibliográficas servem aqui meramente como um auxílio para estudo posterior da literatura.

As notas históricas no desenvolvimento das linguagens de programação e compiladores até a chegada de Fortran podem ser encontradas em Knuth e Trabb Pardo [1977]. Wexelblat [1981] contém depoimentos históricos, sobre várias linguagens de programação, fornecidos por participantes de seus desenvolvimentos.

Alguns artigos iniciais, fundamentais na compilação, foram coletados em Rosen [1967] e Pollack [1972]. A edição de janeiro de 1961 do *Communications of ACM* fornece um retrato do estado da arte da escrita de compiladores àquela época. Uma prestação de contas detalhada de um compilador Algol inicial é feita por Randell e Russell [1964].

Começando ao início dos anos 60, com o estudo da sintaxe, os estudos teóricos tiveram uma profunda influência no desenvolvimento da tecnologia de compiladores e, talvez, no mínimo, tanta influência quanto em qualquer outra área da ciência da computação. O aspecto fascinante da sintaxe há muito tempo se desvaneceu, mas a compilação como um todo continua a ser assunto de uma pesquisa viva. Os frutos dessa pesquisa se tornarão evidentes quando examinarmos a compilação em mais detalhes nos próximos capítulos.

CAPÍTULO 2

UM COMPILADOR SIMPLES DE UMA PASSAGEM

Este capítulo é uma introdução ao material existente nos Capítulos 3 a 8 deste livro. Apresenta um número de técnicas de compilação que são ilustradas pelo desenvolvimento de um programa C executável, que traduz expressões infixas para a forma posfixa. Aqui, a ênfase está nos módulos da vanguarda do compilador, ou seja, na análise léxica, na gramatical e na geração do código intermediário. Os Capítulos 9 e 10 cobrem a geração de código e a otimização.

2.1 VISÃO GERAL

Uma linguagem de programação pode ser definida pela descrição da aparência de seus programas (a *sintaxe* da linguagem) e do que os mesmos significam (a *semântica* da linguagem). Para especificar a sintaxe de uma linguagem, apresentamos uma notação amplamente aceita, chamada de gramática livre de contexto ou BNF (para Forma de Backus-Naur). Com as notações correntemente disponíveis, a semântica é muito mais difícil de se descrever do que a sintaxe. Conseqüentemente, para especificar a semântica de uma linguagem usaremos descrições informais e exemplos sugestivos.

Além de especificar a sintaxe da linguagem, uma gramática livre de contexto pode ser usada como auxílio para guiar a tradução de programas. Uma técnica de compilação, orientada por gramáticas, conhecida como *tradução dirigida pela sintaxe*, é de muita ajuda na organização das partes da vanguarda do compilador e será usada extensivamente ao longo deste capítulo.

No curso da discussão da tradução dirigida pela sintaxe, iremos construir um compilador que traduz expressões infixas para a forma posfixa, uma notação na qual os operadores figuram após seus operandos. Por exemplo, a forma posfixa da expressão `9-5+2` é `95-2+`. A notação posfixa pode ser convertida diretamente no código de um computador que realize todas as suas operações usando uma pilha. Começamos pela construção de um programa simples para traduzir na forma posfixa expressões consistindo em dígitos separados por sinais de adição e subtração. À medida que as idéias se tornem mais claras, estenderemos o programa de forma a tratar das construções mais gerais das linguagens de programação. Cada um de nossos tradutores é formado pela ampliação sistemática do tradutor anterior.

Em nosso compilador, o *analisador léxico* converte o fluxo de caracteres de entrada num fluxo de *tokens* que se torna a entrada para a fase seguinte, como mostrado na Fig. 2.1. O "tradutor dirigido pela sintaxe," na figura, é uma combinação de um analisador sintático e de um gerador de código intermediário. Uma razão para se começar com expressões consistindo em dígitos e operadores é a de tornar a análise léxica inicialmente bastante simples; cada caractere de entrada forma um único *token*. Mais tarde, ampliaremos a linguagem de forma a incluir construções tais como números, identificadores e palavras-chave. Para essa linguagem expandida, construiremos um analisador léxico que colete caracteres consecutivos de entrada nos *tokens* apropriados. A construção de analisadores léxicos será discutida em detalhes no Capítulo 3.

2.2 DEFINIÇÃO DA SINTAXE

Nesta secção, introduzimos uma notação chamada gramática livre de contexto (gramática, de forma simplificada), para especificar a sintaxe da linguagem. Será usada através deste livro como parte da especificação da vanguarda de um compilador.

Uma gramática descreve naturalmente a estrutura hierárquica de muitas construções das linguagens de programação. Por exemplo, um comando *if-else*, em C, possui a forma

if (expressão) comando **else** comando

Ou seja, o comando é uma concatenação da palavra-chave **if,** um parênteses à esquerda, uma expressão, um parênteses à direita, um comando, a palavra-chave **else** e um outro comando. (Em C, não há a palavra-chave **then.**) Usando-se a variável *expr* a fim de denotar uma expressão e a variável *cmd* para um comando (ou enunciado), esta regra de estruturação pode ser expressa como

$$cmd \rightarrow \textbf{if}\ (expr)\ cmd\ \textbf{else}\ cmd \qquad (2.1)$$

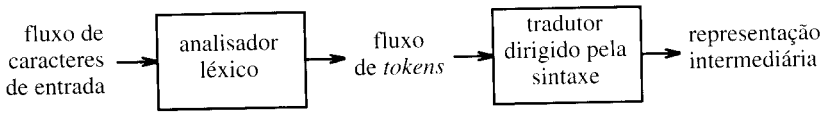


Fig. 2.1. Estrutura da vanguarda de nosso compilador.

onde a seta deve ser lida como "pode ter a forma". Tal regra é chamada de uma *produção*. Numa produção, os elementos léxicos, como a palavra-chave **if** e os parênteses, são chamados de *tokens*. As variáveis como *expr* e *cmd* representam seqüências de *tokens* e são chamadas de *não-terminais*.

Uma *gramática livre de contexto* possui quatro componentes:

1. Um conjunto de *tokens*, conhecidos como símbolos *terminais*.
2. Um conjunto de não-terminais.
3. Um conjunto de produções, onde uma produção consiste em um não-terminal, chamado de *lado esquerdo* da produção, uma seta e uma seqüência de *tokens* e/ou não-terminais, chamados de *lado direito* da produção.
4. Uma designação a um dos não-terminais como o *símbolo de partida*.

Seguimos a convenção de especificar gramáticas pela listagem de suas produções, com aquelas para o símbolo de partida figurando à frente das demais. Assumimos que os dígitos, os sinais, tais como <=, e as cadeias de caracteres em negrito, como **while,** sejam terminais. Um nome em itálico é um não-terminal e qualquer nome ou símbolo que não esteja em itálico deve ser assumido como um *token*.[1] Por uma conveniência de notação, as produções com o mesmo não-terminal à esquerda podem ter todos os seus lados direitos agrupados, com os diferentes lados alternativos à direita separados pelo símbolo |, o qual é lido como "ou".

Exemplo 2.1. Vários exemplos neste capítulo usam expressões constituídas de dígitos e sinais de mais e menos, como em 9−5+2, 3−1 e 7. Como um sinal de mais ou de menos precisa figurar entre dois dígitos, referimo-nos a tais expressões como "listas de dígitos separados por sinais de mais ou de menos". A gramática seguinte descreve a sintaxe dessas expressões. As produções são:

$$lista \rightarrow lista + dígito \quad (2.2)$$
$$lista \rightarrow lista - dígito \quad (2.3)$$
$$lista \rightarrow dígito \quad (2.4)$$
$$dígito \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \quad (2.5)$$

Os lados direitos das três produções, com o não-terminal *lista* à esquerda, poderiam ser equivalentemente agrupados:

$$lista \rightarrow lista + dígito \mid lista - dígito \mid dígito$$

De acordo com nossas convenções, os *tokens* da gramática são os símbolos

+ − 0 1 2 3 4 5 6 7 8 9

Os não-terminais são os nomes em itálico *lista* e *dígito*, com *lista* sendo o não-terminal de partida porque suas produções são fornecidas primeiro □

Dizemos que uma produção é *para um não-terminal* se o último figurar no lado esquerdo da primeira. Uma cadeia de *tokens* é uma seqüência de zero ou mais *tokens*. A cadeia contendo zero *tokens*, escrita ϵ, é chamada de *cadeia vazia*.

Uma gramática deriva cadeias começando pelo símbolo de partida e, então, substituindo repetidamente um não-terminal pelo lado direito de uma produção para aquele não-terminal. As cadeias de *tokens* que podem ser derivadas a partir do símbolo de partida formam a *linguagem* definida pela gramática.

[1] As letras individuais em itálico serão usadas para propósitos adicionais quando as gramáticas forem estudadas em detalhes no Capítulo 4. Por exemplo, usaremos *X*, *Y* e *Z* para falar sobre um símbolo que seja um *token* ou um não-terminal. Entretanto, qualquer nome em itálico contendo dois ou mais caracteres continuará a representar um não-terminal.

Exemplo 2.2. A linguagem definida pela gramática do Exemplo 2.1 consiste em listas de dígitos separados por sinais de mais e de menos.

As dez produções para o não-terminal *dígito* permitem que o mesmo figure em lugar de quaisquer dos *tokens* 0, 1, ..., 9. Pela produção (2.4), um único dígito é, por si mesmo, uma lista. As produções (2.2) e (2.3) expressam o fato de que se tomarmos uma lista e a seguirmos por um sinal de mais ou de menos e, daí, por um outro dígito, teremos uma nova lista.

Segue-se que as produções (2.2) a (2.5) são tudo o que necessitamos para definir a linguagem na qual estamos interessados. Por exemplo, podemos deduzir que 9−5+2 é uma *lista*, como se segue.

a) 9 é uma *lista* pela produção (2.4), pois 9 é um *dígito*.
b) 9−5 é uma *lista* pela produção (2.3), pois 9 é uma *lista* e 5 é um *dígito*.
c) 9−5+2 é uma *lista* pela produção (2.2), pois 9−5 é uma *lista* e 2 é um *dígito*.

Essa sustentação está ilustrada pela árvore da Fig. 2.2. Cada nó da árvore está rotulado por um símbolo da gramática. Um nó interior e seus filhos correspondem a uma produção. O nó interior corresponde ao lado esquerdo da produção, os filhos, ao lado direito. Tais árvores são chamadas de árvores gramaticais e são discutidas abaixo.

Exemplo 2.3. Um tipo um tanto distinto de lista é a seqüência de comandos, separados por ponto e vírgula, encontrada nos blocos *begin-end* de Pascal. Uma nuance de tais listas está em que uma lista vazia de comandos pode ser encontrada entre os *tokens* **begin** e **end.** Podemos começar a desenvolver uma gramática para blocos *begin-end* através da inclusão das produções:

$$bloco \rightarrow \textbf{begin}\ cmds_opcs\ \textbf{end}$$
$$cmds_opcs \rightarrow lista_cmds \mid \epsilon$$
$$lista_cmds \rightarrow lista_cmds;\ cmd \mid cmd$$

Note que o segundo possível lado direito para *cmds_opcs* é ϵ, o que significa a cadeia vazia de símbolos. Ou seja, *cmds_opcs* pode ser substituído pela cadeia vazia e, por conseguinte, um *bloco* pode consistir somente na cadeia de *tokens* **begin end.** Note que as produções para *lista_cmds* são análogas àquelas para *lista* no Exemplo 2.1, com o ponto e vírgula no lugar do operador aritmético e *cmd* no lugar de *dígito*. Não exibimos as produções para *cmd*. Brevemente discutiremos as produções apropriadas para os vários tipos de comandos, tais como comandos_*if*, comandos de atribuição e assim por diante. □

Árvores Gramaticais

Uma árvore gramatical mostra, pictoricamente, como o símbolo de partida de uma gramática deriva uma cadeia de linguagem. Se um não-terminal *A* possui uma produção $A \rightarrow XYZ$, então uma árvore gramati-

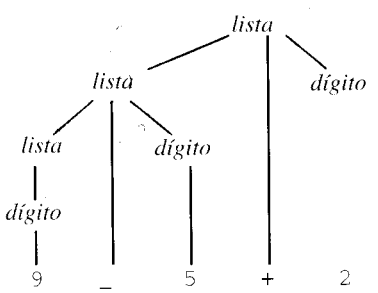
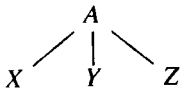


Fig. 2.2. Árvore gramatical para 9−5+2, de acordo com a gramática do Exemplo 2.1.

cal pode ter um nó interior rotulado A, com três filhos rotulados X, Y e Z, da esquerda para a direita:



Formalmente, dada uma gramática livre de contexto, uma *árvore gramatical* possui as seguintes propriedades:

1. A raiz é rotulada pelo símbolo de partida.
2. Cada folha é rotulada por um *token* ou por ϵ.
3. Cada nó interior é rotulado por um não-terminal.
4. Se A é um não-terminal rotulando algum nó interior e $X_1, X_2, ..., X_n$ são os rótulos dos filhos daquele nó, da esquerda para a direita, então $A \rightarrow X_1 X_2 ... X_n$ é uma produção. Aqui, $X_1, X_2, ..., X_n$ figuram no lugar de símbolos que sejam terminais ou não-terminais. Como um caso especial, se $A \rightarrow \epsilon$, então um nó rotulado A deve possuir um único filho rotulado ϵ.

Exemplo 2.4. Na Fig. 2.2, a raiz é rotulada *lista*, o símbolo de partida da gramática do Exemplo 2.1. Os filhos da raiz são rotulados, da esquerda para a direita, *lista*, + e *dígito*. Note que

lista → *lista* + *dígito*

é uma produção na gramática do Exemplo 2.1. O mesmo padrão com − é repetido para o filho à esquerda da raiz, e os três nós rotulados *dígito*, têm, cada um, um filho que é rotulado por um dígito. □

As folhas da árvore gramatical, lidas da esquerda para a direita, formam o *produto* da árvore, que é a cadeia *gerada* ou *derivada* a partir do não-terminal à raiz da árvore gramatical. Na Fig. 2.2, a cadeia gerada é `9-5+2`. Naquela figura, todas as folhas são mostradas no nível mais fundo. Contudo, não iremos caracterizar as folhas dessa forma. Qualquer árvore impõe uma ordem natural, da esquerda para a direita, às suas folhas, baseada na idéia de que se a e b são dois filhos com um mesmo pai e a está à esquerda de b, então todos os descendentes de a estão à esquerda dos descendentes de b.

Outra definição da linguagem gerada por uma gramática é a do conjunto de cadeias que podem ser geradas por alguma árvore gramatical. O processo de encontrar uma árvore gramatical para uma dada cadeia de *tokens* é chamado de *análise gramatical ou análise sintática* daquela cadeia.

Ambigüidade

Temos que ser cuidadosos ao falar sobre *a* estrutura de uma cadeia segundo uma gramática. Conquanto seja claro que cada árvore gramatical dê origem exatamente à cadeia formada por suas folhas, uma gramática pode ter mais de uma árvore gramatical gerando uma dada cadeia de *tokens*. Tal gramática é dita *ambígua*. A fim de mostrar que uma gramática é ambígua, tudo o que precisamos fazer é encontrar uma cadeia de *tokens* que tenha mais de uma árvore gramatical. Como uma

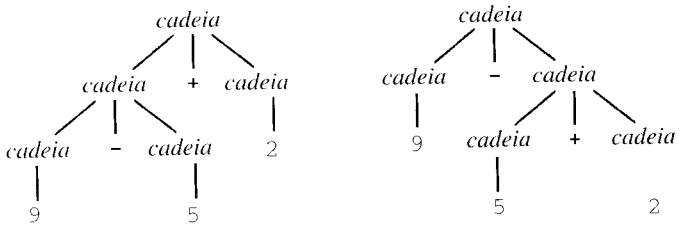


Fig. 2.3. Duas árvores gramaticais para `9-5+2`.

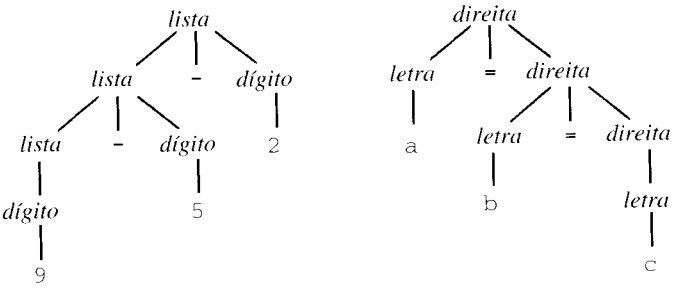


Fig. 2.4. Árvore gramatical para operadores associativos à esquerda e à direita.

cadeia com mais de uma árvore possui usualmente mais de um significado, precisamos projetar, para as aplicações de compilação, gramáticas não ambíguas ou usar gramáticas ambíguas com regras adicionais para resolver as ambigüidades.

Exemplo 2.5. Suponha que no Exemplo 2.1 não fizéssemos distinção entre dígitos e listas. Poderíamos ter escrito a gramática

cadeia → *cadeia* + *cadeia* | *cadeia* − *cadeia* | `0|1|2|3|4|5|6|7|8|9`

Combinarem-se as noções de *dígito* e *lista* no não-terminal *cadeia* faz sentido apenas superficialmente, pois um *dígito* é um caso especial de uma *lista*.

Entretanto, a Fig. 2.3 mostra que uma expressão como `9-5+2` possui, agora, mais de uma árvore gramatical. As duas árvores para `9-5+2` correspondem às duas formas de *parentetizar* a expressão `(9-5)+2` e `9-(5+2)`. Esta segunda forma de parentetização dá à expressão o valor 2, ao invés do valor costumeiro 6. A gramática do Exemplo 2.1 não permitiria essas interpretações. □

Associatividade dos Operadores

Convencionalmente, `9+5+2` é equivalente a `(9+5)+2` e `9-5-2` a `(9-5)-2`. Quando um operando, como `5`, possui operadores à esquerda e à direita, são necessárias convenções para decidir que operador recebe que operando. Dizemos que o operador + *associa à esquerda* porque um operando com sinais de adição em ambos os lados é absorvido pelo operador à sua esquerda. Na maioria das linguagens de programação, os quatro operadores aritméticos — de adição, subtração, multiplicação e divisão — são associativos à esquerda.

Alguns operadores comuns, tais como a exponenciação, são associativos à direita. Como outro exemplo, o operador de atribuição = em C é associativo à direita; em C, a expressão `a=b=c` é tratada da mesma forma que a expressão `a=(b=c)`.

Cadeias, como `a=b=c`, com um operador associativo à direita são geradas pela seguinte gramática:

direita → *letra* = *direita* | *letra*
letra → `a|b|...|z`

O contraste entre uma árvore gramatical para um operador associativo à esquerda, como −, e uma árvore gramatical para um operador associativo à direita, como =, é mostrado na Fig. 2.4. Note que a árvore gramatical para `9-5-2` cresce para baixo em direção à esquerda, enquanto que a de `a=b=c` cresce para baixo e à direita.

Precedência de Operadores

Considere a expressão `9+5*2`. Existem duas possíveis interpretações para a mesma: `(9+5)*2` ou `9+(5*2)`. A associatividade de + e de * não resolve essa ambigüidade. Por esse motivo, precisamos co-

nhecer a precedência relativa dos operadores quando mais de um tipo deles estiver presente.

Dizemos que * possui *precedência mais alta* do que +, se * capturar seus operandos antes de + o fazer. Na aritmética ordinária, a multiplicação e a divisão têm maior precedência que a adição e a subtração. Por conseguinte, `5` é capturado por * tanto em `9+5*2` quanto em `9*5+2`, isto é, as expressões são equivalentes a `9+(5*2)` e a `(9*5)+2`, respectivamente.

Sintaxe das expressões. Uma gramática para expressões aritméticas pode ser construída a partir de uma tabela mostrando a associatividade e a precedência dos operadores. Começamos com os quatro operadores aritméticos comuns e uma tabela de precedência mostrando-os em ordem de precedência crescente, com aqueles no mesmo nível de precedência figurando à mesma linha:

associatividade à esquerda	+ −
associatividade à esquerda	* /

Criamos dois não-terminais, *expr* e *termo*, para os dois níveis de precedência e um não-terminal extra, *fator*, para gerar as unidades básicas das expressões, que são presentemente dígitos e expressões parentetizadas.

$$fator \rightarrow \textbf{dígito} \mid (expr)$$

Consideremos agora os operadores binários * e /, que possuem a precedência mais alta. Como esses operadores associam à esquerda, as produções são similares àquelas para listas que associam identicamente.

$$\begin{aligned} termo &\rightarrow termo * fator \\ &\mid termo \ / \ fator \\ &\mid fator \end{aligned}$$

Similarmente, *expr* gera listas de termos separados pelos operadores aditivos.

$$\begin{aligned} expr &\rightarrow expr + termo \\ &\mid expr - termo \\ &\mid termo \end{aligned}$$

A gramática resultante é, por conseguinte,

$$\begin{aligned} expr &\rightarrow expr + termo \mid expr - termo \mid termo \\ termo &\rightarrow termo * fator \mid termo \ / \ fator \mid fator \\ fator &\rightarrow \textbf{dígito} \mid (expr) \end{aligned}$$

Esta gramática trata uma expressão como uma lista de fatores separados pelos sinais * e /. Note que qualquer expressão parentetizada é fator e, por conseguinte, com parênteses podemos desenvolver expressões que tenham níveis arbitrários de aninhamento (e, também, árvores arbitrariamente profundas).

Sintaxe dos comandos. As palavras-chave nos permitem reconhecer os comandos na maioria das linguagens. Todos os comandos Pascal começam por uma palavra-chave, exceto as atribuições e as chamadas de procedimentos. Alguns comandos Pascal são definidos pela seguinte gramática (ambígua), na qual o *token* **id** representa um identificador.

$$\begin{aligned} cmd \rightarrow\ & \textbf{id} := expr \\ \mid\ & \textbf{if}\ expr\ \textbf{then}\ cmd \\ \mid\ & \textbf{if}\ expr\ \textbf{then}\ cmd\ \textbf{else}\ cmd \\ \mid\ & \textbf{while}\ expr\ \textbf{do}\ cmd \\ \mid\ & \textbf{begin}\ cmds_opcs\ \textbf{end} \end{aligned}$$

O não-terminal *cmds_opcs* gera uma lista possivelmente vazia de comandos separados por ponto e vírgula usando as produções do Exemplo 2.3.

2.3 TRADUÇÃO DIRIGIDA PELA SINTAXE

A fim de traduzir uma construção de linguagem de programação, um compilador pode precisar manter o valor de muitas quantidades além do código gerado para a construção. Por exemplo, o compilador pode necessitar saber o tipo de construção ou a localização da primeira instrução no código-alvo ou o número de instruções geradas. Podemos, conseqüentemente, falar abstratamente sobre os *atributos* associados às construções. Um atributo pode representar qualquer quantidade, por exemplo, um tipo, uma cadeia de caracteres, uma localização de memória ou o que for.

Nesta seção, apresentamos um formalismo, chamado de definição dirigida pela sintaxe, para especificar traduções para construções de linguagens de programação. Uma definição dirigida pela sintaxe especifica a tradução de uma construção em termos dos atributos associados aos seus componentes sintáticos. Em capítulos posteriores, definições dirigidas pela sintaxe são usadas para especificar muitas das traduções que têm lugar na vanguarda de um compilador.

Introduzimos, também, uma notação mais procedimental, chamada de um esquema de tradução, para especificar traduções. Ao longo deste capítulo, usamos esquemas de tradução para passar expressões infixas para a notação posfixa. Uma discussão mais detalhada das definições dirigidas pela sintaxe e suas implementações está contida no Capítulo 5.

Notação Posfixa

A *notação posfixa* para uma expressão E pode ser definida indutivamente como se segue:

1. Se E for uma variável ou uma constante, então a notação posfixa para E será o próprio E.
2. Se E for uma expressão da forma $E_1\ op\ E_2$, onde *op* é qualquer operador binário, então a forma posfixa para E *será* $E_1'\ E_2'\ op$, onde E_1' e E_2' são as notações posfixas para E_1 e E_2, respectivamente.
3. Se E for uma expressão da forma (E_1), então a notação posfixa para E_1 será também a notação posfixa para E.

Os parênteses não são necessários na notação posfixa porque a posição e a aridade (número de argumentos) dos operadores permitem somente uma única decodificação de uma expressão posfixa. Por exemplo, a notação posfixa para `(9-5)+2` é `95-2+` e a notação posfixa para `9-(5+2)` é `952+-`.

Definições Dirigidas pela Sintaxe

Uma *definição dirigida pela sintaxe* usa uma gramática livre de contexto para especificar a estrutura sintática da entrada. A cada símbolo da gramática associa um conjunto de atributos, e a cada produção associa um conjunto de *regras semânticas* para computar os valores dos atributos associados aos símbolos que figuram naquela produção. A gramática e o conjunto de regras semânticas constituem a definição dirigida pela sintaxe.

A tradução é um mapeamento de entrada e saída. A saída para cada entrada x é especificada da seguinte maneira. Primeiro, construa uma árvore gramatical para x. Suponha que um nó n na árvore gramatical seja rotulado pelo símbolo da gramática X. Escrevemos $X.a$ a fim de denotar o valor do atributo a de X àquele nó. O valor de $X.a$ em n é computado usando-se a regra semântica para o atributo a associado com a produção de X usada no nó n. Uma árvore gramatical mostrando os valores dos atributos a cada nó é denominada uma árvore gramatical *anotada*.

Atributos Sintetizados

Um atributo é dito *sintetizado* se seu valor num nó da árvore gramatical é determinado a partir dos valores dos atributos dos filhos daquele nó. Os atributos sintetizados possuem a desejável propriedade de que

PRODUÇÃO	REGRA SEMÂNTICA
$expr \rightarrow expr_1 + termo$	$expr.t := expr_1.t \parallel termo.t \parallel '+'$
$expr \rightarrow expr_1 - termo$	$expr.t := expr_1.t \parallel termo.t \parallel '-'$
$expr \rightarrow termo$	$expr.t := termo.t$
$termo \rightarrow 0$	$termo.t := '0'$
$termo \rightarrow 1$	$termo.t := '1'$
...	...
$termo \rightarrow 9$	$termo.t := '9'$

Fig. 2.5. Definição dirigida pela sintaxe para a tradução da notação infixa para a posfixa.

podem ser avaliados durante um único caminhamento *bottom-up* (do fundo para cima) da árvore gramatical. Neste capítulo, somente atributos sintetizados serão usados; os atributos "herdados" são considerados no Capítulo 5.

Exemplo 2.6. Na Fig. 2.5 é mostrada uma definição dirigida pela sintaxe para traduzir expressões, que consistem em dígitos separados por sinais de mais ou de menos na notação posfixa. Associado a cada não-terminal está um atributo *t*, cujo valor é uma cadeia de caracteres que representa a notação posfixa para a expressão gerada por aquele não-terminal numa árvore gramatical.

A forma posfixa de um dígito é o próprio dígito; por exemplo, a regra semântica associada à produção $termo \rightarrow 9$ define que $termo.t$ seja 9 sempre que essa produção seja usada num nó da árvore gramatical. Quando a produção $expr \rightarrow termo$ é aplicada, o valor de $termo.t$ se torna o valor de $expr.t$.

A produção $expr \rightarrow expr_1 + termo$ dá origem a uma expressão contendo um operador de adição (o subscrito em $expr_1$ distingue a instância de *expr* do lado direito daquela do lado esquerdo). O operando à esquerda do operador de adição é dado por $expr_1$ e o da direita é dado por *termo*. A regra semântica

$$expr.t := expr_1.t \parallel termo.t \parallel '+'$$

associada a esta produção, define o valor do atributo $expr.t$ pela concatenação das formas posfixas $expr_1.t$ e $termo.t$ dos operandos à esquerda e à direita, respectivamente, e então atrelando o sinal de adição. O operador $\parallel$ nas regras semânticas representa a concatenação de cadeias.

A Fig. 2.6 contém a árvore gramatical anotada correspondente à árvore da Fig. 2.2. O valor do atributo-*t* a cada nó foi computado utilizando-se a regra semântica associada à produção usada naquele nó. O valor do atributo à raiz é a notação posfixa para a cadeia gerada pela árvore gramatical. □

Exemplo 2.7. Suponha que um robô possa ser instruído para se mover um passo, numa das direções cardeais norte, sul, leste ou oeste, a partir de sua posição atual. Uma seqüência de tais instruções é gerada pela seguinte gramática:

$$seq \rightarrow seq\ instr \mid \textbf{início}$$
$$instr \rightarrow \textbf{leste} \mid \textbf{norte} \mid \textbf{oeste} \mid \textbf{sul}$$

As mudanças na posição do robô para a entrada

início oeste sul leste leste leste norte norte

são mostradas na Fig. 2.7.

Na figura, uma posição é marcada por um par (x,y), onde x e y representam o número de passos para o leste e norte, respectivamente, a partir da posição inicial. (Se x for negativo, então o robô estará a oeste da posição de partida; similarmente, se y for negativo, então o robô estará ao sul da posição inicial.)

Vamos construir uma definição dirigida pela sintaxe para traduzir uma seqüência de instruções na posição do robô. Iremos usar dois atributos, $seq.x$ e $seq.y$, a fim de monitorar a posição resultante a partir de uma seqüência de instruções gerada pelo não-terminal *seq*. Inicialmente, *seq* gera **início** e tanto $seq.x$ quanto $seq.y$ são, ambos, inicializados em 0 como mostrado no nó interior mais à esquerda da árvore gramatical para **início Oeste Sul,** na Fig. 2.8.

A mudança na posição devido a uma instrução individual derivada a partir de *instr* é dada pelos atributos $instr.dx$ e $instr.dy$. Por exemplo, se *instr* deriva **oeste,** então $instr.dx = -1$ e $instr.dy = 0$. Suponha que uma seqüência *seq* seja formada seguindo-se uma seqüência seq_1 por uma nova instrução *instr*. A nova posição do robô é, então, dada pelas regras

$$seq.x := seq_1.x + instr.dx$$
$$seq.y := seq_1.y + instr.dy$$

Uma definição dirigida pela sintaxe para traduzir uma seqüência de instruções numa posição do robô é mostrada na Fig. 2.9.

Caminhamento em Profundidade

Uma definição dirigida pela sintaxe não impõe qualquer ordem específica para a avaliação dos atributos de uma árvore gramatical; qual-

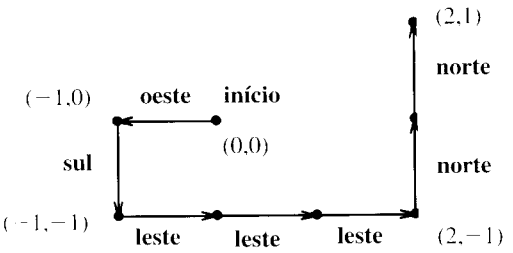


Fig. 2.7. Acompanhando a posição do robô.

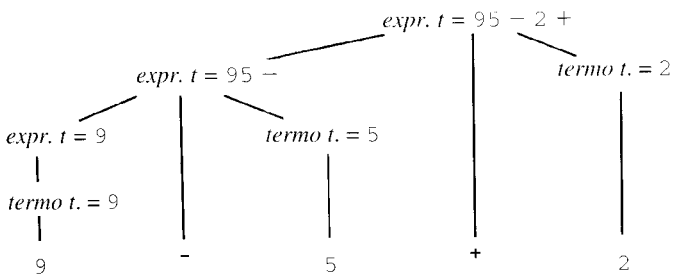


Fig. 2.6. Valores de atributos nos nós de uma árvore gramatical.

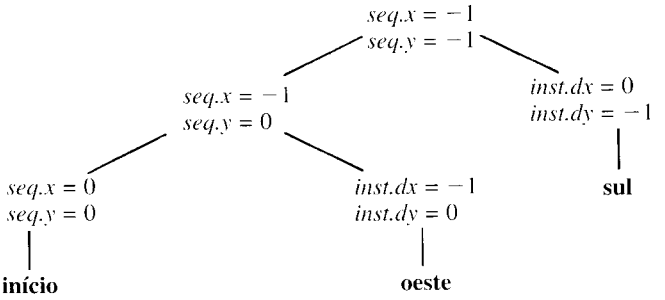


Fig. 2.8. Árvore gramatical anotada para **início oeste sul.**

PRODUÇÃO	REGRAS SEMÂNTICAS
$seq \rightarrow$ **início**	$seq{\cdot}x := 0$ $seq{\cdot}y := 0$
$seq \rightarrow seq_1\ instr$	$seq{\cdot}x := seq_1{\cdot}x + instr{\cdot}dx$ $seq{\cdot}y := seq_1{\cdot}y + instr{\cdot}dy$
$instr \rightarrow$ **leste**	$instr.dx := 1$ $instr{\cdot}dy := 0$
$instr \rightarrow$ **norte**	$instr{\cdot}dx := 0$ $instr{\cdot}dy := 1$
$instr \rightarrow$ **oeste**	$instr{\cdot}dx := -1$ $instr{\cdot}dy := 0$
$instr \rightarrow$ **sul**	$instr{\cdot}dx := 0$ $instr{\cdot}dy := -1$

Fig. 2.9. Definição dirigida pela sintaxe da posição do robô.

quer ordem de avaliação que compute um atributo *a*, após o cômputo de todos os outros atributos dos quais *a* dependa, é aceitável. Em geral, durante o caminhamento da árvore gramatical, alguns atributos terão que ser avaliados quando um nó for atingido pela primeira vez, outros quando todos os seus filhos tiverem sido visitados ou em algum ponto entre as visitas aos filhos daquele nó. As ordens de avaliação mais apropriadas são discutidas em mais detalhes no Capítulo 5.

As traduções neste capítulo podem ser todas implementadas pela avaliação das regras semânticas para os atributos numa árvore gramatical, numa ordem pré-determinada. Um *caminhamento* de uma árvore se inicia à raiz e visita cada nó da mesma em alguma ordem. Neste capítulo, as regras semânticas serão avaliadas usando-se o caminhamento em profundidade definido na Fig. 2.10. Começa na raiz e visita recursivamente os filhos de cada nó numa ordem da esquerda para a direita, como mostrado na Fig. 2.11. As regras semânticas a um dado nó serão avaliadas uma vez que todos os descendentes do mesmo já tenham sido visitados. É chamado de "caminhamento em profundidade" porque visita um filho não visitado de um nó sempre que o mesmo possua um e, então, tenta visitar nós tão distantes da raiz quanto possível, e mais rapidamente possível.

Esquemas de Tradução

No resto deste capítulo usaremos uma especificação procedimental para definir uma tradução. Um *esquema de tradução* é uma gramática livre de contexto na qual fragmentos de programas, chamados de *ações semânticas*, são inseridos nos lados direitos das produções. Um esquema de tradução é como uma definição dirigida pela sintaxe, exceto que a ordem de avaliação das regras semânticas é explicitamente mostrada. A posição à qual uma ação deve ser executada é mostrada envolvendo-a entre chaves e escrevendo-a no lado direito da produção, como em

$$resto \rightarrow +\ termo\ \{imprimir\ ('+')\}\ resto_1$$

procedimento *visitar* (*n*:nó);
início
  para cada filho *m* de *n*, da esquerda para a direita **faça**
    visitar (*m*);
  avaliar as regras semânticas ao nó *n*
fim

Fig. 2.10. Um caminhamento em profundidade de uma árvore.

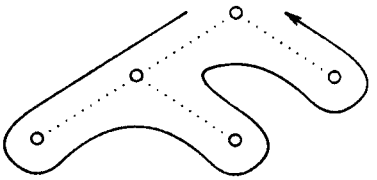


Fig. 2.11. Exemplo de um caminhamento em profundidade de uma árvore.

Um esquema de tradução gera uma saída para cada sentença *x*, que tenha sido gerada pela gramática subjacente, através da execução das ações na ordem em que figurem durante o caminhamento em profundidade de uma árvore gramatical para *x*. Por exemplo, considere uma árvore gramatical com um nó rotulado *resto* representando essa produção. A ação { *imprimir* ('+') } será realizada depois que a subárvore para *termo* seja percorrida, mas antes que o filho para $resto_1$ seja visitado.

Ao desenharmos uma árvore gramatical para um esquema de tradução, indicamos uma ação construindo para a mesma um filho extra, conectado por uma linha pontilhada ao nó de sua produção. Por exemplo, a parte da árvore gramatical para a produção e ação acima é desenhada como na Fig. 2.12. Um nó para uma ação semântica não possui filhos e, então, a ação é realizada quando aquele nó é primeiramente visitado.

Emitindo Uma Tradução

Neste capítulo, as ações semânticas nos esquemas de tradução irão escrever a saída de uma tradução num arquivo, uma cadeia ou um caractere a cada vez. Por exemplo, traduzimos 9−5+2 em 95−2+ imprimindo cada caractere em 9−5+2 exatamente uma vez, sem usar qualquer armazenamento para a tradução de subexpressões. Quando a saída é criada incrementalmente dessa forma, a ordem na qual os caracteres são impressos é importante.

Note que as definições dirigidas pela sintaxe mencionadas até então têm a seguinte importante propriedade: a cadeia representando a tradução do não-terminal ao lado esquerdo de cada produção é a concatenação das traduções dos não-terminais à direita, na mesma ordem que na produção, com algumas cadeias adicionais (talvez nenhuma) entremeadas. Uma definição dirigida pela sintaxe com esta propriedade é denominada *simples*. Por exemplo, considere a primeira produção e regra semântica da definição dirigida pela sintaxe na Fig. 2.5:

PRODUÇÃO	REGRA SEMÂNTICA	
$expr \rightarrow expr_1 + termo$	$expr.t := expr_1.t \,\|\| \, term.t \,\|\| \, '+'$	(2.6)

Aqui, a tradução de *expr.t* é a concatenação das traduções de $expr_1$ e *termo*, seguida pelo símbolo +. Note que $expr_1$ figura antes de *termo* no lado direito da produção.

Uma cadeia adicional aparece entre *termo.t* e $resto_1.t$ em

PRODUÇÃO	REGRA SEMÂNTICA	
$resto \rightarrow +\ termo\ resto_1$	$resto.t := termo.t \,\|\| \, '+' \,\|\| \, resto_1.t$	(2.7)

mas, de novo, o não-terminal *termo* aparece antes de $resto_1$ no lado direito.

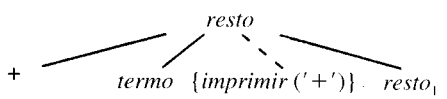


Fig. 2.12. Uma folha extra é construída para uma ação semântica.

expr	→	*expr* + *termo*	{ *imprimir* ('+') }
expr	→	*expr* − *termo*	{ *imprimir* ('−') }
expr	→	*termo*	
termo	→	0	{ *imprimir* ('0') }
termo	→	1	{ *imprimir* ('1') }
	...		
termo	→	9	{ *imprimir* ('9') }

Fig. 2.13. Ações traduzindo expressões na notação posfixa.

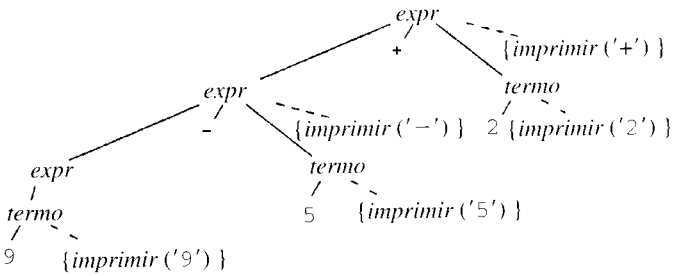


Fig. 2.14. Ações traduzindo 9−5+2 em 95−2+.

Definições simples dirigidas pela sintaxe podem ser implementadas através de esquemas de tradução nos quais as ações imprimam as cadeias adicionais na ordem em que apareçam na definição. As ações nas seguintes produções imprimem as cadeias adicionais em (2.6) e (2.7), respectivamente:

$$expr \rightarrow expr_1 + termo \ \{ \ imprimir\ ('+') \ \}$$
$$resto \rightarrow + \ termo \ \{ \ imprimir\ ('+') \ \} \ resto_1$$

Exemplo 2.8. A Fig. 2.5 continha uma definição simples para traduzir expressões na forma posfixa. Um esquema de tradução a partir dessa definição é dado na Fig. 2.13 e uma árvore gramatical com ações para 9−5+2 é mostrada na Fig. 2.14. Note que apesar das Figs. 2.6 e 2.14 representarem o mesmo mapeamento de entrada e saída, a tradução nos dois casos é construída diferentemente; a Fig. 2.6 atrela a saída à raiz da árvore gramatical, enquanto que na Fig. 2.14 a saída é impressa incrementalmente.

A raiz da Fig. 2.14 representa a primeira produção da Fig. 2.13. Num caminhamento em profundidade, iremos primeiro realizar todas as ações na subárvore para o operando esquerdo *expr*, quando caminharmos na subárvore mais à esquerda da raiz. Visitamos, então, a folha +, na qual não existe ação. Em seguida, realizamos as ações na subárvore para o operando direito *termo* e, finalmente, a ação semântica { *imprimir* ('+') } no nó extra.
Como as produções para *termo* têm somente um dígito no lado direito, aquele dígito é impresso pelas ações para as produções. Nenhuma saída é necessária para a produção *expr* → *termo* e somente o operador necessita ser impresso na ação para as duas produções. Quando executadas durante um caminhamento em profundidade da árvore gramatical, as ações na Fig. 2.14 imprimem 95−2+. □

Como uma regra geral, a maioria dos métodos de análise gramatical processa suas entradas da esquerda para a direita de forma "parcimoniosa"; isto é, constroem o máximo possível da árvore gramatical antes de ler o próximo *token* da entrada. Num esquema de tradução simples (aquele derivado a partir de uma definição simples dirigida pela sintaxe), as ações são também realizadas numa ordem da esquerda para a direita. Conseqüentemente, para se implementar um esquema de tradução simples, podemos executar as ações semânticas enquanto analisamos gramaticalmente; não é necessário construir a árvore gramatical de todo.

2.4 A ANÁLISE GRAMATICAL

A análise gramatical é o processo de se determinar se uma cadeia de *tokens* pode ser gerada por uma gramática. Na discussão do problema, serve como apoio ao raciocínio pensar que uma árvore gramatical está sendo construída, ainda que um compilador não a construa efetivamente. No entanto, um analisador gramatical precisa ser capaz de construir uma árvore ou então a compilação não poderá ser garantida correta.

Esta seção introduz um método de análise gramatical que pode ser aplicado para construir tradutores dirigidos pela sintaxe. Um programa C completo, implementando o esquema de tradução da Fig. 2.13, figura nesta seção. Uma alternativa viável é a de usar uma ferramenta de *software* a fim de gerar um tradutor diretamente a partir de um esquema de tradução. Veja a Seção 4.9 para a descrição de uma tal ferramenta; a mesma pode implementar o esquema de tradução da Fig. 2.13 sem modificações.

Um analisador gramatical pode ser construído para qualquer gramática. As gramáticas usadas na prática, entretanto, possuem uma forma especial. Para qualquer gramática livre de contexto existe um analisador gramatical que toma no máximo um tempo proporcional a $O(n^3)$ para analisar gramaticalmente uma cadeia de *n tokens*. Mas o tempo ao cubo é muito caro. Dada uma linguagem de programação, podemos geralmente construir uma gramática que possa ser analisada de forma rápida. Algoritmos lineares bastam para analisar essencialmente todas as gramáticas que surgem na prática. Analisadores gramaticais de linguagens de programação quase sempre realizam um único esquadrinhamento da esquerda para direita sobre a entrada, procurando por um *token* de cada vez.

A maioria dos métodos de análise gramatical cai em uma dentre duas classes, chamadas de *top-down* e *bottom-up*. Esses termos se referem à ordem na qual os nós da árvore gramatical são construídos. No primeiro, a construção se inicia na raiz e prossegue em direção às folhas, enquanto que no último, a construção se inicia nas folhas e procede em direção à raiz. A popularidade dos analisadores gramaticais *top-down* é devida ao fato de que analisadores eficientes podem ser construídos mais facilmente a mão utilizando-se métodos *top-down*. A análise *bottom-up*, entretanto, pode manipular uma classe mais ampla de gramáticas e esquemas de tradução e, então, as ferramentas de *software* para gerar de analisadores gramaticais, diretamente a partir das gramáticas, tendem a usar os métodos *bottom-up*.

A Análise Gramatical *Top-Down*

Introduzimos a análise gramatical *top-down* considerando uma gramática que é bem adequada a essa classe de métodos. Posteriormente, nesta secção, consideraremos a construção de analisadores gramaticais *top-down* em geral. A seguinte gramática gera um subconjunto dos tipos de Pascal. Usamos o *token* **pontoponto** para enfatizar que a seqüência de caracteres é tratada como uma unidade.

tipo	→	*tipo_simples*	
	|	↑ **id**	
	|	**array** [*tipo_simples*] **of** *tipo*	
tipo_simples	→	**integer**	(2.8)
	|	**char**	
	|	**num pontoponto num**	

A construção *top-down* de uma árvore gramatical é feita iniciando-se pela raiz, rotulada pelo não-terminal de partida e realizando-se, repetidamente, os dois seguintes passos (veja a Fig. 2.15 para um exemplo).

1. Ao nó *n*, rotulado por um não-terminal *A*, selecione uma das produções para *A* e construa os filhos de *n* com os símbolos no lado direito da produção.
2. Encontre o próximo nó no qual uma subárvore deva ser construída.

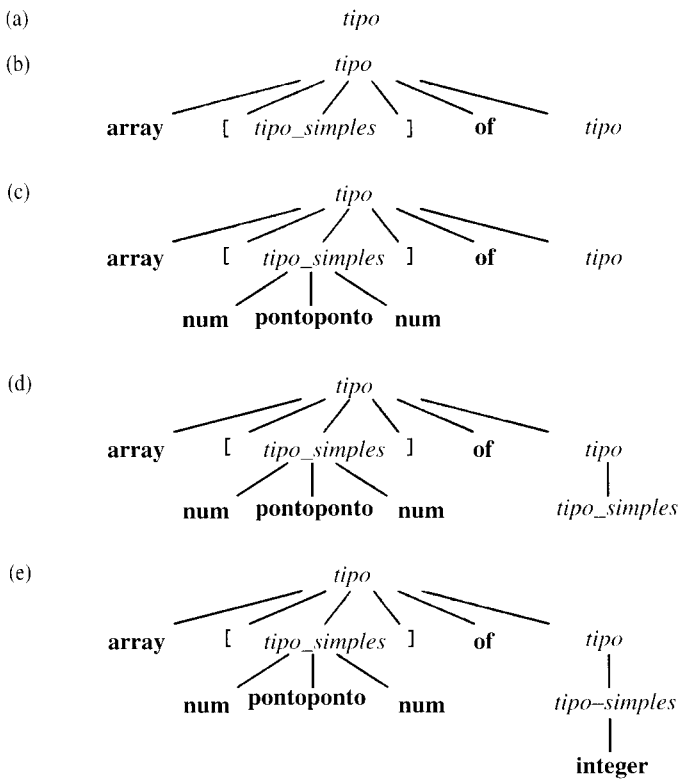


Fig. 2.15. Passos na construção *top-down* de uma árvore gramatical.

Para algumas gramáticas, os passos acima podem ser implementados durante um único esquadrinhamento da cadeia de entrada, da esquerda para a direita. O *token* correntemente esquadrinhado à entrada é freqüentemente referenciado como o símbolo *lookahead*. Inicialmente, o símbolo *lookahead* é o primeiro, isto é, é o *token* mais à esquerda da cadeia de entrada. A Fig. 2.16 ilustra a análise gramatical da cadeia

array [num pontoponto num] of integer

Inicialmente, o *token* **array** é o símbolo *lookahead* e a parte conhecida da árvore gramatical consiste na raiz, rotulada pelo não-terminal de partida *tipo*, na Fig. 2.16(a). O objetivo é construir o resto da árvore gramatical de tal forma que a cadeia gerada pela mesma corresponda à cadeia de entrada.

Para ocorrer uma correspondência, o não-terminal *tipo* na Fig. 2.16 (a) precisa derivar uma cadeia que comece pelo símbolo *lookahead* **array.** Na gramática (2.8) existe somente uma produção para *tipo* que possa derivar uma tal cadeia e, então, a selecionamos e construímos os filhos da raiz rotulados com os símbolos do lado direito da produção.

Cada uma das três fotografias na Fig. 2.16 possui setas marcando o símbolo *lookahead* à entrada e o nó na árvore gramatical em consideração. Quando são construídos os filhos para um nó, consideramos em seguida o filho mais à esquerda. Na Fig. 2.16 (b), os filhos da raiz acabaram de ser construídos e o filho mais à esquerda, rotulado por **array,** está em exame.

Quando o nó em consideração na árvore gramatical é o de um terminal e este último corresponde ao símbolo *lookahead*, avançamos, então, tanto na árvore gramatical quanto na entrada. O próximo *token* à entrada se torna o novo símbolo *lookahead* e o próximo filho na árvore gramatical é considerado. Na Fig. 2.16 (c), a seta na árvore gramatical avançou para o próximo filho da raiz e a seta à entrada avançou para o próximo *token* [. Após o avanço seguinte, a seta na árvore gramatical irá apontar para o filho rotulado pelo não-terminal *tipo_simples*. Quando um nó rotulado por um não-terminal é considerado, repetimos o processo de selecionar uma produção para o não-terminal.

Em geral, a seleção de uma produção para um não-terminal pode envolver tentativa e erro; ou seja, podemos ter que tentar uma produção, retroceder e tentar outra produção, se a primeira se mostrar inadequada. Uma produção é inadequada se, após utilizá-la, não pudermos

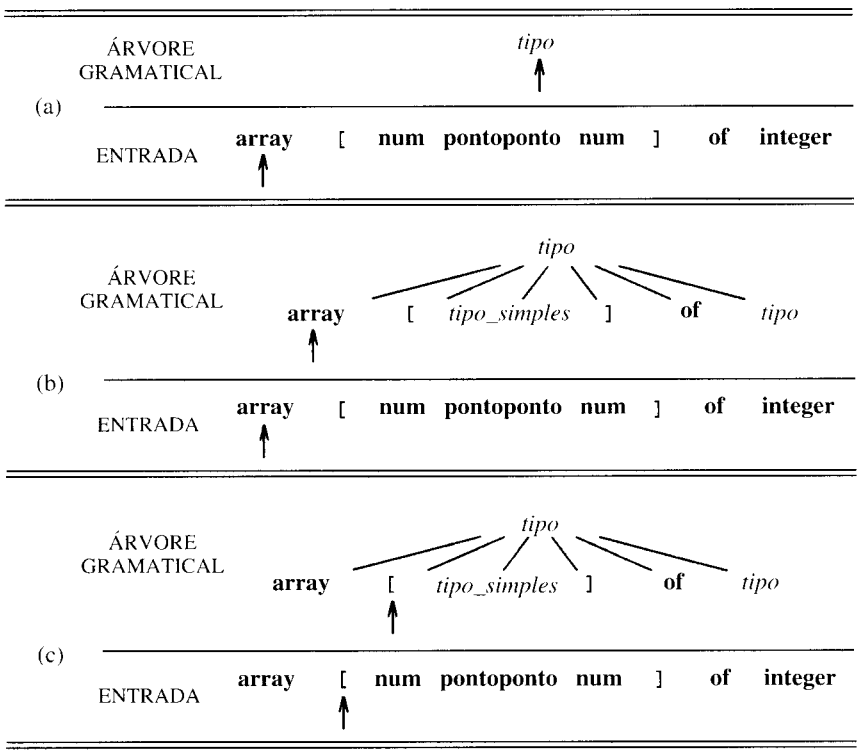


Fig. 2.16. Análise gramatical *top-down* enquanto esquadrinha-se a entrada da esquerda para a direita.

completar uma árvore que corresponda à cadeia de entrada. Existe, no entanto, um caso especial importante, chamado de análise gramatical preditiva, no qual o retrocesso não ocorre.

Análise Gramatical Preditiva

A *análise gramatical descendente recursiva* é um método *top-down* de análise sintática, no qual executamos um conjunto de procedimentos recursivos para processar a entrada. Um procedimento é associado a cada não-terminal de uma gramática. Aqui, consideramos uma forma especial de análise gramatical descendente recursiva, chamada de análise gramatical preditiva, na qual o símbolo *lookahead* determina inambiguamente o procedimento selecionado para cada não-terminal. A seqüência de procedimentos chamada no processamento da entrada define implicitamente uma árvore gramatical para a entrada.

O analisador gramatical preditivo da Fig. 2.17 consiste em procedimentos para os não-terminais *tipo* e *tipo_simples* da gramática (2.8) e um procedimento adicional. Usamos *reconhecer* a fim de simplificar o código para *tipo* e *tipo_simples; reconhecer* avança para o próximo *token* de entrada se seu argumento *t* for igual ao símbolo *lookahead.* Dessa forma, reconhecer modifica a variável *lookahead,* que contém o *token* de entrada correntemente esquadrinhado.

A análise gramatical começa com uma chamada para o procedimento correspondente ao não-terminal de partida de nossa gramática, *tipo.* Com a mesma entrada que a da Fig. 2.16, *lookahead* é inicialmente o primeiro *token* **array.** O procedimento *tipo* executa o código a seguir:

reconhecer **(array);** *reconhecer* ('['); *tipo_simples*;
reconhecer (']'); *reconhecer* **(of);** *tipo* (2.9)

correspondendo ao lado direito da produção

$$tipo \rightarrow \textbf{array}\ [\ tipo_simples\ ]\ \textbf{of}\ tipo$$

```
procedimento reconhecer (t : token);
início
        se lookahead = t então
        lookahead := próximo_token
        senão erro
fim;
procedimento tipo;
início
        se lokahead está em { integer, char, num} então
        tipo_simples
        senão se lookahead = ' ↑ ' então início
                reconhecer (' ↑ '); reconhecer (id)
        fim
        senão se lookahead = array então início
                reconhecer (array); reconhecer ('['); tipo simples;
                reconhecer (']'); reconhecer (of); tipo
        fim
        senão erro
fim;
procedimento tipo_simples;
início
        se lookahead = integer
                então reconhecer (integer)
        senão se lookahead = char então
                reconhecer (char)
        senão se lookahead = num então início
                reconhecer (num); reconhecer (pontoponto);
                reconhecer (num)
        fim
        senão erro
fim;
```

Fig. 2.17. Pseudocódigo para um analisador gramatical preditivo.

Note que cada terminal do lado direito é confrontado com o símbolo *lookahead* e que cada não-terminal ao lado direito leva a uma chamada de seu procedimento correspondente.

Com a entrada da Fig. 2.16, após os *tokens* **array** e [terem sido reconhecidos, o símbolo *lookahead* é **num.** Neste ponto, o procedimento *tipo_simples* é chamado e, dentro do mesmo, é executado o código

reconhecer **(num)**; *reconhecer* **(pontoponto)**; *reconhecer* **(num)**

O símbolo *lookahead* guia a seleção da produção a ser usada. Se o lado direito de uma produção se inicia por um *token,* então a mesma pode ser usada quando o símbolo *lookahead* for igual ao *token.* Considere agora um lado direito começando por não-terminal, como em

$$tipo \rightarrow tipo_simples \qquad (2.10)$$

Essa produção é usada se o símbolo *lookahead* puder ser gerado a partir de *tipo_simples.* Por exemplo, durante a execução do fragmento de código (2.9), suponhamos que, quando o controle atingir a chamada de procedimento *tipo,* o símbolo *lookahead* seja **integer.** Não existe produção para *tipo* que comece com o *token* **integer.** No entanto, uma produção para *tipo_simples* o faz, e, por conseguinte, a produção (2.10) é usada fazendo *tipo* chamar *tipo_simples* para o símbolo *lookahead* **integer.**

A análise gramatical preditiva repousa na informação sobre os primeiros símbolos que podem ser gerados pelo lado direito de uma produção. Mais precisamente, seja α o lado direito de uma produção para um não-terminal A. Definimos PRIMEIRO(α) como sendo o conjunto de *tokens* que figuram como os primeiros símbolos de uma ou mais cadeias geradas a partir de α. Se α é ϵ ou puder gerar ϵ, então ϵ também pertence a PRIMEIRO(α).[2] Por exemplo,

PRIMEIRO(*tipo-simples*) = { **integer, char, num** }
PRIMEIRO (↑ **id**) = { ↑ }
PRIMEIRO(**array** [*tipo-simples*] **of** *tipo*) = { **array** }

Na prática, muitos lados direitos de produções começam por *tokens,* simplificando a construção dos conjuntos PRIMEIRO. Um algoritmo para computar os conjuntos PRIMEIRO é dado na Seção 4.4.

Os conjuntos PRIMEIRO devem ser considerados se existem duas produções $A \rightarrow \alpha$ e $A \rightarrow \beta$. A análise gramatical descendente recursiva sem retrocesso requer que PRIMEIRO(α) e PRIMEIRO(β) sejam disjuntos. O símbolo *lookahead* pode, então, ser usado para decidir a produção a usar; se o mesmo estiver em PRIMEIRO(α), então, α é usado. De outra feita, se o símbolo *lookahead* estiver em PRIMEIRO(β), então β será usado.

Quando Usar Produções-ϵ

As produções com ϵ no lado direito requerem tratamento especial. O analisador gramatical descendente recursivo usa uma produção-ϵ como um *default,* quando nenhuma outra puder ser usada. Como exemplo, considere:

$$\begin{aligned} cmd &\rightarrow \textbf{begin}\ cmds_opcs\ \textbf{end} \\ cmds_opcs &\rightarrow lista_cmds \mid \epsilon \end{aligned}$$

Enquanto analisamos gramaticalmente *cmds_opcs,* se o símbolo *lookahead* não estiver em PRIMEIRO (*lista_cmds*), a produção-ϵ, então, é usada. A escolha estará exatamente correta se o símbolo *lookahead*

[2]Uma produção com ϵ no lado direito complica a determinação dos primeiros símbolos gerados por um não-terminal. Por exemplo, se o não-terminal B pode derivar a cadeia vazia e existe uma produção $A \rightarrow BC$, então o primeiro símbolo gerado por C pode, também, ser o primeiro símbolo gerado por A. Se C também puder gerar ϵ, então, ambos, PRIMEIRO (A) e PRIMEIRO (BC) contêm ϵ.

for **end.** Qualquer outro símbolo *lookahead,* que não **end,** irá resultar num erro, detectado durante a análise de *cmd.*

Projetando um Analisador Gramatical Preditivo

Um *analisador gramatical preditivo* é um programa consistindo em um procedimento para cada não-terminal. Cada procedimento realiza duas coisas:

1. Decide que produção usar, através do exame do símbolo *lookahead.* A produção com um lado direito α é usada se o símbolo *lookahead* estiver em PRIMEIRO(α). Se existir um conflito entre dois lados direitos para qualquer símbolo *lookahead,* então não poderemos usar este método de análise com esta gramática. Uma produção com ϵ no lado direito é usada se o símbolo *lookahead* não estiver no conjunto PRIMEIRO para qualquer outro lado direito.

2. O procedimento usa uma produção imitando o lado direito. Um não-terminal resulta numa chamada a um procedimento para o não-terminal e um *token* se igualando ao símbolo *lookahead* resulta na leitura do próximo *token* de entrada. Se, em algum ponto, o *token* na produção não coincidir com o símbolo *lookahead,* um erro é declarado. A Fig. 2.17 é o resultado da aplicação dessas regras à gramática (2.8).

Exatamente como um esquema de tradução, que é formado expandindo-se uma gramática, um tradutor dirigido pela sintaxe pode ser formado estendendo-se um analisador gramatical preditivo. Um algoritmo com esse propósito é dado na Secção 5.5. Para o presente, a seguinte construção limitada é suficiente, porque os esquemas de tradução implementados neste capítulo não associam atributos aos não-terminais:

1. Construa um analisador gramatical preditivo, ignorando as ações nas produções.

2. Copie as ações a partir do esquema de tradução dentro do analisador gramatical. Se uma ação figurar após o símbolo gramatical X na produção p, então a mesma é copiada exatamente após o código que implementa X. De outra feita, se a mesma figurar ao início da produção, então é copiada exatamente antes do código que implementa a produção.

Iremos construir tal tradutor na próxima secção.

Recursão à Esquerda

É possível um analisador gramatical descendente recursivo rodar para sempre. O problema emerge em produções recursivas à esquerda, tais como

$$expr \rightarrow expr + termo$$

quando o símbolo mais à esquerda do lado direito é o mesmo que o não-terminal no lado esquerdo da produção. Suponhamos que o procedimento para *expr* decida aplicar esta produção. O lado direito da produção começa com *expr,* de tal forma que o procedimento *expr* é chamado recursivamente e o analisador roda para sempre. Note que o símbolo *lookahead* muda somente quando um terminal do lado direito é reconhecido. Como a produção começa pelo não-terminal *expr,* nenhuma mudança à entrada tem lugar entre as chamadas recursivas, causando um laço infinito.

Uma produção recursiva à esquerda pode ser eliminada pela reescrita da produção culpada. Considere um não-terminal A com as duas produções

$$A \rightarrow A\alpha \mid \beta$$

onde α e β são seqüências de terminais e não-terminais que não começam por A. Por exemplo, em

$$expr \rightarrow expr + termo \mid termo$$
$$A = expr,\ \alpha = +\ termo\ e\ \beta = termo.$$

O não-terminal A é *recursivo à esquerda* porque a produção $A \rightarrow A\alpha$ possui o próprio A como seu símbolo mais à esquerda, no lado direito. A aplicação repetida dessa produção constrói uma seqüência de α's à direita de A, como na Fig. 2.18(a). Quando A é finalmente substituído por β, temos β seguido por uma seqüência de zero ou mais α's.

O mesmo efeito pode ser atingido, como na Fig. 2.18(b), reescrevendo-se as produções para A da seguinte maneira:

$$\begin{aligned} A &\rightarrow \beta R \\ R &\rightarrow \alpha R \mid \epsilon \end{aligned} \tag{2.11}$$

Aqui, R é um novo não-terminal. A produção $R \rightarrow \alpha R$ é *recursiva à direita* porque a produção para R possui o próprio R como o último símbolo no lado direito. Produções recursivas à direita levam a árvores que crescem para baixo em direção à direita, como na Fig. 2.18 (b). As árvores que crescem para baixo em direção à direita tornam mais difícil traduzir expressões contendo operadores associativos à esquerda, como o sinal de menos. Na próxima secção, entretanto, veremos que a tradução apropriada de expressões na notação posfixa pode ainda ser atendida por um projeto cuidadoso do esquema de tradução, baseado numa gramática recursiva à direita.

No Capítulo 4, consideraremos formas mais gerais de recursão à esquerda e mostraremos como toda recursão à esquerda pode ser eliminada de uma gramática.

2.5 UM TRADUTOR PARA EXPRESSÕES SIMPLES

Usando-se as técnicas das três últimas secções, construiremos agora um tradutor dirigido pela sintaxe, sob a forma de um programa C que tra-

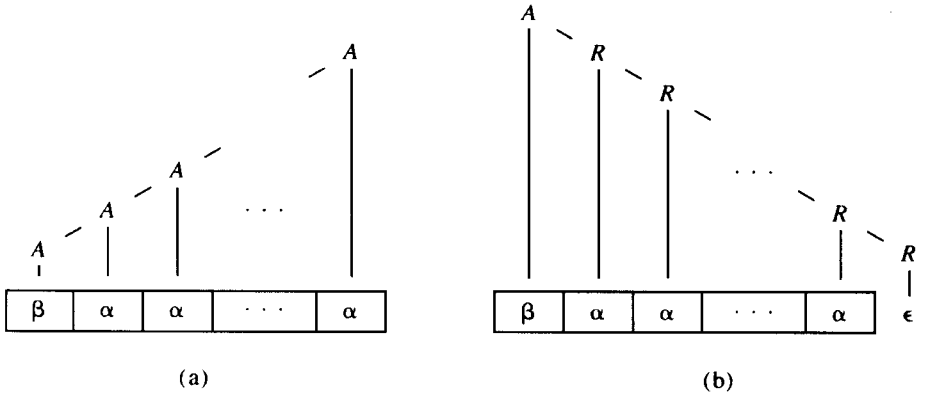


Fig. 2.18. Formas recursivas à esquerda e à direita para gerar uma cadeia.

$$
\begin{array}{lcll}
expr & \rightarrow & expr + termo & \{\ imprimir\ ('+')\ \} \\
expr & \rightarrow & expr - termo & \{\ imprimir\ ('-')\ \} \\
expr & \rightarrow & termo & \\
termo & \rightarrow & 0 & \{\ imprimir\ ('0')\ \} \\
termo & \rightarrow & 1 & \{\ imprimir\ ('1')\ \} \\
 & \ldots & & \\
termo & \rightarrow & 9 & \{\ imprimir\ ('9')\ \}
\end{array}
$$

Fig. 2.19. Especificação inicial de um tradutor da forma infixa para a posfixa.

duz expressões aritméticas para a forma posfixa. A fim de manter o programa inicial gerencialmente pequeno, começamos por expressões constituídas de dígitos separados por sinais de mais e de menos. A linguagem é estendida nas duas próximas seções de forma a incluir números, identificadores e outros operadores. Dado que as expressões figuram como uma construção em muitas linguagens, é valioso estudarmos sua tradução em detalhes.

Um esquema de tradução dirigida pela sintaxe pode freqüentemente servir como a especificação para um tradutor. Usamos o esquema da Fig. 2.19 (repetido da Fig. 2.13) como a definição da tradução a ser realizada. Como freqüentemente é o caso, a gramática subjacente de um dado esquema precisa ser modificada antes que possa ser analisada gramaticalmente por um analisador preditivo. Em particular, a gramática subjacente ao esquema da Fig. 2.19 é recursiva à esquerda e, como vimos na última seção, um analisador preditivo não pode tratar uma gramática recursiva à esquerda. Pela eliminação da recursão à esquerda, podemos obter uma gramática adequada para uso num tradutor preditivo descendente recursivo.

Sintaxe Abstrata e Concreta

Um ponto de partida útil para se pensar sobre a tradução de uma cadeia de entrada é a *árvore sintática abstrata*, na qual cada nó representa um operador e os filhos do nó representam os operandos. Por contraste, uma árvore gramatical é chamada uma *árvore sintática concreta* e a gramática subjacente uma *sintaxe concreta* para a linguagem. Árvores sintáticas abstratas, ou simplesmente *árvores sintáticas*, diferem das árvores gramaticais porque as distinções superficiais de forma, desimportantes para a tradução, não figuram nas árvores sintáticas.

Por exemplo, a árvore sintática para `9-5+2` é mostrada na Fig. 2.20. Como + e − possuem o mesmo nível de precedência e os operadores de mesmo nível de precedência são avaliados da esquerda para a direita, a árvore mostra `9-5` agrupada como uma subexpressão. Comparando a Fig. 2.20 com a árvore gramatical correspondente da Fig. 2.2 notamos que a árvore sintática associa um operador a um nó interior ao invés de fazer do operador um de seus filhos.

É desejável que um esquema de tradução esteja baseado numa gramática cujas árvores gramaticais sejam tão próximas das sintáticas quanto possível. O agrupamento de subexpressões pela gramática da Fig. 2.19 é similar aos agrupamentos na árvore sintática. Infortunadamente, a gramática da Fig. 2.19 é recursiva à esquerda e, conseqüentemente, inadequada à análise gramatical preditiva. Parece existir um conflito: por um lado, precisamos de uma gramática que facilite a análise gramatical; por outro, precisamos de uma gramática radicalmente diferente que facilite a tradução. A solução óbvia é eliminar a recursividade à esquerda. No entanto, isso precisa ser feito cuidadosamente, como o exemplo seguinte mostra.

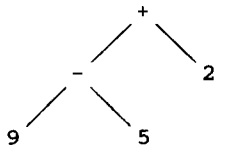


Fig. 2.20. Árvore sintática para `9-5+2`.

Exemplo 2.9. A gramática seguinte é inadequada para a tradução de expressões na forma posfixa, ainda que a mesma gere exatamente a mesma linguagem que a gramática na Fig. 2.19 e possa ser usada para a análise gramatical descendente recursiva.

$$
\begin{array}{rl}
expr \rightarrow & termo\ resto \\
resto \rightarrow & +\ expr \mid -\ expr \mid \epsilon \\
termo \rightarrow & 0 \mid 1 \mid 2 \mid \ldots \mid 9
\end{array}
$$

Esta gramática possui o problema de que os operandos dos operadores gerados por $resto \rightarrow +\ expr$ e $resto \rightarrow -\ expr$ não são óbvios a partir das produções. Nenhuma das seguintes alternativas para formar a tradução de *resto.t* a partir daquela de *expr.t* é aceitável:

$$resto \rightarrow -\ expr \rightarrow resto.t := '-' \parallel expr.t\ \} \qquad (2.12)$$
$$resto \rightarrow -\ expr\ \{\ resto.t := expr.t \parallel '-'\ \} \qquad (2.13)$$

(Mostramos somente a produção e a ação semântica para o operador de −.) A tradução de `9-5` é `95-`. Entretanto, se usarmos a ação em (2.12) então o sinal de menos aparecerá antes de *expr.t* e `9-5` permanece, incorretamente, `9-5` na tradução.

Por outro lado, se usarmos (2.13) e a regra análoga para +, os operadores se movem consistentemente para a direita e `9-5+2` é traduzida incorretamente em `952+-` (a tradução correta é `95-2+`).□

Adaptando o Esquema de Tradução

A técnica de eliminação da recursão à esquerda delineada na Fig. 2.18 também pode ser aplicada às produções contendo ações semânticas. Estendemos a transformação na Seção 5.5 a fim de levarmos os atributos sintetizados em conta. A técnica transforma as produções $A \rightarrow A\alpha \mid A\beta \mid \gamma$ em

$$
\begin{array}{l}
A \rightarrow \gamma R \\
R \rightarrow \alpha R \mid \beta R \mid \epsilon
\end{array}
$$

Quando as ações semânticas estão imiscuídas nas produções, carregamo-las na transformação. Aqui, se fizermos $A = expr$, $\alpha = +\ termo\ \{\ imprimir\ ('+')\ \}$, $\beta = -\ termo\ \{\ imprimir\ ('-')\ \}$ e $\gamma = termo$, a transformação acima produz o esquema de tradução (2.14). As produções de *expr* na Fig. 2.19 foram transformadas, em (2.14), nas produções para *expr* e para o novo não-terminal *resto*. As produções para *termo* são repetidas a partir da Fig. 2.19. Note que a gramática subjacente é diferente daquela do Exemplo 2.9 e que a diferença torna a tradução desejada possível.

$$
\begin{array}{lcll}
expr & \rightarrow & termo\ resto & \\
resto & \rightarrow & +\ termo\ \{\ imprimir\ ('+\ '\ \}\ resto \mid -termo & \\
 & & \{\ imprimir\ ('-'\ \}\ resto \mid \epsilon & \\
termo & \rightarrow & 0\ \ \{\ imprimir\ ('0')\ \} & \qquad (2.14) \\
termo & \rightarrow & 1\ \ \{\ imprimir\ ('1')\ \} & \\
 & \ldots & & \\
termo & \rightarrow & 9\ \ \{\ imprimir\ ('9')\ \} &
\end{array}
$$

A Fig. 2.21 mostra como `9-5+2` é traduzida usando-se a gramática acima.

Procedimentos para os Não-Terminais *expr, termo* e *resto*

Implementamos agora um tradutor em C usando o esquema de tradução dirigida pela sintaxe (2.14). A essência do tradutor é o código C da Fig. 2.22 para as funções `expr`, `termo` e `resto`. Essas funções implementam os não-terminais correspondentes em (2.14).

A função `reconhecer`, apresentada mais tarde, é a contraparte em C do código na Fig. 2.17 para confrontar um *token* com o sím-

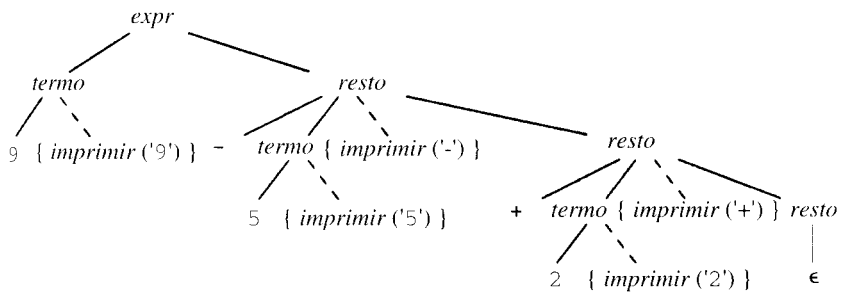


Fig. 2.21. Tradução de 9−5+2 em 95−2+.

bolo *lookahead* e avançar através da entrada. Como cada *token* é um único caractere em nossa linguagem, `reconhecer` pode ser implementada comparando e lendo caracteres.

Para aqueles não familiarizados com a programação na linguagem C, mencionamos as diferenças mais salientes entre C e outras linguagens derivativas de Algol, tais como Pascal, na medida que encontrarmos usos para aquelas figurações de C. Um programa em C consiste em uma seqüência de definições de funções, com a execução começando numa função distinta chamada `main`. Definições de funções não podem ser aninhadas. Parênteses envolvendo listas de parâmetros de funções são necessários mesmo que não existam parâmetros; por conseguinte, escrevemos `expr()`, `termo()` e `resto()`. As funções se comunicam transmitindo parâmetros "por valor" ou tendo acesso a dados globais a todas as funções. Por exemplo, as funções `termo()` e `resto()` examinam o símbolo *lookahead* usando o identificador global `lookahead`.

C e Pascal usam os seguintes símbolos para a atribuição e testes de igualdade:

OPERAÇÃO	C	PASCAL
atribuição	=	:=
teste de igualdade	==	=
teste de desigualdade	!=	<>

```
expr( )
{
    termo( ); resto( );
}

resto ( )
{
    if (lookahead == '+')  {
        reconhecer('+'); termo( ); putchar
        ('+'); resto( );

    {
    else if (lookahead == '-')  {
        reconhecer ('-'); termo( ); putchar
        ('-'); resto( );
    }
    else ;
}

termo( )

{
    if (isdigit (lookahead) )  {
        putchar (lookahead);  reconhecer
        (lookahead);
    }
    else erro( );
}
```

Fig. 2.22. Funções para os não-terminais *expr*, *resto* e *termo*.

As funções para os não-terminais imitam os lados direitos das produções. Por exemplo, a produção para *expr* → *termo resto* é implementada pelas chamadas `termo()` e `resto()` na função `expr()`.

Como um outro exemplo, a função `resto()` usa a primeira produção para *resto* em (2.14), se o símbolo *lookahead* for um sinal de adição, a segunda, se for um sinal de subtração e a produção *resto* → ϵ por *default*. A primeira produção para *resto* é implementada pelo primeiro enunciado *if* na Fig. 2.22. Se o símbolo *lookahead* for +, o sinal de adição é reconhecido pela chamada `reconhecer ('+')`. Após a chamada `termo()`, a rotina da biblioteca-padrão de C `putchar('+')` implementa a ação semântica imprimindo o caractere +. Como a terceira produção para *resto* possui ϵ como lado direito, o último *else* em `resto()` não realiza coisa alguma.

As dez produções para *termo* geram os dez dígitos. Na Fig. 2.22, a rotina `isdigit` testa se o símbolo *lookahead* é um dígito. O dígito é impresso e reconhecido se o teste tiver sucesso; de outra feita, um erro ocorre. (Note que `reconhecer` modifica o símbolo *lookahead*, de tal forma que a impressão deve ocorrer antes do dígito sofrer o reconhecimento.) Antes de mostrar um programa completo, faremos uma transformação para melhorar a velocidade no código da Fig. 2.22.

Otimizando o Tradutor

Certas chamadas recursivas podem ser substituídas por iterações. Quando o último enunciado executado no corpo de um procedimento é uma chamada recursiva para o mesmo procedimento, a mesma é chamada de *epílogo-recursiva*. Por exemplo, as chamadas de `resto()`, ao fim da quarta e sétima linhas da função `resto()`, são *epílogo-recursivas* porque o controle flui para o final do corpo da função após cada uma delas.

Podemos acelerar um programa substituindo a epílogo-recursividade pela iteração. Para um procedimento sem parâmetros, uma chamada epílogo-recursiva pode ser simplesmente substituída por um desvio para o início do procedimento. O código para `resto` pode ser reescrito como:

```
resto()
{
L:      if (lookahead == '+') {
            reconhecer ('+'); termo()
            putchar('+'); goto L;
        }
        else if (lookahead == '-'){
            reconhecer ('-'); termo();
            putchar('-'); goto L;
        }
        else;
}
```

Na medida em que o símbolo *lookahead* seja um sinal de mais ou de menos, o procedimento `resto` reconhece o sinal, chama `termo` a fim de reconhecer um dígito e repete o processo. Note que como reconhecer remove o sinal a cada vez em que é chamado, esse ciclo ocorre somente em seqüências alternantes de sinais e dígitos. Se essa mudança

```
expr( )

{
    termo( );
    while(1)
        if (lookahead == '+')  {
            reconhecer ('+'); termo( );
            putchar('+');
        }
        else if (lookahead == '-')  {
            reconhecer('-'); termo( );
            putchar('-');
        }
        else break;
}
```

Fig. 2.23. Substituição para as funções `expr` e `resto` da Fig. 2.22.

for realizada na Fig. 2.22, a única chamada remanescente para `resto` será proveniente de `expr` (ver linha 3). As duas funções podem conseqüentemente ser integradas numa única, como mostrado na Fig. 2.23. Em C, um enunciado *cmd* pode ser executado repetidamente escrevendo-se

`while(1)` *cmd*

porque a condição 1 é sempre verdadeira. Podemos sair de um laço executando um comando *break*. A forma estilizada do código na Fig. 2.23 permite que outros operadores sejam adicionados convenientemente.

O Programa C Completo

O programa C completo para nosso tradutor é mostrado na Fig. 2.24. A primeira linha, começando com `#include`, carrega `<ctype.h>`, um arquivo de rotinas-padrão que contém o código para o predicado `isdigit`.

Os *tokens*, consistindo em caracteres isolados, são fornecidos pela rotina de biblioteca-padrão `getchar`, que lê o próximo caractere do arquivo de entrada. Entretanto, *lookahead* é declarado ser um inteiro à linha 2 da Fig. 2.24 a fim de antecipar os *tokens* adicionais que não sejam caracteres isolados, os quais serão introduzidos nas seções posteriores. Como *lookahead* é declarada fora de qualquer função, será global a qualquer uma que seja declarada após a linha 2 da Fig. 2.24.

A função `reconhecer` verifica os *tokens*; lê o próximo *token* de entrada se o símbolo *lookahead* for reconhecido e chama uma rotina de erro em caso contrário.

A função `erro` usa a função da biblioteca padrão `printf` a fim de emitir a mensagem `"erro de sintaxe"` e, então, termina a execução através de uma chamada `exit(1)` para uma outra função de biblioteca padrão.

```
#include <ctype.h > /* carrega o arquivo com o predicado isdigit*/ int lookahead;
main( )
{
    lookahead = getchar( );
    expr( );
    putchar ('\n'); /* adiciona caractere de avanço de linha */
}
expr( )
{
    termo( );
    while(1)
        if (lookahead == '+')  {
            reconhecer('+'); termo( ); putchar('+');
        }
        else if (lookahead == '-')  {
            reconhecer('-'); termo( ); putchar('-');
        }
        else break;
}
termo( )

{
    if (isdigit (lookahead))  {
        putchar (lookahead);
        reconhecer (lookahead);
    }
    else erro( );
}

reconhecer(t)
    int(t);
{
    if (lookahead == t)
        lookahead = getchar( );
    else erro( );
}

erro( )
{
    printf("erro de sintaxe\n");  /* imprime mensagem de erro  */
    exit(1);      /* parar */
}
```

Fig. 2.24. Programa C para traduzir uma expressão infixa para a forma posfixa.

2.6 ANÁLISE LÉXICA

Iremos agora adicionar ao tradutor da secção prévia um analisador léxico que lê e converte a entrada para um fluxo de *tokens* a ser analisado pelo *parser*. Relembre que, com base na definição da gramática da Seção 2.2, as sentenças de uma linguagem consistem em cadeias de *tokens*. Uma seqüência de caracteres de entrada que compõe um único *token* é chamada de lexema. Um *scanner* (analisador léxico) pode isolar o *parser* da representação sob a forma de lexema dos *tokens*. Começamos por listar algumas das funções que gostaríamos que um analisador léxico realizasse.

Remoção de Espaços em Branco e Comentários

O tradutor de expressões da última seção examina cada caractere da entrada, de tal forma que caracteres estranhos, como brancos, irão fazer com que o mesmo falhe. Muitas linguagens permitem que "espaços em branco" (brancos, tabulações e avanço de linha) apareçam entre os *tokens*. Os comentários podem igualmente ser ignorados pelo *parser* e tradutor, de tal forma que podem ser tratados também como espaços em branco.

Se um espaço em branco é eliminado pelo analisador léxico, o *parser* jamais terá que considerá-lo. A alternativa de modificar a gramática a fim de incorporar o espaço em branco na sintaxe não é tão facilmente implementável.

Constantes

A qualquer tempo em que um dígito isolado apareça numa expressão, parece razoável admitir uma constante inteira arbitrária no seu lugar. Como uma constante inteira é uma seqüência de dígitos, as constantes inteiras podem ser admitidas quer pela adição à gramática de produções para expressões ou criando um *token* para tais constantes. O trabalho de gerar inteiros através da coleta de dígitos é geralmente entregue a um analisador léxico porque os números podem ser tratados como unidades autônomas durante a tradução.

Seja **num** o *token* que representa um inteiro. Quando uma seqüência de dígitos aparecer no fluxo de entrada, o analisador léxico irá entregar o *token* **num** ao *parser*. O valor de um inteiro será passado junto como um atributo do *token* **num.** Logicamente, o analisador léxico entrega tanto o *token* quanto o atributo ao *parser*. Se escrevermos um *token* e seu atributo como uma tupla envolvida entre < >, a entrada

```
31 + 28 + 59
```

seria transformada na seqüência de tuplas

<**num**, 31> <+, > <**num**, 28> <+, > <**num**, 59>

O *token* + não possui atributos. Os segundos componentes das tuplas, os atributos, não desempenham papel durante a análise gramatical, mas são necessários durante a tradução.

Reconhecendo Identificadores e Palavras-Chave

As linguagens usam identificadores para nomes de variáveis, de *arrays*, e para outros elementos. Uma gramática para uma linguagem freqüentemente trata um identificador como um *token*. Um *parser* baseado numa tal gramática pretende ver o mesmo *token*, digamos **id,** a cada vez que um identificador aparecer na entrada. Por exemplo, a entrada

```
contador = contador + incremento;    (2.15)
```

seria convertida pelo analisador léxico no fluxo de *tokens*

id =id + id; (2.16)

É este fluxo de *tokens* que é usado pela análise gramatical (*parsing*.)

Quando falarmos sobre a análise léxica da linha de entrada (2.15), será útil distinguir entre o *token* **id** e os lexemas `contador` e `incremento`, associados às instâncias desse *token*. O tradutor precisa saber que o lexema `contador` forma as duas primeiras instâncias de **id** em (2.16) e que o lexema `incremento` forma a terceira.

Quando um lexema formando um identificador é examinado à entrada, algum mecanismo é necessário para determinar se o lexema já foi examinado antes. Como mencionado no Capítulo 1, uma tabela de símbolos é usada como tal mecanismo. O lexema é armazenado na tabela de símbolos e o apontador para essa entrada da tabela se torna um atributo do *token* **id.**

Muitas linguagens usam cadeias fixas de caracteres, como `begin`, `end`, `if` etc., como símbolos de pontuação ou para identificar certas construções. Essas cadeias de caracteres, chamadas de *palavras-chave*, geralmente satisfazem as regras para a formação de identificadores, de tal forma que algum mecanismo é necessário para decidir quando um lexema forma uma palavra-chave ou um identificador. O problema é mais fácil de resolver se as palavras-chave são *reservadas*, isto é, se não podem ser usadas como identificadores. Dessa forma, uma cadeia de caracteres forma um identificador somente se não formar uma palavra-chave.

O problema de isolar os *tokens* também emerge se os mesmos caracteres aparecem nos lexemas de mais de um *token*, como em <, <= e <> em Pascal. As técnicas para reconhecer tais *tokens* eficientemente são discutidas no Capítulo 3.

Interface para o Analisador Léxico

Quando um analisador léxico é inserido entre o *parser* e o fluxo de entrada, o mesmo interage com os dois da maneira mostrada na Fig. 2.25: lê os caracteres de entrada, agrupa-os em lexemas e passa os *tokens* formados pelos lexemas, juntamente com os valores dos seus atributos, para os estágios posteriores do compilador. Em algumas situações, o analisador léxico tem que ler alguns caracteres à frente antes de estar apto a decidir a respeito do *token* a ser retornado para o *parser*. Por exemplo, um analisador léxico em Pascal tem que ler à frente caso encontre o caractere >. Se o próximo caractere for =, então a seqüência de caracteres >= é o lexema que forma o *token* para o operador "maior ou igual a". De outra forma, > é o lexema formador do operador "maior do que" e o analisador léxico acabou de ler um caractere a mais. O caractere extra precisa ser empilhado de volta na entrada, pois o mesmo pode ser o início do próximo lexema.

O analisador léxico e o *parser* formam um par *produtor-consumidor*. O analisador léxico produz os *tokens* e o *parser* os consome. Os *tokens* produzidos podem ser guardados num *buffer* até que venham a ser consumidos. A interação entre os dois é restringida somente pelo tamanho do *buffer*, porque o analisador léxico não poderá prosseguir quando o *buffer* estiver cheio e o *parser* quando o *buffer* estiver vazio. Comumente, o *buffer* guarda exatamente um *token*. Nesse caso, a interação pode ser implementada simplesmente fazendo com que o anali-

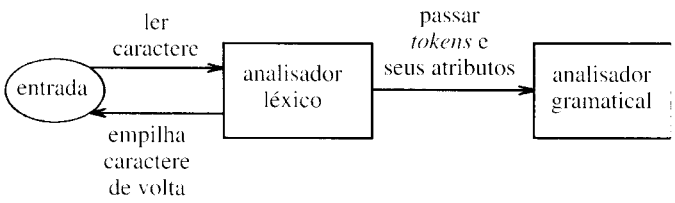


Fig. 2.25. Inserindo um analisador léxico entre a entrada e o *parser*.

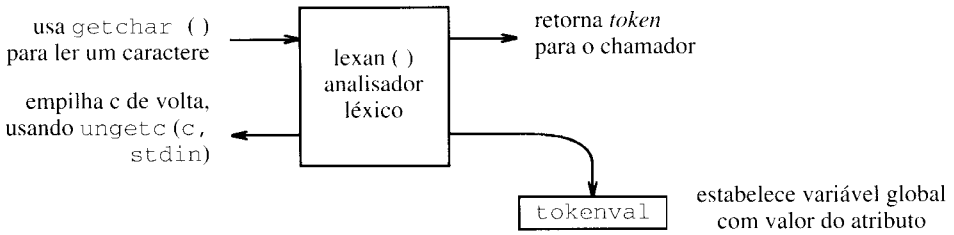


Fig. 2.26. Implementando as interações da Fig. 2.25.

sador léxico seja um procedimento chamado pelo *parser,* retornando os *tokens* por demanda.

A implementação da leitura e devolução dos caracteres é usualmente feita através do estabelecimento de um *buffer* de entrada. Um bloco de caracteres é lido no *buffer* a cada vez; um apontador mantém a posição da porção da entrada que já foi analisada. A devolução dos caracteres é implementada retrocedendo-se o apontador. Os caracteres de entrada também precisam ser salvos para o relatório de erros, pois alguma indicação precisa ser fornecida sobre onde, no texto de entrada, o erro ocorreu. A buferização dos caracteres de entrada pode ser justificada isoladamente no terreno da eficiência. A obtenção de todo um bloco de caracteres é usualmente mais eficiente do que a obtenção de um caractere a cada vez. As técnicas para a buferização da entrada são discutidas na Seção 3.2.

Um Analisador Léxico

Construímos a seguir um analisador léxico rudimentar para o tradutor de expressões da Seção 2.5. O propósito do analisador léxico é o de permitir que espaços em branco e números apareçam dentro das expressões. Na próxima seção, estenderemos o analisador léxico de forma a também aceitar identificadores.

A Fig. 2.26 sugere como o analisador léxico, escrito como a função `lexan` em C, implementa as interações da Fig. 2.25. As rotinas `getchar` e `ungetc`, oriundas do arquivo-padrão de inclusão `<stdio.h>`, tomam conta da buferização da entrada; `lexan` lê e empilha de volta os caracteres da entrada chamando as rotinas `getchar` e `ungetc`, respectivamente. Com `c` declarada como um caractere, o par de enunciados

```
c = getchar(); ungetc (c, stdin);
```

deixa o fluxo de entrada inalterado. A chamada para `getchar()` atribui o próximo caractere de entrada a `c`; a chamada para `ungetc` empilha de volta o valor de `c` na entrada-padrão `stdin`.

Se a linguagem de implementação não permite que estruturas de dados sejam retornadas por funções, então os *tokens* e seus atributos precisam ser passados separadamente. A função `lexan` retorna uma codificação inteira de um *token.* O *token* para um caractere pode ser qualquer codificação inteira convencional daquele caractere. Um *token,* como **num,** pode ser codificado por um valor inteiro maior que qualquer outro que codifique um caractere, digamos `256`. A fim de permitir que a codificação seja modificada facilmente, usamos a constante simbólica `NUM` a fim de referirmo-nos à codificação inteira de **num.** Em Pascal, a associação entre NUM e a codificação pode ser feita por uma declaração **const**; em C, `NUM` pode ser feita figurar no lugar de `256`, usando-se um enunciado *define:*

```
#define NUM 256
```

A função `lexan` retorna `NUM` quando a seqüência de dígitos é examinada à entrada. A variável global `tokenval` é estabelecida com o valor da seqüência de dígitos. Dessa forma, se um 7 for seguido imediatamente por um 6 à entrada, é atribuído a `tokenval` o valor inteiro 76.

Permitir que números figurem em expressões requer uma mudança na gramática da Fig. 2.19. Substituímos os dígitos individuais pelo não-terminal *fator* e introduzimos as seguintes produções e ações semânticas:

fator → (*expr*)
| **num** { *imprimir* (**num**.*valor*) }

O código C para *fator* na Fig. 2.27 é uma implementação direta das produções acima. Quando `lookahead` é igual a `NUM`, o valor do atributo **num**.*valor* é dado pela variável global `tokenval`. A ação de imprimir esse valor é feita pela função da biblioteca padrão `printf`. O primeiro argumento de `printf` é uma cadeia entre aspas especificando o formato a ser usado para a impressão dos argumentos remanescentes. Quando `%d` aparece na cadeia, a representação decimal do próximo argumento é impressa. Dessa forma, o enunciado `printf` na Fig. 2.27 imprime um espaço seguido pela representação decimal de `tokenval` seguida por outro espaço.

A implementação da função `lexan` é mostrada na Fig. 2.28. A cada vez que o corpo do enunciado *while* nas linhas 8-28 é executado, um caractere é lido em t na linha 9. Se o caractere for um espaço ou uma tabulação (escrita '\t'), nenhum *token* é retornado ao *parser;* iremos meramente através do laço *while* mais uma vez. Se o caractere for o de avanço de linha (escrito '\n'), a variável global `clinha` é incrementada, mantendo dessa forma o controle dos números de linha na entrada, mas, de novo, nenhum *token* é retornado. O fornecimento de uma mensagem de erro com o número da linha auxilia na localização dos erros.

O código para a leitura da seqüência de dígitos está nas linhas 14-23. O predicado `isdigit(t)` proveniente do arquivo de inclusão `<ctype.h>` é usado nas linhas 14 e 17 a fim de determinar se um caractere `t` recém-chegado, é um dígito. Se for, seu valor inteiro é dado pela expressão t-'0', tanto em ASCII quanto em EBCDIC. Com outros conjuntos de caracteres, pode ser necessário que a conversão seja feita diferentemente. Na Seção 2.9 incorporamos esse analisador léxico ao nosso tradutor de expressões.

```
fator( )

{
    if (lookahead == '(')  {
        reconhecer (' ('); expr( );
        reconhecer(') ');
    }
    else if (lookahead == NUM)  {
        printf(" %d ", tokenval); reconhecer
        (NUM);
    }
    else erro( );
}
```

Fig. 2.27. Código em C para *fator* quando os operandos podem ser números.

```
(1) #include <stdio.h>
(2) #include <ctype.h>
(3) int clinha = 1;
(4) int tokenval = NONE;
(5) int lexan( )
(6) {
(7)       int t;
(8)       while(1)  {
(9)          t = getchar( );
(10)         if (t == ' ' || t == '\t')
(11)              ; /* retirar fora espaços e*/
                  /* tabulações    */
(12)         else if (t == '\n')
(13)              clinha = clinha + 1;
(14)         else if (isdigit(t))  {
(15)              tokenval = t - '0'
(16)              t = getchar ( );
(17)              while (isdigit(t))  {
(18)                   tokenval = tokenval*10 +
                       t - '0';
(19)                   t = getchar( );
(20)              }
(21)              ungetc (t, stdin);
(22)              return NUM;
(23)         }
(24)         else {
(25)              tokenval = NONE;
(26)              return t;
(27)         }
(28)      }
(29) }
```

Fig. 2.28. Código C para um analisador léxico que elimina os espaços em branco e coleta números.

2.7 INCORPORANDO UMA TABELA DE SÍMBOLOS

Uma estrutura de dados, chamada de tabela de símbolos, é geralmente usada para armazenar informações sobre as várias construções da linguagem-fonte. A informação é coletada pelas fases de análise de um compilador e usada pelas fases de síntese de forma a gerar o código-alvo. Por exemplo, durante a análise léxica, a cadeia de caracteres, ou lexema, que forma um identificador é guardada numa entrada da tabela de símbolos. As fases posteriores do compilador poderiam adicionar informações a essa entrada, tais como o tipo do identificador, seu uso (isto é, procedimento, variável ou rótulo) e sua posição de armazenamento. A fase de geração de código usaria as informações para gerar o código apropriado para armazenar dados e dar acesso à mesma. Na Seção 7.6 discutimos a implementação e uso das tabelas de símbolos em detalhes. Nesta seção, ilustramos como o analisador léxico da seção anterior poderia interagir com a tabela de símbolos.

A Interface com a Tabela de Símbolos

As rotinas da tabela de símbolos estão relacionadas primariamente com o armazenamento e recuperação dos lexemas. Quando um lexema é armazenado, também o é o *token* associado ao mesmo. As seguintes operações serão realizadas na tabela de símbolos.

`inserir(s, t)`: Retorna o índice da nova entrada para a cadeia `s`, *token* `t`.

`buscar(s)`: Retorna o índice de uma entrada para a cadeia `s` ou `0` se `s` não for encontrado.

O analisador léxico usa a operação de busca a fim de determinar se existe uma entrada para um lexema na tabela de símbolos. Se nenhuma entrada existir, a mesma usa a operação inserir para criar uma. Discutiremos uma implementação na qual o analisador léxico e o *parser* sabem, ambos, o formato das entradas da tabela de símbolos.

O Tratamento das Palavras-Chave Reservadas

As rotinas da tabela de símbolos acima podem tratar qualquer coleção de palavras-chave reservadas. Como exemplo, consideremos os *tokens* **div** e **mod** com os lexemas `div` e `mod`, respectivamente. Podemos iniciar a tabela de símbolos usando as chamadas

```
inserir ("div", div) ;
inserir ("mod", mod) ;
```

Qualquer chamada subseqüente `buscar ("div")` retorna o *token* **div**, de tal forma que `div` não pode ser mais usado como um identificador.

Qualquer coleção de palavras-chave reservadas pode ser manipulada dessa forma, através da iniciação apropriada da tabela de símbolos.

Uma Implementação da Tabela de Símbolos

A estrutura de dados para uma implementação particular da tabela de símbolos é delineada na Fig. 2.29. Não desejamos reservar uma quantidade fixa de espaço para guardar os lexemas que formam os identificadores; uma quantidade fixa de espaço pode não ser ampla o suficiente para abrigar um identificador muito longo e pode ser grande demais para um identificador curto, tal como `i`. Na Fig. 2.29, o *array* isolado `lexemas` guarda as cadeias de caracteres que formam os identificadores. Cada cadeia é terminada por um caractere de fim de cadeia, denotado por EOS *(end of string)*, o qual não pode figurar nos identificadores. Cada entrada no *array* da tabela de símbolos `tab_simb` é

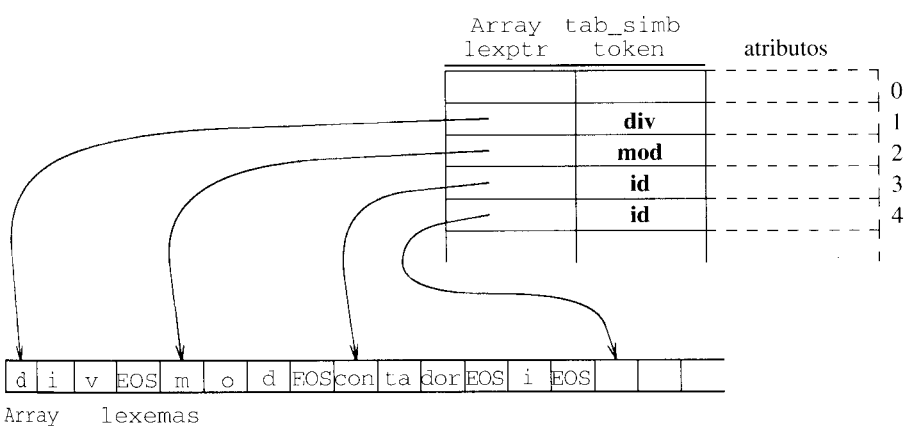


Fig. 2.29. A tabela de símbolos e o *array* para armazenar cadeias.

um registro consistindo em dois campos, `lexptr`, que aponta para o início de um lexema, e *token*. Campos adicionais guardam os valores dos atributos, apesar de não realizarmos tal coisa aqui.

Na Fig. 2.29, a zeroésima entrada é deixada vazia porque `buscar` retornar 0 a fim de indicar que não existe entrada para uma dada cadeia. A primeira e segunda entradas são para as palavras-chave `div` e `mod`. A terceira e quarta para os identificadores `contador` e `i`.

O pseudocódigo para um analisador léxico que manipula identificadores é mostrado na Fig. 2.30; uma implementação em C aparece na Seção 2.9. Os espaços em branco e a manipulação dos inteiros são tratados pelo analisador léxico da mesma forma que na Fig. 2.28 na última seção.

Quando nosso analisador léxico presente lê uma letra, começa por guardar as letras e dígitos subseqüentes no *buffer* `lexbuf`. A cadeia coletada em `lexbuf` é então procurada na tabela de símbolos, usando-se a operação `buscar`. Como a tabela de símbolos é iniciada com entradas para as palavras-chave `div` e `mod`, como mostrado na Fig. 2.29, a operação buscar irá encontrá-las se `lexbuf` contiver `div` ou `mod`. Se não existir entrada para a cadeia em `lexbuf`, ou seja, se buscar retornar 0, então `lexbuf` conterá o lexema de um novo identificador. Uma entrada para o novo identificador é criada usando-se `inserir`. Após a inserção ter sido feita, `p` é o índice da entrada da tabela de símbolos para a cadeia em `lexbuf`. Esse índice é comunicado ao *parser* através da atribuição de `p` a `tokenval`, sendo retornado o *token* do campo `token` da entrada criada para o novo identificador.

A ação *default* é retornar a codificação inteira do caractere como um *token*. Como os *tokens* de um único caractere não possuem atributos, `tokenval` é estabelecido com o valor `NONE`.

```
função lexan: inteiro;
var lexbuf; array [0..100] de caracteres;
    c:     caractere;
início
   repetir início
         ler um caractere em c;
         se c for um espaço ou caractere de tabulação então
            não fazer nada
         senão se c for um caractere de avanço de linha então
            clinha : = clinha + 1
         senão se c for um dígito então início
            fazer tokenval igual ao valor deste e dos dígitos seguintes;
            retornar NUM
         fim
         senão se c for uma letra então início
            colocar c mais as letras e dígitos sucessivos em lexbuf;
            p : = buscar (lexbuf);
            se p = 0 então
                  p : = inserir (lexbuf, ID);
            tokenval : = p;
            retornar o campo token da entrada p da tabela
         fim
         senão início   /* o token é um único caractere */
            fazer tokenval igual a NONE;   /* não existe atributo */
            retornar a codificação inteira do caractere c
         fim
   fim
fim
```

Fig. 2.30. Pseudocódigo para um analisador léxico.

2.8 MÁQUINAS DE PILHA ABSTRATAS

A interface de vanguarda do compilador constrói uma representação intermediária do programa-fonte a partir da qual a interface de retaguarda gera o programa-alvo. Uma forma popular de representação intermediária é o código para uma *máquina de pilha abstrata*. Como mencionado no Capítulo 1, subdividir um compilador numa interface de vanguarda e numa de retaguarda torna-o mais fácil de modificar para que rode numa nova máquina.

Nesta seção, apresentamos uma *máquina de pilha abstrata* e mostramos como o código para a mesma pode ser gerado. A máquina possui memórias de dados e de instruções separadas e todas as operações aritméticas são realizadas sobre valores numa pilha. As instruções são limitadas e caem em três classes: aritmética inteira, manipulação de pilha e fluxo de controle. A Fig. 2.31 ilustra a máquina. O apontador *pc* indica a instrução que estamos prestes e executar. Os significados das instruções mostradas serão discutidos rapidamente.

Instruções Aritméticas

A *máquina de pilha abstrata* precisa implementar cada operador da linguagem intermediária. Uma operação básica, tal como adição ou subtração, é suportada diretamente pela máquina abstrata. Uma operação mais complexa, entretanto, pode necessitar ser implementada como uma seqüência de instruções da máquina abstrada. Simplificaremos a descrição da máquina assumindo que exista uma instrução para cada operador aritmético.

O código da máquina abstrata para uma expressão aritmética simula a avaliação de uma representação posfixa para a expressão, usando uma pilha. A avaliação se dá através do processamento da representação posfixa da esquerda para a direita, empilhando cada operando, na medida que seja encontrado. Quando um operador *k*-ário (com k argumentos) é encontrado, seu argumento mais à esquerda está *k*-1 posições abaixo do topo da pilha e seu argumento mais à direita ao topo. A avaliação aplica o operador aos *k* valores superiores da pilha, desempilha os operandos e empilha o resultado. Por exemplo, na avaliação da expressão posfixa `13 + 5 *`, as seguintes ações são realizadas:

1. Empilhar 1.
2. Empilhar 3.
3. Adicionar os dois elementos superiores da pilha, desempilhá-los e empilhar o resultado 4.
4. Empilhar 5.
5. Multiplicar os dois elementos superiores da pilha, desempilhá-los e empilhar o resultado 20.

O valor ao topo da pilha (neste exemplo, 20) é o valor de toda a expressão.

Na linguagem intermediária, todos os valores serão inteiros, com 0 correspondendo a `falso` e inteiros diferentes de zero, a `verda-`

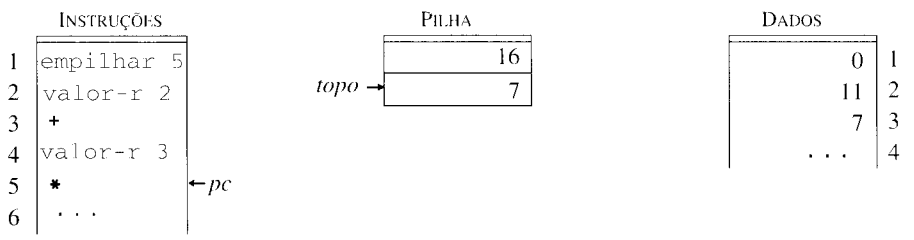


Fig. 2.31. Fotografia da máquina de pilha após a execução das primeiras quatro instruções.

deiro. Os operadores booleanos e e ou requerem, ambos, que seus argumentos sejam avaliados.[1]

Valores-L e Valores-R

Existe uma distinção entre o significado dos identificadores à esquerda e à direita de uma atribuição. Em cada uma das atribuições

```
i := 5 ;
i := i + 1 ;
```

o lado direito especifica um valor inteiro, enquanto que o lado esquerdo especifica onde o valor deve ser armazenado. Similarmente, se p e q são apontadores de caracteres e

```
p↑ := q↑;
```

o lado direito q↑ especifica um caractere, enquanto que p↑ especifica aonde o caractere deve ser armazenado. Os termos *valor-l* e *valor-r* se referem a valores que são apropriados nos lados esquerdo e direito de uma atribuição, respectivamente. Ou seja, os valores-*r* são o que usualmente pensamos a respeito de um "valor", enquanto que os valores-*l* são as suas localizações.

Manipulação da Pilha

Além das instruções óbvias para empilhar uma constante inteira na pilha e desempilhar um valor do topo, existem instruções para dar acesso a dados na memória:

push *v*	empilhar *v*
valor-r *l*	empilhar conteúdo da localização de dados *l*
valor-l *l*	empilhar endereço da localização de dados *l*
pop	desempilhar o valor ao topo da pilha
:=	o valor-*r* ao topo da pilha é colocado no valor-*l* indicado pelo subtopo da
copiar	empilhar uma cópia do valor ao topo da pilha

Tradução de Expressões

O código para avaliar uma expressão na máquina de pilha está estreitamente relacionado à notação posfixa da expressão. Por definição, a forma posfixa da expressão $E + F$ é a concatenação da forma posfixa de E, com a forma posfixa de F e de +. Similarmente, o código da máquina de pilha para avaliar $E + F$ é a concatenação do código para avaliar E, do código para avaliar F e da instrução para adicionar seus valores. A tradução de expressões no código de máquina de pilha pode, dessa forma, ser feita adaptando-se os tradutores das Seções 2.6 e 2.7.

Aqui geramos código de pilha para expressões nas quais as localizações dos dados são endereçadas simbolicamente. (A reserva de localizações de dados para os identificadores é discutida no Capítulo 7.) A expressão a+b é traduzida em

```
valor-r    a
valor-r    b
+
```

Em palavras: empilhar o conteúdo das localizações de dados para a e b; em seguida, remover os dois valores superiores da pilha (os de a e b recém-empilhados), adicioná-los e empilhar o resultado.

A tradução das atribuições em código de máquina de pilha é feita como segue: o valor-*l* do identificador atribuído é empilhado, a expressão é avaliada e seu valor-*r* é atribuído ao identificador. Por exemplo, a atribuição

```
dia := (1461*y) div 4 + (153*m + 2) div 5 + d
```
(2.17)

é traduzida no código da Fig. 2.32.

Esses comentários podem ser expressos formalmente como se segue. Cada não-terminal possui um atributo *t* que fornece a sua tradução. O atributo *lexema* de **id** fornece a representação do identificador sob a forma de cadeia de caracteres.

cmd → **id** := *expr*
{ *cmd.t* := 'valor-l' ‖ **id.** *lexema* ‖ *expr.t* ‖ ':=' }

Fluxo de Controle

A máquina de pilha executa instruções em seqüência numérica, a menos que seja ordenada a agir de outra maneira através de um enunciado de desvio condicional ou incondicional. Várias opções existem para especificar o destino dos desvios:

1. O operando da instrução fornece a localização-alvo.
2. O operando da instrução fornece a distância relativa, positiva ou negativa a ser saltada.
3. O alvo é especificado simbolicamente; isto é, a máquina suporta rótulos.

Com as duas primeiras opções existe a possibilidade adicional de se tomar o operando a partir do topo da pilha.

Escolhemos a terceira opção para a máquina abstrata porque é mais fácil gerar tais desvios. Sobretudo, os endereços simbólicos não precisam ser mudados se, após gerarmos o código para a máquina abstrata, fizermos certos melhoramentos no código que resultem na inserção ou remoção de instruções.

As instruções para o fluxo de controle da máquina de pilha são:

rótulo *l*	designa o alvo dos desvios para o rótulo *l*; não possui outro efeito
goto *l*	a próxima instrução é tomada a partir do enunciado com rótulo *l*
gofalse *l*	desempilhar o valor ao topo da pilha; desviar se for zero
gotrue *l*	desempilhar o valor ao topo da pilha; desviar se não for zero
parar	parar a execução

Tradução de Enunciados

O gabarito na Fig. 2.33 delineia o código para a máquina abstrata para os enunciados condicionais e as repetições. A seguinte discussão se concentra na criação de rótulos.

Considere o gabarito do código para o enunciado *if* na Fig. 2.33. Só pode haver uma única instrução rótulo saída em todo o programa-fonte; de outra forma haverá confusão sobre para onde o con-

```
valor-l dia          push 2
push 1461            +
valor-r y            push 5
*                    div
push 4               +
div                  valor-r d
push 153             +
valor-r m            :=
*
```

Fig. 2.32. Tradução de dia:= (1461*y) div 4 + (153*m+2) div 5 + d.

[1]Ou seja, não são operadores booleanos em curto-circuito. Para uma exposição detalhada sobre o tema ver Pratt [1984] na bibliografia. (N. do T.)

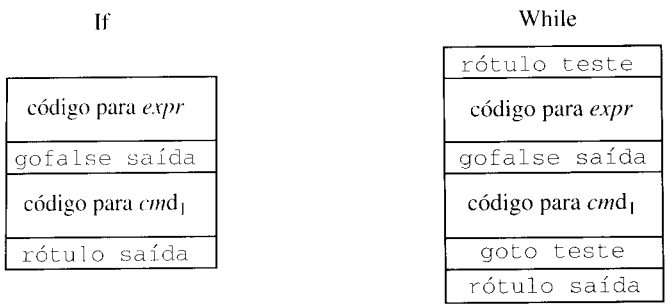


Fig. 2.33. Gabarito do código para comandos condicionais e repetições.

trole deverá fluir a partir de um enunciado `goto saída`. Precisamos, dessa forma, de algum mecanismo para consistentemente substituir `saída` no gabarito do código por um único rótulo cada vez que um enunciado *if* for traduzido.[2]

Suponhamos que *novo_rótulo* seja um procedimento que retorne um rótulo novo a cada vez que seja chamado. Na seguinte ação semântica, o rótulo retornado por uma chamada de *novo_rótulo* é registrado usando-se uma variável local *saída:*

$$
\begin{aligned}
cmd \rightarrow \textbf{if } expr \textbf{ then } cmd_1 \ \{ & saída := novo_rótulo; \\
& cmd.t := expr.t \,\|\, \\
& \quad \texttt{'gofalse'}\ saída \,\|\, \\
& \quad cmd_1.t \,\|\, \\
& \quad \texttt{'rótulo'}\ saída \ \}
\end{aligned}
\qquad (2.18)
$$

Emitindo uma Tradução

Os tradutores de expressões da Secção 2.5 usavam enunciados impressos a fim de gerar a tradução de uma expressão. Enunciados de impressão similares podem ser usados para emitir a tradução de comandos. Em lugar de imprimir os enunciados, usamos o procedimento *emitir* a fim de ocultar os detalhes de impressão. Por exemplo, *emitir* poderia se preocupar com a necessidade de cada instrução da máquina abstrata precisar estar numa linha separada. Usando o procedimento *emitir*, podemos escrever o seguinte em lugar de (2.18):

$cmd \rightarrow$ **if**
  $expr$ { *saída* := *novo-rótulo; emitir* (`'gofalse'`, *saída*);}
  then
  cmd_1 { *emitir* (`'rótulo'`, *saída*); }

Quando as ações semânticas aparecem dentro de uma produção, consideramos os elementos no lado direito da mesma numa ordem da esquerda para a direita. Para a produção acima, a ordem das ações é como se segue: as ações durante a análise gramatical de *expr* são feitas, *saída* é estabelecida com o rótulo retornado por *novo_rótulo* e a instrução `gofalse` é emitida, as ações durante a análise gramatical de cmd_1 são tomadas e, finalmente, a instrução `rótulo` é emitida. Assumindo que as ações durante a análise gramatical de *expr* e de cmd_1 emitam o código para esses não-terminais, as produções acima implementam o gabarito de código da Fig. 2.33.

O pseudocódigo para a tradução de enunciados condicionais e de atribuição é mostrado na Fig. 2.34. Como a variável *saída* é local ao procedimento *cmd*, seu valor não é afetado pelas chamadas aos procedimentos *expr* e *cmd*. A geração de rótulos requer algum exame. Suponhamos que os rótulos na tradução são da forma `L1`, `L2`,... . O pseudocódigo manipula tais rótulos usando o inteiro que se segue ao `L`. Dessa forma, *saída* é declarada como sendo uma variável inteira, *novo_rótulo*

```
procedimento cmd;
var teste, saída: inteiro;  /* para os rótulos */
início
   se lookahead = id então início
         emitir ('valor-e', tokenval); reconhecer (id);
         reconhecer (':='); expr
   fim
   senão se lookahead = 'if' then início
         reconhecer ('if');
         expr;
         saída := novo rótulo;
         emitir ('gofalse', saída);
         reconhecer ('then');
         cmd;
         emitir ('rótulo', saída)
   fim
         /* o código para os enunciados remanescentes entra aqui */
   senão erro;
fim
```

Fig. 2.34. Pseudocódigo para tradução de enunciados.

retorna um inteiro que vem a ser o valor de *saída* e *emitir* precisa ser escrita de forma a imprimir um rótulo dado um inteiro.

O gabarito de código para os enunciados repetitivos (*while*) na Fig. 2.33 pode ser convertido em código de uma maneira similar. A tradução de uma seqüência de enunciados é simplesmente a concatenação dos enunciados em seqüência e é deixada para o leitor.

A tradução da maioria das construções de uma entrada e uma saída é similar àquela dos enunciados repetitivos *while*. Ilustramos isso considerando o fluxo de controle nas expressões.

Exemplo 2.10. O analisador léxico da Secção 2.7 contém uma condicional da forma:

if t = **espaço or** t = **tabulação then** · · ·

Se t for um espaço, então claramente não é necessário testar se t é um caractere de tabulação, porque a primeira igualdade implica que a condição seja verdadeira. A expressão

$expr_1$ **or** $expr_2$

pode ser implementada como

if $expr_1$ **then true else** $expr_2$...

O leitor pode verificar que o código seguinte implementa o operador **or:**

```
código para expr1        /* copiar valor de expr1 */
copiar
gotrue saída
pop                      /* remover valor de expr1 */
código para expr2
rótulo saída
```

Relembremos que as instruções `gotrue` e `gofalse` removem o topo da pilha a fim de simplificar a geração de código dos enunciados condicionais e de repetição. Pela cópia do valor de $expr_1$, asseguramos que o valor no topo da pilha seja verdadeiro se a instrução `gotrue` levar a um salto. □

2.9 JUNTANDO AS TÉCNICAS

Neste capítulo, apresentamos um número de técnicas dirigidas pela sintaxe para construir a vanguarda de um compilador. Para sumarizar essas técnicas, colocamos, nesta seção, um programa C que funciona

[2] Isso significa dizer que rótulo *saída*, para um determinado valor do parâmetro *saída* só poderá ser gerado uma única vez no programa, isto é, não poderão haver duas localizações distintas com a instrução rótulo *saída*, tendo *saída* recebido o mesmo valor de substituição. (N. do T.)

como um tradutor da forma infixa para a posfixa de uma linguagem consistindo em expressões terminadas por ponto e vírgula. As expressões são constituídas de números, identificadores, e os operadores +, −, *, /, `div` e `mod`. A saída do tradutor é uma representação posfixa para cada expressão. O tradutor é uma extensão dos programas desenvolvidos nas Seções 2.5-2.7. Uma listagem completa do programa C é dada ao fim desta seção.

Descrição do Tradutor

O tradutor é projetado usando-se o esquema de tradução dirigida pela sintaxe da Fig. 2.35. O *token* **id** representa uma seqüência não vazia de letras e dígitos, começando por uma letra, **num**, uma seqüência de dígitos e **eof,** um caractere de fim de arquivo. Os *tokens* são separados por seqüências de espaços, tabulações e avanços de linha ("espaços em branco"). O atributo *lexema* do *token* **id** fornece a cadeia de caracteres que forma o *token;* o atributo *valor* do *token* **num** fornece o inteiro representado por **num.**

O código do tradutor está arranjado em sete módulos, cada um armazenado num arquivo separado. A execução começa no módulo `main.c`, o qual consiste em uma chamada para `init()` para a iniciação seguida por uma chamada para `parse()` para a tradução. Os seis módulos remanescentes são mostrados na Fig. 2.36. Existe também um arquivo de cabeçalho global `global.h` que contém definições comuns a mais de um módulo; o primeiro enunciado em cada módulo

```
#include "global.h"
```

leva à inclusão do arquivo de cabeçalho ser como parte do módulo. Antes de mostrar o código para o tradutor, descreveremos brevemente cada módulo e como cada um foi construído.

O Módulo de Análise Léxica `lexer.c`

O analisador léxico é uma rotina chamada `lexan()` a qual é chamada pelo *parser* a fim de encontrar *tokens.* Implementada a partir do pseudocódigo da Fig. 2.30, a rotina lê a entrada um caractere de cada vez e retorna ao *parser* o *token* que encontrar. O valor do atributo associado ao *token* é atribuído à variável global `tokenval`.

Os seguintes *tokens* são esperados pelo *parser:*

```
+ - * / DIV MOD ( ) ID NUM DONE
```

Aqui `ID` representa um identificador, `NUM` um número e `DONE` o caractere de fim de arquivo. O espaço em branco é silenciosamente removido pelo analisador léxico. A tabela na Fig. 2.37 mostra o *token* e o valor de atributo produzido pelo analisador léxico para cada lexema da linguagem_fonte.

início	→	*lista* **eof**	
lista	→	*expr* ; *lista*	
	|	ϵ	
expr	→	*expr* + *termo*	{ *imprimir*('+') }
	|	*expr* − *termo*	{ *imprimir*('−') }
	|	*termo*	
termo	→	*termo* * *fator*	{ *imprimir*('*') }
	|	*termo* / *fator*	{ *imprimir*('/') }
	|	*termo* **div** *fator*	{ *imprimir* ('DIV') }
	|	*termo* **mod** *fator*	{ *imprimir*('MOD') }
	|	*fator*	
fator	→	(*expr*)	
	|	**id**	{ *imprimir*(**id.** *lexema*) }
	|	**num**	{ *imprimir*(**id.** *valor*) }

Fig. 2.35 Especificação para um tradutor infixa-posfixa.

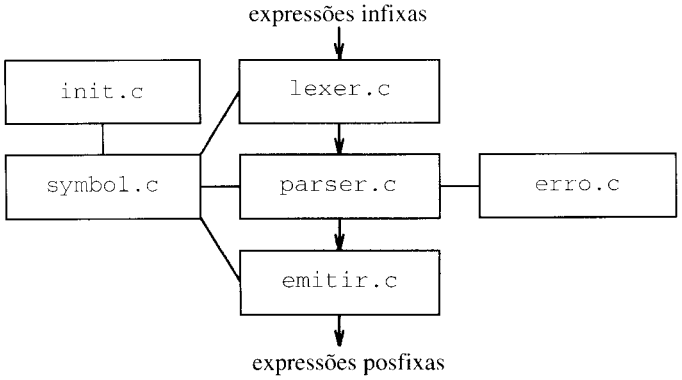


Fig. 2.36. Módulos para um tradutor infixa-posfixa.

O analisador léxico usa a rotina da tabela de símbolo `buscar` a fim de determinar se o lexema de um identificador já foi previamente examinado e a rotina `inserir` para armazenar um novo lexema na tabela. Igualmente, incrementa uma variável global `clinha` a cada vez que examina um caractere de avanço de linha.

O Módulo Parser parser.c

O *parser* é construído usando-se as técnicas da Seção 2.5. Primeiro eliminamos a recursão à esquerda do esquema de tradução da Fig. 2.35 de tal forma que a gramática subjacente possa ser analisada por um *parser* descendente recursivo. O esquema transformado é mostrado à Fig. 2.38.

Construímos, então, funções para os não-terminais *expr, termo* e *fator*, como fizemos na Fig. 2.24. A função *parse* () implementa o símbolo de partida da gramática; chama `lexan` sempre que necessitar de um novo *token.* O *parser* utiliza a função `emitir` para gerar a saída e a função `erro` para relatar um erro de sintaxe.

O Módulo Emissor `emitir.c`

O módulo emissor consiste em uma única função `emitir(t, tval)` que gera a saída para o *token* t com o valor de atributo `tval`.

Módulos da Tabela de Símbolos `symbol.c` e `init.c`

O módulo da tabela de símbolos `symbol.c` implementa a estrutura de dados mostrada na Fig. 2.29 da Seção 2.7. As entradas no *array* `tab_simb` são pares consistindo em um apontador para o *array* de lexemas e um inteiro que denota o *token* lá armazenado. A operação

LEXEMA	TOKEN	VALOR DE ATRIBUTO
espaço em branco		
seqüência de dígitos	NUM	valor numérico da seqüência
div	DIV	
mod	MOD	
outras seqüências de uma única letra seguidas por letras e dígitos	ID	índice para tab_sim.
caractere de fim de arquivo	DONE	
qualquer outro caractere	o próprio caractere	NONE

Fig. 2.37. Descrição dos *tokens.*

$$
\begin{array}{rcl}
\textit{início} & \rightarrow & \textit{lista}\ \textbf{eof} \\
\textit{lista} & \rightarrow & \textit{expr}\ ;\ \textit{lista} \\
 & | & \epsilon \\
\textit{expr} & \rightarrow & \textit{termo maistermos} \\
\textit{maistermos} & \rightarrow & +\ \textit{termo}\ \{\ \textit{imprimir}('+')\ \}\ \textit{maistermos} \\
 & | & -\ \textit{termo}\ \{\ \textit{imprimir}('-')\ \}\ \textit{maistermos} \\
 & | & \epsilon \\
\textit{termo} & \rightarrow & \textit{fator maisfatores} \\
\textit{maisfatores} & \rightarrow & *\ \textit{fator}\ \{\ \textit{imprimir}('*')\ \}\ \textit{maisfatores} \\
 & | & /\ \textit{fator}\ \{\ \textit{imprimir}('/')\ \}\ \textit{maisfatores} \\
 & | & \textbf{div}\ \textit{fator}\ \{\ \textit{imprimir}(\texttt{'DIV'})\ \}\ \textit{maisfatores} \\
 & | & \textbf{mod}\ \textit{fator}\ \{\ \textit{imprimir}(\texttt{'MOD'})\ \}\ \textit{maisfatores} \\
 & | & \epsilon \\
\textit{fator} & \rightarrow & (\ \textit{expr}\ ) \\
 & | & \textbf{id}\ \{\ \textit{imprimir}(\textbf{id.}\ \textit{lexema})\ \} \\
 & | & \textbf{num}\ \{\ \textit{imprimir}(\textbf{num.}\ \textit{valor})\ \}
\end{array}
$$

Fig. 2.38. Esquema de tradução dirigida pela sintaxe após a eliminação da recursão à esquerda.

`inserir(s,t)` retorna o índice da tabela de símbolos para o lexema `s` que forma o *token* `t`. A função `buscar(s)` retorna o índice da entrada em `tab_simb` para o lexema `s` ou `0` se o mesmo não estiver lá.

O módulo `init.c` é usado para pré-carregar a tabela de símbolos com palavras-chave. O lexema e as representações dos *tokens* para todas as palavras-chave são armazenadas no *array* `palavras-chave`, o qual possui o mesmo tipo que o *array* `tab_simb`. A função `init()` percorre seqüencialmente o *array* `palavra_chave` usando a função `inserir` a fim de colocar as palavras-chave na tabela de símbolos. Esse arranjo nos permite modificar a representação dos *tokens* das palavras-chave numa forma conveniente.

O Módulo de Erros `erro.c`

O módulo de erros gerencia o relatório de erros, o qual é extremamente primitivo. Ao encontrar um erro de sintaxe, o compilador imprime uma mensagem informando que um erro ocorreu à linha corrente de entrada e, então, pára. Uma técnica melhor de recuperação de erros poderia saltar até o próximo ponto e vírgula e continuar a análise gramatical; o leitor é encorajado a realizar essa mudança no tradutor. Técnicas mais sofisticadas de recuperação de erros são apresentadas no Capítulo 4.

Criando o Compilador

O código para os módulos aparece em sete arquivos: `lexer.c`, `parser.c`, `emitter.c`, `symbol.c`, `init.c` `error.c` e `main.c`. O arquivo `main.c` contém o módulo principal do programa C que chama `init()`, a seguir `parse()` e no caso de execução com sucesso, `exit(0)`.

Sob o sistema operacional UNIX, o compilador pode ser criado através da execução do comando

```
cc lexer.c parser.c emitter.c symbol.c init.c
                error.c main.c
```

ou compilando separadamente os arquivos, usando

cc -c *nome do arquivo*.c

e ligando os arquivos *nome do arquivo*. `o` resultantes:

```
cc lexer.o parser.o emitter.o symbol.o init.o
                error.o main.o
```

O comando `cc` cria um arquivo `a.out` que contém o tradutor. O tradutor pode, então, ser exercitado digitando-se `a.out` seguido pelas expressões a serem traduzidas; por exemplo,

```
2+3*5;
12 div 5 mod 2;
```

ou qualquer outra expressão de seu gosto. Tente-o.

A Listagem

Aqui está a listagem do programa C que implementa o tradutor. O arquivo global de cabeçalho `global.h` é mostrado, seguido pelos sete arquivos-fonte. Por questão de clareza, o programa foi escrito num estilo elementar de C.

```
/**** global.h ************************************************************/
#include <stdio.h>  /*carrega rotinas de e/s   */
#include <ctype.h>  /*carrega rotinas de teste de caracteres */

#define BSIZE 128   /* tamanho do buffer  */
#define NONE  -1
#define EOS         '\0'

#define NUM         256
#define DIV         257
#define MOD         258
#define ID          259
#define DONE        260

int tokenval;  /*valor do atributo token */
int clinha;    /* contador de linhas     */

struct entry {      /* formata cada entrada da tabela de símbolos  */
       char   *lexptr;
       int    token;
};
struct entry tab-simb[];   /* tabela de símbolos */

/***** lexer.c ***********************************************************
```

```
#include "global.h"

(char lexbuf[BSIZE];
int clinha = 1;
int tokenval = NONE;
int lexan()   /* analisador léxico */
{

      int t;
      while(1) {
            t = getchar();
            if (t == ' ' :: t == '\t')
                  ; /* remover os espaços em branco*/
            else if (t == '\n')
                  clinha = clinha + 1;
            else if (isdigit (t)) {  /* t é um dígito */
                  ungetc(t, stdin);
                  scanf("%d", &tokenval);
                  return NUN;
            }
            else if (isalpha(t))       {      /* t é uma letra    */
                  int p, b = 0;
                  while (isalnum(t)) {       /* t é alfanumérico, */
                        lexbuf[b] = t;
                        t = getchar();
                        b = b + 1;
                        if ( b >= BSIZE)
                              erro("erro do compilador");
                  }
                  lexbuf[b] = EOS;
                  if (t ! = EOF)
                        ungetc(t, stdin);
                  p = buscar (lexbuf);
                  if (p == 0)
                       p = insert(lexbuf, ID);
                  tokenval = p;
                  return tab-simb [p]. token;
            }
            else if (t == EOF)
                  return DONE;
            else {
                  tokenval = NONE;
                  return t;
            }
      }
}

/****        parser.c      ***************************************************
#include "global.h"
int lookahead;
parse()             /* analisa gramaticalmente e traduz uma lista de     */
                    /* expressões
{
      lookahead = lexan();
      while (lookahead != DONE ) {
            expr(); reconhecer (';');
      }
}
expr()
{
      int t;
      termo();
      while(1)
            switch (lookahead) {
            case '+': case '-':
                  t = lookahead;
```

```
                reconhecer (lookahead); termo(); emitir (t, NONE);
                continue;
            default:
                return;
            }
}
termo()
{
            int t;
            fator();
            while(1)
                switch (lookahead) {
                case '*': case '/': case DIV; case MOD;
                    t = lookahead;
                    reconhecer (lookahead); fator(); emitir(t, NONE);
                    continue;
                default:
                    return;
                }
}
fator()
{
            switch (lookahead) {
                case '(':
                    reconhecer ('('); expr(); reconhecer (')'); break;
                case NUM:
                    emitir (NUM, tokenval); reconhecer (NUM); break;
                case ID:
                    emitir (ID, tokenval); reconhecer(ID); break;
                default:
                    erro ("erro de sintaxe");
            }
}
reconhecer (t)
            int t;
{
            if (lookahead == t)
                lookahead = lexan();
            else erro ("erro de sintaxe");
}

/****  emitir.c  ******************************************************/
#include "global.h"
emitir(t, tval)   /* gera a saída */
            int t, tval;
{
            switch(t)    {
                case '+': case '-': case '*' :case '/':
                    printf("%c\n", t); break;
                case DIV:
                    printf("DIV\n"); break;
                case MOD:
                    printf ("MOD\n"); break;
                case NUM:
                    printf("%d\n", tval); break;
                case ID:
                    (printf("%s\n", tab_simb[tval].lexptr); break;
                default:
                    (printf("token %d, tokenval %d\n", t, tval);
            }
}

/****        symbol.c      ***************************************************/

#include      "global.h"
#define       STRMAX 999    /*tamanho do array de lexemas     */
#define       SYMMAX 100    /*tamanho da tabela de símbolos   */
```

```
char          lexemas[STRMAX];
int lastchar = -1:  /* última posição usada em lexemas      */
struct entry tab_simb[SYMMAX];
int           lastentry = 0;       /* última posição usada em tab_simb     */
int           buscar(s)     /* retorna a posição da entrada para s*/
              char s[];
{
              int p;
              for (p = lastentry; p > 0; p = p - 1)
                     if (strcmp(tab-simb[p].lexptr, s) == 0)
                     return p;
              return 0;
}
int insert(s, tok)  /*retorna a posição da entrada para s*/
              char s[];
              int tok;
{
              int len;
              len = strlen(s);     /*strlen computa o comprimento de s */
              if (lastentry + 1 >= SYMMAX)
                     erro("tabela de símbolos está cheia");
              if (lastchar + len + 1 >= STRMAX)
                     erro("array de lexemas está dcheio"):
              lastentry = lastentry + 1;
              tab-simb[lastentry].token = tok;
              tab-simb[lastentry].lexptr = &lexemas [lastchar + 1];
              strcpy(tab-simb[lastentry].lexptr, s);
              return lastentry;
}

/****  init.c  ***************************************************/
#include "global.h"
struct entry palavras-chave[] = {
              "div, DIV,
              "mod" MOD,
              0,     0
};
init()                /* carrega palavras-chave em tab-simb   */
{
              struct entry *p;
              for (p = palavras-chave; p -> token; p++)
                  insert(p -> lexptr token);
}

/****  erro.c  ***************************************************/
#include "global.h"
erro (m)      /* gera todas as mensagens de erro  */
     char *m;
{
              fprintf(stderr, "linha %d: %s\n", clinha,m);
              exit(1);      /* término do programa sem sucesso  */
}
/****          main.c ****************************************************/
#include "global.h"
main()
{
              init();
              parse();
              exit(0);      /* término com sucesso  */
}

/*****************************************************************************/
```

N. do T. Recomendamos que a listagem dos módulos acima seja toda incorporada num único programa, retirando-se os enunciados "# include global.h" de cada módulo particular e compilando o programa resultante através dos diversos compiladores existentes no mercado.

EXERCÍCIOS

2.1 Considere a gramática livre de contexto

$$S \rightarrow SS + \mid SS * \mid a$$

a) Mostre que a cadeia `aa+a*` pode ser gerada por esta gramática.
b) Construa a árvore gramatical para essa cadeia.
c) Qual é a linguagem gerada por essa gramática? Justifique a sua resposta.

2.2 Quais são as linguagens geradas pelas seguintes gramáticas? Em cada caso, justifique a sua resposta.
a) $S \rightarrow 0\ S\ 1 \mid 0\ 1$
b) $S \rightarrow +\ S\ S \mid -\ S\ S \mid a$
c) $S \rightarrow S\ (\ S\ )\ S \mid \epsilon$
d) $S \rightarrow a\ S\ b\ S \mid b\ S\ a\ S \mid \epsilon$
e) $S \rightarrow a \mid S + S \mid S\ S \mid S * \mid (\ S\ )$

2.3 Quais das gramáticas no Exercício 2.2 são ambíguas?

2.4 Construa gramáticas livres de contexto não ambíguas para cada uma das seguintes linguagens. Em cada caso mostre que sua gramática está correta.
a) Expressões aritméticas na notação posfixa.
b) Listas associativas à esquerda de identificadores separados por vírgulas.
c) Listas associativas à direita de identificadores separados por vírgulas.
d) Expressões aritméticas de inteiros e identificadores com os quatro operadores binários +, −, *, /.
e) Inclua o mais e o menos unários aos operadores aritméticos de (d).

***2.5** a) Mostre que todas as cadeias binárias geradas pela seguinte gramática possuem valores divisíveis por 3. *Dica:* Use a indução no número de nós da árvore gramatical.

$$num \mid 11 \mid 1001 \mid num\ 0 \mid num\ num$$

b) A gramática gera de todas as cadeias binárias com valores divisíveis por 3?

2.6 Construa uma gramática livre de contexto para os números romanos.

2.7 Construa um esquema da tradução dirigida pela sintaxe que traduza expressões aritméticas da notação infixa para a prefixa, na qual um operador figura antes de seus operandos; por exemplo, $-xy$ é a notação prefixa para $x - y$. Forneça as árvores gramaticais anotadas para as entradas `9-5+2` e `9-5*2`.

2.8 Construa um esquema de tradução dirigida pela sintaxe que traduza expressões aritméticas da notação posfixa para a infixa. Forneça as árvores gramaticais anotadas para as entradas `95-2*` e `952*-`.

2.9 Construa um esquema de tradução dirigida pela sintaxe que traduza inteiros em números romanos.

2.10 Construa um esquema de tradução dirigida pela sintaxe que traduza números romanos em inteiros.

2.11 Construa *parsers* descendentes recursivos para as gramáticas no Exercício 2.2 (a), (b) e (c).

2.12 Construa um tradutor dirigido pela sintaxe que verifique se os parênteses numa cadeia de entrada estão balanceados apropriadamente.

2.13 As seguintes regras definem a tradução de uma palavra inglesa para o *latim estilizado:*
a) Se a palavra começa por uma cadeia não vazia de consoantes, mova a cadeia inicial de consoantes para o final da palavra e adicione o sufixo `AY`; por exemplo, `DAY` se torna `AYDAY`.
b) Se a palavra começa por uma vogal, adicione o sufixo `YAY`; por exemplo, `owl` se torna `owlyay`.
c) `U` seguindo um `Q` é uma consoante.
d) `Y` ao início de uma palavra é uma vogal, se não for seguido por uma vogal.
e) Palavras com uma única letra não são alteradas.

Construa um esquema de tradução dirigida pela sintaxe para o *latim estilizado*.

2.14 Na linguagem de programação C, o enunciado *for* possui a forma:

$$\texttt{for}\ (\ expr_1\ ;\ expr_2\ ;\ expr_3\ )\ cmd$$

A primeira expressão é executada antes do laço; é tipicamente usada para inicializar o índice do laço. A segunda expressão é um teste feito antes de cada iteração do laço; o laço é abandonado se a expressão se torna 0. O laço em si consiste do enunciado { *cmd* $expr_3$;}. A terceira expressão é executada ao fim de cada iteração; é tipicamente usada para incrementar o índice do laço. O significado do enunciado **for** é similar a

$$expr_1\ ;\ \texttt{while}\ (\ expr_2\ )\ \{\ cmd\ expr_3\ \}$$

Construa um esquema de tradução dirigida pela sintaxe para traduzir enunciados **for** em C em código de máquina de pilha.

***2.15** Considere o seguinte enunciado *for:*

for $i := 1$ **step** $10 - j$ **until** $10 * j$ **do** $j := j + 1$

Três definições semânticas podem ser fornecidas para esse enunciado. Um possível significado é que o limite $10 * j$ e o incremento $10 - j$ devam ser avaliados uma vez antes do laço, como em PL/I. por exemplo, se $j = 5$ antes do laço, deveríamos rodar dez vezes através do mesmo e sair. Um segundo significado, completamente diferente, seria obtido se fôssemos requeridos avaliar o limite e o incremento a cada vez durante o laço. Um terceiro significado é dado por linguagens tais como Algol. Quando o incremento é negativo, o teste feito para a terminação do laço é $i < 10 * j$, ao invés de $i > 10 * j$. Para cada uma dessas três definições semânticas, construa um esquema de tradução dirigida pela sintaxe a fim de traduzir esses laços, *for* em código de máquina de pilha.

2.16 Considere o seguinte fragmento de gramática para enunciados *if-then* e *if-then-else:*

$$\begin{aligned} cmd \rightarrow\ & \textbf{if}\ expr\ \textbf{then}\ cmd \\ \mid\ & \textbf{if}\ expr\ \textbf{then}\ cmd\ \textbf{else}\ cmd \\ \mid\ & \textbf{other} \end{aligned}$$

onde **other** está no lugar de outros enunciados da linguagem.
a) Mostre que essa gramática é ambígua.
b) Construa uma gramática equivalente não ambígua que associe cada **else** com o **then** mais próximo não associado.
c) Construa um esquema de tradução baseado nesta gramática que traduza os enunciados condicionais em código de máquina de pilha.

***2.17** Construa um esquema de tradução dirigida pela sintaxe que traduza expressões aritméticas na notação infixa em expressões aritméticas na notação infixa sem ter parênteses redundantes. Mostre a árvore gramatical anotada para a entrada `(((1 + 2) * (3 * 4)) + 5)`.

EXERCÍCIOS DE PROGRAMAÇÃO

P2.1 Implemente um tradutor de inteiros para números romanos baseado no esquema de tradução dirigido pela sintaxe desenvolvido no Exercício 2.9.

P2.2 Modifique o tradutor da Seção 2.9 para que produza como saída código para a máquina abstrata de pilha da Seção 2.8.

P2.3 Modifique o módulo de recuperação de erros do tradutor da Secção 2.9 a fim de que salte para a próxima expressão de entrada ao encontrar um erro.

P2.4 Expanda o tradutor da Seção 2.9 a fim de traduzir em enunciados de código de máquina de pilha os enunciados gerados pela seguinte gramática:

$$
\begin{aligned}
cmd &\rightarrow \textbf{id} := expr \\
&\mid \textbf{if}\ expr\ \textbf{then}\ cmd \\
&\mid \textbf{while}\ expr\ \textbf{do}\ cmd \\
&\mid \textbf{begin}\ opt_cmd\ \textbf{end} \\
opt_cmds &\rightarrow cmd_list \mid \epsilon \\
cmd_list &\rightarrow cmd_list\ ;\ cmd \mid cmd
\end{aligned}
$$

***P2.6** Construa um conjunto de expressões de teste para o compilador da Seção 2.9, de tal forma que cada produção seja usada pelo menos uma vez na derivação de alguma expressão de teste. Construa um programa de teste que possa ser usado como uma ferramenta geral de teste de compiladores. Use seu programa de teste para rodar seu compilador em tais expressões de teste.

P2.7 Construa um conjunto de enunciados para seu compilador do Exercício P2.5, de tal forma que cada produção seja usada pelo menos uma vez, a fim de gerar algum enunciado de teste. Use o programa de testes do Exercício P2.6 para rodar seu compilador sobre essas expressões de testes.

NOTAS BIBLIOGRÁFICAS

Este capítulo introdutório toca num número de assuntos que são tratados em mais detalhes pelo resto do livro. Remissivas à literatura aparecem nos capítulos contendo material posterior.

As gramáticas livres de contexto foram introduzidas por Chomsky [1956] como parte de um estudo das linguagens naturais. Seus usos na especificação da sintaxe das linguagens de programação emergiram independentemente. Enquanto trabalhava numa versão preliminar de Algol 60, John Backus "rapidamente adaptou [as produções de Emil Post] para aquele uso" (Wexelblat [1981, p. 162]). A notação resultante foi uma variante das gramáticas livres de contexto. O estudioso Panini divisou uma notação sintática equivalente para especificar as regras da gramática Sânscrita entre 400 a.C. e 200 a.C. (Ingerman[1967]).

A proposição de que BNF, que começou como uma abreviatura de Backus Normal Form, seja lida como Backus-Naur Form, para reconhecer as contribuições de Naur como editor do relatório de Algol 60 (Naur [1963]), está contida numa carta de Knuth[1964].

As definições dirigidas pela sintaxe são formas de definições indutivas na qual a indução está na estrutura sintática. Como tal, têm sido usadas há tempo informalmente na matemática. Suas aplicações às linguagens de programação chegam com o uso de uma gramática para estruturar o relatório Algol 60. Pouco depois, Irons [1961] construiu um compilador dirigido pela sintaxe.

A análise gramatical descendente recursiva tem sido usada desde o início dos anos 60. Bauer [1976] atribui o método a Lucas [1961]. Hoare [1962, p. 128] descreve um compilador Algol organizado como um "conjunto de procedimentos, cada um dos quais é capaz de processar uma das unidades sintáticas do relatório Algol 60". Foster [1968] discute a eliminação da recursão à esquerda das produções contendo ações semânticas que não afetam valores de atributos.

McCarthy [1963] advogou que a tradução de uma linguagem fosse baseada numa sintaxe abstrata. No mesmo *paper*, McCarthy [1963, p. 24] deixou o "leitor convencer a si mesmo" de que uma formulação epílogo-recursiva da função fatorial é equivalente a um programa interativo.

Os benefícios de se dividir um compilador numa vanguarda e numa retaguarda foram explorados num relatório de comitê por Strong *et al.* [1958]. O relatório cunhou o nome UNCOL (proveniente de Linguagem Universal Orientada para Computadores — *Universal Computer Oriented Language*) para uma linguagem intermediária universal. O conceito permaneceu como um ideal.

Uma boa forma de se aprender as técnicas de implementação é ler o código de compiladores existentes. Infelizmente, o código não é freqüentemente publicado. Randell e Russell [1964] fornecem uma contabilidade extensiva de um dos primeiros compiladores Algol. O código do compilador pode ser também visto em McKeeman, Horning e Wortman [1970]. Barron [1981] é uma coleção de artigos sobre implementação de Pascal, incluindo as notas de implementação distribuídas com o compilador Pascal P (Nori *et al.* [1981]), detalhes de geração de código (Ammann [1977]), e o código para uma implementação de Pascal S, um subconjunto de Pascal projetado por Wirth [1981] para uso de estudantes. Knuth [1985] fornece uma descrição inusitadamente clara e detalhada do tradutor TEX.

Kernighan e Pike [1984] descrevem em detalhes como construir um programa para simular uma calculadora de mesa em torno de um esquema de tradução dirigida pela sintaxe usando as ferramentas de construção de compiladores disponíveis no sistema operacional UNIX. A Equação (2.17) é proveniente de Tantzen [1963].

CAPÍTULO 3

A ANÁLISE LÉXICA

Este capítulo lida com as técnicas para especificar e implementar analisadores léxicos. Uma forma simples de se construir um analisador léxico é escrever um diagrama que ilustre a estrutura dos *tokens* da linguagem-fonte e então traduzi-lo manualmente num programa que os localize. Analisadores léxicos eficientes podem ser produzidos dessa forma.

As técnicas usadas para implementar os analisadores léxicos podem também ser aplicadas a outras áreas, tais como linguagens de interrogação e sistemas de recuperação de informações. Em cada aplicação, o problema subjacente é a especificação e o projeto de programas que executem ações disparadas por padrões nas cadeias de caracteres. Como a programação dirigida para o processamento de padrões é útil em geral, introduzimos uma linguagem de ações baseadas em padrões, chamada Lex, para a especificação de analisadores léxicos. Nessa linguagem, os padrões são especificados por expressões regulares, e um compilador Lex pode gerar um autômato finito que reconheça expressões regulares.

Várias outras linguagens usam expressões regulares para descrever padrões. Por exemplo, a linguagem de reconhecimento de padrões AWK usa expressões regulares para selecionar linhas de entrada para processamento e o *shell* do sistema UNIX permite que o usuário se refira a um conjunto de nomes de arquivos escrevendo uma expressão regular. O comando UNIX `rm *.o`, por exemplo, remove todos os arquivos com nomes terminados em "`.o`".[1]

Uma ferramenta de *software* que automatize a construção de analisadores léxicos permite que pessoas com diferentes formações usem o reconhecimento de padrões em suas próprias áreas de aplicação. Por exemplo, Jarvis [1976] usou um gerador de analisadores léxicos para criar um programa que reconhecesse as imperfeições de placas de circuitos impressos. Os circuitos eram varridos digitalmente e convertidos em "cadeias" de segmentos de linhas em ângulos diferentes. O "analisador léxico" procurava pelos padrões que correspondessem às imperfeições na cadeia de segmentos de linha. Uma vantagem maior de um gerador de analisadores léxicos é que o mesmo pode utilizar os algoritmos de reconhecimento de padrões mais conhecidos e dessa forma criar analisadores léxicos eficientes para pessoas que não sejam especialistas em técnicas de reconhecimento.

3.1 O PAPEL DO ANALISADOR LÉXICO

O analisador léxico é a primeira fase de um compilador. Sua tarefa principal é a de ler os caracteres de entrada e produzir uma seqüência de *tokens* que o *parser* utiliza para a análise sintática. Essa interação, sumarizada esquematicamente na Fig. 3.1, é comumente implementada fazendo-se com que o analisador léxico seja uma sub-rotina ou uma co-rotina do *parser*. Ao receber do *parser* um comando "obter o próximo *token*", o analisador léxico lê os caracteres de entrada até que possa identificar o próximo *token*.

Como o analisador léxico é a parte do compilador que lê o texto-fonte, também pode realizar algumas tarefas secundárias ao nível da *interface* com o usuário. Uma delas é a de remover do programa-fonte os comentários e espaços em branco, os últimos sob a forma de espaços, tabulações e caracteres de avanço de linha. Uma outra é a de correlacionar as mensagens de erro do compilador com o programa-fonte. Por exemplo, o analisador léxico pode controlar o número de caracteres examinados, de tal forma que um número de linha possa ser relacionado a uma mensagem de erro. Em alguns compiladores, o analisador léxico fica com a responsabilidade de fazer uma cópia do programa-fonte com as mensagens de erro associadas ao mesmo. Se a linguagem-fonte suporta algumas funções sob a forma de macros pré-processadas, as mesmas também podem ser implementadas na medida em que a análise léxica vá se desenvolvendo.

Algumas vezes, os analisadores léxicos são divididos em duas fases em cascata, a primeira chamada de "varredura" (*scanning*) e a segunda de "análise léxica". O *scanner* é responsável por realizar tarefas simples, enquanto o analisador léxico propriamente dito realiza as tarefas mais complexas. Por exemplo, um compilador Fortran poderia usar um *scanner* para eliminar os espaços da entrada.

Temas da Análise Léxica

Existem várias razões para se dividir a fase de análise da compilação em análise léxica e análise gramatical (*parsing*).

1. Um projeto mais simples talvez seja a consideração mais importante. A separação das análises léxica e sintática freqüentemente nos permite simplificar uma ou outra dessas fases. Por exemplo, um *parser* que incorpore as convenções para comentários e espaços em branco é significativamente mais complexo do que um que assuma que os mesmos já tenham sido removidos pelo analisador léxico. Se estivermos projetando uma nova linguagem, separar as convenções léxicas das sintáticas pode levar a um projeto global de linguagem mais claro.
2. A eficiência do compilador é melhorada. Um analisador léxico separado nos permite construir um processador especializado e potencialmente mais eficiente para a tarefa. Uma grande quantidade de tempo é gasta lendo-se o programa-fonte e particionando-o em

[1] A expressão `*.o` é uma variante da notação usual para expressões regulares. Os Exercícios 3.10 e 3.14 mencionam algumas variantes das notações para expressões regulares mais comumente usadas.

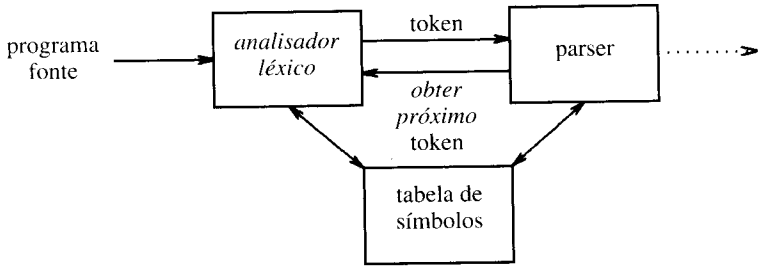


Fig. 3.1. Interação do analisador léxico com o *parser*.

tokens. Técnicas de **buferização** especializadas para a leitura de caracteres e o processamento de *tokens* podem acelerar significativamente o desempenho de um compilador.

3. A portabilidade do compilador é realçada. As peculiaridades do alfabeto de entrada e outras anomalias específicas de dispositivos podem ser restringidas ao analisador léxico. A representação de símbolos especiais ou não-padrão, tais como ↑ em Pascal, pode ser isolada no analisador léxico.

Ferramentas especializadas têm sido projetadas para auxiliar a automação da construção de analisadores léxicos e *parsers* quando os mesmos estão separados. Veremos exemplos globais de tais ferramentas neste livro.

Tokens, Padrões, Lexemas

Quando se fala sobre a análise léxica, usamos os termos "*token*", "padrão" e "lexema" com significados específicos. Exemplos de seus usos são mostrados na Fig. 3.2. Em geral, existe um conjunto de cadeias de entrada para as quais o mesmo *token* é produzido como saída. Esse conjunto de cadeias é descrito por uma regra chamada de um *padrão* associado ao *token* de entrada. O padrão é dito *reconhecer* cada cadeia do conjunto. Um lexema é um conjunto de caracteres no programa-fonte que é reconhecido pelo padrão de algum *token*. Por exemplo, no enunciado Pascal

```
const pi = 3.1416;
```

a subcadeia `pi` é um lexema para o *token* "identificador".

Tratamos os *tokens* como símbolos terminais na gramática para a linguagem-fonte, usando nomes em negrito para representá-los. Os lexemas reconhecidos pelo padrão do *token* representam cadeias de caracteres no programa-fonte, e podem receber um tratamento conjunto, como instâncias de uma mesma unidade léxica (por exemplo, instâncias de identificadores, números etc.).

Na maioria das linguagens de programação, as seguintes construções são tratadas como *tokens*: palavras-chave, operadores, identificadores, constantes, literais, cadeias e símbolos de pontuação, como parênteses, vírgulas e ponto-e-vírgulas. No exemplo acima, quando a seqüência de caracteres `pi` aparece no programa-fonte, um *token* representando um identificador é repassado ao *parser*. O repasse de um *token* é freqüentemente implementado transmitindo-se um inteiro associado ao *token*. É justamente esse inteiro que é designado por **id** na Fig. 3.2.

Um padrão é uma regra que descreve o conjunto de lexemas que podem representar um *token* particular nos programas-fonte. O padrão para o *token* **const** na Fig. 3.2 é exatamente a cadeia singela `const`, que soletra a palavra-chave. O padrão para o *token* **relação** é o conjunto de todos os seis operadores relacionais de Pascal. Para descrever precisamente os padrões para *tokens* mais complexos, como **id** (identificador) e **num** (número), iremos usar a notação de expressões regulares desenvolvida na Seção 3.3.

Certas convenções de linguagem aumentam a dificuldade da análise léxica. Linguagens como Fortran requerem certas construções em posições fixas na linha de entrada. Dessa forma, o alinhamento de um lexema pode ser importante na determinação da correção de um programa-fonte. A tendência no projeto moderno de linguagens de programação está na direção de entradas em formato livre, permitindo que as construções sejam colocadas em qualquer local da linha de entrada, e, por conseguinte, esse aspecto da análise léxica vem se tornando menos importante.

O tratamento dos espaços varia grandemente de linguagem para linguagem. Em algumas linguagens, como Fortran e Algol 68, os espaços não são significativos, exceto quando dentro de literais do tipo cadeia de caracteres. Podem ser adicionados à vontade a fim de melhorar a legibilidade de um programa. As convenções relacionadas aos espaços podem complicar grandemente a tarefa de identificação dos *tokens*.

Um exemplo popular que ilustra a dificuldade potencial em se reconhecer *tokens* é o enunciado `DO` de Fortran. No comando

```
DO 5 I = 1.25
```

não podemos afirmar que `DO` seja parte do identificador `DO5I`, e não um identificador em si, até que tenhamos examinado o ponto decimal. Por outro lado, no enunciado

```
DO 5 I = 1,25
```

TOKEN	LEXEMAS EXEMPLO	DESCRIÇÃO INFORMAL DO PADRÃO
const	`const`	`const`
if	`if`	`if`
relação	`<, <=, =, <>, >, >=`	`< ou <= ou = ou <> ou >= ou >`
id	`pi, contador, D2`	letra seguida por letras e/ou dígitos
num	`3.1416, 0, 6.02E23`	qualquer constante numérica
literal	`"conteúdo da memória"`	quaisquer caracteres entre aspas, exceto aspa

Fig. 3.2. Exemplos de *tokens*.

temos sete *tokens*: a palavra-chave `DO`, o rótulo de enunciado `5`, o identificador `I`, o operador `=`, a constante `1`, a vírgula e a constante `25`. Aqui não podemos estar certos, até que tenhamos examinado a vírgula, de que `DO` seja uma palavra-chave. Para aliviar esta incerteza, Fortran 77 permite que uma vírgula opcional seja colocada entre o rótulo e o índice do enunciado `DO` (no exemplo, a variável I). O uso dessa vírgula é encorajado porque a mesma ajuda a tornar o enunciado `DO` mais claro e legível.

Em muitas linguagens, certas cadeias são *reservadas*, isto é, seus significados são pré-definidos e não podem ser modificados pelo usuário. Se uma palavra-chave não for reservada, o analisador léxico precisará distinguir uma palavra-chave de um identificador definido pelo usuário. Em PL/I, as palavras-chave não são reservadas; conseqüentemente, as regras para essa distinção são um tanto complicadas, como o seguinte enunciado PL/I ilustra:

```
IF THEN THEN THEN = ELSE; ELSE ELSE = THEN;
```

Atributos para os *Tokens*

Quando um lexema for reconhecido por mais de um padrão, o analisador léxico precisará providenciar informações adicionais para as fases subseqüentes do compilador a respeito do lexema particular que foi reconhecido. Por exemplo, o padrão **num** reconhece as duas cadeias 0 e `1`, mas é essencial para o gerador de código ser informado sobre que cadeia foi efetivamente reconhecida.

O analisador léxico coleta informações a respeito dos *tokens* em seus atributos associados. Os *tokens* influenciam decisões na análise gramatical; os atributos influenciam a tradução dos *tokens*. Do ponto de vista prático, o *token* possui usualmente somente um único atributo — um apontador para a entrada da tabela de símbolos na qual as informações sobre o mesmos são mantidas; o apontador se torna o atributo do *token*. Para fins de diagnóstico, podemos estar interessados tanto no lexema de um identificador quanto no número da linha na qual o mesmo foi primeiramente examinado. Esses dois itens de informação podem, ambos, ser armazenados na entrada da tabela de símbolos para o identificador.

Exemplo 3.1. Os *tokens* e os valores de atributos associados ao enunciado Fortran

```
E = M * C ** 2
```

são escritos abaixo como uma seqüência de pares:

<**id,** apontador para a entrada da tabela de símbolos para `E`>

<**operador_de_atribuição,**>

<**id,** apontador para a entrada da tabela de símbolos para `M`>

<**operador_de_multiplicação,**>

<**id,** apontador para a entrada da tabela de símbolos para `C`>

<**operador_de_exponenciação,**>

<**num,** valor inteiro 2>

Note-se que em certos pares não existe a necessidade de um valor de atributo; o primeiro componente é suficiente para identificar o lexema. Nesse pequeno exemplo, ao *token* **num** foi associado um atributo de valor inteiro. O compilador pode armazenar a cadeia de caracteres que forma o número numa tabela de símbolos e deixar o atributo do *token* **num** ser o apontador para a entrada da tabela. □

Erros Léxicos

Poucos erros são distinguíveis somente no nível léxico, uma vez que um analisador léxico possui uma visão muito local do programa-fonte. Se a cadeia `fi` for encontrada pela primeira vez num programa C, no contexto

```
fi ( a == f(x) ) ...
```

um analisador léxico não poderá dizer se `fi` é a palavra-chave `if` incorretamente grafada ou um identificador de função não declarada. Como `fi` é um identificador válido, o analisador léxico precisa retornar o *token* identificador e deixar alguma fase posterior do compilador tratar o eventual erro.

Mas, suponhamos que emerja uma situação na qual o analisador léxico seja incapaz de prosseguir, porque nenhum dos padrões reconheça um prefixo na entrada remanescente. Talvez a estratégia mais simples de recuperação seja a da "modalidade pânico". Removemos sucessivos caracteres da entrada remanescente até que o analisador léxico possa encontrar um *token* bem-formado. Essa técnica de recuperação pode ocasionalmente confundir o *parser*, mas num ambiente de computação interativo pode ser razoavelmente adequada.

Outras possíveis ações de recuperação de erros são:

1. remover um caractere estranho
2. inserir um caractere ausente
3. substituir um caractere incorreto por um correto
4. transpor dois caracteres adjacentes.

Transformações de erros como essas podem ser experimentadas numa tentativa de se consertar a entrada. A mais simples de tais estratégias é a de verificar se um prefixo da entrada remanescente pode ser transformado num lexema válido através de uma única transformação. Essa estratégia assume que a maioria dos erros léxicos seja resultante de um único erro de transformação, uma suposição usualmente confirmada na prática, embora nem sempre.

Uma forma de se encontrar erros num programa é computar o número mínimo de transformações de erros requeridas para tornar um programa errado num que seja sintaticamente bem-formado. Dizemos que um programa errado possui k erros se a menor seqüência de transformações de erros que irá mapeá-lo em algum programa válido possui comprimento k. A correção de erros de distância mínima é uma conveniente ferramenta teórica de longo alcance, mas que não é geralmente usada por ser custosa demais de implementar. Entretanto, uns poucos compiladores experimentais têm usado o critério da distância mínima para realizar correções localizadas.

3.2 BUFERIZAÇÃO DA ENTRADA

Esta seção cobre alguns temas de desempenho relacionados à buferização da entrada. Mencionamos primeiramente um esquema de entrada com dois *buffers*, que é útil quando uma pré-varredura da entrada é necessária para a identificação de *tokens*. Introduzimos, então, algumas técnicas úteis para acelerar o analisador léxico, tais como o uso de "sentinelas", a fim de marcar o final do *buffer*.

Existem três enfoques gerais para a implementação de um analisador léxico:

1. Usar um gerador de analisadores léxicos, tal como o compilador Lex discutido na Seção 3.5, a fim de produzir um analisador léxico a partir de uma especificação baseada em expressões regulares. Nesse caso, o gerador providencia as rotinas para a leitura e a buferização da entrada.
2. Escrever um analisador léxico numa linguagem de programação de sistemas convencional, usando as facilidades de E/S da mesma para ler a entrada.

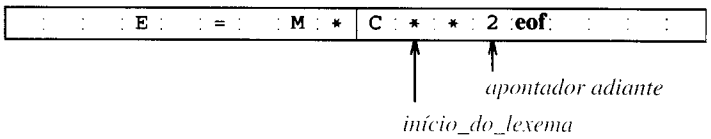


Fig. 3.3. Um *buffer* de entrada em duas metades.

3. Escrever o analisador léxico em linguagem de montagem e manipular explicitamente a leitura da entrada.

As três escolhas estão listadas em ordem de complexidade crescente para o implementador. Infelizmente, as abordagens mais difíceis de implementar produzem freqüentemente os analisadores léxicos mais rápidos. Como o analisador léxico é a única fase do compilador a ler o programa-fonte caractere a caractere, é possível se gastar uma quantidade considerável de tempo na fase de análise léxica, ainda que as fases posteriores sejam conceitualmente mais complexas. Conseqüentemente, a velocidade do analisador léxico é um tema do projeto de compiladores. Ainda que o núcleo deste capítulo esteja devotado à primeira abordagem — projeto e uso de um gerador automático —, também consideramos técnicas que sejam úteis num projeto manual. A Seção 3.4 discute os diagramas de transições, os quais são um conceito valioso para a organização de um analisador léxico projetado à mão.

Pares de *Buffers*

Em muitas linguagens-fonte, existem momentos em que o analisador léxico precisa examinar vários caracteres à frente do lexema, antes que seja anunciado um reconhecimento. Os analisadores léxicos no Capítulo 2 usaram a função `ungetc` para empilhar de volta no fluxo de entrada os caracteres pré-esquadrinhados. Como uma grande quantidade de tempo pode ser consumida movendo-se caracteres, técnicas especializadas de buferização têm sido desenvolvidas, de forma a reduzir a sobrecarga imposta no processamento de um caractere. Muitos esquemas de buferização podem ser usados, mas, como as técnicas são um tanto dependentes de parâmetros de sistema, iremos delinear aqui somente os princípios por trás de uma única classe de esquemas.

Iremos usar um *buffer* dividido em duas metades, com N caracteres cada uma, conforme mostrado na Fig. 3.3. Tipicamente, N é o número de caracteres em um bloco de disco, por exemplo, 1024 ou 4096.

Lemos N caracteres de entrada em cada metade do *buffer* através de um único comando de leitura do sistema, em lugar de invocar um comando de leitura para cada caractere de entrada. Se restarem menos do que N caracteres na entrada, um caractere especial **eof** (fim de arquivo) é lido no *buffer*, após os N caracteres, como na Fig. 3.3. Ou seja, **eof** marca o final do arquivo-fonte e é diferente de qualquer caractere de entrada.

Dois apontadores para o *buffer* de entrada são mantidos. A cadeia de caracteres entre os dois apontadores é o lexema corrente. Inicialmente, ambos os apontadores indicam o primeiro caractere do próximo lexema a ser levantado. Um, chamado de apontador adiante, esquadrinha à frente até que ocorra o reconhecimento de um padrão. Uma vez que o próximo lexema seja determinado, o apontador adiante passa a indicar o caractere mais à direita do mesmo. Após o lexema ser processado, ambos os apontadores passam a sinalizar o caractere imediatamente após esse lexema. Com esse esquema, os comentários e espaços em branco passam a ser tratados como padrões que não produzem *token* algum.

Se o apontador adiante estiver prestes a ser deslocado para além da marca de meio, a metade à direita do *buffer* é preenchida com N novos caracteres de entrada. Se o apontador adiante estiver prestes a ser deslocado para além do fim à direita do *buffer* a metade à esquerda é preenchida com N novos caracteres e o apontador adiante volta para o início do *buffer*.

Esse esquema de buferização funciona bem a maior parte do tempo, mas o esquadrinhamento adiante é limitado e, dada tal circunstância, pode ser impossível reconhecer *tokens* nas situações em que o apontador adiante precise viajar uma distância maior do que o comprimento do *buffer*. Por exemplo, se examinarmos

```
DECLARE ( ARG1, ARG2, . . ., ARGn )
```

num programa PL/I, não podemos determinar se `DECLARE` é uma palavra-chave ou um nome de *array* até que vejamos o caractere que se segue ao parênteses à direita. Num e noutro casos, o lexema termina no segundo `E`, mas a quantidade de esquadrinhamento adiante necessitada é proporcional ao número de argumentos, o qual, em princípio, é ilimitado.

Sentinelas

Se usarmos o esquema da Fig. 3.3 exatamente como mostrado, precisaremos verificar, a cada vez em que movermos o apontador adiante, se não o deslocamos para além de uma das metades do *buffer*; se o fizermos, precisaremos recarregar a outra metade. Ou seja, nosso código para avançar o apontador adiante realiza testes como aqueles mostrados na Fig. 3.4.

Exceto aos finais das metades dos *buffers*, o código da Fig. 3.4 requer dois testes para cada avanço do apontador adiante. Podemos reduzi-los para um, se ampliarmos cada metade do *buffer* de forma a que contenham um caractere de *sentinela* em cada final. O sentinela é um caractere especial que não pode ser parte do programa-fonte. Uma escolha natural é **eof;** a Fig. 3.5 mostra o mesmo arranjo de *buffer* que a Fig. 3.3, com os sentinelas adicionados.

Com o arranjo da Fig. 3.5, podemos usar o código mostrado na Fig. 3.6 para avançar o apontador adiante (e testar pelo fim do arquivo-fonte). O código realiza, na maior parte do tempo, somente um teste para verificar se o apontador *adiante* endereça uma marca de fim de arquivo **(eof).** Somente quando atingirmos o fim de uma metade de um *buffer* ou o fim do arquivo precisaremos realizar mais testes. Como N caracteres de entrada são encontrados entre **eof**'s, o número médio de testes por caractere de entrada é muito próximo a 1.

se *apontador_adiante* estiver ao fim da primeira metade
então início
    recarregar a segunda metade;

apontador_adiante : = *apontador_adiante* + 1

fim
senão se *apontador_adiante* estiver ao fim da segunda metade
então início
    recarregar a primeira metade;
    deslocar *apontador_adiante* para o início da primeira metade

fim
senão *apontador_adiante* : = *apontador_adiante* + 1;

Fig. 3.4. Código para avançar o apontador adiante.

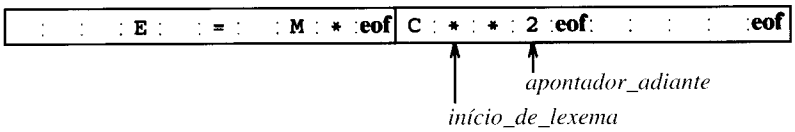


Fig. 3.5. Sentinelas ao fim de cada metade do *buffer*.

Também precisamos decidir como processar o caractere esquadrinhado pelo apontador adiante: marca o final do *token*, representa uma antecipação na procura de uma palavra-chave particular ou o quê? Uma forma de estruturar esses testes é usar um enunciado *case*, se a implementação da linguagem possuir um. O teste

se *apontador_adiante* ↑ = **eof**

pode ser implementado como uma das diferentes alternativas de um enunciado *case*.

3.3 ESPECIFICAÇÃO DOS *TOKENS*

As expressões regulares são uma notação importante para especificar padrões. Cada padrão corresponde a um conjunto de cadeias e, dessa forma, as expressões regulares servirão como nomes para conjuntos de cadeias. A Seção 3.5 estende esta notação numa linguagem orientada a padrões para a análise léxica.

Cadeias e Linguagens

O termo *alfabeto* ou *classe de caracteres* denota qualquer conjunto finito de símbolos. Exemplos típicos de símbolos são letras e caracteres. O conjunto {0,1} é o *alfabeto binário*. EBCDIC e ASCII são dois exemplos de alfabetos de computadores.

Uma *cadeia* sobre algum alfabeto é uma seqüência finita de símbolos retirados do mesmo. Na teoria das linguagens, os termos *sentença* e *palavra* são freqüentemente usados como sinônimos para o termo "cadeia". O comprimento da cadeia s, usualmente escrito $|s|$, é o número de ocorrências de símbolos em s. Por exemplo, banana é uma cadeia de comprimento seis. A cadeia *vazia*, denotada ϵ, é uma cadeia especial de comprimento zero. Alguns termos comuns, associados a partes de uma cadeia, estão sumarizados na Fig. 3.7.

O termo *linguagem* denota qualquer conjunto de cadeias sobre algum alfabeto fixo. A definição é muito ampla. Linguagens abstratas como $\varnothing$, o conjunto *vazio*, ou $\{\epsilon\}$, o conjunto contendo somente a cadeia vazia, são linguagens sob essa definição. Também o são o conjunto de todos os programas Pascal sintaticamente bem-formados, e o conjunto de todas as sentenças gramaticalmente corretas em inglês, apesar dos dois últimos conjuntos serem muito difíceis de se especificar. Note-se também que essa definição não prescreve nenhum significado às cadeias de uma linguagem. Os métodos para a prescrição de significados são discutidos no Capítulo 5.

Se x e y são cadeias, então a *concatenação* de x e y, escrita xy é a cadeia formada atrelando-se y a x. Por exemplo, se x = cão e y = casa, então xy = cãocasa. A cadeia vazia é o elemento identidade da concatenação. Isto é, $s\epsilon = \epsilon s = s$.

Se pensarmos na concatenação como um "produto", podemos definir a cadeia "exponenciação" como segue. Definir s^0 como sendo ϵ e, para $i > 0$, definir s^i como sendo $s^{i-1}s$. Como ϵs é a própria s, $s^1 = s$. Dessa forma, $s^2 = ss$, $s^3 = sss$ e assim por diante.

Operações em Linguagens

Existem várias operações importantes que podem ser aplicadas às linguagens. Para a análise léxica, estaremos primariamente interessados na união, concatenação e fechamento, as quais são definidas na Fig. 3.8. Podemos também generalizar o operador de "exponenciação" para linguagens, definindo-se L^0 como sendo $\{\epsilon\}$ e L^i como $L^{i-1}L$. Dessa forma, L^i é L concatenada consigo mesma $i-1$ vezes.

Exemplo 3.2. Seja L o conjunto {A, B, ..., Z, a, b, ..., z} e D o conjunto {0, 1, ..., 9}. Podemos pensar de L e D em duas formas. L pode ser o alfabeto que consiste no conjunto de letras maiúsculas e minúsculas e D o dos dez dígitos decimais. Alternativa-

```
apontador_adiante : = apontador_adiante + 1;
se apontador_adiante ↑ = eof
então início
        se apontador_adiante estiver ao fim da primeira metade
        então início
                recarregar segunda metade;
                apontador_adiante : = apontador_adiante + 1
        fim
        senão se apontador_adiante estiver ao fim da segunda
        metade
        então início
                recarregar primeira metade;
                mover apontador_adiante para o início da
                primeira metade
        fim
        senão / * eof está no buffer indicando fim da entrada * /
                terminar a análise léxica
fim
```

Fig. 3.6. Código de esquadrinhamento antecipado com sentinelas.

TERMO	DEFINIÇÃO
prefixo de s	Uma cadeia obtida pela remoção de zero ou mais símbolos ao fim da cadeia s; por exemplo, ban é um prefixo de banana.
sufixo de s	Uma cadeia obtida pela remoção de zero ou mais símbolos ao início da cadeia s; por exemplo, nana é um sufixo de banana.
subcadeia de s	Uma cadeia obtida pela remoção de um prefixo e de um sufixo de s; por exemplo, nan é uma subcadeia de banana. Cada prefixo e cada sufixo de s é uma subcadeia de s, mas nem toda subcadeia de s é um prefixo ou um sufixo de s. Para cada cadeia s, tanto s quanto ϵ são prefixos, sufixos e subcadeias de s.
prefixo, sufixo ou subcadeia *própria* de s	Qualquer cadeia não vazia x que seja, respectivamente, prefixo, sufixo ou subcadeia de s, tal que $s \neq x$.
subseqüência de s	Qualquer cadeia formada pela remoção de zero ou mais símbolos não necessariamente contíguos de s; por exemplo, baaa é uma subseqüência de banana.

Fig. 3.7. Termos para partes de uma cadeia.

OPERAÇÃO	DEFINIÇÃO
união de L e M, escrita $L \cup M$	$L \cup M = \{ s \mid s$ está em L ou s está em $M \}$
concatenação de L e M, escrita LM	$LM = \{ st \mid s$ está em L e t está em $M \}$
fechamento Kleene de L, escrito L^*	$L^* = \bigcup_{i=0}^{\infty} L^i$ L^* denota "zero ou mais concatenações de" L.
fechamento positivo de L, escrito L^+	$L^+ = \bigcup_{i=1}^{\infty} L^i$ L^+ denota "uma ou mais concatenações de" L.

Fig. 3.8. Definições de operações em linguagens.

mente, como um símbolo pode ser considerado uma cadeia de comprimento um, os conjuntos L e D são, cada um, linguagens finitas. Aqui estão alguns exemplos de novas linguagens criadas a partir de L e D, através da aplicação dos operadores definidos na Fig. 3.8.

1. $L \cup D$ é o conjunto de letras e dígitos.
2. LD é o conjunto de cadeias consistindo em uma letra seguida por um dígito.
3. L^4 é o conjunto de todas as cadeias com quatro letras.
4. L^* é o conjunto de todas as cadeias de letras, incluindo ϵ, a cadeia vazia.
5. $L(L \cup D)^*$ é o conjunto de todas as cadeias de letras e dígitos, que começam por uma letra.
6. D^+ é o conjunto de todas as cadeias de um ou mais dígitos. □

Expressões Regulares

Em Pascal, um identificador é uma letra seguida por zero ou mais letras ou dígitos; isto é, um identificador é membro do conjunto definido na parte (5) do Exemplo 3.2. Nesta seção, apresentamos uma notação, chamada de expressões regulares, que nos permite definir precisamente conjuntos daquela natureza. Com essa notação, definiríamos identificadores Pascal como

letra (letra | dígito)*

Aqui, a barra vertical significa "ou", os parênteses são usados para agrupar subexpressões, o asterisco significa "zero ou mais instâncias" da expressão parentetizada e a justaposição de **letra** com o resto da expressão significa concatenação.

Uma expressão regular é constituída de expressões regulares mais simples usando-se um conjunto de regras de definição. Cada expressão regular r denota uma linguagem $L(r)$. As regras de definição especificam como $L(r)$ é formada através da combinação, em várias formas, das linguagens denotadas pelas subexpressões de r.

A seguir estão as regras que definem as *expressões regulares sobre um alfabeto* Σ. Associada a cada regra existe uma especificação da linguagem denotada pela expressão regular sendo definida.

1. ϵ é uma expressão regular que denota $\{\epsilon\}$, isto é, o conjunto que contém a cadeia vazia.
2. Se a é um símbolo em Σ, então a é uma expressão regular que denota $\{a\}$, isto é, o conjunto contendo a cadeia a. Apesar de usarmos a mesma notação para todos os três, tecnicamente, a expressão regular a é diferente da cadeia a e do símbolo a. Ficará mais claro a partir do contexto se estaremos falando sobre a como uma expressão regular, cadeia ou símbolo.
3. Suponhamos que r e s sejam expressões regulares denotando as linguagens $L(r)$ e $L(s)$. Dessa forma,

   a) $(r) \mid (s)$ é uma expressão regular denotando $L(r) \cup L(s)$.
   b) $(r)(s)$ é uma expressão regular denotando $L(r)L(s)$.
   c) $(r)^*$ é uma expressão regular denotando $(L(r))^*$.
   d) (r) é uma expressão regular denotando $L(r)$.[2]

A linguagem denotada por uma expressão regular é dita ser um *conjunto regular*.

A especificação de uma expressão regular é um exemplo de definição recursiva. As regras (1) e (2) formam a base da definição; usamos o termo *símbolo básico* a fim de nos referirmos a ϵ ou a um símbolo em Σ figurando numa expressão regular. A regra (3) providencia o passo de indução.

Os parênteses desnecessários podem ser evitados nas expressões regulares, se adotarmos convenções em que:

1. o operador unário * possua a maior precedência e seja associativo à esquerda;
2. a concatenação tenha a segunda maior precedência e seja associativa à esquerda;
3. | possua a menor precedência e seja associativo à esquerda.

Sob essas convenções, $(a) \mid ((b)^*(c))$ é equivalente a $a \mid b^*c$. Ambas as expressões denotam o conjunto de cadeias que sejam um único a ou zero ou mais b's seguidos por um único c.

Exemplo 3.3. Seja $\Sigma = \{a, b\}$.

1. A expressão regular $a \mid b$ denota o conjunto $\{a, b\}$.
2. A expressão regular $(a \mid b)(a \mid b)$ denota $\{aa, ab, ba, bb\}$, o conjunto de todas as cadeias de a's e b's de comprimento dois. Outra expressão regular para esse mesmo conjunto é $aa \mid ab \mid ba \mid bb$.
3. A expressão regular a^* denota o conjunto de todas as cadeias de zero ou mais a's, isto é, $\{\epsilon, a, aa, aaa, \ldots\}$.
4. A expressão regular $(a \mid b)^*$ denota o conjunto de todas as cadeias contendo zero ou mais instâncias de um a ou um b, ou seja, o con-

[2] A regra informa que os pares extras de parênteses podem ser colocados em torno das expressões regulares se o desejarmos.

AXIOMA	DESCRIÇÃO
$r \mid s = s \mid r$	$\mid$ é comutativa
$r \mid (s \mid t) = (r \mid s) \mid t$	$\mid$ é associativa
$(rs)\,t = r\,(st)$	a concatenação é associativa
$r\,(s \mid t) = rs \mid rt$ $(s \mid t)\,r = sr \mid tr$	a concatenação se distribui sobre $\mid$
$\epsilon r = r$ $r\epsilon = r$	ϵ é o elemento identidade da concatenação
$r^* = (r \mid \epsilon)^*$	relação entre ϵ e *
$r^{**} = r^*$	* é idempotente

Fig. 3.9. Propriedades algébricas das expressões regulares.

junto de todas as cadeias de a's e b's. Outra expressão regular para esse conjunto é $(a^*b^*)^*$.

5. A expressão regular $a \mid a^*b$ denota o conjunto contendo a cadeia a e todas as cadeias consistindo em zero ou mais a's seguidos por um b. □

Se duas expressões regulares r e s denotam a mesma linguagem, dizemos que r e s são *equivalentes* e escrevemos $r = s$. Por exemplo, $(a \mid b) = (b \mid a)$.

Existe um número de leis algébricas, obedecidas pelas expressões regulares, que pode ser usado para manipulá-las em formas equivalentes. A Figura 3.9 mostra algumas leis algébricas que vigoram para as expressões regulares r, s e t.

Definições Regulares

Por uma conveniência de notação, podemos desejar dar nomes a expressões regulares, bem como definir outras, usando esses nomes como se fossem símbolos. Se Σ for um alfabeto de símbolos básicos, então uma *definição regular* é uma seqüência de definições da forma

$$d_1 \rightarrow r_1$$
$$d_2 \rightarrow r_2$$
$$\ldots$$
$$d_n \rightarrow r_n$$

onde cada d_i é um nome distinto e cada r_i, uma expressão sobre os símbolos em $\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$, isto é, os símbolos básicos e os nomes previamente definidos. Restringindo-se cada r_i a símbolos de Σ e aos nomes previamente definidos, podemos construir uma expressão regular sobre Σ para qualquer r_i substituindo-se repetidamente os nomes de expressões regulares pelas expressões que os mesmos denotam. Se r_i usasse d_j para algum $j \geq i$, então r_i poderia ser recursivamente definida e o processo de substituição não terminaria.

A fim de distinguir os nomes dos símbolos, imprimimos os nomes nas definições regulares em negrito.

Exemplo 3.4. Como havíamos estabelecido, o conjunto de identificadores Pascal é o conjunto de cadeias de letras e dígitos começando por uma letra. A seguir vem uma definição regular para esse conjunto.

letra → `A | B ... | Z | a | b | ... | z`
dígito → `0 | 1 | ... | 9`
id → **letra** | (**letra** | **dígito**)* □

Exemplo 3.5. Os números sem sinal em Pascal são cadeias tais como `5280`, `39.37`, `6.336E4` ou `1.894E-4`. A seguinte definição regular providencia uma precisa especificação para essa classe de cadeias:

dígito → `0 | 1 | ... | 9`
dígitos → **dígito dígito***
fração_opcional → `.` **dígitos** | ϵ
expoente_opcional → (`E` (`+` | `-` | ϵ) **dígitos**) | ϵ
num → **dígitos fração_opcional expoente_opcional**

Esta definição diz que uma **fração_opcional** ou é um ponto decimal seguido por um ou mais dígitos, ou está ausente (a cadeia vazia). Um **expoente_opcional,** se não estiver ausente, é um `E`, seguido por um sinal `+` ou `-` opcional, seguido por um ou mais dígitos. Note-se que pelo menos um dígito precisa se seguir ao ponto, e, dessa forma, **num** não reconhece `1.`, mas reconhece `1.0`. □

Simplificações Notacionais

Algumas construções ocorrem tão freqüentemente nas expressões regulares que é conveniente introduzir algumas simplificações notacionais para as mesmas.

1. *Uma ou mais ocorrências.* O operador unário pós-fixo $^+$ significa "uma ou mais ocorrências de". Se r for uma expressão regular que denote a linguagem $L(r)$, então $(r)^+$ é uma expressão regular que denota a linguagem $(L(r))^+$. Dessa forma, a expressão regular a^+ denota o conjunto de todas as cadeias de um ou mais a's. O operador $^+$ possui a mesma precedência e associatividade que o operador *. As duas identidades algébricas $r^* = r^+ \mid \epsilon$ e $r^+ = rr^*$ relacionam os operadores de fechamento Kleene e positivo.
2. *Zero ou mais ocorrências.* O operador pós-fixo unário? significa "zero ou uma ocorrência de". A notação r? é uma simplificação para $r \mid \epsilon$. Se r for uma expressão regular, então (r)? denota a linguagem $L(r) \cup \{\epsilon\}$. Por exemplo, usando-se os operadores $^+$ e ?, podemos reescrever a definição regular de **num** no Exemplo 3.5 como

dígito → `0 | 1 | ... | 9`
dígitos → **dígito**$^+$
fração_opcional → (`.` **dígitos**)?
expoente_opcional → (`E` (`+` | `-`)? **dígitos**)?
num → **dígitos fração_opcional expoente_opcional**

3. *Classes de caracteres.* A notação `[abc]`, onde `a`, `b` e `c` são símbolos de alfabeto, denota a expressão regular `a | b | c`. Uma classe de caracteres abreviada, tal como `[a-z]` denota a expressão regular `a | b | ... | z`. Usando-se classes de caracteres, podemos descrever identificadores como sendo cadeias geradas pela expressão regular

```
[A-Za-z][A-Za-z0-9]*
```

Conjuntos Não-regulares

Algumas linguagens não podem ser descritas por qualquer expressão regular. A fim de ilustrar os limites do poder descritivo das expressões regulares, fornecemos, neste ponto, exemplos de construções de linguagens de programação que não podem ser descritas por expressões regulares. As provas dessas assertivas podem ser encontradas nas referências.

As expressões regulares não podem ser usadas para descrever parênteses balanceados ou construções aninhadas. Por exemplo, o conjunto de todas as cadeias de parênteses balanceados não pode ser des-

crito por uma expressão regular. Por outro lado, tal conjunto pode ser descrito por uma gramática livre de contexto.

As cadeias repetitivas não podem ser descritas pelas expressões regulares. O conjunto

$$\{wcw \mid w \text{ é uma cadeia de } a\text{'s e } b\text{'s}\}$$

não pode ser denotado por qualquer expressão regular, nem pode ser descrito por uma gramática livre de contexto.

As expressões regulares podem ser usadas para denotar somente um número fixo de repetições ou um número não especificado de repetições de uma dada construção. Dois números arbitrários não podem ser comparados a fim de se verificar se são os mesmos. Dessa forma não podemos descrever, através de uma expressão regular, cadeias Hollerith da forma $n\mathtt{H}a_1a_2 \ldots a_n$, das versões iniciais de Fortran, porque o número de caracteres seguintes ao `H` precisa se igualar ao número decimal antes de `H`.

3.4 O RECONHECIMENTO DE *TOKENS*

Na seção anterior, consideramos o problema de como especificar *tokens*. Nesta, endereçamos a questão de como reconhecê-los. Ao longo desta seção, usamos uma linguagem gerada pela gramática seguinte como um exemplo itinerante.

Exemplo 3.6. Consideremos o seguinte fragmento de gramática:

$$\begin{array}{lll}
cmd & \rightarrow & \textbf{if}\ expr\ \textbf{then}\ cmd \\
 & \mid & \textbf{if}\ expr\ \textbf{then}\ cmd\ \textbf{else}\ cmd \\
 & \mid & \epsilon \\
expr & \rightarrow & termo\ \textbf{relop}\ termo \\
 & \mid & termo \\
termo & \rightarrow & \textbf{id} \\
 & \mid & \textbf{num}
\end{array}$$

onde os terminais **if, then, else, relop, id** e **num** geram conjuntos de cadeias dados pelas seguintes definições regulares:

$$\begin{array}{rl}
\textbf{if} & \rightarrow \texttt{if} \\
\textbf{then} & \rightarrow \texttt{then} \\
\textbf{else} & \rightarrow \texttt{else} \\
\textbf{relop} & \rightarrow \texttt{<} \mid \texttt{<=} \mid \texttt{=} \mid \texttt{<>} \mid \texttt{>} \mid \texttt{>=} \\
\textbf{id} & \rightarrow \textbf{letra}\ (\ \textbf{letra} \mid \textbf{dígito}\ )^* \\
\textbf{num} & \rightarrow \textbf{dígito}^+\ (\ \cdot\ \textbf{dígito}^+\ )?\ (\ \texttt{E}(\ +\ \mid\ -\ )?\ \textbf{dígito}^+\ )?
\end{array}$$

onde **letra** e **dígito** são como definidos anteriormente.

Para esse fragmento de linguagem, o analisador léxico irá reconhecer as palavras-chave `if`, `then`, `else`, bem como os lexemas denotados por **relop, id** e **num**. Para simplificar as coisas, assumimos que as palavras-chave sejam reservadas; ou seja, não podem ser usadas como identificadores. Como no Exemplo 3.5, **num** representa o inteiro sem sinal e os números reais em Pascal.

Adicionalmente, assumimos que os lexemas sejam separados por espaços em branco, consistindo em seqüências não nulas de espaços, tabulações e avanços de linha. Nosso analisador léxico irá remover o espaço em branco. Tal tarefa será realizada confrontando-se uma cadeia com a definição regular **ws** (para *white space* — espaço em branco) abaixo:

$$\begin{array}{rl}
\textbf{delim} & \rightarrow \textbf{branco} \mid \textbf{tabulação} \mid \textbf{avanço de linha} \\
\textbf{ws} & \rightarrow \textbf{delim}^-
\end{array}$$

Se um reconhecimento para **ws** for atingido, o analisador léxico não retorna *token* algum para o *parser*. Ao invés, prossegue tentando encontrar um *token* que siga ao espaço em branco e o retorna ao *parser*.

EXPRESSÃO REGULAR	*TOKEN*	VALOR DE ATRIBUTO
ws	–	–
`if`	**if**	–
`then`	**then**	–
`else`	**else**	–
id	**id**	apontador para entrada da tabela
num	**num**	apontador para entrada da tabela
`<`	**relop**	`LT`
`<=`	**relop**	`LE`
`=`	**relop**	`EQ`
`<>`	**relop**	`NE`
`>`	**relop**	`GT`
`>=`	**relop**	`GE`

Fig. 3.10. Padrões de expressões regulares para *tokens*.

Nossa meta é a de construir um analisador léxico que irá isolar o lexema para o próximo *token* no *buffer* de entrada e produzir como saída um par consistindo no *token* apropriado e no valor de atributo, usando a tabela de tradução dada na Fig. 3.10. Os valores de atributo para os operadores relacionais são dados pelas constantes simbólicas `LT`, `LE`, `EQ`, `NE`, `GT`, `GE`. □

Diagramas de Transições

Como um passo intermediário na geração de um analisador léxico, produziremos primeiro um fluxograma estilizado, chamado de *diagrama de transições*. Os diagramas de transições delineiam as ações que tomam lugar quando um analisador léxico é chamado pelo *parser* a fim de obter o próximo *token*, como sugerido pela Fig. 3.1. Suponhamos que o *buffer* de entrada seja como na Fig. 3.3 e que o apontador de início de lexema indique o caractere que se segue ao último lexema encontrado. Usamos um diagrama de transições a fim de controlar as informações a respeito dos caracteres que são examinados na medida em que o apontador adiante esquadrinhe a entrada. Realizamos isso pela movimentação posição a posição no diagrama, à medida que os caracteres são lidos.

As posições num diagrama de transições são desenhadas como círculos e são chamadas de *estados*. Os estados são conectados por setas, chamadas de *lados*. Os lados que deixam o estado s possuem rótulos indicando os caracteres de entrada que podem aparecer após o diagrama de transições ter atingido o estado s. O rótulo **outro** se refere a qualquer caractere que não seja indicado por qualquer um dos lados que deixam s.

Assumimos que os diagramas de transições desta seção sejam *determinísticos*; isto é, o mesmo símbolo não pode figurar como rótulo de dois lados diferentes que deixem um mesmo estado. Começando pela Seção 3.5, relaxaremos esta condição, tornando muito mais simples a vida de um projetista de analisadores léxicos e, com as ferramentas adequadas, não mais difícil para o implementador.

Um estado é rotulado como o estado *de partida*; é o estado inicial do diagrama de transições, onde reside o controle quando iniciamos o reconhecimento de um *token*. Certos estados podem ter ações

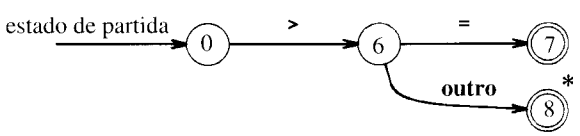


Fig. 3.11. Diagrama de transições para >=.

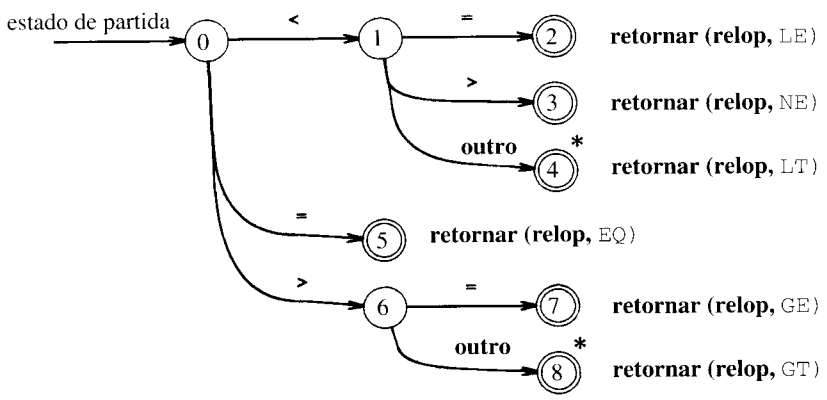


Fig. 3.12. Diagrama de transições para operadores relacionais.

que são executadas quando atingidos pelo fluxo de controle. Ao entrar num estado, lemos o próximo caractere de entrada. Se existir um lado a partir do estado corrente cujo rótulo se iguale a esse caractere de entrada, nos dirigimos, então, para o estado apontado pelo lado. De outra forma, indicamos uma falha.

A Figura 3.11 mostra um diagrama de transições para os padrões >= e >. O diagrama funciona como se segue: seu estado inicial é o estado 0. No mesmo, lemos o próximo caractere de entrada. O lado rotulado > a partir do estado 0 deve ser seguido até o estado 6, se esse caractere de entrada for >. De outra forma, falhamos em reconhecer > ou >=.

Ao atingirmos o estado 6, lemos o próximo caractere de entrada. O lado rotulado = a partir do estado 6 deve ser seguido até o estado 7, se o caractere de entrada for um =. De outra forma, o lado rotulado **outro** indica que devemos nos dirigir ao estado 8. O círculo duplo no estado 7 informa que o mesmo é um estado de aceitação, no qual o *token* >= foi encontrado.

Note-se que o caractere > e um outro caractere extra são lidos, na medida em que sigamos a seqüência de lados a partir do estado de partida até o estado de aceitação 8. Como o caractere extra não é parte do operador relacional >, precisamos retrair o apontador adiante em exatamente um caractere, nesse caso. Usamos um * para indicar estados nos quais essa ação de retração da entrada precise tomar lugar.

Em geral, pode haver vários diagramas de transições, cada qual especificando um grupo de *tokens*. Se ocorrer uma falha enquanto estivermos seguindo um diagrama de transições, retraímos o apontador adiante para onde o mesmo estava quando no estado de partida do diagrama e ativamos o próximo diagrama de transições. Como os apontadores de início de lexema e adiante marcavam a mesma posição no estado de partida do diagrama, o apontador adiante é retraído à posição marcada pelo apontador de início de lexema. Se uma falha ocorrer em todos os diagramas de transições, um erro léxico foi detectado e invocamos uma rotina de recuperação de erros.

Exemplo 3.7. Um diagrama de transições para o *token* **relop** é mostrado à Fig. 3.12. Note-se que a Fig. 3.11 é uma parte desse diagrama de transições mais complexo. □

Exemplo 3.8. Como as palavras-chave são seqüências de letras, constituem-se em exceções à regra que estabelece que uma seqüência de letras e dígitos iniciada por uma letra seja um identificador. Ao invés de codificar as exceções num diagrama de transições, uma saída engenhosa é a de tratar palavras-chave como identificadores especiais, tal como na Seção 2.7. Quando o estado de aceitação na Fig. 3.13 é atingido, executamos algum código para determinar se o lexema que leva ao estado de aceitação é uma palavra-chave ou um identificador.

Uma técnica simples para separar as palavras-chave dos identificadores é a de inicializar apropriadamente a tabela de símbolos, onde as informações sobre os identificadores são guardadas. Para os *tokens* da Fig. 3.10, precisamos dar entrada às cadeias `if`, `then` e `else` na tabela de símbolos, antes que quaisquer caracteres da entrada sejam examinados. Também fazemos uma anotação na (entrada da) tabela de símbolos a respeito do *token* a ser retornado, quando uma dessas cadeias for reconhecida. O enunciado de retorno em seguida ao estado de aceitação na Fig. 3.13 usa *obter_token* () e *instalar_id*(), respectivamente, a fim de obter o *token* e o valor de atributo a serem retornados. O procedimento *instalar_id*() tem acesso ao *buffer* onde o lexema do identificador foi localizado. A tabela de símbolos é examinada e, se o lexema for encontrado lá, marcado como uma palavra-chave, *instalar_id*() retorna 0. Se o lexema for encontrado na tabela de símbolos como uma variável de programa, *instalar_id*() retorna um apontador para a entrada da tabela de símbolos. Se o lexema não for encontrado na tabela de símbolos, o mesmo é instalado como uma variável de programa e um apontador para a entrada recém-criada é retornado.

O procedimento *obter_token*() similarmente procura pelo lexema na tabela de símbolos. Se o lexema for uma palavra-chave, o *token* correspondente é retornado; de outra feita, o *token* **id** é retornado.

Note-se que o diagrama de transições não muda se palavras-chave adicionais precisarem ser reconhecidas; simplesmente inicializamos a tabela de símbolos com as cadeias e *tokens* das palavras-chave adicionais. □

A técnica de se colocar palavras-chave na tabela de símbolos é quase que essencial se o analisador léxico tiver que ser codificado à mão. Sem realizá-lo, o número de estados de um analisador léxico para uma típica linguagem de programação é de várias centenas, enquanto que, usando-se esta saída, menos do que uma centena de estados provavelmente serão suficientes.

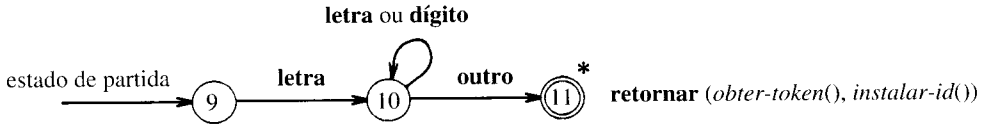


Fig. 3.13. Diagrama de transições para identificadores e palavras-chave.

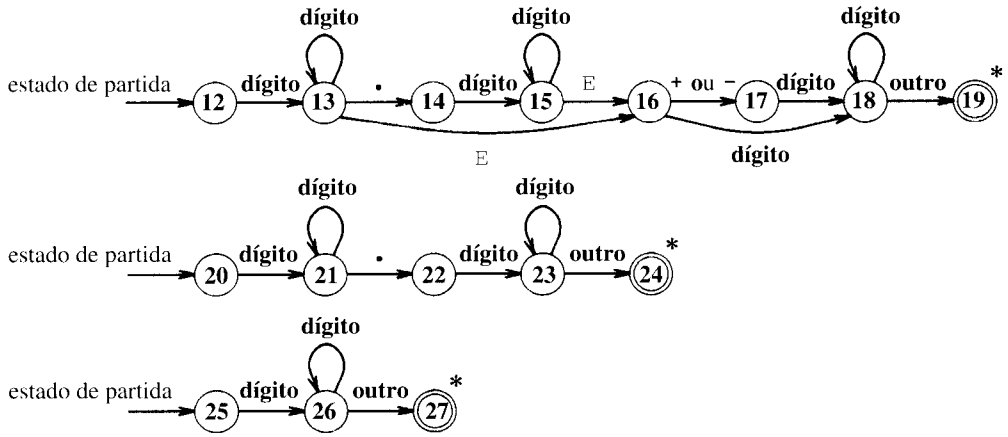


Fig. 3.14. Diagramas de transições para números sem sinal em Pascal.

Exemplo 3.9. Vários temas emergem ao construirmos um reconhecedor para números sem sinal, dados pela seguinte definição regular

num → **dígito**$^+$ (· **dígito**$^+$)? (E(+ | −)? **dígito**$^+$)?

Note-se que a definição é da forma **dígitos fração**? **expoente**?, na qual **fração** e **expoente** são opcionais.

O lexema para um dado *token* precisa ser o mais longo possível. Por exemplo, o analisador léxico não pode parar após enxergar `12` ou mesmo `12.3`, quando a entrada for `12.3E4`. Começando-se pelos estados 25, 20 e 12 na Fig. 3.14, os estados de aceitação serão atingidos após `12`, `12.3` e `12.3E4` serem respectivamente examinados, se `12.3E4` for seguido por um não-dígito à entrada. Os diagramas de transições com os estados de partida 25, 20 e 12 são para **dígitos, dígitos fração** e **dígitos fração**? **expoente**, respectivamente, e, conseqüentemente, os estados de partida precisam ser testados na ordem reversa, 12, 20 e 25.

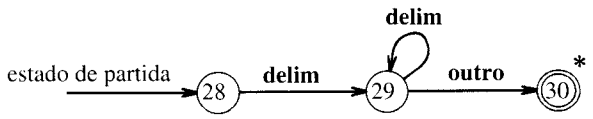
A ação, quando qualquer dos estados de aceitação 19, 24 ou 27 for atingido, é chamar um procedimento *instalar_num* que dá entrada do lexema na tabela de números e retorna um apontador para a entrada criada. O analisador léxico retorna o *token* **num** juntamente com esse apontador como o valor léxico. □

As informações sobre a linguagem que não estejam nas definições regulares dos *tokens* podem ser usadas para assinalar erros na entrada. Por exemplo, com a entrada `1.<x`, falhamos nos estados 14 e 22 da Fig. 3.14, com o próximo caractere de entrada <. Ao invés de retornarmos o número `1`, podemos querer relatar um erro e continuar como se a entrada fosse `1.0 < x`. Tal conhecimento pode também ser usado para simplificar diagramas de transições, porque o tratamento de erros pode ser usado para se recuperar de algumas situações que de outra forma levariam a uma falha.

Existem várias formas pelas quais o reconhecimento redundante nos diagramas de transições da Fig. 3.14 pode ser evitado. Um enfoque é o de reescrever os diagramas combinando-os num único, uma tarefa não trivial em geral. Outro é o de mudar a resposta no caso de uma falha no processo de se acompanhar o diagrama. Uma abordagem explorada mais tarde neste capítulo nos permite passar através de vários estados de aceitação; retornamos para o último estado de aceitação que passarmos antes de a falha ocorrer.

Exemplo 3.10. Uma seqüência de diagramas de transições para todos os *tokens* do Exemplo 3.6 é obtida se pusermos juntos os diagramas de transições das Figs. 3.12, 3.13 e 3.14. Estados de partida com numeração mais baixa devem ser tentados antes dos estados com numeração mais alta.

Os únicos temas remanescentes se relacionam aos espaços em branco. O tratamento de **ws,** representando espaço em branco, é diferente daquele dos padrões discutidos acima, porque nada é retornado ao *parser* quando o mesmo é encontrado na entrada. Um diagrama de transições para reconhecer **ws** por si só é



Nada é retornado quando o estado de aceitação é atingido; meramente retornamos para o estado de partida do primeiro diagrama de transições a fim de procurar por outro padrão.

Sempre que possível, é melhor procurar por *tokens* que incidem freqüentemente antes daqueles que ocorrem menos freqüentemente, porque um diagrama de transições é atingido somente após termos falhado nos diagramas anteriores. Como o espaço em branco é esperado ocorrer freqüentemente, a colocação do diagrama de transições para espaço em branco próximo ao início deve demonstrar ser um melhoramento em relação à colocação ao final. □

Implementando um Diagrama de Transições

Uma seqüência de diagramas de transições pode ser convertida num programa que procure pelos *tokens* especificados pelos diagramas. Adotamos um enfoque sistemático que funciona para todos os diagramas de transições e constrói programas cujo tamanho seja proporcional ao número de estados e lados dos diagramas.

Cada estado recebe um segmento de código. Se existirem lados deixando um estado, então seu código lê um caractere e seleciona um lado para seguir, se possível. A função `próximo-caractere()` é usada para ler o próximo caractere a partir do *buffer* de entrada, avançar o apontador adiante a cada chamada e retornar o caractere lido.[3] Se existir um lado rotulado pelo caractere lido, ou por uma classe de caracteres que o contenha, o controle é, então, transferido para o código do estado apontado por aquele lado. Se não existir tal lado, o estado corrente não é um daqueles que indica que um *token* foi encontrado, e, nesse caso, a rotina `falhar()` é invocada para retrair o apontador adiante para a posição de

[3]Uma implementação mais eficiente usaria uma macro em linha no lugar da função `próximo-caractere()`.

```
int estado = 0, partida = 0;
int valor_léxico;
    /* para "retornar" o segundo componente
de um token */
int falhar ()
{
    apontador_adiante = início_de_token;
    switch (partida) {
        case 0:   partida = 9; break;
        case 9:   partida = 12; break;
        case 12:  partida = 20; break;
        case 20:  partida = 25; break;
        case 25:  recuperar(); break;
        default:  /* erro do compilador */
    }
    retornar partida;
}
```

Fig. 3.15. Código C para se descobrir o próximo estado de partida.

início[1] e começar nova busca por um *token*, especificada pelo próximo diagrama de transições. Se não existirem outros diagramas de transições a tentar, `falhar()` chama uma rotina de recuperação de erros.

Para retornar *tokens*, usamos uma variável global `valor_léxico`, à qual são atribuídos os apontadores retornados pelas funções `instala_id()` e `instala_num()`, quando for encontrado, respectivamente, um identificador ou um número. A classe do *token* é retornada pelo procedimento principal do analisador léxico, chamado `próximo_token()`.

Usamos um enunciado *case* para descobrir o estado de partida do próximo diagrama de transições. Na implementação C da Fig. 3.15, duas variáveis, `estado` e `partida` cuidam do estado presente e do estado de partida do diagrama de transições corrente. Os números de estado no código são para os diagramas de transições das Figs. 3.12 a 3.14.

Os lados nos diagramas de transições podem ser acompanhados selecionando-se repetidamente o fragmento de código para um estado e executando-se o mesmo para determinar o próximo estado, como mostrado na Fig. 3.16. Mostramos o código para o estado 0, tal como modificado no Exemplo 3.10, de forma a tratar os espaços em branco, e o código para dois dos diagramas de transições das Figs. 3.13 e 3.14. Note-se que a construção C

`while(1)` *cmd*

repete *cmd* "para sempre", isto é, até que ocorra um `retornar`.

Como C não permite que sejam retornados um *token* e um valor de atributo, `instalar_id()` e `instalar_num()` estabelecem apropriadamente alguma variável global com o correspondente valor de atributo da entrada da tabela para o **id** ou **num** em questão.

Se a linguagem de implementação não possui um enunciado *case*, podemos criar um *array* para cada estado, indexado por caracteres. Se *estado* 1 é um tal *array*, então *estado* 1 [*c*] é um apontador para um trecho de código que precisa ser executado sempre que o caractere *lookahead* for *c*. Esse código terminaria normalmente com um desvio para o código do próximo estado. O *array* para o estado *s* é referenciado como a tabela indireta de transferências para *s*.

3.5 UMA LINGUAGEM PARA ESPECIFICAÇÃO DE ANALISADORES LÉXICOS

Várias ferramentas têm sido construídas para os analisadores léxicos a partir de notações de propósito especial baseadas nas expressões regulares. Já vimos o uso das expressões regulares para especificar os padrões dos *tokens*. Antes de considerarmos os algoritmos para compilar expressões regulares em programas de reconhecimento de padrões, daremos um exemplo de uma ferramenta que poderia usar um tal algoritmo.

Nesta seção, descrevemos uma ferramenta particular, chamada Lex, que tem sido amplamente usada para especificar analisadores léxicos para uma variedade de linguagens. Referimo-nos à ferramenta como o *Compilador Lex*, e sua especificação de entrada como a *Linguagem Lex*. A discussão de uma ferramenta existente nos permite mostrar como a especificação de padrões usando-se expressões regulares pode ser combinada com ações, como por exemplo realizando entradas numa tabela de símbolos, coisa que a um analisador léxico pode ser exigido realizar. Especificações ao estilo Lex podem ser usadas mesmo que um compilador não esteja disponível; as especificações podem ser manualmente transcritas num programa usando as técnicas dos diagramas de transições da seção anterior.

Lex é geralmente usado da forma delineada na Fig. 3.17. Primeiro, a especificação de um analisador léxico é preparada criando-se o programa (fonte) `lex.l`, na linguagem Lex. Em seguida, `lex.l` é processado pelo compilador Lex a fim de produzir um programa C,

```
token próximo_token()
{   while(1)    {
        switch (estado) {
        case 0: c = próximo_caractere();
            /* c é o caractere lookahead */
            if (c == espaço || c == tabulação ||
                c = avanço de linha){
                estado = 0,
                início_de_lexema++;
                    /* avançar início de
                    lexema */
            }
            else if (c = = '<') estado = 1;
            else if (c = = '=') estado = 5;
            else if (c = = '>') estado = 6;
            else estado = falhar();
            break;
            . . . /* inserir código para
            casos 1-8 aqui */
            break;
        case 9: c = próximo_caractere();
            if (isletter (c)) estado = 10;
            else estado = falhar();
            break;
        case 10: c = próximo_caractere();
            if (isletter (c)) estado = 10;
            else if (isdigit (c) ) state = 10;
            else estado = 11;
            break;
        case 11: retrair(1); instalar_id();
            return ( obter_token() );
            . . . /* inserir casos 12-24
            aqui*/
        case 25: c = próximo_caractere();
            if (isdigit (c)) estado = 26;
            else estado = falhar();
            break;
        case 26: c = próximo_caractere();
            if (isdigit (c)) estado = 26;
            else estado = 27;
            break;
        case 27; retrair(1); instalar_num();
            return ( NUM );
        }
    }
}
```

Fig. 3.16. Código C para o analisador léxico

[1] Isto é, a posição no *buffer* onde começou a busca por um *token* no diagrama de transições atual. (N. do T.)

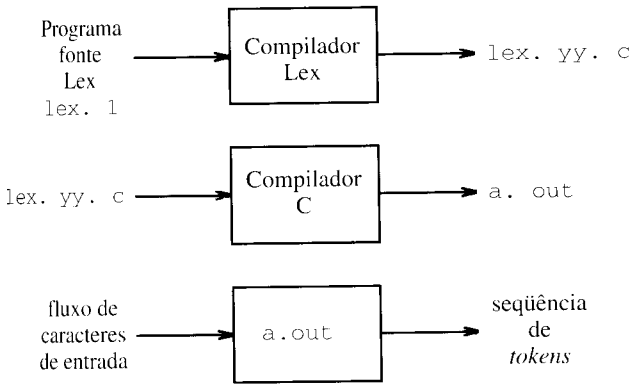


Fig. 3.17. Criando um analisador léxico com Lex.

`lex.yy.c`. O programa `lex.yy.c` consiste em uma representação tabular de um diagrama de transições construído a partir das expressões regulares de `lex.l`, juntamente com uma rotina-padrão que usa a tabela a fim de reconhecer os lexemas. As ações associadas às expressões regulares em `lex.l` são trechos de código escritos em C que são carregados diretamente em `lex.yy.c`. Finalmente, `lex.yy.c` é processado por um compilador C a fim de produzir um programa objeto `a.out`, o qual é o analisador léxico que transforma um fluxo de caracteres de entrada numa seqüência de *tokens*.

Especificações Lex

Um programa *Lex* é constituído de três partes:

declarações
%%
regras de tradução
%%
procedimentos auxiliares

A seção de declarações inclui declarações de variáveis, de constantes manifestas e de definições regulares. (Uma constante manifesta é um identificador que é declarado representar uma constante.) As definições regulares são enunciados similares àqueles fornecidos à Seção 3.3 e são usadas como componentes das expressões regulares que aparecem nas regras de tradução.

As regras de tradução de um programa Lex são enunciados da forma

p_1 {*ação*$_1$}
p_2 {*ação*$_2$}
... ...
p_n {*ação*$_n$}

onde cada p_i é uma expressão regular e cada *ação*$_i$ é um fragmento de programa descrevendo que ação o analisador léxico deverá tomar quando o padrão p_i reconhecer um lexema. Em Lex, as ações são escritas em C; em geral podem, no entanto, estar em qualquer linguagem de implementação.

Uma terceira seção contém quaisquer procedimentos auxiliares que sejam necessitados pelas ações. Alternativamente, esses procedimentos podem ser compilados separadamente e carregados com o analisador léxico.

Um analisador léxico criado por Lex se comporta em concerto com um *parser* da seguinte forma. Quando ativado pelo *parser*, o analisador léxico começa lendo sua entrada remanescente, um caractere de cada vez, até que tenha encontrado o mais longo prefixo da entrada que seja reconhecido por uma das expressões regulares p_i. Em seguida, executa *ação*$_i$. Tipicamente, *ação*$_i$ irá retornar o controle para o *parser*. Entretanto, se não o fizer, o analisador léxico prossegue a fim de encontrar mais lexemas até que uma ação cause o retorno do controle ao *parser*. A procura repetida por lexemas até um que ocorra um retorno explícito permite que o analisador léxico processe o espaço em branco e os comentários convenientemente.

O analisador léxico retorna uma única quantidade, o *token*, para o *parser*. A fim de passar um valor de atributo com informações sobre o lexema, podemos atribuir valores à variável global chamada `yylval`.

Exemplo 3.11. A Fig. 3.18 é um programa Lex que reconhece os *tokens* da Fig. 3.10 e retorna o *token* encontrado. Umas poucas observações a respeito do código irão nos introduzir a muitas das figurações importantes de Lex.

Na seção de declarações, vemos (um lugar para) a declaração de certas constantes manifestas usadas pelas regras de tradução.[4] Essas declarações são envolvidas por chaves especiais `%{` e `%}`. Qualquer coisa figurando entre essas chaves é copiada diretamente no analisador léxico `lex.yy.c` e não é tratada como parte das definições regulares ou das regras de tradução. Exatamente o mesmo tratamento é estabelecido para os procedimentos auxiliares na terceira seção. Na Fig. 3.18, existem dois procedimentos, `instalar_id` e `instalar_num` que são usados pelas regras de tradução; esses procedimentos serão copiados em `lex.yy.c` *ipsis litteris*.

Também incluídas na seção de definições estão algumas definições regulares. Cada tal definição consiste em um nome e em uma expressão regular denotada por aquele nome. Por exemplo, o primeiro nome definido é `delim`; figura no lugar da classe de caracteres `[ \t\n]`, isto é, qualquer dos três símbolos, espaço, tabulação (representada por `\t`) ou avanço de linha (representado por `\n`). A segunda definição é a do espaço em branco, denotada pelo nome `ws`. Espaço em branco é qualquer seqüência de um ou mais caracteres delimitadores. Note-se que a palavra `delim` precisa ser envolvida por chaves em Lex, a fim de distingui-la do padrão constituído pelas cinco letras `delim`.

Na definição de `letra`, vemos o uso de uma classe de caracteres. A forma simplificada `[A-Za-z]` significa qualquer uma das letras maiúsculas, de `A` até `Z`, ou as letras minúsculas, de `a` até `z`. A quinta definição de `id` usa parênteses, que são metassímbolos em Lex, com seus significados naturais de agrupadores. Similarmente, a barra vertical é um metassímbolo em Lex que representa a união.

Na última definição regular, a de `número`, observamos uns poucos detalhes a mais. Vemos ? usado como um metassímbolo, com seu significado usual de "zero ou uma ocorrência de". Também notamos a barra invertida usada como um caractere de escape, a fim de deixar um caractere que é um metassímbolo Lex ter seu significado natural. Em particular, o ponto decimal na definição de `número` é expresso por `\.` porque um ponto por si só representa a classe de todos os caracteres exceto o avanço de linha, tanto em Lex quanto em muitos outros programas de sistema UNIX que lidam com expressões regulares. Na classe de caracteres `[+\-]`, colocamos uma barra invertida antes do sinal de menos porque o mesmo figurando isolado poderia ser confundido com seu uso para denotar um intervalo, como em `[A-Z]`.[5]

Existe uma outra forma de se fazer com que os caracteres tenham seus significados naturais, ainda que sejam metassímbolos de Lex: envolvê-los entre aspas. Mostramos um exemplo dessa convenção na seção de regras de tradução, onde seis operadores relacionais foram envolvidos em aspas.[6]

---

[4]É comum para o programa `lex.yy.c` ser usado como uma sub-rotina de um *parser* gerado por Yacc, um gerador de *parsers* discutido no Capítulo 4. Nesse caso, a declaração de constantes manifestas seria providenciada pelo *parser*, quando fosse compilado com o programa `lex.yy.c`.

[5]De fato, Lex trata a classe de caracteres `[+-]` corretamente sem a barra invertida, porque o sinal de menos figurando ao fim não pode significar um intervalo.

[6]Assim o fizemos porque <e> são metassímbolos de Lex; envolvem os nomes de "estados", habilitando Lex a mudar de estado ao encontrar certos *tokens*, tais como comentários e cadeias entre aspas, que podem ser tratados diferentemente do texto usual. Não existe necessidade de se envolver um sinal de igual entre aspas, mas também não é proibido.

```
%{
      /* definições das constantes manifestas
      LT, LE, EQ, NE, GT, GE,
      IF, THEN, ELSE, ID, NÚMERO, RELOP */
%}

/* definições regulares */
delim       [ \t\n]
ws          {delim}+
letra       [A-Za-z]
dígito      [0-9]
id          {letra} ({letra} | {dígito})*
número      {dígito} + (\. {dígito}+)? (E[+\-]? {dígito}+)?

%%

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

%%

instalar_id () {

      /* procedimento para instalar o lexema, cujo primeiro caractere
         é apontado por yytext e cujo comprimento é yyleng, na
         tabela de símbolos e retornar um apontador para o mesmo */
}

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

Fig. 3.18. Programa Lex para os *tokens* da Fig. 3.10.

Agora, vamos considerar as regras de tradução na seção que se segue ao primeiro %%. A primeira regra diz que se enxergarmos `ws`, isto é, qualquer seqüência maximal de espaços, tabulações e avanços de linha, não realizamos qualquer ação. Em particular, não retornamos ao *parser*. Relembremos que a estrutura do analisador léxico é tal que o mesmo se mantém tentando reconhecer *tokens*, até que uma ação associada a um *token* encontrado provoque o retorno.

A segunda regra diz que se as letras `if` forem vistas, deve-se retornar o *token* `IF`, que é uma constante manifesta representando algum inteiro compreendido pelo *parser* como sendo o *token* **if.** As duas regras seguintes tratam das palavras-chaves `then` e `else` similarmente.

Na regra para **id,** vemos dois enunciados na ação associada. Primeiro, a variável `yylval` é estabelecida com o valor retornado pelo procedimento `instalar_id`; a definição daquele procedimento está na terceira seção. `yylval` é uma variável cuja definição aparece na saída de Lex `lex.yy.c` e que também está disponível para o *parser*. O propósito de `yylval` é o de guardar o valor léxico retornado, uma vez que o segundo enunciado da ação, `return (ID)`, pode somente retornar um código para a classe do *token*.

Não mostraremos os detalhes do código de `instalar_id`. Entretanto, podemos supor que o mesmo procure na tabela de símbolos pelo *lexema* reconhecido pelo padrão **id.** Lex torna o lexema disponível para as rotinas que figuram na terceira seção através das duas variáveis `yytext` e `yyleng`. A variável `yytext` corresponde à variável que temos chamado de *início_de_lexema*, ou seja, um apontador para o primeiro caractere do lexema. Por exemplo, se `instalar_id` falhar em encontrar o identificador na tabela de símbolos, poderíamos criar uma nova entrada para o mesmo. Os `yyleng` caracteres da entrada, começando em `yytext`, poderiam ser copiados num *array* de caracteres e delimitados por um marcador de fim de cadeia, como na Seção 2.7. A nova entrada da tabela de símbolos apontaria para o início dessa cópia.

Os números são tratados similarmente pela regra seguinte, e, para as últimas seis regras, `yylval` é usada para retornar um código para o operador relacional particular encontrado, enquanto o valor efetivo de retorno é o código para o *token* **relop** em cada caso.

Suponhamos que ao analisador léxico resultante do programa da Fig. 3.18 seja fornecida uma entrada constituída de duas tabulações, as letras `if` e um espaço. As duas tabulações são o mais longo prefixo inicial da entrada reconhecido por um padrão, explicitamente o padrão `ws`. A ação para `ws` é a de não realizar coisa alguma e, então, o analisador léxico move o apontador de início de lexema, `yytext`, para o `i` e começa a procurar por outro *token*.

O próximo lexema a ser reconhecido é `if`. Note-se que os padrões `if` e `{id}` reconhecem esse lexema e nenhum dos dois reconhece uma cadeia mais longa. Como o padrão para a palavra-chave `if` precede o padrão para os identificadores na lista da Fig. 3.18, o confli-

to é resolvido em favor da palavra-chave. Em geral, esta estratégia de resolução de conflitos torna fácil reservar palavras-chave listando-as à frente dos padrões para os identificadores.

Como um outro exemplo, suponhamos que <= sejam os dois primeiros caracteres lidos. Enquanto o padrão < reconhece o primeiro caractere, o mesmo não é o padrão mais longo que reconhece um prefixo da entrada. Conseqüentemente, a estratégia de Lex em selecionar o prefixo mais longo reconhecido por um padrão facilita a resolução do conflito entre < e <= da forma esperada – pela seleção de <= como o próximo *token*. □

O Operador *Lookahead*

Como vimos na Seção 3.1, os analisadores léxicos para certas construções de linguagem de programação precisam examinar além do fim do lexema antes que um *token* possa ser determinado sem sombra de dúvida. Relembremos o exemplo da Fortran a respeito do par de enunciados

```
DO 5 I = 1.25
DO 5 I = 1,25
```

Como em Fortran os espaços não são significativos fora dos comentários e das cadeias Hollerith, suponhamos, então, que todos os espaços removíveis sejam estirpados antes que a análise léxica comece. Os enunciados acima apareceriam para o analisador léxico como

```
DO5I=1.25
DO5I=1,25
```

No primeiro enunciado, não podemos dizer, até que tenhamos examinado o ponto decimal, que a cadeia `DO` seja parte do identificador `DO5I`. No segundo enunciado, `DO` é uma palavra-chave por si mesma.

Em Lex, podemos escrever um padrão da forma r_1/r_2, onde r_1 e r_2 sejam expressões regulares, significando reconhecer uma cadeia em r_1 somente se seguida por uma cadeia reconhecida em r_2. A expressão regular r_2, após o operador *lookahead* /, indica que contexto à direita deve ser usado no reconhecimento; mas esse contexto deve ser usado apenas para restringir o reconhecimento, não para fazer parte do mesmo. Por exemplo, uma especificação Lex que reconheça a palavra-chave `DO` no contexto acima é

```
DO/({letra} | {dígito})* = ({letra} | {dígito})*,
```

Com essa especificação, o analisador léxico irá procurar à frente em seu *buffer* de entrada por uma seqüência de letras e dígitos seguida por um sinal de igual, seguida por letras e dígitos e seguida por uma vírgula, a fim de assegurar que não haja um enunciado de atribuição (e sim o comando repetitivo `DO`). Então, somente os caracteres `D` e `O`, precedendo o operador *lookahead* /, seriam parte do lexema reconhecido. Após um reconhecimento com sucesso, `yytext` aponta para o `D` e `yyleng` = 2. Note-se que esse simples padrão de esquadrinhamento adiante permite que `DO` seja reconhecido quando seguido por lixo, como `Z4=6Q`, mas jamais irá reconhecer o `DO` que seja parte de um identificador.

Exemplo 3.12. O operador *lookahead* pode ser usado para colaborar em outro difícil problema de análise léxica em Fortran: distinguir palavras-chave e identificadores. Por exemplo, a entrada

```
IF(I, J) = 3
```

é um enunciado de atribuição perfeitamente válido em Fortran e não um enunciado lógico `IF`. Uma forma de especificar a palavra-chave `IF` em Lex é definir seus possíveis contextos à direita usando o operador *lookahead*. A forma simples do enunciado IF lógico é

```
IF(condição) enunciado
```

Fortran 77 introduziu uma outra forma de enunciado IF lógico:

```
IF(condição) THEN
   bloco_then
ELSE
   bloco_else
END IF
```

Notamos que cada enunciado Fortran sem rótulo se inicia por uma letra e que cada parênteses à direita usado para a subscrição ou agrupamento de operandos precisa ser seguido por um símbolo de operação, tal como =, + ou vírgula, outro parênteses à direita ou o fim do enunciado. Um tal parênteses à direita não pode ser seguido por uma letra. Nesta situação, a fim de confirmar que `IF` é uma palavra-chave ao invés de um nome de *array*, esquadrinhamos à frente procurando por um parênteses à direita seguido por uma letra antes de vermos um caractere de avanço de linha (assumimos que os cartões de continuação "cancelam" o caractere de avanço de linha prévio). Esse padrão para a palavra-chave `IF` pode ser escrito como

```
IF/ \(.* \) {letra}
```

O ponto figura no lugar de "qualquer caractere menos avanço de linha" e as barras invertidas à frente dos parênteses informam ao compilador Lex para tratar esses últimos literalmente, não como metassímbolos, ao agrupar expressões regulares (ver Exercício 3.10). □

Outra forma de atacar o problema imposto pelos enunciados `IF` em Fortran é, após enxergar `IF`, determinar se o mesmo foi declarado como um *array*. Esquadrinhamos o padrão completo indicado acima somente se assim o tiver sido declarado. Tais testes tornam mais difícil a implementação automática de um analisador léxico a partir de uma especificação Lex e podem custar tempo numa perspectiva mais ampla, pois verificações freqüentes precisam ser feitas no programa que simula um diagrama de transições, a fim de determinar se algum de tais testes é necessário. Deveria ser notado que tokenizar Fortran é uma tarefa tão irregular que freqüentemente é mais fácil se escrever um analisador léxico *ad hoc* para Fortran numa linguagem convencional de programação do que usar um gerador automático de analisadores léxicos.

3.6 AUTÔMATOS FINITOS

Um *reconhecedor* para uma linguagem é um programa que toma como entrada uma cadeia x e responde "sim" se x for uma sentença da linguagem e "não" em caso contrário. Compilamos expressões regulares num reconhecedor através da construção de um diagrama de transições generalizado chamado de autômato finito. Um autômato finito pode ser determinístico ou não-determinístico, onde "não-determinístico" significa que mais de uma transição para fora de um estado pode ser possível para o mesmo símbolo de entrada.

Tanto os autômatos finitos determinísticos quanto os não-determinísticos são capazes de reconhecer precisamente os conjuntos regulares. Conseqüentemente, ambos podem reconhecer exatamente o que as expressões regulares podem denotar. Entretanto, existe uma barganha tempo-espaço: enquanto os autômatos finitos determinísticos podem levar a reconhecedores mais rápidos do que os não-determinísticos, um autômato finito-determinístico pode ser muito maior do que um autômato finito não determinístico equivalente. Na próxima seção, apresentamos métodos para converter expressões regulares em ambos os tipos de autômatos finitos. A conversão num autômato finito não determinístico é mais direta e, então, discutiremos esses autômatos primeiro.

Os exemplos desta seção e da próxima lidam primariamente com a linguagem denotada pela expressão regular $(a \mid b)^*abb$, consistindo no conjunto de todas as cadeias de a's e b's terminadas em abb. Linguagens similares emergem na prática. Por exemplo, uma expressão

regular para os nomes de todos os arquivos que terminem em .o é da forma (. | o | *c*)*.o, com *c* representando qualquer caractere que não um ponto ou um . o. Como outro exemplo, os comentários em C consistem em qualquer seqüência de caracteres começada por /* e terminada por */, com a exigência adicional de que nenhum prefixo próprio termine em */.

Autômatos Finitos Não-Determinísticos

Um *autômato finito não-determinístico* (AFN, simplificadamente) é um modelo matemático que consiste em

1. um conjunto de *estados S*
2. um conjunto de símbolos de entrada Σ (o *alfabeto de símbolos de entrada*)
3. uma função de transição, *movimento*, que mapeia pares estado-símbolo em conjuntos de estados
4. um estado s_0 que é distinguido como o *estado de partida* (ou *inicial*).
5. um conjunto de estados *F* distinguidos como *estados de aceitação* (ou *finais*).

Um AFN pode ser representado diagramaticamente por um grafo dirigido e rotulado, chamado de *grafo de transições*, no qual os nós são os estados e os lados rotulados representam a função de transição. Esse grafo se parece com um diagrama de transições, mas o mesmo caractere pode rotular duas ou mais transições para fora de um mesmo estado e os lados podem ser rotulados pelo símbolo especial ϵ bem como pelos símbolos de entrada.

Um grafo de transições para um AFN que reconheça a linguagem $(a \mid b)^*abb$ é mostrado na Fig. 3.19. O conjunto de estados do AFN é $\{0, 1, 2, 3\}$ e o alfabeto de símbolos de entrada é $\{a, b\}$. O estado 0 na Fig. 3.19 é distinguido como o estado de partida e o estado de aceitação 3 é indicado por um círculo duplo.

Ao descrever um AFN, usamos a representação de grafos de transições. Como veremos, a função de transição de um AFN pode ser implementada de várias formas diferentes num computador. A implementação mais fácil é a *tabela de transições*, na qual existe uma linha para cada estado e uma coluna para cada símbolo de entrada e para ε se necessário. A entrada para a linha *i* e símbolo *a* na tabela é o conjunto de estados (ou mais provavelmente na prática, um apontador para um conjunto de estados) que podem ser atingidos através de uma transição a partir do estado *i* e entrada *a*. A tabela de transições para o AFN da Fig. 3.19 é mostrada na Fig. 3.20.

A representação sob a forma de tabela de transições possui a vantagem de providenciar acesso rápido às transições de um dado estado e caractere; sua desvantagem é que pode ocupar um grande espaço quando o alfabeto de entrada for grande e a maioria das transições for para o conjunto vazio. Representações da função de transição sob a forma de listas de adjacências de transição providenciam implementações mais compactas, mas o acesso a uma dada transição é mais lento. Deveria ficar claro que podemos facilmente converter qualquer uma dessas implementações de um autômato finito para outra.

Um AFN *aceita* uma cadeia de entrada *x* se e somente se existir algum percurso no grafo de transições, a partir do estado inicial até algum estado de aceitação, tal que os rótulos dos lados ao longo do

Estado	Símbolo de Entrada	
	a	*b*
0	{0,1}	{0}
1	—	{2}
2	—	{3}

Fig. 3.20. Tabela de transições para o autômato finito da Fig. 3.19.

percurso soletrem a cadeia *x*. O AFN da Fig. 3.19 aceita as cadeias de entrada *abb, aabb, babb, aaabb, ...* . Por exemplo, *aabb* é aceita pelo percurso a partir de 0, seguindo o lado rotulado *a* até o estado 0 de novo e, então, para os estados 1, 2 e 3 através dos lados rotulados *a, b* e *b*, respectivamente.

Um percurso pode ser representado por uma seqüência de transições de estado chamadas *movimentos*. O seguinte diagrama mostra os movimentos realizados ao se aceitar a cadeia de entrada *aabb:*

$$0 \xrightarrow{a} 0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{b} 3$$

Em geral, mais de uma seqüência de movimentos pode levar a um estado de aceitação. Note-se que várias outras seqüências de movimentos podem ser realizadas para a cadeia de entrada *aabb*, mas a nenhuma dessas outras ocorre terminar num estado de aceitação. Por exemplo, uma outra seqüência de movimentos para a entrada *aabb* se mantém reentrando no estado não final 0:

$$0 \xrightarrow{a} 0 \xrightarrow{a} 0 \xrightarrow{b} 0 \xrightarrow{b} 0$$

A *linguagem definida por* um AFN é o conjunto de cadeias de entrada que o mesmo aceita. Não é difícil mostrar que o AFN da Fig. 3.19 aceita $(a \mid b)^*abb$.

Exemplo 3.13. Na Fig. 3.21, vemos um AFN para reconhecer aa^* bb^*. A cadeia *aa* é aceita através da movimentação dos estados 0, 1, 2, 2 e 2. Os rótulos desses lados são ϵ, *a*, *a* e *a*, cuja concatenação é *aaaa*. Note-se que os ϵ's "desaparecem" na concatenação. □

Autômatos Finitos Determinísticos

Um *autômato finito determinístico* (AFD, simplificadamente) é um caso especial de autômato finito não-determinístico, no qual

1. nenhum estado possui uma transição-ϵ, isto é, uma transição à entrada ϵ, e
2. para cada estado *s* e símbolo de entrada *a* existe no máximo *um* lado rotulado *a* deixando *s*.

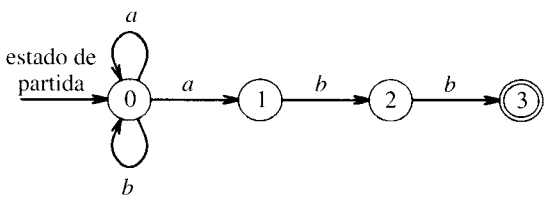


Fig. 3.19. Um autômato finito não-determinístico.

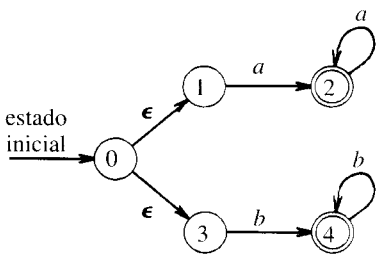


Fig. 3.21. AFN que aceita $aa^* \mid bb^*$.

Um autômato finito possui no máximo uma transição, a partir de cada estado, para qualquer símbolo de entrada. Se estivermos usando uma tabela de transições para representar a função de transição de um AFD, então cada entrada na tabela de transições será um único estado. Como conseqüência, é muito fácil determinar se um autômato finito determinístico aceita uma cadeia de entrada, dado que existe no máximo um único percurso, rotulado por aquela cadeia, a partir do estado inicial. O algoritmo seguinte mostra como simular o comportamento de um AFD, dada uma cadeia de entrada.

Algoritmo 3.1. Simulando um AFD.

Entrada. Uma cadeia de entrada x terminada por um caractere de fim de arquivo **eof.** Um AFD D com estado de partida s_0 e conjunto de estados de aceitação F.

Saída. A resposta "sim" se D aceitar x, "não" em caso contrário.

Método. Aplicar o algoritmo da Fig. 3.22 para a cadeia de entrada x. A função *movimento (s,c)* fornece o estado para o qual existe uma transição a partir do estado s e caractere de entrada c. A função *próximo* retorna o próximo caractere da cadeia de entrada x. □

Exemplo 3.14. Na Fig. 3.23, vemos um grafo de transições de um autômato finito determinístico que aceita a mesma linguagem $(a \mid b)^*abb$, aceita pelo AFN da Fig. 3.19. Com este AFD e a cadeia de entrada *ababb*, o Algoritmo 3.1 segue a seqüência de estados 0, 1, 2, 1, 2, 3 e retorna "sim". □

Conversão de um AFN num AFD

Note-se que o AFN da Fig. 3.19 possui duas transições do estado 0 e entrada a; ou seja, pode-se ir para o estado 0 ou 1. Similarmente, o AFN da Fig. 3.21 possui duas transições em ϵ a partir do estado 0. Conquanto não tenhamos mostrado um exemplo, uma situação onde pudéssemos escolher entre uma transição em ϵ ou num símbolo real de entrada também causaria ambigüidade. Essas situações, nas quais a função de transição é multiavaliada, tornam difícil simular um AFN com um programa de computador. A definição de aceitação meramente estabelece que precisa existir algum percurso rotulado pela cadeia de entrada em questão, indo do estado inicial até um estado de aceitação. Mas, se existirem vários percursos que soletrem a mesma cadeia de entrada, podemos ter que considerá-los todos antes de encontrarmos um que leve à aceitação ou descobrir que nenhum deles o faz.

Apresentamos agora um algoritmo para construir um AFD a partir de um AFN que reconheça a mesma linguagem. Esse algoritmo, freqüentemente chamado de *construção de subconjuntos*, é útil na simulação de um AFN por um programa de computador. Um algoritmo estreitamente relacionado desempenha um papel fundamental na construção de *parsers* LR, no próximo capítulo.

Na tabela de transições de um AFN, cada entrada é um conjunto de estados; na tabela de transições de um AFD, cada entrada é exatamente um único estado. A idéia geral por trás da construção AFN-para-AFD é que cada estado do AFD corresponde a um conjunto de estados do AFN. O AFD usa seu estado para controlar todos os possíveis estados que o AFN poderia estar após ler cada símbolo de entrada. Isto significa dizer, após ler a entrada $a_1a_2 \ldots a_n$, o AFD estará num estado que representa o subconjunto T dos estados do AFN que são atingíveis a partir do estado inicial do AFN juntamente com algum percurso rotulado $a_1a_2 \ldots a_n$. O número de estados do AFD pode ser uma exponencial do número de estados do AFN, mas na prática esse caso extremo ocorre raramente.

```
s := s0;
c := próximo_caractere;
enquanto  c ≠ eof faça
          s := movimento (s, c);
          c := próximo_caractere
fim;
se s estiver em F então
          retornar "sim"
senão retornar "não";
```

Fig. 3.22. Simulando um AFD.

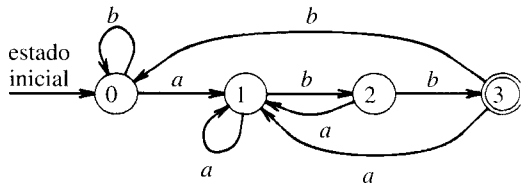


Fig. 3.23. AFD que aceita $(a \mid b)^*abb$.

Algoritmo 3.2. (*Construção de subconjuntos*). Construindo um AFD a partir de um AFN.

Entrada. Um AFN N.

Saída. Um AFD D aceitando a mesma linguagem.

Método. Nosso algoritmo constrói uma tabela de transições *Dtran* para D. Cada estado do AFD é um conjunto de estados do AFN e construímos *Dtran* de tal forma que D simule em "paralelo" todos os possíveis movimentos que N possa fazer a uma dada cadeia de entrada.

Usamos as operações da Fig. 3.24 para controlar os conjuntos de estados do AFN (s representa um estado do AFN e T, o conjunto de estados do AFN).

Antes que o mesmo enxergue o primeiro símbolo de entrada, N pode estar em qualquer um dos estados no conjunto *fechamento-*ϵ (s_0), onde s_0 é o estado inicial de N. Suponhamos que exatamente os estados do conjunto T sejam atingíveis a partir de s_0 a uma dada seqüência de símbolos de entrada e seja a o próximo símbolo de entrada. Ao enxergar a, N pode ir para qualquer um dos estados do conjunto *movimento* (T, a). Quando permitimos transições-ϵ, N pode estar em qualquer um dos estados em *fechamento-*ϵ *(movimento* (T, a)*)*, após ver a.

Construímos *Estados-D*, o conjunto de estados de D, e *Dtran*, a tabela de transições para D, da seguinte maneira. Cada estado de D corresponde a um conjunto de estados do AFN que poderiam ser atin-

OPERAÇÃO	DESCRIÇÃO
fechamento-$\epsilon(s)$	Conjunto de estados do AFN atingíveis a partir de um estado s (do AFN) somente através de transições-ϵ.
fechamento-$\epsilon(T)$	Conjunto de estados do AFN atingíveis a partir de algum estado s do AFN, pertencente a T, somente em transições-ϵ.
movimento (T, a)	Conjunto de estados do AFN para o qual existe uma transição no símbolo de entrada a, a partir de algum estado s do AFN, pertencente a T.

Fig. 3.24. Operações sobre os estados do AFN.

```
inicialmente, fechamento-ϵ(s0) é o único estado em Estados-
D e está não marcado; enquanto existir um estado T não
marcado em Estados-D faça início
    marcar T;
    para cada símbolo de entrada a faça início
        U := fechamento-ϵ(movimento (T, a));
        se U não está em Estados-D então
            adicione U como um estado não marcado a
Estados-D;
        Dtran[T, a] := U
    fim
fim
```

Fig. 3.25. A construção de subconjuntos.

gidos em N após ler alguma seqüência de símbolos de entrada, incluindo todas as possíveis transições-ϵ antes ou depois dos símbolos terem sido lidos. O estado de partida de D é *fechamento*-ϵ (s_0). Os estados e transições são adicionados a D usando-se o algoritmo da Fig. 3.25. Um estado de D é de aceitação se for um conjunto de estados do AFN contendo pelo menos um estado de aceitação de N.

O cômputo do *fechamento*-ϵ (T) é um processo típico de busca dos nós atingíveis a partir de um dado conjunto de nós num grafo. Nesse caso, os estados de T são os conjuntos de nós dados e o grafo consiste exatamente nos lados rotulados ϵ no AFN. Um algoritmo simples para computar o *fechamento*-ϵ (T) usa uma pilha para guardar os estados que ainda não foram checados pela existência de lados representando transições ϵ. Tal procedimento é mostrado na Fig. 3.26. □

Exemplo 3.15. A Fig. 3.27 mostra outro AFN N que aceita a linguagem $(a \mid b)^*abb$. (Acontece que é o mesmo AFN da próxima seção, que será construído mecanicamente a partir da expressão regular.) Vamos aplicar o Algoritmo 3.2 a N. O estado inicial do AFD equivalente é *fechamento*-ϵ (0), que é A = {0, 1, 2, 4, 7}, já que esses são exatamente os estados atingíveis a partir do estado 0 através de um percurso no qual cada lado é rotulado ϵ. Note-se que um percurso pode não ter lados, de tal forma que 0 é atingido a partir de si mesmo dessa forma.

O alfabeto de símbolos de entrada aqui é $\{a, b\}$. O algoritmo da Fig. 3.25 nos diz para marcar A e, em seguida, computar

fechamento-ϵ *(movimento (A, a))*.

Primeiro computamos *movimento (A, a)*, o conjunto de estados de N tendo transições em a a partir de membros da A. Dentre os estados 0, 1, 2, 4 e 7, somente 2 e 7 têm tais transições, para 3 e 8, e, então,

fechamento-ϵ *(movimento* ({0, 1, 2, 4, 7}, *a*)) = *fechamento*-ϵ ({3,8}) = {1, 2, 3, 4, 6, 7, 8}

Vamos chamar esse conjunto de B. Então, *Dtran* $[A, a] = B$.

```
empilhar todos os estados em T na pilha;
inicializar fechamento-ϵ (T) com T;
enquanto pilha não estiver vazia executar início
    desempilhar t, o elemento de topo, para fora da pilha;
    para cada estado u com um lado de t para u rotulado ϵ
    executar se u não estiver em fechamento-ϵ(T) executar
        início adicionar u ao fechamento-ϵ(T);
        empilhar u na pilha
    fim
fim
```

Fig. 3.26. Cômputo de *fechamento*-ϵ.

Dos estados em A, somente 4 possuem uma transição em b para 5, e, conseqüentemente, o AFD possui uma transição em b de A para

$$C = \textit{fechamento-}\epsilon\ (\{5\}) = \{1,2,4,5,6,7\}$$

Dessa forma, *Dtran* $[A, b] = C$.

Se continuarmos esse processo com os conjuntos ainda não marcados B e C, atingiremos eventualmente o ponto onde todos os conjuntos que sejam estados do AFD estejam marcados. Isto é certo, dado que existem "somente" 2^{11} subconjuntos diferentes num conjunto de onze estados e um conjunto, uma vez marcado, está marcado para sempre. Os cinco conjuntos de estados diferentes que efetivamente construímos são:

$A = \{0, 1, 2, 4, 7\}$ $\quad$ $D = \{1, 2, 4, 5, 6, 7, 9\}$
$B = \{1, 2, 3, 4, 6, 7, 8\}$ $\quad$ $E = \{1, 2, 4, 5, 6, 7, 10\}$
$C = \{1, 2, 4, 5, 6, 7\}$

O estado A é o estado de partida e o estado E é o único estado de aceitação. A tabela de transições completa *Dtran* é mostrada na Fig. 3.28.

Um grafo de transições para o AFD resultante é mostrado na Fig. 3.29. Deveria ser notado que o AFD da Fig. 3.23 também aceita $(a \mid b)^*abb$ e possui um estado a menos. Discutiremos a questão da minimização do número de estados do AFD na Seção 3.9. □

3.7 DE UMA EXPRESSÃO REGULAR PARA UM AFN

Existem muitas estratégias para se construir um reconhecedor a partir de uma expressão regular, cada uma com suas próprias fraquezas e vantagens. Uma estratégia que tem sido usada em alguns programas de edição de texto é a de construir um AFN a partir de uma expressão regular e então simular o comportamento do AFN numa cadeia de entrada usando os algoritmos 3.3 e 3.4 desta seção. Se o tempo de execução for essencial, podemos converter o AFN num AFD usando a construção de subconjuntos da seção anterior. Na Seção 3.9, examinamos uma alter-

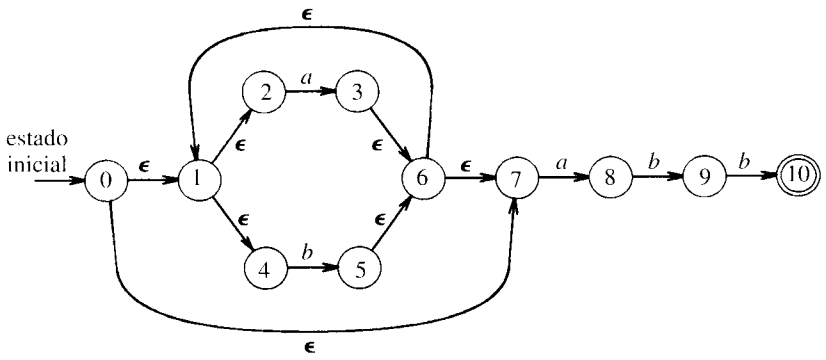


Fig. 3.27. AFN N para $(a \mid b)^*\ abb$.

ESTADO	SÍMBOLO DE ENTRADA	
	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C

Fig. 3.28. Tabela de transições *Dtran* para o AFD.

nativa de implementação de um AFD a partir de uma expressão regular, na qual um AFN intermediário não é explicitamente construído. Esta seção conclui com uma discussão sobre as barganhas de tempo-espaço na implementação de reconhecedores baseados em AFNs e AFDs.

Construção de um AFN a partir de uma Expressão Regular

Daremos agora um algoritmo para construir um AFN a partir de uma expressão regular. Existem muitas variantes desse algoritmo, mas aqui apresentamos uma versão simples, que é fácil de implementar. O algoritmo é dirigido pela sintaxe na medida em que usa a estrutura sintática da expressão regular para guiar o processo de construção. As alternativas no algoritmo seguem as alternativas na definição de uma expressão regular. Primeiro mostramos como construir autômatos que reconheçam ϵ e qualquer símbolo no alfabeto. Em seguida, mostramos como construir autômatos para expressões contendo um operador de alternação, concatenação e de fechamento Kleene. Por exemplo, para a expressão $r \quad s$, construímos um AFN indutivamente a partir do AFN para r e s.

À medida que a construção prossegue, cada passo introduz pelo menos dois novos estados, e, então, o AFN resultante, construído a partir de uma expressão regular, possui como número de estados, no máximo o dobro do número de símbolos e de operadores existentes na expressão regular.

Algoritmo 3.3. (*Construção de Thompson*). Um AFN a partir de uma expressão regular.

Entrada. Uma expressão regular r sobre um alfabeto Σ.

Saída. Um AFN N que aceita $L(r)$.

Método. Primeiro, dividimos r gramaticalmente em suas expressões constituintes. Em seguida, usando as regras (1) e (2) abaixo, construímos AFNs para cada um dos símbolos básicos em r (aqueles que sejam ϵ ou um símbolo de alfabeto). Os símbolos básicos correspondem às partes (1) e (2) na definição de uma expressão regular. É importante compreender que se um símbolo a ocorrer várias vezes em r, um AFN separado é construído para cada ocorrência.

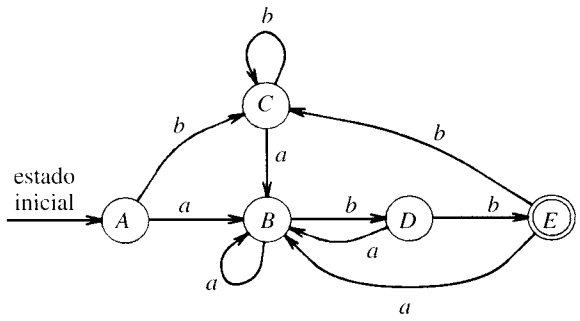
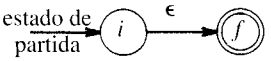


Fig. 3.29. Resultado da aplicação da construção de subconjuntos à Fig. 3.27.

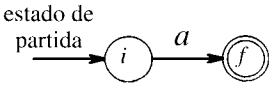
Em seguida, guiados pela estrutura sintática da expressão regular r, combinamos esses AFNs indutivamente usando a regra (3) abaixo, até que obtenhamos o AFN para toda a expressão. Cada AFN intermediário produzido durante o curso da construção corresponde a uma subexpressão de r e possui várias propriedades importantes; possui exatamente um estado final, nenhum lado entra no estado de partida e nenhum lado deixa o estado final.

1. Para ϵ, construímos o AFN



Aqui, i é um novo estado de partida e f um novo estado de aceitação. Este AFN claramente reconhece $\{\epsilon\}$.

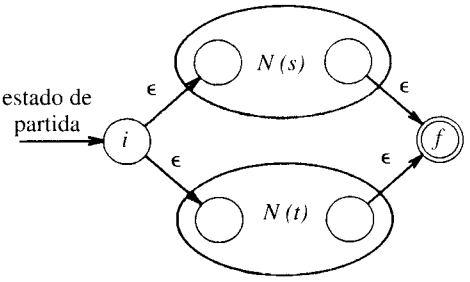
2. Para a em Σ, construímos o AFN



De novo, i é um novo estado e f um novo estado de aceitação. Essa máquina reconhece $\{a\}$.

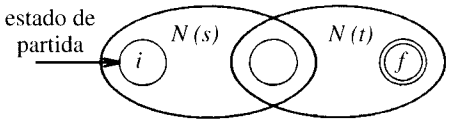
3. Suponhamos que $N(s)$ e $N(t)$ sejam AFNs para as expressões regulares s e t.

a) Para a expressão regular $s \quad t$, construímos o seguinte AFN composto $N(s \quad t)$:



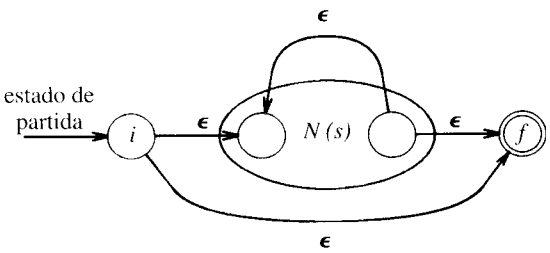
Aqui, i é um novo estado de partida e f um novo estado de aceitação. Existe uma transição em ϵ a partir de i para os estados de partida de $N(s)$ e $N(t)$. Existe uma transição em ϵ a partir dos estados de aceitação de $N(s)$ e $N(t)$ para o novo estado de aceitação f. Os estados inicial e de aceitação de $N(s)$ e de $N(t)$ não são os estados inicial e de aceitação de $N(s \mid t)$. Note-se que qualquer percurso de i para f precisa passar exclusivamente através de $N(s)$ ou de $N(t)$. Dessa forma, vemos que o AFN composto reconhece $L(s) \cup L(t)$.

b) Para a expressão regular st, construímos o AFN composto $N(st)$:



O estado de partida de $N(s)$ e o de aceitação de $N(t)$ se tornam, respectivamente, os estados de partida e de aceitação do AFN composto. O estado de aceitação de $N(s)$ é fundido ao estado de partida de $N(t)$; isto é, todas as transições que emanam do estado de partida de $N(t)$ se tornam transições provenientes do estado de aceitação de $N(s)$. O novo estado resultante da fusão perde tanto o status de estado de aceitação (de $N(s)$) quanto o de partida (de $N(t)$).Um percurso de i para f precisa ir primeiro através de $N(s)$ e, em seguida, através de $N(t)$, de tal forma que o rótulo daquele percurso será uma cadeia em $L(s)\ L(t)$. Como nenhum lado entra no estado de partida de $N(t)$ ou deixa o estado de acei tação de $N(s)$, não pode haver percurso de i para f que trafegue de volta de $N(t)$ para $N(s)$. Por conseguinte, o AFN composto reconhece $L(s)\ L(t)$.

c) Para uma expressão regular s^*, construímos o AFN composto $N(s^*)$:



Aqui, i é o novo estado de partida e f, o novo estado de aceitação. No AFN composto, podemos ir diretamente de i para f ao longo do lado rotulado ϵ, representando o fato de que ϵ está em $(L(s))^*$, ou podemos ir de i para f passando através de $N(s)$ uma ou mais vezes. O AFN composto claramente reconhece $(L(s))^*$.

d) Para uma expressão regular parentetizada (s), usamos o próprio $N(s)$ como o AFN.

A cada vez que construímos um novo estado, fornecemos um nome distinto ao mesmo. Dessa maneira, dois estados de qualquer AFN componente não poderão ter o mesmo nome. Ainda que o mesmo símbolo apareça várias vezes em r, criamos para cada instância do símbolo um AFN separado com seus próprios estados. □

Podemos verificar que cada passo da construção do Algoritmo 3.3 produz um AFN que reconhece a linguagem correta. Adicionalmente, a construção produz um AFN $N(r)$ com as seguintes propriedades:

1. $N(r)$ possui, como número de estados, no máximo o dobro de símbolos e operadores em r. Isto segue do fato de que cada passo da construção cria no máximo dois novos estados.

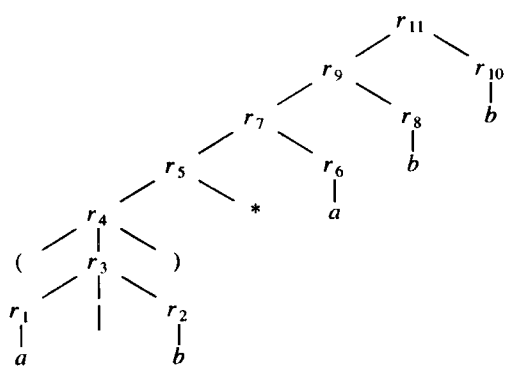
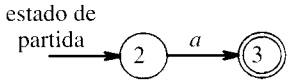


Fig. 3.30. Decomposição de $(a\ \ b)^*abb$.

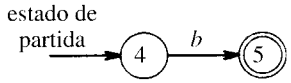
2. $N(r)$ possui exatamente um estado de partida e um de aceitação. O estado de aceitação não possui transições para fora. Esta propriedade vigora igualmente em cada um dos autômatos constituintes.

3. Cada estado de $N(r)$ possui ou uma transição para fora num símbolo de Σ ou no máximo duas transições para fora em ϵ.

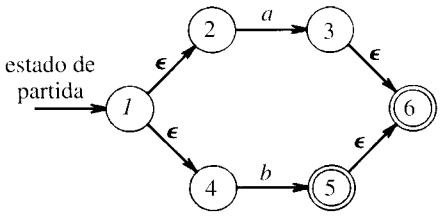
Exemplo 3.16. Vamos usar o Algoritmo 3.3 para construir $N(r)$ para a expressão regular $r = (a \mid b)^*abb$. A Fig. 3.30 mostra uma árvore gramatical para r que é análoga às árvores gramaticais construídas para as expressões aritméticas na Seção 2.2. Para o constituinte r_1, o primeiro a, construímos o AFN



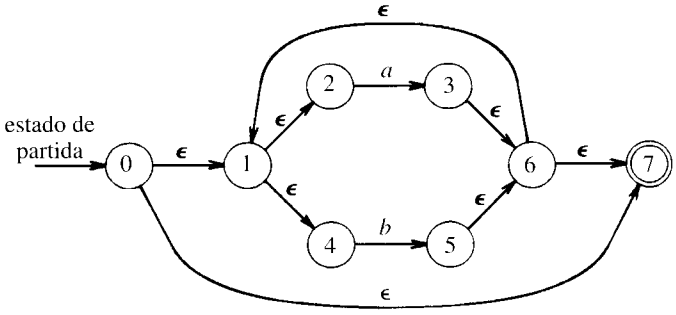
Para r_2, construímos



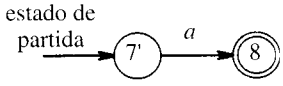
Podemos agora combinar $N(r_1)$ e $N(r_2)$ usando a regra da união a fim de obter o AFN para $r_3 = r_1 \mid r_2$



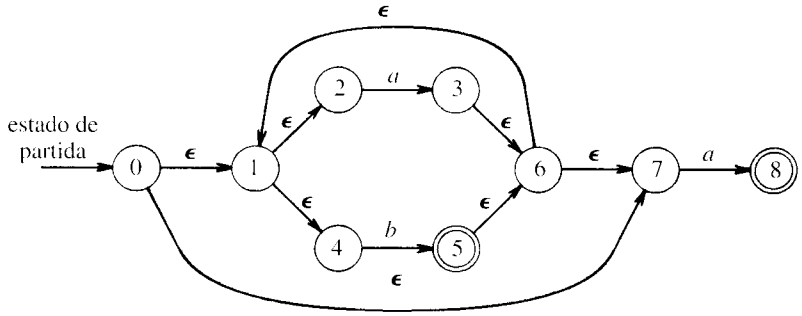
O AFN para (r_3) é o mesmo que o para r_3. O AFN para $(r_3)^*$ é, por conseguinte:



O AFN para $r_6 = a$ é



A fim de se obter o autômato para r_5r_6, combinamos os estados 7 e 7', chamando o estado resultante de 7, para obter



Continuando dessa maneira obtemos o AFN para $r_{11} = (a \mid b)^*abb$, que foi primeiramente exibido na Fig. 3.27. □

Simulação em Duas Pilhas de um AFN

Apresentamos agora um algoritmo que, dado um AFN N construído pelo Algoritmo 3.3, com uma cadeia de entrada x, determina se N aceita x. O algoritmo trabalha através da leitura de um caractere de entrada de cada vez e computa o conjunto completo de estados em que N poderia estar após ter lido cada prefixo da entrada. O algoritmo aproveita a vantagem das propriedades especiais do AFN produzido pelo Algoritmo 3.3 para computar cada conjunto de estados não determinísticos eficientemente. Pode ser implementado para rodar num tempo proporcional a $|N| \times |x|$ onde $|N|$ é o número de estados em N e $|x|$ é o comprimento de x.

Algoritmo 3.4. Simulação de um AFN.

Entrada. Um AFN construído pelo Algoritmo 3.3 e uma cadeia de entrada x. Assumimos que x seja terminada pelo caractere de fim de arquivo **eof.** N possui estado de partida s_0 e um conjunto de estados de aceitação F.

Saída. A resposta "sim" se N aceitar x; "não" em caso contrário.

Método. Aplicar o algoritmo delineado na Fig. 3.31 para a cadeia de entrada x. O algoritmo, com efeito, realiza a construção de subconjuntos em tempo de execução. Computa, em dois estágios, uma transição a partir do conjunto de estados correntes S para o próximo conjunto. Primeiro, determina *movimento* (S, a), isto é, todos os estados que podem ser atingidos a partir de qualquer estado em S através de uma transição em a, o caractere corrente de entrada. Em seguida, computa o *fechamento*-ϵ de *movimento* (S, a), ou seja, todos os estados que podem ser atingidos a partir de *movimento* (S, a) através de zero ou mais transições ϵ. O algoritmo usa a função *próximo_caractere* para ler os caracteres de x, um de cada vez. Quando todos os caracteres de x tiverem sido examinados, o algoritmo retorna "sim" se um estado de aceitação estiver no conjunto S de estados correntes; "não" em caso contrário. □

O Algoritmo 3.4 pode ser eficientemente implementado usando-se duas pilhas e um vetor de *bits* indexado pelos estados do AFN.

```
S := fechamento ϵ ( {s0} );
a := próximo_caractere;
enquanto a ≠ eof faça início
        S := fechamento ϵ (movimento (S, a) );
        a := próximo_caractere;
fim
se S ∩ F ≠ ∅ então
        retornar "sim";
senão retornar "não";
```

Fig. 3.31. Simulação de um AFN do Algoritmo 3.3.

Usamos uma pilha para controlar o conjunto corrente de estados não-determinísticos e a outra pilha para computar o próximo conjunto de estados não-determinísticos. Podemos usar o algoritmo da Fig. 3.26 para computar o *fechamento*-ϵ. O vetor de *bits* pode ser usado para determinar em tempo constante se um estado não-determinístico já está na pilha, de forma a não o adicionarmos duas vezes à mesma. Uma vez que tenhamos computado o próximo estado numa segunda pilha, podemos intercambiar os papéis das duas. Como cada estado não-determinístico possui no máximo duas transições para fora, cada estado pode dar origem a, no máximo, dois novos estados numa transição. Vamos denominar de $|N|$ o número de estados de N. Como podem existir no máximo $|N|$ estados numa pilha, o cômputo do próximo conjunto de estados a partir do conjunto de estados corrente pode ser feito num tempo proporcional a $|N|$. Conseqüentemente, o tempo total necessário para simular o comportamento de N para a entrada x é proporcional a $|N| \times |x|$.

Exemplo 3.17. Seja N o AFN da Fig. 3.27 e seja x a cadeia constituída do único caractere a. O estado de partida é o *fechamento*-ϵ $(\{0\}) = \{0, 1, 2, 4, 7\}$. Ao símbolo de entrada a existe uma transição de 2 para 3 e de 7 para 8. Por conseguinte, T é $\{3, 8\}$. Tomando o *fechamento* -ϵ de T, temos o próximo estado $\{1, 2, 3, 4, 6, 7, 8\}$. Como nenhum desses estados não-determinísticos é de aceitação, o algoritmo retorna "não".

Note-se que o Algoritmo 3.4 realiza a construção de subconjuntos em tempo de execução. Por exemplo, comparemos as transições acima com os estados do AFD da Fig. 3.29 construído a partir do AFN da Fig. 3.27. Os conjuntos de estados inicial e próximo à entrada a correspondem aos estados A e B do AFD. □

Barganhas de Tempo-Espaço

Dada uma expressão regular r e uma cadeia de entrada x, temos agora dois métodos para determinar se x está em $L(r)$. Um enfoque é o de usar o Algoritmo 3.3 para construir um AFN N a partir de r. Esta construção pode ser feita num tempo $O(|r|)$ (proporcional a $|r|$), onde $|r|$ é o comprimento de r. N possui um número de estados que é no máximo o dobro de $|r|$ e, também, duas transições, no máximo, a partir de cada estado, de tal forma que a tabela de transições para N pode ser armazenada num espaço $O(|r|)$. Podemos, então, usar o Algoritmo 3.4 para determinar se N aceita x num tempo $O(|r| \times |x|)$. Dessa forma, usando esta abordagem, podemos determinar se x está em $L(r)$ num tempo total proporcional ao comprimento de r vezes o comprimento de x. Esse enfoque tem sido usado em alguns editores de texto a fim de procurar por padrões de expressões regulares quando a cadeia-alvo x não é geralmente muito longa.

Uma segunda abordagem é a de construir um AFD a partir de uma expressão regular r através da aplicação da construção de Thompson para r e, em seguida, usar a construção de subconjuntos, Algoritmo 3.2, para o AFN resultante (uma implementação que evita a construção explícita de um AFN é dada na Seção 3.9). Implementada a função de transição com uma tabela de transições, podemos usar o Algoritmo 3.1 para simular o AFD para a entrada x num tempo proporcional ao comprimento de x, independentemente do número de estados do AFD. Esse enfoque tem sido freqüentemente usado em programas de

AUTÔMATO	ESPAÇO	TEMPO
NFA	$O(\|r\|)$	$O(\|r\| \times \|x\|)$
DFA	$O(2^{\|r\|})$	$O(\|x\|)$

Fig. 3.32. Espaço e tempo requeridos para reconhecer expressões regulares.

reconhecimento de padrões que percorrem arquivos de texto procurando padrões de expressões regulares. Uma vez que o autômato finito tenha sido construído, a procura pode proceder muito rapidamente, e, então, esse enfoque é vantajoso quando a cadeia-alvo x é muito longa.

Existem, entretanto, certas expressões regulares cujo menor AFD possui um número de estados cujo número é uma exponencial do tamanho da expressão regular. Por exemplo, a expressão regular $(a \mid b)^* a (a \mid b)(a \mid b) \ldots (a \mid b)$, onde existem $n - 1$ $(a \mid b)$'s ao fim, não possui um AFD com menos de 2^n estados. Essa expressão regular denota qualquer cadeia de a's e b's na qual o enésimo caractere a partir da extremidade à direita é um a. Não é difícil provar que qualquer AFD para esta expressão precisa controlar os últimos n caracteres que examina na entrada; de outra forma, poderia fornecer uma resposta errada. Claramente, são necessários pelo menos 2^n estados para controlar todas as possíveis seqüências de n a's e b's. Felizmente, expressões como essa não ocorrem muito freqüentemente em aplicações de análise léxica, mas existem aplicações onde expressões similares efetivamente emergem.

Um terceiro enfoque é o de usar um AFD, mas evitar construir toda a tabela de transições utilizando uma técnica chamada "avaliação preguiçosa de transições". Aqui, as transições são computadas em tempo de execução, mas uma transição a partir de um dado estado e caractere não é determinada até que seja efetivamente necessitada. As transições computadas são armazenadas num *cache*. A cada vez que uma transição estiver prestes a ser realizada, o *cache* é consultado. Se a transição não estiver lá, é então computada e armazenada no *cache*. Se o *cache* encher, podemos apagar algumas transições previamente computadas a fim de abrir espaço para a nova transição.

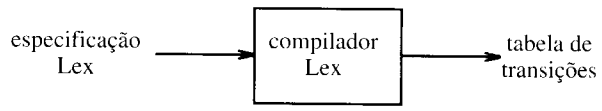
A Fig. 3.32 sumariza o pior caso de exigência de espaço e tempo para se determinar se uma cadeia de entrada x está na linguagem denotada por uma expressão regular r, usando reconhecedores construídos a partir de autômatos determinísticos e não-determinísticos. A técnica "preguiçosa" combina as exigências de espaço do método do AFN com as de tempo da abordagem do AFD. Sua exigência de espaço é o tamanho da expressão regular mais o tamanho do *cache;* seu tempo de execução observado é quase tão curto quanto aquele de um reconhecedor determinístico. Em algumas aplicações, a técnica "preguiçosa" é consideravelmente mais rápida do que o enfoque do AFD, porque nenhum tempo é desperdiçado computando-se transições de estados que jamais serão usadas.

3.8 O PROJETO DE UM GERADOR DE ANALISADORES LÉXICOS

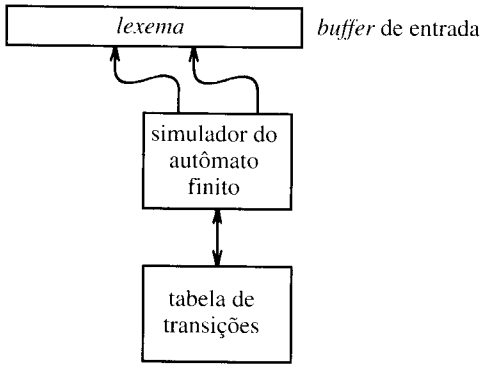
Nesta seção, consideramos o projeto de uma ferramenta de *software* que constrói automaticamente um analisador léxico a partir de um programa na linguagem Lex. Apesar de discutirmos vários métodos, e de nenhum ser precisamente idêntico àquele usado pelo comando Lex do sistema UNIX, referimo-nos a esses programas para construir analisadores léxicos como compiladores Lex.

Assumimos que temos uma especificação de um analisador léxico da forma

$$\begin{array}{ll} p_1 & \{ação_1\} \\ p_2 & \{ação_2\} \\ \ldots & \ldots \\ p_n & \{ação_n\} \end{array}$$



(a) Compilador Lex.



(b) Analisador léxico esquematizado.

Fig. 3.33. Modelo de um compilador Lex.

onde, como na Seção 3.5, cada padrão p_i é uma expressão regular e cada $ação_i$ é um fragmento de programa que deve ser executado sempre que um lexema reconhecido por p_i for encontrado na entrada.

Nosso problema é o de construir um reconhecedor que procure por lexemas no *buffer* de entrada. Se mais de um padrão for reconhecido, o reconhecedor deverá escolher o mais longo lexema encontrado. Se existirem dois ou mais padrões que reconheçam o lexema mais longo, o primeiro padrão de reconhecimento listado será escolhido.

Um autômato finito é um modelo natural em torno do qual se pode construir um analisador léxico e aquele construído por nosso compilador Lex possui a forma mostrada na Fig. 3.33 (b). Lá está o *buffer* de entrada com dois apontadores para o mesmo, o início_de_lexema e o apontador_adiante, como discutido na Seção 3.2. O compilador Lex constrói uma tabela de transições para um autômato finito a partir de padrões de expressões regulares na especificação Lex. O analisador léxico por si só consiste em um simulador de autômato finito que usa esta tabela de transições para procurar pelos padrões de expressões regulares no *buffer* de entrada.

O resto desta seção mostra que a implementação de um compilador *Lex* pode ser baseada quer num autômato determinístico, quer num não-determinístico. Ao fim da última seção, vimos que a tabela de transições de um AFN para um padrão de expressão regular podia ser consideravelmente menor do que a de um AFD, mas que o AFD possuía a decidida vantagem de ser capaz de reconhecer padrões mais rapidamente do que um AFN.

Reconhecimento de Padrões Baseado em AFN's

Um método é o de construir a tabela de transições de um autômato finito não-determinístico N para o padrão composto $p_1 \mid p_2 \mid \ldots \mid p_n$. Isto pode ser feito primeiro criando-se um AFN $N(p_i)$ para cada padrão p_i, usando o Algoritmo 3.3 e, em seguida, adicionando-se um novo estado de partida s_0 e, finalmente, ligando-se s_0 ao estado de partida de cada $N(p_i)$ através de uma transição ϵ, como mostrado na Fig. 3.34.

Para simular este AFN, podemos usar uma modificação do Algoritmo da Fig. 3.4. A modificação assegura que o AFN combinado reconheça o mais longo prefixo da entrada que seja reconhecido por um padrão. No AFN combinado, existe um estado de aceitação para cada padrão p_i. Ao simularmos o AFN usando o Algoritmo 3.4, cons-

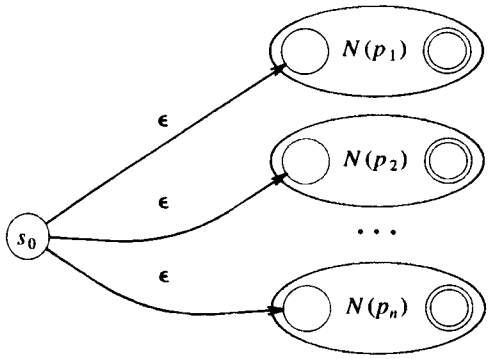


Fig. 3.34. AFN construído a partir de uma especificação Lex.

truímos a seqüência de conjuntos de estados em que o AFN combinado possa estar após examinar cada caractere de entrada. Mesmo se encontrarmos um conjunto de estados que contenha um estado de aceitação, para obtermos a correspondência mais longa precisaremos continuar a simular o AFN até que o mesmo atinja a *terminação*, isto é, o conjunto de estados a partir do qual não existam transições no símbolo corrente de entrada.

Presumimos que a especificação Lex seja projetada de tal forma que um programa válido de entrada não possa preencher inteiramente o *buffer* de entrada sem que o AFN tenha atingido a terminação. Por exemplo, cada compilador coloca alguma restrição no comprimento de um identificador e as violações desse limite serão detectadas quando o *buffer* de entrada estourar a capacidade de armazenamento, se não mais cedo.

Para atingir o reconhecimento correto, fazemos duas modificações ao Algoritmo 3.4. Primeiro, sempre que adicionarmos um estado de aceitação ao conjunto corrente de estados, registramos a posição corrente de entrada e o padrão p_i correspondente a esse estado de aceitação. Se o conjunto corrente entrada já contiver um estado de aceitação, então somente o padrão que aparece primeiro na especificação de Lex é registrado. Segundo, continuamos fazendo transições até encontrarmos a terminação. Ao final, retraímos o apontador adiante para a posição na qual o último reconhecimento ocorreu. O padrão que realiza esse reconhecimento identifica o *token* encontrado e o lexema reconhecido é a cadeia entre os apontadores de início de lexema e adiante.

Usualmente, a especificação Lex é tal que algum padrão, possivelmente um padrão de erro, irá sempre reconhecer. Se nenhum padrão o fizer, entretanto, temos uma condição de erro para a qual nenhuma provisão foi feita e o analisador léxico deveria transferir o controle para alguma rotina *default* de recuperação de erros.

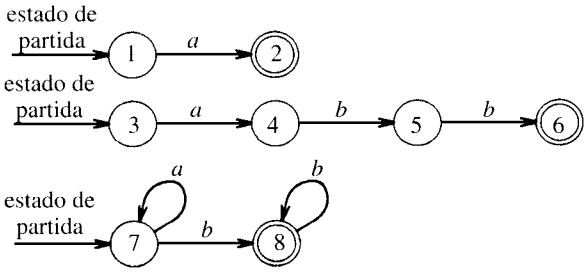
Exemplo 3.18. Um único exemplo ilustra as idéias acima. Suponhamos ter o seguinte programa Lex, consistindo em três expressões regulares e nenhuma definição regular.

```
a     { } /* as ações são omitidas aqui */
abb   { }
a*b+  { }
```

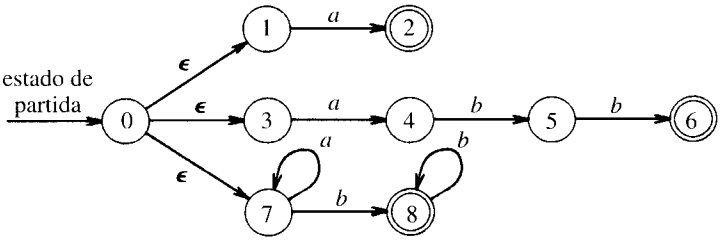
Os três *tokens* acima são reconhecidos pelos autômatos da Fig. 3.35(a). Simplificamos o terceiro autômato em algumas partes em relação àquele que teria sido produzido pelo Algoritmo 3.3. Como indicado acima, podemos converter os AFNs da Fig. 3.35(a) no único AFN combinado N mostrado à Fig. 3.35(b).

Vamos considerar o comportamento de N na cadeia de entrada *aaba* usando nossa modificação do Algoritmo 3.4. A Fig. 3.36 mostra os conjuntos de estados e padrões que reconhecem à medida que cada caractere da entrada *aaba* é processado. Essa figura mostra que o conjunto inicial de estados é $\{0, 1, 3, 7\}$. Os estados 1, 3 e 7 têm cada um uma transição em a para os estados 2, 4 e 7, respectivamente. Como o estado 2 é o de aceitação para o primeiro padrão, registramos o fato de que o primeiro padrão reconhece positivamente após a leitura do primeiro a.

Entretanto, existe uma transição a partir do estado 7 para o estado 7 no segundo caractere de entrada e, por conseguinte, precisamos continuar realizando as transições. Existe uma transição a partir do estado 7 para o 8 no caractere de entrada b. O estado 8 é o estado de aceitação para o terceiro padrão. Uma vez que atinjamos o estado 8, não existem transições possíveis para o próximo caractere de entrada a e, nesse caso, atingimos terminação. Como o último reconhecimento ocorreu após lermos o terceiro caractere de entrada, relatamos que o terceiro padrão reconheceu o lexema *aab*. □



(a) AFN para a, abb e a^*b^+.



(b) AFN combinado.

Fig. 3.35. AFN reconhecendo três diferentes padrões.

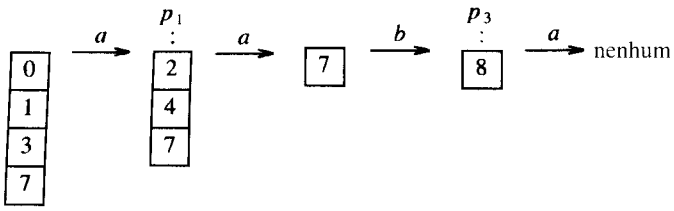


Fig. 3.36. Seqüência de estados atingidos no processamento da entrada *aaba*.

O papel da *ação*$_i$ associada ao padrão p_i na especificação Lex é como se segue. Quando uma instância de p_i é reconhecida, o analisador léxico executa o programa associado *ação*$_i$. Note-se que *ação*$_i$ não é executada somente porque o AFN entra num estado que inclua um estado de aceitação para p_i; a *ação*$_i$ somente é executada se p_i vier a se tornar o padrão que produza o reconhecimento mais longo.

AFDs para Analisadores Léxicos

Um outro enfoque para a construção de um analisador léxico é o de usar um AFD para realizar o reconhecimento de padrões. A única nuança é certificar que encontramos os padrões de reconhecimento adequados. A situação é completamente análoga à simulação modificada do AFN já descrito. Quando convertemos um AFN para um AFD usando a construção de subconjuntos do Algoritmo 3.2, pode haver vários estados de aceitação num dado subconjunto de estados não-determinísticos. Em tal situação, o estado de aceitação correspondente ao padrão listado à frente na especificação Lex possui prioridade. Como na simulação do AFN, *a única outra modificação que precisamos fazer é a de continuar* realizando transições até que tenhamos atingido um estado sem nenhum próximo estado (isto é, o estado $\varnothing$) para o símbolo corrente de entrada. Para encontrar o lexema reconhecido, retornamos à última posição de entrada a qual o AFD entrou num estado de aceitação.

Exemplo 3.19. Se convertermos o AFN da Fig. 3.35 para um AFD, obtemos a tabela de transições da Fig. 3.37, onde os estados do AFD foram designados por listas de estados do AFN. A última coluna da Fig. 3.37 indica um dos padrões reconhecidos ao se entrar naquele estado do AFD. Por exemplo, dentre os estados 2, 4 e 7 do AFN, somente 2 é um estado de aceitação e é o estado de aceitação do autômato para a expressão regular *a* na Fig. 3.35(a). Conseqüentemente, o estado 247 do AFD reconhece o padrão *a*.

Note-se que a cadeia *abb* é reconhecida por dois padrões, *abb* e a^*b^+, reconhecidos pelos estados 6 e 8 do AFN. O estado 68 do AFD, na última linha da tabela de transições inclui, pois, dois estados de aceitação para o AFN. Notamos que *abb* aparece antes de a^*b^+ nas regras de tradução de nossa especificação Lex e, então, anunciamos que *abb* foi encontrado no estado 68 do AFD.

Com a cadeia de entrada *aaba*, o AFD entra nos estados sugeridos pela simulação do AFN mostrada na Fig. 3.36. Consideremos um segundo exemplo, a cadeia de entrada *aba*. O AFD da Fig. 3.37 começa no estado 0137. A entrada *a* vai para o estado 247. Em seguida, à entrada *b*, progride até o estado 58 e, à entrada *a*, não há próximo estado. Atingimos, por conseguinte, a terminação, progredindo através dos estados 0137, em seguida 247 e então 58. O último desses estados inclui

Estado	Símbolo de Entrada		Padrão Anunciado
	a	*b*	
0137	247	8	nenhum
247	7	58	*a*
8	—	8	a^*b^+
7	7	8	nenhum
58	—	68	a^*b^+
68	—	8	*abb*

Fig. 3.37. Tabela de transições para um AFD.

o estado de aceitação 8 da Fig. 3.35(a). Conseqüentemente, no estado 58, o AFD anuncia que o padrão a^*b+ foi reconhecido e seleciona *ab*, o prefixo da entrada que levou ao estado 58, como o lexema. □

Implementando o Operador *Lookahead*

Relembremos da Seção 3.4 que o operador *lookahead* / é necessário em algumas situações, dado que o padrão que denota um *token* particular pode precisar descrever algum contexto após o lexema efetivo. Quando convertemos um padrão com / para um AFN, podemos tratar a / como se fosse ϵ, e, então, não precisamos efetivamente procurar por / à entrada. Entretanto, se uma cadeia denotada por esta expressão regular for reconhecida no *buffer* de entrada, o final do lexema não é a posição do estado de aceitação do AFN. Ao invés, é a última ocorrência de estado deste AFN que tenha uma transição (imaginária) em /.

Exemplo 3.20. O AFN que reconhece o padrão `IF` dado no Exemplo 3.12 é mostrado na Fig. 3.38. O estado 6 indica a presença da palavra-chave `IF`; entretanto, encontramos o *token* `IF` esquadrinhando de volta até a última ocorrência do estado 2. □

3.9 OTIMIZAÇÃO DE RECONHECEDORES DE PADRÕES BASEADOS EM AFDs

Nesta seção, apresentamos três algoritmos que têm sido usados para implementar e otimizar reconhecedores de padrões construídos a partir de expressões regulares. O primeiro algoritmo é adequado para inclusão num compilador Lex, porque constrói um AFD diretamente a partir de uma expressão regular, sem construir um AFN intermediário ao longo do caminho.

O segundo algoritmo minimiza o número de estados de qualquer AFD, e, então, pode ser usado para reduzir o tamanho do reconhecedor de padrões baseado em AFDs. O algoritmo é eficiente; seu tempo de execução é $O(n \log n)$, onde n é o número de estados do AFD. O terceiro algoritmo pode ser usado para produzir representações rápidas porém mais compactas para a representação da tabela de transições do AFD do que uma representação direta sob a forma de uma tabela bidimensional.

Estados Importantes de um AFN

Vamos chamar de *importante* um estado de um AFN que tenha uma transição não-ϵ. A construção de subconjuntos na Fig. 3.25 usa somente

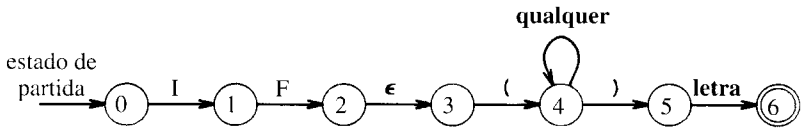


Fig. 3.38. AFN reconhecendo a palavra-chave Fortran `IF`.

os estados importantes de um subconjunto T quando determina o *fechamento-*ϵ (*movimento* (T, a)), o conjunto de estados que são atingíveis a partir de T e entrada a. O conjunto *movimento* (s, a) é não vazio somente se o estado s for importante. Durante a construção, dois subconjuntos podem ser unificados se possuírem os mesmos estados importantes e, ambos ou nenhum dos dois, incluírem estados de aceitação do AFN.

Quando a construção de subconjuntos é aplicada a um AFN obtido pelo Algoritmo 3.3, a partir de uma expressão regular, podemos explorar as propriedades especiais do AFN para combinar as duas construções. A construção combinada relaciona os estados importantes do AFN com os símbolos na expressão regular. A construção de Thompson modela um estado importante exatamente quando um símbolo no alfabeto aparece numa expressão regular. Por exemplo, os estados importantes serão construídos para cada a e b em $(a \mid b)^*abb$.

Sobretudo, o AFN resultante possui exatamente um estado de aceitação, mas o mesmo não é importante pois não possui transições para fora. Pela concatenação de um marcador único de fim à direita # a uma expressão regular r, damos ao estado de aceitação de r uma transição em #, tornando-o um estado importante do AFN para r#. Em outras palavras, usando-se a expressão regular expandida (r)# podemos esquecer a respeito dos estados de aceitação à medida que a construção de subconjuntos prossiga; quando a mesma estiver completa, qualquer estado do AFD com uma transição em # tem que ser um estado de aceitação.

Representamos uma expressão regular expandida através de uma árvore sintática com os símbolos básicos nas folhas e os operadores nos nós interiores. Referimo-nos a um nó interior como um *nó-cat*, *nó-alt* ou *nó-rep* se for rotulado por uma concatenação, | (alternação) ou * (repetição), respectivamente. A Fig. 3.39 (a) mostra a árvore sintática para uma expressão regular expandida, com os nós-cat marcados por pontos. A árvore sintática para uma expressão regular pode ser construída da mesma maneira que uma árvore sintática para uma expressão aritmética (veja o Capítulo 2).

As folhas na árvore sintática para uma expressão regular são rotuladas por símbolos de alfabeto ou por ϵ. Para cada nó rotulado por ϵ, atrelamos um único inteiro e referimo-nos ao mesmo como a *posição* da folha e, também, como a posição de seu símbolo. Um símbolo repetido possui várias posições. As posições são mostradas abaixo dos símbolos na árvore sintática da Fig. 3.39 (a). Os estados numerados no AFN da Fig. 3.39 (c) correspondem às posições das folhas na árvore sintática da Fig. 3.39 (a). Não é uma mera coincidência que estes sejam os estados importantes do AFN. Os estados não importantes são designados por letras maiúsculas na Fig. 3.39 (c).

O AFD na Fig. 3.39 (b) pode ser obtido a partir do AFN da Fig. 3.39 (c) se aplicarmos a construção de subconjuntos e unificarmos aqueles contendo os mesmos estados importantes. A unificação resulta na construção de um estado a menos, como mostra a comparação da Fig. 3.29.

De uma Expressão Regular para um AFD

Nesta seção, mostramos como construir um AFD diretamente a partir de uma expressão regular expandida (r)#. Começamos construindo uma árvore sintática T para (r)# e, em seguida, computamos quatro funções, *anulável*, *primeira_pos*, *última_pos* e *pos_seguinte*, através de travessias na árvore T. Finalmente, construímos um AFD a partir de *pos_seguinte*. As funções *anulável*, *primeira_pos* e *última_pos* são

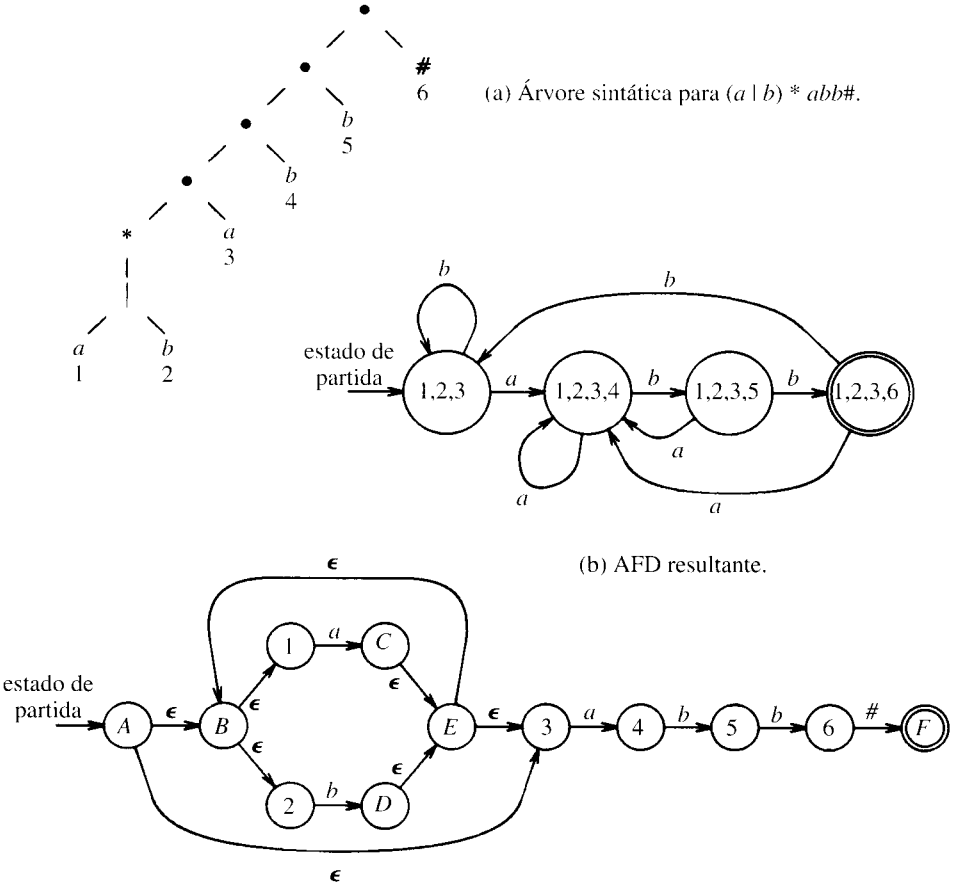


Fig. 3.39. AFD e AFN construídos a partir de $(a \mid b)^*\ abb\ \#$.

definidas sobre os nós da árvore sintática e usadas para computar *pos_seguinte*, que é definida sobre o conjunto de posições.

Relembrando a equivalência entre os estados importantes de um AFN e as posições das folhas na árvore sintática da expressão regular, podemos tomar um atalho na construção do AFN através da modelagem de um AFD cujos estados correspondem aos conjuntos de posições na árvore. As transições-ϵ do AFN representam alguma estrutura razoavelmente complicada das posições; em particular, codificam a informação correspondente quando uma posição pode se seguir a uma outra. Ou seja, cada símbolo numa cadeia de entrada para um AFD pode ser reconhecido por certas posições. Um símbolo de entrada c pode somente ser reconhecido pelas posições nas quais exista um c, mas nem toda posição que tenha um c pode necessariamente reconhecer uma ocorrência particular de c no fluxo de entrada.

A noção de uma posição reconhecendo um símbolo de entrada será definida em termos da função *pos_seguinte* sobre as posições da árvore sintática. Se i for uma posição, então *pos_seguinte* (i) é o conjunto de posições j tal que exista alguma cadeia de entrada ... cd ... em que i corresponda a esta ocorrência de c e j a esta ocorrência de d.

Exemplo 3.21. Na Fig. 3.39 (a), *pos_seguinte* (1) = {1, 2, 3}. A argumentação para tal é que se examinarmos um a correspondente à posição 1, então acabamos de ver uma ocorrência de $a \mid b$ no fechamento $(a \mid b)^*$. Poderíamos em seguida examinar a primeira posição de outra ocorrência de $a \mid b$, o que explica por que 1 e 2 estão em *pos_seguinte* (1). Poderíamos também em seguida examinar a primeira posição do que segue a $(a \mid b)^*$ isto é, a posição 3. □

A fim de computar a função *pos_seguinte*, necessitamos conhecer que posições podem reconhecer o primeiro ou o último símbolo de uma cadeia gerada por uma dada subexpressão de uma expressão regular. (Tal informação foi usada informalmente no exemplo 3.21.) Se r^* é uma tal subexpressão, cada posição que pode ser a primeira em r segue cada posição que pode ser a última em r. Similarmente, se rs é uma subexpressão, cada primeira posição de s segue cada última posição de r.

A cada nó n da árvore sintática de uma expressão regular, definimos uma função *primeira_pos* (n) que fornece o conjunto de posições que podem reconhecer o primeiro símbolo de uma cadeia gerada por uma subexpressão enraizada em n. Simetricamente, definimos uma função *última_pos* (n) que nos dá o conjunto de posições que podem reconhecer o último símbolo de uma tal cadeia. Por exemplo, se n é a raiz de toda a árvore na Fig. 3.39 (a), então *primeira_pos* (n) = {1, 2, 3} e *última_pos* (n) = {6}. Momentaneamente, fornecemos um algoritmo para computar essas funções.

Para computar *primeira_pos* e *última_pos*, precisamos saber que nós são as raízes das subexpressões que geram linguagens que incluem a cadeia vazia. Tais nós são chamados *anuláveis* e definimos *anulável* (n) como sendo verdadeiro se o nó n for anulável e falso em caso contrário.

Precisamos agora fornecer as regras para computar as funções *anulável*, *primeira_pos*, *última_pos* e *pos_seguinte*. Para as três primeiras funções, temos uma regra de base que nos diz a respeito de um símbolo básico e em seguida três regras indutivas que nos permitem determinar o valor de funções trabalhando a sintaxe de uma árvore a partir do fundo; em cada caso, as regras indutivas correspondem aos três operadores, de união, concatenação e fechamento. As regras para *anulável* e *primeira_pos* são dadas na Fig. 3.40. As regras para *última_pos* são as mesmas que as regras para *primeira_pos* (n), mas com c_1 e c_2 invertidas, e não são mostradas.

A primeira regra para *anulável* estabelece que se n é uma folha rotulada ϵ, então *anulável* (n) é certamente verdadeira. A segunda regra estabelece que se n é uma folha rotulada por um símbolo de alfabeto, então *anulável* (n) é falsa. Neste caso, cada folha corresponde a um único símbolo de entrada e conseqüentemente não pode gerar ϵ. A última regra para *anulável* estabelece que se n é um nó-rep com filho c_1, então *anulável* (n) é verdadeira porque o fechamento de uma expressão gera uma linguagem que inclui ϵ.

Como um outro exemplo, a quarta regra para *primeira_pos* estabelece que se n for um nó-*cat* com filho à esquerda c_1 e filho à direita c_2 e se *anulável* (c_1) é verdadeira, então

$$primeira_pos\ (n) = primeira_pos\ (c_1) \cup primeira_pos\ (c_2)$$

de outra forma, *primeira_pos* (n) = *primeira_pos* (c_1). O que esta regra diz é que se uma expressão rs r gera ϵ, então as primeiras posições de s "atravessam" r e são também primeiras posições de rs; de outra forma somente as primeiras posições de r são as primeiras posições de rs. A argumentação por trás das regras remanescentes para *anulável* e *primeira_pos* são similares.

A função *pos_seguinte* (i) nos diz que posições podem seguir a posição i na árvore sintática. Duas regras definem todas as formas em que uma posição pode se seguir a uma outra.

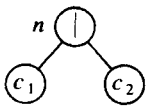
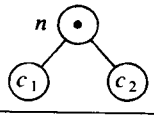
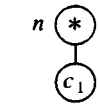
NÓ n	*anulável* (n)	*primeira-pos* (n)
n é uma folha rotulada ϵ	**verdadeiro**	$\varnothing$
n é uma folha rotulada com posição i	**falso**	$\{i\}$
	*anulável* (c_1) **ou** *anulável* (c_2)	*primeira-pos* (c_1) $\cup$ *primeira-pos* (c_2)
	*anulável* (c_1) **e** *anulável* (c_2)	**se** *anulável* (c_1) **então** *primeira-pos* (c_1) $\cup$ *primeira-pos* (c_2) **senão** *primeira-pos* (c_1)
	**verdadeiro**	*primeira-pos* (c_1)

Fig. 3.40. Regras para computar *anulável* e *primeira_pos*.

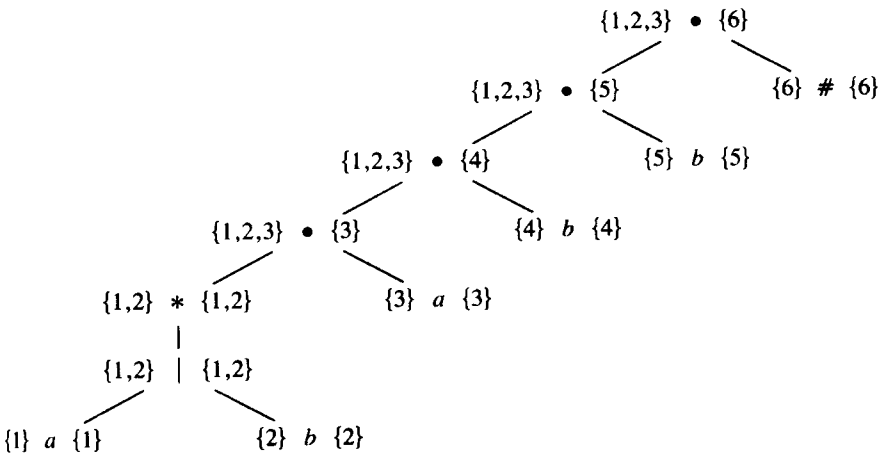


Fig. 3.41. *primeira_pos* e *última_pos* para os nós da árvore sintática para $(a \mid b)^*abb\ \#$.

1. Se n é um nó-*cat* com filho à esquerda c_1 e filho à direita c_2 e i é uma posição em *última_pos* (c_1), então todas as posições em *primeira_pos* (c_2) estão em *pos-seguinte* (i).

2. Se n é um nó-rep e i é uma posição em *última_pos* (n), então todas as posições em *primeira_pos* (n) estão em *pos_seguinte* (i).

Se *primeira_pos* e *última_pos* forem computadas para cada nó, *pos_seguinte* de cada posição pode sê-lo realizando-se uma travessia em profundidade (*depth first*) na árvore sintática.

Exemplo 3.22. A Fig. 3.41 mostra os valores de *primeira_pos* e de *última_pos* em todos os nós da árvore da Fig. 3.49 (a); *primeira_pos* (n) aparece à esquerda do nó n e *última_pos* (n) à direita. Por exemplo, *primeira_pos* ao nó à folha mais à esquerda, rotulada a é $\{1\}$, pois esta folha está rotulada coma a posição 1. Similarmente, *primeira_pos* da segunda folha é $\{2\}$, pois essa folha está rotulada com a posição 2. Pela terceira regra na Fig. 3.40, *primeira_pos* do pai de ambas é $\{1, 2\}$.

O nó rotulado* é o único nó anulável. Conseqüentemente, pela condição **se** da quarta regra, *primeira_pos* para o pai deste nó (aquele a representar a expressão $(a \mid b)^*a$) é a união de $\{1, 2\}$ e $\{3\}$, que são *primeira_pos* de seus filhos à esquerda e à direita. Por outro lado, a condição **senão** se aplica para *última_pos* deste nó, dado que uma folha à posição 3 não é anulável. Conseqüentemente, o pai do nó-rep possui *última_pos* contendo somente 3.

Vamos agora computar *pos_seguinte* de baixo para cima para cada nó da árvore sintática da Fig. 3.41. No nível do nó-rep, adicionamos ambos, 1 e 2, a *pos-seguinte* (1) e a *pos_seguinte* (2), usando a regra (2). Ao nível do pai do nó-rep, adicionamos 3 a *pos_seguinte* (1) e a *pos_seguinte* (2) usando a regra (1). Ao próximo nó-*cat*, adicionamos 4 a *pos-seguinte* (3) usando a regra (1). Aos próximos dois nós-*cat* adicionamos 5 a *pos_seguinte* (4) e 6 a *pos_seguinte* (5) usando a mesma regra. Isto completa a construção de *pos_seguinte*. A Fig. 3.42 sumariza *pos_seguinte*.

Nó	*pos_seguinte*
1	{1,2,3}
2	{1,2,3}
3	{4}
4	{5}
5	{6}
6	—

Fig. 3.42. A função *pos_seguinte*.

Podemos ilustrar a função *pos_seguinte* através da criação de um grafo orientado com um nó para cada posição e um lado direcionado do nó i para o nó j se j estiver em *pos_seguinte* (i). A Fig. 3.43 mostra esse grafo orientado para *pos-seguinte* da Fig. 3.42.

É interessante notar que esse diagrama viria a se tornar um AFN sem as transições-ϵ para a expressão regular em questão se:

1. fizéssemos com que todas as transições em *primeira_pos* da raiz fossem estados de partida;
2. rotulássemos cada lado orientado (i, j) pelo símbolo à posição j;
3. fizéssemos a posição associada com a # ser o único estado de aceitação.

Não deveria haver surpresa alguma se pudéssemos converter o grafo *pos_seguinte* num AFD utilizando a construção de subconjuntos. A construção inteira pode ser concretizada através das posições usando-se o algoritmo seguinte. □

Algoritmo 3.5. Construção de um AFD a partir de uma expressão regular r.

Entrada. Uma expressão regular r.

Saída. Um AFD D que reconhece $L(r)$.

Método.

1. Construir uma árvore sintática para a expressão regular expandida $(r)\ \#$, onde # é um marcador único de fim , atrelado ao fim de (r).

2. Construir as funções *anulável, primeira-pos, última-pos* e *pos-seguinte* através de uma travessia em profundidade (*depth first*) de T.

3. Construir *Estados-D*, o conjunto de estados de D, e *Dtran*, a tabela de transições para D, através do procedimento na Fig. 3.44. Os estados em *Estados-D* são conjuntos de posições: inicialmente, cada estado está "não-marcado", e um estado se torna "marcado" exatamente antes de considerarmos suas transições para fora. O esta-

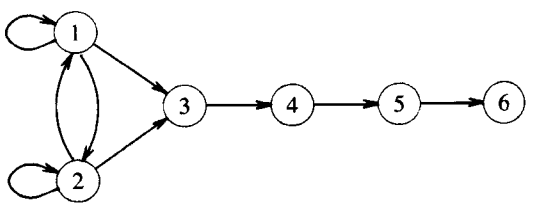


Fig. 3.43. Grafo orientado para a função *pos_seguinte*.

```
inicialmente, o único estado não-marcado em Estados-D é
    primeira_pos (raiz), onde raiz é a raiz da árvore sintática
    para (r) #;
enquanto existir um estado não-marcado T em Estados-D
    faça início
    marque T;
    para cada símbolo de entrada a faça início
        seja U o conjunto de posições que estão em
            pos_seguinte (p), para alguma posição p em T, tal
            que o símbolo à posição p seja a;
        se U não estiver vazio e não estiver em Estados-D
            então
            adicione U como um estado não-marcado a Estados-D;
        Dtran [T, a] := U
    fim
fim
```

Fig. 3.44. Construção de um AFD.

do de partida de D é *primeira_pos* (*raiz*), e os estados de aceitação são todos aqueles contendo a posição associada ao marcador de fim #. □

Exemplo 3.23. Vamos construir um AFD para a expressão regular $(a \mid b)^*abb$. A árvore sintática para $((a \mid b)^*abb)\#$, é mostrada na Fig. 3.39 (a). *Anulável* é verdadeira somente para o nó rotulado*. As funções *primeira_pos* e *última_pos* são mostradas na Fig. 3.41 e a *pos_seguinte* mostrada na Fig. 3.42.

A partir da Fig. 3.41, *primeira_pos* da raiz é $\{1, 2, 3\}$. Seja A este conjunto e consideremos o símbolo de entrada a. As posições 1 e 3 são para a e, então, façamos $B = pos_seguinte(1) \cup pos_seguinte(3) = \{1, 2, 3, 4\}$. Como este conjunto ainda não foi examinado, fazemos $Dtran[A, a] := B$.

Quando consideramos a entrada b, notamos que das posições em A, somente 2 está associada a b e precisamos então considerar o conjunto $pos_seguinte(2) = \{1, 2, 3\}$. Como este conjunto já foi examinado, não o adicionamos a *Estados-D*, mas adicionamos a transição $Dtran[A, b] := A$.

Continuamos, agora, com $B = \{1, 2, 3, 4\}$. Os estados e transições que finalmente obtemos são os mesmos que aqueles mostrados na Fig. 3.39 (b). □

Minimizando o Número de Estados de um AFD

Um importante resultado teórico é que cada conjunto regular é reconhecido por um AFD mínimo que é único, a menos dos nomes dos estados. Nesta seção, mostramos como construir este AFD mínimo pela redução do número de estados de um dado AFD sem afetar a linguagem reconhecida. Vamos supor que temos um AFD M com o conjunto de estados S e alfabeto de símbolos de entrada Σ. Assumimos que cada estado possui uma transição em cada símbolo de entrada. Se esse não for o caso, introduzimos um novo "estado morto" d, com transições de d para d em todas as entradas e adicionamos uma transição do estado s para d à entrada a se não existir transição de s em a.

Dizemos que a cadeia w *distingue* o estado s do estado t se, ao começarmos o AFD M pelo estado s e o alimentarmos com a entrada w, terminamos num estado de aceitação, mas, se começarmos pelo estado t e o alimentarmos com a entrada w, terminamos num estado de não-aceitação, ou vice-versa. Por exemplo, ϵ distingue qualquer estado de aceitação de qualquer estado de não-aceitação e, no AFD da Fig. 3.29, os estados A e B são distinguidos pela entrada bb, uma vez que A vai para o estado de não-aceitação C à entrada bb enquanto B vai para o estado de aceitação E à mesma entrada.

Nosso algoritmo para minimizar o número de estados de um AFD trabalha encontrando todos os grupos de estados que podem ser distinguidos por alguma cadeia de entrada. Os grupos de estados que não podem ser distinguidos são, então, unificados num único estado. O algoritmo trabalha mantendo e refinando uma partição do conjunto de estados que não foram distinguidos uns dos outros ainda e qualquer par de estados escolhidos a partir de diferentes grupos foram distinguidos por alguma entrada.

Inicialmente, a partição consiste em dois grupos: os estados de aceitação e os de não-aceitação. O passo fundamental é tomar algum grupo de estados, digamos $A = \{s_1, s_2, \ldots, s_k\}$ e algum símbolo de entrada a e verificar que transições os estados $s_1, s_2, \ldots, s_k$ possuem à entrada a. Se essas transições forem para estados que caem em dois ou mais grupos diferentes da partição corrente, temos então que partir A, de tal forma que as transições provenientes de cada subconjunto de A estejam todas confinadas a um único grupo da partição corrente. Suponhamos, por exemplo, que s_1 e s_2 se dirijam para os estados t_1 e t_2 à entrada a e que t_1 e t_2 estejam em diferentes grupos de partição. Precisamos, então, dividir A em pelo menos dois subconjuntos, de tal forma que um subconjunto contenha s_1 e o outro, s_2. Note-se que t_1 e t_2 são distinguidos por alguma cadeia w, de tal forma que s_1 e s_2 são distinguidos pela cadeia aw.

Repetimos o processo de subdivisão dos grupos da partição corrente até que não existam mais grupos que precisem ser subdivididos. Conquanto tenhamos justificado por que os estados que foram repartidos em grupos diferentes podem realmente ser distinguidos, não indicamos por que os estados que não foram espalhados em grupos diferentes são tidos como não distinguíveis por qualquer seqüência de entrada. Tal é verdade, no entanto, e deixamos a prova desse fato para o leitor interessado na teoria (ver, por exemplo, Hopcroft & Ullman [1979]). Também deixada para o leitor interessado é a prova do fato de que o AFD construído tomando-se um estado de cada grupo da partição final e jogando-se fora o estado morto e os estados não-atingíveis a partir do estado de partida possui tão poucos estados quanto qualquer AFD que aceite a mesma linguagem.

Algoritmo 3.6. Minimizando o número de estados de um AFD.

Entrada. Um AFD M com um conjunto de estados S, conjunto de entradas Σ, transições definidas para todos os estados, estado de partida s_0, e conjunto de estados de aceitação F.

Saída. Um AFD M' que aceita a mesma linguagem que M e que tem tão poucos estados quanto o possível.

Método.

1. Construir uma partição inicial Π do conjunto de estados com dois grupos: os estados de aceitação F e os de não-aceitação S-F.

2. Aplicar o procedimento da Fig. 3.45 a Π a fim de construir uma nova partição Π_{new}.

3. Se $\Pi_{new} = \Pi$, fazer $\Pi_{final} = \Pi$ e continuar com o passo (4). Em caso contrário, repetir o passo (2) com $\Pi := \Pi_{new}$.

4. Escolher um estado em cada grupo da partição Π_{final} como o *representante* daquele grupo. Os representantes serão os estados do AFD

```
para cada grupo G de Π faça início
    particionar G em subgrupos tais que dois estados s e t
        de G estejam no mesmo subgrupo se e somente se para
        todos os símbolos de entrada a, os estados s e t
        possuírem transições em a para estados do mesmo
        grupo de Π;
    /* na pior situação, um estado estará num subgrupo por
        si mesmo */
    substituir G em Π_new pelo conjunto de todos os
        subgrupos formados
fim
```

Fig. 3.45. Construção de Π_{new}.

reduzido M'. Seja s um estado representante e suponhamos que à entrada a haja uma transição em M de s para t. Seja r o representante do grupo de t's(r pode ser t). Então, M' tem uma transição de s para r em a. Seja o estado de partida de M' o representante do grupo contendo o estado de partida s_0 de M e sejam os estados de aceitação de M' os representantes que estão em F. Note-se que cada grupo de Π_{final} ou é constituído somente de estados em F ou não possui estados em F.

5. Se M' possuir um estado morto, isto é, um estado d que não seja de aceitação e possua transições para si mesmo em todos os símbolos de entrada, remover d de M'. Também remover quaisquer estados não atingíveis a partir de qualquer estado de partida. Quaisquer transições para d de outros estados se tornam indefinidas. □

Exemplo 3.24. Vamos reconsiderar o AFD representado na Fig. 3.29. A partição inicial Π consiste em dois grupos: (E), o grupo do estado de aceitação e $(ABCD)$, os estados de não-aceitação. Para construir Π_{new}, o algoritmo da Fig. 3.45 primeiro considera (E). Como este grupo consiste em um único estado, não pode ser dividido mais, e (E) é colocado em Π_{new}. O algoritmo então considera o grupo $(ABCD)$. À entrada a, cada um desses estados possui uma transição para B, e, conseqüentemente, todos esses estados poderiam permanecer num mesmo grupo, na medida em que a entrada a fosse considerada. À entrada b, entretanto, A, B e C vão para membros do grupo $(ABCD)$ de Π, enquanto D vai para E, um membro de outro grupo. Então, em Π_{new}, o grupo $(ABCD)$ precisa ser repartido em dois novos grupos, (ABC) e (D); Π_{new} é, conseqüentemente, (ABC) (D) (E).

Na passagem seguinte, do algoritmo da Fig. 3.45, de novo não temos divisões à entrada a, mas (ABC) precisa ser repartido em dois novos grupos (AC) e (B) pois, à entrada b, A e C têm, cada um, uma transição para C, enquanto B possui uma transição para D, um membro de um grupo de uma partição diferente da de C. O próximo valor de Π é (AC) (B) (D) (E).

Na passagem seguinte do algoritmo da Fig. 3.45 não podemos mais dividir quaisquer grupos constituídos de um único estado. A única possibilidade é a de subdividir (AC). Mas A e C vão para o mesmo estado B à entrada a e vão para o mesmo estado C à entrada b. Então, após essa passagem, $\Pi_{new} = \Pi$. Π_{final} é, então, (AC) (B) (D) (E).

Se escolhermos A como o representante do grupo (AC) e escolhermos B, D e E para os grupos com único estado, obtemos o autômato reduzido cuja tabela de transições é mostrada na Fig. 3.46. O estado A é o estado de partida e o estado E é o único estado de aceitação.

Por exemplo, no autômato reduzido, o estado E possui uma transição para o estado A à entrada b, dado que A é o representante do grupo de C e existe uma transição de E para C à entrada b no autômato original. Uma mudança similar teve lugar na interseção de A e entrada b; nos demais casos, todas as outras transições foram copiadas a partir da Fig. 3.29. Não existe estado morto na Fig. 3.46 e todos os estados são atingíveis a partir do estado de partida A. □

Estado	Símbolo de Entrada	
	a	b
A	B	A
B	B	D
D	B	E
E	B	A

Fig. 3.46. Tabela de transições para o AFD reduzido.

Minimização de Estados em Analisadores Léxicos

Para aplicar o procedimento de minimização para o AFD construído na Seção 3.7, precisamos começar o Algoritmo 3.5 com uma partição inicial que coloque em diferentes grupos todos os estados que indiquem *tokens* diferentes.

Exemplo 3.25. No caso do AFD da Fig. 3.37, a partição inicial agruparia 0137 com 7, uma vez que nenhum deles dá indicação de reconhecer um *token*; 8 e 58 também seriam agrupados, dado que indicam o *token* a^*b^+. Os outros estados estariam em grupos de si mesmos. Então, descobriríamos imediatamente que 0137 e 7 pertencem a grupos diferentes, pois vão para grupos distintos à entrada a. Igualmente, 8 e 58 não pertencem ao mesmo grupo por causa de suas transições à entrada b. Conseqüentemente, o AFD da Fig. 3.37 é o autômato de estado mínimo que realiza o trabalho. □

Métodos de Compressão de Tabelas

Como indicamos, existem várias formas de se implementar a função de transição de um autômato. O processo de análise léxica ocupa uma razoável parte do tempo do compilador, uma vez que é o único processo que precisa examinar um caractere de cada vez na entrada. Conseqüentemente, o analisador léxico deveria minimizar o número de operações que realiza por caractere de entrada. Se um AFD é usado para auxiliar a implementação de um analisador léxico, é desejável uma representação eficiente da função de transição. Um *array* bidimensional, indexado por estados e caracteres, providencia o acesso mais rápido, mas pode ocupar muito espaço (digamos várias centenas de estados por 128 caracteres). Um esquema mais compacto, porém mais lento, é o de usar uma lista ligada para armazenar as transições para fora de cada estado, com uma transição "*default*" ao final da lista. A transição que ocorre mais freqüentemente é uma escolha óbvia para o *default*.

Existe uma implementação mais sutil que combina o rápido acesso da representação em *array* com a compactação das estruturas de listas. Aqui usamos uma estrutura de dados que consiste em quatro *ar-*

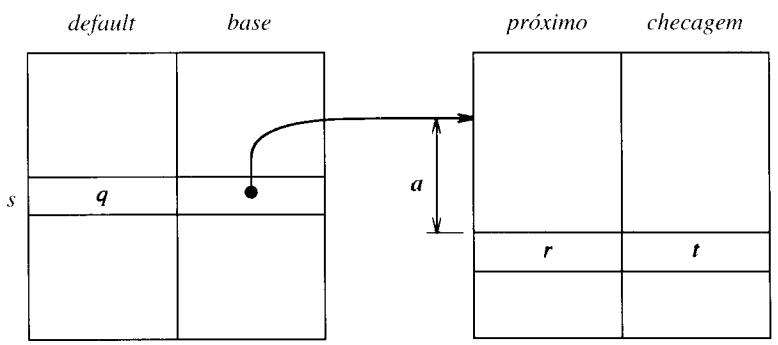


Fig. 3.47. Estrutura de dados para representar tabelas de transições.

rays indexados pelos números de estado, como delineado pela Fig. 3.47.[7] O *array base* é usado para determinar a localização-base das entradas de cada estado armazenado nos *arrays próximo* e *checagem*. O *array default* é usado para determinar uma localização base alternativa no caso da localização base corrente ser inválida.

Para computar *próximo_estado* (s, a), a transição para o estado s e símbolo de entrada a, consultamos primeiro o par de *arrays próximo* e *checagem*. Em particular, encontramos suas entradas para o estado s na localização l = *base* [s] + a, onde o a é tratado como inteiro. Tomamos *próximo* [l] como o próximo estado de s à entrada a se *checagem* [l] = s. Se *checagem* [l] ≠ s, determinamos q = *default* [s] e repetimos todo o procedimento recursivamente, usando q em lugar de s. O procedimento é como se segue:

procedimento *próximo_estado* (s, a);
  se *checagem* [*base* [s] + a] = s **então**
    retornar *próximo* [*base* [s] + a]
  senão
    retornar *próximo_estado* (*default* [s], a)

O uso projetado da estrutura da Fig. 3.47 é tornar os *arrays próximo e checagem* pequenos, aproveitando da vantagem das similaridades entre os estados. Por exemplo, o estado q, *default* para o estado s, poderia ser o que informa estarmos "trabalhando num identificador", tal como o estado 0 da Fig. 3.13. Talvez entremos em s após examinarmos o `th` como um prefixo da palavra-chave `then` e também como um prefixo de um identificador. Ao caractere de entrada e precisamos ir para um estado especial que se lembre de que já examinamos `the`, mas que em caso contrário o estado s se comporte como o estado q o faz. Então, fazemos s igual a *checagem* [*base* [s] + `e`] e *próximo* [*base* [s] + `e`] como o estado para o `the`.

Conquanto não estejamos capacitados a escolher os valores de *base*, de tal forma que nenhuma entrada de *próximo e checagem* permaneça sem uso, a experiência mostra que esta estratégia simples de se estabelecer *base* para o menor número tal que as entradas especiais possam ser preenchidas sem entrar em conflito com as entradas já existentes é razoavelmente boa e se utiliza pouco espaço além do mínimo possível.

Podemos encurtar *checagem* num *array* indexado por estados se o AFD possuir a propriedade de que os lados incidentes para cada estado t terão todos o mesmo rótulo a. Para implementar este esquema, fazemos *checagem* [t] = a e substituímos o teste na linha 2 do procedimento *próximo_estado* por

se *checagem* [*próximo*[*base* [s] + a]] = a **então**

EXERCÍCIOS

3.1 Qual é o alfabeto de entrada de cada uma das seguintes linguagens?
a) Pascal
b) C
c) Fortran 77
d) Ada
e) Lisp

3.2 Quais são as convenções relacionadas ao uso de espaços em cada uma das linguagens do Exercício 3.1?

3.3 Identifique os lexemas que compõem os *tokens* nos programas seguintes. Dê valores de atributo razoáveis para os *tokens*.

a) Pascal

```
function max (i, j : integer ) : inte-
ger ;
{retorna o maior dos inteiros entre i
e j }
begin
    if i > j then max := i
    else max := j
end;
```

b) C

```
int max (i, j) int i, j;
/* retorna o maior dos inteiros entre
i e j */
{
    return i>j? i : j;
}
```

c) Fortran 77

```
      FUNCTION MAX (I, J)
C     RETORNA O MAIOR DOS INTEIROS ENTRE I
      E J
          IF (I .GT. J) THEN
              MAX = I
          ELSE
              MAX = J
          END IF
      RETURN
```

3.4 Escreva um programa para a função `próximo caractere ()` da Seção 3.4 usando o esquema de buferização com sentinelas, descrito na Seção 3.2.

3.5 Numa cadeia de comprimento n, quantos dos seguintes elementos estarão lá?
a) prefixos
b) sufixos
c) subcadeias
d) prefixos próprios
e) subseqüências

***3.6** Descreva as linguagens denotadas pelas seguintes expressões regulares:
a) $0(0 \mid 1)^*0$
b) $((\epsilon \mid 0)1^*)^*$
c) $(0 \mid 1)^*0(0 \mid 1)(0 \mid 1)$
d) $0^* \; 10^* \; 10^* \; 10^*$
e) $(00 \mid 11)^* \; ((01 \mid 10)(00 \mid 11)^*(01 \mid 10)(00 \mid 11)^*)^*$

***3.7** Escreva definições regulares para as seguintes linguagens:
a) Todas as cadeias de letras que contenham as cinco vogais ordenadas.
b) Todas as cadeias de letras nas quais as letras estão em ordem lexicográfica ascendente.
c) Comentários consistindo em uma cadeia envolvida por /* e */ sem */ intervenientes a menos que figurem entre aspas.
*d) Todas as cadeias de dígitos sem nenhum dígito repetido.
e) Todas as cadeias de dígitos com no máximo um dígito repetido.
f) Todas as cadeias de zeros e uns com um número par de zeros e ímpar de uns.
g) O conjunto de movimentos de xadrez, tais como *P4R* ou *PBR* × *CD*.
h) Todas as cadeias de zeros e uns que não contenham a subcadeia 011.
i) Todas as subcadeias de zeros e uns que não contenham a subseqüência 011.

[7] Na prática, deveria haver um outro *array* indexado por s, fornecendo o padrão reconhecido, se algum, quando se entra no estado s. Essa informação é derivada dos estados do AFN que compõem o estado s do AFD.

3.8 Especifique a forma léxica das constantes numéricas das linguagens do Exercício 3.1.

3.9 Especifique a forma léxica dos identificadores e palavras-chave nas linguagens do Exercício 3.1.

3.10 As construções de expressões regulares permitidas por Lex estão listadas na Fig. 3.48 em ordem decrescente de precedência. Nessa tabela, *c* está no lugar de qualquer caractere único, *r* no de expressão regular e *s* para cadeia.

a) O significado especial dos símbolos de operação

```
\ " . ^ $ [ ] * + ? { } | /
```

precisa ser desligado se o símbolo do operador deve ser usado como um caractere de reconhecimento. Isto pode ser feito aspeando-se o caractere em dois estilos diferentes. A expressão "*s*" reconhece a cadeia *s* literalmente, uma vez que nenhuma aspa figure em *s*. Por exemplo, "**" reconhece a cadeia **. Poderíamos também ter reconhecido essa cadeia através de **. Note-se que um asterisco não aspeado (através de "ou de \) é uma instância do operador de fechamento Kleene. Escreva uma expressão regular que reconheça a cadeia "\.

b) Em Lex, uma classe de caracteres *complementados* é aquela na qual o primeiro símbolo é ^. Uma tal classe reconhece qualquer caractere que não esteja na mesma. Conseqüentemente, [^a] reconhece qualquer caractere que não seja um a, [^A-Za-z] reconhece qualquer caractere que não seja uma letra maiúscula ou minúscula, e assim por diante. Mostre que para cada definição regular com classes complementadas de caracteres existe uma expressão regular equivalente sem as classes complementadas de caracteres.

c) A expressão regular *r*{*m*, *n*} reconhece de *m* a *n* ocorrências do padrão *r*. Por exemplo, a{1,5} reconhece uma cadeia de um a cinco a's. Mostre que para expressão regular contendo operadores repetitivos, existe uma expressão regular equivalente sem os mesmos.

d) O operador ^ reconhece o fim mais à esquerda de uma linha. Este é o mesmo operador que introduz uma classe complementada de caracteres, mas o contexto no qual esse operador figurar irá sempre determinar um único significado para esse operador. O operador $ reconhece o fim mais à direita de uma linha. Por exemplo, ^[^aeiou]*$ reconhece qualquer linha que não contenha uma vogal minúscula. Para cada expressão regular contendo os operadores ^ e $, existe uma expressão regular equivalente sem esses operadores?

EXPRESSÃO	ELEMENTO(S) RECONHECIDO (S)	EXEMPLO
c	qualquer caractere *c* não-operador	a
c	caractere *c* literalmente	*
"*s*"	cadeia *s* literalmente	"**"
.	qualquer caractere que não avanço de linha	a.*b
^	início de linha	^abc
$	fim de linha	abc$
[*s*]	qualquer caractere em *s*	[abc]
[^*s*]	qualquer caractere fora de *s*	[^abc]
*r**	zero ou mais *r*'s	a*
r+	um ou mais *r*'s	a+
r?	zero ou um *r*	a?
r {*m*, *n*}	de *m* a *n* ocorrências de *r*	a{1,5}
$r_1 r_2$	r_1 e, em seguida, r_2	ab
$r_1 \mid r_2$	r_1 ou r_2	a|b
(*r*)	*r*	(a|b)
r_1 / r_2	r_1 quando seguida por r_2	abc/123

Fig. 3.48. Expressões regulares de Lex.

3.11 Escreva um programa Lex que copie um arquivo, substituindo cada seqüência não nula de espaços em branco por um único espaço.

3.12 Escreva um programa Lex que copie um programa Fortran, substituindo todas as instâncias de DOUBLE PRECISION por REAL.

3.13 Use sua especificação para palavras-chave e identificadores de Fortran 77 a partir do Exercício 3.9 para identificar os *tokens* dos seguintes enunciados:

```
IF(I) = TOKEN
IF(I) ASSIGN5TOKEN
IF(I) 10,20,30
IF(I) GOTO15
IF(I) THEN
```

Você pode escrever sua especificação para palavras-chave e identificadores em Lex?

3.14 No sistema UNIX, o comando de *shell* sh usa os operadores da Fig. 3.49 nas expressões de nomes de arquivos a fim de descrever conjuntos de nomes de arquivos. Por exemplo, a expressão de nome de arquivo *.O reconhece todos os nomes de arquivo terminados em .O; sort.? reconhece todos os que são da forma sort.*c*, onde *c* é qualquer caractere. As classes de caracteres podem ser abreviadas como em [a-z]. Mostre como as expressões de nomes de arquivo num *shell* podem ser representadas por expressões regulares.

3.15 Modifique o Algoritmo 3.1 a fim de que encontre o mais longo prefixo da entrada que seja aceito pelo AFD.

3.16 Construa um autômato finito não-determinístico para as seguintes expressões regulares usando o Algoritmo 3.3. Mostre a seqüência de movimentos feita por cada uma no processamento da cadeia de entrada *ababbab*.

a) $(a \mid b)^*$
b) $(a^* \mid b^*)^*$
c) $((\epsilon \mid a)\, b^*)^*$
d) $(a \mid b)^*\, abb\, (a \mid b)^*$

3.17 Converta os AFNs do Exercício 3.16 em AFDs usando o Algoritmo 3.2. Mostre as seqüências de movimentos feitos por cada um no processamento da cadeia de entrada *ababbab*.

3.18 Construa AFDs para as expressões regulares do Exercício 3.16 usando o Algoritmo 3.5. Compare os tamanhos dos AFDs com aqueles construídos no Exercício 3.17.

3.19 Construa um autômato finito determinístico para os diagramas de transições dos *tokens* da Fig. 3.10.

3.20 Amplie a tabela da Fig. 3.40 de forma a incluir os operadores de expressões regulares ? e +.

3.21 Minimize o número de estados dos AFDs do Exercício 3.18 usando o Algoritmo 3.6.

3.22 Podemos provar que duas expressões regulares são equivalentes mostrando que seus AFDs mínimos são os mesmos, a menos dos nomes de estados. Usando esta técnica, mostre que as seguintes expressões regulares são equivalentes.

a) $(a \mid b)^*$
b) $(a^* \mid b^*)^*$
c) $((\epsilon \mid a)\, b^*)^*$

EXPRESSÃO	ELEMENTOS RECONHECIDOS	EXEMPLO
'*s*'	cadeia *s* literalmente	'\'
c	caractere *c* literalmente	\'
*	qualquer cadeia	*.o
?	qualquer caractere	sort 1.?
[*s*]	qualquer caractere em *s*	sort. [cso]

Fig. 3.49. Expressões de nomes de arquivo no programa sh.

3.23 Construa um AFD mínimo para as seguintes expressões regulares.

a) $(a \mid b)^* a (a \mid b)$

b) $(a \mid b)^* a (a \mid b) (a \mid b)$

c) $(a \mid b)^* a (a \mid b) (a \mid b) (a \mid b)$

**d) Prove que qualquer autômato finito determinístico para a expressão regular $(a \mid b)^* a (a \mid b) (a \mid b) \ldots (a \mid b)$, onde existam $n - 1$ $(a \mid b)$'s ao fim, precisa ter pelo menos 2^n estados.

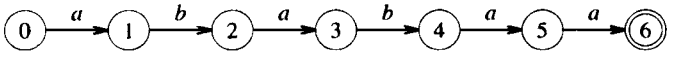
3.24 Construa uma representação da Fig. 3.47 para a tabela de transições do Exercício 3.19. Obtenha os estados *defaults* e tente os dois seguintes métodos para construir o *array próximo* e comparar as quantidades de espaço usadas:

a) Começando pelos estados mais densos (aqueles com o maior número de entradas diferindo de seus estados *default*) coloque primeiro as entradas para aqueles estados no *array próximo*.

b) Coloque as entradas para os estados no *array próximo* numa ordem randômica.

3.25 Uma variante do esquema de compressão de tabela da Seção 3.9 seria evitar o procedimento recursivo *próximo_estado* usando uma localização fixa para cada localização *default* de cada estado. Construa a representação da Fig. 3.47 para a tabela de transições do Exercício 3.19 usando esta técnica não recursiva. Compare as exigências de espaço com aquelas do Exercício 3.24.

3.26 Seja $b_1 b_2 \ldots b_m$ uma cadeia padrão, chamada de *palavra-chave*. Um *trie* para uma palavra-chave é um diagrama de transições com $m + 1$ estados, no qual cada estado corresponde a um prefixo de palavra-chave. Para $1 \leq s \leq m$, existe uma transição do estado $s - 1$ para o estado s no símbolo b_s. Os estados inicial e final correspondem à cadeia vazia e à palavra-chave completa, respectivamente. O *trie* para a palavra-chave *ababaa* é:



Definimos agora uma *função de falha* f em cada estado do diagrama de transições, exceto para o estado de partida. Suponhamos que os estados s e t representem os prefixos u e v da palavra-chave. Definimos, então, $f(s) = t$ se e somente v for o mais longo sufixo próprio de u que também seja prefixo de uma palavra-chave. A função de falha f para o *trie* acima é

s	1	2	3	4	5	6
$f(s)$	0	0	1	2	3	1

Por exemplo, os estados 3 e 1 representam os prefixos *aba* e *a* da palavra-chave *ababaa*. $f(3) = 1$ pois *a* é o sufixo próprio mais longo de *aba* que é prefixo da palavra-chave.

a) Construa a função de falha para a palavra-chave *abababaab*.

```
/* computar a função de falha f para b1...bm /*
t := 0; f(1) := 0;
para s := 1 até m — 1 faça início
        enquanto t > 0 e b_{s+1} ≠ b_{t+1} faça t := f(t);
        se b_{s+1} = b_{t+1} então início t := t + 1; f(s+1) := t fim;
        se não f(s + 1) := 0
fim
```

Fig. 3.50. Algoritmo para computar a função de falha para o Exercício 3.26.

```
/* é a1... an uma subcadeia de b1 ... bm ? */
s := 0;
para i := 1 até n faça início
        enquanto s > 0 e a_i ≠ b_{s+1} faça s := f(s);
        se a_i = b_{s+1} então s := s + 1
        se s = m então retornar "sim"
fim;
retornar "não"
```

Fig. 3.51. Algoritmo KMP.

*b) Sejam os estados do *trie* $0, 1, \ldots, m$, com 0 sendo o estado de partida. Mostre que o algoritmo da Fig. 3.50 computa corretamente a função de falha.

*c) Mostre que na execução global do algoritmo da Fig. 3.50, o enunciado $t = f(t)$ no laço mais interno é executado no máximo m vezes.

*d) Mostre que o algoritmo executa num tempo $O(m)$.

3.27 O Algoritmo KMP da Fig. 3.51 usa a função de falha f construída no Exercício 3.26 para determinar se uma palavra-chave $b_1 b_2 \ldots b_m$ é uma subcadeia de uma cadeia-alvo $a_1 \ldots a_n$. Os estados para o *trie* $b_1 b_2 \ldots b_m$ são numerados de 0 a m, como no Exercício 3.26 (b).

a) Aplique o Algoritmo KMP para determinar se *ababaa* é uma subcadeia de *abababaab*.

*b) Prove que o Algoritmo KMP retorna "sim" se e somente se $b_1 \ldots b_m$ for uma subcadeia de $a_1 \ldots a_n$.

*c) Mostre que o Algoritmo KMP roda num tempo $O(m + n)$.

d) Dada uma palavra-chave y, mostre que a função de falha pode ser usada para construir num tempo $O(|y|)$ um AFD com $|y| + 1$ estados para a expressão regular . y.* onde os pontos estão no lugar de qualquer caractere de entrada.

****3.28** Defina-se um *período* de uma cadeia s como sendo um inteiro p tal que s possa ser expressa como $(uv)^k u$, para algum $k \geq 0$, onde $|uv| = p$ e v não seja a cadeia vazia. Por exemplo, 2 e 4 são períodos da cadeia *abababa*.

a) Mostre que p é um período de uma cadeia s se e somente se $st = us$ para algumas cadeias t e u de comprimento p.

b) Mostre que se p e q são períodos de uma cadeia s e que se $p + q \leq |s| + \text{mdc}(p, q)$, então o mdc (p, q) é um período de s, onde mdc (p, q) é o maior divisor comum de p e q.

c) Seja $sp(s_i)$ o menor período do prefixo de comprimento i de uma cadeia s. Mostre que a função de falha f tem a propriedade que $f(j) = j - sp(s_{j-1})$.

***3.29** Seja o *menor prefixo repetitivo* de uma cadeia s o mais curto prefixo u de s tal que $s = u^k$, para algum $k \geq 1$. Por exemplo, *ab* é o mais curto período repetitivo de *abababab* e *aba* é o mais curto prefixo de *aba*. Construa um algoritmo que encontre o mais curto período repetitivo de uma cadeia s num tempo $O(|s|)$. *Sugestão.* Use a função de falha do Exercício 3.26.

3.30 *Uma cadeia de Fibonacci* é definida como se segue:

$$s_1 = b$$
$$s_2 = a$$
$$s_k = s_{k-1} s_{k-2}, \text{ para } k > 2.$$

Por exemplo, $s_3 = ab$, $s_4 = aba$ e $s_5 = abaab$.

a) Qual é o comprimento de s_n?

**b) Qual é o menor período de s_n?

c) Construa a função de falha para s_6.

*d) Usando a função, mostre que a função de falha para s_n pode ser expressa por $f(j) = j - s_{k-1}$, onde k é tal que $s_k \leq j + 1 < |s_{k+1}|$, para $1 \leq j \leq |s_n|$.

e) Aplique o Algoritmo KMP para determinar se s_6 é uma subcadeia da cadeia-alvo s_7.

f) Construa um AFD para a expressão regular $.^* s_6 .^*$.

**g) No algoritmo KMP, qual é o número máximo de aplicações consecutivas da função de falha executadas ao se determinar se s_k é uma subcadeia da cadeia-alvo s_{k+1}?

3.31 Podemos estender os conceitos de *trie* e de função de falha, do Exercício 3.26, de uma palavra-chave para um conjunto de palavras-chave como se segue. Cada estado no *trie* corresponde a um prefixo de uma ou mais palavras-chave. O símbolo de partida corresponde à cadeia vazia e um estado que corresponda a uma palavra-chave completa é um estado final. Estados adicionais podem ser tornados estados finais durante o cômputo da função de falha. O diagrama de transições para o conjunto de palavras-chave `{he, she, his, hers}` é mostrado na Fig. 3.52.

Para o *trie*, definimos uma *função de transição* g que mapeia pares estado-símbolo para estados tais que $g(s, b_{j-1}) = s'$ se o estado s corresponde a um prefixo $b_1b_2 \ldots b_j$ de alguma palavra-chave e s' corresponde ao prefixo $b_1b_2 \ldots b_jb_{j-1}$. Se s_0 é o estado de partida, definimos $g(s_0, a) = s_0$ para todos os símbolos de entrada a que não são símbolo inicial de qualquer palavra-chave. Fazemos então $g(s, a) = falha$ para qualquer transição não definida. Note-se que não existem transições *falha* para o estado de partida.

Suponhamos que os estados s e t representem os prefixos u e v de algumas palavras-chave. Definimos, então, $f(s) = t$ se e somente se v é o prefixo próprio mais longo de u que também seja prefixo de alguma palavra-chave. A função de falha f para o diagrama de transições abaixo é

s	1	2	3	4	5	6	7	8	9
$f(s)$	0	0	0	1	2	0	3	0	3

Por exemplo, os estados 4 e 1 representam os prefixos `sh` e `h`. $f(4) = 1$ porque `h` é o prefixo próprio mais longo de alguma palavra-chave. A função de falha f pode ser computada para os estados de profundidade crescente usando-se o algoritmo da Fig. 3.53. A profundidade de um estado é a sua distância a partir do estado de partida.

Note-se que, como $g(s_0, c) \neq falha$ para qualquer caractere c, o laço enquanto na Fig. 3.53 está garantido terminar. Após fazer $g(t, a)$ igual $f(s')$, se $g(t, a)$ for um estado final, fazemos também s' um estado final, se já não o for.

a) Construir a função de falha para o conjunto de palavras-chave $\{aaa, abaaa, ababaaa\}$.

*b) Mostre que o algoritmo da Fig. 3.53 computa corretamente a função de falha.

*c) Mostre que a função de falha pode ser computada num tempo proporcional à soma dos comprimentos das palavras-chave.

3.32 Seja g a função de transição e f a função de falha do Exercício 3.31 para um conjunto de palavras-chave $K = \{y_1, y_2, \ldots, y_k\}$. O Algoritmo AC na Fig. 3.54 usa g e f para determinar se uma cadeia-alvo $a_1 \ldots a_n$ contém uma subcadeia que seja uma palavra-chave. O estado s_0 é o estado de partida do diagrama de transições para K e, F, é o conjunto de estados finais.

a) Aplique o Algoritmo AC à cadeia de entrada `ushers` usando as funções de transição e de falha do Exercício 3.31.

*b) Prove que o Algoritmo AC retorna "sim" se e somente se alguma palavra-chave y_i for uma subcadeia de $a_1 \ldots a_n$.

*c) Mostre que o Algoritmo AC realiza no máximo $2n$ transições ao processar uma cadeia de entrada de comprimento n.

*d) Mostre que a partir de um diagrama de transições e da função de falha para um conjunto de palavras-chave $\{y_1, y_2, \ldots, y_k\}$, um AFD com no máximo $\sum_{i=1}^{k} |y_i| + 1$ estados pode ser construído num tempo linear para a expressão regular $.*(y_1 | y_2 | \ldots | y_k).*$.

e) Modifique o Algoritmo AC para imprimir cada palavra-chave encontrada na cadeia-alvo.

3.33 Use o Algoritmo do Exercício 3.32 para construir um analisador léxico para as palavras-chave em Pascal.

3.34 Defina $scl(x, y)$ como a *subseqüência comum mais longa* de duas cadeias x e y, como a cadeia que seja uma subseqüência de ambos, x e y, e seja tão longa quanto qualquer subseqüência. Por exemplo, `tie` é a mais longa subseqüência de `striped` e `tiger`. Defina-se $d(x, y)$, a *distância* entre x e y, como sendo o número mínimo de inserções e remoções requeridas para transformar x em y. Por exemplo, d `(striped, tiger)` $= 6$.

a) Mostre que para quaisquer duas cadeias x e y, a distância entre x e y e o comprimento da subseqüência comum mais longa de ambas estão relacionados por $d(x, y) = |x| + |y| - (2 * |scl(x, y)|)$.

*b) Escreva um algoritmo que tome duas cadeias x e y como entrada e produza a subseqüência comum mais longa de x e y como saída.

3.35 Defina $e(x, y)$ como a *distância de edição* entre duas cadeias x e y, como sendo o número mínimo de inserções, remoções e substituições que são requeridas para transformar x em y. Sejam $x = a_1 \ldots a_m$ e $y = b_1 \ldots b_n$. $e(x, y)$ pode ser computada por um algoritmo de programação dinâmica usando um *array* de distâncias $d[0..m, 0..n]$, no qual $d[i, j]$ é a distância entre $a_1 \ldots a_i$

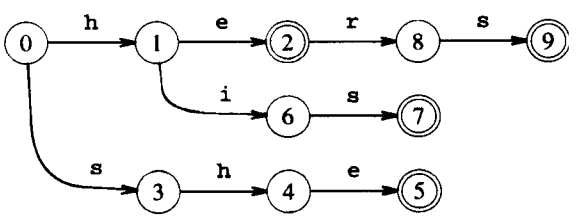


Fig. 3.52. *Trie* para as palavras-chave `{he, she, his, hers}`.

```
para cada estado s de profundidade 1 faça
    f(s) := s0;
para cada profundidade d ≥ 1 faça
    para cada estado sd de profundidade d e caractere a tal
        que g(sd, a) = s' faça início
        s := f(sd);
        enquanto g(s, a) = falha faça s := f(s);
        f(s') := g(s, a);
fim
```

Fig. 3.53. Algoritmo para computar a função de falha para um `trie` de palavras-chave.

```
/* a1 ... an contém uma palavra chave como subcadeia */
s := s0;
para i := 1 até n faça início
    enquanto g(s, ai) = falha faça s = f(s);
    s := g(s, ai);
    se s estiver em F então retornar "sim"
fim;
retornar "não"
```

Fig. 3.54. Algoritmo AC.

```
para i := 0 até m faça d[i, 0] := i;
para j := 1 até n faça d [0, j] := j;
para i := 1 até m faça
    para j := 1 até n faça
        D [i, j] := min (d [i − 1, j −1] + repl (a_i, b_j),
                         d [i − 1, j] + 1,
                         d [i, j − 1] + 1)
```

Fig. 3.55. Algoritmo para computar a distância de edição entre duas cadeias.

e $b_1 ... b_j$. O algoritmo da Fig. 3.55 pode ser usado para computar a matriz d. A função *repl* é tão-somente o custo da substituição de um caractere: $repl\ (a_i, b_j) = 0$ se $a_i = b_j$, 1 em caso contrário.

a) Qual é a relação entre a medida de distância do Exercício 3.34 e a distância de edição?
b) Use o algoritmo na Fig. 3.55 para computar a distância de edição entre *ababb* e *babaaa*.
c) Construa um algoritmo que imprima a seqüência mínima de transformações de edição requeridas para transformar x em y.

3.36 Dê um algoritmo que tome como entrada uma cadeia x e uma expressão regular r e produza como saída uma cadeia y em $L(r)$, tal que $d(x, y)$ seja tão pequena quanto possível, onde d é a função de distância do Exercício 3.34.

EXERCÍCIOS DE PROGRAMAÇÃO

P3.1 Escreva um analisador léxico em Pascal ou C para os *tokens* mostrados na Fig. 3.10.

P3.2 Escreva uma especificação para os *tokens* de Pascal e a partir da mesma construa diagramas de transições. Use os últimos para implementar uma analisador léxico para Pascal numa linguagem como C ou Pascal.

P3.3 Complete o programa Lex na Fig. 3.18. Compare o tamanho e velocidade do analisador léxico resultante produzido por Lex e com o programa escrito no Exercício P3.1.

P3.4 Escreva uma especificação Lex para os *tokens* em Pascal e use o compilador Lex para construir um analisador léxico para Pascal.

P3.5 Escreva um programa que tome como entrada uma expressão regular e o nome de um arquivo e produza como saída todas as linhas do arquivo que contenha uma subcadeia denotada pela expressão regular.

P3.6 Adicione um esquema de recuperação de erros ao programa Lex na Fig. 3.18 a fim de capacitá-lo a continuar a procurar por *tokens* na presença de erros.

P3.7 Programe um analisador léxico a partir do AFD construído no Exercício 3.18 e compare este analisador léxico com aquele construído nos Exercícios P3.1 e P3.3.

P3.8 Construa uma ferramenta que produza um analisador léxico a partir da descrição, sob a forma de expressões regulares, de um conjunto de *tokens*.

NOTAS BIBLIOGRÁFICAS

As restrições impostas nos aspectos léxicos de uma linguagem são freqüentemente determinadas pelo ambiente no qual a linguagem foi criada. Quando Fortran foi projetada em 1954, os cartões perfurados eram um meio comum de entrada. Os espaços eram ignorados parcialmente em Fortran porque os digitadores, que preparavam os cartões a partir de anotações manuscritas, tendiam a errar na contagem dos espaços (Backus [1981]). A separação, em Algol 58, da representação de *hardware* e da linguagem de referência era um compromisso atingido após um membro do comitê de projeto ter insistido "Não! Jamais irei confundir o uso de um ponto indicando um final de frase com o de um ponto decimal"[2]. (Wegstein [1981]).

Knuth [1973a] apresenta técnicas adicionais para buferização da entrada. Feldman [1979b] discute as dificuldades práticas do reconhecimento de *tokens* em Fortran 77.

As expressões regulares foram primeiro estudadas por Kleene [1956], que estava interessado na descrição dos eventos que poderiam ser representados pelo modelo de autômato finito da atividade nervosa, de McCulloch e Pitts [1943]. A minimização dos autômatos finitos foi primeiramente estudada por Huffman [1954] e Moore [1956]. A equivalência entre os autômatos determinísticos e não-determinísticos bem como as suas capacidades em reconhecer linguagens foi mostrada por Rabin e Scott [1959]. McNaughton e Yamada [1960] descrevem um algoritmo para construir AFDs diretamente a partir de uma expressão regular. Mais a respeito da teoria dos autômatos pode ser encontrado em Hopcroft e Ullman [1979].

Foi rapidamente compreendido que as ferramentas para construir analisadores léxicos a partir de expressões regulares seriam úteis na implementação de compiladores. Johnson *et al.* [1968] discutem um sistema precursor. Lex, a linguagem discutida neste capítulo, é devida a Lesk [1975] e tem sido usada para construir analisadores léxicos para muitos compiladores utilizando o sistema UNIX. O esquema de implementação compacta das tabelas de transições na Seção 3.9 é devido a S.C. Johnson, que primeiro o usou na implementação do gerador de *parsers* Yacc (Johnson [1975]). Outros esquemas de compressão de tabelas são discutidos e avaliados em Dencker, Dürre e Heuft [1984].

O problema da implementação compactada das tabelas de transições foi teoricamente estudado num nível global por Tarjan e Yao [1979] e por Fredman, Komlós e Szemerédi [1984]. Cormack, Horspool e Kaiserwerth [1985] apresentam um algoritmo perfeito de *hashing* baseado nesse trabalho.

As expressões regulares e os autômatos finitos têm sido usados em muitas aplicações além da compilação. Muitos editores de texto usam expressões regulares para pesquisas de contexto. Thompson [1968], por exemplo, descreve a construção de um AFN a partir de uma expressão regular (Algoritmo 3.3) no contexto do editor de texto QED. O sistema UNIX possui três programas de propósito geral de pesquisa em expressões regulares: `grep`, `egrep` e `fgrep`. `grep` não permite a união ou os parênteses para agrupar suas expressões regulares, mas permite uma forma limitada de referenciamento posterior (isto é, referenciarmos posteriormente um elemento previamente definido), como em Snobol. `grep` emprega os Algoritmos 3.3 e 3.4 para pesquisar seus padrões sob a forma de expressões regulares. As expressões regulares em `egrep` são similares àquelas em Lex, exceto para iteração e o *lookahead*. `egrep` usa um AFD com uma construção preguiçosa de estados a fim de pesquisar seus padrões sob a forma de expressões regulares, como delineado na Seção 3.7. `fgrep` busca pelos padrões que consistem em conjuntos de palavras-chave usando o algoritmo em Aho e Corasick [1975], o qual é discutido nos Exercícios 3.31 e 3.32. Aho [1980] discute o desempenho relativo desses programas.

As expressões regulares têm sido amplamente usadas em sistemas de recuperação de textos, em linguagens de interrogação de bancos de dados e linguagens de processamento de arquivos, como AWK (Aho, Kernigham e Weinberger [1979]). Jarvis [1976] usou expressões regulares para descrever as imperfeições dos circuitos impressos. Cherry [1982] usou o algoritmo de reconhecimento de palavras-chave do Exercício 3.32 para pesquisar palavras mal escritas em manuscritos.

O algoritmo de reconhecimento de padrões para cadeias dos Exercícios 3.26 e 3.27 é de Knuth, Morris e Pratt [1977]. Esse artigo

[2] A frase original foi *"No! I will never use a period for a decimal point"*. O que tal membro frisava era não haver necessidade em se diferenciar o caractere usado para um ponto decimal do de fim de frase, uma vez que não haveria possibilidade de se confundir um com o outro, dado o contexto; internamente, o compilador faria a distinção, fazendo com que a representação de *hardware* e a da linguagem pudessem ser diferentes. (N. do T.)

também contém uma boa discussão de frases em cadeias. Outro algoritmo eficiente para reconhecimento de cadeias foi inventado por Boyer e Moore [1977], que mostraram que um reconhecimento de uma subcadeia pode usualmente ser determinado sem se ter que examinar todos os caracteres da cadeia-alvo. O *hashing* também foi provado como uma técnica efetiva para o reconhecimento de padrões de cadeias (Harrison [1971]).

A noção de subseqüência comum mais longa, discutida no Exercício 3.34, foi usada no projeto do programa de comparação do sistema de arquivos do UNIX `diff` (Hunt e McIlroy [1976]). Um algoritmo eficiente e prático para o cômputo das subseqüências comuns mais longas é descrito em Hunt e Szymansnki [1977]. O algoritmo para computar as distâncias mínimas de edição do Exercício 3.35 está em Wagner e Fischer [1974]. Wagner [1974] contém uma solução para o Exercício 3.36. Sankof e Kruskal [1983] contém uma discussão fascinante sobre o amplo leque de aplicações dos algoritmos de reconhecimento de distância mínima para o estudo de padrões em seqüências genéticas a problemas no processamento da linguagem falada.

CAPÍTULO 4

ANÁLISE SINTÁTICA

Cada linguagem de programação possui as regras que descrevem a estrutura sintática dos programas bem-formados. Em Pascal, por exemplo, um programa é constituído por blocos, um bloco por comandos, um comando por expressões, uma expressão por *tokens* e assim por diante. A sintaxe das construções de uma linguagem de programação pode ser descrita pelas gramáticas livres de contexto ou pela notação BNF (Forma de Backus-Naur), introduzidas na seção 2.2. As gramáticas oferecem vantagens significativas tanto para os projetistas de linguagens quanto para os escritores de compiladores.

- Uma gramática oferece, para uma linguagem de programação, uma especificação sintática precisa e fácil de entender.
- Para certas classes de gramáticas, podemos construir automaticamente um analisador sintático* que determine se um programa-fonte está sintaticamente bem-formado. Como benefício adicional, o processo de construção do analisador pode revelar ambigüidades sintáticas bem como outras construções difíceis de se analisar gramaticalmente, as quais poderiam, de outra forma, seguir indetectadas na fase de projeto inicial de uma linguagem e de seu compilador.
- Uma gramática propriamente projetada implica uma estrutura de linguagem de programação útil à tradução correta de programas-fonte em códigos-objeto e também à detecção de erros. Existem ferramentas disponíveis para a conversão de descrições de traduções, baseadas em gramáticas, em programas operativos.
- As linguagens evoluíram ao longo de um certo período de tempo, adquirindo novas construções e realizando tarefas adicionais. Essas novas construções podem ser mais facilmente incluídas quando existe uma implementação baseada numa descrição gramatical da linguagem.

O núcleo deste capítulo está devotado aos métodos de análise sintática que são tipicamente usados nos compiladores. Apresentamos primeiramente os conceitos básicos, em seguida as técnicas adequadas à implementação manual e finalmente os algoritmos usados nas ferramentas automatizadas. Como os programas podem conter erros sintáticos, estendemos os métodos de análise, de forma que se recuperem dos erros que ocorrem mais comumente.

4.1 O PAPEL DO ANALISADOR SINTÁTICO

Em nosso modelo de compilador, o analisador sintático obtém uma cadeia de *tokens* proveniente do analisador léxico, como mostrado na Fig. 4.1, e verifica se a mesma pode ser gerada pela gramática da linguagem-fonte. Esperamos que o analisador sintático relate quaisquer erros de sintaxe de uma forma inteligível. Deve também se recuperar dos erros que ocorram mais comumente, a fim de poder continuar processando o resto de sua entrada.

Existem três tipos gerais de analisadores sintáticos. Os métodos universais de análise sintática, tais como o algoritmo de Cocke-Younger-Kasami e o de Earley, podem tratar qualquer gramática (ver as notas bibliográficas). Esses métodos, entretanto, são muito ineficientes para se usar num compilador de produção. Os métodos mais comumente usados nos compiladores são classificados como *top-down* ou *bottom-up*. Como indicado por seus nomes, os analisadores sintáticos *top-down* constroem árvores do topo (raiz) para o fundo (folhas), enquanto que os *bottom-up* começam pelas folhas e trabalham árvore acima até a raiz. Em ambos os casos, a entrada é varrida da esquerda para a direita, um símbolo de cada vez.

Os métodos de análise sintática mais eficientes, tanto *top-down* quanto *bottom-up*, trabalham somente em determinadas subclasses de gramáticas, mas várias dessas subclasses, como as das gramáticas LL e LR, são suficientemente expressivas para descrever a maioria das construções sintáticas das linguagens de programação. Os analisadores implementados manualmente trabalham freqüentemente com gramáticas LL; por exemplo, a abordagem da Seção 2.4 constrói analisadores

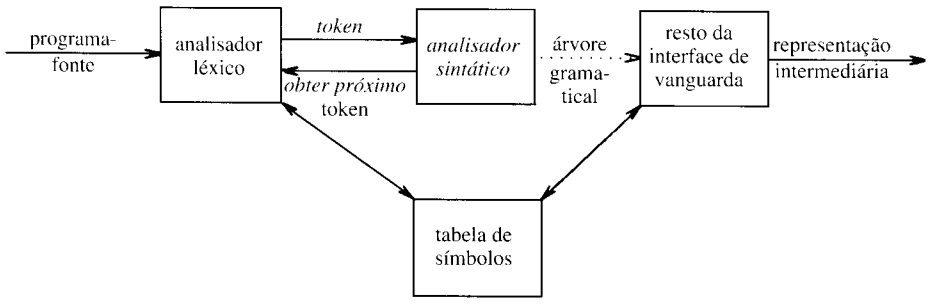


Fig. 4.1. Posição de um analisador sintático num modelo de compilador.

*Usaremos os termos analisador sintático e analisador gramatical com acepção idêntica, equivalendo ao termo original em inglês *parser*. (N. do T.)

sintáticos para gramáticas LL. Os da classe mais ampla das gramáticas LR são usualmente construídos através de ferramentas automatizadas.

Neste capítulo, assumimos que a saída de um analisador sintático seja alguma representação da árvore gramatical para o fluxo de *tokens* produzido pelo analisador léxico. Na prática, existe um certo número de tarefas que poderiam ser conduzidas durante a análise sintática, tais como coletar informações sobre os vários *tokens* na tabela de símbolos, realizar a verificação de tipos e outras formas de análise semântica, assim como gerar o código intermediário, conforme o Capítulo 2. Juntamos todos esses tipos de atividades na caixa "resto da *interface* de vanguarda" da Fig. 4.1. Iremos discutir essas atividades em detalhes nos próximos três capítulos.

No resto desta seção, consideramos a natureza dos erros sintáticos e as estratégias gerais para a sua recuperação. Duas dessas estratégias, chamadas "modalidade do desespero" e recuperação em nível de frase, são discutidas mais pormenorizadamente junto com os métodos individuais de análise sintática. A implementação de cada estratégia requer o julgamento do produtor do compilador, mas daremos algumas diretrizes gerais relacionadas a essas abordagens.

Tratamento dos Erros de Sintaxe

Se um compilador tivesse que processar somente programas corretos, seu projeto e sua implementação seriam grandemente simplificados. Mas os programadores freqüentemente escrevem programas incorretos, e um bom compilador deveria assistir o programador na identificação e localização de erros. É gritante que, apesar dos erros serem lugar-comum, poucas linguagens sejam projetadas tendo-se o tratamento de erros em mente. Nossa civilização seria radicalmente diferente se as linguagens faladas tivessem as mesmas exigências de correção sintática que as das linguagens de computadores. A maioria das especificações das linguagens de programação não descreve como um compilador deveria responder aos erros; tal tarefa é deixada para o projetista do compilador. O planejamento do tratamento de erros exatamente desde o início poderia tanto simplificar a estrutura de um compilador quanto melhorar sua resposta aos erros.

Sabemos que os programas podem conter erros em muitos níveis diferentes. Por exemplo, os erros podem ser:

- léxicos, tais como errar a grafia de um identificador, palavra-chave ou operador
- sintáticos, tais como uma expressão aritmética com parênteses não-balanceados
- semânticos, tais como um operador aplicado a um operando incompatível
- lógicos, tais como uma chamada infinitamente recursiva

Freqüentemente, boa parte da detecção e recuperação de erros num compilador gira em torno da fase de análise sintática. Isto porque os erros ou são sintáticos por natureza ou são expostos quando o fluxo de *tokens* proveniente do analisador léxico desobedece às regras gramaticais que definem a linguagem de programação. Outra razão está na precisão dos modernos métodos de análise sintática: podem detectar muito eficientemente a presença de erros sintáticos num programa. Detectar precisamente a presença de erros semânticos ou lógicos em tempo de compilação é uma tarefa muito mais difícil. Nesta seção, apresentamos umas poucas técnicas básicas para a recuperação de erros sintáticos; suas implementações são discutidas em conjunto com os métodos de análise sintática deste capítulo.

O tratador de erros num analisador sintático possui metas simples de serem estabelecidas:

- Deve relatar a presença de erros clara e acuradamente.
- Deve se recuperar de cada erro suficientemente rápido a fim de ser capaz de detectar erros subseqüentes.
- Não deve retardar significativamente o processamento de programas corretos.

A realização efetiva dessas metas apresenta desafios difíceis.

Felizmente, os erros comuns são simples e freqüentemente basta um mecanismo de tratamento de erros relativamente direto. Em alguns casos, entretanto, um erro pode ter ocorrido muito antes de sua presença ter sido detectada e sua natureza precisa pode ser muito difícil de ser deduzida. Em casos difíceis, o tratador de erros pode ter que adivinhar o que o programador tinha em mente quando o programa foi escrito.

Vários métodos de análise sintática, tais como os métodos LL e LR, detectam os erros tão cedo quanto possível. Mais precisamente, possuem a *propriedade do prefixo viável*, significando que detectam que um erro ocorreu tão logo tenham examinado um prefixo da entrada que não seja o de qualquer cadeia da linguagem.

Exemplo 4.1. A fim de se ter uma apreciação dos tipos de erros que ocorrem na prática, vamos examinar os erros que Ripley e Druseikis [1978] encontraram numa amostra de programa Pascal de estudantes.

Ripley e Druseikis descobriram que os erros não ocorrem com tanta freqüência; 60% dos programas compilados estavam semântica e sintaticamente corretos. Mesmo quando os erros ocorriam de fato, eram um tanto dispersos; 80% dos enunciados contendo erros possuíam apenas um, 13% dois. Finalmente, a maioria constituía-se de erros triviais; 90% eram erros em um único *token*.

Muitos dos erros poderiam ser classificados simplificadamente: 60% eram erros de pontuação, 20% de operadores e operandos, 15% de palavras-chave e os 5% restantes de outros tipos. O grosso dos erros de pontuação girava em torno do uso incorreto do ponto-e-vírgula.

Para alguns erros concretos, consideremos o seguinte programa Pascal.

```
(1)  program prmax ( input, output ) ;
(2)  var
(3)     x, y: integer;

(4)  function max (i: integer; j: integer) :
        integer;
(5)  { return maximum of integers i and j }
(6)  begin
(7)     if i > j then max := i
(8)     else max := j
(9)  end;

(10) begin
(11)    readln (x, y);
(12)    writeln (max (x, y))
(13) end.
```

Um erro comum de pontuação é o de se usar uma vírgula em lugar de ponto-e-vírgula na lista de argumentos de uma declaração de função (por exemplo, usar uma vírgula em lugar do primeiro ponto-e-vírgula à linha (4)); outro é o de omitir um ponto-e-vírgula obrigatório ao final de uma linha (por exemplo, o ponto-e-vírgula ao final da linha (4)); um terceiro é o de colocar um ponto-e-vírgula estranho ao fim de uma linha antes de um `else` (por exemplo, colocar um ponto-e-vírgula ao fim da linha (7)).

Talvez uma razão pela qual os erros de ponto-e-vírgula sejam tão comuns é que seu uso varia grandemente de uma linguagem para outra. Em Pascal, um ponto-e-vírgula é um separador de enunciados; em PL/1 e C é um terminador. Alguns estudos têm sugerido que a última utilização é menos propensa a erros (Gannon e Horning [1975]).

Um exemplo típico de um erro de operador é o de omitir os dois pontos em `:=`. Erros de grafia em palavras-chave são usualmente raros, mas omitir o `i` de `writeln` seria um exemplo representativo.

Muitos compiladores Pascal não têm dificuldades em tratar os erros comuns de inserção, remoção e transformação. De fato, vários compiladores Pascal irão tratar corretamente o programa acima com um erro comum de pontuação ou de operador; irão emitir somente um diagnóstico de alerta, apontando a construção ilegal.

No entanto, um outro tipo de erro é muito mais difícil de se reparar corretamente. É o caso de um `begin` ou `end` ausente (por exemplo, a omissão da linha (9)). A maioria dos compiladores não irá tentar reparar esse tipo de erro. □

Como deveria um tratador de erros reportar a presença de um erro? No mínimo, deveria informar o local no programa-fonte onde o

mesmo foi detectado, uma vez que existe uma boa chance de o erro efetivo ter ocorrido uns poucos *tokens* antes. Uma estratégia comum empregada por muitos compiladores é a de imprimir a linha ilegal com um apontador para a posição na qual o erro foi detectado. Se existir um razoável prognóstico do que o erro realmente foi, uma compreensível mensagem de diagnóstico informativa é também incluída; por exemplo, "ponto-e-vírgula ausente nesta posição".

Uma vez que o erro tenha sido detectado, como deveria o analisador sintático se recuperar? Como veremos, existe um número de estratégias gerais, mas nenhum método claramente se impõe sobre os demais. Na maioria dos casos, não é adequado para o analisador sintático encerrar logo após detectar o primeiro erro, porque o processamento da entrada restante ainda pode revelar outros. Usualmente, existe alguma forma de recuperação de erros na qual o analisador tenta restaurar a si mesmo para um estado onde o processamento da entrada possa continuar com uma razoável esperança de que o resto correto da entrada será analisado e tratado adequadamente pelo compilador.

Um trabalho inadequado de recuperação pode introduzir uma avalancha de erros "espúrios", que não foram cometidos pelo programador, mas introduzidos pelas modificações no estado do analisador sintático durante a recuperação de erros. Numa forma similar, uma recuperação de erros sintáticos pode introduzir erros semânticos espúrios que serão detectados posteriormente pelas fases de análise semântica e de geração de código. Por exemplo, ao se recuperar de um erro, o analisador pode pular a declaração de alguma variável, digamos `zap`. Quando `zap` for posteriormente encontrada nas expressões, não haverá nada sintaticamente errado, mas como não há uma entrada na tabela de símbolos para `zap`, a mensagem "`zap` não definido" será gerada.

Uma estratégia cautelosa para o compilador é a de inibir as mensagens de erro que provenham de erros descobertos muito proximamente no fluxo de entrada. Em alguns casos, pode haver erros demais para o compilador continuar um processamento sensível (por exemplo, como deveria um compilador Pascal responder ao receber um programa Fortran como entrada?). Parece que uma estratégia de recuperação de erros tem que ser um compromisso cuidadosamente considerado levando em conta os tipos de erros que são mais propensos a ocorrer e razoáveis de processar.

Como mencionamos, alguns compiladores tentam reparar os erros, num processo em que tentam adivinhar o que o programador queria escrever. O compilador PL/C (Conway e Wilcox [1973]) é um exemplo desse tipo. Exceto, possivelmente, num ambiente de pequenos programas escritos por estudantes principiantes, a reparação extensiva de erros não é propensa a pagar o seu custo. De fato, com a ênfase crescente na computação interativa e bons ambientes de programação, a tendência parece estar na direção de mecanismos simples de recuperação de erros.

Estratégias de Recuperação de Erros

Existem muitas estratégias gerais diferentes que um analisador sintático pode empregar para se recuperar de um erro sintático. Apesar de nenhuma delas ter provado ser universalmente aceitável, uns poucos métodos têm ampla aplicabilidade. Introduzimos aqui as seguintes estratégias:

- modalidade do desespero
- nível de frase
- produções de erro
- correção global

Recuperação na modalidade do desespero. Este é o método mais simples de implementar e pode ser usado pela maioria dos métodos de análise sintática. Ao descobrir um erro, o analisador sintático descarta símbolos de entrada, um de cada vez, até que seja encontrado um *token* pertencente a um conjunto designado de *tokens* de sincronização. Os *tokens* de sincronização são usualmente delimi-tadores, tais como o ponto-e-vírgula ou o **end**, cujo papel no programa-fonte seja claro. Naturalmente, o projetista do compilador precisa selecionar os *tokens* de sincronização apropriados à linguagem-fonte. A correção na modalidade do desespero, que freqüentemente pula uma parte considerável da entrada sem verificá-la, procurando por erros adicionais, possui a vantagem da simplicidade e, diferentemente dos outros métodos a serem enfocados adiante, tem a garantia de não entrar num laço infinito. Nas situações em que os erros múltiplos num mesmo enunciado sejam raros, esse método pode ser razoavelmente adequado.

Recuperação de frases. Ao descobrir um erro, o analisador sintático pode realizar uma correção local na entrada restante. Isto é, pode substituir um prefixo da entrada remanescente por alguma cadeia que permita ao analisador seguir em frente. Correções locais típicas seriam substituir uma vírgula por um ponto-e-vírgula, remover um ponto-e-vírgula estranho ou inserir um ausente. A escolha da correção local é deixada para o projetista do compilador. Naturalmente, devemos ser cuidadosos, escolhendo substituições que não levem a laços infinitos, como seria o caso, por exemplo, se inseríssemos para sempre na entrada algo à frente do seu símbolo corrente.

Esse tipo de substituição pode corrigir qualquer cadeia e tem sido usado em vários compiladores de correção de erros. O método foi primeiramente usado na análise sintática *top-down*. Sua maior desvantagem está na dificuldade que tem ao lidar com situações nas quais o erro efetivo ocorreu antes do ponto de detecção.

Produções de erro. Se tivéssemos uma boa idéia dos erros comuns que poderiam ser encontrados, poderíamos aumentar a gramática para a linguagem em exame com as produções que gerassem construções ilegais. Usamos, então, a gramática aumentada com essas produções de erro para construir um analisador sintático. Se uma produção de erro for usada pelo analisador, podemos gerar diagnósticos apropriados para indicar a construção ilegal que foi reconhecida na entrada.

Correção global. Idealmente, gostaríamos que um compilador fizesse tão poucas mudanças quanto possível, ao processar uma cadeia de entrada ilegal. Existem algoritmos para escolher uma seqüência mínima de mudanças de forma a se obter uma correção global de menor custo. Dadas uma cadeia de entrada incorreta x e uma gramática G, esses algoritmos irão encontrar uma árvore gramatical para uma cadeia relacionada y, de tal forma que as inserções, remoções e mudanças de *tokens* requeridas para transformar x em y sejam tão pequenas quanto possível. Infelizmente, esses métodos são em geral muito custosos de implementar, em termos de tempo e espaço e, então, essas técnicas são correntemente apenas de interesse teórico.

Devemos assinalar que o programa correto mais próximo pode não ser aquele que o programador tinha em mente. Apesar de tudo, a noção de correção de custo mínimo fornece um padrão de longo alcance para avaliar as técnicas de recuperação de erros e tem sido usada para encontrar cadeias ótimas de substituição para a recuperação em nível de frase.

4.2 GRAMÁTICAS LIVRES DE CONTEXTO

Muitas construções de linguagens de programação possuem uma estrutura inerentemente recursiva que pode ser identificada por gramáticas livres de contexto. Por exemplo, poderíamos ter um enunciado condicional definido por uma regra tal como:

se S_1 e S_2 são enunciados e E é uma expressão, então
"if E **then** S_1 **else** S_2" é um enunciado. (4.1)

Essa forma de enunciado condicional não pode ser especificada usando-se a notação para expressões regulares; no Capítulo 3, vimos que as expressões regulares podem especificar a estrutura léxica dos *tokens*. Por outro lado, usando-se a variável sintática *cmd* para denotar a classe de comandos e *expr* para a classe de expressões, podemos prontamente expressar (4.1) usando a produção gramatical

$$cmd \rightarrow \textbf{if}\ expr\ \textbf{then}\ cmd\ \textbf{else}\ cmd \qquad (4.2)$$

Nesta seção, revisamos a definição de gramática livre de contexto e introduzimos a terminologia para falarmos a respeito da análise sintática. Da Seção 2.2, uma gramática livre de contexto (gramática,

simplificadamente) consiste em terminais, não-terminais, um símbolo de partida e produções.

1. Os terminais são os símbolos básicos a partir dos quais as cadeias são formadas. A palavra "*token*" será um sinônimo de "terminal" ao falarmos a respeito de gramáticas para linguagens de programação. Em (4.2), cada uma das palavras-chave **if, then** e **else** é um terminal.
2. Os não-terminais são variáveis sintáticas que denotam cadeias de caracteres. Em (4.2), *cmd* e *expr* são não-terminais. Os não-terminais definem conjuntos de cadeias que auxiliam a definição da linguagem gerada pela gramática. Também impõem uma estrutura hierárquica na linguagem que é útil tanto para a análise sintática quanto para a tradução.
3. Numa gramática, um não-terminal é distinguido como o símbolo de partida, e o conjunto de cadeias que o mesmo denota é a linguagem definida pela gramática.
4. As produções de uma gramática especificam a forma pela qual os terminais e não-terminais podem ser combinados a fim de formar cadeias. Cada produção consiste em um não-terminal, seguido por uma seta (às vezes o símbolo ::= é usado no lugar da seta), seguido por uma cadeia de não-terminais e terminais.

Exemplo 4.2. A gramática com as seguintes produções define expressões aritméticas simples.

$$
\begin{aligned}
&\mathit{expr} \rightarrow \mathit{expr}\ \mathit{op}\ \mathit{expr}\\
&\mathit{expr} \rightarrow (\mathit{expr})\\
&\mathit{expr} \rightarrow -\ \mathit{expr}\\
&\mathit{expr} \rightarrow \mathbf{id}\\
&\mathit{op} \rightarrow +\\
&\mathit{op} \rightarrow -\\
&\mathit{op} \rightarrow *\\
&\mathit{op} \rightarrow /\\
&\mathit{op} \rightarrow \uparrow
\end{aligned}
$$

Nesta gramática, os símbolos terminais são

$$\mathbf{id}\ +\ -\ *\ /\ \uparrow\ (\ )$$

Os símbolos não-terminais são *expr* e *op*, e *expr* é o símbolo de partida. □

Convenções Notacionais

A fim de evitar especificações do tipo "esses são terminais", "esses são não-terminais" e assim por diante, iremos empregar as seguintes convenções notacionais relacionadas às gramáticas ao longo do resto deste livro.

1. Símbolos terminais:

   i) Letras minúsculas do início do alfabeto, tais como *a, b, c.*
   ii) Símbolos de operadores, tais como +, − etc.
   iii) Símbolos de pontuação, tais como parênteses, vírgula etc.
   iv) Os dígitos 0, 1,...,9.
   v) Cadeias em negrito como **id** ou **if.**

2. Símbolos não-terminais:

   i) Letras maiúsculas do início do alfabeto, tais como *A, B, C.*
   ii) A letra *S*, que, quando aparece, é usualmente o símbolo de partida.
   iii) Os nomes em itálico formados por letras minúsculas, como *expr* ou *cmd.*

3. As letras maiúsculas do final do alfabeto, tais como *X, Y, Z,* representam *símbolos gramaticais,* isto é, terminais ou não-terminais.
4. Letras minúsculas, ao fim do alfabeto, principalmente *u, v,...,z,* representam cadeias de terminais.
5. Letras gregas minúsculas, α, β e γ, por exemplo, representam cadeias de símbolos gramaticais. Dessa forma, uma produção genérica poderia ser escrita como $A \rightarrow \alpha$, indicando que existe um único não-terminal *A* à esquerda da seta (o *lado esquerdo* da produção) e uma cadeia de símbolos gramaticais α à direita da seta (o *lado direito* da produção).
6. Se $A \rightarrow \alpha_1, A \rightarrow \alpha_2,\ldots,A \rightarrow \alpha_k$ são todas produções com *A* à esquerda (chamamos de *produções-A*), podemos escrever $A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_k$. Chamamos $\alpha_1, \alpha_2,\ldots,\alpha_k$ de *alternativas* para *A*.
7. A menos que seja explicitamente estabelecido, o lado esquerdo da primeira produção é o símbolo de partida.

Exemplo 4.3. Usando as simplificações, poderíamos escrever a gramática do Exemplo 4.2 concisamente como

$$
\begin{aligned}
E &\rightarrow E\,A\,E \mid (\,E\,) \mid -E \mid \mathbf{id}\\
A &\rightarrow + \mid - \mid * \mid / \mid \uparrow
\end{aligned}
$$

Nossas convenções notacionais nos dizem que *E* e *A* são não-terminais, tendo *E* como símbolo de partida. Os símbolos restantes são terminais.□

Derivações

Existem várias formas de se enxergar o processo pelo qual uma gramática define uma linguagem. Na Seção 2.2, examinamos este processo como sendo o de construir árvores gramaticais, mas existe também uma visão derivacional relacionada, que freqüentemente achamos útil. De fato, a visão derivacional fornece uma precisa descrição da construção *top-down* da árvore gramatical. A idéia central aqui é que uma produção seja tratada como uma regra de reescrita, na qual o não-terminal à esquerda é substituído pela cadeia no lado direito da produção.

Por exemplo, considere a seguinte gramática para expressões aritméticas, com o não-terminal *E* representando uma expressão.

$$E \rightarrow E + E \mid E * E \mid (\,E\,) \mid -E \mid \mathbf{id} \tag{4.3}$$

A produção $E \rightarrow -E$ significa que uma expressão precedida por um sinal de menos também é uma expressão. Essa produção pode ser usada para gerar expressões mais complexas a partir das mais simples, permitindo que se substitua qualquer instância de *E* por $-E$. No caso mais simples, podemos substituir um único *E* por $-E$. Podemos descrever essa ação escrevendo

$$E \Rightarrow -E$$

que é lido "*E* deriva $-E$." A produção $E \rightarrow (E)$ nos diz que poderíamos também substituir uma instância de um *E* em qualquer cadeia de símbolos gramaticais por (*E*); por exemplo, $E*E \Rightarrow (E)*E$ ou $E*E \Rightarrow E*(E)$.

Podemos tomar um único *E* e aplicar repetidamente as produções em qualquer ordem, a fim de obtermos uma seqüência de substituições. Por exemplo,

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(\mathbf{id})$$

Chamamos uma tal seqüência de substituições de uma *derivação* de $-(\mathbf{id})$ a partir de *E*. Essa derivação providencia uma prova de que uma instância particular de uma expressão é a cadeia $-(\mathbf{id})$.

Num posicionamento mais abstrato, dizemos que $\alpha A \beta \Rightarrow \alpha\gamma\beta$ se $A \rightarrow \gamma$ for uma produção e α e β forem cadeias arbitrárias de símbolos gramaticais. Se $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_n$, dizemos que α_1 *deriva* α_n. O símbolo ⇒ significa "deriva em um passo". Freqüentemente desejamos dizer "deriva em zero ou mais passos". Para esse propósito usamos o símbolo $\overset{*}{\Rightarrow}$. Dessa forma,

1. $\alpha \overset{*}{\Rightarrow} \alpha$ para qualquer cadeia α, e
2. Se $\alpha \overset{*}{\Rightarrow} \beta$ e $\beta \Rightarrow \gamma$, então $\alpha \overset{*}{\Rightarrow} \gamma$.

Do mesmo modo, usamos $\overset{+}{\Rightarrow}$ para significar "deriva em um ou mais passos".

Dada uma gramática *G*, com símbolo de partida *S*, podemos usar a relação $\overset{+}{\Rightarrow}$ para definir *L (G),* a *linguagem gerada por G.* As cadei-

as em $L(G)$ só podem conter símbolos terminais de G. Dizemos que uma cadeia de terminais w está em $L(G)$ se e somente se $S \overset{+}{\Rightarrow} w$. A cadeia w é chamada de uma *sentença* de G. Uma linguagem que possa ser gerada por uma gramática é dita ser uma *linguagem livre de contexto*. Se duas gramáticas geram a mesma linguagem, as gramáticas são ditas *equivalentes*.

Se $S \overset{*}{\Rightarrow} \alpha$, onde α pode conter não-terminais, dizemos, então, que α é uma *forma sentencial* de G. Uma sentença é uma forma sentencial despida de não-terminais.

Exemplo 4.4. A cadeia $-(\mathbf{id}+\mathbf{id})$ é uma sentença da gramática (4.3) porque existe a derivação

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(\mathbf{id}+E) \Rightarrow -(\mathbf{id}+\mathbf{id}) \quad (4.4)$$

As cadeias E, $-E$, $-(E)$,..., $-(\mathbf{id}+\mathbf{id})$, figurando nesta derivação, são todas formas sentenciais desta gramática. Escrevemos $E \overset{+}{\Rightarrow} -(\mathbf{id}+\mathbf{id})$ para indicar que $(\mathbf{id}+\mathbf{id})$ pode ser derivada a partir de E.

Podemos mostrar, por indução no comprimento de uma derivação, que cada sentença na linguagem da gramática (4.3) é uma expressão aritmética envolvendo os operadores + e *, o operador unário −, parênteses e o operando **id**. Similarmente, podemos mostrar por indução no comprimento de uma expressão aritmética que todas essas expressões podem ser geradas por essa gramática. Por conseguinte, a gramática (4.3) gera precisamente o conjunto de todas as expressões aritméticas envolvendo os operadores binários + e *, o unário −, parênteses e o operando **id**. □

A cada passo numa derivação, existem duas escolhas a serem feitas. Primeiro, precisamos escolher qual não-terminal substituir e, segundo, tendo feito tal escolha, que alternativa usar na substituição daquele não-terminal. Por exemplo, a derivação (4.4) do Exemplo 4.4 poderia continuar a partir de $-(E+E)$ como se segue

$$-(E+E) \Rightarrow -(E+\mathbf{id}) \Rightarrow -(\mathbf{id}+\mathbf{id}) \quad (4.5)$$

Cada não-terminal em (4.5) é substituído pelo mesmo lado direito, como no Exemplo 4.4, mas a ordem das substituições é diferente.

Para compreender como certos analisadores sintáticos funcionam, precisamos considerar derivações nas quais somente o não-terminal mais à esquerda em qualquer forma sentencial seja substituído a cada passo. Tais derivações são ditas *mais à esquerda*. Se $\alpha \Rightarrow \beta$ por um passo no qual o não-terminal mais à esquerda em α é substituído, escrevemos $\alpha \underset{mdl}{\Rightarrow} \beta$. Como a derivação (4.4) é mais à esquerda, podemos reescrevê-la como:

$$E \underset{mdl}{\Rightarrow} -E \underset{mdl}{\Rightarrow} -(E) \underset{mdl}{\Rightarrow} -(E+E) \underset{mdl}{\Rightarrow} -(\mathbf{id}+E) \underset{mdl}{\Rightarrow} -(\mathbf{id}+\mathbf{id})$$

Usando nossas convenções notacionais, cada passo mais à esquerda pode ser escrito $wA\gamma \underset{mdl}{\Rightarrow} w\delta\gamma$, onde w consiste em terminais somente, $A \rightarrow \delta$ é a produção aplicada, e γ é uma cadeia de símbolos gramaticais. Para enfatizar o fato de que α deriva β por uma derivação mais à esquerda, escrevemos $\alpha \overset{*}{\underset{mdl}{\Rightarrow}} \beta$. Se $S \overset{*}{\underset{mdl}{\Rightarrow}} \alpha$, dizemos, então, que α é uma *forma sentencial mais à esquerda* da gramática em questão.

Definições análogas valem para derivações *mais à direita*, nas quais o não-terminal mais à direita é substituído a cada passo. Derivações mais à direita são algumas vezes chamadas de derivações *canônicas*.

Árvores Gramaticais e Derivações

Uma árvore gramatical pode ser vista como uma representação gráfica para uma derivação que filtre a escolha relacionada à ordem de substituição. Relembremos, da Seção 2.2, que cada nó interior de uma árvore gramatical é rotulado por algum não-terminal A e que os filhos de um nó são rotulados, da esquerda para a direita, pelos símbolos do lado direito da produção pelos quais A foi substituído na derivação. As folhas da árvore gramatical são rotuladas por não-terminais ou terminais e, lidos da esquerda para a direita, constituem uma forma sentencial chamada de produto ou fronteira da árvore. Por exemplo, a árvore gramatical para $-(\mathbf{id}+\mathbf{id})$ implicada pela derivação (4.4) é mostrada na Fig. 4.2.

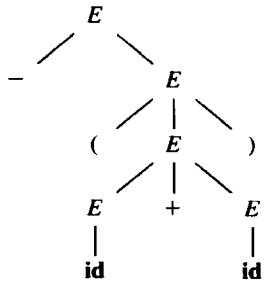


Fig. 4.2. Árvore gramatical para $-(\mathbf{id}+\mathbf{id})$.

Para se compreender a relação entre as derivações e as árvores gramaticais, consideremos uma derivação genérica $\alpha_1 \Rightarrow \alpha_2 \ldots \Rightarrow \alpha_n$, onde α_1 é um não-terminal único A. Para cada forma sentencial α_i na derivação, construímos uma árvore gramatical cujo produto é α_i. O processo é uma indução em i. Como base da indução, a árvore para $\alpha_1 \equiv A$ é um único nó rotulado A. Para realizar a indução, suponhamos já ter construído uma árvore gramatical cujo produto seja $\alpha_{i-1} = X_1X_2 \ldots X_k$. (Relembremos nossas convenções, cada X_i é um terminal ou um não-terminal.) Suponhamos que α_i seja derivada a partir de α_{i-1} pela substituição de X_j, um não-terminal, por $\beta = Y_1Y_2 \ldots Y_r$. Ou seja, no iésimo passo da derivação, a produção $X_j \rightarrow \beta$ é aplicada a α_{i-1} a fim de derivar $\alpha_i = X_1X_2 \ldots X_{j-1}\beta X_{j+1} \ldots X_k$.

Para modelar este passo de derivação, encontramos a j-ésima folha a partir da esquerda na árvore gramatical corrente. Esta folha é rotulada X_j. Damos a esta folha r filhos, rotulados $Y_1, Y_2, \ldots Y_r$, a partir da esquerda. Como um caso especial, se $r = 0$, isto é, $\beta = \epsilon$, então damos à j-ésima folha um filho rotulado ϵ.

Exemplo 4.5. Consideremos a derivação (4.4). A seqüência de árvores gramaticais construída a partir desta derivação é mostrada na Fig. 4.3. No primeiro passo de derivação, $E \Rightarrow -E$. Para modelar este pas-

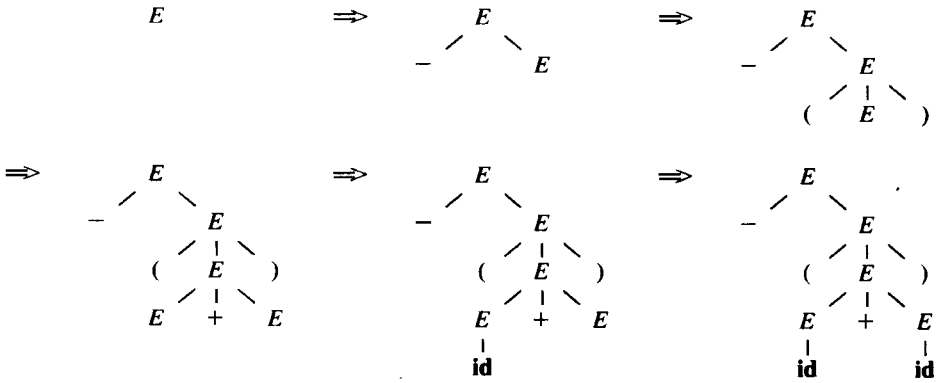


Fig. 4.3. Construindo a árvore gramatical a partir da derivação (4.4).

so, adicionamos dois filhos, rotulados $-$ e E, à raiz E da árvore inicial a fim de criarmos a segunda árvore.

No segundo passo de derivação, $-E \Rightarrow -(E)$. Conseqüentemente, adicionamos três filhos, rotulados (, E e), à folha rotulada E da segunda árvore para obtermos a terceira árvore com produto $-(E)$. Continuando dessa maneira, obtemos na sexta etapa a árvore gramatical completa. □

Como mencionamos, uma árvore gramatical ignora as variações na ordem pela qual os símbolos foram substituídos nas formas sentenciais. Por exemplo, se a derivação (4.4) fosse continuada como na linha (4.5), a mesma árvore gramatical final da Fig. 4.3 resultaria. Essas variações da ordem pela qual as produções são aplicadas também podem ser eliminadas considerando-se apenas derivações mais à esquerda (ou mais à direita). Não é difícil constatar que cada árvore gramatical possui associada a si uma única derivação mais à esquerda ou mais à direita. Na seqüência, iremos freqüentemente analisar sintaticamente através da reconstituição de uma derivação mais à esquerda ou mais à direita, subentendendo que, em lugar da derivação, poderíamos produzir a própria árvore gramatical em si. Não devemos assumir, entretanto, que cada sentença tenha necessariamente somente uma árvore gramatical ou uma única derivação mais à esquerda ou à direita.

Exemplo 4.6. Vamos considerar a gramática de expressões aritméticas (4.3). A sentença **id**+**id***__id__ possui duas derivações distintas mais à esquerda:

$$\begin{aligned} E &\Rightarrow E+E \\ &\Rightarrow \mathbf{id}+E \\ &\Rightarrow \mathbf{id}+E*E \\ &\Rightarrow \mathbf{id}+\mathbf{id}*E \\ &\Rightarrow \mathbf{id}+\mathbf{id}*\mathbf{id} \end{aligned} \qquad \begin{aligned} E &\Rightarrow E*E \\ &\Rightarrow E+E*E \\ &\Rightarrow \mathbf{id}+E*E \\ &\Rightarrow \mathbf{id}+\mathbf{id}*E \\ &\Rightarrow \mathbf{id}+\mathbf{id}*\mathbf{id} \end{aligned}$$

com as duas árvores gramaticais correspondentes mostradas na Fig. 4.4. □

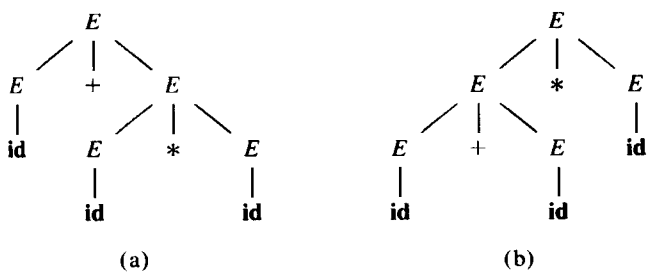


Fig. 4.4. Duas árvores gramaticais para **id**+**id***__id__.

Note-se que a árvore gramatical da Fig. 4.4(a) reflete a precedência comumente assumida de + e *, enquanto que a árvore da Fig. 4.4(b) não a reflete. Ou seja, é usual tratarmos o operador * como tendo maior precedência do que +, correspondendo ao fato de que normalmente iríamos avaliar a expressão $a+b*c$ como $a+(b*c)$, ao invés de $(a+b)*c$.

Ambigüidade

Uma gramática que produza mais de uma árvore gramatical para alguma sentença é dita *ambígua*. Colocado de outra forma, uma gramática ambígua é aquela que produz mais de uma derivação à esquerda, ou à direita, para a mesma sentença. Para certos tipos de analisadores sintáticos, é desejável que a gramática seja inambígua, porque se não o for, não poderemos selecionar, de forma única, a árvore gramatical para uma dada sentença. Para algumas aplicações, também consideraremos métodos através dos quais possamos usar gramáticas ambíguas, juntamente com *regras de inambigüidade* que "descartam" árvores gramaticais indesejáveis, deixando-nos somente com uma árvore para cada sentença.

4.3 ESCREVENDO UMA GRAMÁTICA

As gramáticas são capazes de descrever a maioria, mas não a totalidade das sintaxes das linguagens de programação. Uma parte limitada da análise sintática é realizada pelo analisador léxico, na medida em que produz uma seqüência de *tokens* a partir dos caracteres de entrada. Certas restrições feitas à entrada, tais como a exigência de que os identificadores sejam declarados antes de serem usados, não podem ser descritas por uma gramática livre de contexto. Conseqüentemente, as seqüências de *tokens* aceitas por um analisador sintático formam um subconjunto de uma linguagem de programação; as fases subseqüentes precisam analisar a sua saída a fim de assegurar a concordância com as regras que não são checadas pelo analisador sintático (veja o Capítulo 6).

Começamos esta seção considerando a divisão de trabalho entre um analisador léxico e um analisador sintático. Como cada método de análise sintática pode tratar de gramáticas que tenham uma certa conformação, a gramática inicial pode ter que ser reescrita a fim de se tornar analisável pelo método escolhido. As gramáticas adequadas às expressões podem ser freqüentemente construídas usando-se informações sobre a associatividade e a precedência, como na Seção 2.2. Nesta seção, consideramos as transformações que sejam úteis para reescrever as gramáticas de forma a se tornarem adequadas à análise *top-down*. Concluímos esta seção considerando algumas construções de linguagens de programação que não podem ser descritas por qualquer gramática.

Expressões Regulares *vs.* Gramáticas Livres de Contexto

Cada construção que possa ser descrita por uma expressão regular também pode ser descrita por uma gramática. Por exemplo, a expressão regular $(a \mid b)^*abb$ e a gramática

$$\begin{aligned} A_0 &\rightarrow aA_0 \mid bA_0 \mid aA_1 \\ A_1 &\rightarrow bA_2 \\ A_2 &\rightarrow bA_3 \\ A_3 &\rightarrow \epsilon \end{aligned}$$

descrevem a mesma linguagem, o conjunto de cadeias de a's e b's terminadas em abb.

Podemos converter mecanicamente um autômato finito não-determinístico (AFN) numa gramática que gere a mesma linguagem reconhecida pelo AFN. A gramática acima foi construída a partir do AFN da Fig. 3.23, usando-se a seguinte construção: para cada estado i do AFN, criar um símbolo não-terminal A_i. Se o estado i possuir uma transição para o estado j no símbolo a, introduzir a produção $A_i \rightarrow aA_j$. Se o estado i vai para o estado j à entrada ϵ, introduzir a produção $A_i \rightarrow A_j$. Se i for um estado de aceitação, introduzir $A_i \rightarrow \epsilon$. Se i for o estado de partida, fazer de A_i o símbolo de partida da gramática.

Uma vez que cada expressão regular é uma linguagem livre de contexto, podemos razoavelmente indagar "por que usar expressões regulares para definir a estrutura léxica da linguagem"? Existem várias razões.

1. As regras léxicas de uma linguagem são freqüentemente simples e para descrevê-las não precisamos de uma notação tão poderosa quanto a das gramáticas.
2. As expressões regulares geralmente providenciam, para os *tokens* da gramática, uma notação mais concisa e facilmente compreendida.
3. A partir de expressões regulares, podem ser construídos automaticamente analisadores léxicos mais eficientes do que a partir de gramáticas arbitrárias.
4. A separação da estrutura sintática da linguagem nas partes léxica e não-léxica providencia uma forma conveniente de modularizar a vanguarda de um compilador em componentes administravelmente dimensionados.

Não existem diretrizes firmemente estabelecidas sobre o que colocar nas regras léxicas, ao contrário das regras sintáticas. As expres-

sões regulares são mais úteis para descrever a estrutura de construções léxicas tais como identificadores, constantes palavras-chave e assim por diante. Por outro lado, as gramáticas são mais úteis na descrição de estruturas aninhadas tais como parênteses balanceados, *begin-ends* emparelhados, *if-then-elses* correspondentes e assim por diante. Como tínhamos assinalado, tais estruturas aninhadas não podem ser descritas por expressões regulares.

Verificando a Linguagem Gerada por uma Gramática

Apesar de os projetistas de compiladores raramente o fazerem para uma gramática de uma linguagem de programação completa, é importante estarmos capacitados a sustentar que um dado conjunto de produções gere uma linguagem particular. As construções problemáticas podem ser estudadas escrevendo-se uma gramática abstrata concisa e em seguida analisando-se a linguagem que a mesma gera. Adiante, iremos construir uma gramática para expressões condicionais.

Uma prova de que uma gramática G gera uma linguagem L possui duas partes: precisamos mostrar que cada cadeia gerada por G está em L e, reciprocamente, que cada cadeia em L pode ser gerada por G.

Exemplo 4.7. Consideremos a gramática (4.6)

$$S \rightarrow (S)S \mid \epsilon \tag{4.6}$$

Pode não ser inicialmente aparente, mas esta gramática simples gera todas as cadeias de parêntesis balanceados, e somente tais cadeias. Para verificarmos isso, mostraremos primeiro que cada sentença derivável de S é balanceada e, em seguida, que cada cadeia balanceada é derivável a partir de S. Para mostrar que cada sentença derivável de S é balanceada, usamos uma prova indutiva sobre o número de passos numa derivação. Como base da indução, notamos que a única cadeia de terminais derivável a partir de S em um único passo é a cadeia vazia, a qual certamente é balanceada.

Vamos assumir agora que todas as derivações com menos de n passos produzam sentenças balanceadas e consideremos uma derivação mais à esquerda com exatamente n passos. Tal derivação é da forma

$$S \Rightarrow (S)S \overset{*}{\Rightarrow} (x)S \overset{*}{\Rightarrow} (x)y$$

As derivações de x e y a partir de S têm menos que n passos, e, então, pela hipótese indutiva, x e y são balanceadas. Dessa forma, a cadeia $(x)y$ tem que ser balanceada.

Mostramos, então, que qualquer cadeia derivável a partir de S é balanceada. Precisamos mostrar em seguida que qualquer cadeia balanceada é derivável a partir de S. Para realizar isso, usamos a indução no comprimento de uma cadeia. Como base da indução, a cadeia vazia é derivável a partir de S.

Vamos assumir agora que toda cadeia balanceada de comprimento menor do que $2n$ seja derivável a partir de S e consideremos uma cadeia balanceada w de comprimento $2n$, $n \geq 1$. Certamente w começa por um parêntesis à esquerda. Seja (x) o menor prefixo de w tendo um número igual de parêntesis à esquerda e à direita. Então, w pode ser escrita como $(x)y$, onde ambos x e y são balanceadas. Como x e y são de comprimento menor do que $2n$, são deriváveis a partir de S pela hipótese indutiva. Dessa forma, podemos encontrar uma derivação da forma

$$S \Rightarrow (S)S \overset{*}{\Rightarrow} (x)S \overset{*}{\Rightarrow} (x)y$$

provando que $w = (x)y$ também é derivável a partir de S. □

Eliminando a Ambigüidade

Algumas vezes uma gramática ambígua pode ser reescrita de forma a eliminar a ambigüidade. Como um exemplo, vamos eliminar a ambigüidade da seguinte gramática do "*else-vazio*":

$$\begin{aligned} cmd \rightarrow\ & \textbf{if}\ expr\ \textbf{then}\ cmd \\ \mid\ & \textbf{if}\ expr\ \textbf{then}\ cmd\ \textbf{else}\ cmd \\ \mid\ & \textbf{outro} \end{aligned} \tag{4.7}$$

Aqui **"outro"** significa qualquer outro enunciado. De acordo com esta gramática, o enunciado condicional composto

$$\textbf{if}\ E_1\ \textbf{then}\ S_1\ \textbf{else if}\ E_2\ \textbf{then}\ S_2\ \textbf{else}\ S_3$$

possui a árvore gramatical mostrada na Fig. 4.5. A gramática (4.7) é ambígua, uma vez que a cadeia

$$\textbf{if}\ E_1\ \textbf{then if}\ E_2\ \textbf{then}\ S_1\ \textbf{else}\ S_2 \tag{4.8}$$

possui as duas árvores mostradas na Fig. 4.6.

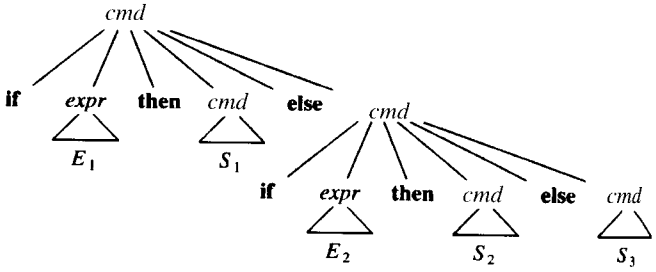


Fig. 4.5. Árvore gramatical para o enunciado condicional.

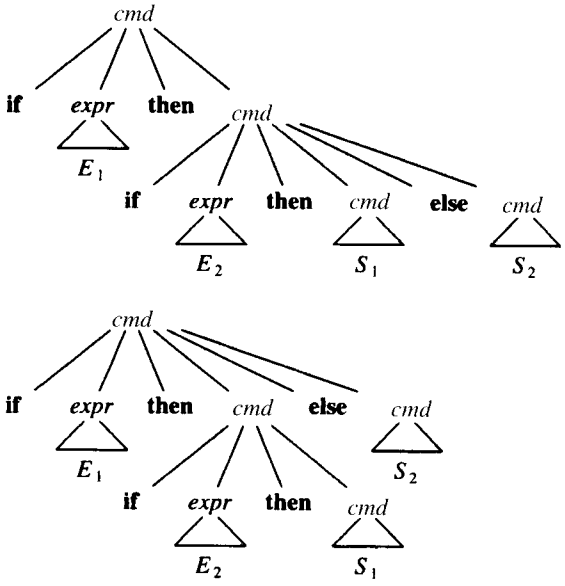


Fig. 4.6. Duas árvores gramaticais para uma sentença ambígua.

Em todas as linguagens de programação com enunciados condicionais desta forma, a primeira árvore gramatical é preferida. A regra geral é "associar cada **else** ao **then** anterior mais próximo ainda não associado". Essa regra de inambigüidade pode ser incorporada diretamente à gramática. Por exemplo, podemos reescrever a gramática (4.7) sob a forma inambígua abaixo. A idéia está em que um enunciado figurando entre um **then** e um **else** precisa ser "associado", isto é, não pode terminar com um **then** ainda não associado seguido por qualquer outro enunciado, pois o **else** seria forçado a se associar a esse **then** não associado. Um enunciado associado ou é um enunciado **if-then-else** contendo somente enunciados associados ou é qualquer outro tipo de enunciado incondicional. Dessa forma podemos usar a gramática

$$
\begin{array}{rl}
cmd \rightarrow & cmd_associado \\
| & cmd_não_associado \\
cmd_associado \rightarrow & \textbf{if}\ expr\ \textbf{then}\ cmd_associado\ \textbf{else}\ cmd_associado \\
| & \textbf{outro} \qquad (4.9) \\
cmd_não_associado \rightarrow & \textbf{if}\ expr\ \textbf{then}\ cmd \\
| & \textbf{if}\ expr\ \textbf{then}\ cmd_associado\ \textbf{else}\ cmd_não_associado
\end{array}
$$

Esta gramática gera o mesmo conjunto de cadeias que (4.7), mas permite somente uma estruturação gramatical para (4.8), especificamente aquela que associa cada **else** com o **then** anterior mais próximo ainda não associado.

Eliminação da Recursão à Esquerda

Uma gramática é *recursiva à esquerda* se possui um não-terminal A tal que exista uma derivação $A \overset{+}{\Rightarrow} A\alpha$ para alguma cadeia α. Os métodos de análise sintática *top-down* não podem processar gramáticas recursivas à esquerda e, conseqüentemente, uma transformação que elimine a recursão à esquerda é necessária. Na Seção 2.4, discutimos a recursão simples à esquerda, onde havia uma produção da forma $A \rightarrow A\alpha$. Aqui, estudamos o caso geral. Na Seção 2.4, mostramos como o par de produções recursivas à esquerda $A \rightarrow A\alpha \mid \beta$ poderia ser substituído pelas produções não-recursivas

$$
\begin{array}{l}
A \rightarrow \beta A' \\
A' \rightarrow \alpha A' \mid \epsilon
\end{array}
$$

sem mudar o conjunto de cadeias de caracteres deriváveis a partir de A. Esta regra por si mesma é suficiente para muitas gramáticas.

Exemplo 4.8. Considere a seguinte gramática para expressões aritméticas.

$$
\begin{array}{l}
E \rightarrow E + T \mid T \\
T \rightarrow T * F \mid F \\
F \rightarrow (\ E\ ) \mid \textbf{id}
\end{array}
\qquad (4.10)
$$

Eliminando a *recursão imediata à esquerda* (produções da forma $A \rightarrow A\alpha$) nas produções para E e T, obtemos

$$
\begin{array}{l}
E \rightarrow TE' \\
E' \rightarrow +TE' \mid \epsilon \\
T \rightarrow FT' \\
T' \rightarrow *FT' \mid \epsilon \\
F \rightarrow (E) \mid \textbf{id}
\end{array}
\qquad (4.11)
$$

□

Não importa quantas produções-A existam, podemos eliminar a recursão imediata das mesmas pela seguinte técnica. Primeiro, agrupamos as produções-A como

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

onde nenhum β_i começa por um A. Em seguida, substituímos as produções–A por

$$
\begin{array}{l}
A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta n A' \\
A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid \epsilon
\end{array}
$$

O não-terminal A gera as mesmas cadeias que antes, mas já não é recursivo à esquerda. Este procedimento elimina todas as recursões à esquerda das produções A e A' (uma vez nenhum α_i seja ϵ), mas não elimina a recursão à esquerda envolvendo derivações em um ou mais passos. Por exemplo, considere a gramática

$$
\begin{array}{l}
S \rightarrow Aa \mid b \\
A \rightarrow Ac \mid Sd \mid \epsilon
\end{array}
\qquad (4.12)
$$

O não-terminal S é recursivo à esquerda porque $S \Rightarrow Aa \Rightarrow Sda$, mas não é imediatamente recursivo.

O algoritmo 4.1, abaixo, irá sistematicamente eliminar a recursão à esquerda de uma gramática. É garantido funcionar se a gramática não possuir ciclos (derivações da forma $A \overset{+}{\Rightarrow} A$) ou produções-$\epsilon$ (produções da forma $A \rightarrow \epsilon$). Os ciclos podem ser sistematicamente eliminados de uma gramática da mesma forma que as produções-ϵ (veja os Exercícios 4.20 e 4.22).

Algoritmo 4.1. Eliminação da recursão à esquerda.

Entrada. Uma gramática G sem ciclos ou produções-ϵ.

Saída. Uma gramática equivalente sem recursão à esquerda.

Método. Aplicar o algoritmo na Fig. 4.7 a G. Note que a gramática resultante não-recursiva à esquerda pode ter produções-ϵ. □

1. Colocar os não-terminais em alguma ordem $A_1, A_2, \ldots, A_n$.

2. **para** $i := 1$ **até** n **faça início**
   para $j := 1$ **até** $i - 1$ **faça início**
   substituir cada produção da forma $A_i \rightarrow A_j\gamma$
   pelas produções $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$,
   onde $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ são todas as produções-A_j correntes;
   fim
   eliminar a recursão imediata à esquerda entre as produções-A_i
   fim

Fig. 4.7. Algoritmo para eliminar a recursão à esquerda de uma gramática.

A razão pela qual o procedimento da Fig. 4.7 funciona está em que, após a iésima menos uma iteração do laço **para** mais externo no passo (2), qualquer produção da forma $A_k \rightarrow A_l\alpha$, onde $k < i$, terá necessariamente $l > k$. Como resultado, na iteração seguinte, o laço mais interno (em j) progressivamente suspende o limite inferior m para qualquer produção da forma $A_i \rightarrow A_m\alpha$, até que tenhamos $m \geq i$. Então, a eliminação da recursão imediata à esquerda para as produções–A_i obriga m a se tornar maior do que i.

Exemplo 4.9. Vamos aplicar este procedimento à gramática (4.12). Tecnicamente, o algoritmo 4.1 não é garantido funcionar, por causa da produção-ϵ, mas, neste caso particular, a produção $A \rightarrow \epsilon$ é inofensiva.

Ordenamos os não-terminais S, A. Não existe recursão imediata à esquerda entre as produções-S, e, conseqüentemente, nada acontece durante o passo (2) para o caso $i = 1$. Para $i = 2$, substituímos as produções-S em $A \rightarrow Sd$ a fim de obter as seguintes produções-A.

$$A \rightarrow Ac \mid Aad \mid bd \mid \epsilon$$

A eliminação da recursão imediata à esquerda entre as produções-A gera a seguinte gramática.

$$
\begin{array}{l}
S \rightarrow Aa \mid b \\
A \rightarrow bdA' \mid A' \\
A' \rightarrow cA' \mid adA' \mid \epsilon
\end{array}
$$

□

Fatoração à Esquerda

A fatoração à esquerda é uma transformação gramatical útil para a criação de uma gramática adequada à análise sintática preditiva. A idéia básica está em, quando não estiver claro qual das duas produções alternativas usar para expandir um não-terminal A, estarmos capacitados a reescrever as produções–A e postergar a decisão até que tenhamos visto o suficiente da entrada para realizarmos a escolha certa.

Por exemplo, se tivermos as duas produções

$$
\begin{array}{rl}
cmd \rightarrow & \textbf{if}\ expr\ \textbf{then}\ cmd\ \textbf{else}\ cmd \\
| & \textbf{if}\ expr\ \textbf{then}\ cmd
\end{array}
$$

ao enxergarmos o *token* de entrada **if,** não podemos imediatamente dizer qual produção escolher a fim de expandir *cmd*. Em geral, se $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ são duas produções-A, e a entrada começa por uma cadeia não vazia derivada a partir de α, não sabemos se vamos expandir A em $\alpha\beta_1$ ou em $\alpha\beta_2$. Entretanto, podemos postergar a decisão expandindo A para $\alpha A'$. Então, após enxergarmos a entrada derivada a partir de α, expandimos A' em β_1 ou em β_2. Isto é, as produções originais, fatoradas à esquerda, se tornam:

$$A \rightarrow \alpha A'$$
$$A' \rightarrow \beta_1 \mid \beta_2$$

Algoritmo 4.2. Fatoramento à esquerda de uma gramática.

Entrada. Gramática G.

Saída. Uma gramática equivalente fatorada à esquerda.

Método. Para cada não-terminal A, encontrar o mais longo prefixo α comum a duas ou mais de suas alternativas. Se $\alpha \neq \epsilon$, isto é, existe um prefixo comum não-trivial, substituir todas as produções de A, $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma$, onde γ representa todas as alternativas que não começam com α, por

$$A \rightarrow \alpha A' \mid \gamma$$
$$A' \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

Aqui, A' é um novo não-terminal. Aplicar repetidamente esta transformação até que não haja duas alternativas com um prefixo comum. □

Exemplo 4.10. A seguinte gramática abstrai o problema do *else-vazio:*

$$S \rightarrow iEtS \mid iEtSeS \mid a$$
$$E \rightarrow b \qquad (4.13)$$

Aqui, i, t e e estão no lugar de **if, then** e **else**, E e S no de "expressão" e "comando", respectivamente. Fatorada à esquerda, esta gramática se torna:

$$S \rightarrow iEtSS' \mid a$$
$$S' \rightarrow eS \mid \epsilon \qquad (4.14)$$
$$E \rightarrow b$$

Dessa forma podemos expandir S para $iEtSS'$ à entrada i e esperar até que $iEtS$ tenha sido visto, para decidir em expandir S' para eS ou ϵ. Naturalmente, as gramáticas (4.13) e (4.14) são ambas ambíguas e, à entrada e, não estará claro que alternativa de S' deverá ser escolhida. O Exemplo 4.19 discute uma forma de se sair desse dilema. □

Construções de Linguagens Não Livres de Contexto

Não deveria soar como uma surpresa que algumas linguagens não possam ser geradas por qualquer gramática. De fato, umas poucas construções sintáticas encontradas em muitas linguagens de programação não podem ser especificadas usando-se somente as gramáticas. Nesta seção, apresentaremos várias dessas construções, usando linguagens abstratas simples para ilustrar essas dificuldades.

Exemplo 4.11. Consideremos que a linguagem abstrata $L_1 = \{wcw \mid w$ está em $(a \mid b)^*\}$. L_1 é constituído de todas as palavras compostas de uma cadeia de a's e b's repetidos, separados por um c, tais como $aabcaab$. Pode ser provado que esta linguagem não é livre de contexto. Esta linguagem abstrai o problema de verificar que os identificadores num programa sejam declarados antes de seus usos. Ou seja, o primeiro w em wcw representa a declaração de um identificador w. O segundo representa o seu uso. Conquanto esteja além do escopo deste livro prová-lo, a não liberdade de contexto de L_1 implica diretamente a não liberdade de contexto de linguagens como Algol e Pascal, que requerem declarações de identificadores antes de seus usos e que permitem identificadores de comprimento arbitrário.

Por esta razão, uma gramática para a sintaxe de Algol ou de Pascal não especifica os caracteres dos identificadores. Ao invés disso, são representados na gramática por um *token*, como **id.** Num compilador para uma gramática desse tipo, a análise semântica verifica se os identificadores foram declarados antes de serem usados. □

Exemplo 4.12. A linguagem $L_2 = \{a^n b^m c^n d^m \mid n \geq 1$ e $m \geq 1\}$ não é livre de contexto. Ou seja, L_2 é constituído de cadeias de caracteres na linguagem gerada pela expressão regular $a^*b^*c^*d^*$, tais que os números de a's e de c's sejam iguais e também os números de b's e d's. (Relembremos que a^n significa a escrito n vezes.) L_2 abstrai o problema de se verificar que o número de parâmetros formais na declaração de um procedimento coincida com o número de parâmetros atuais no uso do mesmo. Isto é, a^n e b^m poderiam representar as listas de parâmetros formais em dois procedimentos declarados como tendo n e m argumentos, respectivamente. Conseqüentemente, c^n e d^m representam as listas de parâmetros nas chamadas a esses dois procedimentos.

De novo, notemos que a sintaxe típica das definições e usos de procedimentos não se preocupa em contar o número de parâmetros. Por exemplo, o enunciado CALL numa linguagem ao estilo de Fortran poderia ser descrito

$$cmd \rightarrow \textbf{call id}\ (lista_de_expressões)$$
$$lista_de_expressões \rightarrow lista_de_expressões\ ,\ expr \mid expr$$

com produções adequadas para *expr*. A verificação de que o número de parâmetros atuais está correto na chamada é usualmente feita durante a fase de análise semântica. □

Exemplo 4.13. A linguagem $L_3 = \{a^n b^n c^n \mid n \geq 0\}$, isto é, cadeias em $L(a^*b^*c^*)$ com um número igual de a's, b's e c's, não é livre de contexto. Um exemplo de problema que envolve L_3 é o que se segue. Um texto composto tipograficamente usa o itálico onde ordinariamente um texto datilografado usaria o sublinhado. Na conversão de um arquivo, com texto destinado a ser impresso através de impressora de linhas, para um outro, com texto adequado à composição fotográfica de tipos, é necessário se substituir as palavras sublinhadas por palavras em itálico. Uma palavra sublinhada é uma cadeia de letras seguida por um número igual de retrocessos e um número igual de travessões. Se considerarmos a como uma letra qualquer, b como retrocesso e c como travessão, a linguagem L_3 representa palavras sublinhadas. A conclusão é que não podemos usar uma gramática para descrever palavras sublinhadas dessa maneira. Por outro lado, se definirmos uma palavra sublinhada como uma seqüência de triplas letra-retrocesso-travessão poderemos representar as palavras sublinhadas através da expressão regular $(abc)^*$. □

É interessante notar que linguagens muito semelhantes a L_1, L_2 e L_3 são livres de contexto. Por exemplo, $L'_1 = \{wcw^R \mid w$ está em $(a \mid b)^*\}$, onde w^R significa w na ordem inversa, é livre de contexto. É gerada pela gramática

$$S \rightarrow aSa \mid bSb \mid c$$

A linguagem $L'_2 = \{a^n b^m c^m d^n \mid n \geq 1$ e $m \geq 1\}$ é livre de contexto, tendo por gramática

$$S \rightarrow aSd \mid aAd$$
$$A \rightarrow bAc \mid bc$$

Igualmente, $L''_2 = \{a^n b^n c^m d^m \mid n \geq 1$ e $m \geq 1\}$ é livre de contexto, tendo por gramática

$$S \rightarrow AB$$
$$A \rightarrow aAb \mid ab$$
$$B \rightarrow cBd \mid cd$$

Finalmente, $L'_3 = \{a^n b^n \mid n \geq 1\}$ é livre de contexto, tendo por gramática

$$S \rightarrow aSb \mid ab$$

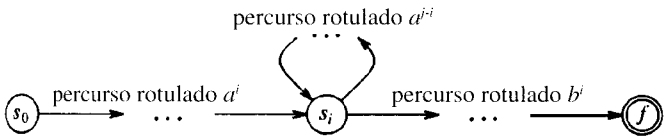


Fig. 4.8. AFD D que aceita a^ib^i e a^jb^i.

É importante notar que L'_3 é o protótipo do exemplo de uma linguagem não definível por qualquer expressão regular. Para confirmar isso, suponhamos que L'_3 seja uma linguagem definida por alguma expressão regular. Equivalentemente, suponhamos que pudéssemos construir um AFD D que aceitasse L'_3. D precisa ter algum número finito de estados, digamos k. Consideremos a seqüência de estados $s_0, s_1, s_2, \ldots, s_k$ seguidos por D, tendo-se lido ϵ, a, aa, aaa,..., a^k. Ou seja, s_i é o estado atingido por D tendo-se lido i a's.

Como D possui somente k estados diferentes, pelo menos dois estados na seqüência $s_0, s_1,\ldots,s_k$ precisam ser os mesmos, digamos s_i e s_j. A partir do estado s_j, uma seqüência de i b's leva D a um estado de aceitação f, uma vez que a^ib^i está em L'_3. Mas então existe também um percurso a partir do estado inicial s_0 até s_i até f rotulado a_jb_i, como mostrado na Fig. 4.8. Dessa forma, D também aceita a^jb^i que não está em L'_3, contradizendo a suposição de que L'_3 é a linguagem aceita por D.

Coloquialmente, dizemos que "um autômato finito não pode realizar contagens", significando que não pode aceitar uma linguagem como L'_3, que exigiria que o mesmo mantivesse uma contagem do número de a's antes de enxergar os b's. Similarmente, dizemos que uma "gramática pode manter uma contagem de dois itens mas não de três", uma vez que com uma gramática podemos definir L'_3 mas não L_3.

4.4 ANÁLISE SINTÁTICA *TOP-DOWN*

Nesta seção, introduzimos as idéias básicas por trás da análise sintática *top-down* e mostramos como construir uma forma eficiente de analisador sintático *top-down* sem retrocesso, chamado de analisador sintático preditivo. Definimos a classe de gramáticas LL(1) a partir da qual os analisadores preditivos podem ser construídos automaticamente. Além de formalizar a discussão dos analisadores preditivos da Seção 2.4, consideramos analisadores sintáticos preditivos não recursivos. Esta seção conclui com uma discussão da recuperação de erros. Os analisadores sintáticos *bottom-up* são discutidos nas Seções 4.5 – 4.7.

Análise Sintática de Descendência Recursiva*

A análise sintática *top-down* pode ser vista como uma tentativa de se encontrar uma derivação mais à esquerda para uma cadeia de entrada. Equivalentemente, pode ser vista como uma tentativa de se construir uma árvore gramatical, para a cadeia de entrada, a partir da raiz, criando os nós da árvore gramatical em pré-ordem. Na Seção 2.4, discutimos um caso especial da análise sintática de descendência recursiva, a análise sintática preditiva, onde nenhum retrocesso era exigido. Consideramos agora uma forma geral de análise sintática *top-down*, chamada de descendência recursiva, que pode envolver retrocesso, ou seja, a realização de esquadrinhamentos repetidos da entrada. Por outro lado, os analisadores sintáticos com retrocesso não são vistos muito freqüentemente. Uma razão está em que o retrocesso é raramente necessitado para analisar sintaticamente construções de linguagens de programação. Em situações tais como a análise sintática de linguagens naturais, o retrocesso ainda é ineficiente e métodos tabulares, tais como o algoritmo de programação dinâmica do Exercício 4.63, ou o método de Earley [1970], são preferidos. Ver Aho e Ullman [1972b] para uma descrição dos métodos gerais de análise sintática.

O retrocesso é exigido no próximo exemplo, e iremos sugerir uma forma de controlar a entrada quando o mesmo ocorrer.

*O termo recursivo-descendente também é usado ao longo deste livro. (N. do T.)

Exemplo 4.14. Consideremos a gramática

$$\begin{aligned} S &\rightarrow cAd \\ A &\rightarrow ab \mid a \end{aligned} \tag{4.15}$$

e a cadeia de entrada $w = cad$. Para construir uma árvore gramatical para esta cadeia, de cima para baixo, criamos inicialmente uma árvore consistindo de um único nó rotulado S. O apontador da entrada aponta para c, o primeiro símbolo de w. Em seguida, usamos a primeira produção para S a fim de expandir a árvore e obter a da Fig. 4.9(a).

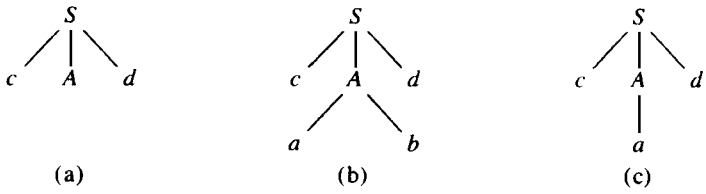


Fig. 4.9. Etapas numa análise sintática *top-down*.

A folha mais à esquerda, rotulada c, reconhece o primeiro símbolo de w e, por conseguinte, avançamos o apontador da entrada para a, o segundo símbolo de w, e consideramos a próxima folha, rotulada A. Em seguida, expandimos A usando a sua primeira alternativa, obtendo a árvore da Fig. 4.9(b). Temos agora um reconhecimento para o segundo símbolo da entrada e, conseqüentemente, avançamos o apontador da entrada para d, o terceiro símbolo da entrada, e comparamos d com a próxima folha, rotulada b. Como b não é igual a d, reportamos uma falha e retornamos a A a fim de verificar se existe uma outra alternativa que não tenhamos tentado ainda, mas que poderia produzir um reconhecimento.

Ao irmos de volta para A, precisamos restabelecer o apontador da entrada para a posição 2, aquela que o mesmo detinha quando passamos pela primeira vez por A, o que significa que o procedimento para A (análogo ao procedimento para não-terminais na Fig. 2.17) precisa armazenar o apontador da entrada numa variável local. Tentamos agora a segunda alternativa de A a fim de obter a árvore na Fig. 4.9(c). A folha a reconhece o segundo símbolo de w e a folha d o terceiro. Uma vez que produzimos uma árvore gramatical para w, paramos e anunciamos o término com sucesso da análise sintática. □

Uma gramática recursiva à esquerda pode levar um analisador sintático de descendência recursiva, mesmo com retrocesso, a um laço infinito. Isto é, quando tentamos expandir A, podemos eventualmente nos encontrar de novo tentando expandir A sem ter consumido nenhum símbolo da entrada.

Analisadores Sintáticos Preditivos

Em muitos casos, escrevendo-se cuidadosamente uma gramática, eliminando-se a recursão à esquerda e fatorando-se à esquerda a gramática resultante, podemos obter uma nova gramática processável por um analisador sintático de descendência recursiva que não necessite de retrocesso, isto é, um analisador sintático preditivo, discutido na Seção 2.4. Para construir um analisador sintático preditivo, precisamos conhecer, dado o símbolo corrente de entrada a e o não-terminal A a ser expandido, qual das alternativas da produção $A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$ é a única que deriva uma cadeia começando por a. Ou seja, a alternativa adequada precisa ser detectável examinando-se apenas para o primeiro símbolo da cadeia que a mesma deriva. As construções de controle de fluxo na maioria das linguagens de programação, com suas palavras-chave distintivas, são usualmente detectáveis dessa forma. Por exemplo, se tivermos as produções.

$$\begin{aligned} cmd \rightarrow\ & \textbf{if}\ expr\ \textbf{then}\ cmd\ \textbf{else}\ cmd \\ \mid\ & \textbf{while}\ expr\ \textbf{do}\ cmd \\ \mid\ & \textbf{begin}\ lista_de_comandos\ \textbf{end} \end{aligned}$$

então as palavras-chave **if, while** e **begin** nos informam qual alternativa é a única que possivelmente teria sucesso, se quiséssemos encontrar um comando.

Diagramas de Transições para Analisadores Sintáticos Preditivos

Na Seção 2.4, discutimos a implementação de analisadores sintáticos preditivos através de procedimentos recursivos, como, por exemplo, aqueles da Fig. 2.17. Exatamente dentro do mesmo espírito com que um diagrama de transições foi considerado um plano ou fluxograma útil para o analisador léxico, podemos criar um diagrama de transições como um plano ou fluxograma útil para um analisador sintático preditivo.

As várias diferenças entre os diagramas de transições para um analisador léxico e um analisador sintático preditivo são imediatamente aparentes. No caso de um analisador sintático, existe um diagrama para cada não-terminal. Os rótulos dos lados são *tokens* e não-terminais. Uma transição em um *token* (terminal) significa que devemos realizá-la se aquele *token* for o próximo símbolo da entrada. Uma transição num não-terminal A é uma chamada do procedimento para A.

Para construir um diagrama de transições de um analisador sintático preditivo a partir de uma gramática, eliminamos primeiro da gramática a recursividade à esquerda e, em seguida, a fatoramos à esquerda. Para cada não-terminal A, então, fazemos o seguinte:

1. Criamos um estado inicial e um final (de retorno).
2. Para cada produção $A \rightarrow X_1X_2...X_n$, criamos um percurso a partir do estado inicial até o estado final, com os lados rotulados $X_1, X_2, ..., X_n$.

O analisador preditivo ao trabalhar sobre os diagramas de transições se comporta como segue. Começa no estado inicial para o símbolo de partida. Se após algumas ações estiver no estado s, o qual possui um lado rotulado pelo terminal a apontando para o estado t, e se o próximo símbolo de entrada for a, move o cursor de entrada uma posição à direita e vai para o estado t. Se, de outra feita, o lado for rotulado pelo não-terminal A, vai para o estado de partida de A, sem movimentar o cursor da entrada. Se em algum instante for atingido o estado final de A, vai imediatamente para o estado t, tendo, com efeito, "lido" A a partir da entrada, durante o tempo em que se movia do estado s para t. Finalmente, se existir um lado de s para t rotulado ϵ, vai, a partir do estado s, imediatamente para o estado t, sem avançar na entrada.

Um programa de análise sintática preditiva baseado num diagrama de transições tenta reconhecer símbolos terminais na entrada e faz uma chamada de procedimento potencialmente recursiva sempre que precisar seguir um lado rotulado por um não-terminal. Uma implementação não-recursiva pode ser obtida empilhando-se o estado s quando existir uma transição em um não-terminal para fora de s e removendo-se o topo da pilha quando o estado final para o não-terminal for atingido. Discutiremos a implementação dos diagramas de transições em mais detalhes brevemente.

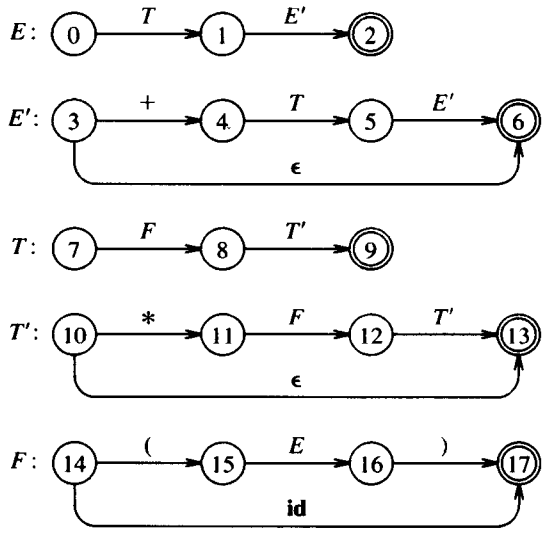


Fig. 4.10. Diagramas de transições para a gramática (4.11).

A abordagem acima funcionará se o diagrama de transições dado for determinístico, isto é, não existir mais de uma transição de um mesmo estado para outros à mesma entrada. Se a ambigüidade ocorrer, deveremos estar capacitados a resolvê-la de uma forma *ad-hoc*, como no próximo exemplo. Se o não-determinismo não puder ser eliminado, não poderemos construir um analisador sintático preditivo, mas poderemos construir um analisador de descendência recursiva com retrocesso, de forma a tentar sistematicamente todas as possibilidades, se esta fosse a melhor estratégia de análise que pudéssemos encontrar.

Exemplo 4.15. A Fig. 4.10 contém uma coleção de diagramas de transições para a gramática (4.11). As únicas ambigüidades unicamente dizem respeito a se seguir ou não um lado-ϵ. Se interpretarmos os lados para fora do estado inicial E' como dizendo realizar a transição em + sempre que este for o próximo símbolo de entrada e seguir a transição em ϵ em caso contrário, e fizermos suposições análogas para T', a ambigüidade será removida, e poderemos escrever um programa de análise preditiva para a gramática (4.11). □

Os diagramas de transições podem ser simplificados pela substituição dos diagramas uns pelos outros; essas substituições são similares às transformações feitas nas gramáticas usadas na Seção 2.5. Por exemplo, na Fig. 4.11(a), a chamada de E' foi substituída por um desvio para o início do diagrama para E'.

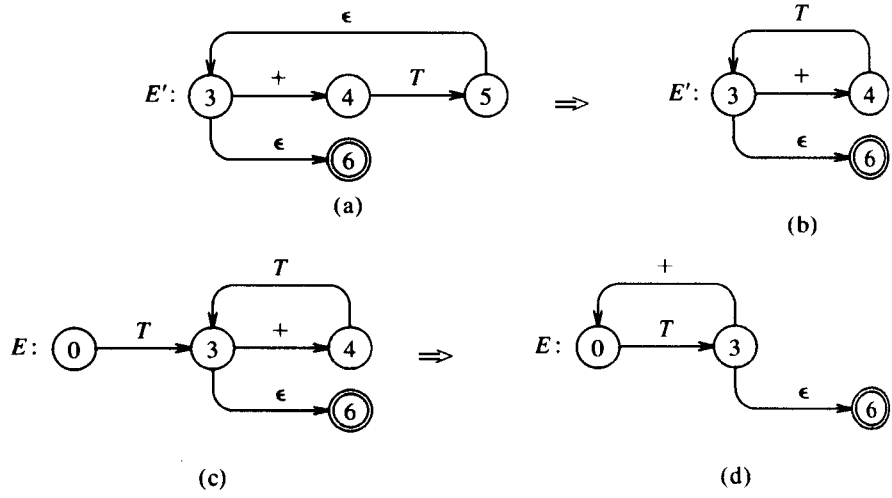


Fig. 4.11. Diagramas de transições simplificados.

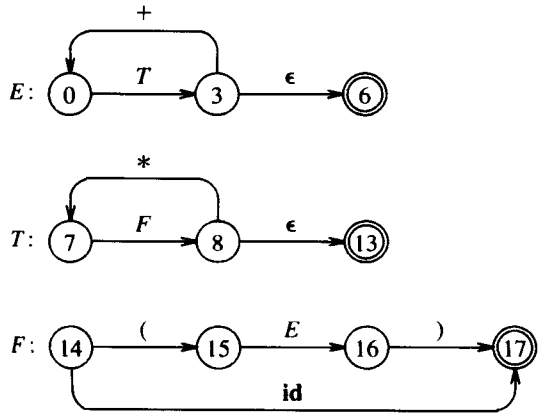


Fig. 4.12. Diagramas de transições simplificados para expressões aritméticas.

A Fig. 4.11(b) mostra um diagrama de transições equivalente para E'. Podemos substituir o diagrama da Fig. 4.11(b) pela transição em E', no diagrama para E, na Fig. 4.10, produzindo o diagrama da Fig. 4.11(c). Por fim, observamos que os primeiro e terceiro nós na Fig. 4.11(c) são equivalentes e os combinamos. O resultado, a Fig. 4.11(d), é repetido como o primeiro diagrama da Fig. 4.12. As mesmas técnicas se aplicam aos diagramas para T e T'. O conjunto completo de diagramas resultantes é mostrado na Fig. 4.12. Uma implementação C para esse analisador preditivo roda de 20 a 25% mais rapidamente do que uma implementação C para o da Fig. 4.10.

Análise Sintática Preditiva Não-Recursiva

É possível construir um analisador preditivo não-recursivo mantendo explicitamente uma pilha, ao invés de implicitamente através de chamadas recursivas. O problema-chave durante a análise preditiva é determinar que produção deve ser aplicada a um dado não-terminal. O analisador não-recursivo da Fig. 4.13 procura pela produção a ser aplicada numa tabela sintática. No que se segue, iremos ver como a tabela pode ser construída diretamente a partir de certas gramáticas.

Um analisador sintático preditivo dirigido por uma tabela possui um *buffer* de entrada, uma pilha, uma tabela sintática e um fluxo de saída. O *buffer* de entrada possui a cadeia a ser analisada, seguida por um \$ à direita para indicar o fim da cadeia de entrada. A pilha contém uma seqüência de símbolos gramaticais, com \$ indicando o fundo da pilha. Inicialmente, a pilha contém o símbolo de partida da gramática acima de \$. Uma tabela sintática é um *array* bidimensional $M[A, a]$, onde A é um não-terminal e a é um terminal ou o símbolo \$.

O analisador sintático é controlado por um programa que se comporta como segue. O programa considera X o símbolo ao topo da pilha e a o símbolo corrente de entrada. Esses dois símbolos determinam a ação do analisador. Existem três possibilidades:

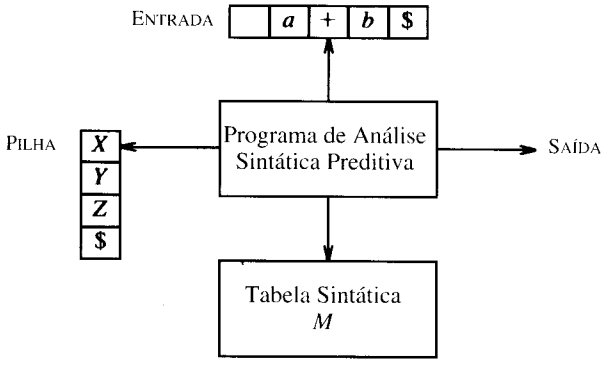


Fig. 4.13. Modelo de um analisador sintático preditivo não-recursivo.

1. Se $X = a = \$$, o analisador pára e anuncia o término com sucesso da análise sintática.
2. Se $X = a \neq \$$, o analisador sintático remove X da pilha e avança o apontador da entrada para o próximo símbolo.
3. Se X é um não-terminal, o programa consulta a entrada $M[X, a]$ da tabela sintática M. Essa entrada será uma produção-X da gramática ou uma entrada de erro. Se, por exemplo, $M[X, a] = \{X \rightarrow UVW\}$, o analisador substitui X no topo da pilha por WVU (com U ao topo). Como saída, iremos assumir que o analisador sintático simplesmente imprima a produção usada; de fato, qualquer outro código poderia ser executado aqui. Se $M[X, a]$ = **erro,** o analisador chama uma rotina de recuperação de erros.

O comportamento do analisador sintático pode ser descrito em termos de suas *configurações*, que dão o conteúdo da pilha e a entrada restante.

Algoritmo 4.3. Análise sintática preditiva não recursiva.

Entrada. Uma cadeia w e uma tabela sintática M para a gramática G.

Saída. Se w estiver em $L(G)$, uma derivação mais à esquerda de w; caso contrário, uma indicação de erro.

Método. Inicialmente, o analisador sintático está numa configuração na qual possui somente $\$S$ na pilha, com S, o símbolo de partida de G ao topo e $w\$$ no *buffer* de entrada. O programa, que utiliza a tabela sintática preditiva M para realizar uma análise sintática da entrada é mostrado na Fig. 4.14. □

```
faça ip apontar para o primeiro símbolo de w$;
repetir
        seja X o símbolo ao topo da pilha e a o símbolo apontado por ip;
        se X for um terminal ou $ então
                se X = a então
                        remover X da pilha e avançar ip
                        senão erro( )
        senão /* X é um não-terminal */
                se M[X, a] = X → Y1Y2 . . . Yk então início
                                remover X da pilha;
                                empilhar Yk, Yk-1, . . . Y1, com Y1 ao topo
                                da pilha;
                                escrever a produção X → Y1Y2 . . . Yk
                fim
                senão erro( )
até que X = $           /* a pilha está vazia */
```

Fig. 4.14. Programa de análise sintática preditiva.

Exemplo 4.16. Considere a gramática (4.11) do Exemplo 4.8. Uma tabela sintática preditiva para a mesma é mostrada na Fig. 4.15. Os brancos são entradas de erro; não-brancos indicam uma produção com a qual se deve expandir o não-terminal ao topo da pilha. Note que ainda não indicamos como essas entradas podem ser selecionadas, mas iremos fazê-lo brevemente.

Com a entrada **id** + **id** * **id** o analisador preditivo realiza a seqüência de movimentos da Fig. 4.16. O apontador da entrada aponta para o símbolo mais à esquerda da cadeia na coluna ENTRADA. Se observarmos cuidadosamente as ações deste analisador, veremos que o mesmo está rastreando uma derivação mais à esquerda da entrada, isto é, as produções escritas são aquelas de uma derivação mais à esquerda. Os símbolos de entrada que já foram esquadrinhados, seguidos pelos símbolos gramaticais na pilha (de cima para baixo), constituem as formas sentenciais mais à esquerda na derivação. □

PRIMEIRO e SEGUINTE

A construção de um analisador sintático preditivo é auxiliada por duas funções associadas à gramática G. Essas funções, PRIMEIRO e SE-

NÃO-TERMINAL	SÍMBOLO DE ENTRADA					
	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow$ **id**			$F \rightarrow (E)$		

Fig. 4.15. Tabela sintática M para a gramática (4.11).

PILHA	ENTRADA	SAÍDA
$\$E$	**id** + **id** * **id**$	
$\$E'T$	**id** + **id** * **id**$	$E \rightarrow TE'$
$\$E'T'F$	**id** + **id** * **id**$	$T \rightarrow FT'$
$\$E'T'$**id**	**id** + **id** * **id**$	F → **id**
$\$E'T'$	+ **id** * **id**$	
$\$E'$	+ **id** * **id**$	$T' \rightarrow \epsilon$
$\$E'T+$	+ **id** * **id**$	$E' \rightarrow +TE'$
$\$E'T$	**id** * **id**$	
$\$E'T'F$	**id** * **id**$	$T \rightarrow FT'$
$\$E'T'$**id**	**id** * **id**$	$F \rightarrow$ **id**
$\$E'T'$	* **id**$	
$\$E'T'F*$	* **id**$	$T' \rightarrow *FT'$
$\$E'T'F$	**id**$	
$\$E'T'$**id**	**id**$	$F \rightarrow$ **id**
$\$E'T'$	$	
$\$E'$	$	$T' \rightarrow \epsilon$
$	$	$E' \rightarrow \epsilon$

Fig. 4.16. Movimentos feitos pelo analisador sintático preditivo para a entrada **id** + **id** * **id**.

GUINTE, nos permitem preencher as entradas de uma tabela sintática preditiva para G, sempre que possível. Os conjuntos de *tokens* produzidos pela função SEGUINTE podem também ser usados como *tokens* de sincronização durante a recuperação de erros na modalidade do desespero.

Se α for qualquer cadeia de símbolos gramaticais, seja *PRIMEIRO*(α) o conjunto de terminais que começam as cadeias derivadas a partir de α. Se $\alpha \overset{*}{\Rightarrow} \epsilon$ então ϵ também está em PRIMEIRO(α).

Definamos *SEGUINTE(A)*, para o não-terminal A, como sendo o conjunto de terminais a que podem figurar imediatamente à direita de A em alguma forma sentencial, isto é, o conjunto de terminais a tais que exista uma derivação de forma $S \overset{*}{\Rightarrow} \alpha A a \beta$, para algum α e β. Note que podem ter existido, em algum tempo durante a derivação, símbolos entre A e a, mas, se assim o foi, os mesmos derivaram ϵ e desapareceram. Se A puder ser o símbolo mais à direita em alguma forma sentencial, então $ está em SEGUINTE*(A)*

Para computar PRIMEIRO(X) para todos os símbolos gramaticais X, aplique as seguintes regras até que nenhum terminal ou ϵ possa ser adicionado a qualquer conjunto PRIMEIRO.

1. Se X for um terminal, então PRIMEIRO*(X)* é $\{X\}$.

2. Se $X \rightarrow \epsilon$ for uma produção, adicionar ϵ a PRIMEIRO(X).

3. Se X for um não-terminal e $X \rightarrow Y_1 Y_2 ... Y_k$ uma produção, colocar a em PRIMEIRO(X) se, para algum i, a estiver em PRIMEIRO(Y_i) e ϵ estiver em todos PRIMEIRO(Y_1), ..., PRIMEIRO(Y_{i-1}); isto é, se $Y_1 ... Y_{i-1} \overset{*}{\Rightarrow} \epsilon$. Se ϵ estiver em PRIMEIRO(Y_j) para todos os $j = 1, 2, ..., k$, adicione, então, ϵ a PRIMEIRO(X). Por exemplo, tudo o que estiver em PRIMEIRO(Y_1) estará certamente em PRIMEIRO(X). Se Y_1 não derivar ϵ, então não adicionamos mais nada a PRIMEIRO(X); mas se $Y_1 \overset{*}{\Rightarrow} \epsilon$, e precisamos adicionar PRIMEIRO(Y_2) e assim por diante.

Agora, podemos computar PRIMEIRO para qualquer cadeia $X_1 X_2 ... X_n$ como segue. Adicionar a PRIMEIRO($X_1 X_2 ... X_n$) todos os símbolos não-ϵ de PRIMEIRO (X_1). Adicionar, também, todos os símbolos não-ϵ de PRIMEIRO (X_2) se ϵ estiver em PRIMEIRO(X_1), os símbolos não-ϵ de primeiro(X_3) se ϵ estiver em ambos, PRIMEIRO(X_1) e PRIMEIRO(X_2), e assim por diante. Finalmente, adicionar ϵ a PRIMEIRO($X_1 X_2 ... X_n$) se, para todos os valores i, PRIMEIRO(X_i) contiver ϵ.

Para computar SEGUINTE(A) para todos os não-terminais A, aplique as seguintes regras até que nada mais possa ser adicionado a qualquer conjunto SEGUINTE.

1. Colocar $ em SEGUINTE(S), onde S é o símbolo de partida e $ o marcador de fim de entrada à direita.

2. Se existir uma produção $A \rightarrow \alpha B \beta$, então tudo em PRIMEIRO(β), exceto ϵ, é colocado em SEGUINTE(B).

3. Se existir uma produção $A \rightarrow \alpha B$ ou uma produção $A \rightarrow \alpha B \beta$ onde PRIMEIRO(β) contém ϵ (isto é, $\beta \overset{*}{\Rightarrow} \epsilon$), então, tudo em SEGUINTE(A) está em SEGUINTE(B).

Exemplo 4.17. Consideremos a gramática (4.11), repetida abaixo:

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

Então:

PRIMEIRO (E) = PRIMEIRO (T) = PRIMEIRO (F) = {(, **id**}.

PRIMEIRO (E') = {+, ϵ}

PRIMEIRO (T') = {*, ϵ}

SEGUINTE (E) = SEGUINTE(E') = {), $}

SEGUINTE (T) = SEGUINTE(T') = {+,),$}

SEGUINTE (F) = {+, *), $}

Por exemplo, **id** e o parêntesis à esquerda são adicionados a PRIMEIRO(F) pela regra (3) na definição de PRIMEIRO, com $i = 1$ em cada caso, uma vez que PRIMEIRO(**id**) = {**id**} e PRIMEIRO ('(') = { (} pela regra (1). Pela regra (3), com $i = 1$, a produção $T = FT'$ implica que **id** e o parênteses à esquerda estão em PRIMEIRO(T) igualmente. Como um outro exemplo, ϵ está em PRIMEIRO(E') pela regra (2).

Para computar os conjuntos SEGUINTE, colocamos $ em SEGUINTE(E) pela regra (1) para SEGUINTE. Pela regra (2), aplicada à produção $F \rightarrow (E)$, o parênteses à direita está em SEGUINTE (E'). Pela regra (3), aplicada à produção $E \rightarrow TE'$, o $ e o parêntesis direito são em SEGUINTE (E'). Como $E' \overset{*}{\Rightarrow} \epsilon$, também estão em SEGUINTE(T). Como um último exemplo de como as regras para SEGUINTE podem ser aplicadas, a produção $E \rightarrow TE'$ implica, pela regra (2), que tudo que não seja ϵ em PRIMEIRO (E') precisa ser colocado em SEGUINTE(T). Já vimos que $ está em SEGUINTE(T). □

Construção de Tabelas Sintáticas Preditivas

O próximo algoritmo pode ser usado para construir uma tabela sintática preditiva para uma gramática G. A idéia por trás do algoritmo é a seguinte: suponhamos que $A \rightarrow \alpha$ seja uma produção com a em PRIMEIRO(α); por conseguinte, o analisador sintático irá expandir A através de α quando o símbolo de entrada corrente for a. A única complicação ocorre quando $\alpha = \epsilon$ ou $\alpha \stackrel{*}{\Rightarrow} \epsilon$. Nesse caso, devemos expandir A de novo através de α se o símbolo corrente de entrada estiver em SEGUINTE(A) ou se o \$ na entrada foi atingido e \$ está em SEGUINTE(A).

Algoritmo 4.4. Construção de uma tabela sintática preditiva.

Entrada. Gramática G.

Saída. Tabela sintática M.

Método.

1. Para cada produção $A \rightarrow \alpha$ da gramática, execute os passos 2 e 3.
2. Para cada terminal a em PRIMEIRO(α), adicione $A \rightarrow \alpha$ a $M[A, a]$.
3. Se ϵ estiver em PRIMEIRO(α), adicione $A \rightarrow \alpha$ a $M[A, b]$, para cada terminal b em SEGUINTE(A). Se ϵ estiver em PRIMEIRO(α) e \$ em SEGUINTE(A), adicione $A \rightarrow \alpha$ a $M[A, \$]$.
4. Faça cada entrada indefinida de M ser **erro.**

Exemplo 4.18. Vamos aplicar o algoritmo 4.4 à gramática (4.11). Como PRIMEIRO(TE') = PRIMEIRO(T) = {(, **id**}, a produção $E \rightarrow TE'$ faz com que $M[E, (]$ e $M[E, \mathbf{id}]$ adquiram a entrada $E \rightarrow TE'$.

A produção $E' \rightarrow +TE'$ causa $M[E',+]$ adquirir $E' \rightarrow +TE'$. A produção $E' \rightarrow \epsilon$ causa $M[E',(]$ e $M[E',\$]$ adquirirem $E' \rightarrow \epsilon$ uma vez que SEGUINTE(E') = {),\$}.

A tabela sintática produzida pelo algoritmo 4.4 para a gramática (4.11) foi mostrada na Fig. 4.15. □

Gramáticas LL(1)

O algoritmo 4.4 pode ser aplicado a qualquer gramática G para produzir uma tabela sintática M. Para algumas gramáticas, entretanto, algumas entradas serão multiplamente definidas. Por exemplo, se G for recursiva à esquerda ou ambígua, M terá pelo menos uma entrada multiplamente definida.

Exemplo 4.19. Vamos considerar a gramática (4.13) do Exemplo 4.10 de novo; a mesma é repetida aqui por uma questão de conveniência.

$$
\begin{aligned}
S &\rightarrow iEtSS' \mid a \\
S' &\rightarrow eS \mid e \\
E &\rightarrow b
\end{aligned}
$$

A tabela sintática para esta gramática é mostrada na Fig. 4.17.

A entrada para $M[S', e]$ contém tanto $S' \rightarrow eS$ e $S' \rightarrow \epsilon$, uma vez que SEGUINTE(S') = {e, \$}. A gramática é ambígua e a ambigüidade é manifestada pela escolha sobre que produção usar quando um e (**else**) for enxergado. Podemos resolver a ambigüidade se escolhermos $S' \rightarrow eS$. Esta escolha corresponde a associar os **else**'s aos **then**'s anteriores mais próximos. Note-se que a escolha $S' \rightarrow \epsilon$ impediria para sempre que e viesse a ser colocado na pilha ou removido da entrada, e, por conseguinte, está seguramente errada. □

NÃO-TERMINAL	SÍMBOLO DE ENTRADA					
	a	b	e	i	t	\$
S	$S \rightarrow a$			$S \rightarrow iEtSS'$		
S'			$S' \rightarrow \epsilon$ $S' \rightarrow eS$			$S' \rightarrow \epsilon$
E		$E \rightarrow b$				

Fig. 4.17. Tabela sintática M para a gramática (4.13).

Uma gramática cuja tabela sintática não possui entradas multiplamente definidas é dita *LL*(1). O primeiro "L" em LL(1) significa a varredura da entrada da esquerda para a direita (*left to right*); o segundo, a produção de uma derivação mais à esquerda (*left linear*); e o "1", o uso de um único símbolo de entrada como *lookahead* a cada passo para tomar as decisões sintáticas. Pode ser mostrado que o algoritmo 4.4 produz, para cada gramática LL(1) G, uma tabela sintática que decompõe todas e somente as sentenças de G.

As gramáticas LL(1) possuem várias propriedades distintivas. Nenhuma gramática ambígua ou recursiva à esquerda pode ser LL(1). Pode também ser mostrado que uma gramática G é LL(1) se, e somente se, sempre que $A \rightarrow \alpha \mid \beta$ forem duas produções distintas de G, vigorarem as seguintes condições:

1. α e β não derivem, ao mesmo tempo, cadeias começando pelo mesmo terminal a, qualquer que seja a.
2. No máximo um dos dois, α ou β, derive a cadeia vazia.
3. Se $\beta \stackrel{*}{\Rightarrow} \epsilon$, então α não deriva qualquer cadeia começando por um terminal em SEGUINTE(A).

Claramente, a gramática (4.11) para expressões aritméticas é LL(1). A gramática (4.13), modelando comandos *if-then-else*, não o é.

Resta a questão sobre o que deveria ser feito quando a tabela sintática possuir entradas multiplamente definidas. Um recurso é o de transformar a gramática através da eliminação de toda a recursividade à esquerda e a subseqüente fatoração à esquerda, sempre que possíveis, na esperança de produzir uma gramática para a qual a tabela sintática não possua entradas multiplamente definidas. Infelizmente, existem algumas gramáticas para as quais nenhuma alteração irá produzir uma gramática LL(1). A gramática (4.13) é um desses casos; sua linguagem não possui gramática LL(1). Como vimos, podemos ainda decompor (4.13) através de um analisador sintático preditivo, fazendo arbitrariamente $M[S',e] = \{S' \rightarrow eS\}$. Em geral, não existem regras universais pelas quais as entradas multiplamente definidas possam se tornar univocamente definidas sem afetar a linguagem reconhecida pelo analisador sintático.

A dificuldade principal em se usar a análise preditiva está na escrita de uma gramática para a linguagem-fonte tal que um analisador sintático preditivo possa ser construído a partir da mesma. Apesar da eliminação da recursividade à esquerda e da fatoração à esquerda serem fáceis de aplicar, ambas tornam a gramática resultante difícil de ler e usar para os fins da tradução. Para avaliar algumas dessas dificuldades, uma forma de organização comum para um analisador sintático de um compilador está em se usar um analisador preditivo para construções de controle e usar a precedência de operadores (discutida na Seção 4.6) para as expressões. Entretanto, se um gerador de analisadores sintáticos LR estiver disponível, como o discutido na Seção 4.9, pode-se obter automaticamente os benefícios da análise preditiva e de precedência de operadores.

Recuperação de Erros na Análise Preditiva

A pilha de um analisador preditivo não-recursivo torna explícitos os terminais e não-terminais que o mesmo espera reconhecer com o restante da entrada. Iremos conseqüentemente nos referir aos símbolos na pilha do analisador na discussão que se segue. Um erro é detectado durante a análise preditiva quando o terminal ao topo da pilha não reconhece o próximo símbolo de entrada ou quando o não-terminal A está ao topo da pilha, a é o próximo símbolo de entrada e a entrada da tabela sintática $M[A,a]$ está vazia.

A recuperação de erros na modalidade do desespero está baseada na idéia de se pular símbolos na entrada até que surja um *token* pertencente a um conjunto pré-selecionado de *tokens* de sincronização. Sua efetividade depende da escolha do conjunto de sincronização. Os conjuntos deveriam ser escolhidos de tal forma que o analisador se recuperasse rapidamente dos erros que tendessem a ocorrer na prática. Algumas técnicas heurísticas são:

1. Como ponto de partida, podemos colocar todos os símbolos de SEGUINTE(A) no conjunto de *tokens* de sincronização para o não-terminal A. Se pularmos *tokens* até que um elemento de SEGUINTE(A) seja visto e removermos A da pilha, é provável que a análise sintática possa continuar.
2. Não é suficiente usar SEGUINTE(A) como o conjunto de sincronização para A. Por exemplo, se os pontos-e-vírgulas terminarem os enunciados, como em C, então as palavras-chave que iniciam os enunciados não devem aparecer no conjunto SEGUINTE do não-terminal que gera expressões. Um ponto-e-vírgula ausente após uma atribuição pode conseqüentemente resultar na omissão da palavra-chave que inicia o próximo enunciado. Freqüentemente, existe uma estrutura hierárquica nas construções da linguagem; por exemplo, as expressões aparecem dentro de enunciados, que figuram dentro de blocos e assim por diante. Podemos adicionar ao conjunto de sincronização de uma construção mais baixa os símbolos que começam as construções mais altas. Por exemplo, poderíamos adicionar palavras-chave que iniciam comandos aos conjuntos de sincronização para os não-terminais que geram expressões.
3. Se adicionarmos os símbolos em PRIMEIRO(A) ao conjunto de sincronização para o não-terminal A, pode ser possível retornar a análise a partir de A, se um símbolo em PRIMEIRO(A) figurar na entrada.
4. Se um não-terminal puder gerar a cadeia vazia, então a produção que deriva ϵ pode ser usada como *default*. Agindo-se assim, pode-se postergar a detecção de algum erro, mas não se pode fazer com que um erro seja perdido. Esse enfoque reduz o número de não-terminais que tenham de ser considerados durante a recuperação de erros.
5. Se um terminal ao topo da pilha não puder ser reconhecido, uma idéia simples é a de removê-lo, emitir uma mensagem informando da remoção e prosseguir a análise sintática. Com efeito, este enfoque faz com que o conjunto de sincronização de um *token* consista em todos os demais *tokens*.

Exemplo 4.20. Usar os símbolos de SEGUINTE e PRIMEIRO como *tokens* de sincronização funciona razoavelmente bem quando as expressões são decompostas de acordo com a gramática (4.11). A tabela sintática da Fig. 4.15 para esta gramática é repetida na Fig. 4.18, com "sinc" indicando os *tokens* de sincronização obtidos a partir do conjunto SEGUINTE do não-terminal em exame. Os conjuntos SEGUINTE para o não-terminal são obtidos a partir do Exemplo 4.17.

A tabela na Fig. 4.18 deve ser usada como segue. Se o analisador sintático procurar pela entrada $M[A, a]$ e encontrar que a mesma está em branco, então o símbolo a é pulado. Se a entrada for "sinc", então o não-terminal ao topo da pilha é removido numa tentativa de se retomar a análise sintática. Se um *token* ao topo da pilha não reconhecer o símbolo de entrada, é removido da pilha, como mencionado acima.

NÃO-TERMINAL	SÍMBOLO DE ENTRADA					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$	sinc	sinc
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$	sinc		$T \to FT'$	sinc	sinc
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to$ **id**	sinc	sinc	$F \to (E)$	sinc	sinc

Fig. 4.18. *Tokens* de sincronização adicionados à tabela sintática da Fig. 4.15.

PILHA	ENTRADA	COMENTÁRIO
$E	) **id** * + **id**$	erro, pular)
$E	**id** * + **id**$	**id** está em PRIMEIRO (E)
$E'T	**id** * + **id**$	
$E'T'F	**id** * + **id**$	
$E'T'**id**	**id** * + **id**$	
$E'T'	* + **id**$	
$E'T'F*	* + **id**$	
$E'T'F	+ **id**$	erro, $M[F, +]$ = sinc
$E'T'	+ **id**$	F foi removido do topo da pilha
$E'	+ **id**$	
$E'T+	+ **id**$	
$E'T	**id**$	
$E'T'F	**id**$	
$E'T'**id**	**id**$	
$E'T'	$	
$E'	$	
$	$	

Fig. 4.19. Movimentos da análise sintática e recuperação de erros feitos pelo analisador sintático preditivo.

À entrada incorreta)**id***+**id**, o analisador sintático e o mecanismo de recuperação de erros da Fig. 4.18 se comportam como na Fig. 4.19. □

A discussão acima, da recuperação na modalidade do desespero, não endereça o importante tema das mensagens de erro. Em geral, as mensagens de erro informativas têm que ser fornecidas pelo projetista do compilador.

Recuperação em nível de frases. A recuperação em nível de frases é implementada preenchendo-se as entradas em branco da tabela sintática preditiva com apontadores para rotinas de erro. Essas rotinas podem modificar, inserir ou remover símbolos da entrada e emitir as mensagens de erro apropriadas. Podem também remover o topo da pilha. É questionável se deveríamos permitir a alteração dos símbolos da pilha ou empilharmos novos símbolos, uma vez que os passos dados pelo analisador poderiam não corresponder à derivação de palavra alguma da linguagem. Em qualquer circunstância, devemos estar certos de que não há possibilidade de um laço infinito. A verificação de que uma ação de recuperação resulte num símbolo de entrada sendo consumido (ou a pilha sendo encurtada se o final da entrada tiver sido atingido) é uma boa forma de nos protegermos contra tais laços.

4.5 ANÁLISE SINTÁTICA *BOTTOM-UP*

Nesta seção, introduzimos o estilo geral de análise sintática *bottom-up*, conhecido como análise de empilhar e reduzir. Uma forma fácil de implementá-la, chamada de análise sintática de precedência de operadores, é apresentada na Seção 4.6. Um método muito mais geral de análise de empilhar e reduzir, chamado de análise LR, é discutido na Seção 4.7. A decomposição LR é usada em vários geradores automáticos de analisadores sintáticos.

A análise gramatical de empilhar e reduzir tenta construir uma árvore gramatical para uma cadeia de entrada começando pelas folhas (o fundo) e trabalhando árvore acima em direção à raiz (o topo). Podemos pensar neste processo como o de "reduzir" uma cadeia w ao símbolo de partida de uma gramática. A cada passo de *redução*, uma subcadeia particular, que reconheça o lado direito de uma produção, é substituída pelo símbolo à esquerda daquela produção e, se a subcadeia tiver sido escolhida corretamente a cada passo, uma derivação mais à direita terá sido rastreada na ordem inversa.

Exemplo 4.21. Considere a gramática

$$S \to aABe$$
$$A \to Abc \mid b$$
$$B \to d$$

A sentença *abbcde* pode ser reduzida a *S* pelos seguintes passos:

$$\begin{array}{l} abbcde \\ aAbcde \\ aAde \\ aABe \\ S \end{array}$$

Podemos esquadrinhar *abbcde* procurando por uma subcadeia que reconheça o lado direito de alguma produção. As subcadeias *b* e *d* se qualificam. Vamos escolher o *b* mais à esquerda e substituí-lo por *A*, o lado esquerdo da produção $A \rightarrow b$; obtemos dessa forma a cadeia *aAbcde*. Agora as subcadeias *Abc*, *b* e *d* reconhecem o lado direito de alguma produção. Apesar de *b* ser a subcadeia mais à esquerda que reconheça o lado direito de alguma produção, escolhemos substituir a subcadeia *Abc* por *A*, o lado esquerdo da produção $A \rightarrow Abc$. Obtemos agora *aAde*. Com a substituição de *d* por *B*, o lado esquerdo da produção $B \rightarrow d$, obtemos *aABe*. Podemos agora substituir toda esta cadeia por *S*. Conseqüentemente, através de uma seqüência de quatro reduções, estamos capacitados a reduzir *abbcde* a *S*. Essas reduções, de fato, rastreiam a seguinte derivação mais à direita, na ordem reversa:

$$S \underset{mad}{\Rightarrow} aABe \underset{mad}{\Rightarrow} aAde \underset{mad}{\Rightarrow} aAbcde \underset{mad}{\Rightarrow} abbcde \qquad \square$$

Handles

Informalmente, um *handle* é uma subcadeia que reconhece o lado direito de uma produção e cuja redução ao não-terminal do lado esquerdo da produção representa um passo ao longo do percurso de uma derivação mais à direita. Em muitos casos, a subcadeia β mais à esquerda que reconhece o lado direito de uma produção $A \rightarrow \beta$ não é um *handle*, porque uma redução pela produção $A \rightarrow \beta$ produz uma cadeia que não pode ser reduzida ao símbolo de partida. No Exemplo 4.21, se substituíssemos *b* por *A* na segunda cadeia *aAbcde* obteríamos a cadeia *aAAcde* que não pode ser subseqüentemente reduzida a *S*. Por esta razão, precisamos fornecer uma definição mais precisa de um *handle*.

Formalmente, um *handle* de uma forma sentencial mais à direita g é uma produção $A \rightarrow \beta$ e uma posição dentro de γ onde a cadeia β possa ser encontrada e substituída por *A* de forma a produzir a forma sentencial à direita anterior numa derivação mais à direita de γ. Ou seja, se $S \overset{*}{\underset{mad}{\Rightarrow}} \alpha A w \overset{*}{\underset{mad}{\Rightarrow}} \alpha\beta w$, então, $A \rightarrow \beta$, na posição seguinte, a α, é um handle de $\alpha\beta w$. A cadeia *w* à direita do *handle* contém somente símbolos terminais. Note-se que dissemos "um *handle*" ao invés de "o *handle*" porque a gramática poderia ser ambígua, com mais de uma derivação mais à direita para $\alpha\beta w$. Se a gramática for inambígua, cada forma sentencial à direita da gramática possui exatamente um *handle*.

No exemplo acima, *abbcde* é uma forma sentencial à direita cujo *handle* é $A \rightarrow b$ à posição 2. Algumas vezes dizemos que "a subcadeia β é um *handle* de $\alpha\beta w$" se a posição de β e a produção $A \rightarrow \beta$ que tivermos em vista forem claras.

A Fig. 4.20 retrata o *handle* $A \rightarrow \beta$ na árvore gramatical da forma sentencial à direita $\alpha\beta w$. O *handle* representa a subárvore completa mais à esquerda, consistindo em um nó e todos os seus filhos. Na Fig. 4.20, *A* é o nó interior mais ao fundo e mais à esquerda com todos os seus filhos na árvore. A redução de β para *A* em $\alpha\beta w$ pode ser pensada como a "poda do *handle*", ou seja, a remoção dos filhos de *A* da árvore gramatical.

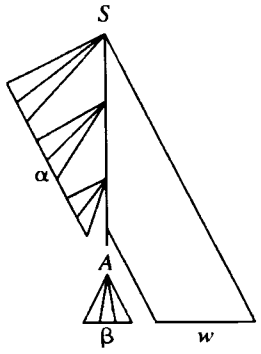


Fig. 4.20. O *handle* $A \rightarrow \beta$ na árvore gramatical de $\alpha\beta w$.

Exemplo 4.22. Considere a seguinte gramática

$$\begin{array}{ll} (1) & E \rightarrow E + E \\ (2) & E \rightarrow E * E \\ (3) & E \rightarrow (E) \\ (4) & E \rightarrow \mathbf{id} \end{array} \qquad (4.16)$$

e a derivação mais à direita

$$\begin{aligned} E &\underset{mad}{\Rightarrow} \underline{E + E} \\ &\underset{mad}{\Rightarrow} E + \underline{E * E} \\ &\underset{mad}{\Rightarrow} E + E * \underline{\mathbf{id}_3} \\ &\underset{mad}{\Rightarrow} E + \underline{\mathbf{id}_2} * \mathbf{id}_3 \\ &\underset{mad}{\Rightarrow} \underline{\mathbf{id}_1} + \mathbf{id}_2 * \mathbf{id}_3 \end{aligned}$$

Subscrevemos os **id**'s por uma conveniência de notação e sublinhamos o *handle* de cada forma sentencial à direita. Por exemplo, $\mathbf{id}_1$ é um *handle* da forma sentencial $\mathbf{id}_1 + \mathbf{id}_2 * \mathbf{id}_3$ porque **id** é o lado direito da produção $E \rightarrow \mathbf{id}$ e a substituição de $\mathbf{id}_1$ por *E* produz a forma sentencial prévia $E + \mathbf{id}_2 * \mathbf{id}_3$. Note-se que a cadeia à direita de um *handle* contém somente símbolos terminais.

Como a gramática (4.16) é ambígua, existe uma outra derivação mais à direita para a mesma cadeia:

$$\begin{aligned} E &\underset{mad}{\Rightarrow} \underline{E * E} \\ &\underset{mad}{\Rightarrow} E * \underline{\mathbf{id}_3} \\ &\underset{mad}{\Rightarrow} \underline{E + E} * \mathbf{id}_3 \\ &\underset{mad}{\Rightarrow} E + \underline{\mathbf{id}_2} * \mathbf{id}_3 \\ &\underset{mad}{\Rightarrow} \underline{\mathbf{id}_1} + \mathbf{id}_2 * \mathbf{id}_3 \end{aligned}$$

Consideremos a forma sentencial à direita $E + E * \mathbf{id}_3$. Nesta derivação, $E + E$ é um *handle* de $E + E * \mathbf{id}_3$, enquanto que $\mathbf{id}_3$ por si mesmo é um *handle* desta mesma forma sentencial à direita, de acordo com a derivação acima.

As duas derivações mais à direita neste exemplo são análogas às duas derivações mais à esquerda no Exemplo 4.6. A primeira derivação confere a * uma maior precedência do que a +, enquanto que a segunda dá a + a maior precedência. $\square$

A Poda do *Handle*

Uma derivação mais à direita na ordem inversa pode ser obtida "podando-se os *handles*". Os seja, começamos pela primeira cadeia de terminais *w* que desejamos decompor. Se *w* for uma sentença da gramática em questão, então $w = \gamma_n$, onde γ_n é a enésima forma sentencial à direita de alguma derivação mais à direita ainda desconhecida

$$S = \gamma_0 \underset{mad}{\Rightarrow} \gamma_1 \underset{mad}{\Rightarrow} \gamma_2 \underset{mad}{\Rightarrow} \ldots \underset{mad}{\Rightarrow} \gamma_{n-1} \underset{mad}{\Rightarrow} \gamma_n = w.$$

Para reconstruir esta derivação na ordem inversa, localizamos o *handle* β_n em γ_n e substituímos β_n pelo lado direito de alguma produção $A_n \rightarrow \beta_n$, de modo a obtermos a enésima menos uma forma sentencial à direita γ_{n-1}. Note-se que ainda não sabemos como os *handles* são encontrados, mas em breve veremos os métodos para tal.

Repetimos, em seguida, esse processo. Isto é, localizamos o *handle* β_{n-1} em γ_{n-1} e o reduzimos de forma a obter a forma sentencial à direita γ_{n-2}. Continuando esse processo, produzimos uma forma sentencial à direita consistindo somente no símbolo de partida *S* e en-

tão paramos e anunciamos o término com sucesso da análise sintática. O reverso da seqüência de produções usadas nas reduções é uma derivação mais à direita para a cadeia de entrada.

Exemplo 4.23. Considere a gramática (4.16) do Exemplo 4.22 e a cadeia de entrada $\mathbf{id}_1 + \mathbf{id}_2 * \mathbf{id}_3$. A seqüência de reduções mostrada na Fig. 4.21 reduz $\mathbf{id}_1 + \mathbf{id}_2 * \mathbf{id}_3$ ao símbolo de partida E. O leitor deveria observar que a seqüência de formas sentenciais à direita neste exemplo é somente o reverso da seqüência da primeira derivação mais à direita no Exemplo 4.22. □

FORMA SENTENCIAL À DIREITA	HANDLE	PRODUÇÃO REDUTORA
$\mathbf{id}_1 + \mathbf{id}_2 * \mathbf{id}_3$	$\mathbf{id}_1$	$E \rightarrow \mathbf{id}$
$E + \mathbf{id}_2 * \mathbf{id}_3$	$\mathbf{id}_2$	$E \rightarrow \mathbf{id}$
$E + E * \mathbf{id}_3$	$\mathbf{id}_3$	$E \rightarrow \mathbf{id}$
$E + E * E$	$E * E$	$E \rightarrow E * E$
$E + E$	$E + E$	$E \rightarrow E + E$
E		

Fig. 4.21. Reduções realizadas por analisador sintático de empilhar e reduzir.

Implementação de Pilha da Análise Sintática de Empilhar e Reduzir

Existem dois problemas que precisam ser resolvidos se estivermos dispostos a analisar sintaticamente através da poda de *handles*. O primeiro é o de localizar a subcadeia a ser reduzida numa forma sentencial à direita e o segundo é o de determinar que produção escolher no caso de existir mais de uma produção com aquela subcadeia no lado direito. Antes de entrarmos nessas questões, vamos considerar os tipos de estruturas de dados usadas num analisador sintático de empilhar e reduzir.

Uma forma conveniente de implementar um analisador sintático de empilhar e reduzir é usar uma pilha para guardar os símbolos gramaticais e um *buffer* de entrada para a cadeia w a ser decomposta. Usamos \$ para marcar o fundo da pilha e também o final à direita da entrada. Inicialmente, a pilha está vazia e a cadeia w está à entrada como segue

PILHA	ENTRADA
\$	w \$

O analisador sintático opera empilhando zero ou mais símbolos (na pilha) até que um *handle* β surja no topo da pilha. Reduz, então, β para o lado esquerdo da produção apropriada. Repete este ciclo até que tenha detectado um erro ou que a pilha contenha o símbolo de partida e a entrada esteja vazia:

PILHA	ENTRADA
\$$S$	\$

Após entrar nesta configuração, pára e anuncia término com sucesso da análise sintática.

Exemplo 4.24. Vamos rastrear as ações que um analisador sintático de empilhar e reduzir realizaria ao decompor a cadeia de entrada $\mathbf{id}_1 + \mathbf{id}_2 * \mathbf{id}_3$, de acordo com a gramática (4.16), usando a primeira derivação do Exemplo 4.22. A seqüência é mostrada na Fig. 4.22. Note que, como a gramática (4.16) possui duas derivações mais à direita para a entrada, existe uma outra seqüência de passos que um analisador sintático de empilhar e reduzir poderia seguir. □

Conquanto as operações primárias do analisador sejam obviamente empilhar e reduzir, existem efetivamente quatro ações possíveis

	PILHA	ENTRADA	AÇÃO
(1)	\$	$\mathbf{id}_1 + \mathbf{id}_2 * \mathbf{id}_3$\$	empilhar
(2)	\$$\mathbf{id}_1$	$+ \mathbf{id}_2 * \mathbf{id}_3$\$	reduzir por $E \rightarrow \mathbf{id}$
(3)	\$$E$	$+ \mathbf{id}_2 * \mathbf{id}_3$\$	empilhar
(4)	\$$E +$	$\mathbf{id}_2 * \mathbf{id}_3$\$	empilhar
(5)	\$$E + \mathbf{id}_2$	$* \mathbf{id}_3$\$	reduzir por $E \rightarrow \mathbf{id}$
(6)	\$$E + E$	$* \mathbf{id}_3$\$	empilhar
(7)	\$$E + E *$	$\mathbf{id}_3$\$	empilhar
(8)	\$$E + E * \mathbf{id}_3$	\$	reduzir por $E \rightarrow \mathbf{id}$
(9)	\$$E + E * E$	\$	reduzir por $E \rightarrow E * E$
(10)	\$$E + E$	\$	reduzir por $E \rightarrow E + E$
(11)	\$$E$	\$	aceitar

Fig. 4.22. Configurações de um analisador sintático de empilhar e reduzir para a entrada $\mathbf{id}_1 + \mathbf{id}_2 * \mathbf{id}_3$.

que o mesmo pode realizar: (1) empilhar, (2) reduzir, (3) aceitar e (4) erro.

1. Numa ação de *empilhar*, o próximo símbolo de entrada é colocado no topo da pilha.
2. Numa ação de *reduzir*, o analisador sabe que o final à direita de um *handle* está no topo da pilha. Precisa, então, localizar o início à esquerda do *handle* dentro da pilha e decidir qual não-terminal irá substituir o *handle*.
3. Numa ação de *aceitar*, o analisador anuncia o término com sucesso da operação de decomposição.
4. Numa ação de *erro*, o analisador descobre que um erro sintático ocorreu e chama uma rotina de recuperação de erros.

Existe um fato importante que justifica o uso de uma pilha numa análise sintática de empilhar e reduzir; o *handle* irá sempre aparecer no topo da pilha, nunca dentro da mesma. Este fato se torna óbvio quando consideramos as possíveis formas que dois passos numa derivação mais à direita podem ter. Os dois passos podem ser da forma

$$(1)\quad S \underset{mad}{\overset{*}{\Rightarrow}} \alpha A z \underset{mad}{\overset{*}{\Rightarrow}} \alpha\beta B y z \underset{mad}{\overset{*}{\Rightarrow}} \alpha\beta\gamma y z$$

$$(2)\quad S \underset{mad}{\overset{*}{\Rightarrow}} \alpha B x A z \underset{mad}{\overset{*}{\Rightarrow}} \alpha B x y z \underset{mad}{\overset{*}{\Rightarrow}} \alpha\gamma x y z$$

No caso (1), A é substituído por βBy e em seguida B, o terminal mais à direita naquele lado direito, é substituído por γ. No caso (2), A é de novo substituído em primeiro lugar, mas, desta vez, o lado direito é uma cadeia somente de terminais. O próximo não-terminal mais à direita B estará em algum lugar à esquerda de y.

Vamos considerar o caso (1) ao contrário, onde um analisador de empilhar e reduzir acabou de atingir a configuração

PILHA	ENTRADA
\$$\alpha\beta\gamma$	yz\$

O analisador agora reduz o *handle* γ para B e atinge a configuração

PILHA	ENTRADA
\$$\alpha\beta B$	yz\$

Como B é o não-terminal mais à direita em $\alpha\beta By z$, o fim à direita do *handle* de $\alpha\beta By z$ não pode ocorrer dentro da pilha.* Por conseguinte, o analisador pode empilhar a cadeia y de modo a atingir a configuração

PILHA	ENTRADA
\$$\alpha\beta By$	z\$

na qual βBy é o *handle*, o qual se vê reduzido a A.

*Uma vez que a substituição realizada imediatamente antes deste passo na derivação mais à direita trocou necessariamente um não-terminal à direita de B ou o próprio B. (N. do T.)

No caso (2), na configuração

PILHA	ENTRADA
$\$\alpha\gamma$	$xyz\$$

o *handle* γ está no topo da pilha. Após reduzi-lo a B, o analisador pode empilhar a cadeia xy de modo a obter o próximo *handle* y no topo da pilha:

PILHA	ENTRADA
$\$\alpha Bxy$	$z\$$

E agora reduz y para A.

Em ambos os casos, após realizar uma redução, o analisador sintático teve que empilhar zero ou mais símbolos de modo a ter o próximo *handle* no topo da pilha. Jamais teve que ir dentro da pilha para encontrá-lo. É este aspecto da poda do *handle* que torna a pilha uma estrutura de dados particularmente conveniente para implementar um analisador sintático de empilhar e reduzir. Precisamos ainda explicar como devem ser feitas as escolhas das ações a serem realizadas de forma que o analisador sintático de empilhar e reduzir funcione corretamente. A precedência de operadores e os analisadores sintáticos LR são duas dessas técnicas que brevemente iremos discutir.

Prefixos Viáveis

Os prefixos de uma forma sentencial à direita que podem figurar na pilha de um analisador sintático de empilhar e reduzir são chamados de *prefixos viáveis*. Uma definição equivalente de um prefixo viável é a de ser um prefixo de uma forma sentencial à direita, o qual não se estende para além do limite à direita do *handle* mais à direita, daquela forma sentencial. Por esta definição, é sempre possível adicionar símbolos terminais ao final de um prefixo viável de modo a obter uma forma sentencial à direita. Por conseguinte, não há aparentemente erro na medida em que a porção da entrada enxergada até um dado ponto possa ser reduzida a um prefixo viável.

Conflitos Durante a Análise Sintática de Empilhar e Reduzir

Existem gramáticas livres de contexto para as quais a análise de empilhar e reduzir não pode ser usada. Cada analisador de empilhar e reduzir para uma tal gramática pode atingir uma configuração na qual, mesmo conhecendo o conteúdo de toda a pilha e o próximo símbolo de entrada, não pode decidir entre empilhar ou reduzir (um *conflito empilhar/reduzir*) ou não pode decidir qual das diversas reduções alternativas realizar (um *conflito reduzir/reduzir*). Damos agora alguns exemplos de construções sintáticas que dão origem a tais gramáticas. Tecnicamente, essas gramáticas não estão na classe de gramáticas LR(k), definida na Seção 4.7; referimo-nos às mesmas como gramáticas não-LR. O k em LR(k) se refere ao número de símbolos de *lookahead* na entrada. As gramáticas usadas na compilação usualmente caem na classe LR(1), com um único símbolo *lookahead*.

Exemplo 4.25. Uma gramática ambígua jamais poderá ser LR. Por exemplo, consideremos a gramática (4.7) do *else*-vazio, na Seção 4.3:

$$
\begin{aligned}
cmd \rightarrow\ & \textbf{if}\ expr\ \textbf{then}\ cmd \\
|\ & \textbf{if}\ expr\ \textbf{then}\ cmd\ \textbf{else}\ cmd \\
|\ & \textbf{outro}
\end{aligned}
$$

Se tivermos um analisador sintático de empilhar e reduzir na configuração

PILHA	ENTRADA
... **if** *expr* **then** *cmd*	**else** ... $

não podemos dizer se **if** *expr* **then** *cmd* é o *handle*, não importa o que apareça abaixo do mesmo na pilha. Aqui há um conflito empilhar/reduzir. Dependendo do que siga o **else** na entrada, poderia ser correto reduzir **if** *expr* **then** *cmd* a *cmd* ou empilhar o **else** e em seguida procurar por outro *cmd* para completar a alternativa **if** *expr* **then** *cmd* **else** *cmd*. Conseqüentemente, não podemos decidir entre empilhar ou reduzir neste caso, e a gramática não é LR(1). Mais geralmente, nenhuma gramática ambígua, como é certamente esse caso, pode ser LR(k) para qualquer que seja o valor de k.

Deveríamos mencionar, entretanto, que a análise sintática de empilhar e reduzir pode ser facilmente adaptada para decompor certas gramáticas ambíguas, tais como a gramática acima do *if-then-else*. Quando construirmos um analisador para uma gramática contendo as duas produções acima, haverá um conflito empilhar/reduzir: no **else**, empilhar ou, alternativamente, reduzir através de *cmd* → **if** *expr* **then** *cmd*. Se resolvermos o conflito em favor de empilhar, o analisador sintático irá se comportar naturalmente. Discutimos os analisadores sintáticos para gramáticas ambíguas na Seção 4.8. □

Outra causa comum da não-linearidade à direita ocorre quando sabemos ter um *handle*, mas o conteúdo da pilha e o próximo símbolo de entrada não são suficientes para determinar que produção deveria ser usada numa redução. O próximo exemplo ilustra esta situação.

Exemplo 4.26. Suponhamos ter um analisador léxico que retorne o *token* **id** para todos os identificadores, independentemente do uso. Suponhamos, também, que nossa linguagem invoque procedimentos fornecendo seu nome, com os parâmetros envolvidos entre parênteses, e que os *arrays* sejam referenciados através da mesma sintaxe. Como a tradução dos índices nas referências aos *arrays* e dos parâmetros nas chamadas de procedimentos são diferentes, desejamos usar produções distintas para gerar as listas de parâmetros atuais e de índices. Nossa gramática poderia ter, por conseguinte, produções (dentre outras) tais como:

(1)	$cmd \rightarrow$ **id** (*lista_de_parâmetros*)
(2)	$cmd \rightarrow expr := expr$
(3)	*lista_de_parâmetros* → *lista_de_parâmetros*, *parâmetro*
(4)	*lista_de_parâmetros* → *parâmetro*
(5)	*parâmetro* → **id**
(6)	*expr* → **id** (*lista_expr*)
(7)	*expr* → **id**
(8)	*lista_expr* → *lista_expr*, *expr*
(9)	*lista_expr* → *expr*

Um enunciado, começando por `A(I, J)`, apareceria para o analisador sintático como o fluxo de *tokens* **id(id, id)**. Após empilhar os três primeiros *tokens*, um analisador de empilhar e reduzir estaria na configuração

PILHA	ENTRADAS
. . . **id (id**	**, id) . . .**

É evidente que o **id** ao topo da pilha deveria ser reduzido, mas para que produção? A escolha correta seria a produção (5) se `A` fosse um procedimento e (7) se fosse um *array*. A pilha não nos diz qual; informações na tabela de símbolos obtidas a partir da declaração de `A` precisam ser usadas.

Uma solução é a de mudar o *token* **id** na produção (1) para **procid** e usar um analisador léxico mais sofisticado que retorne o *token* **procid** ao reconhecer um identificador que seja o nome de um procedimento. Fazer isso requer que o analisador léxico consulte a tabela de símbolos antes de retornar um *token*.

Se fizermos a modificação, no processamento de `A(I, J)` o analisador sintático estaria ou na configuração

PILHA	ENTRADAS
. . . procid (id	**, id) . . .**

ou na configuração mostrada anteriormente. No caso anterior, escolhemos a redução através da produção (5); neste último, através da produção (7). Observe como o terceiro símbolo a partir do topo da pilha de-

termina a redução a ser realizada, ainda que não esteja envolvido na redução. A análise sintática de empilhar e reduzir pode se utilizar de informações muito abaixo na pilha para guiar a decomposição. □

4.6 ANÁLISE SINTÁTICA DE PRECEDÊNCIA DE OPERADORES

A mais ampla classe de gramáticas, para a qual os analisadores sintáticos de empilhar e reduzir podem ser construídos com sucesso — as gramáticas LR —, será discutida na Seção 4.7. Entretanto, para uma pequena, porém importante, classe de gramáticas, podemos facilmente construir manualmente eficientes analisadores sintáticos de empilhar e reduzir. Essas gramáticas possuem a propriedade (dentre outras exigências essenciais) de que nenhum lado direito de produção seja ϵ, ou tenha dois não-terminais adjacentes. Uma gramática com a última propriedade é chamada de uma *gramática de operadores*.

Exemplo 4.27. A seguinte gramática para expressões

$$E \rightarrow EAE \mid (E) \mid -E \mid \mathbf{id}$$
$$A \rightarrow + \mid - \mid * \mid / \mid \uparrow$$

não é uma gramática de operadores porque o lado direito EAE possui dois (de fato três) não-terminais consecutivos. Entretanto, se substituirmos A por cada uma de suas alternativas, obtemos a seguinte gramática de operadores:

$$E \rightarrow E + E \mid E - E \mid E * E \mid E/E \mid E \uparrow E \mid (E) \mid -E \mid \mathbf{id} \qquad (4.17)$$

Descrevemos agora uma técnica de análise sintática fácil de implementar chamada de análise sintática de precedência de operadores. Historicamente, esta técnica foi primeiramente descrita como uma manipulação sobre *tokens* sem qualquer referência a uma gramática subjacente. De fato, uma vez que tenhamos terminado de construir um analisador sintático de precedência de operadores a partir de uma gramática, podemos ignorar esta última usando os não-terminais na pilha apenas como marcadores de lugar para os atributos associados aos não-terminais.

Como uma técnica geral de análise sintática, a de precedência de operadores possui uma série de desvantagens. Por exemplo, é difícil tratar os *tokens* como o sinal de menos, que possui duas diferentes precedências (dependendo de ser unário ou binário). Sobretudo, uma vez que o relacionamento entre a gramática para a linguagem analisada e o analisador sintático de precedência de operadores é tênue, não se pode estar sempre certo de que o analisador aceite exatamente a linguagem desejada. Finalmente, somente uma pequena classe de gramáticas pode ser decomposta usando as técnicas de precedência de operadores.

Apesar de tudo, em virtude de sua simplicidade, numerosos compiladores usando as técnicas de análise sintática de precedência de operadores têm sido construídos com sucesso. Freqüentemente, esses analisadores são de descendência recursiva, como descrito na Seção 4.4, para enunciados e outras construções de alto nível. Analisadores sintáticos de precedência de operadores têm sido construídos até mesmo para linguagens inteiras.

Na análise sintática de precedência de operadores, definimos três *relações de precedência* disjuntas, $<\cdot$, $\doteq$ e $\cdot>$, entre certos pares de terminais. Essas relações de precedência guiam a seleção de *handles* e têm os seguintes significados:

RELAÇÃO	SIGNIFICADO
$a <\cdot b$	a "confere precedência" a b
$a \doteq b$	a "possui a mesma precedência que" b
$a \cdot> b$	a "tem precedência sobre" b

Devemos prevenir o leitor de que, conquanto essas relações possam parecer similares às relações aritméticas "menor do que", "igual a" e "maior do que", as relações de precedência possuem propriedades diferentes. Por exemplo, poderíamos ter $a <\cdot b$ e $a \cdot> b$ para a mesma linguagem, ou não termos nenhuma das relações vigorando para o mesmo par de terminais a e b.

Existem duas formas comuns de determinar quais relações de precedência devem vigorar para um par de terminais. O primeiro método que discutimos é intuitivo e está baseado nas noções tradicionais de associatividade e precedência de operadores. Por exemplo, se * deve ter maior precedência do que +, fazemos $+ <\cdot *$ e $* \cdot>$. Este enfoque será visto resolvendo as ambigüidades da gramática (4.17) e nos capacitará a escrever um analisador sintático de precedência de operadores para a mesma (apesar de o sinal de menos unário causar problemas).

O segundo método de selecionar as relações de precedência de operadores consiste em, primeiro, construir, para a linguagem, uma gramática não ambígua, que reflita a associatividade e precedência corretas em suas árvores gramaticais. Esta tarefa não é difícil para as expressões; a sintaxe das expressões na Seção 2.2 providencia um paradigma. Para as outras fontes comuns de ambigüidade, como o **else**-vazia, a gramática (4.9) é um modelo útil. Tendo obtido uma gramática inambígua, existe um método mecânico para se construir relações de precedência de operadores a partir da mesma. Essas relações podem não ser disjuntas e podem decompor uma linguagem que não a gerada pela gramática, mas, com os tipos padronizados de expressões aritméticas, poucos problemas são encontrados na prática. Não iremos discutir esta construção aqui; ver Aho e Ullman [1972b].

Usando Relações de Precedência de Operadores

O objetivo das relações de precedência é delimitar o *handle* de uma forma sentencial à direita, com $<\cdot$ assinalando o limite à esquerda, $\doteq$ figurando no interior do *handle* e $\cdot>$ marcando o limite à direita. Para sermos mais precisos, suponhamos ter uma forma sentencial à direita de uma gramática de operadores. O fato de não haverem dois não-terminais adjacentes nos lados direitos das produções implica igualmente que nenhuma forma sentencial à direita terá dois não-terminais adjacentes. Por conseguinte, podemos escrever a forma sentencial à direita como $\beta_0 a_1 \beta_1 \ldots a_n \beta_n$, onde cada β_i ou é ϵ (a cadeia vazia) ou um único não-terminal e cada a_i é um único terminal.

Suponhamos que entre a_i e a_{i+1}, exatamente uma das relações $<\cdot$, $\doteq$ ou $\cdot>$ vigore. Adicionalmente, vamos usar \$ para assinalar o final da cadeia e definir que $\$ <\cdot b$ e $b \cdot> \$$ para todos os terminais b. Suponhamos agora que tenhamos removido os não-terminais da cadeia e colocado a relação correta, $<\cdot$, $\doteq$ ou $\cdot>$, entre cada par de terminais e entre os terminais das extremidades e os cifrões (\$) que marcam os finais (à esquerda e à direita) da cadeia. Por exemplo, suponhamos que inicialmente tenhamos a forma sentencial à direita **id** + **id** * **id** e as relações de precedência sejam aquelas fornecidas na Fig. 4.23. Essas relações são algumas daquelas que escolheríamos de forma a decompor uma expressão de acordo com a gramática (4.17).

	id	+	*	**$**
id		$\cdot>$	$\cdot>$	$\cdot>$
+	$<\cdot$	$\cdot>$	$<\cdot$	$\cdot>$
*	$<\cdot$	$\cdot>$	$\cdot>$	$\cdot>$
$	$<\cdot$	$<\cdot$	$<\cdot$	

Fig. 4.23. Relações de precedência de operadores.

Conseqüentemente, a cadeia com as relações de precedência inseridas é:

$$\$ <\cdot \mathbf{id} \cdot> + <\cdot \mathbf{id} \cdot> * <\cdot \mathbf{id} \cdot> \$ \qquad (4.18)$$

Por exemplo, $<\cdot$ é inserida entre o \$ mais à esquerda e **id**, uma vez que $<\cdot$ é a entrada na linha \$ e coluna **id.** O *handle* pode ser encontrado pelo seguinte processo:

1. Esquadrinhar a cadeia a partir da extremidade esquerda até que o primeiro $\cdot>$ seja encontrado. Em (4.18) acima, isso ocorre entre o primeiro **id** e +.

2. Esquadrinhar, então, de volta (para a esquerda) por sobre quaisquer relações $\doteq$ até que $<\cdot$ seja encontrada. Em (4.18), esquadrinhamos de volta até o \$.
3. O *handle* contém tudo à esquerda do primeiro $\cdot>$ e à direita do $<\cdot$ encontrado no passo (2), incluindo quaisquer não-terminais intervenientes ou envolventes. (A inclusão de não-terminais envolventes é necessária para que dois não-terminais adjacentes não figurem numa forma sentencial à direita.) Em (4.18), o *handle* é o primeiro **id**.

Se estivermos lidando com a gramática (4.17), reduzimos, então, **id** para E. Neste ponto, temos a forma sentencial E + **id** * **id**. Após reduzirmos os dois **id**'s restantes a E, obtemos a forma sentencial à direita $E + E * E$. Consideremos agora a cadeia \$ + * \$ obtida pela remoção dos não-terminais. Inserindo as relações de precedência, obtemos

$$\$ <\cdot + <\cdot * \cdot> \$$$

indicando que a extremidade à esquerda do *handle* recai entre o + e o * e a extremidade direita entre * e \$. As relações de precedência indicam que, na forma sentencial à direita $E + E * E$, o *handle* é $E * E$. Note-se que os E's envolvendo o * se tornam parte do *handle*.

Uma vez que os não-terminais não influenciam a decomposição, não precisamos nos preocupar com a distinção entre os mesmos. Um único "não-terminal" marcador pode ser mantido na pilha de um analisador sintático de empilhar e reduzir a fim de indicar guardadores de lugar para os valores de atributos.

Pode parecer, a partir da discussão acima, que toda a forma sentencial à direita precise ser esquadrinhada a cada passo para se encontrar o *handle*. Tal não é o caso se usarmos uma pilha para armazenar os símbolos de entrada já esquadrinhados e se as relações de precedência forem usadas para guiar as ações de um analisador sintático de empilhar e reduzir. Se uma das relações de precedência $<\cdot$ ou $\doteq$ vigorar entre o símbolo terminal ao topo da pilha e o próximo símbolo de entrada, o analisador empilha; ainda não se atingiu a extremidade direita do *handle*. Se a relação $\cdot>$ vigorar, uma redução é pedida. A essa altura, o analisador encontrou a extremidade direita do *handle* e as relações de precedência podem ser usadas para encontrar a extremidade esquerda do *handle* dentro da pilha.

Se nenhuma relação de precedência vigorar entre um par de terminais (indicada por uma entrada em branco na Fig. 4.23), então um erro sintático foi detectado e uma rotina de recuperação de erros deve ser invocada, como será discutido mais tarde nesta seção. As idéias acima podem ser formalizadas pelo algoritmo que se segue.

Algoritmo 4.5. Algoritmo para a análise sintática de precedência de operadores.

```
(1)  fazer ip apontar para o primeiro símbolo de w$;
(2)  repetir para sempre
(3)     se $ estiver no topo da pilha e ip apontar para $ então
(4)     retornar
     senão início
(5)     seja a o símbolo terminal ao topo da pilha
          e seja b o símbolo apontado por ip na entrada;
(6)     se a <· b ou a ≐ b então início
(7)       empilhar b;
(8)       avançar ip de forma a apontar para o próximo símbolo
          entrada;
        fim;
(9)     senão se a ·> b então          /* reduzir */
(10)      repetir
(11)        remover o topo da pilha
(12)      até que o terminal ao topo da pilha esteja relacionado
            por <· ao terminal mais recentemente removido
(13)    senão erro ( )
     fim
```

Fig. 4.24. Algoritmo de análise sintática de precedência de operadores.

Entrada. Uma cadeia w e uma tabela de relações de precedência.

Saída. Se w for bem formada, um *esqueleto* de árvore gramatical, com um não-terminal guardador de lugar E rotulando todos os nós interiores; de outra forma, uma indicação de erro.

Método. Inicialmente, a pilha contém \$ e o *buffer* de entrada a cadeia w\$. Para decompor, executamos o programa da Fig. 4.24. □

Relações de Precedência de Operadores a partir da Associatividade e Precedência

Estamos sempre livres para criar relações de precedência de operadores a qualquer ponto em que as vejamos adequadas e esperamos que o algoritmo de análise sintática de precedência de operadores irá funcionar corretamente quando guiado por elas. Para uma linguagem de expressões aritméticas, tal como a gerada pela gramática (4.17), podemos usar o seguinte procedimento heurístico para produzir o conjunto apropriado de relações de precedência. Note-se que a gramática (4.17) é ambígua e que as formas sentenciais à direita poderiam conter vários *handles*. Nossas regras são projetadas para selecionar os *handles* "apropriados" que reflitam um conjunto de regras de precedência e associatividade para operadores binários.

1. Se o operador θ_1 possui maior precedência do que o operador θ_2, fazer $\theta_1 \cdot> \theta_2$ e $\theta_2 <\cdot \theta_1$. Por exemplo, se * tiver maior precedência do que +, fazer $* \cdot> +$ e $+ <\cdot *$. Essas relações asseguram que, numa expressão da forma $E + E * E + E$, o $E * E$ central é o *handle* que será reduzido primeiro.
2. Se θ_1 e θ_2 são operadores de igual precedência (podem de fato ser o mesmo operador), fazer $\theta_1 \cdot> \theta_2$ e $\theta_2 \cdot> \theta_1$, se os operadores forem associativos à esquerda, ou fazer $\theta_1 <\cdot \theta_2$ e $\theta_2 <\cdot \theta_1$, se forem associativos à direita. Por exemplo, se + e − forem associativos à esquerda, então fazer $+ \cdot> +$, $+ \cdot> -$, $- \cdot> -$ e $- \cdot> +$. Se ↑ for associativo à direita, então fazer $\uparrow <\cdot \uparrow$. Essas relações asseguram que $E - E + E$ terá o *handle* $E - E$ selecionado e que $E \uparrow E \uparrow E$ terá o último $E \uparrow E$ selecionado.
3. Fazer $\theta <\cdot$ **id**, **id** $\cdot> \theta$, $\theta <\cdot (, (<\cdot \theta,)) \cdot> \theta$, $\theta \cdot>)$, $\theta \cdot> \$$ e $\$ <\cdot \theta$ para todos os operadores θ. Fazer também

$(\doteq)$	$\$ <\cdot ($	$\$ <\cdot$ **id**
$(<\cdot ($	**id** $\cdot> \$$	$) \cdot> \$$
$(<\cdot$ **id**	**id** $\cdot>)$	$) \cdot>)$

Essas regras asseguram que tanto **id** quanto *(E)* serão reduzidos a E. Igualmente, \$ serve tanto como marcador da extremidade esquerda quanto da direita, fazendo com que os *handles* sejam encontrados entre os cifrões sempre que possível.

Exemplo 4.28. A Fig. 4.25 contém relações de precedência de operadores para a gramática (4.17) assumindo que

1. ↑ possua a maior precedência e seja associativo à direita,

	+	−	*	/	↑	**id**	(	)	\$
+	$\cdot>$	$\cdot>$	$<\cdot$	$<\cdot$	$<\cdot$	$<\cdot$	$<\cdot$	$\cdot>$	$\cdot>$
−	$\cdot>$	$\cdot>$	$<\cdot$	$<\cdot$	$<\cdot$	$<\cdot$	$<\cdot$	$\cdot>$	$\cdot>$
*	$\cdot>$	$\cdot>$	$\cdot>$	$\cdot>$	$<\cdot$	$<\cdot$	$<\cdot$	$\cdot>$	$\cdot>$
/	$\cdot>$	$\cdot>$	$\cdot>$	$\cdot>$	$<\cdot$	$<\cdot$	$<\cdot$	$\cdot>$	$\cdot>$
↑	$\cdot>$	$\cdot>$	$\cdot>$	$\cdot>$	$<\cdot$	$<\cdot$	$<\cdot$	$\cdot>$	$\cdot>$
id	$\cdot>$	$\cdot>$	$\cdot>$	$\cdot>$	$\cdot>$			$\cdot>$	$\cdot>$
(	$<\cdot$	$<\cdot$	$<\cdot$	$<\cdot$	$<\cdot$	$<\cdot$	$<\cdot$	$\doteq$	
)	$\cdot>$	$\cdot>$	$\cdot>$	$\cdot>$	$\cdot>$			$\cdot>$	$\cdot>$
\$	$<\cdot$	$<\cdot$	$<\cdot$	$<\cdot$	$<\cdot$	$<\cdot$	$<\cdot$		

Fig. 4.25. Relações de precedência de operadores.

2. * e / tenham a próxima precedência mais baixa e sejam associativos à esquerda e
3. + e − tenham a mais baixa precedência e sejam associativos à esquerda.

(Brancos denotam entradas de erro.) O leitor deveria tentar compreender a tabela e verificar que a mesma funciona corretamente, ignorando o problema com o menos unário por enquanto. Tente testar a tabela, por exemplo, para a entrada **id * (id ↑ id) − id / id.** □

Tratando as Operações Unárias

Se tivermos um operador unário tal como ¬ (negação lógica), que também não seja um operador binário, podemos incorporá-lo ao esquema acima, de modo a criar relações de precedência de operadores. Supondo que ¬ seja um operador unário infixo, fazemos ¬ $\cdot>$ θ, se ¬ tiver maior precedência do que θ, e ¬ $<\cdot$ θ em caso contrário. Por exemplo, se ¬ possuir maior precedência do que & e & for associativo à esquerda, podemos através dessas regras agrupar $E\&\neg E\&E$ como $(E\&(\neg E))\&E$. A regra para os operadores unários posfixos é análoga.

Essa situação muda quando temos um operador como o sinal —, que é tanto prefixo unário quando infixo binário. Mesmo se atribuirmos a mesma precedência aos operadores unário e binário, a tabela da Fig. 4.25 irá falhar ao analisar gramaticalmente cadeias como **id*–id** de forma correta. A melhor abordagem neste caso é usar o analisador léxico para distinguir entre o menos unário e o binário, fazendo com que retorne um *token* diferente ao examinar um menos unário. Infelizmente, o analisador léxico não pode usar o *lookahead* para distinguí-los; precisa relembrar o *token* anterior. Em Fortran, por exemplo, um sinal de menos é considerado unário se o *token* anterior for um operador, um parênteses à esquerda, uma vírgula ou um símbolo de atribuição.

Funções de Precedência

Os compiladores que usam analisadores gramaticais de precedência de operadores não precisam armazenar a tabela de relações de precedência. Na maioria das vezes, a tabela pode ser codificada por duas *funções de precedência*, f e g, que mapeiam símbolos terminais em inteiros. Devemos selecionar f e g de tal forma que, para quaisquer símbolos a e b,

1. $f(a) < g(b)$, sempre que $a <\cdot\, b$,
2. $f(a) = g(b)$, sempre que $a \doteq b$
3. $f(a) > g(b)$, sempre que $a \cdot> b$.

Por conseguinte, a relação de precedência entre a e b pode ser determinada através de uma comparação numérica entre $f(a)$ e $g(b)$. Note, entretanto, que as entradas em erro na matriz de precedência são obscuras, uma vez que uma das relações, (1), (2) ou (3), sempre vigora, não importando o que $f(a)$ e $g(b)$ sejam. A perda na capacidade de detecção de erros não é considerada séria o suficiente para impedir o uso das relações de precedência, quando cabíveis; os erros ainda podem ser capturados quando uma redução for pedida e nenhum *handle* puder ser achado.

Nem toda tabela de relações de precedência possui funções de precedência para codificá-la, mas em casos práticos a função usualmente existe.

Exemplo 4.29. A tabela de precedência da Fig. 4.25 possui os seguintes pares de funções de precedência,

	+	−	*	/	↑	(	)	id	$
f	2	2	4	4	4	0	6	6	0
g	1	1	3	3	5	5	0	5	0

Por exemplo, * $<\cdot$ **id** e $f(*) < g$(**id**). Note-se que f(**id**) $> g$(**id**) sugere que **id** $\cdot>$ **id**; mas, de fato, nenhuma relação de precedência vigora entre **id** e **id.** Outras entradas de erro na Fig. 4.25 são similarmente substituídas por uma ou outra relação de precedência. □

Um método simples para se encontrar as funções de precedência para uma tabela, se tais funções existirem, é o seguinte.

Algoritmo 4.6. Construção de funções de precedência.

Entrada. Uma matriz de precedência de operadores.

Saída. Funções de precedência representando a matriz de entrada ou uma indicação de que as funções não existem.

Método.

1. Criar os símbolos f_a e g_a para cada a que seja um terminal ou $.
2. Particionar os símbolos criados em tantos grupos quantos sejam possíveis, de tal forma que se $a \doteq b$, então f_a e g_b estão no mesmo grupo. Note-se que podemos ter que colocar símbolos no mesmo grupo mesmo que não estejam relacionados por $\doteq$. Por exemplo, se $a \doteq b$ e $c \doteq b$, então f_a e f_c precisam estar no mesmo grupo, uma vez que ambos estão no mesmo grupo que g_b. Se, adicionalmente, $c \doteq d$, então, f_a e g_d estão no mesmo grupo, ainda que $a \doteq d$ possa não vigorar.
3. Criamos um grafo dirigido cujos nós são os grupos encontrados em (2). Para quaisquer a e b, se $a <\cdot\, b$, colocar um lado a partir do grupo de g_b para o grupo de f_a. Se $a \cdot> b$, colocar um lado a partir do grupo de f_a para aquele de g_b. Note-se que um lado ou percurso de f_a para g_b significam que $f(a)$ tem que exceder $g(b)$; um percurso de g_b até f_a significa que $g(b)$ precisa exceder $f(a)$.
4. Se o grafo construído em (3) possuir um ciclo, então não há funções de precedência. Se não existirem ciclos, fazer $f(a)$ igual ao comprimento do mais longo percurso começando no grupo de f_a; fazer $g(a)$ igual ao comprimento do mais longo percurso a partir do grupo de g_a. □

Exemplo 4.30. Consideremos a matriz da Fig. 4.23. Não existem relacionamentos $\doteq$ e cada símbolo está num grupo constituído por si só. A Fig. 4.26 mostra o grafo construído usando-se o algoritmo 4.6.

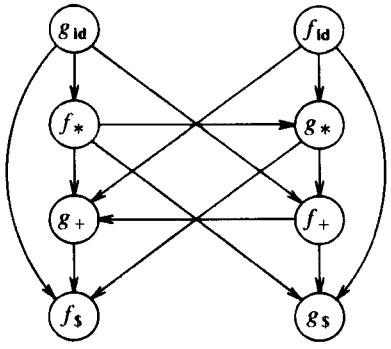


Fig. 4.26. Grafo representando as funções de precedência.

Não existem ciclos; portanto, as funções de precedência existem. Como $f_\$$ e $g_\$$ não têm lados para fora, $f(\$) = g(\$) = 0$. O mais longo percurso a partir de $g+$ possui comprimento 1, e, então, $g(+) = 1$. Existe um percurso a partir de g_{id} para f_*, para g_*, para f_+, para g_+ para $f_\$$ e, por conseguinte, g(**id**) = 5. As funções de precedência resultantes são:

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

□

Recuperação de Erros na Análise Sintática de Precedência de Operadores

Existem dois pontos no processo de análise gramatical nos quais um analisador sintático de precedência de operadores pode descobrir erros sintáticos:

1. Se nenhuma relação vigorar entre o terminal ao topo da pilha e a entrada corrente.[1]
2. Se um *handle* for encontrado, mas não existir produção tendo tal *handle* como lado direito.

Relembremos que o algoritmo de análise sintática de precedência de operadores (algoritmo 4.5) parece reduzir *handles* constituídos somente por terminais. Entretanto, conquanto os não-terminais sejam tratados anonimamente, ainda possuem os seus lugares guardados na pilha sintática. Então, ao falarmos em (2) sobre um *handle* reconhecer o lado direito de uma produção, significa que os terminais são os mesmos e também as posições ocupadas pelos não-terminais.

Deveríamos observar que, além de (1) e (2) acima, não existem pontos nos quais os erros poderiam ser detectados. Ao varrermos pilha abaixo para encontrar a extremidade esquerda do *handle* nos passos (10-12) da Fig. 4.24, do algoritmo de análise sintática de precedência de operadores, estamos certos de encontrar uma relação $<\cdot$, uma vez que \$ marca o fundo da pilha e está relacionado por $<\cdot$ a qualquer outro símbolo que possa aparecer imediatamente acima do mesmo na pilha. Note também que jamais permitiremos símbolos adjacentes na pilha, a menos que estejam relacionados por $<\cdot$ ou $\doteq$. Por conseguinte, os passos (10-12) têm que ter sucesso ao realizar uma redução.

O fato de encontrarmos exatamente uma seqüência de símbolos $a <\cdot b_1 \doteq b_2 \doteq \ldots \doteq b_k$ na pilha não significa que $b_1 b_2 \ldots b_k$ seja uma cadeia de símbolos terminais à direita de alguma produção. Não verificamos esta condição na Fig. 4.24, mas claramente podemos fazê-lo, e, de fato, temos que fazê-lo, se desejamos associar regras semânticas às reduções. Conseqüentemente, temos uma oportunidade de detectar erros na Fig. 4.24, se modificada nos passos (10-12) de forma a determinar qual produção é o *handle* numa redução.

Tratando Erros Durante as Reduções

Podemos dividir a rotina de detecção e recuperação de erros em diversas partes. Uma, trata os erros do tipo (2). Por exemplo, essa rotina poderia remover da pilha os símbolos, exatamente como nos passos (10-12) da Fig. 4.24. Entretanto, como não há produção a reduzir, nenhuma ação semântica é executada; ao invés, é impressa uma mensagem de diagnóstico. Para determinar o que o diagnóstico deveria informar, a rotina que tratasse o caso (2) precisaria decidir com que produção se "assemelharia" o lado direito da pilha que está sendo removido. Por exemplo, suponhamos que *abc* esteja sendo desempilhado e não exista lado direito de produção que consista em *a*, *b* e *c*, juntamente com zero ou mais não-terminais. Precisamos, então, considerar se a remoção de um dentre *a*, *b* e *c* produz um lado direito legal (não-terminais omitidos). Por exemplo, se existisse um lado direito *aEcE*, poderíamos emitir o diagnóstico

b `inválido à linha` (linha contendo *b*)

Poderíamos também considerar a mudança ou a inserção de um terminal. Dessa forma, se *abEdc* estivesse num lado direito, poderíamos emitir o diagnóstico

d `ausente à linha` (linha contendo *c*)

Podemos também concluir que exista um lado direito com a seqüência própria de terminais, mas com o padrão errado de não-terminais. Por exemplo, se *abc* for desempilhado sem não-terminais intervenientes ou envolventes e *abc* não for um lado direito mas sim *aEbc*, poderíamos emitir o diagnóstico

E `ausente à linha` (linha contendo *b*)

Aqui *E* está no lugar de uma categoria sintática apropriada, representada pelo não-terminal *E*. Por exemplo, se *a*, *b* ou *c* fossem operadores, poderíamos dizer "expressão"; se *a* fosse uma palavra-chave como **if**, diríamos "condicional".

[1]Nos compiladores que usam as funções de precedência para representar as tabelas de precedência, esta fonte de detecção de erros pode estar indisponível.

Em geral, a dificuldade em se determinar os diagnósticos apropriados, quando nenhum lado direito legal é encontrado, depende da existência de um número finito ou infinito de possíveis cadeias que poderiam ser removidas às linhas (10-12) da Fig. 4.24. Qualquer cadeia $b_1 b_2 \ldots b_k$ precisa possuir relações $\doteq$ vigorando entre os símbolos adjacentes, e, então, $b_1 \doteq b_2 \doteq \ldots \doteq b_k$. Se uma tabela de precedência de operadores nos diz que existe somente um número finito de seqüências de terminais relacionados por $\doteq$, podemos tratar essas cadeias numa base caso a caso. Para cada tal cadeia *x*, podemos determinar antecipadamente um lado direito legal de distância mínima e emitir um diagnóstico indicando ter sido *x* encontrado quando, de fato, era *y* o esperado.

É fácil determinar todas as cadeias que poderiam ser desempilhadas nos passos (10-12) da Fig. 4.24. São evidentes nos grafos dirigidos, cujos nós representam os terminais, com um lado de *a* para *b* se e somente se $a \doteq b$. Conseqüentemente, as possíveis cadeias são os rótulos dos nós ao longo de percursos naqueles grafos. Percursos consistindo em um único nó são possíveis. Entretanto, para que um percurso $b_1 b_2 \ldots b_k$ seja "desempilhável" dada alguma entrada, deve haver um símbolo *a* (possivelmente \$) tal que $a <\cdot b_1$. Chamemos tal símbolo b_1 de *inicial*. Também deve haver um símbolo *c* (possivelmente \$) tal que $b_k \cdot> c$. Chamemos tal símbolo b_k de *final*. Somente então uma redução poderia ser invocada e $b_1 b_2 \ldots b_k$ ser a seqüência de símbolos desempilhada. Se o grafo possuir um percurso, de um nó inicial para um final, contendo um ciclo, existe uma infinidade de cadeias que poderiam ser desempilhadas; caso contrário, existe somente um número finito.

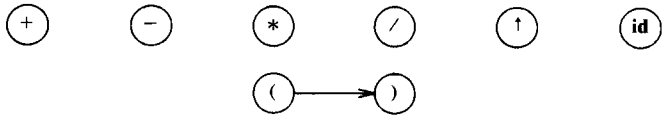


Fig. 4.27. Grafo para a matriz de precedência da Fig. 4.25.

Exemplo 4.31. Vamos reconsiderar a gramática (4.17):

$$E \rightarrow E + E \mid E - E \mid E * E \mid E/E \mid E \uparrow E \mid (E) \mid -E \mid \textbf{id}$$

A matriz de precedência para a mesma foi mostrada na Fig. 4.25 e seu grafo é fornecido na Fig. 4.27. Existe somente um lado, porque o único par relacionado por $\doteq$ é constituído pelo parênteses à esquerda e pelo à direita. Tudo menos o parêntese à direita é inicial e tudo menos o parêntese à esquerda é final. Por conseguinte, os únicos percursos que vão de um inicial até um final são +, −, *, /, **id** e ↑ de comprimento um e o percurso de "("até")", de comprimento dois. Não existe nada mais do que um número finito (de percursos) e cada um corresponde aos terminais de algum lado direito de uma produção na gramática. Dessa forma, o verificador de erros para as reduções só precisa checar se o conjunto apropriado de não-terminais marcadores aparece junto com as cadeias de terminais sendo reduzidas. Especificamente, o verificador realiza o seguinte:

1. Se +, −, *, / ou ↑ for reduzido, verifica se os não-terminais aparecem em ambos os lados. Se não, emite o diagnóstico

   `operando ausente`

2. Se **id** for reduzido, checa se não existe não-terminal à direita ou à esquerda. Se existir, pode avisar

   `operador ausente`

3. Se ( ) for reduzido, verifica se existe um não-terminal entre os parênteses. Se não, pode informar

   `nenhuma expressão entre os parênteses`

Igualmente, precisa assegurar que nenhum não-terminal figure ao lado de qualquer um dos dois parênteses. Se existir algum, emite o mesmo diagnóstico que em (2). □

Se existir uma infinidade de cadeias que possam ser desempilhadas, as mensagens de erro não podem ser tabuladas numa base caso a caso. Poderíamos usar uma rotina geral para determinar se algum lado direito de produção está próximo (digamos, distância 1 ou 2, onde a distância é medida em termos dos *tokens* inseridos, removidos ou modificados, e não em caracteres) da cadeia desempilhada e, se assim o for, emitir o diagnóstico específico na suposição de que aquela produção era a pretendida. Se nenhuma produção estiver próxima à cadeia desempilhada, podemos emitir um diagnóstico geral especificando que "alguma coisa está errada na linha corrente".

Tratando Erros de Empilhar/Reduzir

Precisamos agora discutir a outra forma pela qual o analisador sintático de precedência de operadores pode descobrir os erros. Ao consultar a matriz de precedência para decidir se empilha ou reduz (linhas (6) e (9) da Fig. 4.24), podemos encontrar que nenhuma relação vigore entre o topo da pilha e o primeiro símbolo de entrada. Por exemplo, suponhamos que a e b sejam os dois símbolos de topo da pilha (b está ao topo), c e d são os próximos dois símbolos de entrada e não existe relação de precedência entre b e c. Para recuperarmos, precisamos modificar a pilha, a entrada ou ambas. Podemos modificar símbolos, inseri-los na entrada ou na pilha ou removê-los da entrada ou da pilha. Se inserirmos ou modificarmos, precisamos ter cuidado para não entrarmos num laço infinito, onde, por exemplo, inseriríamos perpetuamente símbolos ao início da entrada sem estarmos capacitados a reduzir ou empilhar quaisquer dos símbolos inseridos.

Um enfoque que irá nos assegurar a exclusão dos laços infinitos é garantir que, após a recuperação, o símbolo corrente de entrada possa ser empilhado (se a entrada corrente for \$, garantir que nenhum símbolo seja colocado na entrada e a pilha seja eventualmente encurtada). Por exemplo, dado ab na pilha e cd na entrada, se $a \leq\cdot\, c$[2], poderíamos desempilhar b. Outra escolha é remover c da entrada se $b \leq\cdot\, d$. Uma terceira escolha é a de encontrar um símbolo e tal que $b \leq\cdot\, e \leq\cdot\, c$ e inserir e à frente de c na entrada. Mais geralmente, poderíamos inserir uma cadeia de símbolos tal que

$$b \leq\cdot\, e_1 \leq\cdot\, e_2 \leq \ldots \leq\cdot\, e_n \leq\cdot\, c$$

se um único símbolo para inserção não puder ser encontrado. A ação exata escolhida deveria refletir a intuição do projetista do compilador relacionada ao que mais provavelmente o erro seria em cada caso.

Para cada entrada em branco na matriz de precedência precisamos especificar uma rotina de recuperação de erros; a mesma rotina poderia ser usada em vários locais. Quando o analisador sintático consulta a entrada para a e b no passo (6) da Fig. 4.24 e nenhuma relação de precedência vigora entre a e b, o mesmo encontra um apontador para a rotina de recuperação daquele erro.

Exemplo 4.32. Consideremos a matriz de precedência da Fig. 4.25 de novo. Na Fig. 4.28, mostramos as linhas e colunas dessa matriz que tenham uma ou mais entradas em branco e as preenchemos com os nomes das rotinas de tratamento de erro.

	id	**(**	**)**	**\$**
id	e3	e3	$\cdot>$	$\cdot>$
(	$<\cdot$	$<\cdot$	$\doteq$	e4
)	e3	e3	$\cdot>$	$\cdot>$
\$	$<\cdot$	$<\cdot$	e2	e1

Fig. 4.28. Matriz de precedência de operadores com entradas de erro.

[2]Usamos $\leq\cdot$ para significar $<\cdot$ ou $\doteq$

O núcleo dessas rotinas de tratamento de erro é como se segue:

```
e1: /* chamada quando toda a expressão estiver faltando */
    inserir id na entrada
    emitir diagnóstico: "operando ausente"
e2: /* chamada quando uma expressão começa por um parêntese à */
    /* direita */
    emitir diagnóstico: "parêntese à direita não-
    balanceado"
e3: /* chamada quando id ou) é seguido por id ou ( */
    inserir + na entrada
    emitir diagnóstico: "operando ausente"
e 4: /* chamada quando a expressão termina por um parênteses à */
    /* esquerda */
    desempilhar (
    emitir diagnóstico: "parêntese à direita ausente"
```

Vamos considerar como esse mecanismo de tratamento de erros manipularia a entrada incorreta **id** +). As primeiras ações tomadas pelo analisador são empilhar **id**, reduzi-lo para E (usamos de novo E para o não-terminal anônimo na pilha) e, então, empilhar o +. Temos agora a configuração

PILHA	ENTRADA
\$$E$+	)\$

Uma vez que + $\cdot>$), uma redução é pedida e o *handle* é +. O verificador de erros para reduções é requerido para inspecionar E's à esquerda e à direita. Ao encontrar que algum desses E's está faltando, emite o diagnóstico

```
operando ausente
```

e de qualquer forma realiza a redução.

Nossa configuração é agora

\$$E$	)\$

Não existe relação de precedência entre \$ e) e a entrada da Fig. 4.28 para este par de símbolos é e2. A rotina e2 provoca a impressão do diagnóstico

```
parêntese à direita não-balanceado
```

e remove o parêntese da entrada. Somos deixados com a configuração final do analisador sintático.

\$$E$	\$	□

4.7 ANALISADORES SINTÁTICOS LR

Esta seção apresenta uma técnica eficiente de análise sintática *bottom-up*, que pode ser usada para decompor uma ampla classe de gramáticas livres de contexto. A técnica é chamada análise sintática LR(k); o "L" significa varredura da entrada da esquerda para a direita (*left-to-right*), o "R", construir uma derivação mais à direita ao contrário (*rightmost derivation*) e o k, o número de símbolos de entrada de *lookahead* que são usados ao se tomar decisões na análise sintática. Quando (k) for omitido, k é assumido ser 1. A técnica de análise sintática LR é atrativa por uma série de razões:

- Analisadores sintáticos LR podem ser elaborados para reconhecer virtualmente todas as construções de linguagens de programação para as quais as gramáticas livres de contexto podem ser escritas.
- O método de decomposição LR é o mais geral dentre os métodos sem retrocesso de empilhar e reduzir conhecidos e ainda pode ser implementado tão eficientemente quanto os demais métodos de empilhar e reduzir.
- A classe de gramáticas que podem ser decompostas usando-se os métodos LR é um superconjunto próprio da classe de gramáticas que podem ser decompostas usando-se analisadores sintáticos preditivos.
- Um analisador sintático LR pode detectar um erro sintático tão cedo quanto possível numa varredura da entrada da esquerda para a direita.

A principal desvantagem deste método está em ser muito trabalhoso construir um analisador sintático LR manualmente para uma gramática típica de linguagem de programação. Necessita-se em geral de uma ferramenta especializada — um gerador de analisadores LR. Felizmente, muitos de tais geradores estão disponíveis e iremos discutir o projeto e uso de um, o Yacc, na Seção 4.9. Com um tal gerador, podemos escrever uma gramática livre de contexto e usá-lo para produzir automaticamente um analisador sintático para a mesma. Se a gramática contiver ambigüidades ou outras construções que sejam difíceis de decompor, numa varredura da entrada da esquerda para a direita, o gerador de analisadores pode localizá-las e informar ao projetista do compilador de suas ocorrências.

Após discutir a operação de um analisador LR, apresentamos três técnicas para construir uma tabela sintática para uma gramática. O primeiro método, chamado de LR simples (SLR — de *simple* LR), é o mais fácil de implementar, mas o menos poderoso dos três. Pode falhar em produzir uma tabela sintática para algumas gramáticas em que os outros dois possam ter sucesso. O segundo método, chamado de LR canônico, é o mais poderoso e o mais caro. O terceiro, chamado LR *lookahead* (LALR — de *lookahead* LR) é de poder e custo intermediários. O método LALR irá funcionar na maioria das gramáticas das linguagens de programação e, com algum esforço, poderá ser implementado eficientemente. Algumas técnicas para comprimir o tamanho das tabelas sintáticas LR são consideradas mais adiante nesta seção.

O Algoritmo de Análise Sintática LR

A forma esquemática de um analisador sintático LR é mostrada na Fig. 4.29. Consiste em uma entrada, uma saída, uma pilha, um programa diretor e uma tabela sintática que possui duas partes (*ação* e *desvio*). O programa diretor é o mesmo para todos os três tipos de analisadores LR; somente a tabela sintática muda de um analisador para outro. O programa de análise sintática lê os caracteres provenientes de um *buffer* de entrada, um de cada vez. Usa uma pilha para armazenar as cadeias sob a forma $s_0X_1s_1X_2s_2 \ldots X_ms_m$, onde s_m está ao topo. Cada X_i é um símbolo gramatical e cada s_i, um símbolo chamado de *estado*. Cada estado sumariza a informação contida na pilha abaixo do mesmo e a combinação do símbolo do estado no topo da pilha e o símbolo corrente de entrada é usada para indexar a tabela sintática e determinar a decisão de empilhar ou reduzir durante a análise. Numa implementação, os símbolos gramaticais não precisam figurar na pilha; entretanto, iremos sempre incluí-los em nossas discussões, a fim de auxiliar a explicação do comportamento de um analisador sintático LR.

A tabela sintática consiste em duas partes, uma função de ações sintáticas, *ação*, e uma função de desvio, *desvio*. O programa diretor do analisador LR se comporta como se segue. Determina s_m, o estado correntemente no topo da pilha, e a_i, o símbolo corrente de entrada. Consulta, então, *ação* $[s_m, a_i]$, a entrada da tabela de ações sintáticas para o estado s_m e entrada a_i, que pode ter um dos quatro seguintes valores:

1. empilhar s, onde s é um estado,
2. reduzir através da produção gramatical $A \rightarrow \beta$,
3. aceitar, e
4. erro.

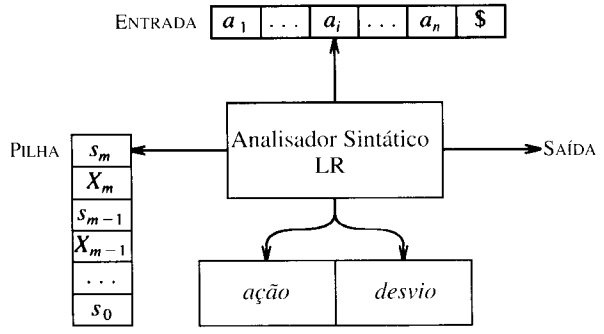


Fig. 4.29. Modelo de um analisador sintático LR.

A função *desvio* toma um estado e um símbolo gramatical como argumentos e produz um estado como saída. Veremos que a função *desvio* de uma tabela sintática, construída a partir de uma gramática G, usando os métodos SLR, LR canônico ou LALR, é a função de transição de um autômato finito determinístico que reconhece os prefixos viáveis de G. Relembremos que os prefixos viáveis de G são aqueles das formas sentenciais à direita que podem aparecer na pilha de um analisador sintático de empilhar e reduzir, porque os mesmos não se estendem para depois do *handle* mais à direita. O estado inicial deste AFD é o estado inicialmente colocado no topo da pilha do analisador sintático LR.

Uma *configuração* de um analisador sintático LR é um par cujo primeiro componente é o conteúdo da pilha e cujo segundo componente é a entrada ainda não consumida:

$$(s_0X_1s_1X_2s_2 \ldots X_ms_m, a_ia_{i+1} \ldots a_n\$)$$

Esta configuração representa a forma sentencial à direita

$$X_1X_2 \ldots X_m a_i a_{i+1} \ldots a_n$$

essencialmente da mesma maneira que um analisador de empilhar e reduzir faria; somente a presença dos estados na pilha é inovadora.

O próximo movimento do analisador é determinado pela leitura de a_i, o símbolo corrente da entrada e de s_m, o estado no topo da pilha, e pela consulta à entrada da tabela de ações, *ação*$[s_m, a_i]$. As configurações resultantes após cada um dos quatro tipos de movimentos são como se segue:

1. Se *ação*$[s_m, a_i]$ = empilhar s, o analisador executa um movimento de empilhar, entrando na configuração

   $$(s_0X_1s_1X_2s_2 \ldots X_ms_m a_i s, a_{i+1} \ldots a_n\$)$$

   Aqui, o analisador sintático empilhou tanto o símbolo corrente de entrada quanto o próximo estado s, que é dado por *ação*$[s_m, a_i]$; a_{i+1} se torna o símbolo corrente da entrada.
2. Se *ação*$[s_m, a_i]$ = reduzir $A \rightarrow \beta$, o analisador sintático executa um movimento de redução, entrando na configuração

   $$(s_0X_1s_1X_2s_2 \ldots X_{m-r}s_{m-r}A\, s, a_i a_{i+1} \ldots a_n\$)$$

   onde s = *desvio*$[s_{m-r}, A]$ e r é o comprimento de β, o lado direito da produção. Aqui o analisador sintático remove primeiro $2r$ símbolos para fora da pilha (r símbolos de estados e r símbolos gramaticais), expondo o estado s_{m-r}. Em seguida, empilha tanto A, o lado esquerdo da produção, quanto s, a entrada para *desvio*$[s_{m-r}, A]$. Para os analisadores sintáticos LR que iremos construir, $X_{m-r+1} \ldots X_m$, a seqüência de símbolos gramaticais removidos da pilha irá sempre reconhecer β, o lado direito da produção redutora.
   A saída de um analisador sintático LR é gerada após um movimento de redução, através da execução de uma ação semântica associada à produção redutora. Para o momento, assumiremos que a saída consista somente na impressão da produção redutora.
3. Se *ação*$[s_m, a_i]$ = aceitar, a análise sintática estará completa.
4. Se *ação*$[s_m, a_i]$ = erro, o analisador sintático descobriu um erro e chama um procedimento de recuperação de erros.

O algoritmo de análise sintática LR é sumarizado abaixo. Todos os analisadores sintáticos LR se comportam desta maneira; a única diferença entre dois analisadores LR são as informações nos campos de ação e desvio da tabela sintática.

Algoritmo 4.7. Algoritmo de análise sintática LR.

Entrada. Uma cadeia de entrada w e uma tabela sintática com as funções *ação* e *desvio* para uma gramática G.

Saída. Se w estiver em $L(G)$, uma decomposição *bottom-up* para w; em caso contrário, uma indicação de erro.

Método. Inicialmente, o analisador sintático possui s_0 na pilha, onde s_0 é o estado inicial, e $w\$$ no *buffer* de entrada. O analisador sintático executa, então, o programa da Fig. 4.30 até que uma ação de aceitação ou de erro seja atingida. □

```
fazer ip apontar para o primeiro símbolo w$;
repetir para sempre início
        seja s o estado ao topo da pilha e
        a o símbolo apontado por ip;
        se ação [s, a] = empilhar s' então início
                empilhar a e em seguida s' no topo da pilha;
                avançar ip para o próximo símbolo de entrada
        fim
        senão se ação [s, a] = reduzir A → β então início
                desempilhar 2*|β| símbolos para fora
                da pilha; seja s' o estado agora ao topo da pilha;
                empilhar A e em seguida desvio[s', A];
                escrever a produção A → β
        fim
        senão se ação [s, a] = aceitar então
                retornar
        senão erro ( )
fim
```

Fig. 4.30. Programa de análise sintática LR.

Exemplo 4.33. A Fig. 4.31 mostra as funções de ação e desvio de uma tabela sintática LR para a seguinte gramática para expressões aritméticas como os operadores binários + e *:

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

O código para cada ação é:

1. *si* significa empilhar o símbolo de entrada mais o estado *i*,
2. r*j* significa reduzir através da produção de número *j*,
3. acc significa aceitar,
4. uma entrada em branco significa um erro.

Note que o valor de *desvio* [*s*, *a*], para um terminal *a*, é encontrado no campo ação, conectado à ação de empilhar à entrada *a* e estado *s*. O campo desvio fornece *desvio* [*s*, *A*] para os não-terminais *A*. Igualmente, tenhamos em mente que ainda não explicamos como as entradas para a Fig. 4.31 foram selecionadas; em breve iremos lidar com este tema.

ESTADO	*ação*						*desvio*		
	id	+	*	(	)	$	*E*	*T*	*F*
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Fig. 4.31. Tabela sintática para a gramática de expressões.

	PILHA	ENTRADA	AÇÃO
(1)	0	**id** * **id** + **id** $	empilhar
(2)	0 **id** 5	* **id** + **id** $	reduzir por $F \rightarrow$ **id**
(3)	0 *F* 3	* **id** + **id** $	reduzir por $T \rightarrow F$
(4)	0 *T* 2	* **id** + **id** $	empilhar
(5)	0 *T* 2 * 7	**id** + **id** $	empilhar
(6)	0 *T* 2 * 7 **id** 5	+ **id** $	reduzir por $F \rightarrow$ **id**
(7)	0 *T* 2 * 7 *F* 10	+ **id** $	reduzir por $T \rightarrow T*F$
(8)	0 *T* 2	+ **id** $	reduzir por $E \rightarrow T$
(9)	0 *E* 1	+ **id** $	empilhar
(10)	0 *E* 1 + 6	**id** $	empilhar
(11)	0 *E* 1 + 6 **id** 5	$	reduzir por $F \rightarrow$ **id**
(12)	0 *E* 1 + 6 *F* 3	$	reduzir por $T \rightarrow F$
(13)	0 *E* 1 +6 *T* 9	$	$E \rightarrow E + T$
(14)	0 *E* 1	$	aceitar

Fig. 4.32. Movimentos de um analisador sintático LR para **id** * **id** + **id.**

Para a entrada **id** * **id** + **id,** a seqüência de conteúdos da pilha da entrada é mostrada na Fig. 4.32. Por exemplo, à linha (1), o analisa dor sintático LR está no estado 0, tendo **id** como o primeiro símbolo d entrada. A ação na linha 0 e coluna **id** do campo ação da Fig. 4.31 é s5 significando empilhar o símbolo de entrada **(id)** e cobrir o topo da p lha com o estado 5. Isso foi o que aconteceu à linha (2): o primeiro *toke* **id** e o símbolo de estado 5 foram ambos empilhados e **id** foi removid da entrada.

Em seguida, * se torna o símbolo corrente da entrada e a açã do estado 5 para a entrada * é reduzir através de $F \rightarrow$ **id.** Os dois sím bolos são removidos da pilha (um símbolo de estado e um símbol gramatical). O estado 0 é, então, exposto. Uma vez que desvio do esta do 0 em *F* é 3, *F* e 3 são empilhados. Temos agora a configuração d linha (3). Cada um dos movimentos restantes é determinado de form similar. □

Gramática LR

Como construímos uma tabela sintática LR para uma dada gramática Uma gramática para a qual podemos construir uma tabela sintática denominada uma *gramática LR.* Existem gramáticas livres de contex to que não são LR, mas que podem ser geralmente evitadas para a construções típicas das linguagens de programação. Intuitivamente, par uma gramática ser LR, é suficiente existir um analisador sintático d empilhar e reduzir que, processando a entrada da esquerda para a direi ta, seja capaz de reconhecer os *handles* desta gramática, quando o mesmos surgirem no topo da pilha.

Um analisador LR não precisa varrer toda a pilha para saber quan do um *handle* surge no topo. De fato, o símbolo de estado ao topo da pi lha contém toda a informação necessária. É um fato digno de nota que, s for possível reconhecer um *handle* conhecendo-se somente os símbolo gramaticais na pilha, existe então um autômato finito que pode, atravé da leitura desses símbolos gramaticais na pilha, do topo para o fundo determinar qual *handle,* se houver algum, está ao topo da mesma. A fun ção desvio de uma tabela sintática LR representa essencialmente tal au tômato finito. O autômato não precisa, entretanto, ler a pilha a cada mo vimento. O símbolo de estado armazenado ao topo da pilha é o estado que o autômato finito reconhecedor de *handles* estaria se tivesse lido o símbolos gramaticais na pilha, do fundo para o topo. Por conseguinte o analisador sintático LR pode determinar, a partir do estado ao topo da pilha, tudo o que necessita saber sobre o que está abaixo dele.

Uma outra fonte de informações que o analisador sintático LR pode usar para auxiliá-lo a tomar as decisões de reduzir e empilhar são os próximos *k* símbolos de entrada. Os casos $k = 0$ e $k = 1$ são de inte resse prático e iremos considerar aqui somente analisadores sintáticos com $k \leq 1$. Por exemplo, a tabela de ações na Fig. 4.31 usa um símbo lo de *lookahead.* Uma gramática que possa ser decomposta por um analisador sintático LR examinado até *k* símbolos de entrada a cada movimento é chamada de uma *gramática LR(k).*

Existe uma significativa diferença entre uma gramática LL e uma LR. No caso de uma gramática ser LR*(k)*, precisamos ser capazes de reconhecer a ocorrência do lado direito de uma produção tendo visto tudo do que foi derivado a partir daquele lado direito mais o esquadrinhamento antecipado de *k* símbolos de entrada (isto é, *k* símbolos *lookahead*). Essa exigência é muito menos restritiva do que aquela para uma gramática LL*(k)*, onde precisamos ser capazes de reconhecer o uso de uma produção examinando somente os primeiros *k* símbolos que seu lado direito pode derivar. Dessa forma, as gramáticas LR podem descrever mais linguagens do que as gramáticas LL.

Construindo Tabelas Sintáticas SLR

Mostraremos agora como construir, a partir de uma gramática, uma tabela sintática LR. Iremos fornecer três métodos, que variam em poder e facilidade de implementação. O primeiro, chamado de "LR simples", ou SLR abreviadamente, é o mais fraco dos três em termos do número de gramáticas para as quais têm sucesso, mas é o mais fácil de implementar. Iremos nos referir à tabela sintática construída por esse método como a tabela SLR e a um analisador sintático que use tal tabela, como um analisador sintático SLR. Uma gramática para a qual um analisador SLR possa ser construído é dita uma gramática SLR. Os outros dois métodos expandem o método SLR com informações de *lookahead*, de forma que o método SLR é um bom ponto de partida para o estudo da análise sintática LR.

Um *item LR(0) (item,* simplificadamente) para uma gramática *G* é uma produção de *G* com um ponto em alguma de suas posições no lado direito. Por conseguinte, a produção $A \rightarrow XYZ$ produz os quatro itens seguintes

$$
\begin{aligned}
A &\rightarrow \cdot XYZ \\
A &\rightarrow X \cdot YZ \\
A &\rightarrow XY \cdot Z \\
A &\rightarrow XYZ \cdot
\end{aligned}
$$

A produção $A \rightarrow \epsilon$ gera somente um item, $A \rightarrow \cdot$. Um item pode ser representado por um par de inteiros, o primeiro fornecendo o número da produção e o segundo a posição do ponto. Intuitivamente, um item indica quanto de uma produção já examinamos a uma dada altura do processo de análise sintática. Por exemplo, o primeiro item acima indica que desejamos em seguida examinar uma cadeia na entrada, derivável a partir de *XYZ*. O segundo item indica que acabamos de examinar na entrada uma cadeia derivável a partir de *X* e que esperamos em seguida ver uma cadeia derivável a partir de *YZ*.

A idéia central do método SLR é construir primeiro a partir da gramática um autômato finito determinístico que reconheça prefixos viáveis. Agrupamos esses itens em conjuntos, os quais dão origem aos estados do analisador sintático SLR. Os itens podem ser vistos como os estados do AFD que reconhece os prefixos viáveis; o agrupamento dos mesmos é, de fato, o processo de construção de subconjuntos discutido na Seção 3.6.

Uma coleção de conjuntos de itens LR(0), que chamamos uma coleção LR(0) *canônica*, providencia a base para a construção de analisadores sintáticos SLR. Para construir a coleção canônica LR(0) para uma gramática, definimos uma gramática aumentada e duas funções, *fechamento* e *desvio*.

Se *G* for uma gramática com símbolo de partida *S*, então *G'*, a *gramática aumentada* para *G* é *G* com um novo símbolo de partida, *S'*, mais a produção $S' \rightarrow S$. O propósito desta nova produção de partida é o de indicar ao analisador sintático quando o mesmo deve parar de analisar e anunciar a aceitação da entrada. Ou seja, a aceitação ocorre quando e somente quando o analisador sintático estiver para reduzir através de $S' \rightarrow S$.

A Operação de Fechamento

Se *I* for um conjunto de itens para uma gramática *G*, então o *fechamento (I)* é o conjunto de itens construídos a partir de *I* por essas duas regras:

1. Inicialmente, cada item em *I* é adicionado ao *fechamento (I)*.
2. Se $A \rightarrow \alpha \cdot B\beta$ estiver em *fechamento (I)* e $B \rightarrow \gamma$ for uma produção, adicionar o item $B \rightarrow \cdot\gamma$ a *I*, se já não estiver lá. Aplicamos esta regra até que não possam ser adicionados novos itens ao *fechamento (I)*.

Intuitivamente, $A \rightarrow \alpha \cdot B\beta$ em *fechamento (I)* indica que, em algum ponto do processo de análise gramatical, esperamos poder ver em seguida na entrada uma subcadeia derivável a partir de $B\beta$. Se $B \rightarrow \gamma$ for uma produção, também esperamos ver uma subcadeia derivável de γ àquele ponto. Por esta razão também incluímos $B \rightarrow \cdot\gamma$ no *fechamento (I)*.

Exemplo 4.34. Consideremos a gramática de expressões aumentada:

$$
\begin{aligned}
E' &\rightarrow E \\
E &\rightarrow E + T \mid T \\
T &\rightarrow T * F \mid F \\
F &\rightarrow (E) \mid \mathbf{id}
\end{aligned}
\tag{4.19}
$$

Se *I* for o conjunto de um item $\{[E' \rightarrow \cdot E]\}$, então o *fechamento (I)* contém os itens

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

Aqui, $E' \rightarrow \cdot E$ é colocado em *fechamento (I)* pela regra (1). Como existe um *E* imediatamente à direita do ponto, adicionamos, pela regra (2), as produções-*E* com pontos nas extremidades à esquerda, isto é, $E \rightarrow \cdot E + T$ e $E \rightarrow \cdot T$. Existe agora um *T* imediatamente à direita do ponto e conseqüentemente adicionamos $T \rightarrow \cdot T * F$ e $T \rightarrow \cdot F$. Em seguida, o *F* à direita do ponto força a inclusão de $F \rightarrow \cdot(E)$ e de $F \rightarrow \cdot\mathbf{id}$. Nenhum outro item é colocado no *fechamento (I)* pela regra (2). □

A função de *fechamento* pode ser computada como na Fig. 4.33. Uma forma conveniente de implementá-la é manter o *array* booleano *adicionado*, indexado pelos não-terminais de *G*, de tal forma que *adicionado* [*B*] seja feito **verdadeiro** se e somente se tivermos adicionado os itens $B \rightarrow \cdot\gamma$, para cada produção-$B \rightarrow \gamma$.

```
função fechamento ( I );
início
    J := I;
    repetir
        para cada item A → α · Bβ em J e cada produção
            B → γ de G tal que B → · γ não esteja em J faça
                    incluir B → · γ a J
    até que não possam ser adicionados mais itens a J;
    retornar J
fim
```

Fig. 4.33. Cômputo de *fechamento*

Note que se uma produção-*B* for incluída no fechamento de *I*, com um ponto na extremidade esquerda, todas as produções-*B* serão similarmente incluídas. De fato, não é necessário, em certas circunstâncias, listar efetivamente os itens $B \rightarrow \cdot\gamma$ adicionados a *I* através de *fechamento*. Uma lista de não-terminais cujas produções tenham sido adicionadas irá bastar. De fato, podemos dividir todos os conjuntos de itens que estejamos interessados em duas classes:

1. *Itens nucleares*, a qual inclui o item inicial e todos os demais cujos pontos não estejam na extremidade esquerda.
2. *Itens não-nucleares*, a qual inclui todos os itens que tenham o ponto na extremidade esquerda.

Sobretudo, cada conjunto de itens em que estamos interessados é formado tomando-se um conjunto de itens nucleares; os itens adiciona-

dos ao fechamento não podem nunca ser itens nucleares, naturalmente. Dessa forma podemos representar os conjuntos de itens que estejamos interessados com muito pouca memória se eliminarmos todos os itens não nucleares, sabendo que podem ser regenerados pelo processo de fechamento.

A Operação Desvio

Uma segunda função útil é *desvio (I, X)*, onde *I* é um conjunto de itens e *X* um símbolo gramatical. *Desvio (I, X)* é definida como o fechamento do conjunto de todos os itens $[A \rightarrow \alpha X \cdot \beta]$ tais que $[A \rightarrow \alpha \cdot X\beta]$ esteja em *I*. Intuitivamente, se *I* for o conjunto de itens válidos para algum prefixo viável γ, então *desvio (I, X)* será o conjunto de itens válidos para o prefixo viável γX.

Exemplo 4.35. Se *I* for o conjunto de dois itens $\{[E' \rightarrow E\cdot], [E \rightarrow E\cdot + T]\}$, então *desvio (I, +)* consiste em

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

Computamos *desvio (I, +)* através do exame dos itens com + imediatamente à direita do ponto. $E' \rightarrow E\cdot$ não é um desses itens, mas $E \rightarrow E \cdot + T$ o é. Movemos o ponto por sobre o + para obtermos $\{E \rightarrow E + \cdot T\}$ e em seguida realizamos o fechamento deste conjunto. □

A Construção dos Conjuntos de Itens

Estamos agora prontos para fornecer o algoritmo para construir *C*, a coleção canônica de conjuntos de itens LR(0) para uma gramática aumentada *G'*; o algoritmo é mostrado na Fig. 4.34.

procedimento *itens* (*G'*);
início
C := { *fechamento* ({ $[S' \rightarrow \cdot S]$ }) };
repetir
  para cada conjunto de itens *I* em *C* e cada símbolo gramatical *X* tal que *desvio (I, X)* não seja vazio e não esteja em *C* **faça** incluir *desvio (I, X)* a *C*
até que não haja mais conjuntos de itens a serem incluídos a *C*
fim

Fig. 4.34. A construção dos conjuntos de itens.

Exemplo 4.36. A coleção canônica de conjuntos de itens LR(0) para a gramática (4.19) do Exemplo 4.34 é mostrada na Fig. 4.35. A função *desvio* para este conjunto de itens é mostrada como o diagrama de transições de um autômato finito determinístico *D* na Fig. 4.36. □

Se cada estado *D* da Fig. 4.36 for um estado final e I_0 for o estado inicial, então *D* reconhece exatamente os prefixos viáveis da gramática (4.19). Isto não é acidente. Para cada gramática *G*, a função *desvio* da coleção canônica de conjuntos de itens define um autômato finito determinístico que reconhece os prefixos viáveis de *G*. De fato, pode-se visualizar um autômato finito não-determinístico *N* cujos estados sejam os próprios itens em si. Há uma transição de $A \rightarrow \alpha \cdot X\beta$ para $A \rightarrow \alpha X \cdot \beta$, rotulada *X*, e existe uma transição de $A \rightarrow \alpha \cdot B\beta$ para $B \rightarrow \cdot\gamma$ rotulada ϵ. Então, o *fechamento (I)* para o conjunto de itens (estados de *N*) *I* é exatamente o *fechamento*-ϵ de um conjunto de estados de um AFN; definido na Seção 3.6. Por conseguinte, *desvio (I, X)* fornece a transição a partir de *I*, no símbolo *X*, do AFD elaborado a partir de *N*, através da construção de subconjuntos. Visto dessa forma, o procedimento *itens (G)* na Fig. 4.34 é exatamente a construção de subconjuntos aplicada ao AFN construído a partir de *G'*, como tínhamos descrito.

I_0: $E' \rightarrow \cdot E$
$E \rightarrow \cdot E + T$
$E \rightarrow \cdot T$
$T \rightarrow \cdot T * F$
$T \rightarrow \cdot F$
$F \rightarrow \cdot(E)$
$F \rightarrow \cdot \mathbf{id}$

I_1: $E' \rightarrow E\cdot$
$E \rightarrow E\cdot + T$

I_2: $E \rightarrow T\cdot$
$T \rightarrow T\cdot * F$

I_3: $T \rightarrow F\cdot$

I_4: $F \rightarrow (\cdot E)$
$E \rightarrow \cdot E + T$
$E \rightarrow \cdot T$
$T \rightarrow \cdot T * F$
$T \rightarrow \cdot F$
$F \rightarrow \cdot(E)$
$F \rightarrow \cdot \mathbf{id}$

I_5: $F \rightarrow \mathbf{id}\cdot$

I_6: $E \rightarrow E + \cdot T$
$T \rightarrow \cdot T * F$
$T \rightarrow \cdot F$
$F \rightarrow \cdot(E)$
$F \rightarrow \cdot \mathbf{id}$

I_7: $T \rightarrow T * \cdot F$
$F \rightarrow \cdot(E)$
$F \rightarrow \cdot \mathbf{id}$

I_8: $F \rightarrow (E\cdot)$
$E \rightarrow E\cdot + T$

I_9: $E \rightarrow E + T\cdot$
$T \rightarrow T\cdot * F$

I_{10}: $T \rightarrow T * F\cdot$

I_{11}: $F \rightarrow (E)\cdot$

Fig. 4.35. Coleção canônica LR(0) para a gramática (4.19).

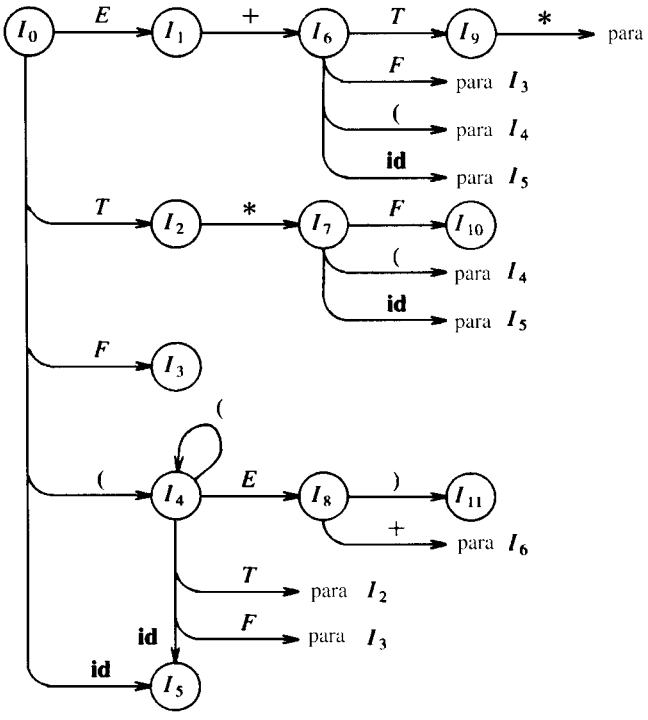


Fig. 4.36. Diagrama de transições para o AFD *D* para os prefixos viáveis.

Itens válidos. Dizemos que o item $A \rightarrow \beta_1 \cdot \beta_2$ é *válido* para prefixo viável $\alpha\beta_1$, se existir uma derivação $S' \overset{*}{\underset{mdd}{\Rightarrow}} \alpha A w \overset{*}{\underset{mdd}{\Rightarrow}} \alpha\beta_1\beta_2$ Em geral, um item será válido para muitos prefixos viáveis. O fato d $A \rightarrow \beta_1 \cdot \beta_2$ ser válido para $\alpha\beta_1$ nos diz muito a respeito de empilh ou reduzir ao encontrarmos $\alpha\beta_1$ na pilha do analisador sintático. E particular, se $\beta_2 \neq \epsilon$, isto sugere que ainda não empilhamos o *hand* e nosso movimento agora é empilhar. Se $\beta_2 = \epsilon$, então tudo indica q $A \rightarrow \beta_1$ é o *handle*, e podemos reduzir através desta produção. Nat

ralmente, dois itens válidos podem nos dizer duas diferentes coisas sobre o mesmo prefixo viável. Alguns desses conflitos podem ser resolvidos examinando-se o próximo símbolo de entrada e os demais conflitos através de métodos desenvolvidos nesta seção, mas não devemos supor que todos os conflitos das ações da análise sintática possam ser resolvidos se o método LR for usado para construir uma tabela sintática para uma gramática arbitrária.

Podemos facilmente computar o conjunto de itens válidos para cada prefixo viável que possa aparecer na pilha do analisador sintático LR. De fato, é um teorema central da teoria da análise sintática LR que o conjunto de itens válidos para um prefixo viável γ seja exatamente o de itens atingidos a partir do estado inicial, juntamente com um percurso rotulado γ no AFD construído a partir da coleção canônica de conjuntos de itens, com as transições dadas pela função *desvio*. Em essência, o conjunto de itens válidos incorpora todas as informações úteis que podem ser vislumbradas na pilha. Conquanto não venhamos a provar este teorema aqui, daremos um exemplo.

Exemplo 4.37. Vamos considerar a gramática (4.19) de novo, cujos conjuntos de itens e a função *desvio* são exibidos nas Figs. 4.35 e 4.36. Claramente, a cadeia $E + T*$ é um prefixo viável de (4.19). O autômato da Fig. 4.36 estará no estado I_7 após ter lido $E + T*$. O estado I_7 contém os itens

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

que são precisamente os itens válidos para $E + T*$. Para ver isto, consideremos as três seguintes derivações mais à direita

$$\begin{aligned} E' &\Rightarrow E \\ &\Rightarrow E + T \\ &\Rightarrow E + T * F \end{aligned} \qquad \begin{aligned} E' &\Rightarrow E \\ &\Rightarrow E + T \\ &\Rightarrow E + T * F \\ &\Rightarrow E + T * (E) \end{aligned} \qquad \begin{aligned} E' &\Rightarrow E \\ &\Rightarrow E + T \\ &\Rightarrow E + T * F \\ &\Rightarrow E + T * \mathbf{id} \end{aligned}$$

A primeira derivação mostra a validade de $T \rightarrow T * \cdot F$, a segunda a validade de $F \rightarrow \cdot(E)$ e a terceira a validade de $F \rightarrow \cdot\mathbf{id}$ para o prefixo viável $E + T*$. Pode ser mostrado que não existem outros itens válidos para $E + T*$, e deixamos a prova para o leitor interessado. □

Tabelas Sintáticas SLR

Agora vamos mostrar como construir as funções de ação e desvio a partir de um autômato finito determinístico que reconhece os prefixos viáveis. Nosso algoritmo não irá produzir tabelas de ações sintáticas unicamente definidas para toda e qualquer gramática, mas terá sucesso em muitas gramáticas para linguagens de programação. Dada uma gramática, G, nós a aumentamos de forma a produzir G' e a partir de G' construímos C, a coleção canônica de conjuntos de itens para G'. Construímos *ação*, a função de ações sintáticas e *desvio*, a função de desvio, a partir de C, usando o algoritmo seguinte. É requerido que conheçamos SEGUINTE (A) para cada não-terminal A da gramática (ver a Seção 4.4).

Algoritmo 4.8. Construção de uma tabela sintática SLR.

Entrada. Uma gramática aumentada G'.

Saída. As funções sintáticas SLR *ação* e *desvio* para G'.

Método.

1. Construir $C = \{I_0, I_1, \ldots, I_n\}$, a coleção de conjuntos de itens LR(0) para G'.
2. O estado i é construído a partir de I_i. As ações sintáticas para o estado i são determinadas como se segue:
   a) Se $[A \rightarrow \alpha \cdot a\beta]$ estiver em I_i e $desvio(I_i, a) = I_j$, então estabelecer *ação*$[i, a]$ em "empilhar j". Aqui, a precisa ser um terminal.
   b) Se $[A \rightarrow \alpha \cdot]$ estiver em I_i, então estabelecer a *ação*$[i, a]$ em "reduzir através de $A \rightarrow \alpha$", para todo a em SEGUINTE (A); aqui, A não pode ser S'.
   c) Se $[S' \rightarrow S \cdot]$ estiver em I_i então fazer *ação*$[i, \$]$ igual a "aceitar".

   Se quaisquer ações conflitantes forem geradas pelas regras anteriores, dizemos que a gramática não é SLR(1). O algoritmo falha em produzir um analisador sintático neste caso.
3. As transições de desvio para o estado i são construídas para todos os não-terminais A usando-se a seguinte regra: se $desvio(I_i, A) = I_j$, então $desvio[i, A] = j$.
4. Todas as entradas não definidas pelas regras (2) e (3) são tornadas "erro".
5. O estado inicial do analisador sintático é aquele construído a partir do conjunto de itens contendo $[S' \rightarrow \cdot S]$. □

A tabela sintática, contendo as funções sintáticas de ação e desvio, determinada pelo algoritmo 4.8, é chamada de *tabela SLR(1) para G*. Um analisador sintático LR usando uma tabela SLR(1) para G é denominado um analisador sintático SLR(1) para G e uma gramática tendo uma tabela sintática SLR(1) é dita ser *SLR(1)*. Usualmente omitimos o "(1)" após o símbolo SLR, uma vez que não iremos lidar aqui com analisadores sintáticos que tenham mais de um símbolo de *lookahead*.

Exemplo 4.38. Vamos construir a tabela SLR para a gramática (4.19). A coleção canônica de conjuntos de itens LR(0) para (4.19) foi mostrada na Fig. 4.35. Consideremos primeiro o conjunto de itens I_0:

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

O item $F \rightarrow \cdot(E)$ dá origem à entrada *ação*[0, (] = empilhar 4, o item $F \rightarrow \cdot\mathbf{id}$ à entrada *ação*[0, **id**] = empilhar 5. Os outros itens em I_0 não produzem ações. Agora consideremos I_1:

$$\begin{aligned} E' &\rightarrow E \cdot \\ E &\rightarrow E \cdot + T \end{aligned}$$

O primeiro item produz *ação*[1, \$] = aceitar, o segundo produz *ação*[1, +] = empilhar 6. Agora, consideremos I_2:

$$\begin{aligned} E &\rightarrow T \cdot \\ T &\rightarrow T \cdot * F \end{aligned}$$

Uma vez que SEGUINTE$(E) = \{\$, +,)\}$, o primeiro item torna *ação* [2, \$] = *ação* [2, +] = *ação* [2,)] reduzir $E \rightarrow T$. O segundo item torna *ação*[2, *] = empilhar 7. Continuando dessa forma obtemos as tabelas sintáticas de ações e de desvio que foram mostradas na Fig. 4.31. Naquela figura, os números de produções nas ações de redução são os mesmos que os da ordem na qual aparecem na gramática (4.18) original. Isto é, $E \rightarrow E + T$ é o número 1, $E \rightarrow T$ é 2 e assim por diante. □

Exemplo 4.39. Toda gramática SLR(1) é inambígua, mas existem muitas gramáticas inambíguas que não são SLR(1). Consideremos a gramática com as produções

$$\begin{aligned} S &\rightarrow L = R \\ S &\rightarrow R \\ L &\rightarrow * R \\ L &\rightarrow \mathbf{id} \\ R &\rightarrow L \end{aligned} \tag{4.20}$$

Podemos pensar de L e R como o valor-l e o valor-r, respectivamente, e de * como o operador indicando "conteúdo de".[3] A coleção canônica

[3] Como na Seção 2.8, um valor-l designa uma localização, e um valor-r é um valor que pode ser armazenado numa localização.

de conjuntos de itens LR(0) para a gramática (4.20) é mostrada na Fig. 4.37.

I_0: $S' \to \cdot S$
$S \to \cdot L = R$
$S \to \cdot R$
$L \to \cdot *R$
$L \to \cdot$ **id**
$R \to \cdot L$

I_1: $S' \to S \cdot$

I_2: $S \to L \cdot = R$
$R \to L \cdot$

I_3: $S \to R \cdot$

I_4: $L \to * \cdot R$
$R \to \cdot L$
$L \to \cdot *R$
$L \to \cdot$ **id**

I_5: $L \to$ **id** $\cdot$

I_6: $S \to L = \cdot R$
$R \to \cdot L$
$L \to \cdot *R$
$L \to \cdot$ **id**

I_7: $L \to *R \cdot$

I_8: $R \to L \cdot$

I_9: $S \to L = R \cdot$

Fig. 4.37. Coleção canônica LR(0) para a gramática (4.20).

Consideremos o conjunto de itens I_2. O primeiro item neste conjunto faz com que *ação*[2, =] = "empilhar 6". Uma vez que SEGUINTE*(R)* contém = (para ver por que, considere $S \Rightarrow L = R \Rightarrow * R = R$), o segundo item faz *ação*[2, =] igual a "reduzir para $R \to L$". Dessa forma, a entrada *ação*[2, =] é multiplamente definida. Uma vez que existe tanto uma entrada de empilhar quanto uma de reduzir em *ação*[2, =], o estado 2 possui um conflito de empilhar/reduzir no símbolo de entrada =.

A gramática (4.20) não é ambígua. O conflito de empilhar/reduzir emerge do fato de o método de construção do analisador sintático SLR não ser poderoso o suficiente para se lembrar o bastante do contexto à esquerda, de forma a decidir que ação o analisador sintático deverá tomar à entrada =, tendo examinado uma entrada redutível a L. Os métodos canônico e LALR, discutidos em seguida, irão ter sucesso sobre uma coleção mais ampla de gramáticas, incluindo a gramática (4.20). Deveria ser assinalado, entretanto, que existem gramáticas inambíguas para as quais qualquer método de construção de analisadores sintáticos LR irá produzir tabelas sintáticas com conflitos nas ações sintáticas. Felizmente, tais gramáticas podem ser geralmente evitadas nas aplicações de linguagens de programação. □

Construindo Tabelas Sintáticas LR Canônicas

Vamos agora apresentar a técnica mais geral para construir uma tabela sintática LR, a partir de uma gramática. Relembremos que, no método SLR, o estado i chama pela redução por $A \to \alpha$ se o conjunto de itens I_i, contiver o item $[A \to \alpha\cdot]$ e a estiver em SEGUINTE*(A)*. Em algumas situações, entretanto, quando o estado i aparecer no topo da pilha, o prefixo viável $\beta\alpha$ na pilha será tal que βA não poderá ser seguido por a numa forma sentencial à direita. Por conseguinte, uma redução através de $A \to \alpha$ será inválida à entrada a.

Exemplo 4.40. Vamos reconsiderar o Exemplo 4.39, onde, no estado 2, tínhamos o item $R \to L\cdot$, que corresponderia a $A \to \alpha$ acima, e a, que poderia ser o sinal de =, que está em SEGUINTE*(R)*. Dessa forma o analisador sintático SLR chama pela redução por $R \to L$ no estado 2 com = como a próxima entrada (a ação de empilhar também é chamada por causa do item $S \to L\cdot = R$ no estado 2). Entretanto, não existe forma sentencial à direita na gramática do Exemplo 4.39 que comece por $R = \cdots$. Conseqüentemente, o estado 2, que é o estado correspondente ao prefixo viável L somente, não deveria realmente chamar pela redução daquele L para R. □

É possível se carregar mais informações no estado, as quais irão proscrever algumas dessas reduções inválidas por $A \to \alpha$. Partindo-se os estados, quando necessário, podemos arranjar para ter cada estado de um analisador sintático LR indicando exatamente quais símbolos de entrada podem se seguir a um *handle* α para o qual exista uma possível redução para A.

A informação extra é incorporada dentro do estado pela redefinição dos itens, de forma a que incluam um símbolo terminal como um segundo componente. A forma geral de um item se torna $[A \to \alpha\cdot\beta, a]$, onde $A \to \alpha\beta$ é uma produção e a um terminal ou o marcador de fim à direita \$. Chamamos a um tal objeto de um *item LR(1)*. O 1 se refere ao comprimento do segundo componente, chamado de *lookahead* do item.[4] Esses *lookaheads* não possuem efeito num item da forma $[A \to \alpha\cdot\beta, a]$, onde β não seja ϵ, mas um item da forma $[A \to \alpha\cdot, a]$ chama pela redução $A \to \alpha$ somente se o próximo símbolo de entrada for a. Por conseguinte, somos compelidos a reduzir por $A \to \alpha$ somente naqueles símbolos de entrada para os quais $[A \to \alpha\cdot, a]$ for um item LR(1) no estado ao topo da pilha. O conjunto de tais a's será sempre um subconjunto de SEGUINTE*(A)*, mas poderia ser um subconjunto próprio, como no Exemplo 4.40.

Formalmente, dizemos que um item LR(1) $[A \to \alpha\cdot\beta, a]$ é *válido* para um prefixo viável γ se existir uma derivação $S \overset{*}{\underset{mdd}{\Rightarrow}} \delta A w \underset{mdd}{\Rightarrow} \delta\alpha\beta w$ onde

1. $\gamma = \delta\alpha$ e
2. ou a é o primeiro símbolo de w ou w é ϵ e a é \$.

Exemplo 4.41. Vamos considerar a gramática

$$S \to BB$$
$$B \to aB \mid b$$

Existe uma derivação mais à direita $S \overset{*}{\underset{mdd}{\Rightarrow}} aaBaB \underset{mdd}{\Rightarrow} aaaBab$. Podemos ver que o item $[B \to a\cdot B, a]$ é válido para o prefixo viável $\gamma = aaa$, fazendo $\delta = aa$, $A = B$, $w = ab$, $\alpha = a$ e $\beta = B$ na definição acima. Existe também uma derivação mais à direita $S \overset{*}{\underset{mdd}{\Rightarrow}} BaB \underset{mdd}{\Rightarrow} BaaB$. A partir desta derivação podemos ver que o item $[B \to a\cdot B, \$]$ é válido para o prefixo viável Baa. □

O método para a construção da coleção de conjuntos de itens LR(1) válidos é essencialmente o mesmo que aquele com que construímos a coleção canônica de conjuntos de itens LR(0). Necessitamos apenas modificar os dois procedimentos *fechamento* e *desvio*.

Para apreciarmos a nova definição da operação de *fechamento*, consideremos um item da forma $[A \to \alpha\cdot B\beta, a]$ no conjunto de itens válidos para algum prefixo viável γ. Existe, então, uma derivação mais à direita $S \overset{*}{\underset{mdd}{\Rightarrow}} \delta Aax \underset{mdd}{\Rightarrow} \delta\alpha B\beta ax$, onde $\gamma = \delta\alpha$. Suponhamos que βax derive uma cadeia de terminais by. Então, para cada produção da forma $\beta \to \eta$, para algum η, temos a derivação $S \overset{*}{\underset{mdd}{\Rightarrow}} \gamma Bby \underset{mdd}{\Rightarrow} \gamma\eta by$. Conseqüentemente, $[\beta \to \cdot\eta, b]$ é válido para γ. Note-se que b pode ser o primeiro terminal derivado a partir de β, ou é possível que β derive ϵ na derivação $\beta ax \overset{*}{\Rightarrow} by$ e b pode, conseqüentemente, ser a. Para sumarizar ambas as possibilidades, dizemos que b pode ser qualquer terminal em PRIMEIRO(βax), onde PRIMEIRO é a função da Seção 4.4. Note-se que x não pode conter o primeiro terminal de by, e, então, PRIMEIRO(βax) = PRIMEIRO(βa). Fornecemos agora a construção de conjuntos de itens LR(1).

Algoritmo 4.9. Construção dos conjuntos de itens LR(1).

Entrada. Uma gramática aumentada G'.

Saída. Os conjuntos de itens LR(1), que são os conjuntos de itens válidos para um ou mais prefixos viáveis de G'.

Método. Os procedimentos *fechamento* e *desvio* e a rotina principal *itens* para construir os conjuntos de itens são mostrados na Fig. 4.38. □

[4] Os *lookaheads* que sejam cadeias de comprimento maior do que 1 são possíveis, naturalmente, mas não os consideraremos aqui.

```
função fechamento (I);
início
    repetir
        para cada item [A → α·Bβ, a] em I,
            cada produção B → γ em G',
            e cada terminal b em PRIMEIRO(βa)
            tal que [B → ·γ, b] não está em I faça
            incluir [B → ·γ, b] em I;
    até que não possam ser adicionados mais itens a I;
    retornar I
fim;

função desvio (I, X);
início
        seja J o conjunto de itens [A → αX·β, a] tais que
            [A → α·Xβ, a] esteja em I;
        retornar fechamento (J)
fim;

procedimento itens(G');
início
    C := {fechamento ({[S' → ·S, $]})};
    repetir
        para cada conjunto de itens I em C e cada símbolo
            gramatical X tal que desvio (I, X) não é vazio
            e não está em C faça
            incluir desvio (I, X) a C
    até que não possam ser adicionados itens a C
fim
```

Fig. 4.38. Construção de conjuntos de itens LR(1) para a gramática G'.

Exemplo 4.42. Consideremos a seguinte gramática aumentada.

$$\begin{aligned} S' &\rightarrow S \\ S &\rightarrow CC \\ C &\rightarrow cC \mid d \end{aligned} \tag{4.21}$$

Começamos pelo cômputo do *fechamento* de $\{[S' \rightarrow \cdot S, \$]\}$. Para fechar, confrontamos o item $[S' \rightarrow \cdot S, \$]$ com o item $[A \rightarrow \alpha \cdot B\beta, a]$ no procedimento *fechamento*. Ou seja, $A = S'$, $\alpha = \epsilon$, $B = S$, $\beta = \epsilon$ e $a = \$$. A função *fechamento* nos diz para adicionar $[B \rightarrow \cdot\gamma, b]$ para cada produção $B \rightarrow \gamma$ e terminal b em PRIMEIRO(βa). Em termos da presente gramática, $B \rightarrow \gamma$ precisa ser $S \rightarrow CC$, e, uma vez que β é ϵ e a é \$, b pode somente ser \$. Conseqüentemente, adicionamos $[S \rightarrow \cdot C\,C, \$]$.

Continuamos a computar o fechamento, adicionando todos os itens $[C \rightarrow \cdot\gamma, b]$ para b em PRIMEIRO(C\$). Isto é, confrontando-se $[S \rightarrow \cdot CC, \$]$ com $[A \rightarrow \alpha \cdot B\beta, a]$ temos $A = S$, $\alpha = \epsilon$, $B = C$, $\beta = C$ e $a = \$$. Como C não deriva a cadeia vazia, PRIMEIRO(C\$) = PRIMEIRO(C). Uma vez que PRIMEIRO(C) contém os terminais c e d, adicionamos os itens $[C \rightarrow \cdot cC, c]$, $[C \rightarrow \cdot cC, d]$, $[C \rightarrow \cdot d, c]$ e $[C \rightarrow \cdot d, d]$. Nenhum dos novos itens possui um não-terminal imediatamente à direita do ponto, e, então, completamos nosso primeiro conjunto de itens LR(1). O conjunto inicial de itens é:

$$\begin{aligned} I_0\colon\quad & S' \rightarrow \cdot S, \$ \\ & S \rightarrow \cdot CC, \$ \\ & C \rightarrow \cdot cC, c/d \\ & C \rightarrow \cdot d, c/d \end{aligned}$$

Os colchetes foram omitidos por conveniência de notação e usamos a forma $[C \rightarrow \cdot cC, c/d]$ como uma abreviação para os dois itens $[C \rightarrow \cdot cC, c]$ e $[C \rightarrow \cdot cC, d]$.

Computamos agora *desvio*(I_0, X) para os vários valores de X. Para $X = S$ precisamos fechar o item $[S' \rightarrow S\cdot, \$]$. Nenhum fechamento adicional é possível, uma vez que o ponto está à extremidade direita. Por conseguinte, temos o próximo conjunto de itens:

$$I_1\colon S' \rightarrow S\cdot, \$$$

Para $X = C$, fechamos $[S \rightarrow C\cdot C, \$]$. Adicionamos as produções-C com segundo componente \$ e, então, não podemos adicionar mais, produzindo:

$$\begin{aligned} I_2\colon\quad & S \rightarrow C\cdot C, \$ \\ & C \rightarrow \cdot cC, \$ \\ & C \rightarrow \cdot d, \$ \end{aligned}$$

Em seguida, seja $X = c$. Precisamos fechar $\{[C \rightarrow c\cdot C, c/d]\}$. Adicionamos as produções-C com o segundo componente c/d, produzindo:

$$\begin{aligned} I_3\colon\quad & C \rightarrow c\cdot C, c/d \\ & C \rightarrow \cdot cC, c/d \\ & C \rightarrow \cdot d, c/d \end{aligned}$$

Finalmente, para $X = d$, decolamos com o conjunto de itens:

$$I_4\colon\quad C \rightarrow d\cdot, c/d$$

Terminamos de considerar *desvio* em I_0. Não temos conjuntos de I_1, mas I_2 possui *desvios* em C, c e em d. Em C obtemos:

$$I_5\colon\quad S \rightarrow CC\cdot, \$$$

e nenhum fechamento é necessário. Em c tomamos o fechamento de $\{[C \rightarrow c\cdot C, \$]\}$, para obter:

$$\begin{aligned} I_6\colon\quad & C \rightarrow c\cdot C, \$ \\ & C \rightarrow \cdot cC, \$ \\ & C \rightarrow \cdot d, \$ \end{aligned}$$

Note-se que I_6 difere de I_3 somente nos segundos componentes. Veremos que é comum vários conjuntos de itens LR(1) de uma gramática terem os mesmos primeiros componentes e diferirem nos segundos. Quando construímos a coleção de conjuntos de itens LR(0) para a mesma gramática, cada conjunto de itens LR(0) irá coincidir com o conjunto de primeiros componentes de um ou mais conjuntos de itens LR(1). Teremos mais a dizer sobre este fenômeno quando discutirmos a análise sintática LALR.

Continuando com a função *desvio* para I_2, *desvio*(I_2, d) é achado ser:

$$I_7\colon\quad C \rightarrow d\cdot, \$$$

Voltando agora para I_3, os *desvios* de I_3 para c e d são I_3 e I_4, respectivamente, e *desvio*(I_3, C) é:

$$I_8\colon\quad C \rightarrow cC\cdot, c/d$$

I_4 e I_5 não possuem *desvios*. Os *desvios* de I_6 em c e d são I_6 e I_7, respectivamente, e *desvio*(I_6, C) é:

$$I_9\colon\quad C \rightarrow cC\cdot, \$$$

Os conjuntos remanescentes de itens não produzem *desvios*, e, então, terminamos. A Fig. 4.39 mostra os dez conjuntos de itens com seus *desvios*. □

Fornecemos as regras pelas quais as funções sintáticas de ação e desvio LR(1) são construídas a partir dos conjuntos de itens LR(1). As funções de ação e desvio são representadas por uma tabela como antes. A única diferença está nos valores das entradas.

Algoritmo 4.10. Construção da tabela sintática LR canônica.

Entrada. Uma gramática aumentada G'.

Saída. As funções sintáticas canônicas LR *ação* e *desvio* para G'.

Método.

1. Construir $C = \{I_0, I_1, \ldots, I_n\}$, a coleção de conjuntos de itens LR(1) para G'.

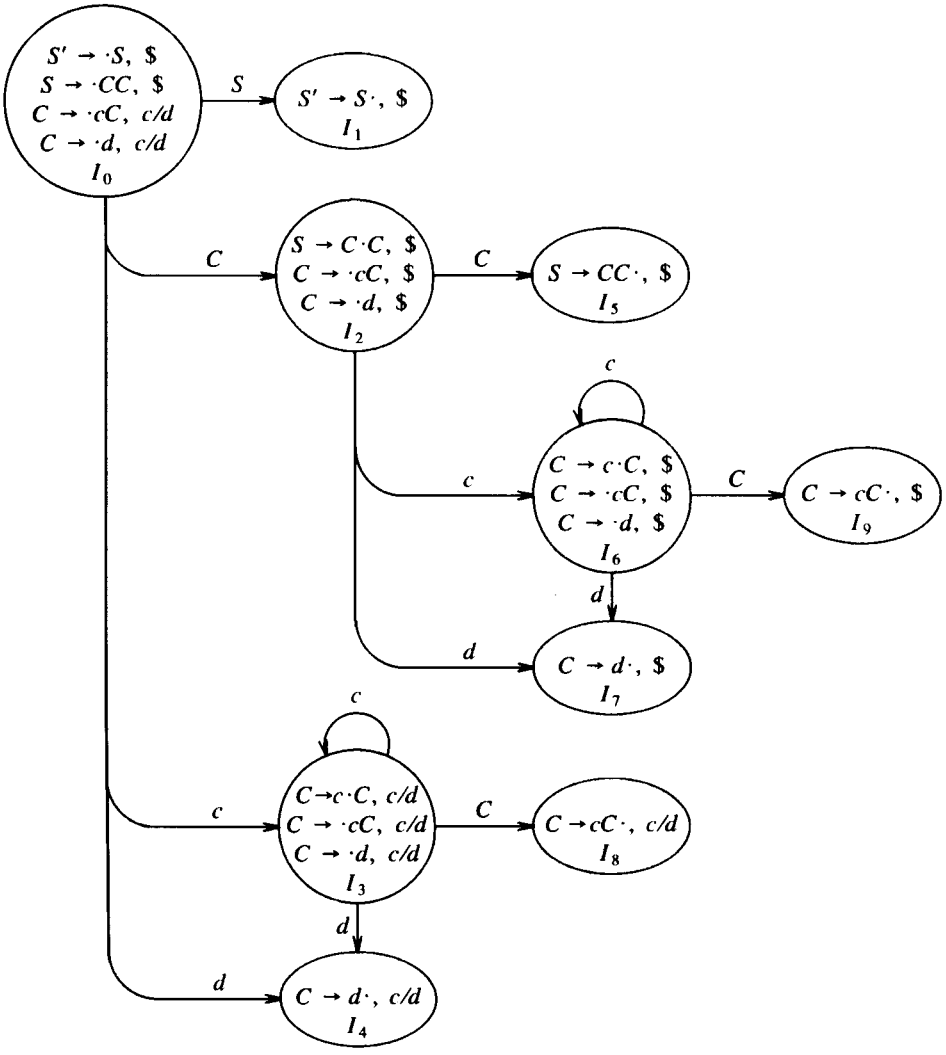


Fig. 4.39. O grafo *desvio* para a gramática (4.21).

2. O estado i do analisador sintático é construído a partir de I_i. As ações sintáticas para o estado i são determinadas como se segue:

   a) Se $[A \rightarrow \alpha \cdot a\beta, b]$ estiver em I_i e $desvio(I_i, a) = I_j$ então fazer *ação*$[i, a]$ igual a "empilhar j". Aqui, a é exigido ser um terminal.
   b) Se $[A \rightarrow \alpha \cdot, a]$ estiver em I_i, $A \neq S'$, então fazer *ação*$[i, a]$ igual a "reduzir $A \rightarrow \alpha$".
   c) Se $[S' \rightarrow S \cdot, \$]$ estiver em I_i, então fazer *ação*$[i, \$]$ igual a "aceitar".

Se um conflito resultar das regras acima, a gramática não é considerada LR(1) e o algoritmo falha.

3. As transições *desvio* para o estado i são determinadas como se segue: se $desvio(I_i, A) = I_j$ então $desvio[i, a] = j$.

4. Todas as entradas não definidas pelas regras (2) e (3) são tornadas "erro".
5. O estado inicial do analisador sintático é aquele construído a partir do conjunto contendo o item $[S' \rightarrow \cdot S, \$]$. □

A tabela formada a partir das funções sintáticas de ação e desvio, produzidas pelo algoritmo 4.10, é chamada tabela sintática LR(1) *canônica*. Um analisador sintático que utilize esta tabela é chamado de um analisador sintático LR(1) canônico. Se a função sintática de ação não possuir entradas multiplamente definidas, então a gramática é chamada uma *gramática LR(1)*. Como antes, omitimos o "(1)" se o mesmo for subentendido.

Exemplo 4.43. A tabela sintática canônica para a gramática (4.21) é mostrada na Fig. 4.40. As produções 1, 2 e 3 são $S \rightarrow CC$, $C \rightarrow cC$ e $C \rightarrow d$. □

Cada gramática SLR(1) possui uma gramática LR(1), mas, para uma gramática SLR(1), o analisador sintático LR pode ter mais esta-

ESTADO	*ação*			*desvio*	
	c	*d*	$	*S*	*C*
0	s3	s4		1	2
1			ac		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

Fig. 4.40. Tabela sintática canônica para a gramática (4.21).

dos do que o analisador sintático SLR para a mesma gramática. A gramática dos exemplos anteriores é SLR e possui um analisador sintático SLR com sete estados, comparados com os dez da Fig. 4.40.

Construindo Tabelas Sintáticas LALR

Vamos introduzir agora nosso último método de construção de analisadores sintáticos, a técnica LALR (*lookahead* LR). Este método é freqüentemente usado na prática porque as tabelas obtidas são consideravelmente menores do que as tabelas LR canônicas e, além do mais, a maioria das construções sintáticas comuns das linguagens de programação pode ser expressa convenientemente por gramáticas LALR. O mesmo é quase verdadeiro para as gramáticas SLR, mas existem umas poucas construções que não podem ser convenientemente tratadas pelas técnicas SLR (ver o Exemplo 4.39, por exemplo).

Para uma comparação do tamanho do analisador sintático, as tabelas SLR e LALR para uma gramática possuem o mesmo número de estados, e esse número é tipicamente de várias centenas para uma linguagem como Pascal. A tabela LR canônica teria tipicamente vários milhares de estados para uma linguagem com o mesmo tamanho. Conseqüentemente, é mais fácil e econômico construir tabelas SLR e LALR do que tabelas LR canônicas.

Como uma forma de introdução, vamos considerar de novo a gramática (4.21), cujos conjuntos de itens LR(1) foram mostrados na Fig. 4.39. Tomemos um par de estados aparentemente semelhantes, tais como I_4 e I_7. Cada um desses estados possui itens com o primeiro componente somente igual a $C \rightarrow d\cdot$. Em I_4, os *lookaheads* são c ou d; em I_7, \$ é o único *lookaheads*.

Para vermos as diferenças entre os papéis de I_4 e I_7 no analisador sintático, notemos que a gramática (4.21) gera o conjunto regular c^*dc^*d. Ao se ler uma entrada $cc \ldots cdcc \ldots cd$, o analisador sintático empilha o primeiro grupo de c's e seus d's seguintes, entrando no estado 4 após ter lido d. Chama em seguida pela redução $C \rightarrow d$, dado que o próximo símbolo de entrada é c ou d. A exigência de que c ou d venha em seguida faz sentido, pois esses são os símbolos que poderiam iniciar cadeias em c^*d. Se \$ se segue ao primeiro d, temos uma entrada como ccd, que não está na linguagem e o estado 4 declara corretamente um erro se \$ for o próximo símbolo de entrada.

O analisador sintático entra no estado 7 após ler o segundo d. Precisa em seguida enxergar \$ na entrada, ou começou a trabalhar numa cadeia que não está em c^*dc^*d. Por conseguinte, faz sentido que o estado 7 deva reduzir através de $C \rightarrow d$ à entrada \$ e declarar um erro com as entradas c ou d.

Vamos agora substituir I_4 e I_7 por I_{47}, a união de I_4 e I_7, consistindo do conjunto de três itens representados por $[C \rightarrow d\cdot, c/d/\$]$. Os desvios em d para I_4 ou I_7, a partir de I_0, I_2, I_3 e I_6, agora entram em I_{47}. A ação do estado 47 é de reduzir a qualquer entrada. O analisador sintático revisado se comporta essencialmente como o original, apesar de reduzir em circunstâncias em que o original declararia um erro, por exemplo, a uma entrada como ccd ou $cdcdc$. O erro será eventualmente capturado; de fato, o será antes que quaisquer outros símbolos de entrada sejam empilhados.

Mais geralmente, podemos olhar para os conjuntos de itens LR(1) como tendo o mesmo *núcleo*, isto é, conjunto de primeiros componentes, e podemos combinar esses conjuntos com núcleos comuns num único conjunto de itens. Por exemplo, na Fig. 4.39, I_4 e I_7 formam um tal par, com núcleo $\{C \rightarrow d\cdot\}$. Similarmente, I_3 e I_6 formam um outro par, com o núcleo $\{C \rightarrow c\cdot C, C \rightarrow \cdot cC, C \rightarrow \cdot d\}$. Existe mais um par, I_8 e I_9, com núcleo $\{C \rightarrow cC\cdot\}$. Note que, em geral, um núcleo é um conjunto de itens LR(0) para a gramática em exame e que uma gramática LR(1) pode produzir mais de dois conjuntos de itens com o mesmo núcleo.

Uma vez que o núcleo de *desvio(I, X)* depende somente do núcleo de I, os desvios dos conjuntos combinados podem também ser combinados. Dessa forma, não há problema em se revisar a função desvio na medida em que combinarmos conjuntos de itens. As funções de ação são modificadas de forma a refletir ações de não-erro para todos os conjuntos de itens que foram combinados.

Suponhamos ter uma gramática LR(1), isto é, uma cujos conjuntos de itens LR(1) não produzam conflitos nas ações sintáticas. Se substituirmos todos os estados que tenham o mesmo núcleo por sua união, é possível que a união resultante possua um conflito, mas é pouco provável pelo seguinte motivo: suponhamos que na união haja um conflito no *lookahead* a porque existe um item $[A \rightarrow \alpha\cdot, a]$ chamando por uma redução através de $A \rightarrow \alpha$ e exista um outro item $[B \rightarrow \beta\cdot a\gamma, b]$ chamando por um empilhamento. Então, algum conjunto de itens a partir dos quais a união foi realizada possui o item $[A \rightarrow \alpha; a]$ e, como os núcleos de todos esses estados são os mesmos, o conjunto também deve ter um item $[B \rightarrow \beta\cdot a\gamma, c]$, para algum c. Então, este estado possui o mesmo conflito de empilhar/reduzir em a e a gramática já não era LR(1) como foi assumido. Dessa forma, a combinação de estados com núcleos comuns jamais poderá produzir um conflito empilhar/reduzir que já não estivesse presente em algum dos estados originais, porque as ações de empilhar dependem somente do núcleo, nunca do *lookahead*.

É possível, entretanto, que o combinador produza um conflito reduzir/reduzir, como o seguinte exemplo mostra.

Exemplo 4.44. Consideremos a gramática

$$
\begin{aligned}
&S' \rightarrow S \\
&S \rightarrow aAd \mid bbd \mid bBe \mid bAe \\
&A \rightarrow c \\
&B \rightarrow c
\end{aligned}
$$

que gera as quatro cadeias acd, ace, bcd e bce. O leitor pode verificar que a gramática é LR(1) construindo os conjuntos de itens. Após fazer isso, encontramos o conjunto de itens $\{[A \rightarrow c\cdot, d], [B \rightarrow c\cdot, e]\}$, válido para o prefixo viável ac e $\{[A \rightarrow c\cdot, e], [B \rightarrow c\cdot, d]\}$, válido para bc. Nenhum desses conjuntos gera um conflito e seus núcleos são os mesmos. Sua união, entretanto, que é

$$
\begin{aligned}
&A \rightarrow c\cdot, d/e \\
&B \rightarrow c\cdot, d/e
\end{aligned}
$$

gera um conflito reduzir/reduzir, uma vez que as duas reduções, $A \rightarrow c$ e $B \rightarrow c$, são chamadas às entradas d e e. □

Estamos agora preparados para fornecer o primeiro dentre dois algoritmos de construção da tabela LALR. A idéia geral é a de construir os conjuntos de itens LR(1) e, se nenhum conflito emergir, combinar os conjuntos com núcleos comuns. Em seguida, construímos uma tabela sintática a partir da coleção conjuntos de itens combinados. O método que estamos prestes a descrever serve primariamente como uma definição das gramáticas LALR(1). A construção de toda a coleção de conjuntos de itens LR(1) requer espaço e tempo em demasia para ser útil na prática.

Algoritmo 4.11. Uma construção fácil da tabela sintática LALR, que, entretanto, ocupa muito espaço.

Entrada. Uma gramática aumentada G'.

Saída. As funções da tabela sintática LALR *ação* e *desvio* para G'.

Método.

1. Construir $C = \{I_0, I_1, \ldots, I_n\}$, a coleção conjuntos de itens LR(1).
2. Para cada núcleo presente entre os conjuntos de itens LR(1), encontrar todos os conjuntos que tenham o mesmo núcleo e substituí-los por sua união.
3. Seja $C' = \{J_0, J_1, \ldots, J_m\}$ a coleção de conjuntos de itens LR(1) resultante. As ações sintáticas para o estado i são construídas a partir de J_i da mesma maneira que no algoritmo 4.10. Se existir um conflito de ação sintática, o algoritmo falha em produzir um analisador sintático e a gramática não é considerada LALR(1).
4. A tabela *desvio* é construída como se segue. Se J for a união de um ou mais conjuntos de itens LR(1), isto é, $J = I_1 \cup I_2 \cup \ldots \cup I_k$, então os núcleos de *desvio*(I_1, X), *desvio*(I_2, X), ..., *desvio* (I_k, X) são os mesmos, uma vez que $I_1, I_2, \ldots, I_k$ possuem todos o mesmo núcleo. Seja K a união de todos os conjuntos de itens que tenham o mesmo núcleo que *desvio*(I_1, X). Então, *desvio* $(J, X) = K$. □

A tabela produzida pelo algoritmo 4.11 é chamada de *tabela sintática LALR* para G. Se não existirem conflitos de ações sintáticas, então a gramática é dita *LALR(1)*. A coleção de conjuntos de itens construída no passo (3) é chamada de *coleção LALR(1)*.

Exemplo 4.45. Consideremos de novo a gramática (4.21) cujo grafo de *desvio* foi mostrado na Fig. 4.39. Como mencionamos, existem três pares conjuntos de itens que podem ser combinados. I_3 e I_6 são substituídos por sua união:

$$\begin{array}{ll} I_{36}: & C \rightarrow c \cdot C,\ c/d/\$ \\ & C \rightarrow \cdot cC,\ c/d/\$ \\ & C \rightarrow \cdot d,\ c/d/\$ \end{array}$$

I_4 e I_7 são substituídos por sua união:

$$I_{47}: \quad C \rightarrow d \cdot,\ c/d/\$$$

e, analogamente, para I_8 e I_9:

$$I_{89}: \quad C \rightarrow cC \cdot,\ c/d/\$$$

As funções de ação e desvio LALR para os conjuntos condensados de itens são mostradas na Fig. 4.41.

ESTADO	*ação*			*desvio*	
	c	*d*	$	*S*	*C*
0	s36	s47		1	2
1			ac.		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

Fig. 4.41. Tabela sintática LALR para a gramática (4.21).

Para vermos como os desvios são computados, consideremos *desvio*(I_{36}, C). No conjunto original de itens LR(1), *desvio*(I_3, C) = I_8 e I_8 é agora parte de I_{89}, e, por conseguinte, fazemos *desvio*(I_{36}, C) ser I_{89}. Poderíamos ter chegado à mesma conclusão se considerássemos I_6, a outra parte de I_{36}. Isto é, *desvio*(I_6, C) = I_9 e I_9 é agora parte de I_{89}. Para um outro exemplo, consideremos *desvio*(I_2, c), uma entrada que é exercitada após a ação de empilhar de I_2 à entrada c. Nos conjuntos originais de itens LR(1), *desvio*(I_2, c) = I_6. Uma vez que I_6 é agora parte de I_{36}, *desvio*(I_2, c) se torna I_{36}. Conseqüentemente, a entrada na Fig. 4.41 para o estado 2 e entrada c é feita igual a s36, significando empilhar o símbolo de entrada e o estado 36. □

Quando apresenteados a uma cadeia proveniente da linguagem c^*dc^*d, tanto o analisador sintático LR da Fig. 4.40 quanto o LALR da Fig. 4.41 realizam exatamente a mesma seqüência de empilhamentos e reduções, apesar dos nomes dos estados na pilha poderem diferir; isto é, se o analisador sintático LR coloca I_3 ou I_6 na pilha, o analisador sintático LALR irá colocar I_{36}. O relacionamento vigora para uma gramática LALR em geral. Os analisadores sintáticos LR e LALR irão se imitar mutuamente para as entradas corretas.

Entretanto, quando apresentado a entradas incorretas, o analisador sintático LALR pode proceder a realização de algumas reduções após o analisador sintático LR ter declarado um erro, apesar do analisador LALR jamais empilhar um novo símbolo de entrada após o LR ter declarado um erro. Por exemplo, à entrada *ccd* seguida por $, o analisador sintático LR da Fig. 4.34 irá colocar

$$0\ c\ 3\ c\ 3\ d\ 4$$

na pilha e no estado 4 irá descobrir um erro porque $ é o próximo símbolo de entrada e o estado 4 possui uma ação de erro em $. Em contraste, o analisador LALR da Fig. 4.41 irá realizar os seguintes movimentos correspondentes, colocando

$$0\ c\ 36\ c\ 36\ d\ 47$$

na pilha. Mas o estado 47 à entrada $ possui a ação de reduzir através de $C \rightarrow d$. O analisador LALR irá conseqüentemente modificar a pilha para

$$0\ c\ 36\ c\ 36\ C\ 89$$

Agora, a ação do estado 89 à entrada $ é reduzir através de $C \rightarrow cC$. A pilha se torna, agora,

$$0\ c\ 36\ C\ 89$$

onde uma redução similar é chamada, obtendo a pilha

$$0\ C\ 2$$

Finalmente, o estado 2 possui uma ação de erro à entrada $, sendo então o erro agora descoberto. □

A Construção Eficiente de Tabelas Sintáticas LALR

Existem várias modificações que podemos fazer no algoritmo 4.11 para evitar a construção da coleção completa de conjuntos de itens LR(1), no processo de criação de uma tabela sintática LALR(1). A primeira observação é que podemos representar um conjunto de itens I pelo seu núcleo, isto é, por aqueles itens que ou são o item inicial $[S' \rightarrow \cdot S, \$]$ ou possuem o ponto em algum local que não o início de um lado direito.

Segundo, podemos computar as ações sintáticas geradas por I a partir do núcleo somente. Qualquer item que chame por uma redução através de $A \rightarrow \alpha$ estará no núcleo a menos que $\alpha = \epsilon$. A redução através de $A = \epsilon$ é invocada à entrada a se e somente se existir um item do núcleo $[B \rightarrow \gamma \cdot C\delta, b]$, tal que $C \overset{*}{\underset{mdd}{\Rightarrow}} A\eta$ para algum η e a estiver em PRIMEIRO($\eta\delta b$). O conjunto de não-terminais A tais que $C \overset{*}{\underset{mdd}{\Rightarrow}} A\eta$ pode ser pré-computado para cada não-terminal C.

As ações de empilhamento geradas por I podem ser determinadas a partir do núcleo de I como se segue. Empilhamos à entrada a se existir um item de núcleo $[B \rightarrow \gamma \cdot C\delta, b]$, onde $C \overset{*}{\underset{mdd}{\Rightarrow}} ax$ numa derivação na qual o último passo não usa uma produção-ϵ. O conjunto de tais a's pode também ser pré-computado para cada C.

Aqui está como as transições de desvio para I podem ser computadas a partir do núcleo. Se $[B \rightarrow \gamma \cdot X\delta, b]$ estiver no núcleo de I, então $[B \rightarrow \gamma X \cdot \delta, b]$ estará no núcleo de *desvio*(I, X). O item $[A \rightarrow X \cdot \beta, a]$ também estará no núcleo de *desvio* (I, X) se existir um item $[B \rightarrow \gamma \cdot C\delta, b]$ de I e $C \overset{*}{\underset{mdd}{\Rightarrow}} A\eta$ para algum η. Se pré-computarmos para cada par de não-terminais C e A se $C \overset{*}{\underset{mdd}{\Rightarrow}} A\eta$ para algum η, o cômputo dos conjuntos de itens a partir dos núcleos é só ligeiramente menos eficiente do realizado através dos conjuntos fechados de itens.

Para computar os conjuntos de itens LALR(1) de uma gramática aumentada G', começamos pelo núcleo $S' \rightarrow S$ do conjunto inicial de itens I_0. Em seguida, computamos os núcleos das transições de desvio a partir de I_0, como mencionado acima. Continuamos computando as transições de desvio para cada novo núcleo gerado até que tenhamos os núcleos de toda a coleção de itens LR(0).

Exemplo 4.46. Vamos considerar de novo a gramática aumentada

$$\begin{array}{l} S' \rightarrow S \\ S \rightarrow L = R \mid R \\ L \rightarrow * R \mid \mathbf{id} \\ R \rightarrow L \end{array}$$

Os núcleos dos conjuntos de itens LR(0) para esta gramática são mostrados na Fig. 4.42. □

Expandimos agora os núcleos atrelando a cada item LR(0) os símbolos *lookahead* apropriados (os segundos componentes). Para ver

I_0:	$S' \to \cdot S$	I_5:	$L \to \mathbf{id} \cdot$
I_1:	$S' \to S \cdot$	I_6:	$S \to L = \cdot R$
I_2:	$S \to L \cdot = R$	I_7:	$L \to * R \cdot$
	$R \to L \cdot$	I_8:	$R \to L \cdot$
I_3:	$S \to R \cdot$	I_9:	$S \to L = R \cdot$
I_4:	$L \to * \cdot R$		

Fig. 4.42. Núcleos dos conjuntos de itens LR(0) para a gramática (4.20).

como os símbolos *lookahead* se propagam de um conjunto de itens I para *desvio(I, X)*, consideremos um item LR(0) $B \to \gamma \cdot C\delta$ no núcleo de I. Suponhamos que $C \overset{*}{\underset{mdd}{\Rightarrow}} A\eta$ para algum η (talvez $C = A$) e $\eta = \epsilon$) e $A \to X\beta$ seja uma produção. Então o item LR(0) $A \to X \cdot \beta$ está em *desvio(I, X)*.

Suponhamos agora que estejamos computando não itens LR(0), mas itens LR(1) e $[B \to \gamma \cdot C\delta, b]$ esteja no conjunto I. Para que valores de a, então, irá $[A \to X \cdot \beta, a]$ estar em *desvio(I, X)*? Certamente se algum a estiver em PRIMEIRO($\eta\delta$), a derivação $C \overset{*}{\underset{mdd}{\Rightarrow}} A\eta$ nos diz que $[A \to X \cdot \beta, a]$ precisa estar em *desvio(I, X)*. Neste caso, o valor de b é irrelevante e dizemos que a, como um *lookahead* para $A \to A \cdot \beta$, é gerado *espontaneamente*. Por definição, \$ é gerado espontaneamente como um *lookahead* para o item $S' \to \cdot S$ no conjunto inicial de itens.

Mas existe uma outra fonte de *lookaheads* para o item $A \to X \cdot \beta$. Se $\eta\delta \overset{*}{\Rightarrow} \epsilon$, então $[A \to X \cdot \beta, b]$ também estará em *desvio(I, X)*. Dizemos neste caso que os *lookaheads* se *propagam* de $B \to \gamma \cdot C\delta$ para $A \to X \cdot \beta$. Um método simples para determinar quando um item LR(1) em I gera um *lookahead* em *desvio(I, X)* espontaneamente ou, quando os *lookaheads* se propagam, está contido no próximo algoritmo.

Algoritmo 4.12. Determinação dos *lookaheads*.

Entrada. O núcleo K de um conjunto de itens LR(0) I e um símbolo gramatical X.

Saída. Os *lookaheads* gerados espontaneamente pelos itens em I para os itens de núcleo em *desvio(I, X)* e os itens em I a partir dos quais os *lookaheads* são propagados para os itens de núcleo em *desvio (I, X)*.

Método. O algoritmo é fornecido na Fig. 4.43. É usado um símbolo fictício de *lookahead* # para detectar as situações nas quais o *lookahead* se propaga. □

para cada item $B \to \gamma \cdot \delta$ em K **faça início**
  J' := *fechamento* ($\{[B \to \gamma \cdot \delta, \#]\}$);
  se $[A \to \alpha \cdot X\beta, a]$ estiver em J' onde a não seja # **então**
    o *lookahead* a é gerado espontaneamente para o item
      $A \to \alpha X \cdot \beta$ em *desvio(I, X)*;
  se $[A \to \alpha \cdot X\beta, \#]$ estiver em J' **então**
    os *lookaheads* se propagam de $B \to \gamma \cdot \delta$ em I para
      $A \to \alpha X \cdot \beta$ em *desvio (I, X)*
fim

Fig. 4.43. Descobrindo os *lookaheads* propagados e os espontâneos.

Vamos agora considerar como faremos para encontrar os *lookaheads* associados aos itens nos núcleos dos conjuntos de itens LR(0). Primeiro, sabemos que \$ é um *lookahead* para $S' \to \cdot S$ no conjunto inicial de itens LR(0). O algoritmo 4.12 nos dá todos aqueles *lookaheads* gerados espontaneamente. Após listá-los todos, precisamos permitir que se propaguem até que nenhuma propagação ulterior seja possível. Existem muitas abordagens diferentes, todas elas em algum sentido controlando os *lookaheads* que se propagaram para um item mas que ainda não se propagaram para fora dele. O próximo algoritmo descreve uma técnica para propagar *lookaheads* para todos os itens.

Algoritmo 4.13. O cômputo eficiente dos núcleos da coleção de conjuntos de itens LALR(1).

Entrada. Uma gramática aumentada G'.

Saída. Os núcleos da coleção de itens LALR(1) para G'.

Método.

1. Usando o método delineado acima, construir os núcleos dos conjuntos de itens LR(0) para G.
2. Aplicar o algoritmo 4.12 ao núcleo de cada conjunto de itens LR(0) e símbolo gramatical X para determinar quais *lookaheads* são gerados espontaneamente para itens de núcleo em *desvio(I, X)* e para quais itens em I são propagados *lookaheads* para os itens de núcleo em *desvio(I, X)*.
3. Inicializar uma tabela que forneça, para cada item de núcleo em cada conjunto de itens, os *lookaheads* associados. Inicialmente, cada item tem associado a si somente aqueles *lookaheads* que foram determinados em (2) como tendo sido gerados espontaneamente.
4. Realizar repetidas passagens sobre os itens de núcleo em todos os conjuntos. Ao visitarmos um item i, procuramos por itens de núcleo para os quais i propague seus *lookaheads*, usando a informação tabulada em (2). O conjunto corrente de *lookaheads* para i é adicionado àqueles já associados a cada um dos itens para os quais i propague seus *lookaheads*. Continuamos realizando passagens sobre os itens de núcleo até que não possam ser propagados novos *lookaheads*. □

Exemplo 4.47. Vamos construir os núcleos dos itens LALR(1) para a gramática do exemplo anterior. Os núcleos dos itens LR(0) foram mostrados na Fig. 4.42. Quando aplicamos o algoritmo 4.12 ao núcleo do conjunto de itens I_0 computamos *fechamento* ($\{[S' \to \cdot S, \#]\}$), que é

$$
\begin{aligned}
S' &\to \cdot S, \# \\
S &\to \cdot L = R, \# \\
S &\to \cdot R, \# \\
L &\to \cdot * R, \#/= \\
L &\to \cdot \mathbf{id}, \#/= \\
R &\to \cdot L, \#
\end{aligned}
$$

Dois itens neste fechamento provocam a geração espontânea dos *lookaheads*. O item $[L \to \cdot * R, =]$ faz com que o *lookahead* = seja espontaneamente gerado para o item de núcleo $L \to * \cdot R$ em I_4 e o item $[L \to \cdot \mathbf{id}, =]$ faz com que = seja espontaneamente gerado para o item de núcleo $L \to \mathbf{id} \cdot$ em I_5.

O padrão de propagação de *lookaheads* entre os itens de núcleo determinados no passo (2) do algoritmo 4.13 está sumarizado na Fig. 4.44. Por exemplo, os desvios de I_0 para os símbolos S, L, R, * e **id** são, respectivamente, I_1, I_2, I_3, I_4, e I_5. Para I_0, computamos somente o fe-

De	Para
I_0: $S' \to \cdot S$	I_1: $S' \to S \cdot$
	I_2: $S \to L \cdot = R$
	I_3: $R \to L \cdot$
	I_3: $S \to R \cdot$
	I_4: $L \to * \cdot R$
	I_5: $L \to \mathbf{id} \cdot$
I_2: $S \to L \cdot = R$	I_6: $S \to L = \cdot R$
I_4: $L \to * \cdot R$	I_4: $L \to * \cdot R$
	I_5: $L \to \mathbf{id} \cdot$
	I_7: $L \to * R \cdot$
	I_8: $R \to L \cdot$
I_6: $S \to L = \cdot R$	I_4: $L \to * \cdot R$
	I_5: $L \to * \cdot R$
	I_8: $R \to L \cdot$
	I_9: $S \to L = R \cdot$

Fig. 4.44. Propagação de *lookaheads*.

CONJUNTO	ITEM	LOOKAHEADS			
		INÍCIO	PASSAGEM 1	PASSAGEM 2	PASSAGEM 3
I_0:	$S' \rightarrow \cdot S$	$	$	$	$
I_1:	$S' \rightarrow S \cdot$		$	$	$
I_2:	$S \rightarrow L \cdot = R$		$	$	$
I_3:	$R \rightarrow L \cdot$		$	$	$
I_4:	$L \rightarrow * \cdot R$	=	=/$	=/$	=/$
I_5:	$L \rightarrow$ **id**$\cdot$	=	=/$	=/$	=/$
I_6:	$S \rightarrow L = \cdot R$			$	$
I_7:	$L \rightarrow *R \cdot$		=	=/$	=/$
I_8:	$R \rightarrow L \cdot$		=	=/$	=/$
I_9:	$S \rightarrow L = R \cdot$				$

Fig. 4.45. Cômputo de *lookaheads*.

chamento do único item de núcleo singelo [$S' \rightarrow \cdot S$, #]. Conseqüentemente, $S' \rightarrow \cdot S$ propaga seu *lookahead* para o item de núcleo em I_1 até I_5.

Na Fig. 4.45, mostramos os passos (3) e (4) do algoritmo 4.13. A coluna rotulada `INICIO` mostra os *lookaheads* gerados espontaneamente para cada item nuclear. Na primeira passagem, o *lookahead* $ se propaga a partir de $S' \rightarrow S$ em I_0 para os seis itens listados na Fig. 4.44. O *lookahead* = se propaga a partir de $L \rightarrow *\cdot R$ em I_4 para os itens $L \rightarrow *R\cdot$ em I_7 e $R \rightarrow L\cdot$ em I_8. Também se propaga para $L \rightarrow$ **id**$\cdot$ em I_5, mas esses *lookaheads* já estão presentes. Na segunda e terceira passagens o único novo *lookahead* propagado é $, descoberto para os sucessores de I_2 e I_4 na passagem 2 e para o sucessor de I_6 à passagem 3. Nenhum novo *lookahead* é propagado à passagem 4, e, por conseguinte, o conjunto final de *lookaheads* é mostrado na coluna mais à direita da Fig. 4.45.

Note-se que o conflito de empilhar/reduzir, encontrado no Exemplo 4.39, ao se utilizar o método SLR, desapareceu com a técnica LALR. A razão está em que somente o *lookahead* $ está associado a $R \rightarrow L\cdot$ em I_2, e conseqüentemente não há conflito com a ação sintática de empilhar com =, gerada com o item $S \rightarrow L\cdot = R$ em I_2. □

Compactação das Tabelas Sintáticas LR

Uma gramática típica de linguagem de programação com 50 a 100 terminais e 100 produções pode ter uma tabela sintática LALR com várias centenas de estados. A função de ação pode facilmente ter 20.000 entradas, cada uma requerendo pelo menos 8 *bits* para codificar. Claramente, uma codificação mais eficiente do que um *array* bidimensional pode ser importante. Iremos rapidamente mencionar umas poucas técnicas que são usadas para comprimir os campos de ação e desvio de uma tabela sintática LR.

Uma técnica útil para compactar o campo de ação é reconhecer que usualmente muitas linhas da tabela de ação são idênticas. Por exemplo, na Fig. 4.40, os estados 0 e 3 possuem entradas idênticas de ação como, também, as entradas 2 e 6. Podemos, por conseguinte, ganhar um espaço considerável, com um pequeno custo de tempo, se criarmos um apontador parta cada estado num *array* unidimensional. Os apontadores dos estados com as mesmas ações apontam para a mesma localização. Para se ter acesso às informações desse *array*, associamos a cada terminal um número de zero até o número de terminais menos um, e usamos esse inteiro como um deslocamento para o valor do apontador de cada estado. Num dado estado, a ação sintática para o iésimo terminal será encontrada *i* localizações além do valor do apontador para aquele estado.

Uma eficiência adicional pode ser atingida às expensas de um analisador sintático um tanto mais lento (geralmente considerado um negócio razoável, uma vez que um analisador sintático LR consome somente uma pequena fração do tempo total de compilação) através da criação da lista para as ações de cada estado. A lista consiste em pares (símbolo-terminal, ação). A ação com maior número de ocorrências para um estado pode ser colocada ao fim da lista e no lugar do terminal podemos usar a notação "qualquer", significando que, se o símbolo de entrada não foi encontrado até então na lista, deveríamos realizar aquela ação, não importa o que a entrada seja. Sobretudo, as entradas de erro podem ser substituídas seguramente por ações de redução, para uma futura uniformidade ao longo de uma linha. Os erros serão detectados posteriormente, antes de um movimento para empilhar o símbolo de entrada.

Exemplo 4.48. Consideremos a tabela sintática da Fig. 4.31. Notemos primeiro que as ações para os estados 0, 4, 6 e 7 se combinam. Podemos representá-las todas pela lista:

SÍMBOLO	AÇÃO
id	s5
(	s4
qualquer	erro

O estado 1 possui uma lista similar:

+	s6
$	aceitar
qualquer	erro

No estado 2, podemos substituir as entradas de erro por r2, de tal forma que a redução através da produção 2 irá ocorrer a qualquer entrada, menos *. Conseqüentemente, a lista para o estado 2 é:

*	s7
qualquer	r2

O estado 3 possui somente as entradas erro e r4. Podemos substituir a primeira pela última, de tal forma que a lista para o estado 3 consista somente do par (qualquer, r4). Os estados 5, 10 e 11 podem ser tratados similarmente. A lista para o estado 8 é:

+	s6
)	s11
qualquer	erro

e para o estado 9:

*	s7
qualquer	r1

□

Podemos também codificar a tabela *desvio* através de uma lista, mas parece mais eficiente criar uma lista de pares para cada não-terminal *A*. Cada par da lista para *A* é da forma *(estado_corrente, próximo_estado)*, indicando

desvio [*estado_corrente*, *A*] = *próximo_estado*

Esta técnica é útil porque nela tende a haver um número menor de estados em qualquer coluna da tabela *desvio*. A razão está em que o desvio para o não-terminal *A* pode somente ser um estado derivável a partir de um conjunto de itens no qual alguns itens possuem *A* imediata-

mente à esquerda de um ponto. Nenhum conjunto possui itens com X e Y imediatamente à esquerda de um ponto se $X \neq Y$. Conseqüentemente, cada estado aparece no máximo em uma coluna de *desvio*.

Para maior redução de espaço, notamos que as entradas de erro na tabela de *desvio* nunca são consultadas. Por conseguinte, podemos substituir cada entrada de erro pela entrada de não-erro mais comum na sua coluna. Essa entrada se torna o *default;* é representada na lista para cada coluna por um par com "qualquer" em lugar do estado corrente.

Exemplo 4.49. Consideremos a Fig. 4.31 de novo. A coluna para F possui a entrada 10 para o estado 7 e todas as outras entradas ou são 3 ou são erro. Podemos substituir erro por 3 e criamos para a coluna F a lista:

estado_corrente	*próximo_estado*
7	10
qualquer	3

Similarmente, uma lista adequada para a coluna T é:

6	9
qualquer	2

Para a coluna E podemos escolher ou 1 ou 8 para ser o *default;* duas entradas são necessárias em cada caso. Por exemplo, poderíamos criar para a coluna E a lista

4	8
qualquer	1

□

Se o leitor totalizar o número de entradas nas listas criadas neste exemplo e no anterior e adicionar os apontadores provenientes dos estados para as listas de ação e dos não-terminais para as listas de próximo estado, não ficará impressionado com os ganhos de espaço sobre a implementação de matriz da Fig. 4.31. Não devemos ser enganados por este pequeno exemplo, entretanto. Para gramáticas reais, o espaço necessitado para a representação de listas é tipicamente menos do que dez por cento daquele necessitado pela representação sob a forma de matriz.

Deveríamos também assinalar que os métodos de compressão de tabelas para autômatos finitos que foram discutidos na Seção 3.9 podem também ser usados para representar tabelas sintáticas LR. A aplicação desses métodos é discutida nos exercícios.

4.8 USANDO GRAMÁTICAS AMBÍGUAS

É um teorema o fato de qualquer gramática ambígua falhar em ser LR e conseqüentemente não estar em quaisquer das classes discutidas na seção anterior. Certos tipos de gramáticas ambíguas, entretanto, são úteis na especificação e implementação de linguagens, como veremos nesta seção. Para construções de linguagens tais como expressões, uma gramática ambígua providencia uma especificação mais curta e natural do que uma gramática inambígua equivalente. Um outro uso das gramáticas ambíguas está em isolar construções sintáticas que ocorrem comumente para otimização de casos especiais. Com uma gramática ambígua, podemos especificar as construções de casos especiais adicionando cuidadosamente novas produções à mesma.

Poderíamos enfatizar que apesar das gramáticas que usamos serem ambíguas, em todos os casos podemos especificar regras de inambigüidade que permitem somente uma árvore gramatical para cada sentença. Desta forma, a especificação global da linguagem ainda se mantém inambígua. Frisamos também que as construções ambíguas deveriam ser usadas com parcimônia e de forma estritamente controlada; de outra forma não poderá haver garantia sobre que linguagem é reconhecida pelo analisador sintático.

Usando a Precedência e a Associatividade para Resolver Conflitos de Ações Sintáticas

Consideremos as expressões nas linguagens de programação. A seguinte gramática para expressões aritméticas com os operadores + e *

$$E \rightarrow E + E \mid E * E \mid (E) \mid \mathbf{id} \tag{4.22}$$

é ambígua, porque não especifica a associatividade ou a precedência dos operadores + e *. A gramática inambígua

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid \mathbf{id} \end{aligned} \tag{4.23}$$

gera a mesma linguagem, mas confere a + uma precedência menor do que * e torna ambos operadores associativos à esquerda. Existem duas razões pelas quais desejaríamos usar a gramática (4.22). Primeiro, como veremos, podemos facilmente modificar os níveis de associatividade e de precedência dos operadores + e * sem perturbar as produções de (4.22) ou o número de estados do analisador sintático resultante. Segundo, o analisador sintático para (4.23) irá gastar uma parte substancial de seu tempo reduzindo através das produções $E \rightarrow T$ e $T \rightarrow F$, cuja única função é garantir a associatividade e precedência. O analisador sintático para a Fig. (4.22) não irá desperdiçar tempo reduzindo através dessas produções *singelas,* como são chamadas.

I_0: $E' \rightarrow \cdot E$
$E \rightarrow \cdot E + E$
$E \rightarrow \cdot E * E$
$E \rightarrow \cdot (E)$
$E \rightarrow \cdot \mathbf{id}$

I_1: $E' \rightarrow E \cdot$
$E \rightarrow E \cdot + E$
$E \rightarrow E \cdot * E$

I_2: $E \rightarrow (\cdot E)$
$E \rightarrow \cdot E + E$
$E \rightarrow \cdot E * E$
$E \rightarrow \cdot (E)$
$E \rightarrow \cdot \mathbf{id}$

I_3: $E \rightarrow \mathbf{id} \cdot$

I_4: $E \rightarrow E + \cdot E$
$E \rightarrow \cdot E + E$
$E \rightarrow \cdot E * E$
$E \rightarrow \cdot (E)$
$E \rightarrow \cdot \mathbf{id}$

I_5: $E \rightarrow E * \cdot E$
$E \rightarrow \cdot E + E$
$E \rightarrow \cdot E * E$
$F \rightarrow \cdot (E)$
$E \rightarrow \cdot \mathbf{id}$

I_6: $E \rightarrow (E \cdot)$
$E \rightarrow E \cdot + E$
$E \rightarrow E \cdot * E$

I_7: $E \rightarrow E + E \cdot$
$E \rightarrow E \cdot + E$
$E \rightarrow E \cdot * E$

I_8: $E \rightarrow E * E \cdot$
$E \rightarrow E \cdot + E$
$E \rightarrow E \cdot * E$

I_9: $E \rightarrow (E) \cdot$

Fig. 4.46. Conjuntos de itens LR(0) para a gramática aumentada (4.22).

Os conjuntos de itens LR(0) para (4.22), aumentada por $E' \rightarrow E$, , são mostradas na Fig. 4.46. Uma vez que a gramática (4.22) é ambígua, os conflitos sintáticos serão gerados quando tentarmos produzir uma tabela sintática LR a partir dos conjuntos de itens. Os estados correspondentes aos conjuntos de itens I_7 e I_8 geram esses conflitos. Suponhamos que usemos a abordagem SLR para construir a tabela de ações sintáticas. O conflito gerado por I_7, entre reduzir através de $E \rightarrow E + E$ e empilhar + ou *, não pode ser resolvido porque + e * estão, ambos, em SEGUINTE(E). Por conseguinte, ambas as ações deveriam ser chamadas para as entradas + e *. Um conflito similar é gerado em I_8, entre reduzir através de $E \rightarrow E * E$ e empilhar às entradas + e *. De fato, cada um de nossos métodos de construção de tabelas sintáticas irá gerar esses conflitos.

Entretanto, esses problemas podem ser resolvidos usando as informações de precedência e associatividade para + e *. Consideremos a entrada **id** + **id** * **id**, que faz com que o analisador sintático baseado na Fig. 4.46 entre no estado 7 após processar **id** + **id**; em particular, o analisador sintático atinge a configuração

PILHA	ENTRADA
$0\,E\,1 + 4\,E\,7$	* **id** $

Assumindo que * tenha precedência sobre +, sabemos que o analisador sintático empilharia *, preparando para reduzir o * e seus **id**'s envolventes à uma expressão. Isto é o que o analisador sintático SLR da Fig. 4.31, para a mesma linguagem, faria e é o que um analisador sintático de precedência de operadores faz. Por outro lado, se + tem precedência sobre *, sabemos que o analisador sintático deveria reduzir $E + E$ para E. Por conseguinte, a precedência relativa de + seguido por * determina unicamente como o conflito de ações sintáticas entre reduzir através de $E \rightarrow E + E$ e empilhar *, no estado 7, deveria ser resolvido.

Se, por outro lado, a entrada fosse **id** + **id** + **id,** o analisador sintático atingiria uma configuração na qual tivesse a pilha $0E1 + 4E7$, após processar a entrada **id** + **id.** A entrada + existe de novo um conflito empilhar/reduzir no estado 7. Agora, entretanto, a associatividade do operador + determina como esse conflito deverá ser resolvido. Se + for associativo à esquerda, a ação correta é reduzir através de $E \rightarrow E + E$. Isto é, os **id**'s que envolvem o primeiro + precisam ser agrupados primeiro. De novo, esta escolha coincide com o que analisadores sintáticos SLR e de precedência de operadores fariam para a gramática do Exemplo 4.34.

Em síntese, assumindo que + seja associativo à esquerda, a ação do estado 7 à entrada + deveria ser reduzida através de $E \rightarrow E + E$ e, assumindo que * tenha precedência sobre +, a ação do estado 7 à entrada * deveria ser empilhar. Semelhantemente, assumindo que * seja associativo à esquerda e tenha precedência sobre +, podemos argumentar que o estado 8, que pode aparecer no topo da pilha somente quando $E * E$ forem os três símbolos gramaticais de topo, deveria ter a ação de reduzir $E \rightarrow E * E$ às entradas + e *. No caso da entrada +, a razão está em que * tem precedência sobre +, enquanto que no caso da entrada *, argumentamos que * seja associativo à esquerda.

Procedendo dessa forma, obtemos a tabela sintática LR mostrada na Fig. 4.47. As produções 1-4 são $E \rightarrow E + E$, $E \rightarrow E * E$, $E \rightarrow (E)$ e $E \rightarrow$ **id,** respectivamente. É interessante que uma tabela de ações sintáticas similar seria produzida pela eliminação das reduções através de produções singelas $E \rightarrow T$ e $T \rightarrow F$ a partir da tabela SLR para a gramática (4.23), mostrada na Fig. 4.31. As gramáticas ambíguas como (4.22) podem ser tratadas numa forma similar no contexto das análises sintáticas LALR e LR canônica.

ESTADO	*ação*						*desvio*
	id	+	*	(	)	$	*E*
0	s3			s2			1
1		s4	s5			ac.	
2	s3			s2			6
3		r4	r4		r4	r4	
4	s3			s2			8
5	s3			s2			8
6		s4	s5		s9		
7		r1	s5		r1	r1	
8		r2	r2		r2	r2	
9		r3	r3		r3	r3	

Fig. 4.47. Tabela sintática para a gramática (4.22).

A Ambigüidade do "*Else*-Vazio"

Consideremos de novo a seguinte gramática para enunciados condicionais:

$$\begin{array}{lll} cmd & \rightarrow & \textbf{if}\ expr\ \textbf{then}\ cmd\ \textbf{else}\ cmd \\ & | & \textbf{if}\ expr\ \textbf{then}\ cmd \\ & | & \textbf{outro} \end{array}$$

Como notamos na Seção 4.3, esta gramática é ambígua porque não resolve a ambigüidade do *else*-vazio. Para simplificar a discussão, vamos considerar uma abstração da gramática acima, onde *i* está no lugar de **if** *expr* **then**, *e* no de **else** e *a* no de "todas as demais produções". Podemos, então, escrever a gramática, com a produção adicional $S' \rightarrow S$, como:

$$\begin{array}{l} S' \rightarrow S \\ S \rightarrow iSeS \mid iS \mid a \end{array} \qquad (4.24)$$

Os conjuntos de itens LR(0) para a gramática (4.24) são mostrados na Fig. 4.48. A ambigüidade em (4.24) dá origem a um conflito de empilhar/reduzir em I_4. Nela, $S \rightarrow iS{\cdot}eS$ chama por um empilhamento de *e* e, uma vez que SEGUINTE(S) = {*e*, $}, o item $S \rightarrow iS{\cdot}$ chama por uma redução através de $S \rightarrow iS$ à entrada *e*.

Voltando à terminologia do **if . . . then . . . else,** dado que tivéssemos

if *expr* **then** *cmd*

na pilha e **else** como o próximo símbolo de entrada, deveríamos empilhar **else** (isto é, empilhar *e*) ou reduzir **if** *expr* **then** *cmd* para *cmd* (isto é, reduzir através de $S \rightarrow iS$)? A resposta é que deveríamos empilhar **else,** porque está "associado" ao **then** prévio. Na terminologia da gramática (4.24), o *e* à entrada, figurando no lugar de **else,** pode somente compor uma parte do lado direito que começa por *iS*, no topo da pilha. Se o que se seguir ao *e* na entrada não puder ser estruturado como um *S*, completando o lado direito *iSeS*, então pode ser mostrado que não existe outra estruturação possível.

Somos levados a concluir que o conflito de empilhar/reduzir em I_4 deveria ser resolvido em favor de empilhar à entrada *e*. A tabela sintática SLR, construída a partir dos conjuntos de itens da Fig. 4.48, usando esta solução para o conflito de ações sintáticas em I_4 à entrada *e* é mostrada na Fig. 4.49. As produções 1 a 3 são $S \rightarrow iSeS$, $S \rightarrow iS$ e $S \rightarrow a$, respectivamente.

Por exemplo, à entrada *iiaea*, o analisador sintático realiza os movimentos mostrados na Fig. 4.50, correspondentes à solução corre-

I_0: $S' \rightarrow {\cdot}S$
$S \rightarrow {\cdot}iSeS$
$S \rightarrow {\cdot}iS$
$S \rightarrow {\cdot}a$

I_1: $S' \rightarrow S{\cdot}$

I_2: $S \rightarrow i{\cdot}SeS$
$S \rightarrow i{\cdot}S$
$S \rightarrow {\cdot}iSeS$
$S \rightarrow {\cdot}iS$
$S \rightarrow {\cdot}a$

I_3: $S \rightarrow a{\cdot}$

I_4: $S \rightarrow iS{\cdot}eS$
$S \rightarrow iS{\cdot}$

I_5: $S \rightarrow iSe{\cdot}S$
$S \rightarrow {\cdot}iSeS$
$S \rightarrow {\cdot}iS$
$S \rightarrow {\cdot}a$

I_6: $S \rightarrow iSeS{\cdot}$

Fig. 4. 48. Estados LR(0) para a gramática aumentada (4.24).

ESTADO	*ação*				*desvio*
	i	*e*	*a*	$	*S*
0	s2		s3		1
1				ac.	
2	s2		s3		4
3		r3		r3	
4		s5		r2	
5	s2		s3		6
6		r1		r1	

Fig. 4.49. Tabela sintática LR para a gramática abstrata do "*else*-vazio".

	PILHA	ENTRADA
(1)	0	*iiaea*$
(2)	0*i* 2	*iaea*$
(3)	0*i* 2*i* 2	*aea*$
(4)	0*i* 2*i* 2*a* 3	*ea*$
(5)	0*i* 2*i* 2*S* 4	*ea*$
(6)	0*i* 2*i* 2*S* 4*e* 5	*a*$
(7)	0*i* 2*i* 2*S* 4*e* 5*a* 3	$
(8)	0*i* 2*i* 2*S* 4*e* 5*S* 6	$
(9)	0*i* 2*S* 4	$
(10)	0*S* 1	$

Fig. 4.50. Ações sintáticas realizadas para a entrada *iiaea*.

ta do *else*-vazio. À linha (5), o estado 4 seleciona a ação de empilhar à entrada *e*, enquanto que à linha (9), o estado 4 chama pela redução através de $S \rightarrow iS$ com $ à entrada.

Como um padrão de comparação, se formos incapazes de usar uma gramática ambígua para especificar enunciados condicionais, teremos que usar uma gramática mais inteligente ao longo das linhas de (4.9).

Ambigüidades para Produções Tipo Caso Especial

Nosso exemplo final a sugerir a utilidade das gramáticas ambíguas ocorre se introduzirmos uma produção adicional para especificar um caso especial de uma construção sintática gerada numa forma mais geral pelo resto da gramática. Ao adicionarmos a produção extra, geramos o conflito de ações sintáticas. Podemos freqüentemente resolver o conflito satisfatoriamente através de uma regra de inambigüidade que oriente para reduzir através da produção do caso especial. A ação semântica associada à produção adicional permite que o caso especial seja manipulado por um mecanismo mais específico.

Um uso interessante das produções de caso especial foi feito por Kernighan e Cherry [1975] em seu pré-processador de composição de equações EQN, que foi usado para auxiliar a composição do original desta tradução. Em EQN, a sintaxe de uma expressão matemática é descrita por uma gramática que usa o operador de subscritos **sub** e um operador de superescritos **sup**, como mostrado no fragmento de gramática (4.25). As chaves são usadas pelo pré-processador para envolver expressões compostas e *c* é usado como um *token* representando qualquer cadeia de texto.

$$
\begin{array}{ll}
(1) & E \rightarrow E \ \mathbf{sub}\ E \ \mathbf{sup}\ E \\
(2) & E \rightarrow E \ \mathbf{sub}\ E \\
(3) & E \rightarrow E \ \mathbf{sup}\ E \\
(4) & E \rightarrow \{ E \} \\
(5) & E \rightarrow c
\end{array}
\qquad (4.25)
$$

A gramática (4.25) é ambígua por várias razões. Não especifica nem a associatividade e nem precedência dos operadores **sub** e **sup.** Mesmo que resolvêssemos as ambigüidades que emergem da associatividade e precedência de **sub** e **sup**, digamos, tornando esses operadores de igual precedência e associativos à direita, a gramática ainda seria ambígua. Isto porque a produção (1) isola um caso especial de produções geradas pelas produções (2) e (3), especificamente, as produções da forma E **sub** E **sup** E. A razão para se tratar de maneira especial expressões com essa conformação é que muitos compositores de tipos prefeririam montar uma expressão como a **sub** i **sup** 2 como a_i^2 ao invés de a_i^2. Adicionando-se meramente uma produção tipo caso especial, Kernighan e Cherry habilitaram EQN a produzir esse caso especial de saída.

Para ver como este tipo de ambigüidade pode ser tratado na abordagem LR, vamos construir um analisador sintático SLR para a gramática (4.25). Os conjuntos de itens LR(0) para esta gramática são mostrados na Fig. 4.51. Nesta coleção, três conjuntos de itens produzem conflitos de ações sintáticas. I_7, I_8 e I_{11} geram conflitos de empilhar/reduzir nos *tokens* **sub** e **sup** porque a associatividade e precedência desses operadores não foi especificada. Resolvemos esses conflitos de ações sintáticas quando fazemos **sub** e **sup** com a mesma precedência e associativos à direita. Por conseguinte, empilhar é a ação de escolha em cada caso.

I_0: $E' \rightarrow \cdot E$
$E \rightarrow \cdot E$ **sub** E **sup** E
$E \rightarrow \cdot E$ **sub** E
$E \rightarrow \cdot E$ **sup** E
$E \rightarrow \cdot \{ E \}$
$E \rightarrow \cdot c$

I_1: $E' \rightarrow E \cdot$
$E \rightarrow E \cdot$**sub** E **sup** E
$E \rightarrow E \cdot$**sub** E
$E \rightarrow E \cdot$**sup** E

I_2: $E \rightarrow \{ \cdot E \}$
$E \rightarrow \cdot E$ **sub** E **sup** E
$E \rightarrow \cdot E$ **sub** E
$E \rightarrow \cdot E$ **sup** E
$E \rightarrow \cdot \{ E \}$
$E \rightarrow \cdot c$

I_3: $E \rightarrow c \cdot$

I_4: $E \rightarrow E$ **sub**$\cdot E$ **sup** E
$E \rightarrow E$ **sub**$\cdot E$
$E \rightarrow \cdot E$ **sub** E **sup** E
$E \rightarrow \cdot E$ **sub** E
$E \rightarrow \cdot E$ **sup** E
$E \rightarrow \cdot \{ E \}$
$E \rightarrow \cdot c$

I_5: $E \rightarrow E$ **sup**$\cdot E$
$E \rightarrow \cdot E$ **sub** E **sup** E
$E \rightarrow \cdot E$ **sub** E
$E \rightarrow \cdot E$ **sup** E
$E \rightarrow \cdot \{ E \}$
$E \rightarrow \cdot c$

I_6: $E \rightarrow E \cdot$**sub** E **sup** E
$E \rightarrow E \cdot$**sub** E
$E \rightarrow E \cdot$**sup** E
$E \rightarrow \{ E \cdot \}$

I_7: $E \rightarrow E \cdot$**sub** E **sup** E
$E \rightarrow E$ **sub** $E \cdot$**sup** E
$E \rightarrow E \cdot$**sub** E
$E \rightarrow E$ **sub** $E \cdot$
$E \rightarrow E \cdot$**sup** E

I_8: $E \rightarrow E \cdot$**sub** E **sup** E
$E \rightarrow E \cdot$**sub** E
$E \rightarrow E \cdot$**sup** E
$E \rightarrow E$ **sup** $E \cdot$

I_9: $E \rightarrow \{ E \} \cdot$

I_{10}: $E \rightarrow E$ **sub** E **sup**$\cdot E$
$E \rightarrow E$ **sup**$\cdot E$
$E \rightarrow \cdot E$ **sub** E **sup** E
$E \rightarrow \cdot E$ **sub** E
$E \rightarrow \cdot E$ **sup** E
$E \rightarrow \cdot \{ E \}$
$E \rightarrow \cdot c$

I_{11}: $E \rightarrow E \cdot$**sub** E **sup** E
$E \rightarrow E$ **sub** E **sup** $E \cdot$
$E \rightarrow E \cdot$**sub** E
$E \rightarrow E \cdot$**sup** E
$E \rightarrow E$ **sup** $E \cdot$

Fig. 4.51. Conjuntos de itens LR(0) para a gramática (4.25).

I_{11} também gera um conflito reduzir/reduzir, no caso das entradas } e $, entre as duas produções

$$
\begin{array}{l}
E \rightarrow E \ \mathbf{sub}\ E \ \mathbf{sup}\ E \\
E \rightarrow E \ \mathbf{sup}\ E
\end{array}
$$

O estado I_{11} estará no topo da pilha quando tivermos enxergado uma entrada que tenha sido reduzida para E **sub** E **sup** na pilha. Se resolvermos o conflito reduzir/reduzir em favor da produção (1), estaremos tratando uma equação da forma E **sub** E **sup** E como um caso especial. Usando essas regras de inambigüidade, obtemos a tabela sintática SLR mostrada na Fig. 4.52.

Escrever gramáticas inambíguas que fatorem casos especiais de construções sintáticas é muito difícil. Para apreciar esse grau de dificuldade, o leitor é convidado a construir uma gramática inambígua equivalente para (4.25) que isole expressões da forma E **sub** E **sup** E.

Recuperação de Erros na Análise Sintática LR

Um analisador sintático LR irá detectar um erro ao consultar a tabela de ações sintáticas e encontrar uma entrada de erro. Os erros nunca são

ESTADO	*ação*						*desvio*
	sub	**sup**	{	}	*c*	$	*E*
0			s2		s3		1
1	s4	s5				ac.	
2			s2		s3		6
3	r5	r5		r5		r5	
4			s2		s3		7
5			s2		s3		8
6	s4	s5		s9			
7	s4	s10		r2		r2	
8	s4	s5		r3		r3	
9	r4	r4		r4		r4	
10			s2		s3		11
11	s4	s5		r1		r1	

Fig. 4.52. Tabela sintática para a gramática (4.25).

detectados pela consulta à tabela de desvio. Diferentemente de um analisador sintático de precedência de operadores, um analisador LR irá anunciar erro tão logo não exista continuação válida para a porção da entrada processada até então. Uma análise sintática canônica LR jamais irá realizar uma única redução sequer, antes de anunciar um erro. Os analisadores sintáticos SLR e LR podem realizar várias reduções antes de enunciar um erro, mas jamais irão empilhar um símbolo incorreto de entrada.

Na análise sintática LR, podemos implementar a recuperação na modalidade do desespero como segue. Podemos varrer a pilha até que o estado *s* com um desvio para um símbolo não-terminal particular *A* seja encontrado. Zero ou mais símbolos de entrada são descartados até que seja encontrado um símbolo *a* que possa legitimamente se seguir a *A*. O analisador sintático empilha, então, o estado *desvio*[*s*, *A*] e continua a análise sintática normal. Pode haver mais de uma escolha para o não-terminal *A*. Normalmente, esses seriam não-terminais representando trechos maiores do programa, tais como expressões, enunciados ou blocos. Por exemplo, se *A* fosse o não-terminal *cmd*, *a* poderia ser um ponto-e-vírgula ou **end.**

Este método de recuperação tenta isolar a frase que contém o erro sintático. O analisador sintático determina que uma cadeia derivável a partir de *A* contém um erro. Parte daquela cadeia já foi processada e o resultado é uma seqüência de estados no topo da pilha. O resto da cadeia ainda está na entrada e o analisador tenta pular sobre o resto dessa cadeia, procurando por um símbolo da entrada que possa legitimamente se seguir a *A*. Através da remoção dos estados que estão na pilha, pulando sobre a entrada e empilhando *desvio*[*s*, *A*], o analisador sintático presume ter encontrado uma instância de *A* e reassume a análise sintática normal da entrada.

A recuperação em nível de frase é implementada através do exame de cada entrada de erro na tabela sintática LR e decidindo-se, com base no uso da linguagem, que erro do programador é mais propenso a dar origem àquele outro detectado. Um procedimento apropriado de recuperação pode ser então construído: presumivelmente, o topo da pilha e/ou os primeiros símbolos de entrada seriam modificados numa forma considerada apropriada a cada entrada de erro.

Comparado com analisadores sintáticos de precedência de operadores, o projeto de rotinas específicas de tratamento de erro para um analisador sintático LR é relativamente fácil. Em particular, não temos que nos preocupar com as reduções equivocadas; qualquer redução pedida por um analisador sintático LR está certamente correta. Podemos, então, preencher cada entrada em branco no campo ação com um apontador para uma rotina de erro que irá tomar a ação apropriada selecionada pelo projetista do compilador. As ações devem incluir a inserção ou remoção de símbolos da pilha, da entrada ou de ambas, ou a alteração e transposição de símbolos da entrada, exatamente como para um analisador sintático de precedência de operadores. Como aquele analisador, precisamos fazer nossas escolhas sem permitir a possibilidade do analisador sintático LR entrar num laço infinito. Uma estratégia que assegure que pelo menos um símbolo de entrada será removido ou eventualmente empilhado ou que a pilha será eventualmente encurtada, se o fim da entrada tiver sido atingido, é suficiente para esse propósito. Deve ser evitado remover um estado que cubra um não-terminal, porque essa mudança elimina da pilha uma construção que já foi estruturada com sucesso.

Exemplo 4.50. Consideremos de novo a gramática de expressões.

$$E \rightarrow E + E \mid E * E \mid (E) \mid \mathbf{id}$$

A Fig. 4.53 mostra a tabela sintática LR da Fig. 4.47 para esta gramática, modificada para a detecção e recuperação de erros. Modificamos cada estado que chame por uma redução particular em alguns símbolos de entrada, substituindo as entradas de erro naquele estado pela redução. Essa modificação possui o efeito de postergar a detecção de erros até que uma ou mais reduções sejam feitas, mas o erro ainda será capturado antes que tome lugar qualquer movimento de empilhar. As entradas de erro remanescentes, provenientes da Fig. 4.47, foram substituídas por chamadas para rotinas de erro.

ESTADO	*ação*						*desvio*
	id	+	*	(	)	$	*E*
0	s3	e1	e1	s2	e2	e1	1
1	e3	s4	s5	e3	e2	ac.	
2	s3	e1	e1	s2	e2	e1	6
3	r4	r4	r4	r4	r4	r4	
4	s3	e1	e1	s2	e2	e1	7
5	s3	e1	e1	s2	e2	e1	8
6	e3	s4	s5	e3	s9	e4	
7	r1	r1	s5	r1	r1	r1	
8	r2	r2	r2	r2	r2	r2	
9	r3	r3	r3	r3	r3	r3	

Fig. 4.53. Tabela sintática LR com rotinas de erro.

As rotinas de erro são como segue. A similaridade dessas ações, com os erros que elas representam para as ações de erro do Exemplo 4.32 (precedência de operadores) deveria ser notada. Entretanto, o caso e1 do analisador sintático LR é freqüentemente manipulado pelo processador de reduções do analisador sintático de precedência de operadores.

e1: /* Esta rotina é chamada a partir dos estados 0, 2, 4 e 5, isto é, aqueles estados que esperam pelo início de um operando, ou **id** ou um parênteses à esquerda. Em lugar, um operador, + ou *, ou o fim da entrada foi encontrada. */
empilhar um **id** imaginário e cobri-lo com o estado 3 (o desvio dos estados 0, 2, 4 e 5 para **id**)[5]

e2: /* Esta rotina é chamada a partir dos estados 0, 1, 2, 4 e 5 ao se encontrar um parênteses à direita. */
remover os parênteses à direita a partir da entrada
emitir o diagnóstico "parênteses à direita não balanceado"

e3: /* Esta rotina é chamada a partir dos estados 1 ou 6 ao se esperar um operador e um **id** ou parênteses à direita. */
empilhar + e cobri-lo com o estado 4.
emitir o diagnóstico "operador ausente"

e4: /* Esta rotina é chamada a partir do estado 6 quando o fim da entrada é atingido. O estado 6 espera um operador ou parênteses à direita. */
empilhar um parênteses à direita e cobri-lo com o estado 9.
emitir o diagnóstico "parênteses à direita ausente"

Ao se receber a entrada incorreta **id** +), discutida no Exemplo 4.32, a seqüência de configurações atingidas pelo analisador sintático é mostrada na Fig. 4.54. □

[5]Note que na prática os símbolos gramaticais não são colocados na pilha. É útil imaginá-los lá de forma a nos lembrarmos dos símbolos que os estados "representam".

PILHA	ENTRADA	MENSAGENS DE ERRO E AÇÕES
0	**id**+)$	
0**id**3	+)$	
0*E*1	+)$	
0*E*1 + 4	)$	
0*E*1 + 4	$	"parênteses à direita não balanceado" e2 remove parênteses à direita
0*E*1 + 4**id**3	$	"operando ausente" e1 empilha **id** 3
0*E*1 + 4*E*7	$	
0*E*1	$	

Fig. 4.54. Análise sintática e os movimentos para a recuperação de erros feitos por um analisador sintático LR.

4.9 GERADORES DE ANALISADORES SINTÁTICOS

Esta seção mostra como um gerador de analisadores sintáticos pode ser usado para facilitar a construção da vanguarda de um compilador. Iremos usar o gerador de analisadores sintáticos LALR Yacc como base de nossa discussão, uma vez que o mesmo implementa muitos dos conceitos discutidos nas duas seções anteriores e está amplamente disponível. Yacc significa "mais um compilador de compiladores"*, refletindo a popularidade dos geradores de analisadores sintáticos ao início dos anos 70, quando a primeira versão do Yacc foi criada por S. C. Johnson. Yacc está disponível como um comando do sistema UNIX e tem sido usado para auxiliar a implementação de centenas de compiladores.

O Gerador de Analisadores Sintáticos Yacc

Um tradutor pode ser construído utilizando-se o Yacc da forma ilustrada na Fig. 4.55. Primeiro, um arquivo, digamos `translate.y`,

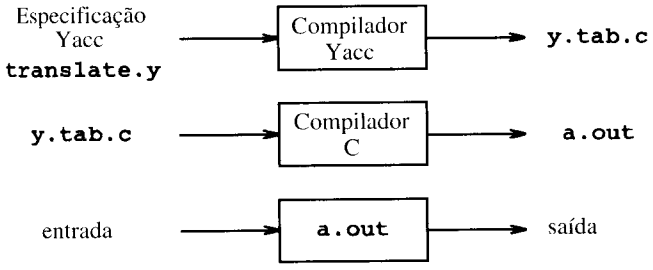


Fig. 4.55. Criando um tradutor de entrada/saída através do Yacc.

contendo uma especificação Yacc do tradutor é preparada. O comando de sistema UNIX

```
yacc translate.y
```

transforma o arquivo `translate.y` num programa C chamado `y.tab.c` usando o método LALR delineado no algoritmo 4.13. O programa `y.tab.c` é uma representação de um analisador sintático LALR escrito em C, juntamente com outras rotinas C que o usuário poderia ter preparado. A tabela sintática LALR é compactada como descrito na Seção 4.7. Através da compilação de `y.tab.c` juntamente com a biblioteca `ly` que contém o programa de análise sintática LR, usando o comando

```
cc y.tab.c -ly
```

obtemos o programa-objeto desejado `a.out`, que realiza a tradução especificada pelo programa Yacc original.[6] Se outros procedimentos forem necessitados, podem ser compilados ou carregados com `y.tab.c` exatamente como em qualquer programa C.

Um programa-fonte Yacc possui três partes:

```
declarações
%%
regras de tradução
%%
rotinas de suporte C
```

Exemplo 4.51. Para ilustrar o preparo de um programa-fonte Yacc, vamos construir uma calculadora de mesa simples que leia uma expressão aritmética, a avalie e imprima o seu valor numérico. Iremos construir a calculadora de mesa com a seguinte gramática para expressões aritméticas:

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid \textbf{dígito}$$

O *token* **dígito** é um único dígito entre 0 e 9. Um programa Yacc para a calculadora de mesa, derivado desta gramática é mostrado na Fig. 4.56. □

```
%{
#include <ctype.h>
%}

%token DIGITO

%%
linha  : expr '\n'     { printf("%d\n", $1); }
       ;
expr   : expr '+' termo     { $$ = $1 + $3; }
       | termo
       ;
termo  : termo '*' fator    { $$ = $1 * $3; }
       | fator
       ;
fator  : '(' expr ')'       { $$ = $2; }
       | DIGITO
       ;

%%
yylex () {
       int c;
       c = getchar ();
       if (isdigit (c)) {
          yylval = c - '0';
          return DIGITO;
       }
       return c;
}
```

Fig. 4.56. Especificação Yacc de uma calculadora de mesa simples.

A parte de declaração. Existem duas seções opcionais na parte de declarações de um programa Yacc. Na primeira, colocamos declarações C ordinárias, delimitadas por %{ e %}. Aqui colocamos declarações de quaisquer variáveis temporárias usadas pelas regras de tradução das segunda e terceira seções. Na Fig. 4.56, esta seção contém somente o enunciado *include*

```
#include <ctype.h>
```

que causa o pré-processador C incluir o arquivo de cabeçalho-padrão `<ctype.h>` que contém o predicado `isdigit`.

*Do original em inglês: "Yet Another Compiler-Compiler". O termo "yet" é usado apenas para dar ênfase ao significado pretendido (N. do T.).

[6] O nome `ly` é dependente do sistema.

Também na parte de declarações estão os *tokens* da gramática. Na Fig. 4.56, o enunciado

```
%token DIGITO
```

declara `DIGITO` como um *token*. Os *tokens* declarados nessa seção podem ser usados nas segunda e terceira partes da especificação Yacc.

A parte das regras de tradução. Na parte da especificação Yacc após o primeiro par `%%`, colocamos as regras de tradução. Cada regra consiste numa produção gramatical e da ação semântica associada. Um conjunto de produções que vínhamos escrevendo como

<lado esquerdo> → <alt 1> | <alt 2> | . . . | <alt *n*>

seria escrito em Yacc como

```
<lado esquerdo> : <alt 1> { ação semântica 1 }
                | <alt 2> { ação semântica 2 }
                . . .
                | <alt n> { ação semântica n }
                ;
```

Numa produção Yacc, o caractere singelo entre apóstrofos `'c'` é considerado o símbolo terminal `c`, e cadeias de letras e dígitos sem apóstrofos, não declarados como *tokens*, são considerados não-terminais. Lados direitos alternativos podem ser separados por barras verticais e um ponto-e-vírgula se segue a cada lado esquerdo com suas alternativas e ações semânticas. O primeiro lado esquerdo é considerado ser o símbolo de partida.

Uma ação semântica Yacc é uma seqüência de enunciados em C. Numa ação semântica, o símbolo `$$` se refere ao valor de atributo associado ao não-terminal à esquerda, enquanto que `$i` se refere ao valor associado ao iésimo símbolo gramatical (não-terminal ou terminal) à direita. A ação semântica é realizada sempre que reduzirmos através da produção associada, de tal forma que normalmente a ação semântica computa um valor para `$$` em função dos `$i`'s. Na especificação Yacc, escrevemos as duas produções-*E*

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

e suas ações semânticas associadas como

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

Note que o não-terminal termo na primeira produção é o terceiro símbolo gramatical à direita, enquanto que `'+'` é o segundo. Uma ação semântica associada à primeira produção adiciona o valor de `expr` e o de `termo` à direita e atribui o resultado como o valor do não-terminal `expr` à esquerda. Omitimos a ação semântica para a segunda produção, uma vez que a cópia do valor é a ação *default* para produções com um único símbolo gramatical à direita. Em geral, `{ $$ = $1; }` é a ação semântica *default*.

Note que adicionamos uma nova produção de partida

```
linha  : expr '\n'  {printf ("%d\n", $1); }
```

à especificação Yacc. A produção diz que uma entrada para a calculadora de mesa deve ser uma expressão seguida por um caractere de avanço de linha. A ação semântica associada a esta produção imprime o valor decimal da expressão seguida por um caractere de avanço de linha.

A parte de rotinas C de suporte. A terceira parte de uma especificação Yacc consiste em rotinas C de suporte. Um analisador léxico com o nome `yylex( )` precisa ser providenciado. Outros procedimentos, tais como rotinas de recuperação de erros, podem ser adicionados na medida do necessário.

O analisador léxico `yylex( )` produz pares consistindo em um *token* e o valor de atributo associado. Se um *token* tal como `DIGITO` for retornado, terá que ter sido declarado na primeira seção da especificação Yacc. O valor de atributo associado e um *token* é comunicado ao analisador sintático através da variável definida pelo Yacc `yylval`.

O analisador léxico da Fig. 4.56 é tosco. Lê caracteres de entrada, um de cada vez, usando a função C `getchar( )`. Se o caractere for um dígito, o valor do mesmo é armazenado na variável `yylval` e o token `DIGITO` é retornado. De outra forma, o próprio caractere é retornado como o *token*.

Usando o Yacc com Gramáticas Ambíguas

Vamos agora modificar a especificação Yacc de tal forma que calculadora de mesa resultante se torne mais útil. Primeiro, iremos permitir que a mesma avalie uma seqüência de expressões, uma a cada linha. Iremos também permitir linhas em branco entre expressões. Fazemos isso modificando a primeira regra para

```
linhas : linhas expr '\n'  {printf ("%g\n", $2); }
       | linhas '\n'
       |
       ;
```

Em Yacc, uma alternativa vazia, como a terceira, denota ϵ.

Segundo, vamos expandir a classe de expressões de forma a incluir números em lugar de dígitos isolados e também os operadores aritméticos +, − (ambos unários e binários), * e /. A forma mais fácil de especificar esta classe de expressões é usar a gramática ambígua

$$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid (E) \mid -E \mid \textbf{número}$$

A especificação Yacc resultante é mostrada na Fig. 4.57.

Uma vez que a gramática de especificação da Fig. 4.57 é ambígua, o algoritmo LALR irá gerar conflitos de ações sintáticas. Yacc irá reportar o número de conflitos de ações gerados. Uma descrição dos conjuntos de itens e dos conflitos de ações sintáticas pode ser obtida invocando-se o Yacc com a opção `-v`. Esta opção gera um arquivo adicional, `y.output`, que contém os núcleos dos conjuntos de itens encontrados na análise sintática, uma descrição dos conflitos gerados entre as ações sintáticas pelo algoritmo LALR e uma representação legível da tabela sintática LR, mostrando como os conflitos entre as ações sintáticas foram resolvidos. Sempre que Yacc relata que encontrou um conflito de ação sintática, é inteligente criar e consultar o arquivo `y.output` para verificar por que os conflitos foram gerados e se foram resolvidos corretamente.

A menos que seja orientado de outra forma, o Yacc irá resolver todos os conflitos de ações sintáticas usando as duas regras seguintes:

1. Um conflito de reduzir/reduzir é resolvido através da escolha das produções conflitantes listadas à frente na especificação Yacc. Por conseguinte, para se tomar a decisão correta na gramática de composição de tipos (4.25), é suficiente listar a produção (1) à frente da produção (3).
2. Um conflito de empilhar/reduzir é resolvido em favor de empilhar. Esta regra resolve o conflito empilhar/reduzir que emerge da ambigüidade da gramática do *else*-vazio corretamente.

Uma vez que essas regras *default* podem nem sempre ser o que o escritor do compilador deseja, o Yacc providencia um mecanismo geral para resolver conflitos de empilhar/reduzir. Na parte de declarações, podemos associar precedências e associatividades aos terminais. A declaração

```
%left '+' '-'
```

faz com que + e − tenham a mesma precedência e sejam associativos à esquerda. Podemos declarar um operador como associativo à direita dizendo

```
%right '^'
```

```
%{
#include <ctype.h>
#include <stdio.h>
#define YYSTYPE double /* real de precisão dupla para os da pilha do Yacc */
%}

%token NUMERO
%left '+' '-'
%left '*' '/'
%right UMINUS

%%
linhas        :  linhas expr  '\n'    { printf("%g\n", $2); }
              |  linhas '\n'
              |  /* ε */
              ;
expr          :  expr '+' expr    { $$ = $1 + $3; }
              |  expr '-' expr    { $$ = $1 - $3; }
              |  expr '*' expr    { $$ = $1 * $3; }
              |  expr '/' expr    { $$ = $1 / $3; }
              |  '(' expr ')'     { $$ = $2; }
              |  '-' expr %prec UMINUS     { $$ = - $2; }
              |  NUMERO
              ;
%%
yylex() {
     int c;
     while ( ( c = getchar() ) == '' );
     if (( c == '.' )||( isdigit(c)) ) {
        ungetc(c, stdin);
        scanf("%lf", &yylval);
        return NUMERO;
     }
     return c;
}
```

Fig. 4.57. Especificação Yacc para uma calculadora de mesa mais avançada.

e podemos forçar um operador a ser binário não associativo (isto é, duas ocorrências do operador não podem ser combinadas de todo) dizendo

```
%nonassoc '<'
```

Aos *tokens* são dadas precedências na ordem em que aparecem na parte de declarações, menores à frente. Os *tokens* na mesma declaração possuem a mesma precedência. Conseqüentemente, a declaração

```
%right UMINUS
```

na Fig. 4.57 confere ao token `UMINUS` um nível de precedência maior do que aqueles dos cinco terminais precedentes.

Yacc resolve os conflitos empilhar/reduzir atrelando uma precedência e associatividade a cada produção envolvida num conflito, bem como para cada terminal igualmente envolvido. Se tiver que escolher entre empilhar um símbolo de entrada a e reduzir através da produção $A \rightarrow \alpha$, Yacc reduz se a precedência da produção for maior do que aquela de a ou se as precedências forem as mesmas e a associatividade da produção for `left`. De outra forma, empilhar será a ação escolhida.

Normalmente, a precedência de uma produção é considerada a mesma que a do terminal mais à direita. Esta é uma decisão sensível na maioria dos casos. Por exemplo, dadas as produções

$$E \rightarrow E + E \mid E * E$$

preferiríamos reduzir através de $E \rightarrow E + E$ com o *lookahead* +, porque o + ao lado direito possui a mesma precedência que o *lookahead*, mas é associativo à esquerda. Com o *lookahead* *, preferiríamos empilhar, porque o *lookahead* possui maior precedência do que o + na produção.

Naquelas situações em que o terminal mais à direita não confere a precedência adequada a uma produção, podemos forçar uma precedência atrelando a uma produção o rótulo

```
%prec <terminal>
```

A precedência e associatividade da produção será então a mesma que a daquele terminal, que presumivelmente está definido na seção de declarações. O Yacc não relata os conflitos de empilhar/reduzir que são resolvidos usando-se este mecanismo de precedência e associatividade.

Este "terminal" pode ser um guardador de lugar, como o `UMINUS` na Fig. 4.57; este terminal não é retornado pelo analisador léxico mas é declarado somente para definir uma precedência para a produção. Na Fig. 4.57, a declaração

```
%right UMINUS
```

atribui uma precedência que é maior do que aquela de * e /. Na parte de regras de tradução, o rótulo

```
%prec UMINUS
```

ao final da produção

```
expr   :   '-' expr
```

faz com que o operador menos unário nesta produção tenha uma precedência maior do que a de qualquer outro operador.

Criando Analisadores Léxicos Yacc com Lex

Lex foi projetado para produzir analisadores léxicos que poderiam ser usados com Yacc. A biblioteca Lex `ll` irá providenciar um programa diretor denominado `yylex()`, o nome requerido pelo Yacc para seu analisador léxico. Se Lex for usado para produzir o analisador léxico, substituímos a rotina `yylex()` na terceira parte da especificação de Yacc pelo enunciado

```
#include "lex.yy.c"
```

e teremos que cada ação Lex retornará um terminal conhecido pelo Yacc. Pelo uso do enunciado `#include "lex.yy.c"`, o programa `yylex` tem acesso aos nomes dos *tokens* de Yacc, uma vez o arquivo de saída Lex é compilado como parte do arquivo de saída `y.tab.c`

Sob o sistema UNIX, se a especificação Lex estiver no arquivo `first.l` e a especificação Yacc em `second.y` podemos dizer

```
lex first.l
yacc second.y
cc y.tab.c-ly-ll
```

para obter o tradutor desejado.

A especificação Lex na Fig. 4.58 pode ser usada em lugar do analisador léxico na Fig. 4.57. O último padrão é `\n¦.` uma vez que .em Lex reconhece qualquer caractere que não seja um avanço de linha.

```
numero     [0-9]+\.?¦[0-9]*\.[0-9]+
%%
[ ]        { /* saltar espaços */ }
{numero}   { sscanf(yytext, "%lf", &yylval);
             return NUMERO; }
\n¦.       { return yytext[0]; }
```

Fig. 4.58. Especificação Lex para `yylex()` na Fig. 4.57.

Recuperação de Erros em Yacc

Em Yacc, a recuperação de erros pode ser realizada utilizando-se uma forma de regra de produção de erro. Primeiro, o usuário decide que não-terminais "mais importantes" terão uma recuperação de erros associada a si. As escolhas típicas são para algum subconjunto dos não-terminais que geram expressões, enunciados, blocos e procedimentos. O usuário, então, adiciona à gramática produções de erro da forma $A \rightarrow$ **error** α, onde A é um terminal importante e α uma cadeia de símbolos gramaticais, possivelmente a cadeia vazia; **error** é uma palavra reservada Yacc. Yacc irá gerar um analisador sintático a partir de tal especificação, tratando as produções de erro como produções ordinárias.

Entretanto, quando o analisador sintático gerado por Yacc encontra um erro, trata aqueles estados cujos conjuntos de itens contêm produções de erro de uma forma especial. Ao encontrar um erro, Yacc desempilha símbolos até que encontre no topo da pilha um estado cujo conjunto subjacente de itens inclua um item da forma $A \rightarrow \cdot$**error** α. O analisador sintático "empilha" um *token* fictício **error,** como se o tivesse enxergado na entrada.

Quando α é ϵ, uma redução para A ocorre imediatamente e a ação semântica associada à produção $A \rightarrow \cdot$**error** (que poderia ser uma rotina de recuperação de erro especificada pelo usuário) é invocada. O analisador sintático passa a descartar símbolos de entrada até que encontre um a partir do qual a análise sintática normal possa prosseguir.

Se α não for vazio, Yacc pula à frente na entrada, procurando por uma subcadeia que possa ser reduzida a α. Se α for constituído inteiramente de terminais, procura por essa cadeia de terminais na entrada e os "reduz" empilhando-os. A esse ponto, o analisador sintático terá **error** α ao topo da pilha. Reduzirá, então, **error** α a A e reassumirá a análise sintática normal.

Por exemplo, uma produção de erro da forma

$$cmd \rightarrow \textbf{error ;}$$

especificaria que o analisador sintático deveria saltar até o próximo ponto-e-vírgula ao enxergar um erro e assumir que um comando foi encontrado. A rotina semântica para esta produção de erro não necessitaria manipular a entrada, mas poderia gerar uma mensagem de diagnóstico e estabelecer um sinalizador para inibir a geração de código objeto, por exemplo.

Exemplo 4.52. A Fig. 4.59 mostra a calculadora de mesa Yacc da Fig. 4.57 com a produção de erro

```
linhas  :  error '\n'
```

```
%{
#include <ctype.h>
#include <stdio.h>
#define YYSTYPE double /* real de precisão dupla para os da pilha do Yacc */
%}

%token NUMERO
%left '+' '-'
%left '*' '/'
%right UMINUS

%%
linhas  :  linhas expr '\n'  { printf("%g\n", $2); }
        ¦  linhas '\n'
        ¦  /* vazio */
        ¦  error'\n' { yyerror("redigite a última linha :");yyerrok; }
        ;
expr    :  expr '+' expr     { $$ = $1 + $3; }
        ¦  expr '-' expr     { $$ = $1 - $3; }
        ¦  expr '*' expr     { $$ = $1 * $3; }
        ¦  expr '/' expr     { $$ = $1 / $3; }
        ¦  '(' expr ')'      { $$ = $2; }
        ¦  '-' expr %prec UMINUS  { $$ = - $2; }
        ¦  NUMERO
        ;
%%
#include "lex·yy·c"
```

Fig. 4.59. Calculadora de mesa com recuperação de erros.

Esta produção de erro faz com que a calculadora de mesa suspenda a análise sintática quando um erro for encontrado na entrada. Ao encontrar o erro, o analisador sintático da calculadora de mesa começa a remover símbolos da pilha até que encontre um estado que tenha uma ação de empilhar ao *token* **error.** O estado 0 é um desses estados (neste exemplo, é o único estado), uma vez que seus itens incluem

$$linhas \rightarrow \cdot\ \mathbf{error}\ \texttt{'\textbackslash n'}$$

Igualmente, o estado 0 está sempre ao fundo da pilha. O analisador sintático empilha o *token* **error** e então prossegue para saltar à frente na entrada até que tenha encontrado um caractere de avanço de linha. A esse ponto o analisador sintático empilha o caractere de avanço de linha, reduz **error** `'\n'` para *linhas* e emite a mensagem de diagnóstico "redigite a última linha:". A rotina Yacc especial `yyerrok` restabelece o analisador sintático em seu modo normal de operação. □

EXERCÍCIOS

4.1 Considere a gramática

$$S \rightarrow (\ L\ ) \mid a$$
$$L \rightarrow L\ ,\ S \mid S$$

a) Quais são os terminais, não-terminais e o símbolo de partida?
b) Encontre as árvores gramaticais para as seguintes sentenças:
   i) *(a, a)*
   ii) *(a, (a, a))*
   iii) *(a, ((a, a), (a, a)))*
c) Construa uma derivação mais à esquerda para cada uma das sentenças em (b).
d) Construa uma derivação mais à direita para cada uma das sentenças em (b).
*e) Que linguagem esta gramática gera?

4.2 Considere a gramática

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

a) Mostre que esta gramática é ambígua, construindo duas derivações mais à esquerda diferentes para a sentença *abab*.
b) Construa as derivações mais à direita correspondentes para *abab*.
c) Construa as árvores gramaticais correspondentes para *abab*.
*d) Que linguagem esta gramática gera?

4.3 Considere a gramática

$$bexpr \rightarrow bexpr\ \mathbf{or}\ btermo \mid btermo$$
$$btermo \rightarrow btermo\ \mathbf{and}\ bfator \mid bfator$$
$$bfator \rightarrow \mathbf{not}\ bfator \mid (\ bexpr\ ) \mid \mathbf{true} \mid \mathbf{false}$$

a) Construa uma árvore gramatical para a sentença **not (true or false).**
b) Mostre que esta gramática gera todas as expressões booleanas.
*c) Esta gramática é ambígua? Por quê?

4.4 Considere a gramática

$$R \rightarrow R\ ' \mid '\ R \mid RR \mid R^* \mid (R) \mid a \mid b$$

Note que a primeira barra vertical é o símbolo "ou", não um separador entre alternativas.
a) Mostre que esta gramática gera todas as expressões regulares sobre os símbolos *a* e *b*.
b) Mostre que essa gramática é ambígua.
c) Construa uma gramática equivalente inambígua que confira aos operadores concatenação e | as precedências e associatividades definidas na Seção 3.3.
d) Construa uma árvore gramatical em ambas as gramáticas para a sentença $a \mid b^*c$.

4.5 A seguinte gramática para os enunciados **if-then-else** é proposta para remediar a ambigüidade do *else*-vazio:

$$\begin{array}{rcl} cmd & \rightarrow & \mathbf{if}\ expr\ \mathbf{then}\ cmd \\ & \mid & cmd_associado \\ cmd_associado & \rightarrow & \mathbf{if}\ expr\ \mathbf{then}\ cmd_associado\ \mathbf{else}\ cmd \\ & \mid & \mathbf{outro} \end{array}$$

Mostre que esta gramática ainda é ambígua.

***4.6** Tente projetar uma gramática para cada uma das seguintes linguagens. Quais delas são regulares?
a) O conjunto de todas as cadeias de 0's e 1's tais que cada 0 é imediatamente seguido por pelo menos um 1.
b) Cadeias de 0's e 1's com um número igual de 0's e 1's.
c) Cadeias de 0's e 1's com um número desigual de 0's e 1's.
d) Cadeias de 0's e 1's nas quais 011 não figura como uma subcadeia.
e) Cadeias de 0's e 1's da forma *xy* onde $x \neq y$.
f) Cadeias de 0's e 1's da forma *xx*.

4.7 Construir uma gramática para as expressões de cada uma das seguintes linguagens:
a) Pascal
b) C
c) Fortran 77
d) Ada
e) Lisp

4.8 Construa gramáticas inambíguas para as sentenças de cada uma das linguagens do Exercício 4.7.

4.9 Podemos usar operadores semelhantes a expressões regulares nos lados direitos de produções gramaticais. Colchetes podem ser usados para denotar uma parte opcional de uma produção. Por exemplo, poderíamos escrever

$$cmd \rightarrow \mathbf{if}\ expr\ \mathbf{then}\ cmd\ [\ \mathbf{else}\ cmd\ ]$$

para denotar um comando *else* opcional. Em geral, $A \rightarrow \alpha\ [\ \beta\ ]\ \gamma$ é equivalente às duas produções $A \rightarrow \alpha\ \beta\ \gamma$ e $A \rightarrow \alpha\ \gamma$.
Chaves podem ser usadas para denotar uma frase que possa ser repetida zero ou mais vezes. Por exemplo,

$$cmd \rightarrow \mathbf{begin}\ cmd\ \{\ ;\ cmd\ \}\ \mathbf{end}$$

denota uma lista de comandos separados por ponto-e-vírgula e envolvidos por **begin** e **end.** Em geral, $A \rightarrow \alpha\ \{\ \beta\ \}\ \gamma$ é equivalente a $A \rightarrow \alpha\beta\gamma$ e $B \rightarrow \beta B \mid \epsilon$.
Num certo sentido, $[\ \beta\ ]$ figura no lugar da expressão regular $\beta \mid \epsilon$ e $\{\ \beta\ \}$ no de β^*. Podemos generalizar essas notações de forma a permitir quaisquer expressões regulares de símbolos gramaticais no lado direito de produções.
a) Modifique a produção-*cmd* acima de tal forma que uma lista de *cmd*'s terminada por ponto-e-vírgula apareça no lado direito.
b) Forneça um conjunto de produções livres de contexto que gere o mesmo conjunto de cadeias que $A \rightarrow \beta^* a(C \mid D)$.
c) Mostre como substituir qualquer produção $A \rightarrow r$, onde *r* é uma expressão regular, por uma coleção finita de produções livres de contexto.

4.10 A seguinte gramática gera declarações para um único identificador:

$$\begin{array}{lcl} cmd & \rightarrow & \mathbf{declare\ id}\ lista_de_opções \\ lista_de_opções & \rightarrow & lista_de_opções\ opção \mid \epsilon \\ opção & \rightarrow & modo \mid escala \mid precisão \mid base \\ modo & \rightarrow & \mathbf{real} \mid \mathbf{complex} \\ escala & \rightarrow & \mathbf{fixed} \mid \mathbf{floating} \\ precisão & \rightarrow & \mathbf{single} \mid \mathbf{double} \\ base & \rightarrow & \mathbf{binary} \mid \mathbf{decimal} \end{array}$$

a) Mostre como esta gramática pode ser generalizada de forma a permitir *n* opções A_i, $1 \leq i \leq n$, cada uma das quais podendo ser a_i ou b_i.

b) A gramática anterior permite declarações redundantes ou contraditórias, tais como

```
declare zap real fixed real floating
```

Poderíamos insistir em que a sintaxe da linguagem proíbe tais declarações. Somos deixados, então, com um número finito de seqüências de *tokens* que são sintaticamente corretos. Obviamente, essas declarações legais formam uma linguagem livre de contexto, de fato um conjunto regular. Escreva uma gramática para declarações com n opções, cada opção figurando no máximo uma vez.

**c) Mostre que a gramática para a parte (b) possui pelo menos 2^n símbolos.

d) O que (c) diz a respeito da viabilidade de se instaurar a não-redundância e a não-contradição entre as opções nas declarações através da definição sintática da linguagem?

4.11 a) Elimine a recursividade à esquerda da gramática do Exercício 4.1.

b) Construa um analisador sintático preditivo para a gramática em (a). Mostre o comportamento do analisador sintático nas sentenças do Exercício 4.1 (b).

4.12 Construa um analisador sintático de descendência recursiva com retrocesso para a gramática do Exercício 4.2. Você pode construir um analisador sintático preditivo para esta gramática?

4.13 A gramática

$$S \rightarrow aSa \mid aa$$

gera todas as cadeias de a's de comprimento par, exceto a cadeia vazia.

a) Construa um analisador sintático de descendência recursiva com retrocesso para esta gramática que tente a alternativa aSa antes de aa. Mostre que o procedimento para S tem sucesso para 2, 4, ou 8 a's, mas falha para 6 a's.

*b) Que linguagem o seu analisador sintático reconhece?

4.14 Construa um analisador sintático preditivo para a gramática do Exercício 4.3.

4.15 Construa um analisador sintático preditivo a partir da gramática inambígua para as expressões regulares no Exercício 4.4.

***4.16** Mostre que nenhuma gramática recursiva à esquerda pode ser LL(1).

***4.17** Mostre que nenhuma gramática LL (1) pode ser ambígua.

4.18 Mostre que uma gramática sem produções-ϵ na qual cada alternativa comece por um terminal distinto é sempre LL(1).

4.19 Um símbolo gramatical X é *inútil* se não existir derivação da forma $S \overset{*}{\Rightarrow} wXy \overset{*}{\Rightarrow} wxy$. Isto é, X não precisaria aparecer na derivação de qualquer sentença.

*a) Escreva um algoritmo para eliminar todas as produções contendo símbolos inúteis de uma gramática.

b) Aplique seu algoritmo à gramática

$$S \rightarrow 0 \mid A$$
$$A \rightarrow AB$$
$$B \rightarrow 1$$

4.20 Dizemos que uma gramática é ϵ-*livre* se ou não possuir produções-ϵ ou se existir exatamente uma produção-ϵ $S \rightarrow \epsilon$ e o símbolo de partida S não aparecer no lado direito de qualquer produção.

a) Escreva um algoritmo para converter uma dada gramática numa gramática ϵ-livre equivalente. *Sugestão.* Determine primeiro todos os não-terminais que podem gerar a cadeia vazia.

b) Aplique seu algoritmo à gramática do Exercício 4.2.

4.21 Uma produção *singela* é aquela com um único não-terminal do lado direito.

a) Escreva um algoritmo para converter uma gramática numa outra equivalente que não tenha produções singelas.

b) Aplique seu algoritmo à gramática do de expressões (4.10)

4.22 Uma gramática *livre-de-ciclos* não possui derivações da forma $A \overset{+}{\Rightarrow} A$ para qualquer não-terminal A.

a) Escreva um algoritmo para converter uma gramática numa outra equivalente que seja livre de ciclos.

b) Aplique seu algoritmo à gramática $S \rightarrow SS \mid (S) \mid \epsilon$.

4.23 a) Usando a gramática do Exercício 4.1, construa uma derivação mais à direita para $(a, (a, a))$ e mostre o *handle* de cada forma sentencial à direita.

b) Mostre os passos de um analisador sintático de empilhar e reduzir correspondentes à derivação mais à direita de (a).

c) Mostre, num processo de construção *bottom-up* de uma árvore gramatical, os passos realizados durante a análise sintática de empilhar e reduzir de (b).

4.24 A Fig. 4.60 mostra as relações de precedência de operadores para a gramática do Exercício 4.1. Usando-se essas relações de precedência, analise sintaticamente as sentenças do Exercício 4.1(b).

	a	(	)	,	$
a			$\cdot>$	$\cdot>$	$\cdot>$
(	$<\cdot$	$<\cdot$	$\doteq$	$<\cdot$	
)			$\cdot>$	$\cdot>$	$\cdot>$
,	$<\cdot$	$<\cdot$	$\cdot>$	$\cdot>$	
$	$<\cdot$	$<\cdot$			

Fig. 4.60 Relações de precedência para a gramática do Exercício 4.1.

4.25 Encontre as funções de precedência de operadores para a tabela da Fig. 4.60.

4.26 Existe uma forma mecânica de se produzir relações de precedência de operadores a partir de uma gramática de operadores, incluindo aquelas com muitos não-terminais diferentes. Defina-se *frente(A)*, para o não-terminal A, como sendo o conjunto de terminais a tal que a seja o terminal mais à esquerda em alguma cadeia derivada a partir de A e defina-se a *cauda(A)* como o conjunto de terminais que possam ser o símbolo mais à direita numa cadeia derivada a partir de A. Então, para os terminais a e b, dizemos que $a \doteq b$ se existir um lado direito da forma $\alpha a \beta b \gamma$ onde β ou é vazio ou é um único não-terminal e α e γ são arbitrários. Dizemos que $a \lessdot b$ se existir um lado direito da forma $\alpha a A \beta$ e b estiver em *frente(A)*; dizemos que $a \gtrdot b$ se existir um lado direito da forma $\alpha A b \beta$ e a estiver na *cauda(A)*. Em ambos os casos, α e β são cadeias arbitrárias. Igualmente, $\$ \lessdot b$ sempre que b estiver na *frente(S)*, onde S é o símbolo de partida e $a \gtrdot \$$ sempre que a estiver na *cauda(S)*.

a) Para a gramática do Exercício 4.1, compute *frente* e *cauda* para S e T.

b) Verifique que as relações de precedência da Fig. 4.60 são aquelas derivadas a partir desta gramática.

4.27 Gere relações de precedência de operadores para seguintes gramáticas.

a) A gramática do Exercício 4.2.

b) A gramática do Exercício 4.3.

c) A gramática de expressões (4.10).

4.28 Construir um analisador sintático de precedência de operadores para expressões regulares.

4.29 Uma gramática é denominada *gramática de precedência de operadores* (unicamente inversível) se for uma gramática de operadores na qual não hajam dois lados direitos que tenham o mesmo padrão de terminais e o método do Exercício 4.26 produza no máximo uma relação de precedência entre qualquer par de terminais. Quais das gramáticas do Exercício 4.27 são gramáticas de precedência de operadores?

4.30 Uma gramática é dita estar na *forma normal de Greibach* (GNF) se for livre de produções-ϵ e cada produção (exceto $S \rightarrow \epsilon$, se existir) é da forma $A \rightarrow a\alpha$, onde a é um terminal e α é uma cadeia de não-terminais, possivelmente vazia.

**a) Escreva um algoritmo para converter uma gramática na sua forma normal de Greibach equivalente.
b) Aplique seu algoritmo para converter a gramática de expressões (4.10).

4.31 Mostre que toda gramática pode ser convertida numa gramática de operadores equivalente. *Sugestão:* transforme primeiro a gramática para a forma normal de Greibach.

4.32 Mostre que toda gramática pode ser convertida para uma gramática de operadores na qual cada produção está numa das seguintes formas

$$A \rightarrow aBcC \qquad A \rightarrow aBb \qquad A \rightarrow aB \qquad A \rightarrow a$$

Se ϵ estiver na linguagem, então $S \rightarrow \epsilon$ também é uma produção.

4.33 Considere a gramática ambígua

$$S \rightarrow AS \mid b$$
$$A \rightarrow SA \mid a$$

a) Construa a coleção de conjuntos LR(0) para esta gramática.
b) Construa um AFN no qual cada estado seja um item LR(0) de (a). Mostre que o grafo de desvio da coleção canônica de itens LR(0) para esta gramática é o mesmo que o AFD construído a partir do AFN usando a construção de subconjuntos.
c) Construa a tabela sintática usando o algoritmo SLR 4.8.
d) Mostre todos os movimentos permitidos pela tabela a partir de (c) à entrada *abab*.
e) Construa a tabela sintática canônica.
f) Construa a tabela sintática canônica usando o algoritmo LALR 4.11.
g) Construa a tabela sintática usando o Algoritmo LALR 4.13.

4.34 Construa uma tabela sintática SLR para a gramática do Exercício 4.3.

4.35 Considere a seguinte gramática

$$E \rightarrow E + T \mid T$$
$$T \rightarrow TF \mid F$$
$$F \rightarrow F^* \mid a \mid b$$

a) Construir a tabela sintática SLR para esta gramática.
b) Construir a tabela sintática LALR.

4.36 Compacte as tabelas sintáticas construídas nos Exercícios 4.33, 4.34 e 4.35, de acordo com o método da Seção 4.7.

4.37 a) Mostre que a seguinte gramática

$$S \rightarrow AaAb \mid BbBa$$
$$A \rightarrow \epsilon$$
$$B \rightarrow \epsilon$$

é *LL(1)* mas não *SLR(1)*.
**b) Mostre que toda gramática *LL(1)* é *LR(1)*.

***4.38** Mostre que nenhuma gramática LR(1) pode ser ambígua.

4.39 Mostre que a seguinte gramática

$$S \rightarrow Aa \mid bAc \mid dc \mid bda$$
$$A \rightarrow d$$

é LALR(1) mas não é SLR(1).

4.40 Mostre que a seguinte gramática

$$S \rightarrow Aa \mid bAc \mid Bc \mid bBa$$
$$A \rightarrow d$$
$$B \rightarrow d$$

é LR(1) mas não é LALR(1).

***4.41** Considere a família de gramáticas G_n definida por

$$S \rightarrow A_i b_i \qquad 1 \le i \le n$$
$$A_i \rightarrow a_j A_i \mid a_j \qquad 1 \le i, j \le n \text{ e } j \ne i$$

a) Mostre que G_n possui $2n^2 - n$ produções e $2^n + n^2 + n$ conjuntos de itens LR(0). O que este resultado diz a respeito do tamanho a que um analisador sintático LR pode chegar, comparado com o tamanho da gramática?
b) G_n é SLR(1)?
c) G_n é LALR(1)?

4.42 Escreva um algoritmo para computar, para cada não-terminal A numa gramática, o conjunto de não-terminais B tais que $A \overset{*}{\Rightarrow} \beta\alpha$ para alguma cadeia de símbolos gramaticais α.

4.43 Escreva um algoritmo para computar para cada não-terminal A numa gramática o conjunto de terminais a tais que $A \overset{*}{\Rightarrow} aw$ para alguma cadeia de terminais w, onde o último passo da derivação não use uma produção-ϵ.

4.44 Construa uma tabela sintática SLR para a gramática do Exercício 4.4. Resolva os conflitos de ações sintáticas de tal forma que as expressões regulares sejam analisadas sintaticamente de forma natural.

4.45 Construir um analisador sintático SLR para a gramática (4.7), do *else*-vazio, tratando *expr* como um terminal. Resolva o conflito de ações sintáticas da forma usual.

4.46 a) Construa uma tabela sintática SLR para a gramática

$$E \rightarrow E \textbf{ sub } R \mid E \textbf{ sup } E \mid \{ E \} \mid c$$
$$R \rightarrow E \textbf{ sup } E \mid E$$

Resolva o conflito de ações sintáticas de tal forma que as expressões venham a ser estruturadas da mesma forma que pelo analisador sintático LR da Fig. 4.52.
b) Pode todo conflito do tipo reduzir/reduzir, gerado no processo de construção da tabela sintática LR, ser convertido num conflito do tipo empilhar/reduzir através da transformação da gramática?

***4.47** Construa uma gramática LR equivalente para a gramática de composição de tipos (4.25), que fatora expressões da forma E **sub** E **sup** E como um caso especial.

***4.48** Considere a seguinte gramática ambígua para n operadores binários infixos:

$$E \rightarrow E\,\theta_1\,E \mid E\,\theta_2\,E \mid \ldots \mid E\,\theta_n\,E \mid (\,E\,) \mid \textbf{id}$$

Assuma que todos os operadores sejam associativos à esquerda e que θ_i tenha precedência sobre θ_j se $i > j$.
a) Construa o conjunto de itens SLR para esta gramática. Quantos conjuntos de itens existem, como uma função de n?
b) Construa a tabela sintática SLR para esta gramática e compacte-a usando a representação de listas da Seção 4.7. Qual é o comprimento total de todas as listas usadas na representação, como função de n?
c) Quantos passos são consumidos para decompor **id** θ_i **id** θ_j?

***4.49** Repita o Exercício 4.48 para a gramática inambígua

$$E_1 \rightarrow E_1\,\theta_1\,E_2 \mid E_2$$
$$E_2 \rightarrow E_2\,\theta_2\,E_3 \mid E_3$$
$$\ldots$$
$$E_n \rightarrow E_n\,\theta_n\,E_{n+1} \mid E_{n+1}$$
$$E_{n+1} \rightarrow (\,E_1\,) \mid \textbf{id}$$

O que as suas respostas aos Exercícios 4.48 e 4.49 dizem a respeito da eficiência relativa dos analisadores sintáticos para gramáticas ambíguas e inambíguas? E sobre a eficiência relativa para construir os analisadores sintáticos?

4.50 Escreva um programa Yacc que irá receber expressões aritméticas como entrada e produzir como saídas as expressões posfixas correspondentes.

4.51 Escreva um programa de "calculadora de mesa" Yacc que irá avaliar expressões booleanas.

4.52 Escreva uma programa Yacc que irá tomar uma expressão regular como entrada e produzir sua árvore gramatical como saída.

4.53 Rastreie os movimentos que seriam feitos pelos analisadores sintáticos preditivo, de precedência de operadores e LR dos Exemplos 4.20, 4.32 e 4.50 para as seguintes entradas erradas:
a) (**id** + (* **id**)
b) * + **id**) + (**id** *

***4.54** Construir analisadores sintáticos de precedência de operadores e LR, com correção de erros, para a seguinte gramática:

$$
\begin{array}{lcl}
cmd & \rightarrow & \textbf{if } e \textbf{ then } cmd \\
 & | & \textbf{if } e \textbf{ then } cmd \textbf{ else } cmd \\
 & | & \textbf{while } e \textbf{ do } cmd \\
 & | & \textbf{begin } lista \textbf{ end} \\
 & | & s \\
lista & \rightarrow & lista \ ; \ cmd \\
 & | & cmd
\end{array}
$$

***4.55** A gramática do Exercício 4.54 pode ser tornada LL através da substituição das produções para *lista* por

$$
\begin{array}{lcl}
lista & \rightarrow & cmd\ lista' \\
lista' & \rightarrow & ;\ cmd \mid \epsilon
\end{array}
$$

Construa um analisador sintático preditivo de correção de erros para a gramática revisada.

4.56 Mostre o comportamento de seus analisadores sintáticos dos Exercícios 4.54 e 4.55 para as entradas incorretas
a) **if** *e* **then** *s* ; **if** *e* **then** *s* **end**
b) **while** *e* **do begin** *s* ; **if** *e* **then** *s* ; **end**

4.57 Escreva analisadores sintáticos LR, de precedência de operadores e preditivo, com recuperação de erros na modalidade do desespero, para as gramáticas dos Exercícios 4.54 e 4.55, usando o ponto-e-vírgula e **end** como *tokens* de sincronização. Mostre o comportamento de seus analisadores sintáticos para as entradas incorretas do Exercício 4.56.

4.58 Na Seção 4.6, propusemos um método orientado por grafos para determinar o conjunto de cadeias que poderiam ser desempilhadas num movimento de redução de um analisador sintático de precedência de operadores.
*a) Forneça um algoritmo para encontrar uma expressão regular que denote todas essas cadeias.
b) Forneça um algoritmo para determinar se o conjunto de tais cadeias é finito ou infinito, listando-as no primeiro caso.
c) Aplique seus algoritmos provenientes de (a) e (b) à gramática do Exercício 4.54.

****4.59** Fizemos a afirmativa para os analisadores sintáticos com correção de erros das Figs. 4.18, 4.28 e 4.53 de que qualquer correção de erro resultaria eventualmente em pelo menos um símbolo a mais sendo removido da entrada ou da pilha em processo de encurtamento, se o fim da entrada fosse atingido. As correções escolhidas, entretanto, não provocaram, todas, o consumo imediato de um símbolo de entrada. Pode V. provar que nenhum laço infinito é possível para os analisadores sintáticos das Figs. 4.18, 4.28 e 4.53? *Sugestão:* ajuda observar que, para um analisador sintático de precedência de operadores, os terminais consecutivos na pilha estão relacionados por $\leq \cdot$, ainda que tenham havido erros. Para o analisador sintático LR a pilha ainda conterá um prefixo viável, mesmo na presença de erros.

****4.60** Forneça um algoritmo para detectar entradas inatingíveis em tabelas sintáticas LR, preditivas e de precedência de operadores.

4.61 O analisador sintático LR da Fig. 4.53 trata as quatro situações, nas quais o estado de topo é 4 ou 5 (que ocorrem quando + e * estão no topo da pilha, respectivamente) e o próximo símbolo de entrada é + ou *, exatamente da mesma forma: chamando a rotina e1, que insere um **id** entre eles. Poderíamos facilmente divisar um analisador sintático LR para expressões que envolvessem o conjunto completo de operadores aritméticos, se comportando exatamente da mesma forma: inserir um **id** entre dois operadores adjacentes. Em algumas linguagens (como PL/1 ou C, mas não Fortran ou Pascal), seria inteligente tratar, de forma especial, o caso no qual / está ao topo da pilha e * é o próximo símbolo de entrada. Por quê? O que poderia ser um curso de ação razoável para o corretor de erros seguir?

4.62 Uma gramática é dita estar na *firma normal de Chomsky* (CNF) se for ϵ-livre e cada produção não-ϵ está na forma $A \rightarrow BC$ ou na forma $A \rightarrow a$.
*a) Forneça um algoritmo para converter uma gramática numa outra equivalente na forma normal de Chomsky.
b) Aplique seu algoritmo à gramática de expressões (4.10).

4.63 Dadas uma gramática G na forma normal de Chomsky e uma cadeia de entrada $w = a_1 a_2 \ldots a_n$, escreva um algoritmo para determinar se w está em $L(G)$. *Sugestão:* usando a programação dinâmica, preencha uma tabela $T\ n \times n$ na qual $T[i, j] = \{A \mid A \Rightarrow a_i a_{i+1} \ldots a_j\}$. A cadeia de entrada w está em $L(G)$ se e somente se S estiver em $T[1, n]$.

***4.64** a) Dada uma forma normal de Chomsky para uma gramática G mostre como adicionar produções para inserção, remoção e mutação de um único erro à gramática, de tal forma que a gramática expandida gere todas as possíveis cadeias de *tokens*.
b) Modifique o algoritmo de decomposição sintática do Exercício 4.63 de tal forma que, dada qualquer cadeia w, o mesmo encontre uma estruturação gramatical para w que use o menor número de produções de erro.

4.65 Escreva um analisador sintático Yacc para expressões aritméticas que usem o mecanismo de recuperação de erros do Exemplo 4.50.

NOTAS BIBLIOGRÁFICAS

O relatório altamente influente Algol 60 (Naur [1963]) usou a forma normal de Backus-Naur (BNF) para definir a sintaxe de uma linguagem maior de programação. A equivalência da BNF e das gramáticas livres de contexto foi rapidamente notada, e a teoria das linguagens formais recebeu uma grande parte das atenções nos anos 60. Hopcroft e Ullman [1979] cobrem as bases deste campo.

Os métodos de análise sintática se tornaram muito mais sistemáticos após o surgimento das gramáticas livres de contexto. Várias técnicas gerais de análise sintática de qualquer gramática livre de contexto foram inventadas. Uma das mais antigas é a da programação dinâmica sugerida no Exercício 4.63, que foi descoberta por J. Cocke, Younger [1967] e Kasami [1965]. Como sua tese de Ph. D, Earley [1970] também desenvolveu um algoritmo universal de análise sintática para todas as gramáticas livres de contexto. Aho e Ullman [1972b e 1973a] discutem esses e outros métodos de decomposição gramatical em detalhe.

Muitos diferentes métodos de análise gramatical têm sido empregados nos compiladores. Sheridan [1959] descreve o método de análise sintática usado no compilador Fortran original que introduziu parênteses adicionais em volta dos operandos de forma a torná-lo capaz de analisar expressões. A idéia da precedência dos operadores e o uso das funções de precedência é de Floyd [1963]. Nos anos 60, um grande número de estratégias de análise sintática *bottom-up* foi proposto. Incluíam a precedência simples (Wirth e Weber [1966]), contexto limitado (Floyd [1964], Graham [1964]), estratégia mista de precedência (McKeeman, Horning e Wortman [1970]) e precedência fraca (Ichbiah e Morse [1970]).

A análise sintática de descendência recursiva e a preditiva são amplamente usadas na prática. Em decorrência da flexibilidade, a análise sintática de descendência recursiva foi usada em muitos sistemas de escrita de compiladores primordiais, tais como META (Schorre [1964]) e TMG (McClure [1965]). Uma solução para o Exercício 4.13 pode ser encontrada em Birman e Ullman [1973], juntamente com alguma teoria deste método de análise sintática. Pratt [1973] propõe um método *top-down* de análise sintática de precedência de operadores.

As gramáticas LL foram estudadas por Lewis e Stearns [1968] e suas propriedades foram desenvolvidas em Rosenkrantz e Stearns [1970]. Analisadores sintáticos preditivos foram estudados extensivamente por Knuth [1971a], Lewis, Rosenkrantz e Stearns [1976] des-

crevem o uso dos analisadores sintáticos preditivos nos compiladores. Algoritmos para transformar as gramáticas para a forma LL(1) são apresentados em Foster [1968], Wood [1969], Stearns [1971] e Soisalon-Soininen e Ukkonen [1979].

As gramáticas e os analisadores sintáticos LR foram primeiramente introduzidos por Knuth [1965], que descreveu a construção de tabelas sintáticas LR canônicas. O método LR não foi considerado prático até que Korenjak [1969] mostrou que, com o mesmo, analisadores sintáticos razoavelmente dimensionados poderiam ser produzidos para gramáticas de linguagens de programação. Quando DeRemer [1969, 1971] divisou os métodos SLR e LALR, que são mais simples que o de Korenjak, a técnica LR se tornou o método de escolha para os geradores automáticos de analisadores sintáticos. Hoje em dia, geradores de analisadores sintáticos LR são comuns nos ambientes de construção de compiladores.

Uma grande parte das pesquisas foi para engenharia dos analisadores sintáticos LR. O uso de gramáticas ambíguas na análise sintática LR é devido a Aho, Johnson e Ullman [1975] e Earley [1975a]. A eliminação de reduções através de produções singelas foi discutida em Anderson, Eve e Horning [1973], Aho e Ullman [1973b], Demers [1975], Backhouse [1976], Joliat [1976], Pager [1977b], Soisalon-Soininen [1980] e Tokuda [1981].

As técnicas para computar conjuntos de *lookaheads* LALR(1) foram propostas por LaLonde [1971], Anderson, Eve e Horning [1973], Pager [1977a], Kristensen e Madsen [1981], DeRemer e Pennello [1982] e Park, Choe e Chang [1985] que também providenciaram algumas comparações experimentais.

Aho e Johnson [1974] fornecem uma pesquisa geral da análise sintática LR e discutem alguns dos algoritmos subjacentes ao gerador de analisadores sintáticos Yacc, incluindo o uso de produções de erro para a recuperação de erros. Aho e Ullman [1972b e 1973a] fornecem um extensivo tratamento da análise sintática LR e de seus alicerces teóricos.

Muitas das técnicas de recuperação de erros para analisadores sintáticos foram propostas. As técnicas de recuperação de erros são pesquisadas por Ciesinger [1979] e por Sippu [1981]. Irons [1963] propôs um enfoque baseado na gramática para a recuperação de erros sintáticos. As produções de erro foram empregadas por Wirth [1968] para o tratamento de erros num compilador PL360. Leinius [1970] propôs a estratégia de recuperação em nível de frase. Aho e Peterson [1972] mostram como uma recuperação de erros de menor custo global pode ser atingida utilizando-se produções de erro em conjunto com os algoritmos para gramáticas livres de contexto. Mauney e Fischer [1982] estendem essas idéias para a reparação de menor custo local em analisadores sintáticos LL e LR, usando a técnica de análise sintática de Graham, Harrison e Ruzzo [1980]. Graham e Rhodes [1975] discutem a recuperação de erros no contexto da análise sintática de precedência de operadores.

Horning [1976] discute as qualidades que as mensagens de erro deveriam ter. Sippu e Soisalon-Soininen [1983] comparam o desempenho da técnica de recuperação de erros no *Helsinki Language Processor* (Processador de Linguagem Helsinki) (Räihä et al. [1983]) com a técnica de recuperação do "movimento para a frente" de Pennello e De Remer [1978], a técnica de recuperação de erros de Graham, Haley e Joy [1979] e a técnica de recuperação de erros de "contexto global" de Pai e Kieburtz [1980].

A correção de erros durante a análise sintática é discutida por Conway e Maxwell [1963], Moulton e Muller [1967], Conway e Wilcox [1973], Levy [1975], Tai [1978] e Röhrich [1980]. Aho e Peterson [1972] contém uma solução para o Exercício 4.63.

CAPÍTULO 5

TRADUÇÃO DIRIGIDA PELA SINTAXE

Este capítulo desenvolve o tema da Seção 2.3, a tradução de linguagens guiada por gramáticas livres de contexto. Associamos informações a uma construção de linguagem de programação atrelando os atributos aos símbolos gramaticais que representam a construção. Os valores para os atributos são computados através de "regras semânticas" associadas às produções da gramática.

Existem duas notações para associar regras semânticas às produções, definições dirigidas pela sintaxe e esquemas de tradução. As definições dirigidas pela sintaxe são especificações de alto nível para as traduções. Escondem muitos detalhes de implementação e liberam o usuário de especificar exatamente a ordem na qual as traduções têm lugar. Os esquemas de tradução indicam a ordem na qual as regras semânticas são avaliadas e assim permitem que alguns detalhes de implementação sejam evidenciados. Usamos ambas as notações no Capítulo 6 para especificar a verificação semântica, particularmente na determinação de tipos, e no Capítulo 8, para gerar o código intermediário.

Conceitualmente, com os dois esquemas, de definições dirigidas pela sintaxe e de tradução, analisamos sintaticamente o fluxo de *tokens* de entrada, construímos a árvore gramatical e, em seguida, a percorremos da forma necessária, avaliando as regras semânticas a cada nó (veja a Fig. 5.1). A avaliação das regras semânticas pode gerar código, salvar informações numa tabela de símbolos, emitir mensagens de erro ou realizar quaisquer outras atividades. A tradução de um fluxo de *tokens* é o resultado obtido através da avaliação das regras semânticas.

Uma implementação não tem que seguir literalmente o delineado na Fig. 5.1. Casos especiais de definições dirigidas pela sintaxe podem ser implementados numa única passagem através da avaliação das regras semânticas durante a análise sintática, sem explicitamente construir uma árvore gramatical ou um grafo que exiba as dependências entre os atributos. Uma vez que a implementação em uma passagem é importante para a eficiência em tempo de compilação, muito deste capítulo é devotado ao estudo dos casos especiais. Uma subclasse importante, chamada de definições "L-atribuídas", abrange virtualmente todas as traduções que podem ser realizadas sem a construção explícita de uma árvore gramatical.

5.1 DEFINIÇÕES DIRIGIDAS PELA SINTAXE

Uma definição dirigida pela sintaxe é uma generalização de uma gramática livre de contexto na qual cada símbolo gramatical possui um conjunto associado de atributos, particionados em dois subconjuntos, chamados de atributos sintetizados e atributos herdados daquele símbolo gramatical. Se pensarmos em um nó para um símbolo gramatical numa árvore gramatical como sendo um registro com campos para manter informações, então um atributo corresponde a um nome de campo.

Um atributo pode representar qualquer coisa que escolhermos: uma cadeia, um número, um tipo, uma localização de memória etc. O valor para um atributo em um nó da árvore gramatical é definido por uma regra semântica associada à produção usada naquele nó. O valor de um atributo sintetizado em um nó é computado a partir dos valores dos atributos dos filhos daquele nó na árvore gramatical. O valor de um atributo herdado é computado a partir dos valores dos atributos dos irmãos e pai daquele nó.

As regras semânticas estabelecem dependências entre os atributos, as quais serão representadas por um grafo. A partir do grafo de dependências, derivamos uma ordem de avaliação para as regras semânticas. A avaliação das regras semânticas define os valores dos atributos nos nós da árvore gramatical para uma dada cadeia de entrada. Uma regra semântica também pode ter efeitos colaterais, como, por exemplo, imprimir um valor ou atualizar uma variável global. Naturalmente, uma implementação não precisa construir explicitamente uma árvore gramatical ou um grafo de dependências; tem apenas que produzir a mesma saída para cada cadeia de entrada.

Uma árvore gramatical mostrando os valores dos atributos a cada nó é denominada de uma árvore gramatical anotada. O processo de computar os valores dos atributos a cada nó é chamado de *anotação* ou *decoração* da árvore.*

Forma das Definições Dirigidas pela Sintaxe

Numa definição dirigida pela sintaxe, cada produção gramatical $A \rightarrow \alpha$ tem associada a si um conjunto de regras semânticas da forma $b := f(c_1,$

*Daremos preferência ao primeiro termo. (N. do T.)

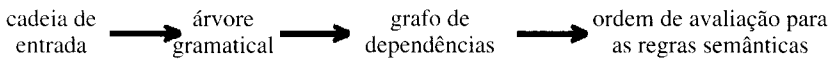


Fig. 5.1. Visão conceitual da tradução dirigida pela sintaxe.

$c_2, ..., c_k$), onde f é uma função e vigora uma das duas situações seguintes, mas não ambas:

1. b é um atributo sintetizado de A e $c_1, c_2, ..., c_k$ são atributos pertencentes aos símbolos gramaticais da produção ou
2. b é um atributo herdado, pertencente a um dos símbolos gramaticais do lado direito da produção, e $c_1, c_2, ..., c_k$ são atributos pertencentes aos símbolos gramaticais da produção.

Num ou noutro caso, dizemos que o atributo b *depende* dos atributos $c_1, c_2, ..., c_k$. Uma *gramática de atributos* é uma definição dirigida pela sintaxe, na qual as funções nas regras semânticas não têm efeitos colaterais.*

As funções nas regras semânticas serão freqüentemente escritas como expressões. Ocasionalmente, o único propósito de uma regra semântica numa definição dirigida pela sintaxe será o de criar um efeito colateral. Tais regras semânticas são escritas como chamadas de procedimentos ou fragmentos de programas. Podem ser pensadas como regras que definem os valores de atributos sintetizados fictícios do não-terminal ao lado esquerdo da produção associada; o atributo fictício e o sinal := não são mostrados na regra semântica.

Exemplo 5.1. A definição dirigida pela sintaxe na Fig. 5.2 é para um programa de calculadora de mesa. Esta definição associa um atributo sintetizado, chamado *val*, com valor do tipo inteiro, a cada um dos não-terminais E, T e F. Para cada produção-E, T e F, a regra semântica computa o atributo *val* para o não-terminal do lado esquerdo a partir dos valores de *val* para os não-terminais ao lado direito.

O *token* **dígito** possui um atributo sintetizado *lexval*, cujo valor assume-se que será fornecido pelo analisador léxico. A regra associada à produção $L \rightarrow E$ **n** para o não-terminal de partida L é justamente um procedimento que imprime o valor da expressão aritmética gerada por E; podemos pensar nesta regra como definindo um atributo fictício para o não-terminal L. Uma especificação Yacc para esta calculadora de mesa foi apresentada na Fig. 4.56, a fim de ilustrar a tradução durante a análise sintática LR. □

Numa definição dirigida pela sintaxe, assume-se que os terminais tenham somente atributos sintetizados, na medida em que a definição não providencie quaisquer regras semânticas para os mesmos. Os valores para os atributos dos terminais são usualmente fornecidos pelo analisador léxico, como discutido na Seção 3.1. Sobretudo, é assumido para o símbolo de partida que o mesmo não tenha quaisquer atributos herdados, a menos que seja estabelecido o contrário.

Atributos Sintetizados

Os atributos sintetizados são usados extensivamente na prática. Uma definição dirigida pela sintaxe que use exclusivamente atributos sintetizados é dita uma *definição S-atribuída*. Uma árvore gramatical para uma definição S-atribuída pode ser sempre anotada através da avaliação das regras semânticas para os atributos a cada nó, de baixo para cima, das folhas para a raiz. A Seção 5.3 descreve como um gerador de analisadores sintáticos LR pode ser adaptado para implementar mecanicamente uma definição S-atribuída baseada numa gramática LR.

Exemplo 5.2. A definição S-atribuída no Exemplo 5.1 especifica uma calculadora de mesa que lê uma linha de entrada, contendo uma expressão aritmética, envolvendo dígitos, parênteses, operadores + e * e um caractere de avanço de linha **n** ao fim, e imprime o valor da expressão. Por exemplo, dada a expressão 3*5+4 seguida por um avanço de linha, o programa imprime o valor 19. A Fig. 5.3 contém uma árvore gramatical anotada para a entrada 3*5+4**n**. A saída impressa à raiz da árvore é o valor de *E.val*, que está ao primeiro filho da raiz da árvore.

PRODUÇÃO	REGRAS SEMÂNTICAS
$L \rightarrow E$ **n**	*Imprimir* (*E. val*)
$E \rightarrow E_1 + T$	$E.val := E_1.val + T.val$
$E \rightarrow T$	$E.val := T.val$
$T \rightarrow T_1 * F$	$T.val := T_1.val \times F.val$
$T \rightarrow F$	$T.val := F.val$
$F \rightarrow (E)$	$F.val := E.val$
$F \rightarrow$ **dígito**	$F.val :=$ **dígito.***lexval*

Fig. 5.2. Definição dirigida pela sintaxe de uma calculadora de mesa simples.

Para vermos como os valores dos atributos são computados, consideremos primeiro o nó interior mais à esquerda e mais ao fundo, que corresponde ao uso da produção $F \rightarrow$ **dígito.** A regra semântica correspondente, $F.val :=$ **dígito.***lexval*, define o atributo $F.val$ àquele nó como tendo o valor 3, porque o valor de **dígito.***lexval* ao filho daquele nó é 3. Do mesmo modo, o atributo $T.val$ possui o valor 3, ao pai desse nó-F.

Consideremos agora o nó para a produção $T \rightarrow T * F$. O valor do atributo $T.val$ a esse nó é definido por

PRODUÇÃO	REGRA SEMÂNTICA
$T \rightarrow T_1 * F$	$T.val := T_1.val \times F.val$

Ao aplicarmos a regra semântica a esse nó, $T_1.val$ possui o valor 3, proveniente do filho à esquerda, e $F.val$ o valor 5, do filho à direita. Por conseguinte, $T.val$ adquire o valor 15 a esse nó.

A regra associada à produção para o não-terminal de partida $L \rightarrow E$ **n** imprime o valor da expressão gerada por E. □

Atributos Herdados

Um atributo herdado é aquele cujo valor a um nó de uma árvore gramatical é definido em termos do pai e/ou irmãos daquele nó. Os atributos herdados são convenientes para expressar a dependência de uma construção de linguagem de programação no contexto em que a mesma figurar. Por exemplo, podemos usar um atributo herdado para controlar se um identificador aparece ao lado esquerdo ou direito de um comando de atribuição a fim de decidirmos se é necessário o endereço ou o valor de um identificador. Apesar de ser sempre possível se reescrever uma definição dirigida pela sintaxe de forma a se usar somente atributos sintetizados, estas definições definidas pela sintaxe com atributos herdados são freqüentemente mais naturais.

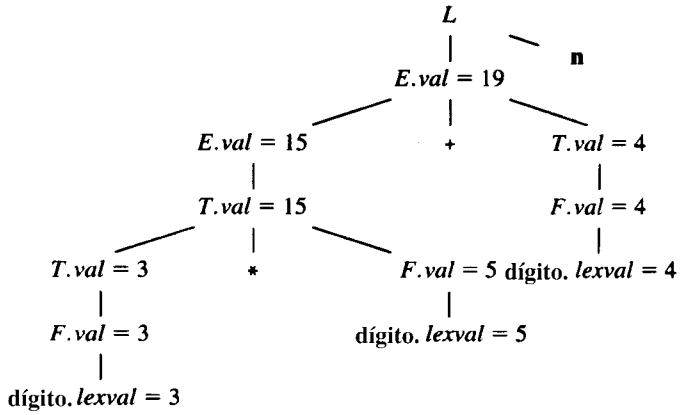


Fig. 5.3. Árvore gramatical anotada para 3*5+4**n**.

*Uma função, procedimento ou operação é dita produzir efeitos colaterais (*side effects*) quando altera um ou mais de seus parâmetros ou modifica uma variável não local. Ver Pratt [1984], na bibliografia, para uma discussão dos efeitos colaterais. (N. do T.)

PRODUÇÃO	REGRAS SEMÂNTICAS
$D \rightarrow T\,L$	$L.in := T.tipo$
$T \rightarrow$ **int**	$T.\ tipo := inteiro$
$T \rightarrow$ **real**	$T.\ tipo := real$
$L \rightarrow L_1$, **id**	$L_1.in := L.\ in$
	incluir_tipo (**id.** *entrada, L.in*)
$L \rightarrow$ **id**	*incluir_tipo* (**id.** *entrada, L.in*)

Fig. 5.4. Definição dirigida pela sintaxe tendo *L.in* como atributo herdado.

No exemplo seguinte, um atributo herdado distribui informações de tipo para os vários identificadores numa declaração.

Exemplo 5.3. Uma declaração gerada pelo não-terminal *D* numa definição dirigida pela sintaxe na Fig. 5.4 consiste na palavra-chave **int** ou **real,** seguida por uma lista de identificadores. O não-terminal *T* possui um atributo sintetizado *tipo,* cujo valor é determinado pela palavra-chave na declaração. A regra semântica *L.in* := *T.tipo,* associada à produção $D \rightarrow T\,L$, faz o atributo herdado *L.in* igual ao tipo na declaração. As regras então propagam esse tipo pela árvore gramatical abaixo, usando o atributo herdado *L.in.* As regras associadas às produções para *L* chamam o procedimento *incluir_tipo* para incluir o tipo de cada identificador na sua entrada respectiva na tabela de símbolos (apontada pelo atributo *entrada*).

A Fig. 5.5 mostra uma árvore gramatical anotada para a sentença **real id**$_1$, **id**$_2$, **id**$_3$. Os valores de *L.in* nos três nós fornecem o tipo dos identificadores **id**$_1$, **id**$_2$ e **id**$_3$. Esses valores são determinados pelo cômputo do valor do atributo *T.tipo* no filho à esquerda da raiz e, em seguida, pela avaliação de *L.in* de cima para baixo nos três nós da subárvore direita da raiz. A cada nó-*L*, podemos também chamar o procedimento *incluir_tipo* para inserir na tabela de símbolos o fato de cada identificador em cada filho à direita desse nó possuir o tipo real. □

Grafos de Dependências

Se um atributo *b* a um nó da árvore gramatical depender de um atributo *c*, a regra semântica para *b* àquele nó precisa ser avaliada após a regra semântica que define *c*. As interdependências entre os atributos herdados e sintetizados nos nós da árvore gramatical podem ser delineadas através de um grafo chamado de *grafo de dependências.*

Antes de construir um grafo de dependências para uma árvore gramatical, colocamos cada regra semântica sob a forma $b := f(c_1, c_2, ..., c_k)$, através da introdução de um atributo sintetizado fictício *b* para cada regra semântica que consista em uma chamada de procedimento. O grafo possui um nó para cada atributo e um lado em direção ao nó para *b*, a partir do nó para *c*, se o atributo *b* depender do atributo *c*. Mais detalhadamente, o grafo de dependências para uma dada árvore gramatical é construído como segue.

para cada nó *n* na árvore gramatical **faça**
  para cada atributo *a* do símbolo gramatical em *n* **faça**
    construir um nó no grafo de dependências para *a;*
para cada nó *n* na árvore gramatical **faça**
  para cada regra semântica $b := f(c_1, c_2, ..., c_k)$
    associada à produção usada em *n* **faça**
    para *i* := 1 **até** *k* **faça**
      construir um lado a partir do nó c_i até o nó *b*;

Por exemplo, suponhamos que $A.a := f(X.x, Y.y)$ seja uma regra semântica para a produção $A \rightarrow XY$. Esta regra define o atributo sintetizado *A.a* que depende dos atributos *X.x* e *Y.y*. Se esta produção vier a ser usada na árvore gramatical, existirão, no grafo de dependências, três nós, *A.a, X.x* e *Y.y,* com um lado para *A.a,* a partir de *X.x,* uma vez que *A.a* depende de *X.x,* e um lado para *A.a,* a partir de *Y.y,* uma vez que *A.a* também depende de *Y.y.*

Se a produção $A \rightarrow XY$ possuir a regra semântica $X.i := g(A.a, Y.y)$ associada a si, existirá um lado para *X.i* proveniente de *A.a* e também uma lado para *X.i* proveniente de *Y.y,* uma vez que *X.i* depende de ambos, *A.a* e *Y.y.*

Exemplo 5.4. Sempre que a seguinte produção for usada numa árvore gramatical, adicionaremos os lados mostrados na Fig. 5.6 ao grafo de dependências.

PRODUÇÃO	REGRA SEMÂNTICA
$E \rightarrow E_1 + E_2$	$E.val := E_1.val + E_2.val$

Os três nós do grafo de dependências, marcados por ●, representam os atributos sintetizados *E.val,* $E_1.val$ e $E_2.val$ nos nós correspondentes da árvore gramatical. O lado para *E.val* a partir de $E_2.val$ mostra que *E.val* também depende de $E_2.val$. As linhas pontilhadas representam a árvore gramatical e não fazem parte do grafo de dependências. □

Exemplo 5.5. A Fig. 5.7 mostra o grafo de dependências para a árvore gramatical na Fig. 5.5. Os nós nos grafos de dependências são marcados por números; esses números serão usados abaixo. Existe um lado em direção ao nó 5, associado a *L.in,* a partir do nó 4, associado a *T.tipo,* porque o atributo herdado *L.in* depende do atributo *T.tipo,* de acordo com a regra semântica *L.in* := *T.tipo* para a produção $D \rightarrow TL$. Os dois lados para baixo em direção aos nós 7 e 9 emergem porque $L_1.in$ depende de *L.in,* de acordo com a regra semântica $L_1.in := L.in$ para a produção $L \rightarrow L_1$, **id.** Cada uma das regras semânticas *incluir_tipo* (**id.***entrada, L.in*) associadas às produções-*L* leva à criação de um atributo fictício. Os nós 6, 8 e 10 são construídos para esses atributos fictícios. □

Ordem de Avaliação

Uma *classificação topológica* de um grafo acíclico dirigido é qualquer ordenamento $m_1, m_2, ..., m_k$ dos nós do grafo, de tal forma que os lados vão dos primeiros nós do ordenamento para os últimos; isto é, se $m_i \rightarrow m_j$ é um lado de m_i para m_j, então m_i aparece antes de m_j no ordenamento.

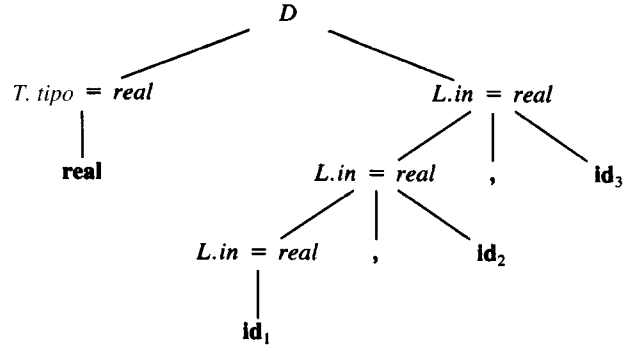


Fig. 5.5. Árvore gramatical com atributo herdado *in* a cada nó rotulado *L*.

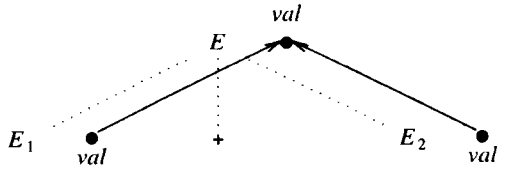


Fig. 5.6. *E.val* é sintetizado a partir de $E_1.val$ e $E_2.val$.

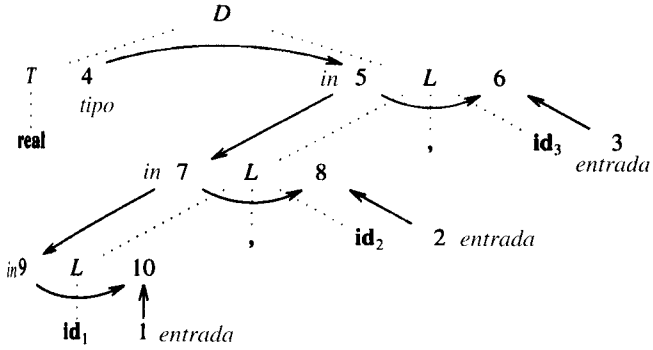


Fig. 5.7. Grafo de dependências para a árvore gramatical da Fig. 5.5.

Qualquer classificação topológica de um grafo de dependências fornece uma ordem válida na qual as regras semânticas associadas aos nós na árvore gramatical podem ser avaliadas. Ou seja, na classificação topológica, os atributos $c_1, c_2, \ldots c_k$, dos quais uma regra semântica $b := f(c_1, c_2, \ldots, c_k)$ depende, estão disponíveis num nó antes de f ser avaliada.

A tradução especificada por uma definição dirigida pela sintaxe pode ser tornada precisa como segue. A gramática subjacente é usada para construir uma árvore gramatical para a entrada. O grafo de dependências é construído como discutido acima. A partir de uma classificação topológica do grafo de dependências, obtemos uma ordem de avaliação para as regras semânticas. A avaliação das regras semânticas nessa ordem produz a tradução da cadeia de entrada.

Exemplo 5.6. Cada um dos lados no grafo de dependências da Fig. 5.7 vai de um nó de numeração menor para um de numeração maior. Por conseguinte, uma classificação topológica do grafo de dependências é obtida escrevendo-se os nós pela ordem de seus números. A partir desta classificação topológica, obtemos o programa seguinte. Escrevemos a_n para significar o atributo associado ao nó numerado n no grafo de dependências.

a_4 := *real;*
$a_5 := a_4$;
incluir_tipo (**id**$_3$.*entrada*, a_5);
$a_7 := a_5$;
incluir_tipo (**id**$_2$.*entrada*, a_7);
$a_9 := a_7$;
incluir_tipo (**id**$_1$.*entrada*, a_9);

A avaliação dessas regras semânticas armazena o tipo *real* na entrada da tabela de símbolos para cada identificador. □

Vários métodos foram propostos para a avaliação das regras semânticas:

1. *Métodos das árvores gramaticais.* Em tempo de compilação, esses métodos obtêm uma ordem de avaliação a partir da classificação topológica do grafo de dependências construído a partir da árvore gramatical para cada entrada. Esses métodos irão falhar em encontrar uma ordem de avaliação somente se o grafo de dependências para uma árvore gramatical particular contiver um ciclo.
2. *Métodos baseados em regras.* Em tempo de construção do compilador, as regras semânticas associadas às produções são analisadas manualmente ou por uma ferramenta especializada. Para cada produção, a ordem na qual os atributos associados àquela produção são avaliados é predeterminada em tempo de construção do compilador.
3. *Métodos alienados.* Uma ordem de avaliação é escolhida, sem levar em consideração as regras semânticas. Por exemplo, se a tradução tem lugar durante a análise sintática, a ordem de avaliação é forçada pelo método de decomposição gramatical, independentemente das regras semânticas. Uma avaliação alienada restringe a classe de definições dirigidas pela sintaxe que podem ser implementadas.

Os métodos baseados em regras e os alienados não precisam construir explicitamente uma árvore gramatical em tempo de compilação e podem, então, ser mais eficientes no uso do tempo e do espaço durante a compilação.

Uma definição dirigida pela sintaxe é dita *circular* se o grafo de dependências para alguma árvore gramatical gerada pela gramática contiver um ciclo. A Seção 5.10 discute como testar a circularidade de uma definição dirigida pela sintaxe.

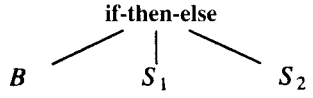
5.2 CONSTRUÇÃO DE ÁRVORES SINTÁTICAS

Nesta seção, mostramos como as definições dirigidas pela sintaxe podem ser usadas para especificar a construção de árvores sintáticas e outras representações gráficas das construções de linguagem.

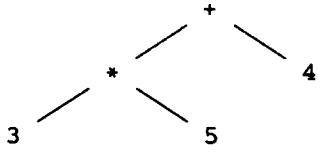
O uso de árvores sintáticas como uma forma de representação intermediária permite que a tradução seja desacoplada da análise sintática. As rotinas de tradução que são invocadas durante a análise sintática precisam conviver com dois tipos de restrições. Primeiro, uma gramática que seja adequada à análise sintática pode não refletir a estrutura hierárquica natural das construções da linguagem. Por exemplo, uma gramática para Fortran pode enxergar uma sub-rotina como consistindo somente numa lista de enunciados. No entanto, a análise da sub-rotina pode ser mais fácil se usarmos uma representação em árvore que reflita o aninhamento dos laços DO. Segundo, o método de análise sintática restringe a ordem na qual os nós na árvore gramatical são considerados. Essa ordem pode não coincidir com a ordem na qual as informações sobre uma construção se tornam disponíveis. Por esta razão, os compiladores para C usualmente constroem árvores sintáticas para as declarações.

Árvores Sintáticas

Uma árvore sintática (abstrata) é uma forma condensada de árvore gramatical, útil para a representação das construções de linguagem. A produção $S \rightarrow$ **if** B **then** S_1 **else** S_2 poderia aparecer numa árvore sintática como



Numa árvore sintática, os operadores e palavras-chave não figuram como folhas, mas, em lugar, são associados ao nó interior que seria o pai daquelas folhas na árvore gramatical. Outra simplificação encontrada nas árvores sintáticas é que as cadeias de produções singelas podem ser eliminadas; a árvore gramatical da Fig. 5.3 se torna árvore sintática



A tradução dirigida pela sintaxe pode ser usada em árvores sintáticas ou em árvores gramaticais. O enfoque é o mesmo em cada caso; atrelamos os atributos aos nós como na árvore gramatical.

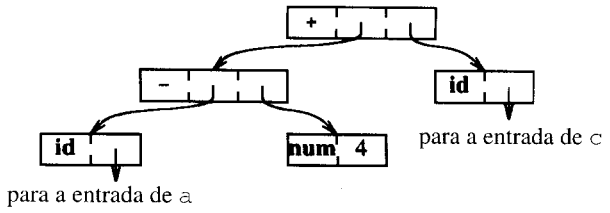


Fig. 5.8. Árvore sintática para a-4+c.

Construindo Árvores Sintáticas para Expressões

A construção de uma árvore sintática para uma expressão é similar à tradução de uma expressão para a forma posfixa. Construímos subárvores para subexpressões através da criação de um nó para cada operador e operando. Os filhos de um nó operador são as raízes dos nós que representam as subexpressões que constituem os operandos daquele operador.

Cada nó numa árvore sintática pode ser implementado como um registro com vários campos. No nó para um operador, um campo identifica o operador e os campos restantes contêm apontadores para os nós dos operandos. O operador é freqüentemente chamado de *rótulo* do nó. Quando usado numa tradução, os nós numa árvore sintática podem ter campos adicionais para guardar os valores (ou apontadores para os valores) dos atributos atrelados ao nó. Nesta seção, usaremos as seguintes funções para criar os nós das árvores sintáticas para as expressões com operadores binários. Cada função retorna um apontador para o nó recém-criado.

1. *criar_nó* (*op*, *esquerdo*, *direito*) cria um nó de operador com rótulo *op* e dois campos contendo apontadores para *esquerdo* e *direito*.
2. *criar_folha* (**id**, *entrada*) cria um nó de identificador, com rótulo **id** e um campo contendo *entrada*, um apontador para a entrada do identificador na tabela de símbolos.
3. *criar_folha* (**num**, *val*) cria um nó número, com rótulo **num** e um campo contendo *val*, o valor do número.

Exemplo 5.7. A seguinte seqüência de chamadas de funções cria a árvore sintática para a expressão a - 4 + c na Fig. 5.8. Nesta seqüência, $p_1, p_2, ..., p_5$ são apontadores para nós e *entrada-a* e *entrada-c* são apontadores para as entradas na tabela de símbolos dos identificadores a e c, respectivamente.

(1) p_1 := *criar_folha* (**id**, *entrada-a*);
(2) p_2 := *criar_folha* (**num**, 4);
(3) p_3 := *criar_nó* ('-', p_1, p_2);
(4) p_4 := *criar_folha* (**id**, *entrada_c*);
(5) p_5 := *criar_nó* ('+', p_3, p_4);

PRODUÇÃO	REGRAS SEMÂNTICAS
$E \rightarrow E_1 + T$	*E.nptr* := *criar-nó* ('+', E_1*.nptr*, *T.nptr*)
$E \rightarrow E_1 - T$	*E.nptr* := *criar-nó* ('-', E_1*.nptr*, *T.nptr*)
$E \rightarrow T$	*E.nprt* := *T.nptr*
$T \rightarrow (E)$	*T.nptr* := *E.nptr*
$T \rightarrow$ **id**	*T.nptr* := *criar_folha* (**id, id.***entrada*)
$T \rightarrow$ **num**	*T.nptr* := *criar_folha* (**num, num.***val*)

Fig. 5.9. Definição dirigida pela sintaxe para a construção da árvore sintática de uma expressão.

A árvore é construída de baixo para cima. As chamadas de função *criar_folha* (**id**, *entrada_a*) e *criar_folha* (**num**, 4) constroem as folhas para a e 4; os apontadores para esses nós são salvos usando p_1 e p_2. Em seguida, a chamada *criar_nó* ('-', p_1, p_2) constrói o nó interior tendo as folhas para a e 4 como filhas. Dois passos depois, p_5 está apontando para a raiz. □

Uma Definição Dirigida pela Sintaxe para Construir Árvores Sintáticas

A Fig. 5.9 contém uma definição S-atribuída para construir uma árvore sintática para uma expressão contendo os operadores + e −. Utiliza as produções subjacentes da gramática a fim de agendar as chamadas para as funções *criar_nó* e *criar_folha* de modo a construir a árvore. O atributo sintetizado *nptr* controla os apontadores retornados pelas chamadas de função.

Exemplo 5.8. Uma árvore gramatical anotada ilustrando a construção de uma árvore sintática para a expressão a - 4 + c é mostrada na Fig. 5.10. A árvore gramatical é exibida em linhas pontilhadas. Os nós da árvore gramatical rotulados pelos não-terminais *E* e *T* usam o atributo sintetizado *nptr* com a finalidade de guardar o apontador para o nó da árvore sintática da expressão representada pelo não-terminal.

As regras semânticas associadas às produções $T \rightarrow$ **id** e $T \rightarrow$ **num** definem o atributo *T.nptr* como sendo um apontador de uma nova folha para um identificador e um número, respectivamente. Os atributos **id**.*entrada* e **num**.*val* são os valores léxicos que se espera que sejam retornados pelo analisador léxico junto com os *tokens* **id** e **num**.

Na Fig. 5.10, quando a expressão *E* for um único termo, correspondendo a um uso da produção $E \rightarrow T$, o atributo *E.nptr* fica com o valor de *T.nptr*. Quando a regra semântica *E.nptr* := *criar_folha* ('-', E_1*.nptr*, *T.nptr*), associada à produção $E \rightarrow E_1 - T$ for invocada, as regras anteriores já fizeram E_1*.nptr* e *T.nptr* serem apontadores das folhas para a e 4, respectivamente.

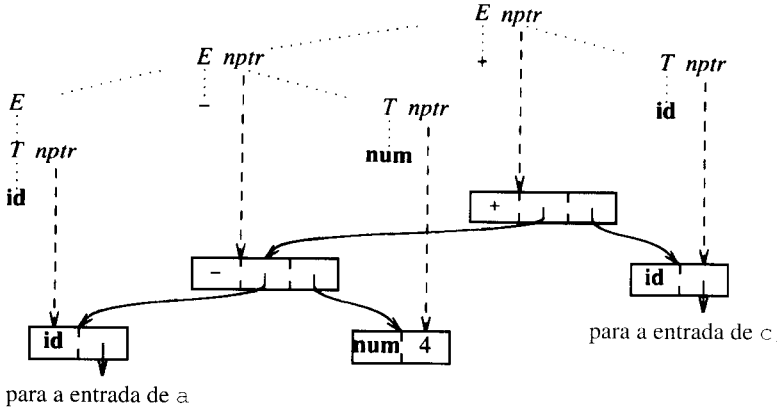


Fig. 5.10. Construção de uma árvore sintática para a-4+c.

Ao se interpretar a Fig. 5.10, é importante compreender que a árvore de nível mais baixo, formada a partir de registros, é uma árvore sintática "real" que compõe a saída, enquanto que a árvore pontilhada acima é a árvore gramatical, que pode existir somente num sentido figurativo. Na próxima seção, mostramos como uma definição S-atribuída pode ser implementada de forma simples usando-se a pilha de um analisador sintático *bottom-up* para controlar os valores dos atributos. De fato, com esta implementação, as funções de construção de nós são invocadas na mesma ordem que na Fig. 5.7. □

Grafos Dirigidos Acíclicos

Um grafo dirigido acíclico (daqui por diante chamado de GDA) para uma expressão identifica as subexpressões comuns existentes na mesma. Como uma árvore sintática, um GDA possui um nó para cada subexpressão de uma expressão; um nó interior representa um operador e os filhos representam seus operandos. A diferença está em que um nó de um GDA, representando uma subexpressão comum, possui mais de um "pai"; numa árvore sintática, a subexpressão comum seria representada por uma subárvore duplicada.

A Fig. 5.11 contém um GDA para a expressão

```
a + a * (b - c) + (b - c) * d
```

A folha para `a` possui dois pais porque `a` é comum a duas subexpressões, `a` e `a * (b - c)`. Igualmente, ambas as ocorrências da subexpressão comum `b - c` são representadas pelo mesmo nó, que também possui dois pais.

A definição dirigida pela sintaxe da Fig. 5.9 irá construir um GDA em lugar de uma árvore sintática, se modificarmos as operações para construir os nós. Um GDA é obtido se a função que constrói um nó verificar primeiro se já não existe um nó idêntico. Por exemplo, antes de construir um novo nó com rótulo *op* e campos com apontadores para *esquerdo* e *direito*, *criar_nó* (*op, esquerdo, direito*) pode verificar se um tal nó já não foi construído. Se já o tiver sido, *criar_nó* (*op, esquerdo, direito*) pode retornar um apontador para aquele nó previamente construído. A função de construção de folhas *criar_folha* pode se comportar de forma semelhante.

Exemplo 5.9. A seqüência de instruções na Fig. 5.12 constrói o GDA da Fig. 5.11, providenciado que *criar_nó* e *criar_folha* criem novos nós somente quando necessário, retornando apontadores para os nós já existentes com os rótulos e filhos corretos sempre que possível. Na Fig. 5.12, *a, b, c* e *d* apontam para entradas na tabela de símbolos para os identificadores `a`, `b`, `c` e `d`.

Quando a chamada para *criar_nó* (**id,** *a*) é repetida à linha 2, o nó construído pela chamada anterior *criar_nó* (**id,** *a*) é retornado, e, por conseguinte, $p_1 = p_2$. Semelhantemente, os nós retornados às linhas 8 e 9 são os mesmos que aqueles retornados às linhas 3 e 4, respectivamente. Conseqüentemente, o nó retornado à linha 10 precisa ser o mesmo que aquele construído pela chamada de *criar_nó* à linha 5. □

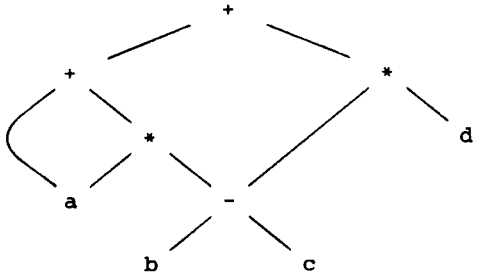


Fig. 5.11. GDAs para a expressão `a + a * (b - c) + (b - c) * d`.

(1) p_1 := *criar_folha* (**id,** *a*);
(2) p_2 := *criar_folha* (**id,** *a*);
(3) p_3 := *criar_folha* (**id,** *b*);
(4) p_4 := *criar_folha* (**id,** *c*);
(5) p_5 := *criar_nó* ('-', p_3, p_4);
(6) p_6 := *criar_nó* ('*', p_2, p_5);
(7) p_7 := *criar_nó* ('+', p_1, p_6);
(8) p_8 := *criar_folha* (**id,** *b*);
(9) p_9 := *criar_folha* (**id,** *c*);
(10) p_{10} := *criar_nó* ('-', p_8, p_9);
(11) p_{11} := *criar_folha* (**id,** *d*);
(12) p_{12} := *criar_nó* ('*', p_{10}, p_{11});
(13) p_{13} := *criar_nó* ('+', p_7, p_{12});

Fig. 5.12. Instruções para construir o GDA da Fig. 5.11.

Em muitas aplicações, os nós são implementados como registros armazenados num *array*, como na Fig. 5.13. Naquela figura, cada registro possui um campo de rótulo que determina a natureza do nó. Podemos nos referir a um nó por seu índice ou posição no *array*. O índice inteiro de um nó é freqüentemente chamado de *número de valor*, por razões históricas. Por exemplo, usando os números de valor, podemos dizer que o nó 3 possui um rótulo +, que seu filho à esquerda é o nó 1 e que o filho à direita é o nó 2. O seguinte algoritmo pode ser usado a fim de criar os nós para uma representação sob a forma de GDA para uma expressão.

Algoritmo 5.1. Método dos números de valor para construir um nó num GDA.

Suponhamos que os nós sejam armazenados num *array*, como na Fig. 5.13, e que cada nó seja referenciado através de seu número de valor. Seja a *assinatura* de um nó de operador uma tripla $< op, l, r >$ consistindo em seu rótulo *op*, filho à esquerda *l* e filho à direita *r*.

Entrada. Rótulo *op*, nó *l* e nó *r*.

Saída. Um nó com assinatura $<op, l, r>$.

Método. Procurar no *array* por um nó *m* com rótulo *op*, filho à esquerda *l* e filho à direita *r*. Se existir um tal nó, retornar *m*; de outra forma, criar um novo nó *n* com rótulo *op*, filho à esquerda *l* e filho à direita *r* e retornar *n*.

Uma forma óbvia de determinar se o nó *m* já está no *array* é manter todos os nós previamente criados numa lista e examinar cada nó procurando pela assinatura desejada. A pesquisa para *m* pode ser tornada mais eficiente usando-se *k* listas, chamadas de *buckets*, e usando-se uma função de *hash h* para determinar o *bucket* a pesquisar.[1]

A função de *hash h* computa o número do *bucket* a partir do valor de *op, l* e *r*. Irá sempre retornar o mesmo número de *bucket*, dados os mesmos argumentos. Se *m* não estiver no *bucket h(op, l, r)*, então um novo nó *n* é criado e adicionado ao mesmo, de forma que as pesquisas subseqüentes o encontrarão lá. Várias assinaturas podem colidir no mesmo número de *bucket*, mas na prática, esperamos que cada *bucket* contenha somente um pequeno número de nós.

Cada *bucket* pode ser implementado como uma lista ligada, como mostrado na Fig. 5.14. Cada célula numa lista ligada representa um nó. Os cabeçalhos de *buckets*, consistindo em apontadores para a primeira célula na lista, são armazenados num *array*. O número de

[1]Qualquer estrutura de dados que implemente dicionários no sentido de Aho, Hopcroft e Ullman [1983] é suficiente. A propriedade importante da estrutura é que dada uma chave, isto é, um rótulo *op* e dois nós *l* e *r*, possamos rapidamente obter um nó *m* com assinatura $<op, l, r>$ ou determinar que nenhum existe.

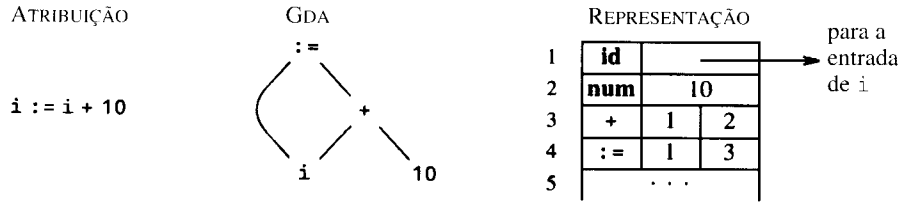


Fig. 5.13. Nós em um GDA para `i := i + 10` alocados a partir de um *array*.

bucket retornado por *h(op, l, r)* é um índice para esse *array* de cabeçalhos de *buckets*.

Este algoritmo pode ser adaptado para ser aplicado a nós que não são reservados seqüencialmente a partir do *array*. Em muitos compiladores, os nós são reservados à medida que forem necessários, de forma a evitar a pré-alocação de um *array* que abrigue nós demais a maior parte do tempo e, em algum instante, não tenha nós suficientes. Neste caso, não podemos assumir que os nós estejam localizados em memória seqüencial e, por conseguinte, teremos que usar apontadores para nos referirmos aos mesmos. Se a função de *hash* pode ser construída para computar o número do *bucket* a partir de um rótulo e de apontadores para os filhos, podemos usar apontadores para os nós em lugar dos números de valor. Caso contrário, podemos numerar os nós de qualquer forma e usar esse número como o número de valor do nó. □

Os GDAs podem também ser usados para representar conjuntos de expressões, uma vez que um GDA pode ter mais de uma raiz. Nos Capítulos 9 e 10, as computações realizadas por uma seqüência de enunciados de atribuição serão representadas por um GDA.

5.3 AVALIAÇÃO *BOTTOM-UP* DE DEFINIÇÕES S-ATRIBUÍDAS

Agora que já vimos como usar as definições dirigidas pela sintaxe para especificar traduções, podemos começar o estudo sobre como implementar os tradutores para as mesmas. Pode ser difícil de se construir um tradutor para uma definição dirigida pela sintaxe, arbitrária. Entretanto, existem amplas classes de definições dirigidas pela sintaxe, úteis, para as quais é fácil construir tradutores. Nesta seção, examinamos uma de tais classes: a das definições S-atribuídas, ou seja, as definições dirigidas pela sintaxe somente com atributos sintetizados. As seções subseqüentes consideram a implementação de definições que tenham atributos herdados.

Os atributos sintetizados podem ser avaliados por um analisador sintático *bottom-up* na medida em que a entrada seja decomposta gramaticalmente. O analisador sintático controla os valores dos atributos sintetizados associados aos símbolos gramaticais em sua pilha. Sempre que uma redução for realizada, os valores dos novos atributos sintetizados são computados a partir dos atributos que figuram na pilha, para os símbolos gramaticais do lado direito da produção redutora. Esta seção mostra como a pilha do analisador sintático pode ser ampliada de forma a abrigar os valores desses atributos sintetizados. Veremos na Seção 5.6 que esta implementação também suporta alguns atributos herdados.

Somente os atributos sintetizados figuram na definição dirigida pela sintaxe da Fig. 5.9 para a construção da árvore sintática de uma expressão. A abordagem desta seção pode subseqüentemente ser aplicada na construção de árvores sintáticas durante a análise sintática *bottom-up*. Como veremos na Seção 5.5, a tradução de expressões durante a análise sintática *top-down* freqüentemente usa atributos herdados. Por conseguinte, postergaremos a tradução durante a análise sintática *top-down* até que as dependências "esquerda-direita" sejam examinadas na próxima seção.

Atributos Sintetizados na Pilha do Analisador Sintático

Um tradutor para uma definição S-atribuída, pode freqüentemente ser implementado com o auxílio de um gerador de analisadores sintáticos LR, tal como aquele discutido na Seção 4.9. A partir de uma definição S-atribuída, o gerador de analisadores sintáticos pode construir um tradutor que avalie os atributos à medida que decomponha gramaticalmente a entrada.

Um analisador sintático *bottom-up* utiliza uma pilha para guardar informações sobre as subárvores que tenham sido estruturadas. Podemos usar campos extra na pilha do analisador sintático para abrigar os valores dos atributos sintetizados. A Fig. 5.15 mostra um exemplo de uma pilha sintática com espaço para um valor de atributo. Vamos supor, como na figura, que a pilha seja implementada através de um par de *arrays estado* e *valor*. Cada entrada de *estado* é um apontador (ou índice) de uma tabela sintática LR(1) (note-se que o símbolo gramatical está implícito no estado e não precisa ser armazenado na pilha). É conveniente, entretanto, referirmo-nos ao estado através do único símbolo gramatical que o mesmo cobre quando colocado na pilha sintática, de acordo com o descrito na Seção 4.7. Se o *i*-ésimo símbolo de *estado* for *A*, então *val*[*i*] irá abrigar o valor do atributo associado ao nó da árvore gramatical correspondente a esse *A*.

O topo corrente da pilha é indicado através do apontador *topo*. Assumimos que os atributos sintetizados sejam avaliados exatamente antes de cada redução. Suponhamos que a regra semântica $A.a := f(X.x, Y.y, Z.z)$ esteja associada à produção $A \rightarrow XYZ$. Antes que *XYZ* seja reduzida a *A*, o valor do atributo *Z.z* está em *Val*[*topo*], o de *Y.y* em *val*[*topo*-1] e o de *X.x* em *val*[*topo*-2]. Se um símbolo não possuir atributo, então a entrada correspondente no *array val* é indefinida. Após a redução, *topo* é decrementado de 2, o estado que cobre *A* é colocado

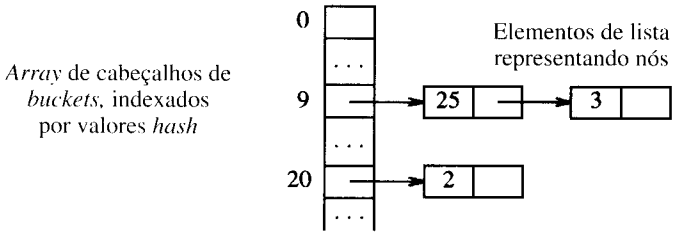


Fig. 5.14. Estrutura de dados para pesquisar *buckets*.

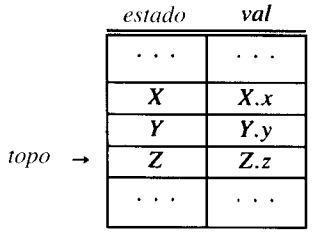


Fig. 5.15. Pilha do analisador sintático com um campo para atributos sintetizados.

em *estado* [*topo*] (isto é, onde *X* estava) e o valor do atributo sintetizado *A.a* é colocado em *val*[*topo*].

Exemplo 5.10. Consideremos de novo a definição dirigida pela sintaxe da calculadora de mesa da Fig. 5.2. Os atributos sintetizados na árvore sintática anotada da Fig. 5.3 podem ser avaliados por um analisador sintático LR durante a decomposição gramatical *bottom-up* da linha de entrada 3*5+4**n**. Como antes, assumimos que o analisador sintático forneça o valor do atributo **dígito.***lexval*, que é o valor numérico de cada *token* que representa um dígito. Quando o analisador sintático empilha um **dígito,** o *token* **dígito** é colocado em *estado* [*topo*] e seu valor de atributo é colocado em *val* [*topo*].

Podemos usar as técnicas da Seção 4.7 para construir um analisador sintático LR para a gramática subjacente. Para avaliar os atributos, modificamos o analisador sintático de forma a executar os fragmentos de código mostrados na Fig. 5.16, exatamente antes de realizar as reduções apropriadas. Note-se que podemos associar a avaliação de atributos às reduções porque cada redução determina a produção a ser aplicada. Os fragmentos de código foram obtidos a partir das regras semânticas na Fig. 5.2 substituindo-se cada atributo por uma posição no *array val*.

Os fragmentos de código não exibem como as variáveis *topo* e *ntopo* são gerenciadas. Quando uma produção com *r* símbolos à direita é reduzida, o valor de *ntopo* é feito igual a *topo-r*+1. Após cada fragmento de código ser executado, *topo* é igualado a *ntopo*.

A Fig. 5.17 mostra a seqüência de movimentos feita pelo analisador sintático para a entrada 3*5+4**n.** O conteúdo dos campos *estado* e *val* da pilha sintática é mostrado após cada movimento. Tomamos de novo a liberdade de substituir os estados da pilha pelos seus símbolos gramaticais correspondentes. Tomamos a liberdade adicional de mostrar, em lugar do *token* **dígito,** o valor efetivo do dígito de entrada.

Consideremos a seqüência de eventos ao se enxergar o símbolo de entrada 3. No primeiro movimento, o analisador sintático empilha o estado correspondente ao *token* **dígito** (cujo valor de atributo é 3) na pilha sintática (o estado é representado por 3 e o valor 3 está no campo *val*). No segundo movimento, o analisador sintático reduz através da produção $T \rightarrow F$. Nenhum fragmento de código está associado a esta produção e, conseqüentemente, o *array val* é deixado sem modificações. Note-se que, após cada redução, o topo da pilha *val* contém o valor de atributo associado ao lado esquerdo da produção redutora. □

Na implementação esboçada acima, os fragmentos de código são executados exatamente antes da redução ocorrer. As reduções providenciam um "gancho" no qual ações constituídas de fragmentos de código arbitrários podem ser penduradas. Ou seja, podemos permitir que o usuário associe, a uma produção, uma ação que seja executada quando a redução relativa àquela produção tiver lugar. Os esquemas de tradução considerados na próxima seção providenciam uma notação para se entremear as ações com a análise sintática. Na Seção 5.6, veremos como uma classe mais ampla de definições dirigidas pela sintaxe pode ser implementada durante uma análise sintática *bottom-up*.

PRODUÇÃO	FRAGMENTO DE CÓDIGO
$L \rightarrow E$**n**	*imprimir* (*val*[*topo*])
$E \rightarrow E_1 + T$	*val*[*ntopo*] := *val*[*topo* − 2] + *val*[*topo*]
$E \rightarrow T$	
$T \rightarrow T_1 * F$	*val*[*ntopo*]: = *val*[topo − 2] × *val*[*topo*]
$T \rightarrow F$	
$F \rightarrow (E)$	*val*[*ntopo*] := *val*[*topo* − 1]
$F \rightarrow$ **dígito**	

Fig. 5.16. Implementação de uma calculadora de mesa com um analisador sintático LR.

ENTRADA	*estado*	*val*	PRODUÇÃO USADA
3 * 5 + 4 **n**	−	−	
* 5 + 4 **n**	3	3	
* 5 + 4 **n**	*F*	3	$F \rightarrow$ **dígito**
* 5 + 4 **n**	*T*	3	$T \rightarrow F$
5 + 4 **n**	*T* *	3 −	
+ 4 **n**	*T* * 5	3 − 5	
+ 4 **n**	*T* * *F*	3 − 5	$F \rightarrow$ **dígito**
+ 4 **n**	*T*	15	$T \rightarrow T * F$
+ 4 **n**	*E*	15	$E \rightarrow T$
4 **n**	*E* +	15 −	
n	*E* + 4	15 − 4	
n	*E* + *F*	15 − 4	$F \rightarrow$ **dígito**
n	*E* + *T*	15 − 4	$T \rightarrow F$
n	*E*	19	$E \rightarrow E + T$
	E **n**	19 −	
	L	19	$L \rightarrow E$ **n**

Fig. 5.17. Movimentos realizados pelo tradutor da entrada 3*5+4 **n**.

5.4 DEFINIÇÕES L-ATRIBUÍDAS

Quando a tradução tem lugar durante a análise sintática, a ordem de avaliação dos atributos é ligada à ordem na qual os nós da árvore gramatical são "criados" pelo método de análise gramatical. Uma ordem natural, que caracteriza muitos métodos de tradução *top-down* e *bottom-up*, é aquela obtida através da aplicação do procedimento *visitar-df* na Fig. 5.18 à raiz da árvore gramatical. Chamamos esta ordem de avaliação de *ordem de pesquisa em profundidade.** Ainda que a árvore gramatical não seja construída de fato, é útil estudar a tradução durante a análise sintática considerado a avaliação dos atributos, através de um método de pesquisa em profundidade sobre os nós da árvore gramatical.

Introduzimos agora uma classe de definições dirigidas pela sintaxe, chamada de definições L-atribuídas, cujos atributos podem sempre ser avaliados numa ordem de pesquisa em profundidade (o L está no lugar de *left* (esquerda), porque as informações de atributos aparecem no fluxo da esquerda para a direita). A implementação de classes progressivamente mais amplas de definições L-atribuídas é coberta nas próximas três seções deste capítulo. As definições L-atribuídas incluem todas as definições dirigidas pela sintaxe baseadas nas gramáticas LL (1); a Seção 5.5 fornece um método de implementar tais definições numa única passagem usando métodos de análise sintática preditiva. Uma classe mais ampla de definições L-atribuídas é implementada na Seção 5.6 durante a análise sintática *bottom-up*, ampliando-se os métodos da Seção 5.3. Um método geral de implementar todas as definições L-atribuídas é delineado na Seção 5.7.

Definições L-Atribuídas

Uma definição dirigida pela sintaxe é *L-atribuída* se cada atributo herdado de X_j, $1 \leq j \leq n$, do lado direito de $A \rightarrow X_1X_2 \ldots X_n$ depender somente

1. dos atributos dos símbolos $X_1, X_2, \ldots, X_{j-1}$ à esquerda de X_j na produção e
2. dos atributos herdados de *A*.

Note-se que cada definição S-atribuída é L-atribuída porque as restrições (1) e (2) se aplicam somente aos atributos herdados.

Exemplo 5.11. A definição dirigida pela sintaxe da Fig. 5.19 não é L-atribuída porque o atributo herdado *Q.i* do símbolo gramatical *Q* de-

*Do original em inglês: *depth first order*. (N. do T.)

```
procedimento visitar_df (n: nó);
início
    para cada filho m de n, da esquerda para a direita faça
        início
            avaliar os atributos herdados de m;
            visitar_df (m)
    fim;
    avaliar os atributos sintetizados de n
fim
```

Fig. 5.18. Ordem de avaliação em profundidade para os atributos de uma árvore gramatical.

PRODUÇÃO	REGRAS SISTEMÁTICAS
$A \rightarrow L\ M$	$L.i := l(A.i)$
	$M.i := m(L.s)$
	$A.s := f(M.s)$
$A \rightarrow Q\ R$	$R.i := r(A.i)$
	$Q.i := q(R.s)$
	$A.s := f(Q.s)$

Fig. 5.19. Uma definição dirigida pela sintaxe que é L-atribuída

pende do atributo $R.s$ do símbolo gramatical à sua direita. Outros exemplos de definições que não são L-atribuídas podem ser encontrados nas Seções 5.8 e 5.9. □

Esquemas de Tradução

Um esquema de tradução é uma gramática livre de contexto na qual os atributos são associados aos símbolos gramaticais e as ações semânticas envolvidas entre chaves { } são inseridas nos lados direitos das produções, como na Seção 2.3. Usaremos os esquemas de tradução neste capítulo como uma notação útil para a especificação da tradução durante a análise sintática.

Os esquemas de tradução considerados neste capítulo tanto podem ter atributos sintetizados quanto herdados. Nos esquemas simples de tradução considerados no Capítulo 2, os atributos eram do tipo cadeia, um para cada símbolo, e, para cada produção $A \rightarrow X_1 \ldots X_n$, a regra semântica formava a cadeia para A através da concatenação das cadeias para $X_1, \ldots, X_n$, nessa ordem, com algumas cadeias adicionais e opcionais entremeadas. Vimos que podíamos realizar a tradução simplesmente imprimindo as cadeias literais na ordem em que apareciam nas regras semânticas.

Exemplo 5.12. Aqui está um esquema simples de tradução que mapeia expressões infixas, com adição e subtração, nas expressões posfixas correspondentes. É uma pequena revisão do trabalho feito no esquema de tradução (2.14) do Capítulo 2.

$E \rightarrow T\ R$
$R \rightarrow$ **op_aditivo** T {*imprimir* (**op_aditivo**.*lexema*)} $R_1 \mid \epsilon$ (5.1)
$T \rightarrow$ **num** {*imprimir* (**num**.*val*)}

A Fig. 5.20 mostra a árvore gramatical para a entrada 9-5+2, com cada ação semântica atrelada apropriadamente como filho de um nó o qual corresponde ao lado esquerdo da produção em que figura a ação semântica. Com efeito, tratamos as ações como se fossem símbolos terminais, um ponto de vista que é um mneumônico conveniente para se estabelecer quando as ações devem ser executadas. Tomamos a liberdade de mostrar os números efetivos e o operador aditivo em lugar dos *tokens* **num** e **op_aditivo.** Quando executadas na ordem de pesquisa em profundidade, as ações na Fig. 5.20 imprimem a saída 95-2+. □

Quando designamos um esquema de tradução, precisamos observar algumas restrições de forma a assegurar que um valor de atributo esteja disponível quando uma ação o referenciar. Essas restrições, motivadas pelas definições L-atribuídas, aseguram que uma ação não se refira a um atributo que ainda não foi computado.

O caso mais simples ocorre quando somente atributos sintetizados são necessários. Para esse caso, podemos construir um esquema de tradução através da criação de uma ação, que consista numa atribuição, para cada regra semântica e colocando-se esta ação ao fim do lado direito associado a essa produção. Por exemplo, a produção e a regra semântica

PRODUÇÃO	REGRA SEMÂNTICA
$T \rightarrow T_1 * F$	$T.val := T_1.val \times F.val$

produzem a seguinte produção e ação semântica

$$T \rightarrow T_1 * F\ \{T.val := T_1.val \times F.val\}$$

Se tivermos atributos tanto herdados quanto sintetizados, precisamos ser mais cuidadosos:

1. Um atributo herdado para um símbolo no lado direito de uma produção precisa ser computado numa ação antes daquele símbolo.

2. Uma ação não pode se referir a um atributo sintetizado de um símbolo à direita da ação.

3. Um atributo sintetizado para um não-terminal à esquerda pode somente ser computado após o cômputo de todos os atributos que o mesmo referencie. A ação de computar tais atributos pode ser usualmente colocada ao fim do lado direito da produção.

Nas próximas duas seções, mostramos como um esquema de tradução, que satisfaz essas três exigências, pode ser implementado através de generalizações de analisadores sintáticos *bottom-up* e *top-down*.

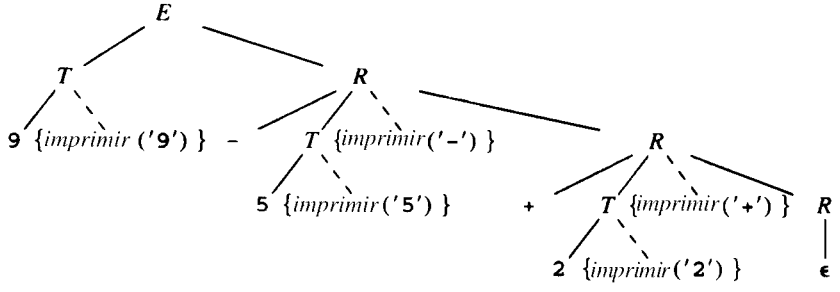


Fig. 5.20. Árvore gramatical para 9-5+2 mostrando as ações.

$$E_{1}.val$$

Fig. 5.21. Colocação de compartimentos dirigida pela sintaxe.

O seguinte esquema de tradução não satisfaz à primeira dessas três exigências.

$$S \rightarrow A_1 A_2 \ \{A_1.in := 1; A_2.in := 2\}$$
$$A \rightarrow a \ \{imprimir\ (A.in)\}$$

Encontramos que o atributo herdado $A.in$ na segunda produção não estará definido quando for realizada uma tentativa de imprimir seu valor, durante uma travessia em profundidade da árvore gramatical para a cadeia de entrada *aa*. Isto é, uma travessia em profundidade começa por S e visita a subárvore para A_1 e A_2 antes que os valores de $A_1.in$ e de A_2 sejam estabelecidos. Se a ação definindo os valores de $A_1.in$ e $A_2.in$ estivesse inserida antes dos A's no lado direito de $S \rightarrow A_1 A_2$, ao invés de depois, então $A.in$ estaria definida a cada vez em que *imprimir (A.in)* fosse executada.

É sempre possível se começar com uma definição dirigida pela sintaxe e construir um esquema de tradução que satisfaça as três exigências acima. O próximo exemplo ilustra esta construção. Está baseado na linguagem de formatação matemática EQN, descrita brevemente na Seção 1.2. Dada a entrada

```
E sub 1 . val
```

EQN coloca E, 1 e $.val$ nas posições relativas e tamanhos mostrados na Fig. 5.21. Note-se que o subscrito 1 é impresso num tamanho e fontes menores e é movido para baixo em relação a E e $.val$.

Exemplo 5.13. A partir da definição L-atribuída na Fig. 5.22, iremos construir o esquema de tradução da Fig. 5.23. Nas figuras, o não-terminal B (para *box*, no original em inglês) representa uma fórmula. A produção $B \rightarrow B\,B$ representa a justaposição de dois quadros e $B \rightarrow B$ **sub** B representa a colocação do segundo quadro subscrito num tamanho menor do que o primeiro quadro na posição relativa própria a um subscrito.

O atributo herdado tp (para tamanho de ponto) afeta a largura de uma fórmula. A regra para a produção $B \rightarrow$ **texto** faz com que a largura normalizada do texto seja multiplicada pelo tamanho de ponto a fim de se obter a largura efetiva do texto. O atributo l, de **texto,** é obtido através de uma pesquisa de tabela, dado o caractere representado pelo *token* **texto.** Quando a produção $B \rightarrow B_1 B_2$ é aplicada, B_1 e B_2 herdam o tamanho de ponto por causa das regras de cópia. A largura de B, representada pelo atributo sintetizado lg, é a maior das larguras de B_1 e B_2.

PRODUÇÃO	REGRAS SEMÂNTICAS
$S \rightarrow B$	$B.tp := 10$
	$S.lg := B.lg$
$B \rightarrow B_1 B_2$	$B_1.tp := B.tp$
	$B_2.tp := B.tp$
	$B.lg := max(B_1.lg, B_2.lg)$
$B \rightarrow B_1$ **sub** B_2	$B_1.tp := B.tp$
	$B_2.tp := comprimir\ (B.tp)$
	$B.lg := desloc(B_1.lg, B_2.lg)$
$B \rightarrow$ **texto**	$B.lg :=$ **texto.**$l \times B.tp$

Fig. 5.22. Definição dirigida pela sintaxe para os tamanhos e largura de quadros.

$S \rightarrow$		$\{B.tp := 10\}$
	B	$\{S.lg := B.lg\}$
$B \rightarrow$		$\{B_1.tp := B.tp\}$
	B_1	$\{B_2.tp := B.tp\}$
	B_2	$\{B.lg := max\ (B_1.lg, B_2.lg)\ \}$
$B \rightarrow$		$\{B_1.tp := B.tp\}$
	B_1	
	sub	$\{B_2.tp := comprimir\ (B.tp)\ \}$
	B_2	$\{B.lg := desloc(B_1.lg, B_2.lg)\ \}$
$B \rightarrow$ **texto**		$\{B.lg :=$ **texto.**$l \times B.tp\}$

Fig. 5.23. Esquema de tradução construído a partir da Fig. 5.22.

Quando a produção $B \rightarrow B_1$ **sub** B_2 é usada, a função *comprimir* reduz o tamanho de ponto de B_2 em 30%. A função *desloc* permite um deslocamento para baixo do quadro B_2 enquanto computa a largura de B. As regras que geram os comandos efetivos do compositor de tipos como saída não são mostradas.

A definição da Fig. 5.22 é L-atribuída. O único atributo herdado é tp, do não-terminal B. Cada regra semântica define tp somente em termos do atributo herdado do não-terminal à esquerda da produção. Por conseguinte, a definição é L-atribuída.

O esquema de tradução na Fig. 5.23 é obtido através da inserção de atribuições correspondentes às regras semânticas na Fig. 5.22 dentro das produções, seguindo as três exigências anteriores. Para melhorar a legibilidade, cada símbolo gramatical numa produção é escrito numa linha separada e as ações são mostradas à direita. Conseqüentemente,

$$S \rightarrow \{B.tp := 10\}\ B\ \{S.lg := B.lg\}$$

é escrita como

$$S \rightarrow \{B.tp := 10\}$$
$$B\ \{S.lg := B.lg\}$$

Note-se que as ações que estabelecem os valores dos atributos herdados $B_1.tp$ e $B_2.tp$ figuram exatamente antes de B_1 e B_2 nos lados direitos das produções. □

5.5 TRADUÇÃO *TOP-DOWN*

Nesta seção, as definições L-atribuídas serão implementadas durante a análise gramatical preditiva. Trabalhamos com esquemas de tradução ao invés de definições dirigidas pela sintaxe de forma a sermos explícitos a respeito da ordem na qual as ações e as avaliações de atributos ocorrem. Estendemos igualmente o algoritmo de eliminação da recursão à esquerda para os esquemas de tradução com atributos sintetizados.

Eliminação da Recursão à Esquerda de um Esquema de Tradução

Uma vez que a maioria dos operadores aritméticos é associativa à esquerda, é natural usar gramáticas recursivas à esquerda para expressões. Estendemos o algoritmo para eliminação da recursão à esquerda nas Seções 2.4 e 4.3 de forma a permitir atributos quando a gramática subjacente de um esquema de tradução for transformada. A transformação se aplica a esquemas de tradução com atributos sintetizados. Permite que muitas das definições dirigidas pela sintaxe das Seções 5.1 e 5.2 sejam implementadas utilizando a análise sintática preditiva. O próximo exemplo motiva a transformação.

Exemplo 5.14. O esquema de tradução da Fig. 5.24 é transformado abaixo no esquema de tradução da Fig. 5.25. O novo esquema produz a árvore gramatical anotada da Fig. 5.26 para a expressão `9-5+2`. As

$$
\begin{array}{lll}
E \rightarrow & E_1 + T & \{E.\,val := E_1.val + T.val\} \\
E \rightarrow & E_1 - T & \{E.\,val := E_1.val - T.val\} \\
E \rightarrow & T & \{E.\,val := T.val\} \\
T \rightarrow & (\,E\,) & \{T.\,val := E.val\} \\
\mathrm{T} \rightarrow & \textbf{num} & \{T.\,val := \textbf{num}.val\}
\end{array}
$$

Fig. 5.24. Esquema de tradução com uma gramática recursiva à esquerda.

setas na figura sugerem uma forma para a determinação do valor da expressão.

Na Fig. 5.26, os números individuais são gerados por T, e $T.val$ toma seu valor a partir do valor léxico do número, dado pelo atributo **num**.*val*. O 9 na subexpressão 9-5 é gerado pelo T mais à esquerda, mas o operador menos e o 5 são gerados pelo R do filho à direita da raiz. O atributo herdado $R.i$ obtém o valor 9 a partir de $T.val$. A subtração 9-5 e a passagem abaixo do resultado 4 ao nó do meio, para R, são feitas inserindo-se a seguinte ação entre T e R_1 em $R \rightarrow - \; T \; R_1$:

$$\{R_1.i := R.i - T.val\}$$

Uma ação similar adiciona 2 ao valor de 9-5 produzindo o resultado $R.i = 6$ ao nó de fundo para R. Esse resultado é necessário à raiz, como o valor de $E.val$; o atributo sintetizado s para R, não mostrado na Fig. 5.26, é usado para copiar o resultado acima até a raiz. □

Para a análise sintática *top-down*, podemos assumir que uma ação seja executada no tempo em que um símbolo na mesma posição é expandido. Por conseguinte, na segunda produção da Fig. 5.25 a primeira ação (atribuição a $R.i$) é realizada após T ter sido completamente expandido em terminais e a segunda após R_1 ter sido completamente expandido. Como notado na discussão das definições L-atribuídas da Seção 5.4, um atributo herdado de um símbolo precisa ser computado por uma ação que figure antes do símbolo e um atributo sintetizado do não-terminal à esquerda precisa ser computado após o cômputo de todos os atributos dos quais dependa.

Com a finalidade de adaptar à análise sintática preditiva outros esquemas de tradução recursivos à esquerda, expressaremos o uso dos atributos $R.i$ e $R.s$ na Fig. 5.25 mais abstratamente. Suponhamos ter o seguinte esquema de tradução

$$
\begin{array}{ll}
A \rightarrow A_1 \; Y & \{A.a := g(A_1.a,\; Y.y)\} \\
A \rightarrow X & \{A.a := f(X.x)\}
\end{array}
\tag{5.2}
$$

Cada símbolo gramatical possui um atributo sintetizado, escrito usando a letra minúscula correspondente, e f e g são funções arbitrárias. A generalização para produções-A adicionais e para produções com cadeias em lugar dos símbolos X e Y pode ser feita como no Exemplo 5.15 abaixo.

$$
\begin{array}{lll}
E \rightarrow & T & \{\;\; R.i := T.val\} \\
 & R & \{E.val := R.s\} \\
R \rightarrow & + & \\
 & T & \{\; R_1.i := R.i + T.val\} \\
 & R_1 & \{\;\; R.s := R_1.s\} \\
R \rightarrow & - & \\
 & T & \{\; R_1.i := R.i - T.val\} \\
 & R_1 & \{\;\; R.s := R_1.s\} \\
R \rightarrow & \epsilon & \{\;\; R.s := R.i\} \\
T \rightarrow & (& \\
 & E & \\
 &) & \{T.val := E.val\} \\
T \rightarrow & \textbf{num} & \{T.val := \textbf{num}.val\}
\end{array}
$$

Fig. 5.25. Esquema de tradução transformado com gramática recursiva à direita.

O algoritmo para eliminar a recursão à esquerda na Seção 2.4 constrói a seguinte gramática a partir de (5.2):

$$
\begin{array}{l}
A \rightarrow X \; R \\
R \rightarrow Y \; R \mid \epsilon
\end{array}
\tag{5.3}
$$

Levando em conta as ações semânticas, o esquema transformado se torna

$$
\begin{array}{ll}
A \rightarrow X & \{R.i := f(X.x)\} \\
\quad R & \{A.a := R.s\} \\
R \rightarrow Y & \{R_1.i := g(R.i, Y.y)\} \\
\quad R_1 & \{R.s := R_1.s\} \\
R \rightarrow \epsilon & \{R.s := R.i\}
\end{array}
\tag{5.4}
$$

O esquema transformado usa os atributos i e s para R, como na Fig. 5.25. Para vermos por que os resultados de (5.2) e de (5.4) são os mesmos, consideremos as duas árvores gramaticais anotadas na Fig. 5.27. O valor de $A.a$ é computado de acordo com (5.2) na Fig. 5.27(a). A Fig. 5.27(b) contém os cômpulos sucessivos de $R.i$ árvore abaixo, de acordo com (5.4). O valor de $R.i$ ao fundo é passado acima imodificado e se torna o valor correto de $A.a$ à raiz ($R.s$ não é mostrado na Fig. 5.27(b)).

Exemplo 5.15. Se a definição dirigida pela sintaxe na Fig. 5.9 para a construção de árvores sintáticas for convertida para um esquema de tradução, as produções e ações semânticas para E se tornam:

$$
\begin{array}{ll}
E \rightarrow E_1 + T & \{E.nptr := criar\text{-}nó\,('+', E_1.nptr, T.nptr)\} \\
E \rightarrow E_1 - T & \{E.nptr := criar\text{-}nó\,('-', E_1.nptr, T.nptr)\} \\
E \rightarrow T & \{E.nptr := T.nptr\}
\end{array}
$$

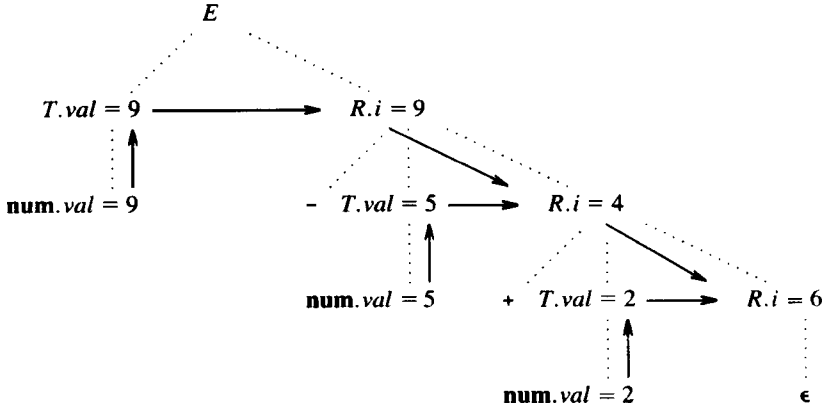


Fig. 5.26. Avaliação da expressão 9-5+2.

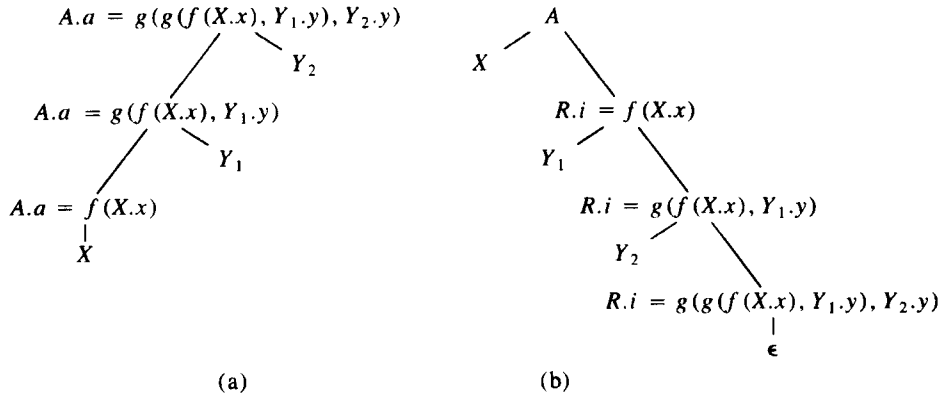


Fig. 5.27. Duas formas de computar um valor de atributo.

Quando a recursão à esquerda é eliminada deste esquema de tradução, o não-terminal E corresponde a A em (5.2) e as cadeias $+\ T$ e $-\ T$, nas duas primeiras produções, correspondem a Y; o não-terminal T na terceira produção corresponde a X. O esquema de tradução transformado é mostrado na Fig. 5.28. As produções e as ações semânticas para T são similares àquelas na definição original na Fig. 5.9.

A Fig. 5.29 mostra como as ações na Fig. 5.28 constroem a árvore sintática para `a-4+c`. Os atributos sintetizados são mostrados à direita do nó para um símbolo gramatical e os atributos herdados são mostrados à esquerda. Uma folha na árvore sintática é construída pelas ações associadas às produções $T \rightarrow$ **id** e $T \rightarrow$ **num**, como no Exemplo 5.8. Ao T mais à esquerda, o atributo $T.nptr$ aponta para a folha de `a`. Um apontador para o nó de `a` é herdado como o atributo $R.i$ no lado direito de $E \rightarrow T\ R$.

Quando a produção $R \rightarrow -T\ R_1$ é aplicada ao filho à direita da raiz, $R.i$ aponta para o nó de `a` e $T.nptr$ para o de `4`. O nó para `a-4` é construído aplicando-se *criar_nó* ao operador menos juntamente com esses apontadores.

Finalmente, quando a produção $R \rightarrow \epsilon$ é aplicada, $R.i$ aponta para a raiz de toda a árvore sintática. Toda a árvore sintática é retornada através dos atributos s dos nós para R (não mostrados na Fig. 5.29) até que se torne o valor de $E.nptr$. □

Projeto de um Tradutor Preditivo

O próximo algoritmo generaliza a construção de analisadores sintáticos preditivos para implementar um esquema de tradução baseado numa gramática adequada à análise sintática *top-down*.

Algoritmo 5.2. Construção de um tradutor preditivo dirigido pela sintaxe.

$E \rightarrow$	T	$\{\ R.i := T.nptr\}$
	R	$\{E.nptr := R.s\}$
$R \rightarrow$	$+$	
	T	$\{\ R_1.i := criar_nó\ ('+',\ R.i,\ T.nptr)\ \}$
	R_1	$\{\ R.s := R_1 \cdot s\}$
$R \rightarrow$	$-$	
	T	$\{\ R_1.i := criar\text{-}nó\ ('-';\ R.i,\ T.nptr)\ \}$
	R_1	$\{\ R.s := R_1.s\}$
$R \rightarrow$	ϵ	$\{\ R.s := R.i\}$
$T \rightarrow$	(	
	E	
	)	$\{T.nptr := E.nptr\}$
$T \rightarrow$	**id**	$\{T.nptr := criar_folha\ (\mathbf{id}, \mathbf{id} \cdot entrada)\ \}$
$T \rightarrow$	**num**	$\{T.nptr := criar_folha\ (\mathbf{num}, \mathbf{num}.val)\ \}$

Fig. 5.28. Esquema de tradução transformado para construir árvores sintáticas.

Entrada. Um esquema de tradução dirigido pela sintaxe com uma gramática subjacente adequada à análise sintática preditiva.

Saída. Código para um tradutor dirigido pela sintaxe.

Método. A técnica é uma modificação da construção de um analisador sintático preditivo da Seção 2.4.

1. Para cada não-terminal A, construir uma função que tenha um parâmetro formal para cada atributo herdado de A e que retorne os valores dos atributos sintetizados de A (possivelmente como um registro, como um apontador para um registro com um campo para cada atributo ou usando um mecanismo de passagem por referência para a transmissão de parâmetros, discutido na Seção 7.5). Por simplicidade, assumimos que cada não-terminal tenha exatamente um atributo sintetizado. A função para A possui uma variável local para cada atributo de cada símbolo gramatical que figure numa produção para A.

2. Como na Seção 2.4, o código para o não-terminal A decide que produção usar, baseado no símbolo corrente de entrada.

3. O código associado a cada produção realiza o seguinte. Consideramos os *tokens*, os não-terminais e as ações no lado direito da produção, da esquerda para a direita.

   i) Para o *token* X com atributo sintetizado x, salvar o valor de x na variável declarada para $X.x$. Em seguida, gerar uma chamada para reconhecer o *token* X e avançar a entrada.

   ii) Para o não-terminal B, gerar uma atribuição $c := B(b_1, b_2, ..., b_k)$ com uma chamada de função no lado direito, onde $b_1, b_2, ..., b_k$ são as variáveis para os atributos herdados de B e c é a variável para o atributo sintetizado de B.

   iii) Para uma ação, copiar o código dentro do analisador sintático, substituindo cada referência a um atributo pela variável para aquele atributo. □

O Algoritmo 5.2 é estendido na Seção 5.7 de forma a implementar qualquer definição L-atribuída, uma vez que uma árvore sintática já tenha sido construída. Na Seção 5.8, consideramos formas de melhorar os tradutores construídos pelo Algoritmo 5.2. Por exemplo, pode ser possível eliminar enunciados de cópia da forma $x := y$ ou usar uma única variável para abrigar os valores de vários atributos. Algumas de tais melhorias podem também ser feitas automaticamente utilizando-se os métodos do Capítulo 10.

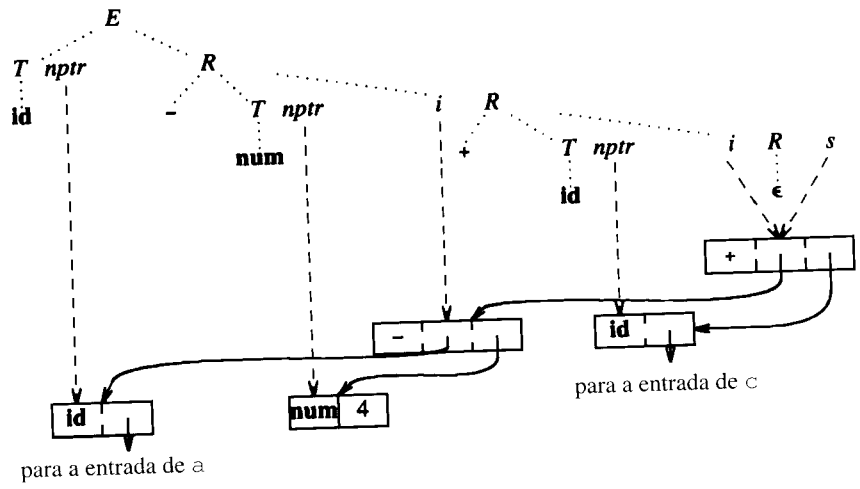


Fig. 5.29. Uso dos atributos herdados para construir árvores sintáticas.

Exemplo 5.16. A gramática na Fig. 5.28 é LL(1) e, por conseguinte, adequada à análise sintática *top-down*. A partir dos atributos dos não-terminais da gramática, obtemos os seguintes tipos de argumentos e resultados para as funções de *E*, *R* e *T*. Uma vez que *E* e *T* não têm atributos herdados, não possuem argumentos.

função *E*: ↑ nó_árvore_sintática;
função *R* (*i*: ↑ nó_árvore_sintática): ↑ nó_árvore_sintática;
função *T*: ↑ nó_árvore_sintática;

Combinamos duas das produções-*R* na Fig. 5.28 para tornar o tradutor menor. As novas produções usam o *token* **op_aditivo** para representar + e −:

$$
\begin{array}{lll}
R \rightarrow \textbf{op_aditivo} & & \\
\quad T & \{R_1.i := criar_nó\ (\textbf{op_aditivo}.lexema,\ R.i,\ T.nptr)\} & \\
\quad R_1 & \{R.s := R_1.s\} & (5.5) \\
R \rightarrow \epsilon & \{R.s := R.i\} &
\end{array}
$$

O código para *R* está baseado no procedimento de análise sintática da Fig. 5.30. Se o símbolo de *lookahead* for **op_aditivo,** a produção $R \rightarrow$ **op_aditivo** *T R* é aplicada utilizando-se o procedimento *reconhecer* a fim de ler o próximo *token* de entrada após **op_aditivo** e, então, chamando os procedimentos para *R* e *T*. Em caso contrário, o procedimento não faz nada, para imitar a produção $R \rightarrow \epsilon$.

O procedimento para *R* na Fig. 5.31 contém código para a avaliação de atributos. O valor léxico *lexval* do *token* **op_aditivo** é guardado em *lexema-op-aditivo*, **op_aditivo** é reconhecido, *T* é chamado e o resultado é guardado usando-se *nptr*. A variável *il* corresponde ao atributo herdado $R_1.i$ e *sl* ao atributo sintetizado $R_1.s$. O enunciado **retornar** retorna o valor de *s* imediatamente antes do controle deixar a função. As funções para *E* e *T* são construídas similarmente. □

```
procedimento R;
início
        se lookahead = op_aditivo então início
                reconhecer (op_aditivo); T; R
        fim
        senão início / * não fazer nada * /
        fim
fim;
```

Fig. 5.30. Procedimento de análise sintática para as produções $R \rightarrow$ **op-aditivo** $T\,R \mid \epsilon$.

5.6 AVALIAÇÃO *BOTTOM-UP* DOS ATRIBUTOS HERDADOS

Nesta seção, apresentamos um método de implementação de definições L-atribuídas sob uma estrutura de análise sintática *bottom-up*. O método é capaz de tratar todas as definições L-atribuídas consideradas na seção precendente, na medida em que pode implementar qualquer definição L-atribuída baseada numa gramática LL(1). Pode também implementar muitas (mas não todas) definições L-atribuídas baseadas em gramáticas LR(1). O método é uma generalização da técnica de tradução *bottom-up*, introduzida na Seção 5.3.

Removendo Ações Infiltradas dos Esquemas de Tradução

No método de tradução *bottom-up* da Seção 5.3, confiamos em que todas as ações de tradução estivessem na extremidade direita das produções, enquanto que no método de análise preditiva necessitamos que as ações fossem infiltradas em vários pontos no lado direito. Para começar nossa discussão sobre como os atributos herdados podem ser tratados de baixo para cima, introduzimos uma transformação que faz to-

```
função R(i: ↑nó_árvore_sintática): ↑nó_árvore_sintática;
        var nptr, il, sl, s: ↑nó-árvore-sintática;
                lexema_op_aditivo: caractere;
início
        se lookahead = op_aditivo então início
                / * produção R → op_aditivo T R */
                lexema_op_aditivo : = lexval;
                reconhecer (op_aditivo);
                nptr : = T;
                il : = criar_nó(lexema_op_aditivo, i, nptr);
                sl : = R(il);
                s : = sl
        fim
        senão s : = i;    / * produção R → ε */
        retornar s
fim;
```

Fig. 5.31. Construção recursivo-descendente de árvores sintáticas.

das as ações infiltradas num esquema de tradução ocorrerem nos finais à direita de suas produções.

A transformação insere novos não-terminais *marcadores* gerando ϵ dentro da gramática básica. Substituímos cada ação infiltrada por um não-terminal marcador distinto M e atrelamos a ação ao final da produção $M \rightarrow \epsilon$. Por exemplo, o esquema de tradução

$E \rightarrow T\ R$

$R \rightarrow +\ T\ \{imprimir\ ('+')\}\ R \mid -\ T\ \{imprimir\ ('-')\}\ R \mid \epsilon$

$T \rightarrow \textbf{num}\ \{imprimir\ (\textbf{num}.val)\}$

é transformado, usando não-terminais marcadores M e N, em

$E \rightarrow T\ R$

$R \rightarrow +\ T\ M\ R \mid -\ T\ N\ R \mid \epsilon$

$T \rightarrow \textbf{num}\ \{imprimir\ (\textbf{num}.val)\}$

$M \rightarrow \epsilon\ \{imprimir\ ('+')\}$

$N \rightarrow \epsilon\ \{imprimir\ ('-')\}$

As gramáticas nos dois esquemas de tradução aceitam exatamente a mesma linguagem e, desenhando uma árvore gramatical com os nós extra para as ações, podemos mostrar que as ações são realizadas na mesma ordem. As ações no esquema transformado de tradução terminam produções e, por conseguinte, podem ser realizadas exatamente antes do lado direito ser reduzido durante a análise sintática *bottom-up*.

Herdando Atributos na Pilha do Analisador Sintático

Um analisador sintático *bottom-up* reduz o lado direito de produção $A \rightarrow XY$ através da remoção de X e de Y do topo da pilha do analisador sintático, substituindo-os por A. Suponhamos que X possua o atributo sintetizado $X.s$, o qual a implementação da Seção 5.3 manteve juntamente com X na pilha do analisador sintático.

Uma vez que o valor de $X.s$ já esteja na pilha e antes que quaisquer reduções tenham lugar na subárvore abaixo de Y, este valor pode ser herdado por Y. Isto é, se o atributo herdado $Y.i$ for definido pela regra de cópia $Y.i := X.s$, o valor de $X.s$ pode ser usado onde $Y.i$ for pedido. Como veremos, as regras de cópia desempenham um importante papel na avaliação dos atributos herdados durante a análise sintática *bottom-up*.

Exemplo 5.17. O tipo de um identificador pode ser passado pelas regras de cópia utilizando-se atributos herdados, como mostrado na Fig. 5.32 (adaptada da Fig. 5.7). Examinaremos primeiro os movimentos feitos por um analisador sintático *bottom-up* para a entrada

```
real p, q, r
```

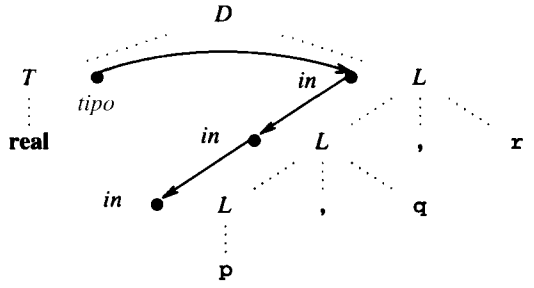


Fig. 5.32. A cada nó para L, $L.in = T.tipo$.

ENTRADA	*estado*	PRODUÇÃO USADA
real p, q, r	–	
p, q, r	**real**	
p, q, r	T	$T \rightarrow$ **real**
,q, r	T p	
,q, r	$T\ L$	$L \rightarrow$ **id**
q, r	$T\ L$,	
, r	$T\ L$, q	
, r	$T\ L$	$L \rightarrow L$, **id**
r	$T\ L$,	
	$T\ L$, r	
	$T\ L$	$L \rightarrow L$, **id**
	D	$D \rightarrow T\ L$

Fig. 5.33. Sempre que um lado direito para L for reduzido, T estará exatamente abaixo do lado direito.

Em seguida, mostraremos como o valor do atributo $T.tipo$ pode receber acesso quando as produções para L forem aplicadas. O esquema de tradução que desejamos implementar é

$D \rightarrow$	T	$\{L.in := T.tipo\}$
	L	
$T \rightarrow$	**int**	$\{T.tipo := inteiro\}$
$T \rightarrow$	**real**	$\{T.tipo := real\}$
$L \rightarrow$		$\{L_1.in := L.in\}$
	L_1, **id**	$\{incluir_tipo\ (\textbf{id}.entrada, L.in)\}$
$L \rightarrow$	**id**	$\{incluir_tipo\ (\textbf{id}.entrada, L.in)\}$

Se ignorarmos as ações no esquema de tradução acima, a seqüência de movimentos realizados pelo analisador sintático para a entrada da Fig. 5.32 é como na Fig. 5.33. Para maior clareza, mostramos na pilha o símbolo gramatical correspondente em lugar do estado e o identificador efetivo ao invés do *token* **id.**

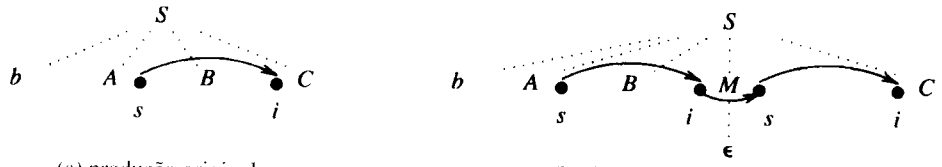
Suponhamos, como na Seção 5.3, que a pilha do analisador sintático seja implementada como um par de *arrays estado* e *val*. Se a entrada *estado*[i] estiver associada ao símbolo gramatical X, então *val*[i] deterá o valor do atributo sintetizado $X.s$. O conteúdo do *array estado* é mostrado na Fig. 5.33. Note-se que a cada vez que o lado direito de uma produção para L for reduzido na Fig. 5.33, T estará na pilha exatamente abaixo desse lado direito. Podemos usar este fato para ter acesso ao valor de atributo $T.tipo$.

A implementação da Fig. 5.34 usa o fato do atributo $T.tipo$ estar em local conhecido, relativo ao topo, na pilha *val*. Sejam *topo* e *ntopo* os índices da entrada de todo da pilha exatamente antes e após a redução ter tido lugar, respectivamente. A partir das regras de cópia que definem $L.in$, sabemos que $T.tipo$ pode ser usado em lugar de $L.in$.

Quando a produção $L \rightarrow$ **id** for aplicada, **id**.*entrada* estará ao topo da pilha *val* e $T.topo$ estará exatamente abaixo da mesma. Por conse-

PRODUÇÃO	CODE FRAGMENT
$D \rightarrow T\ L$;	
$T \rightarrow$ **int**	*val*[*ntopo*] : = *inteiro*
$T \rightarrow$ **real**	*val*[*ntopo*] : = *real*
$L \rightarrow L$, **id**	*incluir_tipo* (*val*[*topo*], *val* [*topo*-3])
$L \rightarrow$ **id**	*incluir_tipo* (*val*[*topo*], *val* [*topo*-1])

Fig. 5.34. O valor de $T.tipo$ é usado em lugar de $L.in$.



(a) produção original (b) dependências modificadas

Fig. 5.35. Copiando um valor de atributo através de um marcador *M*.

guinte, *incluir-tipo (val[topo], val [topo-1]* é equivalente a *incluir_tipo* (**id**.*entrada, T. tipo*). Analogamente, uma vez que o lado direito da produção $L \rightarrow L$, **id** possui três símbolos, *T.tipo* vai aparecer em *val[topo-3]*, quando essa redução tiver lugar. As regras de cópia envolvendo *L.in* são eliminadas porque o valor de *T.tipo* é usado em substituição. □

Simulando a Avaliação de Atributos Herdados

A procura na pilha do analisador sintático por um valor de atributo funciona somente se a gramática permitir que a posição do valor de atributo seja antecipadamente conhecida.

Exemplo 5.18. Como uma instância na qual não podemos predizer a posição, consideremos o seguinte esquema de tradução:

PRODUÇÃO	REGRAS SEMÂNTICAS	
$S \rightarrow aAC$	$C.i := A.s$	
$S \rightarrow bABC$	$C.i := A.s$	(5.6)
$C \rightarrow c$	$C.s := g(C.i)$	

C herda o atributo sintetizado *A.s* através de uma regra de cópia. Note-se que pode ou não haver um *B* entre *A* e *C* na pilha. Quando a redução através de $C \rightarrow c$ for realizada, o valor de *C.i* estará ou em *val[topo-1]* ou em *val [topo-2]*, mas não é claro que caso estará presente.

Na Fig. 5.35, um novo não-terminal marcador *M* é inserido exatamente antes de *C* no lado direito da segunda produção em (5.6). Se estamos analisando sintaticamente de acordo com a produção $S \rightarrow bABMC$, então *C.i* herda o valor de *A.s* indiretamente através de *M.i* e de *M.s*. Quando a produção $M \rightarrow \epsilon$ for aplicada, uma regra de cópia $M.s := M.i$ assegura que o valor de $M.s = M.i = A.s$ figure exatamente antes da parte da pilha usada para estruturar a subárvore para *C*. Por conseguinte, o valor de *C.i* pode ser encontrado em *val[topo-1]* quando $C \rightarrow c$ for aplicada, independentemente da primeira ou segunda produção estiver sendo usada, na seguinte modificação de 5.6.

PRODUÇÃO	REGRAS SEMÂNTICAS
$S \rightarrow aAC$	$C.i := A.s$
$S \rightarrow bABMC$	$M.i := A.s$; $C.i := M.s$
$C \rightarrow c$	$C.s := g(C.i)$
$M \rightarrow \epsilon$	$M.s := M.i$

Não-terminais marcadores podem também ser usados para simular regras semânticas que não sejam regras de cópia. Por exemplo, consideremos

PRODUÇÃO	REGRAS SEMÂNTICAS	
$S \rightarrow aAC$	$C.i := f(A.s)$	(5.7)

Desta vez, a regra que define *C.i* não é uma regra de cópia e, então, o valor de *C.i* ainda não está na pilha *val*. Este problema pode também ser resolvido usando-se um marcador.

PRODUÇÃO	REGRAS SEMÂNTICAS	
$S \rightarrow aANC$	$N.i := A.s$; $C.i := N.s$	
$N \rightarrow \epsilon$	$N.s := f(N.i)$	(5.8)

O não-terminal distinto *N* herda *A.s* através de uma regra de cópia. Seu atributo sintetizado *N.s* é estabelecido com o valor de *f(A.s)*; em seguida, *C.i* herda este valor usando uma regra de cópia. Ao reduzirmos através de $N \rightarrow \epsilon$, encontramos o valor de *N.i* no lugar de *A.s*, isto é, em *val[topo-1]*. Ao reduzirmos através de $S \rightarrow aANC$, o valor de *C.i* é também encontrado em *val[topo-1]*, porque é *N.s*. De fato, não necessitamos de *C.i* nesse momento; anteriormente sim, durante a redução de uma cadeia terminal para o não-terminal *C*, quando seu valor foi com segurança armazenado na pilha juntamente com *N*.

Exemplo 5.19. Três não-terminais marcadores, *L*, *M* e *N*, são usados na Fig. 5.36 para assegurar que o valor do atributo herdado *B.tp* apareça em uma posição conhecida na pilha do analisador sintático enquanto a subárvore para *B* estiver sendo reduzida. A gramática de atributos original aparece na Fig. 5.22 e sua relevância para a formatação de texto é explicada no Exemplo 5.13.

A inicialização é feita usando-se *L*. A produção para *S* é $S \rightarrow L\ B$ na Fig. 5.36 e, conseqüentemente, *L* permanecerá na pilha enquanto a subárvore abaixo de *B* for reduzida. O valor 10 do atributo herdado $B.tp = L.s$ é colocado na pilha do analisador sintático pela regra $L.s := 10$, associada à $L \rightarrow \epsilon$.

O marcador *M* em $B \rightarrow B_1\ M\ B_2$ desempenha um papel similar àquele de *M* na Fig. 5.35; assegura que o valor de *B.tp* apareça exatamente abaixo de B_2 na pilha do analisador sintático. Na produção $B \rightarrow B_1$ **sub** $N\ B_2$, o não-terminal *N* é usado como em (5.8). *N* herda, através da regra de cópia $N.i := B.tp$, o valor de atributo do qual B_2.tp depende e sintetiza o valor de $B_2.tp$ pela regra $N.s := comprimir\ (N.i)$. A conseqüência, que deixamos como um exercício, é que o valor de *B.tp* estará sempre imediatamente abaixo do lado direito quando reduzirmos para *B*.

Os fragmentos de código que implementam a definição dirigida pela sintaxe da Fig. 5.36 são mostrados na Fig. 5.37. Todos os atributos herdados são estabelecidos pelas regras de cópia da Fig. 5.36, e, por conseguinte, a implementação obtém seus valores controlando suas posições na pilha *val*. Como nos exemplos prévios, *topo* e *ntopo* forne-

PRODUÇÃO	REGRAS SEMÂNTICAS
$S \rightarrow L\ B$	$B.tp := L.s$
	$S.lg := B.lg$
$L \rightarrow \epsilon$	$L.s := 10$
$B \rightarrow B_1\ M\ B_2$	$B_1.tp := B.tp$
	$M.i := B.tp$
	$B_2.tp := M.s$
	$B.lg := max(B_1.lg, B_2.lg)$
$B \rightarrow B_1$ **sub** $N\ B_2$	$B_1.tp := B.tp$
	$N.i := B.tp$
	$B_2.tp := N.s$
	$B.lg := desloc(B_1.lg, B_2.lg)$
$B \rightarrow$ **texto**	$B.lg :=$ **texto**$.l \times B.tp$
$M \rightarrow \epsilon$	$M.s := M.i$
$N \rightarrow \epsilon$	$N.s := comprimir(N.i)$

Fig. 5.36. Todos os atributos herdados são estabelecidos usando-se regras de cópia.

PRODUÇÃO	FRAGMENTO DE CÓDIGO
$S \rightarrow L\ B$	$val[ntopo] := val[topo]$
$L \rightarrow \epsilon$	$val[ntopo] := 10$
$B \rightarrow B_1\ M\ B_2$	$val[ntopo] := max(val[topo\text{-}2], val[topo])$
$B \rightarrow B_1$ **sub** $N\ B_2$	$val[ntopo] := disp(val[topo\text{-}3], val[topo])$
$B \rightarrow$ **texto**	$val[ntopo] := val[topo] \times val[topo\text{-}1]$
$M \rightarrow \epsilon$	$val[ntopo] := val[topo\text{-}1]$
$N \rightarrow \epsilon$	$val[ntopo] := comprimir(val[topo\text{-}2])$

Fig. 5.37. Implementação da definição dirigida pela sintaxe na Fig. 5.36.

cem os índices do topo da pilha antes e depois de uma redução, respectivamente. □

A introdução sistemática de marcadores, como nas modificações de (5.6) e de (5.7), pode tornar possível avaliar definições L-atribuídas durante a análise sintática LR. Uma vez que existe somente uma produção para cada marcador, uma gramática se mantém LL(1) quando os marcadores são inseridos. Qualquer gramática LL(1) também é uma gramática LR(1) e nenhum conflito sintático emerge quando os marcadores são adicionados a uma gramática LL(1). Infelizmente, o mesmo não pode ser dito de todas as gramáticas LR(1); isto é, os conflitos sintáticos podem emergir se os marcadores forem introduzidos em certas gramáticas LR(1).

As idéias das seções precedentes podem ser formalizadas pelo seguinte algoritmo.

Algoritmo 5.3. Análise sintática e tradução *bottom-up* com atributos herdados.

Entrada. Uma definição L-atribuída com uma gramática LL(1) subjacente.

Saída. Um analisador sintático que compute os valores de todos os atributos na sua pilha sintática.

Método. Vamos assumir por simplicidade que cada não-terminal A possua um atributo herdado, $A.i$ e que cada símbolo gramatical X possua um atributo sintetizado $X.s$. Se X for um terminal, então seu atributo sintetizado é realmente o valor léxico retornado com X pelo analisador léxico; aquele valor léxico figurará na pilha, num *array val*, como nos exemplos anteriores.

Para cada produção $A \rightarrow X_1 \ldots X_n$, introduzir n novos não-terminais marcadores, $M_1, \ldots, M_n$ e substituir a produção por $A \rightarrow M_1X_1, \ldots M_nX_n$.[2] O atributo sintetizado $X_j.s$ irá para a pilha do analisador sintático na entrada do *array val* associada a X_j. O atributo herdado $X_j.i$, se existir um, aparece no mesmo *array*, mas associado a M_j.

Uma invariante importante é que, à medida que analisamos sintaticamente, o atributo $A.i$, se existir, será encontrado na posição do *array val* imediatamente abaixo da posição para M_1. Como assumimos que o símbolo de partida não possui atributo herdado, não há problema no caso do símbolo de partida ser A, mas, mesmo que houvesse um tal atributo herdado, o mesmo poderia ser colocado abaixo do fundo da pilha. Podemos provar a invariante através de uma fácil indução no número de passos da análise sintática *bottom-up*, notando o fato de que os atributos herdados estão associados aos não-terminais marcadores M_j e que o atributo $X_j.i$ é computado em M_j antes de começarmos a redução para X_j.

Para vermos como os atributos podem ser computados de acordo com o desejado durante a análise sintática *bottom-up*, consideremos dois casos. Primeiro, se reduzirmos a um não-terminal marcador M_j, sabemos a que produção $A \rightarrow M_1X_1 \ldots M_nX_n$ aquele marcador pertence. Conseqüentemente, sabemos as posições de quaisquer atributos que o atributo herdado $X_j.i$ necessita para sua computação. $A.i$ está em $val[topo-2j+2]$, $X_1.i$ está em $val[topo-2j+3]$, $X_1.s$ está em $val[topo-2j+4]$, $X_2.i$ está em $val[topo-2j+5]$, e assim por diante. Por conseguinte, podemos computar $X_j.i$ e armazená-lo em $val[topo+1]$, que se torna o novo símbolo de topo da pilha após a redução. Note-se como o fato da gramática ser LL(1) é importante, ou de outra forma não poderíamos estar certos de que estávamos reduzindo ϵ a um não-terminal marcador particular e, logicamente, não poderíamos localizar os atributos próprios ou mesmo saber, em geral, que fórmula aplicar. Pedimos ao leitor para que confie (ou derive uma prova) em que qualquer gramática LL(1) com marcadores é ainda LR (1).

O segundo caso ocorre quando reduzimos a um símbolo não marcador, digamos através da produção $A \rightarrow M_1X_1 \ldots M_nX_n$. Conseqüentemente, temos somente que computar o atributo sintetizado $A.s$; note-se que $A.i$ já foi computado e mora na posição da pilha exatamente abaixo da posição na qual inserimos o próprio A. Os atributos necessários para computar $A.s$ estão claramente disponíveis em posições conhecidas na pilha, às posições dos vários X_j's durante a redução.

As seguintes simplificações reduzem o número de marcadores; a segunda, em particular, evita conflitos sintáticos nas gramáticas recursivas à esquerda.

1. Se X_j não possui atributo herdado, não precisamos utilizar o marcador M_j. Naturalmente, as posições esperadas para os atributos na pilha irão mudar se M_j for omitido, mas essa mudança pode ser incorporada facilmente ao analisador sintático.
2. Se $X_1.i$ existir, mas for computado por uma regra de cópia $X_1.i = A.i$, então podemos omitir M_1, uma vez que sabemos através de nossa invariante que $A.i$ já estará localizado onde o queríamos, exatamente abaixo de X_1 na pilha, e esse valor pode subseqüentemente servir para X_1 igualmente. □

Substituindo Atributos Herdados por Sintetizados

É possível algumas vezes se evitar o uso de atributos herdados através da mudança da gramática subjacente. Por exemplo, uma declaração em Pascal pode consistir em uma lista de identificadores seguida por um tipo, por exemplo, `m, n : integer`. Uma gramática para tais declarações pode incluir produções da forma

$$D \rightarrow L : T$$
$$T \rightarrow \textbf{integer} \mid \textbf{char}$$
$$L \rightarrow L,\ \textbf{id}, \mid \textbf{id}$$

Como os identificadores são gerados por L mas o tipo não está na subárvore para L, não podemos associar o tipo a um identificador usando isoladamente os atributos sintetizados. De fato, se o não-terminal L herda um tipo a partir de T à sua direita na primeira produção, temos uma definição dirigida pela sintaxe que não é L-atribuída e a tradução baseada na mesma não pode ser realizada durante a análise sintática.

Uma solução para esse problema está em reestruturar a gramática de forma a incluir o tipo como o último elemento da lista de identificadores:

$$D \rightarrow \textbf{id}\ L$$
$$L \rightarrow \textbf{,}\ \textbf{id}\ L \mid \textbf{:}\ T$$
$$T \rightarrow \textbf{integer} \mid \textbf{char}$$

Agora, o tipo pode ser carregado junto como o atributo sintetizado $L.tipo$. À medida que cada identificador estiver sendo gerado por L, seu tipo pode ser colocado na tabela de símbolos.

Uma Difícil Definição Dirigida pela Sintaxe

O Algoritmo 5.3, para implementar atributos herdados durante a análise sintática *bottom-up*, se estende para algumas gramáticas LR, mas não todas. A definição L-atribuída na Fig. 5.38 está baseada numa gramá-

[2] Apesar da inserção de M_1 antes de X_1 simplificar a discussão dos não-terminais marcadores, possui o indesejável efeito colateral de introduzir conflitos sintáticos numa gramática recursiva à esquerda. Veja o Exercício 5.21. Como notado abaixo, M_1 pode ser eliminado.

PRODUÇÃO			REGRAS SEMÂNTICAS
S	$\rightarrow$	L	$L.cont := 0$
L	$\rightarrow$	$L_1\ 1$	$L_1.cont := L.cont + 1$
L	$\rightarrow$	ϵ	*imprimir (L.cont)*

Fig. 5.38. Uma difícil definição dirigida pela sintaxe.

tica LR(1) simples, mas não pode ser implementada tal como é numa análise sintática LR. O não-terminal L em $L \rightarrow \epsilon$ herda a contagem do número de 1's gerados por S. Como a produção $L \rightarrow \epsilon$ é a primeira que um analisador sintático *bottom-up* reduziria através, o tradutor àquela altura não pode saber o número de 1's na entrada.

5.7 AVALIADORES RECURSIVOS

Funções recursivas, que avaliam atributos à medida que realizam uma travessia numa árvore gramatical, podem ser construídas a partir de uma definição dirigida pela sintaxe usando-se uma generalização das técnicas para a tradução preditiva na Seção 5.5. Tais funções nos permitem implementar definições dirigidas pela sintaxe que não podem ser implementadas simultaneamente com a análise sintática. Nesta seção, associamos uma única função de tradução a cada não-terminal. A função visita os filhos de um nó para um não-terminal em alguma ordem determinada pela produção para o nó; não é necessário que os filhos sejam visitados numa ordem da esquerda para a direita. Na Seção 5.10, veremos que o efeito da tradução em mais de uma passagem pode ser simulado através da associação de múltiplos procedimentos aos não-terminais.

Travessias da Esquerda para a Direita

No Algoritmo 5.2, mostramos como uma definição L-atribuída, baseada numa gramática LL(1), pode ser implementada através da construção de uma função recursiva que analisa sintaticamente e traduz cada não-terminal. Todas as definições L-atribuídas, dirigidas pela sintaxe, podem ser implementadas se uma função recursiva similar for invocada a cada nó para aquele não-terminal, numa árvore gramatical previamente construída. Através do exame da produção àquele nó, a função pode determinar o que são seus filhos. A função para um não-terminal A toma um nó e os valores dos atributos herdados para A como argumentos e, como resultado, retorna os valores dos atributos sintetizados para A.

Os detalhes da construção são exatamente como no Algoritmo 5.2, exceto para o passo 2, onde a função para um não-terminal decide que produção usar baseada no símbolo corrente de entrada. A função neste ponto emprega um enunciado *case** para determinar a produção usada a um nó. Damos um exemplo para ilustrar o método.

Exemplo 5.20. Consideremos a definição dirigida pela sintaxe para determinar o tamanho e a largura de fórmulas na Fig. 5.22. O não-terminal B possui um atributo herdado tp e um atributo sintetizado lg. Usando-se o Algoritmo 5.2, modificado como mencionado acima, construímos a função para B mostrada na Fig. 5.39.

A função B toma, como argumentos, um nó n e o valor correspondente a $B.tp$ àquele nó e retorna o valor correspondente a $B.lg$ ao nó n. A função possui um enunciado *case* para cada produção com B à esquerda. O código correspondente a cada produção simula as regras semânticas associadas à produção. A ordem na qual as regras são

```
função B(n, tp);
    var tp1, tp2, lg1, lg2;
início
    caso a produção ao nó n seja
    'B → B1B2':
        tp1 : = tp;
        lg1 : = B (filho(n, 1), tp1);
        tp2 : = tp;
        lg2 : = B (filho(n, 2), tp2);
        retornar max(lg1, lg2);
    'B → B1 sub B2':
        tp1 : = tp;
        lg1 : = B (filho (n, 1), tp1);
        tp2 : = comprimir (tp);
        lg2 : = B (filho (n, 3), tp2);
        retornar desloc(lg1, lg2);
    'B → texto':
        retornar tp × texto.l;
    default:
        erro
    fim
fim;
```

Fig. 5.39. Função para o não-terminal B na Fig. 5.22.

aplicadas precisa ser tal que os atributos herdados de um não-terminal sejam computados antes que a função para o não-terminal seja chamada.

No código correspondente à produção $B \rightarrow B$ **sub** B, as variáveis tp, $tp1$ e $tp2$ guardam os valores dos atributos herdados $B.tp$, $B_1.tp$ e $B_2.tp$. Analogamente, lg, $lg1$ e $lg2$ guardam os valores de $B.lg$, $B_1.lg$ e $B_2.lg$. Usamos a função *filho (m, i)* para referirmo-nos ao *i*ésimo filho do nó m. Como B_2 é o rótulo do terceiro filho do nó m, o valor de $B_2.lg$ é determinado pela chamada de função *B(filho(n, 3), tp2)*. □

Outras Travessias

Uma vez que uma árvore gramatical explícita está disponível, estamos livres para visitar os filhos de um nó em qualquer ordem. Consideremos a definição não L-atribuída do Exemplo 5.21. Numa tradução especificada por aquela definição, os filhos de um nó para uma produção precisam ser visitados da esquerda para a direita, enquanto que os filhos de um nó para a outra produção precisam ser visitados da direita para a esquerda.

PRODUÇÃO			REGRAS SEMÂNTICAS
A	$\rightarrow$	$L\ M$	$L.i := l(A.i)$ $M.i := m(L.s)$ $A.s := f(M.s)$
A	$\rightarrow$	$Q\ R$	$R.i := r(A.i)$ $Q.i := q(R.s)$ $A.s := f(Q.s)$

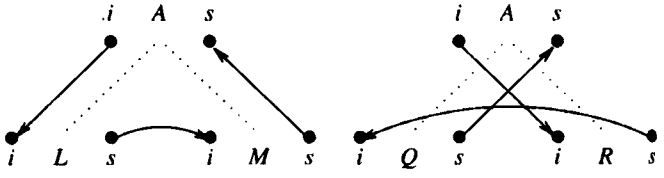


Fig. 5.40. Produções e regras semânticas para o não-terminal A.

*Traduzido em nosso pseudocódigo pela estrutura **caso...seja**, com a acepção usual dessa estrutura de controle. (N. do T.)

```
função A(n, ai);
início
    caso a produção ao nó n seja
    'A → L M':    / * ordem da esquerda para a direita * /
        li : = l(ai);
        ls : = L(filho(n, 1), li);
        mi : = m(ls ;
        ms : = M(filho(n, 2), mi)
        retornar f(ms);
    'A → Q R':    / * ordem da direita para a esquerda * /
        ri : = r(ai);
        rs : = R(filho(n, 2), ri);
        qi : = q(rs);
        qs : = Q(filho(n, 1), qi);
        retornar f(qs);
    default:
        erro
    fim
fim;
```

Fig. 5.41. As dependências na Fig. 5.40 determinam a ordem dos filhos visitados.

O exemplo abstrato ilustra o poder de se usar funções mutuamente recursivas para avaliar os atributos aos nós de uma árvore sintática. As funções não precisam depender da ordem na qual os nós da árvore gramatical são criados. A consideração principal para a avaliação durante uma travessia é a de que os atributos herdados a um nó sejam computados antes que o nó seja primeiro visitado e que os atributos sintetizados sejam computados antes que deixemos o nó pela última vez.

Exemplo 5.21. Cada um dos não-terminais na Fig. 5.40 possui um atributo herdado i e um sintetizado s. Os grafos de dependências para as duas produções também são mostrados. As regras associadas a $A \rightarrow LM$ estabelecem dependências da esquerda para a direita enquanto que as associadas a $A \rightarrow Q\ R$ estabelecem-nas da direita para a esquerda.

A função para o não-terminal A é mostrada na Fig. 5.41; assumimos que as funções para L, M, Q e R possam ser construídas. As variáveis na Fig. 5.41 são nomeadas após o não-terminal e seu atributo; por exemplo, li e ls são as variáveis correspondentes a $L.i$ e $L.s$.

O código correspondente à produção $A \rightarrow L\ M$ é construído como no Exemplo 5.20. Isto é, determinamos o atributo herdado de L, chamamos a função para L a fim de determinar o atributo sintetizado do mesmo, e repetimos o processo para M. O código correspondente a $A \rightarrow Q\ R$ visita a subárvore para R antes que visite a subárvore para Q. Nos demais casos, o código para as duas produções é muito similar. □

PRODUÇÃO	REGRAS
$D \rightarrow T\ L$	$L.in := T.tipo$
$T \rightarrow$ **int**	$T.tipo :=$ *inteiro*
$T \rightarrow$ **real**	$T.tipo :=$ *real*
$L \rightarrow L_1$, I	$L_1.in := L.in$
	$I.in := L.in$
$L \rightarrow I$	$I.in := L.in$
$I \rightarrow I_1$ [**num**]	$I_1.in :=$ *array* (**num.** *val*, $I.in$)
$I \rightarrow$ **id**	*incluir_tipo* (**id.** *entrada*, $I.in$)

Fig. 5.42. Passando o tipo para os identificadores numa declaração.

5.8 ESPAÇO PARA OS VALORES DE ATRIBUTOS EM TEMPO DE COMPILAÇÃO

Nesta seção, consideramos a atribuição de espaço em tempo de compilação para os valores de atributos. Iremos usar as informações provenientes do grafo de dependências para uma árvore gramatical, e, por conseguinte, esta seção é adequada aos métodos que, utilizando árvores gramaticais, determinam a ordem de avaliação a partir dos grafos de dependências. Na próxima seção, consideramos o caso em que a ordem de avaliação pode ser prevista antecipadamente, de forma a decidirmos a respeito do espaço para os atributos uma vez e para todo sempre quando o compilador for construído.

Dada uma ordem de avaliação para os atributos, não necessariamente de pesquisa em profundidade, o *tempo de vida* de um atributo começa quando o mesmo é primeiramente computado e termina quando todos os atributos que dele dependam o tenham sido igualmente. Podemos economizar espaço guardando o valor de um atributo somente durante o seu tempo de vida.

Para enfatizar que as técnicas desta seção se aplicam a qualquer ordem de avaliação, iremos considerar a seguinte definição dirigida pela sintaxe, não L-atribuída, para transmitir a informação de tipo de dados aos identificadores numa declaração.

Exemplo 5.22. A definição dirigida pela sintaxe na Fig. 5.42 é uma extensão daquela da Fig. 5.4 a fim de permitir declarações da forma

```
real c[12][31];        (5.9)
int x[3], y[5];        (5.10)
```

Uma árvore gramatical para (5.10) é mostrada pelas linhas pontilhadas na Fig. 5.43(a). Os números aos nós são discutidos no próximo exem-

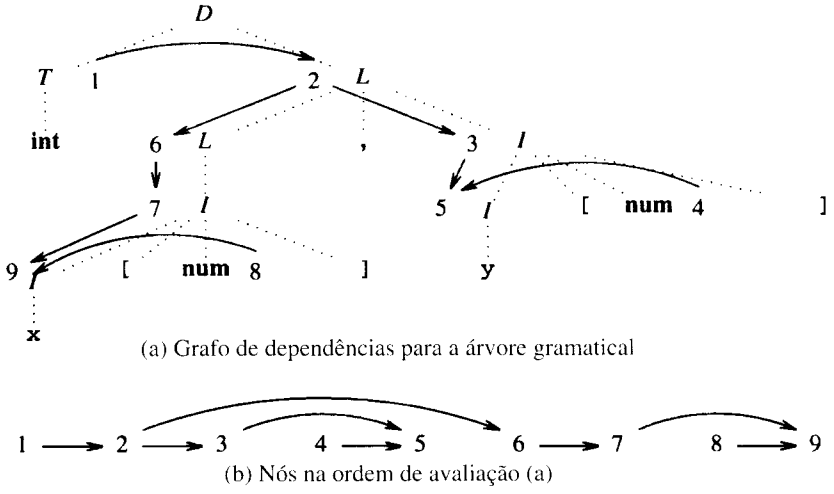


Fig. 5.43. Determinação dos tempos de vida dos valores de atributos.

para cada nó $m_1, m_2, \ldots, m_n$ **faça início**
  para cada nó n cujo tempo de vida termine com a avaliação m **faça**
    marcar o registrador de n;
  se algum registrador r estiver marcado **então início**
    desmarcar r;
    avaliar m no registrador r;
    retornar os registradores marcados para o *pool;*
  fim
  se não / * nenhum registrador foi marcado * /
    avaliar m num registrador proveniente do *pool;*
  / * as ações usando o valor de m podem ser inseridas aqui * /
  se o tempo devida de m terminou **então**
    retornar o registrado de m para o *pool*
fim

Fig. 5.44. Atribuindo valores de atributos a registradores.

plo. Como na Fig. 5.3, o tipo obtido a partir de T é herdado por L e transmitido abaixo para os identificadores na declaração. Um lado a partir de *T.tipo* para *L.in* mostra que *L.in* depende de *T.tipo*. A definição dirigida pela sintaxe na Fig. 5.42 não é L-atribuída porque I_1.*in* depende de **num**.*val* e **num**.*val* está à direita de I_1 em $I \rightarrow I_1$ [**num**]. □

Reservando Espaço para os Atributos em Tempo de Compilação

Suponhamos que seja dada uma seqüência de registradores para guardar valores de atributos. Por conveniência, assumimos que cada registrador possa guardar qualquer valor de atributo. Se os atributos representarem tipos diferentes, podemos formar grupos de atributos que ocupem a mesma memória e considerar cada grupo separadamente. Confiamos nas informações sobre os tempos de vida dos atributos para determinar os registradores nos quais serão avaliados.

Exemplo 5.23. Suponhamos que os atributos sejam avaliados na ordem dada pelos números de nó no grafo de dependências da Fig. 5.43,[3] construído no último exemplo. O tempo de vida de cada nó começa quando o atributo for avaliado e termina quando for usado pela última vez. Por exemplo, o tempo de vida do nó 1 termina quando o 2 for avaliado, porque 2 é o único nó que depende de 1. O tempo de vida de 2 termina quando 6 for avaliado. □

Um método para avaliar os atributos que usa tão poucos registradores quanto possível é fornecido na Fig. 5.44. Consideramos os nós do grafo de dependências D para uma árvore gramatical na ordem em que devem ser avaliados. Inicialmente, temos um *pool* de registradores $r_1, r_2, \ldots$ Se o atributo b for definido pela regra semântica $b := f(c_1, c_2, \ldots, c_k)$, o tempo de vida de um ou mais dos $c_1, c_2, \ldots, c_k$ poderia terminar com a avaliação de b; os registradores que guardam tais atributos são retornados ao *pool* após b ter sido avaliado. Sempre que possível, b é avaliado num registrador que detinha um dos $c_1, c_2, \ldots, c_k$.

Os registradores usados durante a avaliação do grafo de dependências da Fig. 5.43 são mostrados na Fig. 5.45. Começamos pela avaliação do nó no registrador r_1. O tempo de vida do nó 1 termina quando o 2 for avaliado e, por conseguinte, 2 é avaliado em r_1. O nó 3 obtém um registrador novo r_2, porque o nó 6 irá necessitar do valor de 2.

Evitando Cópias

Podemos melhorar o método da Fig. 5.44 tratando as regras de cópia como um caso especial. Uma regra de cópia possui a forma $b := c$, e, conseqüentemente, se o valor de c estiver no registrador r, o valor de b já aparece em r. O número de atributos definidos pelas regras de cópia pode ser significativo e, dessa forma, desejamos evitar a realização de cópias explícitas.

Um conjunto de nós tendo o mesmo valor forma uma classe de equivalência. O método da Fig. 5.44 pode ser modificado como se segue, a fim de abrigar o valor de uma classe de equivalência num registrador. Quando o nó m for considerado, verificamos primeiro se está definido por alguma regra de cópia. Se o estiver, seu valor já estará num registrador e m se une à classe de equivalência com valores naquele registrador. Sobretudo, um registrador é retornado ao *pool* somente ao fim dos tempos de vida de todos os nós com valores naquele registrador.

Exemplo 5.24. O grafo de dependências na Fig. 5.43 é redesenhado na Fig. 5.46, com um sinal de igual antes de cada nó definido por uma regra de cópia. A partir da definição dirigida pela sintaxe da Fig. 5.42, encontramos que o tipo determinado ao nó 1 é copiado a cada elemento na lista de identificadores, resultando em que os nós 2, 3, 6 e 7 da Fig. 5.43 sejam cópias de 1.

Uma vez que 2 e 3 são cópias de 1, seus valores são obtidos a partir do registrador r_1 na Fig. 5.46. Note-se o tempo de vida de 3 termina quando 5 for avaliado, mas o registrador r_1, que abriga o valor de 3, não é retornado ao *pool* porque o tempo de vida de 2 na sua classe de equivalência não terminou ainda.

O seguinte código mostra como a declaração (5.10) do Exemplo 5.22 poderia ser processada por um compilador:

```
r1 := inteiro;                  /* avalia os nós 1, 2, 3, 6 e 7 */
r2 := 5;                        /* avalia o nó 4 */
r2 := array (r2, r1);           /* tipo de y */
incluir_tipo (y, r2);
r2 := 3;                        /* avalia nó 8 */
r2 := array (r2, r1);           /* tipo de x */
incluir_tipo (x, r2);
```

Acima, x e y apontam para entradas da tabela de símbolos para `x` e `y` e o procedimento *incluir_tipo* precisa ser chamado nos pontos apropriados de forma a adicionar os tipos de `x` e de `y` às suas entradas na tabela de símbolos. □

5.9 RESERVA DE ESPAÇO EM TEMPO DE CONSTRUÇÃO DO COMPILADOR

Apesar de ser possível guardar valores numa única pilha durante uma travessia, podemos algumas vezes evitar a realização de cópias através da utilização de múltiplas pilhas. Em geral, se as dependências entre os atributos tornam inconveniente colocar certos valores de atributos numa pilha, podemos guardá-los nos nós, numa árvore sintática explicitamente construída.

[3] O grafo de dependências da Fig. 5.43 não mostra os nós correspondentes à regra semântica *incluir_tipo*(**id**.*entrada*, *I.in*) porque nenhum espaço é reservado para atributos fictícios. Note-se, entretanto, que esta regra semântica não deve ser avaliada até que o valor de *I.in* esteja disponível. Um algoritmo, para determinar este fato, precisa trabalhar com um grafo de dependências contendo nós para esta regra semântica.

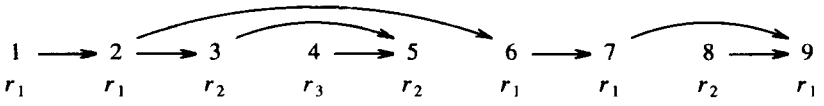


Fig. 5.45. Registradores usados para valores de atributos na Fig. 5.43.

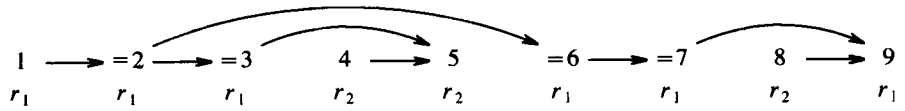


Fig. 5.46. Registradores usados, levando em conta as regras de cópia.

Já examinamos o uso de uma pilha para guardar valores de atributos durante a análise sintática *bottom-up* nas Seções 5.3 e 5.6. Uma pilha também é usada implicitamente através de um analisador sintático de descendência recursiva para o controle de chamadas de procedimentos; este tema será discutido no Capítulo 7.

O uso de uma pilha pode ser combinado com outras técnicas para economizar espaço. As ações de impressão usadas extensivamente nos esquemas de tradução do Capítulo 2 emitem atributos com valores do tipo cadeia de caracteres para um arquivo de saída sempre que possível. Enquanto construíamos árvores sintáticas na Seção 5.2, passávamos apontadores para os nós, em lugar de subárvores completas. Em geral, ao invés de passar objetos extensos, podemos ganhar espaço transmitindo apontadores para os mesmos. Essas técnicas serão aplicadas nos Exemplos 5.27 e 5.28.

Predizendo os Tempos de Vida a Partir da Gramática

Quando a ordem de avaliação para os atributos é obtida a partir de uma travessia particular de uma árvore gramatical, podemos predizer os tempos de vida dos atributos em tempo de construção do compilador. Por exemplo, suponhamos que os filhos sejam visitados da esquerda para a direita durante uma travessia em profundidade, como na Seção 5.4. Começando pelo nó para a produção $A \rightarrow B\ C$, a subárvore para B é visitada, a subárvore para C é visitada e retornamos ao nó para A. O pai de A não pode se referir aos atributos de B e de C e, por conseguinte, seus tempos de vida têm que terminar quando retornamos para A. Note-se que essas observações estão baseadas na produção $A \rightarrow B\ C$ e na ordem pela qual os nós para esses terminais são visitados. Nada precisamos saber a respeito das subárvores em B e C.

Com qualquer ordem de avaliação, se o tempo de vida do atributo c estiver contido naquele de b, então o valor de c pode ser guardado numa pilha abaixo do valor de b. Aqui, b e c não têm que ser atributos para o mesmo não-terminal. Para a produção $A \rightarrow B\ C$, podemos usar uma pilha durante uma travessia em profundidade da seguinte forma.

Começamos pelo nó para A com os atributos herdados de A que já estejam na pilha. Em seguida, avaliamos e empilhamos os valores dos atributos herdados de B. Esses atributos permanecem na pilha à medida que caminhamos na subárvore de B, retornando com os atributos sintetizados de b acima deles. Este processo é repetido com C; isto é, empilhamos seus atributos herdados, percorremos sua subárvore e retornamos com seus atributos sintetizados ao topo. Usando $\mathbf{H}(X)$ e $\mathbf{S}(X)$ para os atributos herdados e sintetizados de X, respectivamente, a pilha agora contém.

$$\mathbf{H}(A),\ \mathbf{H}(B),\ \mathbf{S}(B),\ \mathbf{H}(C),\ \mathbf{S}(C) \tag{5.11}$$

Todos os valores de atributos necessitados para computar os atributos sintetizados de A estão agora na pilha, e podemos, então, retornar para A com a pilha contendo

$$\mathbf{H}(A),\ \mathbf{S}(A)$$

Note-se que o número (e presumivelmente o tamanho) dos atributos herdados e sintetizados de um símbolo gramatical é fixo. Por conseguinte, em cada passo do processo acima, sabemos o quanto teremos que atingir abaixo na pilha para encontrar um atributo.

Exemplo 5.25. Suponhamos que os valores de atributos para a tradução da composição de tipos do Exemplo 5.22 sejam guardados numa pilha, como discutido acima. Começando pelo nó para a produção $B \rightarrow B_1\ B_2$ com $B.tp$ no topo da pilha, os conteúdos da pilha antes e após a visita a um nó são mostrados na Fig. 5.47, à esquerda e à direita do nó, respectivamente. Como de praxe, a pilha cresce para baixo.

Note-se que, exatamente antes de um nó para um não-terminal B ser visitado pela primeira vez, seu atributo tp está ao topo da pilha. Exatamente antes da última visita, isto é, quando o percurso se dirigir de volta para aquele nó, seus atributos lg e tp estarão nas duas posições ao topo da pilha. □

Quando um atributo b é definido por uma regra de cópia $b := c$ e o valor de c está ao topo da pilha de valores de atributos, pode não ser necessário empilhar uma cópia de c na pilha do analisador sintático. Podem haver mais oportunidades para se eliminar regras de cópia, se mais de uma pilha for usada para abrigar valores de atributos. No próximo exemplo, usamos pilhas distintas para os atributos sintetizados e herdados. Uma comparação com o Exemplo 5.25 nos mostra que regras de cópia adicionais podem ser eliminadas, se forem usadas pilhas separadas.

Exemplo 5.26. Com a definição dirigida pela sintaxe da Fig. 5.22, suponhamos que usemos pilhas separadas para o atributo herdado tp e para o atributo sintetizado lg. Mantemos as pilhas de tal forma que $B.tp$

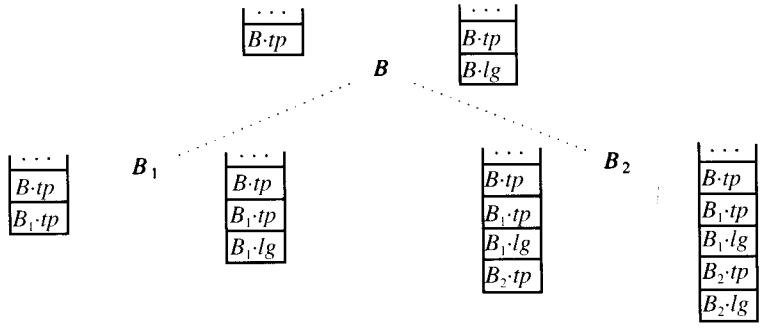


Fig. 5.47. Conteúdo da pilha antes e após a visita a um nó.

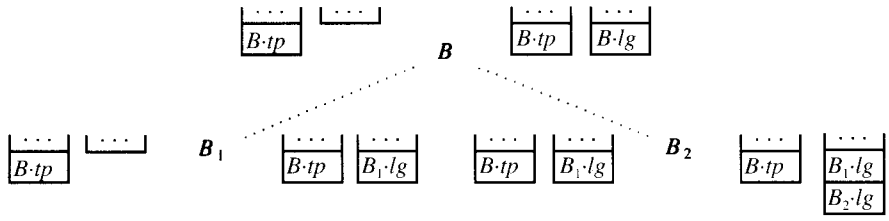


Fig. 5.48. Usando pilhas separadas para os atributos *tp* e *lg*.

estará no topo da pilha de *tp* exatamente antes de *B* ser primeiramente visitado e exatamente após o ser por último. *B.lg* estará ao topo da pilha para *lg* exatamente antes de *B* ser visitado.

Com pilhas separadas, podemos nos beneficiar de ambas as regras de cópia $B_1.tp := B.tp$ e $B_2.tp := B.tp$ associadas a $B \rightarrow B_1\ B_2$. Como mostrado na Fig. 5.48, não precisamos empilhar $B_1.tp$ porque seu valor já está no topo da pilha como *B.tp*.

Um esquema de tradução baseado na definição dirigida pela sintaxe da Fig. 5.22 é mostrado na Fig. 5.49. A operação *empilhar (v, p)*, empilha o valor *v* na pilha *p* e *desempilhar(p)* desempilha o valor no topo da pilha *p*. Usamos *topo(p)* para referirmo-nos ao elemento de topo da pilha *p*.

O próximo exemplo combina o uso de uma pilha para valores de atributos com as ações para emitir código.

Exemplo 5.27. Aqui, consideramos as técnicas para implementar uma definição dirigida pela sintaxe especificando a geração de código intermediário. O valor de uma expressão *booleana E* **e** *F* é falso se *E* for falso. Em *C*, a subexpressão *F* não precisa ser avaliada se *E* for falso. A avaliação de uma tal expressão *booleana* é considerada na Seção 8.4.

As expressões *booleanas* na definição dirigida pela sintaxe na Fig. 5.50 são construídas a partir de identificadores e do operador **e.** Cada expressão *E* herda dois rótulos, *E.v* e *E.f*, marcando os pontos para os quais o controle deve desviar se *E* for verdadeiro ou falso, respectivamente.

Suponhamos que $E \rightarrow E_1$ e E_2. Se E_1 é avaliado como falso, o controle flui para o rótulo herdado *E.f*; em caso contrário, E_1 é avaliado como verdadeiro e o controle flui para o código que avalia E_2. Um novo rótulo gerado pela função *criar_rótulo* marca o início do código para E_2. As instruções individuais são formadas usando *gerar*. Para uma discussão posterior da relevância da Fig. 5.50 para a geração de código intermediário ver a Seção 8.4.

A definição dirigida pela sintaxe da Fig. 5.50 é L-atribuída, e, por conseguinte, podemos construir um esquema de tradução para a mesma. O esquema de tradução da Fig. 5.51 usa o procedimento *emitir* de forma a gerar e emitir instruções incrementalmente. Também mostradas na figura são as ações para o estabelecimento dos valores dos atributos herdados, inseridos antes dos símbolos gramaticais apropriados, como discutido na Seção 5.4.

$S \rightarrow$		{*empilhar* (10, *tp*) }
	B	
$B \rightarrow$	B_1	
	B_2	{ *l2* : = *topo* (*lg*); *desempilhar*(*lg*); *l1* : = *topo* (*lg*); *desempilhar* (*lg*); *empilhar* (*max*(*l1*, *l2*), *lg*) }
$B \rightarrow$	B_1	
	sub	{*empilhar*(*comprimir*(*topo*(*tp*)), *tp*) }
	B_2	{*desempilhar* (*tp*); *l2* : = *topo*(*lg*); *desempilhar* (*lg*); *l1* : = *topo*(*lg*); *desempilhar* (*lg*); *empilhar* (*desloc*(*l1*, *l2*), *lg*) }
$B \rightarrow$	**texto**	{*empilhar* (**texto.**l × *topo*(*tp*), *lg*) }

Fig. 5.49. Esquema de tradução mantendo as pilhas *tp* e *lg*.

O esquema de tradução da Fig. 5.52 vai adiante; utiliza-se de pilhas separadas para guardar os valores dos atributos herdados *E.v* e *E.f*. Como no Exemplo 5.26, as regras de cópia não possuem efeito sobre as pilhas. Para implementar a regra $E_1.v$:= *criar_rótulo*, um novo rótulo é empilhado na pilha *v* antes de E_1 ser visitado. O tempo de vida desse rótulo termina com a ação *emitir* ('**rótulo**' *topo(v)*), correspondente a *emitir* ('**rótulo**' $E_1.v$) e conseqüentemente a pilha *v* tem o seu topo removido após a ação. A pilha *f* não muda neste exemplo, mas é necessária quando o operador **ou** é adicionalmente permitido figurar juntamente com o operador **e**. □

Tempos de Vida Não-Superpostos

Um único registrador é um caso especial de pilha. Se cada operação de empilhar for seguida por uma operação de desempilhar, poderá haver, no máximo, a cada vez, um único elemento na pilha. Nesse caso, podemos usar um registrador no lugar de uma pilha. Em termos de tempos de vida, se os tempos de vida de dois atributos não se superpuserem, seus valores podem ser guardados no mesmo registrador.

Exemplo 5.28. A definição dirigida pela sintaxe da Fig. 5.53 constrói árvores sintáticas para expressões do tipo lista, com operadores a um único nível de precedência. É obtida a partir do esquema de tradução da Fig. 5.28.

Afirmamos que o tempo de vida de cada atributo de *R* termina quando o atributo que depende dele é avaliado. Podemos mostrar que, para qualquer árvore gramatical, os atributos de *R* podem ser avaliados no mesmo registrador *r*. A seguinte argumentação é típica daquela necessitada para analisar gramáticas. A indução é feita no tamanho da subárvore atrelada a *R*, no fragmento de árvore gramatical da Fig. 5.54.

A menor subárvore é obtida se $R \rightarrow \epsilon$ for aplicada, caso em que *R.s* é uma cópia de *R.i*, e, por conseguinte, os dois valores estão no registrador *r*. Para uma subárvore maior, a produção à raiz da subárvore terá que ser para $R \rightarrow$ **op_aditivo** $T\ R_1$. O tempo de vida de vida de *R.i* termina quando $R_1.i$ é avaliado, e, então, $R_1.i$ pode ser avaliado no registrador *r*. A partir da hipótese indutiva, todos os atributos para as instâncias do não-terminal *R* na subárvore para R_1 podem ser atribuídos ao mesmo registrador. Finalmente, *R.s* é uma cópia de $R_1.s$ e, conseqüentemente, seu valor já está em *r*.

PRODUÇÃO	REGRAS SEMÂNTICAS
$E \rightarrow E_1$ **and** E_2	$E_1.v$:= *novo_rótulo;* $E_1.f$:= $E.f$ $E_2.v$:= $E.v$ $E_2.f$:= $E.f$ *E*.código := E_1·*código* ‖ *gerar* ('**rótulo**' $E_1.v$) ‖ E_2·*código*
$E \rightarrow$ **id**	*E.código* := *gerar* ('**se**' **id**.*loc* '**ir_para**' *E.v*) ‖ *gerar* ('**ir_para**' *E.f*)

Fig. 5.50. Avaliação de expressões *booleanas* em curto-circuito.

$E \rightarrow$ $\{E_1.v := criar_rótulo;$
$E_1.f := E \cdot f\}$
E_1
e $\{emitir$ ('**rótulo**' $E_1.v$);
$E_2.v := E.v;$
$E_2.f := E.f\}$
E_2
$E \rightarrow$ **id** $\{emitir$ ('**se**' **id**.*loc* }**ir_para**'
$E.v$);
$emitir$ ('**ir_para**' $E.f$) }

Fig. 5.51. Emitindo código para expressões *booleanas*.

PRODUÇÃO	REGRAS SEMÂNTICAS
$E \rightarrow T\,R$	$R.i := T.nptr$ $E.nptr := R.s$
$R \rightarrow$ **op_aditivo** $T\,R_1$	$R.i :=$ *criar_nó* (**op_aditivo**. *lexema, R.i., T.nptr*) $R.s := R_1.s$
$R \rightarrow \epsilon$	$R.s := R.i$
$T \rightarrow$ **num**	$T.nptr :=$ *criar_folha* (**num**, **num**.*val*)

Fig. 5.53. Uma definição dirigida pela sintaxe adaptada a partir da Fig. 5.28.

O esquema de tradução na Fig. 5.55 avalia os atributos na gramática de atributos da Fig. 5.53, usando o registrador r para guardar os valores dos atributos $R.i$ e $R.s$ para todas as instâncias do não-terminal R.

Para completar, mostramos na Fig. 5.56 o código para a implementação do esquema acima; é construído de acordo com o Algoritmo 5.2. O não-terminal R já não possui mais atributos, e R se transformou num procedimento em lugar de uma função. A variável r foi tornada local à função E possibilitando que E seja chamada recursivamente, apesar de não necessitarmos fazê-lo no esquema da Fig. 5.55. Este código pode ser aprimorado adicionalmente pela eliminação da epílogo-recursividade e substituição da chamada remanescente de R pelo próprio corpo do procedimento resultante, como na Seção 2.5. □

5.10 ANÁLISE DE DEFINIÇÕES DIRIGIDAS PELA SINTAXE

Na Seção 5.7, os atributos eram avaliados durante uma travessia de uma árvore, usando um conjunto de funções mutuamente recursivas. As funções para um não-terminal mapeavam os valores dos atributos herdados a um nó nos valores dos atributos sintetizados àquele nó.

O enfoque da Seção 5.7 se estende a traduções que não podem ser realizadas durante um única travessia em profundidade. Aqui iremos usar uma função separada para cada atributo sintetizado de cada não-terminal, apesar de grupos de atributos sintetizados poderem ser avaliados por uma única função. A construção na Seção 5.7 lida com o caso especial no qual todos os atributos sintetizados formam um grupo. O agrupamento dos atributos é determinado a partir das regras semânticas numa definição dirigida pela sintaxe. O seguinte exemplo abstrato ilustra a construção de um avaliador recursivo.

Exemplo 5.29. A definição dirigida pela sintaxe na Fig. 5.57 é motivada por um problema que iremos considerar no Capítulo 6. Resumidamente, o problema é como segue. Um identificador "sobrecarregado" pode ter um conjunto de possíveis tipos; como resultado, uma expressão pode ter um conjunto de tipos possíveis. Informações contextuais são usadas para selecionar um dos possíveis tipos para cada subexpressão. O problema pode ser resolvido realizando-se uma passagem *bottom-up* para sintetizar o conjunto de tipos possíveis, seguida de uma passagem *top-down* para afunilar o conjunto num único tipo.

As regras semânticas na Fig. 5.57 são uma abstração desse problema. O atributo sintetizado s na Fig. 5.57 representa o conjunto de possíveis tipos do atributo herdado i, a informação contextual. Um atributo adicional sintetizado t, que não pode ser avaliado na mesma passagem que s, poderia representar o código gerado ou o tipo selecionado para uma subexpressão. Os grafos de dependências para as produções na Fig. 5.57 são mostrados na Fig. 5.58. □

Avaliação Recursiva de Atributos

O grafo de dependências para uma árvore gramatical é formado encaixando-se grafos menores correspondentes às regras semânticas para uma produção. O grafo de dependências D_p para a produção p é baseado somente nas regras semânticas para uma única produção, isto é, nas regras semânticas para os atributos sintetizados do lado esquerdo e para os atributos herdados dos símbolos gramaticais do lado direito da produção. Isto é, o grafo D_p mostra as dependências locais somente. Por exemplo, todos os lados do grafo de dependências para $E \rightarrow E_1\,E_2$ na Fig. 5.58 são entre instâncias do mesmo atributo. A partir deste grafo de dependências, não podemos dizer se os atributos s precisam ser computados antes de outros atributos.

Um exame mais apurado do grafo de dependências para a árvore gramatical da Fig. 5.59 mostra que os atributos de cada instância do não-terminal E precisa ser avaliada na ordem $E.s$, $E.i$, $E.t$. Note-se que todos os atributos da Fig. 5.59 podem ser avaliados em três passagens; uma passagem *bottom-up* para avaliar os atributos s, uma passagem *top-down* para avaliar os atributos i e uma passagem *bottom-up* final para avaliar os atributos t.

Num avaliador recursivo, a função para um atributo sintetizado toma os valores de alguns dos atributos herdados como parâmetros. Em geral, se o atributo sintetizado $A.a$ pode depender do atributo herdado $A.b$, então a função para $A.a$ toma $A.b$ como parâmetro. Antes de analisar as dependências, consideramos um exemplo mostrando seus usos.

Exemplo 5.30. As funções Es e Et na Fig. 5.60 retornam os valores dos atributos sintetizados s e t a um nó n rotulado E. Como na Seção 5.7, existe um enunciado *case* para cada produção na função para um não-terminal. O código executado em cada enunciado *case* simula as regras semânticas associadas à produção na Fig. 5.57.

Da discussão acima a respeito do grafo de dependências na Fig. 5.59 sabemos que o atributo $E.t$ a um nó numa árvore gramatical pode depender de $E.i$. Conseqüentemente, passamos o atributo herdado i como parâmetro para a função Et que é associada ao atributo t. Como

$E \rightarrow$ $\{empilhar$ ($criar_rótulo$, v) }
E_1
e $\{emitir$ ('**rótulo**' $topo$ (v));
$desempilhar$ (v) }
E_2
$E \rightarrow$ **id** $\{emitir$ ('**se**' **id**.*loc* '**ir_para**'
$topo$ (v));
$emitir$ ('**ir_para**' $topo$ (f)) }

Fig. 5.52. Emitindo código para expressões *booleanas*.

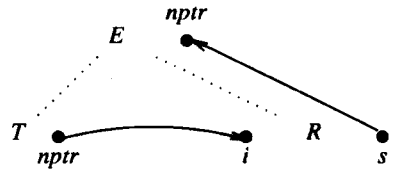


Fig. 5.54. Grafo de dependência para $E \rightarrow T\,R$.

$E \rightarrow$ T {r : = $T.nptr$ / * r agora contém $R.i$ * /}
R {E, $nptr$: = r / * r retornou com $R.s$ * /}
$R \rightarrow$ **op_aditivo**
T {r : = *criar_nó* (**op-aditivo.***lexema*, r, $T.nptr$) }
R
$R \rightarrow$ ϵ
$T \rightarrow$ **num** {$T.nptr$: = *criar_folha* (**num, num.***val*) }

Fig. 5.55. Esquema de tradução transformado para construir árvores sintáticas.

o atributo $E.s$ não depende de quaisquer atributos herdados, a função Es não possui parâmetros correspondentes a valores de atributos. □

Definições Dirigidas pela Sintaxe Fortemente Não-Circulares

Os avaliadores recursivos podem ser construídos para uma classe de definições dirigidas pela sintaxe, chamadas de "fortemente não-circulares". Para uma definição nesta classe, os atributos a cada nó para um não-terminal podem ser avaliados de acordo com a mesma ordem (parcial). Quando construímos a função para um atributo sintetizado de um não-terminal, esta ordem é usada para selecionar os atributos sintetizados que vêm a ser os parâmetros da função.

Daremos agora uma definição desta classe e mostraremos que a definição dirigida pela sintaxe na Fig. 5.57 cai neste caso. Forneceremos, então, um algoritmo para testar a circularidade e a não-circularidade forte e mostraremos como a implementação do Exemplo 5.30 se estende a todas as definições fortemente não-circulares.

Consideremos o não-terminal A ao nó n de uma árvore gramatical. O grafo de dependências para árvore gramatical pode em geral ter percursos que comecem num atributo de um nó n, passam através dos atributos dos outros nós na árvore gramatical e terminam num outro atributo de n. Para nossos propósitos, é suficiente procurar por percursos que se mantenham dentro da árvore gramatical abaixo de A. Um pouco de raciocínio revela que tais percursos partem de algum atributo herdado de A para algum outro atributo sintetizado de A. Iremos fazer uma estimativa (possivelmente muito pessimista) do conjunto de tais percursos através da consideração de ordens parciais entre os atributos de A.

Seja a produção p com os não-terminais $A_1, A_2, ..., A_n$ ocorrendo no lado direito. Seja RA_j uma ordem parcial sobre os atributos de A_j, para $1 \leq j \leq n$. Escrevemos $D_p[RA_1, RA_2, ..., RA_n]$ para o grafo obtido pela adição de lados a D_p como segue: se RA_j ordena o atributo $A_j.b$ antes de $A_j.c$, adicionamos um lado de $A_j.b$ para $A_j.c$.

Uma definição dirigida pela sintaxe é dita ser *fortemente não-circular* se, para cada não-terminal A, pudermos encontrar uma ordem parcial RA sobre os atributos de A, tal que, para cada produção p, com lado esquerdo A e não-terminais $A_1, A_2, ..., A_n$ ocorrendo no lado direito

1. $D_p[RA_1, RA_2, ..., RA_n]$ for acíclico e
2. se existir um lado do atributo $A.b$ para $A.c$ em $D_p[RA_1, RA_2, ..., RA_n]$ então RA ordena $A.b$ antes de $A.c$.

Exemplo 5.31. Seja p a produção $E \rightarrow E_1\ E_2$ da Fig. 5.57, cujo grafo de dependências D_p está ao centro da Fig. 5.58. Seja RE a relação de ordem parcial (ordem total neste caso) $s \rightarrow i \rightarrow t$. Existem duas ocorrências de não-terminais no lado direito de p, escritas E_1 e E_2, como de praxe. Por conseguinte, RE_1 e RE_2 são os mesmos que RE e o grafo $D_p[RE_1, RE_2]$ é como mostrado na Fig. 5.61.

Dentre os atributos associados à raiz E da Fig. 5.61, os únicos percursos são de i para t. Como RE faz i preceder t, não existe violação da condição (2). □

Dada uma definição fortemente não-circular e uma ordem parcial RA para cada não-terminal A, a função para o atributo sintetizado s de A toma os argumentos como segue: se RA odena o atributo herdado i antes de s, então i é um argumento para a função, caso contrário não o é.

PRODUÇÃO	REGRAS SEMÂNTICAS
$S \rightarrow E$	$E.i := g(E.s)$
	$S.r := E.t$
$E \rightarrow E_1\ E_2$	$E.s := fs(E_1{\cdot}s, E_2{\cdot}s)$
	$E_1{\cdot}i := fi1(E.i)$
	$E_2{\cdot}i := fi2(E.i)$
	$E.t := ft(E_1{\cdot}t, E_2{\cdot}t)$
$E \rightarrow$ **id**	$E.s :=$ **id**$.s$
	$E.t := h(E.i)$

Fig. 5.57. Atributos sintetizados s e t não podem ser avaliados juntos.

Um Teste de Circularidade

Uma definição dirigida pela sintaxe é dita circular se o grafo de dependências para alguma árvore gramatical possuir um ciclo; as definições circulares são malformadas e sem significado. Não existe forma para que possamos começar a computar quaisquer valores de atributos no ciclo. O cômputo de relações de ordem parcial que assegurem que uma definição seja fortemente não-circular está estritamente relacionado ao teste de circularidade de uma definição. Iremos, por conseguinte, considerar primeiro um teste para a circularidade.

```
função E: ↑ nó_árvore_sintática;
    var r: ↑ nó_árvore_sintática;
        lexema_op_aditivo: caractere;
    procedimento R;
    início
        se lookahead = op_aditivo então início
        lexema_op_aditivo : = lexval;
        reconhecer (op_aditivo);
        r : = criar_nó (lexema_op_aditivo, r, T);
        R
    fim
    fim;
início
    r : = T; R
    retornar r
fim;
```

Fig. 5.56. Compare o procedimento R com o código na Fig. 5.31.

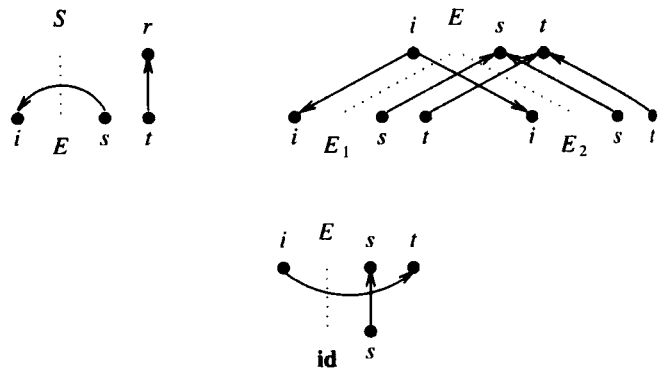


Fig. 5.58. Grafos de dependências para as produções na Fig. 5.57.

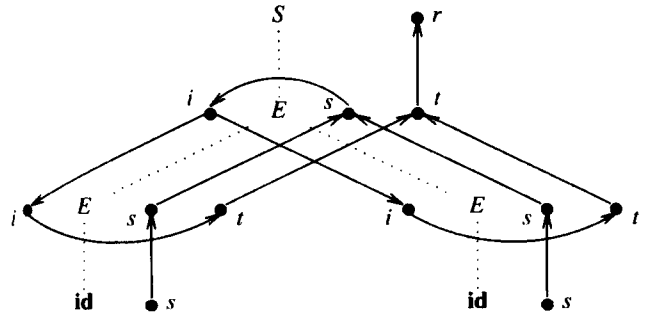


Fig. 5.59. Grafo de dependências para uma árvore gramatical.

Exemplo 5.32. Na seguinte definição dirigida pela sintaxe, os percursos entre os atributos de A dependem de que produção é aplicada. Se $A \rightarrow 1$ for aplicada, então $A.s$ depende de $A.i$; em caso contrário, não depende. Para uma completa informação sobre as possíveis dependências, teremos que controlar os conjuntos de relações de ordem parcial sobre os atributos de um não-terminal.

PRODUÇÃO	REGRAS SEMÂNTICAS
$S \rightarrow A$	$A.i := c$
$A \rightarrow 1$	$A.s := f(A.i)$
$A \rightarrow 2$	$A.s := d$

□

```
função Es (n);
  início
        caso a produção ao nó n seja
        'E → E1 E2':
              s1 := Es (filho (n, 1));
              s2 := Es (filho (n, 2));
              retornar fs (s1, s2);
        'E → id':
              retornar id.s;
        default:
                    erro
        fim
  fim;

  função Et (n, i);
  início
        caso a produção ao nó n seja
        'E → E1 E2';
              i1 := fi1 (i);
              t1 := Et (filho (n, 1), i1);
              i2 := fi2 (i);
              t2 := Et (filho (n, 2), i2);
              retornar ft (t1, t2);
        'E → id':
              retornar h (i);
        default:
              erro
        fim
  fim;

  função Sr (n);
  início
        s := Es (filho (n, 1));
        i := g(s);
        t := Et (filho (n, 1), i);
        retornar t
  fim;
```

Fig. 5.60. Funções para atributos sintetizados na Fig. 5.57.

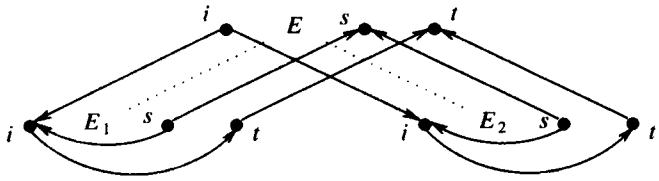


Fig. 5.61. Grafo aumentado de dependências para uma produção.

A idéia por trás do Algoritmo da Fig. 5.62 é como se segue. Representamos relações de ordem parcial por grafos dirigidos acíclicos. Dados dois GDAs para os atributos dos símbolos ao lado direito de uma produção, podemos determinar um GDA para os atributos do lado esquerdo como se segue:

Seja a produção p igual a $A \rightarrow X_1 X_2 \ldots X_k$ com grafo de dependências D_p. Seja D_j um GDA para X_j, $1 \leq j \leq k$. Cada lado $b \rightarrow a$ em D_j é temporariamente adicionado ao grafo de dependências D_p para a produção. Se o grafo resultante possuir um ciclo, a definição dirigida pela sintaxe é circular. Em caso contrário, os percursos no grafo resultante determinam um novo GDA para os atributos no lado esquerdo da produção e o GDA resultante é adicionado a $\mathcal{F}(A)$.

O teste de circularidade da Fig. 5.52 gasta um tempo que é uma exponencial do número de grafos nos conjuntos $\mathcal{F}(X)$ para qualquer símbolo gramatical X. Existem definições dirigidas pela sintaxe que não podem ser testadas pela circularidade num tempo polinomial.

Podemos converter o Algoritmo da Fig. 5.62 num teste mais eficiente, se uma definição dirigida pela sintaxe for fortemente não-circular, como se segue. Ao invés de manter uma família de grafos $\mathcal{F}(X)$ para cada X, sumarizamos a informação da família mantendo um único grafo $F(X)$. Note-se que cada grafo em $\mathcal{F}(X)$ possui os mesmos nós para os atributos de X, mas pode ter lados diferentes. $F(X)$ é o grafo sobre os nós para os atributos do X que tenha um lado entre $X.b$ e $X.c$, se algum grafo em $\mathcal{F}(X)$ o tiver. $F(X)$ representa uma "estimativa do pior caso" das dependências entre os atributos de X. Em particular, se $F(X)$ for acíclico, a definição dirigida pela sintaxe é garantida ser não-circular. No entanto, a recíproca não é verdadeira necessariamente; isto

```
para o símbolo gramatical X faça
   𝓕(X) possui um único grafo com os atributos de X e nenhum
   lado;
repetir
   mudou := falso;
  para a produção p dada por A → X1X2 ... Xk faça início
    para GDAs G1 ∈ 𝓕 (X1), . . ., Gk ∈ 𝓕 (Xk) faça início
          D := Dp;
          para o lado b → c em Gj, 1 ≤ j ≤ k faça
             adicionar um lado em D entre os atributos b e c de Xj;
          se D possui um ciclo então
             falhar o teste de circularidade
          senão início
             G := um novo grafo com nós para os atributos
                  de A e nenhum lado;
             para cada par de atributos b e c de A faça se existe
             um percurso em D de b para c então
                     adicionar b → c a G;
             se G não estiver ainda em 𝓕 (A) então início
                     adicionar G a 𝓕 (A);
                     mudou := verdadeiro
             fim
          fim
    fim
  fim
até que mudou = falso
```

Fig. 5.62. Um teste de circularidade.

é, se $F(X)$ possuir um ciclo, não é necessariamente verdadeiro que a definição dirigida pela sintaxe seja circular.

O teste de circularidade modificado constrói grafos acíclicos $F(X)$ para cada X se o mesmo tiver sucesso. A partir desses grafos podemos construir um avaliador para a definição dirigida pela sintaxe. O método é uma generalização direta do Exemplo 5.30. A função para o atributo sintetizado $X.s$ toma como argumento todos e somente os atributos herdados que precedem s em $F(X)$. A função, chamada ao nó n, chama outras funções para computar os atributos sintetizados necessitados ao filho de n. Para as rotinas que computam esses atributos são passados os valores para os atributos herdados dos quais necessitam. O fato de que um teste de não forte circularidade ter tido sucesso garante que esses atributos herdados podem ser computados.

EXERCÍCIOS

5.1 Para a expressão de entrada `(4 * 7 + 1) * 2`, construa uma árvore gramatical anotada de acordo com a definição dirigida pela sintaxe da Fig. 5.2.

5.2 Construa as árvores gramatical e sintática para a expressão `((a) + (b))` de acordo com

a) A definição dirigida pela sintaxe da Fig. 5.9 e

b) O esquema de tradução d a Fig. 5.28.

5.3 Construa um GDA e identifique os valores numéricos para as subexpressões da expressão seguinte, assumindo que + associe a partir da esquerda:

```
a + a + (a + a + a (a + a + a + a))
```

***5.4** Dê uma definição dirigida pela sintaxe para traduzir expressões infixas em expressões infixas sem parêntesis redundantes. Por exemplo, uma vez que + e * associam à esquerda, `((a*(b+c))*(d))` pode ser reescrita como `a * (b + c) * d`.

5.5 Dê uma definição dirigida pela sintaxe que gere a derivada de expressões formadas pela aplicação dos operadores aritméticos + e * à variável `x` e a constantes; por exemplo, `x*(3*x + x*x)`. Assuma que nenhuma simplificação tome lugar, de tal forma que `3*x` avalie em `3 * 1 + 0 * x`.

5.6 A seguinte gramática gera expressões formadas pela aplicação do operador aritmético + a constantes inteiras e reais. Quando dois inteiros são adicionados, o tipo resultante é inteiro, em caso contrário é real.

$$E \rightarrow E + T \mid T$$
$$T \rightarrow \textbf{num . num} \mid \textbf{num}$$

a) Dê uma definição dirigida pela sintaxe para determinar o tipo de cada subexpressão.

b) Estenda a definição dirigida pela sintaxe de (a) para traduzir na notação posfixa bem como determinar os tipos. Use o operador unário **inttoreal** para converter um valor inteiro num valor real equivalente, de forma a que ambos os operandos de + na forma posfixa tenham o mesmo tipo.

5.7 Estenda a definição dirigida pela sintaxe da Fig. 5.22 de forma a controlar o comprimento dos quadros em adição ao controle de suas larguras. Assuma que o terminal **texto** possua o atributo sintetizado w que forneça o comprimento normalizado do texto.

5.8 Seja o atributo sintetizado *val*, que fornece o valor do número binário gerado por S na gramática seguinte. Por exemplo, para a entrada `101.101`, $S.val = 5.625$.

$$S \rightarrow L \,.\, L \mid L$$
$$L \rightarrow L\,B \mid B$$
$$B \rightarrow \texttt{0} \mid \texttt{1}$$

a) Use atributos sintetizados para determinar $S.val$.

b) Determine $S.val$ com uma definição dirigida pela sintaxe na qual o único atributo sintetizado de B é c, fornecendo a contribuição do *bit* gerado por B para o valor final. Por exemplo, as contribuições dos primeiro e último *bits* em `101.101` para o valor 5.625 são, respectivamente, 4 e 0.125.

5.9 Reescreva a gramática subjacente na definição dirigida pela sintaxe do Exemplo 5.3, de tal forma que a informação de tipo possa ser propagada usando-se somente atributos sintetizados.

***5.10** Quando os enunciados gerados pela gramática seguinte forem traduzidos num código de máquina abstrata, um enunciado *break* é traduzido como um desvio para a instrução seguindo-se ao enunciado *while* envolvente mais próximo. Por simplicidade, as expressões são representadas pelo terminal **expr** e outros tipos de enunciados pelo terminal **other.** Esses terminais possuem um atributo sintetizado *código* fornecendo as suas traduções.

$$S \rightarrow \textbf{while expr do begin } S \textbf{ end}$$
$$\mid S \,;\, S$$
$$\mid \textbf{break}$$
$$\mid \textbf{other}$$

Forneça uma definição dirigida pela sintaxe que traduza enunciados em código para a máquina de pilha da Seção 2.8. Assegure-se de que os enunciados *break* dentro dos enunciados *while* aninhados sejam traduzidos corretamente.

5.11 Elimine a recursão à esquerda das definições dirigidas pela sintaxe nos Exercícios 5.6(a) e (b).

5.12 As expressões geradas pela gramática seguinte podem ter atribuições dentro de si.

$$S \rightarrow E$$
$$E \rightarrow E := E \mid E + E \mid (E) \mid \textbf{id}$$

A semântica das expressões é como em C. Isto é, `b:=c` é uma expressão que atribui o valor de `c` a `b`; o valor-r desta expressão é o mesmo do de `c`. Sobretudo, `a:=(b:=c)` atribui o valor de `c` a `b` e então a `a`.

a) Construa uma definição dirigida pela sintaxe para verificar se o lado esquerdo de uma expressão é um valor-l. Use o atributo herdado *lado* para o não-terminal E a fim de indicar se a expressão gerada por E aparece no lado esquerdo ou direito de uma atribuição.

b) Estenda a definição dirigida pela sintaxe em (a) para gerar código intermediário para a máquina de pilha da Seção 2.8 à medida que examina a entrada.

5.13 Reescreva a gramática subjacente do Exercício 5.12 de tal forma que agrupe as subexpressões de := à direita e as subexpressões de + à esquerda.

a) Construa um esquema de tradução que simule uma definição dirigida pela sintaxe do Exercício 5.12(b).

b) Modifique o esquema de tradução de (a) de forma que emita código incrementalmente para um arquivo de saída.

5.14 Forneça um esquema de tradução que verifique se o mesmo identificador não aparece duas vezes numa lista de identificadores.

5.15 Suponha que as declarações sejam geradas pela seguinte gramática.

$$D \rightarrow \textbf{id}\ L$$
$$L \rightarrow \textbf{, id}\ L \mid \textbf{:}\ T$$
$$T \rightarrow \textbf{integer} \mid \textbf{real}$$

a) Construa um esquema de tradução para inserir o tipo de cada identificador na tabela de símbolos, como no Exemplo 5.3.

b) Construa um tradutor preditivo a partir do esquema de tradução em (a).

5.16 A gramática seguinte é uma versão inambígua da gramática subjacente da Fig. 5.22. As chaves { } são usadas somente para agrupar quadros e são eliminadas durante a tradução.

$$S \rightarrow L$$
$$L \rightarrow L\,B \mid B$$
$$B \rightarrow B\ \textbf{sub}\ F \mid F$$
$$F \rightarrow \{\ L\ \} \mid \textbf{texto}$$

a) Adapte a definição dirigida pela sintaxe da Fig. 5.22 de forma que use a gramática acima.
b) Converta a definição dirigida pela sintaxe de (a) num esquema de tradução.

***5.17** Estenda a transformação para eliminar a recursão na Seção 5.5 de forma a permitir para o não-terminal A em (5.2):
a) Atributos herdados através de regras de cópia.
b) Atributos herdados.

5.18 Elimine a recursão à esquerda do esquema de tradução do Exercício 5.16(b).

***5.19** Suponhamos ter uma definição L-atribuída cuja gramática subjacente ou é LL(1) ou é uma daquelas para as quais podemos resolver ambigüidades e construir um analisador sintático preditivo. Mostre que podemos manter os atributos herdados e sintetizados na pilha do analisador sintático *top-down* dirigido pela tabela sintática preditiva.

***5.20** Prove que a adição de não-terminais marcadores únicos a uma gramática LL(1) resulta numa gramática que é LR(1).

5.21 Considere a seguinte modificação da gramática LR(1) $L \rightarrow L\,b \mid a$:

$$L \rightarrow M\,L\,b \mid a$$
$$M \rightarrow \epsilon$$

a) Em que ordem iria um analisador sintático *bottom-up* aplicar as produções na árvore gramatical para a cadeia de entrada *abbb*?
*b) Mostre que a gramática modificada não é LR(1).

***5.22** Mostre que num esquema de tradução baseado na Fig. 5.36, o valor do atributo herdado *B.tp* está sempre imediatamente abaixo do lado direito, sempre que reduzirmos um lado direito para *B*.

5.23 O Algoritmo 5.3 para a tradução e análise sintática *bottom-up* com atributos herdados utiliza não-terminais marcadores para guardar valores dos atributos herdados em posições previsíveis na pilha do analisador sintático. Poucos marcadores podem ser necessários se os valores forem colocados numa pilha separada da pilha sintática.
a) Converta a definição dirigida pela sintaxe na Fig. 5.36 num esquema de tradução.
b) Modifique o esquema de tradução construído em (a) de tal forma que o valor do atributo herdado *tp* figure numa pilha separada. Elimine o não-terminal marcador M no processo.

***5.24** Considere a tradução durante a análise sintática como no Exercício 5.23. S. C. Johnson sugere o seguinte método para simular uma pilha separada para os atributos herdados, usando uma variável global para cada atributo herdado. Na seguinte produção, o valor v é empilhado na pilha i pela primeira ação e é removido pela segunda:

$$A \rightarrow \alpha\ \{\mathit{empilhar}\ (v,\ i)\ \}\ \beta\ \{\ \mathit{desempilhar}\ (i)\ \}$$

A pilha i pode ser simulada pelas seguintes produções que usam uma variável global g e um não-terminal marcador M com o atributo sintetizado s:

$$A \rightarrow \alpha\ M\ \beta\ \{\ g := M.s\ \}$$
$$M \rightarrow \epsilon\ \{\ M.s := g;\ g := v\}$$

a) Aplique esta transformação ao esquema de tradução do Exercício 5.23(b). Substitua todas as referências ao topo da pilha separada por referências à variável global.
b) Mostre que o esquema de tradução construído em (a) computa os mesmos valores para o atributo sintetizado do símbolo de partida que o do Exercício 5.23(b).

5.25 Use a abordagem da Seção 5.8 para implementar todos os atributos de *E.lado* usando o esquema de tradução do Exercício 5.12(b), através de uma única variável *booleana*.

5.26 Modifique o uso da pilha durante uma travessia em profundidade no Exemplo 5.26 de tal forma que os valores na pilha correspondam àqueles mantidos na pilha do analisador sintático do Exemplo 5.19.

NOTAS BIBLIOGRÁFICAS

O uso de atributos sintetizados para especificar a tradução de uma linguagem aparece em Irons [1961]. A idéia de um analisador sintático chamar por ações sintáticas é discutida em Samuelson e Bauer [1960] e Brooker e Morris [1962]. Juntamente com os atributos herdados, os grafos de dependências e um teste pela não-circularidade forte aparecem em Knuth [1968]; um teste de circularidade aparece num artigo de correção. O exemplo expandido no artigo usa efeitos colaterais disciplinados nos atributos globais atrelados à raiz de uma árvore gramatical. Se os atributos podem ser funções, os atributos herdados podem ser eliminados; como feito na semântica denotacional, podemos associar uma função dos atributos herdados para os sintetizados com um não-terminal. Tais observações aparecem em Mayoh [1981].

Uma aplicação na qual os efeitos colaterais nas regras semânticas são indesejáveis está na edição dirigida pela sintaxe. Suponhamos que um editor seja gerado a partir de uma gramática de atributos para a linguagem-fonte, como em Reps [1984], e consideremos uma mudança de edição no programa-fonte que resulte na remoção de uma parte da árvore gramatical para o programa. Na medida em que não hajam efeitos colaterais, os valores dos atributos para o programa modificado podem ser recomputados incrementalmente.

Ershov [1958] usa *hashing* para controlar as subexpressões comuns.

A definição de gramáticas L-atribuídas em Lewis, Rosenkrantz e Stearns [1974] é motivada pela tradução durante a análise sintática. Restrições similares sobre as dependências entre os atributos se aplicam a cada uma das travessias em profundidade, da esquerda para a direita, em Bochmann [1976]. Gramáticas afixas, como introduzidas por Koster [1971], estão relacionadas às gramáticas L-atribuídas. As restrições sobre as gramáticas L-atribuídas são propostas por Koskimies e Räihä [1983] para controlar o acesso aos atributos globais.

A construção mecânica de um tradutor preditivo, similar àqueles construídos pelo Algoritmo 5.2 é descrita por Bochmann e Ward [1978]. A impressão de que a análise sintática *top-down* permite maior flexibilidade para a tradução mostra-se falsa por uma prova em Brosgol [1974] de que o esquema baseado numa gramática LL(1) pode ser simulado durante a análise sintática LR(1). Independentemente, Watt [1977] usou não-terminais marcadores para assegurar que os valores dos atributos herdados aparecessem na pilha durante a análise sintática *bottom-up*. As posições nos lados direitos das produções onde os não-terminais marcadores podem ser inseridos com segurança, sem se perder a propriedade LR(1), são consideradas em Purdom e Brown [1980] (ver o Exercício 5.21). Simplesmente exigir que os atributos herdados sejam definidos através de regras de cópia não é o bastante para assegurar que os atributos possam ser avaliados durante a análise sintática *bottom-up;* as condições de suficiência sobre as regras semânticas são dadas por Jones e Madsen [1980]. Como um exemplo de uma tradução

que não pode ser feita durante a análise sintática. Giegerich e Wilhelm [1978] consideram a geração de código para expressões *booleanas*. Veremos na Seção 8.6 que a retrocorreção *(backpatching)* pode ser usada para esse problema e conseqüentemente uma segunda passagem completa não é necessária.

Um número de ferramentas especializadas para a implementação de definições dirigidas pela sintaxe foram desenvolvidas, começando com FOLDS, por Fang [1972], mas poucas têm sido vistas em uso disseminado. DELTA por Lorho [1977] construiu um grafo de dependências em tempo de compilação. O espaço era economizado através do controle dos tempos de vida dos atributos e eliminação das regras de cópia. Métodos de avaliação de atributos baseados em árvores sintáticas são discutidos por Kennedy e Ramanathan [1979] e Cohen e Harry [1979].

Métodos de avaliação de atributos são examinados de forma extensiva *(survey)* por Engelfriet [1984]. Um artigo correlato, por Courcelle [1984], pesquisa os fundamentos teóricos. HLP, descrito por Räihä et al. [1983], realiza travessias alternativas, em profundidade, como sugerido por Jazayeri e Walter [1975]. LINGUIST por Farrow [1984] também realiza passagens alternativas. Ganzinger et al. [1982] reporta que MUG permite que a ordem na qual os filhos de um nó são visitados seja determinada pela produção ao nó. GAG, devido a Kastens, Hutt e Zimmerman [1982], permite repetidas visitas aos filhos de um nó. GAG implementa a classe gramática de atributos ordenados definida por Kastens [1980]. A idéia de repetidas visitas aparece num artigo anterior por Kennedy e Warren [1976], onde avaliadores de uma classe maior de gramáticas fortemente não-circulares são construídos. Saarinen [1978] descreve uma modificação do método de Kennedy e Warren que economiza espaço mantendo os valores de atributos numa pilha se não forem necessários durante uma visita posterior. Uma implementação descrita por Jourdan [1984] constrói avaliadores recursivos para esta classe. Avaliadores recursivos também são construídos por Katayama [1984]. Um enfoque um tanto diferente é tomado em NEATS, por Madsen [1980], onde um GDA é construído para expressões que representam valores de atributos.

A análise das dependências em tempo de construção do compilador pode economizar tempo e espaço em tempo de compilação. O teste de circularidade é um problema típico de análise. Jazayeri, Ogden e Rounds [1975] provam que um teste de circularidade requer uma quantidade de tempo exponencial em função do tamanho da gramática. Técnicas para aprimorar a implementação dos testes de circularidade são considerados por Lorho e Pair [1975], Räihä e Saarinen [1982] e Deransart, Jourdan e Lorho [1984].

O espaço usado por avaliadores ingênuos tem levado ao desenvolvimento de técnicas para a conservação do espaço. O algoritmo para atribuição de valores de atributos a registradores na Seção 5.8 foi descrito num contexto diferente por Marill [1962]. O problema de se encontrar uma classificação topológica do grafo de dependências que minimize o número de registradores usados é mostrado ser NP-completo em Sethi [1975]. A análise em tempo de compilação de tempos de vida num avaliador de múltiplas passagens aparece em Räihä [1981] e Jazayeri e Pozefsky [1981]. Branquart et al. [1976] menciona o uso de pilhas separadas para aguardar atributos sintetizados e herdados durante uma travessia. GAG realiza a análise dos tempos de vida e coloca os valores dos atributos e variáveis globais, pilhas e árvores gramaticais, na medida do necessário. Uma comparação das técnicas de economia de espaço usadas por GAG e LINGUIST é feita por Farrow e Yellin [1984].

CAPÍTULO 6

VERIFICAÇÃO DE TIPOS

Um compilador precisa verificar se o programa segue as convenções sintáticas e semânticas da linguagem fonte. Essa checagem, chamada de *verificação estática* (para distinguir da verificação *dinâmica*, durante a execução do programa-alvo), assegura que certos tipos de erro de programa serão detectados e reportados. Os exemplos de verificação estática incluem:

1. *Verificação de tipos.* Um compilador deveria relatar um erro se um operador for aplicado a um operando incompatível; por exemplo; se uma variável tipo *array* e uma variável função forem adicionadas.

2. *Verificação do fluxo de controle.* Os enunciados que fazem o fluxo de controle deixar uma construção precisam ter algum local para onde transferir o controle. Por exemplo, um enunciado *break* em C faz com que o controle deixe o *while*, *for* ou *switch* envolvente mais interno; um erro ocorre se um tal enunciado envolvente não existir.

3. *Verificações de unicidade.* Existem situações nas quais um objeto precisa ser definido exatamente uma vez. Por exemplo, em Pascal, um identificador precisa ser declarado univocamente, os rótulos em enunciados *case* precisam ser distintos, e os elementos num tipo escalar não podem ser repetidos.

4. *Verificações relacionadas aos nomes.* Algumas vezes, o mesmo nome precisa figurar duas ou mais vezes. Por exemplo, em Ada, um laço ou bloco precisa ter um nome que apareça ao início e ao final da construção. O compilador precisa verificar se o mesmo nome é usado em ambos os locais.

Neste capítulo, focalizamos a verificação de tipos. Como os exemplos acima indicam, a maioria das outras verificações estáticas é rotineira e pode ser implementada usando-se as técnicas do último capítulo. Algumas delas podem ser inseridas em outras atividades. Por exemplo, na medida em que damos entrada às informações a respeito de um nome numa tabela de símbolos, podemos verificar se o mesmo foi declarado univocamente. Muitos compiladores Pascal combinam a verificação estática e a geração de código intermediário com a análise sintática. Com construções mais complexas, como as de Ada, pode ser conveniente ter uma passagem separada de verificação de tipos entre a análise sintática e a geração de código intermediário, como indicado na Fig. 6.1.

Um verificador de tipos checa se o tipo de uma construção corresponde exatamente àquele esperado no contexto. Por exemplo, o operador aritmético embutido `mod` em Pascal exige operandos inteiros e, por conseguinte, um verificador de tipos precisa checar se os operandos de `mod` têm o tipo inteiro. Similarmente, o verificador de tipos precisa assegurar que o derreferenciamento seja aplicado somente a um apontador, que a indexação seja feita somente sobre um *array*, que uma função definida pelo usuário seja aplicada ao número e tipos corretos de argumentos e assim por diante. Uma especificação de um verificador simples de tipo aparece na Fig. 6.2. A representação dos tipos e a questão de quando dois tipos se igualam são discutidas na Seção 6.3.

As informações a respeito dos tipos capturadas pelo verificador de tipos podem ser necessárias quando o código estiver sendo gerado. Por exemplo, operadores aritméticos, como +, usualmente se aplicam a inteiros ou reais, talvez a outros tipos, e precisamos olhar o contexto de + para determinar o significado pretendido. Um símbolo que possa representar diferentes operações em diferentes contextos é dito "sobrecarregado". A sobrecarga pode ser acompanhada pela coerção de tipos, onde o compilador fornece um operador para converter um operando no tipo esperado pelo contexto.

Uma noção distinta de sobrecarga é a de "polimorfismo". O corpo de uma função polimórfica pode ser executado com argumentos de vários tipos. Um algoritmo de unificação para inferir os tipos das funções polimórficas conclui este capítulo.

6.1 SISTEMAS DE TIPOS

O projeto de um verificador para uma linguagem é baseado nas informações sobre as construções sintáticas de linguagem, a noção de tipos de dados e as regras para atribuição de tipos às construções das linguagens. Os seguintes excertos provenientes do relatório Pascal e do manual de referência de C são, respectivamente, exemplos de informações com que um escritor de compiladores poderia ter que lidar.

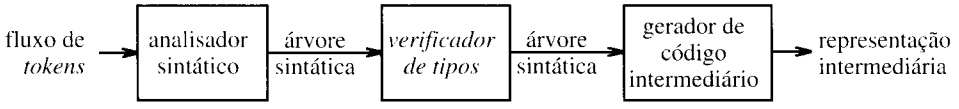


Fig. 6.1. Posição do verificador de tipos.

- "Se ambos os operandos dos operadores aritméticos de adição, subtração e multiplicação são do tipo inteiro, o resultado também o é ."

- "O resultado do operador unário & é um apontador para o objeto referido pelo operando. Se o tipo do operando for do tipo '...', o resultado é um 'apontador para ...'."

Implícita nos excertos acima é a idéia de que cada expressão possui um tipo associado à mesma. Sobretudo, os tipos têm estrutura; o tipo "apontador para ..." é construído a partir do tipo que "..." referencia.

Tanto em Pascal quanto em C, os tipos ou são básicos ou são construídos. Os tipos básicos são os atômicos, sem estrutura interna na medida em que o programador está envolvido com o tema. Em Pascal, os tipos básicos são os booleanos, caracteres, inteiros e reais. Tipos subintervalo, como `1..10`, e tipos enumerados, como

```
(violeta, índigo, azul, verde, amarelo, laranja, vermelho)
```

podem ser tratados como tipos básicos. Pascal permite que o programador construa tipos a partir dos tipos básicos e de outros tipos construídos, sendo exemplos os *arrays*, registros e conjuntos. Adicionalmente, os apontadores e funções podem ser também tratados como tipos construídos.

Expressões de Tipos

O tipo de uma construção de linguagem será denotado por uma "expressão de tipo". Informalmente, uma expressão de tipo ou é um tipo básico ou é formada através da aplicação de um operador, chamado de um *construtor de tipos*, a outras expressões de tipo. Os conjuntos de tipos e de construtores básicos dependem da linguagem que está sendo verificada.

Este capítulo usa a seguinte definição para *expressões de tipo:*

1. Um tipo básico é uma expressão de tipo. Dentre os tipos básicos estão os *booleanos, caracteres* e *reais*. Um tipo especial de tipo, o *tipo erro*, irá sinalizar um erro durante a verificação. Finalmente, o tipo básico *vazio*, denotando "a ausência de um valor" permitirá que todos os comandos sejam verificados.

2. Uma vez que as expressões podem receber nomes, um nome de tipo é uma expressão de tipo. Um exemplo do uso dos nomes de tipo aparece em 3(c) abaixo; as expressões de tipo contendo nomes são discutidas na Seção 6.3.

3. Um construtor de tipos aplicado a uma expressão de tipo é uma expressão de tipo. Os construtores abrangem:

   a) *Arrays.* Se T é uma expressão de tipo, então *array (I, T)* é uma expressão de tipo denotando o tipo *array* de elementos com tipo T e conjunto de índices I. I é freqüentemente um intervalo dos inteiros. Por exemplo, a declaração

   ```
   var A: array[1..10] of integer;
   ```

   associa a expressão de tipo *array* (1..10, *integer*) a `A`.

   b) *Produtos.* Se T_1 e T_2 são expressões de tipo, o produto cartesiano $T_1 \times T_2$ é uma expressão de tipo. Assumimos que $\times$ associe à esquerda.

   c) *Registros.* A diferença entre um registro e um produto é que os campos num registro têm nomes. Um construtor do tipo *registro* será aplicado a uma tupla formada a partir dos nomes de campos e dos tipos de campos (tecnicamente, os nomes de campo deveriam ser parte do construtor de tipos, mas é conveniente manter os nomes de campo junto aos seus tipos associados. No Capítulo 8, o construtor de tipos *registro* é aplicado a um apontador para a tabela de símbolos contendo entradas para os nomes de campo). Por exemplo, o fragmento de programa Pascal

   ```
   type linha = record
                  endereco: integer;
                  lexema: array [1..15] of char
                end;

   var tabela: array [1..101] of linha;
   ```

   declara o nome de tipo `linha` como representando a seguinte expressão de tipo

   registro ((`endereco` $\times$ *inteiro*) $\times$ (`lexema` $\times$ *array* (1..15, *caractere*)))

   e a variável `tabela` como sendo um *array* de registros desse tipo.

   d) *Apontadores.* Se T é uma expressão de tipo, então *apontador (T)* é uma expressão de tipo denotando o tipo "apontador para um objeto de tipo T". Por exemplo, em Pascal, a declaração

   ```
   var p: ↑linha
   ```

   declara a variável `p` como tendo o tipo *apontador* (`linha`).

   e) *Funções.* Matematicamente, uma função mapeia elementos de um conjunto, o *domínio*, em outro conjunto, o *intervalo*. Devemos tratar as funções nas linguagens de programação como mapeando o *domínio de tipos D* em um *intervalo de tipos R*. O tipo de tal função será denotado pelas expressões de tipo $D \rightarrow R$. Por exemplo, a função embutida *mod* de Pascal possui domínio de tipos *int* $\times$ *int*, isto é, um par de inteiros, e intervalo de tipos *int*. Por conseguinte, dizemos que `mod` possui o tipo[1]

   $$int \times int \rightarrow int$$

   Como um outro exemplo, a declaração Pascal

   ```
   function f(a, b: char) : ↑integer; ...
   ```

   diz que o domínio de tipos de `f` é denotado por *caractere* $\times$ *caractere* e o intervalo de tipos por *apontador (inteiro)*. O tipo de `f` é então denotado pela expressão de tipo

   $$caractere \times caractere \rightarrow apontador\ (inteiro)$$

   Freqüentemente, por motivos de implementação, discutidos no próximo capítulo, existem limitações sobre o tipo que uma função pode retornar; por exemplo, *arrays* ou funções não podem ser retornadas. Existem, entretanto, linguagens, das quais LISP é o exemplo mais proeminente, que permitem retornar objetos de tipos arbitrários, e, conseqüentemente, podemos definir uma função `g` do tipo

   $$(inteiro \rightarrow inteiro) \rightarrow (inteiro \rightarrow inteiro)$$

   Isto é, `g` toma como argumento uma função que mapeia um inteiro em um inteiro e produz como resultado uma outra função do mesmo tipo.

4. Expressões de tipo podem conter variáveis cujos valores sejam expressões de tipo. As variáveis de tipo serão introduzidas na Seção 6.6.

[1] Assumimos que $\times$ possua maior precedência do que $\rightarrow$, de forma que *int* $\times$ *int* é o mesmo que (*int* $\times$ *int*) $\rightarrow$ *int*. Adicionalmente, $\rightarrow$ associa à direita.

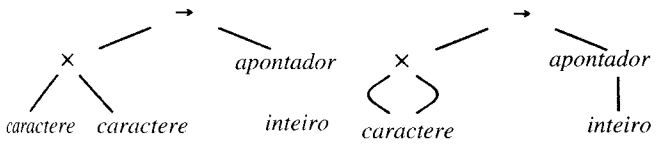


Fig. 6.2. Árvore e GDA, respectivamente, para *caractere* × *caractere* → *apontador (inteiro)*.

Uma forma conveniente de representar expressões de tipo é usar um grafo. Utilizando a abordagem dirigida pela sintaxe na Seção 5.2, podemos construir uma árvore ou GDA para uma expressão de tipo, com os nós interiores para os construtores de tipo e as folhas para os tipos básicos, nomes de tipos e variáveis de tipo (ver a Fig. 6.2). Os exemplos de representações de expressões de tipo que têm sido usadas nos compiladores são fornecidos na Seção 6.3.

Sistemas de Tipos

Um *sistema de tipos* é uma coleção de regras para expressões de tipos das várias partes de um programa. Um verificador de tipos implementa um sistema de tipos. Os sistemas de tipos deste capítulo são especificados numa forma dirigida pela sintaxe, para que possam ser prontamente implementados usando as técnicas do capítulo anterior.

Diferentes sistemas de tipos podem ser usados por diferentes compiladores ou processadores de uma mesma linguagem. Por exemplo, em Pascal, o tipo de um *array* inclui o conjunto de índices do *array* e, por conseguinte, uma função tendo um *array* por argumento pode somente ser aplicada a *arrays* com aquele conjunto de índices. No entanto, muitos compiladores Pascal permitem que o conjunto de índices seja deixado inespecificado quando um *array* é transmitido como argumento. Por conseguinte, esses compiladores usam um diferente sistema de tipos daquele da definição da linguagem Pascal. Similarmente, no sistema UNIX, o comando `lint` examina programas C procurando possíveis erros utilizando um sistema de tipos mais detalhado do que aquele que o próprio compilador C utiliza.

Verificação Estática e Dinâmica de Tipos

A verificação feita por um compilador é dita estática, enquanto que aquela feita enquanto o programa-alvo roda é denominada dinâmica. Em princípio, qualquer verificação pode ser feita dinamicamente, se o código-alvo carregar o tipo do elemento juntamente com o seu valor.

Um sistema *sonoro* de tipos elimina a necessidade de verificação dinâmica de tipos em busca de erros de tipo, porque nos permite determinar estaticamente que esses erros não poderão ocorrer quando o programa-alvo rodar. Isto é, se um sistema sonoro de tipos atribui um tipo que não *tipo-erro* a uma parte do programa, erros de tipo não poderão ocorrer quando o código-alvo para essa parte do programa for posto para rodar. Uma linguagem é fortemente tipada se seu compilador puder garantir que o programa que o mesmo aceita irá executar sem erros de tipo.

Na prática, algumas verificações só podem ser feitas dinamicamente. Por exemplo, se primeiro declararmos

```
tabela: array[0..255] of char;
i: integer;
```

e então computarmos `tabela[i]`, um compilador não poderá em geral garantir que, durante a execução, o valor de `i` repouse no intervalo de 0 a 255.[2]

[2]Podem ser usadas técnicas de análise de fluxo de dados, similares àquelas do Capítulo 10, para inferir se `i` estará entre os limites em alguns programas. Entretanto, nenhuma técnica poderá realizar a decisão corretamente em todos os casos.

Recuperação de Erros

Uma vez que a verificação de tipos tem o potencial de capturar erros em programas, é importante para um verificador de tipos realizar algo razoável quando um erro for descoberto. No mínimo, deveria reportar a natureza e a localização do erro. É desejável que o verificador de tipos se recupere dos erros, de forma a que possa verificar o resto da entrada. Como o tratamento de erros afeta as regras de verificação de tipos, precisa ser projetado no sistema de tipos desde o início; as regras precisam ser preparadas para lidar com os erros.

A inclusão do tratamento de erros pode resultar num sistema de tipos que vá além daquele necessitado para especificar programas corretos. Por exemplo, uma vez que o erro tenha ocorrido, podemos não saber o tipo do fragmento de programa incorretamente formado. Lidar com essa situação requer técnicas similares àquelas necessitadas para as linguagens que não requerem que os identificadores sejam declarados antes que sejam usados. As variáveis de tipo, discutidas na Seção 6.6, podem ser usadas para assegurar uma utilização consistente dos identificadores não declarados ou aparentemente incorretamente declarados.

6.2 ESPECIFICAÇÃO DE UM VERIFICADOR SIMPLES DE TIPOS

Nesta seção, especificamos um verificador de tipos para uma linguagem simples, na qual o tipo de cada identificador precisa ser declarado antes de ser usado. O verificador de tipos é um esquema de tradução que sintetiza o tipo de cada expressão a partir do tipo de suas subexpressões. O verificador de tipos pode tratar *arrays*, apontadores, enunciados e funções.

Uma Linguagem Simples

A gramática na Fig. 6.3 gera programas, representados pelo não-terminal P, consistindo em uma seqüência de declarações D seguidas por uma única expressão E.

Um programa gerado pela gramática da Fig. 6.3 é:

```
chave: integer:
chave mod 1999
```

Antes de discutir expressões, consideremos os tipos na linguagem. A linguagem em si possui dois tipos básicos, *caractere (char)* e *inteiro (integer)*, um terceiro tipo básico, *tipo_erro*, é usado para sinalizar erros. Por simplicidade, assumimos que todos os *arrays* comecem por 1. Por exemplo,

```
array [256] of char
```

leva à expressão de tipo *array* (1..256, *caractere*) consistindo no construtor *array* aplicado ao subintervalo 1..256 e ao subtipo *caractere*. Como em Pascal, o operador prefixo ↑ nas declarações constrói um tipo apontador, e, por conseguinte,

```
↑integer
```

$$
\begin{array}{lcl}
P & \rightarrow & D\,;\,E \\
D & \rightarrow & D\,;\,D \mid \textbf{id} : T \\
T & \rightarrow & \textbf{char} \mid \textbf{integer} \mid \textbf{array}\ [\ \textbf{num}\ ]\ \textbf{of}\ T \mid \uparrow T \\
E & \rightarrow & \textbf{literal} \mid \textbf{num} \mid \textbf{id} \mid E\ \textbf{mod}\ E \mid E\,[\,E\,] \mid E\uparrow
\end{array}
$$

Fig. 6.3. Gramática para uma linguagem-fonte.

P	$\rightarrow$	$D\ ;\ E$	
D	$\rightarrow$	$D\ ;\ D$	
D	$\rightarrow$	**id** : T	{*incluir-tipo*(**id**.*entrada*, *T.tipo*)}
T	$\rightarrow$	**char**	{*T.tipo* := *caractere*}
T	$\rightarrow$	**integer**	{*T.tipo* := *inteiro*}
T	$\rightarrow$	$\uparrow T_1$	{*T.tipo* := *apontador* (T_1.*tipo*)}
T	$\rightarrow$	**array** [**num**] **of** T_1	{*T.tipo* := *array* (1..**num**.*val*, T_1.*tipo*)}

Fig. 6.4. A parte de um esquema de tradução que salva o tipo de um identificador.

leva a uma expressão de tipo *apontador (inteiro)*, consistindo no construtor *apontador* aplicado ao tipo *inteiro*.

No esquema de tradução da Fig. 6.4, a ação associada à produção $D \rightarrow$ **id** : T guarda, para um identificador, um tipo numa entrada da tabela de símbolos. A ação *incluir_tipo* (**id**.*entrada*, *T.tipo*) é aplicada ao atributo sintetizado *entrada*, que aponta para a entrada de **id** na tabela de símbolos, e a uma expressão de tipo, representada pelo atributo sintetizado *tipo*, para um não-terminal T.

Se T gerar **char** ou **integer**, então *T.tipo* é definido como sendo *caractere* ou *inteiro*, respectivamente. O limite superior de um *array* é obtido a partir do atributo *val* do *token* **num**, que dá o inteiro representado por **num.** Assume-se que os *arrays* começam por 1 e, então, o tipo construtor *array* é aplicado ao subintervalo 1..**num**.*val* e ao tipo do elemento.

Como D aparece antes de E no lado direito de $P \rightarrow D;\ E$, podemos estar certos de que os tipos de todos os identificadores declarados serão salvos antes da expressão gerada por E ser verificada (veja o Capítulo 5). De fato, modificando-se adequadamente a gramática da Fig. 6.3, podemos implementar os esquemas de tradução desta seção durante a análise sintática *top-down* ou *bottom-up*, como desejado.

Verificação de Tipos das Expressões

Nas regras seguintes, o atributo sintetizado *tipo* para E fornece a expressão de tipo, atribuída pelo sistema de tipos, à expressão gerada por E. As seguintes regras semânticas dizem que as constantes representadas pelos *tokens* **literal** e **num** têm os tipos *caractere* e *inteiro*, respectivamente:

$E \rightarrow$ **literal**	{ *E.tipo* := *caractere* }
$E \rightarrow$ **num**	{ *E.tipo* := *inteiro* }

Usamos uma função *procurar (e)* para recuperar o tipo guardado na entrada da tabela de símbolos apontada por *e*. Quando um identificador aparece numa expressão, seu tipo declarado é recuperado e atribuído ao atributo *tipo*:

$E \rightarrow$ **id**	{ *E.tipo* := *procurar* (**id**.*entrada*) }

A expressão formada pela aplicação do operador mod às duas subexpressões de tipo *inteiro* possui tipo *inteiro*; caso contrário, seu tipo é *tipo_erro*. A regra é

$E \rightarrow E_1$ **mod** E_2	{ *E.tipo* := **se** E_1.*tipo* = *inteiro* **e** E_2.*tipo* = *inteiro* **então** *inteiro* **senão** *tipo_erro* }

Numa referência a *array* E_1 [E_2], a expressão de índice E_2 precisa ter tipo inteiro, caso em que o resultado é o elemento de tipo t obtido a partir do tipo *array (s, t)* de E_1; não fazemos uso do conjunto de indexação s do *array*.

$E \rightarrow E_1$ [E_2]	{ *E.tipo* := **se** E_2.*tipo* = *inteiro* **e** E_1.*tipo* = *array* (*s*, *t*) **então** *t* **senão** *tipo_erro* }

S	$\rightarrow$	**id** := E	{ *S. tipo* := **se id**·*tipo* = *E. tipo* **então** *vazio* **senão** *tipo_erro* }
S	$\rightarrow$	**if** E **then** S_1	{ *S. tipo* := **se** *E. tipo* = *booleano* **então** S_1·*tipo* **senão** *tipo_erro* }
S	$\rightarrow$	**while** E **do** S_1	{ *S. tipo* := **se** *E. tipo* = *booleano* **então** S_1·*tipo* **senão** *tipo_erro* }
S	$\rightarrow$	$S_1\ ;\ S_2$	{ *S. tipo* := **se** S_1·*tipo* = *vazio* **e** S_2·*tipo* = *vazio* **então** *vazio* **senão** *tipo_erro* }

Fig. 6.5. Esquema de tradução para a verificação de tipo de comandos.

Dentro das expressões, o operador posfixo $\uparrow$ produz o objeto apontado por seu operando. O tipo de $E \uparrow$ é o tipo t do objeto apontado pelo apontador E:

$E \rightarrow E_1 \uparrow$	{ *E.tipo* := **se** E_1.*tipo* = *apontador* **então** *t* **senão** *tipo_erro* }

Deixamos para o leitor a adição de produções e regras semânticas que permitam tipos e operações adicionais dentro das expressões. Por exemplo, para permitir que identificadores tenham o tipo *booleano*, podemos introduzir a produção $T \rightarrow$ **boolean** à gramática da Fig. 6.3. A introdução dos operadores de comparação, como <, e dos conectivos lógicos, como **and** (**e** lógico), nas produções para E permitiriam a construção de expressões do tipo *booleano*.

Verificação de Tipos dos Comandos

Uma vez que as construções de linguagem tais como os comandos não produzem valores, o tipo básico especial *vazio* pode lhes ser atribuído. Se um erro for detectado dentro de um comando, o tipo atribuído ao comando é *tipo_erro*.

Os comandos que consideramos são a atribuição, os condicionais e os comandos *while*. As seqüências de comandos são separadas por pontos-e-vírgulas. As produções da Fig. 6.5 podem ser combinadas com aquelas da Fig. 6.3 se modificarmos a produção para um programa completo em $P \rightarrow D;\ S$. Um programa agora consiste em declarações seguidas de comandos; as regras acima para a verificação de expressões são ainda necessárias porque os comandos podem ter expressões dentro de si.

As regras para a verificação de comandos são dadas na Fig. 6.5. A primeira verifica se os lados esquerdo e direito de um comando de atribuição têm o mesmo tipo.[3] A segunda e terceira regras especificam que as expressões nos comandos condicionais e comandos *while* precisam ter tipo *booleano*. Os erros são propagados pela última regra na Fig. 6.5 porque a seqüência de comandos possui tipo *vazio* somente se cada subcomando possuir tipo *vazio*. Nessas regras, uma desigualdade de tipos produz o tipo *tipo_erro;* um verificador de tipos amigável deveria, naturalmente, reportar a natureza e localização da desigualdade de tipos.

A Verificação de Funções

A aplicação de uma função a um argumento pode ser capturada pela produção

$$E \rightarrow E\ (E)$$

[3]Se uma expressão for permitida no lado esquerdo de uma atribuição, teremos então que distinguir entre valores-*l* e valores-*r*. Por exemplo, 1 := 2 é incorreto porque a constante 1 não pode receber uma atribuição.

na qual uma expressão é a aplicação de uma expressão à outra. As regras para associar expressões de tipo ao não-terminal T podem ser expandidas pelas seguintes produção e ação para permitir tipos de funções em declarações.

$$T \rightarrow T_1 \text{ '}\rightarrow\text{' } T_2 \qquad \{\ T.tipo := T_1.tipo \rightarrow T_2.tipo\ \}$$

Os apóstrofos em torno da seta usada como um construtor de função distinguem-na do mestassímbolo de uma produção.

A regra para verificar o tipo de uma aplicação de função é

$$E \rightarrow E_1\ (E_2) \qquad \{\ E.tipo := \textbf{se } E_2.tipo\ s \textbf{ e } E_1.tipo = s \rightarrow t \textbf{ então } t \textbf{ senão } tipo_erro\ \}$$

Esta regra diz que numa expressão formada pela aplicação de E_1 a E_2, o tipo de E_1 precisa ser uma função $s \rightarrow t$, proveniente do tipo s de E_2 para algum tipo intervalo t; o tipo de E_1 (E_2) é t.

Muitos temas relacionados à verificação de tipos em presença de funções podem ser discutidos com respeito à sintaxe simples acima. A generalização para funções com mais de um argumento é feita pela construção de um tipo produto consistindo nos argumentos. Note-se que os n argumentos de tipo $T_1, \ldots, T_n$ podem ser vistos como um único argumento do tipo $T_1 \times \ldots \times T_n$. Por exemplo, poderíamos escrever

```
raiz : ( real → real ) × real → real        (6.1)
```

para declarar uma função `raiz` que toma uma função dos reais para os reais e um real como argumentos e retorna um real. A sintaxe ao estilo de Pascal para esta declaração é

```
function raiz ( function f (real): real; x:
real): real
```

A sintaxe em (6.1) separa a declaração do tipo de uma função dos nomes de seus parâmetros.

6.3 EQUIVALÊNCIA DAS EXPRESSÕES DE TIPO

As regras de verificação na última seção possuem a forma, "**se** duas expressões de tipo forem iguais **então** retornar um certo tipo **senão** retornar *tipo_erro*". É por conseguinte importante se ter uma definição precisa de quando duas expressões de tipo são equivalentes. Ambigüidades potenciais afloram quando são dados nomes às expressões, os quais são usados em expressões de tipo subseqüentes. O tema-chave é se um nome numa expressão de tipo vale por si mesmo ou se é uma abreviação para uma outra expressão de tipo.

Uma vez que existe interação entre a noção de equivalência de tipos e a representação de tipos, vamos falar sobre ambas juntas. Por uma questão de eficiência, os compiladores usam representações permitindo uma rápida determinação da equivalência de tipos. A noção de equivalência de tipos implementada por um compilador específico pode freqüentemente ser explicada usando-se os conceitos de equivalência de nomes e equivalência estrutural, discutidas nesta seção. A discussão é em termos de uma representação sob a forma de grafos para as expressões de tipo, a qual deixa as folhas para os tipos básicos e para os nomes de tipos e os nós interiores para os construtores de tipos, como na Fig. 6.2. Como veremos, tipos definidos recursivamente levam a ciclos no grafo de tipos se um nome for tratado como uma abreviatura de uma expressão de tipo.

Equivalência Estrutural para as Expressões de Tipo

Na medida em que as expressões de tipo são construídas a partir dos tipos básicos e dos construtores, uma noção natural de equivalência entre duas expressões de tipos é a *equivalência estrutural;* isto é, duas expressões ou são do mesmo tipo básico ou são formadas pela aplicação do mesmo construtor a tipos estruturalmente equivalentes. Isto é, duas expressões de tipos são estruturalmente equivalentes se e somente se forem idênticas. Por exemplo, a expressão de tipo *inteiro* é equivalente somente a *inteiro* porque ambas representam o mesmo tipo básico. Similarmente, *apontador (inteiro)* é equivalente somente o *apontador (inteiro)* porque os dois são formados através da aplicação do mesmo construtor *apontador* a tipos equivalentes. Se usarmos o método dos números de valor do Algoritmo 5.1, para construir uma representação sob a forma de GDA para expressões de tipo, então expressões idênticas de tipos serão representadas pelo mesmo nó.

Na prática, freqüentemente são necessitadas modificações da noção de equivalência estrutural, de forma a refletir as regras efetivas de verificação de tipos de linguagem fonte. Por exemplo, quando os *arrays* são transmitidos como parâmetros, podemos desejar não incluir seus limites como parte do tipo.

O algoritmo para testar a equivalência estrutural na Fig. 6.6 pode ser adaptado para testar noções modificadas de equivalência. O algoritmo assume que os únicos construtores de tipos são os *arrays*, produtos, apontadores e funções. O algoritmo compara recursivamente a estrutura das expressões de tipo sem checar pelos ciclos, de forma que pode ser aplicado a uma representação de árvore ou GDA. Expressões de tipo idênticas não precisam ser representadas pelo mesmo nó no GDA. A equivalência estrutural dos nós em grafos de tipos com ciclos pode ser testada usando um algoritmo da Seção 6.7.

Os limites de *arrays* s_1 e t_1 em

$$s = array\ (s_1, s_2)$$
$$t = array\ (t_1, t_2)$$

são ignorados se o teste para a equivalência de *arrays* nas linhas 4 e 5 da Fig. 6.6. for reformulado para

senão se $s = array\ (s_1, s_2)$ **e** $t = array\ (t_1, t_2)$ **então**
retornar $sequiv\ (s_2, t_2)$

Em certas situações, podemos encontrar uma representação para expressões de tipo que seja significativamente mais compacta do que a notação de grafos de tipos. No próximo exemplo, algumas das informações provenientes das expressões de tipos são codificadas como uma seqüência de *bits*, que podem ser codificados como um único inteiro. A codificação é tal que inteiros distintos representam expressões de tipo estruturalmente não equivalentes. O teste para a equivalência estrutural pode ser acelerado testando-se primeiro pela inequivalência estrutural através da comparação das representações inteiras dos tipos e então aplicando o algoritmo da Fig. 6.6 somente se os inteiros forem os mesmos.

(1) **função** *sequiv* (s, t) : booleano;
início
(2) **se** s e t são o mesmo tipo básico **então**
(3) **retornar verdadeiro**
(4) **senão se** $s = array\ (s_1, s_2)$ **e** $t = array\ (t_1, t_2)$ **então**
(5) **retornar** $sequiv\ (s_1, t_1)$ **e** $sequiv\ (s_2, t_2)$
(6) **senão se** $s = s_1 \times s_2$ **e** $t = t_1 \times t_2$ **então**
(7) **retornar** $sequiv\ (s_1, t_1)$ **e** $sequiv\ (s_2, t_2)$
(8) **senão se** $s = apontador\ (s_1)$ **e** $t = apontador\ (t_1)$ **então**
(9) **retornar** $sequiv\ (s_1, t_1)$
(10) **senão se** $s = s_1 \rightarrow s_2$ **e** $t = t_1 \rightarrow t_2$ **então**
(11) **retornar** $sequiv\ (s_1, t_1)$ **e** $sequiv\ (s_2, t_2)$
senão
(12) **retornar falso**
fim

Fig. 6.6. O teste da equivalência estrutural das duas expressões de tipos s e t.

Exemplo 6.1 A codificação das expressões de tipo neste exemplo é proveniente de um compilador C escrito por D. M. Ritchie. É também usado pelo compilador C descrito em Johnson [1979].

Consideremos expressões de tipo com os seguintes construtores de tipos para apontadores, funções e *arrays: apontador (t)* denota um apontador para o tipo *t*, *frets (t)* denota uma função de alguns argumentos que retorna um objeto de tipo *t* e *array (t)* denota um *array* (de algum tamanho indeterminado) de elementos de tipo *t*. Note-se que simplificamos os construtores de tipos para o *array* e a função. Iremos controlar o número de elementos num *array*, mas tal número será mantido em algum local e dessa forma não é parte do construtor de tipo *array*. Similarmente, o único operando do construtor *frets* é o tipo do resultado de uma função; os tipos dos argumentos das funções serão armazenados em algum local. Por conseguinte, objetos com expressões estruturalmente equivalentes desse sistema de tipos poderão ainda falhar em atender o teste da Fig. 6.6, quando aplicados ao sistema de tipos mais detalhado lá usado.

Uma vez que cada um desses construtores é um operador unário, as expressões de tipo formadas pela aplicação desses construtores aos tipos básicos têm uma estrutura muito uniforme. Exemplos de tais expressões de tipos são:

caractere
frets (caractere)
apontador (frets (caractere))
array (apontador (frets (caractere)))

Cada uma das expressões acima pode ser representada por uma seqüência de *bits* representando um esquema de codificação simples. Como existem apenas três construtores de tipos, podemos usar dois *bits* para codificar um construtor como segue:

CONSTRUTOR DE TIPO	CODIFICAÇÃO
apontador	01
array	10
frets	11

Os tipos básicos de C são codificados usando-se quatro *bits* em Johnson [1979]; nossos quatro tipos básicos poderiam ser codificados como:

TIPO BÁSICO	CODIFICAÇÃO
booleano	0000
caractere	0001
inteiro	0010
real	0011

Expressões restritas de tipos podem ser agora codificadas como seqüências de *bits*. Os quatro *bits* mais à direita codificam os tipos básicos numa expressão de tipo. Indo-se da direita para a esquerda, os dois *bits* seguintes indicam o construtor aplicado ao tipo básico, os dois seguintes descrevem o construtor aplicado àquele último e assim por diante. Por exemplo,

EXPRESSÃO DE TIPO	CODIFICAÇÃO
caractere	000000 0001
frets (caractere)	000011 0001
apontador (frets (caractere))	000111 0001
array (apontador (frets (caractere)))	100111 0001

Ver o Exercício 6.12 para mais detalhes.

Além de economizar espaço, tal representação controla os construtores que aparecem em qualquer expressão de tipo. Duas diferentes seqüências de *bits* não podem representar o mesmo tipo porque ou o tipo básico ou construtores na expressão de tipo são diferentes. Naturalmente, tipos diferentes poderiam ter a mesma seqüência de *bits* uma vez que o tamanho de um *array* e os argumentos de uma função não estão representados.

A codificação desse exemplo pode ser estendida de forma a incluir tipos de registros. A idéia é tratar cada registro como um tipo básico na codificação; uma seqüência separada de *bits* codifica o tipo de cada campo no registro. A equivalência de tipos em C é examinada posteriormente no Exemplo 6.4. □

Nomes para Expressões de Tipos

Em algumas linguagens, podem ser dados nomes aos tipos. Por exemplo, no fragmento de programa Pascal

```
type   link  =  ↑ celula;
var proximo  :  link;
   anterior  :  link;
          p  :  ↑ celula;
       q, r  :  ↑ celula;                    (6.2)
```

o identificador `link` é declarado como sendo um nome para o tipo ↑ célula. A questão desponta: as variáveis, `proximo`, `anterior`p, `q` e `r`, têm todas o mesmo tipo? Surpreendentemente, a resposta depende da implementação. O problema emergiu porque o Relatório Pascal não definiu o termo "tipo idêntico".

Para modelar esta situação, permitiremos que expressões de tipo recebam nomes e também que esses nomes figurem em expressões de tipo, onde previamente tínhamos apenas tipos básicos. Por exemplo, se `celula` é o nome de uma expressão de tipo, *apontador* (`celula`) é uma expressão de tipo. Por hora, suponhamos que não existam definições circulares de expressões de tipo, tais como uma que defina `celula` como sendo um nome de uma expressão de tipo contendo `celula`.

Quando os nomes são permitidos em expressões de tipo, duas noções de equivalência emergem, dependendo do tratamento dado aos nomes. A *equivalência por nome* enxerga cada nome de tipo como o de um tipo distinto e, dessa forma, duas expressões são nomes equivalentes se, e somente se, os nomes forem idênticos. Sob a ótica da *equivalência estrutural*, os nomes são substituídos pelas expressões de tipo que definem e, dessa forma, duas expressões são estruturalmente equivalentes se representarem duas expressões de tipo estruturalmente equivalente após a substituição de todos os nomes.

Exemplo 6.2. As expressões de tipo que poderiam ser associadas às variáveis nas declarações (6.2) são dadas na seguinte tabela.

VARIÁVEL	EXPRESSÃO DE TIPO
`proximo`	`link`
`anterior`	`link`
`p`	*apontador* (`celula`)
`q`	*apontador* (`celula`)
`r`	*apontador* (`celula`)

Sob a equivalência por nome, as variáveis `proximo` e `anterior` têm o mesmo tipo, porque foram associadas às mesmas expressões de tipo. As variáveis `p`, `q` e `r` também têm o mesmo tipo, mas `p` e `proximo` não, uma vez que suas expressões de tipo associadas são diferentes. Sob a equivalência estrutural, todas as cinco variáveis têm o mesmo tipo porque `link` é um nome para a expressão de tipo *apontador* (`celula`). □

Os conceitos de equivalência por nome e estrutural são úteis para se explicar as regras usadas pelas várias linguagens a fim de associar tipos aos identificadores que figuram nas declarações.

Exemplo 6.3. A confusão emerge em Pascal a partir do fato de que muitas implementações associam um nome implícito de tipo a cada identificador declarado. Se uma declaração contém uma expressão de tipo que não seja um nome, um nome implícito é criado. Um novo nome implícito é criado a cada vez que uma expressão de tipo figurar numa declaração de variável.

Por conseguinte, nomes implícitos são criados para as expressões de tipo nas duas declarações contendo `p`, `q` e `r` em (6.2). Isto é, as declarações são tratadas como se fossem

```
type link      =  ↑ celula;
     np        =  ↑ celula;
     nqr       =  ↑ celula;
var  proximo   :  link;
     anterior  :  link;
     p         :  np;
     q         :  nqr;
     r         :  nqr;
```

Aqui, os novos nomes de tipo `np` e `nqr` foram introduzidos. Sob a equivalência por nome, uma vez que `proximo` e `anterior` foram declarados com o mesmo nome de tipo, ambos são tratados como tendo tipos equivalentes. Similarmente, `q` e `r` são tratados como tendo tipos equivalentes porque o mesmo nome implícito de tipo está associado aos mesmos. No entanto, `p`, `q` e `proximo` não têm tipos equivalentes, já que todos têm tipos com nomes diferentes.

A implementação típica constrói um grafo de tipos para representá-los. A cada vez que um construtor de tipos ou um tipo básico for enxergado, um novo nó é criado. A cada vez que um novo tipo é visto, uma folha é criada, mas, entrementes, controlamos a expressão de tipo à qual o nome da expressão se refere. Com essa representação, duas expressões de tipo são equivalentes se forem representadas pelo mesmo nó no grafo de tipos. A Fig. 6.7 mostra um grafo de tipos para as declarações de (6.2). Linhas pontilhadas mostram a associação entre as variáveis e os nós no grafo de tipos. Note-se que o nome de tipo `celula` possui três pais, todos rotulados *apontador*. Um sinal de igual aparece entre o nome de tipo `link` e o nó no grafo de tipos ao qual o mesmo se refere. □

Ciclos na Representação de Tipos

As estruturas de dados básicos, como as listas ligadas e árvores, são freqüentemente definidas recursivamente; por exemplo, uma lista ligada ou é vazia ou consiste em uma célula com um apontador para uma lista ligada. Tais estruturas de dados são usualmente implementadas utilizando-se registros, que contêm apontadores para registros similares, e os nomes de tipos desempenham um papel essencial na definição dos tipos de tais registros.

Consideremos uma lista ligada de células, cada uma contendo algumas informações do tipo inteiro e um apontador para a próxima célula na lista. As declarações Pascal dos nomes de tipo correspondentes a `links` e células são:

```
type  link     =  ↑ celula;
      celula   =  record
                     info : integer;
                  proximo : link
                end;
```

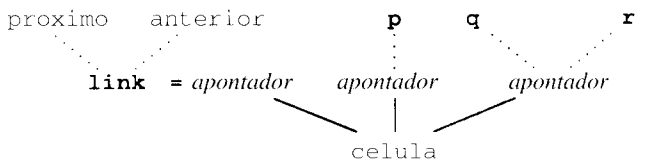


Fig. 6.7. Associação de variáveis e nós no grafo de tipos.

Note-se que o nome de tipo `link` é definido em termos de `celula` e que `celula` é definida em termos de `link`, e, por conseguinte, suas definições são recursivas.

Nomes de tipos recursivamente definidos podem ser substituídos se estivermos ávidos por introduzir ciclos nos grafos de tipos. Se *apontador* `(celula)` for substituído por `link`, a expressão de tipo mostrada na Fig. 6.8(a) é obtida para `celula`. Usando ciclos, como na Fig. 6.8(b), podemos eliminar a menção a `celula` da parte do grafo de tipos abaixo do nó rotulado *registro*.

Exemplo 6.4. C evita ciclos nos grafos de tipos pelo uso da equivalência estrutural para todos os tipos, exceto registros. Em C, a declaração de `celula` seria como

```
struct celula  {
     int info;
     struct celula *próximo;
};
```

C usa a palavra-chave `struct` ao invés de `record` e o nome `celula` se torna parte do tipo do registro. Com efeito, C usa a representação acíclica da Fig. 6.8(a).

C requer que os nomes de tipos sejam declarados antes de serem usados, exceção feita aos apontadores para tipos registro não declarados. Todos os ciclos potenciais, conseqüentemente, são devidos a apontadores para registros. Uma vez que o nome de um registro é parte de seu tipo, o teste para a equivalência estrutural pára quando um construtor de registro é atingido — ou os tipos sendo comparados são equivalentes porque são o mesmo tipo de registro designado pelo mesmo nome ou são inequivalentes. □

6.4 CONVERSÕES DE TIPO

Consideremos expressões como `x + i`, onde `x` é do tipo real e `i` do tipo inteiro. Como a representação dos inteiros e reais é diferente dentro do computador e diferentes instruções de máquina são usadas para as operações sobre inteiros e reais, o compilador pode ter que primeiro converter um dos operandos de + para assegurar que ambos os operandos sejam do mesmo tipo quando a adição tiver lugar.

A definição da linguagem especifica quais as conversões que são necessárias. Quando um inteiro é atribuído a um real ou vice-versa, a

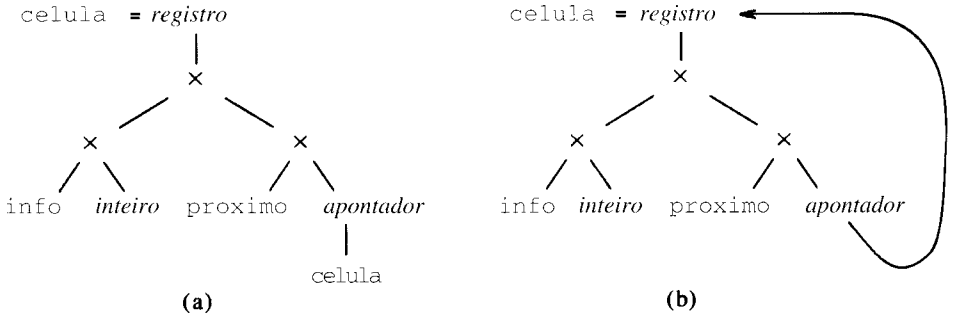


Fig. 6.8. Nome de tipo `celula` recursivamente definido.

conversão deve ser feita para o tipo do lado esquerdo da atribuição. Nas expressões, a transformação usual converte um inteiro para um número real e, em seguida, realiza a operação no par resultante de operandos reais. O verificador de tipos num compilador pode ser usado para inserir essas operações de conversão na representação intermediária do programa-fonte. Por exemplo, a notação posfixa para `x + i` poderia ser

`x i` **inttoreal** +

Aqui, o operador `x i` **inttoreal** + converte `i` de inteiro para real e em seguida **real**+ realiza a adição real de seus operandos.

A conversão de tipos freqüentemente emerge em outro contexto. Um símbolo que tenha diferentes significados dependendo do seu contexto é dito sobrecarregado. A sobrecarga será discutida na próxima seção, mas é mencionada aqui porque as conversões de tipo freqüentemente acompanham a sobrecarga.

Coerções

A conversão de um tipo para outro é dita *implícita* se for realizada automaticamente pelo compilador. Conversões implícitas de tipo, também chamadas de *coerções*, estão limitadas em muitas linguagens às situações onde nenhuma informação é perdida em princípio; por exemplo, um inteiro pode ser convertido para um real mas não vice-versa. Na prática, entretanto, alguma perda é possível quando um número real precisa caber no mesmo número de *bits* que o inteiro.

A conversão é dita *explícita* se o programador precisa escrever alguma coisa para provocar a conversão. Para todos os propósitos práticos, todas as conversões em Ada são explícitas. As conversões explícitas se parecem exatamente com aplicações de funções para um verificador de tipos, e dessa forma não apresentam problemas novos.

Por exemplo, em Pascal, a função embutida na linguagem `ord` mapeia um caractere em um inteiro e `chr` realiza o mapeamento inverso, de um inteiro para um caractere, e assim essas conversões são explícitas. C, por outro lado, coage (isto é, converte implicitamente) os caracteres ASCII para inteiros na faixa entre `0` e `127` nas expressões aritméticas.

Exemplo 6.5. Consideremos expressões formadas pela aplicação do operador aritmético **op** a constantes e identificadores, como na gramática da Fig. 6.9. Suponhamos que existam dois tipos — real e inteiro, com os inteiros convertidos para reais quando necessário. O atributo *tipo* para o não-terminal E pode ser inteiro ou real e as regras de verificação de tipos são mostradas na Fig. 6.9. Como na Seção 6.2, a função *procurar* (*e*) retorna o tipo guardado na entrada da tabela de símbolos apontada por *e*. □

PRODUÇÃO	REGRA SEMÂNTICA
$E \rightarrow$ **num**	$E.tipo$:= *inteiro*
$E \rightarrow$ **num . num**	$E.tipo$:= *real*
$E \rightarrow$ **id**	$E.tipo$:= *procurar* (**id**.*entrada*)
$E \rightarrow E_1$ **op** E_2	$E.tipo$:= **se** $E_1.tipo$ = *inteiro* **e** $E_2.tipo$ = *inteiro* **então** *inteiro* **senão se** $E_1.tipo$ = *inteiro* **e** $E_2.tipo$ = *real* **então** *real* **senão se** $E_1.tipo$ = *real* **e** $E_2.tipo$ = *inteiro* **então** *real* **senão se** $E_1.tipo$ = *real* **e** $E_2.tipo$ = *real* **então** *real* **senão** *tipo_erro*

Fig. 6.9. Regras para a verificação de tipos para a coerção de inteiro para real.

A conversão implícita de constantes pode usualmente ser feita em tempo de compilação, freqüentemente com um grande ganho no tempo de execução do programa objeto. Nos seguintes fragmentos de código, `X` é um *array* de reais que está sendo inicializado todo em 1's. Usando um compilador Pascal, Bentley [1982] encontrou que o fragmento de código

```
for I : = 1 to N do X[I] := 1
```

levava 48.4N microssegundos para executar, enquanto que o fragmento

```
for I : = 1 to N do X[I] := 1.0
```

levava 5.4N. Ambos os fragmentos atribuem o valor um aos elementos de um *array* de reais. No entanto, o código gerado (por este compilador) para o primeiro fragmento continha uma chamada para uma rotina em tempo de execução que convertia a representação inteira de 1 numa representação sob a forma de número real. Uma vez que é sabido em tempo de compilação que `X` é um *array* de reais, um compilador mais completo converteria 1 para 1.0 em tempo de compilação.

6.5 SOBRECARGA DE FUNÇÕES E OPERADORES

Um símbolo *sobrecarregado* é aquele que possui diferentes significados dependendo de seu contexto. Na Matemática, o operador de adição + é sobrecarregado, porque + em $A + B$ possui diferentes significados quando A e B são inteiros, reais, números complexos ou matrizes. Em Ada, os parêntesis ( ) são sobrecarregados; a expressão `A(I)` pode significar o `i`ésimo elemento do *array* `A`, uma chamada para a função `A` com argumento `I` ou uma conversão explícita da expressão `I` para o tipo `A`.

A sobrecarga é *resolvida* quando um único significado para uma ocorrência de um símbolo sobrecarregado é determinado. Por exemplo, se + puder denotar a adição inteira ou a real, então as duas ocorrências de + em `x ( i + j)` podem denotar diferentes formas de adição, dependendo dos tipos de `x`, `i` e `j`. A resolução da sobrecarga é algumas vezes mencionada como *identificação de operador*, pois determina que operador um símbolo de operação denota.

Os operadores aritméticos são sobrecarregados na maioria das linguagens. No entanto, a sobrecarga envolvendo operadores como + pode ser resolvida olhando-se somente para os argumentos do operador. A análise de caso para se determinar o uso da versão inteira ou real de + é similar àquela na regra semântica para $E \rightarrow E_1$ **op** E_2 na Fig. 6.9, onde o tipo de E é determinado pelo exame dos possíveis tipos para E_1 e E_2.

O Conjunto de Possíveis Tipos para uma Subexpressão

Nem sempre é possível se resolver a sobrecarga somente através do exame dos argumentos de uma função, como o próximo exemplo mostra. Em lugar de um único tipo, uma subexpressão figurando sozinha pode ter um conjunto de possíveis tipos. Em Ada, o contexto precisa providenciar informações suficientes para estreitar o leque de escolhas para um único tipo.

Exemplo 6.6. Em Ada, uma das interpretações padrão (isto é, embutidas) do operador * é aquela de uma função de um par de inteiros para um inteiro. O operador pode ser sobrecarregado pela adição de declarações como as seguintes:

PRODUÇÃO	REGRA SEMÂNTICA
$E' \rightarrow E$	$E'.tipos := E.tipos$
$E \rightarrow$ **id**	$E.tipos := procurar$ (**id**.*entrada*)
$E \rightarrow E_1 (E_2)$	$E.tipos := \{ t$ exista um s em $E_2.tipos$ tal que $s \rightarrow t$ esteja em $E_1.tipos \}$

Fig. 6.10. Determinando o conjunto de possíveis tipos de uma expressão.

```
function "*" ( i, j : integer ) return complex;
function "*" ( x, y : complex ) return complex;
```

Após as declarações acima, os possíveis tipos para $*$ incluem:

$$\begin{array}{rcl} \textit{inteiro} \times \textit{inteiro} & \rightarrow & \textit{inteiro} \\ \textit{inteiro} \times \textit{inteiro} & \rightarrow & \textit{complexo} \\ \textit{complexo} \times \textit{complexo} & \rightarrow & \textit{complexo} \end{array}$$

Suponhamos que o único tipo possível para `2`, `3` e `5` seja inteiro. Com as declarações acima, a subexpressão `3*5` ou tem o tipo inteiro ou tem o tipo complexo, dependendo de seu contexto. Se a expressão completa for `2*( 3*5 )`, `3*5` terá necessariamente o tipo inteiro porque $*$ ou toma um par de inteiros ou um par de números complexos como argumento. Por outro lado, 3*5 precisará ter o tipo complexo se a expressão completa for `( 3*5 )*z` e `z` for declarado como complexo. □

Na Seção 6.2, assumimos que cada expressão tinha um único tipo, e, por conseguinte, a regra de checagem de tipos para uma aplicação de função era:

$$E \rightarrow E_1 (E_2) \qquad \{ E.tipo := \textbf{se } E_2.tipo = s \textbf{ e } E_1.tipo = s \rightarrow t \textbf{ então } t \textbf{ senão } tipo_erro \}$$

A generalização natural desta regra para conjuntos de tipos é mostrada na Fig. 6.10. A única operação na Fig. 6.10 é a de aplicação de uma função; as regras para verificar outros operadores em expressões são similares. Podem haver várias declarações para um operador sobrecarregado, de forma que assumimos que uma entrada numa tabela de símbolos possa conter um conjunto de tipos possíveis; esse conjunto é retornado pela função *procurar*. O não-terminal de partida E' gera a expressão completa. Seu papel é esclarecido abaixo.

Expressa em palavras, a terceira regra da Fig. 6.10 diz que se s for um dos tipos de E_2 e um dos tipos de E_1 pode mapear s em t, t é um dos tipos de E_1 (E_2). Um não batimento de tipos durante uma aplicação de função resulta em que o conjunto *E.tipos* se torne vazio, uma condição que usamos temporariamente para sinalizar um tipo-erro.

Exemplo 6.7. Além de ilustrar a especificação da Fig. 6.10, este exemplo sugere como a abordagem se estende a outras construções. Em particular, consideramos a expressão `3*5`. Sejam as declarações do operador $*$ como no Exemplo 6.6. Isto é, $*$ pode mapear um par de inteiros quer para um inteiro, quer para um número complexo, dependendo do contexto. O conjunto de possíveis tipos para as subexpressões de `3*5` é mostrado na Fig. 6.11, onde i e c abreviam *inteiro* e *complexo*, respectivamente.

De novo, suponhamos que o único tipo possível para `3` e `5` seja *inteiro*. O operador $*$ é, por conseguinte, aplicado a um par de inteiros. Se tratarmos este par de inteiros como uma unidade, seu tipo será dado por *inteiro* $\times$ *inteiro*. Existem duas funções no conjunto de tipos para $*$ que se aplicam a pares de inteiros; uma retorna um inteiro enquanto

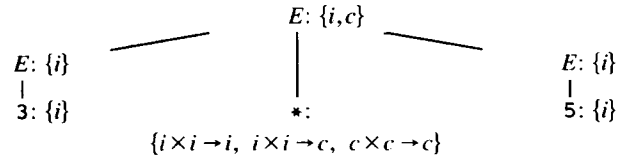


Fig. 6.11. Conjunto de possíveis tipos para a expressão `3*5`.

que a outra retorna um número complexo, e, dessa forma, a raiz pode ter ou o tipo *inteiro* ou o tipo *complexo*. □

Estreitando o Conjunto de Possíveis Tipos

Ada requer que uma expressão completa tenha um único tipo. Dado um único tipo a partir do contexto, podemos estreitar as escolhas de tipos para cada subexpressão. Se esse processo não resultar num único tipo para cada subexpressão, um erro é declarado para toda a expressão.

Antes de trabalharmos de cima para baixo, de uma expressão para suas subexpressões, faremos um exame detalhado nos conjuntos *E. tipos* construídos pelas regras da Fig. 6.10. Mostramos que cada tipo t em *E. tipos* é um tipo *viável*; isto é, é possível escolher dentre os tipos sobrecarregados dos identificadores em E de uma forma em que E assuma o tipo t. A propriedade vigora para os identificadores por declaração, uma vez que cada elemento de **id**.*tipos* é viável. Para o passo indutivo, consideremos o tipo t em *E.tipos*, onde E é E_1 (E_2). Da regra para a aplicação de função na Fig. 6.10 para algum tipo s, s precisará estar em $E_2.tipos$ e um tipo $s \rightarrow t$ precisará estar em $E_1.tipos$. Por indução, s e $s \rightarrow t$ são tipos viáveis para E_1 e E_2, respectivamente. Segue que t é um tipo viável para E.

Pode haver várias maneiras de se chegar a um tipo viável. Por exemplo, consideremos a expressão `f(x)` onde `f` pode ter os tipos $a \rightarrow c$ e $b \rightarrow c$ e `x` pode ter os tipos a e b. Então, `f(x)` possui tipo c mas `x` pode ter o tipo a ou b.

A definição dirigida pela sintaxe da Fig. 6.12 é obtida a partir daquela da Fig. 6.10 pela adição de regras semânticas para determinar o atributo herdado *único* de E. O atributo sintetizado *código* de E é discutido abaixo.

Uma vez que toda a expressão é gerada por E', desejamos que $E'.tipos$ seja um conjunto contendo um único tipo t. Este único tipo é

PRODUÇÃO	REGRAS SEMÂNTICAS
$E' \rightarrow E$	$E'.tipos := E.tipos$ $E.único :=$ **se** $E'.tipos = \{t\}$ **então** t **senão** *tipo_erro* $E'.código := E.código$
$E \rightarrow$ **id**	$E.tipos := procurar$ (**id**.*entrada*) $E.código := gerar$ (**id**.*lexema* ':' *E.único*)
$E \rightarrow E_1 (E_2)$	$E.tipos := \{ s'$ exista um s em $E_2.tipos$ tal que $s \rightarrow s'$ esteja em $E_1.tipos \}$ $t := E.único$ $S := \{ s \mid s \in E_2.tipos$ **e** $s \rightarrow t \in E_1.tipos \}$ $E_2.único :=$ **se** $S = \{s\}$ **então** s **senão** *tipo_erro* $E_1.único :=$ **se** $S = \{s\}$ **então** $s \rightarrow t$ **senão** *tipo_erro* $E.código := E_1.código \parallel E_2.código \parallel$ *gerar* (' **aplicar** ' ':' *E.único*)

Fig. 6.12. Estreitando o conjunto de tipos de uma expressão.

herdado como o valor de *E.único*. Novamente, o tipo básico *tipo_erro* assinala um erro.

Se uma função E_1 (E_2) retorna o *tipo t*, podemos encontrar um tipo *s* que seja viável para o argumento E_2; ao mesmo tempo, $s \rightarrow t$ é viável para a função. O conjunto *S* na regra semântica correspondente na Fig. 6.12 é usado para checar se existe um único tipo *s* com essa propriedade.

A definição dirigida pela sintaxe na Fig. 6.12 pode ser implementada realizando duas travessias em profundidade da árvore sintática para uma expressão. Durante a primeira travessia, o atributo *tipos* é sintetizado de baixo para cima. Durante a segunda, o atributo *único* é propagado de cima para baixo e, na medida em que retornarmos de um nó, o atributo *código* pode ser sintetizado. Na prática, o verificador de tipos pode simplesmente atrelar um único tipo a cada nó da árvore sintática. Na Fig. 6.12, geramos uma notação posfixa para sugerir como o código intermediário poderia ser gerado. Na notação posfixa, cada identificador e instância do operador **aplicar** possui um tipo atrelado a si pela função *gerar*.

6.6 FUNÇÕES POLIMÓRFICAS

Um procedimento ordinário permite que os comandos em seu corpo sejam executados com argumentos de tipos fixos; a cada vez que um procedimento polimórfico é chamado, os enunciados em seu corpo podem ser executados com argumentos de diferentes tipos. O termo "polimórfico" pode também ser aplicado a qualquer trecho de código que possa ser executado com argumentos de tipos diferentes e dessa forma podemos falar igualmente de funções e de operadores polimórficas.

Operadores embutidos para indexar *arrays*, aplicar funções e manipular apontadores são usualmente polimórficos, porque não estão restritos a um tipo particular de *array*, função ou apontador. Por exemplo, o manual de referência de C estabelece a respeito do operador de endereçamento &: "se o tipo do operando for '...', o tipo do resultado é um apontador para '...'". Uma vez que qualquer tipo pode ser substituído em lugar de '...', o operador & em C é polimórfico.

Em Ada, as funções "genéricas" são polimórficas, mas o polimorfismo em Ada é restrito. Como o termo "genérico" também tem sido usado para referenciar as funções sobrecarregadas e a coerção dos argumentos de funções, iremos evitar o uso desse termo.

Esta seção endereça os problemas que emergem no projeto de um verificador de tipos para uma linguagem com funções polimórficas. Para lidarmos com o polimorfismo, expandiremos o nosso conjunto de expressões de tipo para incluir expressões com variáveis de tipo. A inclusão de variáveis de tipo levanta alguns temas algorítmicos relacionados à equivalência de expressões.

Por que Funções Polimórficas?

As funções polimórficas são atraentes porque facilitam a implementação de algoritmos que manipulam estruturas de dados, independentemente dos tipos de seus elementos. Por exemplo, é conveniente termos um programa que determine o comprimento de uma lista sem termos que saber os tipos dos elementos da mesma.

Linguagens como Pascal requerem especificações completas dos tipos dos parâmetros de funções, de forma que uma função, para determinar o comprimento de uma lista ligada de inteiros, não pode ser aplicada a uma lista de reais. O código Pascal da Fig. 6.13 é para listas de inteiros. A função `comprimento` segue a cadeia de apontadores para o próximo elemento da lista até que um *link* com valor `nil` seja atingido. Apesar da função não depender de maneira alguma do tipo de informação numa célula, Pascal requer que o tipo de `info` seja declarado quando a função `comprimento` for escrita.

Numa linguagem com funções polimórficas, como ML (Milner [1984]), a função `comprimento` pode ser escrita de forma a se aplicar a qualquer tipo de lista, como mostrado na Fig. 6.14. A palavra-

```
type link = ↑ celula
   celula = registro
                  info : integer;
                  proximo : link
     end;
function comprimento ( lptr : link ) : integer;
var comp : integer;
begin
     comp = 0;
     while lptr <> nil do begin
           comp := comp + 1;
           lptr := lptr ↑·proximo
     end;
     comprimento := comp
end;
```

Fig. 6.13. Programa Pascal para o comprimento de uma lista.

chave `fun` indica que `comprimento` é uma função recursiva. As funções `null` e `tl` são pré-definidas; `null` testa se uma lista está vazia e `tl` retorna o resto da lista após o primeiro elemento ter sido removido. Com a definição mostrada na Fig. 6.14, as duas seguintes aplicações da função `comprimento` produzem 3:

```
comprimento (["sol", "lua", "ter"]);
comprimento ([10, 9, 8]);
```

Na primeira, `comprimento` é aplicada a uma lista de cadeias de caracteres; na segunda, é aplicada a uma lista de inteiros.

Variáveis de Tipo

As variáveis representando expressões de tipo nos permitem falar a respeito de tipos desconhecidos. No resto desta seção, iremos usar as letras gregas, α, β etc., para variáveis de tipo em expressões de tipo.

Uma aplicação importante das variáveis de tipo é a de verificar o uso consistente dos identificadores numa linguagem que não requeira que os identificadores sejam declarados antes de serem usados. Uma variável representa o tipo do identificador não declarado. Examinando o programa, podemos dizer se o identificador não declarado é usado, digamos, como um inteiro num enunciado e como um *array* num outro. Tal utilização inconsistente pode ser reportada como um erro. Por outro lado, se a variável for sempre usada como um inteiro, então não somente asseguramos o uso consistente; no processo, inferimos o que seu tipo tem que ser.

A *inferência de tipos* é o problema de determinar o tipo de uma construção de linguagem a partir da forma que for usada. O termo é freqüentemente aplicado ao problema de inferir o tipo de uma função a partir de seu corpo.

Exemplo 6.8. As técnicas de inferência de tipos podem ser aplicadas a programas em linguagens como C e Pascal para preencher as informações de tipo ausentes em tempo de compilação. O fragmento de código da Fig. 6.15 mostra o procedimento `mlist`, que possui um parâmetro `p` que é ele próprio um procedimento; em particular, não conhecemos o número ou tipos dos argumentos tomados por `p`. Tais especificações incompletas do tipo de `p` são permitidas por C e pelo manual de referência de Pascal.

O procedimento `mlist` aplica o parâmetro `p` a cada célula numa lista ligada. Por exemplo, `p` pode ser usado para inicializar ou impri-

```
fun comprimento ( lptr ) =
     if null ( lptr ) then 0
     else comprimento ( tl ( lptr )) + 1;
```

Fig. 6.14. Programa ML para o comprimento de uma lista.

mir um inteiro guardado numa célula. A despeito do fato dos tipos dos argumentos de `p` não serem especificados, podemos inferir, a partir do uso de `p` na expressão `p (lptr)`, que o tipo de `p` terá que ser:

$$\texttt{link} \rightarrow \textit{vazio}^*$$

Qualquer chamada de `mlist` com um parâmetro do tipo procedimento que não tenha esse tipo é um erro. Um procedimento pode ser pensado como uma função que não retorne um valor, e, por conseguinte, o tipo do seu resultado é *vazio*. □

As técnicas para a inferência e verificação de tipos possuem muito em comum. Em cada caso, temos de lidar com expressões de tipo contendo variáveis. Uma argumentação similar àquela do exemplo seguinte é usada mais adiante nesta seção por um verificador de tipos a fim de inferir os tipos representados pelas variáveis.

Exemplo 6.9. Um tipo pode ser inferido para a função polimórfica `derref` no seguinte pseudoprograma. A função `derref` possui o mesmo efeito que o operador Pascal ↑ para o derrefenciamento de apontadores.

```
function derref(p);
begin
     return p↑
end;
```

quando a primeira linha

```
function derref(p);
```

é vista, não sabemos nada a respeito do tipo de `p`, e então vamos representá-lo por uma variável de tipo β. Por definição, o operador posfixo ↑ toma um apontador para um objeto e retorna o objeto. Uma vez que o operador ↑ é aplicado a `p` na expressão `p↑`, segue que `p` tem que ser um apontador para um objeto de tipo desconhecido α, e, então, precisamos aprender que

$$\beta = \textit{apontador}\ (\alpha)$$

onde α é uma outra variável de tipo. Sobretudo, a expressão `p↑` possui tipo α, e, por conseguinte, podemos escrever a expressão de tipo

$$\text{para qualquer tipo } \alpha,\ \textit{apontador}\ (\alpha) \rightarrow \alpha \qquad (6.3)$$

para o tipo da função `derref`. □

Uma Linguagem com Funções Polimórficas

Tudo o que dissemos até então a respeito de funções polimórficas é que podem ser executadas com argumentos de "tipos diferentes". Enunciados precisos sobre o conjunto de tipos a que uma função polimórfica pode ser aplicada são feitos usando-se o símbolo ∀, significando "para qualquer tipo". Conseqüentemente,

$$\forall\alpha.\ \textit{apontador}\ (\alpha) \rightarrow \alpha \qquad (6.4)$$

é como escrevemos a expressão de tipo (6.3) para o tipo da função `derref` no Exemplo 6.9. A função polimórfica `comprimento` na Fig. 6.14 toma uma lista de elementos de qualquer tipo e retorna um inteiro, e seu tipo pode ser escrito como

$$\forall\alpha.\ \textit{lista}\ (\alpha) \rightarrow \textit{inteiro} \qquad (6.5)$$

*Do original em inglês, e utilizado na sintaxe de C, *void*. (N. do T.)

```
type link ↑ celula;
procedure mlist ( lptr : link; procedure p ) ;
begin
    while lptr <> nil do begin
        p ( lptr );
        lptr := lptr↑.proximo
    end
end;
```

Fig. 6.15. Procedimento `mlist` com parâmetro tipo procedimento `p`.

Aqui, *lista* é um construtor de tipos. Sem o símbolo ∀, podemos apenas dar exemplos de possíveis domínios e intervalos de tipos para `comprimento`:

$$\begin{aligned} \textit{lista (inteiro)} &\rightarrow \textit{inteiro} \\ \textit{lista (lista (caractere))} &\rightarrow \textit{inteiro} \end{aligned}$$

As expressões como (6.5) são os enunciados mais gerais que podemos fazer a respeito do tipo de uma função polimórfica.

O símbolo ∀ é o *quantificador universal* e a variável de tipo à qual é aplicado é dita estar *ligada* (ou *amarrada*) ao mesmo. As variáveis ligadas podem ser renomeadas à vontade, providenciado que todas as ocorrências da variável sejam renomeadas. Por conseguinte, na expressão de tipo

$$\forall\gamma.\ \textit{apontador}\ (\gamma) \rightarrow \gamma$$

é equivalente a (6.4). Uma expressão de tipo com o símbolo ∀ na mesma será referenciada informalmente como um "tipo polimórfico".

A linguagem que iremos usar para verificar funções polimórficas é gerada pela gramática da Fig. 6.16.

Os programas gerados por esta gramática consistem em uma seqüência de declarações seguidas pela expressão *E* a ser verificada, por exemplo:

$$\begin{aligned} &\texttt{derref}: \ \forall\alpha.\ \textit{apontador}\ (\alpha) \rightarrow \alpha; \\ &\quad\ \texttt{q}: \ \textit{apontador(apontador (inteiro));} \\ &\texttt{derref(derref(q))} \end{aligned} \qquad (6.6)$$

Minimizamos a notação fazendo com que o não-terminal *T* gere expressões diretamente. Os construtores → e × formam funções e produtos de tipos. Os construtores unários, representados por **construtor_unário**, permitem que tipos como *apontador (inteiro)* e *lista (inteiro)* sejam escritos. Os parênteses são usados simplesmente para agrupar os tipos. As expressões cujos tipos devem ser verificados possuem uma sintaxe muito simples: podem ser identificadores, seqüências de expressões formando uma tupla ou aplicação de uma função a um argumento.

As regras de verificação de tipos para as funções polimórficas diferem de três maneiras daquelas para as funções ordinárias da Seção

$$\begin{aligned} P &\rightarrow D\ ;\ E \\ D &\rightarrow D\ ;\ D \mid \textbf{id} : Q \\ Q &\rightarrow \forall\textbf{variável_de_tipo} \cdot Q \mid T \\ T &\rightarrow T\ '\rightarrow'\ T \\ &\mid T \times T \\ &\mid \textbf{construtor_unário}\ (\ T\ ) \\ &\mid \textbf{tipo_básico} \\ &\mid \textbf{variável_de_tipo} \\ &\mid (\ T\ ) \\ E &\rightarrow E\ (\ E\ ) \mid E\ ,\ E \mid \textbf{id} \end{aligned}$$

Fig. 6.16. Gramática para linguagem com funções polimórficas.

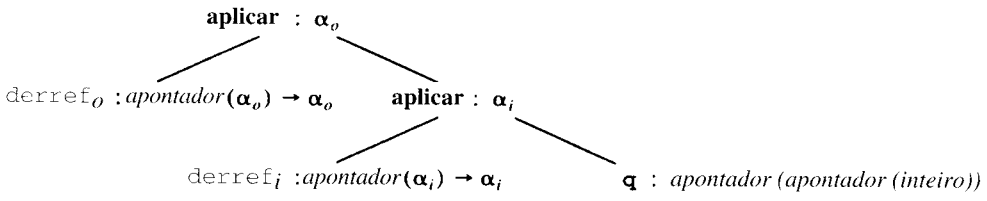


Fig. 6.17. Árvore sintática rotulada para `derref(derref(q))`.

6.2. Antes de apresentá-las, ilustramos essas diferenças considerando a expressão `derref(derref(q))` no programa (6.6). Uma árvore sintática para esta expressão é mostrada na Fig. 6.17. Atrelados a cada nó existem dois rótulos. O primeiro nos informa a subexpressão representada pelo nó e o segundo é uma expressão de tipo associada à subexpressão. Os subscritos o e i distinguem as ocorrências mais externas e mais internas de `derref`, respectivamente.

As diferenças para as regras das funções ordinárias são:

1. As ocorrências distintas para uma função polimórfica na mesma expressão não precisam ter argumentos do mesmo tipo. Na expressão `derref`$_o$ `(derref`$_i$`(q))`, `derref`$_i$ remove um nível de indireção do apontador, de tal forma que `derref`$_o$ é aplicada um argumento de tipo diferente. A implementação dessa propriedade está baseada na interpretação de $\forall\alpha$ como "para qualquer tipo α". Cada ocorrência de `derref` possui sua própria perspectiva sobre o que a variável ligada α em (6.4) significa. Por conseguinte, atribuímos a cada ocorrência de `derref` uma expressão de tipo formada pela substituição de α em (6.4) por uma variável nova e removendo o quantificador $\forall$ no processo. Na Fig. 6.17, as variáveis novas α_o e α_i são usadas nas expressões de tipo atribuídas às ocorrências mais externas e mais internas de `derref`, respectivamente.

2. Uma vez que as variáveis podem aparecer em expressões de tipo, temos que reexaminar a noção de equivalência de tipos. Suponhamos que E_1 do tipo $s \rightarrow s'$ seja aplicada a E_2 do tipo t. Em lugar de simplesmente determinar a equivalência de s e de t, precisamos "uniformizá-los". A unificação é definida abaixo; informalmente, determinamos se s e t podem ser tornados estruturalmente equivalentes através da substituição das variáveis de tipo s e t por expressões de tipo. Por exemplo, no nó mais interno rotulado **aplicar** na Fig. 6.17, a igualdade

   $$apontador\ (\alpha_i) = apontador\ (apontador\ (inteiro))$$

   é verdadeira se α_i for substituída por *apontador (inteiro)*.

3. Necessitamos de um mecanismo para registrar o efeito de unificar duas expressões. Em geral, uma variável de tipo pode aparecer em várias expressões de tipo. Se a unificação de s e de s' resultar numa variável α representando o tipo t, então α precisa continuar a representar t à medida que a verificação de tipos prossiga. Por exemplo, na Fig. 6.17, α_i é o tipo intervalo de `derref`$_i$, e, dessa forma, podemos usá-la para o tipo de `derref`$_i$`(q)`. A unificação do tipo do domínio de `derref`$_i$ com o tipo de `q` afeta, por conseguinte, a expressão de tipo no nó mais interno rotulado **aplicar**. A outra variável de tipo α_o na Fig. 6.17 representa *inteiro*.

Substituições, Instâncias e Unificação

A informação sobre os tipos representada nas variáveis é formalizada definindo-se um mapeamento, a partir das variáveis de tipo para as expressões de tipo, chamado de uma *substituição*. A seguinte função recursiva *subst(t)* torna precisa a noção de aplicação de uma substituição S para repor todas as variáveis de tipo numa expressão t. Como de praxe, tomamos o construtor de tipos de função como sendo o construtor "típico".

função *subst (t : expressão_de_tipo): expressão_de_tipo;*
início
  se t é um tipo básico **então retornar** t
  senão se t é uma variável **então retornar** $S\ (t)$
  senão se t é $t_1 \rightarrow t_2$ **então retornar** $subst\ (t_1) \rightarrow subst\ (t_2)$
fim

Por conveniência, escrevemos $S\ (t)$ para a expressão de tipo que resulta quando *subst* é aplicada a t; o resultado $S\ (t)$ é chamado de uma *instância* de t. Se a substituição S não especifica uma expressão para uma variável α, assumimos que $S\ (\alpha)$ é α; isto é, S é o mapeamento identidade para tais variáveis.

Exemplo 6.10. No que se segue, escrevemos $s < t$ para indicar que s é uma instância de t;

$$\begin{array}{rcl} apontador\ (inteiro) & < & apontador\ (\alpha) \\ apontador\ (real) & < & apontador\ (\alpha) \\ inteiro \rightarrow inteiro & < & \alpha \rightarrow \alpha \\ apontador\ (\alpha) & < & \beta \\ \alpha & < & \beta \end{array}$$

No entanto, no que se segue, o tipo de uma expressão à esquerda não é uma instância daquela à direita (pela razão indicada):

inteiro	*real*	As substituições não se aplicam aos tipos básicos
inteiro $\rightarrow$ *real*	$\alpha \rightarrow \alpha$	Substituição inconsistente para α
inteiro $\rightarrow \alpha$	$\alpha \rightarrow \alpha$	Todas as ocorrências de α devem ser substituídas. □

Duas expressões de tipo t_1 e t_2 se *unificam* se existir alguma substituição S tal que $S(t_1) = S(t_2)$. Na prática, estamos interessados no *unificador mais geral*, que é uma substituição que impõe o menor número de restrições sobre as variáveis nas expressões Mais precisamente, o unificador mais geral das expressões t_1 e t_2 é uma substituição S com as seguintes propriedades:

1. $S(t_1) = S(t_2)$ e

2. para qualquer outra substituição S' tal que $S'\ (t_1) = S'\ (t_2)$, a substituição é uma instância de S (isto é, para qualquer t, $S'\ (t)$ é uma instância de $S(t)$).

No que segue, quando dizemos "unifica", estaremos nos referindo ao unificador mais geral.

Verificando as Funções Polimórficas

As regras para a verificação de expressões geradas pela gramática da Fig. 6.16 serão escritas em termos das seguintes operações numa representação de tipos sob a forma de grafos.

1. *novo*(t) substitui as variáveis ligadas na expressão de tipo t por variáveis novas e retorna um apontador para um nó representando a expressão de tipo resultante. Quaisquer ocorrências do símbolo $\forall$ são removidas no processo.

2. *unificar*(m, n) unifica as expressões de tipos representadas pelos nós apontados por m e n. Possui o efeito colateral de controlar a substituição que torna as expressões equivalentes. Se as expressões falham em unificar, todo o processo de verificação de tipos falha.[4]

As folhas individuais e os nós interiores do grafo de tipos são construídos utilizando as operações *criar_folha* e *criar_nó*, similares àquelas da Seção 5.2. É necessário que haja uma única folha para cada variável de tipo, mas outras expressões estruturalmente equivalentes não precisam ter nós únicos.

A operação *unificar* está baseada na seguinte formulação grafo-teorética de unificação e substituições. Suponhamos que os nós m e n de um grafo representem as expressões e e f, respectivamente. Dizemos que os nós m e n, são *equivalentes sob* a substituição S se $S(e) = S(f)$. O problema de se encontrar o unificador mais geral S pode ser reenunciado como sendo o problema de se agrupar, em conjuntos, os nós que precisam ser equivalentes sob S. Para as expressões serem equivalentes, suas raízes precisam ser equivalentes. Igualmente, dois nós m e n são equivalentes se e somente se representarem o mesmo operador e seus filhos correspondentes forem equivalentes.

Um algoritmo para unificar um par de expressões será retardado até a próxima seção. O algoritmo controla os conjuntos de nós que são equivalentes sob as substituições que ocorreram.

As regras de verificação de tipos para as expressões são mostradas na Fig. 6.18. Não mostramos como as declarações são processadas. Na medida em que as expressões de tipo geradas pelos não-terminais T e Q são examinadas, *criar_folha* e *criar_nó* adicionam nós ao grafo de tipo, seguindo a construção do GDA na Seção 5.2. Quando um identificador é declarado, o tipo na declaração é guardado na tabela de símbolos sob a forma de um apontador para o nó que representa o tipo. Na Fig. 6.18, esse apontador é referenciado como o atributo sintetizado **id**.*tipo*. Como mencionado acima, a operação *novo* remove os símbolos $\forall$ à medida que substitui as variáveis ligadas por variáveis novas. A ação associada à produção $E \rightarrow E_1, E_2$ estabelece *E.tipo* com o produto dos tipos de E_1 e E_2.

A regra de verificação de tipos para a aplicação de funções $E \rightarrow E_1(E_2)$ é motivada pela consideração do caso onde E_1.*tipo* e E_2.*tipo* são ambas variáveis de tipo, digamos que $E_1.tipo = \alpha$ e $E_2.tipo = \beta$. Aqui, E_1.*tipo* precisa ser uma função tal que, para algum tipo desconhecido γ, tenhamos $\alpha = \beta \rightarrow \gamma$. Na Fig. 6.18, uma variável nova correspondente a γ é criada e E_1.*tipo* é unificada com $E_2.tipo \rightarrow \gamma$. Uma nova variável de tipo é retornada a cada chamada de *nova_var_de_tipo*, uma folha para a mesma é construída por *criar_folha* e um nó representando a função a ser unificada com E_1.*tipo* é construída por *criar_nó*. Após a unificação ter sucesso, a nova folha representa o tipo resultante.

As regras da Fig. 6.18 serão ilustradas trabalhando-se sobre um exemplo simples em detalhe. Sumarizamos o funcionamento do algoritmo escrevendo a expressão de tipo atribuída a cada subexpressão, como na Fig. 6.19. A cada aplicação de função, a operação *unificar* pode ter o efeito colateral de registrar uma expressão de tipo para algumas das variáveis de tipo. Tais efeitos colaterais são sugeridos pela coluna para a substituição na Fig. 6.19.

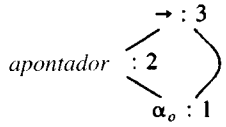
Exemplo 6.11. A verificação de tipos da expressão `derref`$_o$ `(derref`$_i$ `(q))` no programa (6.6) procede de baixo para cima, a partir das folhas. Mais uma vez, os subscritos o e i distiguem as ocorrências de

[4]A razão para se abortar o processo de verificação de tipos está em que os efeitos colaterais de algumas unificações podem ser registrados antes da falha ser detectada. A recuperação de erros pode ser implementada se os efeitos colaterais da operação *unificar* forem postergados até que as expressões tenham sido unificadas com sucesso.

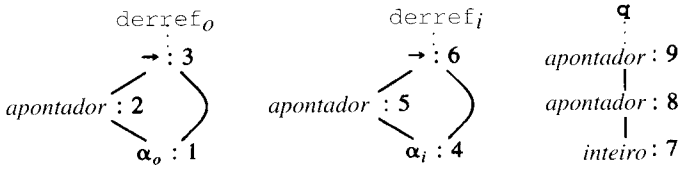
E	$\rightarrow$	E_1 (E_2)	{ p := *criar_folha*(*nova_var_de_tipo*); *unificar* (E_1.*tipo*, *criar_nó*('$\rightarrow$', E_2.*tipo*, p)); E. *tipo* := p }
E	$\rightarrow$	E_1, E_2	{ E. *tipo* := *criar_nó*('x', E_1.*tipo*, E_2.*tipo*) }
E	$\rightarrow$	**id**	{ E. *tipo* := *novo*(**id**.*tipo*) }

Fig. 6.18. Esquema de tradução para a checagem de funções polimórficas.

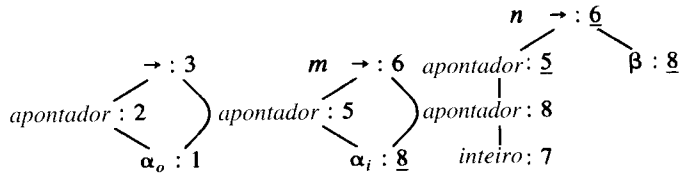
`derref`. Quando a subexpressão `derref`$_o$ é considerada, *novo* constrói nós usando uma nova variável de tipo α_o.



O número no nó indica a classe de equivalência a que o mesmo pertence. A parte do grafo de tipos para os três identificadores é mostrada abaixo. As linhas pontilhadas indicam que os nós numerados 3, 6 e 9 são para `derref`$_o$, `derref`$_i$ e `q`, respectivamente.



A aplicação de função `derref`$_i$ `(q)` é verificada pela construção do nó n para uma função a partir do tipo de `q` para uma nova variável de tipo β. Essa função se unifica com sucesso com o tipo de `derref`$_i$, representado pelo nó m abaixo. Antes dos nós m e n serem unificados, cada nó possui um número distinto. Após a unificação, os nós equivalentes são aqueles abaixo com o mesmo número; os números modificados estão sublinhados:



Note-se que o nó para α_i e *apontador(inteiro)* são ambos numerados 8, isto é, α_i é unificado com sua expressão de tipo, como mostrado na Fig. 6.19. Subseqüentemente, α_o é unificada com *inteiro*. □

O próximo exemplo relaciona a inferência de tipos de funções polimórficas em ML para as regras de verificação de tipos na Fig. 6.18. A sintaxe das definições de funções em ML é dada por

$$\textbf{fun id}_0(\ \textbf{id}_1, \ldots, \textbf{id}_k\ ) = E\ ;$$

onde $\textbf{id}_0$ representa o nome da função e $\textbf{id}_1, \ldots, \textbf{id}_k$ representam seus parâmetros. Para simplificar, assumimos que a sintaxe da expressão E é como na Fig. 6.16 e que os únicos identificadores em E são o nome da função, de seus parâmetros e de funções embutidas na linguagem.

A abordagem é uma formalização do Exemplo 6.9, onde um tipo polimórfico foi inferido para `derref`. Novas variáveis de tipo são

EXPRESSÃO	:	TIPO	SUBSTITUIÇÃO
q	:	*apontador* (*apontador* (*inteiro*))	
$\texttt{derref}_i$	:	*apontador* $(\alpha_i) \rightarrow \alpha_i$	
$\texttt{derref}_i$ (q)	:	*apontador* (*inteiro*)	α_i = *apontador* (*inteiro*)
$\texttt{derref}_o$	:	*apontador* $(\alpha_0) \rightarrow \alpha_o$	
$\texttt{derref}_o$ ($\texttt{derref}_i$ (q))	:	*inteiro*	α_o = *inteiro*

Fig. 6.19. Sumário de determinação de tipos *bottom-up*.

estabelecidas para o nome de função e seus parâmetros. Em geral, as funções embutidas têm tipos polimórficos; quaisquer variáveis de tipo que apareçam naqueles tipos estão ligadas a quantificadores $\forall$. Verificamos em seguida se os tipos das expressões $\mathbf{id}_0$ ($\mathbf{id}_1, \ldots, \mathbf{id}_k$) e E se correspondem. Quando uma correspondência ocorrer, teremos inferido um tipo para um nome de função. Finalmente, quaisquer variáveis no tipo inferido são amarradas por quantificadores $\forall$ para fornecer o tipo polimórfico para a função.

Exemplo 6.12. Relembremos a função ML na Fig. 6.14 para determinar o comprimento de uma lista:

```
fun comprimento (lptr) =
      if null(lptr) then 0
      else comprimento (tl (lptr)) + 1;
```

As variáveis de tipo β e γ são introduzidas para os tipos de `comprimento` e de `lptr`, respectivamente. Encontramos que o tipo de `comprimento (lptr)` corresponde ao tipo da expressão que forma o corpo da função e que `comprimento` deve ter o tipo

para qualquer tipo α, *lista* $(\alpha) \rightarrow$ *inteiro*

e dessa forma o tipo de `comprimento` é

$\forall\alpha$. *lista* $(\alpha) \rightarrow$ *inteiro*

Mais detalhadamente, construímos o programa mostrado na Fig. 6.20, para o qual as regras de verificação de tipos da Fig. 6.18 podem ser aplicadas. As declarações no programa associam novas variáveis de tipo β e γ a `comprimento` e `lptr` e tornam explícitos os tipos das funções embutidas. Escrevemos condicionais ao estilo da Fig. 6.16 pela aplicação do operador polimórfico `if` a três operandos, representando a expressão a ser testada, a parte *then* e a parte *else*; a declaração diz que as partes *then* e *else* podem ser de quaisquer tipos que se igualem, que será também o tipo do resultado.

Claramente, `comprimento (lptr)` precisa ter o mesmo tipo que o corpo da função; essa verificação é codificada usando-se o operador `corresponder`. O uso de `corresponder` é uma conveniência técnica que permite que toda verificação seja feita usando um programa ao estilo da Fig. 6.16.

```
  comprimento : β;
         lptr : γ;
           if : ∀α. booleano × α × α → α;
         null : ∀α. lista (α) → booleano;\
           tl : ∀α. lista (α) → lista (α)
            0 : inteiro;
            1 : inteiro;
            + : inteiro × inteiro → inteiro;
 corresponder : ∀α. α × α → α;
 corresponder(
              comprimento (lptr),
              if ( null ( lptr ), 0, comprimento
              ( tl ( lptr )) + 1
             )
```

Fig. 6.20. Declarações seguidas pela expressão a ser verificada.

O efeito da aplicação das regras de verificação de tipos na Fig. 6.18 ao programa na Fig. 6.20 é sumarizado na Fig. 6.21. As novas variáveis introduzidas pela operação *novo* aplicada aos tipos polimórficos das operações embutidas são distinguidas por subscritos em α. Aprendemos na linha (3) que `comprimento` precisa ser uma função de γ para algum tipo desconhecido δ. Em seguida, quando a subexpressão `null(lptr)` for verificada, encontramos na linha (6) que γ se unifica com *lista* (α_n), onde α é um tipo desconhecido. A esse ponto, sabemos que o tipo de `comprimento` precisará ser

para qualquer tipo α_n, *lista* $(\alpha_n) \rightarrow \delta$

Eventualmente, quando a adição verificada à linha (15) — tomamos a liberdade de escrever + entre os argumentos por uma questão de clareza — δ é unificado com *inteiro*.

Quando a verificação estiver completa, a variável de tipo α_n permanece no tipo de `comprimento`. Como nenhuma suposição foi feita sobre α_n, a mesma pode ser substituída por qualquer tipo quando a função for usada. Conseqüentemente, tornamo-la uma variável ligada e escrevemos

$\forall\alpha_n$. *lista* $(\alpha_n) \rightarrow$ *inteiro*

para o tipo de `comprimento`. □

6.7 UM ALGORITMO PARA A UNIFICAÇÃO

Informalmente, unificação é o problema de determinar se duas expressões e e f podem ser tornadas idênticas pela substituição das variáveis por expressões em e e f. O teste de igualdade de expressões é um caso especial da unificação; se e e f possuem constantes, mas não variáveis, e e f se unificam se e somente se forem idênticas. O algoritmo de unificação nesta seção pode ser aplicado a grafos com ciclos, e, por conseguinte, pode ser usado para testar a equivalência estrutural dos tipos circulares.[5]

A unificação foi definida na última seção em termos de uma função S, chamada de uma substituição, mapeando variáveis em expressões. Escrevemos $S(e)$ para a expressão obtida quando cada variável α em e é substituída por $S(\alpha)$. S é um unificador para e e f se $S(e) = S(f)$. O algoritmo desta seção determina a substituição que é o unificador mais geral para um par de expressões.

Exemplo 6.13. Para uma perspectiva sobre os unificadores mais gerais, consideremos as duas expressões de tipos

$$((\alpha_1 \rightarrow \alpha_2) \times list\ (\alpha_3)) \rightarrow list\ (\alpha_2)$$
$$((\alpha_3 \rightarrow \alpha_4) \times list\ (\alpha_3)) \rightarrow list\ \alpha_5$$

[5]Em algumas aplicações, é um erro unificar uma variável e uma expressão contendo aquela variável. O Algoritmo 6.1 permite tais substituições.

LINHA	EXPRESSÃO	:	TIPO	SUBSTITUIÇÃO
(1)	`lptr`	:	γ	
(2)	`comprimento`	:	β	
(3)	`comprimento (lptr)`	:	δ	$\beta = \gamma \rightarrow \delta$
(4)	`lptr`	:	γ	
(5)	`null`	:	*lista* $(\alpha_n) \rightarrow$ *booleano*	
(6)	`null (lptr)`	:	*booleano*	$\gamma =$ *lista* (α_n)
(7)	`0`	:	*inteiro*	
(8)	`lptr`	:	*lista* (α_n)	
(9)	`tl`	:	*lista* $(\alpha_t) \rightarrow$ *lista* (α_t)	
(10)	`tl (lptr)`	:	*lista* (α_n)	$\alpha_t = \alpha_n$
(11)	`comprimento`	:	*lista* $(\alpha_n) \rightarrow \delta$	
(12)	`comprimento (tl (lptr ))`	:	δ	
(13)	`1`	:	*inteiro*	
(14)	`+`	:	*inteiro* $\times$ *inteiro* $\rightarrow$ *inteiro*	
(15)	`comprimento ( tl ( lptr )) + 1`	:	*inteiro*	$\delta =$ *inteiro*
(16)	`if`	:	*booleano* $\times \alpha_i \times \alpha_i \rightarrow \alpha_i$	
(17)	`if ( . . . )`	:	*inteiro*	$\alpha_i =$ *inteiro*
(18)	`corresponder`	:	$\alpha_m \times \alpha_m \rightarrow \alpha_m$	
(19)	`corresponder ( . . . )`	:	*inteiro*	$\alpha_m =$ *inteiro*

Fig. 6.21. Inferindo o tipo *lista*$(\alpha_n) \rightarrow$ *inteiro* para `comprimento`.

Dois unificadores, S e S', para essas expressões são:

x	$S(x)$	$S'(x)$
α_1	α_3	α_1
α_2	α_2	α_1
α_3	α_3	α_1
α_4	α_2	α_1
α_5	*lista* (α_2)	*lista* (α_1)

Essas substituições mapeiam e e f como segue:

$$S(e) = S(f) = ((\alpha_3 \rightarrow \alpha_2) \times \textit{lista}\ (\alpha_3)) \rightarrow \textit{lista}\ (\alpha_2)$$
$$S'(e) = S'(f) = ((\alpha_1 \rightarrow \alpha_1) \times \textit{lista}\ (\alpha_1)) \rightarrow \textit{lista}\ (\alpha_1)$$

A substituição S é o unificador mais geral para e e f. Note-se que $S'(e)$ é uma instância de $S(e)$ porque podemos substituir ambas as variáveis em $S(e)$ por α_1. O inverso, no entanto, é falso, pois cada ocorrência de α_1 em $S'(e)$ precisa ser substituída pela mesma expressão e dessa forma não podemos obter $S(e)$ substituindo a variável α_1 em $S'(e)$. □

Quando as expressões a serem unificadas representam árvores, o número de nós na árvore para a expressão substituída $S(e)$ pode ser uma exponencial do número de nós nas árvores para e e f, mesmo que S seja o unificador mais geral. Entretanto, tal explosão de tamanho não precisa ocorrer se forem usados grafos em lugar de árvores para representar expressões e substituições.

Iremos implementar uma formulação grafo-teorética da unificação, também apresentada na última seção. O problema é o de agrupar, em conjuntos, os nós que precisam ser equivalentes sob o unificador mais geral de duas expressões. As duas expressões do Exemplo 6.13 são representadas pelos dois nós rotulados $\rightarrow$: 1 na Fig. 6.22. Os inteiros aos nós indicam as classes de equivalência às quais os nós pertencem após os nós rotulados 1 serem unificados. Essas classes de equivalência possuem a propriedade de todos os nós interiores numa classe de equivalência serem para o mesmo operador. Os filhos correspondentes dos nós interiores numa classe de equivalência são também equivalentes.

Algoritmo 6.1. Unificação de um par de nós numa grafo.

Entrada. Um grafo e um par de nós m e n a serem unificados.

Saída. O valor *booleano* verdadeiro se as duas expressões representadas pelos nós m e n se unificarem; falso, em caso contrário. A versão da operação *unificar* necessitada para as regras de verificação de tipos da Fig. 6.18 é obtida se a função neste algoritmo é modificada para falhar ao invés de retornar falso.

Método. Um nó é representado por um registro, como na Fig. 6.23, com campos para um operador binário e apontadores para os filhos à esquerda e à direita.

Os conjuntos de nós equivalentes são mantidos usando-se o campo *conjunto*. Um nó em cada classe de equivalência é escolhido como o único elemento representativo da classe de equivalência fazendo-se seu campo *conjunto* conter o apontador nil. Os campos *conjunto* dos

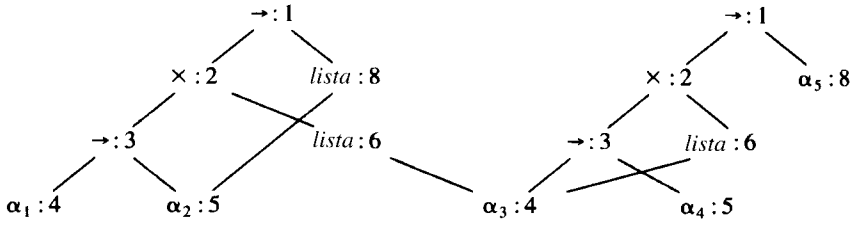


Fig. 6.22. Classes de equivalência após a unificação.

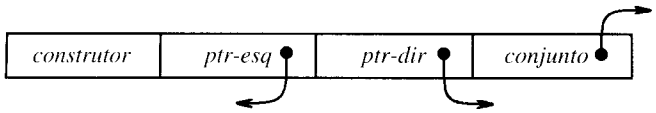


Fig. 6.23. Estrutura de dados para um nó.

nós remanescentes na classe de equivalência irão apontar (possivelmente de forma indireta através dos outros nós no conjunto) para o elemento representativo. Inicialmente, cada nó *n* é uma classe de equivalência por si só, tendo o próprio *n* como o elemento representativo.

O algoritmo de unificação, mostrado na Fig. 6.24, usa as seguintes operações sobre os nós:

1. *localizar (n)* retorna o nó representativo da classe de equivalência contendo corretamente o nó *n*.

2. *união (m, n)* combina as classes de equivalência contendo os nós *m* e *n*. Se um dos elementos representativos para as classes de equivalência de *m* e *n* for um nó não-variável, *união* faz o nó não-variável ser o elemento representativo da classe de equivalência resultante da união; em caso contrário, *união* elege um dos dois elementos representativos como o novo representante. Essa assimetria na especificação de *união* é importante porque uma variável não pode ser usada como o elemento representativo de uma classe de equivalência para uma expressão contendo um construtor de tipo ou um tipo básico. De outra forma, duas expressões não equivalentes poderiam ser unificadas através daquela variável.

A operação de *união* sobre conjuntos é implementada simplesmente mudando-se o campo *conjunto* do elemento representativo de uma classe de equivalência de forma que o mesmo aponte para o elemento representativo da outra. Para encontrar a classe de equivalência a que um nó pertence, seguimos os apontadores *conjunto* dos nós até que o elemento representativo (o nó com um apontador **nil** no campo conjunto) seja atingido.

Note-se que o algoritmo da Fig. 6.24 usa *s* = *localizar (m)* e *t* = *localizar (n)* ao invés de *m* e *n*, respectivamente. Se os nós representativos *s* e *t* representarem o mesmo tipo básico, a chamada *unificar (m, n)* retorna verdadeiro. Se *s* e *t* são ambos nós interiores para um construtor binário de tipos, combinamos as suas classes de equivalência numa base especulativa e verificamos recursivamente se seus filhos respectivos são equivalentes. Combinando-se primeiro, diminuímos o número de classes de equivalência antes de verificar recursivamente os filhos e dessa forma o algoritmo termina.

A substituição de uma variável por uma expressão é implementada pela adição da folha para a variável à classe de equivalência que contém o nó para a expressão. Se um dentre *m* e *n*, mas não ambos, for uma folha para uma variável que tenha sido colocada numa classe de equivalência, que contenha um nó que represente uma expressão com um construtor de tipos ou com um tipo básico, *localizar* irá retornar um elemento representativo que reflita o construtor de tipos ou tipo

função *unificar* (*m, n* : *nó*) : *booleano;*
início
  s := *localizar* (*m*);
  t := *localizar* (*n*);
  se *s* = *t* **então**
    retornar verdadeiro
  senão se *s* e *t* são nós que representam o mesmo tipo básico
    então retornar verdadeiro
  senão se *s* é um nó de *operador* com filhos s_1 e s_2 **e**
      t é um nó de *operador* com filhos t_1 e t_2 **então início**
    união (*s, t*);
    retornar *unificar* (s_1, t_1) **e** *unificar* (s_2, t_2)
  fim
  senão se *s* ou *t* representa uma variável **então início**
    união (*s, t*);
    retornar verdadeiro
  fim
  senão retornar falso
    /* os nós interiores com operadores diferentes não podem ser unificados */
fim

Fig. 6.24. Algoritmo de unificação.

básico e, por conseguinte, uma variável não pode ser unificada a duas expressões diferentes. □

Exemplo 6.14. Mostramos o grafo inicial para duas expressões do Exemplo 6.13 na Fig. 6.25 com cada nó numerado e em sua própria classe de equivalência. Para computar *unificar* (1, 9), o algoritmo nota que os nós 1 e 9 representam o mesmo operador e, por conseguinte, combina 1 e 9 na mesma classe de equivalência e chama *unificar* (2, 10) e *unificar* (8, 14). O resultado do cômputo de *unificar* (1, 9) é o grafo previamente mostrado na Fig. 6.22. □

Se o Algoritmo 6.1 retornar verdadeiro, podemos construir, como se segue, uma substituição *S* que atue como um unificador. Façamos cada nó *n* do grafo resultante representar a expressão associada a *localizar (n)*. Dessa forma, para cada variável α, *localizar* (α) fornece o nó *n* que é o elemento representativo da classe de equivalência de α. A expressão representada por *n* é $S(\alpha)$. Por exemplo, na Fig. 6.22, vemos que o elemento representativo para α_3 é o nó 4, que representa α_1. O elemento representativo para α_5 é o nó 8, que representa *lista*(α_2).

Exemplo 6.15. O Algoritmo 6.1 pode ser usado para testar a equivalência estrutural das duas expressões de tipo

e : *real* → e
f : *real* → (*real* → f)

Os grafos de tipo para essas expressões são mostrados na Fig. 6.26. Por uma questão de conveniência, cada nó foi numerado.

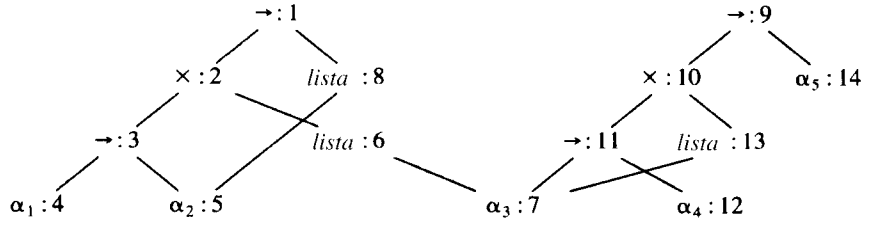


Fig. 6.25. GDA inicial com cada nó em sua própria classe de equivalência.

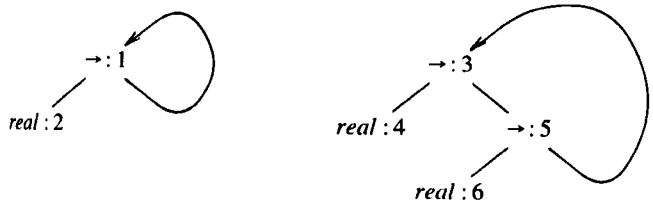


Fig. 6.26. Grafos para dois tipos circulares.

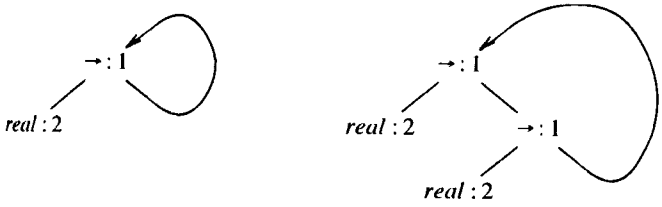


Fig. 6.27. Grafo de tipos mostrando as classes de equivalência dos nós.

Chamamos *unificar*(1, 3) para testar a equivalência estrutural dessas duas expressões. O algoritmo combina os nós 1 e 3 em uma única classe de equivalência e, recursivamente, chama *unificar* (2, 4) e *unificar*(1, 5). Uma vez que 2 e 4 representam o mesmo tipo básico, a chamada *unificar* (2, 4) retorna verdadeiro. A chamada *unificar* (1, 5) adiciona 5 à classe de equivalência de 1 e de 3 e chama recursivamente *unificar*(2, 6) e *unificar*(1, 3).

A chamada *unificar*(2, 6) retorna verdadeiro porque 2 e 6 também representam o mesmo tipo básico. A segunda chamada de *unificar* (1, 3) termina porque já havíamos combinado os nós 1 e 3 na mesma classe de equivalência. O algoritmo então termina, retornando verdadeiro para mostrar que as duas expressões são de fato equivalentes. A Fig. 6.27 mostra as classes de equivalência de nós resultantes, onde os nós com o mesmo número inteiro estão na mesma classe de equivalência. □

EXERCÍCIOS

6.1 Escreva expressões de tipo para os seguintes tipos.

a) Um *array* de apontadores de reais, onde os índices do *array* variam de 1 a 100.

b) Um *array* bidimensional de inteiros (isto é, um *array* de *arrays*) cujas linhas sejam indexadas de 0 a 9 e as colunas de −10 a 10.

c) Funções cujos domínios sejam funções de inteiros para apontadores de inteiros e cujos intervalos sejam registros consistindo em um inteiro e um caractere.

6.2 Vamos supor que temos as seguintes declarações C:

```
typedef  struct  {
         int a, b;
         }  CELULA, *PCELULA;
CELULA foo[100];
PCELULA barra(x, y) int x; CELULA y { ... }
```

Escreva expressões para os tipos `foo` e `barra`.

6.3 A seguinte gramática define listas de listas de literais. A interpretação dos símbolos é a mesma que a da gramática da Fig. 6.3 com a adição do tipo **list**, que indica uma lista de elementos do tipo T que o segue.

$$
\begin{array}{lcl}
P & \rightarrow & D\,;E \\
D & \rightarrow & D\,;D \mid \textbf{id}\,:\,T \\
T & \rightarrow & \textbf{list of}\ T \mid \textbf{char} \mid \textbf{integer} \\
E & \rightarrow & (L) \mid \textbf{literal} \mid \textbf{num} \mid \textbf{id} \\
L & \rightarrow & E, L \mid E
\end{array}
$$

Escreva regras de tradução semelhantes àquelas na Seção 6.2 para determinar os tipos das expressões (E) e as listas (L).

6.4 Adicione à gramática do Exercício 6.3 a produção

$$E \rightarrow \textbf{nil}$$

significando que uma expressão pode ser a lista nula (vazia). Revise as regras na sua resposta ao Exercício 6.2, de forma a levar em conta o fato de que **nil** pode figurar em lugar de uma lista vazia de elementos de qualquer tipo.

6.5 Usando o esquema de tradução da Seção 6.2, compute os tipos das expressões nos seguintes fragmentos de programa. Mostre os tipos a cada nó da árvore gramatical.

a)
```
c: char; i: integer;
c mod i mod 3
```

b)
```
p: ↑ integer; a: array [10] of integer;
a[p↑]
```

c)
```
f: integer → boolean;
i: integer; j: integer; k: integer;
while f(i) do
      k: = i;
      i: = j mod i;
      j: = k
```

6.6 Modifique o esquema de tradução para a verificação de expressões na Seção 6.2 de forma a que imprima uma mensagem descritiva quando um erro for detectado e continue a verificação como se o tipo esperado tivesse sido enxergado.

6.7 Reescreva as regras de verificação de tipos para expressões na Fig. 6.2 de forma a que se refiram aos nós de uma representação sob a forma de grafo para as expressões de tipo. As regras reescritas devem usar estruturas de dados e operações suportadas por uma linguagem como Pascal. Use a.equivalência estrutural para as expressões de tipo quando:

a) as expressões de tipo forem representadas por árvores, como na Fig. 6.2 e

b) o grafo de tipos for um GDA com um único nó para cada expressão de tipo.

6.8 Modifique o esquema de tradução da Fig. 6.5 para tratar o que se segue.

a) Comandos que possuam valor. O valor de um comando de atribuição é o da expressão à direita do símbolo : =. O valor de um enunciado condicional ou de repetição (*while*) é o valor do corpo do enunciado, o valor de uma lista de enunciados é o valor do último elemento da lista.

b) Expressões booleanas. Adicione produções para os operadores lógicos **e, ou** e **não** e para os operadores de comparação (<, etc.). Em seguida, adicione as regras de tradução apropriadas para as expressões de tipo.

6.9 Generalize as regras de verificação de tipos para as funções dadas ao final da Seção 6.2 para tratar funções *n-árias*.

6.10 Assuma que os nomes de tipo `link` e `celula` sejam definidos como na Seção 6.3. Quais das seguintes expressões são estruturalmente equivalentes?

i) `link`

ii) *apontador* (`celula`)

iii) *apontador* (`link`)

iv) *apontador*(*registro* ((`info` × *inteiro*) × (`proximo` × *apontador* (`celula`))))

6.11 Reorganize o algoritmo para testar a equivalência estrutural na Fig. 6.6, de forma que os argumentos de *sequiv* sejam apontadores para os nós de um GDA.

6.12 Considere a codificação de expressões restringidas de tipos como seqüências de *bits* no Exemplo 6.1. Em Johnson [1979], os campos de dois *bits* para os construtores apareceram na ordem oposta, com o campo para o construtor mais externo figurando em seguida aos quatro *bits* do tipo básico; por exemplo,

EXPRESSÃO DE TIPO	CODIFICAÇÃO
caractere	000000 0001
frets(*caractere*)	000011 0001
apontador(*frets*(*caractere*))	001101 0001
array(*apontador* (*frets*(*caractere*)))	110110 0001

Usando os operadores de C, escreva um código para construir a representação da *array(t)* a partir daquela de *t* e vice-versa, assumindo que a codificação esteja em:

a) Johnson [1979].
b) Exemplo 6.1.

6.13 Suponhamos que o tipo do identificador seja um subintervalo dos inteiros. Para expressões com os operadores `+`, `-`, `*`, `div` e `mod`, como em Pascal, escreva regras de verificação de tipos que associem a cada subexpressão o subintervalo em que o valor precisa estar contido.

6.14 Dê um algoritmo para testar a equivalência dos tipos de C (veja o exemplo 6.4).

6.15 Algumas linguagens, como PL/I, irão coagir um valor *booleano* num valor inteiro, com verdadeiro identificado por 1 e falso por 0. Por exemplo, 3<4<5 é agrupado como (3<4) <5 e tem o valor "verdadeiro" (ou 1) porque 3<4 tem o valor 1 e 1<5 é verdadeiro. Escreva regras de tradução para expressões *booleanas* que realizem essa coerção. Utilize enunciados condicionais na linguagem intermediária para associar valores inteiros a temporários que representem o valor de uma expressão *booleana*, quando necessário.

6.16 Generalize os algoritmos de (a) Fig. 6.9 e (b) Fig. 6.12 para expressões com os construtores de tipos *array, apontador* e produto cartesiano.

6.17 Quais das seguintes expressões de tipo recursivas são equivalentes?

```
e1 = inteiro → e1
e2 = inteiro → (inteiro → e2)
e3 = inteiro → (inteiro → e1)
```

6.18 Usando as regras do Exemplo 6, determine quais das seguintes expressões possuem tipos únicos. Assuma que `z` seja um número complexo

a) `1 * 2 * 3`
b) `1 * (z * 2)`
c) `(1 * z) * z`

6.19 Suponhamos que sejam permitidas conversões de tipos no Exemplo 6.6. Sob que condições sobre os tipos de *a, b* e *c* (inteiro ou complexo) terá a expressão $(a * b) * c$ um único tipo?

6.20 Expresse, usando variáveis de tipo, os tipos das seguintes funções:

a) A função *ref* que toma como argumento um objeto de qualquer tipo e retorna um apontador para o objeto.
b) Uma função que tome como argumento um *array* indexado pelos inteiros, com elementos de qualquer tipo, e retorne um *array* cujos elementos sejam os objetos apontados pelos elementos do *array* dado.

6.21 Encontre o unificador mais geral para as seguintes expressões de tipo

i) $(apontador\ (\alpha)) \times (\beta \rightarrow \gamma)$
ii) $\beta \times (\gamma \rightarrow \delta)$

Como seria se em (ii) δ fosse α?

6.22 Encontre o unificador mais geral para cada par de expressões da lista seguinte ou estabeleça que nenhum existe.

a) $\alpha_1 \rightarrow (\alpha_2 \rightarrow \alpha_1)$
b) $array(\beta_1) \rightarrow (apontador\ (\beta_1) \rightarrow \beta_3)$
c) $\gamma_1 \rightarrow \gamma_2$
d) $\delta_1 \rightarrow (\delta_1 \rightarrow \delta_2)$

6.23 Expanda as regras de verificação de tipos no Exemplo 6.6 para atender registros. Use a seguinte sintaxe adicional para expressões de tipo e expressões:

$$\begin{array}{rcl} T & \rightarrow & \textbf{record}\ campos\ \textbf{end} \\ E & \rightarrow & E\ .\ \textbf{id} \\ campos & \rightarrow & campos\ ;\ campo \mid campo \\ campo & \rightarrow & \textbf{id}\ :\ T \end{array}$$

Que restrições a ausência de nomes de tipos impõe sobre os tipos que podem ser definidos?

***6.24** A resolução para a sobrecarga na Seção 6.5 procede através de duas fases: primeiro, o conjunto de tipos possíveis para cada subexpressão é determinado e, em seguida, estreitado numa segunda fase para um único tipo, após o tipo único de toda a expressão ter sido determinado. Que estruturas de dados V. usaria para resolver a sobrecarga numa única passagem de baixo para cima (*bottom-up*)?

****6.25** A resolução da sobrecarga se torna mais difícil se as declarações de identificadores forem opcionais. Mais precisamente, suponhamos que as declarações possam ser usadas para sobrecarregar identificadores que representem símbolos de funções, mas que todas as ocorrências de um identificador não declarado tenham o mesmo tipo. Mostre que o problema de determinar se uma expressão nesta linguagem possui um tipo válido é NP-completo. Este problema emerge durante a verificação de tipos na linguagem experimental Hope (Berstall, MacQueene Sannella [1980]).

6.26 Seguindo o Exemplo 6.12, infira o seguinte tipo polimórfico para `map`:

$$\texttt{map} : \forall\alpha.\ \forall\beta.\ ((\alpha \rightarrow \beta) \times lista\ (\alpha)) \rightarrow lista\ (\beta)$$

A definição ML de `map` é

```
fun map(f, l) =

    if null(l) then nil
    else cons(f(hd(l)), map(f, tl (l)))
```

Os tipos dos identificadores embutidos no corpo da função são:

`null`	:	$\forall\alpha.$	$lista\ (\alpha) \rightarrow booleano;$
`nil`	:	$\forall\alpha.$	$lista\ (\alpha);$
`cons`	:	$\forall\alpha.$	$(\alpha \times lista(\alpha)) \rightarrow lista(\alpha);$
`hd`	:	$\forall\alpha.$	$lista\ (\alpha) \rightarrow \alpha;$
`tl`	:	$\forall\alpha.$	$lista\ (\alpha) \rightarrow lista(\alpha);$

****6.27** Mostre que o algoritmo de unificação da Seção 6.7 determina o unificador mais geral.

***6.28** Modifique o algoritmo da Seção 6.7 de tal forma que o mesmo não unifique uma variável com uma expressão contendo aquela mesma variável.

****6.29** Suponhamos que as expressões sejam representadas por árvores. Encontre as expressões *e* e *f* tais que para qualquer unificador *S* o número de nós em *S*(*e*) seja uma exponencial do número de nós em *e* e *f*.

```
(1)  int n;
(2)  int f(g)
(3)  int g( );
(4)  {
(5)     int m;
(6)     m = n;
(7)     if ( m = = 0 ) return 1;
(8)     else {
(9)        n = n − 1; return m * g(g);
(10)    }
(11) }
(12) main ( )
(13) {
(14)    n = 5; printf("%d o fatorial é %d\n",
        n,f (f) );
(15) }
```

Fig. 6.28. Um programa C contendo aplicações de uma função a si mesma.

6.30 Dois nós são ditos *congruentes* se representam expressões equivalentes. Mesmo que não existam pares de nós congruentes no grafo original de tipos, após a unificação, é possível que nós distintos sejam congruentes.

a) Dê um algoritmo para combinar uma classe de nós mutuamente congruentes num único nó.

**b) Estenda o algoritmo em (a) para combinar nós congruentes até que não hajam dois nós distintos que sejam congruentes.

***6.31** A expressão `g(g)` à linha 9 no programa C completo na Fig. 6.28 é a aplicação de uma função a si mesma. A declaração à linha 3 fornece *inteiro* como o tipo intervalo `g`, mas os tipos dos argumentos de `g` não são especificados. Tente executar o programa. O compilador deve emitir um aviso porque `g` é declarado como uma função à linha 3 em lugar de um apontador para uma função.

a) O que V. pode dizer a respeito do tipo de `g`?

b) Use as regras de verificação de tipos para as funções polimórficas na Fig. 6.18 para inferir um tipo para `g` no programa seguinte.

```
    m   : inteiro;
vezes   : inteiro × inteiro → inteiro;
    g   : α;
    vezes ( m, g(g) )
```

NOTAS BIBLIOGRÁFICAS

Os tipos básicos e os construtores de tipos das primeiras linguagens como Fortran e Algol 60 eram limitados o suficiente para que a verificação de tipos não se constituísse um problema sério. Como resultado, as descrições da verificação de tipos em seus compiladores são esquecidas nas discussões da geração de código para as expressões. Sheridan [1959] descreve a tradução de expressões no compilador Fortran original. O compilador controlava se o tipo de uma expressão era inteiro ou real, mas as coerções não eram permitidas pela linguagem. Backus [1981, p. 54] relembra, "Penso que justamente porque não gostávamos das regras sobre o que acontecia com as expressões em modo misto, decidimos 'Vamos jogá-las fora. É mais fácil' ". Naur [1965] é um artigo primordial na verificação de tipos num compilador Algol; as técnicas usadas pelo compilador são similares àquelas discutidas na Seção 6.2.

Facilidades de estruturação de dados tais como *arrays* e registros foram antecipadas nos anos 40 por Zuse em seu Plankalkül, que teve pouca influência direta (Bauer e Wössner [1972]). Uma das primeiras linguagens de programação a permitir que expressões de tipos fossem construídas sistematicamente é Algol 68. Duas expressões podem ser recursivamente definidas e a equivalência estrutural é usada. Uma distinção clara entre a equivalência estrutural e a equivalência por nome é encontrada em EL1, a escolha sendo deixada para o programador (Wegbreit [1974]), A Crítica de Pascal por Welsh, Sneeringer e Hoare [1977] chamou a atenção para a discussão.

A combinação de coerção e sobrecarga pode levar a ambigüidades: coagir um argumento pode levar à resolução da sobrecarga em favor de um algoritmo diferente. As restrições são, por conseguinte, colocadas numa ou noutra. Uma abordagem liberadora da coerção foi tomada em PL/I, onde o primeiro critério de projeto era "*Qualquer coisa pode.* Se uma combinação particular de símbolos possuir um significado razoavelmente sensível, aquele significado será tornado oficial (Radin e Rogoway [1965]". Um ordenamento é freqüentemente colocado no conjunto de tipos básicos — por exemplo, Hext [1967] descreve uma estrutura de reticulado imposta sobre os tipos básicos de CPL — e os tipos mais abaixo podem ser coagidos para os tipos mais altos.

A resolução em tempo de compilação da sobrecarga em linguagens tais como APL (Iverson [1962]) e SETL (Schwartz [1973]) tem o potencial de melhorar o tempo de execução dos programas (Bauer e Saal [1973]). Tennenbaum [1974] distinguiu entre a resolução "à frente" que determina o conjunto de possíveis tipos de um operador a partir de seus operandos e a resolução "para trás" baseada no tipo esperado pelo contexto. Usando um reticulado de tipos, Jones e Muchnick [1976] e Kaplan e Ullman [1980] resolvem restrições sobre tipos obtidas a partir de análises à frente e para trás. A sobrecarga em Ada pode ser resolvida fazendo-se uma única passagem à frente, e em seguida uma única passagem para trás, como na Seção 6.5. Essa observação emerge de um número de artigos: Ganzinger e Ripken [1980], Pennello, DeRemer e Meyers [1980]; Janas [1980]; Persch et al. [1980]. Cormack [1981] oferece uma implementação recursiva e Baker [1982] evita uma passagem para trás explícita mantendo um GDA dos possíveis tipos.

A inferência de tipos foi estudada por Curry (ver Curry e Feys [1958]) em conexão com a lógica combinatória e o cálculo lambda (*Lambda Calculus)* de Church [1941]. Tem sido há muito observado que o cálculo lambda está no núcleo das linguagens funcionais. Temos repetidamente usado a aplicação de uma função a um argumento para discutir conceitos de verificação de tipos neste capítulo. As funções podem ser definidas e aplicadas independentemente dos tipos no cálculo lambda e Curry estava interessado em seus "caracteres funcionais" e em determinar o que chamaríamos agora de tipo polimórfico mais geral, consistindo em uma expressão de tipo com quantificadores universais, como na Seção 6.6. Motivado por Curry, Hindley [1969] observou que a unificação poderia ser usada para inferir os tipos. Independentemente, em sua tese, Morris [1968a] atribuiu tipos a expressões lambda montando uma série de equações e resolvendo-as de forma a determinar os tipos associados às variáveis. Desavisado do trabalho de Hindley, Milner [1978] também observou que a unificação podia ser usada para resolver o conjunto de equações e aplicou a idéia de inferir tipos na linguagem de programação ML.

A pragmática da verificação de tipos em ML é descrita por Cardelli [1984]. Essa abordagem tem sido aplicada a uma linguagem de Meertens [1983]; Suzuki [1981] explora sua aplicação em Smalltalk 1976 (Ingalls [1978]). Mitchell [1984] mostra como as coerções poderiam ser incluídas.

Morris [1968a] observa que os tipos recursivos ou circulares permitem que os tipos sejam inferidos para expressões contendo a aplicação de uma função a si mesma. O programa C à Fig. 6.28, contendo uma aplicação de uma função a si mesma, é motivado por um programa Algol em Ledgard [1971]. O Exercício 6.31 é proveniente de MacQueen, Plotkin e Sethi [1984], onde é dado um modelo semântico para tipos polimórfico recursivo. Enfoques diferentes aparecem em McCracken [1979] e Cartwright [1985]. Reynolds [1985] pesquisa o sistema de tipos ML, fornece diretrizes teóricas para evitar anomalias envolvendo coerções, sobrecarga e funções polimórficas de mais alta ordem.

A unificação foi primeiramente estudada por Robinson [1965]. O algoritmo de unificação na Seção 6.7 pode ser prontamente adaptado a partir de algoritmos para testar a equivalência de (1) autômatos finitos e (2) listas ligadas com ciclos (Knuth [1973a]), Seção 2.3.5, Exercício 11). O algoritmo quase linear para testar a equivalência de autômatos finitos devido a Hopcroft e Karp [1971] pode ser visto como uma implementação do esboço à pag. 594 de Knuth [1973a]. Através do uso esclarecido de estruturas de dados, algoritmos lineares para o caso acíclico são apresentados por Paterson e Wegman [1978] e por Martelli e Montanari [1982]. Um algoritmo para encontrar os nós congruentes (ver Exercício 6.30) aparece em Downey, Sethi e Tarjan [1980].

Despeyroux [1984] descreve um gerador de verificador de tipos que utiliza o reconhecimento de padrões para criar um verificador de tipos a partir de uma especificação semântico-operacional baseada em regras de inferência.

CAPÍTULO 7

AMBIENTES EM TEMPO DE EXECUÇÃO

Antes de considerar a geração de código, precisamos relacionar o texto-fonte estático do programa às ações que precisam ocorrer em tempo de execução para implementar o programa. À medida que a execução prossegue, o mesmo nome no texto fonte pode denotar objetos de dados diferentes na máquina alvo. Este capítulo examina o relacionamento entre os nomes e os objetos de dados.

A alocação e liberação de objetos de dados são gerenciadas pelo pacote de *suporte em tempo de execução,* consistindo em rotinas carregadas com o código-alvo gerado. O projeto do pacote de suporte em tempo de execução é influenciado pela semântica dos procedimentos. Pacotes de suporte para linguagens como Fortran, Pascal e Lisp podem ser construídos usando-se as técnicas neste capítulo.

Cada execução de um procedimento é referenciada como uma *ativação* do procedimento. Se o procedimento for recursivo, várias de suas ativações podem estar vivas ao mesmo tempo. Cada chamada de procedimento em Pascal leva a uma ativação que pode manipular os objetos de dados alocados para seu uso.

A representação de um objeto de dados em tempo de execução é determinada por seu tipo. Freqüentemente, os tipos de dados elementares, tais como caracteres, inteiros e reais, podem ser representados por objetos de dados equivalentes na máquina-alvo. No entanto, agregados de dados tais como *arrays,* cadeias de caracteres e estruturas são usualmente representados por coleções de objetos de dados primitivos; suas disposições são estudadas no Capítulo 8.

7.1 TEMAS DA LINGUAGEM-FONTE

Para simplificar, suponhamos que um programa seja composto por procedimentos, como em Pascal. Esta seção distingue o texto-fonte de um procedimento de suas ativações em tempo de execução.

Procedimentos

Uma *definição de procedimento* é uma declaração que, na sua forma mais simples, associa um identificador a um enunciado. O identificador é o *nome do procedimento* e o enunciado é o *corpo do procedimento.* Por exemplo, o código Pascal na Fig. 7.1 contém a definição do procedimento denominado `readarray` às linhas 3-7; o corpo do procedimento está nas linhas 5-7. Em muitas linguagens, os procedimentos que retornam valores são chamados de *funções;* entretanto, é conveniente que nos refiramos às mesmas como procedimentos. Um programa completo também será tratado como um procedimento.

Quando um nome de procedimento aparece dentro de um enunciado executável, dizemos que o procedimento é *chamado* àquele ponto. A idéia básica está em que uma chamada de procedimento execute o corpo do mesmo. O programa principal nas linhas 21-25 da Fig. 7.1 chama o procedimento `readarray` à linha 23 e em seguida chama `quicksort` à linha 24. Note-se que as chamadas de procedimentos podem ocorrer dentro de expressões, como na linha 16.

Alguns dos identificadores que aparecem numa definição de procedimento são especiais e são chamados de *parâmetros formais* (ou simplesmente, *formais*) do procedimento (C os chama de "argumentos formais" e Fortran de "argumentos fictícios"). Os identificadores `m` e `n` na linha 12 são os parâmetros formais de `quicksort`. Argumentos, conhecidos como *parâmetros atuais* (ou *atuais*), podem ser transmitidos ao procedimento chamado; substituem os parâmetros formais no corpo do procedimento. Os métodos para estabelecer a correspon-

```
(1)   program sort (input,output);
(2)         var a : array [0..10] of integer;

(3)         procedure readarray;
(4)                 var i : integer;
(5)                 begin
(6)                       for i := 1 to 9 do read (a[i])
(7)                 end;

(8)         function partition (y, z: integer) : integer;
(9)                 var i, j, x, v: integer;
(10)                begin ...
(11)                end;

(12)        procedure quicksort (m, n: integer);
(13)                var i : integer;
(14)                begin
(15)                      if ( n > m ) then begin
(16)                            i := partition (m,n);
(17)                            quicksort (m,i-1);
(18)                            quicksort (i+1,n)
(19)                      end
(20)                end;

(21)        begin
(22)                a[0] := -9999; a[10] := 9999;
(23)                readarray;
(24)                quicksort (1,9)
(25)        end.
```

Fig. 7.1. Um programa Pascal para ler e classificar inteiros.

dência entre os parâmetros atuais e formais são discutidos na Seção 7.5. A linha 18 da Fig. 7.1 é uma chamada de `quicksort` com parâmetros atuais `i+1` e `n`.

Árvores de Ativações

Fazemos as seguintes suposições sobre o fluxo de controle entre procedimentos durante a execução de um programa:

1. O controle flui seqüencialmente; isto é, a execução de um programa consiste em uma seqüência de passos, com o controle, a cada passo, em algum ponto específico do programa.
2. Cada execução de um procedimento começa ao início do corpo do procedimento e eventualmente retorna o controle para o ponto imediatamente seguinte ao local de onde foi chamado. Isso significa que o fluxo de controle entre os procedimentos pode ser delineado usando-se árvores, como veremos em breve.

Cada execução do corpo de um procedimento é referenciada como uma ativação do procedimento. O *tempo de vida* de uma ativação de um procedimento `p` é a seqüência de passos entre o primeiro e o último passos na execução do corpo do procedimento, incluindo o tempo dispendido executando-se procedimentos chamados por `p`, por procedimentos chamados por esses últimos e assim por diante. Em geral, o termo "tempo de vida" se refere a uma seqüência de passos consecutivos durante a execução de um programa.

Em linguagens como Pascal, a cada vez que o controle entrar num procedimento `q`, proveniente de um procedimento `p`, retornará eventualmente ao procedimento `p` (na ausência de um erro fatal). Mais precisamente, a cada vez que o controle fluir de uma ativação de um procedimento `p` para uma ativação de um procedimento `q`, retornará para a mesma ativação de `p`.

Se a e b são ativações de procedimentos, então seus tempos de vida ou são não sobrepostos ou são aninhados. Isto é, se a ativação b é iniciada antes de a ser deixada, o controle terá que abandonar b antes de deixar a.

Esta propriedade de aninhamento dos tempos de vida das ativações pode ser ilustrada inserindo-se dois enunciados de impressão em cada procedimento, um antes do primeiro comando do corpo do procedimento e o outro após o último. O primeiro comando imprime a cadeia `entrando` seguida pelo nome do procedimento e dos valores dos parâmetros atuais; o último enunciado imprime a cadeia `abandonando` seguida pela mesma informação. Uma execução do programa na Fig. 7.1 com esses enunciados de impressão produziu a saída mostrada na Fig. 7.2. O tempo de vida da ativação `quicksort (1,9)` é a seqüência de passos executados entre a impressão de `entrando quicksort (1,9)` e a impressão de `abandonando quicksort (1,9)`. Na Fig. 7.2, é assumido que o valor retornado por `partition (1,9)` é 4.

Um procedimento é *recursivo* se uma nova aplicação puder começar antes que uma ativação anterior do mesmo procedimento tenha terminado. A Fig. 7.2 mostra que o controle entra na ativação de `quicksort (1,9)`, a partir da linha 24, ao início da execução do programa, mas abandona essa ativação quase ao fim. Nesse meio tempo, existem diversas outras ativações de `quicksort` e, dessa forma, `quicksort` é recursivo.

Um procedimento recursivo `p` não precisa chamar a si mesmo diretamente; `p` pode chamar um outro procedimento `q` que pode chamar `p` através de alguma seqüência de chamadas de procedimentos. Podemos usar uma árvore, chamada de *árvore de ativações*, para delinear a forma como o controle entra e abandona as ativações. Numa árvore de ativações,

1. cada nó representa uma ativação de um procedimento,
2. a raiz representa a ativação do programa principal,
3. o nó para a é o pai do nó para b se e somente se o controle fluir da ativação a para b e

```
A execução inicia...
entrando readarray
abandonando readarray
entrando quicksort (1,9)
entrando partition (1,9)
abandonado partition (1,9)
entrando quicksort (1,3)
      ...
deixando quicksort (1,3)
entrando quicksort (5,9)
      ...
deixando quicksort (5,9)
deixando quicksort (1,9)
Execução terminada.
```

Fig. 7.2. Saída sugerindo as ativações de procedimentos na Fig. 7.1

4. o nó para a está à esquerda do nó para b se e somente se o tempo de vida de a decorrer antes do tempo de vida de b.

Como cada nó representa uma única ativação e vice-versa, é conveniente falarmos do controle estando a um nó quando estiver na ativação representada por aquele nó.

Exemplo 7.1. A árvore de ativações na Fig. 7.3 é construída a partir da saída na Fig. 7.2.[1] Para economizar espaço, somente a primeira letra de cada procedimento é mostrada. A raiz da árvore de ativações é para todo o programa `sort`. Durante a execução de `sort`, existe um registro de ativação de `readarray`, representado pelo primeiro filho da raiz, tendo rótulo `r`. A ativação seguinte, representada pelo segundo filho da raiz, é de `quicksort`, com parâmetros atuais 1 e 9. Durante essa ativação, as chamadas de `partition` e de `quicksort` nas linhas 16-18 da Fig. 7.1 levam às ativações `p(1,9)`, `q(1,3)` e `q(5,9)`. Note-se que as ativações `q(1,3)` e `q(5,9)` são recursivas e que começam e terminam antes de `q(1,9)` terminar. □

Pilhas de Controle

O fluxo de controle num programa corresponde a uma travessia em profundidade da árvore de ativações que começa à raiz, visita um nó antes de seus filhos e recursivamente visita os filhos a cada nó numa ordem da esquerda para a direita. A saída da Fig. 7.2 pode ser, por conseguinte, reconstruída através de uma travessia da árvore de ativações na Fig. 7.3, imprimindo a cadeia `entrando` quando o nó de uma ativação for atingido pela primeira vez e imprimindo a cadeia `abandonando` após toda a subárvore do nó ter sido visitada durante a travessia.

Podemos usar uma pilha, chamada de *pilha de controle*, para controlar a vida das ativações dos procedimentos. A idéia é a de empilhar o nó para uma ativação na pilha de controle quando a ativação se iniciar e remover o nó quando a mesma terminar. Dessa forma, o conteúdo da pilha de controle está relacionado a percursos até a raiz da árvore de ativações. Quando um nó n estiver ao topo da pilha de controle, a mesma conterá os nós ao longo do percurso de n até a raiz.

Exemplo 7.2. A Fig. 7.4 mostra os nós da árvore de ativações que foram atingidos quando o controle entrou na ativação representada por `q(2,3)`. As ativações com rótulos `r`, `p(1,9)`, `p(1,3)` e `q(1,0)` foram executadas até o fim e, conseqüentemente, a figura contém linhas tracejadas para seus nós. As linhas sólidas marcam o percurso de `q(2,3)` até a raiz.

A esse ponto, a pilha de controle contém os seguintes nós ao longo desse percurso até a raiz (o topo da pilha está à direita)

[1] As chamadas efetivas feitas por `quicksort` dependem do que `partition` retornar (ver Aho, Hopcroft e Ullman [1983] para detalhes do algoritmo). A Fig. 7.3 representa uma possível árvore de chamadas. É consistente com a Fig. 7.2, apesar de certas chamadas mais baixas na árvore não serem mostradas na Fig. 7.2.

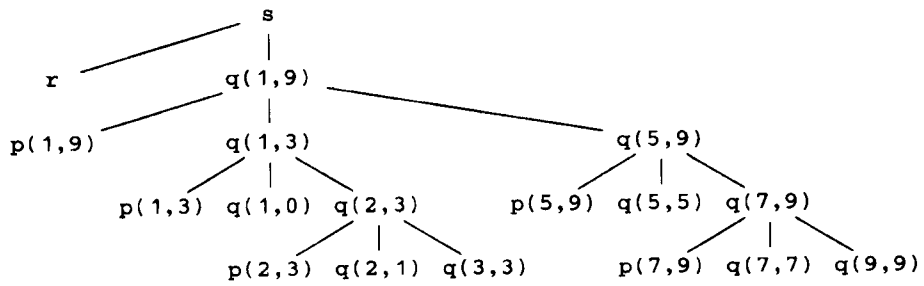


Fig. 7.3. Uma árvore de ativações correspondente à saída da Fig. 7.2.

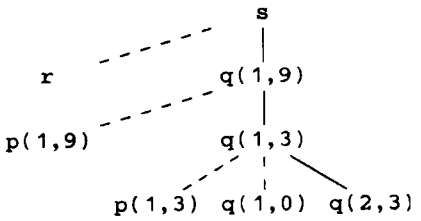


Fig. 7.4. A pilha de controle contém nós ao longo de um percurso até a raiz.

```
s, q(1,9), q(1,3), q(2,3)
```

e nenhum outro nó. □

O tema das pilhas de controle se estende até a técnica de alocação de memória de pilha usada para implementar linguagens como Pascal e C. Essa técnica é discutida em detalhes nas Seções 7.3 e 7.4.

O Escopo de uma Declaração

Uma declaração numa linguagem é uma construção sintática que associa informações a um nome. As declarações podem ser explícitas, como no fragmento Pascal.

```
var i : integer;
```

ou podem ser implícitas. Por exemplo, num programa Fortran, qualquer nome de variável começando por `I` é assumido denotar um inteiro, a menos que seja declarado explicitamente algo diferente. As *regras de escopo* de uma linguagem determinam a declaração de nome que se aplica, quando o mesmo figurar num programa-fonte. No programa Pascal da Fig. 7.1, `i` é declarado três vezes, às linhas 4, 9 e 13, e os usos do nome `i` nos procedimentos `readarray`, `partition` e `quicksort` são independentes uns dos outros. A declaração à linha 4 se aplica aos usos de `i` à linha 6. Isto é, as ocorrências de `i` à linha 6 estão no escopo da declaração à linha 4. As três ocorrências de `i` às linhas 16-18 estão no escopo da declaração de `i` à linha 13.

A parte do programa à qual uma declaração se aplica é chamada de *escopo* daquela declaração. Uma ocorrência de um nome dentro de um procedimento é dita *local* ao mesmo se estiver no escopo de uma declaração dentro do daquele procedimento; caso contrário, a ocorrência é *não-local*. A distinção entre nomes locais e não-locais recai sobre qualquer construção sintática que possa ter declarações dentro de si.

Conquanto o escopo seja uma propriedade da declaração de um nome, é algumas vezes conveniente usar a abreviação "escopo de um nome `x`" em lugar de "escopo da declaração do nome `x` que se aplica a esta ocorrência de `x`". Neste sentido, o escopo de `i` à linha 17 da Fig. 7.1 é o corpo de `quicksort`.[2]

[2]A maior parte do tempo, os termos nome, identificador, variável e lexema podem ser usados intercambiavelmente sem causar qualquer confusão a respeito da construção conotada.

Em tempo de compilação, a tabela de símbolos pode ser usada para encontrar a declaração que se aplica à ocorrência de um nome. Quando a declaração é enxergada, é criada para a mesma uma entrada na tabela de símbolos. Na medida em que estivermos no escopo de uma declaração, sua entrada é retornada quando o nome for procurado. As tabelas de símbolos são discutidas na Seção 7.6.

Amarrações de Nomes

Mesmo que num programa cada nome seja declarado uma única vez, o mesmo nome poderá denotar objetos de dados diferentes em tempo de execução. O termo informal "objeto de dados" corresponde a uma localização de memória que pode abrigar valores.

Na semântica das linguagens de programação, o termo *ambiente* se refere a uma função que mapeia um nome em uma localização de memória e o termo *estado* a uma função que mapeia uma localização de memória no valor guardado nela, como na Fig. 7.5. Usando os termos valor-*l* e valor-*r*, provenientes do Capítulo 2, um ambiente mapeia um nome em um valor-*l* e um estado o valor-*l* em um valor-*r*.

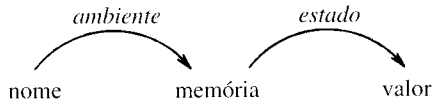


Fig. 7.5. Um mapeamento em dois estágios de nomes em valores.

Os ambientes e estados são diferentes; uma atribuição modifica o estado, mas não o ambiente. Por exemplo, suponhamos que o endereço de memória 100, associado à variável `pi`, abrigue o valor 0. Após a atribuição `pi := 3.14`, o mesmo endereço de memória ainda está associado a `pi`, mas o valor guardado lá é `3.14`.

Quando um ambiente associa uma localização de memória *s* a um nome `x`, dizemos que `x` está *amarrado* a *s*; a associação é, em si, referenciada como uma *amarração* de `x`. O termo "localização" de memória é tomado figurativamente. Se `x` não for de um tipo básico, o armazenamento para `x` pode ser uma coleção de palavras de memória.

Uma amarração é a contraparte dinâmica de uma declaração, como mostrado na Fig. 7.6. Como temos visto, mais de uma ativação de um procedimento recursivo pode estar viva ao mesmo tempo. Em Pascal, um nome de variável local a um procedimento estará amarrado a diferentes localizações de memória, a cada ativação do mesmo. As técnicas para se amarrar nomes de variáveis locais são consideradas na Seção 7.3.

NOÇÃO ESTÁTICA	CONTRAPARTE DINÂMICA
definição de um procedimento	ativação de um procedimento
declaração de um nome	amarrações de um nome
escopo de uma declaração	tempo de vida de uma amarração

Fig. 7.6. Correspondência entre as noções estáticas e dinâmicas.

Perguntas a Responder

A forma pela qual um compilador de uma linguagem precisa organizar sua memória e amarrar nomes é amplamente determinada por respostas a questões como as seguintes:

1. Os procedimentos podem ser recursivos?
2. O que acontece aos valores dos nomes locais quando o controle retorna da ativação de um procedimento?
3. Um procedimento pode se referir a nomes não-locais?
4. Como são transmitidos os parâmetros quando um procedimento é chamado?
5. Os procedimentos podem ser transmitidos como parâmetros?
6. Os procedimentos podem ser retornados como resultados?
7. A memória pode ser reservada dinamicamente sob controle do programa?
8. A memória precisa ser liberada explicitamente?

O efeito desses temas no suporte necessitado em tempo de execução por uma dada linguagem de programação é examinado no restante deste capítulo.

7.2 ORGANIZAÇÃO DE MEMÓRIA

A organização de memória em tempo de execução, discutida nesta seção, pode ser usada em linguagens tais como Fortran, Pascal e C.

Subdivisão da Memória em Tempo de Execução

Suponhamos que o compilador obtenha um bloco de memória do sistema operacional para o programa compilado executar no mesmo. A partir da discussão da última seção, essa memória em tempo de execução poderia ser subdividada de forma a abrigar:

1. o código-alvo gerado,
2. objetos de dados, e
3. a contraparte da pilha de controle para controlar as ativações de procedimentos.

O tamanho do código gerado é estabelecido em tempo de compilação e dessa forma o compilador pode colocá-lo numa área estaticamente determinada, talvez na parte mais baixa da memória. Analogamente, os tamanhos de alguns objetos de dados também já devem ser conhecidos em tempo de compilação, e esses objetos de dados também podem ser colocados numa área estaticamente determinada, como na Fig. 7.7. Uma razão para a alocação estática de tantos objetos de dados quanto o possível está em que os endereços desses objetos de dados podem ser compilados no código-alvo. Todos os objetos de dados em Fortran podem ser alocados estaticamente.

As implementações de linguagens como Pascal e C usam extensões da pilha de controle para gerenciar as ativações dos procedimentos. Quando ocorre uma chamada, a execução de uma ativação é interrompida e as informações sobre o estado da máquina, tais como o valor do apontador da próxima instrução e dos registradores de máquina, são salvas na pilha. Quando o controle retorna da chamada, essa ativação pode ser retomada após a restauração dos valores dos registradores relevantes e o estabelecimento do apontador da próxima instrução para um ponto imediatamente após a chamada. Os objetos de dados cujos tempos de vida estão contidos naquela ativação podem ser estabelecidos na pilha, juntamente com outras informações associadas à ativação. Essa estratégia é discutida na próxima seção.

Uma área separada de memória em tempo de execução, chamada *heap*, abriga todas as outras informações. Pascal permite que os dados sejam alocados sob controle do programa, como discutido na Seção 7.7; o armazenamento para tais dados é obtido a partir do *heap*. As implementações de linguagens nas quais os tempos de vida das ativações não possam ser representados por uma árvore de ativações poderiam usar o *heap* para manter informações sobre as mesmas. A forma controlada pela qual os dados são reservados e liberados numa pilha torna mais barato colocá-los numa pilha ao invés de num *heap*.

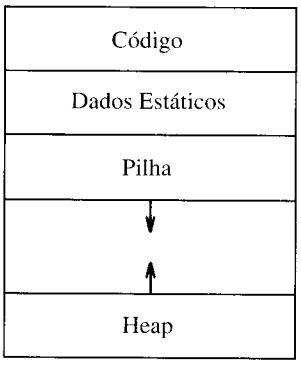


Fig. 7.7. Subdivisão típica da memória em tempo de execução em código e áreas de dados.

Os tamanhos da pilha e do *heap* podem mudar à medida que o programa executa e dessa forma os mostramos em extremidades opostas da memória na Fig. 7.7, onde podem crescer em direção um do outro, na medida do necessário. Pascal e C precisam tanto da pilha em tempo de execução quanto do *heap*, mas nem todas as linguagens precisam de ambos.

Por convenção, a pilha cresce para baixo. Isto é, o "topo" da pilha é desenhado em direção ao fundo da página. Como os endereços de memória aumentam à medida que vamos para baixo na página, o "crescimento para baixo" significa em direção aos endereços maiores. Se *topo* marca o topo da pilha, os deslocamentos a partir do topo da pilha podem ser computados subtraindo-se o deslocamento de *topo*. Em muitas máquinas, este cômputo pode ser feito eficientemente mantendo-se o valor de *topo* num registrador. Os endereços da pilha podem, então, ser representados como deslocamentos a partir de *topo*.[3]

Registros de Ativação

As informações necessitadas por uma única execução de um procedimento são gerenciadas utilizando-se um único bloco contíguo de armazenamento chamado de um *registro de ativação* ou *moldura*, consistindo em uma coleção de campos mostrada na Fig. 7.8. Nem todas as linguagens, nem todos os compiladores, usam todos esses campos; freqüentemente, os registradores podem tomar o lugar de um ou mais deles. Para linguagens como Pascal e C, é usual empilhar o registro de ativação do procedimento na pilha em tempo de execução quando o procedimento é chamado, e removê-lo da mesma quando o controle retornar ao chamador.

O propósito dos campos de um registro de ativação é como o seguinte, começando com o campo para as áreas de armazenamento de dados temporários.

1. Valores temporários, tais como aqueles que emergem da avaliação de expressões, são armazenados no campo para os temporários.
2. O campo para dados locais abriga aqueles dados que são locais a uma execução do procedimento. A disposição desse campo é discutida em seguida.

[3]Organização da Fig. 7.7 supõe que a memória em tempo de execução se constitua de um único bloco contíguo de armazenamento, obtido ao início da execução. Esta suposição impõe um limite fixo nos tamanhos combinados da pilha e do *heap*. Se o limite for grande o suficiente de forma que raramente seja excedido, poderá ser disperdiçantemente grande para a maioria dos programas. A alternativa de se ligar objetos na pilha e no *heap* pode tornar mais caro controlar o topo da pilha. Sobretudo, a máquina-alvo pode favorecer uma diferente colocação de áreas. Por exemplo, algumas máquinas permitem somente deslocamentos positivos a partir de um endereço num registrador.

3. O campo para o estado salvo da máquina detém informações sobre o estado da máquina exatamente antes do procedimento ser chamado. Essa informação inclui os valores de um contador de programa* e dos registradores de máquina que precisam ser restaurados quando o controle retornar do procedimento.
4. O *elo de acesso* opcional é usado na Seção 7.4 para referir aos dados locais abrigados em outros registros de ativação. Os elos de acesso não são necessários para uma linguagem como Fortran, porque os dados não locais são mantidos em localizações fixas. Os elos de acesso ou o mecanismo relacionado de *"display"* são necessitados em Pascal.
5. O *elo de controle* opcional aponta para o registro de ativação do chamador.
6. O campo para os parâmetros atuais é usado pelo procedimento chamador para fornecer os parâmetros do procedimento chamado. Mostramos o espaço para os parâmetros no registro de ativação, mas, na prática, os mesmos são passados em registradores de máquina, para maior eficiência.
7. O campo para o valor retornado é usado pelo procedimento chamado para retornar um valor para o procedimento chamador. De novo, na prática, esse valor é freqüentemente retornado num registrador, para maior eficiência.

O tamanho de cada um desses campos pode ser determinado em tempo de chamada do procedimento. De fato, os tamanhos de quase todos os campos podem ser determinados em tempo de compilação. Uma exceção ocorre quando um procedimento puder ter um *array* local cujo tamanho seja determinado pelo valor de um parâmetro atual, disponível somente quando o procedimento for chamado em tempo de execução. Veja a Seção 7.3 para o estabelecimento de dados de tamanho variável no registro de ativação.

Gabarito para os Dados Locais em Tempo de Compilação

Suponhamos que o armazenamento em tempo de execução venha em blocos de *bytes* contíguos, onde um *byte* é a menor unidade endereçável de memória. Em muitas máquinas, um *byte* é constituído por oito *bits* e algum número consecutivo de *bytes* forma uma palavra de máquina. Múltiplos objetos são armazenados em *bytes* consecutivos, sendo-lhes atribuídos os endereços do primeiro *byte* de cada seqüência.

A quantidade de memória necessitada por um nome é determinada a partir de seu tipo. Um tipo de dados elementar, tal como caractere, inteiro ou real, pode ser usualmente armazenado num número integral de *bytes*. A memória de armazenamento de um agregado, como um *array* ou registro, precisa ser suficientemente ampla para abrigar todos os seus componentes. Para facilidade de acesso aos componentes, o armazenamento para os agregados é tipicamente mantido em blocos de *bytes* contíguos. Veja as Seções 8.2 e 8.3 para mais detalhes.

O campo para dados locais é estabelecido na medida em que as declarações de um procedimento são examinadas em tempo de compilação. Dados de tamanho variável são mantidos fora desse campo. Mantemos um contador de localizações de memória que tenham sido reservadas para as declarações anteriores. A partir do contador, determinamos o endereço *relativo* de armazenamento para um objeto de dados local em relação a alguma posição, tal como a do início do registro de ativação. O endereço relativo, ou *deslocamento*, é a diferença entre os endereços da posição e do objeto de dados.

A disposição da memória para os objetos de dados é fortemente influenciada pelas restrições de endereçamento da máquina-alvo. Por exemplo, as instruções para adicionar inteiros podem esperar que os mesmos sejam *alinhados*, isto é, colocados em certas posições na memória, tal como num endereço divisível por 4. Apesar de um *array* de dez caracteres precisar somente de dez *bytes* para abrigá-los, um compilador pode, no entanto, reservar 12 *bytes*, deixando 2 *bytes* sem uso. O espaço deixado sem uso, devido às considerações de alinhamento, é denominado de *enchimento*. Quando o espaço estiver a prêmio, um compilador pode *compactar* dados, de forma que nenhum enchimento seja gerado; podem ser necessitadas instruções adicionais em tempo de execução para posicionar os dados compactados, de forma que possam ser operados como se estivessem propriamente alinhados.

*Usamos os termos contador do programa e apontador da próxima instrução com o mesmo significado. (N. do T.)

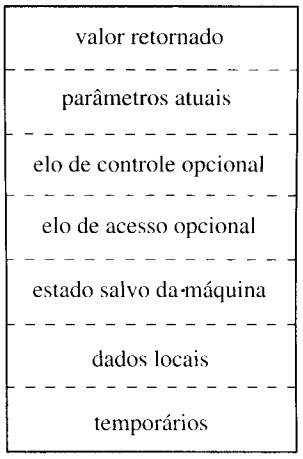


Fig. 7.8. Um registro de ativação geral.

Exemplo 7.3. A Fig. 7.9 é uma simplificação da disposição de dados usada em compiladores C para duas máquinas, que chamamos Máquina 1 e Máquina 2. C providencia três tamanhos de inteiros, declarados usando-se as palavras-chave `short`, `int` e `long`. Os conjuntos de instruções para as duas máquinas são tais que o compilador para a Máquina 1 reserva 16, 32 e 32 *bits* para os três tamanhos de inteiros, enquanto que o compilador para a Máquina 2 reserva 24, 48 e 64 *bits*, respectivamente. Para uma comparação entre as máquinas, os comprimentos são medidos em *bits* na Fig. 7.9, ainda que nenhuma das máquinas permita que *bits* sejam endereçados diretamente.

A memória da Máquina 1 é organizada em *bytes* consistindo em 8 *bits* cada um. Ainda que cada *byte* possua um endereço, o conjunto de instruções favorece que os inteiros curtos sejam posicionados em endereços divisíveis por 4. O compilador coloca os inteiros curtos em endereços pares, mesmo que no processo tenha que pular um *byte* de enchimento. Por conseguinte, quatro *bytes*, consistindo em 32 *bits*, podem ser reservados para um caractere seguido por um inteiro curto.

Na Máquina 2, cada palavra consiste em 64 *bits* e são permitidos 24 *bits* para o endereçamento de uma palavra. Existem 64 possibilidades para os *bits* individuais dentro de uma palavra, de forma que 6 *bits* adicionais não necessários para distingui-los. Por projeto, um apontador para um caractere na Máquina 2 gasta 30 *bits* — 24 para encontrar a palavra e 6 para a posição do *bit* inicial do caractere dentro da palavra.

A forte orientação para palavra do conjunto de instruções da Máquina 2 levou o compilador a reservar uma palavra completa de cada vez, mesmo que um número menor de *bits* fosse suficiente para representar todos os possíveis valores de um tipo; por exemplo, somente 8 *bits* são necessários para representar um caractere. Por conseguinte, sob o alinhamento, a Fig. 7.9 mostra os 64 *bits* para cada tipo. Dentro de cada palavra, os *bits* para cada tipo básico estão nas posições especificadas. Duas palavras consistindo em 128 *bits* seriam reservadas para um caractere seguido por um inteiro curto, com o caractere usando apenas 8 *bits* da primeira palavra e o inteiro curto somente 24 *bits* da segunda. □

TIPO	TAMANHO (*Bits*)		ALINHAMENTO (*Bits*)	
	Máquina 1	Máquina 2	Máquina 1	Máquina 2
char	8	8	8	64[a]
short	16	24	16	64
int	32	48	32	64
long	32	64	32	64
float	32	64	32	64
double	64	128	32	64
apontador de caracteres	32	30	32	64
outros apontadores	32	24	32	64
estruturas	8	64	32	64

[a]Os caracteres nos *arrays* são alinhados a cada 8 *bits*.

Fig. 7.9. Disposições de dados usadas por dois compiladores C.

7.3 ESTRATÉGIAS PARA A ALOCAÇÃO DE MEMÓRIA

Uma estratégia diferente de reserva de memória é usada em cada uma das três áreas de dados na organização da Fig. 7.7.

1. A alocação de memória estática dispõe o armazenamento para todos os objetos de dados em tempo de compilação.
2. A alocação de memória de pilha gerencia a memória em tempo de execução como uma pilha.
3. A alocação de memória *heap* reserva e libera as áreas de armazenamento na medida do necessitado em tempo de execução, a partir de uma área de dados conhecida como *heap*.

Essas estratégias de alocação são aplicadas, nesta seção, aos registros de ativação. Também descrevemos como o código-alvo de um procedimento tem acesso à memória amarrada a um nome local.

Alocação de Memória Estática

Na alocação estática, os nomes são amarrados às localizações de memória à medida que o programa é compilado, de forma que não há necessidade de um pacote de suporte em tempo de execução. Uma vez que as amarrações não se modificam em tempo de execução, a cada vez que um procedimento é ativado, seus nomes são amarrados às mesmas localizações de memória. Essa propriedade permite que os valores dos nomes locais sejam *retidos* através das ativações de um procedimento. Isto é, quando o controle retorna a um procedimento, os valores dos objetos de dados locais são os mesmos que detinham quando o controle o deixou pela última vez.

A partir do tipo de um nome, o compilador determina a quantidade de memória a ser reservada para o mesmo, como discutido na Seção 7.2. O endereço dessa memória consiste no deslocamento a partir de uma das extremidades do registro de ativação do procedimento. O compilador terá eventualmente que decidir onde os registros de ativação irão ficar, tanto quanto à posição relativa ao código-alvo do programa bem como às posições relativas entre si. Uma vez que essa decisão tenha sido tomada, a posição de cada registro de ativação e, por conseguinte, a do armazenamento de cada nome no registro são fixadas. Em tempo de compilação podemos, por conseguinte, preencher os endereços aos quais o código-alvo poderá encontrar os dados sobre os quais opera. Similarmente, os endereços aos quais as informações devem ser salvas quando ocorre uma chamada de procedimento também são conhecidos em tempo de compilação.

No entanto, algumas limitações acompanham a alocação de memória estática, quando somente a mesma é utilizada:

1. O tamanho dos objetos de dados e as restrições a respeito de suas posições na memória precisam ser conhecidos em tempo de compilação.
2. Os procedimentos recursivos estão restringidos, porque todas as ativações de um procedimento usam as mesmas amarrações para os nomes locais.
3. As estruturas de dados não podem ser criadas dinamicamente, uma vez que não há mecanismo para alocação de memória em tempo de execução.

Fortran foi projetada para possibilitar a reserva de memória estática. Um programa Fortran consiste em um programa principal, de sub-rotinas e funções (chamêmo-las todas de *procedimentos*), como no exemplo da Fig. 7.10. Usando a organização de memória da Fig. 7.7, a disposição do código e dos registros de ativação para esse programa é mostrada na Fig. 7.11. Dentro do registro de ativação para CNSUME (leia "consume"* — Fortran não gosta de identificadores longos), existe espaço para os objetos de dados locais BUF, NEXT e C. A memória amarrada a BUF abriga uma cadeia de 50 caracteres. É seguida por espaço para guardar um valor inteiro para NEXT e um valor de caractere para C. O fato de NEXT ser também declarado em PRDUCE não apresenta problema porque os objetos de dados locais para os dois procedimentos obtêm espaço nos seus respectivos registros de ativação.

Como os tamanhos do código executável e do registro de ativação são conhecidos em tempo de compilação, são possíveis organiza-

```
(1)      PROGRAM CNSUME
(2)        CHARACTER * 50 BUF
(3)        INTEGER NEXT
(4)        CHARACTER C, PRDUCE
(5)        DATA NEXT /1/, BUF /' '/
(6)  6       C = PRDUCE( )
(7)          BUF (NEXT:NEXT) = C
(8)          NEXT = NEXT + 1
(9)          IF ( C. NE. ' ' ) GOTO 6
(10)         WRITE (*, '(A)') BUF
(11)         END

(12)     CHARACTER FUNCTION PRDUCE( )
(13)       CHARACTER * 80 BUFFER
(14)       INTEGER NEXT
(15)       SAVE BUFFER, NEXT
(16)       DATA NEXT /81/
(17)         IF ( NEXT .GT. 80 )THEN
(18)           READ (*, ' (A) ') BUFFER
(19)           NEXT = 1
(20)         END IF
(21)         PRDUCE = BUFFER(NEXT:NEXT)
(22)         NEXT = NEXT+1
(23)         END
```

Fig. 7.10. Um programa Fortran 77.

*Consumir, na língua original. (N. do T.)

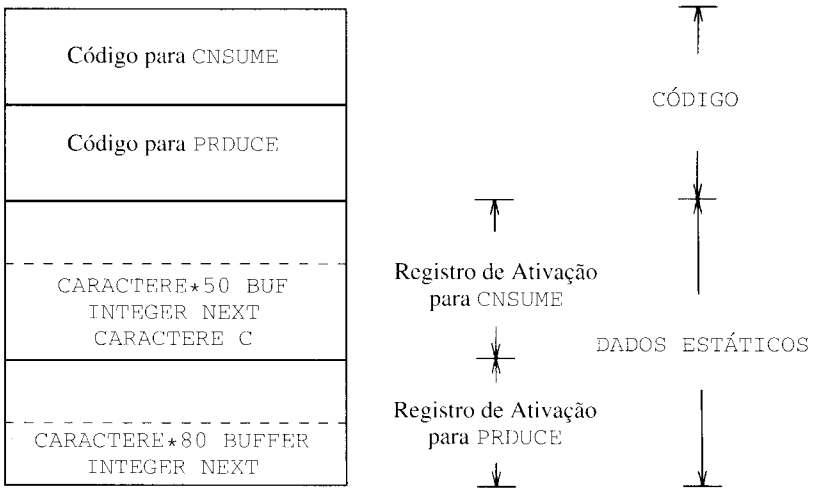


Fig. 7.11. Armazenamento estático para os objetos de dados associados a identificadores locais num programa Fortran 77.

ções de memória que não aquela da Fig. 7.11. Um compilador Fortran poderia colocar o registro de ativação para um procedimento juntamente com o código desse procedimento. Em alguns sistemas de computação é viável deixar a posição relativa dos registros de ativação inespecificada e permitir que o editor de ligações ligue os registros de ativação e o código executável.

Exemplo 7.4. O programa da Fig. 7.10 confia em que os valores dos objetos de dados locais sejam retidos ao longo das ativações dos procedimentos. Um enunciado SAVE em Fortran 77 especifica que o valor de um objeto de dados local ao início de uma ativação precisa ser o mesmo que aquele ao fim da última ativação. Os valores iniciais para esses objetos de dados locais podem ser especificados usando-se um enunciado DATA.

O enunciado à linha 18 do procedimento PRDUCE lê uma linha de texto de cada vez num *buffer*. O procedimento delibera caracteres sucessivos a cada vez que for ativado. O programa principal CNSUME também possui um *buffer* no qual acumula caracteres até que um espaço seja enxergado. À entrada

```
Bom dia vida!
```

os caracteres retornados pelas ativações de PRDUCE são delineados na Fig. 7.12; a saída do programa é

```
Bom
```

O *buffer* no qual PRDUCE lê linhas precisa reter os valores entre as ativações. O enunciado SAVE à linha 15 assegura que, quando o controle retornar a PRDUCE, os objetos de dados locais associados a BUFFER e a NEXT tenham os valores que tinham quando o controle deixou o procedimento pela última vez. Na primeira vez que o controle atinge PRDUCE, o valor do objeto de dados local associado a NEXT é obtido a partir do valor especificado no enunciado DATA, à linha 16. Dessa forma, NEXT é inicializado com 81. □

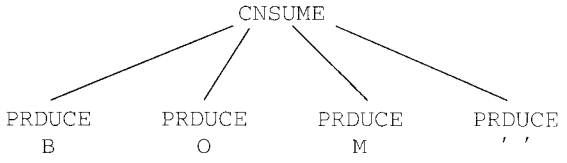


Fig. 7.12. Caracteres retornados pelas ativações de PRDUCE.

Alocação de Memória de Pilha

A alocação de memória de pilha está baseada na idéia de uma pilha de controle; a memória é organizada como uma pilha e os registros de ativação são empilhados e desempilhados à medida que as ativações começam e terminam, respectivamente. O armazenamento para os objetos de dados locais a cada chamada de um procedimento está contido no registro de ativação daquela chamada. Dessa forma os objetos de dados locais estão amarrados à memória nova a cada ativação porque um novo registro de ativação é empilhado quando uma chamada é realizada. Sobretudo, os valores para os objetos de dados locais são *removidos* quando a ativação termina, isto é, os valores estão perdidos porque a memória para os mesmos desaparece quando o registro de ativação é desempilhado.

Primeiro, descrevemos a forma para uma alocação de memória de pilha, na qual os tamanhos de todos os registros de ativação são conhecidos em tempo de compilação. As situações nas quais estão disponíveis informações incompletas a respeito dos tamanhos em tempo de compilação são consideradas mais adiante.

Suponhamos que o registrador *topo* marque o topo da pilha. Em tempo de execução, um registro de ativação pode ser alocado e liberado incrementando-se e decrementando-se *topo*, respectivamente, pelo tamanho do registro. Se o procedimento q possui um registro de ativação de tamanho *a*, então *topo* é incrementado de *a* exatamente antes do código alvo de q ser executado. Quando o controle retorna de q, *topo* é decrementado de *a*.

Exemplo 7.5. A Fig. 7.13 mostra os registros de ativação que são empilhados e desempilhados em tempo de execução à medida que o controle flui através da árvore de ativações da Fig. 7.3. As linhas pontilhadas na árvore vão para ativações que já terminaram. A execução começa com uma ativação do procedimento s. Quando o controle atinge a primeira chamada no corpo de s, o procedimento r é ativado e seu registro de ativação é empilhado. Quando o controle retorna dessa ativação, o registro é desempilhado deixando somente o registro para s na pilha. Na ativação de s, o controle atinge a chamada de q com parâmetros atuais 1 e 9 e um registro de ativação, para a ativação de q, é alocado ao topo da pilha. Sempre que o controle estiver numa ativação, seu registro de ativação estará ao topo da pilha.

Várias ativações ocorrem entre os dois instantâneos da Fig. 7.13. No último instantâneo, as ativações p(1,3) e q(1,0) começaram e terminaram durante o tempo de vida de q(1,3) e, dessa forma, seus registros de ativação vieram e se foram da pilha, deixando o registro de ativação para q(1,3) ao topo. □

Num procedimento Pascal, podemos determinar o endereço para os dados locais no registro de ativação, como discutido na Seção 7.2.

Posição na Árvore de Ativações	Registros de Ativação na Pilha	Comentários
s	s a : array	Moldura para s
s r	s a : array r i : integer	r é ativado
s r q(1,9)	s a : array q(1,9) i : integer	A moldura para r foi desempilhada e a de q(1,9) empilhada
s r q(1,9) p(1,9) q(1,3) p(1,3) q(1,0)	s a : array q(1,9) i : integer q(1,3) i : integer	O controle acabou de retornar para q(1,3)

Fig. 7.13. Alocação de registros de ativação numa pilha que cresce para baixo.

Em tempo de execução, suponhamos que *topo* marque a localização do fim de um registro. O endereço para um nome local x no código-alvo para o procedimento poderia, conseqüentemente, ser escrito como *dx*(*topo*), para indicar que os dados amarrados a x podem ser encontrados pela adição de *dx* ao valor no registrador *topo*. Note-se que os endereços podem ser alternativamente obtidos como deslocamentos a partir do valor de qualquer outro registrador *r* apontando para uma localização fixa no registro de ativação.

Seqüências de Chamada

As chamadas de procedimento são implementadas através da geração do que é denominado de *seqüências de chamada* no código-alvo. Uma *seqüência de chamada* aloca um registro de ativação e introduz informações em seus campos. Uma *seqüência de retorno* restaura o estado da máquina de tal forma que o procedimento chamador possa continuar a sua execução.

As seqüências de chamada e os registros de ativação diferem, mesmo para as implementações de uma mesma linguagem. O código numa seqüência de chamada é freqüentemente dividido entre o procedimento chamador (o chamador) e o procedimento que o mesmo chama (o procedimento chamado). Não existe uma divisão exata das tarefas em tempo de execução entre o procedimento chamador e o chamado — a linguagem-fonte, a máquina-alvo e o sistema operacional impõem exigências que podem favorecer uma solução em detrimento de outra.[4]

Um princípio que auxilia o projeto das seqüências de chamada e dos registros de ativação é que os campos cujos tamanhos sejam fixados mais cedo sejam colocados ao meio. No molde geral de um registro de ativação da Fig. 7.8, o elo de controle, o de acesso e os campos de estado da máquina fugiram ao meio. A decisão a respeito de se usar ou não os elos de controle e de acesso é parte do projeto do compilador, de forma que esses campos podem ser fixados em tempo de construção do compilador. Se exatamente a mesma quantidade de informações de estado da máquina for guardada a cada ativação, o mesmo código pode realizar o armazenamento e restauração para todas as ativações. Sobretudo, programas tais como os depuradores terão a vida facilitada ao decifrar o conteúdo da pilha quando ocorrer um erro.

Ainda que o tamanho do campo para os objetos de dados temporários seja eventualmente estabelecido em tempo de compilação, esse

[4]Se um procedimento for chamado *n* vezes, o trecho da seqüência de chamada é gerado *n* vezes, nos diversos chamadores. No entanto, o trecho no procedimento chamado é compartilhado por todas as chamadas e, dessa forma, é gerado somente uma vez. Por conseguinte, é desejável colocar tanto quanto possível da seqüência de chamada dentro do procedimento chamado.

tamanho pode não ser conhecido pela vanguarda do compilador. Uma criteriosa geração de código ou de otimização pode reduzir o número de objetos de dados temporários necessitados pelo procedimento mas, na medida em que a vanguarda do compilador estiver envolvida, o tamanho desse campo é desconhecido. Por conseguinte, no modelo genérico de registro de ativação, mostramos esse campo após aquele para os dados locais, onde as mudanças em seu tamanho não afetam os deslocamentos relativos dos objetos de dados dos campos no meio do registro de ativação.

Uma vez que cada chamada possui os seus próprios parâmetros, o chamador usualmente avalia os parâmetros atuais e os comunica ao registro de ativação do procedimento chamado. Os métodos de transmissão de parâmetros são discutidos na Seção 7.5. Na pilha em tempo de execução, o registro de ativação do chamador está exatamente abaixo daquele para o chamado, como na Fig. 7.14. Há uma vantagem em se colocar os campos para os parâmetros e o valor potencialmente retornado em seguida ao registro de ativação do chamador. O chamador poderá ter acesso a esses campos utilizando deslocamentos a partir do fim de seu próprio registro de ativação, sem conhecer a disposição de dados do registro de ativação do procedimento chamado. Em particular, não existe razão para o chamador saber a respeito de dados locais ou temporários do procedimento chamado. Um benefício dessa ocultação de informações está em que os procedimentos com um número variável de argumentos, tais como `printf` em C, podem ser tratados, como discutido abaixo.

Linguagens como Pascal requerem que os *arrays* locais aos procedimentos tenham um tamanho que possa ser determinado em tempo de compilação. Mais freqüentemente, o comprimento de um *array* local pode depender do valor de um parâmetro transmitido a um procedimento. Nesse caso, o tamanho de todos os dados locais ao procedimento não pode ser determinado até que o procedimento seja chamado. As técnicas para o tratamento de dados de tamanho variável são discutidas adiante nesta seção.

A seqüência de chamada seguinte é motivada pela discussão acima. Como na Fig. 7.14, o registrador *ap_topo* aponta para o final do campo de estado da máquina no registro de ativação. Esta posição é conhecida pelo chamador, de forma que o mesmo pode ser tornado responsável pelo estabelecimento de *ap_topo* antes do controle fluir para o procedimento chamado. O código para o procedimento chamado pode ter acesso a seus temporários e dados locais usando deslocamentos a partir de *ap_topo*. A seqüência de chamada é:

1. O chamador avalia seus parâmetros atuais.
2. O chamador armazena um endereço de retorno e o valor antigo de *ap_topo* no registro de ativação do procedimento chamado. O chamador então incrementa *ap_topo* para a posição mostrada na Fig. 7.14. Isto é, *ap_topo* é movido para além dos dados locais e temporários do chamador e dos parâmetros e campos de estado do procedimento chamado.
3. O procedimento chamado salva os valores dos registradores e outras informações de estado.
4. O procedimento chamado inicializa seus dados locais e começa a execução.

Uma possível seqüência de retorno é:

1. O procedimento chamado coloca um valor de retorno em seguida ao registro de ativação do chamador.
2. Usando as informações do campo de estado da máquina, o procedimento chamado restaura *ap_topo* e outros registradores e desvia para um endereço de retorno no código do procedimento chamador.
3. Apesar de *ap_topo* ter sido decrementado, o procedimento chamador pode copiar o valor retornado em seu próprio registro de ativação e usá-lo para avaliar uma expressão.

As seqüências de chamada acima permitem que o número de argumentos do procedimento chamado dependa da chamada. Note-se que, em tempo de compilação, o código-alvo do chamador conhece o número de argumentos que está fornecendo ao procedimento chamado. Por conseguinte, o chamador conhece o tamanho do campo de parâmetros. Entretanto, o código-alvo do procedimento chamado precisa ser preparado para tratar outras chamadas igualmente e, conseqüentemente, espera até ser chamado para, então, examinar o campo de parâmetros. Usando-se a organização da Fig. 7.14, as informações descrevendo os parâmetros precisam ser contíguas ao campo de estado, para que o procedimento chamado possa encontrá-las. Por exemplo, consideremos a função de biblioteca padrão `printf` da linguagem C. O primeiro argumento de `printf` especifica a natureza dos argumentos re-

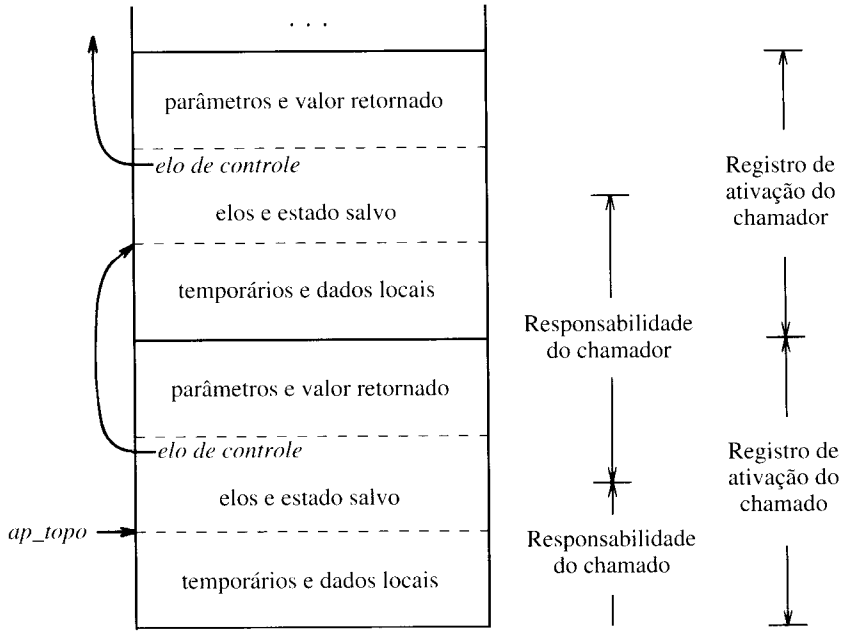


Fig. 7.14. Divisão de tarefas entre o chamador e o chamado.

manescentes e uma vez que `printf` tenha encontrado o primeiro argumento, poderá localizar os restantes.

Dados de Tamanho Variável

Uma estratégia comum para tratar dados de tamanho variável é sugerida na Fig. 7.15, onde um procedimento `p` possui três *arrays* locais. O armazenamento para esses *arrays* não faz parte do registro de ativação para `p`; somente um apontador para o início de cada *array* figura no registro de ativação. Os endereços relativos desses apontadores são conhecidos em tempo de compilação e, por conseguinte, o código-alvo pode ter acesso aos elementos do *array* através dos apontadores.

Igualmente mostrado na Fig. 7.15 é o procedimento `q`, chamado por `p`. O registro de ativação para `q` começa após os *arrays* de `p` e os *arrays* de tamanho variável de `q` começam além daí.

O acesso aos dados na pilha se dá através de dois apontadores, *topo* e *ap_topo*. O primeiro deles marca o topo atual da pilha; aponta para a posição na qual o próximo registro de ativação irá começar. O segundo é usado para se encontrar os dados locais. Por uma questão de consistência com a organização da Fig. 7.14, suponhamos que *ap_topo* aponte para o final do campo de estado da máquina. Na Fig. 7.15, *ap_topo* aponta para o final desse campo no registro de ativação para `q`. Dentro desse campo há um elo de controle apontando para o valor prévio de *ap_topo*, quando o controle estava na ativação chamadora de `p`.

O código para reposicionar *topo* e *ap_topo* pode ser gerado em tempo de compilação, usando os tamanhos dos campos nos registros de ativação. Quando `q` retorna, o novo valor de *topo* é *ap_topo*, menos o tamanho dos campos de estado da máquina e de parâmetros no registro de ativação de `q`. Esse comprimento é conhecido em tempo de compilação, pelo menos pelo chamador. Após ajustar *topo*, o novo valor de *ap_topo* pode ser copiado a partir do elo de controle de `q`.

Referências Ocas

Sempre que a memória puder ser liberada, emerge o problema das referências ocas. Uma *referência oca* ocorre quando existe uma referência a uma memória que já foi liberada. O uso das referências ocas é um erro

```
main( )
{
    int *p;
    p = dangle( );
}
int *dangle( )
{
     int i = 23;
     return &i;
}
```

Fig. 7.16. Um programa C que deixa `p` apontando para a memória liberada.

lógico, uma vez que o valor da memória liberada é indefinido, de acordo com a semântica da maioria das linguagens de programação. Pior, como aquela memória pode ser posteriormente reservada para um outro dado, erros misteriosos podem aparecer nos programas que tenham referências ocas.

Exemplo 7.6. O procedimento `dangle` no programa C da Fig. 7.16 retorna um apontador para a memória amarrada ao nome local `i`. O apontador é criado pelo operador & aplicado a `i`. Quando o controle retorna para `main` proveniente de `dangle`, a memória para os objetos de dados locais é liberada e pode ser usada para outros propósitos. Uma vez que `p` em `main` se refere a essa memória, o uso de `p` é uma referência oca. □

Um exemplo da Seção 7.7 envolve a liberação feita sob controle do programa.

Alocação de Memória *Heap*

A estratégia de alocação de memória de pilha discutida acima não pode ser usada se uma das duas situações for possível:

1. Os valores dos nomes locais precisarem ser retidos quando a ativação terminar.
2. A ativação chamada sobreviver ao chamador. Essa possibilidade não pode ocorrer para aquelas linguagens onde as árvores de ativação delineam corretamente o fluxo de controle entre os procedimentos.

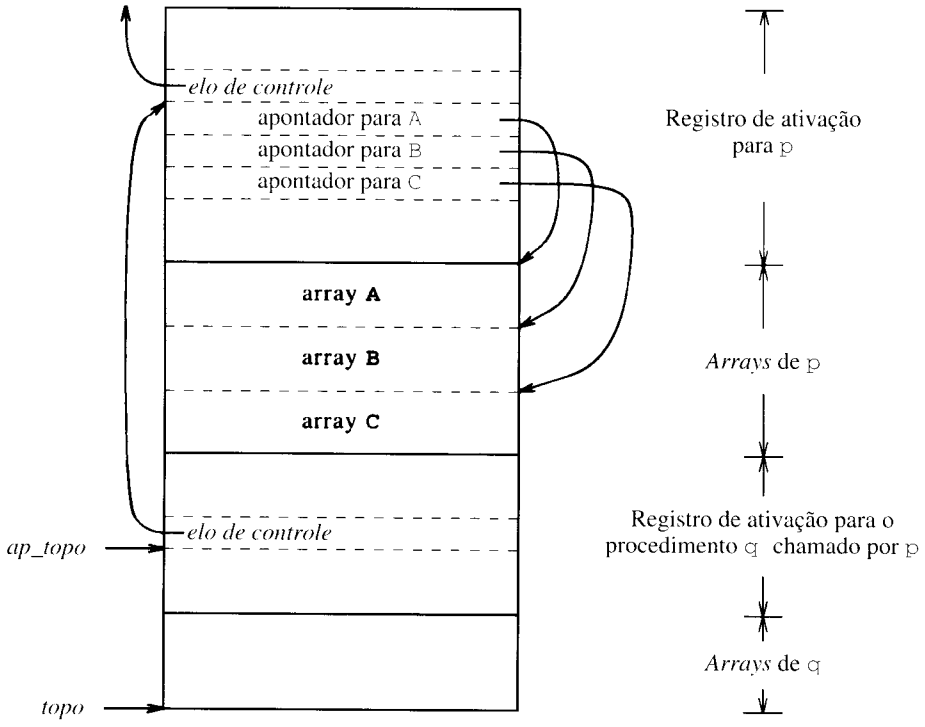


Fig. 7.15. Acesso a *arrays* alocados dinamicamente.

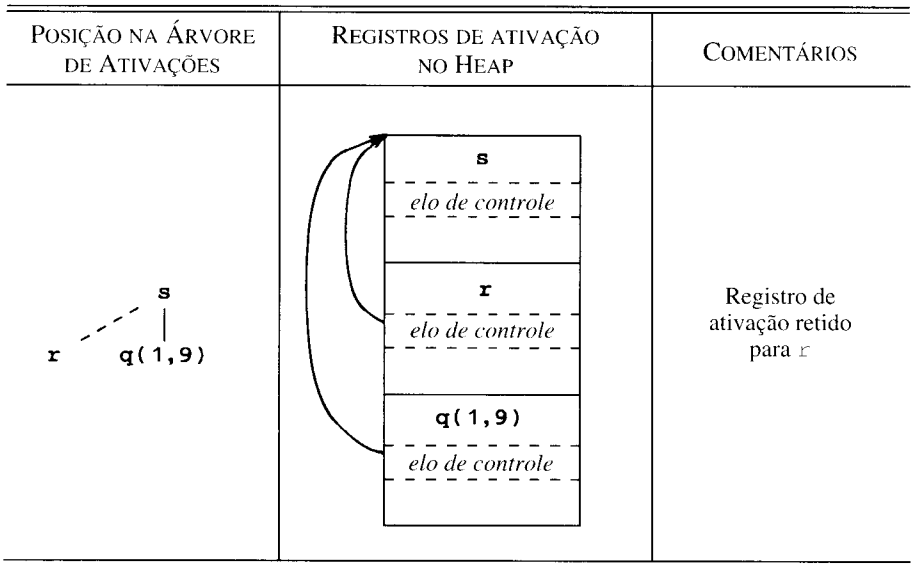


Fig. 7.17. Os registros para as ativações vivas não precisam ser adjacentes no *heap*.

Em cada um dos casos acima, a liberação de registros de ativação não necessita ocorrer numa modalidade FIFO, de forma que a memória não pode ser organizada como uma pilha.

A alocação de memória *heap* fornece blocos de memória contígua, na medida do necessitado, para os registros de ativação ou outros objetos. As áreas de memória podem ser liberadas em qualquer ordem, de forma que, ao longo do tempo, o *heap* irá se constituir de áreas alternativamente livres e em uso.

A diferença entre a alocação de memória *heap* e a de pilha para os registros de ativação pode ser vista a partir das Figs. 7.17 e 7.13. Na Fig. 7.17, o registro para uma ativação do procedimento `r` é retido ao término da mesma. Por conseguinte, o registro para a nova ativação `q(1,9)` não pode seguir fisicamente àquele para `s`, como feito na Fig. 7.13. Agora, se o registro de ativação retido para `r` for liberado existirá espaço livre no *heap* entre os registros de ativação para `s` e `q(1,9)`. É deixado para o gerenciador do *heap* fazer uso desse espaço.

A questão do gerenciamento eficiente de memória *heap* é um tema um tanto especializado na teoria das estruturas de dados; algumas técnicas são revistas na Seção 7.8. Existe geralmente alguma sobrecarga no tempo e espaço associados ao uso de um gerenciador de memória *heap*. Por razões de eficiência, pode ser vantajoso manipular pequenos registros de ativação ou registros de tamanhos previsíveis como casos especiais, tal como segue:

1. Para cada tamanho específico de interesse, manter uma lista ligada de blocos livres daquele tamanho.
2. Se possível, atender a uma requisição para um tamanho s com um bloco de tamanho s', onde s' é o menor tamanho maior ou igual a s. Quando o bloco for eventualmente liberado, é retornado à lista ligada da qual é oriundo.
3. Para grandes blocos de memória, usar o gerenciador de *heap*.

Esse enfoque resulta na rápida alocação e liberação de pequenas quantidades de memória, uma vez que a obtenção e o retorno de um bloco, a partir de uma lista ligada, são operações eficientes. Para grandes quantidades de memória, esperamos que a computação tome algum tempo para se utilizar da memória e, conseqüentemente, o tempo gasto pelo alocador é freqüentemente desprezível comparado ao tempo da computação.

7.4 ACESSO AOS NOMES NÃO LOCAIS

As estratégias de alocação de memória da última seção são adaptadas nesta seção de forma a permitir o acesso aos nomes não locais. Apesar da discussão estar baseada na reserva de memória de pilha para os registros de ativação, as mesmas idéias se aplicam à alocação de memória *heap*.

As regras de escopo de uma linguagem determinam o tratamento das referências aos nomes não locais. Uma regra comum, chamada de *regra de escopo léxico ou estático*, determina a declaração que se aplica a um nome pelo exame isolado do texto do programa. Pascal, C e Ada estão entre as muitas linguagens que usam o escopo léxico, com a estipulação adicional do "aninhamento mais interno", o qual é discutido abaixo. Uma regra alternativa, chamada de *regra do escopo dinâmico*, determina a declaração aplicável a um nome em tempo de execução pela consideração das ativações correntes. Lisp, Apl e Snobol estão dentre as linguagens que usam o escopo dinâmico.

Começamos pelos blocos e a regra do "aninhamento mais interno". Consideremos, então, os nomes não locais em linguagens como C, onde os escopos são léxicos e todos os nomes não locais podem ser amarrados à memória alocada estaticamente e nenhuma declaração aninhada de procedimento é permitida.

Em linguagens como Pascal, que possuem procedimentos aninhados e escopo léxico, os nomes pertencentes a procedimentos diferentes podem ser parte do ambiente em um dado instante. Discutimos duas formas de se encontrar os registros de ativação contendo a memória amarrada aos nomes não locais: elos de acesso e *displays*.

Uma subseção final discute a implantação do escopo dinâmico.

Blocos

Um bloco é um enunciado contendo suas próprias declarações de dados locais. O conceito de bloco foi originado em Algol. Em C, um bloco possui a sintaxe.

{ *declarações comandos* }

Uma característica dos blocos é a sua estrutura de aninhamento. Os delimitadores marcam o início e o final de um bloco. C utiliza chaves, { e }, enquanto que a tradição de Algol é usar `begin` e `end`. Os delimitadoress asseguram que um bloco ou é independente ou está aninhado dentro de um outro. Isto é, não é possível para dois blocos B_1 e B_2 se sobreporem, de forma que o primeiro, B_1, comece, em seguida B_2, mas que B_1 termine antes de B_2 terminar. Essa propriedade de aninhamento é algumas vezes referenciada como *estrutura de bloco*.

O escopo de uma declaração numa linguagem estruturada em blocos é dado pela *regra do aninhamento mais interno:*

```
main( )
{
        int a = 0;
        int b = 0;
        {
                int b = 1;
                {
                        int a = 2;
                }
                        printf ("%d %d\n", a, b);
        }
        {
                {
                        int b = 3;
                }
                        printf("%d %d\n", a, b);
                }
                printf("%d %d\n", a, b);
        }
        printf("%d %d\n", a, b);
}
```

DECLARAÇÃO	ESCOPO
int a = 0;	$B_0 - B_2$
int b = 0;	$B_0 - B_1$
int b = 1;	$B_1 - B_3$
int a = 2;	B_2
int b = 3;	B_3

Fig. 7.18. Blocos num programa C.

1. O escopo de uma declaração num bloco B inclui B.
2. Se um nome x não for declarado num bloco B, uma ocorrência de x em B estará no escopo de uma declaração de x num bloco envolvente B' tal que
   i) B' possui uma declaração para x e
   ii) B' é o bloco mais internamente aninhado envolvendo B, em relação a qualquer outro bloco que contenha uma declaração para x.

Por projeto, cada declaração da Fig. 7.18 inicializa cada nome declarado com o número de bloco no qual a mesma aparece. O escopo da declaração de b em B_0 não inclui B_1, porque b é redeclarado em B_1, indicado por $B_0 - B_1$ na figura. Uma tal ausência é chamada de um *buraco* no escopo de uma declaração.

A regra do aninhamento mais interno é refletida pela saída do programa da Fig. 7.18. O controle flui para um bloco a partir de um ponto exatamente antes do mesmo e flui a partir do bloco para o ponto exatamente após o mesmo no texto-fonte. Os enunciados de impressão são, por conseguinte, executados na ordem B_2, B_3, B_1 e B_0, a ordem na qual o controle deixa os blocos. Os valores de a e de b nesses blocos são:

```
2  1
0  3
0  1
0  0
```

A estrutura de blocos pode ser implementada usando-se uma alocação de memória de pilha. Uma vez que o escopo de uma declaração não se estende para fora do bloco na qual aparece, o espaço para o nome declarado pode ser reservado quando o controle entra no bloco e liberado quando o deixa. Essa visão trata os blocos como "procedimentos sem parâmetros", chamados somente a partir do ponto exatamente antes do bloco e retornando somente para o ponto exatamente após o mesmo. O ambiente não local para um bloco pode ser mantido usando-se as técnicas para os procedimentos examinados mais adiante nesta seção. Note-se, entretanto, que os blocos são mais simples do que os procedimentos, porque não há parâmetros passados e o fluxo de controle para o bloco e a partir do bloco segue o texto estático do programa.[5]

[5]Um salto para fora de um bloco em direção a um bloco envolvente pode ser implementado desempilhando-se os registros de ativação dos blocos intervenientes. Um salto para dentro de um bloco é permitido em algumas linguagens. Antes do controle ser transferido dessa forma, os registros de ativação têm que ser estabelecidos para os blocos intervenientes. A semântica da linguagem determina como os dados locais nesses blocos são inicializados.

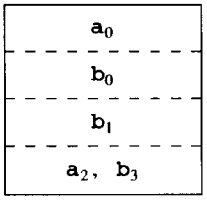


Fig. 7.19. Armazenamento para os nomes declarados na Fig. 7.18.

Uma implementação alternativa consiste em se reservar memória para todo o corpo do procedimento de uma só vez. Se existirem blocos dentro de um procedimento, é feita provisão para o armazenamento necessitado pelas declarações dentro dos blocos. Para o bloco B_0 na Fig. 7.18, podemos reservar memória como na Fig. 7.19. Os subscritos dos objetos de dados locais para a e b identificam os blocos nos quais são declarados. Note-se que a_2 e b_3 podem ser associados à mesma memória porque estão em blocos que não estão vivos ao mesmo tempo.

Na ausência de dados de tamanho variável, a quantidade máxima de memória necessitada durante a execução de um bloco pode ser determinada em tempo de compilação (dados de tamanho variável podem ser tratados usando-se apontadores, como na Seção 7.3). Ao se fazer essa determinação, assumimos conservativamente que todos os percursos de controle no programa possam de fato ser executados. Isto é, assumimos que ambas as partes de um *if-then-else* possam ser executadas e que todos os comandos dentro de laço *while* possam ser atingidos.

Escopo Léxico sem Procedimentos Aninhados

As regras de escopo estático para C são mais simples do que aquelas para Pascal, discutidas em seguida, porque as definições de procedimentos não podem ser aninhadas em C. Isto é, uma definição de procedimento não pode figurar dentro de outra. Como na Fig. 7.20, um programa C consiste em uma seqüência de declarações de variáveis e procedimentos (C as chama de funções). Se existir uma referência não local a um nome a em alguma função, a precisará ser declarado fora de qualquer função. O escopo de uma declaração fora de uma função consiste nos corpos das funções que seguem a declaração, com os buracos, se o nome for redeclarado dentro de uma função. Na Fig. 7.20, as ocorrências não locais de a em readarray, partition e main se referem ao *array* declarado à linha 1.

```
(1) int a[11];
(2) readarray ( ) { ... a ... }
(3) int partition (y,z) int y, z; { ... a ... }
(4) quicksort (m,n) int m, n; { ... }
(5) main ( ) { ... a ... }
```

Fig. 7.20. Programa C com ocorrências não locais de a.

Na ausência de procedimentos aninhados, a estratégia de alocação de pilha da Fig. 7.3, para os nomes locais, pode ser usada diretamente para uma linguagem com escopo léxico como C. O armazenamento para todos os nomes declarados fora de quaisquer procedimentos pode ser alocado estaticamente. A posição desse armazenamento é conhecida em tempo de compilação e, dessa forma, se um nome for não local ao corpo de um procedimento, usamos simplesmente o endereço determinado estaticamente. Qualquer outro nome terá que ser local à ativação que está ao topo da pilha, acessível através do apontador *topo*. Os procedimentos aninhados fazem esse esquema falhar porque um nome não local pode estar relacionado a dados no fundo da pilha, como será discutido abaixo.

Um benefício importante da alocação estática para nomes não locais está em que os procedimentos podem ser transmitidos como

parâmetros e retornados como resultados (uma função em C é transmitida usando-se um apontador para a mesma). Com o escopo léxico e sem procedimentos aninhados, qualquer nome não local a um procedimento é não local a todos os procedimentos. Seu endereço estático pode ser usado em todos os procedimentos, independentemente de como sejam ativados. Similarmente, se os procedimentos são retornados como resultado, os nomes não locais nos procedimentos retornados se referem à memória reservada estaticamente para os mesmos.

Por exemplo, consideremos o programa Pascal da Fig. 7.21. Todas as ocorrências do nome m, circuladas na Fig. 7.21, estão no escopo da declaração da linha 2. Uma vez que m é não local a todos os procedimentos no programa, sua memória pode ser reservada estaticamente. Sempre que os procedimentos f e g são executados, podem usar o endereço estático para ter acesso ao valor de m. O fato de f e g serem passados como parâmetros afeta somente o instante de suas ativações; não afeta como têm acesso ao valor de m.

```
(1)  program pass (input, output);
(2)    var (m) : integer;

(3)    function f (n : integer) : integer;
(4)      begin f := (m) + n end { f };

(5)    function g(n : integer) : integer;
(6)      begin g := (m) * n end { g };

(7)    procedure b (function h(n : integer) : integer);
(8)      begin write(h(2)) end { b };

(9)    begin
(10)     (m) := 0;
(11)     b(f); b(g); writeln
(12)   end.
```

Fig. 7.21. Programa Pascal com ocorrências não locais de m.

Em mais detalhes, a chamada b(f) à linha 11 associa a função f com o parâmetro formal h do procedimento b. Dessa forma, quando o parâmetro formal h é chamado à linha 8, em write(h(2)), a função f é ativada. A ativação de f retorna 2 porque o objeto de dados associado ao nome não local m possui valor 0 e o parâmetro formal n possui valor 2. No seguimento da execução, a chamada b(g) associa g a h; dessa vez, a chamada de h ativa g. A saída do programa é

```
2   0
```

Escopo Léxico com Procedimentos Aninhados

Uma ocorrência não local de um nome a num procedimento Pascal está no escopo da declaração de a mais internamente aninhada, envolvendo o procedimento no texto estático do programa.

O aninhamento das definições de procedimentos no programa Pascal da Fig. 7.22 é indicado pela seguinte indentação:

```
sort
    readarray
    exchange
    quicksort
        partition
```

A ocorrência de a, à linha 15 na Fig. 7.22, está dentro da função partition, a qual está aninhada no procedimento quicksort. A declaração de a mais proximamente aninhada está à linha 2 do procedimento que consiste no próprio programa. A regra do aninhamento mais interno se aplica a nomes de procedimentos igualmente. O procedimento exchange, chamado por partition à linha 17, é não local a partition. Aplicando a regra, primeiro verificamos se exchange está

```
(1)  program sort(input, output);
(2)    var a : array [0 .. 10] of integer;
(3)        x : integer;

(4)    procedure readarray;
(5)      var i : integer;
(6)      begin ... a ... end { readarray };

(7)    procedure exchange ( i, j: integer);
(8)      begin
(9)          x := a[i]; a[i] := a[j]; a[j] := x
(10)     end { exchange };
(11)   procedure quicksort (m, n: integer);
(12)     var k, v : integer;

(13)     function partition(y, z: integer) : integer;
(14)         var i, j : integer;
(15)         begin ... a ...
(16)               ... v ...
(17)               ... exchange (i,j); ...
(18)         end { partition } ;

(19)     begin ... end { quicksort };

(20)   begin ... end { sort }
```

Fig. 7.22. Programa Pascal com procedimentos aninhados.

definido dentro de quicksort; como não está, procuramos no programa principal sort.

Profundidade de Aninhamento

A noção de *profundidade de aninhamento* de um procedimento é usada abaixo para implementar o escopo léxico. Façamos o nome do programa principal estar à profundidade de aninhamento 1; adicionamos 1 à profundidade de aninhamento à medida que nos encaminhamos de um procedimento envolvente para um procedimento envolvido. Na Fig. 7.22, o procedimento quicksort à linha 11 está à profundidade de aninhamento 2, enquanto que partition à linha 13 está à profundidade de aninhamento 3. A cada ocorrência de um nome associamos a profundidade de aninhamento do procedimento no qual é declarado. As ocorrências de a, v e i, às linhas 15-17 em partition, têm, por conseguinte, profundidades de aninhamento 1, 2 e 3, respectivamente.

Elos de Acesso

Uma implementação direta do escopo léxico para procedimentos aninhados é obtida adicionando-se um apontador, chamado de *elo de acesso*, a cada registro de ativação. Se o procedimento p estiver aninhado imediatamente dentro de q no texto-fonte, então o elo de acesso num registro de ativação para p aponta para o elo de acesso do registro da ativação mais recente de q.

Instantâneos da pilha em tempo de execução, durante uma execução do programa na Fig. 7.22, são mostrados na Fig. 7.23. De novo, para economizar espaço na figura, somente a primeira letra de cada nome de procedimento é mostrada. O elo de acesso para a ativação de sort está vazio, já que não há procedimento envolvente. O elo de acesso de quicksort aponta para o registro de sort. Note-se que na Fig. 7.23 o elo de acesso no registro de ativação para partition (1, 3) aponta para o elo de acesso no registro da ativação mais recente de quicksort, explicitamente, quicksort (1, 3).

Suponhamos que o procedimento p à profundidade de aninhamento n_p se refira ao nome não local a com nível de aninhamento $n_a \leq n_p$. O armazenamento de a pode ser encontrado como segue.

1. Quando o controle estiver em p, um registro de ativação para p está ao topo da pilha. Seguir $n_p - n_a$ elos de acesso a partir do registro ao topo da pilha. O valor de $n_p - n_a$ pode ser pré-computado em tempo de compilação. Se o elo de acesso em um registro aponta o elo de

acesso noutro, então um elo pode ser seguido realizando-se uma operação singela de indireção.

2. Após seguir $n_p - n_a$ elos, atingimos um registro de ativação para o procedimento ao qual `a` é local. Como discutido na última seção, seu armazenamento estará a um deslocamento fixo relativo à posição do registro. Em particular, o deslocamento pode ser relativo ao elo de acesso.

Por conseguinte, o endereço associado a um nome não local `a`, num procedimento `p`, é dado pelo seguinte par pré-computado em tempo de compilação e armazenado na tabela de símbolos:

($n_p - n_a$, deslocamento dentro do registro de ativação contendo `a`)

O primeiro componente fornece o número de elos de acesso a serem atravessados.

Por exemplo, nas linhas 15-16 da Fig. 7.22, o procedimento `partition`, à profundidade de aninhamento 3, referencia os nomes não locais `a` e `v` às profundidades de aninhamento 1 e 2, respectivamente. Os registros de ativação contendo a memória para esses nomes não locais são encontrados seguindo-se 3−1=2 e 3−2=1 elos de acesso, respectivamente, a partir do registro de `partition`.

O código para estabelecer os elos de acesso é parte da seqüência de chamada. Suponhamos que o procedimento `p` com profundidade de aninhamento n_p chame o procedimento `x` à profundidade de aninhamento n_x. O código para estabelecer o elo de acesso no procedimento chamado depende desse último estar aninhado ou não no chamador.

1. Caso $n_p < n_x$. Uma vez que o procedimento chamado `x` está aninhado mais profundamente do que `p`, terá que estar declarado dentro de `p`, ou não estaria acessível a `p`. Esse caso ocorre quando `sort` chama `quicksort` na Fig. 7.23(a) e quando `quicksort` chama `partition` na Fig. 7.23(c). Nesse caso, o elo de acesso no procedimento chamado terá de apontar para o elo de acesso no registro de ativação do chamador, exatamente abaixo na pilha.
2. Caso $n_p \geq n_x$. A partir das regras de escopo, os procedimentos envolventes às profundidades de aninhamento 1, 2, . . ., $n_x - 1$, a partir dos procedimentos chamado e chamador, precisam ser os mesmos, como quando `quicksort` chama a si mesmo na Fig. 7.23(b) ou quando `partition` chama `exchange` na Fig. 7.23(d). Seguindo-se $n_p - n_x + 1$ elos de acesso a partir do chamador atingimos o registro de ativação mais recente do procedimento que envolve estática e mais proximamente tanto o procedimento chamado quanto o chamador. O elo de acesso atingido é aquele para o qual o elo de acesso no procedimento chamado precisará apontar. De novo, $n_p - n_x + 1$ pode ser computado em tempo de compilação.

Parâmetros Tipo Procedimento

As regras de escopo léxico se aplicam mesmo quando um procedimento aninhado é passado como parâmetro. A função `f`, às linhas 6-7 do programa Pascal na Fig. 7.24, possui um nome `m` não local; todas as ocorrências de `m` estão circuladas na figura. À linha 8, o procedimento `c` atribui 0 a `m` e então passa `f` como parâmetro a `b`. Note-se que o escopo da declaração de `m` à linha 5 não inclui o corpo de `b` às linhas 2-3.

```
(1)   program param(input, output);

(2)       procedure b(function h(n:integer): integer);
(3)           begin writeln(h(2)) end {b};

(4)       procedure c;
(5)           var (m) : integer;

(6)           function f (n : integer) : integer;
(7)             begin f := (m) + n end {f};

(8)           begin (m) := 0; b(f) end {c};

(9)       begin
(10)      c
(11)      end.
```

Fig. 7.24. Um elo de acesso precisa ser passado com o parâmetro atual `f`.

Dentro do corpo de `b`, o enunciado `writeln(h(2))` ativa `f` porque o parâmetro formal `h` se refere a `f`. Isto é, `writeln` imprime o resultado da chamada `f(2)`.

Como estabelecemos o elo de acesso para a ativação de `f`? A resposta é que um procedimento aninhado, que seja passado como parâmetro, precisa carregar seu elo de acesso junto, como mostrado na Fig. 7.25. Quando o procedimento `c` passa `f`, determina o elo de acesso para `f`, como faria se estivesse chamando `f`. Esse elo é passado junto com `f` para `b`. Subseqüentemente, quando `f` for ativado a partir de `b`, o elo é usado para estabelecer o elo de acesso no registro de ativação para `f`.

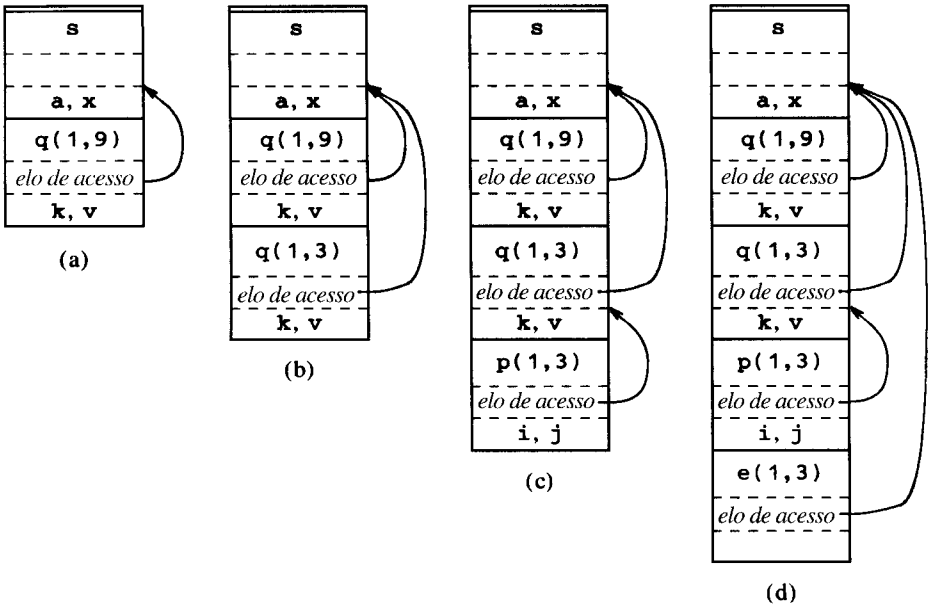


Fig. 7.23. Elos de acesso para se encontrar o armazenamento de nomes não locais.

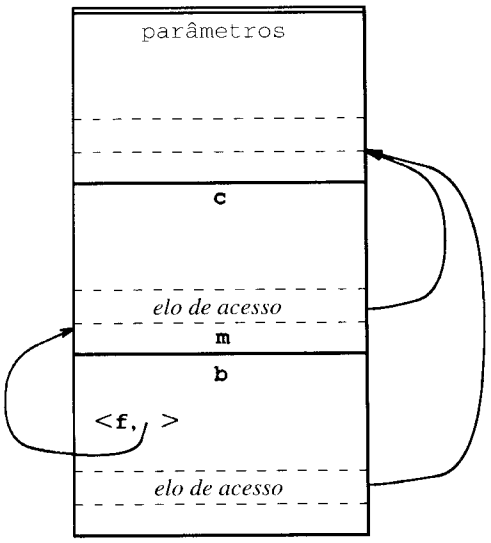


Fig. 7.25. O parâmetro atual tipo procedimento carrega junto seu elo de acesso .

Displays

Um acesso aos nomes não locais, mais rápido do que o obtido através dos elos de acesso, pode ser atingido usando-se um *array d* de apontadores para os registros de ativação, chamado *display*. Mantemos o *display* de tal forma que a memória para um não-local à profundidade de aninhamento *i* esteja no registro de ativação apontado pelo elemento *d*[*i*] do *display*.

Suponhamos que o controle esteja na ativação de um procedimento `p` à profundidade de aninhamento *j*. Então, os primeiros $j - 1$ elementos do *display* apontam para as ativações mais recentes dos procedimentos que envolvem lexicamente o procedimento `p` e *d*[*j*] aponta para a ativação de `p`. Usar um *display* é geralmente mais rápido do que seguir os elos de acesso, porque o registro de ativação que abriga um não local é encontrado através do acesso a um elemento de `d`, ao invés de se seguir exatamente um apontador.

Um arranjo simples para se manter o *display* usa elos de acesso adicionalmente ao *display*. Como parte das seqüências de chamada e retorno, o *display* é atualizado de acordo com a cadeia de elos de acesso. Quando um elo para um registro de ativação à profundidade de aninhamento *n* é seguido, o elemento do *display d*[*n*] é estabelecido de forma a apontar para o registro de ativação. Com efeito, o *display* duplica as informações na cadeia de elos de acesso.

O arranjo simples acima pode ser melhorado. O método ilustrado na Fig. 7.26 requer menos trabalho à entrada e à saída de um procedimento no caso usual em que os procedimentos não são passados como parâmetros. Na Fig. 7.26, o *display* consiste em um *array* global, separado da pilha. Os instantâneos na figura se referem à execução do texto-fonte na Fig. 7.22. De novo, somente a primeira letra de cada procedimento é mostrada.

A Fig. 7.26(a) mostra a situação exatamente antes da ativação `q(1,3)` começar. Como `quicksort` está à profundidade de aninhamento 2, o elemento do `display` *d*[2] é afetado quando uma nova ativação de `quicksort` começa. O efeito da ativação `q(1,3)` sobre *d*[2] é mostrado na Fig. 7.26(b), onde *d*[2] agora aponta para um

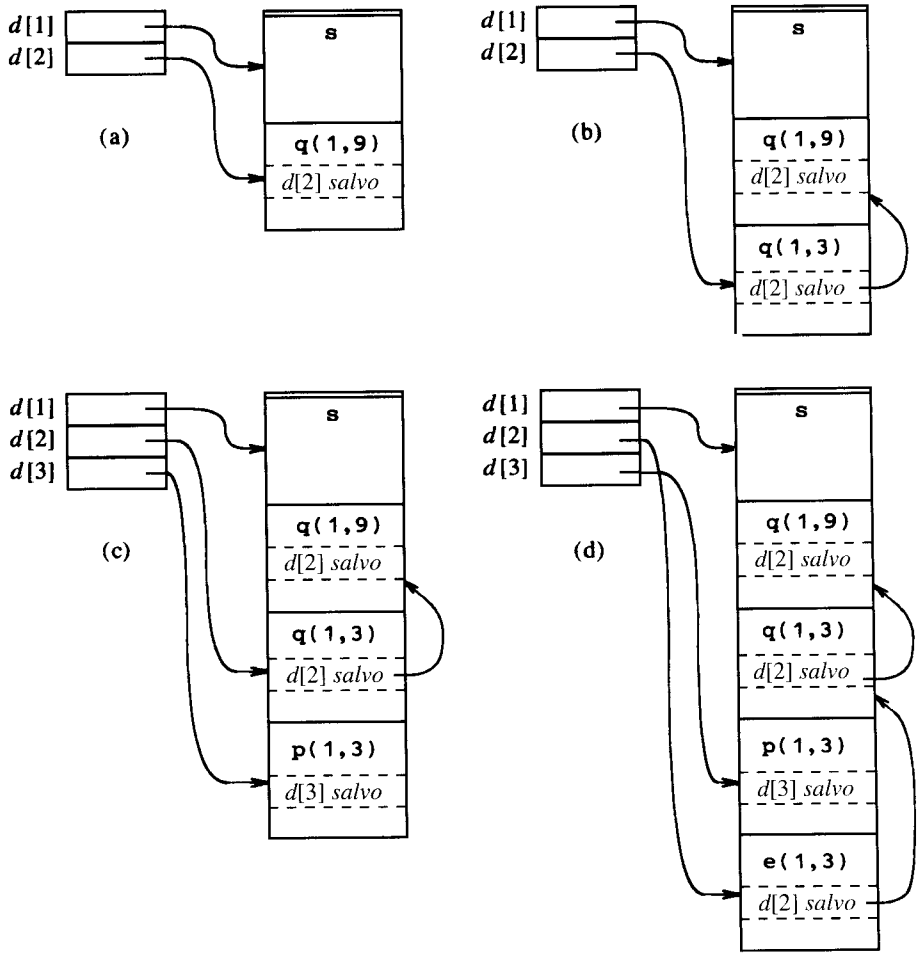


Fig. 7.26. Mantendo o *display* quando procedimentos não são transmitidos como parâmetros.

novo registro de ativação; o valor antigo de $d[2]$ é salvo dentro do novo registro de ativação.[6] O valor salvo será necessitado mais tarde para restaurar o *display* a seu estado na Fig. 7.26(a), quando o controle retornar à ativação `q(1,9)`.

O *display* muda quando ocorre uma nova ativação e precisa ser restabelecido quando o controle retorna da nova ativação. As regras de escopo de Pascal e de outras linguagens com escopo léxico permitem que o *display* seja mantido através dos seguintes passos. Discutimos somente o caso fácil no qual os procedimentos não são passados como parâmetros (ver o Exercício 7.8). Quando um novo registro de ativação para um procedimento à profundidade de aninhamento i é estabelecido,

1. salvamos o valor de $d[i]$ no novo registro de ativação e
2. fazemos $d[i]$ apontar para o novo registro de ativação.

Exatamente antes da ativação terminar, $d[i]$ é restabelecido com o valor salvo.

Esses passos são justificados como se segue. Suponhamos que um procedimento à profundidade de aninhamento j chame um procedimento à profundidade i. Existem dois casos, dependendo do procedimento chamado estar ou não aninhado no chamador, no texto-fonte do programa, como na discussão dos elos de acesso.

1. Caso $j < i$. Então $i = j + 1$ e o procedimento chamado está aninhado dentro do chamador. Os primeiros j elementos do *display* conseqüentemente não precisam ser modificados e fazemos $d[i]$ apontar para o novo registro de ativação. Esse caso é ilustrado na Fig. 7.26(a), quando `sort` chama `quicksort` e também quando `quicksort` chama `partition` na Fig. 7.26(c).
2. Caso $j \geq i$. De novo, os procedimentos envolventes às profundidades $1, 2, \ldots, i - 1$ do procedimento chamado e do chamador precisam ser os mesmos. Aqui, precisamos guardar o valor antigo de $d[i]$ no novo registro de ativação e fazer $d[i]$ apontar para esse mesmo novo registro. O *display* é mantido corretamente porque os primeiros $i - 1$ elementos são deixados como estão.

Um exemplo do Caso 2, com $i = j = 2$, ocorre quando `quicksort` é chamado recursivamente na Fig. 7.26(b). Um exemplo mais interessante ocorre quando a ativação `p(1,3)`, à profundidade de aninhamento 3, chama `e(1,3)`, à profundidade 2 e o procedimento envolvente de ambos é `s`, à profundidade 1, como na Fig. 7.26(d) (o programa está na Fig. 7.22). Note-se que quando `e(1,3)` é chamado, o valor de $d[3]$, pertencente a `p(1,3)`, ainda está no *display*, apesar de não poder se ter acesso ao mesmo enquanto o controle estiver em `e`. Chame `e` um outro procedimento à profundidade 3, e aquele procedimento armazenará $d[3]$ e a restauraria quando do retorno para `e`. Podemos, por conseguinte, mostrar que cada procedimento vê o *display* correto para todas as profundidades até a sua própria.

Existem vários locais onde um *display* pode ser mantido. Se existirem registradores suficientes, o *display*, descrito como um *array*, pode ser uma coleção de registradores. Note-se que o compilador pode determinar o tamanho máximo desse *array*; é a profundidade máxima de aninhamento dos procedimentos no programa. De outra forma, o *display* pode ser mantido em memória estaticamente alocada e todas as referências aos registros de ativação começam utilizando um endereçamento indireto através do apontador apropriado do *display*. Esse enfoque é razoável numa máquina com endereçamento indireto, apesar de cada indireção custar um ciclo de memória. Outra possibilidade é armazenar o *display* na própria pilha em tempo de execução e criar uma nova cópia do mesmo a cada entrada de procedimento.

[6]Note-se que `q(1,9)` também salvou $d[2]$, apesar de acontecer que o segundo elemento do *display* jamais ter sido usado e não precisar ser restaurado. É mais fácil para as chamadas de `q` armazenar $d[2]$ do que decidir em tempo de execução se tal armazenamento é necessário.

```
(1)  program dynamic (input, output);
(2)    var r : real;

(3)    procedure show;
(4)      begin write ( r : 5:3 ) end;

(5)    procedure small;
(6)      var r : real;
(7)      begin r := 0.125; show end;

(8)    begin
(9)      r := 0.25;
(10)     show; small; writeln;
(11)     show; small; writeln
(12)   end.
```

Fig. 7.27. A saída depende de ser usado o escopo léxico ou o dinâmico.

Escopo Dinâmico

Sob o escopo dinâmico, uma nova ativação herda as amarrações existentes para os nomes não locais na memória. Um nome não local `a`, na ativação chamada, se refere à mesma memória que tinha associada quando na ativação que a chamou. As novas amarrações são estabelecidas para os nomes locais do procedimento chamado; os nomes se referem à memória no novo registro de ativação.

O programa na Fig. 7.27 ilustra o escopo dinâmico. O procedimento `show`, às linhas 3-4, escreve o valor para o `r` não local. Sob o escopo léxico em Pascal, o `r` não local está no escopo da declaração à linha 2, de forma que a saída do programa é

```
0.250   0.250
0.250   0.250
```

Entretanto, sob o escopo dinâmico, a saída é

```
0.250   0.125
0.250   0.125
```

Quando `show` é chamado às linhas 10-11 do programa principal, `0.250` é escrito porque a variável local ao programa principal `r` é usada. Entretanto, quando `show` é chamado à linha 7, de dentro de `small`, `0.125` é escrito, porque a variável `r`, local a `small`, é usada.

Os dois enfoques seguintes para implementar o escopo dinâmico têm alguma semelhança com o uso dos elos de acesso e *displays*, respectivamente, na implementação do escopo léxico.

1. *Acesso profundo.* Conceitualmente, o escopo dinâmico resulta se os elos de acesso apontarem para os mesmos registros de ativação que os elos de controle apontam. Uma implementação simples é se dispensar os elos de acesso e se utilizar os elos de controle para pesquisar na pilha, procurando pelo primeiro registro de ativação que contenha memória para o nome não local. O termo *acesso profundo* provém do fato da procura precisar "ir fundo" na pilha. A profundidade até a qual a procura pode ir depende da entrada para o programa e não pode ser determinada em tempo de compilação.
2. *Acesso superficial.* Aqui a idéia é guardar o valor corrente de cada nome em memória estaticamente reservada. Quando ocorre uma nova ativação de um procedimento `p`, um nome local `n`, em `p`, ocupa a memória reservada estaticamente para `n`. O valor prévio de `n` é salvo no registro de ativação para `p` e precisa ser restaurado quando a ativação de `p` terminar.

A barganha entre os dois enfoques está em que o acesso profundo exige mais tempo para se ter acesso a um nome não local, mas não há sobrecarga associada ao início e término de uma ativação. O acesso superficial, por outro lado, permite que os não locais sejam procurados diretamente, mas é gasto tempo para manter esses valores quando as ativações começam e terminam. Quando as funções são passadas como parâmetros e retornadas como resultados, uma implementação mais direta é obtida através do acesso profundo.

7.5 TRANSMISSÃO DE PARÂMETROS

Quando um procedimento chama outro, o método usual de comunicação entre os mesmos é através de nomes não locais e parâmetros para o procedimento chamado. Ambos, nomes não locais e parâmetros, são usados pelo procedimento da Fig. 7.28 para trocar os valores de `a[i]` e de `a[j]`. Aqui o *array* `a` é não local ao procedimento `swap` e `i` e `j` são parâmetros.

```
(1)  procedure swap (i, j: integer);
(2)     var x : integer;
(3)     begin
(4)        x := a[i]; a[i] := a[j]; a[j] := x
(5)     end
```

Fig. 7.28. O procedimento Pascal `swap` com nomes não locais e parâmetros.

Vários métodos comuns para associar os parâmetros atuais e formais são discutidos nesta seção. São: chamada por valor, referência, cópia-e-restauração, nome e por expansão de macro.* É importante conhecer o método de transmissão de parâmetros que uma linguagem (ou compilador) usa, porque o resultado de um programa pode depender do método usado.

Por que tantos métodos? Os diferentes métodos emergem a partir de diferentes interpretações do que uma expressão representa. Num enunciado como

```
a[i] := a[j]
```

a expressão `a[j]` representa um valor enquanto que `a[i]` representa uma localização de memória na qual o valor de `a[j]` é colocado. A decisão de se usar a localização ou o valor representado por uma expresão é determinada dependendo da mesma figurar ao lado esquerdo ou direito, respectivamente, do símbolo de atribuição. Como especificado no Capítulo 2, o termo valor-*l* se refere à memória representada por uma expressão e valor-*r* ao valor contido na memória. Os prefixos *l* e *r* simbolizam esquerda *(left)* e direita *(right)* de uma atribuição.

As diferenças entre os métodos de passagem de parâmetros estão baseadas primariamente no que o parâmetro atual representa, se um valor-*r*, um valor-*l* ou o texto do próprio parâmetro atual.

Chamada por Valor

Este é, num certo sentido, o método mais simples possível de se transmitir parâmetros. Os parâmetros atuais são avaliados e seus valores-*r* são passados para o procedimento chamado. A chamada por valor é usada em C, e os parâmetros em Pascal são usualmente transmitidos dessa forma. Todos os programas deste capítulo repousam sobre esse método para a transmissão de parâmetros. A transmissão por valor pode ser implementada tal como segue.

1. Um parâmetro formal é tratado exatamente como um nome local, de forma que a memória para os parâmetros formais está no registro de ativação do procedimento chamado.
2. O chamador avalia os parâmetros atuais e coloca seus valores-*r* na memória para os parâmetros formais.

Uma figuração distintiva da transmissão por valor está em que os parâmetros formais não afetam os valores no registro de ativação do chamador. Se a palavra-chave `var` à linha 3 da Fig. 7.29 for omitida, Pascal irá transmitir `x` e `y` por valor para o procedimento `swap`. A chamada `swap(a,b)` à linha 12 deixa os valores de `a` e `b` intocados. Sob a transmissão por valor, o efeito da chamada `swap(a,b)` é equivalente à seqüência de passos

```
   x := a
   y := b
temp := x
   x := y
   y := temp
```

onde `x`, `y` e `temp` são locais a `swap`. Apesar dessas atribuições mudarem os valores dos locais `x`, `y` e `temp`, as mudanças estarão perdidas quando o controle retornar da chamada e o registro de ativação de `swap` for liberado. A chamada, por conseguinte, não possui efeito no registro de ativação do chamador.

```
(1)  program reference (input, output);
(2)  var a, b: integer;
(3)  procedure swap (var x, y: integer);
(4)     var temp : integer;
(5)     begin
(6)        temp := x;
(7)           x := y;
(8)           y := temp
(9)     end;

(10) begin
(11)    a := 1; b := 2;
(12)    swap (a,b);
(13)    writeln(' a =', a); writeln('b =', b)
(14) end.
```

Fig. 7.29. Um programa Pascal com o procedimento `swap`.

Um procedimento chamado por valor somente pode afetar seu chamador através de nomes não locais (ver `swap` na Fig. 7.28) ou através de apontadores que sejam explicitamente passados como valores. No programa C da Fig. 7.30, `x` e `y` são declarados à linha 2 como apontadores para inteiros; o operador `&` na chamada `swap(&a,&b)` à linha 8 resulta nos apontadores para `a` e `b` sendo passados para `swap`. A saída desse programa é

```
a é agora 2, b é agora 1
```

O uso dos apontadores neste exemplo sugere como o compilador usando uma chamada por referência trocaria os valores.

```
(1)  swap (x, y)
(2)  int *x, *y;
(3)  {     int temp;
(4)        temp = *x; *x = *y; *y = temp;
(5)  }
(6)  main( )
(7)  {     int a = 1, b = 2;
(8)        swap ( &a, &b );
(9)        printf ("a e agora %d, b e agora %d\n",a,b);
(10) }
```

Fig. 7.30. Programa C usando apontadores num procedimento chamado por valor.

*Dos originais em inglês: *call-by-value, call-by-reference, copy-restore, call-by-name* e *macro-expansion*. Também são utilizados na literatura em língua portuguesa os termos "transmissão por valor", "por referência" e "por valor-resultado" (de *value-result*) para os três primeiros termos. (N. do T.)

Chamada por Referência

Quando os parâmetros são transmitidos por *referência* (também conhecida *chamada por endereço* ou *chamada por localização*), o chamador passa para o procedimento chamado um apontador para o endereço de armazenamento de cada parâmetro atual.

1. Se um parâmetro atual for um nome ou uma expressão tendo um valor-*l*, então aquele valor-*l* é passado.
2. Se, no entanto, o parâmetro atual for uma expressão, como `a+b` ou `2`, que não têm valor-*l*, a expressão é avaliada numa nova localização e o endereço daquela localização é passado.

Uma referência a um parâmetro formal no procedimento chamado se torna, no código-alvo, uma referência indireta através do apontador transmitido para o procedimento chamado.

Exemplo 7.7. Consideremos o procedimento `swap` da Fig. 7.29. Uma chamada para `swap` com parâmetros atuais `i` e `a[i]`, isto é, `swap(i,a[i])` teria o mesmo efeito que a seguinte seqüência de passos:

1. Copiar o endereço (valores-*l*) de `i` e `a[i]` no registro de ativação do procedimento chamado, digamos, nas localizações `arg1` e `arg2`, correspondentes a `x` e `y`, respectivamente.
2. Estabelecer `temp` com o conteúdo da localização apontada por `arg1` (isto é, fazer `temp` igual a I_0, onde I_0 é o valor inicial de `i`). Esse passo corresponde a `temp := x` à linha 6 de `swap`.
3. Estabelecer o conteúdo da localização apontada por `arg1` com o valor da localização apontada por `arg2`; isto é, `i := a[`I_0`]`. Esse passo corresponde a `x := y` à linha 7 de `swap`.
4. Estabelecer o conteúdo da localização apontada por `arg2` com um valor igual ao `temp`; isto é, fazer `a[`I_0`] := i`. Esse passo corresponde a `y := temp`. □

A chamada por referência é usada numa série de linguagens; os parâmetros `var` em Pascal são passados dessa forma. Os *arrays* são usualmente transmitidos por referência.

Cópia-e-Restauração

Um sistema híbrido entre a chamada por valor e a por referência é a *ligação da cópia-e-restauração*, (também conhecida como *cópia-para-dentro-cópia-para-fora* ou *valor-resultado*).

1. Antes do controle fluir para o procedimento chamado, os parâmetros atuais são avaliados. Os valores-*r* dos parâmetros atuais são passados para o procedimento chamado, como na transmissão por valor. Adicionalmente, entretanto, os valores-*l* daqueles parâmetros atuais que tenham valores-*l* são determinados antes da chamada.
2. Quando o controle retorna, os valores-*r* correntes dos parâmetros formais são copiados de volta dentro dos valores-*l* dos parâmetros atuais, usando os valores-*l* computados antes da chamada. Somente os parâmetros atuais que tenham valores-*l* são copiados de volta, naturalmente.

O primeiro passo "copia para dentro" do registro de ativação do procedimento chamado os valores dos parâmetros atuais (na memória para os parâmetros formais). O segundo passo "copia para fora", no registro de ativação do procedimento chamador, os valores finais dos parâmetros formais (dentro dos valores-*l* computados a partir dos parâmetros atuais antes da chamada).

Note-se que `swap(i, a[i])` trabalha corretamente usando a cópia-e-restauração, uma vez que a localização de `a[i]` é computada e preservada pelo programa chamador antes da chamada. Por conseguinte, o valor final do parâmetro formal `y`, que será o valor inicial de

```
(1)  program copyout (input, output);
(2)    var a : integer;
(3)    procedure unsafe (var x : integer);
(4)      begin x := 2; a := 0 end;
(5)    begin
(6)      a := 1; unsafe(a); writeln(a)
(7)    end.
```

Fig. 7.31. A saída muda se a chamada por referência for mudada para cópia-e-restauração.

`i`, é copiado na localização correta, ainda que a localização de `a[i]` seja modificada pela chamada (porque o valor de `i` muda).

A cópia-e-restauração é usada em algumas implementações Fortran. No entanto, outras implementações usam a chamada por referência. As diferenças entre as duas podem emergir se o procedimento chamado possuir mais de uma forma de acesso a uma localização do registro de ativação do procedimento chamador. A ativação estabelecida pela chamada `unsafe(a)` à linha 6 da Fig. 7.31 pode ter acesso a `a` como um nome não local e através do parâmetro formal `x`. Sob a chamada por referência, as atribuições a `x` e a `a` afetam imediatamente a `a`, de forma que o valor final de `a` é `0`. Sob a cópia-e-restauração, entretanto, o valor `1` do parâmetro atual `a` é copiado no parâmetro formal `x`. O valor final `2` de `x` é copiado dentro do valor-*l* de `a` exatamente antes do controle retornar, de forma que o valor final de `a` é `2`.

Chamada por Nome

A chamada por nome é definida tradicionalmente pela *regra da cópia* de Algol, a qual é:

1. O procedimento é tratado como se fosse uma macro; isto é, seu corpo é substituído pela chamada feita no chamador, com os parâmetros atuais substituindo literalmente os parâmetros formais. Tal substituição literal é chamada de *expansão de macro* ou *expansão em linha*.
2. Os nomes locais no procedimento chamado são mantidos distintos dos nomes do procedimento chamador. Podemos pensar em cada nome local no procedimento chamado como sendo sistematicamente renomeado para um nome distinto antes da expansão da macro ser feita.
3. Os parâmetros atuais são envolvidos por parênteses, se necessário, para preservar suas integridades.

Exemplo 7.8. A chamada `swap(i,a[i])` proveniente do Exemplo 7.7 seria implementada como se fosse

```
temp := i
   i := a[i]
a[i] := temp
```

Por conseguinte, sob a chamada por nome, `swap` estabelece `i` com o valor de `a[i]`, mas possui o inesperado resultado de estabelecer `a[a[`I_0`]]` — ao invés de `a[`I_0`]` — em I_0, onde I_0 é o valor inicial de `i`. Esse fenômeno ocorre porque a localização de `x` na atribuição `x := temp` de `swap` não é avaliada até que seja necessitada, tempo em que o valor de `i` já mudou. Uma versão operacional de `swap` aparentemente não pode ser escrita se a chamada por nome for usada (ver Fleck [1976]). □

Apesar da chamada por nome ser primariamente de interesse teórico, a técnica conceitualmente relacionada da expansão em linha tem sido sugerida para reduzir o tempo de execução de um programa. Existe um certo custo associado ao estabelecimento de um procedimento — espaço é reservado para o registro de ativação, o estado da máquina é salvo, os elos são estabelecidos e, em seguida, o controle é

transferido. Quando o corpo do procedimento é pequeno, o código devotado às seqüências de chamada pode superar o código no corpo do procedimento. Pode ser, conseqüentemente, mais eficiente usar a expansão em linha do corpo do procedimento no código do chamador, mesmo que o tamanho do programa cresça um pouco. No próximo exemplo, a expansão em linha é aplicada a um procedimento chamado por valor.

Exemplo 7.9. Suponhamos que a função `f` na atribuição

```
x := f(A) + f(B)
```

seja chamada por valor. Aqui, os parâmetros atuais A e B são expressões. A substituição de cada ocorrência de parâmetro formal no corpo de `f` pelas expressões A e B leva a uma chamada por nome; relembremos `a[i]` no último exemplo.

Variáveis temporárias novas podem ser usadas para forçar a avaliação dos parâmetros atuais antes da execução do corpo do procedimento:

```
t1 := A;
t2 := B;
t3 := f(t1);
t4 := f(t2);
x := t3 + t4;
```

Agora a expansão em linha irá substituir todas as ocorrências do parâmetro formal t_1 e t_2 quando a primeira e a segunda chamadas forem expandidas.[7] □

A implementação usual da chamada por nome é passar para o procedimento chamado sub-rotinas sem parâmetros, comumente chamadas de *avaliadoras*,* que podem avaliar o valor-*l* e o valor-*r* do parâmetro atual. Como qualquer procedimento passado como parâmetro numa linguagem usando o escopo léxico, um avaliador carrega um elo de acesso consigo, apontando para o registro de ativação corrente do procedimento chamador.

7.6 TABELAS DE SÍMBOLOS

Um compilador usa uma tabela de símbolos para controlar as informações de escopo e das amarrações a respeito dos nomes. A tabela de símbolos é pesquisada a cada vez que um nome é encontrado no texto-fonte. As mudanças na tabela ocorrem se um novo nome ou uma nova informação a respeito de um nome já existente for descoberta.

Um mecanismo de tabela de símbolos precisa permitir que adicionemos novas entradas e encontremos eficientemente as já existentes. Os dois mecanismos de tabelas de símbolos apresentados nesta seção são as listas lineares e as tabelas *hash*. Avaliamos cada esquema na base do tempo requerido para adicionar n entradas e realizar e inquisições. Uma lista linear é o mais simples de implementar, mas seu desempenho é pobre quando e e n se tornam grandes. Os esquemas de *hashing* providenciam um melhor desempenho para um esforço de programação um tanto maior e uma sobrecarga de espaço também maior. Ambos os mecanismos podem ser adaptados prontamente para tratar a regra de escopo do aninhamento mais interno.

É útil para um compilador ser capaz de crescer a tabela de símbolos dinamicamente, se necessário, em tempo de compilação. Se o tamanho da tabela de símbolos for fixado quando o compilador for escrito, o mesmo deve ser escolhido grande o suficiente para tratar qualquer programa-fonte que lhe seja apresentado. Tal tamanho fixo está inclinado a ser grande para a maioria dos programas e, por outro lado, inadequado para alguns.

Entradas da Tabela de Símbolos

Cada entrada da tabela de símbolos é destinada à declaração de um nome. O formato das entradas não tem que ser uniforme, porque a informação guardada a respeito de um nome depende do uso do mesmo. Cada entrada pode ser implementada como um registro consistindo em uma seqüência de palavras consecutivas na memória. Para manter a tabela de símbolos uniforme, pode ser conveniente que algumas das informações a respeito de um nome sejam mantidas fora da entrada da tabela, com somente um apontador para cada uma dessas informações armazenado no registro.

As informações são introduzidas na tabela de símbolos em vários momentos. As palavras-chave são introduzidas inicialmente na tabela, se o forem de todo. O analisador léxico da Seção 3.4 procura seqüências de letras e dígitos na tabela de símbolos para determinar se uma palavra-chave reservada ou um nome foi coletado. Com essa abordagem, as palavras-chave precisam estar na tabela de símbolos antes da análise léxica começar. Alternativamente, se o analisador léxico intercepta as palavras-chave reservadas, as mesmas não precisam figurar na tabela de símbolos. Se a linguagem não reserva palavras-chave, é essencial que as mesmas sejam introduzidas na tabela de símbolos com um aviso a respeito de seus possíveis usos como palavras-chave.

A entrada da tabela de símbolos em si pode ser estabelecida quando o papel de um nome for esclarecido, com o preenchimento dos valores dos atributos à medida que as informações se tornarem disponíveis. Em alguns casos, a entrada pode ser iniciada a partir do analisador léxico tão logo um nome seja visto à entrada. Mais freqüentemente, um nome pode denotar vários objetos diferentes, talvez no mesmo bloco ou procedimento. Por exemplo, as declarações C

```
int x;
struct x { float y, z; };        (7.1)
```

usam `x` tanto como um inteiro quanto como o rótulo de uma estrutura com dois campos. Em tais casos, o analisador léxico pode somente retornar ao analisador sintático o nome (ou um apontador para o lexema que forma o nome), ao invés de um apontador para a entrada da tabela de símbolos. O registro na tabela de símbolos será criado quando o papel sintático desempenhado por esse nome for descoberto. Para as declarações em (7.1), seriam criadas duas entradas para `x` na tabela de símbolos: uma tendo `x` como um inteiro e outra como uma estrutura.

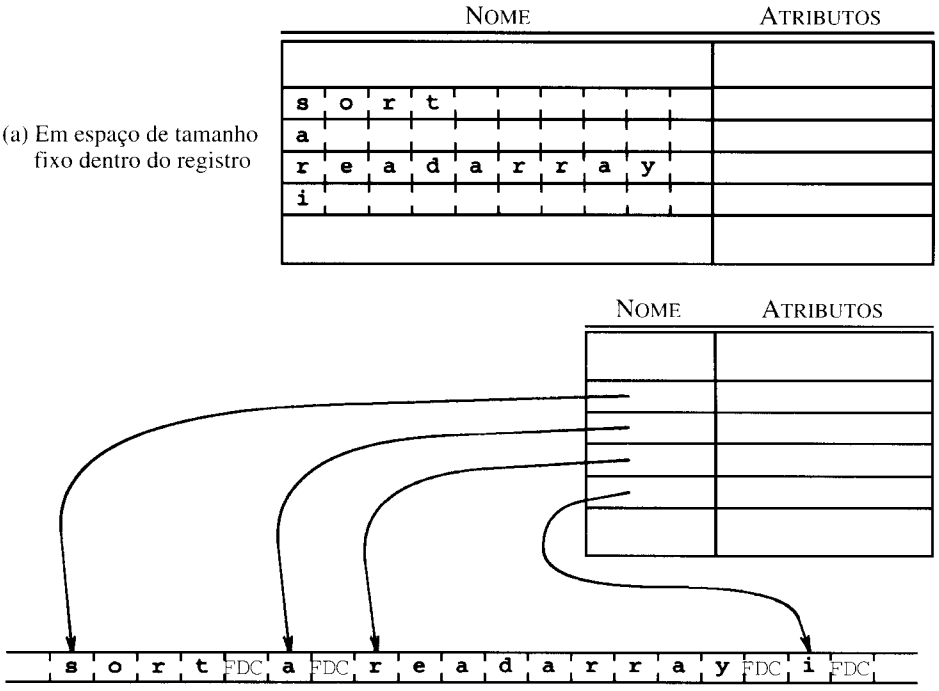
Os atributos de um nome são introduzidos em resposta às declarações, que podem ser implícitas. Os rótulos são freqüentemente identificadores seguidos por um ponto-e-vírgula e dessa forma uma ação associada ao reconhecimento de um identificador pode introduzir esse fato na tabela de símbolos. Similarmente, a sintaxe das declarações dos procedimentos especifica que certos identificadores são parâmetros formais.

Caracteres num Nome

Como no Capítulo 3, existe uma distinção entre o *token* **id** para um identificador ou nome, o lexema que consiste na cadeia de caracteres para o nome e os atributos do nome. As cadeias de caracteres podem ser inapropriadas para se trabalhar e dessa forma os compiladores freqüentemente usam alguma representação de tamanho fixo para o nome ao invés do lexema. O lexema é necessitado quando uma entrada para a tabela de símbolos é estabelecida pela primeira vez e quando procuramos por um lexema encontrado à entrada, a fim de se determinar se é um nome que já apareceu antes. Uma representação comum para

[7] Existem custos associados às variáveis temporárias. As mesmas podem provocar a alocação de espaço extra num registro de ativação. Se os objetos de dados locais no registro de ativação forem inicializados, os temporários resultam igualmente em desperdício de tempo.

*Do original em inglês: *thunks*. (N. do T.)



Nota: FDC simboliza um caractere de fim de cadeia. (b) Num *array* separado

Fig. 7.32. Armazenando os caracteres de um nome.

um nome é um apontador para a entrada da tabela de símbolos para o mesmo.

Se existir um limite superior modesto no comprimento dos nomes, os caracteres de cada nome podem ser armazenados na entrada da tabela de símbolos, como na Fig. 7.32(a). Se não existe limite no tamanho de um nome, ou se o limite é raramente atingido, o esquema indireto da Fig. 7.32(b) pode ser usado. Ao invés de se alocar em cada entrada da tabela de símbolos a quantidade máxima de espaço para abrigar um lexema, podemos utilizar o espaço mais eficientemente se existir espaço somente para o apontador nas entradas da tabela de símbolos. Num registro para um nome, colocamos um apontador para um *array* de caracteres separado (a *tabela de cadeias de caracteres*) dando a posição do primeiro caractere do lexema. O esquema indireto da Fig. 7.32(b) permite que o tamanho do campo nome, da entrada da tabela de símbolos, permaneça uma constante.

O esquema completo para se constituir um nome precisa ser armazenado, a fim de assegurar que todos os usos do mesmo nome possam ser associados ao mesmo registro da tabela de símbolos. Precisamos, entretanto, distinguir entre as ocorrências de um mesmo lexema que estejam em escopos de declarações diferentes.

Informações de Alocação de Memória

As informações a respeito das localizações de memória que serão amarradas aos nomes em tempo de execução são mantidas na tabela de símbolos. Consideremos primeiro os nomes com memória estática. Se o código-alvo estiver em linguagem de montagem, podemos fazer com que o montador cuide das localizações de memória para os vários nomes. Tudo o que temos de fazer é esquadrinhar a tabela de símbolos, após gerar o código de montagem do programa, e adicionar as definições de dados para cada nome no programa em linguagem de montagem.

Se, entretanto, o compilador deve gerar código de máquina, a posição de cada objeto de dados, relativa a uma origem fixa, tal como o início de um registro de ativação, precisa ser acertada. O mesmo comentário se aplica a um bloco de dados carregado como um módulo separado do programa. Por exemplo, os blocos `COMMON` em Fortran são carregados separadamente e as posições dos nomes, relativas ao início do bloco no qual residem, precisam ser determinadas. Pelas razões discutidas na Seção 7.9, o enfoque da Seção 7.3 precisa ser modificado em Fortran, na medida em que precisamos atribuir deslocamentos para os nomes depois que todas as declarações para um procedimento tenham sido examinadas e os enunciados `EQUIVALENCE` tenham sido processados.

No caso de nomes cuja memória seja alocada numa pilha ou *heap*, o compilador não reserva qualquer memória, apenas delineia o registro de ativação para cada procedimento, como na Seção 7.3.

A Estrutura de Lista para a Tabela de Símbolos

A estrutura mais simples e fácil de se implementar uma tabela de símbolos é uma lista linear de registros, mostrada na Fig. 7.33. Usamos um *array* singelo ou, de modo equivalente, vários *arrays*, para armazenar os nomes e suas informações associadas. Os novos nomes são adicionados à lista na ordem em que são encontrados. A posição do final do *array* é marcada pelo apontador *próxima-entrada-disponível*, que aponta para onde a próxima entrada da tabela de símbolos será colocada. A pesquisa de um nome se dá do final do *array* para o início. Quando o nome é localizado, as informações associadas podem ser encontradas nas próximas palavras adjacentes. Se atingimos o início do *array* sem ter encontrado o nome, um erro fatal ocorre — um nome esperado não está na tabela de símbolos.

Note-se que a constituição de uma entrada para um nome e a procura do nome da tabela de símbolos são operações independentes — podemos desejar realizar uma sem termos que realizar a outra. Numa linguagem estruturada em blocos a ocorrência de um nome está no escopo da declaração mais proximamente aninhada ao mesmo. Podemos implementar essa regra de escopo usando a estrutura de lista de dados através da constituição de uma nova entrada para um nome a cada vez que o mesmo for declarado. Uma nova entrada é feita nas palavras imediatamente seguintes ao apontador *próxima_entrada_disponível;* esse apontador é incrementado pelo tamanho do registro da tabela de símbolos. Uma vez que essas entradas são inseridas em ordem, começando

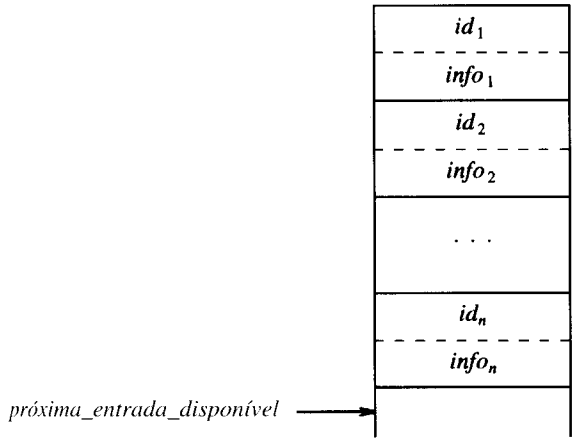


Fig. 7.33. Uma lista linear de registros.

pelo início do *array*, figuram na ordem que são criadas. Pela pesquisa a partir de *próxima_entrada_disponível*, em direção ao início do *array*, estaremos seguros de encontrar a entrada mais recentemente criada.

Se a tabela de símbolos contém n nomes, o trabalho necessário para inserir um novo nome será constante se realizarmos a inserção sem verificarmos se o nome já não está na tabela. Se não são permitidas múltiplas entradas para o mesmo nome, necessitamos, então, procurar ao longo de toda a tabela antes de descobrir se um nome já não está lá, realizando, no processo, um trabalho proporcional a n. Para encontrar os dados a respeito de um nome, pesquisamos, em média, $n/2$ nomes, de forma que o custo de uma inquisição é também proporcional a n. Por conseguinte, uma vez que as inserções e as inquisições tomam um tempo proporcional a n, o trabalho total para se inserir n nomes e realizar e inquisições é no máximo $cn\,(n + e)$, onde c é uma constante que representa o tempo necessário para se realizar umas poucas operações de máquina. Num programa de porte médio, poderíamos ter $n = 100$ e $e = 1000$, de forma que várias centenas de milhares de operações de máquina seriam utilizadas na manutenção da tabela. Isto pode não ser doloroso, uma vez que estamos falando de menos de um segundo de tempo. No entanto, se n e e forem multiplicados por 10, o custo é multiplicado por 100 e a manutenção da tabela se torna proibitiva. O estabelecimento de um perfil de comportamento produz dados valiosos sobre onde o compilador gasta tempo e pode ser usado para decidir se tempo demais está sendo gasto pesquisando-se através de listas lineares.

Tabelas *Hash*

Variações na técnica de pesquisa, conhecidas como *hashing*, têm sido implementadas em muitos compiladores. Aqui consideramos uma variante um tanto simples, conhecida como *hashing aberto*, onde o termo "aberto" se refere à propriedade de que não precisa haver limite no número de entradas que podem ser feitas numa tabela. Esse esquema nos dá ainda a capacidade de realizar e entradas sobre n nomes num tempo proporcional a $n\,(n + e)/m$, para qualquer constante m de nossa escolha. Uma vez que m pode ser feita tão grande quanto desejemos, até o limite de n, esse método é geralmente mais eficiente do que as listas lineares e é o método de escolha para as tabelas de símbolos na maioria das situações. Como poderia ser esperado, o espaço ocupado pela estrutura de dados cresce com m, e dessa forma uma barganha espaço-tempo está envolvida.

O esquema básico de *hashing* é ilustrado na Fig. 7.34. Existem duas partes para a estrutura de dados:

1. Uma *tabela hash*, consistindo em um *array* fixo de m apontadores para entradas da tabela.
2. As entradas da tabela são organizadas em m listas ligadas separadas, chamadas de *buckets* (algumas das quais podem ficar vazias).

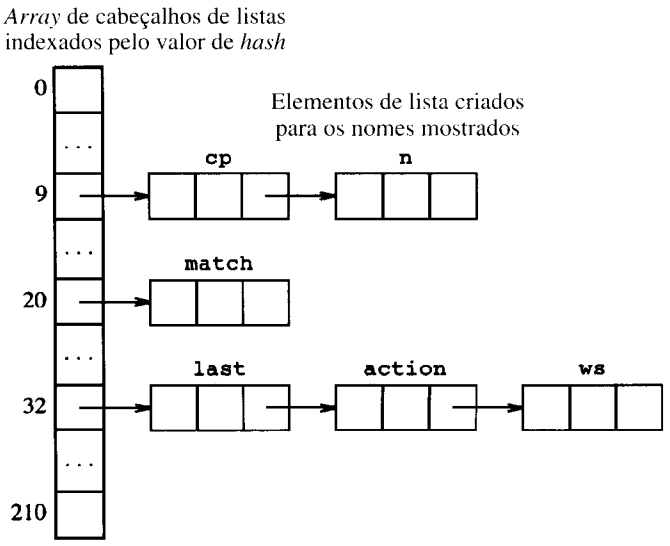


Fig. 7.34. Uma tabela *hash* de tamanho 211.

Cada registro na tabela de símbolos figura exatamente em uma dessas listas. O armazenamento para os registros pode ser delineado a partir de um *array* de registros, como discutido na próxima seção. Alternativamente, as facilidades de alocação dinâmica de memória da linguagem de implementação podem ser usadas para se obter espaço para os registros, freqüentemente às expensas de alguma perda de eficiência.

Para se determinar se existe uma entrada para uma cadeia de caracteres s na tabela de símbolos, aplicamos uma *função de hash* h a s, de tal forma que $h(s)$ retorne um inteiro entre 0 e $m-1$. Se s estiver na tabela de símbolos, estará na lista numerada por $h(s)$. Se s ainda não estiver na tabela de símbolos, é introduzida através da criação de um registro para a mesma, que é ligado ao início da lista numerada por $h(s)$.

Grosso modo, a lista média tem um comprimento de n/m registros, se existirem n nomes numa tabela de comprimento m. Pela escolha de m de tal forma que n/m esteja limitado a uma pequena constante, 2 por exemplo, o tempo para se ter acesso a uma entrada da tabela é essencialmente constante.

O espaço ocupado pela tabela de símbolos consiste em m palavras para a tabela *hash* e cn palavras para as entradas da tabela, onde c é o número de palavras por entrada da tabela. Por conseguinte, o espaço para a tabela *hash* depende somente de m e o espaço para as entradas da tabela depende somente do número de entradas.

A escolha do valor m depende da aplicação pretendida para a tabela de símbolos. Escolhê-lo como sendo de umas poucas centenas deve fazer com que a pesquisa em tabela seja uma fração desprezível do tempo total despendido pelo compilador, mesmo para programas de tamanho moderado. Quando a entrada para o compilador puder ser gerada por um outro programa, entretanto, o número de nomes pode exceder em muito o da maioria dos programas de mesmo tamanho que sejam gerados por pessoas e, nesse caso, tamanhos maiores de tabelas de símbolos são preferíveis.

Uma boa atenção tem sido dada à questão de como projetar uma função de *hash* que seja fácil de computar para cadeias de caracteres, além de distribuí-las uniformemente dentre as m listas.

Uma abordagem adequada ao cômputo das funções de *hash* está em se proceder da seguinte forma:

1. Determinar um inteiro positivo h a partir dos caracteres $c_1, c_2, \ldots, c_k$ na cadeia s. A conversão de caracteres singelos para inteiros é usualmente suportada pela linguagem de implementação. Pascal providencia a função *ord* com esse propósito; C converte automaticamente um caráter para um inteiro, se uma operação aritmética for realizada sobre o mesmo.

2. Converter o inteiro h determinado acima no número de uma lista, isto é, um inteiro entre 0 e $m - 1$. Simplesmente dividir por m e ficar com o resto é uma política razoável. Ficar com resto parece funcionar melhor se m for primo e, por conseguinte, a escolha 211 ao invés de 200 na Fig. 7.34.

As funções de *hash* que examinam todos os caracteres numa cadeia são menos facilmente iludidas do que, digamos, as funções que examinam apenas alguns caracteres nas extremidades ou no meio da cadeia. Relembremos que a entrada para o compilador pode ter sido criada por um programa e pode ter, conseqüentemente, uma forma estilizada, escolhida para evitar conflitos com os nomes que uma pessoa ou algum outro programa poderiam utilizar. As pessoas tendem a "agrupar" os nomes igualmente, com escolhas como `ponteiro`, `pont`, `pontant` e assim por diante.

Uma técnica simples para computar h está em adicionar os valores inteiros dos caracteres na cadeia. Uma idéia melhor está em multiplicar o valor antigo de h por uma constante α antes de adicionar o próximo caractere. Isto é, fazendo-se $h_0 = 0$, $h_i = \alpha h_{i-1} + c_i$, para $1 \leq i \leq k$ e seja $h = h_k$, onde k é o tamanho da cadeia (relembremos que o valor de *hash* que dá o número de listas é h **mod** m). A simples adição dos caracteres representa o caso em que $\alpha = 1$. Uma estratégia similar consiste em realizar o ou-exclusivo de c_i com αh_{i-1}, ao invés de adicionar.

Para os inteiros de 32 *bits*, se fizermos $\alpha = 65599$, um número primo próximo a 2^{16}, o estouro de capacidade ocorrerá cedo durante o cômputo de αh_{i-1}. Como α é primo, ignorar o estouro e manter somente os 32 *bits* de mais baixa ordem parece funcionar bem.

Em um conjunto de experimentos, a função de *hash hashpjw* na Fig. 7.35 do compilador C de P. J. Weinberger funcionou consistentemente bem em todos os tamanhos de tabelas testados (ver a Fig. 7.36). Os tamanhos incluíam os primeiros números primos maiores do que 100, 200, ..., 1500. Um segundo lugar bastante próximo foi o da função que computou h através da multiplicação do valor antigo de h por 65599, ignorando os estouros de capacidade e adicionando o próximo caractere. A função *hashpjw* é computada começando com $h = 0$. Para cada caractere c, deslocam-se os *bits* de h 4 posições à esquerda e adiciona-se c. Se qualquer um dos 4 *bits* de mais alta ordem de h for 1, deslocam-se os quatro *bits* em 24 posições à direita, faz-se o ou exclusivo dos mesmos com h e zera-se qualquer *bit* de mais alta ordem que seja 1.

```
(1)  #define PRIME 211
(2)  #define EOS '\0'
(3)  int hashpjw(s)
(4)  char    *s;
(5)  {
(6)     char   *p;
(7)     unsigned h = 0, g:
(8)     for ( p = s; *p != EOS; p = p+1 ) {
(9)        h = (h << 4) + (*p);
(10)       if (g = h&0xf0000000) {
(11)          h = h ^ (g >> 24);
(12)          h = h ^ g;
(13)       }
(14)    }
(15)    return h % PRIME;
(16) }
```

Fig. 7.35. Função de *hash hashpjw*, escrita em C.

Exemplo 7.10. Para resultados ótimos, o tamanho da tabela *hash* e da entrada esperada precisam ser levados em conta quando a função de *hash* for projetada. Por exemplo, é desejável que os valores de *hash*, para os nomes que ocorrem mais freqüentemente numa linguagem, sejam distintos. Se as palavras-chave forem introduzidas na tabela de símbolos, estarão propensas a figurar entre os nomes que ocorrem mais freqüentemente, apesar de num dos exemplos de programa C o nome `iter` figurado com uma freqüência três vezes maior do que `while`.

Uma forma de se testar uma função de *hash* é examinar o número de cadeias que caem na mesma lista. Dado um arquivo F, consistindo em n cadeias, suponhamos que b_j cadeias caiam numa lista j, para $0 \leq j \leq m - 1$. Uma medida de quão uniformemente as cadeias estão distribuídas através das listas é obtida pelo cômputo

$$\sum_{j=0}^{m-1} b_j(b_j + 1)/2 \tag{7.2}$$

A justificação intuitiva para esse termo está em que precisamos examinar 1 elemento de lista para encontrar a primeira entrada na lista j, a 2 para encontrar o segundo e assim por diante, até b_j, para encontrar a última entrada. A soma de 1, 2, ..., b_j é $b_j(b_j + 1)/2$.

Do Exercício 7.14, o valor de (7.2) para uma função de *hash* que distribua cadeias aleatoriamente através dos *buckets* é dado por

$$(n / 2m)(n + 2m - 1) \tag{7.3}$$

A média de termos (7.2) e (.7.3) é plotada na Fig. 7.36 para várias funções de *hash* aplicadas a nove arquivos. Os arquivos são:

1. Os nomes e palavras-chave que ocorrem mais freqüentemente numa amostra de programas C.
2. Como em (1), mas com os 100 nomes e palavras-chave que ocorrem mais freqüentemente.
3. Como em (1), mas com os 500 nomes e palavras-chave que ocorrem mais freqüentemente.
4. 952 nomes externos do *kernel* do sistema operacional UNIX.
5. 627 nomes num programa C gerado por C++ (Stroustrup [1986]).
6. 915 cadeias de caracteres geradas randomicamente.
7. 614 palavras da Seção 3.1 deste livro, no original em língua inglesa
8. 1201 palavras em inglês com `xxx` adicionado como prefixo e sufixo.
9. Os 300 nomes `v100, v101, ..., v399`

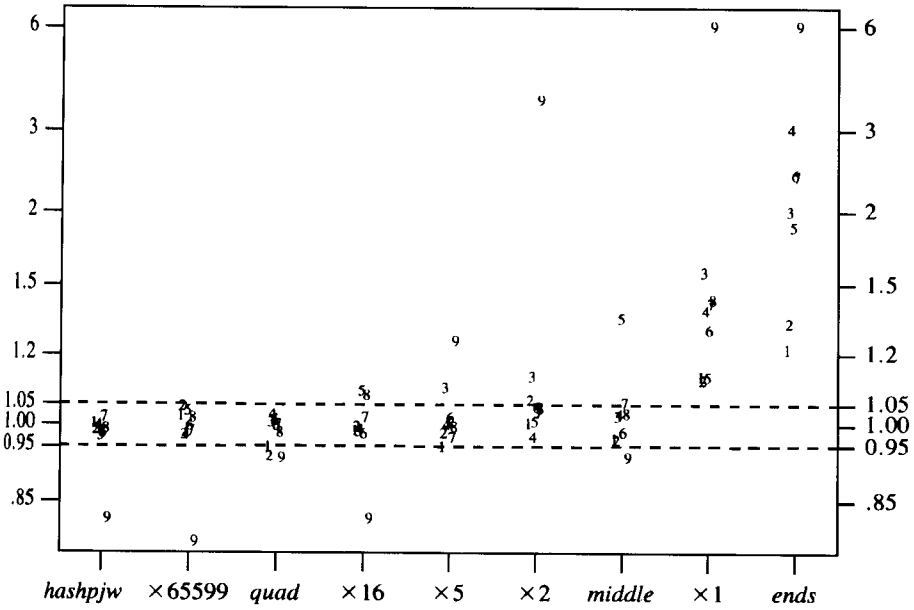
A função *hashpjw* é como na Fig. 7.35. As funções denominadas $x\alpha$, onde α é uma constante inteira, computam h **mod** m, onde h é obtido iterativamente começando com 0, multiplicando o valor antigo por α e adicionando-se o próximo caractere. A função *middle* (meio) forma h a partir dos quatro caracteres centrais da cadeia, enquanto que *ends* (extremidades) adiciona os três primeiros e os três últimos caracteres ao tamanho para formar h. Finalmente, *quad* (quadras) agrupa cada quatro caracteres consecutivos em um inteiro e os adiciona.

Representando a Informação de Escopo

As entradas da tabela de símbolos são para declarações de nomes. Quando uma ocorrência de um nome no texto-fonte é pesquisada na tabela de símbolos, a entrada para a declaração apropriada daquele nome deve ser retornada. As regras de escopo da linguagem-fonte determinam a declaração que é a apropriada.

Uma abordagem simplista está em manter uma tabela de símbolos separada para cada escopo. Com efeito, a tabela de símbolos para um procedimento ou escopo é o equivalente, em tempo de compilação, do registro de ativação. As informações para os nomes não locais de um procedimento são encontradas esquadrinhando-se as tabelas de símbolos para os procedimentos envolventes seguindo-se as regras de escopo da linguagem. De modo equivalente, as informações sobre os nomes locais para um procedimento podem ser atreladas ao nó para o mesmo numa árvore sintática para o programa. Com essa abordagem, a tabela de símbolos é integrada à representação intermediária da entrada.

As regras de escopo do aninhamento mais interno podem ser implementadas adaptando-se as estruturas de dados apresentadas anteriormente nesta seção. Controlamos os nomes locais de um procedimento fornecendo a cada procedimento um único número. Os blocos também precisam ser numerados, se a linguagem for estruturada em



MÉTODOS DE HASH
O número do arquivo é plotado.

Fig. 7.36. Desempenho relativo de funções de *hash* para uma tabela de tamanho 211.

blocos. O número de cada procedimento pode ser computado de uma forma dirigida pela sintaxe a partir das regras semânticas que reconhecem o início e o final de cada procedimento. O número de procedimento é tornado parte de todos os nomes locais declarados naquele procedimento; a representação do nome local na tabela de símbolos é um par consistindo no nome e no número de procedimento (em alguns esquemas, tais como aqueles descritos abaixo, o número de procedimento não precisa efetivamente aparecer, na medida em que pode ser deduzido da posição do registro na tabela de símbolos).

Quando procuramos por um nome recém-esquadrinhado, um reconhecimento ocorre somente quando os caracteres do nome se igualam aos caracteres do nome na entrada na tabela e o número associado à entrada da tabela de símbolos é o número do procedimento sendo processado. As regras de escopo do aninhamento mais interno podem ser implementadas em termos das seguintes operações sobre um nome:

pesquisar:	encontrar a entrada mais recentemente criada
inserir:	constituir uma nova entrada
remover:	remover a entrada mais recentemente criada

As entradas "removidas" precisam ser preservadas; são apenas removidas da tabela de símbolos ativa. Num compilador de uma passagem, as informações sobre um escopo consistindo, digamos, no corpo de um procedimento, não são necessitadas em tempo de compilação após o corpo do procedimento ter sido processado. Entretanto, podem ser necessitadas em tempo de execução, particularmente se um sistema de diagnóstico for implementado. Neste caso, as informações na tabela de símbolos precisam ser adicionadas ao código gerado para utilização pelo editor de ligações ou pelo sistema de diagnóstico em tempo de execução. Veja também o tratamento dos nomes de campos em registros nas Seções 8.2 e 8.3.

Cada uma das estruturas de dados descritas nesta seção — listas e tabelas *hash* — pode ser mantida de forma a dar suporte às operações acima.

Quando uma lista linear consistindo em um *array* de registros foi descrita anteriormente nesta seção, mencionamos como *procurar* poderia ser implementada através da inserção de entradas numa das extremidades da tabela, de tal forma que a ordem das entradas no *array* era a mesma que a ordem de inserção. Um esquadrinhamento, que comece pela extremidade em direção ao início do *array*, encontra a entrada mais recentemente criada para um nome. A situação é similar numa lista ligada, como mostrado na Fig. 7.37. O apontador *frente* aponta para a entrada mais recentemente criada na lista. A implementação de *inserir* torna o tempo constante porque uma entrada é colocada à frente da lista. A implementação de *procurar* é feita esquadrinhando-se a lista, começando pela entrada apontada por *frente*, e seguindo-se os elos, até que o nome desejado seja encontrado ou o fim da lista atingido. Na Fig. 7.37, a entrada para a, declarado no bloco B_2, aninhado no bloco B_0, aparece mais próxima da frente da lista do que a entrada para a declarado em B_0.

Para a operação *remover*, note-se que as entradas para as declarações no procedimento mais profundamente aninhado aparecem à frente da lista. Por conseguinte, não precisamos manter o número de procedimento junto a cada entrada — se controlarmos a primeira entrada para cada procedimento, todas as entradas até a primeira podem ser removidas na tabela de símbolos ativa, ao terminarmos o processamento do escopo do procedimento.

Uma tabela *hash* consiste em *m* listas às quais se tem acesso através de um *array*. Como um nome sempre tem o seu endereço de *hash* calculado para a mesma lista, as listas individuais são mantidas como na Fig. 7.37. Entretanto, para implementar a operação de *remover* não teríamos que percorrer toda a tabela *hash*, procurando por listas contendo as entradas a serem removidas. A abordagem seguinte pode ser usada. Suponhamos que cada entrada possua dois elos:

1. um elo de *hash* que encadeia a entrada com as demais que colidem num mesmo endereço de *hash*
2. um elo de escopo que encadeia todas as entradas que pertencem ao mesmo escopo.

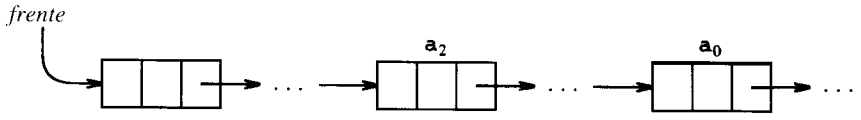


Fig. 7.37. A entrada mais recente para a está mais próxima à frente da lista.

Se o elo de escopo for mantido inalterado ao se remover uma entrada numa tabela de *hash*, a cadeia formada pelos elos de escopo constitui-se numa tabela de símbolos separada (inativa) para o escopo em questão.

A remoção de entradas da tabela de *hash* precisa ser realizada com cuidado, porque a remoção de uma entrada afeta a anterior na lista. Relembremos que removemos a *i*-ésima entrada fazendo com que a *i*-ésima − 1 entrada aponte para a *i*-ésima + 1. Usar simplesmente os elos de escopo para encontrar a *i*-ésima entrada não é o bastante. A *i*-ésima entrada − 1 pode ser encontrada se os elos de *hash* formarem uma lista circular ligada, na qual a última entrada aponta de volta para primeira. Alternativamente, poderíamos usar uma pilha para controlar as listas que contivessem entradas a serem removidas. Um marcador é colocado na pilha quando um novo procedimento é esquadrinhado. Acima do marcador estão os números das listas que contêm entradas para nomes declarados neste procedimento. Ao terminarmos de processar o procedimento, os números de lista podem ser removidos da pilha até que o marcador para o procedimento seja encontrado. Um outro esquema é discutido no Exercício 7.11.

7.7 FACILIDADES DE LINGUAGEM PARA A ALOCAÇÃO DINÂMICA DE MEMÓRIA

Nesta seção, descrevemos brevemente as facilidades providenciadas por algumas linguagens para a alocação dinâmica de memória para dados, sob controle do programa. O armazenamento para tais dados é usualmente obtido a partir do *heap* . Os dados alocados são freqüentemente retidos até que sejam explicitamente liberados. A alocação em si pode ser *explícita* ou *implícita*. Em Pascal, por exemplo, a alocação explícita é realizada usando o procedimento padrão `new`. A execução de `new(p)` reserva memória para o tipo de objeto apontado por `p` e o próprio `p` é deixado apontando para o objeto recém-alocado. A liberação é feita chamando-se o procedimento `dispose` na maioria das implementações de Pascal.

A alocação implícita ocorre quando a avaliação de uma expressão resulta na obtenção de memória para abrigar o valor de uma expressão. Lisp, por exemplo, aloca uma célula numa lista quando `cons` é usada; as células que não podem ser mais atingidas são automaticamente reclamadas de volta. Snobol permite que o comprimento de uma cadeia varie em tempo de execução e gerencia o espaço necessitado para abrigar uma cadeia numa área de armazenamento do tipo `heap`.

Exemplo 7.11. O programa Pascal na Fig. 7.38 constrói a lista ligada mostrada na Fig. 7.39 e imprime os inteiros armazenados nas células.

```
( 1 ) program tabela (input, output);
( 2 ) type link = ↑ celula;
( 3 )       celula = record
( 4 )                    chave, info : integer;
( 5 )                    proximo : link
( 6 )                    end;
( 7 ) var cabeca : link;
( 8 ) procedure inserir (k, i : integer);
( 9 )       var p : link;
(10 )       begin
(11 )             new (p); p↑.chave := k; p↑.info := i;
(12 )             p↑.proximo := cabeca; cabeca := p
(13 )       end;
(14 ) begin
(15 )       cabeca := nil;
(16 )       inserir (7,1); inserir (4,2); inserir (76,3);
(17 )       writeln (cabeca↑.chave, cabeca↑. info);
(18 )       writeln (cabeca↑.proximo↑.chave,cabeca↑.
                  proximo↑.info);
(19)        writeln (cabeca↑.proximo↑.proximo↑.chave,
                  cabeca↑.proximo↑. proximo↑ .info);
(20) end.
```

Fig. 7.38. Alocação dinâmica de células usando `new` em Pascal.

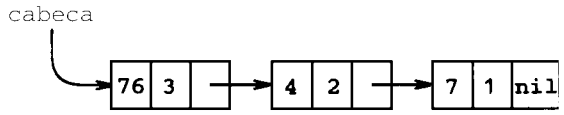


Fig. 7.39. Listas ligadas construídas pelo programa da Fig. 7.38.

Sua saída é

```
76  3
 4  2
 7  1
```

Quando a execução do programa começa à linha 15, o armazenamento do apontador `cabeca` está no registro de ativação para todo o programa. A cada vez que o controle atinge

```
(11)   new(p); p↑·chave := k; p↑.info := i;
```

a chamada `new(p)` resulta numa célula sendo alocada em algum local do *heap;* `p↑` se refere a essa célula na atribuição à linha 11.

Note-se, da saída do programa, que quando o controle retorna para o programa principal, proveniente de `inserir`, as células alocadas estão acessíveis. Em outras palavras, as células alocadas, utilizando-se `new`, durante uma ativação de `inserir`, são retidas quando o controle retorna para o programa principal, proveniente da ativação. □

Lixo

A memória alocada dinamicamente pode se tornar inatingível. A memória que um programa reserva, mas não se refere, é chamada de *lixo.** Na Fig. 7.38, suponhamos que `nil` fosse atribuído a `cabeca↑·proximo`, entre as linhas 16 e 17:

```
(16)   inserir(7,1);inserir(4,2);inserir(76,3);
       cabeca↑.proximo :=nil;
(17)   writeln(cabeca↑.chave, cabeca↑.info);
```

A célula mais à esquerda na Fig. 7.39 irá agora conter um valor de apontador `nil` ao invés de um apontador para a célula do meio. Quando o apontador para a célula do meio está perdido, as células do meio e a mais à direita se tornam lixo.

Lisp realiza a *coleta de lixo,* um processo discutido na próxima seção, que reclama a memória inacessível. Pascal e C não têm coleta de lixo, deixando para o programa liberar explicitamente a memória que já não é mais desejada. Nessas linguagens, a memória liberada pode ser reusada, mas o lixo permanece até o programa terminar.

Referências Ocas

Uma complicação adicional pode emergir com a liberação explícita; as referências ocas podem ocorrer. Como mencionado na Seção 7.3, uma referência oca ocorre quando se referencia uma memória que já foi liberada. Por exemplo, consideremos o feito de se executar `dispose(cabeca↑.proximo)` entre as linhas 16 e 17, na Fig. 7.38:

```
(16)   inserir(7,1); inserir(4,2); inserir(76,3);
       dispose(cabeca↑.proximo);
(17)   writeln(cabeca↑.chave, cabeca↑.info);
```

A chamada a `dispose` libera a célula seguinte àquela apontada por `cabeca`, como mostrado na Fig. 7.40. Entretanto, `cabeca↑.pró-`

*O termo *refugo* é também utilizado. (N. do T.)

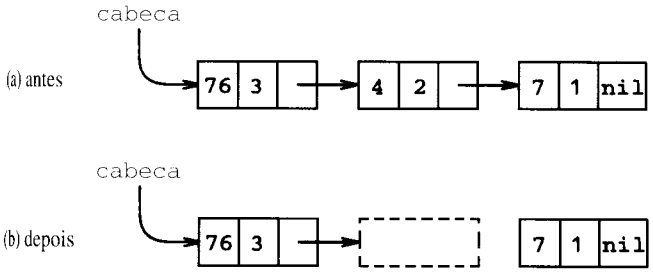


Fig. 7.40. Criação de referências ocas e de lixo.

ximo não foi modificado e, dessa forma, é um apontador oco para a memória liberada.

As referências ocas e o lixo são conceitos relacionados; as referências ocas ocorrem se a liberação ocorrer antes da última referência enquanto que o lixo existe se a última referência ocorrer antes da liberação.

7.8 TÉCNICAS DE ALOCAÇÃO DINÂMICA DE MEMÓRIA

As técnicas necessárias à implementação da alocação dinâmica de memória dependem de como a memória é liberada. Se a liberação for implícita, o pacote de suporte em tempo de execução é responsável pela determinação de quando um bloco não é mais necessitado. Haverá menos a fazer por parte do compilador se a liberação for feita explicitamente pelo programador. Consideramos a liberação explícita primeiro.

Alocação Explícita de Blocos de Tamanho Fixo*

A forma mais simples de alocação dinâmica envolve blocos de tamanho fixo. Através da ligação dos blocos numa lista, como na Fig. 7.41, a reserva e liberação podem ser feitas rapidamente, com pouca ou nenhuma sobrecarga de memória.

Suponhamos que os blocos devam ser retirados de uma área contígua de memória. A inicialização da área é feita usando-se uma parte de cada bloco como um elo para o bloco seguinte. O apontador *próximo_disponível* aponta para o primeiro bloco. A alocação consiste em se retirar um bloco da lista e a liberação em colocar o bloco de volta na mesma.

As rotinas do compilador que gerenciam os blocos não precisam saber o tipo de objeto de dados que será guardado no bloco por parte do programa do usuário. Podemos tratar cada bloco como um registro variante, com as rotinas do compilador enxergando cada bloco como consistindo em um elo para um próximo bloco e o programa do usuário, por sua vez, vendo-o como de algum outro tipo. Por conseguinte, não existe sobrecarga de espaço, porque o programa do usuário pode utilizar todo o bloco para suas próprias finalidades. Quando o bloco é devolvido, as rotinas do compilador usam parte do espaço do próprio bloco para ligá-lo à lista de blocos disponíveis, como mostrado na Fig. 7.41.

Alocação Explícita de Blocos de Tamanhos Variados**

Quando os blocos são alocados e liberados, a memória se torna *fragmentada;* isto é, o *heap* pode consistir em blocos que estejam alternadamente livres e em uso, como na Fig. 7.42.

A situação mostrada na Fig. 7.42 pode ocorrer se um programa reservar cinco blocos e em seguida liberar o segundo e o quarto, por exemplo. A fragmentação não gera conseqüências se os blocos são de tamanho fixo, mas se são de tamanhos variados a situação da Fig. 7.42 é um problema, porque poderíamos não alocar um bloco maior do que qualquer um dos blocos livres, apesar de, em princípio, o espaço estar disponível.

Um método para se alocar blocos de tamanhos variados é chamado de *método do primeiro ajuste****. Quando um bloco de tamanho s é solicitado, pesquisamos pelo primeiro bloco livre que tenha tamanho $f \geq s$. Esse bloco é então subdividido num bloco usado de tamanho s e num bloco livre de tamanho $f - s$. Note-se que a alocação incorre numa sobrecarga de tempo, porque é necessário se procurar por um bloco livre que seja suficientemente grande.

Quando um bloco é liberado, verificamos se o mesmo é adjacente a um bloco livre. Se possível, o bloco liberado é combinado com o bloco livre adjacente a si, de forma a criar um bloco livre maior. A combinação de blocos adjacentes livres num bloco livre maior previne

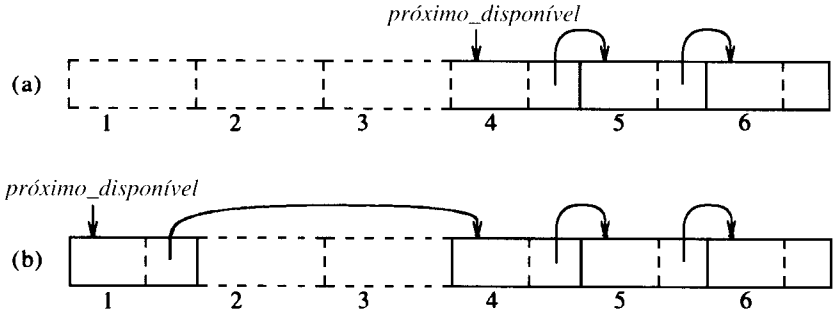


Fig. 7.41. Um bloco desalocado é retornado à lista de blocos disponíveis.

Fig. 7.42. Blocos livres e em uso num *heap*.

*Tamanho fixo, neste contexto, significa que todos os blocos possuem o mesmo tamanho. (N. do T.)

**Em princípio, o tamanho de um bloco não muda, uma vez que tenha sido reservado. Neste contexto, os blocos são de tamanho fixo, podendo, no entanto, existir blocos de diversos tamanhos no *heap*. (N. do T.)

***Do original em inglês: *first fit method*. (N. do T.)

que ocorra uma fragmentação posterior. Existe um número de detalhes sutis relacionados a como os blocos são alocados, liberados e mantidos numa lista, ou listas, de disponibilidade. Existem, também, diversas barganhas entre tempo, espaço e a disponibilidade de grandes blocos. O leitor é remetido a Knuth [1973a] ou a Aho, Hopcroft e Ullman [1983] para uma discussão desses temas.

Liberação Implícita

A liberação requer a cooperação entre o programa do usuário e o pacote de suporte em tempo de execução, porque este último necessita saber quando um bloco de memória já não está mais em uso. Essa cooperação é implementada através da fixação do formato dos blocos de memória. Para a discussão presente, suponhamos que o formato de um bloco de memória seja como na Fig. 7.43.

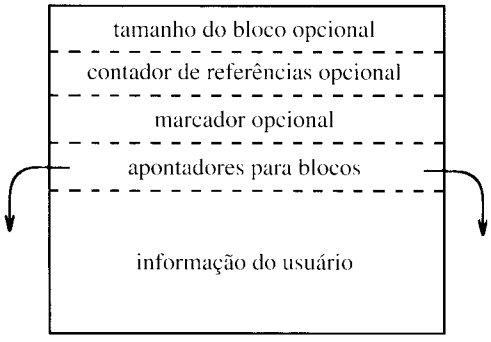


Fig. 7.43. Formato de um bloco.

O primeiro problema é o de reconhecer os limites de um bloco. Se o tamanho do bloco é fixo, informações de posição podem ser usadas. Por exemplo, se cada bloco ocupa 20 palavras, um novo bloco se inicia a cada 20 palavras. Em caso contrário, mantemos o seu tamanho na área inacessível associada ao bloco, de forma a que possamos determinar onde começa o próximo.

O segundo problema é o de reconhecer se um bloco está em uso. Assumimos que um bloco esteja em uso se for possível que o programa do usuário se refira a uma informação no mesmo. A referência pode ocorrer através de um apontador ou após se seguir uma seqüência de apontadores e, por conseguinte, o compilador precisa conhecer a posição de todos os apontadores na memória. Usando-se o formato da Fig. 7.43, os apontadores são mantidos em posições fixas nos blocos. Talvez mais adequada seja a suposição de que a área do bloco reservada às informações do usuário não contenha quaisquer apontadores.

Dois enfoques podem ser usados para a liberação implícita. Aqui, os mesmos são esboçados; para maiores detalhes, ver Aho, Hopcroft e Ullman [1983].

1. *Contadores de referências.* Controlamos o número de blocos que apontam diretamente para o bloco presente. Se esse contador chegar a zero, o bloco pode ser desalocado, uma vez que não pode ser referenciado. Em outras palavras, se tornou lixo e pode ser coletado. A manutenção de contadores de referências pode custar tempo; a atribuição de apontador `p := q` leva a mudanças no contador de referências dos blocos apontados tanto por `p` quanto por `q`. O contador para o bloco apontado por `p` decresce de um, enquanto que o do bloco apontado por `q` aumenta de um. Os contadores de referências são melhor utilizados quando os apontadores de um bloco nunca figuram em ciclos. Por exemplo, na Fig. 7.44, nenhum dos dois blocos está acessível por parte de outro, e, dessa forma, ambos são lixo, mas cada um tem uma contagem de referências diferente de zero (igual a um).

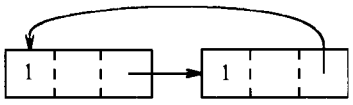


Fig. 7.44. Células que são lixo com contadores de referências diferentes de zero.

2. *Técnicas de marcação.* Um enfoque alternativo é o de suspender temporariamente a execução do programa do usuário e utilizar os apontadores congelados para determinar quais dos blocos estão em uso. Esse enfoque requer que todos os apontadores dentro do *heap* sejam conhecidos. Conceitualmente, "espargimos tinta" no *heap* através dos apontadores. Qualquer bloco que seja atingido pela tinta está em uso, o resto pode ser liberado. Em mais detalhes, varremos o *heap* marcando todos os blocos como *não-usados*. Depois, seguimos os apontadores marcando como *usado* qualquer bloco que tenha sido atingido durante o processo. Um esquadrinhamento seqüencial final do *heap* permite que todos os blocos que ainda estejam marcados como *não-usados* sejam coletados.

Com blocos de tamanhos variados, temos a possibilidade adicional de mover blocos de memória utilizados a partir de suas posições correntes.[8] Esse processo, chamado de *compactação*, move todos os blocos usados para uma extremidade do *heap*, de forma que toda a memória livre possa ser coletada num único grande bloco livre. A compactação também requer informações sobre os apontadores dentro dos blocos, pois quando um bloco é movido, todos os apontadores para o mesmo precisam ser ajustados de forma a refletir a movimentação. A vantagem está em que, em seguida, a fragmentação da memória disponível é eliminada.

7.9 ALOCAÇÃO DE MEMÓRIA EM FORTRAN

Fortran foi projetada para permitir a alocação estática de memória, como estabelecido na Seção 7.3. No entanto, existem alguns temas, tais como o tratamento de declarações `COMMON` e `EQUIVALENCE`, que são razoavelmente especiais em Fortran. Um compilador Fortran pode criar um certo número de *áreas de dados*, isto é, blocos de memória nos quais os valores dos objetos podem ser armazenados. Em Fortran, há uma área de dados para cada procedimento, para cada bloco `COMMON` com nome e para o bloco `COMMON` sem nome (em branco), se for usado. A tabela de símbolos precisa registrar, para cada nome, a área de dados a qual pertence e o seu deslocamento dentro dessa área, isto é, a posição relativa ao início da mesma. O compilador terá que, eventualmente, decidir onde as áreas estarão em relação ao código executável e na relação entre si, mas esta escolha é arbitrária, uma vez que as áreas de dados são independentes.

O compilador precisa computar o comprimento de cada área. Para as áreas de dados dos procedimentos, um único contador basta, uma vez que seus tamanhos são conhecidos depois de cada procedimento ter sido processado. Para os blocos `COMMON`, um registro para cada bloco precisa ser mantido durante o processamento de todos os procedimentos, uma vez que cada procedimento usando um desses blocos pode ter a sua própria idéia a respeito de quão grande o bloco é, e o tamanho efetivo será o maior dos tamanhos implicados pelos vários procedimentos. Se os procedimentos forem compilados separadamente, o editor de ligações precisará ser usado a fim de selecionar o tamanho do bloco `COMMON` para o maior de todos esses blocos com o mesmo nome, dentre os blocos de código que estiverem sendo ligados.

Para cada área, o compilador cria um *mapa de memória*, que é uma descrição do seu conteúdo. Esse "mapa de memória" poderia consistir simplesmente em uma indicação, na entrada da tabela de símbolos para cada nome existente na área, de seu respectivo deslocamento dentro da mesma. Não precisamos ter, necessariamente, uma resposta fácil para a questão "quais são todos os nomes dessa área de dados"? Em Fortran,

[8] Poderíamos fazer isso com blocos de tamanho fixo, sem resultados vantajosos.

entretanto, sabemos a resposta para as áreas de dados dos procedimentos, já que todos os nomes declarados num procedimento, que não sejam COMMON (comuns) ou equiparados a um nome COMMON, estão na área de dados do procedimento. Os nomes COMMON podem ter suas entradas na tabela de símbolos ligadas a uma cadeia para cada bloco COMMON, classificada pela ordem de aparição dos nomes no bloco. De fato, na medida em que os deslocamentos dos nomes na área de dados não podem ser sempre determinados até que todo o procedimento tenha sido processado (os *arrays* em Fortran podem ser declarados antes de suas dimensões o serem), é necessário que sejam criadas essas cadeias de nomes COMMON.

Um programa Fortran consiste em um programa principal, de subrotinas e de funções (chamados a todas de *procedimentos*). Cada ocorrência de um nome possui um escopo que consiste em um procedimento somente. Podemos gerar código objeto para cada procedimento ao atingirmos os seus fins respectivos. Se assim o fizermos, é possível que a maioria das informações na tabela de símbolos possa ser expurgada. Precisamos somente preservar aqueles nomes que sejam externos à rotina recém-processada. Esses são os nomes de outros procedimentos e dos blocos COMMON. Esses nomes não precisam ser verdadeiramente externos a todo o programa que está sendo compilado, mas precisam ser preservados até que toda a coleção de procedimentos seja processada.

Dados em Áreas COMMON

Criamos, para cada bloco, um registro fornecendo o primeiro e último nomes, pertencentes ao procedimento corrente, que são declarados como estando naquele bloco COMMON. Ao processar uma declaração como

```
COMMON /BLOCO1/ NOME1, NOME2
```

o compilador precisará fazer o seguinte:

1. Na tabela para nomes de blocos COMMON, criamos um registro para BLOCO1, se já não houver.
2. Nas entradas da tabela de símbolos para NOME1 e NOME2, estabelecer um apontador para a entrada da tabela de símbolos de BLOCO1, indicando que os mesmos são membros de um bloco COMMON e de nome BLOCO1.
3. a) Se acabou de ser criado um registro para BLOCO1, estabelecer um apontador, nesse registro, para a entrada de NOME1 na tabela de símbolos, indicando o primeiro nome definido nesse bloco COMMON. Em seguida, ligar a entrada da tabela de símbolos de NOME1 àquela de NOME2, usando um campo da tabela de símbolos reservado para ligar os membros do mesmo bloco COMMON. Finalmente, estabelecer um apontador, no registro para BLOCO1, para a entrada da tabela de símbolos de NOME2, indicando o último membro encontrado para aquele bloco.
   b) Se, entretanto, essa não for a primeira declaração para BLOCO1, simplesmente ligar NOME1 e NOME2 ao final da lista de nomes para BLOCO1. O apontador para o fim da lista para BLOCO1, figurando no registro para BLOCO1, é atualizado, naturalmente.

Após o procedimento ter sido processado, aplicamos o algoritmo de equivalência, a ser discutido em breve. Podemos descobrir que alguns nomes pertencem a um bloco COMMON por terem sido tornados equivalentes a nomes que estão em blocos COMMON. Iremos deduzir que não é de fato necessário ligar um tal nome XYZ à cadeia para seu bloco COMMON. Um *bit* na entrada da tabela de símbolos para XYZ é ligado, indicando que XYZ foi equiparado a algo. Uma estrutura de dados, a ser discutida, dará a posição de XYZ relativa a algum nome efetivamente declarado como estando em um bloco COMMON.

Após realizar as operações de equivalência, podemos criar um mapa de memória para cada bloco COMMON esquadrinhando a lista de nomes para aquele bloco. Inicializamos um contador em zero e, para cada nome na lista, fazemos seu deslocamento igual ao valor corrente do contador. Em seguida, adicionamos o número de unidades de memória ocupadas pelo objeto de dados denotado pelo nome. Os registros para os blocos COMMON podem, então, ser removidos e o espaço recusado pelo próximo procedimento.

Se um nome XYZ num bloco COMMON é equiparado a um nome que não esteja num tal bloco, precisamos determinar o deslocamento máximo, a partir do início de XYZ, de qualquer palavra de memória necessitada por qualquer nome equiparado a XYZ. Por exemplo, se XYZ for um real, equiparado a A(5,5), onde A é um *array* 10×10 de reais, A(1,1) figura 44 palavras antes de XYZ e A(10,10) 55 palavras após XYZ, como mostrado na Fig. 7.45. A existência de A não afeta o contador para o bloco COMMON; o mesmo só é incrementado de uma palavra quando XYZ é considerado, independentemente de XYZ ter sido equiparado ao que quer que seja. Entretanto, o fim da área de dados para o bloco COMMON precisa estar distante o bastante do início de forma a acomodar o *array* A. Por conseguinte, registramos o maior deslocamento, a partir do início do bloco COMMON, de qualquer palavra usada por um nome equiparado a um membro daquele bloco. Na Fig. 7.45, esse valor precisa ser, pelo menos, o deslocamento de XYZ mais 55. Verificamos também se o *array* A não se estende para antes do início da área de dados; isto é, o deslocamento de XYZ precisa ser pelo menos 44. Em caso contrário, temos um erro e precisamos produzir uma mensagem de diagnóstico.

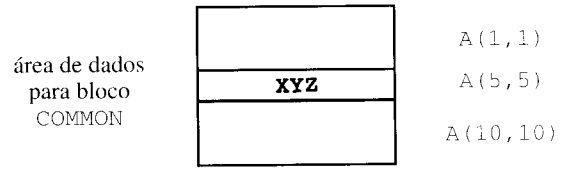


Fig. 7.45. Relação entre os enunciados COMMON e EQUIVALENCE.

Um Algoritmo Simples de Equivalência

Os primeiros algoritmos para processar enunciados de equivalência apareceram nos montadores ao invés de em compiladores. Como esses algoritmos podem ser um pouco complexos, especialmente quando as interações entre os enunciados COMMON e EQUIVALENCE são consideradas, vamos tratar primeiro de uma situação típica de uma linguagem de montagem, onde os únicos enunciados EQUIVALENCE são da forma

EQUIVALENCE A,B + *deslocamento*

onde A e B são nomes de localizações. Esse enunciado faz A denotar a localização que está *deslocamento* unidades de memória além da localização de B.

Uma seqüência de enunciados EQUIVALENCE agrupa, em *conjuntos de equivalência*, os nomes cujas posições relativas entre si são definidas pelos próprios enunciados EQUIVALENCE. Por exemplo, a seqüência de enunciados

```
EQUIVALENCE A, B+ 100
EQUIVALENCE C, D-40
EQUIVALENCE A, C+30
EQUIVALENCE E, F
```

agrupa os nomes nos conjuntos {A, B, C, D} e {E, F}, onde E e F denotam a mesma localização. C está 70 localizações após B, A está 30 após C e D 10 após A.

Para computar os conjuntos de equivalência, criamos uma árvore para cada conjunto. Cada nó da árvore representa um nome e contém o deslocamento daquele nome relativo ao nome que está ao pai desse nó. Denominamos o nome que está à raiz da árvore de *líder*. A posição de qualquer nome, relativa ao líder, pode ser computada se-

guindo-se o percurso a partir do nó para aquele nome e adicionando-se os deslocamentos ao longo do caminho, até a raiz.

Exemplo 7.12. O conjunto de equivalência {A, B C, D}, mencionado acima, poderia ser representado pela árvore mostrada na Fig. 7.46. D é o líder e podemos descobrir que A está localizado 10 posições antes de D, já que a soma dos deslocamentos ao longo do percurso que vai de A até D é 100 + (−110) = −10. □

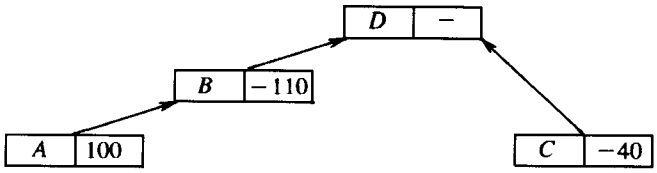


Fig. 7.46. Árvore representando o conjunto de equivalência.

Vamos agora fornecer um algoritmo para a construção de árvores para conjuntos de equivalência. Os campos relevantes nas entradas da tabela de símbolos são:

1. *pai*, apontando para a entrada da tabela de símbolos para o pai, nulo se o nome estiver à raiz (ou não equiparado a qualquer outro) e
2. *deslocamento*, fornecendo o deslocamento de um nome, relativo ao nome do pai.

O algoritmo que fornecemos assume que qualquer nome poderia ser o líder de um conjunto de equivalência. Na prática, numa linguagem de montagem, um, e somente um, nome no conjunto teria uma localização efetiva definida por uma pseudo-operação e esse nome seria tornado o líder. Confiamos em que o leitor possa ver como modificar o algoritmo e fazer um nome particular de líder.

Algoritmo 7.1. Construção de árvores de equivalência.

Entrada. Uma lista de enunciados definindo equivalências da forma

```
EQUIVALENCE   A,B+dist
```

Saída. Uma coleção de árvores tais que, para qualquer nome mencionado na lista de equivalências, possamos determinar a posição do nome relativa ao líder, seguindo o percurso que vai daquele nome até a raiz e somando os *deslocamentos* encontrados ao longo do mesmo.

Método. Repetimos os passos da Fig. 7.47 para cada enunciado de equivalência EQUIVALENCE A,B+*dist*, em turnos. A justificativa para a fórmula à linha (12) para o deslocamento do líder de A em relação ao líder de B é como segue. A localização de A, digamos l_A, é igual a c mais a localização do líder de A, digamos m_A. A localização de B, digamos l_B, é igual a d mais a localização do líder de B, digamos m_B. Mas $l_A = l_B + dist$ e, então, $c + m_A = d + m_B + dist$. Por conseguinte, $m_A - m_B$ é igual a $d - c + dist$. □

Exemplo 7.13. Se processarmos

```
EQUIVALENCE   A, B+100
EQUIVALENCE   C,D-40
```

obtemos a configuração mostrada na Fig. 7.46, mas sem o deslocamento −110 no nó para B e sem elo de B para D. Ao processarmos

```
EQUIVALENCE   A,C+30
```

encontramos que p aponta para B após o laço-enquanto à linha (3) e que q aponta para d após o laço-enquanto à linha (6). Temos igualmente que c = 100 e que d = −40. Por conseguinte, à linha (11) fazemos D o pai de B e estabelecemos o campo *deslocamento* para B com o valor 100, que é o resultado de (−40)−(100)+30. □

```
      início
(1)       sejam p e q apontadores para os nós de A e B respectivamente;
(2)       c := 0; d := 0; /* c e d computam os deslocamentos de A e B
              a partir dos líderes de seus respectivos conjuntos */
(3)       enquanto pai(p) ≠ nulo faça início
(4)           c := c + deslocamento(p);
(5)           p := pai(p)
          fim;  /* mover p para o líder de a, acumulando os
                   deslocamentos à medida que cominhemos */
(6)       enquanto pai(p) ≠ nulo faça início
(7)           d := d + deslocamento(q);
(8)           q := pai(q)
          fim; /* fazer o mesmo para B */
(9)       se p = q então /* A e B já estão equiparados */
(10)          se c — d ≠ dist então erro;
                  /* foram atribuídas duas posições relativas diferentes
                     a A e B */
          senão início  /* combinar os conjuntos de A e B */
(11)          pai(p) := q /* fazer o líder de A um filho do líder de B */
(12)          deslocamento(p) := d − c + dist
          fim
      fim
```

Fig. 7.47. Algoritmo de equivalência.

O Algoritmo 7.1 poderia levar um tempo proporcional a n^2 para processar n equivalências, uma vez que, no pior caso, os percursos seguidos pelos laços às linhas (3) e (6) poderiam incluir cada nó de suas respectivas árvores. A equiparação (através da equivalência) requer apenas uma pequenina fração do tempo gasto durante a compilação, e dessa forma, n^2 passos não é um preço proibitivo e um algoritmo mais complexo do que aquele da Fig. 7.47 não é provavelmente justificável. No entanto, acontece que existem duas coisas fáceis que podemos fazer para tornar o tempo que Algoritmo 7.1 gasta apenas linear em relação ao número de equivalências processadas. Conquanto não seja provável que os conjuntos de equivalência sejam grandes o suficiente, em média, de forma que esses aprimoramentos precisem ser realmente implantados, é importante notar que a equiparação serve como um paradigma para um certo número de processos importantes envolvendo a "combinação de conjuntos". Por exemplo, um número de algoritmos eficientes para a análise do fluxo de dados depende de algoritmos de equivalência rápidos; o leitor interessado é remetido às notas bibliográficas do Capítulo 10.

O primeiro aprimoramento que podemos fazer é manter uma contagem, para cada líder, do número de nós em sua árvore. Então, às linhas (11) e (12), em lugar de ligar arbitrariamente o líder de A ao líder de B, ligue-se o que tiver a menor contagem ao outro. Isto assegura que as árvores cresçam achatadas, de forma que os percursos serão curtos. É deixado como exercício para o leitor comprovar que n equivalências realizadas dessa forma não podem produzir percursos mais longos do que $\log_2 n$ nós.

A segunda idéia é conhecida como compressão de percursos. Ao seguir um percurso até a raiz, nos laços às linhas (3) e (6), fazer todos

```
início
        h := deslocamento (n_{k-1});
        para i := k-2 de_volta_até 1 faça início
            pai(n_i) := n_k;
            h := h + deslocamento (n_i);
            deslocamento (n_i) := h
        fim
fim
```

Fig. 7.48. Ajuste dos deslocamentos.

os nós encontrados filhos do líder, se já não o forem. Isto é, enquanto se seguir o percurso, registrar todos os nós $n_1, n_2, ..., n_k$ encontrados, onde n_1 é o nó para A ou B e n_k é o líder. Em seguida, ajustar os deslocamentos e fazer $n_1, n_2, ..., n_{k-2}$ filhos de n_k através dos passos enumerados na Fig. 7.48.

Um Algoritmo de Equivalência para Fortran

Existem diversas figurações que precisam ser adicionadas ao Algoritmo 7.1, para fazê-lo funcionar para Fortran. Primeiro, precisamos determinar se o conjunto de equivalência está em um bloco COMMON, o que podemos fazer através do registro para cada líder, informando se qualquer um dos nomes no seu conjunto está em algum bloco COMMON, e se estiver, em que bloco.

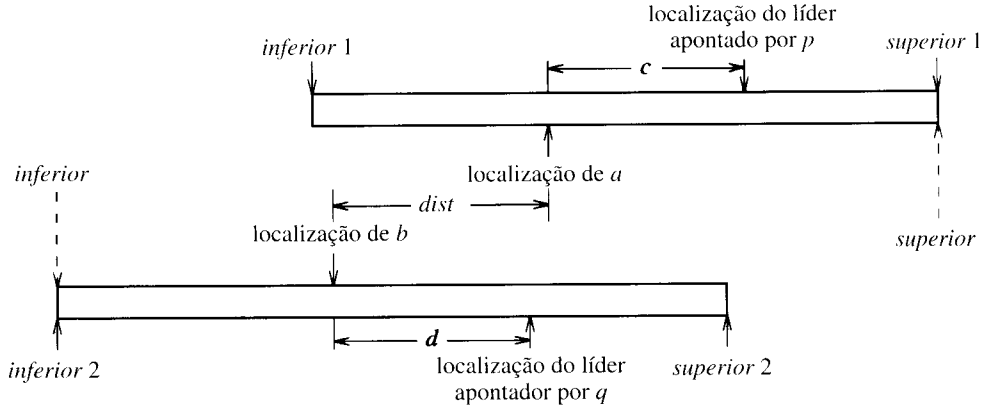
Segundo, numa linguagem de montagem, um membro de um conjunto de equivalência irá materializar todo o conjunto por ser um rótulo de um enunciado, permitindo, por conseguinte, que todos os endereços denotados por todos os elementos daquele conjunto sejam computados em relação àquela localização. Em Fortran, entretanto, é tarefa do compilador determinar as localizações de memória e, dessa forma, um conjunto de equivalência que não esteja num bloco COMMON pode ser considerado como "flutuando" até que o compilador determine a posição de todo o conjunto em sua área de dados apropriada. Para realizar essa tarefa corretamente, o compilador precisa conhecer a extensão do conjunto de equivalência, isto é, o número de localizações que os nomes no conjunto ocupam coletivamente. Para tratar esse problema, associamos ao líder dois campos, *inferior* e *superior*, fornecendo os deslocamentos relativos ao líder das localizações mais baixa e mais alta usadas por qualquer membro do conjunto de equivalência. Terceiro, existem problemas menores introduzidos pelo fato de que os nomes podem ser *arrays* e as localizações no meio de um *array* podem ser equiparadas a localizações em outros *arrays*.

Uma vez que existem três campos (*inferior*, *superior* e um apontador para um bloco COMMON) que precisam ser associados a cada líder, não queremos reservar espaço para esses campos em todas as entradas da tabela de símbolos. Um curso de ação é usar o campo *pai* no Algoritmo 7.1 de forma a apontar, no caso do líder, para um registro numa tabela com três campos, *inferior*, *superior* e *bloco_common*. Na medida em que essa tabela e a de símbolos ocupam áreas separadas, podemos registrar que tabela o apontador endereça. Alternativamente, a tabela de símbolos pode ter um *bit* indicando se um nome é correntemente um líder. Se o espaço estiver realmente a prêmio, um algoritmo alternativo, que evita essa tabela extra às expensas de um pequeno esforço de programação adicional, é discutido nos exercícios.

Vamos considerar o cálculo que deve substituir as linhas (11) e (12) da Fig. 7.47. A situação na qual dois conjuntos de equivalência, cujos líderes são apontados por p e q, precisam ser combinados é delineada na Fig. 7.49(a). A estrutura de dados que representa os dois conjuntos aparece na Fig. 7.49(b). Primeiro, precisamos verificar que não existam dois membros dentre os dois conjuntos de equivalência que estejam em blocos COMMON. Mesmo que ambos estejam no mesmo bloco, o padrão Fortran proíbe que sejam equiparados. Se qualquer bloco COMMON contém um membro de um dos dois conjuntos de equivalência, o conjunto combinado possui um apontador para o registro daquele bloco em *bloco_common*. O código que faz essa verificação, assumindo que o líder apontado por q se torna o líder do conjunto combinado, é mostrado na Fig. 7.50. No lugar das linhas (11) e (12) da Fig. 7.47, precisamos também computar a extensão do conjunto de equivalência combinado. A Fig. 7.49(a) indica as fórmulas para os novos valores de *inferior* e *superior*, relativos ao líder apontado por q.

```
início
    bloco_common_1 := bloco_common(pai(p));
    bloco_common_2 := bloco_common(pai(q));
    se bloco_common_1 ≠ null e bloco_common_2 ≠ null então
        erro;    /* dois nomes em bloco COMMON equivalentes */
    senão se bloco_common_2 = null então
        bloco_common (pai(q)) := bloco_common_1
fim
```

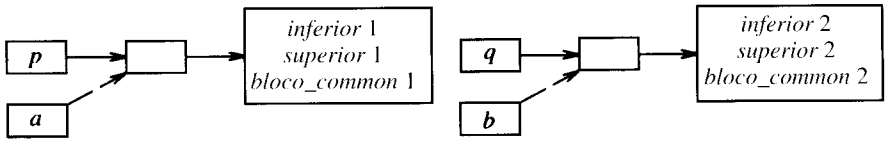
Fig. 7.50. Computando os blocos COMMON.



$$inferior = \text{mínimo}(inferior\ 2,\ inferior\ 1 - c + dist + d)$$

$$superior = \text{máximo}(superior\ 2,\ superior\ 1 - c + dist + d)$$

(a) posições relativas dos conjuntos de equivalência.



(b) estrutura de dados

Fig. 7.49. Combinando conjuntos equivalentes.

Por conseguinte, precisamos fazer

início
inferior (pai(q)) := mínimo *(inferior(pai(q)), inferior(pai(p)) −c +dist+d);*
superior(pai(q)) := máximo *(superior(pai(q)), superior(pai)(p)) −c+dist+d)*
fim

Esses enunciados são seguidos pelas linhas (11) e (12) da Fig. 7.47, para efetuar a combinação dos dois conjuntos de equivalência.

Dois últimos detalhes precisam ser cobertos para fazer o Algoritmo 7.1 funcionar para Fortran. Em Fortran, podemos equiparar, através do enunciado de equivalência, posições ao meio de um *array* às posições em outros *arrays* ou a nomes simples. O deslocamento do *array* `A` a partir do líder significa o deslocamento da primeira localização de `A` a partir da primeira localização do líder. Se uma localização como `A(5,7)` é equiparada, digamos, a `B(20)`, precisamos computar a posição de `A(5,7)`, relativa a `A(1,1)`, e inicializar *c* para o valor negativo dessa distância, na linha (2) da Fig. 7.47. Similarmente, *d* precisa ser inicializado para o negativo da posição de `B(20)` relativa a `B(1)`. As fórmulas da Seção 8.3, juntamente com o conhecimento do tamanho dos elementos dos *arrays* `A` e `B` são suficientes para calcular os valores iniciais de *c* e *d*.

O último detalhe a ser coberto é o fato de Fortran permitir um enunciado `EQUIVALENCE` que envolva muitas localizações, tal como

```
EQUIVALENCE (A(5,7), B(20), C, D(4,5,6))
```

Isto acima pode ser tratado como

```
EQUIVALENCE (B(20), A(5,7))
EQUIVALENCE (C, A(5,7))
EQUIVALENCE (D(4,5,6), A(5,7))
```

Note-se que se fizermos as equivalências nesta ordem, somente `A` se torna o líder de um conjunto com mais de um elemento. Um registro com *inferior, superior* e *bloco_common* pode ser usado muitas vezes para "conjuntos de equivalência" de um único nome.

Mapeando Áreas de Dados

Podemos descrever agora as regras através das quais o espaço nas diversas áreas de dados é atribuído aos nomes pertencentes a cada rotina.

1. Para cada bloco `COMMON`, visitar todos os nomes declarados como figurando àquele bloco, na ordem ds suas declarações (usar as cadeias de nomes `COMMON` criadas na tabela de símbolos para esse propósito). Alocar o número de palavras necessitado para cada nome, mantendo uma contagem do número de palavras alocadas, de forma que os deslocamentos possam ser computados para cada nome. Se um nome `A` for equiparado através de um enunciado de equivalência, a extensão do seu conjunto de equivalência não importa, mas precisamos assegurar que o valor de *inferior* para o líder de `A` não vá para antes do início do bloco `COMMON`. Consultar o valor *superior* para o líder, de forma a colocar um limite inferior na última palavra do bloco. Deixamos as fórmulas exatas para esses cálculos por conta do leitor.
2. Visitar todos os nomes para a rotina em qualquer ordem.
   a) Se um nome estiver em um bloco `COMMON`, não fazer nada. O espaço já foi reservado em (1).
   b) Se um nome não estiver em um bloco `COMMON` e não for equiparado através de enunciado de equivalência, reservar o número adequado de palavras na área de dados para a rotina.
   c) Se um nome `A` for equiparado através de um enunciado de equivalência, encontrar o seu líder, digamos `L`. Se a `L` ainda não foi atribuída uma posição na área de dados para a rotina, computar a posição de `A` adicionando àquela posição todos os *deslocamentos* encontrados no percurso de `A` até `L` na árvore, representando os conjuntos de equivalência de `A` e de `L`. Se a `L` ainda não foi fornecida uma posição, aloque as próximas *superior-inferior* palavras, na área de dados, para o conjunto de equivalência. A posição de `L` nessas palavras está a *-inferior* palavras do início e a posição de `A` pode ser calculada somando os *deslocamentos* como antes.

EXERCÍCIOS

7.1 Usando as regras de escopo de Pascal, determine as declarações que se aplicam a cada ocorrência dos nomes `a` e `b` na Fig. 7.51. A saída do programa consiste nos inteiros de `1` a `4`.

```
program a(input, output);
procedure b(u, v, x, y: integer);
   var a : record a, b : integer end;
       b : record b, a : integer end;
   begin
      with a do begin a := u; b := v end;
      with b do begin a := x; b := y end;
      writeln(a.a, a.b, b.a, b.b)
   end;
begin
   b(1, 2, 3, 4)
end.
```

Fig. 7.51. Programa Pascal com várias declarações para `a` e `b`.

7.2 Considere uma linguagem estruturada em blocos, na qual um nome possa ser declarado como um inteiro ou real. Suponha que as expressões sejam representadas por um terminal **expr** e que os únicos enunciados sejam atribuições, comandos condicionais, comandos *while*, e seqüências de comandos. Assumindo que os inteiros sejam alocados em uma palavra e os reais em duas, forneça uma definição dirigida pela sintaxe (baseada numa gramática razoável para declarações e blocos) para determinar as amarrações, a partir dos nomes, para as palavras de memória que possam ser usadas por uma ativação de um bloco. A sua alocação usa um número mínimo de palavras adequadas a qualquer execução de um bloco?

***7.3** Na Seção 7.4, clamamos que o *display* poderia ser mantido corretamente se cada procedimento à profundidade *i* armazenasse *d*[*i*] ao início da ativação e restaurasse *d*[*i*] ao final. Prove, por indução no número de chamadas, que cada procedimento enxerga o *display* correto.

7.4 Uma *macro* é uma forma de procedimento, implementada substituindo literalmente cada chamada do procedimento pelo seu corpo. A Fig. 7.52 mostra o programa Pic e sua saída. As duas primeiras linhas definem as macros `show` (exibir) e `small`

```
define show % { circulo de raio r aqui } %
define small % [ r = 1/12; show ] %
[
    r = 1/6;
    show; small;
    move; { desloca as coordenadas do centro do
        circulo }
    show; small;
]
```

Fig. 7.52. Círculos desenhados pelo programa Pic.

(reduzir). Os corpos das macros estão contidos entre os dois símbolos % às linhas. Cada um dos quatro círculos na figura são desenhados usando-se `show`; o raio do círculo é dado pelo nome não local `r`. Os blocos em Pic são delimitados por [e]. Cada variável associada a um de um bloco é implicitamente declarada dentro do mesmo. A partir da saída, o que pode ser dito a respeito do escopo de cada ocorrência de `r`?

7.5 Escreva um procedimento para inserir um item numa lista ligada passando-se um apontador para a cabeça da lista. Sob que mecanismos de transmissão de parâmetros esse procedimento funcionará?

7.6 O que é impresso pelo programa na Fig. 7.53, assumindo (a) chamada por valor, (b) chamada por referência, (c) ligação de cópia e restauração, (d) chamada por nome?

```
program main (input, output);
    procedure p(x, y, z);
        begin
            y := y + 1;
            z := z + x;
        end;
    begin
        a := 2;
        b := 3;
        p(a + b, a, a);
        print a
    end.
```

Fig. 7.53. Pseudoprograma ilustrando a transmissão de parâmetros.

7.7 Quando um procedimento é transmitido como parâmetro numa linguagem de escopo léxico, seu ambiente não local pode ser transmitido usando-se um elo de acesso. Forneça um algoritmo para determinar esse elo.

7.8 Os três tipos de ambientes que poderiam ser associados a um procedimento passado como parâmetro são ilustrados pelo programa Pascal na Fig. 7.54. Os ambientes *léxico, de transmissão* e *de ativação* de um tal procedimento consistem nas amarrações de identificadores nos pontos em que o procedimento é definido, transmitido como parâmetro e ativado, respectivamente. Considere a função `f`, transmitida como um parâmetro à linha 11. Usando os ambientes léxico, de transmissão e de ativação de `f`, o nome não local `m` à linha 8 está no escopo para as declarações de `m` às linhas 6, 10 e 3, respectivamente.

a) Desenhe a árvore de ativações para esse programa.

```
(1)  program param(input, output);

(2)  procedure b(function h(n: integer) :integer);
(3)     var m : integer;
(4)     begin m := 3; writeln(h(2)) end { b };

(5)  procedure c;
(6)     var m : integer;

(7)        function f(n : integer) : integer;
(8)           begin f := m + n end { f };

(9)        procedure r;
(10)          var m : integer;
(11)          begin m := 7; b(f) end { r };

(12)    begin m := 0; r end { c };

(13) begin
(14)    c
(15) end.
```

Fig. 7.54. Exemplo dos ambientes léxico, de transmissão e de ativação.

b) Qual é a saída do programa, usando-se os ambientes léxico, de transmissão e de ativação para `f`?

*c) Modifique a implementação do *display* para as linguagens de escopo léxico, de forma a estabelecer o ambiente léxico corretamente quando um procedimento transmitido como parâmetro for ativado.

***7.9** O enunciado $f := a$ à linha 11 do pseudoprograma da Fig. 7.55 chama a função a, que transmite a função *addm* de volta como resultado.

a) Desenhe a árvore de ativações desse programa.

b) Suponha que o escopo léxico seja usado para nomes não locais. Por que o programa irá falhar se a alocação de pilha for usada?

c) Qual é a saída do programa com a alocação de *heap*?

```
(1)  program ret(input, output);
(2)  var f : function (integer) : integer;

(3)     function a : function (integer) : integer;
(4)        var m : integer;
(5)        function addm (n : integer) : integer;
(6)           begin return m + n end;
(7)        begin m := 0; return addm end;

(8)     procedure b(g : function (integer) : integer);
(9)        begin writeln(g(2)) end;

(10)    begin
(11)       f := a ; b(f)
(12)    end.
```

Fig. 7.55. Pseudoprograma no qual a função *addm* é retornada como resultado.

***7.10** Certas linguagens, como Lisp, têm a habilidade de retornar procedimentos recentemente criados em tempo de execução. Na Fig. 7.56, todas as funções, quer definidas no texto do programa-fonte, quer criadas em tempo de execução, recebem no máximo um argumento e retornam um valor, que é uma função ou um real. O operador $\circ$ significa a composição de funções; isto é $(f \circ g)(x) = f(g(x))$.

a) Qual é o valor imprimido por *main*?

*b) Suponhamos que, sempre que um procedimento p for criado e retornado, seu registro de ativação se torna um filho do registro de ativação para a função que retorna p. O am-

```
function f (x : function);
var   y : function;
          y := x ∘ h;    /* cria y quando executado    */
          return y
end { f };

function h( );
      return sin
end { h};

function g(z : function);
var   w : function;
          w := arctan ∘ z;    /* cria w quando executado  */
          return w
end { g };

function main ();
var   a : real;
          u, v : function;
          v := f(g);
          u := v();
          a := u(π/2);
          print a
end { main }.
```

Fig. 7.56. Pseudoprograma que cria funções em tempo de execução.

biente de transmissão de p pode então ser mantido através de uma árvore de registros de ativação em lugar de uma pilha. O que é a árvore de registros de ativação quando a é computado por *main* na Fig. 7.56?

*c) Alternativamente, suponha que um registro de ativação para p seja criado quando p for ativado e que seja tornado filho do registro de ativação para o procedimento chamador p. Esse enfoque pode ser usado para manter o ambiente de ativação para p. Desenhe instantâneos dos registros de ativação e seus relacionamentos pai-filho na medida em que os comandos em *main* sejam executados. É uma pilha suficiente para abrigar os registros de ativação quando esse enfoque for usado?

7.11 Outra forma de se tratar a remoção, nas tabelas *hash*, dos nomes cujo escopo já tenha passado (como na Seção 7.6) é deixar os nomes expirados numa lista, até que a lista seja pesquisada de novo. Assumindo que as entradas incluam o nome do procedimento no qual a declaração é feita, podemos, em princípio, informar se um nome é antigo e removê-lo, se for o caso. Forneça um esquema de indexação para procedimentos que nos habilite informar, num tempo $O(1)$, se um procedimento é "antigo", isto é, se seu escopo já passou.

7.12 Muitas funções *hash* podem ser caracterizadas por uma seqüência de constantes inteiras $\alpha_0, \alpha_1, \ldots$. Se c_i, $1 \le i \le n$, é o valor inteiro do i-ésimo caractere na cadeia s, então a cadeia é *hashenizada* para

$$hash(s) = (\alpha_0 \cdot \sum_{i=1}^{n} \alpha_i c_i) \textbf{ mod } m$$

onde m é o tamanho da tabela *hash*. Para cada um dos seguintes casos, determine a seqüência de constantes $\alpha_0, \alpha_1, \ldots$, ou mostre que não existe uma tal seqüência. Cada caso determina um inteiro; um valor *hash* é obtido tomando-se esse inteiro módulo m.

a) Obter a soma dos caracteres.
b) Obter a soma do primeiro e último caracteres
c) Obter h_n, onde $h_0 = 0$ e $h_i = 2h_{i-1} + c_i$.
d) Tratar os *bits* dos 4 caracteres centrais como um inteiro de 32 *bits*.
e) Um inteiro de 32 *bits* pode ser visto como constituído de 4 *bytes*, onde cada *byte* é um dígito que toma um dentre 256 possíveis valores. Começando por 0000, para $1 \le i \le n$, adicione c_i ao *byte* i **mod** 4, permitindo a ocorrência de vai-um. Isto é, c_1 e c_5 são adicionados ao *byte* 1, c_2 e c_6 ao *byte* 2 e assim por diante. Retornar o valor final.

***7.13** Por que as funções *hash*, caracterizadas por uma seqüência de inteiros $\alpha_0, \alpha_1, \ldots$, como no Exercício 7.12, têm algumas vezes um desempenho pobre se a entrada consistir em cadeias consecutivas, como, por exemplo, `v000`, `v001`, ... ? O sintoma é que, em algum lugar ao longo do caminho, seu comportamento se desvia do randômico e pode ser predito.

****7.14** Quando n cadeias podem ser *hashenizadas* em m listas, o número médio de cadeias por lista é n/m, não importa quão não homogeneamente as cadeias estejam distribuídas. Suponha que d seja a "distribuição", isto é, uma cadeia randômica é colocada à i-ésima lista com probabilidade $d(i)$. Suponhamos que uma função de *hash* com distribuição d coloque b_j cadeias randomicamente selecionadas na lista j, $0 \le j \le m-1$. Mostre que o valor esperado $W = \sum_{j=0}^{m-1} (b_j)(b_j+1)/2$ está linearmente relacionado à variância da distribuição d. Para uma distribuição uniforme, mostre que o valor esperado W é $(n/2m)(n + 2m - 1)$.

7.15 Suponha que tenhamos a seguinte seqüência de declarações num programa Fortran.

```
SUBROUTINE SUB(X, Y)
INTEGER A, B(20), C(10,15), D, E
COMPLEX F, G
COMMON /CBLK/ D, E
EQUIVALENCE (G, B(2))
EQUIVALENCE (D, F, B(1))
```

Mostre o conteúdo das áreas de dados para `SUB` e `CBLK` (pelo menos a parte de `CBLK` acessível a partir de `SUB`). Por que não há espaço para `X` e `Y`?

***7.16** Uma estrutura de dados útil para o cômputo de equivalências é a *estrutura de anel*. Usamos um apontador e um campo de deslocamento em cada entrada da tabela de símbolos para ligar membros de um conjunto de equivalência. Esta estrutura é sugerida pela Fig. 7.57, onde `A`, `B`, `C` e `D` são equivalentes e `E` e `F` também o são, com a localização de `B` estando 20 palavras após aquela de `A` e assim por diante.

a) Forneça um algoritmo para computar o deslocamento de `X` relativo a `Y` assumindo que `X` e `Y` estão no mesmo conjunto de equivalência.
b) Forneça um algoritmo para computar *inferior* e *superior*, conforme definido na Seção 7.9, relativos à localização de algum nome `Z`.
c) Forneça um algoritmo para processar

```
EQUIVALENCE U, V
```

Não assuma que `U` e `V` estejam necessariamente em conjuntos de equivalência diferentes.

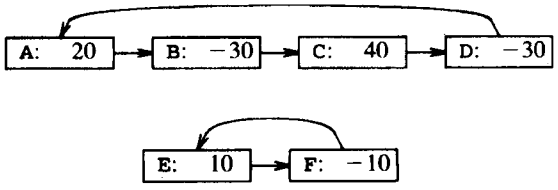


Fig. 7.57. Estruturas em anel.

***7.17** O algoritmo para mapear áreas de dados, fornecido na Seção 7.9, requer que verifiquemos que o valor de *inferior*, para o líder do conjunto de equivalência de `A`, não faça com que o espaço para o conjunto de equivalência de `A` se estenda para antes do começo do bloco `COMMON` e que calculemos o valor de *superior* para o líder de `A`, de forma a incrementar o limite superior do bloco `COMMON`, se necessário. Forneça fórmulas, em termos de *próximo*, o deslocamento de `A` no bloco `COMMON`, e de *última*, a última palavra corrente do bloco, para realizar o teste e atualizar *última*, se necessário.

NOTAS BIBLIOGRÁFICAS

As pilhas desempenharam um papel essencial na implementação das funções recursivas. McCarthy [1981, p. 178] relembra que durante um projeto de implementação de Lisp, começado em 1958, foi decidido "usar uma única pilha contínua e pública para salvar os valores das variáveis e endereços de retorno de sub-rotinas na implementação das sub-rotinas recursivas". A inclusão de blocos e de procedimentos recursivos em Algol 60 — ver Naur [1981, Seção 2.10] para uma contabilidade detalhada de seus projetos — também estimulou o desenvolvimento da alocação de pilha. A idéia de um *display* para se dar acesso aos nomes não locais numa linguagem de escopo léxico é devida a Dijkstra [1960, 1963]. Apesar de Lisp utilizar o escopo dinâmico, é possível se encontrar o efeito do escopo léxico ao se usar de "amarra-

ções profundas"* consistindo em uma função e um elo de acesso; McCarthy [1981] descreve o desenvolvimento desse mecanismo. As linguagens sucessoras de Lisp, como Common Lisp (Steele [1984]), têm se afastado do escopo dinâmico.

Explicações a respeito das amarrações para os nomes podem ser encontradas em livros-texto sobre linguagens de programação, ver por exemplo Abelson e Sussman [1985], Pratt [1984] ou Tenent [1981]. Uma abordagem alternativa, sugerida no Capítulo 2, é ler a descrição de um compilador. O desenvolvimento passo a passo em Kernighan e Pike [1984] começa com uma calculadora para expressões aritméticas e constrói um interpretador para uma linguagem simples com procedimentos recursivos. Ou veja-se o código para Pascal-S em Wirth [1981]. Uma descrição detalhada da alocação de pilha, do uso do *display* e da alocação dinâmica de *arrays* aparece em Randell e Russell [1964].

Johnson e Ritchie [1981] discutem o projeto da seqüência de chamada que permite que o número de argumentos de um procedimento varie de chamada a chamada. Um método geral para se estabelecer um *display* global é o de seguir a cadeia de elos de acesso, estabelecendo os elementos do *display* no processo. O enfoque da Seção 7.4, que toca exatamente um único elemento, parece ter sido "bem conhecido" por algum tempo; uma referência publicada é Rohl [1975]. Moses [1970] discute as distinções cabíveis entre os ambientes quando uma função é transmitida como parâmetro e considera os problemas que emergem quando tais ambientes são implementados usando-se os acessos superficial e profundo. A alocação de pilha não pode ser usada para linguagens com corrotinas ou múltiplos processos. Lampson [1982] considera implementações rápidas usando a alocação de *heap*.

Na lógica matemática, as variáveis quantificadas, de escopo e substituição limitadas, aparecem com Begriffsschrift de Frege [1879]. A substituição e a transmissão de parâmetros têm sido assunto de muitos debates, tanto nas comunidades de lógica matemática quanto nas de linguagens de programação. Church [1956, p. 288] observa: "Especialmente difícil é o assunto do enunciado correto da regra da substituição pelas variáveis funcionais", e relata o desenvolvimento de tal regra para o cálculo proposicional. O *lambda calculus* de Church [1941] tem sido aplicado a ambientes de linguagens de programação, por exemplo, por Landin [1964]. Um par consistindo em uma função e um elo de acesso é freqüentemente referenciado como um *fechamento*, seguindo-se a Landin [1964].

Estruturas de dados para as tabelas de símbolos e algoritmos para pesquisá-las são discutidos em detalhes em Knuth [1973b] e Aho, Hopcroft e Ullman [1974, 1983]. A base do *hashing* é tratada em Knuth [1973b] e Morris [1968b]. O artigo original discutindo o *hashing* é Peterson [1957]. Mais a respeito da organização de tabelas de símbolos pode ser encontrado em McKeeman [1976]. O Exemplo 7.10 é de Bentley, Cleveland e Sethi [1985]. Reiss [1983] descreve um gerador de tabelas de símbolos.

Os algoritmos de equivalência foram descritos por Arden, Galler e Graham [1961] e Galler e Fischer [1964]; adotamos o último enfoque. A eficiência dos algoritmos de equivalência é discutida em Fischer [1972], Hopcroft e Ullman [1973] e Tarjan [1975].

*Do original em inglês: "*funargs*" (as aspas são do original). Esse tipo de amarração é definido como "*deep or funarg binding*" no livro antecessor deste, *Principles of Compiler Design*, citado na bibliografia sob a chamada Aho e Ullman [1977]. (N. do T.)

CAPÍTULO 8

GERAÇÃO DE CÓDIGO INTERMEDIÁRIO

No modelo de análise e síntese de um compilador, os módulos da vanguarda traduzem o programa-fonte numa representação intermediária, a partir da qual os módulos da retaguarda geram o código-alvo. Na medida do possível, os detalhes da linguagem-alvo são confinados ao máximo nos módulos da retaguarda. Apesar de se poder traduzir o programa-fonte diretamente na linguagem-alvo, alguns dos benefícios em se usar uma forma intermediária independente da máquina são:

1. O redirecionamento é facilitado; um compilador para uma máquina diferente pode ser criado atrelando-se à vanguarda existente uma retaguarda para a nova máquina.
2. Um otimizador de código independente da máquina pode ser aplicado à representação intermediária. Tais otimizadores são discutidos em detalhes no Capítulo 10.

Este capítulo mostra como os métodos dirigidos pela sintaxe, dos Capítulos 2 e 5, podem ser usados para traduzir, numa forma intermediária, construções de linguagens de programação, tais como declarações, atribuições e enunciados de fluxo de controle. Por uma questão de simplicidade, assumimos que o programa-fonte já foi analisado sintaticamente e verificado estaticamente, como na organização da Fig. 8.1. A maioria das definições dirigidas pela sintaxe deste capítulo pode ser implementada quer durante uma análise sintática *bottom-up*, quer *top-down*, usando-se as técnicas do Capítulo 5, de forma que, se desejado, a geração de código intermediário pode ser imiscuída na análise sintática.

8.1 LINGUAGENS INTERMEDIÁRIAS

As árvores sintáticas e a notação pós-fixa, introduzidas nas Seções 5.2 e 2.3, respectivamente, são dois tipos de representações intermediárias. Uma terceira, chamada de código de três endereços, será usada neste capítulo. As regras semânticas para gerar o código de três endereços, a partir de construções comuns das linguagens de programação, são similares àquelas para construir árvores sintáticas e gerar a notação pós-fixa.

Representações Gráficas

Uma árvore sintática delineia a estrutura hierárquica natural de um programa-fonte. Um GDA fornece as mesmas informações, mas de uma forma mais compacta, porque as subexpressões comuns são identificadas. Uma árvore sintática e um GDA para o enunciado de atribuição `a := b * - c + b * - c` aparecem na Fig. 8.2.

A notação pós-fixa é uma representação linearizada de uma árvore sintática; é uma lista dos nós da árvore na qual um nó aparece imediatamente após seus filhos. A notação pós-fixa para a árvore sintática na Fig. 8.2(a) é

`a b c uminus * b c uminus * + assign` (8.1)

Os lados numa árvore sintática não aparecem explicitamente na notação pós-fixa. Podem ser recuperados a partir da ordem na qual os nós aparecem e do número de operandos que o operador, a um nó, espera receber. A recuperação dos lados de uma expressão na notação pós-fixa é similar à avaliação usando-se uma pilha. Veja a Seção 2.8 para mais detalhes e o relacionamento entre a notação pós-fixa e o código para uma máquina de pilha.

As árvores sintáticas para os enunciados de atribuição são produzidas pela definição dirigida pela sintaxe na Fig. 8.3; é uma extensão daquela na Seção 5.2. O não-terminal S gera um enunciado de atribuição. Os dois operadores binários + e * são exemplos do conjunto completo de operadores numa linguagem típica. As associatividades e precedências dos operadores são as usuais, ainda que esses não tenham sido colocados na gramática. Essa definição constrói a árvore da Fig. 8.2(a) a partir da entrada `a := b * -c + b * -c`.

Esta mesma definição dirigida pela sintaxe irá produzir o GDA da Fig. 8.2(b) se as funções *criar_nó_un (op, filho)* e *criar_nó (op, filho_esq, filho_dir)* retornarem um apontador para um nó já existente, em lugar de construir novos nós. O *token* **id** possui um atributo *local* que aponta para a entrada da tabela de símbolos relativa ao identificador. Na Seção 8.3, mostramos como uma entrada da tabela de símbo-

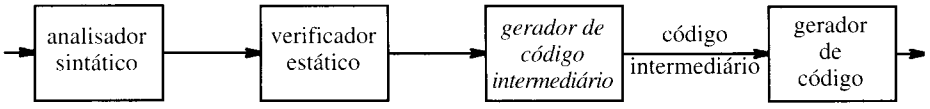
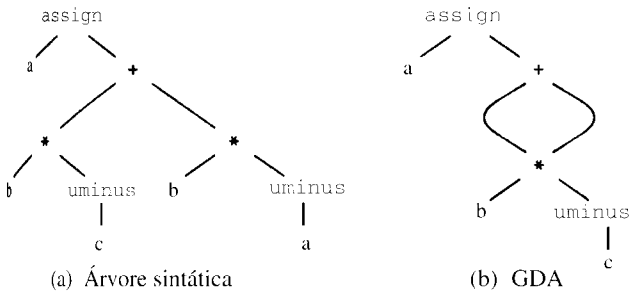


Fig. 8.1. Posição do gerador de código intermediário.



(a) Árvore sintática (b) GDA

Fig. 8.2. Representações gráficas para `a := b * - c + b * - c`.

PRODUÇÃO	REGRA SEMÂNTICA
$S \rightarrow$ **id** $:= E$	$S.nptr$:= *criar_nó* ('`assign`', *criar_folha* (**id, id.***local*), $E.nptr$)
$E \rightarrow E_1 + E_2$	$E.nptr$:= *criar_nó* ('+', $E_1.nptr$, $E_2.nptr$)
$E \rightarrow E_1 * E_2$	$E.nptr$:= *criar_nó* ('*', $E_1.nptr$, $E_2.nptr$)
$E \rightarrow -E_1$	$E.nptr$:= *criar_nó_un* ('`uminus`', $E_1.nptr$)
$E \rightarrow (E_1)$	$E.nptr$:= $E_1.nptr$
$E \rightarrow$ **id**	$E.nptr$:= *criar_folha* (**id, id.** *local*)

Fig. 8.3. Definição dirigida pela sintaxe para produzir árvores sintáticas para comandos de atribuição.

los pode ser encontrada a partir do atributo **id.***nome*, representando o lexema associado àquela ocorrência de **id.** Se o analisador léxico guarda todos os lexemas num único *array* de caracteres, o atributo *nome* poderia ser o índice para o primeiro caractere do lexema.

Duas representações da árvore sintática na Fig. 8.2(a) aparecem na Fig. 8.4. Cada nó é representado como um registro com um campo para seu operador e campos adicionais para os apontadores de seus filhos. Na Fig. 8.4(b), os nós são alocados a partir de um *array* de registros e o índice ou posição do nó serve como o apontador para o nó. Todos os nós na árvore sintática podem ser visitados seguindo-se os apontadores, começando-se da raiz à posição 10.

Código de Três Endereços

O código de três endereços é uma seqüência de enunciados da forma geral

$$x := y \ op \ z$$

onde `x`, `y` e `z` são nomes, constantes ou objetos de dados temporários criados pelo compilador; *op* está no lugar de qualquer operador, tal como um operador de aritmética de ponto fixo ou flutuante ou um operador lógico sobre dados booleanos. Note-se que não são permitidas expressões aritméticas construídas, na medida em que só há um operador no lado direito de um enunciado. Por conseguinte, uma expressão de linguagem-fonte como `x + y * z` poderia ser traduzida na seqüência

$$t_1 := y * z$$
$$t_2 := x + t_1$$

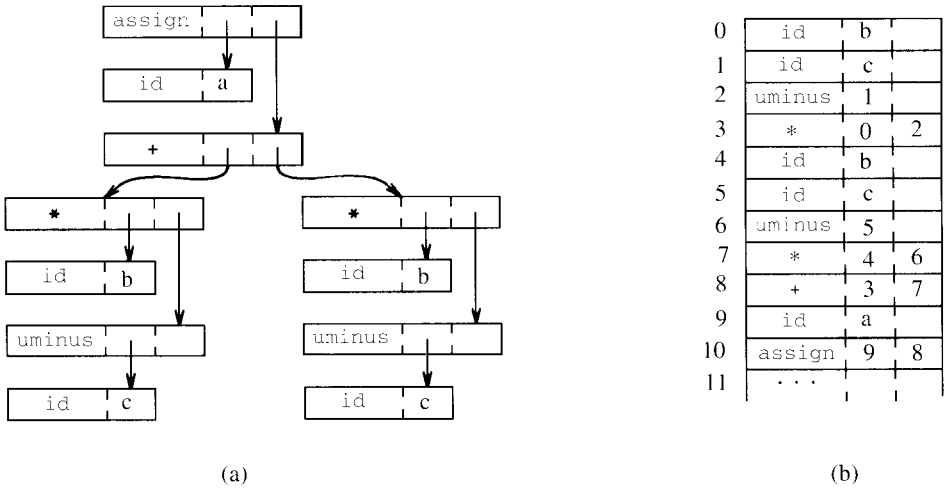
onde t_1 e t_2 são nomes temporários gerados pelo compilador. Esse afastamento das expressões aritméticas complicadas e de enunciados de fluxo de controle aninhados torna o código de três endereços desejável para a geração do código-alvo e a otimização (ver os Capítulos 10 e 12). O uso de nomes para os valores intermediários computados por um programa permite que o código de três endereços seja facilmente rearrumado — diferentemente da notação pós-fixa.

O código de três endereços é uma representação linearizada de uma árvore sintática ou de um GDA, no qual os nomes explícitos correspondem aos nós interiores do grafo. A árvore sintática e o GDA na Fig. 8.2 são representados pelas seqüências de código de três endereços na Fig. 8.5. Os nomes de variáveis podem aparecer diretamente nos comandos de três endereços, e, dessa forma, a Fig. 8.5(a) não possui enunciados correspondentes às folhas na Fig. 8.4.

A razão para o termo "código de três endereços" está em que cada comando contém usualmente três endereços, dois para os operandos e um para o resultado. Nas implementações do código de três endereços fornecidas mais adiante nesta seção, um nome definido pelo programador é substituído por um apontador para a entrada da tabela de símbolos daquele nome.

Tipos de Enunciados de Três Endereços

Os enunciados de três endereços são semelhantes ao código de montagem. Os enunciados podem ter rótulos simbólicos e existem enunciados para o fluxo de controle. Um rótulo simbólico representa o índice de um enunciado de três endereços num *array* que abriga o código intermediário. Os índices efetivos podem substituir os rótulos, quer realizando-se uma passagem separada, quer utilizando-se a "retrocorreção", discutida na Seção 8.6.



(a) (b)

Fig. 8.4. Duas representações da árvore sintática da Fig. 8.2(a).

```
t1 := - c
t2 := b * t1
t3 := - c
t4 := b * t3
t5 := t2 + t4
a  := t5
```

(a) Código para a árvore sintática

```
t1 := - c
t2 := b * t1
t5 := t2 + t2
a  := t5
```

(b) Código para o GDA

Fig. 8.5. Código de três endereços correspondente à árvore e ao GDA da Fig. 8.2.

Aqui estão os enunciados de três endereços comuns usados no resto deste livro:

1. Enunciados de atribuição da forma `x := y` *op* `z`, onde *op* é uma operação binária aritmética ou lógica.

2. Instruções de atribuição da forma `x :=` *op* `y`, onde *op* é uma operação unária. Dentre as operações unárias essenciais estão incluídas o menos unário, a negação lógica, os operadores de deslocamento e os de conversão que, por exemplo, convertem um número de ponto fixo num de ponto flutuante.

3. *Enunciados de cópia,* da forma `x := y` onde o valor de `y` é atribuído a `x`.

4. O desvio incondicional `goto L`. O enunciado de três endereços com rótulo `L` é o próximo a ser executado.

5. Os desvios condicionais tais como `if x` *relop* `y goto L`. Esta instrução aplica um operador relacional (<, =, >=, etc) a `x` e `y` e em seguida executa o enunciado com rótulo `L`, se `x` estiver na relação *relop* com `y`. Caso não esteja, é executado o enunciado de três endereços que se segue ao desvio condicional, como na seqüência usual.

6. `param x` e `call p,` *n*, para chamadas de procedimentos, e `return y,` onde `y`, que representa o valor retornado, é opcional. Seus usos típicos são para seqüências de três endereços como a seguinte

```
param x1
param x2
...
param xn
call p, n
```

geradas como parte da chamada de procedimento $p(x_1, x_2, \ldots x_n)$. O inteiro *n*, indicando o número de parâmetros atuais em "`call p,` *n*" não é redundante porque as chamadas podem ser aninhadas. A implementação de chamadas de procedimentos é delineada na Seção 8.7.

7. Atribuições indexadas da forma `x := y[i]` e `x[i] := y`. A primeira delas estabelece `x` com o valor da localização `i` unidades de memória após a localização de `y`. O enunciado `x[i] := y` estabelece o conteúdo da localização `i` unidades para além de `x` com o valor de `y`. Em ambas as instruções, `x`, `y` e `i` se referem a objetos de dados.

8. Atribuições de endereços e de apontadores da forma `x := &y`, `x := *y`, e `*x := y`. A primeira estabelece o valor de `x` como sendo a localização de `y`. Presumivelmente, `y` é um nome, talvez de um temporário, que denota uma expressão com um valor-*l*, tal como `A[i,j]`, e `x` é um nome de apontador ou temporário. Isto é, o valor-*r* de `x` é o valor-*l* (localização) de algum objeto. No enunciado `x := *y`, `y` é presumivelmente um apontador ou um temporário cujo valor-*r* seja uma localização. O valor-*r* de `x` é igualado ao conteúdo daquela localização. Finalmente, `*x := y` estabelece o valor-*r* do objeto apontado por `x` com o valor-*r* de `y`.

A escolha dos operadores permitidos é um tema importante no projeto de uma forma intermediária. O conjunto de operadores precisa ser rico o bastante para implementar as operações da linguagem-fonte. Um pequeno conjunto de operações é mais fácil de implementar numa nova máquina-alvo. Entretanto, um conjunto restrito de instruções pode forçar a vanguarda do compilador a gerar longas seqüências de comandos para algumas operações da linguagem-fonte. O otimizador e o gerador de código podem ter que arcar um trabalho maior, se deve ser gerado um código de melhor qualidade.

Tradução Dirigida pela Sintaxe em Código de Três Endereços

Quando o código de três endereços é gerado, os nomes temporários são construídos para os nós interiores da árvore sintática. O valor do não-terminal E ao lado esquerdo de $E \rightarrow E_1 + E_2$ será computado numa nova variável temporária `t`. Em geral, o código de três endereços para **id** := E consiste em código para avaliar E em alguma variável temporária `t`, seguido pela atribuição **id**.*local* : = `t`. Se uma expressão se constituir em um único identificador, digamos `y`, o próprio `y` abrigará o valor da expressão. Para o momento, criamos um novo nome a cada vez que uma variável temporária for necessária. As técnicas para a reutilização das variáveis temporárias são fornecidas na Seção 8.3.

A definição S-atribuída na Fig. 8.6 gera código de três endereços para enunciados de atribuição. Dada a entrada `a := b* - c + b* - c`, a definição produz o código na Fig. 8.5(a). O atributo sintetizado *S.código* representa o código de três endereços para a atribuição *S*. O não-terminal E possui dois atributos:

1. *E. local,* o nome que irá abrigar o valor de E e

2. *E. código,* a seqüência de enunciados de três endereços avaliando E.

A função *novo_temporário* retorna uma seqüência de nomes distintos $t_1, t_2, \ldots$ em resposta às sucessivas chamadas.

Por uma questão de conveniência, usamos a notação *gerar* (*x* ':=' *y* '+' *z*), na Fig. 8.6, para representar o enunciado de três endereços *x* := *y* + *z*. As expressões figurando em lugar de variáveis, como *x*, *y* e *z* são avaliadas quando passadas a *gerar,* e os operadores entre apóstrofos, como '+', são considerados literalmente. Na prática, os enunciados de três endereços poderiam ser enviados para um arquivo de saída, ao invés de serem construídos sobre os atributos *código*.

Os enunciados de fluxo de controle podem ser adicionados à linguagem de atribuições da Fig. 8.6 através de produções e regras semânticas como aquelas para os enunciados *while* na Fig. 8.7. Na figura, o código para $S \rightarrow$ **while** E **do** S_1 é gerado usando-se os novos atributos *S.início* e *S.saída* para marcar o primeiro enunciado no código para E e o enunciado que se segue ao código para *S*, respectivamente. Esses atributos representam os rótulos criados pela função *novo_rótulo,* que retorna um novo rótulo a cada vez que for chamada. Note-se que *S.saída* se torna o rótulo do enunciado que vem após o código para o enunciado *while*. Assumimos que uma expressão diferente de zero signifique verdadeiro; isto é, quando o valor de E se torna zero, o controle deixa o enunciado *while*.

As expressões que governam o fluxo de controle podem em geral ser expressões booleanas contendo operadores relacionais e lógicos. As regras semânticas para os enunciados *while* na Seção 8.6 diferem daquelas da Fig. 8.7 de forma a permitir o fluxo de controle dentro de expressões booleanas.

A notação pós-fixa pode ser obtida pela adaptação das regras semânticas na Fig. 8.6 (ou veja a Fig. 2.5). A notação pós-fixa para um iden-

PRODUÇÃO	REGRAS SEMÂNTICAS
$S \rightarrow$ **id** := E	*S.código* := *E.código* ‖ *gerar* (**id**.*local* ':=' *E.local*)
$E \rightarrow E_1 + E_2$	*E.local* := *novo_temporário;* *E.código* := E_1.*código* ‖ E_2.*código* ‖ *gerar* (*E.local* ':=' E_1.*local* '+' E_2.*local*)
$E \rightarrow E_1 * E_2$	*E.local* := *novo_temporário;* *E.código* := E_1.*código* ‖ E_2.*código* ‖ *gerar* (*E.local* ':=' E_1.*local* '*' E_2.*local*)
$E \rightarrow - E_1$	*E.local* := *novo_temporário;* *E.código* := E_1.*código* ‖ *gerar* (*E.local* ':= ' 'uminus' E_1.*local*)
$E \rightarrow (E_1)$	*E.local* := E_1.*local;* *E.código* := E_1.*código*
E $\rightarrow$ **id**	*E.local* := **id**.*local;* *E.código* := ''

Fig. 8.6. Definição dirigida pela sintaxe para produzir código de três endereços para atribuições.

tificador é o próprio identificador. As regras para as outras produções concatenam somente o operador após o código para os operandos. Por exemplo, associada à produção $E \rightarrow - E_1$ está a regra semântica

E.código := E_1.*código* ‖ 'uminus'

Em geral, a forma intermediária produzida pelas traduções dirigidas pela sintaxe neste capítulo podem ser modificadas realizando-se modificações similares às regras semânticas.

Implementações de Enunciados de Três Endereços

Um enunciado de três endereços é uma forma abstrata de código intermediário. Num compilador, esses enunciados podem ser implementados como registros com campos para o operador e os operandos. Três dessas representações são as quádruplas, as triplas e as triplas indiretas.

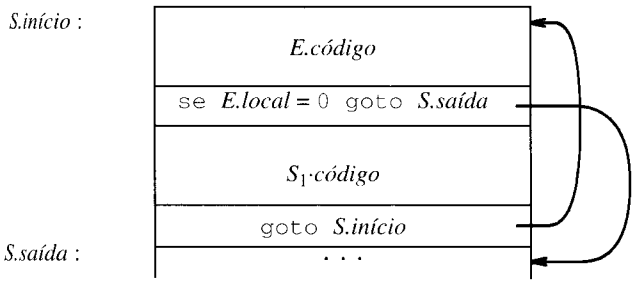
Quádruplas

Uma quádrupla é uma estrutura de registro com quatro campos, os quais chamaremos de *op*, *arg*1, *arg*2 e *resultado*. O campo *op* contém um código interno para o operador. O enunciado de três endereços `x := y op z` é representado colocando-se `y` em *arg*1, `z` em *arg*2 e `x` em *resultado*. Os enunciados com operadores unários, como `x := -y` ou `x := y` não usam *arg*2. Os operadores como `param` não usam nem *arg*2 nem *resultado*. Os desvios condicionais e incondicionais colocam o rótulo-alvo em *resultado*. As quádruplas na Fig. 8.8(a) são para a atribuição `a := b* - c + b * - c`. São obtidas a partir do código de três endereços da Fig. 8.5(a).

Os conteúdos dos campos *arg*1, *arg*2 e *resultado* são normalmente apontadores para as entradas, na tabela de símbolos, dos nomes representados por esses campos. Assim sendo, os nomes dos temporários precisam ser introduzidos na tabela de símbolos, à medida que forem criados.

Triplas

Para se evitar a instalação de nomes de temporários na tabela de símbolos, poderíamos nos referir a um valor temporário pela posição do enunciado que o computa. Se assim o fizermos, os enunciados de três endereços podem ser representados por registros com somente três campos: *op*, *arg*1 e *arg*2, como na Fig. 8.8(b). Os campos *arg*1 e *arg*2,



PRODUÇÃO	REGRAS SEMÂNTICAS
$S \rightarrow$ **while** E **do** S_1	*S.início* := *novo_rótulo;* *S.depois* := *novo_rótulo;* *S.código* := *gerar* (*S. início* ':') ‖ *E.código* ‖ *gerar* ('if' *E.local* '=' '0' 'goto' *S.saída*) ‖ S_1.*código* ‖ *gerar*('goto' *S.início*) ‖ *gerar*(*S.saída* ':')

Fig. 8.7. Regras semânticas para gerar código para um enunciado *while*.

	op	*arg* 1	*arg* 2	*resultado*
(0)	uminus	c		t_1
(1)	*	b	t_1	t_2
(2)	uminus	c		t_3
(3)	*	b	t_3	t_4
(4)	+	t_2	t_4	t_5
(5)	:=	t_5		a

(a) Quádruplas

	op	*arg* 1	*arg* 2
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	assign	a	(4)

(b) Triplas

Fig. 8.8. Representações em quádruplas e triplas de enunciados de três endereços.

	op	*arg* 1	*arg* 2
(0)	`[]=`	`x`	`i`
(1)	`assign`	(0)	`y`

(a) `x[i] := y`

	op	*arg* 1	*arg* 2
(0)	`=[]`	`y`	`i`
(1)	`assign`	`x`	(0)

(b) `x := y[i]`

Fig. 8.9. Mais representações de triplas.

na qualidade de argumentos de *op*, ou são apontadores para a tabela de símbolos (para nomes definidos pelo programador ou constantes) ou apontadores para estruturas de tripla (para valores temporários). Uma vez que três campos são usados, esse formato de código intermediário é conhecido como triplas.[1] Exceto pelo tratamento dos nomes definidos pelo programador, as triplas correspondem à representação da árvore sintática ou GDA através de um *array* de nós, como na Fig. 8.4.

Os números entre parênteses representam apontadores para a estrutura de triplas, enquanto os apontadores para a tabela de símbolos são representados pelos próprios nomes. Na prática, as informações necessitadas para interpretar os diferentes tipos de entradas nos campos *arg*1 e *arg*2 podem ser codificadas no campo *op* ou em alguns outros campos adicionais. As triplas da Fig. 8.8(b) correspondem às quádruplas da Fig. 8.8(a). Note-se que o enunciado de cópia `a := t5` é codificado na representação de triplas colocando-se `a` no campo *arg*1 e utilizando-se o operador de atribuição `assign`.

Uma operação ternária, como `x[i] := y`, requer duas entradas na estrutura de triplas, como mostrado na Fig. 8.9(a), enquanto `x := y[i]` é naturalmente representada como duas operações na Fig. 8.9(b).

Triplas Indiretas

Outra implementação do código de três endereços que tem sido considerada é a que lista apontadores para triplas, ao invés das próprias triplas. Essa representação é naturalmente chamada de triplas indiretas.

Por exemplo, vamos usar o *array comando* para listar apontadores para as triplas na ordem desejada. As triplas, então, podem ser representadas como na Fig. 8.10.

Comparação das Representações: O Uso da Indireção

A diferença entre as triplas e as quádruplas pode ser considerada como uma questão sobre a quantidade de indireção que está presente na representação. Quando, em última análise, produzimos o código-alvo, cada nome, de temporário ou definido pelo programador, terá atribuída a si alguma localização de memória em tempo de execução. Essa localização será colocada na tabela de símbolos para o dado. Usando-se a notação de quádruplas, um enunciado de três endereços definindo ou usando um temporário pode ter acesso imediatamente à localização do mesmo através da tabela de símbolos.

Um benefício mais importante das quádruplas aparece nos compiladores otimizantes, onde os comandos são freqüentemente deslocados de lugar. Usando-se a notação de quádruplas, a tabela de símbolos interpõe um grau extra de indireção entre a computação de um valor e seu uso. Se movermos um enunciado que compute `x`, o enunciado que use `x` não irá requerer qualquer mudança. Entretanto, na notação de triplas, mover um enunciado que defina um valor temporário irá nos exigir que mudemos todas as referências àquele enunciado nos *arrays* *arg*1 e *arg*2. Esse problema torna as triplas difíceis de se usar num compilador otimizante.

	enunciado
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

	op	*arg* 1	*arg* 2
(14)	`uminus`	`c`	
(15)	`*`	`b`	(14)
(16)	`uminus`	`c`	
(17)	`*`	`b`	(16)
(18)	`+`	(15)	(17)
(19)	`assign`	`a`	(18)

Fig. 8.10. Representação em triplas indiretas de enunciados de três endereços.

As triplas indiretas não apresentam tal problema. Um enunciado pode ser removido pelo reordenamento da lista *comando*. Como os apontadores para valores temporários se referem aos *arrays* *op*-*arg*1-*arg*2 que não são mudados, nenhum desses apontadores precisa ser modificado. Por conseguinte, as triplas indiretas se parecem muito com as quádruplas, na medida em que a utilidade é considerada. As duas notações requerem em torno da mesma quantidade de espaço e são igualmente eficientes na reordenação do código. Como ocorre com as triplas ordinárias, a alocação de memória para esses temporários que dela necessitam precisa ser postergada até a fase de geração de código. Entretanto, as triplas indiretas podem ocupar algum espaço extra quando comparadas com as quádruplas se o mesmo valor temporário for usado mais de uma vez. A razão está em que duas ou mais entradas no *array* *comando* podem apontar para a mesma linha na estrutura *op*-*arg*1-*arg*2. Por exemplo, as linhas (14) e (16) da Fig. 8.10 poderiam ser combinadas e, em seguida, poderíamos combinar (15) e (17).

8.2 DECLARAÇÕES

À medida que a seqüência de declarações num procedimento ou bloco é examinada, podemos dispor da memória para atribuí-la aos nomes locais ao procedimento. Para cada nome local, podemos criar uma entrada na tabela de símbolos, com informações tais como o tipo e o endereço relativo da memória para aquele nome. O endereço relativo consiste em um deslocamento a partir da base da área estática de dados ou do campo para os dados locais no registro de ativação.

Quando os módulos de vanguarda do compilador geram endereços, já se pode estar com uma máquina-alvo em mente. Suponhamos que os endereços de inteiros consecutivos difiram de 4, numa máquina com endereçamento em nível de *byte*. Os cômputos de endereço gerados pela vanguarda podem, por conseguinte, incluir multiplicações por 4. O conjunto de instruções na máquina-alvo pode também favorecer certas disposições para os objetos de dados e, por conseguinte, de seus endereços. Ignoramos aqui o alinhamento de dados; o Exemplo 7.3 mostra como os objetos de dados são alinhados por dois compiladores.

Declarações num Procedimento

A sintaxe de linguagem como C, Pascal e Fortran, permite que todas as declarações num único procedimento possam ser processadas como um único grupo. Nesse caso, uma variável global, digamos *deslocamento*, pode controlar o próximo endereço relativo disponível.

No esquema de tradução da Fig. 8.11, o não-terminal *P* gera uma seqüência de declarações da forma **id** : *T*. Antes da primeira declaração ser

[1]Alguns se referem às triplas como "código de dois endereços", preferindo identificar as quádruplas pelo termo "código de três endereços". Iremos, entretanto, tratar o "código de três endereços" como uma noção abstrata com várias implementações, sendo as triplas e as quádruplas as principais.

$P \rightarrow$	{ *deslocamento* := 0 }
D	
$D \rightarrow D \; ; \; D$	
$D \rightarrow$ **id** : T	{ *instalar* (**id**·*nome*, *T.tipo*, *deslocamento*); *deslocamento* := *deslocamento* + *T.largura* }
$T \rightarrow$ **integer**	{ *T.tipo* := *inteiro*; *T.largura* := 4 }
$T \rightarrow$ **real**	{ *T.tipo* := *real*; *T. largura* := 8 }
$T \rightarrow$ **array** [**num**] **of** T_1	{ *T.tipo* := *array* (**num**·*val*, T_1.*tipo*); *T.largura* := **num**·*val* × T_1.*largura* }
$T \rightarrow \uparrow T_1$	{ *T.tipo* := *apontador* (T_1.*tipo*); *T.largura* := 4 }

Fig. 8.11. Computando os tipos e endereços relativos de nomes declarados.

considerada, *deslocamento* é zerado. À medida que cada nome é enxergado, o mesmo é introduzido na tabela de símbolos, com um deslocamento igual ao valor corrente de *deslocamento*, o qual é, em seguida, incrementado pelo tamanho do objeto de dados denotado por aquele nome.

O procedimento *instalar (nome, tipo, deslocamento)* cria uma nova entrada para *nome*, confere ao mesmo o tipo *tipo* e o endereço relativo *deslocamento* em sua área de dados. Usamos os atributos sintetizados *tipo* e *largura* para o não-terminal T para indicar o tipo e a largura ou o número de unidades de memória ocupadas pelos objetos de dados daquele tipo. O atributo *tipo* representa uma expressão de tipo, construída a partir dos tipos básicos *inteiro* e *real*, através da aplicação dos construtores de tipos *apontador* e *array*, como na Seção 6.1. Se as expressões de tipo forem representadas por grafos, o atributo *tipo* poderia ser um apontador para o nó que representa a expressão de tipo.

Na Fig. 8.11, os inteiros têm largura 4 e os reais, 8. A largura de um *array* é obtida pela multiplicação da largura de cada elemento pelo número de elementos dentro do *array*.[2] A largura de cada apontador é assumida igual a 4. Em Pascal e C, um apontador pode ser visto antes que conheçamos o tipo do objeto apontado pelo mesmo (veja a discussão de tipos recursivos na Seção 6.3). A alocação de memória para tais tipos é mais simples se todos os apontadores tiverem a mesma largura.

A inicialização de *deslocamento* no esquema de tradução da Fig. 8.11 é mais evidente se a primeira produção aparecer em uma linha como

$$P \rightarrow \{ \textit{deslocamento} := 0 \} \; D \qquad (8.2)$$

Os não-terminais que geram ϵ, chamados na Seção 5.6 de não-terminais marcadores, podem ser usados para reescrever as produções, de tal forma que todas as ações apareçam nas extremidades dos lados direitos. Utilizando-se um não-terminal marcador M, (8.2) pode ser reescrita como:

$$\begin{array}{ll} P \rightarrow M \; D & \\ M \rightarrow \epsilon & \{ \textit{deslocamento} := 0 \} \end{array}$$

Controlando o Escopo das Informações

Numa linguagem com procedimentos aninhados, os nomes locais a cada procedimento podem ter atribuídos a si endereços relativos, usando-se o enfoque da Fig. 8.11. Quando um procedimento aninhado é enxergado, o processamento das declarações do procedimento envolvente é suspenso temporariamente. Essa abordagem será ilustrada através da adição de regras semânticas à linguagem seguinte.

$$\begin{array}{l} P \rightarrow D \\ D \rightarrow D \; ; \; D \mid \textbf{id} : T \mid \textbf{proc id} \; ; \; D \; ; \; S \end{array} \qquad (8.3)$$

As produções para os não-terminais S (para os enunciados) e T (para os tipos) não são mostradas porque desejamos focalizar as declarações. O não-terminal T possui os atributos sintetizados *tipo* e *largura*, como no esquema de tradução da Fig. 8.11.

Por uma questão de simplicidade, suponhamos que exista uma tabela de símbolos separada para cada procedimento na linguagem (8.3). Uma possível implementação de uma tabela de símbolos seria uma lista ligada de entradas para os nomes. Implementações mais inteligentes podem substituir esta última, se for desejado.

Uma nova tabela de símbolos é criada quando uma declaração de procedimento $D \rightarrow$ **proc id** D_1 ; S é enxergada, sendo as entradas para as declarações em D_1 colocadas nesta nova tabela. A nova tabela aponta de volta para a tabela de símbolos do procedimento envolvente. A única mudança para o tratamento das declarações de variáveis na Fig. 8.11 é que é dito ao procedimento *instalar* em que tabela de símbolos deve ser introduzida a entrada.

Por exemplo, na Fig. 8.12, são mostradas as tabelas de símbolos para cinco procedimentos. A estrutura de aninhamento dos procedimentos pode ser deduzida a partir dos elos entre as tabelas de símbolos; o programa está na Fig. 7.22. As tabelas de símbolos para os procedimentos `readarray`, `exchange` e `quicksort` apontam de volta para aquela que contém o procedimento `sort`, que se constitui em todo o programa. Como `partition` é declarada dentro de `quicksort`, sua tabela aponta para a de `quicksort`.

As regras semânticas são definidas em termos das seguintes operações:

1. *criar_tabela (anterior)* cria uma nova tabela de símbolos retornando um apontador para a mesma. O argumento *anterior* aponta para a tabela anteriormente criada, presumivelmente aquela para o procedimento envolvente. O apontador *anterior* é colocado num cabeçalho para a nova tabela, juntamente com informações adicionais, tais como a profundidade de aninhamento de um procedimento. Podemos também numerar os procedimentos de acordo com a ordem em que sejam declarados e manter esse número no cabeçalho.

[2] Para os *arrays* cujo limite inferior não seja 0, o cômputo de endereços para os elementos de *array* é simplificado se somente o deslocamento introduzido na tabela de símbolos for ajustado, como discutido na Seção 8.3.

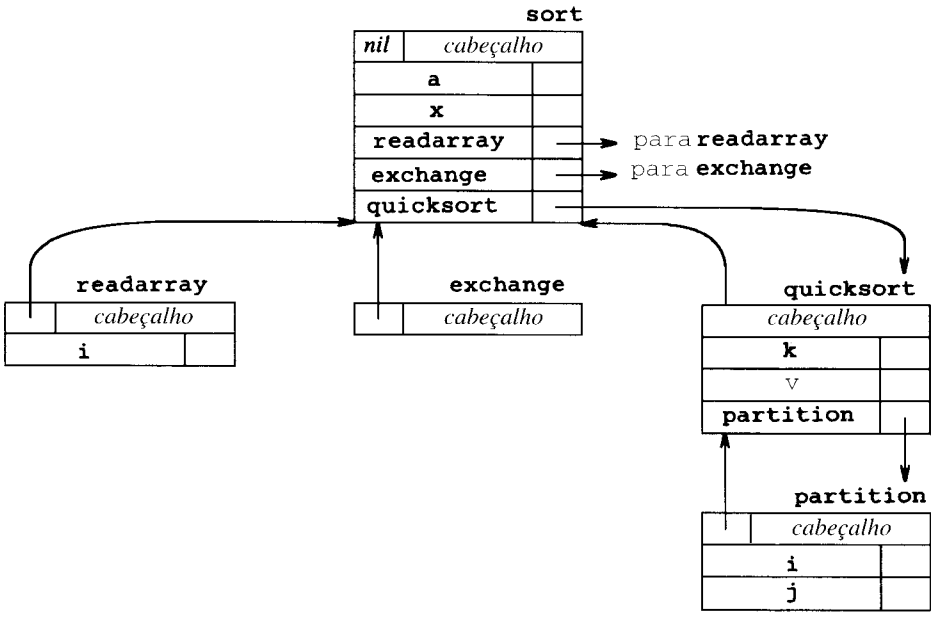


Fig. 8.12. Tabelas de símbolos para procedimentos aninhados.

2. *instalar (tabela, nome, tipo, deslocamento)* cria uma nova entrada para o nome *nome* na tabela de símbolos apontada por *tabela*. De novo, *instalar* coloca o tipo *tipo* e o endereço relativo *deslocamento* nos campos da entrada criada.

3. *registrar_largura (tabela, largura)* registra a largura acumulada de todas as entradas de *tabela* no cabeçalho associado a esta tabela de símbolos.

4. *instalar_proc(tabela, nome, nova_tabela)* cria uma nova entrada para o nome de procedimento *nome* na tabela de símbolos apontada por *tabela*. O argumento *nova_tabela* aponta para a tabela de símbolos do procedimento *nome*.

O esquema de tradução na Fig. 8.13 mostra como os dados podem ser dispostos em uma única passagem, usando-se a pilha *ptr_tab* para guardar os apontadores para as tabelas de símbolos dos procedimentos envolventes. Com as tabelas de símbolos da Fig. 8.12, *ptr_tab* irá conter apontadores para as tabelas de `sort`, `quicksort` e de `partition` quando as declarações dentro de `partition` forem consideradas. O apontador para a entrada corrente da tabela de símbolos está ao topo. A outra pilha *deslocamento* é a generalização natural para os procedimentos aninhados com o atributo *deslocamento* à Fig. 8.11. O elemento de topo de *deslocamento* é o próximo endereço relativo disponível no procedimento corrente.

Todas as ações semânticas nas subárvores para B e C em

$$A \rightarrow B\ C\ \{\ ação_A\ \}$$

são realizadas antes que $ação_A$, ao final da produção, o seja. Por conseguinte, a ação associada ao marcador M na Fig. 8.13 é a primeira a ser realizada.

A ação para o não-terminal M inicializa a pilha *ptr_tab* com a tabela de símbolos para o escopo mais externo, criada pela operação *criar_tabela(nil)*. A ação também empilha o endereço relativo 0 sobre a pilha *deslocamento*. O não-terminal N desempenha um papel similar quando uma declaração de procedimento aparece. Sua ação usa a operação *criar_tabela (topo(tab_ptr))* para criar uma nova tabela de símbolos. Aqui, o argumento *topo(ptr_tab)* dá o escopo envolvente da nova tabela. Um apontador para a nova tabela é empilhado acima daquele para o escopo envolvente. De novo, 0 é empilhado sobre a pilha *deslocamento*.

Para cada declaração de variável **id** : T, uma entrada é criada para **id** na tabela corrente de símbolos. Esta declaração deixa a pilha *ptr_tab*

$P \rightarrow M\ D$	{ *registrar_largura*(*topo*(*ptr_tab*), *topo*(*deslocamento*)); *desempilhar*(*ptr_tab*); *desempilhar*(*deslocamento*) }
$M \rightarrow \epsilon$	{ *t* := *criar_tabela*(*nil*); *empilhar*(*t*, *ptr_tab*); *empilhar*(0, *deslocamento*) }
$D \rightarrow D_1; D_2$	
$D \rightarrow$ **proc id** : $N\ D_1; S$	{ *t* := *topo*(*ptr_tab*); *registrar_largura*(*t*, *topo*(*deslocamento*)); *desempilhar*(*ptr_tab*); *desempilhar*(*deslocamento*); *instalar_proc*(*topo*(*ptr_tab*), **id**.*nome*, *t*) }
$D \rightarrow$ **id** : T	{ *instalar*(*topo*(*ptr_tab*), **id**. *nome*, *T.tipo*, *topo*(*deslocamento*)); *topo*(*deslocamento*) := *topo*(*deslocamento*) + *T.largura* }
$N \rightarrow \epsilon$	{ *t* := *criar_tabela*(*topo*(*ptr_tab*)); *empilhar*(*t*, *ptr_tab*); *empilhar* (0, *deslocamento*) }

Fig. 8.13. Processando declarações em procedimentos aninhados.

inalterada; o topo da pilha *deslocamento* é incrementado por *T.largura*. Quando a ação ao lado direito de $D \rightarrow$ **proc id** ; $N\ D_1$; S ocorre, a largura de todas as declarações geradas por D_1 está ao topo da pilha *deslocamento;* é registrada usando-se *registrar_largura*. As pilhas *ptr_tab* e *deslocamento* têm, então, os topos removidos e voltamos a examinar as declarações do procedimento envolvente. A esse ponto, o nome do procedimento é introduzido na tabela de símbolos de seu procedimento envolvente.

Nomes de Campos em Registros

A seguinte produção permite que o não-terminal T gere registros adicionalmente aos tipos básicos, apontadores e *arrays*:

$$T \rightarrow \textbf{record}\ D\ \textbf{end}$$

As ações no esquema de tradução da Fig. 8.14 enfatizam a similaridade entre a disposição de dados dos registros como construção de linguagem e dos registros de ativação. Como as definições de procedimentos não afetam os cômputos de larguras na Fig. 8.13, fazemos "vista grossa" para o fato da produção acima também permitir que definições de procedimentos apareçam dentro de registros.

Após a palavra-chave **record** ter sido enxergada, a ação associada ao marcador L cria uma nova tabela de símbolos para os nomes de campos. Um apontador para esta tabela de símbolos é empilhado em *ptr_tab* e o endereço relativo 0 é empilhado em *deslocamento*. Por conseguinte, a ação para $D \rightarrow$ **id** : T na Fig. 8.13 entra com informações sobre o nome de campo **id** na tabela de símbolos para o registro. Sobretudo, o topo da pilha *deslocamento* irá abrigar a largura de todos os objetos de dados dentro do registro após os campos terem sido examinados. A ação seguindo-se ao **end** na Fig. 8.14 retorna essa largura como o atributo sintetizado *T.largura*. O tipo *T.tipo* é obtido através da aplicação do construtor *registro* ao apontador da tabela de símbolos para esse registro. Esse apontador será usado na próxima seção para recuperar os nomes, tipos e larguras dos campos no registro de *T.tipo*.

8.3 ENUNCIADOS DE ATRIBUIÇÃO

Nesta seção, as expressões podem ser do tipo inteiro, real, *array* e registro. Como parte da tradução das atribuições em código de três endereços, mostramos como os nomes podem ser pesquisados na tabela de símbolos e como os elementos de *arrays* e registros podem receber acesso.

Nomes na Tabela de Símbolos

Na Seção 8.1, formamos enunciados de três endereços usando os próprios nomes, com a compreensão de que esses nomes estavam em lugar dos apontadores para as respectivas entradas na tabela de símbolos. O esquema de tradução na Fig. 8.15 mostra como tais entradas da tabela de símbolos podem ser encontradas. O lexema para o nome representado por **id** é fornecido pelo atributo **id**.*nome*. A operação *procurar* (**id**.*nome*) verifica se existe uma entrada para esta ocorrência de nome na tabela de símbolos. Se houver, um apontador para a entrada para esta ocorrência é retornado; em caso contrário, *procurar* retorna *nil* para indicar que nenhuma entrada foi encontrada.

As ações semânticas na Fig. 8.15 usam o procedimento *emitir* para emitir enunciados de três endereços num arquivo de saída, ao invés de construir os atributos *código* para os não-terminais, como na Fig. 8.6. A partir da Seção 2.3, a tradução pode ser feita através da emissão para um arquivo de saída, se os atributos *código* para os não-terminais nos lados esquerdos das produções forem formados pela concatenação dos atributos *código* dos não-terminais ao lado direito, na mesma ordem em que os não-terminais figuram ao lado direito, talvez com algumas cadeias adicionais entremeadas.

Pela reinterpretação da operação *procurar* na Fig. 8.15, o esquema de tradução pode ser usado mesmo que a regra do escopo aninhado mais internamente se aplique a nomes não-locais, como em Pascal. Para uma situação mais concreta, suponhamos que o contexto no qual uma atribuição apareça seja dado pela seguinte gramática.

$$\begin{array}{l} P \rightarrow M\ D \\ M \rightarrow \epsilon \\ D \rightarrow D\ ;\ D \mid \textbf{id}\ :\ T \mid \textbf{proc id}\ ;\ N\ D\ ;\ S \\ N \rightarrow \epsilon \end{array}$$

O não-terminal P se torna o novo símbolo de partida quando essas produções são adicionadas àquelas da Fig. 8.15.

Para cada procedimento gerado por esta gramática, o esquema de tradução na Fig. 8.13 estabelece uma tabela de símbolos separada. Cada uma dessas tabelas de símbolos possui um cabeçalho contendo um apontador para a tabela do procedimento envolvente (ver a Fig. 8.12 para um exemplo). Quando um enunciado que forma o corpo do procedimento é examinado, um apontador para a tabela de símbolos para o procedimento aparece ao topo da pilha *ptr_tab*. Esse apontador é empilhado por ações associadas ao não-terminal marcador N, ao lado direito de $D \rightarrow$ **proc id** ; $N\ D_1$; S.

Sejam as produções para o não-terminal S aquelas na Fig. 8.15. Os nomes numa atribuição gerada por S precisam ter sido declarados no procedimento em que S aparece ou em algum procedimento envolvente. Quando aplicada a (um) *nome*, o procedimento modificado *procurar* verifica primeiro se *nome* figura na tabela de símbolos corrente, acessível através de *topo (ptr_tab)*. Caso não figure, *procurar* usa um apontador no cabeçalho da tabela atual para encontrar a tabela de símbolos para o procedimento envolvente e procura o nome por lá. Se o nome não puder ser encontrado em quaisquer desses escopos, *procurar* devolve *nil*.

Por exemplo, suponhamos que as tabelas de símbolos sejam como na Fig. 8.12 e que uma atribuição no corpo do procedimento `partition` esteja sendo examinada. A operação *procurar* `(i)` irá encontrar uma entrada na tabela de símbolos para `partition`. Uma

$T \rightarrow$ **record** $L\ D$ **end**	{ *t.tipo* := *registro*(*topo*(*ptr_tab*)); *T.largura* := *topo*(*deslocamento*); *desempilhar*(*ptr_tab*); *desempilhar*(*deslocamento*) }
$L \rightarrow \epsilon$	{ *t* := *criar_tabela*(*nil*); *empilhar*(*t*, *ptr_tab*); *empilhar*(0, *deslocamento*) }

Fig. 8.14. Estabelecendo uma tabela de símbolos para nomes de campos de um registro.

$S \rightarrow$ **id** := E	{ *p* := *procurar*(**id**.*nome*); **se** *p* ≠ *nil* **então** *emitir*(*p* ':=' *E.local*) **senão** *erro* }
$E \rightarrow E_1 + E_2$	{ *E.local* := *novo-temporário;* *emitir*(*E.local* ':=' E_1.*local* '+' E_2.*local*) }
$E \rightarrow E_1 * E_2$	{ *E.local* := *novo-temporário;* *emitir*(*E.local* ':=' E_1.*local* '*' E_2.*local*) }
$E \rightarrow -E_1$	{ *E.local* := *novo_temporário;* *emitir*(*E.local* ':=' 'uminus' E_1.*local*) }
$E \rightarrow (E_1)$	{ *E.local* := E_1.*local* }
$E \rightarrow$ **id**	{ *p* := *procurar*(**id**.*nome*); **se** *p* ≠ *nil* **então** *E.local* := *p* **senão** *erro*; }

Fig. 8.15. Esquema de tradução para produzir código de três endereços para as atribuições.

vez que `v` não está nesta tabela de símbolos, *procurar* `(v)` irá usar o apontador no cabeçalho desta tabela de símbolos para continuar a busca na tabela de símbolos do procedimento envolvente `quicksort`.

Reusando os Nomes Temporários

Temos prosseguido assumindo que *novo_temporário* gere um novo nome temporário a cada vez que um temporário seja necessitado. É útil, especialmente em compiladores otimizantes, se criar realmente um nome distinto a cada vez que *novo_temporário* for chamado. O Capítulo 10 fornece a justificativa para se agir assim. No entanto, os temporários usados para guardar os valores intermediários em cômputos de expressões tendem a entulhar a tabela de símbolos , e é necessário reservar espaço para abrigar seus valores.

Os temporários podem ser reusados através da modificação de *novo_temporário*. Um enfoque alternativo de se compactar temporários distintos na mesma localização de memória durante a geração de código é explorado no próximo capítulo.

O grosso dos temporários que denotam dados é gerado durante a tradução dirigida pela sintaxe de expressões, através de regras como aquelas da Fig. 8.15. O código gerado pelas regras para $E \rightarrow E_1 + E_2$ possui a forma geral:

avaliar E_1 em t_1
avaliar E_2 em t_2
$t := t_1 + t_2$

A partir das regras para o atributo sintetizado *E.local* verifica-se que t_1 e t_2 não são usadas em qualquer outro lugar do programa. Os tempos de vida desses temporários estão aninhados como pares de parênteses balanceados. De fato, os tempos de vida de todos os temporários usados na avaliação de E_2 estão contidos nos tempos de vida de t_1. É, por conseguinte, possível modificar *novo_temporário* de forma a que usasse, como se fora uma pilha, um pequeno *array* numa área de dados do procedimento, para abrigar os temporários.

Vamos assumir, simplesmente, que estamos lidando somente com inteiros. Mantemos um contador *c*, inicializado em zero. Sempre que um nome temporário for usado como um operando, decrementar *c* de 1. Sempre que um novo nome de temporário for gerado, usar `$c` e incrementar *c* de 1. Note-se que a "pilha" de temporários não sofre empilhamento ou desempilhamento em tempo de execução, apesar de armazenamentos e cargas de valores temporários serem programados pelo compilador para acontecer ao "topo".

Exemplo 8.1. Consideremos a atribuição

```
x := a * b + c * d - e * f
```

A Fig. 8.16 mostra a seqüência de enunciados de três endereços que seria gerada pelas regras semânticas na Fig. 8.15, se *novo_temporário* fosse modificado. A figura também contém uma indicação do valor corrente de *c* após a geração de cada enunciado. Note-se que quando computamos `$0 - $1`, *c* é decrementado para zero, e dessa forma `$0` está disponível , abrigando o último resultado. □

ENUNCIADO	VALOR de *c*
	0
`$0 := a * b`	1
`$1 := c * d`	2
`$0 := $0 + $1`	1
`$1 := e * f`	2
`$0 := $0 - $1`	1
`x := $0`	0

Fig. 8.16. Código de três endereços com temporários empilhados.

Os temporários que podem ser atribuídos e/ou usados mais de uma vez, por exemplo, numa atribuição condicional, não podem ter os nomes associados numa base último a entrar primeiro a sair (LIFO), descrita acima. Como tendem a ser raros, tais valores temporários podem ter os nomes atribuídos de forma particular. O mesmo problema de temporários definidos ou usados mais de uma vez ocorre quando realizamos uma otimização de código, tal como combinar subexpressões comuns ou mover uma computação para fora de um laço (ver o Capítulo 10). Uma estratégia razoável é a de criar um novo nome sempre que criarmos uma definição adicional ou uso para um temporário ou mover a sua computação.

Endereçando Elementos de Arrays

Os elementos de um *array* podem receber acesso rapidamente se os elementos forem armazenados num bloco de localizações consecutivas. Se a largura de cada elemento do *array* é *w*, o *i*ésimo elemento do *array* `A` começa na localização

$$base + (i - linf) \times w \tag{8.4}$$

onde *linf* é o limite inferior do intervalo de subscritos e *base* é o endereço relativo da memória alocada para o *array*. Isto é, base é o endereço relativo de `A` [linf].

A expressão (8.4) pode ser primariamente avaliada em tempo de compilação se for reescrita como

$$i \times w + (base - linf \times w)$$

A subexpressão $c = base - linf \times w$ pode ser avaliada quando a declaração do *array* for enxergada. Assumimos que *c* é salvo na entrada da tabela de símbolos para `A`, de forma que o endereço relativo de `A`[*i*] é obtido simplesmente adicionando-se $i \times w$ a *c*.

A pré-computação em tempo de compilação também pode ser aplicada a cálculos de endereços de elementos de *arrays* multidimensionais. Um *array* bidimensional é normalmente armazenado em uma de duas formas, ou por *linha* (linha a linha) ou por *coluna* (coluna a coluna). A Fig. 8.17 mostra a disposição de um *array* `A` 2×3 (a) organizado por linhas (b) por colunas. Fortran usa a organização por coluna; Pascal, por linha, porque `A[i,j]` é equivalente a `A[i][j]` e os elementos do *array* `A[i]` são armazenados consecutivamente.

No caso de um *array* bidimensional armazenado na ordem por linha, o endereço relativo de `A`$[i_1, i_2]$ pode ser calculado pela fórmula

$$base + ((i_1 - linf_1) \times n_2 + i_2 - linf_2) \times w$$

onde $linf_1$ e $linf_2$ são os limites inferiores sob os valores de i_1 e i_2 e n_2 é o número de valores que i_2 pode assumir. Isto é, se $lsup_2$ é o limite superior sobre os valores de i_2, então $n_2 = lsup_2 - linf_2 + 1$. Assumindo que i_1 e i_2 são os únicos valores que não são conhecidos em tempo de compilação, podemos reescrever a expressão acima como

$$((i_1 \times n_2) + i_2) \times w + (base - ((linf_1 \times n_2) + linf_2) \times w) \tag{8.5}$$

O último termo desta expressão pode ser determinado em tempo de compilação.

Podemos generalizar a organização por linha ou por coluna para várias dimensões. A generalização da organização por linha significa armazenar os elementos de tal forma que, à medida que esquadrinhamos um bloco de memória, os subscritos mais à direita parecem variar mais rapidamente, como os números num odômetro. A expressão (8.5) se generaliza para a expressão seguinte, para o endereço relativo de `A`$[i_1, i_2, ..., i_k]$

$$((\ldots ((i_1 n_2 + i_2) n_3 + i_3) \ldots) n_k + i_k) \times w$$
$$+ base - ((\ldots ((linf_1 n_2 + linf_2) n_3 + linf_3) \ldots) n_k + linf_k) \times w \tag{8.6}$$

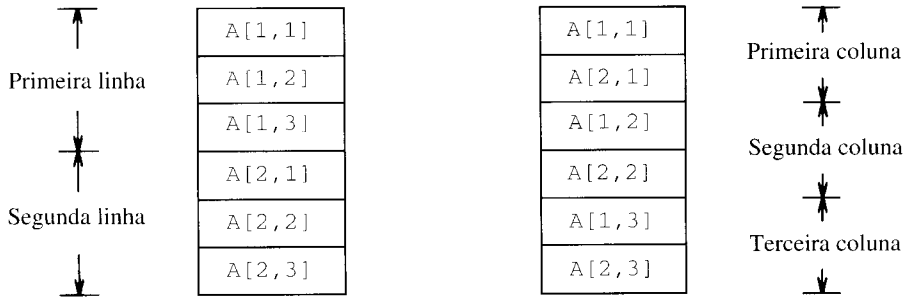


Fig. 8.17. Disposições para um *array* bidimensional.

Como para todo j, $n_j = lsup_j - linf_j + 1$ é assumido fixo, o termo na segunda linha de (8.6) pode ser computado pelo compilador e salvo na entrada da tabela de símbolos para A[3]. A organização por coluna se generaliza num arranjo oposto, no qual os subscritos mais à esquerda variam mais rapidamente.

Algumas linguagens permitem que os comprimentos dos *arrays* sejam especificados dinamicamente, quando o procedimento for chamado em tempo de execução. A alocação de tais *arrays* numa pilha em tempo de execução foi considerada na Seção 7.3. As fórmulas para o acesso aos elementos de tais *arrays* são as mesmas que aquelas para *arrays* de tamanho fixo, mas os limites superior e inferior não são conhecidos em tempo de compilação.

O problema principal na geração de código para referências a *arrays* é o de relacionar o cômputo de (8.6) a uma gramática para referenciar *arrays*. As referências a *arrays* podem ser permitidas nas atribuições se o não-terminal L, como as produções seguintes, for permitido figurar onde **id** aparecer, na Fig. 8.15:

$$L \rightarrow \textbf{id}\ [lista_E] \mid \textbf{id}$$
$$lista_E \rightarrow lista_E,\ E \mid E$$

Com o fito de tornar disponíveis os vários limites dimensionais n_j do *array*, à medida que agruparmos expressões de índices numa *lista_E*, será útil reescrever as produções como

$$L \rightarrow lista_E\ ] \mid \textbf{id}$$
$$lista_E \rightarrow lista_E\ ,\ E \mid \textbf{id}\ [\ E$$

Isto é, o nome do *array* é atrelado à expressão de índice mais à esquerda, ao invés de ficar ligada à *lista_E* quando um L for formado. Essas produções permitem que um apontador para a entrada da tabela de símbolos para o nome do *array* seja passado como o atributo sintetizado *array* de *lista_E*[4].

Também usamos *lista_E.ndim* para registrar o número de dimensões (expressões de índices) na *lista_E*. A função *limite (array, j)* retorna n_j, o número de elementos ao longo da j_ésima dimensão do *array*, cuja entrada da tabela de símbolos é apontada por *array*. Finalmente, *lista_E.local* denota um temporário que abriga um valor computado a partir de expressões de índice em *lista_E*.

Uma *lista_E* que produza os m primeiros índices de uma referência a um *array* k-dimensional $A[i_1, i_2, ..., i_k]$ irá gerar o código de três endereços para computar

$$(\ ...\ ((i_1 n_2 + i_2)n_3 + i_3)\ ...)n_m + i_m \qquad (8.7)$$

usando a fórmula de recorrência

$$e_1 = i_1$$
$$e_m = e_{m-1} \times n_m + i_m \qquad (8.8)$$

Por conseguinte, quando $m = k$, uma multiplicação pela largura w é tudo o que será necessitado para computar o termo da primeira linha de (8.6). Note-se que os i's aqui podem realmente ser valores de expressões e o código para avaliar aquelas expressões será entremeado com o código para computar (8.7).

Um valor-l L terá dois atributos, *L.local* e *L.deslocamento*. No caso em que L for simplesmente um nome simples, *L.local* será um apontador para a entrada da tabela de símbolos para aquele nome, e *L.deslocamento* será **null**, indicando que o valor-l é um nome simples ao invés de uma referência a um *array*. O não-terminal E possui a mesma tradução *E.local*, com o mesmo significado que na Fig. 8.15.

O Esquema de Tradução para o Endereçamento de Elementos de Array

As ações semânticas serão incluídas na gramática:

(1)	S	$\rightarrow$	$L := E$
(2)	E	$\rightarrow$	$E + E$
(3)	E	$\rightarrow$	$(\ E\ )$
(4)	E	$\rightarrow$	L
(5)	L	$\rightarrow$	*lista_E*]
(6)	L	$\rightarrow$	**id**
(7)	*lista_E*	$\rightarrow$	*lista_E, E*
(8)	*lista_E*	$\rightarrow$	**id** [E

Como no caso das expressões sem referências a *arrays*, o código de três endereços em si é produzido pelo procedimento *emitir*, invocado pelas ações semânticas.

Geramos uma atribuição normal se L for um nome simples e uma atribuição indexada na localização denotada por L em caso contrário:

```
(1)  S → L := E   { se L.deslocamento = null então /* L é um
                        identificador simples */
                      emitir (L.local ':=' E.local);
                  senão
                      emitir(L.local ' [ ' L.deslocamento ' ]
                        ':=' E,local)}
```

O código para as expressões aritméticas é exatamente o mesmo que na Fig. 8.15:

```
(2)  E → E1 + E2   { E.local := novo temporário;
                     emitir (E.local ':=' E1.local '+' E2.local)}
```

[3]Em C, um *array* multidimensional é simulado definido-se *arrays* cujos elementos são *arrays*. Por exemplo, suponhamos que seja um *array* de inteiros. A linguagem permite, então, que ambos, x[i] e x[i][j], sejam escritos, sendo as larguras dessas expressões diferentes. No entanto, o limite inferior de um *array* é sempre 0, e o termo à segunda linha de (8.6) se simplifica para *base* em cada caso.

[4]A transformação é similar àquela mencionada ao final da Seção 5.6 para eliminar os atributos herdados. Aqui, igualmente, poderíamos ter resolvido o problema com atributos herdados.

(3) $E \rightarrow (E_1)$ { $E.local := E_1.local$ }

Quando uma referência a um *array* L é reduzida a E, queremos o valor-r de L. Por conseguinte, usamos a indexação para obter o conteúdo da localização *L.local* [*L.deslocamento*]:

(4) $E \rightarrow L$ { **se** *L.deslocamento* = **null então** /* *L* é um identificador simples **id** */
E.local := *L.local*
senão início
E.local := *novo_temporário;*
emitir(E.local ':=' L.local['L.deslocamento']')
fim }

Abaixo, *L.deslocamento* é um novo temporário representando o primeiro termo de (8.6); a função *largura (lista_E.array)* retorna w em (8.6). *L.local* representa o segundo termo de (8.6), retornado pela função *c(lista_E.array)*.

(5) $L \rightarrow lista_E\ ]$ { *L.local := novo_temporário;*
L.deslocamento := novo_temporário;
emitir (L.local ':=' c(lista_E.array));
emitir (L.deslocamento ':=' lista_E.local '' largura (lista_E.array))*}

Um deslocamento nulo indica um nome simples.

(6) $L \rightarrow$ **id** {*L.local* := **id.***local;*
L.deslocamento := **null** }

Quando a próxima expressão de índice for enxergada, aplicamos a fórmula de recorrência (8.8). Na ação seguinte, *lista_E.local* corresponde a e_{m-1} em (8.8) e *lista_E.local* a e_m. Note-se que se *lista_E* possuir m-1 componentes, então *lista_E*, ao lado esquerdo da produção, possui m componentes.

(7) $lista_E \rightarrow lista_E_1\ E$ {*t := novo_temporário;*
m := lista_E$_1$*.ndim* + 1;
emitir (t ':=' lista_E$_1$*.local '*' limite(lista-E*$_1$*.array, m));*
emitir (t ':=' t '+' E.local);
lista_E.array := lista_E$_1$*.array;*
lista_E.local := t;
lista_E.ndim := m}

E.local abriga tanto o valor da expressão E quanto o valor de (8.7) para $m = 1$.

(8) $lista_E \rightarrow$ **id** | E { *lista_E* := **id.***local;*
lista_E.local := *E.local;*
lista_E.ndim := 1 }

Exemplo 8.2. *Seja* A um *array* 10×20 com $linf_1 = linf_2 = 1$. Por conseguinte, $n_1 = 10$ e $n_2 = 20$. Façamos w igual a 4. Uma árvore gramatical anotada para a atribuição `x :=A[y,z]` é mostrada na Fig. 8.18. A atribuição é traduzida na seguinte seqüência de enunciados de três endereços:

```
t1 := y * 20
t1 := t1 + z
t2 := c                    /* constant c = base_A - 84 */
t3 := 4 * t1
t4 := t2 [t3]
x := t4
```

Para cada variável, usamos seu nome em lugar de **id.***local*. □

Conversões de Tipo dentro de Atribuições

Na prática, deverão existir muitos tipos diferentes de variáveis e constantes, de forma que o compilador ou precisa rejeitar certas operações com tipos mistos ou gerar as instruções de coerção (conversão de tipo) apropriadas.

Consideremos a gramática para os enunciados de atribuição, como acima, mas suponhamos que existam dois tipos — real e inteiro, com os inteiros convertidos para reais quando necessário.

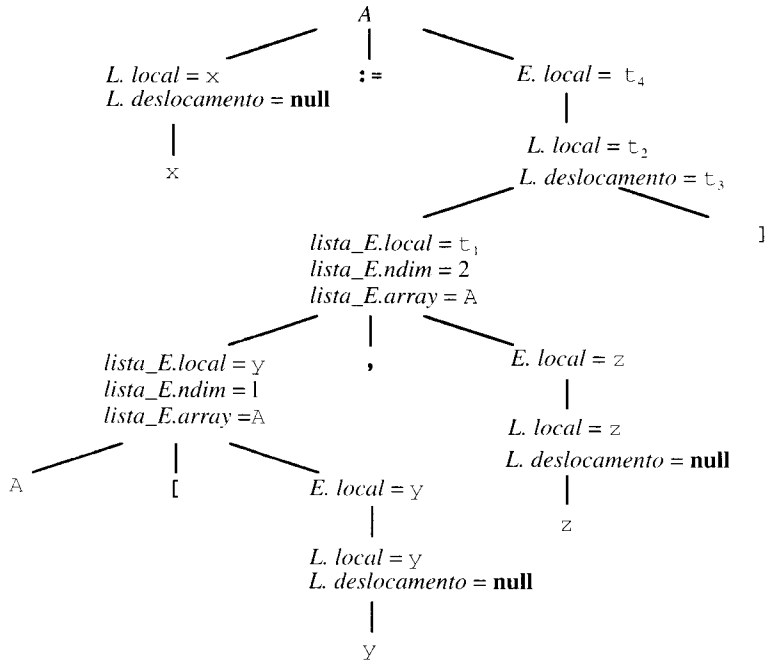


Fig. 8.18. Árvore gramatical anotada para `X := A[y,z]`.

Introduzimos um outro atributo *E.tipo*, cujo valor é *real* ou *inteiro*. A regra semântica para *E.tipo* associada à produção $E \rightarrow E + E$ é

$E \rightarrow E + E$ {*E.tipo* :=
se E_1.*tipo* = *inteiro* **e**
E_2.*tipo* = *inteiro* **então** *inteiro*
senão *real*}

Esta regra está no espírito da Seção 6.4; entretanto, aqui e ali neste capítulo, omitimos as verificações para os erros de tipo; uma discussão da verificação de tipos aparece no Capítulo 6.

Toda a regra semântica para $E \rightarrow E + E$, e para a maioria das outras produções, precisa ser modificada para gerar, quando necessário, enunciados de três endereços da forma `x := inttoreal y`, cujo efeito é o de converter o inteiro `y` para um real de igual valor, chamado `x`. Precisamos também incluir, junto com o código do operador, uma indicação da aritmética pretendida, se de ponto fixo ou flutuante. A ação semântica completa para uma produção da forma $E \rightarrow E_1 + E_2$ é listada na Fig. 8.19.

Por exemplo, para a entrada

```
x := y + i * j
```

assumindo que `x` e `y` tenham tipo *real* e que `i` e `j` tenham tipo *inteiro*, a saída se pareceria com o seguinte

```
t1 := i *int j
t3 := inttoreal t1
t2 := y + real t3
x := t2
```

A ação semântica da Fig. 8.19 usa dois atributos *E.local* e *E.tipo* para o não terminal *E*. À medida que o número de tipos sujeitos à conversão aumenta, o número de casos que emergem cresce quadraticamente (ou pior, se existirem operadores com mais de dois argumentos). Por conseguinte, com um número amplo de tipos, uma organização cuidadosa das ações semânticas se torna importante.

O Acesso aos Campos nos Registros

O compilador precisa controlar tanto os tipos quanto os endereços relativos dos campos de um registro. Uma vantagem de se manter essas informações em entradas da tabela de símbolos para os nomes dos campos está em que a rotina para procurar por nomes na tabela de símbolos também pode ser usada para nomes de campos. Com isso em mente, uma tabela de símbolos separada foi criada para cada tipo de registro pelas ações semânticas na Fig. 8.14, na última seção. Se *t* é um apontador para a tabela de símbolos para um tipo de registro, então o tipo *registro(t)*, formado pela aplicação do construtor *registro* ao apontador, é retornado como *T.tipo*.

Usamos a expressão

```
p↑.info + 1
```

para ilustrar como um apontador para a tabela de símbolos pode ser extraído a partir do atributo *E.tipo*. A partir das operações nesta expressão segue que `p` precisa ser um apontador para um registro com um nome de campo `info` cujo tipo é aritmético. Se os tipos são construídos como nas Figs. 8.13 e 8.14, o tipo de `p` precisa ser fornecido por uma expressão de tipo

apontador (registro(t))

O tipo de `p↑` é, por conseguinte, *registro(t)*, a partir do qual *t* pode ser extraído. O nome de campo `info` é procurado na tabela de símbolos apontada por *t*.

E.local := *novo_temporário;*
se *E.tipo* = *inteiro* **e** E_2.*tipo* = *inteiro* **então início**
emitir(*E.local* ':=' E_1.*local* '+ int' E_2.*local*);
E.tipo := *inteiro*
fim
senão se E_1.*tipo* = *real* **e** E_2.*tipo* = *real* **então início**
emitir(*E.local* ':=' E_1.*local* '+ real' E_2.*local*);
E.tipo := *real*
fim
senão se E_1.*tipo* = *inteiro* e E_2.*tipo* = *real* **então início**
u := *novo_temporário;*
emitir(*u* ':=' inttoreal E_1.*local*);
emitir(*E.local* ':=' u '+ real' E_2.*local*);
E.tipo := *real*
fim
senão se E_1.*tipo* = *real* e E_2.*tipo* = *inteiro* **então início**
u := *novo_temporário;*
emitir(*u* ':=' inttoreal E_2.*local*);
emitir(*E.local* ':=' E_1.*local* '+ real' *u*);
E.tipo := *real*
fim
senão
E.tipo := *tipo_erro*;

Fig. 8.19. Ação semântica para $E \rightarrow E_1 + E_2$.

8.4 EXPRESSÕES BOOLEANAS

Nas linguagens de programação, as expressões booleanas têm dois propósitos primários. São usadas para computar valores lógicos, porém mais freqüentemente são usadas em expressões condicionais em comandos que alteram o fluxo de controle, tais como *if-then-else* ou *while*.

As expressões booleanas são compostas por operadores booleanos (**and** (e), **or** (ou) e **not** (não))* aplicados a elementos que são variáveis booleanas ou expressões relacionais. Por sua vez, as expressões relacionais são da forma E_1 **relop** E_2, onde E_1 e E_2 são expressões aritméticas. Algumas linguagens, tais como PL/I, permitem que expressões mais gerais, onde os operadores booleanos, aritméticos e relacionais podem ser aplicados a expressões de qualquer tipo, sem distinção entre valores booleanos ou aritméticos; uma coerção é realizada quando necessária. Nesta seção, consideramos as expressões booleanas geradas pela seguinte gramática:

$E \rightarrow E$ **or** $E \mid E$ **and** $E \mid$ **not** $E \mid (E) \mid$ **id relop id** $\mid$ **true** $\mid$ **false**

Usamos o atributo *op* para determinar qual dos operadores $<, \leqslant, =, \neq, >$ ou $\geqslant$ é representado por **relop.** Como usual, assumimos que **or** e **and** sejam associativos à esquerda e que **or** tenha a menor precedência, em seguida **and** e então **not**.

Métodos de Tradução de Expressões Booleanas

Existem dois métodos principais para se representar o valor de uma expressão booleana. O primeiro método é codificar verdadeiro e falso numericamente e avaliar uma expressão booleana de forma análoga a uma expressão aritmética. Freqüentemente `1` é usado para denotar verdadeiro e `0` falso, apesar de muitas outras codificações serem também possíveis. Por exemplo, poderíamos fazer qualquer quantidade diferente de zero denotar verdadeiro e zero denotar falso, ou poderíamos fazer qualquer quantidade não negativa denotar verdadeiro e qualquer número negativo, falso.

*Iremos manter esses operadores na língua original inglesa uma vez que essas estruturas não existem como construções de linguagens de programação em língua portuguesa. No pseudocódigo, entretanto, essas operações serão traduzidas para a língua portuguesa. (N. do T.)

O segundo método principal de se implementar expressões booleanas é através do fluxo de controle, isto é, representar o valor de uma expressão booleana através de uma posição atingida num programa. Esse método é particularmente conveniente na implementação de expressões booleanas em enunciados de fluxo de controle, tais como *if-then-else* e *while*. Por exemplo, dada a expressão E_1 **or** E_2, se determinarmos que E_1 é verdadeira poderíamos concluir que a expressão inteira também o é, sem ter que avaliar E_2.

A semântica da linguagem de programação determina se todas as partes de uma expressão booleana precisam ser avaliadas. Se a definição da linguagem permite (ou requer) que partes de uma expressão booleana sigam inavaliadas, o compilador pode otimizar a avaliação das expressões booleanas computando somente o suficiente de uma expressão para determinar o seu valor. Por conseguinte, numa expressão como E_1 **or** E_2, nem E_1 nem E_2 precisam ser avaliadas por inteiro necessariamente. Se E_1 ou E_2 for uma expressão com efeitos colaterais (por exemplo, contenha uma função que modifique uma variável global), uma resposta inesperada pode ser obtida.

Nenhum dos métodos acima é uniformemente superior ao outro. Por exemplo, o compilador otimizante BLISS/11 (Wulf et al. 1975), dentre outros, escolhe o método apropriado para cada expressão individualmente. Esta seção considera ambos os métodos para a tradução de expressões booleanas para o código de três endereços.

Representação Numérica

Vamos primeiro considerar a implementação de expressões booleanas usando `1` para denotar verdadeiro e `0` para falso. As expressões serão avaliadas completamente, da esquerda para a direita, numa forma similar às expressões aritméticas. Por exemplo, a tradução para

```
a or b and not c
```

é a seqüência de três endereços

```
t1 := not c
t2 := b and t1
t3 := a or t2
```

Uma expressão relacional tal como `a < b` é equivalente ao enunciado condicional `if a < b then 1 else 0`, que pode ser traduzido na seqüência de código de três endereços (de novo, começamos os números de comando em 100):

```
100: if a < b goto 103
101: t := 0
102: goto 104
103: t := 1
104:
```

Um esquema de tradução para produzir código de três endereços para expressões booleanas é mostrado na Fig. 8.20. Nesse esquema, assumimos que *emitir* coloca enunciados de três endereços num arquivo de saída no formato correto, que *próximo_estado* fornece o índice do próximo enunciado de três endereços na seqüência de saída e que *emitir* incrementa *próximo_estado* após produzir cada enunciado de três endereços.

Exemplo 8.3. O esquema na Fig. 8.20 geraria o código de três endereços da Fig. 8.21 para a expressão `a < b or c < d and e < f`. □

Código em Curto-Circuito

Podemos também traduzir uma expressão booleana em código de três endereços sem gerar código para quaisquer dos operadores booleanos e sem ter código para avaliar a expressão por inteiro. Esse estilo de avaliação é algumas vezes chamado de código em "curto-circuito" ou código de "jumpeamento". É possível se avaliar expressões booleanas sem gerar código para os operadores booleanos **and, or** e **not** se representamos o valor de uma expressão por uma posição na seqüência de código. Por exemplo, na Fig. 8.21, podemos dizer que valor t_1 terá caso atinjamos o enunciado 101 ou 103, e, dessa forma, o valor de t_1 é redundante. Para muitas expressões booleanas, é possível determinar o valor da expressão sem termos que avaliá-la completamente.

$E \rightarrow E_1$ **or** E_2	{ *E.local* := *novo_temporário*; *emitir* (*E.local*):=' E_1.*local* 'ou' E_2.*local*) }
$E \rightarrow E_1$ **and** E_2	{ *E.local* := *novo_temporário*; *emitir* (*E.local* ':=' E_1.*local* 'e' E_2.*local*) }
$E \rightarrow$ **not** E_1	{ *E.local* := *novo_temporário*; *emitir*(*E.local* ':=' 'not' E_1.*local*) }
$E \rightarrow (E_1)$	{ *E.local* := E_1.*local* }
$E \rightarrow$ **id**$_1$ **relop id**$_2$	{ *E.local* := *novo_temporário*; *emitir*('if' **id**$_1$.*local* **relop**.*op* **id**$_2$.*local* 'goto' *próximo_estado* + 3); *emitir*(*E.local* ':=' '0'); *emitir*('**goto**' *próximo_estado* + 2); *emitir*(*E.local* ':=' '1') }
$E \rightarrow$ **true**	{ *E.local* := *novo_temporário*; *emitir*(*E.local* ':=' '1' }
$E \rightarrow$ **false**	{ *E.local* := *novo_temporário*; *emitir*(*E.local* ':=' '0') }

Fig. 8.20. Esquema de tradução usando uma representação numérica para booleanos.

Enunciados de Fluxo de Controle

Consideremos agora a tradução de expressões booleanas em código de três endereços no contexto de enunciados *if-then, if-then-else* e *while-do*, tais como aqueles gerados pela seguinte gramática:

$$
\begin{array}{lll}
S & \rightarrow & \textbf{if } E \textbf{ then } S_1 \\
 & | & \textbf{if } E \textbf{ then } S_1 \textbf{ else } S_2 \\
 & | & \textbf{while } E \textbf{ do } S_1
\end{array}
$$

Em cada uma dessas produções, E é uma expressão booleana a ser traduzida. Na tradução, assumimos que um enunciado de três endereços possa ser simbolicamente rotulado e que a função *novo_rótulo* retorna um novo rótulo simbólico a cada vez que for chamada.

A uma expressão booleana E, associamos dois rótulos: $E.v$, o rótulo para o qual o controle flui se E for verdadeiro e $E.f$, o rótulo para o qual flui se E for falso. As regras semânticas para traduzir um enunciado de fluxo de controle S permitem que o controle flua da tradução $S.código$ para a instrução de três endereços que se segue imediatamen-

```
100:  if a < b goto 103      107:  t2 := 1
101:  t1 := 0                108:  if e < f goto 111
102:  goto 104               109:  t3 := 0
103:  t1 := 1                110:  goto 112
104:  if c < d goto 107      111:  t3 := 1
105:  t2 := 0                112:  t4 := t2 e t3
106:  goto 108               113:  t5 := t1 ou t4
```

Fig. 8.21. Tradução de `a < b or c < d and e < f`.

te a *S.código*. Em alguns casos, a instrução que se segue a *S.código* é um desvio para algum rótulo *L*. Um desvio para `L`, a partir de dentro de *S.código*, é evitado utilizando-se o atributo herdado *S.próximo*. O valor de *S.próximo* é um rótulo que é atrelado à primeira instrução de três endereços a ser executada após o código para *S*.[5] A inicialização de *S.próximo* não é mostrada.

Ao se traduzir o enunciado condicional $S \rightarrow$ **if** E **then** S_1, um novo rótulo *E.v* é criado e atrelado à primeira instrução de três endereços gerada para o enunciado S_1, como na Fig. 8.22(a). A definição dirigida pela sintaxe aparece na Fig. 8.23. O código para *E* gera um desvio para *E.v* se *E* for verdadeiro e um salto para *E.próximo* se *E* for falso. Conseqüentemente, fazemos *E.f* igual a *S.próximo*.

Na tradução do enunciado `condicional` $S \rightarrow$ **if** E **then** S_1 **else** S_2, o código para a expressão booleana *E* possui desvios, para fora do mesmo, para a primeira instrução do código para S_1, se *E* for verdadeiro, e para a primeira instrução do código para S_2, se *E* for falso, como ilustrado na Fig. 8.22(b). Da mesma forma que com o enunciado *if-then*, o atributo herdado *S.próximo* fornece o próximo rótulo da instrução de três endereços a ser executada em seguida à execução do código para S. Um `goto (desvio)` explícito para *S.próximo* aparece após o código para S_1, mas não após S_2. Deixamos para o leitor mostrar que, com essas regras semânticas, se *S.próximo* não for o rótulo da instrução que se segue imediatamente a *S.código*, então um enunciado envolvente irá fornecer o desvio para o rótulo *S.próximo* após o código para S_2.

O código para $S \rightarrow$ **while** E **do** S_1 é formado como mostrado na Fig. 8.22(c). Um novo rótulo *S.início* é criado e atrelado à primeira instrução gerada para *E*. Um outro novo rótulo, *E.t*, é atrelado à primeira instrução para S_1. O código para *E* gera um desvio para esse rótulo se *E* for verdadeiro e um desvio para *E.f* se *E* for falso; de novo, fazemos *E.falso* igual a *S.próximo*. Após o código para S_1, colocamos a instrução `goto` *S.início*, que provoca um desvio de volta ao início do código para a expressão booleana. Note-se que S_1*.próximo* é estabelecido para esse rótulo *S.início*, de forma que os desvios de dentro de S_1*.código* podem se dirigir diretamente para *S.início*.

Discutimos a tradução dos enunciados de fluxo de controle em mais detalhes na Seção 8.6, onde um método alternativo, chamado de "retrocorreção", emite o código para tais enunciados em uma única passagem.

Tradução de Expressões Booleanas em Fluxo de Controle

Discutimos agora *E.código*, o código produzido para expressões booleanas *E* na Fig. 8.23. Como indicamos, *E* é traduzido numa seqüência de enunciados de três endereços que avalia *E* como uma seqüência de desvios condicionais e incondicionais para uma ou duas localizações: *E.v*, a localização que o controle deve atingir se *E* for verdadeiro e *E.f*, o local que deve ser atingido caso *E* seja falso.

A idéia básica por trás da tradução é a seguinte. Suponhamos que *E* seja da forma `a < b`. O código gerado é, conseqüentemente, da forma

```
if a < b goto E.v
goto E.f
```

Suponhamos que *E* seja da forma E_1 **or** E_2. Se E_1 for verdadeiro, sabemos imediatamente que *E* já é verdadeira, de forma que $E_1.v$ é o mesmo que *E.v*. Se E_1 for falso, então E_2 precisa ser avaliado, e, dessa forma, fazemos $E_1.f$ ser o rótulo do primeiro enunciado no código para E_2. As saídas verdadeira e falsa de E_2 podem ser feitas iguais às saídas verdadeira e falsa de *E*, respectivamente.

Considerações análogas se aplicam à tradução de E_1 **and** E_2. Nenhum código da forma **not** E_1 é necessitado para uma expressão *E*;

[5] Se implementado literalmente, o enfoque de se herdar o rótulo *S.próximo* pode levar a uma proliferação de rótulos. A abordagem da retrocorreção da Seção 8.6 cria rótulos somente quando forem necessários.

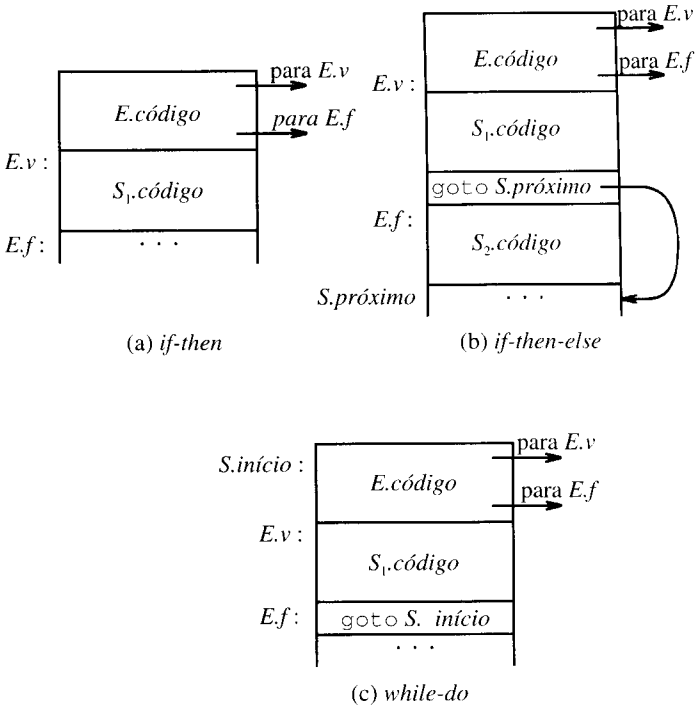


Fig. 8.22 Código para enunciados *if-then*, *if-then-else* e *while-do*.

simplesmente intercambiamos as saídas falsas de E_1 para obter as saídas verdadeira e falsa de *E*. Uma definição dirigida pela sintaxe que gera código de três endereços para expressões booleanas desta forma é mostrada na Fig. 8.24. Note-se que os atributos *v* e *f* são herdados.

Exemplo 8.4. Vamos considerar de novo a expressão

```
a < b or c < d and e < f
```

Suponhamos que as saídas verdadeira e falsa para toda a expressão tenham sido estabelecidas em `Lv` e `Lf`. Então, usando a definição da Fig. 8.24, obteríamos o seguinte código:

```
        if a < b goto Lv
        goto L1
L1:     if c < d goto L2
        goto Lf
L2:     if e < f goto Lv
        goto Lf
```

Note-se que o código gerado não é ótimo, na medida em que o segundo enunciado pode ser eliminado sem mudar seu valor. Instruções redundantes dessa forma podem ser subseqüentemente removidas por um simples otimizador *peephole* (ver o Capítulo 9). Um outro enfoque que evita a geração desses desvios redundantes é o de traduzir uma expressão relacional da forma $\mathbf{id}_1 < \mathbf{id}_2$ no enunciado `if` $\mathbf{id}_1 \geq \mathbf{id}_2$ `goto` *E.f*, com a presunção de que, quando a relação for verdadeira, seguimos em frente no código. □

Exemplo 8.5. Consideremos o enunciado

```
while a < b do
    if c < d then
        x := y + z
    else
        x := y - z
```

PRODUÇÃO	REGRAS SEMÂNTICAS
$S \rightarrow$ **if** E **then** S_1	$E.v$:= *novo_rótulo;* $E.f$:= *S.próximo;* S_1*.próximo* := S_1*.próximo;* *S.código* := *E.código* ‖ *gerar* (*E.v* ':') ‖ S_1*.código*
$S \rightarrow$ **if** E **then** S_1 **else** S_2	$E.v$:= *novo_rótulo;* $E.f$:= *novo_rótulo;* S_1*.próximo* := *S.próximo;* S_2*.próximo* := *S.próximo;* *S.código* := *E.código* ‖ *gerar* (*E.v* ':') ‖ S_1*.código* ‖ *gerar* ('`goto`' *S.próximo*) ‖ *gerar* (*E.f* ':') ‖ S_2*.código*
$S \rightarrow$ **while** E **do** S_1	*S.início* := *novo_rótulo;* $E.v$:= *novo_rótulo;* $E.f$:= *S.próximo;* S_1*próximo* := *S.início;* *S.código* := *gerar* (*S.início* ':') ‖ *E.código* ‖ *gerar* (*E.v* ':') ‖ S_1*.código* ‖ *gerar* ('`goto`' *S.início*)

Fig. 8.23. Definição dirigida pela sintaxe para enunciados de fluxo de controle.

A definição dirigida pela sintaxe acima, acoplada aos esquemas para os enunciados de atribuição e de expressões booleanas, produziriam o seguinte código:

```
L1:     if a < b goto L2
        goto Lpróximo
L2:     if c < d goto L3
        goto L4
L3:     t1 := y + z
        x := t1
        goto L1
L4:     t2 := y - z
        x := t2
        goto L1
Lpróximo:
```

Notamos que os dois primeiros desvios podem ser eliminados pela modificação das direções dos testes. Essa transformação local típica pode ser feita pela otimização *peephole* discutida no Capítulo 9. □

Expressões Booleanas em Modo Misto

É importante compreendermos que temos simplificado a gramática para as expressões booleanas. Na prática, as expressões booleanas freqüentemente contêm subexpressões aritméticas, como em `(a + b) < c`. Em linguagens onde falso possui o valor numérico 0 e verdadeiro o valor 1, a expressão `(a < b) + (b < a)` pode mesmo ser considerada aritmética, com valor 0 se `a` e `b` tiverem o mesmo valor e 1 em caso contrário.

PRODUÇÃO	REGRAS SEMÂNTICAS
$E \rightarrow E_1$ **or** E_2	$E_1.v$:= *E.v;* $E_1.f$:= *novo_rótulo;* $E_2.v$:= *E.v;* $E_2.f$:= *E.f;* *E.código* := E_1*.código* ‖ *gerar* ($E_1.f$ ':') ‖ E_2*.código*
$E \rightarrow E_1$ **and** E_2	$E_1.v$:= *novo-rótulo;* $E_1.f$:= *E.f;* $E_2.v$:= *E.v;* $E_2.f$:= *E.f;* *E.código* := E_1*.código* ‖ *gerar* ($E_1.v$ ':') ‖ E_2*.código*
$E \rightarrow$ **not** E_1	$E_1.v$:= *E.f;* $E_1.f$:= *E.v;* *E.código* := E_1*.código;*
$E \rightarrow (E_1)$	$E_1.v$:= *E.t;* $E_1.f$:= *E.f;* *E.código* := E_1*.código;*
$E \rightarrow$ **id**$_1$ **relop** **id**$_2$	*E.código* := *gerar* ('`if`' **id**$_1$*.local* **relop***.op* **id**$_2$*.local* '`goto`' *E.v*) ‖ *gerar* ('`goto`' *E.f*)
$E \rightarrow$ **true**	*E.código* := *gerar* ('`goto`' *E.v*)
$E \rightarrow$ **false**	*E.código* := *gerar* ('`goto`' *E.f*)

Fig. 8.24. Definição dirigida pela sintaxe para produzir código de três endereços para expressões booleanas.

O método para se representar expressões booleanas saltando-se código pode ainda ser usado, mesmo que as expressões aritméticas sejam representadas por código para computar seus valores. Por exemplo, consideremos a gramática modelo

$$E \rightarrow E + E \mid E \textbf{ and } E \mid E \textbf{ relop } E \mid \textbf{id}$$

Podemos supor que $E + E$ produza um resultado aritmético inteiro (a inclusão do tipo real ou de outros tipos aritméticos torna o assunto mais complicado, mas não adiciona nada ao valor instrutivo deste exemplo), enquanto as expressões E **and** E e E **relop** E produzam valores booleanos representados pelo fluxo de controle. A expressão E **and** E requer que ambos os argumentos sejam booleanos, mas as operações + e **relop** recebem os dois tipos de argumentos, incluindo valores mistos. $E \rightarrow$ **id** é também considerada aritmética, apesar de que poderíamos estender esse exemplo permitindo identificadores booleanos.

Para gerar o código nesta situação, podemos usar o atributo sintetizado *E.tipo*, que terá o valor *aritm* ou *bool*, dependendo do tipo de E. E terá os atributos herdados $E.v$ e $E.f$ para as expressões booleanas e o atributo sintetizado *E.local* para as expressões aritméticas. Parte da regra semântica para $E \rightarrow E_1 + E_2$ é mostrada na Fig. 8.25.

No caso de modo misto, geramos o código para E_1, em seguida para E_2, seguido pelos três enunciados:

$E_2.v$: *E.local* := E_1.*local* + *1*
`goto` *próximo-estado* + 1
$E_2.f$: *E.local* := E_1.local

O primeiro enunciado computa o valor $E_1 + 1$ para E, quando E_2 for verdadeiro, o terceiro, o valor E_1 quando E_2 for falso. O segundo enunciado é um salto sobre o terceiro. As regras semânticas nos casos remanescentes e as outras produções são similares e as deixamos como exercícios.

8.5 ENUNCIADOS DE DESVIO MÚLTIPLO*

O enunciado *switch* ou *case* está disponível numa variedade de linguagens; mesmo os desvios computado e atribuído de Fortran podem ser considerados como variedades do enunciado de desvio múltiplo. Nossa sintaxe do enunciado de desvio múltiplo é mostrada na Fig. 8.26.

Existe uma expressão seletora, que deve ser avaliada, seguida por n valores constantes que a expressão poderia assumir, possivelmente incluindo um "valor" *default*, que sempre se iguala ao valor da expressão se nenhum outro o fizer. A tradução desejada para um desvio múltiplo é um código para:

1. Avaliar a expressão.
2. Encontrar que valor na lista de alternativa é igual ao valor da expressão. Relembremos que o valor *default* sempre se iguala ao da expressão se nenhum dos valores explicitamente mencionados nas alternativas se igualar.
3. Executar o enunciado associado ao valor encontrado.

O passo (2) é uma ramificação de n saídas que pode ser implementada em uma dentre várias formas possíveis. Se o número de alternativas não for muito grande, digamos no máximo 10, é razoável usar uma seqüência de desvios condicionais, cada um dos quais testa por um valor individual e transfere o controle para o código do enunciado correspondente.

Uma forma mais compacta de implementar essa seqüência de desvios condicionais é criar uma tabela de pares, cada qual consistindo em um valor e um rótulo para o código do enunciado correspondente. O código é gerado de forma a colocar ao final dessa tabela o valor da própria expressão associada ao rótulo para o enunciado *default*. Um laço simples pode ser gerado pelo compilador de forma a comparar o valor da expressão a cada valor da tabela, estando asegurado que, se nenhuma igualdade for encontrada, a última entrada *(default)* é garantida se igualar ao valor da expressão.

Se o número de valores exceder a 10, ou algo em torno, é mais eficiente construir uma tabela *hash* (ver Seção 7.6) para os valores, tendo os rótulos dos vários enunciados como entradas. Se nenhuma entrada para o valor computado para a expressão do enunciado de desvio múltiplo for encontrada, um salto para o enunciado *default* pode ser gerado.

Existe um caso especial comum para o qual existe uma implementação ainda mais eficiente do desvio de ramificação múltipla. Se todos os valores recaem em algum pequeno intervalo, digamos, de $i_{mín}$ até $i_{máx}$, e o número de valores for uma fração razoável de $i_{máx}$-$i_{mín}$, podemos então construir um *array* de rótulos, com o rótulo para o enunciado de número j na entrada da tabela com deslocamento j-$i_{mín}$ e o rótulo para o *default* nas entradas não preenchidas. Para realizar o desvio múltiplo, avaliamos a expressão de forma a obtermos o valor de j, verificamos se o mesmo está no intervalo de $i_{mín}$ até $i_{máx}$ e transferimos indiretamente para a entrada da tabela de símbolos que está ao deslocamento j-$i_{mín}$. Por exemplo, se a expressão for do tipo caractere, uma tabela, digamos, com 128 entradas (dependendo do conjunto de caracteres) pode ser criada e receber transferência sem nenhum teste de intervalo.

E.tipo := *aritm;*
se E_1.*tipo* = *aritm* **e** E_2.*tipo* = *aritm* **então início**
/* operação aritmética de adição normal */
E.local := *novo_temporário;*
E.código := E_1.*código* := E_1.*código* || E_2.*código* ||
gerar(*E.local* ':=' E_1.*local* '+' E_2.*local*)
fim
senão se E_1.*tipo* = *aritm* **e** E_2.*tipo* = *bool* **então início**
E.local := *novo_temporário;*
$E_2.v$:= *novo_rótulo;*
$E_2.f$:= *novo_rótulo;*
E.código := E_1.*código* || E_2.*código* ||
gerar($E_2.v$ ':' *E.local* ':=' E_1.*local* + 1) ||
gerar('`goto`' *próximo_estado* + 1) ||
gerar($E_2.f$ ':' *E.local* ':=' E_1.*local*)
senão se . . .

Fig. 8.25. Regra semântica para a produção $E \rightarrow E_1 + E_2$.

switch *expressão*
begin
case *valor* : *comando*
case *valor* : *comando*
. . .
case *valor* : *comando*
default : *comando*
end

Fig. 8.26. Sintaxe do enunciado de desvio múltiplo.

*Do original em inglês: *case statements*. (N. do T.)

Tradução Dirigida pela Sintaxe para Enunciados de Desvios Múltiplos

Consideremos o seguinte enunciado de desvio múltiplo.

```
switch E
    begin
        case V1 :      S1
        case V2 :      S2
            ...
        case Vn-1 :    Sn-1
        default:       Sn
    end
```

Com um esquema de tradução dirigido pela sintaxe, é conveniente se traduzir esse enunciado *case* no código intermediário que possui a forma da Fig. 8.27.

Os testes todos aparecem ao final, de forma que um gerador de código simples pode reconhecer o desvio múltiplo e gerar código eficiente para o mesmo, usando a implementação mais apropriada sugerida ao início desta seção. Se gerarmos a seqüência mais direta mostrada na Fig. 8.28, o compilador teria que realizar uma análise extensiva para encontrar a implementação mais eficiente. Note-se que é inconveniente se colocar os enunciados de ramificação ao início, porque o compilador não iria emitir em seguida o código para cada um dos S_i's à medida que os visse.

Para traduzir na forma da Fig. 8.27, ao enxergarmos a palavra-chave **switch,** geramos dois novos rótulos, `teste` e `próximo`, e um novo temporário `t`. Em seguida, à medida que analisamos sintaticamente a expressão E, geramos código para avaliar E em `t`. Após processarmos E, geramos o salto `goto teste`.

Em seguida, ao enxergarmos cada palavra-chave **case,** criamos um novo rótulo L_i e o introduzimos na tabela de símbolos. Colocamos numa pilha, usada somente para armazenar as alternativas, um apontador para essa entrada da tabela de símbolos e o valor V_i da constante associada ao *case* (se esse desvio múltiplo estiver embutido em algum dos enunciados internos a outro desvio múltiplo, colocamos um marcador na pilha para separar as alternativas de um desvio múltiplo daquelas do desvio múltiplo mais externo).

Processamos cada enunciado **case** V_i **:** S_i emitindo o rótulo recém-criado L_i, seguido pelo código para S_i, seguido pelo desvio `goto próximo`. Em seguida, quando a palavra-chave **end,** que termina o corpo do desvio múltiplo é encontrada, estamos prontos para gerar o código da ramificação. Lendo os pares apontador-valor na pilha das alternativas, do fundo para o topo, podemos gerar uma seqüência de enunciados de três endereços da forma

```
        código para avaliar E em t
        goto teste
L1:     código para S1
        goto próximo
L2:     código para S2
        goto próximo
            . . .
Ln-1:   código para Sn-1
        goto próximo
Ln:     código para Sn
        goto próximo
teste:  if t = V1 goto L1
        if t = V2 goto L2
            . . .
        if t = Vn-1 goto Ln-1
        goto Ln
próximo:
```

Fig. 8.27. Tradução de um enunciado de desvio múltiplo.

```
        código para avaliar E em t
        if t ≠ V1 goto L1
        código para S1
        goto proximo
L1:     if t ≠ V2 goto L2
        código para S2
        goto proximo
L2:
            ...
Ln-2:   if t ≠ Vn-1 goto Ln-1
        código para Sn-1
        goto proximo
Ln-1:   código para Sn
proximo:
```

Fig. 8.28. Outra tradução de um enunciado de desvio múltiplo.

```
caso V1 L1
caso V2 L2
    ...
caso Vn-1 Ln-1
caso t Ln
rotulo proximo
```

onde `t` é o nome que abriga o valor da expressão seletora E e L_n é o rótulo para o enunciado *default.* O enunciado de três endereços `caso` V_i L_i é um sinônimo para `if t = `V_i `goto` L_i, na Fig. 8.27, mas nesse `caso`, é mais fácil para o gerador de código final detectá-lo como um candidato para um tratamento especial. Na fase de geração de código, essas seqüências de enunciados `caso` podem ser traduzidas numa ramificação de n alternativas do tipo mais eficiente, dependendo de quantas existam e se os valores caem num intervalo pequeno.

8.6 RETROCORREÇÃO*

A forma mais fácil de se implementar as definições dirigidas pela sintaxe na Seção 8.4 é se usar duas passagens. Primeiro, construímos uma árvore sintática para a entrada e, em seguida, caminhamos na árvore numa ordem busca em profundidade, computando as traduções dadas na definição. O principal problema de se gerar código para as expressões booleanas e para o fluxo de controle numa única passagem está em que, durante a mesma, não podemos conhecer os rótulos para onde o controle deve ir, no tempo em que os enunciados de desvio são gerados. Podemos solucionar esse problema pela geração de uma série de enunciados de ramificação, com os alvos dos desvios deixados temporariamente inespecificados. Cada enunciado será colocado numa lista de enunciados de desvio cujos rótulos serão preenchidos quando o rótulo apropriado puder ser determinado. Chamamos esse preenchimento subseqüente de rótulos de *retrocorreção.*

Nesta seção, mostramos como a retrocorreção pode ser usada para gerar código para expressões booleanas e fluxo de controle em uma única passagem. As traduções que gerarmos serão da mesma forma que aquelas na Seção 8.4, exceto pela maneira na qual geraremos os rótulos. Especificamente, geramos quádruplas num *array* de quádruplas. Os rótulos serão índices para esse *array.* Para manipular as listas de rótulos, usamos três funções:

*Do original em inglês: *backpatching.* (N. do T.)

1. *criar_lista(i)* cria uma nova lista contendo somente *i*, um índice para o *array* de quádrupla; *criar_lista* retorna um apontador para a lista construída.
2. *concatenar* (p_1, p_2) concatena as listas apontadas por p_1 e p_2 e retorna um apontador para a lista concatenada.
3. *retrocorrigir (p, i)* insere *i* como o rótulo-alvo para cada enunciado na lista apontada por *p*.

Expressões Booleanas

Construímos agora um esquema de tradução adequado à produção de quádruplas para expressões booleanas durante a análise sintática *bottom-up*. Inserimos um não-terminal marcador *M* na gramática a fim de fazer com que uma ação semântica obtenha, em momentos apropriados, o índice da próxima quádrupla a ser gerada. A gramática que usamos é a seguinte:

(1)	$E \rightarrow$	E_1 **or** $M\ E_2$
(2)	|	E_1 **and** $M\ E_2$
(3)	|	**not** E_1
(4)	|	(E_1)
(5)	|	$\mathbf{id}_1$ **relop** $\mathbf{id}_2$
(6)	|	**true**
(7)	|	**false**
(8)	$M \rightarrow$	ϵ

Os atributos sintetizados *lista_v* e *lista_f* do não-terminal *E* são usados para gerar o código de desvio para as expressões booleanas. À medida que o código é gerado para *E*, os desvios para as saídas verdadeira e falsa são deixados incompletos, com o campo de rótulo não preenchido. Esses desvios incompletos são colocados em listas apontadas por *E.lista_v* e *E.lista_f*, na medida do apropriado.

As ações semânticas refletem as considerações mencionadas acima. Consideremos a produção $E \rightarrow E_1$ **and** $M\ E_2$. Se E_1 for falso, então *E* também é falso e os enunciados mencionados em E_1*.lista_f* se tornam parte de *E.lista_f*. Se E_1 for verdadeiro, entretanto, precisamos testar E_2, e dessa forma que os alvos para os enunciados em E_1*.lista_v* precisam estar ao início do código gerado para E_2. Esse alvo é obtido usando o não-terminal marcador *M*. O atributo *M.quad* registra o número do primeiro enunciado de E_2*.código*. Com a produção $M \rightarrow \epsilon$ associamos a ação semântica

{ *M.quad* := *próxima_quádrupla* }

A variável *próxima_quádrupla* abriga o índice da próxima quádrupla a seguir. Esse valor será retrocorrigido sobre a lista E_1*.lista_v*, quando tivermos examinado o resto da produção $E \rightarrow E_1$ **and** $M\ E_2$. O esquema de tradução é como segue.

(1)	$E \rightarrow E_1$ **or** $M\ E_2$	{ *retrocorrigir* (E_1*.lista_f, M.quad*); *E.lista_v* := *concatenar* (E_1*.lista_v*, E_2*.lista_v*); *E.lista_f* := E_2*.lista_f* }
(2)	$E \rightarrow E_1$ **and** $M\ E_2$	{ *retrocorrigir* (E_1*.lista_t, M.quad*); *E.lista_v* := E_2*.lista_v*; *E.lista_f* := *concatenar* (E_1*.lista_f*, E_2*.lista-f*) }
(3)	$E \rightarrow$ **not** E_1	{ *E.lista-v* := E_1*.lista_f*; *E.lista_f* := E_1*.lista_v* }
(4)	$E \rightarrow (E_1)$	{ *E.lista_v* := E_1*.lista_v*; *E.lista_f* := E_1*.lista_f* }
(5)	$E \rightarrow \mathbf{id}_1$ **relop** $\mathbf{id}_2$	{ *E.lista_v*:= *criar_lista* (*próxima_quádrupla*); *E.lista_f* := *criar_lista* (*próxima_quádrupla* + 1); *emitir* ('se' $\mathbf{id}_1$. *local* **relop.***op* $\mathbf{id}_2$. *local* 'goto –') *emitir* ('goto –') }
(6)	$E \rightarrow$ **true**	{ *E.lista_v* := *criar_lista* (*próxima_quádrupla*); *emitir* ('goto –') }
(7)	$E \rightarrow$ **false**	{ *E.lista_f* := *criar_lista* (*próxima_quádrupla*); *emitir* ('goto –') }
(8)	$M \rightarrow \epsilon$	{ *M.quad* := *próxima_quádrupla* }

Por uma questão de simplicidade, a ação semântica (5) gera dois enunciados, um desvio condicional e um incondicional. Nenhum possui o destino do desvio preenchido. O índice do primeiro enunciado gerado é colocado numa lista e à *E.lista_v* é atribuído um apontador para a mesma. O segundo enunciado de desvio gerado `goto_` é também constituído numa lista e cujo apontador também é analogamente atribuído a *E.lista_f*.

Exemplo 8.6. Consideremos de novo a expressão `a < b or c < d and e < f`. Uma árvore gramatical anotada é mostrada na Fig. 8.29. As ações são realizadas durante uma travessia em profundidade da mesma. Como todas as ações aparecem às extremidades dos lados direitos, podem ser realizadas em conjunto com as reduções durante uma análise sintática *bottom-up*. Em resposta à redução de `a < b` a *E* pela produção (5), as duas quádruplas

```
100: if a < b goto _
101: goto
```

são geradas (começamos de novo, arbitrariamente, a numeração em 100). O não-terminal marcador *M* na produção $E \rightarrow E_1$ **or** $M\ E_2$ registra o valor de *próxima_quádrupla*, que é a esse tempo 102. A redução de `c < d` a *E* pela produção (5) gera as quádruplas

```
102: if c < d goto_
103: goto_
```

Vimos agora E_1 na produção $E \rightarrow E_1$ **and** $M\ E_2$. O não-terminal marcador nesta produção registra o valor corrente de *próxima_quádrupla*, o qual é agora 104. A redução de `e < f` para *E* através da produção (5) gera

```
104: if e < f goto_
105: goto_
```

Reduzimos agora através de $E \rightarrow E_1$ **and** $M\ E_2$. A ação semântica correspondente chama *retrocorrigir* ({102}, 104), onde {102}, como argumento, denota um apontador para a lista contendo somente 102, isto é, a lista que é apontada por *E.lista_v*. Essa chamada para *retrocorrigir* preenche com 104 o enunciado 102. Os seis enunciados gerados são pois:

```
100: if a < b goto:_
101: goto_
102: if c < d goto 104
103: goto_
104: if e < f goto_
105: goto_
```

A ação semântica associada à redução final através de $E \rightarrow E_1$ **or** $M\ E_2$ chama *retrocorrigir* ({101}, 102) o que leva os enunciados a se tornarem:

```
100: if a < b goto_
101: goto 102
102: if c < d goto 104
103: goto_
104: if e < f goto_
105: goto_
```

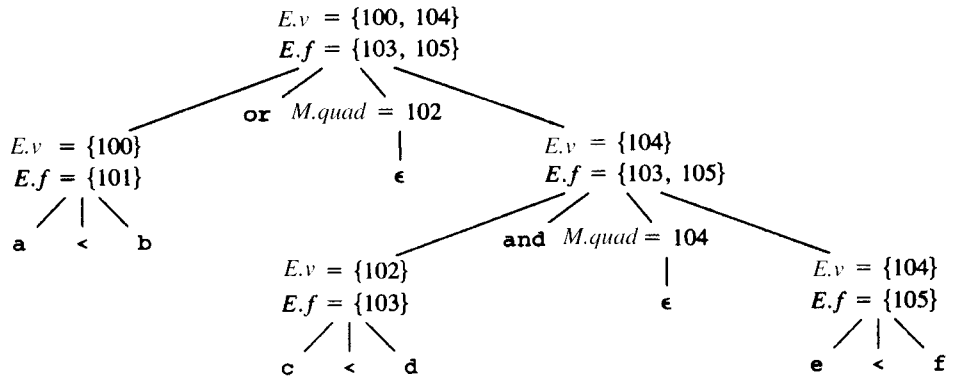


Fig. 8.29. Árvore gramatical anotada para `a < b or c < d and e < f`.

A expressão inteira é verdadeira somente se os desvios dos enunciados 100 ou 104 forem atingidos e é falsa se e somente se os enunciados de desvio 103 ou 105 forem atingidos. Essas instruções terão seus alvos preenchidos mais adiante na compilação, quando for enxergado o que terá que ser feito dependendo da veracidade ou falsidade da expressão. □

Enunciados de Fluxo de Controle

Mostramos agora como a retrocorreção pode ser usada para traduzir os enunciados de fluxo de controle em uma só passagem. Como acima, fixamos nossa atenção na geração de quádruplas e a notação relacionada à tradução de nomes de campos e procedimentos para o tratamento de listas, proveniente da seção anterior, é mantida nesta seção igualmente. Como um exemplo mais amplo, desenvolvemos um esquema de tradução para enunciados gerados pela seguinte gramática:

(1)	$S \rightarrow$	**if** E **then** S
(2)	|	**if** E **then** S **else** S
(3)	|	**while** E **do** S
(4)	|	**begin** L **end**
(5)	|	A
(6)	$L \rightarrow$	L ; S
(7)	|	S

Aqui, S denota um enunciado, L, uma lista de enunciados, A, um enunciado de atribuição, e E, uma expressão booleana. Note-se que pode haver outras produções, tais como aquelas para os enunciados de atribuição. As produções fornecidas, entretanto, serão suficientes para ilustrar as técnicas usadas para traduzir os enunciados de fluxo de controle.

Usamos as mesmas estruturas de código para os enunciados *if-then, if-then-else* e *while-do* que as usadas na Seção 8.4. Fazemos a suposição tácita de que o código que se segue a um dado enunciado em execução também o segue fisicamente no *array* de quádruplas. Se tal não for verdade, um desvio explícito precisará ser providenciado.

Nosso enfoque geral será o de preencher os desvios para fora dos enunciados quando seus destinos forem determinados. Não somente as expressões booleanas precisam de duas listas de desvios, que ocorrem quando uma expressão é verdadeira ou quando é falsa, mas também os enunciados, que precisam igualmente de listas de desvios (dadas pelo atributo *próxima_lista*) para os códigos que os seguem na seqüência de execução.

Esquema para Implementar a Tradução

Descrevemos agora um esquema de tradução dirigida pela sintaxe para gerar traduções para as estruturas de fluxo de controle dadas acima. O não-terminal E possui dois atributos *E.lista_v* e *E.lista_f*, como acima. L e S também necessitam, cada um, de uma lista de quádruplas não preenchidas que eventualmente precisarão ser completadas pela retrocorreção. Essas listas são apontadas pelos atributos *L.próxima_lista* e *S.próxima_lista*. *S.próxima_lista* é um apontador para uma lista de todos os saltos condicionais e incondicionais para a quádrupla que se segue ao enunciado S na ordem de execução, e *L.próxima_lista* é definido similarmente.

No delineamento do código para $S \rightarrow$ **while** E **do** S_1 na Fig. 8.22(c), existem os rótulos *S.início* e *E.v* que marcam o início do código para o enunciado completo S e para o corpo S_1. As duas ocorrências do não-terminal marcador M na seguinte produção registram os números de quádruplas a essas posições:

$$S \rightarrow \textbf{while } M_1\ E \textbf{ do } M_2\ S_1$$

De novo, a única produção para M é $M \rightarrow \epsilon$ com uma ação estabelecendo o atributo *M.quad* com o número da próxima quádrupla. Após o corpo S_1 do enunciado *while* ser executado, o controle flui para o início. Por conseguinte, ao reduzirmos **while** M_1 E **do** M_2 S_1 a S, retrocorrigimos S_1*.próxima_lista* de forma a fazer com que todos os alvos sejam atrelados após o código para S_1 porque o controle pode também "cair além do final". *E.lista_v* é retrocorrigida de forma a ir para o início de S_1 fazendo-se os desvios em *E. lista-y* irem para M_2*.quad*.

Um argumento mais forte para se usar *S.próxima-lista* e *L.próxima_lista* vem quando o código é gerado para o enunciado condicional **if** E **then** S_1 **else** S_2. Se o controle "cai além do fim" de S_1, como quando S_1 é uma atribuição, precisamos incluir ao final do código para S_1 um salto sobre o código para S_2. Usamos um outro não-terminal marcador para introduzir esse salto após S_1. Seja o não-terminal N esse não-terminal marcador com produção $N \rightarrow \epsilon$. N possui o atributo *N.próxima_lista*, o qual será uma lista consistindo em um número de quádrupla do enunciado `goto_` que é gerado pela regra semântica para N. Fornecemos agora as regras semânticas para a gramática revisada.

(1) $S \rightarrow$ **if** E **then** $M_1\ S_1\ N$ **else** $M_2\ S_2$
{ *retrocorrigir (E.lista_v* M_1*.quad);*
retrocorrigir (E.lista_f, M_2*.quad);*
S.próxima_lista := concatenar (S_1*.próxima_lista,*
concatenar (N.próxima_lista, S_2*.próxima_lista))* }

Retrocorrigimos os saltos quando E for verdadeiro para a quádrupla M_1*.quad*, que é o início do código para S_1. Similarmente, retrocorrigimos os desvios quando E for falso de forma que o controle se dirija para o início do código para S_2. A lista *S.próxima-lista* inclui todos os saltos para fora de S_1 e de S_2, bem como o desvio gerado por N.

(2) $N \rightarrow \epsilon$	{ *N.próxima_lista := criar_lista (próxima_quádrupla); emitir* ('goto_') }
(3) $M \rightarrow \epsilon$	{ *M.quad := próxima_quádrupla* }
(4) $S \rightarrow$ **if** E **then** $M\ S_1$	{ *retrocorrigir (E.lista_v, M.quad); S.próxima_lista := concatenar (E.lista_f, S_1.próxima_lista)* }
(5) $S \rightarrow$ **while** $M_1\ E$ **do** $M_2\ S_1$	{ *retrocorrigir (S_1.próxima_lista, M_1.quad); retrocorrigir (E.lista_v, M_2.quad); S.próxima_lista := E.lista_f; emitir* ('goto_' *M_1.quad)* }
(6) $S \rightarrow$ **begin** L **end**	{ *S.próxima_lista := L.próxima_lista* }
(7) $S \rightarrow A$	{ *S. próxima_lista :=* **nil** }

A atribuição *S.próxima_lista :=* **nil** inicializa *S.próxima_lista* com uma lista vazia.

(8) $L \rightarrow L_1; M\ S$	{ *retrocorrigir (L_1.próxima_lista, M.quad); L.próxima_lista := S.próxima_lista* }

O enunciado seguinte a L_1 na ordem de execução é o início de *S*. Por conseguinte, a lista L_1*.próxima_lista* é retrocorrigida para o início do código para *S*, que é dado por *M.quad*.

(9) $L \rightarrow S$	{ *L.próxima_lista := S.próxima_lista* }

Note-se que não são geradas novas quádruplas em qualquer local dessas regras semânticas, a não ser nas regras (2) e (5). Todo o resto do código é gerado pelas ações semânticas associadas aos enunciados de atribuição e expressões. O que o fluxo de controle faz é provocar a retrocorreção adequada, de forma que as avaliações das atribuições e expressões booleanas sejam conectadas adequadamente.

Rótulos e Desvios

A construção de linguagem de programação mais elementar para modificar o fluxo de controle num programa é constituída pelos rótulos e desvios. Quando um compilador encontra um enunciado como `goto L`, precisa verificar se existe exatamente um enunciado com o rótulo `L` no escopo desse enunciado de desvio. Se o rótulo já tiver aparecido, quer num enunciado de declaração de rótulo ou como rótulo de algum enunciado-fonte, a tabela de símbolos já terá uma entrada fornecendo o rótulo gerado pelo compilador para a primeira instrução de três endereços associada ao enunciado-fonte rotulado `L`. Para a tradução, geramos um enunciado de três endereços `goto` tendo aquele rótulo gerado pelo compilador como destino.

Quando um rótulo `L` é encontrado pela primeira vez no programa_fonte, quer numa declaração quer como alvo de um desvio para a frente, introduzimos `L` na tabela de símbolos e geramos um rótulo simbólico para `L`.

8.7 CHAMADAS DE PROCEDIMENTOS

O procedimento[6] é uma construção de programação tão importante e tão freqüentemente usada que é imperativo para um compilador gerar um bom código para as chamadas e retornos dos mesmos. As rotinas em tempo de execução que tratam da transmissão de parâmetros para os procedimentos, das chamadas e retornos são parte do pacote de suporte em tempo de execução. Discutimos os diferentes tipos de mecanismos necessitados para implementar o pacote de suporte em tempo de execução no Capítulo 7. Nesta seção, discutimos o código que é tipicamente gerado para chamadas e retornos de procedimentos.

Vamos considerar uma gramática para um enunciado simples de chamada de procedimento.

(1)	S	$\rightarrow$	**call id** *(lista_E)*
(2)	*lista_E*	$\rightarrow$	*lista_E, E*
(3)	*lista_E*	$\rightarrow$	*E*

Seqüências de Chamada

Como discutido no Capítulo 7, a tradução de uma chamada inclui uma seqüência de chamada, a seqüência de ações tomadas à entrada e à saída de cada procedimento. Conquanto as seqüências de chamada difiram, mesmo para implementações distintas de uma mesma linguagem, as seguintes ações tipicamente têm lugar:

Quando uma chamada de procedimento ocorre, é necessária a reserva de espaço para o registro de ativação do procedimento chamado. Os argumentos do procedimento chamado precisam ser avaliados e tornados disponíveis para o mesmo num local conhecido. Os apontadores de ambientes precisam ser estabelecidos de forma a habilitar o procedimento chamado a ter acesso aos dados nos blocos envolventes. O estado do procedimento chamador precisa ser salvo de forma que o mesmo possa reassumir a execução após a chamada. O endereço de retorno também é salvo numa localização conhecida, a localização para a qual a rotina chamada precisa transferir o controle de volta após ter-se encerrado. O endereço de retorno é usualmente a localização de uma instrução que segue a chamada no procedimento chamador. Finalmente, um salto para o início do código para o procedimento chamado precisa ser gerado.

Quando o procedimento retorna, várias ações precisam ter lugar. Se o procedimento chamado for uma função, o resultado precisa ser armazenado num local conhecido. O registro de ativação do procedimento chamado precisa ser restaurado. Um salto para o endereço de retorno no procedimento chamador precisa ser gerado.

Não existe divisão exata das tarefas entre o procedimento chamador e o chamado. Freqüentemente, a linguagem-fonte, a máquina-alvo e o sistema operacional impõem exigências que favorecem uma solução em relação a outra.

Um Exemplo Simples

Vamos considerar um exemplo simples no qual os parâmetros são transmitidos por referência e a memória é alocada estaticamente. Nesta situação, podemos usar os próprios enunciados `param` como guardadores de lugar para os argumentos. Ao procedimento chamado, é passado um registrador contendo um apontador para o primeiro dos enunciados `param`, e o procedimento chamado pode então obter os apontadores para quaisquer de seus argumentos através do uso do deslocamento adequado a partir desse apontador-base. Ao se gerar código de três endereços para esse tipo de chamada, é suficiente gerar os enunciados de três endereços necessitados para avaliar aqueles argumentos que sejam expressões que não nomes simples e, então, segui-los por uma lista de enunciados de três endereços `param`, um para cada argumento. Se não desejamos misturar os enunciados que avaliam os parâmetros com os enunciados `param`, teremos que salvar o valor de *E.local*, para cada expressão *E* existente em **id** *(E, E, ..., E)*.[7]

[6] Usamos o termo procedimento incluídas aí as funções. Uma função é um procedimento que retorna um valor.

[7] Se os parâmetros forem transmitidos para o procedimento chamado numa pilha, como seria normalmente o caso para os dados alocados dinamicamente, não existe razão para não se misturar os enunciados de avaliação e enunciados `param`. O enunciado `param` é substituído em tempo de geração de código pelo código para empilhar um parâmetro na pilha.

Uma estrutura de dados conveniente na qual podemos salvar esses valores é a fila, uma lista FIFO (primeiro a entrar, primeiro a sair). Nossa rotina semântica para *lista_E* → *lista_E, E* irá incluir um passo para armazenar *E.local* numa fila chamada *queue*. Em seguida, a rotina semântica para *S* → **call id** *(lista_E)* irá gerar um enunciado `param` para cada item em *queue*, fazendo com que esses enunciados sigam os que avaliam as expressões de argumentos. Aqueles enunciados foram gerados quando os próprios argumentos foram reduzidos a *E*. A seguinte tradução dirigida pela sintaxe incorpora essas idéias.

(1) *S* → **call id** *(lista_E)*
{ **para** cada item *p* em *queue* **faça**
emitir ('`param`' *p*);
emitir ('`call`' **id.***local)* }*

O código para *S* é o código para *lista_E*, que avalia os argumentos, seguido por um enunciado `param` *p* para cada argumento, seguido por um enunciado de chamada de procedimento `executar`. Não é gerada uma contagem do número de parâmetros com o enunciado `executar`, mas a mesma poderia ser calculada da forma que computamos *lista_E.ndim* na seção anterior.

(2) *lista_E* → *lista_E, E*
{ atrelar *E.local* ao final de *queue* }
(3) *lista_E* → *E*
{ inicializar *queue* de forma a conter somente *E.local* }

Aqui, *queue* é esvaziada e então obtém um único apontador para a localização da tabela de símbolos para o nome que denota o valor de *E*.

EXERCÍCIOS

8.1 Traduzir a expressão aritmética `a * - (b + c)` numa
(a) árvore sintática
(b) notação pós-fixa
(c) código de três endereços

8.2 Traduza a expressão `- (a + b) * (c + d) + (a + b + c)` em:
(a) quádruplas
(b) triplas
(c) triplas indiretas

8.3 Traduza em enunciados executáveis o seguinte programa C:

```
main ()
{
        int i;
        int a[10];
        i = 1;
        while (i <= 10) {
            a[i] = 0; i = i + 1;
        }
}
```

em:
(a) árvore sintática
(b) notação pós-fixa
(c) código de três endereços

***8.4** Prove que, se todos os operadores são binários, uma cadeia de operadores e operandos é uma expressão pós-fixa se e somente se (1) existir exatamente um operador a menos do que o número de operandos e (2) cada prefixo não vazio da expressão possuir menos operadores do que operandos.

8.5 Modifique o esquema de tradução da Fig. 8.11, que computa os tipos e endereços relativos dos nomes declarados, de forma a permitir listas de nomes em lugar de nomes isolados em declarações da forma *D* → **id** : *T*.

8.6 A forma *prefixa* de uma expressão, na qual o operador θ é aplicado a expressões $e_1, e_2, ..., e_k$, é $\theta p_1 p_2 ... p_k$, onde p_i é a forma prefixa de e_i.
a) Gere a forma prefixa de `a * - (b + c)`
**b) Mostre que as expressões infixas não podem ser traduzidas na forma prefixa por esquemas de tradução nos quais todas as ações sejam de impressão e apareçam nas extremidades dos lados direitos das produções.
c) Forneça uma definição dirigida pela sintaxe para traduzir expressões infixas na forma prefixa. Quais dos métodos do Capítulo 5 você pode usar?

8.7 Escreva um programa para implementar a definição dirigida pela sintaxe para a tradução de expressões booleanas em código de três endereços, dada na Fig. 8.24.

8.8 Modifique a definição dirigida pela sintaxe da Fig. 8.24 para que gere código para a máquina de pilha da Seção 2.8.

8.9 A definição dirigida pela sintaxe da Fig. 8.24 traduz *E* → $\mathbf{id}_1 < \mathbf{id}_2$ no par de enunciados

if $\mathbf{id}_1 < \mathbf{id}_2$ **goto** ...
goto ...

Poderíamos, ao invés, traduzi-la no enunciado único

if $\mathbf{id}_1 \geq \mathbf{id}_2$ **goto-**

e seguirmos em frente pelo código para *E* quando o mesmo for verdadeiro. Modifique a definição na Fig. 8.24 para gerar código dessa natureza.

8.10 Escreva um programa para implementar a definição dirigida pela sintaxe da Fig. 8.23 para os enunciados de fluxo de controle.

8.11 Escreva um programa para implementar o algoritmo de retrocorreção dado na Seção 8.6.

8.12 Traduza o seguinte enunciado de atribuição em código de três endereços usando o esquema de tradução da Seção 8.3.

```
A[i,j] := B[i,j] + C[A[k,l] ] + D[i+j]
```

***8.13** Algumas linguagens, tais como PL/I, permitem que a uma lista de nomes seja fornecida uma lista de atributos e também que as declarações seja aninhadas umas nas outras. A seguinte gramática abstrai o problema:

D → *lista_de_nomes lista_de_atributos*
| *(D) lista_de_atributos*
lista_de_nomes → **id,** *lista_de_nomes*
| **id**
lista_de_atributos → *A lista_de_atributos*
| *A*
A → **decimal** | **fixed** | **float** | **real**

O significado de *D* → *(D) lista_de_atributos* é que a todos os nomes mencionados na declaração entre parênteses sejam associados os atributos em *lista_de_atributos*, não importa quantos níveis de aninhamento existam. Note-se que uma declaração de *n* nomes e *m* atributos pode fazer com que *nm* itens de informação sejam introduzidos na tabela de símbolos. Forneça uma definição dirigida pela sintaxe para as declarações desta gramática.

8.14 Em C, o enunciado `for` possui a seguinte forma:

`for` (e_1 ; e_2; e_3) *cmd*

*Em nosso pseudocódigo, o enunciado `executar` é o equivalente ao `call`. (N. do T.)

Tomando o significado como sendo

```
e1;
while ( e2 ) {
    cmd;
    e3;
}
```

construa uma definição dirigida pela sintaxe para traduzir enunciados `for` ao estilo de C em código de três endereços.

8.15 O Pascal padrão define o enunciado

for *v* := *valor_inicial* **to** *valor_final* **do** *cmd*

como tendo o mesmo significado que a seguinte seqüência de código:

```
início
    t1 := valor_inicial; t2 := valor_final;
    se t1 ≤ t2 então início
        v := t1;
        cmd
        enquanto v ≠ t2 faça início
            v := succ(v);
            cmd
        fim
    fim
fim
```

a) Considere o seguinte programa Pascal:

```
program lacofor (input, output);
    var i, inicial, final: integer;
    begin
        read (inicial, final);
        for i:= inicial to final do
            writeln(i)
    end.
```

Que comportamento esse programa terá para *inicial* = `MAXINT`, − 5 e *final* = `MAXINT`, onde `MAXINT` é o maior inteiro na máquina-alvo?

*b) Construa uma definição dirigida pela sintaxe que gere o código de três endereços correto para os enunciados `for` em Pascal.

NOTAS BIBLIOGRÁFICAS

UNCOL (para Universal Compiler Oriented Language — Linguagem Universal Orientada para Compiladores)— é uma linguagem intermediária universal mítica, pesquisada desde meados da década de 50. Dada uma UNCOL, o relatório de comitê por Strong et al. [1958] mostrava como os compiladores poderiam ser construídos, juntando-se a interface de vanguarda para uma dada linguagem-fonte com uma interface de retaguarda para uma dada linguagem-alvo. As técnicas de *bootstrapping* fornecidas no relatório são rotineiramente usadas para redirecionar compiladores para novas máquinas (ver a Seção 11.2). Steel [1961] contém uma proposta original para UNCOL.

Um compilador reorientável consiste em uma interface de vanguarda que pode ser colocada junto com várias interfaces de retaguarda que implementam uma dada linguagem em várias máquinas. Neliac é um exemplo primordial de linguagem com um compilador reorientável (Huskey, Halstead e McArthur [1960]), escrito em sua própria linguagem. Ver também Richards [1971], para uma descrição de um compilador reorientável para BCPL, Nori el at. [1981] para Pascal e Johnson [1979] para C. Newey, Poole e Waite [1972] aplicam a idéia de se mudar a interface de retaguarda para um macro processador, um editor de texto e um compilador *Basic*.

O ideal de UNCOL de implementar *n* linguagens em *m* máquinas escrevendo *n* interfaces de vanguarda e *m* interfaces de retaguarda em oposição a $n \times m$ compiladores distintos tem sido abordado de diversas maneiras. Uma dessas abordagens é se adaptar uma interface de vanguarda para uma nova linguagem sobre um compilador já existente. Feldman [1979b] descreve a adição de uma interface de vanguarda para Fortran 77 aos compiladores C por Johnson [1979] e Ritchie [1979]. Organizações de compiladores projetadas para acomodar múltiplas interfaces de vanguarda e de retaguarda são descritas por Davidson e Fraser [1984b], Leverett et al. [1980] e Tanenbaum et al. [1983].

Os termos "união" e interseção" de máquinas abstratas usados em Davidson e Fraser [1984b] destacam o papel do conjunto de operadores permitidos numa representação intermediária. O conjunto de instruções e modos de endereçamentos de uma máquina-interseção são limitados e, por conseguinte, as interfaces de vanguarda não têm que fazer muitas escolhas ao gerar o código intermediário. As máquinas-união providenciam formas alternativas de implementação de construções ao nível do código-fonte. Uma vez que nem todas as alternativas podem ser implementadas diretamente por todas as máquinas-alvo, o conjunto mais rico da máquina-união pode permitir uma dependência da máquina-alvo, de forma a se começar do zero e crescer a partir daí. Comentários similares se aplicam a outros tipos de código intermediário, tais como árvores sintáticas e código de três endereços. Fraser e Hanson [1982] consideram formas de expressar o acesso à pilha em tempo de execução usando operações independentes da máquina.

A implementação de Algol 60 é discutida em detalhes por Randell e Russell [1964] e Grau, Hill e Langmaack [1967]. Freiburghouse [1969] discute PL/I, Wirth [1971] Pascal e Branquart et al. [1976] Algol 68.

Minker e Minker [1980] e Giegerich e Wilhelm [1978] discutem a geração de código ótimo para expressões booleanas. O exercício 8.15 é proveniente de Newey e Waite [1985].

CAPÍTULO 9

GERAÇÃO DE CÓDIGO

A fase final em nosso modelo de compilador é o gerador de código. Recebe como entrada a representação intermediária do programa-fonte e produz como saída um programa-alvo equivalente, como indicado na Fig. 9.1. As técnicas de geração de código apresentadas neste capítulo podem ser usadas haja ou não uma fase de otimização antes da geração de código, como nos assim chamados compiladores "otimizantes". Uma tal fase tenta transformar o código intermediário numa forma a partir da qual um código-alvo mais eficiente possa ser produzido. Falaremos a respeito da otimização de código em detalhes no próximo capítulo.

As exigências tradicionalmente impostas a um gerador de código são severas. O código de saída precisa ser correto e de alta qualidade, significando que o mesmo deve tornar efetivo o uso dos recursos da máquina-alvo. Sobretudo, o próprio gerador de código deve rodar eficientemente.

Matematicamente, o problema de se gerar um código ótimo não pode ser solucionado. Na prática, devemos nos contentar com técnicas heurísticas que geram um código bom, mas não necessariamente ótimo. A escolha dos métodos heurísticos é importante, na medida em que um algoritmo de geração de código cuidadosamente projetado pode produzir um código que seja várias vezes mais rápido do que aquele produzido por um algoritmo concebido às pressas.

9.1 TEMAS NO PROJETO DE UM GERADOR DE CÓDIGO

Enquanto os detalhes são dependentes da máquina-alvo e do sistema operacional, temas como a gerência de memória, seleção de instruções, alocação de registradores e a ordem de avaliação são inerentes a quase todos os problemas de geração de código. Nesta seção, iremos examinar os temas genéricos do projeto de geradores de código.

Entrada para o Gerador de Código

A entrada para o gerador de código consiste na representação intermediária do programa-fonte, produzida pela vanguarda do compilador, juntamente com as informações na tabela de símbolos, que são usadas para determinar os endereços, em tempo de execução, dos objetos de dados denotados pelos nomes na representação intermediária.

Como notamos no capítulo anterior, existem várias escolhas para as linguagens intermediárias, incluindo as representações lineares, como a notação posfixa, representações de três endereços, como as quádruplas, representações de máquina virtual, como o código de máquina de pilha, e as representações gráficas, como as árvores sintáticas e os GDAs. Apesar dos algoritmos deste capítulo serem alinhavados em termos de código de três endereços, árvores e GDAs, muitas das técnicas também se aplicam a outras representações intermediárias.

Assumimos que a geração prévia de código, a partir da vanguarda do compilador, analisou léxica e sintaticamente o programa-fonte, bem como o traduziu numa forma razoavelmente detalhada de representação intermediária, de forma que os nomes que figuram na linguagem intermediária possam ser representados por quantidades que a máquina-alvo possa diretamente manipular (*bits*, inteiros, reais, apontadores etc.). Também assumimos que a necessária verificação de tipos já teve lugar, de forma que os operadores de conversão de tipo já foram inseridos onde quer que fossem necessários e que os erros semânticos óbvios (por exemplo, tentar indexar um *array* através de um número em ponto flutuante) já foram detectados. A fase de geração de código pode, por conseguinte, prosseguir na suposição de que a sua entrada está livre de erros. Em alguns compiladores, esse tipo de verificação semântica é feito junto com a geração de código.

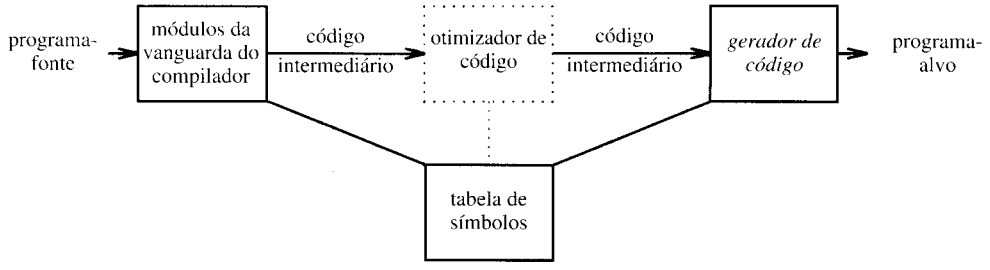


Fig. 9.1 Posição do gerador de código.

Programas-Alvo

A saída do gerador de código é o programa-alvo. Como o código intermediário, essa saída pode assumir uma variedade de formas: linguagem absoluta de máquina, linguagem relocável de máquina ou linguagem de montagem.

Como saída, a produção de um programa em linguagem absoluta de máquina possui a vantagem do mesmo poder ser carregado numa localização fixa de memória e executado imediatamente. Um pequeno programa pode ser compilado e imediatamente executado. Um certo número de compiladores para "trabalhos de estudantes", tais como WATFIV e PL/C, produz código absoluto.

A produção de um programa em linguagem relocável de máquina (módulo objeto) como saída permite que os subprogramas sejam compilados separadamente. Um conjunto de módulos-objeto relocáveis pode ser ligado e carregado para execução por um carregador/editor de ligações. Apesar de termos de pagar o preço adicional de editar as ligações, carregar o programa e produzir módulos-objeto relocáveis, ganhamos uma grande flexibilidade ao ficarmos aptos a compilar as rotinas separadamente e em chamar outros programas previamente compilados a partir de um módulo-objeto. Se a máquina-alvo não trata a relocação automaticamente, o compilador precisa providenciar informações explícitas para o carregador, a fim de ligar os segmentos de programa compilados separadamente.

A produção de um programa em linguagem de montagem como saída torna o processo de geração de código um tanto mais fácil. Podemos gerar instruções simbólicas e usar as facilidades de processamento de macros do montador para auxiliar a geração de código. O preço pago está no passo de montagem após a geração de código. Como a produção do código de montagem não duplica toda a tarefa do compilador, essa escolha é outra alternativa razoável, especialmente para uma máquina com uma memória pequena, onde o compilador precisa realizar diversas passagens. Neste capítulo, usamos o código de montagem como a linguagem-alvo por uma questão de legibilidade. Entretanto, poderíamos enfatizar que, na medida em que os endereços possam ser calculados a partir dos deslocamentos e de outras informações na tabela de símbolos, o gerador de código pode produzir endereços relocáveis ou absolutos para os nomes tão facilmente quanto endereços simbólicos.

Gerenciamento de Memória

O mapeamento dos nomes no programa-fonte para os endereços dos objetos de dados em tempo de execução é feito cooperativamente pela vanguarda e pelo gerador de código. No último capítulo, assumimos que um nome num enunciado de três endereços se referia a uma entrada para o nome na tabela de símbolos. Na Seção 8.2, as entradas na tabela de símbolos eram criadas à medida que as declarações iam sendo examinadas. O tipo numa declaração determina a largura, isto é, a quantidade de memória necessitada para o nome declarado. A partir das informações da tabela de símbolos, pode ser determinado um endereço relativo para o nome na área de dados para o procedimento. Na Seção 9.3, delineamos as implementações da alocação estática e de pilha para áreas de dados e mostramos como os nomes na representação intermediária podem ser convertidos em endereços no código-alvo.

Se deve ser gerado código de máquina, os rótulos nos enunciados de três endereços têm que ser convertidos para endereços de instruções. Esse processo é análogo à técnica de "retrocorreção" da Seção 8.6. Suponhamos que os rótulos se refiram a números de quádruplas num *array* de quádruplas. À medida que esquadrinhamos cada quádrupla, podemos deduzir a localização da primeira instrução de máquina gerada para aquela quádrupla mantendo uma contagem do número de palavras usadas para as instruções geradas até então. A contagem pode ser mantida no *array* de quádruplas (em um campo extra), de forma que se for encontrada uma referência tal como *j*:`goto` *i* e *i* for menor do que *j*, o número da quádrupla corrente, podemos simplesmente gerar uma instrução de desvio com o endereço-alvo igual à localização de máquina da primeira instrução no código para a quádrupla *i*. Se, entretanto, o desvio é para adiante, e, dessa forma, *i* excede *j*, precisamos armazenar, numa lista para quádrupla *i*, a localização da primeira instrução gerada para a quádrupla *j*. Então, ao processarmos a quádrupla *i*, preenchemos a localização adequada em todas as instruções que sejam desvios adiante para *i*.

Seleção de Instruções

A natureza do conjunto de instruções da máquina-alvo determina a dificuldade da seleção de instruções. A uniformidade e completeza do conjunto de instruções são fatores importantes. Se a máquina-alvo suporta cada tipo de dado de uma maneira uniforme, cada exceção à regra geral requer um tratamento especial.

A velocidade das instruções e os dialetos de máquina são fatores importantes. Se não nos importarmos com a eficiência do programa-alvo, a seleção de instruções é um processo direto. Para cada tipo de instrução de três endereços, podemos projetar um esqueleto de código que delineie o código-alvo a ser gerado para aquela construção. Por exemplo, cada enunciado de três endereços da forma `x:= y+z`, onde `x`, `y` e `z` são alocados estaticamente, pode ser traduzido na seqüência de código

```
MOV y,R0   /* carregar y no registrador R0 */
ADD z,R0   /* adicionar z a R0 */
MOV R0,x   /* armazenar R0 em x */
```

Infelizmente, esse tipo de geração de código enunciado a enunciado freqüentemente produz um código de baixa qualidade. Por exemplo, a seqüência de enunciados

```
a := b + c
d := a + e
```

seria traduzida em

```
MOV b,R0
ADD c,R0
MOV R0,a
MOV a,R0
ADD e,R0
MOV R0,d
```

Aqui, o quarto enunciado é redundante, como também o será o terceiro, caso `a` não venha a ser utilizado subseqüentemente.

A qualidade do código gerado é determinada por sua velocidade e tamanho. Uma máquina-alvo, com um rico conjunto de instruções, pode providenciar várias formas de se implementar uma dada operação. Uma vez que as diferenças de custo entre as diferentes implementações podem ser significativas, uma tradução ingênua do código intermediário pode levar a um código-alvo correto, porém inaceitavelmente ineficiente. Por exemplo, se a máquina-alvo possui uma instrução de "incremento" `(INC)`, o enunciado de três endereços `a := a + 1` pode ser implementado mais eficientemente pela instrução singela `INC a` do que por uma seqüência mais óbvia que carregue `a` num registrador, adicione um ao mesmo e, em seguida, armazene o resultado de volta em `a`:

```
MOV a,R0
ADD #1,R0
MOV R0,a
```

As velocidades das instruções são necessárias para se projetar boas seqüências de código, mas, infelizmente, informações acuradas a respeito da cronometrização das instruções são freqüentemente difíceis de se obter. Decidir que seqüência de código de máquina é a melhor para uma dada construção de três endereços requer, também, o conhecimento

a respeito do contexto no qual a instrução aparece. As ferramentas para se construir os seletores de instruções são discutidas na Seção 9.12.

Alocação de Registradores

As instruções envolvendo operadores do tipo registrador são usualmente mais curtas do que aquelas envolvendo operandos na memória. Por conseguinte, a utilização eficiente dos registradores é particularmente importante na geração de código de boa qualidade. O uso dos registradores é freqüentemente subdividido em dois subproblemas:

1. Durante a *alocação de registradores*, selecionamos o conjunto de variáveis que residirão nos registradores a um determinado ponto do programa.
2. Durante a fase subseqüente de *atribuição de registradores*, obtemos o registrador específico no qual a variável irá residir.

Encontrar uma atribuição ótima para os registradores é difícil, ainda que com valores únicos de registradores. Matematicamente, o problema é NP-completo.* O problema é adicionalmente complicado porque o *hardware* e/ou sistema operacional podem exigir que certas convenções de uso dos registradores sejam observadas.

Certas máquinas requerem *pares de registradores* (um registrador de número par e o registrador seguinte de número ímpar) para guardar alguns operandos e resultados. Por exemplo, nas máquinas do Sistema IBM/370, a multiplicação e a divisão inteiras envolvem pares de registradores. A instrução de multiplicação é da forma

```
M  x,y
```

onde `x` é o multiplicando e está num registrador par de uma dupla par/ímpar de registradores. O valor do multiplicando é obtido a partir do registrador ímpar da dupla. O multiplicador `y` é um único registrador. O produto ocupa todo a dupla par/ímpar de registradores.

A instrução de divisão é da forma

```
D  x,y
```

onde o dividendo de 64 *bits* ocupa a dupla par/ímpar de registradores, onde `x` é o registrador par; `y` representa o divisor. Após a divisão, o registrador par abriga o resto e o ímpar o quociente.

Agora, consideremos as duas seqüências de código de três endereços na Fig. 9.2(a) e (b), na qual a única diferença é o operador do segundo enunciado. As menores seqüências de código de montagem para (a) e (b) são fornecidas na Fig. 9.3.

`Ri` está em lugar de registrador *i*. (`SRDA R0,32` desloca o dividendo para `R1` e limpa `R0`, de forma que todos os seus *bits* fiquem iguais ao *bit* de sinal antes da limpeza). `L`, `ST` e `A` figuram em lugar de *load* (carregar), *store* (armazenar) e *add* (adicionar), respectivamente. Note-se que a escolha ótima para o registrador no qual `a` deve ser carregado depende do que acontecer a `t`, em última análise. As estratégias para a alocação de registradores são discutidas na Seção 9.7.

Escolha da Ordem de Avaliação

A ordem na qual as computações são realizadas pode afetar a eficiência do código-alvo. Algumas computações requerem menos registradores para abrigar resultados intermediários do que outras, como teremos oportunidade de examinar. A obtenção da melhor seqüência é um outro difícil problema NP-completo. Inicialmente, evitaremos o problema gerando o código para os enunciados de três endereços na ordem em que foram produzidos pelo gerador de código intermediário.

```
t := a + b
t := t * c
t := t / d
```

(a)

```
t := a + b
t := t + c
t := t / d
```

(b)

Fig. 9.2 Duas seqüências de código de três endereços.

Enfoques para a Geração de Código

Indubitavelmente, o critério mais importante para um gerador de código é que produza um código correto. A correção ganha significado acentuado por causa do número de casos especiais com que o gerador de código pode se defrontar. Dado o prêmio da correção, o projeto de um gerador de código, de forma a que possa ser facilmente implementado, testado e mantido, é uma importante meta de projeto.

A Seção 9.6 contém um algoritmo direto para a geração de código que usa a informação sobre os usos subseqüentes de um operando para gerar código para uma máquina de registradores. O algoritmo considera um enunciado por vez, mantendo os operandos em registradores na medida do possível. A saída de um tal gerador de código pode ser melhorada através de técnicas de otimização *peephole*, como aquelas discutidas na Seção 9.9.

A Seção 9.7 apresenta técnicas para fazer um melhor uso dos registradores através da consideração do fluxo de controle no código intermediário. A ênfase está na alocação de registradores para operandos pesadamente utilizados em laços internos.

As Seções 9.10 e 9.11 apresentam algumas técnicas de seleção de código voltadas para árvores que facilitam a construção de geradores de código reorientáveis. Várias versões do PCC, o compilador C portável, com tais geradores de código têm sido instaladas em numerosas máquinas. A disponibilidade do sistema operacional UNIX numa variedade de máquinas se deve em muito à portabilidade do PCC. A Seção 9.12 mostra como a geração de código pode ser tratada como um processo de reescrita.

9.2 A MÁQUINA-ALVO

Uma familiaridade com a máquina-alvo e seu conjunto de instruções é um pré-requisito para o projeto de um bom gerador de código. Infelizmente, numa discussão geral da geração de código não é possível descrever as nuances de qualquer máquina-alvo com detalhes suficientes para ficarmos aptos a gerar um código de boa qualidade para uma linguagem completa, naquela máquina. Neste capítulo, usaremos como computador-alvo uma máquina de registradores que é representativa para vários minicomputadores. No entanto, as técnicas de geração de

```
L   R1, a
A   R1, b
M   R0, c
D   R0, d
ST  R1, t
```

(a)

```
L     R0, a
A     R0, b
A     R0, c
SRDA  R0, 32
D     R0, d
ST    R1, t
```

(b)

Fig. 9.3 Seqüências ótimas de código de máquina.

*NP-completo é uma tradução abreviada para a classe de problemas não-determinísticos de tempo polinomial completo, a qual é considerada conter somente problemas matematicamente intratáveis, isto é, todos os algoritmos para resolvê-los têm pelo menos uma complexidade exponencial de tempo. O Capítulo 10 de Aho, Hopcroft e Ullman [1974] é um bom texto no assunto. (N. do T.)

[1]*Shift Right Double Arithmetic*, isto é, Deslocamento Aritmético Duplo para a Direita. (Nota original dos autores, expandida pelo tradutor.)

código apresentadas neste capítulo também têm sido usadas em muitas outras classes de máquinas.

Nosso computador-alvo é uma máquina endereçável em nível de *byte*, com quatro *bytes* por palavra, e *n* registradores de propósito geral, `R0, R1, ... , Rn-1`. Possui instruções de dois endereços da forma

op origem, destino

na qual *op* é o código da operação e *origem* e *destino* são campos de dados. A máquina possui os seguintes códigos de operação (dentre outros):

`MOV`	(copiar *origem* para *destino*)
`ADD`	(adicionar *origem* a *destino*)
`SUB`	(subtrair *origem* de *destino*)

Outras instruções serão introduzidas na medida do necessário.

Os campos origem e destino não são largos o suficiente para abrigar endereços de memória, e, dessa forma, determinados padrões de *bits* nesses campos especificam que as palavras seguintes a uma instrução contêm operandos e/ou endereços. A origem e o destino de uma instrução são especificados pela combinação de registradores e localizações de memória com os modos de endereçamento. Na descrição que se segue, *conteúdo (a)* denota o conteúdo de um registrador ou memória representada por *a*.

Os modos de endereçamento, juntamente com suas formas na linguagem de montagem e custos associados, são como se segue:

MODO	FORMA	ENDEREÇO	CUSTO ADICIONADO
absoluto	`M`	`M`	1
registrador	`R`	`R`	0
indexado	*c* `(R)`	*c* + *conteúdo* `(R)`	1
registrador indireto	`*R`	*conteúdo* `(R)`	0
indexado indireto	`*`*c* `(R)`	*conteúdo (c+conteúdo* `(R))`	1

Uma localização de memória `M` ou um registrador `R` representam a si mesmos quando usados como origem ou destino. Por exemplo, a instrução

```
MOV  R0,M
```

armazena o conteúdo do registrador `R0` na localização de memória `M`.

Um deslocamento de endereço *c* a partir do valor no registrador `R` é escrito *c* `(R)`. Por conseguinte,

```
MOV  4(R0),M
```

armazena o valor

conteúdo (4 + *conteúdo* `(R0)`)

na localização de memória `M`.

Versões indiretas dos últimos dois modos de endereçamento são indicadas pelo prefixo `*`. Por conseguinte,

```
MOV *4(R0),M
```

armazena o valor

conteúdo (conteúdo (4 + *conteúdo* `(R0)`))

na localização de memória `M`.

Um modo final de endereçamento permite que a origem seja uma constante:

MODO	FORMA	CONSTANTE	CUSTO ADICIONAL
literal	`#`*c*	*c*	1

Por conseguinte, a instrução

```
MOV  #1,R0
```

carrega a constante 1 no registrador `R0`.

Custos das Instruções

Fazemos o custo de uma instrução ser um mais os custos associados aos modos de endereçamento da origem e do destino (indicados como "custos adicionados" na tabela para os modos de endereçamento anterior). Esse custo corresponde ao comprimento (em palavras) da instrução. Os modos de endereçamento envolvendo registradores têm custo zero, enquanto que aqueles com uma localização de memória ou literal possuem custo um, porque tais operandos precisam ser armazenados junto com a instrução.

Se o espaço for importante, teremos claramente que minimizar o comprimento das instruções. No entanto, agir assim traz dois benefícios adicionais importantes. Para a maioria das máquinas e instruções, o tempo tomado para se carregar uma instrução da memória excede o tempo gasto executando-se a instrução. Por conseguinte, através da minimização do comprimento da instrução também tendemos a minimizar o tempo gasto para executar a instrução igualmente.[2] Alguns exemplos seguem.

1. A instrução `MOV R0,R1` copia o conteúdo do registrador `R0` no registrador `R1`. Essa instrução possui custo um, pois ocupa somente uma palavra de memória.

2. A instrução (de armazenamento) `MOV R5, M` copia o conteúdo do registrador `R5` na localização de memória `M`. Essa instrução tem custo 2, uma vez que o endereço da localização de memória `M` está na palavra que se segue à instrução.

3. A instrução `ADD #1,R3` adiciona a constante 1 ao conteúdo do registrador 3 e possui custo dois, uma vez que a constante 1 precisa aparecer na próxima palavra que se segue à instrução.

4. A instrução `SUB 4(R0),*12(R1)` armazena o valor

conteúdo (conteúdo (12 + *conteúdo* `(R1)`)) −*conteúdo (conteúdo* (4+`R0`))

no destino `*12(R1)`. O custo da instrução é três, pois as constantes 4 e 12 são armazenadas nas duas próximas palavras que se seguem à instrução.

Algumas das dificuldades para se gerar código para esta máquina podem ser enxergadas considerando-se o que deve ser produzido para um enunciado de três endereços da forma `a := b + c`, onde `b` e `c` são variáveis simples residindo em localizações de memória distintas, denotadas por esses nomes. Esse enunciado pode ser implementado atra-

[2] O critério de custo pretende ser instrutivo ao invés de realista. Permitir uma palavra completa para uma instrução simplifica a regra para se determinar o custo. Uma estimativa mais acurada do tempo gasto por uma instrução deveria considerar se a mesma requer que o valor de um operando, bem como de seu endereço (encontrado na instrução), seja carregado da memória.

vés de muitas seqüências diferentes de instruções. Aqui estão alguns exemplos:

```
1.  MOV   b,R0
    ADD   c,R0          custo = 6
    MOV   R0,a

2.  MOV   b,a
    ADD   c,a           custo = 6
```

Assumindo que `R0`, `R1` e `R2` contenham os endereços de `a`, `b` e `c`, respectivamente, podemos usar:

```
3.  MOV   *R1,  *R0
    ADD   *R2,  *R0     custo = 2
```

Assumindo que `R1` e `R2` contenham os valores de `b` e de `c`, respectivamente, e que o valor de `b` não é necessitado após a atribuição, podemos usar:

```
4.  ADD   R2,   R1
    MOV   R1,   a       custo = 3
```

Podemos ver que para gerarmos um código de boa qualidade para esta máquina precisamos utilizar suas capacidades de endereçamento eficientemente. Existe um prêmio para se manter o valor-*l* ou o valor-*r* de um nome num registrador, se possível, caso o mesmo venha a ser usado num futuro próximo.

9.3 GERENCIAMENTO DE MEMÓRIA EM TEMPO DE EXECUÇÃO

Como vimos no Capítulo 7, a semântica dos procedimentos numa linguagem determina como os nomes são amarrados à memória durante a execução. As informações necessitadas durante a execução de um procedimento são mantidas num bloco de memória chamado de registro de ativação; o armazenamento para os nomes locais ao procedimento também aparecem no registro de ativação.

Nesta seção, discutimos que código gerar para gerenciar os registros de ativação em tempo de execução. Na Seção 7.3, duas estratégias padrão de alocação de memória são apresentadas nominalmente: a alocação estática e a alocação de pilha. Na alocação estática, a posição de um registro de ativação na memória é fixada em tempo de compilação. Na alocação de pilha, um novo registro de ativação é empilhado para cada execução de um procedimento. O registro é desempilhado quando a ativação termina. Posteriormente, nesta seção, consideraremos como o código para um procedimento pode se referir aos objetos de dados no registro de ativação.

Como vimos na Seção 7.2, um registro de ativação para um procedimento possui campos para abrigar parâmetros, resultados, informações sobre o estado da máquina, dados locais, temporários e assemelhados. Nesta seção, ilustramos as estratégias de alocação usando um campo de estado da máquina para abrigar o endereço de retorno e o campo para os dados locais. Assumimos que os outros campos são manipulados como discutido no Capítulo 7.

Uma vez que a alocação e a liberação de registros de ativação ocorrem como parte das seqüências de chamada e retorno, focalizamos os seguintes enunciados de três endereços:

1. `call`,
2. `return`,
3. `halt`, e
4. `action`, um marcador de lugar para outros enunciados.

Por exemplo, o código de três endereços para os procedimentos `c` e `p`, na Fig. 9.4, contém exatamente esses tipos de enunciados. O tamanho e a disposição de dados dos registros de ativação são comunicados ao gerador de código através de informações, a respeito dos nomes, os quais estão na tabela de símbolos. Por uma questão de clareza, mostramos a disposição de dados na Fig. 9.4, ao invés da forma das entradas da tabela de símbolos.

Assumimos que a memória em tempo de execução seja dividida em área para código, dados estáticos e uma pilha, como na Seção 7.2 (a área adicional para um *heap* naquela seção não é usada aqui).

Alocação Estática

Consideremos o código necessário para implementar a alocação estática. Um enunciado `call` no código intermediário é implementado por uma seqüência de duas instruções da máquina-alvo. Uma instrução `MOV`, que salva o endereço de retorno, e um `GOTO`, que transfere o controle para o código-alvo do procedimento chamado:

```
MOV     # aqui+20, chamado.área_estática
GOTO    chamado.área_de_código
```

Os atributos *chamado.área-estática* e *chamado.área-de-código* são constantes se referindo, respectivamente, ao endereço do registro de ativação e à primeira instrução do procedimento chamado. O código-fonte #*aqui* + 20 na instrução `MOV` é o literal endereço de retorno; é o endereço da instrução seguinte à instrução `GOTO`. (A partir da discussão na Seção 9.2, as três constantes mais as duas instruções na seqüência de chamada custam cinco palavras ou 20 *bytes*.)

O código para um procedimento termina com um retorno para o procedimento chamador, exceto quando o primeiro procedimento não possui chamador, de modo que sua instrução final é um `HALT` (parar), que presumivelmente retorna o controle para o sistema operacional. Um retorno a partir do procedimento *chamado* é implementado por

```
GOTO    *chamado.área_estática
```

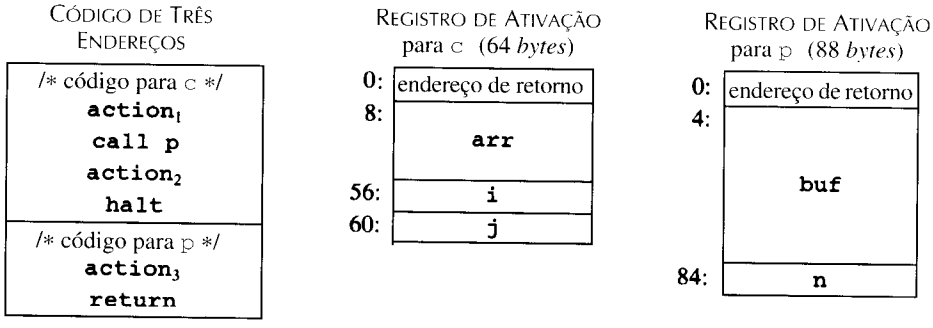


Fig. 9.4 Entrada para o gerador de código.

```
                                    /* código para c */
100:    ACTION1
120:    MOV  #140, 364              /* guarda o endereço de retorno 140 */
132:    GOTO 200                    /* chamar p */
140:    ACTION2
160:    HALT
        . . .

                                    /* código para p */
200:    ACTION3
220:    GOTO *364                   /* retornar para endereço salvo na
                                       localização 364 */
        . . .

                                    /* 300-363 abriga o registro de ativação para c */
300:                                /* endereço de retorno */
304:                                /* dados locais para c */

        . . .

                                    /* 364-451 abriga o registro de ativação para p */
364:                                /* endereço de retorno */
368:                                /* dados locais para p */
```

Fig. 9.5 Código-alvo para a entrada da Fig. 9.4.

que transfere o controle para o endereço salvo ao início do registro de ativação.

Exemplo 9.1 O código na Fig. 9.5 é construído a partir dos procedimentos `c` e `p` na Fig. 9.4. Usamos a pseudo-instrução `ACTION` para implementar o enunciado `action`, que representa um código de três endereços que não é relevante para esta discussão. Começamos arbitrariamente o código para esses procedimentos nos endereços 100 e 200, respectivamente, e assumimos que cada instrução `ACTION` ocupe 20 *bytes*. Os registros de ativação para os procedimentos são alocados estaticamente, começando na localização 300 e 364, respectivamente.

As instruções que começam no endereço 100 implementam os enunciados

```
action1; call p; action2; halt
```

do primeiro procedimento `c`. A execução, por conseguinte, começa com a instrução $\texttt{ACTION}_1$ no endereço 100. A instrução `MOV` no endereço 120 salva o endereço de retorno 140 no campo de estado de máquina, que está à primeira palavra no registro de ativação de `p`. A instrução `GOTO`, no endereço 132, transfere o controle para a primeira instrução do código-alvo do procedimento chamado.

Uma vez que 140 foi salvo ao endereço 364 pela seqüência de chamada acima, `*364` representa 140 quando o enunciado `GOTO`, no endereço 220, for executado. O controle retorna, por conseguinte, para o endereço 140 e a execução do procedimento `c` é retomada. □

Alocação de Pilha

A alocação estática pode se transformar em alocação de pilha pelo uso de endereços relativos no armazenamento dentro dos registros de ativação. A posição do registro para uma ativação de um procedimento não é conhecida até o tempo de execução. Na alocação de pilha, esta posição é usualmente armazenada num registrador, de forma que as palavras no registro de ativação podem receber acesso através de um deslocamento contado a partir do valor no registrador. O modo de endereçamento indexado de nossa máquina-alvo é conveniente para esse propósito.

Os endereços relativos num registro de ativação podem ser considerados como deslocamentos contados a partir de qualquer posição conhecida no registro de ativação, como vimos na Seção 7.3. Por uma questão de conveniência, usaremos deslocamentos positivos mantendo no registrador `SP` um apontador para o início do registro de ativação que está ao topo da pilha. Quando ocorre uma chamada de procedimento, o procedimento chamador incrementa `SP` e transfere o controle para o procedimento chamado. Após o controle retornar para o chamador, o mesmo decrementa `SP`, liberando, por conseguinte, o registro de ativação do procedimento chamado.[3]

O código para o primeiro procedimento inicializa a pilha através do estabelecimento de SP com um valor apontando para o início da área da pilha na memória:

```
MOV  #início_da_pilha, SP          /* inicializar a pilha */
código para o primeiro procedimento
HALT                               /* terminar a execução */
```

Uma seqüência de chamada de procedimento incrementa `SP`, salva o endereço de retorno e transfere o controle para o procedimento chamado:

```
ADD   #chamador.tamanho_de_registro, SP
MOV   #aqui+16, *SP          /* salvar endereço de retorno */
GOTO  chamado.área_de_código
```

O atributo *chamador.tamanho_de_registro* representa o tamanho de um registro de ativação, de forma que a instrução `ADD` deixa o `SP` apontando para o início do próximo registro de ativação. A origem *#aqui* + 16 na instrução `MOV` é o endereço da instrução que se segue à instrução de desvio `GOTO`; é salvo no endereço apontado por `SP`.

A seqüência de retorno consiste em duas partes. O procedimento chamado transfere o controle para o endereço de retorno usando

```
GOTO  *0(SP)                /* retornar ao chamado */
```

a razão de se usar `*0(SP)` na instrução de desvio está em que precisamos de dois níveis de indireção: `0(SP)` é o endereço da primeira palavra no registro de ativação e `*0(SP)` é o endereço de retorno salvo lá.

[3]Com deslocamentos negativos, poderíamos ter `SP` apontando para o final da pilha e fazer o procedimento chamado incrementar `SP`.

A segunda parte da seqüência de retorno está no chamador, que decrementa SP e, dessa forma, restaura SP para o conteúdo prévio. Ou seja, após a subtração, SP aponta para o início do registro de ativação do chamador:

```
SUB    #chamador.tamanho_de_registro, SP
```

Uma discussão mais ampla das seqüências de chamada e das barganhas na divisão de tarefas entre os procedimentos chamador e chamado figura na Seção 7.3.

Exemplo 9.2 O programa na Fig. 9.6 é uma condensação do código de três endereços para o programa Pascal discutido na Seção 7.1. O procedimento q é recursivo e, por conseguinte, mais de uma ativação do mesmo pode estar viva ao mesmo tempo.

Suponhamos que os tamanhos dos registros de ativação para os procedimentos s, p e q tenham sido determinados em tempo de compilação como sendo *tam_s*, *tam_p* e *tam_q*, respectivamente. A primeira palavra em cada registro de ativação irá guardar um endereço de retorno. Assumimos arbitrariamente que o código para esses procedimentos começa nos endereços 100, 200 e 300, respectivamente, e que a pilha começa em 600. O código-alvo para o programa na Fig. 9.6 é como se segue.

CÓDIGO DE TRÊS ENDEREÇOS

/* código para s */ **action$_1$** **call q** **action$_2$** **halt**
/* código para p */ **action$_3$** **return**
/* código para q */ **action$_4$** **call p** **action$_5$** **call q** **action$_6$** **call q** **return**

Fig. 9.6 Código de três endereços para ilustrar a alocação de pilha.

```
                          /* código para s   */
100: MOV    #600, SP      /* inicializar a pilha */
108: ACTION1
128: ADD    #tam_s, SP    /* seqüência de chamada inicial */
136: MOV    #152, *SP     /* empilha endereço de retorno */
144: GOTO 300             /* chamar q */
152: SUB    #tam_s, SP    /* restaurar SP */
160: ACTION2
180: HALT
     . . .
                          /* código para p */
200: ACTION3
220: GOTO *0 (SP)         /* retornar */
     . . .
                          /* código para  q */
300: ACTION4              /* desvio condicional para 456 */
320: ADD    #tam_q, SP
328: MOV    #344, *SP     /* empilhar endereço de retorno */
336: GOTO   200,          /* chamada para p */
344: SUB    #tam_q SP
352: ACTION5
372: ADD    #tam_q, SP
380: MOV    #396, *SP     /* empilhar endereço de retorno */
388: GOTO   300           /* chamar q */
396: SUB    #tam_q, SP
404: ACTION6
424: ADD    #tam_q, SP
432: MOV    #448, *SP     /* empilhar endereço de retorno */
440: GOTO   300,          /* chamada para q */
448: SUB    #tam_q, SP
456: GOTO   *0(SP)        /* retornar */
     . . .

600                       /* a pilha começa aqui */
```

Assumimos que ACTION$_4$ contenha um desvio condicional para o endereço 456 da seqüência de retorno de q; caso contrário, o procedimento recursivo q está condenado a chamar a si próprio para sempre. No exemplo abaixo, consideramos a execução de um programa na qual a primeira chamada de q não retorna imediatamente; mas todas as chamadas subseqüentes o fazem.

Se *tam_s*, *tam_p* e *tam_q* são 20, 40 e 60, respectivamente, SP é inicializado em 600, o início da pilha, pela primeira instrução no endereço 100. SP está com 620, exatamente antes do controle se transferir de s para q, porque *tam_s* é 20. Subseqüentemente, quando q chama p, a instrução no endereço 320 incrementa SP para 680, onde o registro de ativação para q começa; SP tem seu valor revertido para 620 após o controle retornar para q. Se as duas próximas chamadas recursivas de q retornarem imediatamente, o valor máximo de SP durante essa execução será 680. Note-se, entretanto, que a última alocação usada na pilha foi 739, uma vez que o registro de ativação para q, começando na localização 680, se estende por 60 *bytes*. □

Endereços para os Nomes, em Tempo de Execução

A estratégia de alocação de memória em tempo de execução e a disposição de dados no registro de ativação para um procedimento determinam como a memória para os nomes recebe acesso. No Capítulo 8, assumimos que um nome num enunciado de três endereços é realmente um apontador para a entrada do nome na tabela de símbolos. Esse enfoque possui uma vantagem significativa; torna o compilador mais portável, uma vez que a interface de vanguarda não precisa ser modificada mesmo se o compilador for migrado para uma nova máquina onde uma diferente organização em tempo de execução é necessitada (por exemplo, o *display* pode ser mantido em registradores ao invés de na memória). Por outro lado, a seqüência específica de passos de acesso ao se gerar o código intermediário pode ser uma vantagem significativa num compilador otimizante, uma vez que permite ao compilador tirar vantagem de detalhes que sequer enxergaria num enunciado simples de três endereços.

Em qualquer caso, os nomes precisam ser eventualmente substituídos pelo código para se ter acesso às localizações de memória. Consideramos, por conseguinte, algumas elaborações do enunciado singelo de três endereços, de cópia, x := 0. Suponhamos que, após as declarações num procedimento terem sido processadas, a entrada da tabela de símbolos para x registre o endereço relativo 12, para x. Consideremos primeiro o caso em que x está numa área estaticamente alocada que começa no endereço *estático*. Por conseguinte, o endereço efetivo de x, em tempo de execução, é *estático* + 12. Apesar do compilador eventualmente determinar o valor de *estático* + 12 em tempo de compilação, a posição da área estática pode não ser conhecida quando o código intermediário para dar acesso ao nome for gerado. Nesse caso, faz sentido gerar um código de três endereços para "computar" *estático* + 12, com a compreensão de que essa computação será concluída

durante a geração de código ou, possivelmente, pelo carregador, antes do programa rodar. A atribuição `x := 0` se traduz em

```
estático[12] := 0
```

Se a área estática começar no endereço 100, o código-alvo para esse enunciado é

```
MOV    #0,    112
```

Por outro lado, suponhamos que nossa linguagem seja como Pascal e que um *display* seja usado para dar acesso aos nomes não locais, como discutido na Seção 7.4. Suponhamos, também, que o *display* seja mantido em registradores e que `x` seja local ao procedimento ativo cujo apontador do *display* está no registrador `R3`. Podemos, então, traduzir a cópia `x := 0` nos enunciados de três endereços

$$t_1 := 12 + R3$$
$$*t_1 := 0$$

no qual t_1 contém o endereço de `x`. Essa seqüência pode ser implementada pela única instrução de máquina

```
MOV    #0,    12(R3)
```

Note-se que o valor no registrador `R3` não pode ser determinado em tempo de compilação.

9.4 BLOCOS BÁSICOS E GRAFOS DE FLUXO

Uma representação de enunciados de três endereços, sob a forma de grafos, chamada de grafo de fluxo, é útil para a compreensão dos algoritmos de três endereços, mesmo que o grafo não seja explicitamente construído pelo algoritmo de geração de código. Os nós no grafo de fluxo representam computações e os lados representam o fluxo de controle. No Capítulo 10 usamos extensivamente o grafo de fluxo para um programa como um veículo para coletar informações a respeito do programa intermediário. Alguns algoritmos de alocação de registradores usam grafos de fluxo para encontrar os laços internos, nos quais um programa é esperado gastar a maior parte de seu tempo.

Blocos Básicos

Um *bloco básico* é uma seqüência de enunciados consecutivos, na qual o controle entra no início e o deixa no fim, sem uma parada ou possibilidade de ramificação, exceto ao final. A seguinte seqüência de enunciados de três endereços forma um bloco básico:

$$\begin{aligned} t_1 &:= a * a \\ t_2 &:= a * b \\ t_3 &:= 2 * t_2 \\ t_4 &:= t_1 + t_3 \\ t_5 &:= b * b \\ t_6 &:= t_4 + t_5 \end{aligned} \tag{9.1}$$

Um enunciado de três endereços `x := y + z` é dito *definir* `x` e *usar* (ou *referenciar*) `y` e `z`. Um nome num bloco básico é dito estar *vivo* a um dado ponto, se seu valor vier a ser usado após aquele ponto no programa, possivelmente em outro bloco básico.

O algoritmo seguinte pode ser usado para particionar uma seqüência de enunciados de três endereços em blocos básicos.

Algoritmo 9.1 Particionar uma seqüência de enunciados em blocos básicos.

Entrada. Uma seqüência de enunciados de três endereços.

```
begin
    prod := 0;
    i := 1;
    do begin
        prod := prod + a[i] * b[i];
        i := i + 1
    end
    while i <= 20
end
```

Fig. 9.7 Programa para computar o produto escalar.

Saída. Uma lista de blocos básicos, com cada enunciado de três endereços exatamente em um bloco.

Método.

1. Determinamos primeiro o conjunto de *líderes*, os primeiros enunciados dos blocos básicos. As regras que usamos são as seguintes.

   i) O primeiro enunciado é um líder.
   ii) Qualquer enunciado que seja objeto de um desvio condicional ou incondicional é um líder.
   iii) Qualquer enunciado que siga imediatamente um enunciado de desvio condicional ou incondicional é um líder.

2. Para cada líder, seu bloco básico consiste no líder e em todos os enunciados até, mas não incluindo o próximo líder ou o final do programa. □

Exemplo 9.3 Consideremos o fragmento de código-fonte mostrado na Fig. 9.7; o mesmo computa o produto escalar de dois vetores* `a` e `b` de comprimento 20. Uma lista de enunciados de três endereços que realiza essa computação em nossa máquina-alvo é mostrada na Fig. 9.8.

Vamos aplicar o Algoritmo 9.1 ao código de três endereços na Fig. 9.8, a fim de determinar seus blocos básicos. O enunciado (1) é o líder pela regra (i) e o enunciado (3) é um líder pela regra (ii), uma vez que o último enunciado pode saltar para o mesmo. Pela regra (iii), o enunciado seguindo a (12) (relembremos que a Fig. 9.13 é apenas um fragmento de programa) é um líder. Por conseguinte, os enunciados (1) e (2) formam um bloco básico. O resto do programa, começando com o enunciado (3), forma um segundo bloco básico. □

Transformações sobre os Blocos Básicos

Um bloco básico computa um conjunto de expressões. Essas expressões são os valores dos nomes vivos à saída de um bloco. Dois blocos básicos são ditos *equivalentes* se computarem o mesmo conjunto de expressões.

Um número de transformações pode ser aplicado a um bloco básico sem mudar o conjunto de expressões computadas pelo mesmo. Muitas dessas transformações são úteis para melhorar a qualidade do código que será gerado, em última análise, a partir de um bloco básico. No próximo capítulo, mostramos como um código global "otimizador" tenta usar tais transformações para reagrupar as computações num programa, num esforço para reduzir as exigências globais de tempo e espaço para a execução do programa-alvo final. Existem duas importantes classes de transformações locais que podem ser aplicadas aos blocos básicos; são as transformações estrutura invariantes e as algébricas.

*O somatório dos produtos dos elementos de mesmo índice em cada vetor. (N. do T.)

```
(1)  prod := 0
(2)  i := 1
(3)  t1 := 4 * i
(4)  t2 := a [ t1 ]          /* computar a[i] */
(5)  t3 := 4 * i
(6)  t4 := b [ t3 ]          /* computar b[i] */
(7)  t5 := t2 * t4
(8)  t6 := prod + t5
(9)  prod := t6
(10) t7 := i + 1
(11) i := t7
(12) if i <= 20 goto (3)
```

Fig. 9.8 Código de três endereços para computar o produto escalar.

Transformações Estrutura Invariantes

As transformações primárias estrutura invariantes sobre os blocos básicos são:

1. eliminação de subexpressões comuns
2. eliminação de código morto
3. renomeação de variáveis temporárias
4. intercâmbio de dois enunciados adjacentes independentes

Vamos agora examinar essas transformações em um pouco mais de detalhe. Nesse momento, assumimos que os blocos básicos não possuam *arrays*, apontadores ou chamadas de procedimentos.

1. *Eliminação de subexpressões comuns.* Consideremos o bloco básico

```
a := b + c
b := a - d
c := b + c
d := a - d
```
(9.2)

Os segundo e quarto enunciados computam a mesma expressão, nominalmente, `b + c - d`, e, por conseguinte, esse bloco básico pode ser transformado no bloco equivalente

```
a := b + c
b := a - d
c := b + c
d := b
```
(9.3)

Note-se que, apesar dos primeiro e terceiro enunciados em (9.2) e (9.3) parecerem ter a mesma expressão à direita, o segundo enunciado redefine `b`. Conseqüentemente, o valor de `b` no terceiro enunciado é diferente do valor de `b` no primeiro, e o primeiro e terceiro enunciados não computam a mesma expressão.

2. *Eliminação de código morto.* Suponhamos que `x` esteja morto, isto é, jamais será subseqüentemente usado, no ponto em que `x := y + z` aparece num bloco básico. Esse enunciado, então, pode ser removido com segurança, sem mudar o valor do bloco básico.

3. *Renomeação de variáveis temporárias.* Suponhamos ter um enunciado da forma `t := b + c`, onde `t` é um temporário. Se mudarmos esse enunciado para `u := b + c`, onde `u` é uma nova variável temporária, e mudarmos todos os usos dessa instância de `t` para `u`, o valor desse bloco básico não é mudado. De fato, podemos sempre transformar um bloco básico num bloco equivalente no qual cada enunciado que defina um temporário passe a definir um outro novo temporário. Chamamos a um tal bloco de bloco básico *forma normal*.

4. *Intercâmbio de enunciados.* Suponhamos ter um bloco com os dois enunciados adjacentes

```
t1 := b + c
t2 := x + y
```

Podemos, então, intercambiar os dois enunciados sem afetar o valor do bloco se e somente se nem `x` nem `y` forem t_1 e nem `b` nem `c` forem t_2. Note-se que um bloco básico em forma normal permite todas as trocas de enunciados possíveis.

Transformações Algébricas

Transformações algébricas incontáveis podem ser usadas para mudar o conjunto de expressões computadas por um bloco básico num conjunto algebricamente equivalente. As transformações úteis são aquelas que simplificam expressões ou substituem operações caras por outras mais em conta. Por exemplo, um enunciado como

```
x := x + 0
```

ou

```
x := x * 1
```

podem ser eliminados de um bloco básico sem modificar o conjunto de expressões que o mesmo computa. O operador de exponenciação no enunciado

```
x := y ** 2
```

usualmente requer uma chamada de função para ser implementado. Usando-se uma transformação algébrica, esse enunciado pode ser substituído pelo enunciado mais barato, porém equivalente

```
x := y * y
```

As transformações algébricas são discutidas com mais detalhes na Seção 9.9 sobre a otimização *peephole* e na Seção 10.3 sobre otimização de blocos básicos.

Grafos de Fluxo

Podemos adicionar informações a respeito do fluxo de controle no conjunto de blocos básicos, que constituem um programa, através da construção de um grafo direcionado chamado *grafo de fluxo*. Os nós do grafo de fluxo são os blocos básicos. Um nó é distinguido como *inicial*; é o bloco cujo líder é o primeiro enunciado. Existe um lado dirigido do bloco B_1 para o bloco B_2 se B_2 pode se seguir imediatamente a B_1 em alguma seqüência de execução; isto é, se

1. existir um enunciado de desvio condicional ou incondicional, do último enunciado de B_1 para o primeiro enunciado de B_2, ou

2. B_2 seguir imediatamente a B_1 na ordem do programa e B_1 não terminar por um desvio incondicional.

Dizemos que B_1 é um *predecessor* de B_2 e que B_2 é um *sucessor de* B_1.

Exemplo 9.4 O grafo de fluxo do programa da Fig. 7.9 é mostrado na Fig. 9.9. B_1 é o nó inicial. Note-se que no último enunciado o comando de desvio (3) foi substituído por um desvio equivalente para o início do bloco B_2. □

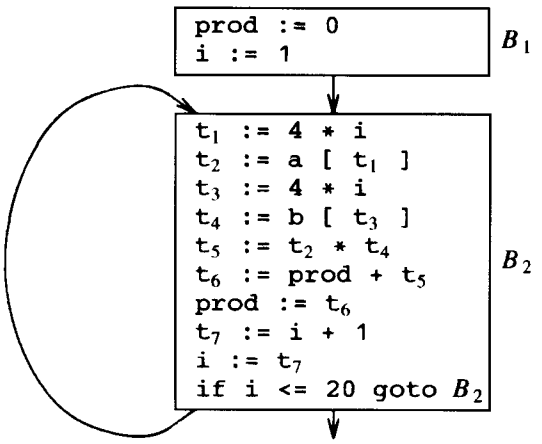


Fig. 9.9 Grafo de fluxo para o programa.

Representação dos Blocos Básicos

Os blocos básicos podem ser representados por uma variedade de estruturas de dados. Por exemplo, após particionar os enunciados de três endereços através do Algoritmo 9.1, cada bloco básico pode ser representado por um registro que consiste em um contador do número de quádruplas no bloco, seguido por um apontador para o líder (primeira quádrupla) do bloco e pelas listas de predecessores e sucessores do bloco. Uma alternativa é se fazer uma lista ligada das quádruplas em cada bloco. Referências explícitas aos números de quádruplas em enunciados de desvio ao final dos blocos básicos podem causar problemas se as quádruplas forem movidas durante a otimização de código. Por exemplo, se o bloco B_2, que abrange os enunciados de (3) a (12) no código intermediário da Fig. 9.9, fosse movido para algum outro lugar no *array* de quádruplas ou fosse encolhido, o (3) no enunciado `if i <= 20 goto` (3) teria que ser modificado. Por conseguinte, preferimos fazer com que os desvios apontassem para os blocos em lugar das quádruplas, como fizemos na Fig. 9.9.

É importante notar que, num grafo de fluxo, um lado proveniente de um bloco B para um bloco B' não especifica as condições sob as quais o controle flui de B para B'. Isto é, o lado não nos diz se o desvio condicional ao final de B (se existir um desvio condicional lá) vai para o líder de B' quando a condição for satisfeita ou quando não for. Essa informação pode ser recuperada quando necessário a partir do enunciado de desvio em B.

Laços

Num grafo de fluxo, o que é um laço e como se encontram todos os laços? Na maioria das vezes é fácil se responder a essa questão. Por exemplo, na Fig. 9.9 existe um laço, que consiste no bloco B_2. As respostas gerais a essas questões, entretanto, são um tanto sutis, e iremos examiná-las em detalhes no próximo capítulo. Para o presente, é suficiente dizer que um laço é uma coleção de nós num grafo de fluxo tal que

1. Todos os nós na coleção são *fortemente conectados;* isto é, a partir de qualquer nó no laço, em direção a qualquer outro, existe um percurso de comprimento um ou mais, totalmente dentro do laço, e

2. A coleção de nós possui uma única *entrada*, isto é, um nó do laço, tal que a única forma de se atingir qualquer nó do laço a partir de um nó externo ao mesmo consiste em se dirigir primeiro ao nó da entrada.

Um laço que não contenha outros laços é chamado de um laço *interno*.

9.5 INFORMAÇÕES DE USO SUBSEQÜENTE

Nesta seção, coletamos informações de uso subseqüente a respeito dos nomes nos blocos básicos. Se um nome associado a um registrador não é mais necessário, o registrador pode ser associado a algum outro nome. Essa idéia de se manter um nome na memória somente se o mesmo vier a ser subseqüentemente usado pode ser aplicada em vários contextos. Foi usada na Seção 5.8 para atribuir espaço aos valores de atributos. O gerador de código simples da próxima seção a aplica à atribuição de registradores. Como uma aplicação final, consideramos a atribuição de memória para nomes temporários.

Computando os Usos Subseqüentes

O *uso* de um nome num enunciado de três endereços é definido como se segue. Suponhamos que o enunciado de três endereços i atribua um valor a `x`. Se o enunciado j tem `x` como um operando e o controle fluir do enunciado i para j ao longo de um percurso que não tenha atribuições intervenientes a `x`, podemos dizer que o enunciado j *usa* o valor de `x` computado em i.

Desejamos determinar, para cada enunciado de três endereços `x := y` *op* `z`, quais são os usos subseqüentes de `x`, `y` e `z`. Para o presente, não estamos preocupados com usos fora do bloco básico que contém esse enunciado de três endereços; mas podemos, se o desejarmos, tentar determinar se existe ou não um tal uso através da técnica de análise das variáveis vivas do Capítulo10.

Nosso algoritmo para determinar os usos subseqüentes realiza uma passagem para trás sobre cada bloco básico. Podemos esquadrinhar facilmente um fluxo de enunciados de três endereços para encontrar as extremidades dos blocos básicos, como no Algoritmo 9.1. Uma vez que os procedimentos podem ter efeitos colaterais arbitrários, assumimos, por uma questão de conveniência, que cada chamada de procedimento inicie um novo bloco básico.

Tendo encontrado o final de um bloco básico, esquadrinhâmolo para trás até o início, registrando (na tabela de símbolos), para cada nome `x`, se o mesmo possui um uso subseqüente no bloco e, caso não o possua, se está vivo à saída daquele bloco. Se a análise de fluxo de dados discutida no Capítulo 10 tiver sido realizada, saberemos que nomes estarão vivos à saída de cada bloco. Se nenhuma análise de variáveis vivas tiver sido feita, podemos supor, assumindo uma postura conservativa, que todos os nomes não-temporários estejam vivos à saída de cada bloco. Se os algoritmos que geram o código intermediário ou que o otimizam permitem que certos temporários sejam usados através dos blocos, os últimos também precisam ser considerados vivos. Seria uma boa idéia marcar quaisquer desses temporários, de forma a não termos de considerar todos os temporários como vivos.

Vamos supor que atingimos o $i$ésimo enunciado de três endereços `x := y` *op* `z`, em nosso esquadrinhamento para trás. Fazemos, então, o seguinte.

1. Atrelamos ao enunciado i as informações, correntemente encontradas na tabela de símbolos, relativas ao uso subseqüente e prova de vida* de `x`, `y` e `z`.[4]
2. Na tabela de símbolos, façamos `x` "não vivo" e "sem uso subseqüente".
3. Na tabela de símbolos, façamos `y` e `z` "vivos" e os usos subseqüentes de `y` e `z` iguais a i. Note-se que a ordem de passos (2) e (3) não pode ser mudada, já que `x` pode ser `y` ou `z`.

Se o enunciado de três endereços i for da forma `x := y` ou `x :=` *op* `y`, os passos são os mesmos, ignorando-se `z`.

*Do original em inglês: *liveness*. (N. do T.)

[4]Se `x` não estiver vivo, esse enunciado pode ser removido; tais transformações serão consideradas na Seção 9.8.

Memória para Nomes Temporários

Apesar de ser útil, num compilador otimizante, criar um nome distinto a cada vez que um nome temporário for necessitado (ver o Capítulo 10 para a justificativa), o espaço precisa ser reservado para abrigar os valores desses temporários. O comprimento da área reservada para os temporários (campo) no registro geral de ativação da Seção 7.2 cresce com o número de temporários.

Podemos, em geral, mapear dois temporários para uma mesma localização (isto é, compactá-los) se não estiverem vivos simultaneamente. Uma vez que quase todos os temporários são definidos e usados dentro dos blocos básicos, as informações de uso subseqüente podem ser aplicadas para compactá-los. Para os temporários que são usados através dos blocos, o Capítulo 10 discute a análise de fluxo de dados necessitada para computar a prova de vida.

Podemos reservar localizações de memória para temporários examinando um a cada vez, atribuindo um temporário à primeira localização que não contenha um temporário vivo, no campo para temporários. Se a um temporário não puder ser atribuída nenhuma localização de memória previamente criada, adicionamos uma nova localização para a área de dados associada ao procedimento corrente. Em muitos casos, os temporários podem ser compactados em registradores ao invés de localizações de memória, como na próxima seção.

Por exemplo, os seis temporários no bloco básico (9.1) podem ser compactados em duas localizações. Essas localizações correspondem a t_1 e t_2 em:

```
t1  :=  a  *  a
t2  :=  a  *  b
t2  :=  2  *  t2
t1  :=  t1 +  t2
t2  :=  b  *  b
t1  :=  t1 +  t2
```

9.6 UM GERADOR DE CÓDIGO SIMPLES

A estratégia de geração de código desta seção produz um código-alvo para uma seqüência de enunciados de três endereços. Considera um enunciado por vez, lembrando se algum dos operandos está correntemente em registradores e tirando vantagem desse fato se possível. Por uma questão de simplicidade, assumimos que para cada operador num enunciado exista um operador correspondente na linguagem-alvo. Assumimos igualmente que os resultados computados possam ser deixados em registradores na medida do possível, armazenando-os somente (a) se os registradores de seus resultados forem necessários em outra computação ou (b) exatamente antes de uma chamada de procedimento, desvio ou enunciado rotulado.[5]

A condição (b) implica que tudo precisa ser armazenado exatamente antes do final de cada bloco básico.[6] A razão pela qual precisamos fazê-lo é que, após deixarmos um bloco básico, precisamos estar aptos a ir a vários blocos básicos distintos ou para um bloco básico particular que possa ser atingido a partir de vários outros. Em qualquer caso, não podemos, sem esforço extra, assumir que um dado usado por um bloco apareça no mesmo registrador, não importa como o controle tenha atingido esse bloco. Dessa forma, para evitar um possível erro, nosso algoritmo gerador de código simples armazena tudo quanto se desloca através dos limites dos blocos básicos, bem como quando as chamadas de procedimentos são feitas. Posteriormente, consideraremos formas para guardar alguns dados em registradores através dos limites dos blocos.

Podemos produzir um código razoável para um enunciado de três endereços `a := b + c` se gerarmos uma única instrução `ADD Rj, Ri` com custo um, deixando o resultado `a` num registrador `Ri`. Essa seqüência é possível somente se o registrador `Ri` contiver `b`, `Rj` contiver `c` e `b` não estiver vivo após o enunciado; isto é, `b` não é usado após o enunciado.

Se `Ri` contiver `b`, mas `c` estiver numa localização de memória (chamada `c` por conveniência), podemos gerar a seqüência

```
ADD c, Ri                    custo = 2
```

ou

```
MOV     c,Rj
ADD     Rj,Ri                custo = 3
```

uma vez providenciado que `b` não esteja subseqüentemente vivo. A segunda seqüência se torna atraente se esse valor de `c` for subseqüentemente usado, na medida em que pudermos obter seu valor a partir do registrador `Rj`. Existem muitos outros casos para considerar, dependendo de onde `b` e `c` estejam correntemente localizados e se o valor de `b` é subseqüentemente usado. Precisamos também considerar os casos em que um ou ambos, `b` e `c`, sejam constantes. O número de casos que precisam ser posteriormente considerados aumenta se assumirmos que o operador `+` seja comutativo. Por conseguinte, podemos ver que a geração de código envolve o exame de um amplo número de casos, e que caso deve prevalecer depende do contexto no qual um enunciado de três endereços é examinado.

Descritores de Registradores e de Endereços

O algoritmo de geração de código usa descritores para controlar o conteúdo dos registradores e dos endereços para os nomes.

1. Um descritor de registradores controla o que está correntemente em cada registrador. É consultado sempre que um novo registrador é necessitado. Assumimos que inicialmente o descritor de registradores informe que todos os registradores estão vazios (se os registradores forem atribuídos através dos blocos, esse não será o caso). À medida que a geração de código progride, cada registrador irá abrigar o valor de zero ou mais nomes a um dado instante.

2. Um descritor de endereços controla a localização (ou localizações) onde o valor corrente de um nome pode ser encontrado em tempo de execução. A localização poderia ser um registrador, uma localização da pilha, um endereço de memória ou algum conjunto desses elementos, já que, uma vez copiado, o valor também permanece onde estava originalmente. Essas informações podem ser armazenadas na tabela de símbolos e são usadas para determinar o método de acesso para um nome.

Um Algoritmo de Geração de Código

O algoritmo de geração de código toma como entrada uma seqüência de enunciados de três endereços que constituem um bloco básico. Para cada enunciado de três endereços da forma `x := y` *op* `z`, realizamos as seguintes ações:

1. Invocamos a função *obter_reg* para determinar a localização `L` onde o resultado do cômputo `y` *op* `z` deverá ficar armazenado. `L` usualmente será um registrador, mas poderia ser também uma localização de memória. Descreveremos *obter_reg* em breve.

2. Consultamos o endereço do descritor para `y` de forma a determinar `y'`, a localização corrente de `y` (ou uma das suas localizações). Pre-

[5] No entanto, para se produzir um *dump simbólico*, que torna disponíveis os valores das localizações de memória e dos registradores em termos dos nomes no programa-fonte, pode ser mais conveniente ter as variáveis definidas pelo programa (mas não necessariamente os temporários gerados pelo compilador) armazenadas imediatamente após o cômputo, dada a iminência de um erro, o qual provoca subitamente uma interrupção precipitada e o encerramento do programa.

[6] Note-se que não estamos assumindo que as quádruplas foram efetivamente particionadas em blocos básicos pelo compilador; a noção de bloco básico é conceitualmente útil em qualquer instância.

ferir para y' o registrador, caso o seu valor esteja tanto na memória quanto num registrador. Se o valor de y ainda não estiver em L, gerar a instrução MOV y', L a fim de colocar uma cópia de y em L.

3. Gerar a instrução OP z', L onde z' é a localização corrente de z. De novo, preferir um registrador em vez de uma localização de memória, se z estiver em ambas. Atualizar o descritor de endereço de x para indicar que x está na localização L. Se L for um registrador, atualizar o descritor para indicar que o mesmo contém o valor de x e remover x de todos os demais descritores de registradores.

4. Se os valores correntes de y e/ou z não possuírem usos subseqüentes, não estão, por conseguinte, vivos à saída do bloco, e estão em registradores; alterar, nesse caso, o descritor de registradores para indicar que, após a execução de x := y *op* z, aqueles registradores não irão conter y e/ou z, respectivamente.

Se o enunciado de três endereços corrente possuir um operador unário, os passos são análogos àqueles acima e, por conseguinte, omitimos os detalhes. Um caso especial importante é o enunciado de três endereços x := y. Se y estiver num registrador, mudar simplesmente o descritor de registradores e de endereços, de modo a que informem que o valor de x é agora encontrado somente no registrador que abriga o valor de y. Se y não tiver uso subseqüente e não estiver vivo à saída do bloco, o registrador já não abriga o valor de y.

Se y estiver somente na memória, podemos, em princípio, registrar que o valor de x está na localização de y, mas essa opção iria complicar nosso algoritmo, uma vez que não poderíamos então mudar o valor de y sem salvar o valor de x. Por conseguinte, se y estiver na memória, usamos *obter_reg* para encontrar um registrador no qual carregar y e fazer do mesmo a nova localização de x.

Alternativamente, podemos gerar uma instrução MOV y,x, o que seria preferível caso o valor de x não tenha uso subseqüente no bloco. É valioso notar que a maioria das instruções de cópia, senão todas, será eliminada se usarmos o algoritmo de aprimoramento de blocos e propagação de cópias do Capítulo 10.

Uma vez que tenhamos processado todos os enunciados de três endereços num bloco básico, armazenamos, através de instruções MOV, aqueles nomes que estão vivos à saída e não estão ainda em suas localizações de memória. Para realizar isso, usamos o descritor de registradores para determinar quais nomes foram deixados em registradores, o descritor de endereços para determinar se o mesmo nome já não está em sua localização de memória e as informações sobre as variáveis vivas para determinar se um nome deve ou não ser armazenado. Se nenhuma informação sobre as variáveis vivas foi computada pela análise de fluxo de dados entre os blocos, devemos assumir que todas as variáveis estão vivas ao final do bloco.

A Função *obter_reg*

A função *obter_reg* retorna a localização L para abrigar o valor de x para a atribuição x := y *op* z. Uma grande quantidade de esforços pode ser gasta na implementação desta função para produzir uma escolha perspicaz de L. Nesta seção, discutimos um esquema simples, fácil de implementar, baseado nas informações de uso subseqüente coletadas na última seção.

1. Se um nome y está num registrador que não abriga o valor de nenhum outro nome (relembremos que as instruções de cópia, tais como x := y, poderiam fazer com que um registrador guardasse o valor de duas ou mais variáveis simultaneamente) e y não está viva e não possui uso subseqüente após a execução de x := y *op* z, confiscar o registrador de y para L. Atualizar o descritor de endereço de y de forma a indicar que o mesmo já não está mais em L.

2. Em caso de (1) falhar, retornar um registrador vazio para L, se existir um.

3. Caso (2) falhe, se x possuir um uso subseqüente no bloco ou se *op* for um operador que requeira um registrador, tal como o de indexação, encontrar um registrador ocupado R. Armazenar o valor de R numa localização de memória (através de MOV R, M) se ainda não estiver na localização própria de memória M, atualizar o descritor de endereço de M e retornar R. Se R abrigar o valor de diversas variáveis, uma instrução MOV precisa ser gerada para cada variável que necessite ser armazenada. Um registrador ocupado adequado poderia ser um cujo conteúdo seja referenciado mais distantemente no futuro ou um cujo valor esteja também na memória. Deixamos a escolha exata inespecificada, já que não existe uma que tenha sido provada como a melhor forma de se fazer a seleção.

4. Se x não é usado no bloco ou nenhum registrador ocupado adequado puder ser encontrado, selecionar L como a localização de memória de x.

Uma função *obter_reg* mais sofisticada também consideraria os usos subseqüentes de x e a comutatividade do operador *op* ao determinar o registrador para abrigar o valor de x. Deixamos tais extensões de *obter_reg* como exercícios.

Exemplo 9.5. A atribuição d := (a − b) + (a − c) + (a − c) poderia ser traduzida na seguinte seqüência de código de três endereços

```
t := a − b
u := a − c
v := t + u
d := v + u
```

Enunciados	Código Gerado	Descritor de Registradores	Descritor de Endereços
		registradores vazios	
t := a − b	MOV a, R0 SUB b, R0	R0 contém t	t em R0
u := a − c	MOV a, R1 SUB c, R1	R0 contém t R1 contém u	t em R0 u em R1
v := t + u	ADD R1, R0	R0 contém v R1 contém u	u em R1 v em R0
d := v + u	ADD R1, R0 MOV R0, d	R0 contém d	d em R0 d em R0 e memória

Fig. 9.10 Seqüência de código.

ENUNCIADO	i NO REGISTRADOR Ri		i NA MEMÓRIA Mi		i NA PILHA	
	CÓDIGO	CUSTO	CÓDIGO	CUSTO	CÓDIGO	CUSTO
a:=b[i]	MOV b(Ri),R	2	MOV Mi,R MOV b (R),R	4	MOV Si(A),R MOV b(R),R	4
a[i]:=b	MOV b,a(Ri)	3	MOV Mi,R MOV b,a(R)	5	MOV Si(A),R MOV b,a(R)	5

Fig. 9.11 Seqüências de código para atribuições indexadas.

com d vivo ao fim. O algoritmo de geração de código dado acima produziria, para essa seqüência de enunciados de três endereços, a seqüência de código mostrada na Fig. 9.10. Mostrados juntos estão os valores aos descritores de registradores e de endereços, à medida que o processo de geração de código progride. Não mostrado no descritor de endereços é o fato de que a, b e c estão sempre na memória. Assumimos também que t, u e v, sendo temporários, não estão em memória, a menos que armazenemos explicitamente seus valores através de uma instrução MOV.

A primeira chamada de *obter_reg* retorna R0 como a localização na qual se deve computar t. Uma vez que a não está em R0, geramos as instruções MOV a, R0 e SUB b, R0. Atualizamos agora o descritor de registradores, indicando que R0 contém t.

A geração de código prossegue dessa maneira, até que o último enunciado de três endereços d := v + u tenha sido processado. Note-se que R1 se torna vazio porque u não possui uso subseqüente. Geramos, então, MOV R0,d para armazenar a variável viva d, ao final do texto do bloco.

O custo do código gerado na Fig. 9.10 é 12. Poderíamos reduzir isso para 11 gerando MOV R0,R1 imediatamente após a primeira instrução e removendo a instrução MOV a,R1, mas fazê-lo requer um algoritmo de geração de código mais sofisticado. A razão para os ganhos está em que é mais econômico carregar R1 a partir de R0 do que da memória. □

Gerando Código para Outros Tipos de Enunciados

As operações de indexação e de referenciamento nos enunciados de três endereços são tratadas da mesma maneira que as operações binárias. A tabela da Fig. 9.11 mostra as seqüências de código geradas para os enunciados de atribuição indexada a := b[i] e a[i] := b, assumindo que b seja estaticamente alocado.

A localização corrente de i determina a seqüência de código. Três casos são cobertos, dependendo de i estar no registrador Ri, na localização de memória Mi ou na pilha, com deslocamento Si e o apontador para o registro de ativação de i estar no registrador A. O registrador R é aquele retornado quando a função *obter_reg* é chamada. Para a primeira atribuição, preferiríamos deixar a num registrador R se a tivesse um uso subseqüente no bloco e o registrador R estivesse disponível. Na segunda atribuição, assumimos que a seja alocada estaticamente.

A tabela na Fig. 9.12 mostra as seqüências de código geradas para as atribuições de apontadores a:=*p e *p:=a. Aqui, a localização corrente de p determina a seqüência de código.

Três casos são cobertos, dependendo de p estar inicialmente num registrador Rp, na localização de memória Mp ou na pilha, com deslocamento Sp e o apontador para o registro de ativação para p estar no registrador A. O registrador R é o retornado quando a função *obter_reg* é chamada. Na segunda atribuição, assumimos que a seja alocada estaticamente.

Enunciados Condicionais

As máquinas implementam os desvios condicionais em uma dentre duas formas possíveis. Uma delas é desviar se o valor designado num registrador atender a uma das seis condições: negativo, zero, positivo, não negativo, não-zero e não-positivo. Numa tal máquina, um enunciado de três endereços, tal como if x < y goto z, pode ser implementado subtraindo-se y de x, no registrador R e, em seguida, desviando-se para z se o valor no registrador R for negativo.

Um segundo enfoque, comum a muitas máquinas, usa um conjunto de *códigos de condição* para indicar se a última quantidade computada ou carregada num registrador é negativa, zero ou positiva. Freqüentemente, uma instrução de comparação (CMP, em nossa máquina) possui a desejável propriedade de estabelecer o código de condição sem efetivamente computar um valor. Isto é, CMP x,y estabelece o código de condição para positivo se x > y, e assim por diante. Uma instrução de máquina de desvio condicional realiza o desvio se a condição designada, <, =, >, ≤, ≠, ou ≥ for atendida. Usamos a instrução CJ <= z para significar "desviar para z se o código de condição for negativo ou zero". Por exemplo, if x < y goto z poderia ser implementada através de

```
CMP  x  ,y
CJ<  z
```

ENUNCIADO	p NO REGISTRADOR Rp		p NA MEMÓRIA Mp		p NA PILHA	
	CÓDIGO	CUSTO	CÓDIGO	CUSTO	CÓDIGO	CUSTO
a:=*p	MOV *Rp,a	2	MOV Mp,R MOV *R,R	3	MOV Sp(A),R MOV *R,R	3
*p:=a	MOV a,*Rp	2	MOV Mp,R MOV a,*R	4	MOV a,R MOV R,*Sp(A)	4

Fig. 9.12 Seqüências de código para atribuição de apontadores.

Se estamos gerando código para uma máquina com códigos de condição, é útil manter-se um descritor do código de condição na medida em que gerarmos o código. Esse descritor informa o nome que estabeleceu o código de condição pela última vez, ou o par de nomes comparados, se o código de condição foi estabelecido dessa forma pela última vez. Por conseguinte, poderíamos implementar

```
x  := y + z
if x < 0 goto z
```

através de

```
MOV  y,  R0
ADD  z,  R0
MOV  R0,x
CJ<  z
```

se estivermos informados de que o código de condição foi determinado através do valor de `x` após `ADD z,R0`.

9.7 ALOCAÇÃO E ATRIBUIÇÃO DE REGISTRADORES

As instruções envolvendo somente operandos do tipo registrador são mais curtas e rápidas do que aquelas que envolvem operandos na memória. Por conseguinte, a utilização eficiente dos registradores é importante na geração de um código de boa qualidade. Esta seção apresenta várias estratégias para decidir que valores num programa deveriam residir em registradores (alocação de registradores) e em que registrador cada valor deveria residir (atribuição de registradores).

Uma abordagem para a alocação e atribuição de registradores é associar valores específicos num programa objeto a certos registradores. Por exemplo, uma decisão pode ser tomada para atribuir endereços base a um grupo de registradores, cômputos aritméticos a outro, o topo da pilha em tempo de execução a um registrador fixo e assim por diante.

Esse enfoque possui a vantagem de simplificar o projeto do compilador. Sua desvantagem está em que, aplicado excessivamente ao pé da letra, utiliza os registradores ineficientemente; certos registradores podem seguir sem uso ao longo de porções substanciais de código, enquanto que são geradas cargas e armazenamentos desnecessários de outros. Apesar de tudo, é razoável na maioria dos ambientes de computação se reservar uns poucos registradores como base, apontadores de pilha e semelhantes a fim de permitir que os registradores remanescentes possam ser usados pelo compilador, na medida em que sejam vistos como adequados.

Alocação Global de Registradores

O algoritmo de geração de código na Seção 9.6 usou os registradores para guardar valores pela duração de um único bloco básico. No entanto, todas as variáveis vivas foram armazenadas ao final de cada bloco. Para economizar alguns desses armazenamentos, e as cargas correspondentes, poderíamos programar para associar registradores às variáveis mais freqüentemente usadas e manter esses registradores consistentes ao longo dos limites dos blocos (*globalmente*). Uma vez que os programas gastam a maior parte do tempo nos seus laços internos, uma abordagem natural para a atribuição global de registradores é tentar manter um valor freqüentemente usado num registrador fixo ao longo de um laço. Para o momento, vamos assumir que conhecemos a estrutura de laço de um grafo de fluxo e que sabemos como os valores computados num bloco básico são usados fora do mesmo. O próximo capítulo cobre as técnicas para computar essas informações.

Uma estratégia para a alocação global de registradores é associar algum número fixo de registradores para guardar a maior parte dos valores ativos em cada laço mais interno. Os valores selecionados podem ser diferentes em diferentes laços. Os registradores ainda não alocados podem ser usados para guardar os valores locais a um bloco, como na Seção 9.6. Esse enfoque possui a desvantagem de que um número fixo de registradores não é sempre o número certo para se deixar disponível para a alocação global de registradores. Ainda assim o método é simples de implementar e foi usado em Fortran H, o compilador otimizante para a série de máquinas IBM/360 (Lowry e Medlock [1969]).

Em linguagens como C e Bliss, o programador pode realizar diretamente alguma alocação de registradores através de declarações de forma a manter certos valores em registradores pela duração de um procedimento. O uso judicioso das declarações de registradores, pode acelerar muitos programas, mas o programador não deveria realizá-lo antes de estabelecer o perfil de comportamento do programa.

Contadores de Utilização

Um método simples para se determinar as economias a serem realizadas mantendo-se uma variável x num registrador pela duração de um laço L é reconhecer que em nosso modelo de máquina economizamos uma unidade de custo para cada referência a x, se x estiver num registrador. Entretanto, se usarmos o enfoque da seção anterior para gerar o código de um bloco, existe uma boa chance de que x, após ter sido computado num bloco, venha a permanecer num registrador, se existirem usos subseqüentes de x naquele bloco. Por conseguinte, contamos com uma economia de um para cada uso de x no laço L, que não seja precedido por uma atribuição a x no mesmo bloco. Também economizamos duas unidades se pudermos evitar um armazenamento de x ao final de um bloco. Por conseguinte, se x for alocado a um registrador, contamos com um ganho de dois para cada bloco de L para o qual x esteja vivo à saída e no qual x receba a atribuição de um valor.

No lado do débito, se x estiver vivo à entrada do cabeçalho do laço, precisamos carregar x em seu registrador exatamente antes de entrar no laço L. Isso custa duas unidades. Semelhantemente, para cada bloco de saída B do laço L, ao qual x esteja vivo à entrada de algum sucessor de B fora de L, precisamos armazenar x a um custo de dois. Entretanto, na suposição de que um laço seja repetido muitas vezes, podemos negligenciar esses débitos, uma vez que ocorrem somente uma vez a cada vez que entramos no laço. Por conseguinte, uma fórmula aproximada para o lucro a ser realizado alocando-se um registrador a x dentro de um laço L é:

$$\sum_{\text{blocos } B \text{ em } L} (\textit{utilização}\ (x, B), + 2 * \textit{vivo}\ (x, B)) \qquad (9.4)$$

onde *utilização* (x, B) é o número de vezes que x é usado em B antes de qualquer definição de x; *vivo* (x, B) é 1 se x está vivo à saída de B e recebe uma atribuição de valor em B, e 0 caso contrário. Note-se que (9.4) é aproximada, porque nem todos os blocos num laço são executados com igual freqüência e também porque está baseada na suposição de que um laço é iterado "muitas" vezes. Em outras máquinas, teria que ser desenvolvida uma fórmula análoga e possivelmente diferente de (9.4).

Exemplo 9.6 Consideremos os blocos básicos do laço mais interno delineado na Fig. 9.13, onde os enunciados de desvio condicional e incondicional foram omitidos. Vamos assumir que os registradores `R0`, `R1` e `R2` são alocados para guardar valores ao longo do laço. Por uma questão de conveniência, as variáveis vivas à entrada e à saída de cada bloco são mostradas na Fig. 9.1, imediatamente acima e abaixo de cada bloco, respectivamente. Existem alguns pontos sutis a respeito das variáveis vivas, os quais endereçamos no Capítulo 10. Por exemplo, note-se que ambos, `e` e `f`, estão vivos ao final de B_1, mas desses, somente `e` está vivo à entrada de B_2 e somente `f` à entrada de B_3. Em geral, as variáveis vivas ao fim de um bloco são a união daquelas vivas ao início de cada um de seus blocos sucessores.

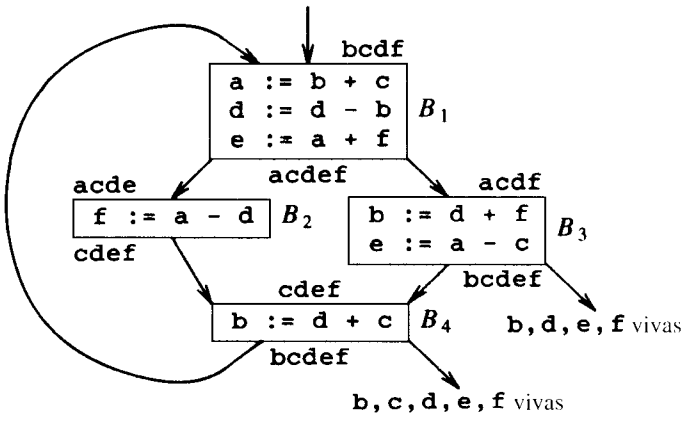


Fig. 9.13 Grafo de fluxo para um laço mais interno.

A fim de avaliar (9.4) para x = a, observamos que a está viva à saída de B_1 e que recebe um valor nele, mas não está viva à saída de B_2, B_3 ou B_4. Por conseguinte,

$$\sum_{B \text{ em } L} 2 * \textit{vivo}(\mathtt{a}\ B) = 2.$$

Igualmente, *utilização* (a, B_1) = 0, uma vez que a é definida em B_1 antes de qualquer uso. Igualmente, *utilização* (a, B_3) = 1 e *utilização* (a, B_4) = 0. Por conseguinte,

$$\sum_{B \text{ em } L} \textit{utilização}(\mathtt{a},\ B) = 2.$$

Dessa forma, o valor de (9.4) para x = a é 4. Isto é, quatro unidades de custo podem ser economizadas selecionando-se a para um dos registradores globais. Os valores de (9.4) para b, c, d, e e f são 6, 3, 6, 4 e 4, respectivamente. Por conseguinte, devemos selecionar a, b e d para os registradores R0, R1 e 2, respectivamente. Usar R0 para e ou f em lugar de a, seria uma outra escolha com o mesmo benefício aparente. A Fig. 9.4 mostra o código de montagem gerado a partir da Fig. 9.13, assumindo que a estratégia da Seção 9.6 é usada para gerar código para cada bloco. Não mostramos o código gerado para os desvios condicionais e incondicionais omitidos que terminam cada bloco na Fig. 9.13 e, por conseguinte, não mostramos o código gerado como um único fluxo, conforme o mesmo apareceria na prática. É importante notar que se não aderíssemos estritamente à nossa estratégia de reservar R0, R1 e R2, poderíamos usar

```
SUB    R2,    RO
MOV    R0,    f
```

para B_2, economizando uma unidade, uma vez que a não está viva à saída do bloco B_2. Um ganho similar poderia ser realizado em B_3. □

Atribuição de Registradores para Laços Mais Externos

Tendo atribuído os registradores e gerado código para os laços mais internos, podemos aplicar a mesma idéia para laços progressivamente maiores. Se um laço mais externo L_1 possui um laço mais interno L_2, os nomes alocados, que foram alocados a registradores em L_2, não precisam ter registradores alocados em L_1-L_2. No entanto, se um nome x tem um registrador alocado no laço L_1 mas não em L_2, precisamos armazenar x à entrada para L_2 e carregarmos x quando o deixarmos e entrarmos num bloco de L_1-L_2. Semelhantemente, se escolhermos alocar para x um registrador em L_2 mas não em L_1, precisamos carregar x à entrada para L_2 e armazenar x à saída. Deixamos como um exercício a derivação de um critério para selecionar nomes a terem registradores alocados num laço mais externo L, dado que as escolhas já tenham sido feitas para todos os laços aninhados dentro de L.

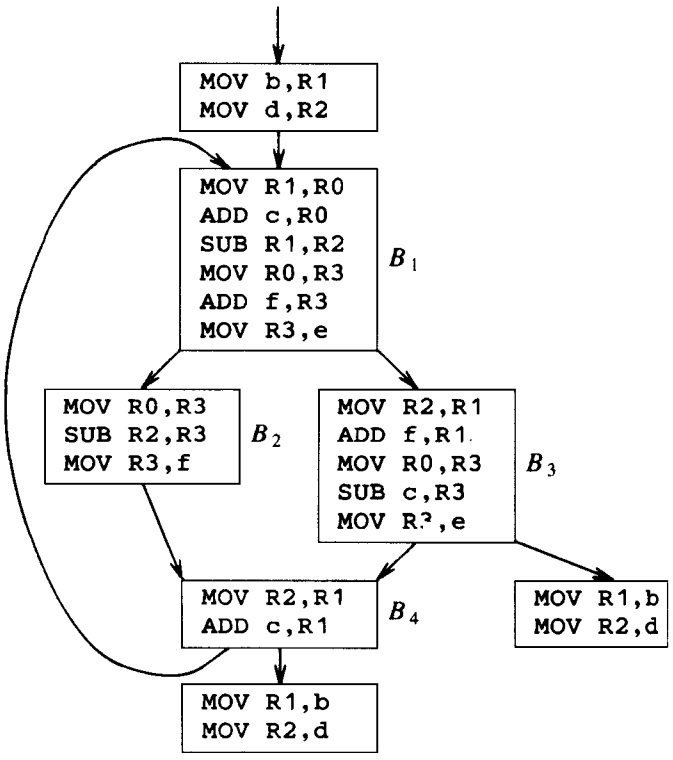


Fig. 9.14 Seqüência de código usando-se a atribuição global de registradores.

Alocação de Registradores Colorindo-se um Grafo

Quando um registrador é necessário para uma computação mas todos os registradores disponíveis estão em uso, o conteúdo de um dos registradores precisa ser armazenado (*drenado*)* numa localização de memória a fim de liberar um registrador. A coloração de grafos é uma técnica sistemática simples para se alocar registradores e manipular a drenagem dos mesmos.

Neste método, são usadas duas passagens. Na primeira, as instruções da máquina-alvo são selecionadas como se existisse um número infinito de registradores simbólicos; com efeito, os nomes usados no código intermediário se tornam os nomes de registradores e os enunciados de três endereços se tornam os enunciados da linguagem de máquina. Se o acesso às variáveis requer instruções que usem apontadores de pilha ou de *display*, registradores base ou outras quantidades que assistem ao acesso, assumimos que as mesmas sejam guardadas em registradores reservados para cada propósito. Normalmente, seus usos são diretamente traduzíveis num modo de acesso para o endereço mencionado na instrução de máquina. Se o acesso é mais complexo, o mesmo deve ser quebrado em várias instruções de máquina e um registrador simbólico temporário (ou vários) pode necessitar ser criado.

Uma vez que as instruções tenham sido selecionadas, uma segunda passagem atribui os registradores físicos aos simbólicos. A meta é encontrar uma atribuição que minimize o custo das drenagens.

Na segunda passagem, é construído um *grafo de interferência de registradores*, para cada procedimento, no qual os nós são os registradores simbólicos e um lado conecta dois nós se um estiver vivo no

*Do original em inglês: *spilled*. (N. do T.)

ponto em que o outro é definido. Por exemplo, um grafo de interferência de registradores para a Fig. 9.13 teria nós para os nomes a e d. No bloco B_1, a está vivo no segundo enunciado, o qual define d; por conseguinte, no grafo haveria um lado entre os nós para a e d.

É feita uma tentativa de colorir o grafo de interferência de registradores usando-se k cores, onde k é o número de registradores atribuíveis (um grafo é dito *colorido* se a cada nó foi associada uma cor, de tal forma que não haja dois nós adjacentes com cores iguais). Uma cor representa um registrador e o processo de colorimento assegura que não haja dois registradores simbólicos que possam interferir um com o outro e sejam associados ao mesmo registrador físico.

Apesar do problema de determinar se um grafo é colorível a k cores em geral ser NP-completo, a seguinte técnica heurística pode ser usualmente utilizada para realizar o colorimento de forma rápida na prática. Suponhamos que um nó n num grafo G possua menos do que k vizinhos (nós conectados a n através de um lado). Removamos n e seus lados de G, de forma a obtermos o grafo G'. Um colorimento-k de G' pode ser estendido a um colorimento-k de G atribuindo-se a n uma cor não associada a nenhum de seus vizinhos.

Através da eliminação repetida dos nós que tenham menos do que k lados a partir do grafo de interferência de registradores, obteremos ou um grafo vazio, caso em que podemos obter um colorimento-k do grafo original através do colorimento dos nós na ordem reversa à qual foram removidos, ou obteremos um grafo no qual cada nó possui k ou mais nós adjacentes. No último caso, um colorimento-k não é mais possível. A esse ponto, um nó é drenado através da introdução de código para armazenar e recarregar o registrador. O grafo de interferência é apropriadamente modificado e o processo de colorimento retomado. Chaitin [1982] e Chaitin et al. [1981] descrevem vários métodos heurísticos para escolher o nó a drenar. Uma regra geral é evitar introduzir código de drenagem nos laços mais internos.

9.8 A REPRESENTAÇÃO DE BLOCOS BÁSICOS SOB A FORMA DE GDAs

Grafos direcionados acíclicos (GDAs) são estruturas de dados úteis para implementar transformações sobre os blocos básicos. Um GDA fornece um quadro sobre como o valor computado por cada enunciado num bloco básico é usado nos enunciados subseqüentes do bloco. A construção de um GDA a partir dos enunciados de três endereços é uma boa forma de se determinar as subexpressões comuns (expressões computadas mais de uma vez) dentro de um bloco, os nomes que são usados dentro do bloco mas avaliados fora do mesmo, e que enunciados do bloco poderiam ter os seus valores computados sendo utilizados fora do mesmo.

Um GDA para um bloco básico (ou simplesmente *GDA*) é um grafo direcionado acíclico com os seguintes rótulos nos nós:

1. As folhas são rotuladas por identificadores únicos, quer nomes de variáveis ou constantes. A partir do operador aplicado a um nome, determinamos se o valor-l ou o valor-r de um nome é necessitado; a maioria das folhas representa valores-r. As folhas representam os valores iniciais dos nomes e os subscrevemos com 0 para evitar confusão com os rótulos denotando valores correntes de nomes como em (3) abaixo.
2. Os nós interiores são rotulados por um símbolo de operador.
3. Aos nós é também dada, opcionalmente, uma seqüência de identificadores como rótulos. A intenção é que os nós interiores representem valores computados e os identificadores que rotulem um nó sejam considerados ter aquele valor.

```
(1)   t1 := 4 * i
(2)   t2 := a [ t1 ]
(3)   t3 := 4 * i
(4)   t4 := b [ t3 ]
(5)   t5 := t2 * t4
(6)   t6 := prod + t5
(7)   prod := t6
(8)   t7 := i + 1
(9)   i := t7
(10)  if i <= 20 goto (1)
```

Fig. 9.15 Código de três endereços para o bloco B_2.

É importante não se confundir os GDAs com os grafos de fluxo. Cada nó de um grafo de fluxo pode ser representado por um GDA, uma vez que cada nó do grafo de fluxo está em lugar de um bloco básico.

Exemplo 9.7. A Fig. 9.15 mostra o código de três endereços correspondente ao bloco B_2 da Fig. 9.9. Os números de enunciado começando de (1) foram usados por uma questão de conveniência. O GDA correspondente é mostrado na Fig. 9.16. Discutiremos o significado do GDA após fornecer um algoritmo para construí-lo. Para o momento, vamos observar que cada nó do GDA representa uma fórmula em termos das folhas, isto é, os valores possuídos pelas variáveis e constantes ao se entrar no bloco. Por exemplo, o nó rotulado t_4 na Fig. 9.16 representa a fórmula

```
b [4 * i]
```

isto é, o valor da palavra cujo endereço está a um deslocamento de 4*i *bytes* a partir do endereço b, que é o valor pretendido para t_4. □

Construção de GDAs

Para se construir um GDA para um bloco básico, processamos um enunciado do bloco de cada vez. Quando enxergamos um enunciado da forma x := y + z, procuramos pelos nós que representam os valores "correntes" de y e z. Esses poderiam ser folhas, ou poderiam ser nós inte-

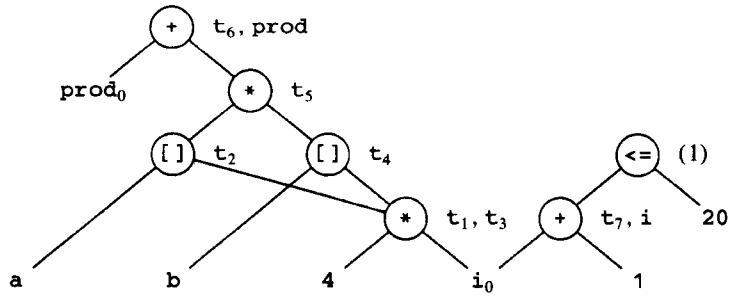


Fig. 9.16 GDA para o bloco da Fig. 9.15.

riores do GDA, se y e/ou z tivessem sido avaliados pelos enunciados anteriores do bloco. Criamos em seguida um nó rotulado + e damos ao mesmo dois filhos; o filho à esquerda é o nó para y e o à direita é o nó para z. Rotulamos, então, esse nó de x. No entanto, se já existir um nó denotando o mesmo valor que y + z, não adicionamos o novo nó ao GDA, mas, ao invés, damos ao nó existente o rótulo adicional x.

Dois detalhes deveriam ser mencionados. Primeiro, se x (não x_0) tiver previamente rotulado algum outro nó, removemos aquele rótulo, uma vez que o valor "corrente" de x é o nó recém-criado. Segundo, para uma atribuição tal como x := y, não criamos um novo nó. Ao invés, atrelamos o rótulo x à lista de nomes no nó para o valor "corrente" de y.

Fornecemos agora o algoritmo para computar o GDA a partir de um bloco. O algoritmo é quase o mesmo que o Algoritmo 5.1, exceto pela lista adicional de identificadores que atrelamos a cada nó aqui. O leitor deve estar consciente de que esse algoritmo pode não operar corretamente se existirem atribuições para *arrays*, se existirem atribuições indiretas através de apontadores ou se uma localização de memória puder ser referenciada através de dois ou mais nomes, devido a enunciados EQUIVALENCE ou a correspondência entre parâmetros formais e atuais de uma chamada de procedimento. Discutimos as modificações necessárias para tratar essas situações ao final desta seção.

Algoritmo 9.2. Construção de um GDA.

Entrada. Um bloco básico.

Saída. Um GDA para o bloco básico contendo as seguintes informações:

1. Um *rótulo* para cada nó. Para as folhas, o rótulo é um identificador (as constantes são permitidas) e para os nós interiores, um símbolo de operador.

2. Para cada nó, uma lista (possivelmente vazia) de identificadores atrelados (as constantes não são permitidas aqui).

Método. Assumimos que as estruturas de dados apropriadas estejam disponíveis para criar os nós com um ou dois filhos, com a distinção entre filho à "esquerda" e filho à "direita" no último caso. Também disponível na estrutura está um local para o rótulo para cada nó e a facilidade de se criar uma lista ligada de identificadores atrelados a cada nó.

Adicionalmente a esses componentes, precisamos manter o conjunto de todos os identificadores (incluindo as constantes) para os quais haja um nó associado. O nó poderia ser ou uma folha rotulada por aquele identificador ou um nó interior com aquele identificador na sua lista de identificadores atrelados. Assumimos a existência da função *nó(identificador)*, que, à medida que construímos o GDA, retorna o nó mais recentemente criado associado a *identificador*. Intuitivamente, *nó(identificador)* é o nó do GDA que representa o valor que aquele identificador tem no ponto corrente do processo de construção do GDA. Na prática, uma entrada no registro da tabela de símbolos para *identificador* indicaria o valor de *nó(identificador)*.

O processo de construção de GDA é executar os seguintes passos, de (1) a (3), para cada enunciado do bloco, sucessivamente. De início, assumimos que não existam nós e que *nó* seja indefinido para todos os argumentos. Suponhamos que o enunciado "corrente" de três endereços seja (i) x := y *op* z ou (ii) x := *op* y ou (iii) x := y.[7] Referimo-nos a esses como casos (i), (ii) e (iii). Tratamos um operador relacional, como if i<= 20 goto como caso (i), com x indefinido.

1. Se *nó* (y) for indefinido, criar uma folha rotulada y e fazer *nó* (y) igual a esse nó. No caso (i), se *nó* (z) for indefinido, criar uma folha rotulada z e fazer aquela folha *nó* (z).

[7]Os operadores são assumidos terem no máximo dois argumentos. A generalização para três ou mais argumentos é direta.

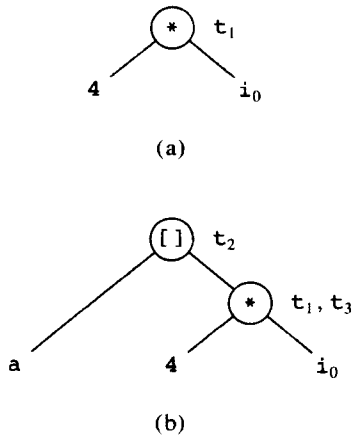


Fig. 9.17 Passos no processo de construção do GDA.

2. No caso (i), determinar se existe um nó rotulado *op*, cujo filho à esquerda seja *nó* (y) e cujo filho à direita seja *nó* (z). (Esta verificação é para capturar as subexpressões comuns.) Caso não haja, criar um tal nó. Em quaisquer dos casos, seja *n* o nó encontrado ou criado. No caso (ii), determinar se existe um nó rotulado *op* cujo único filho seja *nó* (y). Se não existir, criar um tal nó e seja *n* o nó encontrado ou criado. No caso (iii), seja *n* o *nó* (y).

3. Remover x da lista de identificadores atrelados para *nó* (x). Incluir x à lista atrelada de identificadores para o *nó n* encontrado em (2) e fazer *nó* (x) igual a *n*.

Exemplo 9.8. Vamos retornar ao bloco da Fig. 9.15 e ver como o GDA da Fig. 9.16 é construído a partir do mesmo. O primeiro enunciado é t_1 := 4 * i. No passo (1), precisamos criar as folhas rotuladas 4 e i_0 (usamos o subscrito 0, como antes, para auxiliar a distinção dos rótulos dos identificadores atrelados nas figuras, mas o subscrito não é realmente parte do rótulo). No passo (2), criamos um nó rotulado * e, no passo (3), atrelamos o identificador t_1 ao mesmo. A Fig 9.17(a) mostra o GDA nesse estágio.

Para o segundo enunciado, t_2 := a[t_1], criamos numa nova folha rotulada por a e encontramos o nó previamente criado, *nó* (t_1). Criamos, também, um nó rotulado [ ] ao qual atrelamos os nós para a e para t_1 como filhos.

Para o enunciado (3), t_3 := 4 * i, determinamos que *nó* (4) e *nó* (i) já existem. Como o operador é *, não criamos um novo nó para o enunciado(3), mas, ao invés, adicionamos t_3 ao fim da lista de identificadores para o *nó* t_1. O GDA resultante é mostrado na Fig. 9.17(b). O método dos números de valor da Seção 5.2 pode ser usado para descobrir rapidamente que o nó 4 * i já existe.

Convidamos o leitor a completar a construção do GDA. Mencionamos somente os passos dados para o enunciado (9), i := t_2. Antes do enunciado (9), *nó* (i) é a folha rotulada i_0. O enunciado (9) é uma instância do caso (iii); por conseguinte, obtemos *nó* (t_2), atrelamos i ao final da sua lista de identificadores, e atribuímos *nó* (j) a *nó* (t_2). Esse é um dos enunciados — o outro enunciado é (7) — onde o valor de *nó* muda para um identificador. É a mudança que assegura que um novo nó para i é o filho à esquerda do nó do operador <= construído para o enunciado (10). □

Aplicações de GDAs

Existem diversos itens úteis de informação que podemos obter à medida que estivermos executando o Algoritmo 9.2. Primeiro, observemos que detectamos automaticamente as subexpressões comuns. Segundo, podemos determinar que identificadores possuem seus valores usados

no bloco; são exatamente aqueles para os quais uma folha é criada no passo (1) em algum instante. Terceiro, podemos determinar que enunciados computam valores que poderiam ser usados fora do bloco. São exatamente aqueles enunciados *S* cujos nós *n* construídos ou encontrados no passo (2) ainda possuem *nó*(x) = *n* ao final da construção do GDA, onde x é o identificador atribuído pelo enunciado *S*. (Equivalentemente, x ainda é um identificador atrelado a *n*).

Exemplo 9.9. No exemplo 9.8, todos os enunciados atendem à restrição acima porque as únicas vezes em que *nó* era redefinida — para prod e i — o valor anterior de *nó* era o de uma folha. Por conseguinte, todos os nós interiores podem ter seus valores usados fora do bloco. Vamos supor agora que tivéssemos inserido antes do enunciado (9) um novo enunciado *s* que atribuísse um valor a i. No enunciado *s* criaríamos um nó *m* e faríamos *nó*(i) = *m*. No entanto, no enunciado (9) redefiniríamos *nó*(i). Conseqüentemente, o valor computado no enunciado *s* não poderia ser usado fora do bloco. □

Um outro uso importante no qual o GDA pode ser colocado é o de reconstruir uma lista simplificada de quádruplas, tirando-se partido das subexpressões comuns e não realizando-se as atribuições da forma x := y, a menos que sejam absolutamente necessárias. Isto é, sempre que um nó possuir mais de um identificador em suas listas, verificamos quais daqueles identificadores, se algum, são necessitados fora do bloco. Como mencionamos, encontrar as variáveis vivas ao final de um bloco requer uma análise de fluxo de dados chamada de "análise das variáveis vivas", discutida no Capítulo 10. Entretanto, em muitos casos podemos assumir que nenhum nome temporário, tal como t_1, t_2, . . . , t_7 na Fig. 9.15, seja necessitado fora do bloco (mas fique-se avisado sobre como as expressões lógicas são traduzidas; uma expressão pode se espalhar por sobre diversos blocos básicos).

Podemos, em geral, avaliar os nós interiores de um GDA em qualquer ordem que seja uma classificação topológica do mesmo. Numa classificação topológica, um nó não é avaliado até que todos os seus filhos que sejam nós interiores tenham sido avaliados. À medida que avaliamos um nó, atribuímos seu valor a um dos identificadores atrelados x, preferindo aquele cujo valor seja necessitado fora do bloco. Não podemos, entretanto, escolher x se existir um outro nó *m*, cujo valor foi também guardado por x, tal que *m* tenha sido avaliado e esteja ainda "vivo". Aqui, definimos *m* como vivo se seu valor for necessitado fora do bloco ou se *m* possuir um pai que não tenha sido ainda avaliado.

Se existirem identificadores adicionais atrelados $y_1, y_2, \ldots, y_k$, cujos valores sejam também necessitados fora do bloco, atribuímos aos mesmos através dos enunciados y_1 := x, y_2 := x, . . . , y_k := x. Se *n* não possuir de todo identificadores atrelados (isso poderia ocorrer se, digamos, *n* tivesse sido criado por uma atribuição a x, mas x tivesse recebido subseqüentemente uma reatribuição), criaríamos um novo nome temporário para guardar o valor de *n*. O leitor deve estar consciente de que na presença de uma atribuição de *array* ou de apontador, nem toda classificação topológica de um GDA é permitida; iremos endereçar tal matéria em breve.

Exemplo 9.10 Vamos reconstruir um bloco básico da Fig. 9.16, ordenando os nós na mesma ordem em que foram criados: t_1, t_2, t_4, t_5, t_6, t_7 (1). Note-se que os enunciados (3) e (7) do bloco original não criaram novos nós, mas adicionaram os rótulos t_3 e prod às listas dos nós t_1 e t_6, respectivamente. Assumimos que nenhum dos temporários t_i seja necessário fora do bloco.

Começamos com o nó que representa 4 * i. Esse nó possui dois identificadores atrelados, t_1 e t_3. Vamos pinçar t_1 para guardar o valor 4 * i, de forma que o primeiro enunciado reconstruído é

$$t_1 := 4 * i$$

exatamente como no bloco original. O segundo nó considerado é rotulado t_2. O enunciado construído para esse nó é

$$t_2 := a [t_1]$$

como antes. O nó considerado em seguida é rotulado t_4, a partir do qual geramos o enunciado

$$t_4 := b [t_1]$$

O último enunciado usa t_1 como um argumento, ao invés de t_3, como no bloco básico original, porque t_1 é o nome escolhido para carregar o valor de 4 * i.

Em seguida, consideramos o nó rotulado t_5 e geramos o enunciado

$$t_5 := t_2 * t_4$$

Para o nó rotulado t_6, prod, selecionamos prod para carregar o valor, uma vez que esse identificador, e não t_6, irá (presumivelmente) ser necessitado fora do bloco. Como t_3, o temporário t_6 desaparece. O próximo enunciado gerado é

$$prod := prod + t_5$$

Similarmente, escolhemos i ao invés de t_7 para carregar o valor de i + 1. Os dois últimos enunciados gerados são

```
i := i + 1
if i <= 20 goto (1)
```

Note-se que os dez enunciados da Fig. 9.15 foram reduzidos a sete tirando-se partido das subexpressões comuns expostas durante o processo de construção do GDA e eliminando-se as atribuições desnecessárias. □

Arrays, Apontadores e Chamadas de Procedimentos

Consideremos o bloco básico:

```
x := a[i]
a[j] := y                    (9.5)
z := a[i]
```

Se usarmos o Algoritmo 9.2 para construir o GDA para (9.5), a[i] se tornaria uma subexpressão comum e o bloco "otimizado" se tornaria

```
x := a[i]
z := x                       (9.6)
a[j] := y
```

Entretanto, (9.5) e (9.6) computam valores diferentes para z no caso de i = j e y ≠ a[i]. O problema é que quando atribuímos a um array a, podemos estar modificando o valor-*r* da expressão a[i], mesmo que a e i não mudem. Por conseguinte, é necessário que, ao se processar uma atribuição a um *array* a, *matemos* todos os nós rotulados [ ], cujos argumentos à esquerda sejam mais ou menos uma constante (possivelmente zero)[8]. Isto é, tornamos esses nós inelegíveis para receber um rótulo de identificador adicional, impedindo-os de serem falsamente reconhecidos como subexpressões comuns. É dessa forma requerido que mantenhamos um *bit* para cada nó, informando se o mesmo foi ou não morto. Sobretudo, para cada *array* a mencionado no bloco, é conveniente ter uma lista de todos os nós correntemente não mortos mas que terão de sê-lo caso os atribuamos a um elemento de a.

[8]Note-se que o argumento de [ ], indicando o nome do *array*, poderia ser o próprio a, ou uma expressão como a − 4. No último caso, o nó a seria um neto, ao invés de um filho do nó [ ].

Um problema similar ocorre se temos uma atribuição tal como `*p := w`, onde `p` é um apontador. Se não sabemos para o que `p` poderia apontar, cada nó correntemente no GDA sendo construído precisa ser morto no sentido acima. Se o nó *n* rotulado `a` é morto e existe uma atribuição subseqüente a `a`, precisamos criar uma nova folha para `a` e usar a mesma em lugar de *n*. Consideramos mais adiante as restrições na ordem de avaliação causadas pela morte dos nós.

No Capítulo 10, discutimos métodos através dos quais poderíamos descobrir que `p` poderia apontar, somente para alguns subconjuntos de identificadores. Se `p` pudesse somente apontar para `r` ou `s`, então somente *nó*`(r)` e *nós*`(s)` precisam ser mortos. É igualmente concebível podermos descobrir que `i = j` seja impossível no bloco (9.5), caso em que o nó para `a[i]` não precisa ser morto por `a[j] := y`. Entretanto, o último tipo de descoberta não compensa a complicação.

Uma chamada de procedimento num bloco básico mata todos os nós, uma vez que na ausência de conhecimento a respeito do procedimento chamado precisamos assumir que qualquer variável possa ser modificada por um efeito colateral. O Capítulo 10 discute como devemos estabelecer que certos identificadores não sejam modificados por uma chamada de procedimento e, por conseguinte, os nós para esses identificadores não precisarão ser mortos.

Se pretendemos remontar o GDA em um bloco básico, e não desejamos usar a ordem na qual os nós do GDA foram criados, precisamos, então, indicar no GDA que certos nós aparentemente independentes precisam ser avaliados numa determinada ordem. Por exemplo, em (9.5), o enunciado `z := a[i]` precisa seguir a `a[j] := y`, que precisa seguir a `x := a[i]`. Vamos introduzir certos lados $n \rightarrow m$ no GDA, de forma a que o mesmo não indique que *m* seja um argumento de *n*, mas, ao invés, que a avaliação de *n* deve se seguir à de *m* em qualquer cômputo do GDA. As regras que devem passar a vigorar são as seguintes:

1. Qualquer *avaliação de* ou *atribuição a* um elemento de um *array* `a` precisa seguir a atribuição prévia a um elemento daquele *array*, se existir alguma.

2. Qualquer atribuição a um elemento de um *array* `a` precisa seguir qualquer avaliação prévia de `a`.

3. Qualquer uso de qualquer identificador precisa seguir a chamada prévia de procedimento ou atribuição indireta através de um apontador, se existir alguma.

4. Qualquer chamada de procedimento ou atribuição indireta através de um apontador precisa seguir todas as avaliações prévias de qualquer identificador.

Isto é, ao se reordenar o código, os usos de um *array* `a` não podem se atravessar uns aos outros e nenhum enunciado pode atravessar uma chamada de procedimento ou atribuição através de um apontador.

9.9 OTIMIZAÇÃO PEEPHOLE

Uma estratégia de geração de código enunciado a enunciado freqüentemente produz um código-alvo que contém instruções redundantes e construções subótimas. A qualidade de tal código-alvo pode ser melhorada através da aplicação de transformações "otimizantes" ao programa-alvo. O termo "otimizante" é um tanto enganador porque não existe garantia do código-alvo resultante ser ótimo sob qualquer medida matemática. Apesar de tudo, muitas transformações simples podem melhorar significativamente o tempo de execução ou as exigências de espaço do programa-alvo e, dessa forma, é importante saber que tipos de transformações são importantes na prática.

Uma técnica simples, porém efetiva, para melhorar localmente o código-alvo é a *otimização peephole*, um método para tentar melhorar o desempenho do programa-alvo examinando-se pequenas seqüências de instruções-alvo (chamadas de *peepholes*) e substituindo-se essas instruções por uma seqüência mais curta ou mais rápida, sempre que possível. Apesar de discutirmos a otimização *peephole* como uma técnica para melhorar a qualidade do código-alvo, a mesma pode ser aplicada diretamente após a geração do código intermediário a fim de melhorar aquela representação.

Um *peephole* é uma pequena janela móvel no programa-alvo. O código na janela não precisa ser contíguo, apesar de algumas implementações o exigirem. É característica da otimização *peephole* que cada melhoria possa criar oportunidades para outras melhorias adicionais. Em geral, são necessárias passagens repetidas sobre o código-alvo para se obter o benefício máximo. Nesta seção, daremos os seguintes exemplos de transformações de programa que são características da otimização *peephole:*

- eliminação de instruções redundantes
- otimizações de fluxo de controle
- simplificações algébricas
- uso de dialetos de máquina

Cargas e Armazenamentos Redundantes

Se examinarmos a seqüência de instruções

```
(1)   MOV R0, a
(2)   MOV a, R0                                    (9.7)
```

podemos eliminar a instrução (2) porque sempre que (2) for executada, (1) irá assegurar que o valor de `a` já estará no registrador `R0`. Note-se que, se (2) tivesse um rótulo[9], — não poderíamos assegurar que (1) fosse sempre executada imediatamente antes de (2), e, nesse caso, não poderíamos removê-la. Colocado de outra forma, (1) e (2) têm que estar no mesmo bloco básico para que essa transformação seja segura.

Enquanto que um código-alvo tal como (9.7) não seria gerado se o algoritmo sugerido na Seção 9.6 fosse usado, poderia sê-lo caso fosse usado um algoritmo mais ingênuo, como aquele mencionado ao início da Seção 9.1.

Código Inatingível

Outra oportunidade para a otimização *peephole* é a remoção de instruções inatingíveis. Uma instrução não rotulada seguindo-se imediatamente a um desvio incondicional pode ser sempre eliminada. Essa operação pode ser repetida para eliminar uma seqüência de instruções. Por exemplo, para fins de depuração, um grande programa pode ter dentro de si certos segmentos que sejam executados somente se uma variável estiver ligada. Em C, o código-fonte poderia se parecer com

```
#define  debug  0

. . .

if ( debug ) {
   imprimir informações de depuração
}
```

[9]Uma vantagem de se gerar código de montagem é que os rótulos estarão presentes, facilitando uma otimização *peephole* como essa. Se for gerado código de máquina e se pretendemos realizar esse tipo de otimização, pode-se usar um *bit* para marcar as instruções que tiverem rótulos.

Na representação de código intermediário, o enunciado condicional `if` poderia ser traduzido como

```
        if debug = 1 goto L1
        goto L2
L1:     imprimir informações de depuração        (9.8)
L2:
```

Uma otimização óbvia do tipo *peephole* consiste na eliminação dos desvios sobre desvios. Por conseguinte, não importa qual seja o valor de `debug`, (9.8) pode ser substituída por

```
        if debug ≠ 1 goto L2
        imprimir informações de depuração        (9.9)
L2:
```

Agora, como `debug` é estabelecida com 0 ao início do programa[10], a propagação de constantes deveria substituir (9.9) por.

```
        if 0 ≠ 1 goto L2
        imprimir informações de depuração        (9.10)
L2:
```

Como o argumento do primeiro enunciado de (9.10) avalia para uma constante **true**, que pode ser substituído por `goto L2`. Então, todos esse enunciados que imprimem auxílios para a depuração são manifestamente inatingíveis e podem ser eliminados um de cada vez.

Otimizações do Fluxo de Controle

Os algoritmos de geração de código intermediário do Capítulo 8 freqüentemente produzem desvios para desvios, desvios para desvios condicionais ou desvios condicionais para desvios. Esses desvios desnecessários podem ser eliminados tanto no código intermediário quanto no código-alvo pelos tipos de otimização *peephole* que se seguem. Podemos substituir a seqüência de desvios

```
        goto L1
        . . .
L1:     goto L2
```

pela seqüência

```
        goto L2
        . . .
L1:     goto L2
```

Se não existem agora desvios para `L1`[11], pode então ser possível eliminar o enunciado `L1 : goto L2`, se o mesmo for precedido por um desvio incondicional. Similarmente, a seqüência

```
        if a < b goto L1
        . . .
L1:     goto L2
```

pode ser substituída por

```
        if a < b goto L2
        . . .
L1:     goto L2
```

Finalmente, suponhamos que exista somente um desvio para `L1` e que `L1` seja precedido por um desvio incondicional. A seqüência

```
        goto L1
        . . .
L1:     if a < b goto L2        (9.11)
L3:
```

pode ser substituída por

```
        if a < b goto L2        (9.12)
        goto L3
        . . .
L3:
```

Ainda que o número de instruções em (9.11) e (9.12) seja o mesmo, algumas vezes saltamos o desvio incondicional em (9.12), mas nunca em (9.11). Por conseguinte, o código de (9.12) é superior ao de (9.11) em termos do tempo de execução.

Simplificação Algébrica

Não há limite para a quantidade de simplificações algébricas que podem ser tentadas através da otimização *peephole*. No entanto, somente umas poucas identidades algébricas ocorrem com uma freqüência suficiente o bastante para justificar que sejam implementadas. Por exemplo, enunciados como

```
x := x + 0
```

ou

```
x := x * 1
```

são freqüentemente produzidos por algoritmos de geração direta de código intermediário e podem ser eliminados através da otimização *peephole*.

Redução de Capacidade

A redução de capacidade substitui as operações custosas por outras equivalentes, mais baratas, na máquina-alvo. Certas instruções de máquina são consideravelmente mais baratas que outras e podem ser usadas como casos especiais de operadores mais custosos. Por exemplo, x^2 é invariavelmente mais barata de implementar como `x*x` do que como uma chamada para uma rotina de exponenciação. A multiplicação em ponto fixo ou divisão por uma potência de dois é mais barata de implementar como um deslocamento. A divisão por uma constante em ponto flutuante pode ser implementada (com aproximação) como uma multiplicação por uma constante, operação que pode ser menos custosa.

Uso de Dialetos de Máquina

A máquina-alvo pode ter instruções de *hardware* para implementar certas operações específicas eficientemente. Detectar as situações que permitem o uso dessas instruções pode reduzir o tempo de execução significativamente. Por exemplo, algumas máquinas possuem modos de endereçamento auto-incrementantes e decrementantes. Esses modos adicionam ou subtraem um de um operando antes ou depois de usar seu valor. O uso desses modos melhora grandemente a qualidade de código ao se empilhar ou desempilhar dados, como na transmissão de parâmetros. Esses modos podem também ser usados no código de instruções como `i := i + 1`.

[10] Para dizer que `debug` possui o valor 0, precisamos realizar uma análise global de fluxo de dados do tipo "definições incidentes", como discutido no Capítulo 10.

[11] Para que essa otimização *peephole* seja realizada, podemos contar o número de desvios para cada rótulo na entrada para aquele rótulo na tabela de símbolos; uma pesquisa no código não é necessária.

9.10 A GERAÇÃO DE CÓDIGO A PARTIR DE GDAs

Nesta seção, mostramos como gerar código para um bloco básico a partir de sua representação sob a forma de GDA. A vantagem de se fazer isso é que a partir de um GDA podemos mais facilmente ver como rearranjar a ordem da seqüência final de computação do que quando começamos a partir da seqüência linear de enunciados de três endereços ou de quádruplas. Central na nossa discussão é o caso de quando o GDA é uma árvore. Para esse caso, podemos gerar um código que podemos provar ser ótimo sob critérios tais como tamanho de programa ou o menor número de temporários usados. Esse algoritmo para geração de código intermediário a partir de uma árvore é também útil quando o código intermediário é uma árvore sintática.

Rearranjando a Ordem

Vamos rapidamente considerar como a ordem na qual as computações são feitas pode afetar o custo do código objeto resultante. Consideremos o seguinte bloco básico, cuja representação sob a forma de GDA é mostrada na Fig. 9.18 (acontece que o GDA é uma árvore).

```
t1 := a + b
t2 := c + d
t3 := e - t2
t4 := t1 - t3
```

Note-se que a ordem é aquela que seria naturalmente obtida a partir de uma tradução dirigida pela sintaxe para a expressão `(a+b)-(e-(c+d))`, através do Algoritmo 8.3.

Se gerarmos código para os enunciados de três endereços, usando o Algoritmo da Seção 9.6, obtemos a seqüência de código da Fig. 9.19 (assumindo que dois registradores `R0` e `R1` estejam disponíveis e que somente t_4 esteja vivo à saída).

Por outro lado, vamos supor que rearranjemos a ordem dos enunciados de forma que o cômputo de t_1 ocorra imediatamente antes daquele de t_4, como:

```
t2 := c + d
t3 := e - t2
t1 := a + b
t4 := t1 - t3
```

Então, usando o algoritmo de geração de código da Seção 9.6, obtemos a seqüência de código da Fig. 9.20. (De novo, somente `R0` e `R1` estão disponíveis.) Através da realização da computação nesta ordem, somos capazes de economizar duas instruções, `MOV R0, t1` (que armazena o valor de `R0` na localização de memória t_1), e `MOV t2, R1` (que recarrega o valor de t_2 no registrador `R1`).

Um Ordenamento Heurístico para os GDAs

A razão pela qual o reordenamento acima melhorou o código foi que o cômputo de t_4 passou a seguir imediatamente após o cômputo de t_1,

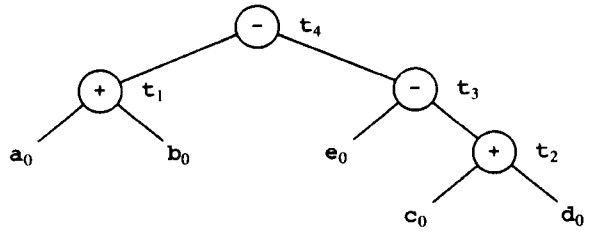


Fig. 9.18 GDA para um bloco básico.

```
MOV a, R0
ADD b, R0
MOV c, R1
ADD d, R1
MOV R0, t1
MOV e, R0
SUB R1, R0
MOV t1, R1
SUB R0, R1
MOV R1, t4
```

Fig. 9.19 Seqüência de código.

seu operando à esquerda na árvore. Que essa disposição é benéfica deveria estar claro. O argumento à esquerda para o cômputo de t_4 precisa estar num registrador para o cômputo eficiente de t_4 e computar t_2 imediatamente antes de t_4 assegura que este será o caso.

Ao selecionar um ordenamento para os nós de um GDA, estamos somente restringidos em assegurar que a ordem preserve os relacionamentos dos lados do GDA. Relembremos da Seção 9.8 que aqueles lados podem representar um relacionamento operador-operando, de restrições implicadas, decorrentes de possíveis interações entre chamadas de procedimentos, atribuições de *arrays* ou de apontadores. Propomos o seguinte algoritmo de ordenamento heurístico, que tenta, na medida do possível, fazer a avaliação de um nó se seguir imediatamente à avaliação do argumento mais à esquerda. O algoritmo da Fig. 9.21 produz o ordenamento da ordem invertida.

Exemplo 9.11. O algoritmo da Fig. 9.21 aplicado à árvore da Fig. 9.18 produz a ordem a partir da qual o código da Fig. 9.20 foi produzido. Para um exemplo mais completo, consideremos o GDA da Fig. 9.22.

Inicialmente, o único nó com todos os pais listados é 1 (uma vez que não possui pai) e, dessa forma, fazemos $n = 1$ à linha (2) e listamos 1 à linha (3). Agora o argumento à esquerda de 1, que é 2, possui todos os seus pais listados (isto é, 1, seu único pai) e, por conseguinte, listamos 2 e fazemos $n = 2$ à linha (6). Agora, à linha (4), encontramos o filho mais à esquerda de (2), que é (6), que possui um pai não listado. Por conseguinte, selecionamos um novo n à linha (2) e o nó (3) é o único candidato. Listamos, então, 3 e, em seguida, prosseguimos abaixo, ao longo de sua cadeia à esquerda, listando 4, 5 e 6. Isso deixa somente 8 entre os nós interiores e, assim, o listamos. A lista resultante é 1234568 e, por conseguinte, a ordem sugerida de avaliação é 8654321. Esse ordenamento corresponde à seqüência de enunciados de três endereços:

```
t8 := d + e
t6 := a + b
t5 := t6 - c
t4 := t5 * t8
t3 := t4 - e
t2 := t6 + t4
t1 := t2 * t3
```

```
MOV c, R0
ADD d, R0
MOV e, R1
SUB R0, R1
MOV a, R0
ADD b, R0
SUB R1, R0
MOV R0, t4
```

Fig. 9.20 Seqüência de código revisada.

```
(1)  enquanto existirem nós interiores não listados faça início
(2)      selecionar um nó n não listado do qual todos os pais
             já foram listados;
(3)      listar n;
(4)      enquanto o filho mais à esquerda m de n tiver todos os
             pais listados e não for uma folha faça
             /* como n acabou de ser listado, m não será listado
             ainda */
             início
(5)              listar m
(6)              n := m
             fim
     fim
```

Fig. 9.21 Algoritmo de listagem de nós.

```
(1)  se n é uma folha então
(2)      se n é o filho mais à esquerda de seu pai então
(3)          rótulo (n) := 1
(4)      senão rótulo (n) := 0
     senão início /* n é um nó interior */
(5)      sejam n1, n2, . . . nk os filhos de n ordenados por
             rótulo, de tal forma que
             rótulo (n1) ≥ rótulo (n2) ≥ . . . ≥ rótulo(nk);
(6)          rótulo(n) := max (rótulo (ni) + i − 1)
                          1 ≤ i ≤ k
     fim
```

Fig. 9.23 Cômputo dos rótulos.

a qual gera um código ótimo para o GDA de nossa máquina, seja qual for o número de registradores, se o algoritmo de geração de código da Seção 9.6 for usado. Deveria ser notado que, neste exemplo, nosso ordenamento heurístico nunca teve qualquer escolha para fazer ao passo (2), mas, em geral, pode ter muitas escolhas. □

Ordenamento Ótimo para Árvores

Segue-se que, para o modelo de máquina da Seção 9.2, podemos fornecer um algoritmo simples para determinar a ordem ótima na qual avaliamos os enunciados de um bloco básico quando a representação do mesmo for uma árvore. Ordem ótima aqui significa aquela que produz a menor seqüência de instruções, sobre todas as seqüências de instruções que avaliam a árvore. O algoritmo, modificado para levar em conta pares de registradores e outras peculiaridades das máquinas-alvo, tem sido usado em compiladores para Algol, Bliss e C.

O algoritmo possui duas partes. A primeira parte rotula cada nó da árvore, de baixo para cima, com um inteiro que denota o menor número de registradores requeridos para avaliar a árvore sem armazenamento de resultados intermediários. A segunda parte do algoritmo é uma travessia da árvore, cuja ordem é governada pelos rótulos computados dos nós. O código de saída é gerado durante a travessia da árvore.

Intuitivamente, o algoritmo opera, dados os dois operandos de um operador binário, avaliando primeiro os operandos que requeiram mais registradores (o operando mais difícil). Se as exigências de ambos os operandos forem as mesmas, qualquer um dos dois operandos pode ser avaliado primeiro.

Algoritmo de Rotulação

Usamos o termo "folha à esquerda" para significar um nó que é uma folha e o descendente mais à esquerda de seu pai. Todas as outras folhas são referenciadas como "folhas à direita".

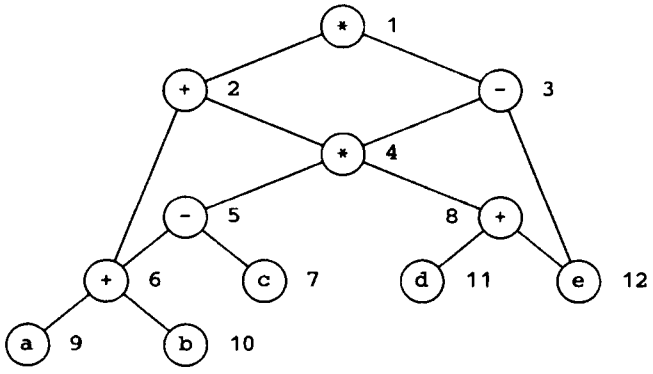


Fig. 9.22 Um GDA.

A rotulação pode ser feita visitando-se os nós até que todos os seus filhos tenham sido rotulados. A ordem na qual os nós da árvore gramatical são criados será adequada se a árvore gramatical for usada como código intermediário, e, nesse caso, por conseguinte, os rótulos podem ser computados como uma tradução dirigida pela sintaxe. A Fig. 9.23 dá o algoritmo para computar o rótulo ao nó n. No importante caso especial em que n seja um nó binário e seus filhos tenham rótulos l_1 e l_2, a fórmula à linha (6) se reduz a

$$rótulo\ (n) = \begin{cases} \max(l_1, l_2) \text{ se} & l_1 \neq l_2 \\ l_1 + 1 \text{ se} & l_1 = l_2 \end{cases}$$

Exemplo 9.12 Consideremos a árvore da Fig. 9.18. Uma travessia pós-ordem[12] da árvore visita os nós na ordem `a b` t_1 e `c d` t_2 t_3 t_4. A pós-ordem é sempre uma importante ordem na qual se pode realizar cômputos de rótulos. O nó `a` é rotulado 1 uma vez que é uma folha à esquerda. O nó `b` é rotulado 0 pois é uma folha à direita. O nó t_1 é rotulado 1 porque os rótulos de seus filhos são desiguais e o rótulo máximo de um filho é 1. A Fig. 9.24 mostra a árvore rotulada resultante. A mesma estabelece que dois registradores são necessários para avaliar t_4 e, de fato, dois registradores são necessários para avaliar t_3.

Geração de Código a Partir de uma Árvore Rotulada

Apresentamos agora o algoritmo que recebe como entrada uma árvore rotulada T e produz como saída uma seqüência de código de máquina que avalia T em `R0`. (`R0` pode ser armazenado na localização apropriada de memória.) Assumimos que T possua somente operadores binários. A generalização para operadores com um número arbitrário de operandos não é difícil e é deixada como um exercício.

O algoritmo usa o procedimento recursivo *gerar_código(n)* para produzir o código de máquina apropriado que avalia a subárvore de T com raiz n num registrador. O procedimento *gerar_código* usa a pilha *pilha_r* para alocar registradores. Inicialmente, *pilha_r* contém todos os registradores disponíveis, os quais assumimos que sejam `R0, R1,` . . . , `R`(r − 1), nesta ordem. Uma chamada para *gerar_código* deve encontrar um subconjunto de registradores, talvez numa ordem diferente, em *pilha_r*. Quando *gerar_código* retorna, deixa os registradores em *pilha_r* na mesma ordem em que os encontrou. O código resultante computa o valor da árvore T no registrador ao topo de *pilha_r*.

A função *trocar (pilha_r)* intercambia os dois registradores ao topo de *pilha_r*. O uso de *trocar* assegura que o filho à esquerda e seu pai sejam avaliados no mesmo registrador.

O procedimento *gerar_código* usa a pilha *pilha_t* para alocar as localizações temporárias de memória. Assumimos que *pilha_t* conte-

[12] Uma travessia pós-ordem visita recursivamente as subárvores enraizadas aos filhos n_1, n_2, ..., n_k de um nó n e, em seguida n. É a ordem na qual os nós de uma árvore gramatical são criados numa análise gramatical *bottom-up*.

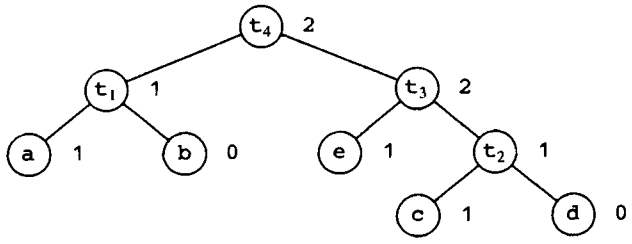
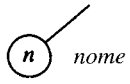


Fig. 9.24 Árvore rotulada.

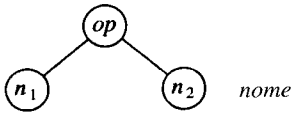
nha inicialmente `T0, T1, T2 ...`. Na prática, *pilha_t* não precisa ser implementada como uma lista, se simplesmente controlarmos o valor específico de i, tal que T_i esteja correntemente no topo. O conteúdo da *pilha_t* é sempre um sufixo de `T0, T1, ...`.

O enunciado X := *desempilhar (pilha)* significa "remover o topo da pilha *pilha* e atribuir o valor removido a X". Por outro lado, usamos *empilhar(pilha, X)* para significar "empilhar X na pilha *pilha*"; *topo(pilha)* se refere ao valor no topo de *pilha*.

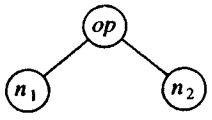
O algoritmo de geração de código chama *gerar_código* à raiz T, onde *gerar_código* é o procedimento mostrado na Fig. 9.25. Pode ser explicado através do exame de cada um dentre cinco casos. Para o caso 0, temos uma subárvore da forma



Isto é, n é folha e é o filho mais à esquerda de seu pai. Por conseguinte, geramos exatamente uma instrução de carga. Para o caso 1, temos uma subárvore da forma



para a qual geramos código para avaliar n_1 no registrador R=*topo(pilha_r)*, seguido pela instrução *op nome, R*. No caso 2, temos uma subárvore da forma



onde n_1 pode ser avaliado sem os armazenamentos, mas n_2 é mais difícil de avaliar (isto é, requer mais registradores do que n_1). Para esse caso, intercambiamos os dois registradores de topo em *pilha_r* e, em seguida, avaliamos n_2 em R := *topo(pilha_r)*. Removemos R da *pilha_r* e avaliamos n_1 em S = *topo(pilha_r)*. Note-se que S era o registrador inicialmente ao topo da *pilha_r* ao início do caso 2. Geramos, em seguida, a instrução *op R, S*, que produz o valor de n (o nó rotulado por *op*) no registrador S. Uma outra chamada para *trocar* deixa *pilha_r* como estava quando a chamada para *gerar_código* começou.

O caso 3 é similar ao caso 2, exceto que aqui a árvore à esquerda é mais difícil e é avaliada primeiro. Não há necessidade de se *trocar* os registradores aqui.

O caso 4 ocorre quando ambas as subárvores requerem r ou mais registradores para avaliar sem armazenamentos. Como precisamos de uma localização de memória temporária, avaliamos primeiro a subárvore à direita na localização temporária T, em seguida, a subárvore à esquerda e, finalmente, a raiz.

Exemplo 9.13. Vamos gerar o código para a árvore rotulada da Fig. 9.24 como *pilhar_r* = `R0, R1`, inicialmente. A seqüência de chamadas para *gerar_código* e os passos de impressão são mostrados na Fig. 9.26. Mostrado junto aos colchetes está o conteúdo de *pilha_r* a cada chamada, com o topo exibido na extremidade direita. A seqüência de código, aqui, é uma permutação daquela da Fig. 9.20. □

Podemos provar que *gerar_código* produz um código ótimo para as expressões de nosso modelo de máquina, assumindo que nenhuma das propriedades algébricas dos operadores seja levada em conta e assumindo que não hajam subexpressões comuns. A prova, deixada como um exercício, está baseada em se mostrar que qualquer seqüência de código terá que realizar.

1. uma operação para cada nó interior,

```
procedimento gerar_código(n);
início
/* caso 0 */
se n for uma folha à esquerda representando um operando nome e n
        for o filho mais à esquerda de seu pai então
    imprimir 'MOV' || nome || ',' || topo(pilha_r)
senão se n for um nó interior com operador op, filho à esquerda
        n1 e filho à direita n2 então

/* caso 1 */
    se rótulo (n2) = 0 então início
        seja nome o operando representado por n2;
        gerar_código (n1);
        imprimir op || nome || ',' || topo(pilha_r)
    fim

/* caso 2 */
    senão se 1 ≤ rótulo (n1) < rótulo(n2) e rótulo (n1) < r
    então início
        trocar(pilha_r);
        gerar_código(n2);
        R := desempilhar(pilha_r) ;
        /* n2 foi avaliado no registro R */
        gerar_código (n1);
        imprimir op || R || ',' || topo(pilha_r);
        empilhar(pilha_r, R);
        trocar(pilha_r, R)
    fim

/* caso 3 */
    senão se 1 ≤ rótulo (n2) < rótulo (n1) e rótulo (n2) < r
        então início
        gerar_código (n1);
        R := desempilhar(pilha_r);
        /* n1 foi avaliado no registrador R */
        gerar_código (n2);
        imprimir op || topo(pilha_r) || ',' || R;
        empilhar(pilha_r, R);
    fim

/* caso 4, ambos os rótulos ≥ r, o número total de registradores */
    senão início
        gerar_código (n2);
        T := desempilhar(pilha_t);
        imprimir 'MOV' || topo(pilha_r) || ',' || T;
        gerar_código(n1);
        empilhar (pilha_t, T)
        imprimir op || T || ',' || topo(pilha_r)
    fim
fim
```

Fig. 9.25 A função *gerar_código*.

```
gerar-código (t4)                          [R1 R0]                  /* caso 2*/
    gerar_código (t3)                          [R0 R2]              /* caso 3*/
        gerar_código (e)                           [R0 R1]          /* caso 0*/
            imprimir MOV e, R1
        gerar_código (t2)                          [R0]             /* caso 1*/
            gerar_código(c)                                [R0]     /* caso 0*/
                imprimir MOV c, R0
            imprimir ADD d, R0
        imprimir SUB R0, R1
    gerar_código (t1)          [R0]                                 /* caso 1*/
        gerar_código (a)                           [R0]             /* caso 0*/
            imprimir MOV a, R0
    imprimir ADD b, R0
imprimir SUB R1, R0
```

Fig. 9.26 Rastreamento da rotina *gerar_código*.

2. uma carga para cada folha que seja um filho mais à esquerda de seu pai e
3. um armazenamento para cada nó, tanto para aqueles cujos filhos tenham rótulos iguais a r quanto maiores do que r.

Como *gerar_código* produz exatamente esses passos, é otimizante.

Operações Multi-Registrador

Podemos modificar nosso algoritmo de rotulação para tratar operações tais como a multiplicação, divisão ou chamadas funcionais, que normalmente requerem mais de um registrador para serem realizadas. Simplesmente modificamos o passo (6) da Fig. 9.23, o algoritmo de rotulação, de forma que *rótulo(n)* seja pelo menos o número de registradores requeridos pela operação. Por exemplo, se é assumido que uma chamada funcional requeira todos os r registradores, substituímos a linha (6) por *rótulo (n)* = r. Se a multiplicação requer dois registradores, no caso binário usamos

$$rótulo(n) = \begin{cases} \max(2, l_1, l_2) & \text{se } l_1 \neq l_2 \\ l_1 + 1 & \text{se } l_1 = l_2 \end{cases}$$

onde l_1 e l_2 são os rótulos dos filhos de n.

Infelizmente, esta modificação não irá garantir que um par de registradores esteja disponível para uma multiplicação ou divisão, ou para operações de precisão múltipla. Um artifício útil para algumas máquinas é supor que a multiplicação e a divisão requeiram três registradores. Se *trocar* não for nunca usado em *gerar_código*, então *pilha_r* irá conter sempre registradores de números altos e consecutivos, $i, i+1, \ldots, r-1$, para algum i. Por conseguinte, os três primeiros desses registradores são assegurados incluir um par de registradores. Tirando-se vantagem do fato de que muitas operações são comutativas, podemos freqüentemente evitar o uso do caso 2 de *gerar_código*, o caso em que *trocar* é chamado. Igualmente, mesmo que *pilha_r* não contenha três registradores consecutivos ao topo, temos uma chance muito boa de encontrar um par de registradores em algum local de *pilha_r*.

Propriedades Algébricas

Se quisermos assumir as leis algébricas para os vários operadores, teremos a oportunidade de substituir uma dada árvore T por uma com rótulos menores (para evitar armazenamentos no caso 4 de *gerar_código*) e/ou menos folhas à esquerda (para evitar as cargas no caso 0). Por exemplo, como + é normalmente considerada uma operação comutativa, podemos substituir a árvore da Fig. 9.27(b), reduzindo de um o número de folhas à esquerda e possivelmente reduzindo também alguns rótulos.

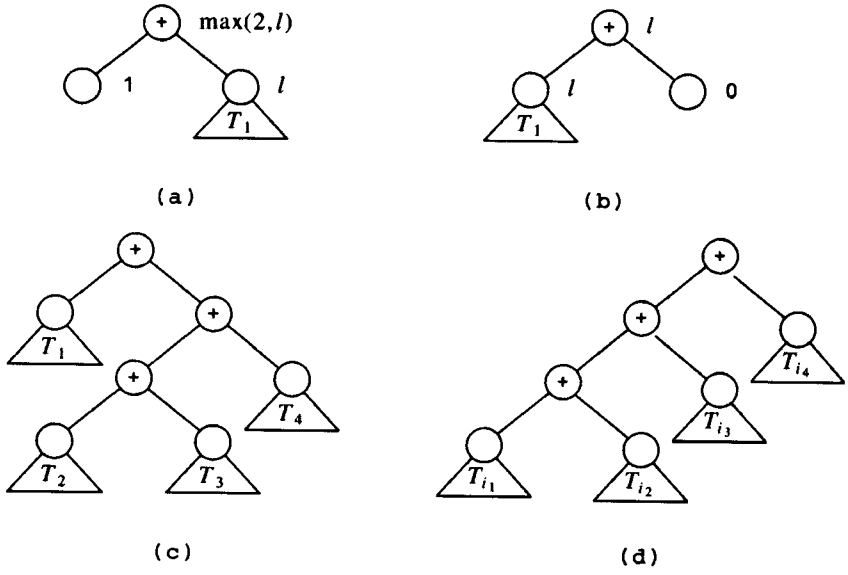


Fig. 9.27 Transformações comutativas e associativas.

Uma vez que + é usualmente tratada como sendo tanto associativa quanto comutativa, podemos tomar um agrupamento de nós rotulados +, como na Fig. 9.27(c), e substituí-lo por uma cadeia à esquerda como na Fig. 9.27(d). Para minimizar o rótulo da raiz, precisamos somente arranjar para que T_{i1} seja um dentre T_1, T_2, T_3 e T_4 com o maior rótulo e que T_{i1} não seja uma folha a menos que todos $T_1, \ldots, T_4$ o sejam.

Subexpressões Comuns

Quando existem subexpressões comuns num bloco básico, o GDA correspondente já não será mais uma árvore. As subexpressões comuns irão corresponder aos nós com mais de um pai, chamados de *nós compartilhados*. Já não podemos aplicar mais o algoritmo de rotulação ou *gerar_código* diretamente. De fato, as subexpressões comuns tornam a geração de código marcadamente mais difícil do ponto de vista matemático. Bruno e Sethi [1976] mostraram que a geração ótima de código para GDAs numa máquina com um registrador é um problema NP-completo. Aho, Johnson e Ullman [1977] mostraram que mesmo com um número ilimitado de registradores o problema permanece NP-completo. A dificuldade emerge ao se tentar determinar uma ordem ótima onde avaliar um GDA da forma mais barata.

Na prática, podemos obter uma solução razoável se particionarmos o GDA num conjunto de subárvores procurando, para cada raiz ou nó compartilhado n, a subárvore maximal que tenha n como raiz, e que não inclua outros nós compartilhados, a não ser como folhas. Por exemplo, o GDA da Fig. 9.22 pode ser particionado nas árvores da Fig. 9.28. Cada nó compartilhado com p pais aparece como uma folha em pelo menos p árvores. Os nós com mais de um pai na mesma árvore podem ser transformados em tantas árvores quantas forem necessárias, de forma que nenhuma folha possua múltiplos pais.

Uma vez que particionamos os GDAs em árvores dessa forma, podemos ordenar a avaliação de árvores e usar qualquer um dos algoritmos precedentes para gerar o código para cada árvore. A ordem das árvores precisa ser tal que os valores compartilhados que sejam folhas de uma árvore tenham que estar disponíveis quando a árvore for avaliada. As quantidades compartilhadas podem ser computadas e armazenadas na memória (ou mantidas em registradores, caso haja registradores disponíveis em número suficiente). Conquanto esse processo não gere necessariamente código otimalizado, o mesmo será freqüentemente satisfatório.

9.11 ALGORITMO DE PROGRAMAÇÃO DINÂMICA PARA A GERAÇÃO DE CÓDIGO

Na seção anterior, o procedimento *gerar_código* produziu código otimalizado a partir de uma árvore de expressão usando uma quantidade de tempo que era uma função linear do tamanho da árvore. Esse procedimento funciona para máquinas nas quais toda a computação seja feita em registradores e cujas instruções consistam em um operador aplicado a dois registradores ou a um registrador e uma localização de memória.

Um algoritmo baseado no princípio da programação dinâmica pode ser usado para estender a classe de máquinas para as quais um código otimalizado pode ser gerado, a partir de árvores de expressões, num tempo linear. O algoritmo de programação dinâmica se aplica a uma ampla classe de máquinas de registradores com conjuntos complexos de instruções.

A Classe das Máquinas de Registradores

O algoritmo de programação dinâmica pode ser usado para gerar código para qualquer máquina com r registradores intercambiáveis `R0`, `R1`, . . . `R`$r-1$ e instruções da forma `R`i := E, onde R é uma expressão contendo operadores, registradores e localizações de memória. Se E envolve um ou mais registradores, `R`i terá de ser um desses registradores. Esse modelo de máquina inclui o modelo de máquina introduzido na Seção 9.2.

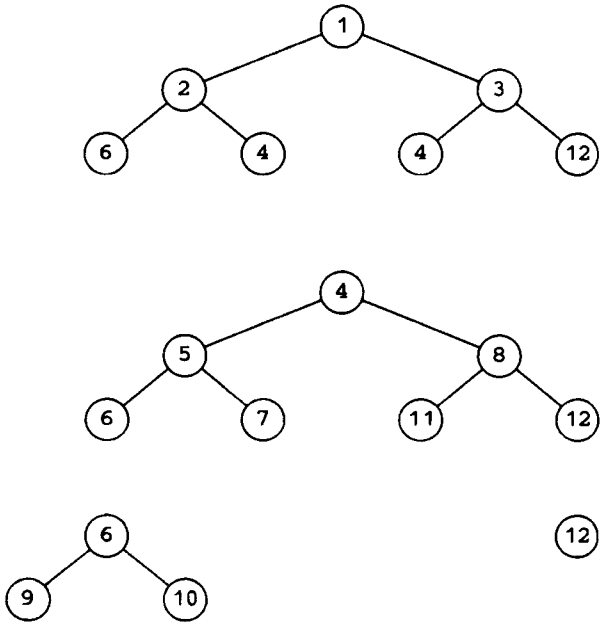


Fig. 9.28 Particionando em árvores.

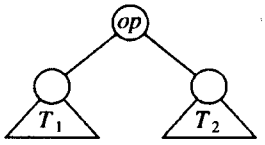
Por exemplo, a instrução `ADD R0,R1` seria representada como `R1 := R1 + R0`. A instrução `ADD *R0, R1` seria representada como `R1 := R1 + ind R0`, onde `ind` está no lugar do operador de indireção.

Assumimos que a máquina possua uma instrução de carga `R`i `:=M`, uma instrução de armazenamento `M := R`i e uma instrução de cópia registrador a registrador `R`i `:= R`j. Por uma questão de simplicidade, também assumimos que cada instrução custe uma unidade, apesar do algoritmo de programação dinâmica poder ser facilmente modificado para funcionar mesmo que cada instrução tenha o seu próprio custo.

O Princípio da Programação Dinâmica

O algoritmo de programação dinâmica particiona o problema de se gerar um código otimal para uma expressão em subproblemas para gerar código otimal para as subexpressões de dada expressão. Como um exemplo simples, consideremos uma expressão E, da forma $E_1 + E_2$. Um programa otimal para E é formado pela combinação de programas otimais para E_1 e E_2, numa ou noutra ordem, seguido pelo código para avaliar o operador +. Os subproblemas de se gerar código ótimo para E_1 e E_2 são resolvidos similarmente.

Um programa otimal produzido pelo algoritmo de programação dinâmica possui uma propriedade importante. Avalia uma expressão $E := E_1\ op\ E_2$ "contiguamente". Podemos apreciar o que isso significa examinando a árvore sintática T para E.



Aqui T_1 e T_2 são árvores para E_1 e E_2, respectivamente.

Avaliação Contígua

Dizemos que um programa P avalia uma árvore *contiguamente* se primeiro avalia, na memória, aquelas subárvores de T que necessitam ser computadas. Em seguida, avalia o restante de T ou na ordem T_1, T_2 e, em seguida, a raiz, ou na ordem T_2, e T_1 e, então, a raiz, em qualquer caso usando os valores previamente computados a partir da memória sempre que necessário. Como um exemplo de avaliação não contígua, P poderia primeiro avaliar parte de T_1, deixando o valor num registrador (ao invés de na memória); em seguida avaliar T_2 e, então, retornar para avaliar o resto de T_1.

Para a máquina de registradores definida acima podemos provar que, dado qualquer programa em linguagem de máquina P para avaliar uma árvore de expressão T, podemos encontrar um programa equivalente P', tal que

1. P' é de custo não maior do que P,
2. P' não usa mais registradores do que P e
3. P' avalia a árvore numa forma contígua.

O resultado implica que cada árvore de expressão pode ser avaliada otimalmente por um programa contíguo.

Para fazermos um contraste, máquinas com pares par-ímpar de registradores, tais como as IBM System /370, não possuem sempre avaliações contíguas otimais. Para essas máquinas, podemos dar exemplos de árvores de expressão nas quais um programa otimal em linguagem de máquina precisa primeiro avaliar num registrador uma parte da subárvore à esquerda da raiz; em seguida, uma parte da subárvore à direita; então outra parte da subárvore à esquerda; a seguir outra parte da subárvore à esquerda e assim por diante. Esse tipo de oscilação é desnecessário para uma avaliação ótima de qualquer árvore de expressão usando a máquina geral de registradores.

A propriedade da avaliação contígua definida acima diz que, para qualquer árvore de expressão T, existe sempre um programa otimal que consiste em programas otimais para as subárvores da raiz. Essa propriedade nos permite usar o algoritmo de programação dinâmica para gerar um programa otimal para T.

O Algoritmo de Programação Dinâmica

O algoritmo de programação dinâmica se desenrola em três fases. Suponhamos que a máquina-alvo possua r registradores. Na primeira fase, computamos, *bottom-up*, para cada nó n da árvore de expressão T, um *array* C de custos, no qual o $i$ésimo componente $C[i]$ é o custo ótimo de se computar a subárvore S, enraizada por n, num registrador, assumindo que i registradores estejam disponíveis para a computação, para $1 \leq i \leq r$. O custo inclui as cargas e armazenamentos necessários para avaliar S no número dado de registradores. Inclui, também, o custo de computar o operador à raiz de S. O zeroésimo componente do vetor de custos é o custo ótimo para se computar a subárvore S na memória. A propriedade da avaliação contígua assegura que um programa ótimo para S pode ser gerado, considerando-se as combinações de programas ótimos, somente para as subárvores da raiz de S. Essa restrição reduz o número de casos que precisam ser considerados.

Para computar $C[i]$ ao nó n, consideremos cada instrução de máquina `R` := E cuja expressão E corresponda à subexpressão enraizada ao nó n. Pelo exame dos vetores de custo para os descendentes correspondentes de n, determinamos os custos de se avaliar os operandos de E. Para aqueles operandos de E que sejam registradores, consideremos todas as possíveis ordens nas quais as subárvores correspondentes de T possam ser avaliadas em registradores. Em cada ordenamento, a primeira subárvore correspondendo a um operando registrador pode ser avaliada usando i registradores disponíveis, a segunda usando i-1 registradores e assim por diante. Para contabilizar o nó n, adicionamos o custo da instrução `R` := E que foi usada para corresponder ao nó n. O valor de $C[i]$ será então mínimo sobre todas as possíveis ordens.

Os vetores de custos para a totalidade da árvore T podem ser computados de baixo para cima num tempo linearmente proporcional ao número de nós de T. É conveniente armazenar a cada nó a instrução usada para atingir o melhor custo para $C[i]$, para cada valor de i. O menor custo no vetor para a raiz de T fornece o custo mínimo para se avaliar T.

Na segunda fase do algoritmo, fazemos uma travessia da árvore usando os vetores de custo para determinar que subárvores de T precisam ser computadas na memória. Na terceira fase, fazemos uma travessia de cada árvore usando os vetores de custos e instruções associadas para gerar o código-alvo final. O código para as subárvores computadas em localizações de memória é gerado primeiro. Essas duas fases podem também ser implementadas de forma a que rodem num tempo linearmente proporcional ao tamanho da árvore da expressão.

Exemplo 9.14. Consideremos uma máquina que tenha dois registradores `R0` e `R1` e as seguintes instruções, cada uma de custo unitário:

```
Ri  :=  Mj
Ri  :=  Rj op Rj
Ri  :=  Ri op Mj
Ri  :=  Rj
Mi  :=  Ri
```

Nessas instruções, `R`i é `R0` ou `R1` e `M`j é uma localização de memória.

Vamos aplicar o algoritmo de programação dinâmica para gerar um código ótimo para a árvore sintática da Fig. 9.29. Na primeira fase, computamos os vetores de custo mostrados a cada nó. Para ilustrar esse cômputo de custo, consideremos o vetor de custos à folha `a`. $C[0]$, o custo de computar `a` na memória é 0, uma vez que já está lá. $C[1]$, o custo de computar `a` num registrador é 1, uma vez que podemos carregar o seu valor num registrador através da instrução `R0 := a`. $C[2]$, o custo de se carregar `a` num registrador, tendo dois registradores disponíveis, é o mesmo do que quando se tem um único registrador. O vetor de custos à folha `a` é, por conseguinte, (0, 1, 1).

Consideremos o vetor de custos à raiz. Primeiro, determinamos o custo mínimo de se computar a raiz com um e dois registradores disponíveis. A instrução de máquina `R0 := R0 + M` se ajusta à raiz, porque a raiz é rotulada com o operador +. Usando esta instrução, o custo mínimo de se avaliar a raiz com um registrador disponível é o custo

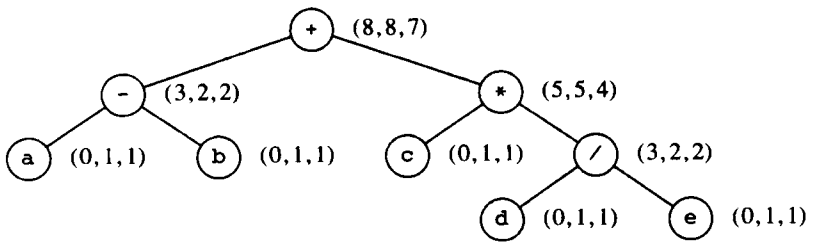


Fig. 9.29 Árvore sintática para `(a-b)+c*(d/e)` com um vetor de custos a cada nó.

mínimo de se computar sua subárvore à direita na memória mais o custo mínimo de se computar sua subárvore à esquerda num registrador, mais 1 para a instrução. Não há outra forma. Os vetores de custos dos filhos à direita e à esquerda da raiz mostram que o custo mínimo de se computar a raiz com um registrador disponível é 5 + 2 + 1 = 8.

Consideremos agora o custo mínimo de se avaliar a raiz com dois registradores disponíveis. As alternativas surgem dependendo de que instrução é usada para computar a raiz e em que ordem as subárvores da esquerda e da direita da raiz são avaliadas.

1. Computar a subárvore à esquerda no registrador `R0`, com dois registradores disponíveis, computar a subárvore à direita no registrador `R1`, com um registrador disponível, e usar a instrução `R0 := R0 + R1` para computar a raiz. O custo dessa seqüência é 2 + 5 + 1 = 8.

2. Computar a subárvore à direita com dois registradores disponíveis em `R1`, computar a subárvore à esquerda com um registrador disponível em `R0` e usar a instrução `R0 := R0 + R1`. Essa seqüência possui custo 4 + 2 + 1 = 7.

3. Computar a subárvore à direita na localização de memória `M`, computar a subárvore à esquerda, no registrador `R0`, com dois registradores disponíveis e usar a instrução `R0 := R0 + M`. Essa seqüência possui custo 5 + 2 + 1 = 8.

A segunda escolha fornece o custo mínimo 7.

O custo mínimo de se computar a raiz na memória é determinado pela adição de um ao custo mínimo de se computar a raiz com todos os registradores disponíveis; isto é, computamos a raiz num registrador e, em seguida, armazenamos o resultado. O vetor de custos à raiz é, por conseguinte, (8, 8, 7).

A partir dos vetores de custos, podemos construir facilmente a seqüência de código fazendo-se uma travessia da árvore. A partir da Fig. 9.29, assumindo que dois registradores estejam disponíveis, uma seqüência ótima é

```
R0 := c
R1 := d
R1 := R1 / e
R0 := R0 * R1
R1 := a
R1 := R1 - b
R1 := R1 + R0
```
□

Esta técnica, originalmente desenvolvida em Aho e Johnson [1976], tem sido usada em vários compiladores, incluindo a segunda versão de S. C. Johnson para um compilador C portável, o PCC2. A técnica facilita a reorientação por causa da aplicabilidade da técnica de programação dinâmica a uma ampla classe de máquinas.

9.12 GERADORES DE GERADORES DE CÓDIGO

A geração de código envolve a escolha de uma ordem de avaliação para as operações, a atribuição dos registradores para a guarda de valores e a seleção das instruções apropriadas da linguagem-alvo para implementar os operadores da representação intermediária. Mesmo que assumíssemos que a ordem de avaliação fosse fornecida e os registradores alocados por um mecanismo separado, o problema de decidir que instruções usar pode ser uma grande tarefa combinatorial, em especial numa máquina rica em modos de endereçamento. Nessa seção, apresentamos técnicas de reescrita de árvores que podem ser usadas para construir automaticamente a fase de seleção de instruções de um gerador de código, a partir de uma especificação em alto nível da máquina-alvo.

Geração de Código através da Reescrita de Árvore

Através desta seção, a entrada para o processo de geração de código será uma seqüência de árvores ao nível semântico da máquina-alvo. As árvores são o que poderíamos obter após inserir os endereços em tempo de execução na representação intermediária, como descrito na Seção 9.3.

Exemplo 9.15. A Fig 9.30 contém uma árvore para o enunciado de atribuição `a[i] := b + 1`, no qual `a` e `i` são nomes locais cujos endereços em tempo de execução são dados como os deslocamentos $const_a$ e $const_i$ a partir de `SP`, o registrador que contém o apontador para o início do registro de ativação corrente. O *array* `a` é armazenado na pilha em tempo de execução. A atribuição a `a[i]` é uma atribuição indireta, na qual o valor-*r* da localização para `a[i]` é estabelecido com o valor-*r* da expressão `b+1`. O endereço do *array* `a` é obtido pela adição do valor da constante $const_a$ ao conteúdo do registrador `SP`; o valor de `i` é a localização obtida pela adição do valor da constante $const_i$ ao conteúdo do registrador `SP`. A variável `b` é um nome global com a localização de memória mem_b. Por uma questão de simplicidade, assumimos que todas as variáveis sejam do tipo caractere.

Na árvore, o operador `ind` trata seu argumento como uma localização de memória. Como filho à esquerda de um operador de atribuição, o nó `ind` fornece a localização na qual o valor-*r* do lado direito do operador de atribuição deve ser armazenado. Se o argumento de um operador + ou `ind` for uma localização de memória ou um registrador, o conteúdo daquela localização de memória ou registrador é tomado como o valor. As folhas da árvore são atributos de tipo com subscritos; o subscrito indica o valor do atributo. □

O código-alvo é gerado durante um processo no qual a árvore de entrada é reduzida num único nó através da aplicação de uma seqüência de regras de reescrita de árvores. Cada regra de reescrita de árvore é um enunciado da forma

substituição ← *gabarito* {*ação*}

onde

1. *substituição* é um único nó,
2. *gabarito* é uma árvore e
3. *ação* é um fragmento de código, como no esquema de tradução dirigida pela sintaxe.

Um conjunto de regras de reescrita de árvores é chamado de *esquema de tradução de árvores.*

Cada gabarito de árvore representa uma computação realizada pela seqüência de instruções de máquina emitida pela ação associada. Usualmente, um gabarito corresponde a uma única instrução de máquina. As folhas do gabarito são atributos com subscritos, como na

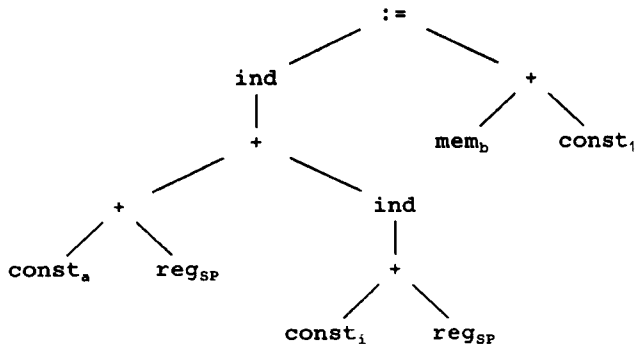
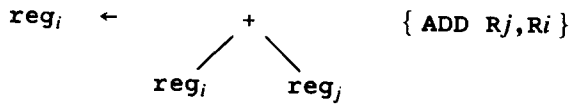


Fig. 9.30 Árvore do código intermediário para `a[i] := b+1`.

árvore de entrada. Freqüentemente, certas restrições se aplicam aos valores dos subscritos nos gabaritos; essas restrições são especificadas como predicados semânticos que precisam ser satisfeitos antes do gabarito ser considerado como correspondido. Por exemplo, um predicado poderia especificar que o valor de uma constante caísse num determinado intervalo.

Um esquema de tradução de árvores é uma forma conveniente de se representar a fase de seleção de instruções de um gerador de código. Como exemplo de uma regra de reescrita de árvores, consideremos a regra para a instrução de adição registrador a registrador:



Esta regra é usada como se segue. Se a árvore de entrada contém uma subárvore que se ajuste a esse gabarito de árvore, isto é, uma subárvore cuja raiz seja rotulada pelo operador + e cujos filhos à esquerda e à direita sejam quantidades nos registradores i e j, podemos substituir aquela subárvore por um único nó rotulado $\texttt{reg}_i$ e emitir a instrução ADD R_j, R_i como saída. Mais de um gabarito pode se ajustar a uma subárvore a um dado instante; iremos descrever em breve alguns mecanismos para decidir que regra aplicar no caso de um conflito. Assumimos que a alocação de registradores seja feita antes da seleção de código.

Exemplo 9.16 A Fig. 9.31 contém regras de reescrita de árvores para umas poucas instruções em nossa máquina-alvo. Essas regras serão usadas num exemplo que perambulará ao longo desta seção. As duas

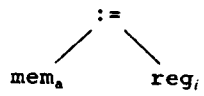
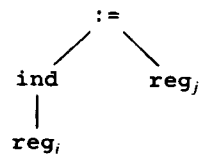
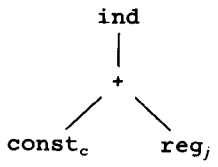
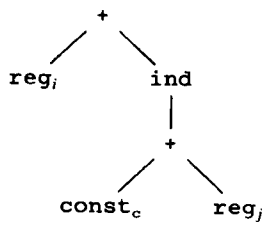
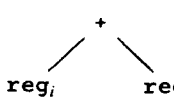
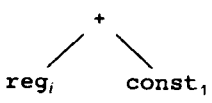
(1)	$\texttt{reg}_i \leftarrow \texttt{const}_c$	{ MOV #c, R*i* }
(2)	$\texttt{reg}_i \leftarrow \texttt{mem}_a$	{ MOV a, R*i* }
(3)	mem ← 	{ MOV R*i*, a }
(4)	mem ← 	{ MOV R*j*, *R*i* }
(5)	$\texttt{reg}_i \leftarrow$ 	{ MOV c(R*j*), R*i* }
(6)	$\texttt{reg}_i \leftarrow$ 	{ ADD c(R*j*), R*i* }
(7)	$\texttt{reg}_i \leftarrow$ 	{ ADD R*j*, R*i* }
(8)	$\texttt{reg}_i \leftarrow$ 	{ INC R*i* }

Fig. 9.31 Regras de reescrita de árvores para algumas instruções da máquina-alvo.

primeiras regras correspondem a instruções de carga, as duas seguintes a instruções de armazenamento e o restante a adições e cargas indexadas. Note-se que a regra (8) requer que o valor da constante seja um. Essa condição seria especificada por um predicado semântico. □

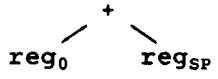
Um esquema de tradução de árvores funciona da seguinte forma. Dada uma árvore de entrada, os gabaritos das regras de reescrita de árvores são aplicados às suas (da árvore) subárvores. Se a ação contém uma seqüência de instruções de máquina, as instruções são emitidas. Este processo é repetido até que a árvore seja reduzida a um único nó ou até que não hajam mais gabaritos que possam se ajustar. A seqüência de instruções de máquina gerada à medida que a árvore de entrada era reduzida a um único nó se constitui na saída do esquema de tradução de árvore da árvore de entrada.

O processo de especificar um gerador de código vem a ser similar àquele de se usar um esquema de tradução dirigida pela sintaxe para especificar um tradutor. Escrevemos um esquema de tradução de árvores para descrever o conjunto de instruções de uma máquina-alvo. Na prática, gostaríamos de encontrar um esquema que fizesse com que fosse gerada uma seqüência de instruções de custo mínimo para cada árvore de entrada. Várias ferramentas estão disponíveis para auxiliar a construção do gerador de código automaticamente a partir de um esquema de tradução de árvores.

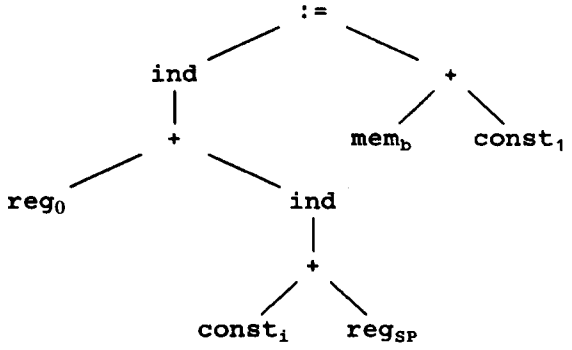
Exemplo 9.17. Vamos usar o esquema de tradução de árvores da Fig. 9.31 para gerar o código para a árvore de entrada na Fig. 9.30. Suponhamos que a primeira regra

(1) $\texttt{reg}_0 \leftarrow \texttt{const}_a$ { MOV #a, R0 }

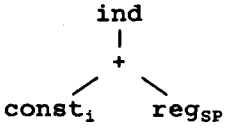
seja aplicada para se carregar a constante a no registrador R0. O rótulo para a folha mais à esquerda muda, então, de $\texttt{const}_a$ para R0 e a instrução MOV #a,R0 é gerada. A sétima regra

(7) $\texttt{reg}_0 \leftarrow$  { ADD SP,R0 }

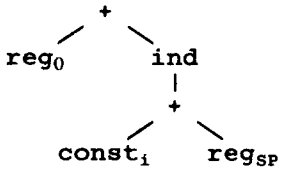
agora se ajusta à subárvore mais à esquerda com raiz rotulada +. Usando esta regra, reescrevemos esta subárvore como um único nó rotulado $\texttt{reg}_0$ e geramos a instrução ADD SP, R0. Agora a árvore está como



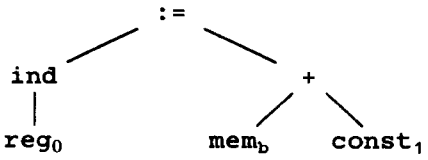
A esse ponto, poderíamos aplicar a regra (5) para reduzir a subárvore



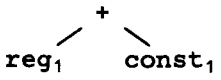
a um único nó rotulado $\texttt{reg}_1$. No entanto, poderíamos usar também a regra (6) para reduzir a subárvore maior num único nó rotulado $\texttt{reg}_0$ e gerar a instrução ADD i(SP), R0.



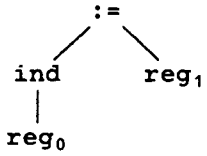
Assumindo que seja mais eficiente usar uma única instrução para computar a subárvore maior ao invés da menor, escolhemos a última redução para obter



Na subárvore à direita, a regra (2) se aplica à folha $\texttt{mem}_b$. Esta regra gera uma instrução para carregar b no registrador 1, digamos. Agora, usando a regra (8), podemos enquadrar a subárvore e gerar a instrução de incremento INC R1. A esse ponto, a árvore de entrada se reduziu a



Essa árvore remanescente é enquadrada pela regra (4), que a reduz a um único nó e gera a instrução MOV R1, #R0.



No processo de redução da árvore a um único nó, geramos a seguinte seqüência de código:

```
MOV    #a,R0
ADD    SP,R0
ADD    i(SP),R0
MOV    b,R1
INC    R1
MOV    R1,*R0
```
□

Vários aspectos desse processo de redução de árvores precisam de explicações subseqüentes. Não especificamos como o reconhecimento do padrão de árvores é feito. Tampouco especificamos uma ordem na qual os gabaritos são reconhecidos ou o que fazer se mais de um gabarito enquadrar a mesma subárvore a um dado instante. Igualmente, note-se que, se nenhum gabarito enquadrar uma subárvore, o processo de geração de código fica bloqueado. No outro extremo, pode ser possível que um único nó seja reescrito indefinidamente, gerando uma seqüência infinita de instruções de movimentação de registradores ou uma seqüência infinita de cargas e armazenamentos.

Uma forma de se realizar o enquadramento dos padrões de árvores eficientemente é estender o algoritmo de reconhecimento de padrões para múltiplas palavras-chave, do Exercício 3.32, num algoritmo *top-down* de reconhecimento de padrões de árvores. Cada gabarito pode ser representado por um conjunto de cadeias de caracteres, nominalmente, pelo conjunto de percursos da raiz até as folhas. A partir dessas coleções de cadeias, podemos construir um reconhecedor de padrões de árvores, como no Exercício 3.32.

(1)	$reg_i \rightarrow const_c$	{ MOV #c, R*i* }
(2)	$reg_i \rightarrow mem_a$	{ MOV a, R*i* }
(3)	$mem \rightarrow := mem_a\ reg_i$	{ MOV R*i*, a }
(4)	$mem \rightarrow := ind\ reg_i\ reg_j$	{ MOV Rj, *Ri }
(5)	$reg_i \rightarrow ind + const_c\ reg_j$	{ MOV c(R*j*), R*i* }
(6)	$reg_i \rightarrow + reg_i\ ind + const_c\ reg_j$	{ ADD c(R*j*), R*i* }
(7)	$reg_i \rightarrow + reg_i\ reg_j$	{ ADD R*j*, R*i* }
(8)	$reg_i \rightarrow + reg_i\ const_1$	{ INC R*i* }

Fig. 9.32 Esquema de tradução dirigida pela sintaxe construído a partir da Fig. 9.31.

Os problemas de ordenamento e reconhecimento múltiplo podem ser resolvidos através do uso do reconhecimento de padrões de árvores em conjunto com o algoritmo de programação dinâmica da seção anterior. Um esquema de tradução de árvores pode ser expandido com informações de custo, associando-se a cada regra de reescrita de árvores o custo da seqüência de instruções de máquina gerada, se aquela regra for aplicada.

Na prática, o processo de reescrita de árvores pode ser implementado rodando-se o reconhecedor de padrões de árvores durante uma travessia em profundidade da árvore de entrada e realizando-se as reduções à medida que os nós são visitados pela última vez. Se rodarmos o algoritmo de programação dinâmica simultaneamente, podemos selecionar uma seqüência ótima de reconhecimentos usando as informações de custo associadas a cada regra. Podemos necessitar postergar a decisão a respeito de um reconhecimento até que o custo de todas as alternativas seja conhecido. Usando esta abordagem, um pequeno e eficiente gerador de código pode ser construído rapidamente a partir de um esquema de reescrita de árvores. Sobretudo, o algoritmo de programação dinâmica libera o projetista do gerador de código de ter que resolver reconhecimentos conflitantes ou decidir a respeito da ordem de avaliação.

Reconhecimento de Padrões Através da Análise Sintática

Uma outra abordagem é usar um analisador sintático LR para realizar o reconhecimento de padrões. A árvore de entrada pode ser tratada como uma cadeia de caracteres através do uso de sua representação prefixa. Por exemplo, a representação prefixa para a árvore da Fig. 9.30 é

$$:= ind\ +\ +\ const_a\ reg_{sp}\ ind\ +\ const_i\ reg_{sp}\ +\ mem_b\ const_i$$

O esquema de tradução de árvores pode ser convertido num esquema de tradução dirigida pela sintaxe substituindo-se as regras de reescrita de árvores por produções de uma gramática livre de contexto na qual os lados direitos sejam representações de prefixos dos gabaritos de instruções.

Exemplo 9.18 O esquema de tradução dirigida pela sintaxe da Fig. 9.32 está baseado no esquema de tradução de árvores da Fig. 9.31. □

A partir do esquema de tradução construiríamos um analisador sintático LR usando uma das técnicas de construção do Capítulo 4. O código-alvo é gerado através da emissão da instrução de máquina correspondente a cada redução.

Uma gramática de geração de código é usualmente ambígua e alguns cuidados devem ser tomados a respeito de como os conflitos das ações sintáticas são resolvidos quando o analisador sintático é construído. Na ausência de informações de custo, uma regra geral é a de favorecer as reduções maiores em detrimento das menores. Isto significa que num conflito reduzir-reduzir, a redução mais longa é preferida; num conflito empilhar-reduzir, o movimento de empilhar é escolhido. Esse enfoque de "estocagem máxima" faz com que um maior número de operações seja realizado por uma única instrução de máquina.

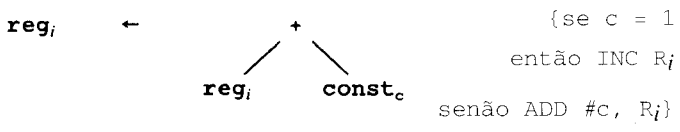
Existem vários aspectos ao se usar a técnica de análise sintática para a geração de código. Primeiro, o método de análise sintática é eficiente e bem conhecido e, por conseguinte, geradores de código confiáveis e eficientes podem ser criados usando-se os algoritmos descritos no Capítulo 4. Segundo, é relativamente fácil reorientar o gerador de código resultante; um seletor de código para uma nova máquina pode ser construído escrevendo-se uma gramática que descreva as instruções da nova máquina. Terceiro, o código gerado pode ser tornado eficiente adicionado-se produções tipo caso especial de forma que tire vantagem dos dialetos da máquina.

No entanto, existem algumas dificuldades, igualmente. Uma ordem de avaliação da esquerda para a direita é fixada pelo método de análise sintática. Adicionalmente, para algumas máquinas com um amplo número de métodos de endereçamento, a gramática de descrição da máquina e o analisador sintático resultante podem se tornar excessivamente grandes. Como conseqüência, técnicas especializadas são necessárias para se codificar e processar as gramáticas de descrição de máquinas. Precisamos também ser cuidadosos para que o analisador sintático resultante não fique bloqueado (não possua próximo movimento) ao analisar sintaticamente uma expressão de árvore em decorrência da gramática não tratar alguns padrões de operadores ou que o analisador sintático tenha tomado a resolução errada em algum conflito sintático. Precisamos também assegurar que o analisador sintático não entre num laço infinito de reduções de produções com símbolos únicos no lado direito. O problema do laço infinito pode ser resolvido usando-se uma técnica de particionamento de estados no tempo em que as tabelas sintáticas são geradas (ver Glanville [1977]).

Rotinas para a Verificação Semântica

As folhas na árvore de entrada são atributos de tipo com subscritos, onde um subscrito associa um valor a um atributo. Num esquema de tradução para geração de código, os mesmos atributos aparecem, mas, freqüentemente, com restrições sobre os valores que os subscritos possam ter. Por exemplo, uma instrução de máquina pode requerer que um atributo caia num intervalo ou que os valores de dois atributos sejam relacionados.

Essas restrições sobre os valores de atributos podem ser especificadas como predicados que são invocados antes de uma redução ser feita. De fato, o uso geral das ações e predicados semânticos pode conferir maior flexibilidade e facilidade de descrição do que uma pura especificação gramatical para um gerador de código. Os gabaritos genéricos podem ser usados para representar classes de instruções e as ações semânticas podem ser usadas para selecionar instruções para casos específicos. Por exemplo, duas formas da instrução de adição podem ser representadas como um gabarito:



Os conflitos de ações sintáticas podem ser resolvidos tornando inambíguos os predicados que permitem diferentes estratégias de seleção a serem usadas em diferentes contextos. Uma descrição menor da máquina-alvo é possível porque certos aspectos da arquitetura de máquina, tais como os modos de endereçamento, podem ser inseridos nos atributos. A complicação desse enfoque é que pode se tornar difícil verificar a correção da gramática como uma descrição fiel da máquina-alvo, apesar do problema ser compartilhado em algum grau por todos os geradores de código.

EXERCÍCIOS

9.1 Gere código para os seguintes enunciados C, para a máquina-alvo da Seção 9.2, assumindo que todas as variáveis sejam estáticas. Assuma que três registradores estejam disponíveis.

a) `x = 1`
b) `x = y`
c) `x = x + 1`
d) `x = a + b * c`
e) `x = a / (b + c) - d * (e + f)`

9.2 Repita o Exercício 9.1 assumindo que todas as variáveis sejam automáticas (reservadas na pilha).

9.3 Gere P-Code para os seguintes enunciados em C, para a máquina-alvo da Seção 9.2, assumindo que todas as variáveis sejam estáticas. Assuma que três registradores estejam disponíveis.

a) `x = a[i] + 1`
b) `a[i] = b [c[i]]`
c) `a[i][j] = b[i][k] * c[k][j]`
d) `a[i] = a[i] + b[j]`
e) `a[i] + = b[j]`

9.4 Faça o Exercício 9.1 usando

a) o algoritmo da Seção 9.6
b) o procedimento *gerar_código*
c) o algoritmo de programação dinâmica da Seção 9.11

9.5 Gere código para os seguintes enunciados C

a) `x = f(a) + f(a) + f(a)`
b) `x = f(a)/g(b,c)`
c) `x = f(f(a))`
d) `x = ++ f(a)`
e) `*pp++ = *q++`

9.6 Gere código para o seguinte programa C

```
main ()
{
     int i;
     int a[10];
     while (i <= 10)
          a[i] = 0;
}
```

9.7 Suponha que, para o laço da Fig. 9.13, optemos por alocar três registradores para abrigar `a`, `b` e `c`. Gere o código para os blocos daquele laço. Compare o custo de seu código com o daquele da Fig. 9.14.

9.8 Construa o grafo de interferência de registradores para o programa da Fig. 9.13.

9.9 Suponha que, por uma questão de simplicidade, armazenemos automaticamente todos os registradores na pilha (ou na memória, se uma pilha não for usada) antes de cada chamada de procedimento e os restauremos após retorno. Como isso iria afetar a fórmula (9.4), usada para avaliar a utilidade de se alocar um registrador a uma dada variável, dentro de um laço?

9.10 Modifique a função *obter_registrador* da Seção 9.6 para que retorne pares de registradores quando necessário.

9.11 Dê um exemplo de um GDA para o qual o processo heurístico de ordenamento de nós de um GDA, fornecido na Fig. 9.21, não devolve o melhor ordenameno possível.

***9.12** Gere um código ótimo para os seguintes enunciados de atribuição:

a) `x := a + b * c`
b) `x := (a * - b) + (c - (d + e))`
c) `x := (a / b - c) / d`
d) `x := a + (b + c/d*e) / (f*g - h*i)`
e) `a[i ,j] := b[i, j] - c[a[k, l]] * d[i+j]`

9.13 Gere o código para o seguinte programa Pascal:

```
program lacofor(input, output) ;
     var i, inicial, final : integer;
     begin
        read (inicial, final);
        for i := inicial to final do
           writeln(i)
end.
```

9.14 Construa o GDA para o seguinte bloco básico.

```
d := b * c
e := a + b
b := b * c
a := e - d
```

9.15 Quais são as ordens legais de avaliação e os nomes para os valores aos nós para o GDA do Exercício 9.14

a) assumindo que `a`, `b` e `c` estejam vivos ao final do bloco básico?
b) assumindo que somente `a` esteja vivo ao final?

9.16 No exercício 9.15(b), se estivermos gerando código para uma máquina com um registrador somente, que ordem de avaliação será a melhor?

9.17 Podemos modificar o algoritmo de construção de GDAs, de forma a levarmos em consideração as atribuições a *arrays* e através de apontadores. Quando qualquer elemento de um *array* participa de uma atribuição, assumimos que esse novo valor seja criado para aquele *array*. Esse novo valor é representado por um nó cujos filhos são o valor antigo do *array*, o valor do índice para o elemento do *array* e o valor atribuído. Quando uma atribuição ocorre através de um apontador, assumimos que criamos um novo valor para cada variável que aquele apontador poderia ter apontado; os filhos do nó para cada novo valor são o valor do apontador e o valor antigo da variável que poderia ter sido atribuída. Usando essas suposições, construa um GDA para o seguinte bloco básico:

```
a[i] := b
*p := c
d := a[j]
e := *p
*p := a[i]
```

Assuma que (a) `p` possa apontar para qualquer local, (b) `p` aponte somente para `b` ou `d`. Não se esqueça de mostrar a ordem implícita das restrições.

9.18 Se um apontador ou expressão de *array*, tal como `a[i]` ou `*p`, é atribuído e então usado sem a possibilidade de seu valor ter mudado nesse ínterim, podemos reconhecer a situação e tirar vantagem dela para simplificar o GDA. Por exemplo, no código do Exercício 9.17, uma vez que `p` não é atribuído entre o segundo e quarto enunciados, o enunciado `e := *p` pode ser substituído por `e := c`, já que estamos seguros de que, qualquer que seja a localização que `p` aponte, a mesma possui o

mesmo valor que `c`, mesmo que não saibamos para o que `p` aponta. Revise o algoritmo de construção de GDAs de forma a que tire vantagem de tais inferências. Aplique seu algoritmo ao código do Exercício 9.17.

****9.19** Conceba um algoritmo para gerar código ótimo para uma seqüência de enunciados de três endereços da forma `a := b + c`, na máquina de *n* registradores do Exemplo 9.14. Os enunciados têm que ser executados na ordem dada. Qual é a complexidade de tempo de seu algoritmo?

NOTAS BIBLIOGRÁFICAS

O leitor interessado em visões gerais sobre a pesquisa de geração de código deveria consultar Waite [1976a,b], Aho e Sethi [1977], Graham [1980 e 1984], Ganapathi, Fischer e Hennessy [1982], Lunell [1983] e Henry [1984]. A geração de código para Bliss é discutida por Wulf et al. [1975], para Pascal por Ammann [1977] e para PL.8 por Auslander e Hopkins [1982].

As estatísticas de uso de programas são úteis para o projeto de compiladores. Knuth [1971b] fez um estudo empírico de programas Fortran. Elshoff [1976] providencia alguns usos estatísticos sobre o uso de PL/I e Shimasaki et al. [1980] e Carter [1982] analisam programas Pascal. O desempenho de vários compiladores sobre diversos conjuntos de instruções é discutido por Lunde [1977], Shustec [1978] e Ditzel e McLellan [1982].

Muitos dos processos heurísticos para a geração de código propostos neste capítulo foram usados em vários compiladores. Freiburghouse [1974] discute as contagens de uso como um apoio na geração de código de qualidade para blocos básicos. A estratégia usada em *obter_reg* para criar um registrador livre, expulsando-se de um registrador a variável cujo valor permanecerá sem uso pelo período mais longo, foi mostrada ser ótima num contexto de troca de páginas* por Belady [1966]. Nossa estratégia de alocar um número fixo de registradores para guardar variáveis pela duração de um laço foi mencionada por Marill [1962] e usada na implementação de Fortran H por Lowry e Medlock [1969].

Horwitz et al. [1966] fornecem um algoritmo para otimizar o uso de registradores índice em Fortran. O colorimento de grafos como uma técnica de alocação de registradores foi proposto por J. Cocke, Ershov [1971] e Schwartz [1973]. O tratamento do colorimento de grafos da Seção 9.7 segue Chaitin et al. [1981] e Chaitin [1982]. Chow e Hennessy [1984] descrevem um algoritmo de colorimento de grafos baseado em prioridades para a alocação de registradores. Outros enfoques para a alocação de registradores são discutidos por Kennedy [1972], Johnson [1975], Harrison [1975], Beaty [1974] e Leverett [1982].

O algoritmo de rotulação para árvore da Seção 9.10 é uma reminiscência do algoritmo para denominar rios: a confluência de um rio maior e um afluente menor continua a usar o nome do maior rio; a confluência de dois rios iguais recebe um novo nome daí em diante. O algoritmo de rotulação apareceu originalmente em Ershov [1958]. Os algoritmos de geração de código que usam este método foram propostos por Anderson [1964], Nievergelt [1965], Nkata [1967], Redziejowski [1969] e Beaty [1972]. Sethi e Ullman [1970] usaram o método de rotulação num algoritmo que foram capazes de provar que gerava um código ótimo para árvores de expressões numa ampla variedade de situações. O procedimento *gerar_código* da Seção 9.10 é uma modificação de algoritmo de Sethi e Ullman, devido a Stockhausen [1973]. Bruno e Lassagne [1975] e Coffman e Sethi [1983] fornecem algoritmos de geração de código ótimo para árvores de expressões se a máquina-alvo possui registradores que precisam ser usados como uma pilha.

Aho e Johnson [1976] divisaram o algoritmo de programação dinâmica descrito na Seção 9.11. Este algoritmo foi usado como base para o gerador de código no compilador de C portável, PCC2 e foi também usado por Ripken [1977] num compilador para a máquina IBM / 370. Knuth [1977] generalizou o algoritmo de programação dinâmica para máquinas com classes assimétricas de registradores, tais como o IBM 7090 e o CDC 6600. Ao desenvolver a generalização, Knuth enxergou a geração de código como um problema de análise sintática para gramáticas livres de contexto.

Floyd [1961] fornece um algoritmo para tratar subexpressões comuns em expressões aritméticas. O desmembramento de GDAs em árvores e o uso de um procedimento como *gerar_código* separadamente sobre as árvores é proveniente de Waite [1976a]. Sethi [1975] e Bruno e Sethi [1976] mostram que o problema da geração de código ótimo para GDAs é NP-completo. Aho, Johnson e Ullman [1977a] mostram que o problema permanece NP-completo com máquinas de um registrador e de número infinito de registradores. Aho, Hopcroft e Ullman [1977a] e Garrey e Johnson [1979] discutem a importância do que significa para um problema ser NP-completo.

As transformações sobre os blocos foram estudadas por Aho e Ullman [1972a] e por Downey e Sethi [1978]. A otimização *peephole* é discutida por McKeeman [1965], Fraser [1979], Davidson e Fraser [1980 e 1984a,b] Lamb [1981] e Giegerich [1983]. Tanenbaum, van Staveren e Stevenson [1982] advogam o uso da otimização *peephole* também sobre o código intermediário.

A geração de código foi tratada como um processo de reescrita de árvore por Wasilew [1971], Weingart [1973], Johnson [1978] e Cattell [1980]. O exemplo de reescrita de árvores de Seção 9.12 é derivado de Henry [1984]. Aho e Ganapathi [1985] propuseram a combinação do reconhecimento eficiente de padrões de árvores com o método de programação dinâmica mencionado naquela mesma seção. Tjiang [1986] implementou uma linguagem de geração de código chamada Twig, baseada nos esquemas de tradução de árvores da Seção 9.12. Kron [1975], Huet e Levy [1979] e Hoffman e O'Donnell [1982] descrevem algoritmos gerais para o reconhecimento de padrões de árvores.

A abordagem Graham-Glanville para a geração de código utilizando um analisador sintático LR para a seleção de instruções é descrita e avaliada em Glanville [1977], Glanville e Graham [1978], Graham [1980 e 1984], Henry [1984] e Aigrain et al. [1984]. Ganapathi [1980] e Ganapathi e Fischer [1982] usaram gramáticas de atributos para especificar e implementar geradores de código.

Outras técnicas para automatizar a construção de geradores de código foram propostas por Fraser [1977], Cattel [1980] e Leverett et al. [1980]. A portabilidade de compiladores também é discutida por Richards [1971 e 1977]. Szymanski [1978] e Leverett e Szymanski [1980] descrevem técnicas para ligar instruções de desvio dependentes do desmembramento. Yannakakis [1985] possui um algoritmo de tempo polinomial para o exercício 9.19.

*Do original em inglês, *page swaping*. (N. do T.)

CAPÍTULO 10

OTIMIZAÇÃO DE CÓDIGO

Idealmente, os compiladores deveriam produzir um código-alvo que fosse tão bom quanto aquele que poderia ser escrito à mão. A realidade é que esta meta é atingida somente em casos limitados, e com dificuldade. No entanto, o código produzido pelos algoritmos diretos de compilação pode ser transformado de forma a rodar mais rapidamente, a ocupar menos espaço ou ambas as situações. Esse melhoramento é atingido através de transformações de programa que são tradicionalmente chamadas de *otimizações*, apesar do termo ser um equívoco, porque só raramente pode ser garantido que o código obtido seja o melhor possível. Os compiladores que aplicam transformações de melhoria de código são chamados de *compiladores otimizantes*.

A ênfase deste capítulo está nas otimizações independentes da máquina, isto é, nas transformações de programa que melhoram o código-alvo sem levar em consideração quaisquer propriedades da máquina-alvo. As otimizações dependentes da máquina, tais como a alocação de registradores e a utilização de seqüências especiais de instruções de máquina (dialetos da máquina) foram discutidas no Capítulo 9.

A maior recompensa em troca do menor esforço é conseguida se pudermos identificar as partes do programa mais freqüentemente usadas e as tornarmos tão eficientes quanto possível. Há um consenso popular que diz que a maioria dos programas gasta noventa por cento do seu tempo de execução em dez por cento de seu código.* Conquanto os percentuais efetivos possam variar, o caso é que freqüentemente uma pequena fração do programa contabiliza a maior parte do tempo de execução. O estabelecimento de um perfil de comportamento do programa em relação ao seu tempo de execução, usando-se amostras significativas de dados, identifica de forma acurada as regiões pesadamente atravessadas pelo mesmo. Infelizmente, um compilador não pode se beneficiar de dados de entrada que sirvam de exemplo, e, dessa forma, precisa fazer o seu melhor palpite a respeito de quais são os pontos críticos do programa.

Na prática, os laços mais internos do programa são bons candidatos para os melhoramentos. Numa linguagem que enfatize as construções de controle, tais como os enunciados *while* e *for*, os laços podem ser evidentes a partir da sintaxe do programa; em geral, um processo chamado de análise de fluxo de controle identifica os laços no grafo de fluxo de um programa.

Este capítulo é uma cornucópia de transformações otimizantes úteis e de técnicas para implementá-las. A melhor técnica para decidir que transformações valem a pena serem colocadas num compilador é coletar estatísticas a respeito dos programas-fonte e avaliar o benefício de um dado conjunto de otimização numa amostra representativa de programas-fonte reais. O Capítulo 12 descreve as transformações que foram comprovadamente úteis de serem usadas em compiladores otimizantes para várias linguagens diferentes.

Um dos temas deste capítulo é a análise de fluxo de dados, um processo de coletar informações a respeito das formas pelas quais as variáveis são usadas no programa. As informações coletadas nos vários pontos de um programa podem ser relacionadas usando-se um conjunto simples de equações. Apresentamos vários algoritmos para a coleta de informações usando a análise de fluxo de dados bem como para usar essas informações na otimização. Também consideramos o impacto, na otimização, das construções de linguagens tais como procedimentos e apontadores.

As últimas quatro seções deste capítulo lidam com material mais avançado. Cobrem algumas idéias da teoria dos grafos, relevantes à análise do fluxo de controle, e as aplicam à análise do fluxo de dados. O capítulo conclui com uma discussão das ferramentas de propósito geral para a análise do fluxo de dados e técnicas para depurar um código otimizado. A ênfase ao longo deste capítulo está nas técnicas de otimização que se aplicam às linguagens em geral. Alguns compiladores que usam essas idéias serão revistos no Capítulo 12.

10.1 INTRODUÇÃO

Para criar um programa eficiente numa linguagem-alvo, um programador precisa mais do que um compilador otimizante. Nesta seção, revisamos as opções disponíveis a um programador e a um compilador para criar programas-alvo eficientes. Mencionamos os tipos de transformações de melhoria de código que um programador e um escritor de compiladores poderiam esperar usar de forma a aprimorar o desempenho de um programa. Consideramos também as representações de programa nas quais as transformações podem ser aplicadas.

Critérios para as Transformações de Melhoria de Código

Enunciado de forma simplista, as melhores transformações de programa são aquelas que produzem o maior benefício com o menor esforço. As transformações providenciadas por um compilador otimizante devem possuir várias propriedades.

Primeiro, uma transformação precisa preservar o significado dos programas. Isto é, uma "otimização" não pode modificar a saída pro-

*Esse efeito, que ocorre em outras áreas totalmente irrelacionadas, tais como, por exemplo, a distribuição de terras, produção literária e científica e citações bibliográficas, é conhecido como *efeito Mateus* (também *lei dos 80/20* ou *90/10*). Esse nome tem origem numa citação bíblica. (N. do T.)

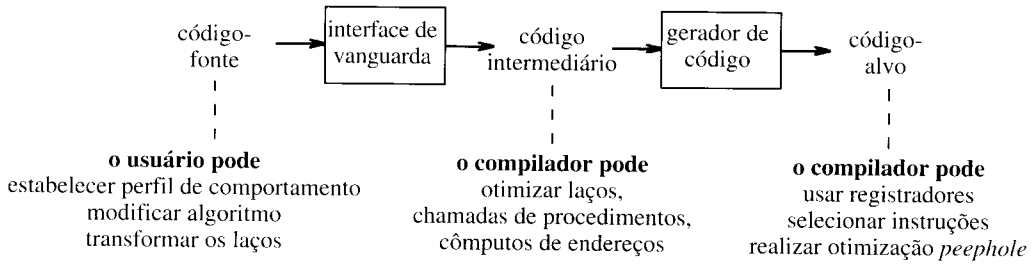


Fig. 10.1. Locais para melhorias potenciais por parte do usuário e do compilador.

duzida por um programa para uma dada entrada, ou causar um erro, tal como uma divisão por zero, que já não estivesse presente na versão original do programa-fonte. A influência deste critério se espalha por este capítulo; em todas as vezes, adotamos o critério "seguro" de perder uma oportunidade de aplicar uma transformação em vez de correr o risco de mudar o que o programa faz.

Segundo, uma transformação precisa, na média, acelerar os programas por um fator mensurável. Algumas vezes, estamos interessados em reduzir o espaço usado pelo código compilado, apesar do tamanho do código ter hoje menos importância do que tinha em outras épocas. Naturalmente, nem toda transformação tem sucesso em melhorar todo e qualquer programa e, ocasionalmente, uma "otimização" pode retardar ligeiramente um programa, à medida que a mesma melhora o estado de coisas em média.

Terceiro, uma transformação precisa valer o esforço. Não faz sentido que um escritor de compiladores dispenda esforço intelectual para implementar uma transformação de melhoria de código, e ter o compilador gastando tempo adicional de compilação com os programas-fonte, se esse esforço não for recompensado quando o programa-alvo for executado. Certas transformações locais ou *peephole*, do tipo discutido na Seção 9.9, são simples e benéficas o suficiente para serem incluídas em qualquer compilador.

Algumas transformações podem somente ser aplicadas após uma análise detalhada, freqüentemente demorada, do programa-fonte e, dessa forma, existe pouca discussão sobre se deveriam ser aplicadas a programas que serão rodados por apenas umas poucas vezes. Por exemplo, um compilador rápido, não otimizante, está propenso a ser mais útil ao longo de uma depuração ou quando usado em "aplicações de estudantes", que serão rodadas com sucesso umas poucas vezes e abandonadas. Somente quando o programa em questão ocupa uma fração significativa dos ciclos da máquina é que a qualidade do código melhorado justifica o tempo gasto rodando-se um compilador otimizante para o mesmo.

Obtendo Melhor Desempenho

Melhorias dramáticas no tempo de execução de um programa — tais como a redução do tempo de execução de umas poucas horas para uns poucos segundos — são usualmente obtidas melhorando-se o programa em todos os níveis, a partir do nível fonte até o do programa-alvo, como sugerido pela Fig. 10.1. A cada nível, as opções disponíveis caem entre os dois extremos de se encontrar um melhor algoritmo ou implementar um dado algoritmo específico de forma que um número menor de operações seja realizado.

As transformações algorítmicas produzem ocasionalmente melhorias espetaculares no tempo de execução. Por exemplo, Bentley [1982] relata que o tempo de execução de um programa para classificar N elementos caiu de $2.02N^2$ microssegundos para $12N\log_2 N$ microssegundos, quando uma "classificação de inserção" cuidadosamente codificada, foi substituída por um "*quicksort*".[1] Para $N = 100$, a substituição acelera o programa por um fator de 2.5. Para $N + 100.000$, a melhoria é substancialmente dramática: acelera o programa por um fator de mais de mil.

Infelizmente, nenhum compilador pode encontrar o melhor algoritmo para um dado programa. Algumas vezes, entretanto, um compilador pode substituir uma seqüência de operações por uma outra algebricamente equivalente e, em conseqüência, reduzir significativamente o tempo de execução de um programa. Tais ganhos são mais comuns quando são aplicadas transformações algébricas a programas em linguagens em nível muito alto, como, por exemplo, às linguagens de interrogação a bancos de dados (ver Ullman [1982]).

Nesta seção e na próxima, um programa de classificação chamado *quicksort* será usado para ilustrar o efeito das várias transformações de melhoria de código. O programa C da Fig. 10.2 é derivado de Sedgewick [1978], onde uma otimização manual do mesmo é discutida. Não iremos discutir aqui os aspectos algorítmicos do programa — de fato, `a[0]` precisa ter o menor elemento a ser classificado, e `a[max]` o maior, para que o programa funcione.

Pode não ser possível realizar certas transformações de melhoria no código ao nível da linguagem-fonte. Por exemplo, numa linguagem como Pascal ou Fortran, o programador pode somente se referir a elementos de um *array* da forma usual, como, por exemplo, `b[i,j]`. Ao nível da linguagem intermediária, entretanto, novas oportunidades para a melhoria de código podem ser expostas. O código de três endereços, por exemplo, faculta muitas oportunidades para a melhoria do cômputo de endereços, especialmente dentro de laços. Consideremos o código de três endereços para determinar o valor de `a[i]`, assumindo que cada elemento do *array* ocupe quatro *bytes*:

$$t_1 := 4*i;\ t_2 := a[t_1]$$

```
void quicksort(m,n)
int m,n;
{
    int i,j;
    int v,x;
    if ( n <= m ) return;
    /* o fragmento começa aqui */
    i = m-1; j = n; v = a[n];
    while(1) {
        do i = i+1; while ( a[i] <v );
        do j = j-1; while ( a[j] >v );
        if (i >= j ) break;
        x = a[i]; a[i] = a[j]; a[j] = x;
    }
    x = a[i]; a[i] = a[n]; a[n] = x;
    /* o fragmento termina aqui */
    quicksort(m, j); quicksort(i+1, n);

}
```

Fig. 10.2. Código C para *quicksort*.

[1] Ver Aho, Hopcroft e Ullman [1983] para uma discussão desses algoritmos e suas velocidades.

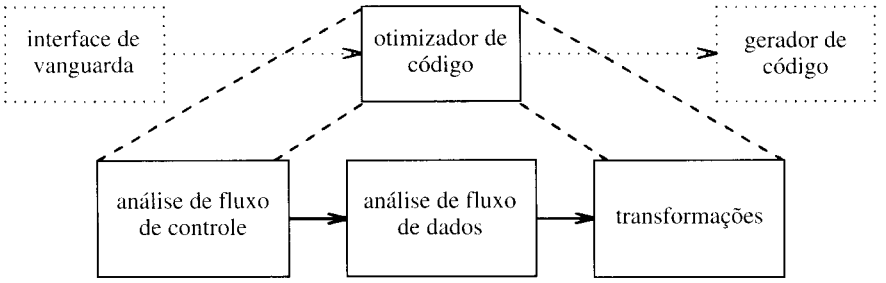


Fig. 10.3. Organização do otimizador de código.

Um código ingênuo de três endereços irá recalcular `4*i` a cada vez que `a[i]` figurar no programa-fonte, e o programador não possui controle sobre os cômputos redundantes de endereços uma vez que estão implícitos na implementação da linguagem, em vez de serem explícitos no código escrito pelo usuário. Nessas situações, é mister do compilador enxugar todos esses cômputos redundantes. Numa linguagem como C, entretanto, essa transformação pode ser feita ao nível da linguagem-fonte, uma vez que as referências aos elementos de *arrays* podem ser sistematicamente reescritas usando apontadores para tornálas mais eficientes. Essa reescrita é similar às transformações que os compiladores otimizantes de Fortran tradicionalmente aplicam.

Ao nível da máquina-alvo, é responsabilidade do compilador fazer o bom uso de seus recursos. Por exemplo, manter as variáveis mais pesadamente usadas em registradores pode reduzir significativamente o tempo de execução, freqüentemente tanto quanto à metade. Novamente, C permite que o programador avise o compilador para manter certas variáveis guardadas em registradores, mas a maioria das linguagens não o faz. Semelhantemente, o compilador pode acelerar significativamente os programas, escolhendo instruções que tirem vantagem dos modos de endereçamento da máquina, realizando em uma instrução o que ingenuamente esperaríamos requerer duas ou três, como discutido no Capítulo 9.

Mesmo que seja possível para o programador melhorar o código, pode ser mais conveniente deixar o compilador realizar algumas das melhorias. Se a um compilador pode ser confiada a geração de um código eficiente, o usuário pode se concentrar em escrever um código correto.

Uma Organização para um Compilador Otimizante

Como mencionamos, existem freqüentemente vários níveis nos quais um programa pode ser melhorado. Como as técnicas para analisar e transformar um programa não mudam significativamente com o nível, este capítulo se concentra nas transformações do código intermediário usando a organização mostrada na Fig. 10.3. A fase de melhoramento do código consiste nas análises de fluxo de controle e de fluxo de dados, seguidas pela aplicação de transformações. O gerador de código discutido no Capítulo 9 produz um programa-alvo a partir do código intermediário transformado.

Por uma questão de conveniência da apresentação, assumimos que o código intermediário consista em um código de três endereços. O código intermediário, do tipo produzido pelas técnicas do Capítulo 8 para uma parte do programa da Fig. 10.2, é mostrado na Fig. 10.4. Com outras representações intermediárias, as variáveis temporárias t_1, t_2, ..., t_{15} não precisam figurar explicitamente, como discutido no Capítulo 8.

A organização da Fig. 10.3 possui as seguintes vantagens:

1. As operações necessárias para implementar as construções de alto nível são tornadas explícitas no código intermediário, e, dessa forma, é possível otimizá-las. Por exemplo, os cômputos de endereços para `a[i]` são explícitos na Fig. 10.4 e o recômputo de expressões como `4*i` pode ser eliminado, como discutido na próxima seção.

2. O código intermediário pode ser (relativamente) independente da máquina-alvo, de forma que o otimizador não tenha que mudar muito se o gerador de código for substituído por um outro de uma máquina diferente. O código intermediário da Fig. 10.4 assume que cada elemento de um *array* `a` ocupe quatro *bytes*. Alguns códigos independentes, como, por exemplo, o Código-P para Pascal, deixam para o gerador de código preencher o tamanho dos elementos de *arrays*, de forma que o código intermediário seja independente do tamanho da palavra de máquina. Poderíamos ter feito o mesmo em nosso código intermediário se substituíssemos 4 por uma constante simbólica.

```
(1)  i := m-1
(2)  j := n
(3)  t1 := 4*n
(4)  v := a[t1]
(5)  i := i+1
(6)  t2 := 4*i
(7)  t3 := a[t2]
(8)  if t3 < v goto (5)
(9)  j := j-1
(10) t4 := 4*j
(11) t5 := a[t4]
(12) if t5 > v goto (9)
(13) if i >= j goto (23)
(14) t6 := 4*i
(15) x := a[t6]
(16) t7 := 4*i
(17) t8 := 4*j
(18) t9 := a[t8]
(19) a[t7] := t9
(20) t10 := 4*j
(21) a[t10] := x
(22) goto (5)
(23) t11 := 4*i
(24) x := a[t11]
(25) t12 := 4*i
(26) t13 := 4*n
(27) t14 := a[t13]
(28) a[t12] := t14
(29) t15 := 4*n
(30) a[t15] := x
```

Fig. 10.4. Código de três endereços para o fragmento de código da Fig. 10.2.

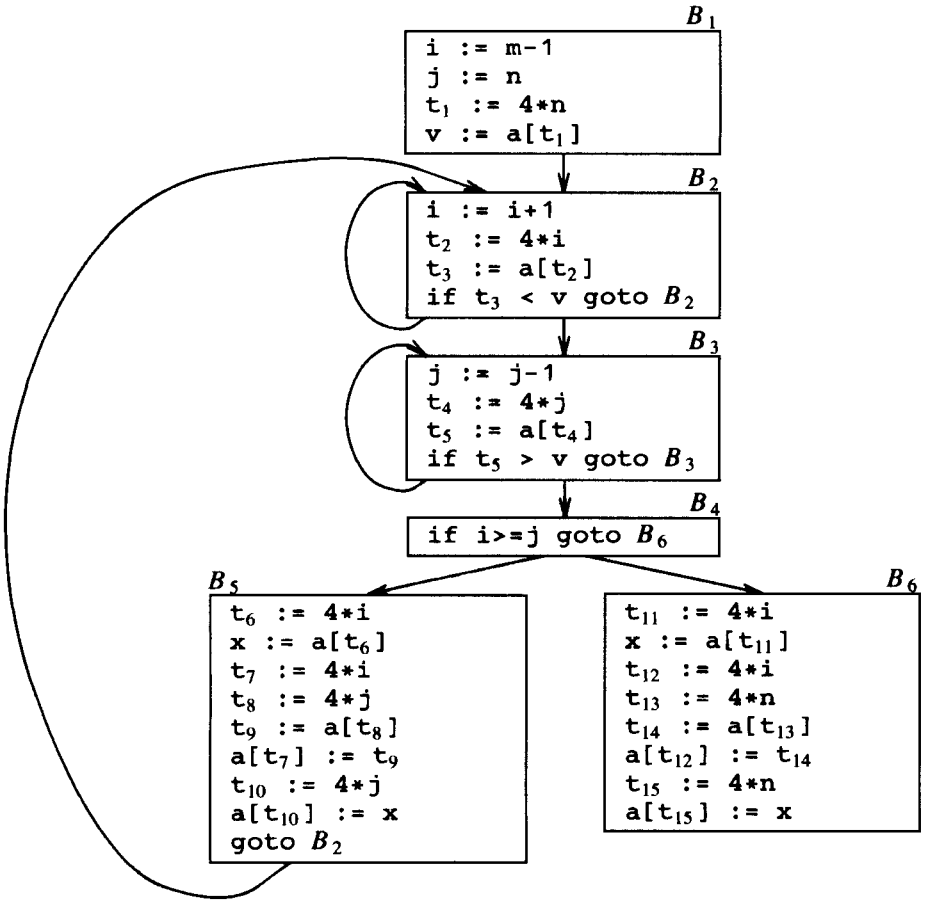


Fig. 10.5. Grafo de fluxo.

No otimizador de código, os programas são representados por grafos de fluxo, nos quais os lados representam os blocos básicos, como discutido na Seção 9.4. A menos que especificado de outra forma, um programa significa um único procedimento. Na Seção 10.8, discutimos a otimização interprocedimental.

Exemplo 10.1. A Fig. 10.5 contém o grafo de fluxo para o programa da Fig. 10.4. B_1 é o nó inicial. Todos os desvios condicionais e incondicionais para os enunciados na Fig. 10.4 foram substituídos, na Fig. 10.5, por desvios para os blocos nos quais os enunciados são os líderes.

Na Fig. 10.5 existem três laços. B_2 e B_3 são laços por si mesmos. Os blocos B_2, B_3 e B_4 e B_5 formam, juntos, um laço, com entrada em B_2. □

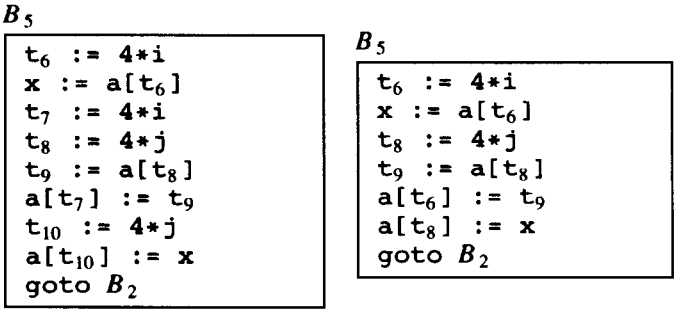
10.2 AS PRINCIPAIS FONTES DE OTIMIZAÇÃO

Nesta seção, introduzimos algumas das mais úteis transformações para melhoria de código. As técnicas para implementar essas transformações são apresentadas nas seções subseqüentes. Uma transformação de programa é chamada *local* se puder ser realizada examinando-se apenas os enunciados que constituem o bloco básico; caso contrário, é chamada de *global*. Muitas transformações podem ser realizadas em ambos os níveis, local e global. As transformações locais são usualmente realizadas primeiro.

Transformações Função-Preservantes

Existem várias formas através das quais um compilador pode melhorar um programa sem modificar a função que o mesmo computa. A eliminação de subexpressões comuns, a propagação de cópias, a eliminação de código morto e a transposição para constantes são exemplos comuns de tais transformações função-preservantes. A Seção 9.8 sobre a representação de blocos básicos sob a forma de GDAs mostrou como as subexpressões comuns locais poderiam ser removidas à medida que o GDA para o bloco básico viesse sendo construído. As outras transformações surgem primariamente quando as otimizações globais são realizadas, e discutiremos uma de cada vez.

Um programa irá incluir freqüentemente vários cálculos do mesmo valor, tais como o deslocamento de um *array*. Como mencionado na Seção 10.1, alguns desses cômputos não podem ser evitados pelo programador, porque recaem abaixo do nível de detalhe acessível dentro da linguagem-fonte. Por exemplo, o bloco B_5, mostrado na Fig. 10.6(a), recalcula `4*i` e `4*j`.



(a) Antes (b) Depois

Fig. 10.6. Eliminação de subexpressões comuns locais.

Subexpressões Comuns

A ocorrência de uma expressão E é chamada de uma *subexpressão comum* se E foi computada previamente e os valores das variáveis em E não mudaram desde o cômputo anterior. Podemos evitar o recômputo de uma expressão se pudermos usar o valor previamente calculado. Por exemplo, as atribuições a t_7 e t_{10} possuem as subexpressões comuns `4*i` e `4*j`, respectivamente, no lado direito da Fig. 10.6(a). Foram eliminadas na Fig. 10.6(b), usando-se t_6, em lugar de t_7, e t_8, em lugar de t_{10}. Essa mudança é o que iria resultar se reconstruíssemos o código intermediário a partir do GDA para o bloco básico.

Exemplo 10.2. A Fig. 10.7 mostra o resultado de se eliminar as subexpressões comuns, tanto locais quanto globais, dos blocos B_5 e B_6, no grafo de fluxo da Fig. 10.5. Discutimos primeiro a transformação de B_5 e, em seguida, mencionamos algumas das sutilezas que envolvem os *arrays*.

Depois que as subexpressões locais comuns forem eliminadas, B_5 ainda estará avaliando `4*i` e `4*j`, como mostrado na Fig. 10.6(b). Ambas são subexpressões comuns; em particular, os três enunciados

$$t_8 := 4*j;\ t_9 := a[t_8];\ a[t_8] := x$$

em B_5, podem ser substituídos por

$$t_9 := a[t_4];\ a[t_4] := x$$

usando-se t_4, computado no bloco B_3. Na Fig. 10.7, observemos que, à medida que o controle passa da avaliação de `4*j` em B_3 para B_5, não existe mudança em `j` e, dessa forma, t_4 pode ser usado quando `4*j` for necessário.

Uma outra subexpressão comum vem à luz em B_5, após t_4 substituir t_8. A nova expressão $a[t_4]$ corresponde, ao nível de código-fonte, ao valor de `a[j]`. Não somente `j` retém seu valor, quando o controle abandona B_3 e, em seguida, entra em B_5, mas, também, `a[j]`, que é um valor computado na variável temporária t_5, porque não existem atribuições a elementos do *array* `a` nesse interim. Os enunciados

$$t_9 := a[t_4];\ a[t_6] := t_9$$

em B_5 podem ser, por conseguinte, substituídos por

$$a[t_6] := t_5$$

Analogamente, o valor atribuído a `x` no bloco B_5 da Fig. 10.6(b) é visto ser o mesmo que o valor atribuído a t_3, no bloco B_2. O bloco B_2, na Fig. 10.7, é o resultado da eliminação das subexpressões comuns correspondentes aos valores das expressões em nível fonte `a[i]` e `a[j]`, de B_5, na Fig. 10.6(b). Uma série similar de transformações foi aplicada a B_6, na Fig. 10.7.

A expressão $a[t_1]$, nos blocos B_1 e B_6 da Fig. 10.7, não é considerada uma subexpressão comum, apesar de t_1 poder ser usada em ambos os locais. O controle, após deixar B_1 e antes de atingir B_6, pode ir através de B_5, onde existem atribuições a `a`. Por conseguinte, $a[t_1]$ pode não ter o mesmo valor ao atingir B_6, como tinha ao deixar B_1, e não é seguro tratar $a[t_1]$ como uma subexpressão comum. □

Propagação de Cópias

O bloco B_5 na Fig. 10.7 pode ser subseqüentemente melhorado através da eliminação de `x`, usando-se duas novas transformações. Uma se relaciona às atribuições da forma `f:=g`, chamadas de *enunciados de cópia* ou *cópias*, abreviadamente. Tivéssemos ido mais a fundo nos detalhes do Exemplo 10.2, as cópias teriam emergido mais cedo, porque o algoritmo para eliminar as subexpressões comuns as introduz, assim como o fazem vários outros algoritmos. Por exemplo, quando a subexpressão comum em `c:= d + e` é eliminada na Fig. 10.8, o algoritmo usa uma nova variável `t` para guardar o valor de `d + e`. Como o controle pode atingir `c:= d + e`, quer depois da atribuição a `a` ou após a atribuição a `b`, seria incorreto substituir `c:= d + e` por `c:= a` ou por `c:= b`.

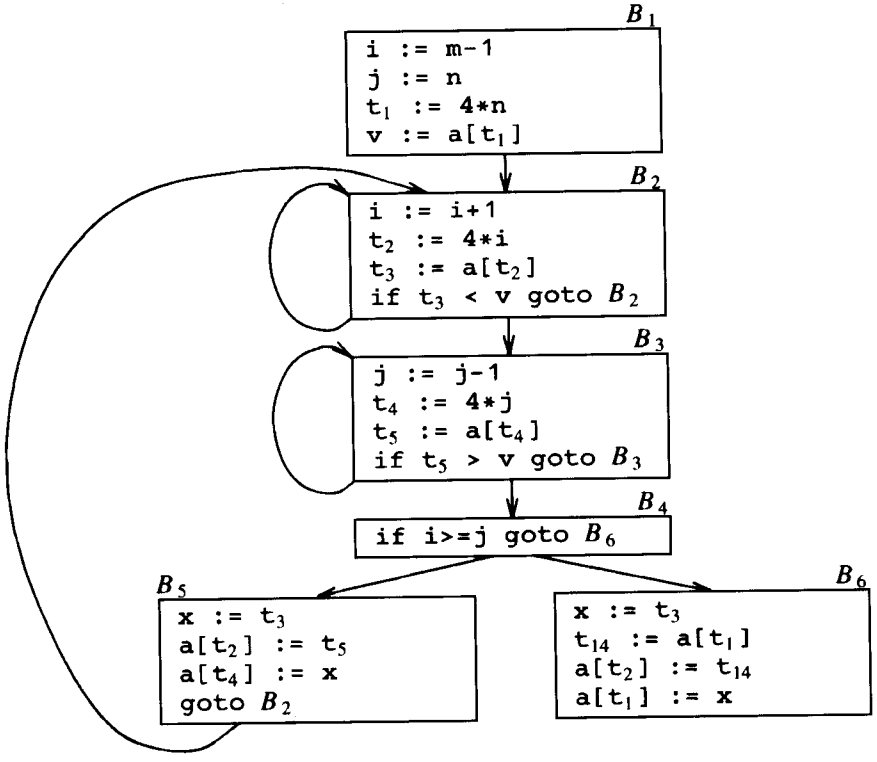


Fig. 10.7. B_5 e B_6 após a eliminação de subexpressões comuns.

A idéia por trás da transformação da propagação de cópias é usar `g` em lugar de `f` sempre que possível após o enunciado de cópia `f:=g`. Por exemplo, a atribuição $x := t_3$ no bloco B_5 da Fig. 10.7 é uma cópia. A propagação de cópias aplicada a B_5 produz:

$$\begin{array}{l} x := t_3 \\ a[t_2] := t_5 \\ a[t_4] := t_3 \\ \text{goto } B_2 \end{array} \qquad (10.1)$$

Isto pode não parecer uma melhoria, mas, como veremos, nos dá a oportunidade de eliminar a atribuição a `x`.

Eliminação de Código Morto

Uma variável está viva a um ponto do programa se seu valor (potencialmente) puder ser usado subseqüentemente; em caso contrário estará morta àquele ponto. Uma idéia relacionada é a do código morto ou inútil, isto é, enunciados que computam um valor que jamais será usado. Enquanto o programador é pouco propenso a incluir qualquer código morto de forma intencional, o mesmo pode surgir como resultado de transformações prévias. Por exemplo, na Seção. 9.9, discutimos o uso de `debug`, que é estabelecida para verdadeiro ou falso em vários pontos no programa e usada em enunciados como

```
if (debug) print ...          (10.2)
```

Através de uma análise de fluxo de dados, pode ser possível deduzir que, a cada vez que o programa atinja este enunciado, o valor de `debug` seja falso. Usualmente, o motivo está em que existe um enunciado particular

```
debug := false
```

que podemos deduzir que seja a última atribuição de `debug` antes do teste (10.2). Não importa que seqüência de ramificações o programa efetivamente siga. Se uma propagação de cópias substituir `debug` por falso, o enunciado de impressão estará morto porque não poderá ser atingido. Podemos eliminar, do código objeto, tanto o teste quanto a impressão. Mais geralmente, deduzir, em tempo de compilação, que o valor de uma expressão é uma constante e usar a constante em seu lugar é conhecido como *transposição para constantes.**

Uma vantagem da propagação de cópias é que, freqüentemente, transforma o enunciado de cópia em código morto. Por exemplo, a propagação de cópias, seguida pela eliminação de código morto, remove a atribuição a `x` e transforma (10.1) em:

$$\begin{array}{l} a[t_2] := t_5 \\ a[t_4] := t_3 \\ \text{goto } B_2 \end{array}$$

Este código representa uma melhoria subseqüente do bloco B_5 na Fig. 10.7.

Otimizações de Laços

Fornecemos agora uma breve introdução a um local muito importante para as otimizações, nominalmente os laços, e especialmente os mais internos, onde os programas tendem a gastar o grosso de seus tempos. O tempo de execução de um programa pode ser melhorado se decrementarmos o número de instruções de um laço mais interno, ainda que aumentemos a quantidade de código fora do mesmo. Três técnicas são importantes para a otimização de laços: a *movimentação de código,* que move código para fora de um laço; a *eliminação das variáveis de indução,* que aplicamos para eliminar `i` e `j` dos laços mais internos B_2 e B_3 da Fig. 10.7; e a *redução de capacidade,* a qual substitui uma operação mais cara e complexa por uma outra mais barata e simples, tal como uma multiplicação por uma adição.

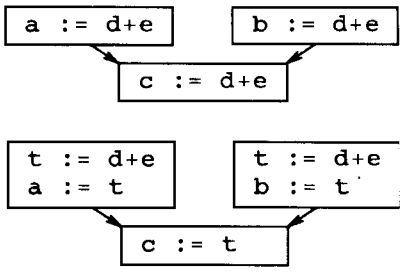


Fig. 10.8. Cópias introduzidas durante a eliminação de subexpressões comuns.

Movimentação de Código

Uma importante modificação, que diminui a quantidade código num laço, é a movimentação de código. Esta transformação toma uma expressão que produz o mesmo resultado independentemente do número de vezes que o laço é executado (*uma computação laço-invariante*) e a coloca antes do mesmo. Note-se que a noção "antes do laço" assume a existência de uma entrada para o laço. Por exemplo, a avaliação de `limite-2` é um cômputo laço-invariante no seguinte enunciado *while*:

```
while (i <= limite-2) /* o enunciado não muda limite */
```

A movimentação de código iria resultar no código equivalente a

```
t = limite-2:
while ( i <= t) /* o enunciado não muda limite ou t */
```

Variáveis de Indução e Redução de Capacidade

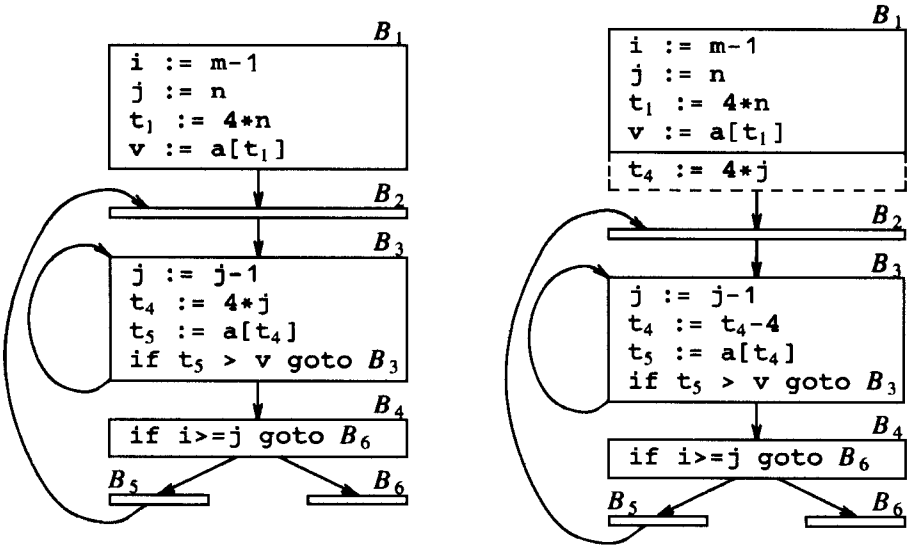
Conquanto a movimentação de código não seja aplicável ao exemplo do *quicksort* que temos considerado, as duas outras transformações o são. Os laços são processados de fora para dentro. Por exemplo, consideremos o laço em torno de B_3. Somente a parte do grafo de fluxo que é relevante às transformações sobre B_3 é mostrada na Fig. 10.9.

Note-se que os valores de `j` e de t_4 permanecem par e passo; a cada vez que o valor de `j` decresce de 1, o de t_4 decresce de 4, já que `4*j` é atribuído a t_4. Tais identificadores são chamados de *variáveis de indução.*

Quando existem duas ou mais variáveis de indução num laço, pode ser possível nos livrarmos de todas menos uma, através do processo de eliminação das variáveis de indução. Para o laço mais interno em torno de B_3, na Fig. 10.9(a), não podemos nos livrar de `j` ou de t_4 completamente; t_4 é usada em B_3 e `j` em B_4. No entanto, podemos ilustrar uma parte do processo de eliminação de variáveis de indução. Eventualmente, `j` será eliminada quando o laço mais externo de B_2–B_5 for considerado.

Exemplo 10.3. Na medida em que o relacionamento t_4= `4*j` é certamente válido após uma tal atribuição a t_4, na Fig. 10.9(a), e t_4 não é mudado em qualquer outra parte no laço mais interno, ao longo de B_3, segue-se que exatamente após o enunciado `j := j-1` a relação t_4=`4*j - 4` terá que ser válida. Podemos, então, substituir a atribuição t_4 := `4*j` por t_4 := t_4-4. O único problema está em que t_4 não tem um valor ao entrarmos no bloco B_3 pela primeira vez. Como precisamos manter a relação t_4 := `4*j` à entrada do bloco B_3, colocamos uma inicialização para t_4 ao fim do bloco onde o próprio `j` é inicializado, mostrada na inclusão, pontilhada, ao bloco B_1 na Fig. 10.9(b).

*Do original em inglês: *constant folding.* (N. do T.)



(a) Antes (b) Depois

Fig. 10.9. Redução de capacidade aplicada a `4*j` no bloco B_3.

A substituição de uma multiplicação por uma subtração irá acelerar o código objeto se a multiplicação levar mais tempo do que a adição ou subtração, como é o caso de muitas máquinas. □

A Seção 10.7 discute como as variáveis de indução podem ser detectadas e que transformações podem ser aplicadas. Concluímos esta seção com mais um exemplo de eliminação de variáveis de indução, o qual trata `i` e `j` no contexto do laço mais externo contendo B_2, B_3, B_4 e B_5.

Exemplo 10.4. Após a redução de capacidade ser aplicada aos laços mais internos em torno de B_2 e B_3, o único uso de `i` e `j` é determinar o resultado do teste no bloco B_4. Sabemos que os valores de `i` e de t_2 satisfazem a relação t_2= `4*i`, enquanto que aqueles de `j` e t_4 satisfazem a relação t_4= `4*j`, de forma que o teste t_2>=t_4 é equivalente a `i >= j`. Uma vez que essa substituição seja feita, `i`, no bloco B_2, e `j`, em B_3, se tornam variáveis mortas e as atribuições às mesmas nesses blocos se tornam código morto, o qual pode ser eliminado, resultando no grafo de fluxo mostrado na Fig. 10.10. □

As transformações de melhoria de código foram efetivas. Na Fig. 10.10, o número de instruções nos blocos B_2 e B_3 reduziu-se de 4 para 3 em cada um, a partir do grafo de fluxo original na Fig. 10.5; em B_5, reduziu-se de 9 para 3 e em B_6, de 8 para 3. Por outro lado, B_1 cresceu de quatro para seis instruções, mas o mesmo só é executado uma vez no fragmento de código, de forma que o tempo total de execução é só levemente afetado pelo tamanho de B_1.

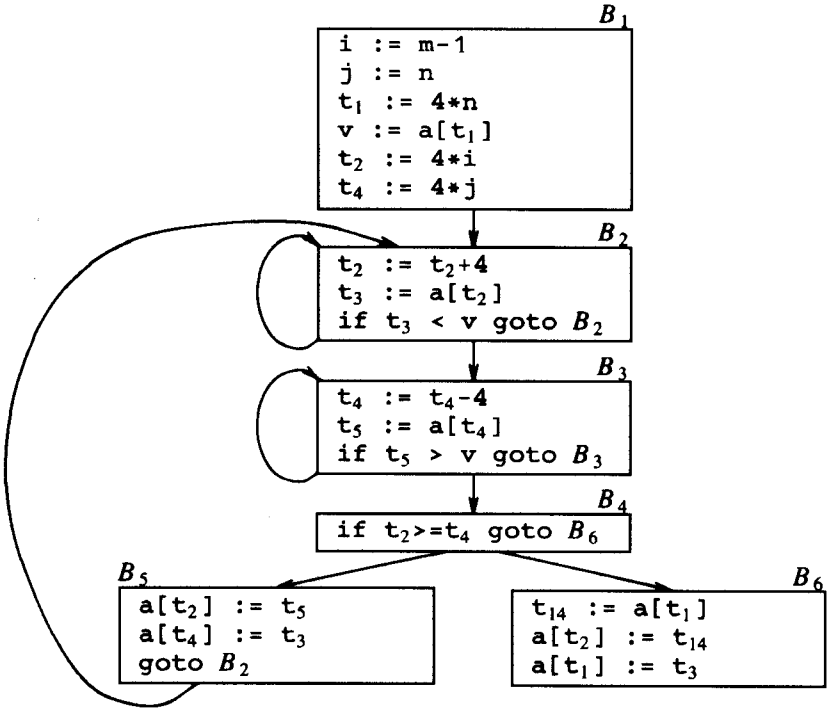


Fig. 10.10. Grafo de fluxo após a eliminação de variáveis de indução.

10.3 OTIMIZAÇÃO DOS BLOCOS BÁSICOS

No Capítulo 9, examinamos várias transformações de melhoria de código para os blocos básicos. Incluíam transformações estrutura-preservantes, tais como a eliminação de código morto e as transformações algébricas, tais como a redução de força.

Muitas das transformações estrutura-preservantes podem ser implementadas através da construção de um GDA para um bloco básico. Relembremos que existe um nó no GDA para cada um dos valores iniciais das variáveis que aparecem no bloco básico e que existe um nó *n* associado a cada enunciado *s* dentro do bloco. Os filhos de *n* são aqueles nós que correspondem aos enunciados que são as últimas definições, antes de *s*, dos operandos usados por *s*. O nó *n* é rotulado pelo operador aplicado a *s* e, também atrelada a *n*, está a lista de variáveis para as quais esta seja a última definição dentro do bloco. Também notamos aqueles nós, se algum, cujos valores estejam vivos à saída do bloco; esses são os nós de saída.

As subexpressões comuns podem ser detectadas verificando-se, na medida em que um novo nó *m* esteja para ser adicionado, se há um nó existente *n* com os mesmos filhos, na mesma ordem, e com o mesmo operador. Se assim o for, *n* computa o mesmo valor que *m* e pode ser usado em seu lugar.

Exemplo 10.5. Um GDA para o bloco (10.3)

```
a := b + c
b := a - d          (10.3)
c := b + c
d := a - d
```

é mostrado na Fig. 10.11. Quando construímos o nó para o terceiro enunciado, `c:=b+c`, sabemos que o uso de `b` em `b+c` se refere ao nó da Fig. 10.11 rotulado $-$, porque esta é a definição mais recente de `b`. Por conseguinte, não confundimos os valores computados nos enunciados um e três.

No entanto, o nó correspondente ao quarto enunciado, `d := a-d`, possui o operador $-$ e os nós rotulados `a` e d_0 como filhos. Uma vez que o operador e os filhos são os mesmos que aqueles para o nó correspondente ao enunciado dois, não criamos este nó, mas adicionamos `d` à lista de definições para o nó rotulado $-$. □

Poderia parecer que, como existem somente três nós no GDA da Fig. 10.11, o bloco (10.3) pudesse ser substituído por um bloco com somente três enunciados. De fato, se `b` ou `d` (uma das duas) estiver viva à saída do bloco, não precisamos computar aquela variável e podemos usar a outra para receber o valor representado pelo nó rotulado $-$, na Fig. 10.11. Por exemplo, se `b` não estivesse viva à saída do bloco, poderíamos usar:

```
a := b + c
d := a - d
c := d + c
```

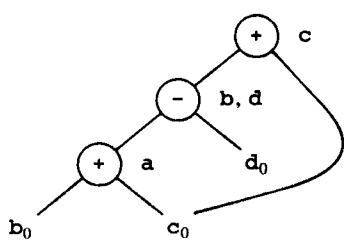


Fig. 10.11. Um GDA para o bloco básico (10.3).

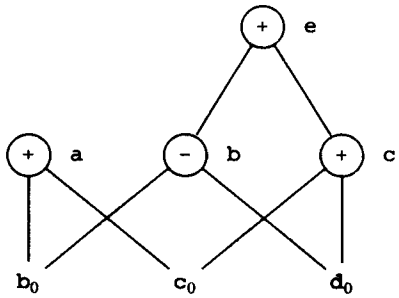


Fig. 10.12. GDA para o bloco básico (10.4).

No entanto, se ambas, `b` e `d`, estiverem vivas à saída, um quarto enunciado precisa ser usado para copiar o valor de uma para outra.[2]

Note-se que, quando procuramos por subexpressões comuns, estamos realmente procurando por expressões que garantam computar o mesmo valor, não importa como o valor seja computado. Por conseguinte, o método do GDA irá omitir o fato de que a expressão computada pelos primeiro e quarto enunciados na seqüência

```
a := b + c
b := b - d
c := c + d          (10.4)
e := b + c
```

é a mesma, nominalmente, `b + c`. No entanto, as identidades algébricas aplicadas ao GDA, como discutido em seguida, podem expor a equivalência. O GDA para esta seqüência é mostrado na Fig. 10.12.

A operação sobre os GDAs que corresponde à eliminação de código morto é claramente direta de se implementar. Removemos do GDA qualquer raiz (nó sem ancestrais) que não tenha variáveis vivas. A repetida aplicação desta transformação irá remover todos os nós do GDA que correspondam a código morto.

O Uso de Identidades Algébricas

As identidades algébricas representam uma outra classe importante de otimizações sobre os blocos básicos. Na Seção 9.9, introduzimos algumas transformações algébricas simples que poderiam ser tentadas durante uma otimização. Por exemplo, podemos aplicar identidades aritméticas, tais como

```
x + 0 = 0 + x = x
x - 0 = x
x * 1 = 1 * x = x
x / 1 = x
```

Outra classe de otimizações algébricas inclui a redução de capacidade, isto é, a substituição de um operador mais dispendioso por um mais econômico, como em

```
x ** 2 = x * x
2.0 * x = x + x
x / 2 = x * 0.5
```

Uma terceira classe de otimizações relacionadas é a transposição para constantes. Aqui, avaliamos as expressões constantes em tempo

[2]Em geral, temos que ser criteriosos ao reconstruir o código a partir dos GDAs, de forma a escolhermos cuidadosamente os nomes das variáveis correspondentes aos nós. Se uma variável `x` for definida duas vezes, ou se for atribuída uma vez e o valor inicial x_0 for também usado, precisamos assegurar que não modificamos o valor de `x` até que tenhamos feito todos os usos do nó cujo valor `x` detinha previamente.

de compilação e as substituímos por seus valores.[3] Por conseguinte, a expressão `2*3.14` seria substituída por `6.28`. Muitas expressões constantes emergem durante o uso de constantes simbólicas.

O processo de construção de GDAs pode nos auxiliar nessas e em outras transformações algébricas mais gerais, tais como as comutativas e as associativas. Por exemplo, suponhamos que `*` seja comutativa, isto é, `x*y=y*x`. Antes de criarmos um novo nó rotulado `*` com filho à direita *m* e filho à esquerda *n*, verificamos se um nó já não existe. Procuramos, em seguida, por um nó tendo operador `*`, filho à direita *n* e filho à esquerda *m*.

Os operadores relacionais `<=`, `>=`, `<`, `>`, `=` e ≠ algumas vezes geram subexpressões comuns inesperadas. Por exemplo, a condição `x>y` pode também ser testada pela subtração de argumentos, e realizando-se um teste no código de condição estabelecido pela subtração. (A subtração pode, no entanto, introduzir estouros de capacidade ou resultados abaixo do valor mínimo representável ou irrepresentáveis a uma dada precisão — *underflow* — que uma instrução de comparação não produziria). Por conseguinte, somente um nó do GDA seria gerado para `x-y` e `x>y`.

As leis associativas também podem ser aplicadas de forma a expor as subexpressões comuns. Por exemplo, se o código-fonte possui as atribuições

```
a := b + c
e := c + d + b
```

o seguinte código intermediário poderia ser gerado:

```
a := b + c
t := c + d
e := t + b
```

Se `t` não for necessitado fora do bloco, mudamos a seqüência para

```
a := b + c
e := a + d
```

usando a associatividade e comutatividade de +.

O escritor de compiladores deveria examinar cuidadosamente a especificação da linguagem, de forma a determinar que rearrumações das computações são permitidas, uma vez que a aritmética de computadores nem sempre obedece às identidades algébricas da matemática. Por exemplo, o padrão para Fortran 77 estabelece que um compilador pode avaliar qualquer expressão matematicamente equivalente, providenciado que a integridade dos parênteses não seja violada. Por conseguinte, um compilador pode avaliar `x*y-x*z` como `x*(y-z)`, mas não pode avaliar `a+(b-c)` como `(a+b)-c`. Um compilador Fortran precisa, conseqüentemente, controlar onde os parênteses estavam presentes nas expressões da linguagem-fonte, se deve otimizar os programas de acordo com a definição da linguagem.

10.4 LAÇOS EM GRAFOS DE FLUXO

Antes de considerar as otimizações de laços, precisamos definir no que se constitui um laço num grafo de fluxo. Usaremos a noção de um nó "dominando" um outro para definir os "laços naturais" e uma importante classe especial de grafos de fluxo "redutíveis". Um algoritmo para se encontrar os dominadores e verificar a redutibilidade dos grafos de fluxo será fornecido na Seção 10.9.

[3] As expressões aritméticas deveriam ser avaliadas da mesma forma que o são em tempo de execução. K. Thompson sugeriu uma solução elegante para a transposição para constantes: compilar a expressão constante, executar o código-alvo em foco e substituir a expressão pelo resultado. O compilador, por conseguinte, não precisa conter um interpretador.

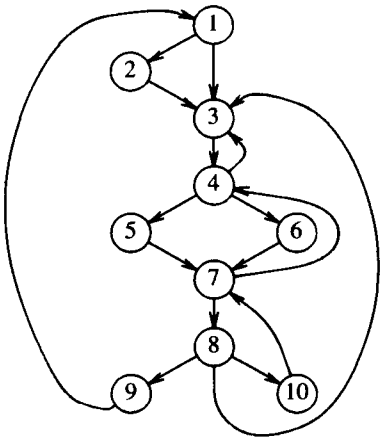


Fig. 10.13. Grafo de fluxo.

Dominadores

Dizemos que um nó *d* de um grafo de fluxo *domina* um nó *n*, escrito *d domina n*, se cada percurso, a partir do nó inicial do grafo de fluxo para *n*, passa através de d. Sob esta definição, cada nó domina a si mesmo e a entrada para um laço (como definido na Seção 9.4) domina todos os nós do laço.

Exemplo 10.6. Consideremos o grafo de fluxo da Fig. 10.13, com nó inicial 1. O nó inicial domina cada nó. O nó 2 domina somente a si mesmo, uma vez que o controle pode atingir qualquer outro nó ao longo de um percurso que comece por 1 → 3. O nó 3 domina todos os nós, menos os nós 1 e 2. O nó 4 domina todos os nós, menos 1, 2 e 3, já que todos os percursos a partir de 1 precisam começar 1 → 2 → 3 → 4 ou 1 → 3 → 4. Os nós 5 e 6 dominam somente a si mesmos, uma vez que o fluxo de controle pode evitar, alternativamente, a um ou a outro. Finalmente, 7 domina 7, 8, 9, 10; 8 domina 8, 9, 10; 9 e 10 dominam somente a si mesmos. □

Uma forma útil de apresentar um dominador é numa árvore, chamada de *árvore dos dominadores*, na qual o nó inicial é a raiz e cada nó *d* domina somente seus descendentes na árvore. Por exemplo, a Fig. 10.4 mostra a árvore de dominadores para o grafo de fluxo da Fig. 10.13.

A existência das árvores de dominadores provém de uma propriedade dos dominadores; cada nó *n* possui um único *dominador imediato m*, que é o último dominador de *n* sobre qualquer percurso a partir do nó inicial até *n*. Em termos da relação de dominação (*domina*), o dominador imediato *m* possui a propriedade de que se $d \neq n$ e *d domina n*, então *d domina m*.

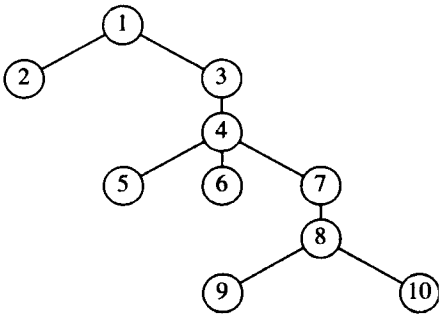


Fig. 10.14. Árvore de dominadores para o grafo de fluxo da Fig.10.13.

Laços Naturais

Uma importante aplicação das informações sobre os dominadores está na determinação dos laços adequados a melhoramentos num grafo de fluxo. Existem duas propriedades essenciais de tais laços.

1. Um laço precisa ter um único ponto de entrada, chamado de "cabeçalho". Esse ponto de entrada domina todos os nós do laço, ou não seria a única entrada para o laço.

2. Deve haver pelo menos uma forma de se iterar o laço, isto é, pelo menos um percurso de volta para o cabeçalho.

Uma boa forma de se encontrar todos os laços num grafo de fluxo é pesquisar pelos lados cujas cabeças dominem as caudas. (Se $a \rightarrow b$ é um lado, b é a *cabeça* e a é a *cauda*). Chamaremos tais lados de *lados refluentes.**

Exemplo 10.7. Na Fig. 10.13, existe um lado $7 \rightarrow 4$ e 4 *domina* 7. Similarmente, $10 \rightarrow 7$ é um lado e 7 *domina* 10. Os outros lados com esta propriedade são $4 \rightarrow 3$, $8 \rightarrow 3$ e $9 \rightarrow 1$. Note-se que esses são exatamente os lados dos quais poderíamos pensar como formadores de laços num grafo de fluxo. □

Dado um lado refluente $n \rightarrow d$, definimos um *laço natural* do mesmo como sendo d mais o conjunto de nós que podem atingir n sem passar através de d. O nó d é o cabeçalho do laço.

Exemplo 10.8. O laço natural do lado $10 \rightarrow 7$ consiste nos nós 7, 8 e 10, uma vez que 8 e 10 são todos aqueles nós que podem atingir 10 sem passar através de 7. O laço natural de $9 \rightarrow 1$ é todo o grafo de fluxo. (Não se esqueça do percurso $10 \rightarrow 7 \rightarrow 8 \rightarrow 9$!) □

Algoritmo 10.1 Construção do laço natural de um lado refluente.

Entrada. Um grafo de fluxo G e um lado refluente $n \rightarrow d$.

Saída. O conjunto *laço* consistindo em todos os nós do laço natural de $n \rightarrow d$.

Método. Começando pelo nó n, consideramos cada nó $m \neq d$ que sabemos que está em *laço*, de forma a assegurar que os predecessores de m sejam também colocados em *laço*. O algoritmo é dado na Fig. 10.15. Cada nó em *laço*, exceto d, é colocado uma vez em *pilha*, de forma que seus predecessores serão examinados. Note-se que, como d é colocado no laço inicialmente, nunca examinaremos seus predecessores e, por conseguinte, iremos encontrar aqueles nós que atingem n sem nunca passar através de d. □

Laços mais Internos

Se usarmos os laços naturais como "os laços", teremos, então, a útil propriedade de que, a menos que dois laços tenham o mesmo cabeçalho, os mesmos ou serão disjuntos ou um estará inteiramente contido *(aninhado dentro)* do outro. Por conseguinte, negligenciando por enquanto os laços com o mesmo cabeçalho, temos a noção de *laço mais interno:* aquele que não contém quaisquer outros laços.

Quando dois laços possuem o mesmo cabeçalho, como na Fig. 10.16, é difícil dizer qual é o mais interno. Por exemplo, se o teste ao fim de B_1 fosse

```
if a = 10 goto B2
```

*Do original em inglês: *back edges*. (N. do T.)

procedimento *inserir(m)*;
se m não está em *laço* **então início**
    laço := *laço* ∪ {m};
    empilhar m em *pilha*
fim;

/* segue o programa principal */

pilha := vazio;
laço := {d };
inserir(n);
enquanto *pilha* não estiver vazia **faça início**
    desempilhar m, o primeiro elemento de *pilha*
    para cada predecessor p de m **faça** *inserir* (p)
fim

Fig. 10.15. Algoritmo para construir o laço natural.

provavelmente o laço $\{B_0, B_1, B_3\}$ seria o laço mais interno. Entretanto, não poderíamos assegurar isso sem um exame detalhado do código. Talvez a seja quase sempre 10 e seja típico executar o laço $\{B_0, B_1, B_2\}$ muitas vezes, antes de desviar para B_3. Conseqüentemente, iremos assumir que, quando dois laços naturais possuírem o mesmo cabeçalho, mas nenhum dos dois estiver aninhado dentro do outro, ambos serão tratados como um único laço.

Pré-Cabeçalhos

Várias transformações requerem que movamos enunciados para "antes do cabeçalho". Começamos, por conseguinte, o tratamento de um laço L através da criação de um novo bloco, chamado de *pré-cabeçalho.* O pré-cabeçalho possui somente o cabeçalho como sucessor e todos os lados que entravam anteriormente no cabeçalho de L, vindos de fora de L, entram, agora no pré-cabeçalho. Os lados de dentro de L para o cabeçalho não são modificados. A disposição é mostrada na Fig. 10.17. Inicialmente, o pré-cabeçalho está vazio, mas as transformações em L podem colocar enunciados no mesmo.

Grafos de Fluxo Redutíveis

Os grafos de fluxo que ocorrem na prática recaem na classe dos grafos de fluxo redutíveis, definida abaixo. O uso exclusivo de enunciados estruturados de fluxo de controle, tais como *if-then-else, while-do, continue* e *break,* produz programas cujos grafos de fluxo são sempre redutíveis. Mesmo os programas escritos usando os enunciados de desvio *goto* por parte de programadores sem conhecimento prévio de projeto estruturado de programas são quase sempre redutíveis.

Uma variedade de definições para "grafo de fluxo redutível" tem sido proposta. A que adotamos aqui carrega uma das mais importantes

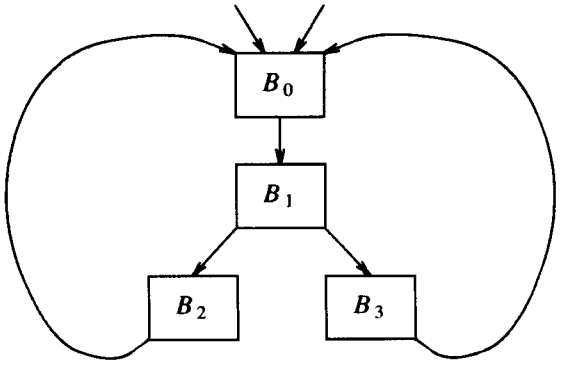


Fig. 10.16. Dois laços com o mesmo cabeçalho.

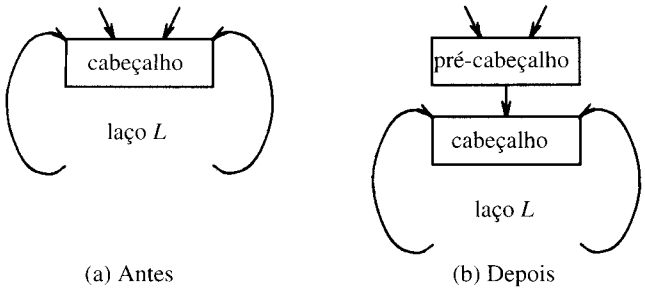


Fig. 10.17. Introdução do pré-cabeçalho.

propriedades dos grafos de fluxo redutíveis; nominalmente, não há desvios para o meio dos laços provenientes de fora dos mesmos; a única forma de se entrar num laço é através de seu cabeçalho. Os exercícios e notas bibliográficas contêm uma breve história do conceito.

Um grafo de fluxo G é *redutível* se e somente se pudermos particionar os lados em dois grupos disjuntos, freqüentemente chamados de lados *afluentes* e lados *refluentes*, com as duas seguintes propriedades:

1. Os lados afluentes formam um grafo acíclico, no qual cada nó pode ser atingido a partir do nó inicial de G.

2. Os lados refluentes consistem somente naqueles lados cujas cabeças dominam suas caudas.

Exemplo 10.9. O grafo de fluxo da Fig. 10.13 é redutível. Em geral, se conhecermos a relação de dominação (*domina*) para um grafo de fluxo, podemos encontrar e remover todos os lados refluentes. Os lados remanescentes têm que ser lados afluentes, se o grafo for redutível, e para verificar se o grafo de fluxo é redutível basta checar se os lados afluentes formam um grafo acíclico. No caso da Fig. 10.13, é fácil verificar que, se removermos os cinco lados refluentes $4 \rightarrow 3$, $7 \rightarrow 4$, $8 \rightarrow 3$, $9 \rightarrow 1$ e $10 \rightarrow 7$, cujas cabeças dominam as suas caudas, o grafo restante é acíclico. □

Exemplo 10.10. Consideremos o grafo de fluxo da Fig. 10.18, cujo nó inicial é 1. Este grafo de fluxo não possui lados refluentes, já que nenhuma cabeça de um lado domina a cauda do mesmo. Por conseguinte, o grafo somente poderia ser redutível se todo o grafo fosse acíclico. Mas, como não o é, o grafo de fluxo não é redutível. Intuitivamente, a razão pela qual este grafo de fluxo não é redutível é que o ciclo 2-3 pode ser iniciado de dois diferentes pontos, os nós 2 e 3. □

A propriedade-chave dos grafos de fluxo redutíveis para a análise de laços é que, em tais grafos de fluxo, cada conjunto de nós, que consideraríamos informalmente como um laço, precisa conter um lado refluente. De fato, precisamos examinar somente os laços naturais dos lados refluentes de forma a encontrar todos os laços de um programa cujo grafo de fluxo seja redutível. Contrastantemente, o grafo de fluxo da Fig. 10.18 parece ter um "laço" consistindo nos nós 2 e 3, mas não há um lado refluente do qual esse seja um laço natural. De fato, aquele "laço" possui dois cabeçalhos, 2 e 3, fazendo com que a aplicação de muitas das técnicas de otimização de código, tais como aquelas introduzidas na Seção 10.2 para a movimentação de código e remoção de variáveis de indução, não sejam diretamente aplicáveis.

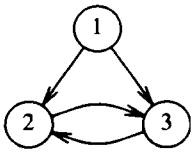


Fig. 10.18. Um grafo de fluxo irredutível.

Felizmente, as estruturas de fluxo de controle não redutíveis, tais como aquelas da Fig. 10.18, aparecem tão raramente na maioria dos programas que tornam o estudo de laços com mais de um cabeçalho de importância secundária. Existem, de fato, linguagens como Bliss e Modula 2, que permitem apenas programas com grafos de fluxo redutíveis, e muitas outras linguagens irão produzir somente grafos de fluxo redutíveis, na medida em que não usarmos o comando de desvio *goto*.

Exemplo 10.11. Retornando de novo à Fig. 10.13, notamos que o único "laço mais interno", isto é, o laço sem sublaços, é $\{7, 8, 10\}$, o laço natural do lado refluente $10 \rightarrow 7$. O conjunto $\{4, 5, 6, 7, 8, 10\}$ é o laço natural de $7 \rightarrow 4$. (Note-se que 8 e 10 podem atingir 7 através do lado $10 \rightarrow 7$.) Nossa intuição de que $\{4, 5, 6, 7\}$ forma um laço está errada, uma vez que 4 e 7 poderiam ser, ambos, entradas do exterior, violando nossa exigência de uma única entrada. Colocado de outra forma, não existe razão para assumir que o controle gaste muito tempo indo através dos nós $\{4, 5, 6, 7\}$; isto é tão plausível quanto que o controle passe de 8 para 7 mais freqüentemente do que o faz para 4. Pela inclusão de 8 e 10 no laço, estamos mais certos de termos isolado uma região pesadamente atravessada no programa.

É importante reconhecer, no entanto, o perigo de se fazer suposições a respeito da freqüência dos desvios. Por exemplo, se movêssemos para fora do laço $\{7, 8, 10\}$ um enunciado invariante que estivesse em 8 ou 10, e, de fato, o controle seguisse o lado $7 \rightarrow 4$ mais freqüentemente do que $7 \rightarrow 8$, poderíamos efetivamente aumentar o número de vezes que enunciado movido seria executado. Discutiremos os métodos para evitar esse problema na Seção 10.7.

O próximo laço maior é $\{3, 4, 5, 6, 7, 8, 10\}$, que é um laço natural dos lados $4 \rightarrow 3$ e $8 \rightarrow 3$. Como antes, nossa intuição de que $\{3, 4\}$ deveria ser considerado um laço viola a exigência de um único cabeçalho. O último laço, aquele para o lado refluente $9 \rightarrow 1$, é todo o grafo de fluxo. □

Existem várias propriedades úteis adicionais dos grafos redutíveis, as quais iremos introduzir quando discutirmos os tópicos da busca em profundidade e a análise de intervalos da Seção 10.9.

10.5 INTRODUÇÃO À ANÁLISE GLOBAL DE FLUXO DE DADOS

Com a finalidade de realizar a otimização de código e um trabalho de qualidade para a geração de código, um compilador precisa coletar informações a respeito do programa como um todo e distribuí-las a cada bloco no grafo de fluxo. Por exemplo, vimos na Seção 9.7 como o conhecimento de quais variáveis estavam vivas à saída de cada bloco poderia melhorar a utilização dos registradores. A Seção 10.2 sugeriu como poderíamos usar o conhecimento das subexpressões comuns globais com a finalidade de eliminar os cômputos redundantes. Igualmente, as Seções 9.9 e 10.3 discutiram como um compilador poderia tirar vantagem das "definições incidentes", tal como quando o conhecimento do último local de definição de uma variável, como `debug`, antes de se atingir um dado bloco, poderia ser usado com a finalidade de realizar transformações como a transposição para constantes e a eliminação de código morto. Esses fatos são simplesmente uns poucos exemplos das *informações de fluxo de dados* que um compilador otimizante captura através de um processo conhecido por *análise do fluxo de dados*.

As informações de fluxo de dados podem ser coletadas estabelecendo e solucionando-se sistemas de equações que relacionam informações em vários pontos do programa. Uma equação típica possui a forma

$$saída[S] = geradas[S] \cup (entrada[S] - mortas[S]) \quad (10.5)$$

e pode ser lida como "as informações ao final de um enunciado ou são geradas dentro do enunciado ou entram ao início e não são mortas à

medida que o controle flui através do enunciado". Tais equações são chamadas de *equações de fluxo de dados*.

Os detalhes sobre como as equações de fluxo de dados são estabelecidas e resolvidas dependem de três fatores.

1. As noções de "gerar" e "matar" dependem da informação desejada, isto é, do problema de análise de fluxo de dados a ser resolvido. Sobretudo, para alguns problemas, ao invés de prosseguir ao longo do fluxo de controle e definir *saída*[*S*] em termos de *entrada*[*S*], precisamos proceder de trás para frente e definir *entrada*[*S*] em termos de *saída*[*S*].

2. Como os dados fluem ao longo de percursos de controle, a análise de fluxo de dados é afetada pelas construções controle do programa. De fato, quando escrevemos *saída* [*S*], assumimos implicitamente que existe um único ponto final onde o controle deixa o enunciado; em geral, as equações são estabelecidas ao nível dos blocos básicos em vez de enunciados, porque os blocos possuem realmente pontos únicos de saída.

3. Existem sutilezas que acompanham enunciados como chamadas de procedimentos, atribuições através de variáveis do tipo apontador e mesmo atribuições a variáveis do tipo *array*.

Nesta seção, consideramos o problema de determinar o conjunto de definições que atingem um ponto num programa e seu uso para encontrar oportunidades para a transposição para constantes. Mais tarde, neste capítulo, os algoritmos para a movimentação de código e a eliminação das variáveis de indução também irão usar estas informações.

Consideramos inicialmente programas construídos usando os enunciados **if** e **do-while.** O fluxo de controle previsível nesses enunciados permite-nos concentrar nas idéias necessárias para estabelecer e resolver as equações de fluxo de dados. As atribuições nesta seção ou são enunciados de cópia ou são da forma `a := b + c`. Neste capítulo, freqüentemente usamos "+" como um operador típico. Tudo o que dissermos se aplica diretamente a outros operadores, incluindo aqueles com um operando ou com mais de dois operandos.

Pontos e Percursos

Dentro de um bloco básico, falamos do *ponto* entre dois enunciados adjacentes, bem como do ponto antes do primeiro enunciado e após o último. Por conseguinte, o bloco B_1, na Fig. 10.19, possui quatro pontos: um antes de qualquer uma das atribuições e um após cada um dos três enunciados.

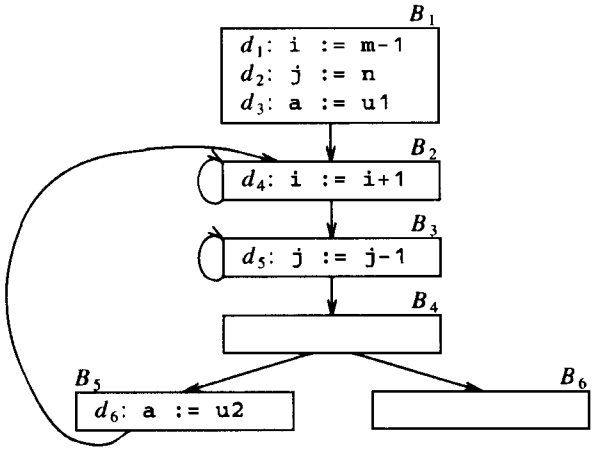


Fig. 10.19. Um grafo de fluxo.

Vamos agora ter uma visão global e considerar todos os pontos em todos os blocos. Um percurso de p_1 para p_n é uma seqüência de pontos $p_1, p_2, ..., p_n$, tais que, para cada i entre 1 e $n-1$, uma das duas situações abaixo é válida:

1. p_i é o ponto precedendo imediatamente a um enunciado e p_{i+1} é o ponto que se segue imediatamente àquele enunciado no mesmo bloco, ou
2. p_i é o final de algum bloco e p_{i+1} é o início de um bloco sucessor.

Exemplo 10.12. Na Fig. 10.19 existe um percurso a partir do bloco B_5 para o início do bloco B_6. O mesmo viaja através do ponto final de B_5 e, em seguida, através de todos os pontos em B_2, B_3 e B_4, antes de atingir o início de B_6. □

Definições Incidentes

Uma *definição* de uma variável `x` é um enunciado que atribui, ou pode atribuir, um valor a `x`. As formas mais comuns de definição são as atribuições a `x` e os enunciados que lêem um valor proveniente de um dispositivo de entrada e saída e o armazenam em `x`. Esses enunciados certamente definem um valor para `x`, e são referenciados como definições *inambíguas* de `x`. Existem outros tipos determinados de enunciados que podem definir um valor para `x`; são chamados de definições *ambíguas*. As formas mais usuais de definições ambíguas de `x` são:

1. Uma chamada de procedimento tendo `x` como parâmetro (que não um parâmetro por valor) ou um procedimento que possa ter acesso a `x` porque `x` está no escopo do procedimento. Temos também de considerar a possibilidade da polionomia*, onde `x` não está no escopo do procedimento, mas foi associado a outra variável que é transmitida como um parâmetro ou está no escopo. Esses temas são retomados na Seção 10.8.

2. Uma atribuição através de um apontador que poderia se referir a `x`. Por exemplo a atribuição `*q:=y` é uma definição de `x` se for possível que `q` aponte para `x`. Os métodos para se determinar para o que um apontador poderia apontar são também discutidos na Seção 10.8, mas na ausência de qualquer conhecimento do contrário, temos que assumir que uma atribuição através de um apontador é uma definição de cada variável.

Dizemos que uma definição *d incide* num ponto *p* (ou *atinge* um ponto *p*) se existir um percurso do ponto que se segue imediatamente a *d* até *p*, tal que *d* não seja "morta" ao longo do percurso. Intuitivamente, se uma definição *d* de alguma variável `a` incide em um ponto *p*, *d* poderia ser o local ao qual o valor de `a` usado em *p* teria sido definido. *Matamos* uma definição de uma variável `a` se, entre dois pontos ao longo do percurso, existir uma nova definição de `a`. Note-se que somente definições inambíguas de `a` matam outras definições de `a`. Por conseguinte, um ponto pode ser atingido por uma definição inambígua e por uma definição ambígua da mesma variável, que apareça mais tarde ao longo de um percurso.

Por exemplo, ambas as definições `i := m-1` e `j:=n` no bloco B_1 da Fig. 10.19 atingem o início do bloco B_2, assim como o faz a definição `j:=j-1`, providenciado que não haja atribuições ou leituras de `j` em B_4 ou B_5, ou na porção de B_3 que se segue àquela definição. No entanto, a atribuição a `j` em B_3 mata a definição `j:=n`, de forma que a última não atinge B_4, B_5 ou B_6.

*Significa a situação em que um elemento de programa apresenta nomes múltiplos. Cada um desses nomes, que não o de batismo, é um *pseudônimo* ou *apelido* do objeto de dados. Os diversos nomes serão considerados *sinônimos* uns dos outros. O termo original em inglês é **aliasing;** os *outros* nomes são denominados **alias** do nome principal. Pratt [1984] é uma boa referência para o tema. (N.do T.)

Definidas dessa forma, permitimos algumas imprecisões na conceituação das definições incidentes. No entanto, todas as imprecisões estão na direção "segura" ou "conservativa". Como exemplo, notemos que nossa suposição de que todos os lados de um grafo de fluxo possam ser percorridos. Isto pode não ser verdadeiro na prática. Por exemplo, para nenhum valor de `a` e de `b` pode o controle atingir efetivamente o enunciado `a:=4`, no seguinte fragmento de programa:

```
if a = b then a := 2
else if a = b then a := 4
```

Em geral, decidir se cada percurso no grafo de fluxo pode ser seguido é um problema indecidível, e não tentaremos resolvê-lo aqui.

Um tema recorrente no projeto de transformações para a melhoria de código é que devemos tomar apenas decisões conservativas ao nos defrontarmos com qualquer dúvida, apesar das estratégias conservativas nos fazerem perder algumas transformações que de fato poderíamos realizar seguramente. Uma decisão é *conservativa* se nunca leva a uma mudança do que o programa computa. Nas aplicações de definições incidentes, é normalmente conservativo assumir que uma definição atinja a um ponto, ainda que, de fato, não possa fazê-lo. Por conseguinte, permitimos os percursos que jamais venham a ser percorridos em qualquer execução do programa e permitimos que passem definições através de definições ambíguas da mesma variável.

Análise de Fluxo e Dados de Programas Estruturados

Os grafos de fluxo para as construções de fluxo de controle tais como os enunciados *do-while* possuem uma propriedade útil; existe um único ponto de início através do qual o controle entra e um único ponto de saída ao qual o controle deixa quando a execução do enunciado estiver terminada. Exploramos esta propriedade quando falamos das definições incidentes ao início e fim de enunciados, que têm a seguinte sintaxe:

$$S \rightarrow \textbf{id} := E \mid S\ ;\ S \mid \textbf{if}\ E\ \textbf{then}\ S\ \textbf{else}\ S \mid \textbf{do}\ S\ \textbf{while}\ E$$
$$E \rightarrow \textbf{id} + \textbf{id} \mid \textbf{id}$$

As expressões nesta linguagem são similares àquelas no código intermediário, mas os grafos de fluxo para os enunciados têm formas restritas que são sugeridas pelos diagramas na Fig. 10.20. Um propósito primário desta seção é o de estudar as equações de fluxo de dados sumarizadas na Fig. 10.21.

Definimos uma porção do grafo de fluxo chamada de uma *região* como sendo um conjunto de nós N que inclua um *cabeçalho*, que domina todos os outros nós da região. Todos os lados entre os nós em N estão na região, exceto (possivelmente) por alguns que entrem no cabeçalho.[4] A porção do grafo de fluxo correspondente a um enunciado S é a região que obedece à restrição adicional de que o controle pode fluir para exatamente um bloco externo quando deixa a região.

Como uma conveniência técnica, assumimos que existam blocos fictícios sem enunciados (indicados por círculos abertos na Fig. 10.20) através dos quais o controle flua justamente antes de entrar e deixar a região. Dizemos que os pontos de início dos blocos fictícios à entrada e saída de um enunciado da região são os pontos *inicial* e *final*, respectivamente, do enunciado.

As equações na Fig. 10.21 são definições indutivas, ou dirigidas pela sintaxe, dos conjuntos *entrada*[S], *saída*[S] e *geradas*[S] para todos os enunciados S. Os conjuntos *geradas*[S] e *mortas*[S] são atributos sintetizados; são computados de baixo para cima, dos menores enunciados para os maiores. Nosso desejo é que a definição d esteja em *geradas*[S], se d atingir o final de S, independentemente de atingir

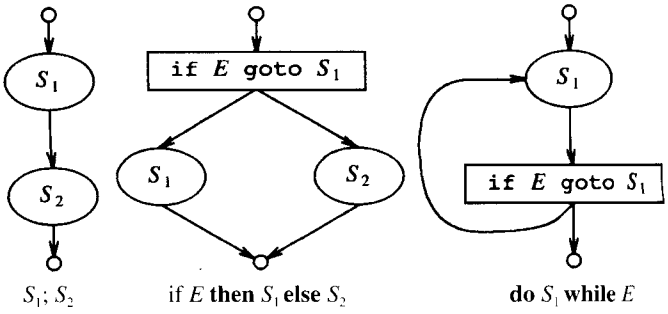


Fig. 10.20. Algumas construções de controle estruturado.

o início de S. Colocado de outra forma, d precisa aparecer em S e atingir o final de S através de um percurso que não vá para fora de S. Esta é a justificativa para dizer que *geradas*[S] é o conjunto de definições "geradas por S".

Similarmente, desejamos que *mortas*[S] seja o conjunto de definições que nunca atinjam o final de S, mesmo que atinjam o início. Conseqüentemente, faz sentido enxergar essas definições como "mortas por S". Para a definição d estar em *mortas*[S], cada percurso a partir do início até o final de S precisa ter uma definição inambígua da mesma variável definida por d e, se d aparecer em S, então, seguindo a cada ocorrência de d ao longo de qualquer percurso, precisa haver uma outra definição da mesma variável.[5]

As regras para *geradas* e *mortas*, sendo traduções sintetizadas, são relativamente fáceis de se compreender. Para começar, observemos as regras da Fig. 10.21(a) para uma única atribuição da variável `a`. Certamente, aquela atribuição é uma definição de `a`, digamos, definição d. Então d é a única definição certa de atingir o final do enunciado independentemente de atingir ou não o início. Por conseguinte,

$$geradas[S] = \{d\}$$

Por outro lado, d "mata" todas as outras definições de `a` e, dessa forma, escrevemos

$$mortas[S] = D_a - \{d\}$$

onde D_a é o conjunto de todas as definições no programa para a variável `a`.

A regra para enunciados em cascata, ilustrada na Fig. 10.21(b), é um tanto mais sutil. Sob que circunstâncias é uma definição de d gerada por $S = S_1\ ;\ S_2$? Antes de mais nada, se for gerada por S_2, então certamente é gerada por S. Se d é gerada por S_1, irá atingir o final de S, senão for morta por S_2. Por conseguinte, escrevemos

$$geradas[S] = geradas[S_2] \cup (geradas[S_1] - mortas[S_2])$$

Um raciocínio similar se aplica à morte de uma definição, e, então, temos

$$mortas[S] = mortas[S_2] \cup (mortas[S_1] - geradas[S_2])$$

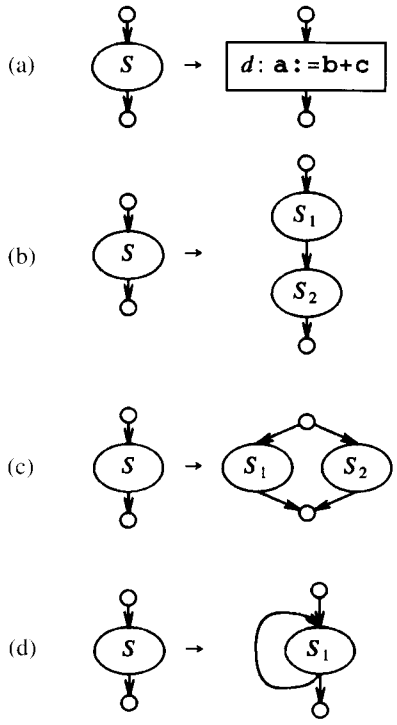
Para o enunciado *if*, ilustrado na Fig. 10.21 (c), notamos que, se um ou outro ramo do comando *if* gera uma definição, a mesma atinge o fim do enunciado S. Então,

$$geradas[S] = geradas[S_1] \cup geradas[S_2]$$

No entanto, a fim de "matar" uma definição d, a variável definida por d precisa ser morta ao longo de qualquer percurso partindo do início

[4]Um laço é um caso especial de uma região que é fortemente conectada e inclui todos os lados refluentes para o cabeçalho.

[5]Nesta seção introdutória estamos assumindo que todas as definições sejam inambíguas. A Seção 10.8 lida com as modificações necessárias para tratar as definições ambíguas.



$geradas[S] = \{d\}$
$mortas[S] = D_a - \{d\}$

$saída[S] = geradas[S] \cup (entrada[S] - mortas[S])$

$geradas[S] = geradas[S_2] \cup (geradas[S_1] - mortas[S_2]$
$mortas[S] = mortas[S_2] \cup (mortas[S_1] - geradas[S_2])$

$entrada[S_1] = entrada[S]$
$entrada[S_2] = saída[S_1]$
$saída[S] = saída[S_2]$

$geradas[S] = geradas[S_1] \cup geradas[S_2]$
$mortas[S] = mortas[S_1] \cap mortas[S_2]$

$entrada[S_1] = entrada[S]$
$entrada[S_2] = entrada[S]$
$saída[S] = saída[S_1] \cup saída[S_2]$

$geradas[S] = geradas[S_1]$
$mortas[S] = mortas[S_1]$

$entrada[S_1] = entrada[S] \cup geradas[S_1]$
$saída[S] = saída[S_1]$

Fig. 10.21. Equações de fluxo de dados para as definições incidentes.

para o final de S. Em particular, precisa ser morta ao longo de um e de outro ramos, e, dessa forma,

$$mortas[S] = mortas[S_1] \cap mortas[S_2]$$

Finalmente, consideremos as regras para os laços na Fig. 10.21(d). Colocado de forma simples, um laço não afeta *geradas* ou *mortas*. Se a definição d é gerada dentro de S, atinge tanto o fim de S_1 quanto o de S. Por outro lado, se d é gerada dentro de S, pode somente ser gerada dentro de S_1. Se d é morta por S_1, ir ao longo do laço não ajuda; a variável de d queda redefinida dentro de S_1 a cada volta. Por outro lado, se d é morta por S, terá de sê-lo seguramente por S_1. Concluímos que

$$geradas[S] = geradas[S_1]$$
$$mortas[S] = mortas[S_1]$$

Estimativa Conservativa das Informações de Fluxo de Dados

Existe um erro sutil de cálculo para *geradas* e *mortas* dadas na Fig. 10.21. Levantamos a suposição de que a expressão condicional E nos enunciados *if* e *do* eram "não interpretadas", isto é, que existiam entradas para o programa que faziam seus desvios irem para uma outra direção. Colocado de outra forma, assumimos que qualquer percurso grafo-teorético no diagrama de fluxo é também um *percurso de execução*, isto é, um percurso que é executado quando o programa é rodado com pelo menos uma possível entrada.

Este não é sempre o caso e, de fato, em geral não podemos decidir se uma ramificação será seguida. Suponhamos, por exemplo, que a expressão E num enunciado *if-then-else* fosse sempre verdadeira. Então, o percurso ao longo de S_2 na Fig. 10.21(c) poderia jamais ser seguido. Isto tem duas conseqüências. Primeira, uma definição gerada por S_2 não é realmente gerada por S, porque não há como se ir do início de S para o enunciado S_2. Segunda, nenhuma definição em $mortas[S_1]$ pode atingir o final de S. Por conseguinte, tal definição deveria logicamente estar em $mortas[S]$ mesmo que não esteja em $mortas[S_2]$.

Quando comparamos o conjunto *geradas* computado com o conjunto *geradas* "verdadeiro", descobrimos que o *geradas* "verdadeiro" é sempre um subconjunto do *geradas* computado. Por outro lado, o conjunto *mortas* verdadeiro é sempre um superconjunto do conjunto *mortas* computado. Essas relações de inclusão são válidas mesmo depois de considerarmos as outras regras da Fig. 10.21. Por exemplo, se a expressão E num enunciado *do-S-while-E* nunca for falsa, podemos jamais sair do laço. Por conseguinte, o *geradas* verdadeiro é $\emptyset$, e cada definição é morta pelo laço. O caso dos enunciados em cascata, na Fig. 10.21(b), onde precisa ser levada em conta a impossibilidade de se sair de S_1 ou S_2 por causa de um laço infinito, é deixada como um exercício.

É natural perguntar se essas diferenças entre os conjuntos *geradas* e *mortas*, verdadeiros e computados, apresentam um sério obstáculo à análise do fluxo de dados. A resposta reside no uso pretendido para esses dados. No caso particular das definições incidentes, usamos normalmente as definições para inferir que o valor de uma variável `x` a um ponto está limitado a algumas possibilidades dentre pequenos números. Por exemplo, se concluirmos que as únicas definições de `x` que atingem aquele ponto são da forma `x := 1`, podemos inferir que `x` possui o valor 1 àquele ponto. Por conseguinte, poderíamos decidir substituir as referências a `x` por referências a 1.

Como conseqüência, superestimar o conjunto de definições que atingem um determinado ponto não parece ser muito sério; meramente nos impede de realizar uma otimização que poderíamos legitimamente fazer. Por outro lado, subestimar o conjunto de definições é um erro fatal; poderia nos levar a realizar uma mudança no programa a qual modifica o que o programa computa. Por exemplo, podemos pensar que todas as definições incidentes de `x` conferem ao mesmo o valor 1 e, conseqüentemente, substituir `x` por 1; mas pode haver uma outra definição incidente, não detectada, que dá a `x` valor 2. Para o caso das definições incidentes, denominamos um conjunto de definições *seguro* ou *conservativo* se a estimativa for um superconjunto (não necessariamente próprio) do conjunto verdadeiro das definições incidentes. Chamamos a definição de *insegura* se não for necessariamente um superconjunto do conjunto verdadeiro.

Para cada problema de fluxo de dados precisamos examinar o efeito das estimativas inacuradas sobre os tipos de modificações de programa que as mesmas podem causar. Aceitamos geralmente as discrepâncias que sejam *seguras*, no sentido em que as mesmas possam proibir otimizações que poderiam ser legalmente feitas, mas não aceitamos as que sejam *inseguras*, no sentido em que possam causar "otimizações" que não preservem o comportamento do programa observado externamente. Em cada problema de fluxo de dados, ou um subconjunto ou um superconjunto da resposta verdadeira (mas não ambos) é usualmente seguro.

Retornando agora às implicações de segurança na estimativa de *geradas* e *mortas* para as definições incidentes, notemos que nossas discrepâncias, superconjuntos para *geradas* e subconjuntos para *mortas*, estão ambas na direção segura. Intuitivamente, aumentar *geradas* adiciona ao conjunto definições que podem atingir um ponto e não pode impedir que uma definição atinja um local que a mesma já atingia efetivamente. Igualmente, decrementar *mortas* pode somente aumentar o conjunto de definições incidentes a um dado ponto.

Cômputo de *Entrada* e *Saída*

Muitos problemas de fluxo de dados podem ser resolvidos por traduções sintetizadas similares àquelas usadas para se computar *geradas* e *mortas*. Por exemplo, podemos querer determinar, para cada enunciado S, o conjunto das variáveis que sejam definidas dentro de S. Essa informação pode ser computada por equações análogas àquelas para o conjunto *geradas*, sem mesmo serem requeridos conjuntos semelhantes a *mortas*. Essa informação pode ser usada para determinar os cômputos laço-invariantes.

No entanto, existem outros tipos de informações de fluxo de dados, tais como os do problema das definições incidentes, que usamos como exemplo, onde também necessitamos computar certos atributos herdados. Acontece que *entrada* é um atributo herdado e *saída* é um atributo sintetizado dependente de *entrada*. Pretendemos que *entrada*[S] seja o conjunto de definições que atinjam o início de S, levando em conta o fluxo de controle através de todo o programa, incluindo os enunciados fora de S ou dentro dos quais S esteja aninhado. O conjunto de *saída*[S] é definido similarmente para o final de S. É importante notar uma distinção entre *saída*[S] e *geradas*[S]. O último é o conjunto de definições que atingem o final de S sem seguir por percursos fora de S.

Como um exemplo simples da diferença, consideremos os enunciados em cascata na Fig. 10.21(b). Um enunciado d pode ser gerado em S_1 e conseqüentemente atingir o início de S_2. Se d não estiver em *mortas*[S_2] irá atingir o final de S_2 e, por conseguinte, estará em *saída* [S_2]. No entanto, d não está em *geradas*[S_2].

Após computar *geradas*[S] e *mortas* [S], de baixo para cima, para todos os enunciados S, devemos computar *entrada* e *saída*, começando pelo enunciado que representa o programa completo, compreendendo que *entrada*[S_0], = $\varnothing$, se S_0 for o programa completo. Isto é, nenhuma definição atinge o início do programa. Para cada um dos quatro tipos de enunciados da Fig. 10.21, podemos assumir que *entrada*[S] seja conhecido. Precisamos usá-lo para computar *entrada* para cada um dos subenunciados de S [o que é trivial nos casos (b) – (d) e irrelevante no caso (a)]. Em seguida, computamos recursivamente (de cima para baixo) *saída* para cada um dos subenunciados S_1 ou S_2 e usamos esses conjuntos para computar *saída*[S].

O caso mais simples é o da Fig. 10.21(a), onde o enunciado é uma atribuição. Assumindo que conheçamos *entrada*[S], computamos *saída* através da equação (10.5), isto é

$$saída[S] = geradas[S] \cup (entrada[S] - mortas[S])$$

Dito em palavras, uma definição atinge o final de S se for gerada por S (isto é, é a definição de d que é o enunciado), ou se atinge o início do enunciado e não for morta pelo mesmo.

Suponhamos ter computado *entrada*[S] e que S sejam dois enunciados em cascata, S_1 ; S_2, como no segundo caso da Fig. 10.21. Começamos observando *entrada*[S_1] = *entrada*[S]. Em seguida, computamos recursivamente saída[S_1], que fornece *entrada*[S_2], uma vez que uma definição atinge o início de S_2 se e somente se atingir o final de S_1. Podemos agora computar recursivamente *saída*[S_2], e esse conjunto é igual a *saída*[S].

Em seguida, consideremos o enunciado *if* da Fig. 10.21(c). Como tínhamos assumido conservativamente que o controle podia seguir uma ou outra ramificação, uma definição atinge o início de S_1 ou S_2 exatamente quando atinge o início de S. Isto é,

$$entrada[S_1] = entrada\ [S_2] = entrada[S]$$

Também segue a partir do diagrama na Fig. 10.21(c) que uma definição atinge o final de S se e somente se a mesma atingir o final de um dos subenunciados; isto é,

$$saída[S] = saída[S_1] \cup saída[S_2]$$

Podemos, por conseguinte, usar essas equações para computar *entrada*[S_1] e *entrada*[S_2] a partir de *entrada*[S], para computar recursivamente *saída*[S_1] e *saída*[S_2] e usá-los para computar *saída*[S].

Lidando com Laços

O último caso, a Fig. 10.21(d), apresenta problemas especiais. Vamos assumir, de novo, que nos sejam dados *geradas*[S_1] e *mortas*[S_1], tendo os mesmos sido computados de baixo para cima, e vamos assumir que nos seja dado *entrada*[S_1], na medida em que estejamos no processo de travessia em profundidade da árvore sintática. Diferentemente dos casos (b) e (c), não podemos simplesmente usar *entrada*[S] como em *entrada*[S_1], porque as definições dentro de S_1, que atingem o final de S_1, são capazes de seguir o arco de volta para o início de S_1 e, por conseguinte, também estão em *entrada*[S_1]. Em vez disso, temos

$$entrada[S_1] = entrada[S] \cup saída[S_1] \qquad (10.6)$$

Temos também a equação óbvia para *saída*[S]:

$$saída[S] = saída[S_1]$$

a qual podemos usar uma vez que tenhamos computado *saída*[S_1]. No entanto, parece que não podemos computar *entrada*[S_1], por (10.6), até que tenhamos computado *saída*[S_1] e o nosso plano geral foi computar *saída* para um enunciado computando primeiro *entrada* para o mesmo.

Felizmente, existe uma forma direta de se expressar *saída* em termos de *entrada;* é dada por (10.5), ou neste caso particular:

$$saída[S_1] = geradas[S_1] \cup (entrada[S_1] - mortas[S_1]) \qquad (10.7)$$

É importante compreender o que está acontecendo aqui. Não sabemos realmente que (10.7) seja verdadeira a respeito de um enunciado arbitrário S_1; apenas suspeitamos que deveria ser verdadeira porque "faz sentido" que uma definição deva atingir o final de um enunciado se e somente se tiver sido gerada dentro do mesmo ou se atingir o seu início e não for morta dentro do mesmo. No entanto, a única maneira de se saber como computar *saída* para um enunciado é através das equações fornecidas na Fig. 10.21(a) – (c). Vamos assumir (10.7) e derivar as equações para *entrada* e *saída* na Fig. 10.21(d). Podemos, em seguida, usar as equações da Fig. 10.21(a) – (d) para provar que (10.7) é válida para um S arbitrário. Poderíamos, em seguida, colocar essas provas juntas para realizar uma prova por indução no comprimento do enunciado S e de todos os seus subenunciados. Não iremos fazê-lo; deixamos as provas como exercício, mas o raciocínio que seguimos aqui deve se provar instrutivo.

Mesmo assumindo (10.6) e (10.7), ainda estamos no meio do mato. Essas duas equações definem uma recorrência para *entrada*[S_1] e *saída*[S_1] simultaneamente. Vamos escrever as equações como

$$\begin{aligned} I &= J \cup O \\ O &= G \cup (I - M) \end{aligned} \tag{10.8}$$

onde I, O, J, G e M correspondem a *entrada*[S_1], *saída*[S_1], *entrada*[S], *geradas*[S_1] e *mortas*[S_1], respectivamente. Os dois primeiros são variáveis; os outros três são constantes.

Para resolver (10.8), vamos assumir que $O = \varnothing$. Poderíamos então usar a primeira equação em (10.8) para computar uma estimativa de I, isto é,

$$I^1 = J$$

Em seguida, podemos usar a segunda equação para obter uma melhor estimativa de O:

$$O^1 = G \cup (I^1 - M) = G \cup (J - M)$$

A aplicação da primeira equação a esta nova estimativa de O nos fornece:

$$I^2 = J \cup O^1 = J \cup G \cup (J - M) = J \cup G$$

Se então reaplicarmos a segunda equação, a próxima estimativa de O será:

$$O^2 = G \cup (I^2 - M) = G \cup (J \cup G - M) = G \cup (J - M)$$

Note-se que $O^2 = O^1$. Por conseguinte, se computarmos a próxima estimativa de I, a mesma será igual a I^1, o que nos dará uma outra estimativa de O igual O^1 e assim por diante. Por conseguinte, os valores limitantes para I e O são aqueles fornecidos para I^1 e O^1 acima. Derivamos, então, as equações da Fig. 10.21(d), que são

$$\begin{aligned} \textit{entrada}[S_1] &= \textit{entrada}[S] \cup \textit{geradas}[S_1] \\ \textit{saída}[S] &= \textit{saída}[S_1] \end{aligned}$$

A primeira dessas equações é proveniente do cômputo acima; a segunda se segue do exame do grafo da Fig. 10.21(d).

Um detalhe que resta é: por que fomos autorizados a começar com a estimativa $O = \varnothing$. Relembremos que, em nossa discussão das estimativas conservativas, sugerimos que conjuntos como *saída*[S_1], em lugar dos quais O está, deveriam ser superestimados ao invés de subestimados. De fato, se fôssemos começar com $O = \{d\}$, onde d fosse uma definição que não aparecesse nem em J, G ou M, d acabaria desaparecendo dos valores limitantes tanto de I quanto de O.

Aqui, precisamos invocar os significados pretendidos para *entrada* e *saída*. Se uma tal definição d realmente pertencesse a *entrada*[S_1], deveria existir um percurso, a partir de qualquer ponto em que d estivesse definida até o início de S_1, que evidenciasse como d atingia aquele ponto. Se d estivesse fora de S, então d teria que estar em *entrada*[S] enquanto que, se d estivesse dentro de S (e por conseguinte dentro de S_1), teria que estar em *geradas*[S_1]. No primeiro caso, d estaria em J e, conseqüentemente, colocado em I por (10.8). No segundo caso, d estaria em G e, de novo, transmitido para I através de O em (10.8). A conclusão é que começando-se com uma estimativa muito pequena, construindo-se acima e juntando-se mais definições para I e O é uma forma segura de estimar *entrada*[S_1].

Representação dos Conjuntos

Definições de conjuntos, tais como *geradas*[S] e *mortas*[S], podem ser representadas compactamente usando-se vetores de *bits*. Atribuímos um número para cada definição de interesse no grafo de fluxo. O vetor de *bits* que representa um conjunto de definições terá 1 na posição i se e somente se uma definição com número i estiver no conjunto.

O número de um enunciado de definição pode ser tomado como o índice do enunciado num *array* que guarde os apontadores para os enunciados. No entanto, nem todas as definições podem ser de interesse durante a análise global de fluxo de dados. Por exemplo, definições de temporários que sejam usados somente dentro de um único bloco, como será a maioria dos temporários gerados para a avaliação de expressões, não precisam ter números atribuídos. Por conseguinte, os números de definições de interesse serão tipicamente registrados numa tabela separada.

A representação sob a forma de um vetor de *bits* para os conjuntos também permite que operações sobre conjuntos sejam implementadas eficientemente. A união e interseção de dois conjuntos podem ser implementadas pelo **ou** e **e** lógicos, respectivamente, operações básicas na maioria das linguagens orientadas para a programação de sistemas. A diferença $A - B$ dos conjuntos A e B pode ser implementada tomando-se o complemento de B e, em seguida, usando-se o **e** lógico para computar $A \wedge \neg B$.

Exemplo 10.13. A Fig. 10.22 mostra um programa com sete definições, indicadas por d_1 a d_7 nos comentários à esquerda das definições. Vetores de *bits* representando os conjuntos *geradas* e *mortas* para os enunciados da Fig. 10.22 são mostrados à esquerda dos nós da árvore sintática na Fig. 10.23. Os conjuntos em si foram computados através da aplicação das equações da Fig. 10.21 aos enunciados representados pelos nós da árvore sintática.

Consideremos o nó para d_7 ao canto à direita e ao fundo da Fig. 10.23. O conjunto *geradas*$\{d_7\}$ é representado por 000 0001 e o conjunto *mortas* $\{d_1, d_4\}$ por 100 1000. Isto é, d_7 mata todas as outras definições de i, sua variável.

Os segundo e terceiro filhos do nó **if** representam as partes do *then* e do *else*, respectivamente, do comando condicional. Notemos que o conjunto *geradas* 000 0011 ao nó **if** é a união dos conjuntos 000 0010 e 000 0001 aos segundo e terceiro filhos. O conjunto *mortas* é vazio porque os conjuntos de definições mortas pelas partes do *then* e do *else* são disjuntos.

As equações de fluxo de dados para enunciados em cascata são aplicadas ao pai do nó **if**. O conjunto *mortas* a esse nó é obtido por

$$000\,0000 \cup (110\,0001 - 000\,0011) = 110\,0000$$

Em palavras, nada é morto pelo enunciado condicional e d_7, morto pelo enunciado d_4, é gerado pelo enunciado condicional e, dessa forma, somente d_1 e d_2 estão no conjunto *mortas* do pai do nó **if**.

Podemos agora computar *entrada* e *saída* para o topo da árvore gramatical. Assumimos que o conjunto *entrada* à raiz da árvore sintática seja vazio. Conseqüentemente, *saída* para o filho à esquerda da raiz é o *geradas* desse nó, ou 111 0000. Este também é o valor do conjunto *entrada* ao nó **do**. A partir das equações de fluxo de dados associadas

```
/* d1 */    i := m-1;
/* d2 */    j := n;
/* d3 */    a := u1;
            do
/* d4 */        i := i+1;
/* d5 */        j := j-1;
                if e1 then
/* d6 */            a := u2
                else
/* d7 */            i := u3
            while e2
```

Fig. 10.22. Programa para ilustrar as definições incidentes.

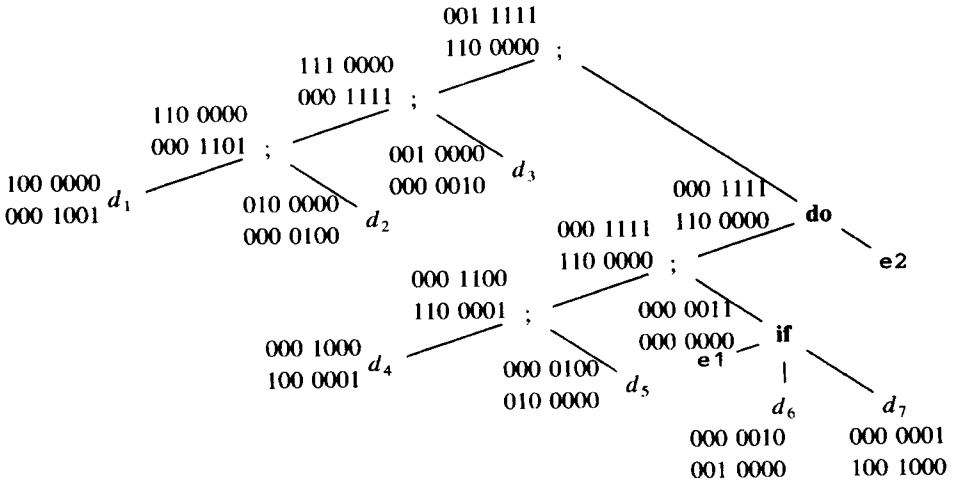


Fig. 10.23. Os conjuntos *geradas* e *mortas* nos nós de uma árvore sintática.

à produção **do** na Fig. 10.21, o conjunto *entrada* para o enunciado dentro do laço **do** é obtido tomando-se a união do conjunto 111 0000 ao nó **do** e do conjunto *geradas* 000 1111 ao próprio enunciado. A união produz 111 1111, e dessa forma todas as definições atingem o início do corpo do laço. No entanto, exatamente no ponto antes da definição d_5, o conjunto *entrada* é 011 1110, uma vez que a definição d_4 mata d_1 e d_7. O balanço dos cômputos de *entrada* e *saída* é deixado como um exercício. □

Definições Incidentes Locais

O espaço para as informações de fluxo de dados pode ser negociado em troca de tempo, salvando-se informações somente em determinados pontos e, na medida do necessário, recomputando-as em determinados pontos intervenientes. Os blocos básicos são usualmente tratados como uma unidade durante a análise de fluxo de dados global, com a atenção restrita somente àqueles pontos que sejam inícios de blocos. Como usualmente existem muito mais pontos do que blocos, restringir nossos esforços aos blocos se constitui numa economia significativa. Quando necessário, as definições incidentes para todos os pontos de um bloco podem ser computadas a partir das definições incidentes para o início do bloco.

Em mais detalhes, consideremos uma seqüência de atribuições S_1 ; S_2 ; ... ; S_n num bloco básico B. Referimo-nos ao início de B como o ponto p_0, ao ponto entre os enunciados S_i e S_{i+1} como p_i e ao final do bloco como o ponto p_n. As definições que atingem o ponto p_j podem ser obtidas a partir de *entrada*[B] considerando-se os enunciados S_1 ; S_2 ; ...; S_j, começando-se com S_1 e aplicando-se as equações de fluxo de dados da Fig. 10.21 para enunciados em cascata. Inicialmente, seja D = *entrada*[B]. Quando S_i é considerado, removemos de D as definições mortas por S_i e adicionamos as definições geradas por S_i. Ao final, D consiste nas definições incidentes em p_j.

Cadeias Uso-Definição

É freqüentemente conveniente armazenar as informações sobre as definições incidentes sob a forma de "cadeias uso-definição" ou *cadeias-ud*, que são listas de todas as definições que atingem aquele uso, para cada uso de uma variável. Se o uso de uma variável a num bloco B não é precedido por definições inambíguas de a, a cadeia-ud para aquele uso de a é o conjunto de definições em *entrada*[B] que sejam definições de a. Se existirem definições inambíguas de a, dentro de B, precedendo esse uso de a, então somente a última de tais definições de a estará na cadeia-ud e *entrada*[B] não é colocado na cadeia-ud. Adicionalmente, se existirem definições ambíguas de a, então, todas aquelas para as quais nenhuma definição inambígua de a recaia entre a mesma (isto é, a definição ambígua) e o uso de a estarão na cadeia-ud para esse uso de a.

Ordem de Avaliação

As técnicas para conservar espaço durante a avaliação de atributos, discutidas no Capítulo 5, também se aplicam ao cômputo das informações de fluxo de dados, usando-se especificações como aquela da Fig. 10.21. Especificamente, a única restrição sobre a ordem de avaliação dos conjuntos *geradas, mortas, entrada* e *saída* para os enunciados é aquela imposta pelas dependências entre os mesmos. Tendo escolhido uma ordem de avaliação, estamos livres para liberar o espaço para o conjunto, após todos os usos do mesmo terem ocorrido.

As equações de fluxo de dados desta seção diferem em um aspecto das regras semânticas para os atributos no Capítulo 5: as dependências circulares entre os atributos não foram permitidas no Capítulo 5, mas vimos que as equações de fluxo de dados podem tê-las, como, por exemplo, *entrada*[S_i] *e saída*[S_i], que dependem um do outro, em 10.8. Para o problema das definições incidentes, as equações de fluxo de dados podem ser reescritas de modo a eliminar a circularidade — comparem-se as equações não circulares da Fig. 10.21 com 10.8. Uma vez que uma especificação não circular seja obtida, as técnicas do Capítulo 5 podem ser aplicadas de forma a se obter soluções eficientes para as equações de fluxo de dados.

Fluxo de Controle Geral

A análise de fluxo de dados precisa levar em conta todos os percursos de controle. Se os percursos de controle forem evidentes a partir da sintaxe, as equações de fluxo de dados podem ser estabelecidas e resolvidas numa forma dirigida pela sintaxe, como nesta seção. Quando os programas podem conter enunciados *goto*, ou mesmo os enunciados mais disciplinados *continue* e *break*, o enfoque que adotamos precisará ser modificado a fim de levar em conta os percursos de controle efetivos.

Várias abordagens podem ser adotadas. O método iterativo da próxima seção funciona para grafos de fluxo arbitrários. Uma vez que os grafos de fluxo obtidos na presença de enunciados *break* e *continue* são redutíveis, tais construções podem ser tratadas sistematicamente usando métodos baseados em intervalos, que serão discutidos na Seção 10.10.

No entanto, a abordagem dirigida pela sintaxe não precisa ser abandonada quando os enunciados *break* e *continue* forem permitidos.

```
/* d1 */    i := m-1;
/* d2 */    j := n;
/* d3 */    a := u1;
            do
/* d4 */        i := i+1;
/* d5 */        j := j-1;
                if e3 then
/* d6 */            a := u2
                else begin
/* d7 */                i := u3;
                        break
                    end
            while e2
```

Fig. 10.24. Programa contendo enunciado *break*.

Antes de deixar esta seção, consideramos um exemplo que sugere como os enunciados *break* podem ser acomodados, deixando o desenvolvimento de idéias para a Seção 10.10.

Exemplo 10.14. O enunciado *break* dentro do laço **do-while** da Fig. 10.24 é equivalente a um desvio para o final do laço. Como devemos definir o conjunto *geradas* para o seguinte enunciado?

```
if e3 then a := u2
else begin i := u3; break end
```

Definimos o conjunto *geradas* como sendo $\{d_6\}$, onde d_6 é a definição `a:= u2`, porque d_6 é a única definição gerada ao longo dos percursos de controle, a partir do ponto de início até o de fim do enunciado. A definição d_7, isto é, `i:=u3`, será levada em conta quando todo o laço **do-while** for tratado.

Existe uma armadilha de programação que nos permite ignorar o desvio causado pelo enunciado *break*, enquanto estivermos processando os enunciados dentro do corpo de um laço: fazemos os conjuntos *geradas* e *mortas* para um enunciado *break* como sendo, respectivamente, o conjunto vazio e U, o conjunto universal de todas as definições, como mostrado na Fig. 10.25. Os conjuntos remanescentes *geradas* e *mortas*, mostrados na Fig. 10.25, são determinados usando-se as equações de fluxo de dados da Fig. 10.21, com o conjunto *geradas* mostrado acima do conjunto *mortas*. Os enunciados S_1 e S_2 representam seqüência de atribuições. Os conjuntos *geradas* e *mortas* ao nó *do* ficam para ser determinados.

O ponto final de qualquer seqüência de enunciados terminando num comando *break* não pode ser atingido, de forma que não há dano em se fazer o conjunto *geradas* para a seqüência ser $\emptyset$ (vazio) e o conjunto *mortas* ser U; o resultado ainda será uma estimativa conservativa de *entrada* e *saída*. Similarmente, o ponto final de um enunciado *if* pode somente ser atingido através da parte *then* e os conjuntos *geradas* e *mortas*, ao nó **if**, na Fig. 10.25, são, de fato, os mesmos que aqueles ao seu segundo filho.

Os conjuntos *geradas* e *mortas*, ao nó *do*, precisam levar em conta todos os percursos, a partir do início até o fim do enunciado *do*, de forma que são afetados pelo enunciado *break*. Vamos agora computar dois conjuntos G e M, inicialmente vazios, à medida que realizamos uma travessia da parte pontilhada, a partir do nó *do* até o *break*. A intuição é que G e M representem as definições geradas e mortas à medida que o controle flua para o enunciado *break*, a partir do início do corpo do laço. O conjunto *geradas* para o enunciado *do-while* pode então ser determinado realizando-se a união de G e do conjunto *geradas* para o corpo do laço, porque o controle pode atingir o final do *do* quer pelo enunciado *break* quer caindo através do corpo do laço. Pela mesma razão, o conjunto *mortas* para o **do** é determinado tomando-se a interseção de M e do conjunto *mortas* para o corpo do laço.

Exatamente antes de atingirmos o nó **if** temos $G = geradas[S_2] = \{d_4, d_5\}$ e $M = mortas[S_2] = \{d_1, d_2, d_7\}$. Ao ponto do nó **if**, estamos interessados no caso quando o controle flui para o enunciado *break* e, dessa forma, a parte *then* do comando condicional não possui efeito sobre G e M. O próximo nó ao longo do percurso pontilhado é uma seqüência de enunciados, e, dessa forma, computamos os novos valores de G e M. Escrevendo S_3 em lugar do enunciado representado pelo filho à esquerda do nó seqüência (aquele rotulado d_7), usamos

$$G := geradas[S_3] \cup (G - mortas[S_3])$$
$$M := mortas[S_3] \cup (M - \text{geradas}[S_3])$$

Por conseguinte, os valores de G e M, ao atingirem o enunciado *break*, são $\{d_5, d_7\}$ e $\{d_1, d_2, d_4\}$, respectivamente. □

10.6 SOLUÇÃO ITERATIVA PARA AS EQUAÇÕES DE FLUXO DE DADOS

O método da última seção é simples e eficiente quando aplicável, mas, para linguagens como Fortran ou Pascal, que permitem grafos de fluxo arbitrários, não é suficientemente geral. A Seção 10.10 discute a "análise de intervalos", uma forma de se obter as vantagens da abordagem dirigi-

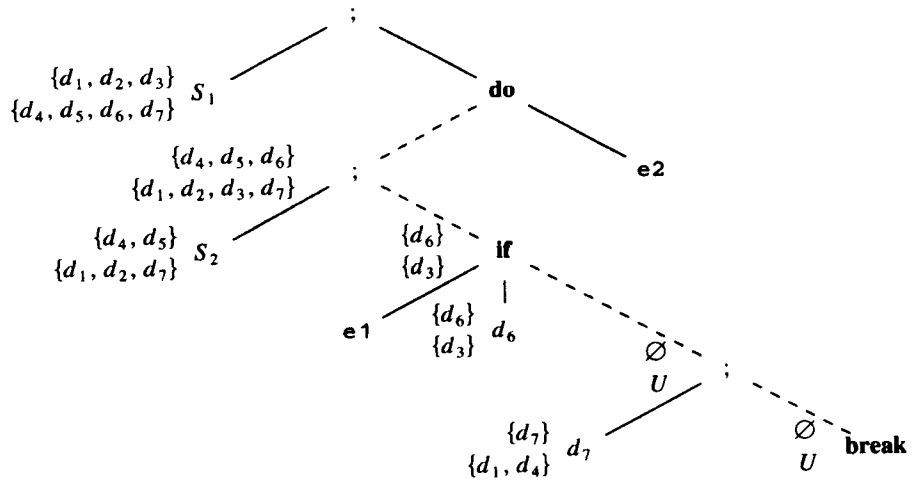


Fig. 10.25. Efeito de um enunciado *break* sobre os conjuntos *geradas* e *mortas*.

da pela sintaxe para a análise de fluxo de dados sobre grafos de fluxo gerais, a expensas de uma complexidade conceitual consideravelmente maior.

Aqui, iremos discutir uma outra importante abordagem à solução de problemas de fluxo de dados. Em vez de tentar usar a árvore gramatical para dirigir o cômputo dos conjuntos *entrada* e *saída*, construímos, primeiro, um grafo de fluxo e, em seguida, computamos *entrada* e *saída*, para cada nó simultaneamente. Ao discutirmos este novo método, iremos também aproveitar a oportunidade de apresentar ao leitor muitos diferentes problemas de análise de fluxo de dados, mostrar algumas de suas aplicações e apontar as diferenças entre os problemas.

As equações para muitos problemas de fluxo de dados são similares na forma em que as informações estão sendo "geradas" e "mortas". No entanto, existem duas formas principais nas quais as equações diferem em detalhes.

1. As equações da última seção para as definições incidentes são equações *para adiante*, no sentido em que os conjuntos *saída* são computados em termos de seus conjuntos *entrada*. Veremos também problemas que sejam *para trás*, à medida que os conjuntos *entrada* sejam computados em função dos conjuntos *saída*.

2. Quando há mais de um lado entrando num bloco B, as definições que atingem o início de B são a união das definições que partem ao longo de cada lado. Dizemos, por conseguinte, que a união é o *operador de confluência*. Em contraste, consideraremos problemas como as expressões globais disponíveis, onde a interseção é o operador de confluência, porque uma expressão somente está disponível ao início de B se estiver disponível ao final de cada predecessor de B. Na Seção 10.11 iremos ver outros exemplos de operadores de confluência.

Nesta seção, iremos ver exemplos tanto de equações para adiante como para trás, com a união e a interseção ocupando, em turnos, a posição de operador de confluência.

Algoritmo Iterativo para Definições Incidentes

Para cada bloco básico B, podemos definir *saída*[B], *geradas*[B] e *entrada*[B] como na última seção, notando que cada bloco B pode ser visto como um enunciado que esteja em cascata com um ou mais enunciados de atribuição. Assumindo que *geradas* e *mortas* tenham sido computados para cada bloco, podemos criar dois grupos de equações, mostrados em (10.9), abaixo, que relacionam *entrada* e *saída*. O primeiro grupo de equações segue da observação que *entrada*[B] é a união das definições que partem de todos os predecessores de B. O segundo grupo é o das equações que são casos especiais da lei geral (10.5), a qual afirmamos ser válida para todos os enunciados. Esses dois grupos são:

$$entrada[B] = \bigcup_{\substack{P \text{ é um} \\ \text{predecessor de } B}} saída[P]$$

$$saída[B] = geradas[B] \cup (entrada[B] - mortas[B]) \qquad (10.9)$$

Se um grafo de fluxo possui n blocos básicos, obtemos $2n$ equações a partir de (10.9). As $2n$ equações podem ser resolvidas tratando-as como recorrências do cômputo dos conjuntos *entrada* e *saída*, exatamente como foram resolvidas, na última seção, as equações de fluxo de dados (10.6) e (10.5), para os enunciados **do-while**. Lá, começamos com um conjunto de definições vazio como estimativa inicial para todos os conjuntos *saída*. Aqui, começaremos com conjuntos *entrada* vazios, uma vez que notamos de (10.9) que os conjuntos *entrada*, sendo a união dos conjuntos *saída*, estarão vazios se os conjuntos *saída* o estiverem. Conquanto estejamos habilitados a concordar que as equações (10.6) e (10.7) necessitavam somente de uma iteração, no caso das equações mais complexas não podemos limitar *a priori* o número de iterações.

Algoritmo 10.2 Definições incidentes.

Entrada. Um grafo de fluxo para o qual *mortas*[B] e *geradas*[B] tenham sido computados para cada bloco.

Saída. *Entrada*[B], e *saída*[B] para cada bloco B.

Método. Usamos uma abordagem iterativa, começando com a "estimativa" entrada[B] = $\emptyset$ para todos os conjuntos B e convergindo para os valores desejados de *entrada* e *saída*. Na medida em que precisamos iterar até que os conjuntos *entrada* (e, por conseguinte, os conjuntos *saída*) convirjam, usaremos a variável *mudou* para registrar, a cada passagem através dos blocos, se algum conjunto *entrada* foi modificado. O algoritmo é delineado na Fig. 10.26. □

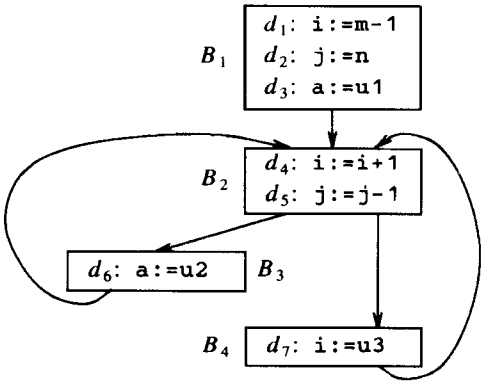
Intuitivamente, o Algoritmo 10.2 propaga as definições na medida em que as mesmas quedem mortas, num sentido em que simula todas as possíveis execuções do programa. As notas bibliográficas contêm referências onde podem ser encontradas provas formais da correção deste e de outros problemas de análise de fluxo de dados.

Podemos ver que o algoritmo irá eventualmente parar porque *saída*[B] nunca decresce em tamanho, para qualquer B; uma vez que uma definição seja adicionada, fica lá para sempre. (A prova deste fato é deixada como exercício.) Como o conjunto de todas as definições é finito, deverá haver eventualmente uma passagem pelo laço *while* na qual *saída-anterior* = *saída*[B], para cada B, à linha (9). Por conseguinte, *mudou* irá permanecer **false,** e o algoritmo terminará. Estamos certos em terminar porque se os conjuntos *saída* não mudarem, os conjuntos *entrada* não mudarão na próxima passagem. E, se os conjuntos *entra-*

```
        /* inicializar saída na suposição de que entrada[B] = ∅, para todos os B's */
(1)     para cada bloco B faça saída[B] := geradas[B];
(2)     mudou := true;          /* para fazer o laço while "pegar no tranco"*/
(3)     enquanto mudou faça início
(4)             mudou := false;
(5)             para cada bloco B faça início
(6)                     entrada[B] := ∪_{P é um predecessor de B} saída[P];
(7)                     saída_anterior := saída[B];
(8)                     saída[B] := geradas[B] ∪(entrada[B] − mortas[B]);
(9)                     se saída[B] ≠ saída_anterior então mudou: = true
                fim
        fim
```

Fig. 10.26. Algoritmo para computar *entrada* e *saída*.

*Os leitores hão de perdoar o tradutor por esta liberalidade, em decorrência da antinaturalidade dessa atribuição, com a única finalidade do laço ser executado a primeira vez. Este problema emerge em outros algoritmos, como, por exemplo, no *bubble sort*. (N. do T.)



$$\begin{aligned} \textit{geradas}\ [B_1] &= \{d_1, d_2, d_3\} \\ \textit{mortas}\ [B_1] &= \{d_4, d_5, d_6, d_7\} \\ \textit{geradas}\ [B_2] &= \{d_4, d_5\} \\ \textit{mortas}\ [B_2] &= \{d_1, d_2, d_7\} \\ \textit{geradas}\ [B_3] &= \{d_6\} \\ \textit{mortas}\ [B_3] &= \{d_3\} \\ \textit{geradas}\ [B_4] &= \{d_7\} \\ \textit{mortas}\ [B_4] &= \{d_1, d_4\} \end{aligned}$$

Fig. 10.27. Grafo de fluxo para ilustrar as definições incidentes.

da não mudarem, os conjuntos *saída* não poderão mudar, de forma que, em todas as passagens subseqüentes, não poderão haver mudanças.

Pode ser mostrado que um limite superior do número de vezes que o laço *while* irá rodar é o número de nós no grafo de fluxo. Intuitivamente, o motivo está em que, se uma definição atinge um determinado ponto, pode fazê-lo ao longo de um percurso livre de ciclos, e o número de nós de um grafo de fluxo é um limite superior do número de nós, num percurso livre de ciclos. Ao se passar pelo laço uma vez, a definição progride pelo menos em um nó, ao longo do percurso em questão.

De fato, ao ordenarmos apropriadamente os blocos à linha (5) do laço *for*, existem evidências empíricas de que o número médio de iterações num programa real está abaixo de 5 (ver a Seção 10.10). Como os conjuntos podem ser representados por vetores de *bits* e as operações sobre esses conjuntos podem ser implementadas através de operações lógicas sobre tais vetores, o Algoritmo 10.2 é surpreendentemente rápido na prática.

Exemplo 10.15. O grafo de fluxo da Fig. 10.27 foi derivado do programa 10.22 da última seção. Iremos aplicar o Algoritmo 10.2 a esse grafo de fluxo de forma que as abordagens das duas seções possam ser comparadas.

Somente as definições $d_1, d_2, ..., d_7$, definindo `i`, `j` e `a` na Fig. 10.27 são de interesse. Como na última seção, representaremos os conjuntos de definições por vetores de *bits*, onde o *bit i*, a partir da esquerda, representa a definição d_i.

O laço da linha (1), na Fig. 10.26, inicializa *saída*[B] = *geradas*[B], para cada B, e esses valores iniciais de *saída*[B] são mostrados na tabela da Fig. 10.28. Os valores iniciais $\varnothing$ para cada *entrada*[B] não são computados ou usados, mas são mostrados por uma questão de completeza. Suponhamos que o laço *for* da linha (5) seja executado com $B = B_1, B_2, B_3, B_4$, nessa ordem. Com $B = B_1$ não há predecessores para o nó inicial, e, dessa forma, *entrada*[B_1] permanece o conjunto vazio, representado por 000 0000; como resultado, *saída*[B_1] permanece igual a *geradas*[B_1]. Este valor não difere de *saída-anterior*, computado à linha (7), e, por conseguinte, não modificamos *mudou* para **true**.

Em seguida, consideramos $B = B_2$ e computamos

$$\begin{aligned} \textit{entrada}[B_2] &= \textit{saída}[B_1] \cup \textit{saída}[B_3] \cup \textit{saída}[B_4] \\ &= 111\ 0000 + 000\ 0010 + 000\ 0001 = 111\ 0011 \\ \textit{saída}[B_2] &= \textit{geradas}[B_2] \cup (\textit{entrada}[B_2] - \textit{mortas}[B_2]) \\ &= 000\ 1100 + (111\ 0011 - 110\ 0001) = 001\ 1110 \end{aligned}$$

Esse cômputo é sumarizado na Fig. 10.28. Ao final da primeira passagem, *saída*[B_4] = 001 0111, refletindo o fato de que d_7 é gerada e que d_3, d_5 e d_6 atingem B_4 e não são mortas em B_4. A partir da segunda passagem não existem mudanças em quaisquer dos conjuntos *saída*, e dessa forma, o algoritmo termina. □

Expressões Disponíveis

Uma expressão `x+y` está *disponível* em um ponto p se cada percurso (não necessariamente livre de ciclos), a partir do nó inicial até p, avalia `x+y` e, após a última de tais avaliações antes de atingir p, não existam atribuições subseqüentes a `x` ou a `y`. Para as expressões disponíveis, dizemos que um bloco *mata* a expressão `x+y` se atribuir (ou puder atribuir) a `x` ou a `y` e não recomputar subseqüentemente `x+y`. Um bloco *gera* a expressão `x+y` se, inquestionavelmente, avalia `x+y` e não redefine subseqüentemente nem `x` nem y.

Note-se que as noções de "matar" e "gerar" uma expressão disponível não são exatamente iguais à noção das definições incidentes. Contudo, essas novas noções de "matar" e "gerar" estão sujeitas às mesmas leis quando aplicadas às definições incidentes. Poderíamos computá-las exatamente como fizemos na Seção 10.5, desde que providenciássemos modificações nas regras em 10.21(a) para um enunciado de atribuição simples.

O uso primário das informações a respeito das expressões disponíveis está destinado à detecção das subexpressões comuns. Por exemplo, na Fig. 10.29, a expressão `4*i` no bloco B_3 será uma subexpressão comum se `4*i` estiver disponível no ponto de entrada do bloco B_3. Estará disponível se `i` não receber uma atribuição de novo va-

Bloco B	Valor Inicial		Passagem 1		Passagem 2	
	entrada [B]	*saída* [B]	*entrada* [B]	*saída* [B]	*entrada* [B]	*saída* [B]
B_1	000 0000	111 0000	000 0000	111 0000	000 0000	111 0000
B_2	000 0000	000 1100	111 0011	001 1110	111 1111	001 1110
B_3	000 0000	000 0010	001 1110	000 1110	001 1110	000 1110
B_4	000 0000	000 0001	001 1110	001 0111	001 1110	001 0111

Fig. 10.28. Cômputo de *entrada* e *saída*.

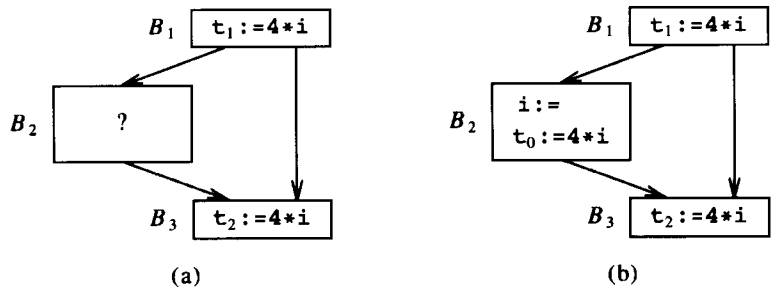


Fig. 10.29. Subexpressões comuns potenciais através de blocos.

lor no bloco B_2 ou se, como na Fig. 10.29, `4*i` for recomputado após `i` ter recebido uma atribuição em B_2.

Podemos facilmente computar o conjunto de expressões geradas para cada ponto num bloco, trabalhando a partir do início para o fim do bloco. No ponto antes do bloco, assumimos que não haja expressões disponíveis. Se a um ponto p, o conjunto A de expressões estiver disponível e q for um ponto após p com o enunciado `x:=y+z` entre ambos, formamos o conjunto de expressões disponíveis em q através dos dois seguintes passos.

1. Adicionamos a A a expressão `y+z`.

2. Removemos de A qualquer expressão envolvendo `x`.

Notemos que os dois passos precisam ser realizados na ordem correta, na medida em que `x` poderia ser o mesmo que `y` ou `z`. Após atingirmos o final do bloco, A é o conjunto de expressões geradas para o bloco. O conjunto de expressões mortas são todas as expressões, digamos, `y+z`, tais que `y` ou `z` sejam definidas no bloco e que `y+z` seja gerada pelo mesmo.

Exemplo 10.16. Consideremos os quatro enunciados da Fig. 10.30. Após o primeiro, `b+c` estará disponível. Após o segundo, `a-d` se torna disponível, porém `b+c` já não o está mais, porque `b` foi redefinida. O terceiro não torna `b+c` disponível de novo porque o valor de `c` é imediatamente modificado. Após o último enunciado, `a-d` já não está mais disponível, porque `d` foi modificada. Por conseguinte, nenhum enunciado é gerado e todos os enunciados envolvendo `a`, `b`, `c`, e `d` estão mortos. □

Podemos encontrar expressões disponíveis numa forma que lembra como as definições incidentes são computadas. Suponhamos que U seja o conjunto "universal" de todas as expressões que figurem no lado direito de um ou mais enunciados do programa. Para cada bloco B, seja *entrada*[B] o conjunto de expressões em U que estejam disponíveis exatamente antes do início de B. Seja *saída*[B] o mesmo para o ponto que se segue ao final de B. Definamos *e_geradas*[B] como sendo as expressões geradas por B e *e_mortas*[B] como sendo o conjunto de expressões em U mortas em B. Note-se que *entrada*, *saída*, *e_geradas* e *e_mortas* podem ser todos representados por vetores de *bits*. As seguintes equações relacionam os desconhecidos *entrada* e *saída*, um ao outro, e as quantidades conhecidas *e_geradas* e *e_mortas*.

ENUNCIADOS	EXPRESSÕES DISPONÍVEIS
.....	nenhuma
`a := b + c`	
.....	somente `b + c`
`b := a - d`	
.....	somente `a - b`
`c := b + c`	
.....	somente `a - d`
`d := a - d`	
.....	nenhuma

Fig. 10.30. Cômputo de expressões disponíveis

$$saída[B] = e_geradas[B] \cup (entrada[B] - e_mortas[B]$$

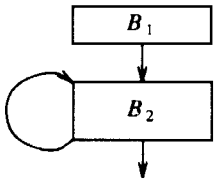
$$entrada[B] = \bigcap_{\substack{p \text{ é um predecessor} \\ \text{de } B}} saída[P] \text{ para } B \text{ não inicial}$$

$$entrada[B_1] = \varnothing \text{ onde } B_1 \text{ é o bloco inicial} \qquad (10.10)$$

As equações (10.10) parecem quase idênticas às equações (10.9) para as definições incidentes. A primeira diferença está em que *entrada* para o nó inicial é tratada como um caso especial. Isto é justificado sobre a base de que nada está disponível se o programa acabou de começar ao nó inicial, mesmo que alguma expressão pudesse estar disponível ao longo de todos os percursos até o nó inicial, de qualquer parte do programa. Senão forçarmos *entrada*[B_1] a ser vazio, poderíamos erroneamente deduzir que certas expressões estivessem disponíveis antes do programa começar. A segunda e mais importante diferença é que o operador de confluência é a interseção ao invés da união. Este operador é o adequado porque uma expressão está disponível ao início de um bloco somente se estiver disponível ao final de todos os seus predecessores. Em contraste, uma definição atinge o início de um bloco sempre que atingir o final de um ou mais de seus predecessores.

O uso de $\cap$ em lugar de $\cup$ faz as equações (10.10) se comportarem diferentemente daquelas de (10.9). Conquanto nenhum dos dois conjuntos possua uma solução única, para (10.9) é a menor solução que corresponde à definição de "incidente", e obtivemos aquela solução começando pela suposição de que nada atingia lugar algum, e a construímos. Naquela forma, nunca assumimos que uma definição d pudesse atingir um ponto p, a menos que o percurso efetivo, propagando d até p, pudesse ser encontrado. Em contraste, para as equações (10.10), desejamos a maior solução possível, e dessa forma começamos com uma aproximação que seja muito grande e trabalhamos reduzindo-a.

Pode não ser óbvio que, começando-se pela suposição de que "tudo, isto é, o conjunto U, esteja disponível em qualquer local" e eliminando-se somente aquelas expressões para as quais possamos descobrir um percurso ao longo do qual não estejam disponíveis, possamos atingir um conjunto de expressões verdadeiramente disponíveis. No caso das expressões disponíveis, é conservativo produzir um subconjunto do conjunto exato de expressões disponíveis e é isso o que iremos fazer. O argumento para que subconjuntos sejam conservativos é que o nosso uso pretendido para as informações é o de substituir o cômputo de uma expressão disponível por um valor previamente computado (ver o Algoritmo 10.5 na próxima seção) e não saber que uma expressão está disponível só e simplesmente nos inibe de modificar o código.



$$O^{j+1} = G \cup (I^j - M)$$
$$I^{j+1} = saída\ [B_1] \cap O^{j+1}$$

$I^0 = \varnothing$	$I^0 = U$
$O^1 = G$	$O^1 = U - M$
$I^1 = saída\ [B_1] \cap G$	$I^1 = saída\ [B_1] - M$
$O^2 = G$	$O^2 = G \cup (saída\ [B_1] - M)$

Fig. 10.31. Inicializar os conjuntos *entrada* com $\varnothing$ é por demais restritivo.

Exemplo 10.17. Iremos nos concentrar num único bloco, B_2 na Fig. 10.31, para ilustrar o efeito da aproximação inicial de *entrada*$[B_2]$ em *saída*$[B_2]$. Sejam G e M abreviaturas para *geradas*$[B_2]$ e *mortas*$[B_2]$, respectivamente. As equações de fluxo de dados para o bloco B_2 são:

$$entrada[B_2] = saída\ [B_1] \cap saída[B_2]$$
$$saída[B_2] = G \cup (entrada[B_2] - M)$$

Essas equações foram reescritas como fórmulas de recorrência na Fig. 10.31, com I^j e O^j sendo as j-ésimas aproximações de *entrada*$[B_2]$ e *saída*$[B_2]$, respectivamente. A figura também mostra que, começando com $I^0 = \varnothing$ obtemos $O^1 = O^2 = G$, enquanto que, começando com $I^0 = U$, obtemos um conjunto maior para O^2. Também acontece que *saída*$[B_2]$ se iguala a O^2 em cada caso, porque as iterações convergem, cada uma, para os pontos mostrados.

Intuitivamente, a solução obtida começando-se com $I^0 = U$ usando

$$saída[B_2] = G \cup (saída[B_1] - M)$$

é mais desejável, porque reflete corretamente o fato de que as expressões em *saída*$[B_1]$ que não são mortas por B_2 estão disponíveis ao final de B_2, exatamente como as expressões geradas por B_2 o estão. □

Algoritmo 10.3 Expressões disponíveis.

Entrada. Um grafo de fluxo G com *e_mortas*$[B]$ e *e_geradas*$[B]$ computados para cada bloco B. O bloco inicial é B_1.

Saída. O conjunto *entrada*$[B]$ para cada bloco B.

Método. Executar o algoritmo da Fig. 10.32. A explicação dos passos é similar àquela para a Fig. 10.26. □

Análise das Variáveis Vivas

Um número de transformações que melhoram a qualidade do código depende de informações computadas na direção oposta ao fluxo de controle do programa; iremos considerar algumas delas agora. Na análise das *variáveis vivas* desejamos saber, para a variável x e ponto p, se o valor de x em p poderia ser usado ao longo de algum percurso no grafo de fluxo, começando, por p. Se puder, dizemos que x está *viva* em p; caso contrário, x estará *morta* em p.

Como vimos na Seção 9.7, um importante uso para as informações a respeito das variáveis vivas surge quando geramos o código objeto. Após um valor ser computado num registrador e presumivelmente usado dentro de um bloco, não é necessário armazenar aquele valor se o mesmo estiver morto ao final do bloco. Igualmente, se todos os registradores estiverem ocupados e precisarmos de um outro registrador, deveríamos favorecer o uso de um registrador com um valor morto, uma vez que aquele valor não tem que ser armazenado.

Vamos definir *entrada*$[B]$ como sendo o conjunto de variáveis vivas no ponto imediatamente antes do bloco B e definir *saída*$[B]$ como sendo o mesmo no ponto imediatamente após o bloco. Seja *definidas*$[B]$ o conjunto de variáveis que tenham valores definitivamente atribuídos em B antes de qualquer uso dessas variáveis em B e seja *usos*$[B]$ o conjunto de variáveis cujos valores possam ser usados em B antes de quaisquer definições dessas variáveis. Por conseguinte, as equações que relacionam *definidas* e *usos* aos conjuntos desconhecidos *entrada* e *saída* são

$$entrada[B] = usos[B] \cup (saída[B] - definidas[B])$$
$$saída[B] = \bigcup_{S \text{ é um sucessor de } B} entrada[S] \qquad (10.11)$$

O primeiro grupo de equações diz que uma variável está viva chegando a um bloco se for usada antes da redefinição dentro do bloco, ou se estiver saindo do bloco e não for redefinida dentro do mesmo. O segundo grupo de equações diz que uma variável está viva à saída de um bloco se e somente se estiver viva indo para um de seus sucessores.

A relação entre (10.11) e as equações das definições incidentes (10.9) deveria ser notada. Aqui, *entrada* e *saída* possuem seus papéis intercambiados e *usos* e *definidas* substituem *geradas* e *mortas*, respectivamente. Como para (10.9), a solução para (10.11) não é necessariamente única e desejamos a menor solução possível. O algoritmo usado para a solução mínima é essencialmente uma versão de trás para a frente do Algoritmo 10.2. Como o mecanismo para detectar mudanças em qualquer um dos conjuntos *entrada* é similar à forma com que

```
entrada[B1] := ∅;
saída[B1] := geradas_e[B1]; /* entrada e saída nunca mudam para o nó inicial, B1 */
para B ≠ B1 faça saída[B] := U− e_mortas[B]; /* a estimativa inicial é muito grande */
mudou := true;
enquanto mudou faça início
    mudou := false;
    para B ≠ B1 faça início
        entrada[B] := ∪ (P é um predecessor de B) saída[P];

        saída-anterior := saída[B];
        saída[B] := e_geradas[B] ∪ (entrada[B] − e_mortas[B]);
        se saída[B] ≠ saída-anterior então mudou := true
    end
end
```

Fig. 10.32. Cômputo das expressões disponíveis.

detectamos as mudanças nos conjuntos *saída* nos Algoritmos 10.2 e 10.3, omitimos os detalhes para verificar a terminação do algoritmo.

Algoritmo 10.4 Análise das variáveis vivas.

Entrada. Um grafo de fluxo com os conjuntos *definidas* e *usos* computados para cada bloco.

Saída. *Saída*[*B*], o conjunto de variáveis vivas à saída de cada bloco *B* do grafo de fluxo.

Método. Executar o programa na Fig. 10.33. □

Cadeias Definição-Uso

Um cômputo feito virtualmente da mesma maneira que a análise das variáveis vivas é o *encadeamento definição-uso* (encadeamento-du). Dizemos que uma variável é usada no enunciado *s* se seu valor-*r* puder ser requerido. Por exemplo, `b` e `c` (mas não `a`) são usados em cada um dos enunciados `a:=b+c` e `a[b]:= c`. O problema do encadeamento-du é o de computar para um ponto *p* o conjunto de usos *s* de uma variável, digamos `x`, tal que exista um percurso de *p* até *s* que não redefina `x`.

Da mesma forma que com as variáveis vivas, se pudermos computar *saída*[*B*], o conjunto de usos atingíveis a partir do final do bloco *B*, poderemos então computar as definições atingidas a partir de qualquer ponto *p* dentro do bloco *B*, esquadrinhando a porção do bloco *B* que se segue a *p*. Em particular, se existir uma definição de uma variável `x` no bloco, podemos determinar a *cadeia-du* para aquela definição, a lista de todos os possíveis usos daquela definição. O método é análogo àquele discutido na Seção 10.5 para computar as cadeias-ud e o deixamos por conta do leitor.

As equações para computar as informações do encadeamento-du se parecem exatamente com (10.11), com a substituição de *definidas* e *usos*. Em lugar de *usos*[*B*], tomar o conjunto de usos *expostos acima* de *B*, isto é, o conjunto de pares (*s*, `x`) tais que *s* seja um enunciado em *B* que use a variável `x` e que nenhuma definição prévia de `x` ocorra em *B*. Em lugar de *definidos* [*B*], considere-se o conjunto de pares (*s*, `x`), tais que *s* seja um enunciado que use `x`, *s* não esteja em *B* e *B* tenha uma definição de `x`. Essas equações são resolvidas pelo análogo óbvio do Algoritmo 10.4 e não iremos alongar a discussão sobre o assunto.

10.7 TRANSFORMAÇÕES DE MELHORIA DE CÓDIGO

Os algoritmos para realizar as transformações de melhoria de código introduzidos na Seção 10.2 repousam nas informações da análise de fluxo de dados. Nas duas últimas seções, vimos como essas informações podiam ser coletadas. Aqui consideramos a eliminação das subexpressões comuns, a propagação de cópias e as transformações para a movimentação de computações laço-invariantes para fora dos laços e a eliminação das variáveis de indução. Para muitas linguagens, melhorias significativas no tempo de execução podem ser atingidas melhorando-se o código dos laços. Quando tais transformações são implementadas num compilador, é possível realizar algumas das transformações conjuntamente. No entanto, iremos apresentar as idéias subjacentes às transformações individualmente.

A ênfase desta seção está nas transformações globais que usam informações a respeito do programa como um todo. Como vimos nas duas últimas seções, a análise global do fluxo de dados não enxerga usualmente pontos dentro dos blocos básicos. As transformações globais não são, por conseguinte, um substitutivo para as transformações locais; ambas precisam ser realizadas. Por exemplo, ao realizarmos a eliminação global de subexpressões comuns, estaremos somente preocupados com o fato de uma expressão ser gerada por um bloco e não se a mesma é recomputada diversas vezes dentro do mesmo.

para cada bloco *B* **faça** *entrada*[*B*] := ∅;
enquanto ocorrerem mudanças a qualquer um dos conjuntos
    entrada **faça**
    para cada bloco *B* **faça início**

$$saída[B] = \bigcup_{\substack{S \text{ é um} \\ \text{sucessor de } B}} entrada[S]$$

$$entrada[B] := usadas[B] \cup (saída[B] - definidas[B])$$

    fim

Fig. 10.33. Cômputo das variáveis vivas.

Eliminação Global das Subexpressões Comuns

O problema de fluxo de dados das expressões disponíveis, discutido na última seção, nos permite determinar se uma expressão em um ponto *p* num grafo de fluxo é uma subexpressão comum. O algoritmo seguinte formaliza as idéias intuitivas apresentadas na Seção 10.2 para eliminar as subexpressões comuns.

Algoritmo 10.5 Eliminação global das subexpressões comuns.

Entrada. Um grafo de fluxo com informações a respeito das expressões disponíveis.

Saída. Um grafo de fluxo revisado.

Método. Para cada enunciado *s* da forma `x:=y+z`[6] tal que `y+z` esteja disponível ao início do bloco de *s* e nem `y` nem `z` sejam definidos antes do enunciado *s* naquele bloco, fazer o seguinte.

1. Para descobrir as avaliações de `y+z` que atingem o bloco de *s*, seguimos os lados do grafo de fluxo, procurando de frente para trás a partir do bloco de *s*. No entanto, não seguimos através de qualquer bloco que avalie `y+z`. A última avaliação de `y+z` em cada bloco encontrado é uma avaliação de `y+z` que atinge *s*.

2. Criar um nova variável `u`.

3. Substituir cada enunciado `w:=y+z` encontrado em (1) por

```
u := y + z
w := u
```

4. Substituir o enunciado *s* por `x:=u`. □

Alguns comentários sobre este algoritmo estão em pauta.

1. A busca no passo (1) do algoritmo, por avaliações de `y+z` que atinjam o enunciado *s*, pode também ser formulada como um problema de análise de fluxo de dados. No entanto, não faz sentido resolvê-lo para todas as expressões `y+z` e todos os enunciados ou blocos, porque muita informação irrelevante é coletada. Antes, devemos realizar uma busca sobre o grafo de fluxo para cada enunciado e expressão relevantes.

2. Nem todas as mudanças realizadas pelo Algoritmo 10.5 são melhorias. Poderíamos provavelmente desejar limitar em 1 o número das diferentes avaliações encontradas no passo (1), que atinjam *s*. No entanto, a propagação de cópias, a ser discutida em seguida, permite freqüentemente que o benefício seja obtido mesmo quando várias avaliações de `y+z` atinjam *s*.

[6] Lembremos que continuamos a usar + como um operador genérico.

3. O Algoritmo 10.5 irá omitir o fato de que `a*z` e `c*z` possuem o mesmo valor em

```
a := x+y          c := x+y
b := a*z    VS.   d := c*z
```

porque esse enfoque simples para as subexpressões comuns considera somente as expressões literais em si, ao invés dos valores computados pelas expressões. Kildall [1973] apresenta um método para capturar tais equivalências em uma passagem; discutiremos essas idéias na Seção 10.11. No entanto, essas equivalências podem ser capturadas através de múltiplas passagens do Algoritmo 10.5 e deve-se considerar repetilo até que não ocorram mais mudanças. Se `a` e `c` são variáveis temporárias que não sejam usadas fora do bloco no qual figuram, a subexpressão comum `(x+y)*z` pode ser capturada tratando-se as variáveis de forma especial, como no próximo exemplo.

Exemplo 10.18. Vamos supor que não existam atribuições ao *array* `a` no grafo de fluxo da Fig. 10.34(a), de forma a que possamos dizer seguramente que $a[t_2]$ e $a[t_6]$ sejam subexpressões comuns. O problema é eliminar essa subexpressão comum.

A subexpressão comum `4*i` na Fig. 10.34(a) foi eliminada na Fig. 10.34(b). Uma forma de determinar que $a[t_2]$ e $a[t_6]$ são também subexpressões comuns é substituir t_2 e t_6 por `u`, usando a propagação de cópias (a ser discutida em seguida); ambas as expressões se tornam `a[u]`, que pode ser eliminada reaplicando-se o Algoritmo 10.5. Notemos que a mesma nova variável `u` é inserida em ambos os blocos da Fig. 10.34(b), de forma que a propagação local de cópias é suficiente para converter ambas, $a[t_2]$ e $a[t_6]$, em `a[u]`.

Existe uma outra forma, que leva em conta o fato das variáveis temporárias serem inseridas pelo compilador e usadas somente dentro dos blocos em que aparecem. Iremos examinar mais detidamente a forma através da qual as expressões são representadas durante o cômputo das expressões disponíveis para nos acercarmos do fato que variáveis temporárias diferentes podem representar a mesma expressão. A técnica recomendada para se representar conjuntos de expressões é a de atribuir um número a cada expressão e usar um vetor de *bits*, com o *bit i* representando a expressão de número *i*. As técnicas dos números de valor da Seção 5.2 podem ser aplicadas durante a numeração de expressões, de forma a tratar as variáveis temporárias de forma especial.

Mais detalhadamente, suponhamos que `4*i` tenha o número de valor 15. As expressões $a[t_2]$ e $a[t_6]$ irão obter o mesmo número de valor se usarmos o número de valor 15 ao invés dos temporários t_2 e t_6. Suponhamos que o número de valor resultante seja 18. Por conseguinte, o *bit* 18 irá representar tanto $a[t_2]$ quanto $a[t_6]$ durante a análise de fluxo de dados e podemos determinar que $a[t_6]$ esteja disponível e possa ser eliminada. O código resultante é indicado na Fig. 10.34(c). Usamos (15) e (18) para representar as variáveis temporárias correspondentes às expressões com seus números de valor. Efetivamente, t_6 é inútil e deveria ser eliminada durante a análise local de variáveis vivas; igualmente, t_7, sendo uma variável temporária, não deveria ser computada em si; em lugar, os usos de t_7 deveriam ser substituídos por usos de (18). □

Propagação de Cópias

O Algoritmo 10.5 recém-apresentado e vários outros algoritmos tais como a eliminação de variáveis de indução, discutidos mais tarde nesta seção, introduzem enunciados de cópia da forma `x:=y`. As cópias também podem ser geradas diretamente pelo gerador de código intermediário, apesar da maioria delas envolver variáveis temporárias, locais a um bloco, e poderem ser removidas pela construção de GDAs discutida na Seção 9.8. É possível algumas vezes eliminar o enunciado *s* de cópia `x:=y` se determinarmos todos os locais onde essa definição de `x` é usada. Podemos, então, substituir `x` por `y` e todos esses locais, uma vez providenciado que as seguintes condições sejam atendidas para cada tal uso *u* de `x`.

1. O enunciado *s* precisa ser a única definição de `x` que atinja *u* (isto é, a cadeia-ud para o uso *u* consiste somente de *s*).

2. A cada percurso de *s* para *u*, incluindo os percursos que vão através de *u* várias vezes (mas não passam por *s* uma segunda vez), não existam atribuições a `y`.

A condição (1) pode ser verificada usando-se as informações do encadeamento-ud, mas, e sobre a condição (2)? Estabeleceremos um novo problema de análise de fluxo de dados no qual *entrada*[*B*] é o conjunto de cópias *s*: `x:=y` tais que cada percurso a partir do nó inicial até o início de *B* contém o enunciado *s* e subseqüentemente à última ocorrência de *s* não existam atribuições a `y`. O conjunto *saída*[*B*] pode ser correspondentemente definido, com respeito, porém, ao final de *B*. Dizemos que o enunciado de cópia *s*: `x:=y` é *gerado* no bloco *B* se *s* ocorrer em *B* e não existir atribuição subseqüente a `y` dentro de *B*. Dizemos que o enunciado *s*: `x:=y` é *morto* em *B* se `x` ou `y` receberem atribuição em *B* e *s* não esteja em *B*. A noção de que as atribuições a `x` "matam" `x:=y` é familiar a partir das definições incidentes, mas a idéia de que as atribuições a `y` também o fazem é especial neste problema. Notemos a importante conseqüência do fato de que diferentes atribuições `x:=y` matam-se uma às outras; *entrada*[*B*] pode conter somente um enunciado de cópia com `x` à esquerda.

Seja *U* o conjunto "universal" de todos os enunciados de cópia no programa. É importante notar-se que os diferentes enunciados `x:=y` são diferentes em *U*. Definamos *c_geradas*[*B*] como sendo o conjunto de todas as cópias geradas no bloco *B* e *c_mortas*[*B*] como sendo o conjunto de cópias em *U* que sejam mortas em *B*. As seguintes equações relacionam as quantidades definidas:

$$saída[B] = c_geradas[B] \cup (entrada[B] - c_mortas[B])$$
$$entrada[B] = \bigcap_{\substack{p \text{ é um} \\ \text{predecessor de } B}} saída[P] \text{ para } B \text{ não inicial}$$
$$entrada[B_1] = \varnothing \text{ onde } B_1 \text{ é o bloco inicial} \qquad (10.12)$$

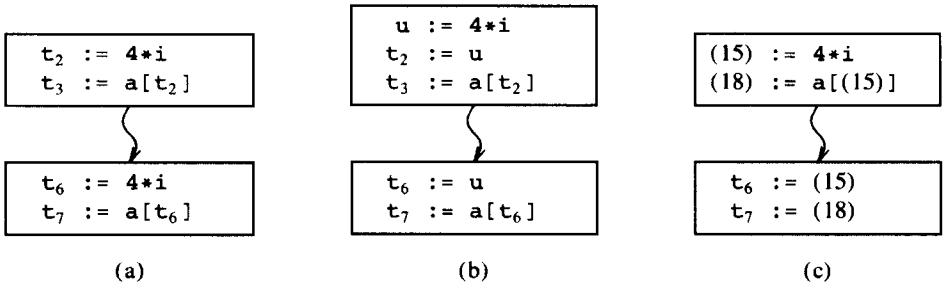


Fig. 10.34. Eliminando a subexpressão comum `4*i`.

As equações 10.12 serão idênticas às equações 10.10, se *c_mortas* for substituído por *e_mortas* e *c_geradas* por *e_geradas*. Por conseguinte, as equações 10.12 podem ser resolvidas pelo Algoritmo 10.3 e não iremos nos alongar mais nesse assunto. Forneceremos, no entanto, um exemplo que expõe algumas das nuances da otimização de cópias.

Exemplo 10.19. Consideremos o grafo de fluxo da Fig. 10.35. Aqui, *c_geradas*[B_1] = {x:=y} e *c_geradas*[B_3] = {x:=z}. Igualmente, *c_mortas* [B_2] = {x:=y}, uma vez que y recebe uma atribuição em B_2. Finalmente, *c_mortas*[B_1] = {x:=z} já que x recebe uma atribuição em B_1 e *c_mortas*[B_3] = {x:=y} pela mesma razão.

Os demais conjuntos *c_geradas* e *c_mortas* são $\varnothing$. Igualmente, *entrada*[B_1] = $\varnothing$ pelas equações 10.12. O Algoritmo 10.3 em uma passagem determina que

entrada[B_2] = *entrada*[B_3] = *saída*[B_1] = {x:=y}

Analogamente, *saída*[B_5] = $\varnothing$ e

saída[B_3] = *entrada*[B_4] = *saída*[B_4] = {x:=z}

Finalmente, *entrada*[B_5] = *saída*[B_2] $\cap$ *saída*[B_4] = $\varnothing$.

Observamos que nenhuma das duas cópias x:=y e x:=z "atinge" o uso de x em B_5, nos termos do Algoritmo 10.5. É uma verdade, ainda que irrelevante, que essas duas definições de x "atinjam" B_5, dentro do significado das definições incidentes. Por conseguinte, nenhuma cópia pode ser propagada, na medida em que não é possível substituir x por y (e, respectivamente, z) em todos os usos de x que a definição x:=y (e, respectivamente, x:=z) atinge. Poderíamos substituir x por z em B_4, mas isso não iria melhorar o código. □

Algoritmo 10.6 Propagação de cópias.

Entrada. Um grafo de fluxo *G* com cadeias-ud fornecendo as definições incidentes ao bloco *B* e com *c_entrada*[*B*] representando a solução para equações 10.12, isto é, o conjunto de cópias x:=y que atingem ao bloco *B* ao longo de cada percurso, sem atribuições a x ou y se seguindo à última ocorrência de x:=y no percurso. Precisamos também cadeias-du fornecendo os usos de cada definição.

Saída. Um grafo de fluxo revisado.

Método. Para cada cópia *s*:x:=y, fazer o seguinte.

1. Determinar aqueles usos de x que são atingidos por esta definição de x, nominalmente, *s*:x:=y.

2. Determinar se para cada uso de x encontrado em (1), *s* está em *c_entrada*[*B*], onde *B* é o bloco deste uso particular e, sobretudo, se nenhuma definição de x ou de y ocorre antes desse uso de x dentro de *B*. Relembrar que se *s* está em *c_entrada*[*B*], então *s* é a única definição de x que atinge *B*.

3. Se *s* atende às condições de (2), então remover *s* e substituir por y, todos os usos de x encontrados em (1). □

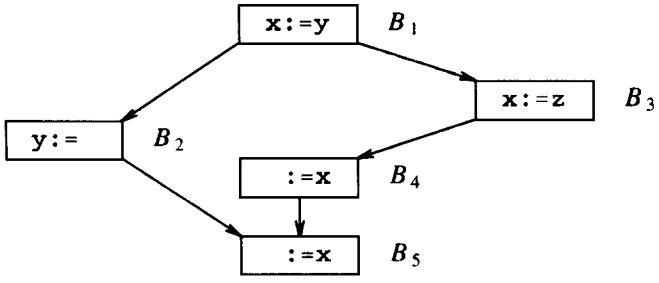


Fig. 10.35. Grafo de fluxo exemplo.

Detecção de Cômputos Laço-Invariantes

Iremos fazer uso das cadeias-ud para detectar as computações que sejam *laço-invariantes*, isto é, aquelas cujo valor não se modifica à medida que o controle percorra o laço. Como discutido na Seção 10.4, um laço é uma região que consiste em um conjunto de blocos com um cabeçalho que domine todos os outros blocos, e que a única maneira de se entrar no laço seja através do cabeçalho. Também exigimos que um laço tenha, pelo menos, uma forma de voltar ao cabeçalho, a partir de qualquer bloco do mesmo.

Se uma atribuição x:=y+z está a uma posição do laço, onde todas as possíveis definições de y e z estejam fora do mesmo (incluindo o caso especial onde y e/ou z sejam constantes), y+z é, por conseguinte, laço-invariante, já que seu valor será o mesmo a cada vez que x:=y+z for encontrado, por tanto tempo quanto o controle estiver dentro do laço. Todas essas atribuições podem ser detectadas a partir das cadeias-ud, isto é, a lista de todos os pontos de definição de y e z que atingem a atribuição x:=y+z.

Tendo reconhecido que o valor de x computado em x:=y+z não se modifica dentro do laço, suponhamos que exista um outro enunciado v:=x+w, onde w poderia somente ter sido definido fora do laço. Nesse caso, x+w também é laço-invariante. Podemos usar as idéias acima para realizar repetidas passagens sobre um laço, descobrindo mais e mais computações cujos valores sejam laço-invariantes. Se tivermos tanto as cadeias-ud quanto as cadeias-du, não precisaremos nem mesmo ter que realizar passagens repetidas sobre o código. A cadeia-du para a definição x:=y+z irá nos dizer onde este valor de x poderá ser usado e necessitamos somente verificar dentre esses usos de x, dentro do laço, aqueles que não usam outra definição de x. Essas atribuições laço-invariantes podem ser movidas para o pré-cabeçalho, providenciado que seus operandos, além de x, sejam também laço-invariantes, como discutido no próximo algoritmo.

Algoritmo 10.7 Detecção de cômputos laço-invariantes.

Entrada. Um laço *L* consistindo em um conjunto de blocos básicos, cada bloco contendo uma seqüência de enunciados de três endereços. Assumimos que as cadeias-ud, computadas como na Seção 10.5, estejam disponíveis para os enunciados individuais.

Saída. O conjunto de enunciados de três endereços que computa o mesmo valor a cada vez que for executado, do momento em que o controle entra no laço *L* até o instante em que o deixa.

Método. Forneceremos uma especificação um tanto informal do algoritmo, acreditando que os princípios ficarão claros.

1. Marcar como "invariantes" aqueles enunciados cujos operandos sejam todos constantes ou tenham todas as suas definições incidentes fora de *L*.

2. Repetir o passo (3) até que em alguma repetição não haja novos enunciados marcados como "invariantes".

3. Marcar como invariantes todos aqueles enunciados não previamente marcados dessa forma, cujos operandos sejam ou constantes, ou tenham todas as suas definições incidentes fora de *L*, ou tenham exatamente uma definição incidente e que essa definição seja um enunciado de *L*, marcado como invariante. □

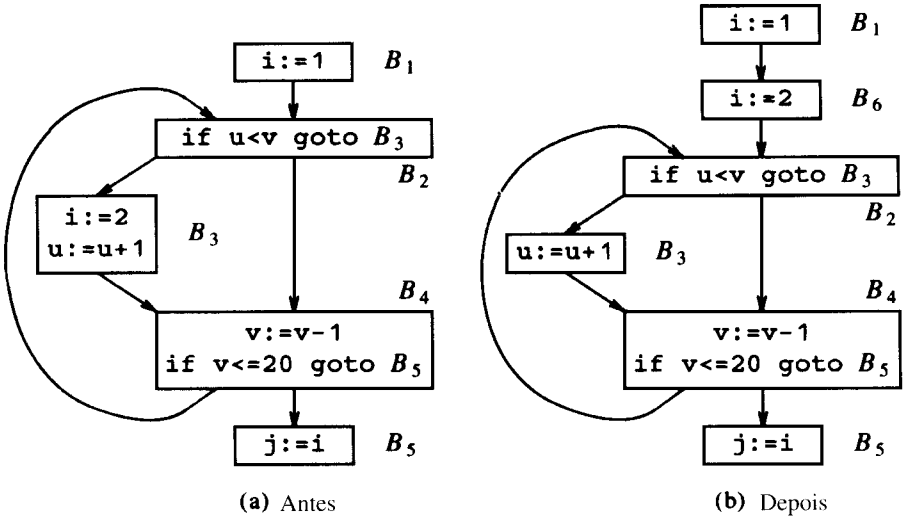


Fig. 10.36. Exemplo de uma movimentação de código ilegal.

Realizando a Movimentação de Código

Tendo encontrado os enunciados invariantes dentro de um laço, podemos aplicar a alguns deles uma otimização conhecida como *movimentação de código*, na qual os enunciados são movidos para o pré-cabeçalho do laço. As três seguintes condições asseguram que a movimentação de código não mude o que o programa computa. Nenhuma das condições é absolutamente essencial; as mesmas foram selecionadas porque são fáceis de verificar e aplicar a situações que ocorrem em programas reais. Iremos discutir mais tarde a possibilidade de relaxar essas condições.

As condições para o enunciado s:`x:=y+z` são:

1. Um bloco contendo s domina todos os nós de saída do laço, onde uma saída de um laço é um nó com um sucessor fora do laço.

2. Não há outro enunciado no laço que atribua valor a `x`. De novo, se `x` for uma variável temporária, recebendo atribuição somente uma vez, esta condição é certamente satisfeita e não precisa ser verificada.

3. Nenhum uso de `x` no laço é atingido por qualquer definição de `x` que não s. Esta condição também será satisfeita, normalmente, se `x` for uma variável temporária.

Os três próximos exemplos motivam as condições acima.

Exemplo 10.20. Mover um enunciado que não precise ser executado dentro de um laço para uma posição fora do mesmo pode mudar o que um programa computa, como mostramos na Fig. 10.36. Esta observação motiva a condição (1), uma vez que um enunciado que domine todas as saídas não poderá deixar de ser executado, assumindo que o laço não rode para sempre.

Consideremos o grafo de fluxo mostrado na Fig. 10.36(a). B_2, B_3 e B_4 formam um laço com cabeçalho B_2. O enunciado `i:=2` em B_3 é claramente laço-invariante. No entanto, B_3 não domina B_4, a única saída do laço. Se movermos `i:=2` para o pré-cabeçalho recentemente criado B_6, como mostrado na Fig. 10.36(b), podemos modificar o valor atribuído a `j` em B_5, naqueles casos em que B_3 jamais quedar executado. Por exemplo, se `u=30` e `v=25`, quando B_2 for atingido pela primeira vez, a Fig. 10.36(a) faz `j` igual a `1` em B_5, já que B_3 jamais será atingido, enquanto que a Fig. 10.36(b) faz `j` igual `2`. □

Exemplo 10.21. A condição (2) é requerida quando existe mais de uma atribuição a `x` no laço. Por exemplo, a estrutura do grafo de fluxo na Fig. 10.37 é a mesma que aquela da Fig. 10.36(a) e temos a opção de criar o pré-cabeçalho B_6, como na Fig. 10.36(b).

Como B_2 na Fig. 10.37 domina a saída B_4, a condição (1) não impede que `i:=3` seja movido para o pré-cabeçalho B_6. No entanto, se o fizermos, iremos atribuir `2` a `i`, sempre que B_3 for executado, e `i` terá o valor 2 quando atingirmos B_5, mesmo se seguirmos uma seqüência

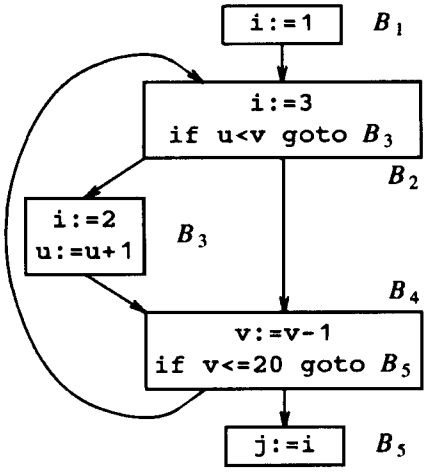


Fig. 10.37. Ilustrando a condição (2).

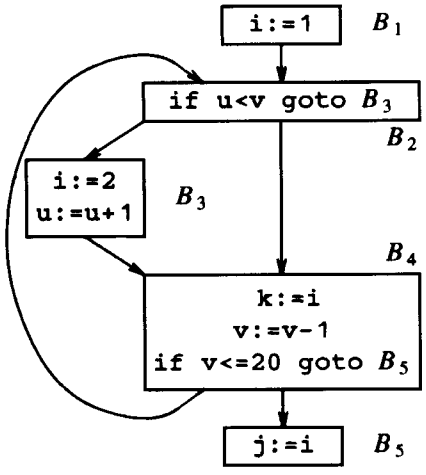


Fig. 10.38. Ilustrando a condição (3).

tal como $B_2 \rightarrow B_3 \rightarrow B_4 \rightarrow B_2 \rightarrow B_4 \rightarrow B_5$. Por exemplo, consideremos o que acontece se o valor de `v` for `22` e o de `u` `21`, quando B_2 for atingido pela primeira vez. Se o enunciado `i:=3` estiver em B_2, faremos `j` igual a `3` em B_5, mas se `i:=3` for removido para o pré-cabeçalho, faremos `j` igual a `2`. □

Exemplo 10.22. Vamos agora considerar a regra (3). O uso `k:=i` no bloco B_4 da Fig. 10.38 é atingido por `i:=1` no bloco B_1, bem como por `i:=2` em B_3. Conseqüentemente, não poderíamos mover `i:=2` para o pré-cabeçalho, porque o valor de `k` incidindo sobre B_5 mudaria no caso de `u>=v`. Por exemplo, se `u=v=0`, então `k` é feito igual a `1` no grafo de fluxo da Fig. 10.38, mas se `i:=2` for movido para o pré-cabeçalho, fazemos `k` igual a `2` uma vez e para todas as situações. □

Algoritmo 10.8 Movimentação de código.

Entrada. Um laço L com informações do encadeamento-ud e informações a respeito do dominador.

Saída. Uma versão revisada do laço com um pré-cabeçalho e (possivelmente) alguns enunciados movidos para o pré-cabeçalho.

Método.

1. Usar Algoritmo 10.7 para encontrar os enunciados laço-invariantes.

2. Para cada enunciado s definindo `x`, encontrado no passo (1), verificar:

   i) se está num bloco que domine todas as saídas de L
   ii) se `x` não é definido em algum local de L e
   iii) se todos os usos em L de `x` podem somente ser atingidos pela definição de `x` no enunciado s.

3. Mover, na ordem encontrada pelo Algoritmo 10.7, cada enunciado s encontrado em (1) e, atendendo às condições (2i), (2ii) e (2iii), para o novo pré-cabeçalho criado, providenciado que quaisquer operandos de s, que sejam definidos no laço L (no caso de s ter sido encontrado no passo (3) do Algoritmo 10.7) tenham tido seus enunciados de definição previamente movidos para o pré-cabeçalho. □

Para compreender por que nenhuma mudança para o que o programa computa pode ocorrer, as condições (2i) e (2ii) do Algoritmo 10.8 asseguram que o valor de `x` computado em s precisa ser o valor de `x` após qualquer bloco de saída de L. Quando movemos s para o pré-cabeçalho, s ainda será a defnição de `x` que atinge o final de qualquer bloco de saída de L. A condição (2iii) assegura que quaisquer usos de `x` dentro de L usaram, e irão continuar a usar, o valor de `x` computado por s.

Para ver por que a transformação não pode aumentar o tempo de execução do programa, temos apenas que observar que a condição (2i) assegura que s seja executado pelo menos uma a cada vez que o controle entrar em L. Após a movimentação de código, o mesmo será exercitado exatamente uma vez no pré-cabeçalho, e não mais em L, sempre que o controle lá entrar.

Estratégias Alternativas de Movimentação de Código

Podemos relaxar a condição (1) em alguma extensão se estivermos dispostos a correr o risco de, efetivamente, aumentarmos um pouco o tempo de execução do programa; naturalmente, jamais mudaremos o que o programa computa. A versão menos restrita da condição (1) da movimentação de código [isto é, o item 2(i) no Algoritmo 10.8] permite que movimentemos um enunciado s que atribui um valor a `x` somente se:

1'. O bloco que contém s ou domina todas as saídas do laço ou `x` não é usado fora do laço. Por exemplo, se `x` for uma variável temporária, podemos estar certos (em muitos compiladores) de que o valor será usado somente em seu próprio bloco. Em geral, a análise das variáveis vivas é necessitada para informar se `x` estará viva em qualquer saída do laço.

Se o Algoritmo 10.8 for modificado para usar a condição (1'), o tempo de execução irá ocasionalmente aumentar ligeiramente, mas podemos esperar que nos saiamos razoavelmente bem na média. O algoritmo modificado deve mover para o pré-cabeçalho certas computações que poderiam não ser executadas dentro do laço. Isto não somente arrisca retardar o programa significativamente, mas, também, pode causar um erro em certas circunstâncias. Por exemplo, a avaliação de uma divisão `x/y` num laço pode ser precedida por um teste para ver se `y=0`. Se movermos `x/y` para o pré-cabeçalho uma divisão por zero pode ocorrer. Por esta razão, não é inteligente usar a condição (1') a menos que a otimização possa ser inibida pelo programador ou que apliquemos a condição mais restrita (1) aos enunciados de divisão.

Mesmo que nenhuma das condições (2i), (2ii) e (2iii) do Algoritmo 10.8 seja atendida para uma atribuição `x:=y+z`, podemos ainda realizar a computação `y+z` fora do laço. Criamos uma nova variável temporária `t` e fazemos `t:=y+z` no pré-cabeçalho. Em seguida substituímos `x:=y+z` por `x:=t` dentro do laço. Em muitos casos podemos, em seguida, propagar o enunciado de cópia `x:=t`, como discutido antes nesta seção. Notemos que, se a condição (2iii) do Algoritmo 10.8 for atendida, isto é, se todos os usos de `x` no laço L forem definidos em `x:=y+z` (agora `x:=t`), podemos ficar seguros ao remover o enunciado `x:=t`, substituindo os usos de `x` em L por usos de `t` e colocando `x:=t` após cada saída do laço.

A Manutenção das Informações de Fluxo de Dados após a Movimentação de Código

As transformações do Algoritmo 10.8 não mudam as informações do encadeamento-ud, uma vez que, pelas condições (2i), (2ii) e (2iii), todos os usos da variável que recebeu a atribuição através de um enunciado movido s e que eram atingidos por s, ainda o são, a partir de sua nova posição (do enunciado movido). As definições das variáveis usadas por s ou estão fora de L, caso em que atingem o pré-cabeçalho, ou estão dentro de L, caso em que, pelo passo (3), seriam movidas para o pré-cabeçalho à frente (isto é, antes) de s.

Se as cadeias-ud são representadas por listas de apontadores para apontadores de enunciados (ao invés de através de listas de apontadores para enunciados), podemos manter as cadeias-ud ao movimentarmos um enunciado s, simplesmente modificando o apontador para s. Isto é, criamos para cada enunciado s um apontador p_s, o qual sempre aponta para s. Colocamos p_s em cada cadeia-ud que contenha s. Como conseqüência, não importa o quanto movamos s, teremos apenas que modificar p_s, independentemente de quantas cadeias-ud s estiverem participando. Naturalmente, o nível extra de indireção custa algum tempo e espaço do compilador.

Se representarmos as cadeias-ud por uma lista de endereços de enunciados (apontadores para endereços), poderemos ainda manter as cadeias-ud à medida que movimentarmos os enunciados. Mas necessitaremos, então, de cadeias-du por uma questão de eficiência. Quando movemos s, percorremos a sua cadeia-du, mudando a cadeia-ud em todos os usos que se refiram a s.

As informações sobre o dominador são modificadas ligeiramente pela movimentação de código. O pré-cabeçalho é, agora, o dominador imediato do cabeçalho e o dominador imediato do pré-cabeçalho é o nó que anteriormente era o dominador do cabeçalho. Isto é, o pré-cabeçalho é inserido na árvore dos dominadores como pai do cabeçalho.

Eliminação das Variáveis de Indução

Uma variável `x` é chamada de uma *variável de indução* de um laço L se, a cada vez que a mesma mudar de valor, for incrementada ou decrementada por alguma constante. Freqüentemente, uma variável de indução é incrementada pela mesma constante a cada vez que o controle passa ao longo do laço, como `i` no laço encabeçado por **for** `i:=1` **to** 10. No entanto, nossos métodos lidam com variáveis que são incrementadas ou decrementadas zero, uma, duas ou mais vezes à medida que vamos ao longo de um laço. O número de mudanças realizadas em uma variável de indução pode inclusive diferir em iterações diferentes.

Uma situação comum é aquela em que uma variável de indução, digamos `i`, indexa um *array* e alguma outra variável de indução, digamos `t`, cujo valor é uma função linear de `i`, é o deslocamento efetivo usado para dar acesso ao *array*. Freqüentemente, o único uso feito de `i` é o teste de terminação do laço. Podemos, então, nos livrar de `i`, substituindo seu teste por um outro sobre `t`.

Os algoritmos que se seguem lidam com uma classe restrita de variáveis de indução, para simplificar a apresentação. Algumas extensões dos algoritmos podem ser feitas adicionando-se mais casos, mas outras requerem que sejam provados teoremas a respeito das expressões envolvendo os operadores aritméticos usuais.

Iremos procurar pelas *variáveis básicas de indução*, que são aquelas variáveis `i` cujas únicas atribuições dentro do laço L são da forma `i:= i`$\pm c$, onde c é uma constante.[7] Procuramos, em seguida, por variáveis de indução adicionais `j` que sejam definidas somente uma vez dentro de L e por aquelas cujo valor seja uma função linear de alguma variável básica de indução `i`, onde `j` esteja definida.

Algoritmo 10.9 Detecção de variáveis de indução.

Entrada. Um laço L com informações das definições incidentes e do cômputo laço-invariante (provenientes do Algoritmo 10.7).

Saída. Um conjunto de variáveis de indução. Associada a cada variável de indução `j` está uma tripla `i`, c, d), onde `i` é uma variável básica de indução e c e d são constantes tais que o valor de `j` é dado por c*`i`+d, no ponto onde `j` é definida. Dizemos que `j` pertence à *família* de `i`. A variável básica de indução `i` pertence à sua própria família.

Método.

1. Encontrar todas as variáveis básicas de indução esquadrinhando os enunciados de L. Usamos aqui as informações do cômputo laço-invariante. Associada a cada variável básica de indução está a tripla (`i`, 1, 0).

2. Procurar pelas variáveis `k`, com uma única atribuição a `k` dentro de L, tendo uma das seguintes formas:

   `k:=j`$*b$, `k:=`b`*j`, `k:=j/`b, `k:+j`$\pm b$, `k:=`$b\pm$`j`

   onde b é uma constante e `j` uma variável de indução, básica ou não.

   Se `j` é básica, então `k` está na família de `j`. A tripla para `k` depende da instrução que a define. Por exemplo, se `k` é definida por `k:=j`$*b$, então a tripla para `k` é (`j`, b, 0). As triplas para os casos remanescentes podem ser determinadas similarmente.

   Se `j` não é básica, tornar `j` da família de `i`. Nossas exigências adicionais são que

   (a) não haja atribuições a `i` entre o único ponto de atribuição a `j` em L e a atribuição a `k`, e

   (b) nenhuma definição de `j` fora de L atinja `k`.

   O caso usual será quando as definições de `k` e `j` estejam em temporários dentro do mesmo bloco, caso que é fácil de se verificar. Em geral, as informações sobre as definições incidentes irão providenciar a verificação que necessitamos se analisarmos o grafo de fluxo do laço L para determinar aqueles blocos (e, por conseguinte, aquelas definições) nos percursos entre a atribuição a `j` e a atribuição a `k`.
   Computamos a tripla para `k` a partir da tripla (`i`, c, d) para `j` e a instrução que define `k`. Por exemplo, a definição `k:=`b`*j` leva a (`i`, $b*c$, $b*d$) para `k`. Notemos que as multiplicações em $b*c$ e $b*d$ podem ser feitas à medida que a análise prosseguir, porque b, c e d são constantes. □

Uma vez que as famílias de variáveis de indução tenham sido encontradas, modificamos as instruções para computar uma variável de indução de forma que use adições ou subtrações em vez de multiplicações. A substituição de uma instrução mais dispendiosa por uma mais econômica é chamada de *redução de capacidade*.

Exemplo 10.23. O laço que consiste no bloco B_2, na Fig. 10.39(a), possui uma variável básica de indução `i`, porque a única atribuição a `i` no laço incrementa seu valor em 1. A família de `i` contém t_2, com lado direito `4*i`. Por conseguinte, a tripla para t_2 é (`i`, 4,0). Semelhantemente, `j` é a única variável básica de indução no laço que consiste em B_3, e t_4, com a tripla (`j`, 4, 0), está na família de `j`.

Podemos também procurar por variáveis de indução no laço mais externo, com cabeçalho B_2 e blocos B_2, B_3, B_4, B_5. Ambos, `i` e `j`, são variáveis básicas de indução nesse laço maior. De novo, t_2 e t_4 são variáveis de indução com triplas (`i`, 4, 0) e (`j`, 4, 0), respectivamente.

O grafo de fluxo da Fig. 10.39(b) é obtido a partir daquele da Fig. 10.39(a), aplicando-se o próximo algoritmo. Discutiremos essa transformação abaixo. □

Algoritmo 10.10 Redução de capacidade aplicada a variáveis de indução.

Entrada. Um laço L com informações das definições incidentes e das famílias de variáveis de indução, computadas usando o Algoritmo 10.9.

Saída. Um laço revisado.

Método. Consideremos uma variável básica de indução `i` de cada vez. Para cada variável de indução `j`, na família de `i`, com tripla (`i`, c, d):

1. Criar uma nova variável `s` (mas, se duas variáveis j_1 e j_2 tiverem triplas iguais, criar somente uma nova variável para ambas).

2. Substituir a atribuição de `j` por `j:=s`.

3. Imediatamente após cada atribuição `i:=i`+n em L, onde n é uma constante, atrelar

   `s := s +` $c*n$

   onde a expressão $c*n$ é avaliada como uma constante, já que c e n são constantes. Colocar `s` na família de `i`, com tripla (`i`, c, d)

4. Resta assegurar que `s` seja inicializada com c*`i`+d à entrada do laço. A inicialização pode ser colocada ao final do pré-cabeçalho. A inicialização consiste em

[7] Em nossa discussão a respeito das variáveis de indução, "+" está em lugar do operador de adição, não de um operador genérico e, igualmente, em lugar de outros operadores aritméticos padrão.

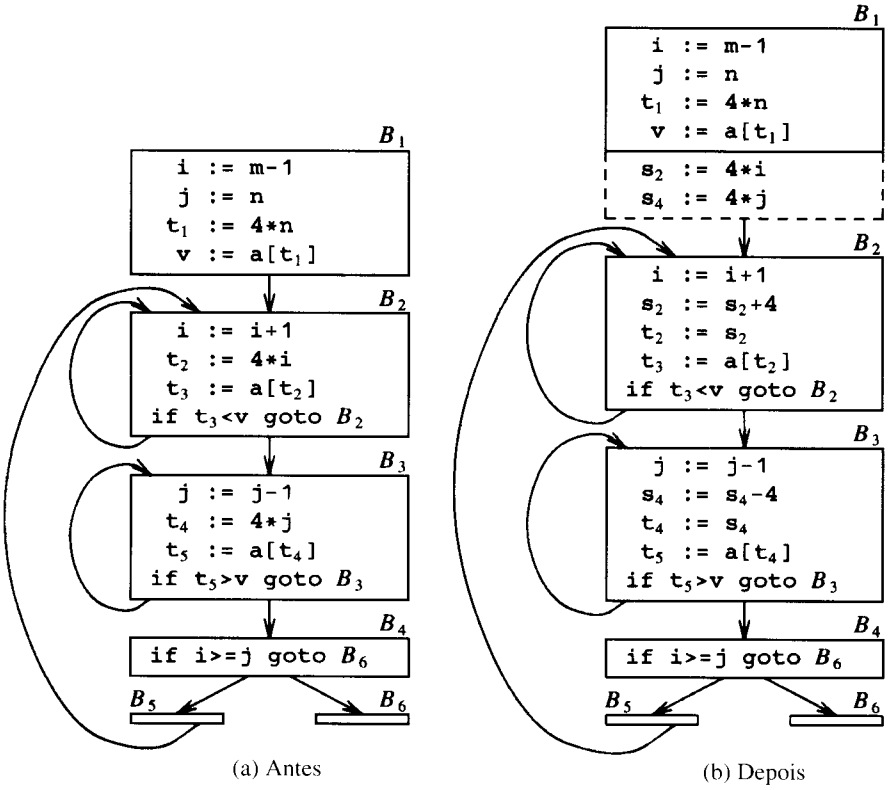


Fig. 10.39. Redução de capacidade.

```
s := c*i      /* somente s:=i se c for 1 */
s := s + d    /* omitir se d for 0 */
```

Note-se que `s` é uma variável de indução da família de `i`. □

Exemplo 10.24. Suponhamos que consideremos os laços da Fig. 10.39(a), de dentro para fora. Como o tratamento dos laços mais internos contendo B_2 e B_3 é muito similar, falaremos somente sobre o laço que envolve B_3. No Exemplo 10.23, observamos que a variável básica de indução, no laço em torno de B_3 é `j`, e que a outra variável de indução é t_4, com tripla (`j`, 4, 0). No passo (1) do Algoritmo 10.10, uma nova variável s_4 é construída. No passo (2), a atribuição t_4 := 4*`j` é substituída por t_4 := s_4. O passo (4) insere a atribuição s_4 := s_4-4, após a atribuição `j:=j-1`, onde o `-4` é obtido multiplicando-se o `-1` da atribuição a `j` e o 4 na tripla (`j`, 4, 0) para t_4.

Como B_1 serve como um pré-cabeçalho para o laço, podemos colocar a inicialização de s_4 ao fim do bloco B_1, que contém a definição de `j`. As instruções adicionadas são mostradas na extensão pontilhada do bloco B_1.

Quando um laço mais externo é considerado, o grafo de fluxo se parece com o mostrado na Fig. 10.39(b). Existem quatro variáveis, `i`, s_2, `j` e s_4, que poderiam ser consideradas variáveis de indução. No entanto, o passo (3) do Algoritmo 10.10 coloca as variáveis recém-criadas nas famílias de `i` e de `j`, respectivamente, para facilitar a eliminação de `i` e `j`, usando o próximo algoritmo. □

Após a redução de capacidade encontramos que o único uso de alguma variável de indução está em testes; podemos substituir um teste de uma tal variável por aquele de uma outra. Por exemplo, se `i` e `t` são variáveis de indução, tais que o valor de `t` seja sempre quatro vezes o valor de `i`, o teste `i>=j` é equivalente a `t>=4*j`. Após essa substituição, pode ser possível eliminar `i`. Notemos, entretanto, que, se `t=-4*i`, podemos necessitar modificar o operador relacional igualmente, porque `i>=j` é equivalente a `t<=-4*j`. No algoritmo seguinte consideramos o caso onde a constante multiplicativa é positiva, deixando a generalização para as constantes negativas como um exercício.

Algoritmo 10.11 Eliminação das variáveis de indução.

Entrada. Um laço L com informações sobre definições incidentes, cômputos laço-invariantes (do Algoritmo 10.7) e variáveis vivas.

Saída. Um laço revisado.

Método.

1. Considerar cada variável básica de indução `i` cujos únicos usos sejam para computar outras variáveis de indução em sua família e em desvios condicionais. Considerar algum `j` na família de `i`, preferivelmente uma em que c e d em sua tripla (`i`, c, d) sejam tão simples quanto possível (isto é, preferimos $c = 1$ e $d = 0$) e modificamos cada teste em que i apareça de modo a usar `j` no lugar. Assumimos, no que se segue, que c seja positivo. Um teste da forma `if i` **op-relacional** `x goto B`, onde `x` não seja uma variável de indução é substituído por

```
r:= c*x        /* r:=x se c for 1 */
r:=r+d         /* omitir se d for 0 */
if j op-relacional r goto B
```

onde `r` é uma nova variável temporária. O caso `if x` **op-relacional** `i goto B` é tratado analogamente. Se existirem duas variáveis de indução i_1 e i_2 no teste `if` i_1 **op-relacional** i_2 `goto B`, checamos então se ambas, i_1 e i_2, podem ser substituídas. O caso simples é quando temos j_1 com tripla (i_1, c_1, d_1) e j_2 com tripla (i_2, c_2, d_2) e $c_1 = c_2$ e $d_1 = d_2$. Então, i_1 **op-relacional** i_2 é equivalente a j_1 **op-relacional** j_2. Em casos mais complexos, a substituição do teste pode não valer a pena, porque podemos necessitar introduzir dois passos multiplicativos e um aditivo, enquanto apenas dois passos podem ser economizados eliminando-se i_1 e i_2.

Finalmente, remover todas as atribuições às variáveis de indução eliminadas a partir do laço L, na medida em que agora serão inúteis.

2. Consideremos agora cada variável de indução `j`, para as quais um enunciado `j:=s` foi gerado pelo Algoritmo 10.10. Primeiro, verificamos que podem não haver atribuições a `s` entre o enunciado introduzido `j:=s` e o uso de `j`. Na situação usual, `j` é usada no bloco no qual é definida, simplificando essa verificação; em caso contrário, as informações a respeito das definições incidentes mais alguma análise do grafo é necessitada para implementar a verificação. Substituir, então, todos os usos de `j` por usos de `s` e remover o enunciado `j:=s`. □

Exemplo 10.25. Consideremos o grafo de fluxo da Fig. 10.39(b). O laço mais interno em torno de B_2 contém duas variáveis de indução `i` e s_2, mas nenhuma das duas pode ser eliminada porque s_2 é usada para indexar o *array* `a` e `i` é usada num teste fora do laço. Semelhantemente, o laço em torno de B_3 contém as variáveis de indução `j` e s_4, mas nenhuma das duas pode ser eliminada.

Vamos aplicar o Algoritmo 10.11 ao laço mais externo. Quando as novas variáveis s_2 e s_4 foram criadas pelo Algoritmo 10.10, como discutido no Exemplo 10.24, s_2 foi colocada na família de `i` e s_4 na de `j`. Consideremos a família de `i`. O único uso de `i` é no teste para a terminação do laço no bloco B_4, e, dessa forma, `i` é uma candidata para eliminação no passo (1) do Algoritmo 10.11. O teste no bloco B_4 envolve as duas variáveis de indução `i` e `j`. Felizmente, as famílias de `i` e `j` contêm s_2 e s_4 com as mesmas constantes em suas triplas, porque as triplas são (`i`, 4, 0) e (`j`, 4, 0), respectivamente. O teste `i>=j` pode, por conseguinte, ser substituído por s_2>=s_4, permitindo que ambas, `i` e `j`, sejam eliminadas.

O passo (2) do Algoritmo 10.11 aplica a propagação de cópias às variáveis recém-criadas, substituindo t_2 e t_4 por s_2 e s_4, respectivamente. □

Variáveis de Indução com Expressões Laço-Invariantes

Nos Algoritmos 10.9 e 10.10, podemos permitir expressões laço-invariantes em lugar de constantes. Entretanto, a tripla (`i`, *c*, *d*), para uma variável de indução `j`, pode conter expressões laço-invariantes em vez de constantes. A avaliação dessas expressões deveria ser realizada fora do laço *L*, no pré-cabeçalho. Sobretudo, uma vez que o código intermediário requer que lá haja no máximo um operador por instrução, precisamos estar preparados para gerar os enunciados do código intermediário para a avaliação de expressões. A substituição dos testes no Algoritmo 10.11 requer que o sinal da constante multiplicativa *c* seja conhecido. Por esta razão, pode ser razoável restringir a atenção aos casos em que *c* seja uma constante conhecida.

10.8 LIDANDO COM PSEUDÔNIMOS

Se duas ou mais expressões denotam a mesma localização de memória, dizemos que as expressões são *sinônimas* uma da outra, ou *pseudônimos* da mesma localização. Nesta seção, iremos considerar a análise do fluxo de dados na presença de apontadores e procedimentos, ambos introduzindo pseudônimos.

A presença de apontadores torna a análise de fluxo de dados mais complexa, já que causa incerteza sobre o que é definido e usado. A única suposição segura, senão soubermos nada a respeito da localização para onde um apontador `p` possa apontar, é assumir que uma atribuição indireta através de um apontador possa potencialmente mudar (isto é, definir) qualquer variável. Precisamos assumir, também, que qualquer uso de dados endereçados por um apontador, como, por exemplo, `x:=*p`, pode potencialmente usar qualquer variável. Essas suposições resultam em mais variáveis vivas e definições incidentes e menos expressões disponíveis, do que seria realístico assumir. Felizmente, podemos usar a análise do fluxo de dados para indicar para o que um apontador poderia apontar, permitindo-nos, por conseguinte, obter uma informação mais acurada a partir de nossas análises de fluxo de dados.

Como nas atribuições envolvendo variáveis do tipo apontador, ao executarmos uma chamada de procedimento, talvez não tenhamos que realizar nossa suposição do pior caso — a de que tudo pode ser mudado — se for possível computar o conjunto de variáveis que o procedimento possa mudar. Como com todas as otimizações de código, podemos ainda cometer erros dentro das suposições conservativas. Isto é, os conjuntos de variáveis cujos valores "possam ser" modificados ou usados deveriam incluir apropriadamente as variáveis que fossem efetivamente modificadas ou usadas em alguma execução do programa. Iremos, como de praxe, simplesmente tentar chegar razoavelmente próximos aos verdadeiros conjuntos de variáveis modificadas e usadas, sem trabalhar excessiva e inadequadamente, ou cometendo erros que alterem o que o programa realiza.

Uma Linguagem Simples de Apontadores

Por uma questão de especificidade, vamos considerar uma linguagem na qual haja itens de dados elementares (por exemplo, inteiros e reais), requerendo uma palavra cada um, e *arrays* desses tipos. Façamos também existirem apontadores para esses elementos e para *arrays*, mas não para outros apontadores. Ficaremos satisfeitos em saber que um apontador `p` está apontando para alguma localização do *array* `a`, sem nos preocuparmos com que elemento particular de `a` está sendo apontado. Esse agrupamento de todos os elementos juntos de um *array*, na medida em que alvos de apontadores estejam sendo considerados, é razoável. Tipicamente, os apontadores serão usados como cursores para se percorrer todo um *array*, de forma que uma análise de fluxo de dados mais detalhada, se pudermos concretizá-la de todo, freqüentemente poderá nos dizer que, de qualquer forma, a um ponto particular do programa, `p` poderá estar apontando para qualquer um dos elementos de `a`.

Precisamos também realizar certas suposições a respeito de que operações aritméticas sobre apontadores são semanticamente significativas. Primeiro, se o apontador `p` aponta para um elemento de dados primitivo (uma palavra), então qualquer operação aritmética sobre `p` produz um valor que pode ser um inteiro, mas não um apontador. Se `p` aponta para um *array*, então uma adição ou subtração de um inteiro deixa `p` apontando para algum lugar no mesmo *array*, enquanto que outras operações aritméticas sobre apontadores produzem valores que não são apontadores. Conquanto nem todas as linguagens proíbam a movimentação de um apontador de um *array* `a` para um outro *array* `b` através de uma adição ao apontador, tal ação seria dependente da implementação particular assegurar que o *array* `b` se segue ao `a` na memória. É nosso ponto de vista que um compilador otimizante deveria aderir somente à definição de linguagem ao decidir que otimizações realizar. Cada implementador de compilador, no entanto, precisa fazer um julgamento sobre que otimizações específicas o compilador deveria ser permitido realizar.

Os Efeitos das Atribuições de Apontadores

Sob essas suposições, as únicas variáveis que poderiam ser possivelmente usadas como apontadores são aquelas declaradas como apontadores e temporários que recebam um valor que seja um apontador adicionado ou subtraído de uma constante. Iremos nos referir a todas essas variáveis como apontadores. Nossas regras para determinar para o que um apontador `p` pode apontar são como se segue.

1. Se existir um enunciado de atribuição *s* `:p:=&a`, então, imediatamente após *s*, `p` aponta somente para `a`. Se `a` for um *array*, então, após qualquer atribuição a `p` da forma `p:=&a±c` onde `c` seja uma constante, `p` poderá apenas apontar para `a`.[8] Como de praxe, `&a` é considerado referenciar a localização do primeiro elemento do *array* `a`.

[8]Nesta seção, + figura como o próprio em vez de simbolizar um operador genérico.

2. Se existir um enunciado de atribuição s : p : =q±c, onde c é um inteiro diferente de zero, e p e q apontadores, então, imediatamente após s, p pode apontar para qualquer *array* que q poderia apontar antes de s, e só para isso.

3. Se existir uma atribuição s : p : =q, então, imediatamente após s, p pode apontar para qualquer coisa que q poderia apontar antes de s.

4. Depois de qualquer outro tipo de atribuição a p, não há objeto para o qual p possa apontar; uma tal atribuição é provavelmente (dependendo da semântica de linguagem) sem significado.

5. Após qualquer atribuição a uma variável que não p, p aponta para o que quer que apontava antes da atribuição. Notemos que esta regra assume que nenhum apontador possa apontar para um outro apontador. O relaxamento desta suposição não torna as coisas mais difíceis particularmente e deixamos a generalização para o leitor.

Iremos definir *entrada*[B], para um bloco B, como sendo a função que fornece, para cada apontador p, o conjunto de variáveis para as quais p poderia estar apontando ao início de B. Formalmente, *entrada*[B] é um conjunto de pares da forma (p, a), onde p é um apontador e a uma variável, significando que p poderia apontar para a. Na prática, *entrada*[B] poderia ser representado como uma lista para cada apontador, a lista para p que fornece o conjunto de a's tais que (p, a) esteja em *entrada*[B]. Definimos *saída*[B] similarmente para o final de B.

Especificamos a função de transferência $trans_B$ que define o efeito do bloco B. Isto é, $trans_B$ é uma função que toma como argumento um conjunto de pares S, cada par da forma (p, a), sendo p um apontador e a uma variável de tipo não apontador, e produzindo um outro conjunto T. Presumivelmente, o conjunto ao qual $trans_B$ será aplicada é *entrada*[B] e o resultado da aplicação será *saída*[B]. Necessitamos somente dizer como computar *trans* para os enunciados singelos; $trans_B$ será, por conseguinte, uma composição de $trans_s$ para cada enunciado s do bloco B. As regras para se computar *trans* são como se segue.

1. Se s é p : =&a ou p : =&a±c, no caso em que a seja um *array*, então

$$trans_s(S) = (S - \{(p, b) \mid \text{qualquer variável } b\}) \cup \{(p, a)\}$$

2. Se s for p : =q±c, para q do tipo apontador e c, do tipo inteiro e diferente de zero, então

$$trans_s(S) = (S - \{(p, b) \mid \text{qualquer variável } b\}) \cup \{(p, b) \mid (q, b) \text{ está em } S \text{ e } b \text{ seja uma variável } array\}$$

Notemos que esta regra faz sentido mesmo se p = q.

3. Se s é p : =q, então

$$trans_s(S) = (S - \{(p, b) \mid \text{qualquer variável } b\}) \cup \{(p, b) \mid (q, b) \text{ está em } S\}$$

4. Se s atribui ao apontador p qualquer outra expressão, então

$$trans_s(S) = S - \{(p, b) \mid \text{qualquer variável } b\}$$

5. Se s não é uma atribuição a um apontador, então $trans_s(S) = S$

Podemos agora escrever as equações que relacionam *entrada*, *saída* e *trans* como segue.

$$saída[B] = trans_B(entrada[B])$$

$$entrada[B] = \bigcup_{\substack{P \text{ é um} \\ \text{predecessor de } B}} saída[P] \qquad (10.13)$$

onde, se B consiste dos enunciados $s_1, s_2, ..., s_k$, então

$$trans_B(S) = trans_{S_k}(trans_{S_{k-1}}(... (trans_{S_2}(trans_{S_1}(S))))...)).$$

As equações (10.13) podem ser resolvidas essencialmente como as definições incidentes no Algoritmo 10.2. Não iremos, por conseguinte, entrar em detalhes do algoritmo, mas nos contentaremos com um exemplo.

Exemplo 10.26. Consideremos o grafo de fluxo da Fig. 10.40. Supomos que a seja um *array* e c um inteiro; p e q são apontadores. Inicialmente, fazemos *entrada*[B_1] igual a ∅ (vazio). Então, $trans_{B_1}$ possui o efeito de remover quaisquer pares com primeiro componente q, e, em seguida, adicionar o par (q, c). Isto é, q é designado como apontando para c. Então,

$$saída[B_1] = trans_{B_1}(\emptyset) = \{(q, c)\}$$

Então, *entrada*[B_2] = *saída*[B_1]. O efeito de p : =&c é o de substituir todos os pares com primeiro componente p pelo par (p, c). O efeito de q : =&(a[2]) é o de substituir pares com primeiro componente q por (q, a). Notemos que q : =&a[2] é efetivamente uma atribuição da forma q : =&a+c para uma constante c. Podemos agora computar

$$saída[B_2] = trans_{B_2}(\{(q, c)\}) = \{(p,c)(q, a)\}$$

Similarmente, *entrada*[B_3] = {(q, c)} e *saída*[B_3] = {(p, a), (q, c)}.

Em seguida, encontramos que *entrada* [B_4] = *saída*[B_2] ∪ *saída*[B_3] ∪ *saída*[B_5]. Presumivelmente, *saída*[B_5] foi inicializado com ∅ (vazio) e não foi modificado nessa passagem ainda. No entanto, *saída*[B_2] = {(p, c), (q, a)}, e *saída*[B_3] = {(p, a), (q, c)}, e, dessa forma,

$$entrada[B_4] = \{(p, a), (p, c), (q, a), (q, c)\}$$

O efeito de p : =p+1 em B_4 é o de descartar a possibilidade de p não apontar para um *array*. Isto é,

$$saída[B_4] = trans_{B_4}(entrada[B_4]) = \{(p, a), (q, a), (q, c)\}$$

Notemos, entretanto, que, sempre que B_2 for executado, fazendo p apontar para c, uma ação semanticamente sem sentido terá lugar se p for usado indiretamente após p : =p+1 em B_4. Por conseguinte, esse grafo de fluxo não é "realista", mas efetivamente ilustra as inferências que podemos fazer a respeito dos apontadores.

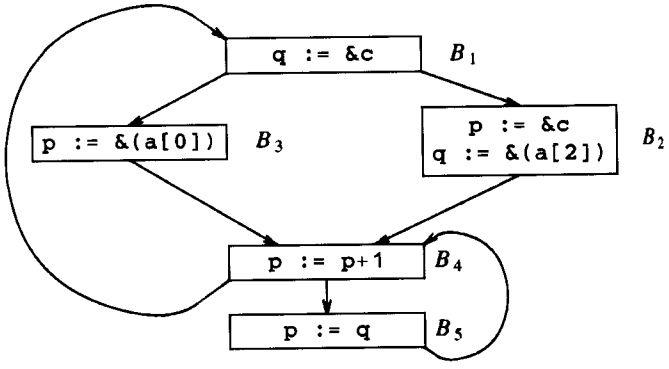


Fig. 10.40. Grafo de fluxo com as operações sobre apontadores exibidas.

Continuando, *entrada*[B_5] = *saída*[B_4] e $trans_{B_4}$ copia os alvos de `q` e os dá igualmente a `p`. Uma vez que `q` pode apontar somente para `a` ou `c` em *entrada*[B_5],

$$saída[B_5] = \{(p,a),\ (p,c),\ (q,a),\ (q,c)\}$$

Na passagem seguinte, encontramos que *entrada*[B_1] = *saída*[B_4] e, por conseguinte, *saída*[B_1] = {(`p,a`), (`q,c`)}. Este valor também são os novos *entrada*[B_2] e *entrada*[B_3], mas esses mesmos valores não mudam *saída*[B_2] ou *saída*[B_3], nem é *entrada*[B_4] modificada. Convergimos, dessa forma, para a resposta desejada. □

Fazendo Uso das Informações sobre os Apontadores

Suponhamos que *entrada*[B] seja o conjunto de variáveis apontadas por cada apontador ao início do bloco B e que tenhamos uma referência a um apontador `p` dentro do bloco B. Começando por *entrada*[B], aplicamos $trans_s$ para cada enunciado s do bloco B que preceda a referência a `p`. Este cômputo nos diz para o que `p` poderia apontar no enunciado particular, onde aquela informação é importante.

Vamos supor agora que tenhamos determinado para o que cada apontador poderia apontar quando um tal apontador fosse usado numa referência indireta, quer à esquerda ou à direita do símbolo de atribuição. Como podemos usar esta informação para obter soluções mais acuradas para os problemas usuais de fluxo de dados? Em cada caso, precisamos considerar em que direção os erros são conservativos e precisamos utilizar as informações dos apontadores numa forma em que somente erros conservativos sejam cometidos. Para ver como esta escolha é feita, vamos considerar dois exemplos: as definições incidentes e a análise das variáveis vivas.

Para calcular as definições incidentes, podemos usar o Algoritmo 10.2, mas necessitamos conhecer os valores para os conjuntos *mortas* e *geradas* de um bloco. O último valor é computado como de praxe para os enunciados que não sejam atribuições indiretas através de apontadores. Uma atribuição indireta `*p:=a` é considerada gerar uma definição para cada variável `b` para a qual `p` possa apontar. A suposição é conservativa, porque, como discutido na Seção 10.5, é geralmente conservativo assumir que as definições atinjam um ponto, ainda que, na realidade, não o façam.

Ao computarmos *mortas*, assumimos que `*p:=a` mate definições de `b` somente se `b` não for um *array* e for a única variável para a qual `p` poderia estar apontando. Se `p` puder apontar para duas ou mais variáveis, não assumimos que as definições de quaisquer delas sejam mortas. De novo, estamos sendo conservativos porque permitimos que as definições de `b` ultrapassem `*p:=a`, e atinjam o que quer que possam, a menos que possamos provar que `*p:=a` tenha redefinido `b`. Em outras palavras, quando temos dúvida, assumimos que uma definição incida.

Para as variáveis vivas, podemos usar o algoritmo 10.4, mas precisamos reconsiderar como *definidas* e *usos* devem ser definidos para os enunciados da forma `*p:=a` e `a:=*p`. O enunciado `*p:=a` usa somente `a` e `p`. Dizemos que o mesmo define `b` somente se `b` for a única variável para a qual `p` possa apontar. Esta suposição permite que os usos de `b` passem, a menos que sejam seguramente bloqueados pela atribuição `*p:=a`. Por conseguinte, jamais poderemos afirmar que a variável `b` esteja morta em um ponto quando está de fato viva. O enunciado `a:=*p` certamente representa uma definição de `a`. Representa igualmente um uso de `p` e um uso de qualquer variável para a qual `p` poderia apontar. Maximizando os possíveis usos, de novo maximizamos nossas estimativas das variáveis vivas. Maximizando as variáveis vivas estamos normalmente sendo conservativos. Por exemplo, poderíamos gerar código para armazenar uma variável morta, mas jamais iremos deixar de armazenar para uma que estivesse viva.

Análise Interprocedimental do Fluxo de Dados *

Até agora, falamos de "programas" que eram procedimentos singelos e, conseqüentemente, grafos de fluxo simples. Iremos agora ver como capturar informações a partir de muitos procedimentos interatuantes. A idéia é agora determinar como cada procedimento influencia as informações a respeito de *geradas, mortas, usos* ou de *definidas* dos demais procedimentos, e, então, computar nossas informações de fluxo de dados para cada procedimento, como antes.

Durante a análise de fluxo de dados, teremos que lidar com os pseudônimos estabelecidos pelos parâmetros nas chamadas de procedimento. Como não é possível para duas variáveis globais denotar o mesmo endereço de memória, pelo menos um, dentro de um par de pseudônimos, terá que ser um parâmetro formal. Como os parâmetros formais podem ser transmitidos para procedimentos, é possível que dois parâmetros formais sejam pseudônimos.

Exemplo 10.27. Suponhamos ter um procedimento `p` com dois parâmetros formais `x` e `y` transmitidos por referência. Na Fig. 10.41, vemos uma situação na qual `b+x` é computado em B_1 e B_3. Suponhamos que os únicos percursos a partir do bloco B_1 para B_3 passem através de B_2 e que não existam atribuições a `b` e a `x` ao longo de quaisquer de tais percursos. Está, `b+x` disponível em B_3? A resposta depende de `x` e `y` poderem denotar ou não o mesmo endereço de memória. Por exemplo, poderia haver uma chamada `p(z,z)`, ou talvez uma chamada de `p(u,v)`, onde `u` e `v` fossem parâmetros formais de outro procedimento `q(u,v)` e uma chamada `q(z,z)` ser possível.

Similarmente, é possível para `x` e `y` serem pseudônimos se `x` for um parâmetro formal, digamos de `p(x,w)` e `y` uma variável com um escopo acessível a algum procedimento `q` que chame `p`, digamos, através de uma chamada `p(y,t)`. Situações ainda mais complicadas poderiam tornar `x` e `y` pseudônimos e iremos desenvolver em breve algumas regras gerais para determinar todos esses pares de pseudônimos. □

Irão acontecer algumas situações em que será conservativo não considerar os nomes de variáveis como pseudônimos. Por exemplo, nas definições incidentes, se desejarmos estabelecer que uma definição de `a` é morta por uma definição de `b`, melhor faremos em nos assegurar que, definitivamente, `a` e `b` sejam pseudônimos sempre que a definição de `b` for executada. Em outras situações, será conservativo considerar os nomes de variáveis como pseudônimos sempre que houver dúvida. O Exemplo 10.27 é um desses casos. Se a expressão disponível `b+x` não deve ser morta por uma definição de `y`, melhor faremos em assegurar que nem `b` nem `x` possam ser pseudônimos de `y`.

Um Modelo de Código com Chamadas de Procedimentos

Para ilustrar como poderíamos lidar com a polionomia, vamos considerar uma linguagem que permita procedimentos recursivos, alguns dos quais possam se referir tanto a variáveis locais quanto globais. Os dados disponíveis a um procedimento consistem somente nas variáveis globais e nas suas próprias variáveis locais; isto é, não há estrutura de blocos na linguagem. Os parâmetros são transmitidos por referência. Requeremos que todos os procedimentos tenham um grafo de fluxo com uma única *entrada* e um único nó de *retorno* que faz o controle retornar para o procedimento chamador. Supomos, por conveniência, que cada nó repouse em algum percurso da entrada para o retorno.

Suponhamos, agora, que estejamos num procedimento `p` e cheguemos a uma chamada de procedimento `q(u,v)`. Se estivermos interessados em computar definições incidentes, expressões disponíveis ou quaisquer outras análises de fluxo de dados, precisamos saber se `q(u,v)`

*O termo **interprocedural** poderia também ser utilizado. Preferimos, no entanto, o termo, já existente na língua portuguesa. (N. do T.)

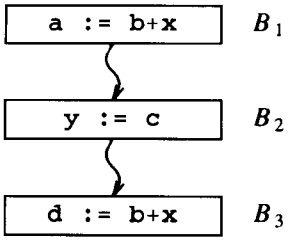


Fig. 10.41. Ilustração de problemas da polionomia.

pode mudar o valor de alguma variável. Notemos que dizemos "pode mudar" em lugar de "mudará". Como em todos os problemas de fluxo de dados, é impossível saber com certeza se o valor de uma variável é modificado ou não. Podemos somente encontrar um conjunto que inclua todas as variáveis cujos valores mudem realmente e talvez algumas que não. Sendo cuidadosos, podemos reduzir a última classe de variáveis, obtendo uma boa aproximação da realidade e errando somente dentro do aspecto conservativo.

As únicas variáveis cujos valores poderiam ser definidos pela chamada `q(u,v)` seriam as globais e as variáveis `u` e `v` que poderiam ser locais a `p`. As definições das variáveis locais a `q` não produzem conseqüências após o retorno das chamadas. Ainda que `p=q`, serão outras cópias das variáveis locais de `q` que mudarão e essas cópias desaparecerão após o retorno. É fácil se determinar que variáveis globais são explicitamente definidas por `q`; simplesmente vemos quais têm definições em `q` ou são definidas em uma chamada de procedimento feita por `q`. Adicionalmente, `u` e/ou `v`, que podem ser globais, mudam se `q` possuir uma definição de seu primeiro e/ou segundo parâmetros formais, respectivamente, ou se esses parâmetros formais forem transmitidos como parâmetros atuais por `q` para outro procedimento que as defina. No entanto, nem todos os parâmetros atuais modificados por uma chamada para `q` precisam ser definidos explicitamente por `q` ou por um dos procedimentos que o mesmo chame, porque as variáveis podem ter pseudônimos.

Cômputo dos Pseudônimos*

Antes de podermos responder à questão sobre que variáveis poderiam mudar num dado procedimento, precisamos desenvolver um algoritmo para encontrar os pseudônimos. O enfoque que iremos usar aqui é simples. Computamos a relação $\equiv$ sobre as variáveis que formalizam a noção "é um pseudônimo de". Agindo dessa maneira, não distinguimos as ocorrências de uma variável em diferentes chamadas do mesmo procedimento, apesar de distinguirmos as variáveis locais a procedimentos diferentes tendo, no entanto, o mesmo identificador.

Para tornar as coisas mais simples, não tentaremos diferenciar os conjuntos de pseudônimos em diferentes pontos do programa. Ao invés, se duas variáveis puderem, de alguma forma, ser pseudônimos, assumimos que sempre o poderão ser. Finalmente, faremos a suposição conservativa de que $\equiv$ é transitiva, de forma que as variáveis sejam agrupadas em classes de equivalência e duas variáveis podem ser pseudônimos somente se estiverem na mesma classe.

Algoritmo 10.12 Cômputo simples de pseudônimos.

Entrada. Uma coleção de procedimentos e variáveis globais.

Saída. Uma relação de equivalência com a propriedade de que, sempre que existir uma posição no programa em que `x` e `y` forem pseudônimos, $x \equiv y$; a recíproca não precisa ser verdadeira.

Método.

1. Renomear as variáveis, se necessário, de forma que não haja dois procedimentos que usem o mesmo parâmetro formal ou identificador de variável local e que não haja variáveis locais, parâmetros formais ou variáveis globais compartilhando o mesmo identificador.

2. Se existir um procedimento $p(x_1, x_2, \ldots, x_n)$ e uma invocação $p(y_1, y_2, \ldots, y_n)$ desse procedimento, fazer $x_i \equiv y_i$, para todo i. Isto é, cada parâmetro formal pode ser um pseudônimo de qualquer um de seus parâmetros atuais correspondentes.

3. Fazer o fechamento reflexivo e transitivo das correspondências entre os atuais e formais, adicionando

   a) $x \equiv y$, sempre que $y \equiv x$.
   b) $x \equiv z$, sempre que $x \equiv y$ e $y \equiv z$, para algum y. □

Exemplo 10.28. Consideremos os esqueletos dos três procedimentos mostrados na Fig. 10.42, onde é assumido que os parâmetros sejam transmitidos por referência. Existem duas variáveis globais, `g` e `h`, e duas variáveis locais, `i` para o procedimento `main` e `k` para o procedimento `two`. O procedimento `one` possui os parâmetros formais `w` e `x`, o procedimento `two`, os formais `y` e `z`, e `main` não possui parâmetros formais. Por conseguinte, nenhuma renomeação de variáveis é necessária. Computamos primeiro a polionomia devida às correspondências entre os parâmetros formais e atuais.

A chamada de `one` por parte de `main` faz $h \equiv w$ e $i \equiv x$. A primeira chamada de `two` por `one` faz $w \equiv y$ e $w \equiv z$. A segunda, $g \equiv y$ e $x \equiv z$.

A chamada de `one` por `two` faz $k \equiv w$ e $y \equiv x$. Ao tomarmos, neste exemplo, o fechamento transitivo dos relacionamentos entre os pseudônimos, representado por $\equiv$, concluímos, então, que todos os nomes de variáveis são possíveis pseudônimos, uns dos outros. □

O cômputo das polionomias pelo Algoritmo 10.12 não resulta freqüentemente um grau de sinonímia tão extenso como encontramos no Exemplo 10.28. Intuitivamente, não devemos esperar que, freqüentemente, duas variáveis de tipos diferentes sejam sinônimas uma da outra. Sobretudo, o programador possui, indubitavelmente, tipos conceituais para essas variáveis. Por exemplo, se o primeiro parâmetro formal de um procedimento `p` representa uma velocidade, pode ser

```
global g, h;
    procedure main( );
        local i;
        g :=  ... ;
        one(h, i)
    end
    procedure one(w, x);
        x :=  ... ;
        two(w, w);
        two(g, x)
    end;
    procedure two(y, z);
        local  k;
        h :=  ... ;
        one(k, y)
    end
```

Fig. 10.42. Procedimentos de exemplo.

*Quando temos mais de um nome para o mesmo objeto de dados no mesmo ambiente de referenciamento. Uma boa referência para esse tema está nos Capítulos 6 e 7 de Pratt [1984]. (N. do T.)

```
(1)  para cada procedimento p faça modificadas [p] := definidas [p]; /*inicializar*/
(2)  enquanto ocorrerem mudanças a qualquer modificadas[p] faça
(3)      para i := 1 até n faça
(4)          para cada procedimento q chamado por p_i faça início
(5)              adicionar quaisquer variáveis globais em modificadas[q] a modificadas[p_i];
(6)              para cada parâmetro formal x (o j-ésimo) de q faça
(7)                  se x está em modificadas [q] então
(8)                      para cada chamada de q por p_i faça
(9)                          se a, o j-ésimo parâmetro atual da chamada,
                                 for global ou parâmetro formal de p_i
(10)                             então adicionar a a modificadas[p_i]
         fim
```

Fig. 10.43. Algoritmo iterativo para computar *modificadas*.

esperado que o primeiro argumento em qualquer chamada a `p` será igualmente pensado pelo programador como uma velocidade. Por conseguinte, esperamos intuitivamente que a maioria dos programas produza pequenos grupos de possíveis pseudônimos.

Análise de Fluxo de Dados na Presença de Chamadas de Procedimentos

Vamos considerar, por exemplo, como as expressões disponíveis podem ser calculadas na presença de chamadas de procedimentos, onde os parâmetros são transmitidos por referência. Como na Seção 10.6, precisamos determinar quando uma variável poderia ser definida, matando, por conseguinte, uma expressão e precisamos determinar quando as expressões são geradas (avaliadas).

Podemos definir, então, para cada procedimento `p`, um conjunto, *modificadas*[`p`], cujo valor é o conjunto de variáveis globais e parâmetros formais de `p` que poderiam ser modificados durante a sua execução. A este ponto, não contamos uma variável como modificada se um membro de sua classe de equivalência de pseudônimos é modificado.

Seja *definidas* [`p`] o conjunto de parâmetros formais e variáveis globais que tenham definições explícitas dentro de `p` (não incluindo aquelas(es) definidas(os) dentro de procedimentos chamados por `p`). Para escrever as equações para *modificadas* [`p`], temos somente que relacionar os nomes de variáveis globais e os parâmetros formais de `p` que sejam usados como parâmetros atuais em chamadas feitas por `p`, aos parâmetros formais correspondentes dos procedimentos chamados. Podemos escrever:

$$modificadas\,[p] = definidas[p] \cup A \cup G \qquad (10.14)$$

onde

1. $A = \{a \mid a$ é uma variável global ou parâmetro formal de `p` tal que, para algum procedimento `q` e inteiro i, `p` chama `q` com `a` como i-ésimo parâmetro atual e o i-ésimo parâmetro formal de `q` está em *modificadas* [`q`] $\}$.

2. $G = \{g \mid g$ é global em *modificadas* [`q`] e `p` chama `q`$\}$.

Não deveria soar com surpresa que a Equação (10.14) possa ser resolvida para um conjunto de procedimentos através de uma técnica iterativa. Apesar da solução não ser única, precisamos somente da menor. Podemos convergir para a solução começando com uma aproximação bem pequena e reiterando. A aproximação bem pequena, óbvia, com a qual começar é *modificadas*[`p`] = *definidas*[`p`]. Os detalhes da iteração são deixados como um exercício para o leitor.

É valioso considerar a ordem na qual os procedimentos deveriam ser visitados na iteração abaixo. Por exemplo, se os procedimentos não são mutuamente recursivos, podemos, então, visitar primeiro aqueles que não chamam a nenhum outro (deve haver pelo menos um). Para esses procedimentos, *modificadas* = *definidas*. Em seguida, podemos computar *modificadas* para aqueles procedimentos que chamam somente procedimentos que não chamam a nada mais. Podemos aplicar (10.14) para esse grupo seguinte de procedimentos diretamente, já que *modificadas* [`q`] será conhecido para qualquer `q` em (10.14).

Esta idéia pode ser tornada mais precisa como se segue. Desenhamos um *grafo de chamadas*, cujos nós são procedimentos, com um lado de `p` para `q` se `p` chamar `q`[9]. Uma coleção de procedimentos que não sejam mutuamente recursivos terá um grafo de chamadas acíclico. Nesse caso, visitamos os nós somente uma vez.

Fornecemos agora um algoritmo para processar *modificadas*.

Algoritmo 10.13 Análise interprocedimental das variáveis modificadas.

Entrada. Uma coleção de procedimentos $p_1, p_2, \ldots, p_n$. Se o grafo de chamadas for acíclico, assumimos que p_i chame p_j somente se $j < i$. Caso contrário, não fazemos suposição a respeito de que procedimento chama o quê.

Saída. Para cada procedimento `p`, produzimos *modificadas*[`p`], o conjunto de variáveis globais e parâmetros formais de `p` que podem ser modificados explicitamente por `p`, sem polionomias.

Método.

1. Computar *definidas* [`p`] para procedimento `p` por inspeção.

2. Executar o programa da Fig. 10.43 para computar *modificadas*. □

Exemplo 10.29. Vamos considerar a Fig. 10.42 de novo. Por inspeção, *definidas* [`main`] = {`g`}, *definidas* [`one`] = {`x`} e *definidas* [`two`] = {`h`}. Esses são os valores iniciais de *modificadas*.O grafo de chamadas dos procedimentos é mostrado na Fig. 10.44. Iremos considerar `two`, `one` e `main` como a ordem na qual visitaremos os procedimentos.

Consideremos p_i = `two` no programa da Fig. 10.42. `q` pode somente ser o procedimento `one` na linha (4). Como inicialmente *modificadas* [`one`] = {`x`}, nada é adicionado a *modificadas* [`two`] à linha (5). Às linhas (6) e (7) precisamos considerar somente o segundo parâmetro formal do procedimento `one`, uma vez que o primeiro parâmetro atual é local a `two`. Na única chamada de `one` por parte de `two`, o segundo parâmetro atual é `y` e seu parâmetro formal correspondente, `x`, é modificado, e, conseqüentemente, fazemos *modificadas* [`two`] igual a {`h`, `y`} à linha (10).

Consideremos, agora, p_i = `one`. À linha (4), `q` pode somente ser o procedimento `two`.

[9]Assumimos aqui que não haja variáveis do tipo procedimento. Essas complicariam a construção do grafo de chamadas e precisaríamos determinar os possíveis parâmetros atuais correspondentes aos parâmetros formais do tipo procedimento ao mesmo tempo que construíssemos os lados do grafo de chamadas.

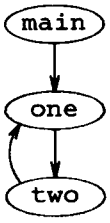


Fig. 10.44. Grafo de chamadas.

À linha (5), h é global em *modificadas*[two], e, assim, fazemos *modificadas* [one] = {h, x}. Às linhas (6) e (7), somente o primeiro parâmetro formal de two está em *modificadas* [two], e, dessa forma, precisamos adicionar g e w a *modificadas* [one], à linha (10), sendo esses os dois primeiros parâmetros atuais nas chamadas do procedimento two. Por conseguinte, *modificadas* [one]={g, h, w, x}.

Consideremos, agora, main. O procedimento one modifica ambos os parâmetros formais, e, por conseguinte, ambos, h e i, serão modificados na chamada de one por parte main. No entanto, i é uma variável local e não precisa ser considerada. Conseqüentemente, fazemos *modificadas*[main] = {g, h}. Finalmente, repetimos o laço **enquanto** da linha (2). Ao reconsiderarmos two, encontramos que one modifica a variável global g. Por conseguinte, a chamada para one(k, y) faz com que g seja modificada e, dessa forma, *modificadas* [two] = {g, h, y}. Não ocorrem mudanças ulteriores na iteração. □

Uso das Informações do Conjunto Modificadas

Como um exemplo da forma que *modificadas* pode ser usado, consideremos o cômputo global das subexpressões comuns. Vamos supor que estejamos computando as expressões disponíveis para um procedimento p e desejemos computar *d_mortas*[*B*] para um bloco *B*. Uma definição da variável a precisa ser encarada como matando qualquer expressão envolvendo a, a menos que a seja um pseudônimo (lembremos que a é um pseudônimo de si mesma) de alguma variável em *modificadas* [q]. Por conseguinte, as informações computadas pelos Algoritmos 10.12 e 10.13 podem ser usadas para construir uma aproximação segura do conjunto de expressões mortas.

Para computar as expressões disponíveis nos programas com chamadas de procedimentos, precisamos também ter uma forma conservativa de estimar o conjunto de expressões geradas por uma chamada de procedimento. Para sermos conservativos, podemos assumir que a + b seja gerada por uma chamada para q se e somente se, em cada percurso da entrada de q até sua saída, encontramos a + b sem nenhuma redefinição subseqüente de a ou b. Quando procuramos por ocorrências de a + b, não podemos aceitar x + y como uma tal ocorrência, a menos que estejamos seguros de que, em cada chamada de q, x seja um pseudônimo de a e y de b.

Fazemos essa exigência porque é conservativo errar assumindo que uma expressão não está disponível quando, de fato, o está. Na mesma trilha, precisamos assumir que a + b seja morta por uma definição de qualquer z que pudesse possivelmente ser um pseudônimo de a ou de b. Por conseguinte, a forma mais simples de se computar as expressões disponíveis para todos os nós de todos os procedimentos é assumir que uma chamada não gere nada e que *d_mortas*[*B*] para todos os blocos *B* é computado como acima. À medida que não se espere que muitas expressões sejam geradas por um procedimento típico, esta abordagem é boa o suficiente para a maior parte dos propósitos.

Uma alternativa complicada e mais acurada para o cômputo das expressões disponíveis é a de computar *geradas* [p] para cada procedimento p iterativamente. Podemos inicializar *geradas* [p] com o conjunto de expressões disponíveis ao fim do nó de retorno de p de acordo com o método acima. Isto é, nenhuma polionomia é permitida para as expressões geradas; a + b representa somente a si mesma, mesmo que outros nomes de variáveis possam ser pseudônimos de a ou de b.

Computemos agora as expressões disponíveis para todos os nós de todos os procedimentos, de novo. No entanto, uma chamada q(a, b) gera aquelas expressões em *geradas* [q] com a e b substituindo os parâmetros formais correspondentes de q. *d_mortas* permanece como antes. Um novo valor de *geradas*[p], para cada procedimento p, pode ser encontrado olhando-se para quais expressões estão disponíveis ao final do retorno de p. Esta iteração pode ser repetida até que não tenhamos mais modificações nas expressões disponíveis a qualquer nó.

10.9 ANÁLISE DE FLUXO DE DADOS DE GRAFOS DE FLUXO ESTRUTURADOS

Programas sem comandos de desvio possuem grafos de fluxo redutíveis; assim também o são os programas encorajados por diversas metodologias de programação. Vários estudos de amplas classes de programas têm revelado que virtualmente todos os programas escritos por pessoas têm grafos de fluxo que são redutíveis.[10] Esta observação é relevante para fins de otimização, porque podemos encontrar algoritmos otimizantes que rodam significativamente mais rápido sobre grafos de fluxo redutíveis. Nesta seção, discutimos uma variedade de conceitos de grafos de fluxo, tais como a "análise de intervalos", que são primariamente relevantes para os grafos de fluxo estruturados. Em essência, iremos aplicar as técnicas de tradução dirigida pela sintaxe, desenvolvidas na Seção 10.5, no âmbito mais geral onde a sintaxe não providencia necessariamente a estrutura, mas o grafo de fluxo o faz.

Busca em Profundidade

Existe um ordenamento útil dos nós num grafo de fluxo, conhecido como *ordenamento em profundidade*, que é uma generalização da travessia (ou caminhamento) em profundidade de uma árvore, introduzida na Seção 2.3. Um ordenamento em profundidade pode ser usado para detectar laços em qualquer grafo de fluxo; também auxilia na aceleração de algoritmos iterativos de fluxo de dados, tais como os discutidos na Seção 10.6. O ordenamento em profundidade é criado começando-se pelo nó inicial e buscando-se todo o grafo, tentando visitar os nós tão distantes quanto possível do nó inicial e tão rapidamente quanto possível (*profundidade em primeiro lugar*). A rota da busca forma uma árvore. Antes de fornecermos o algoritmo, vamos considerar um exemplo.

Exemplo 10.30. Uma possível busca em profundidade do grafo de fluxo da Fig. 10.45 é ilustrada na Fig. 10.46. Os lados contínuos formam uma árvore; os lados tracejados são os outros lados do grafo de fluxo. A busca em profundidade do grafo de fluxo corresponde a um caminhamento pré-ordem da árvore, $1 \rightarrow 3 \rightarrow 4 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 10$, e, então, de volta para 8 e em seguida para 9. Voltamos para 8 mais uma vez, retraindo para 7, 6 e 4 e, em seguida, adiante para 5. Retraímos de 5 para 4, então, de volta para 3 e 1. De 1, vamos para 2, retraímos de 2 de volta para 1 e temos atravessado toda a árvore em pré-ordem. Notemos que ainda não explicamos como a árvore é selecionada a partir do grafo de fluxo. □

O *ordenamento em profundidade* dos nós é o reverso da ordem na qual visitamos, por último, os nós no caminhamento pré-ordem.

Exemplo 10.31. No Exemplo 10.30, a seqüência completa dos nós visitados à medida que atravessamos a árvore é

$$1, 3, 4, 6, 7, 8, 10, 8, 9, 8, 7, 6, 4, 5, 4, 3, 1, 2, 1.$$

[10] "Escritos por pessoas" não é redundante porque conhecemos vários programas que geram código com "a lógica da macarronada" usando comandos de desvio condicional e incondicional. Não existe nada de errado com isso; a estrutura está na entrada para esses programas.

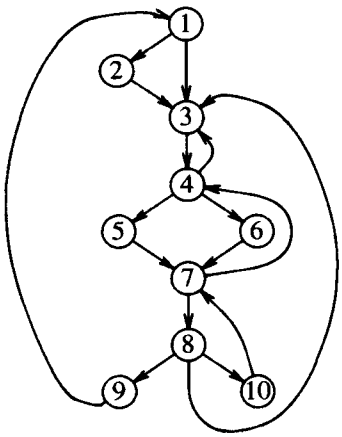


Fig. 10.45. Grafo de fluxo.

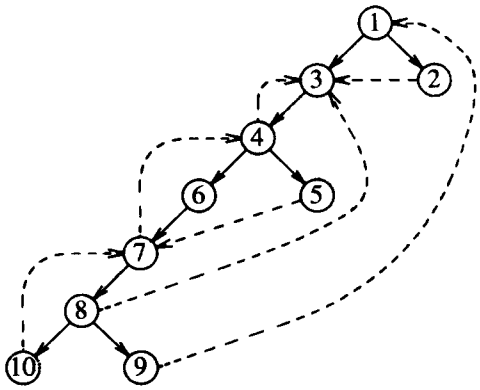


Fig. 10.46. Apresentação da busca em profundidade.

Nesta lista, marcamos a última ocorrência de cada número para obtermos

$$1, 3, 4, 6, 7, 8, \underline{10}, 8, \underline{9}, \underline{8}, \underline{7}, \underline{6}, 4, \underline{5}, \underline{4}, \underline{3}, 1, \underline{2}, \underline{1}$$

O ordenamento em profundidade é a seqüência de números marcados na ordem reversa. Aqui, esta seqüência acontece ser 1, 2 ..., 10. Isto é, inicialmente os nós foram numerados numa ordem em profundidade. □

Fornecemos agora um algoritmo que computa o ordenamento em profundidade de um grafo de fluxo pela construção e travessia de uma árvore enraizada ao nó inicial, tentando construir percursos na árvore na medida do possível. Uma tal árvore é chamada de *árvore de alcance em profundidade* (*aap*).* Este foi o algoritmo usado para construir a Fig. 10.46 a partir da Fig. 10.45.

Algoritmo 10.14 Árvore de alcance em profundidade e ordenamento em profundidade.

Entrada. Um grafo de fluxo G.

Saída. Uma *aap* T de G e um ordenamento dos nós de G.

Método. Usamos o procedimento recursivo *pesquisar*(n) da Fig. 10.47; o algoritmo inicializa todos os nós de G como "não visitados" e, em seguida, chama *pesquisar* (n_0), onde n_0 é o nó inicial. Ao chamarmos *pesquisar*(n), marcamos primeiro n como "visitado", a fim de evitar que n seja adicionado à árvore duas vezes. Usamos i para contar a partir do número dos nós de G até 1, atribuindo números de profundidade $np[n]$ aos nós n, à medida que sigamos em frente. O conjunto de lados T forma a árvore de alcance em profundidade para G, sendo os lados denominados de *lados da árvore*. □

Exemplo 10.32. Consideremos a Fig. 10.47. Fazemos i igual a 10 e chamamos *pesquisar*(1). À linha (2) de *pesquisar* precisamos considerar cada sucessor do nó 1. Suponhamos que consideremos $s = 3$, primeiro. Adicionamos, então, o lado $1 \rightarrow 3$ à árvore e chamamos *pesquisar*(3). Em *pesquisar*(3), adicionamos o lado $3 \rightarrow 4$ a T e chamamos *pesquisar*(4).

Suponhamos que, em *pesquisar*(4), tenhamos escolhido $s = 6$, primeiro. Adicionamos, então, o lado $4 \rightarrow 6$ a T e chamamos *pesquisar*(6). Isto, por sua vez, nos faz adicionar $6 \rightarrow 7$ a T e chamar *pesquisar*(7). O nó 7 possui dois sucessores, 4 e 8. Mas 4 já estava marcado como "visitado" por *pesquisar*(4) e, dessa forma, não fazemos nada quando $s = 4$. Quando $s = 8$, adicionamos o lado $7 \rightarrow 8$ a T e chamamos *pesquisar*(8). Suponhamos que escolhemos, então, $s = 10$. Adicionamos o lado $8 \rightarrow 10$ e chamamos *pesquisar*(10).

Agora, 10 possui um sucessor, 7, mas 7 já está marcado como "visitado", e, conseqüentemente, em *pesquisar*(10), saímos através do passo (6) da Fig. 10.47, fazendo $np[10] = 10$ e $i = 9$. Isto completa a chamada para *pesquisar*(10), e, por conseguinte, retornamos a *pesquisar*(8). Fazemos, agora, $s = 9$, em *pesquisar* (8), adicionamos o lado $8 \rightarrow 9$ a T e chamamos *pesquisar*(9). O único sucessor de 9, o nó, 1, já está como "visitado", fazemos, então, $np[9] = 9$ e $i = 8$. Retornamos, então, para *pesquisar*(8). O último sucessor de 8, o nó 3, é "visitado", e, por isso, não fazemos nada para $s = 3$. A essa altura, tendo considerado todos os sucessores de 8, fazemos, então, $np[8] = 8$ e $i = 7$, retornando a *pesquisar*(7).

Todos os sucessores de 7 foram considerados e, por conseguinte, fazemos $np[7] = 7$ e $i = 6$, retornando a *pesquisar*(6). Similarmente, os sucessores de 6 foram considerados; fazemos, então, $np[6] = 6$ e $i = 5$ e retornamos a *pesquisar*(4). O sucessor 3 de 4 já foi "visitado", mas 5 ainda não; adicionamos, então, $4 \rightarrow 5$ à árvore e chamamos *pesquisar*(5), que não resulta em chamadas posteriores, na medida em que o sucessor 7, de 5, já foi "visitado". Por conseguinte, $np[5] = 5$, i é feito igual a 4 e retornamos a *pesquisar*(4). Tendo completado a consideração dos sucessores de 4, fazemos, então, $np[4] = 4$ e $i = 3$, retornando a *pesquisar*(3). Em seguida, fazemos $np[3] = 3$ e $i = 2$, retornado a *pesquisar*(1).

```
        procedimento pesquisar(n);
        início
(1)         marcar n como "visitado";
(2)         para cada sucessor s de n faça
(3)             se s é "não visitado" então início
(4)                 adicionar lado n → s a T;
(5)                 pesquisar(s)
                fim;
(6)         np[n] := i;
(7)         i := i − 1
        fim;
            /* segue o programa principal */
(8)     T := vazio;    /* conjunto de lados */
(9)     para cada nó n de G faça marcar n "não visitado";
(10)    i := número de nós de G;
(11)    pesquisar(n0)
```

Fig. 10.47. Algoritmo de pesquisa em profundidade.

*Do original em inglês, *depht-first spanning tree (dfst)*. (N. do T.).

Os passos finais são chamar *pesquisar*(2) a partir de *pesquisar* (1), estabelecer $np[2] = 2$ e $i = 1$, retornar a *pesquisar*(1) e fazer $np[1] = 1$ e $i = 0$. Notemos que escolhemos uma numeração de nós tal que $np[i] = i$, mas esta relação não precisa vigorar para um grafo arbitrário ou mesmo para um outro ordenamento em profundidade do grafo da Fig. 10.45. □

Lados numa Representação em Profundidade de um Grafo de Fluxo

Quando construímos uma *aap* para um grafo de fluxo, os lados do grafo de fluxo caem em três categorias.

1. Existem lados que vão de um nó m para um ancestral de m na árvore (possivelmente para o próprio m). A esses lados iremos chamar de *lados de retraentes*. Por exemplo, $7 \rightarrow 4$ e $9 \rightarrow 1$ são lados de retração na Fig. 10.46. É um fato interessante e útil que, se o grafo de fluxo for redutível, os lados retraentes são exatamente os lados refluentes do grafo de fluxo,[11] independentemente da ordem na qual os sucessores sejam visitados no passo (2) da Fig. 10.47. Para qualquer grafo de fluxo, qualquer lado refluente é retraente, apesar de, se o grafo for não redutível, existirem alguns lados retraentes que não sejam lados refluentes.

2. Existem lados, chamados de *lados progressivos*, que vão de um nó m para um descendente próprio de m na árvore. Todos os lados na *aap* são lados progressivos. Não existem outros lados progressivos na Fig. 10.46, mas, por exemplo, se $4 \rightarrow 8$ fosse um lado, estaria nesta categoria.

3. Existem lados $m \rightarrow n$ tais que nem m nem n sejam ancestrais um do outro na *aap*. Os lados $2 \rightarrow 3$ e $5 \rightarrow 7$ são os únicos da Fig. 10.46 que servem de exemplo para tal situação. Chamamos tais lados de *lados cruzados*. Uma importante propriedade dos lados cruzados é que se desenharmos a *aap* de forma que os filhos de um nó sejam colocados da esquerda para a direita na ordem em que são adicionados à árvore, todos os lados cruzados se deslocam da direita para a esquerda.

Deveria ser notado que $m \rightarrow n$ é um lado retraente se e somente se $np[m] \geq np[n]$. Para ver por que, notemos que se m é um descendente de n na *aap*, então *pesquisar* (m) termina antes que *pesquisar* (n) e, então, $np[m] \geq np[n]$. Reciprocamente, se $np[m] \geq np[n]$, *pesquisar* (m) termina antes que *pesquisar* (n) ou $m = n$, mas *pesquisar*(n) precisa ter começado antes de *pesquisar*(m) se houver um lado $m \rightarrow n$ ou, caso contrário, o fato de n ser um sucessor de m teria feito de n descendente de m na *aap*. Por conseguinte, o tempo em que *pesquisar*(m) está ativa é um subintervalo do tempo em que *pesquisar*(n) também o está, donde se conclui que n é um ancestral de m na *aap*.

Profundidade de um Grafo de Fluxo

Existe um importante parâmetro dos grafos de fluxo chamado *profundidade*. Dada uma árvore de alcance em profundidade para um grafo, a profundidade é o maior número de lados retraentes em qualquer percurso livre de ciclos.

Exemplo 10.33. Na Fig. 10.46, a profundidade é 3, já que existe o percurso

$$10 \rightarrow 7 \rightarrow 4 \rightarrow 3$$

com três lados retraentes, mas nenhum percurso livre de ciclos com quatro ou mais lados retraentes. É uma coincidência que o percurso "mais profundo" possua somente lados retraentes; em geral, podemos ter uma mistura de lados retraentes, progressivos e cruzados num percurso "mais profunda". □

Podemos provar que a profundidade nunca é maior do que o que se poderia chamar intuitivamente de profundidade de aninhamento de um laço num grafo de fluxo. Se um grafo de fluxo é redutível, podemos substituir "retraente" por "refluente" na definição de "profundidade", uma vez que os lados retraentes em qualquer *aap* são exatamente os lados refluentes. A noção de profundidade se torna independente da *aap* efetivamente escolhida.

Intervalos

A divisão de um grafo de fluxo em intervalos serve para colocar uma estrutura hierárquica no grafo de fluxo. A estrutura, por sua vez, permite-nos aplicar as regras para a análise de fluxo de dados dirigida pela sintaxe, cujo desenvolvimento começou na Seção 10.5.

Intuitivamente, um "intervalo" num grafo de fluxo é um laço natural mais qualquer estrutura acíclica que se pendure nos nós daquele laço. Uma propriedade importante dos intervalos é que possuem nós de *cabeçalho* que dominam todos os nós no intervalo; isto é, cada intervalo é uma região. Formalmente, dado um grafo de fluxo G, com nó inicial n_0 e um nó n de G, o *intervalo com cabeçalho n*, denotado por $I(n)$, é denotado como se segue:

1. n está em $I(n)$.

2. Se todos os predecessores de algum nó $m \neq n_0$ estão em $I(n)$, então m está em $I(n)$.

3. Nada mais está em $I(n)$.

Podemos, por conseguinte, construir $I(n)$ começando por n e adicionando os nós m pela regra (2). Não importa em que ordem adicionamos os candidatos m porque, uma vez que os predecessores de um nó estejam todos em $I(n)$, permanecem em $I(n)$ e cada candidato será eventualmente adicionado pela regra (2). Eventualmente, não poderão ser adicionados mais nós a $I(n)$ e o conjunto resultante de nós é o intervalo com cabeçalho n.

Partições de Intervalos

Dado qualquer grafo de fluxo G, podemos particionar G em intervalos disjuntos como segue.

Algoritmo 10.15 Análise de intervalos de um grafo de fluxo.

Entrada. Um grafo de fluxo G com nó inicial n_0.

Saída. Uma partição de G num conjunto de intervalos disjuntos.

Método. Para qualquer nó n, computamos $I(n)$ pelo método delineado acima:

$I(n) := \{n\}$;
enquanto existir um nó $m \neq n_0$,
do qual todos os predecessores estejam em $I(n)$ **faça**
$I(n) := I(n) \cup \{m\}$

Os nós particulares que são os cabeçalhos dos intervalos na partição são escolhidos como segue. Inicialmente, nenhum nó é "selecionado".

[11] Relembremos que os lados refluentes de um grafo de fluxo são aqueles cujas cabeças dominam suas caudas.

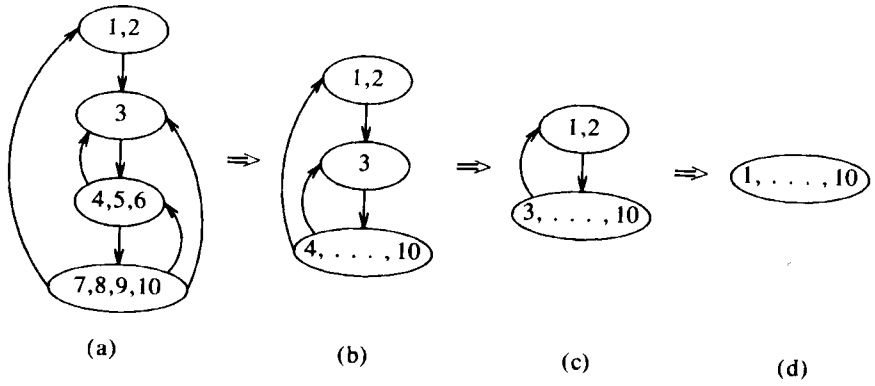


Fig. 10.48. Seqüência de grafos de intervalo.

construir I(n_0) e "selecionar" todos os nós nesse intervalo;
enquanto existir um nó m ainda não "selecionado",
mas com um predecessor selecionado **faça**
construir I(m) e "selecionar" todos os nós desse intervalo. □

Uma vez que um candidato m possua um predecessor p selecionado, m jamais poderá ser adicionado a algum intervalo que não contenha p. Por conseguinte, os candidatos de m se mantêm candidatos até que sejam selecionados para encabeçar seus próprios intervalos. Conseqüentemente, a ordem na qual os cabeçalhos de intervalo m são pinçados no Algoritmo 10.15 não afeta o particionamento final em intervalos. Igualmente, na medida em que todos os nós são alcançáveis a partir de n_0, pode ser mostrado por indução sobre o tamanho do percurso a partir de n_0, para n, que o nó n será eventualmente ou colocado no intervalo de algum outro nó ou irá se tornar o cabeçalho de seu próprio intervalo, mas não ambas as situações. Por conseguinte, o conjunto de intervalos construído pelo Algoritmo 10.15 particiona verdadeiramente G.

Exemplo 10.34. Vamos encontrar a partição de intervalo para a Fig. 10.45. Começamos por construir I(1), porque o nó 1 é o nó inicial. Podemos adicionar 2 a I(1) porque o único predecessor de 2 é 1. No entanto, não podemos adicionar 3 porque o mesmo possui predecessores 4 e 8, que não estão ainda em I(1) e, similarmente, cada outro nó, exceto 1 e 2, possui predecessores ainda não em I(1). Por conseguinte, I(1) = {1,2}.

Podemos agora computar I(3) porque 3 possui alguns predecessores "selecionados", 1 e 2, mas o próprio 3 não está num intervalo. No entanto, nenhum nó pode ser adicionado a I(3), e, dessa forma, I(3) = {3}. Agora, 4 é um cabeçalho porque possui um predecessor, 3, num intervalo. Podemos adicionar 5 e 6 a I(4), porque esses têm somente 4 como predecessor, mas nenhum outro nó pode ser adicionado; por exemplo, 7 possui predecessor 10.

Em seguida, 7 se torna um cabeçalho e podemos adicionar 8 a I(7). Em seguida, podemos adicionar 9 e 10, porque esses têm somente 8 como predecessor. Conseqüentemente, os intervalos na partição da Fig. 10.45 são:

I(1) = {1,2} I(4) = {4,5,6}
I(3) = {3} I(7) = {7,8,9,10} □

Grafos de Intervalos

A partir dos intervalos de um grafo de fluxo G, podemos construir um novo grafo de fluxo I(G) pelas seguintes regras.

1. Os nós de I(G) correspondem aos intervalos da partição de intervalos de G.

2. O nó inicial de I(G) é o intervalo de I(G) que contém o nó inicial de G.

3. Existe um lado, a partir do intervalo I para um intervalo diferente J, se e somente, em G, existir um lado de algum nó em I para o cabeçalho de J. Notemos que não poderia haver um lado entrando em algum nó n de J, de fora de J, que não no do cabeçalho, porque então não haveria forma pela qual n poderia ser adicionado a J no Algoritmo 10.15.

Podemos aplicar o Algoritmo 10.15 e a construção de grafos de intervalo alternativamente, produzindo a seqüência de grafos G, I(G), I(I(G)), ... Eventualmente, chegaremos a um grafo em que cada um dos nós seja um intervalo em si. Esse grafo é chamado de *grafo de fluxo limite* de G. É um fato interessante que um grafo de fluxo seja redutível se e somente se seu grafo de fluxo limite se constituir de um único nó.[12]

Exemplo 10.35. A Fig. 10.48 mostra o resultado da aplicação da construção de intervalos repetidamente à Fig. 10.45. Os intervalos daquele grafo foram fornecidos no Exemplo 10.34 e o grafo de intervalos construído a partir do mesmo, como na Fig. 10.48(a). Notemos que o lado 10 → 7 na Fig. 10.45 não acarreta um lado partindo do nó que representa {7,8,9,10} para si mesmo, na Fig. 10.48(a), porque a construção de grafos de intervalos excluiu explicitamente tais laços. Notemos também que o grafo de fluxo da Fig. 10.45 é redutível porque seu grafo de fluxo limite é um único nó. □

Divisão de Nós

Se atingirmos um grafo de fluxo limite que seja algo que não um único nó, podemos prosseguir adiante somente se dividirmos um ou mais nós. Se um nó n possui k predecessores, podemos substituir n por k nós, n_1, n_2, ..., n_k. O *iésimo* predecessor de n se torna o predecessor de n_i somente, enquanto todos os sucessores de n se tornam sucessores de todo os n_i's.

Se aplicarmos o Algoritmo 10.15 ao grafo resultante, cada n_i possui um único predecessor e, conseqüentemente, irá certamente se tornar parte do intervalo do predecessor. Por conseguinte, uma divisão de nós, mais uma rodada de particionamento de intervalos, resulta num grafo com um número menor de nós. Como conseqüência, a construção de grafos de intervalo, entremeada, quando necessário, com a divisão de nós, terá eventualmente que atingir um grafo com um único nó. O significado desta observação irá se tornar claro na próxima seção, quando projetarmos algoritmos de análise de fluxo de dados que são dirigidos por essas duas operações sobre grafos.

Exemplo 10.36. Consideremos o grafo de fluxo da Fig. 10.49(a), o qual é o seu próprio grafo de fluxo limite. Podemos dividir o nó 2 em 2a e

[12] De fato, historicamente esta é a definição original.

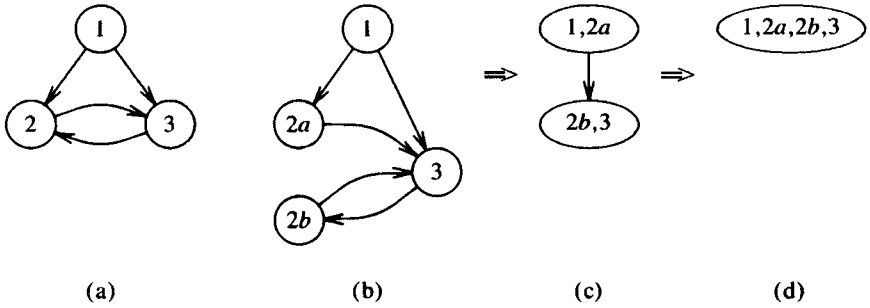


Fig. 10.49. Divisão de nós seguida por particionamento de intervalos.

2*b*, com predecessores 1 e 3, respectivamente. Este grafo é mostrado na Fig. 10.49(b). Se aplicarmos o particionamento de intervalos duas vezes, obtemos a seqüência de grafos mostrada na Fig. 10.49(c) e (d), levando a único nó. □

Análise $T_1 - T_2$

Uma forma conveniente de se alcançar o mesmo efeito que a análise de intervalos é aplicar duas transformações simples aos grafos de fluxo.

T_1: Se *n* é um nó com um laço, isto é, um lado $n \rightarrow n$, remover o lado.

T_2: Se existir um nó *n*, que não o inicial, que tenha um único predecessor *m*, então *m* pode *consumir n* através da remoção de *n* e fazendo-se todos os sucessores de *n* (incluindo *m* possivelmente) serem sucessores de *m*.

Alguns fatos interessantes a respeito das transformações T_1 e T_2 são:

1. Se aplicarmos T_1 e T_2 a um grafo de fluxo *G* em qualquer ordem até que resulte um grafo de fluxo para o qual nenhuma aplicação de T_1 ou de T_2 seja possível, um grafo de fluxo único resulta. A razão está em que um candidato para remoção de laço por T_1 ou consumo por T_2 se mantém um candidato, mesmo que alguma outra aplicação de T_1 ou de T_2 seja feita primeiro.

2. O grafo de fluxo resultante da aplicação exaustiva de T_1 e de T_2 a *G* é o grafo de fluxo limite de *G*. A prova é um tanto sutil e é deixada como exercício. Como conseqüência, uma outra definição de "grafo de fluxo redutível" é que é aquele que pode ser convertido a um único nó por T_1 e T_2.

Exemplo 10.37. Na Fig. 10.50, vemos uma seqüência de reduções T_1 e T_2, começando a partir do grafo de fluxo que é um remanescente da Fig. 10.49(b). Na Fig. 10.50(b), *c* consumiu *d*. Notemos que o laço em *cd* na Fig. 10.50(b) resulta do lado $d \rightarrow c$ na Fig. 10.50(a). O laço é removido por T_1 na Fig. 10.50(c). Notemos, igualmente, que, quando *a* consome *b* na Fig. 10.50(d), os lados provenientes de *a* e *b* para o nó *cd* se tornam um único lado. □

Regiões

Relembremos da Seção 10.5 que uma região num grafo de fluxo é um conjunto de nós *N* que inclui um cabeçalho, que domina todos os outros nós na região. Todos os lados entre nós em *N* estão na região, exceto (possivelmente) para alguns daqueles que estejam no cabeçalho. Por exemplo, cada intervalo é uma região, mas existem regiões que não são intervalos porque, por exemplo, podem omitir alguns nós que um intervalo incluiria ou podem omitir alguns lados de volta para o cabeçalho. Existem, também, regiões maiores do que qualquer intervalo, como iremos ver.

À medida que reduzimos um grafo de fluxo através de T_1 e T_2, a todo instante as seguintes condições são verdadeiras:

1. Um nó representa uma região de *G*.

2. Um lado de *a* até *b* representa um conjunto de lados. Cada um de tais lados vai de algum nó na região representada por *a* até o cabeçalho da região representada por *b*.

3. Cada nó e lado de *G* é representado por exatamente um nó ou lado do grafo corrente.

Para vermos por que essas observações são válidas, notemos primeiro que as mesmas vigoram trivialmente para o próprio *G*. Cada nó é uma região por si mesmo e cada lado representa somente a si mesmo. Suponhamos que apliquemos T_1 a algum nó *n* representando uma região *R*, enquanto o laço $n \rightarrow n$ represente algum conjunto de lados *E*, todos os quais necessariamente têm que entrar no cabeçalho de *R*. Se adicionarmos os lados *E* à região *R*, é ainda uma região, e, por conseguinte, após removermos o lado $n \rightarrow n$, o nó *n* representa *R* e os lados de *E*, o que preserva as condições (1)-(3) acima.

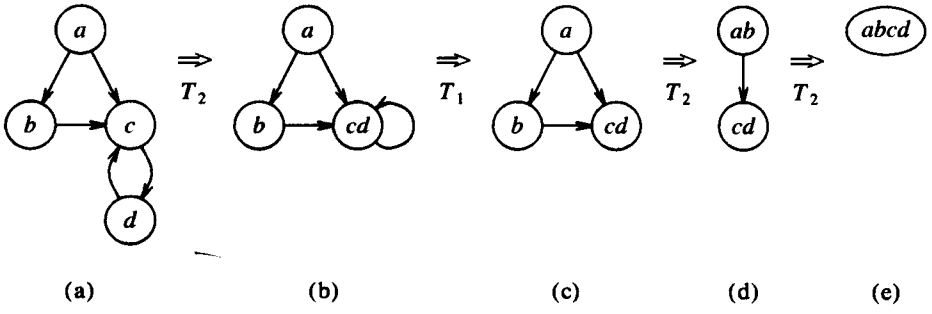


Fig. 10.50. Redução através de T_1 e T_2.

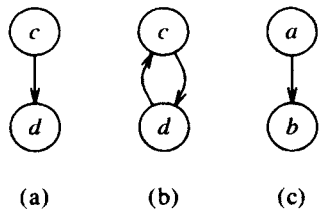


Fig. 10.51. Algumas regiões.

(1) $D(n_0) := \{n_0\}$;
(2) **para** n em $N - \{n_0\}$ **faça** $D(n) := N$;
/* fim da inicialização */
(3) **enquanto** ocorrerem mudanças a qualquer $D(n)$ **faça**
(4) **para** n em $N - \{n_0\}$ **faça**
(5) $D(n) := \{n\} \cup \bigcap_{P \text{ é um predecessor de } n} D(p)$;

Fig. 10.52. Algoritmo de cômputo dos dominadores.

Se, ao invés, usarmos T_2 para consumir o nó b pelo nó a, façamos a e b representarem as regiões R e S, respectivamente. Igualmente, seja E o conjunto de lados representados pelo lado $a \rightarrow b$. Afirmamos que R, S e E, juntas, formam uma região cujo cabeçalho de R domina cada nó em S. Senão, então deverá haver algum percurso para o cabeçalho de S que não termine com um lado de E. Então, o último lado desse percurso teria de ser representado no grafo de fluxo corrente por algum outro lado que entre em b. Mas não poderá haver um tal lado, ou T_2 não poderá ser usada para consumir b.

Exemplo 10.38. O nó rotulado cd na Fig. 10.50(b) representa a região mostrada na Fig. 10.51(a), a qual foi formada ao c ter consumido d. Notemos que o lado $d \rightarrow c$ não é parte da região; na Fig. 10.50(b), aquele lado é representado pelo laço em cd. No entanto, na Fig. 10.50(c), o lado $cd \rightarrow cd$ foi removido e o nó cd agora representa a região mostrada na Fig. 10.51(b).

Na Fig. 10.50(d), o nó cd ainda representa a região da Fig. 10.51(b), enquanto que o nó ab representa a região da Fig. 10.51(c). O lado $ab \rightarrow cd$ na Fig. 10.50(d) representa os lados $a \rightarrow c$ e $b \rightarrow c$ do grafo de fluxo original na Fig. 10.50(a). Ao aplicarmos T_2 atingindo a Fig. 10.50(e), o nó restante representa todo o grafo de fluxo, a Fig. 10.50(a). □

Deveríamos observar que a propriedade de redução de T_1 e T_2 mencionada acima também vigora para a análise de intervalos. Deixamos como exercício o fato de que, à medida que construímos I(G), I(I(G)) e assim por diante, cada nó em cada um desses grafos representa uma região e cada lado, um conjunto de lados satisfazendo a propriedade (2) acima.

Encontrando os Dominadores

Encerramos esta seção com um algoritmo eficiente para um conceito que temos usado freqüentemente, e que continuaremos usando, no desenvolvimento da teoria dos grafos de fluxo e análise de fluxo de dados. Daremos um algoritmo simples para computar os dominadores de cada nó n num grafo de fluxo, baseados no princípio de que, se $p_1, p_2, \ldots, p_k$ são todos predecessores de n e $d \neq n$, então, d *domina* n se e somente se d *domina* p_i, para cada i. O método é semelhante à análise de fluxo de dados para adiante, tendo a interseção como o operador de confluência (por exemplo, expressões disponíveis), na qual tomamos uma aproximação do conjunto de dominadores de n e a refinamos repetidamente, visitando todos os nós a cada vez.

Neste caso, a aproximação inicial que escolhemos possui o nó inicial dominado somente pelo nó inicial e tudo dominando tudo para além do nó inicial. Intuitivamente, a razão pela qual esse enfoque funciona está em que os candidatos a dominadores são proscritos somente quando encontramos um percurso que prove, digamos, que m *domina* n seja falso. Senão pudermos encontrar um tal percurso, a partir do nó inicial até o nó n, evitando m, então m é realmente um dominador de n.

Algoritmo 10.16 Encontrando os dominadores.

Entrada. Um grafo de fluxo G com conjunto de nós N, conjunto de lados E e nó inicial n_0.

Saída. A relação *domina*.

Método. Computamos $D(n)$, o conjunto de dominadores de n, iterativamente, pelo procedimento da Fig. 10.52. Ao final, d está em $D(n)$ se e somente se d *domina* n. O leitor pode fornecer os detalhes relacionados a como as mudanças em $D(n)$ são detectadas; o Algoritmo 10.2 servirá como modelo.

Pode-se mostrar que $D(n)$ computado à linha (5) da Fig. 10.52 é sempre um subconjunto do $D(n)$ corrrente. Como $D(n)$ não pode diminuir indefinidamente, terminaremos eventualmente o laço **enquanto.** A prova que, após a convergência, $D(n)$ é o conjunto de dominadores de n é deixada para o leitor interessado. O algoritmo da Fig. 10.52 é razoavelmente eficiente, na medida em que $D(n)$ possa ser representado por um vetor de *bits* e as operações sobre conjuntos da linha (5) possam feitas com o **e** e o **ou** lógicos. □

Exemplo 10.39. Vamos retornar ao grafo de fluxo da Fig. 10.45 e suponhamos que no laço **para** da linha (4) os nós sejam visitados em ordem numérica. O nó 2 possui somente 1 como predecessor; dessa forma, $D(2) =: \{2\} \cup D(1)$. Como 1 é o nó inicial, a $D(1)$ foi atribuído $\{1\}$ à linha (1). Por conseguinte, $D(2)$ é estabelecido com $\{1,22\}$ à linha (5).

Em seguida, o nó 3, com predecessores 1, 2, 4, e 8 é considerado. A linha (5) nos dá $D(3) = \{3\} \cup (\{1\} \cap \{1,2\} \cap \{1,2, \ldots, 10\}) = \{1,3\}$. Os cômputos restantes são

$$
\begin{aligned}
D(4) &= \{4\} \cup (D(3) \cap D(7)) = \{4\} \cup (\{1,3\} \cap \{1,2,\ldots,10\}) = \{1,3,4\} \\
D(5) &= \{5\} \cup D(4) = \{5\} \cup \{1,3,4\} = \{1,3,4,5\} \\
D(6) &= \{6\} \cup D(4) = \{6\} \cup (\{1,3,4\} = \{1,3,4,6\} \\
D(7) &= \{7\} \cup (D(5) \cap D(6) \cap D(10)) \\
&= \{7\} \cup (\{1,3,4,5\} \cap \{1,3,4,6\} \cap \{1,2,\ldots,10\}) = \{1,3,4,7\} \\
D(8) &= \{8\} \cup D(7) = \{8\} \cup \{1,3,4,7\} = \{1,3,4,7,8\} \\
D(9) &= \{9\} \cup D(8) = \{9\} \cup \{1,3,4,7,8\} = \{1,3,4,7,8,9\} \\
D(10) &= \{10\} \cup D(8) = \{10\} \cup \{1,3,4,7,8\} = \{1,3,4,7,8,10\}
\end{aligned}
$$

A segunda passagem através do laço enquanto não produz mudanças, de forma que os valores acima produzem a relação *domina*. □

10.10 ALGORITMOS DE FLUXO DE DADOS EFICIENTES

Nesta seção, consideraremos duas formas de usar a teoria dos grafos de fluxo para acelerar a análise de fluxo de dados. A primeira é uma aplicação do ordenamento em profundidade, a fim de reduzir o número de passagens que os algoritmos iterativos da Seção 10.6 fazem e a segunda usa intervalos ou as transformações T_1 e T_2 para generalizar a abordagem dirigida pela sintaxe da Seção 10.5.

Ordenamento em Profundidade nos Algoritmos Iterativos

Em todos os problemas estudados até então, tais como as definições incidentes, expressões disponíveis ou variáveis vivas, qualquer evento

de significância para um nó será propagado até o mesmo ao longo de um percurso acíclico. Por exemplo, se uma definição *d* está em *entrada*[*B*], então existe algum percurso acíclico a partir do bloco que contém *d* até *B*, tal que *d* está em todos os conjuntos *entrada* e *saída* ao longo daquele percurso. Similarmente, se uma expressão `x+y` não está disponível à entrada para um bloco *B*, há, então, algum percurso acíclico que demonstra esse fato; ou o percurso é partir de um nó inicial e não inclui enunciados que matem ou gerem `x+y`, ou o percurso é de um bloco que mate `x+y` e ao longo do percurso não existe geração subseqüente de `x+y`. Finalmente, para as variáveis vivas, se `x` está viva à saída do bloco *B*, então existe um percurso acíclico a partir de *B* até um uso de `x`, ao longo do qual não existem definições de `x`.

O leitor deveria verificar que, em cada um desses casos, os percursos com ciclos não adicionam nada. Por exemplo, se um uso de `x` é atingido a partir do final do bloco *B*, ao longo de um percurso com um ciclo, podemos eliminar aquele ciclo de forma a encontrar um percurso mais curto ao longo do qual o uso de `x` ainda é atingido a partir de *B*.

Se todas as informações úteis se propagam ao longo de percursos acíclicos, temos a oportunidade de costurar a ordem na qual visitamos os nós no algoritmo de fluxo de dados iterativos de forma que, após umas relativamente poucas passagens através dos nós, podemos nos assegurar que as informações passaram ao longo de todos os percursos acíclicos. Em particular, as estatísticas capturadas em Knuth [1971b] mostram que os grafos de fluxo típicos têm uma *profundidade de intervalo* muito baixa, a qual é o número de vezes que se deve aplicar o particionamento de intervalos a fim de se atingir o grafo de fluxo limite; foi encontrada uma média de 2.75. Sobretudo, pode ser mostrado que a profundidade de intervalo de um grafo de fluxo nunca é menos do que o temos chamado de "profundidade", isto é, o número máximo de lados retraentes em qualquer percurso acíclico. (Se o grafo de fluxo não for redutível, a profundidade pode depender da árvore de alcance de profundidade escolhida).

Relembrando nossa discussão a respeito da árvore de alcance de profundidade na seção anterior, notamos que, se $a \rightarrow b$ for um lado, então, o número de profundidade de *b* é menor do que aquele de *a* somente quando o lado for retraente. Por conseguinte, substituamos a linha (5) da Fig. 10.26, que nos diz para visitar cada bloco *B* do grafo de fluxo para o qual estamos computando as definições incidentes, por:

para cada bloco *B* em ordem de profundidade **faça**

Suponhamos ter um percurso ao longo do qual uma definição *d* se propague, tal como

$$3 \rightarrow 5 \rightarrow 19 \rightarrow 35 \rightarrow 16 \rightarrow 23 \rightarrow 45 \rightarrow 4 \rightarrow 10 \rightarrow 17$$

onde os inteiros representam os números de profundidade dos blocos ao longo do percurso. Então, na primeira vez que caminharmos através do laço das linhas (5)−(9), na Fig. 10.26, *d* será propagado de *saída*[3] para *entrada*[5], para *saída*[5] e assim por diante até *saída*[35]. Não irá atingir *entrada*[16] nessa rodada, porque, como 16 precede 35, já computamos *entrada*[16] no tempo em que *d* foi colocado em *saída*[35]. No entanto, a próxima vez que rodarmos através do laço das linhas (5)−(9), quando computarmos *entrada*[16], *d* será incluído porque está em *saída*[35]. A definição *d* também irá ser propagada até *saída*[16], *entrada*[23] e assim por diante, até *saída*[45], onde deverá aguardar porque *entrada*[4] já foi computado. A terceira passagem, *d* viaja para *entrada*[4], *saída*[4], *entrada*[10], *saída*[10] e *entrada*[17], de forma que, após três passagens, estabelecemos que *d* atinge o bloco 17.[13]

Não deveria ser difícil extrair o princípio geral a partir deste exemplo. Se usamos a ordem em profundidade da Fig. 10.26, o número de passagens necessitadas para propagar qualquer definição incidente ao longo de qualquer percurso acíclico não é maior do que um mais o número de lados ao longo daquele percurso, o qual vai de um bloco de numeração mais alta até um bloco de numeração mais baixa. Aqueles lados são exatamente os lados retraentes, de forma que o número de passagens necessitadas é um mais a profundidade. Naturalmente, o Algoritmo 10.2 não detecta o fato de que todas as definições tenham atingido as localizações, onde quer que as possam atingir, somente com uma passagem a mais, de forma que o limite superior do número de passagens tomadas por aquele algoritmo, com ordenamento em profundidade dos blocos é, de fato, dois mais a profundidade, ou 5, se crermos que os resultados de Knuth [1971b] sejam típicos.

A ordem em profundidade é também vantajosa para as expressões disponíveis (Algoritmo 10.3) ou para qualquer problema de fluxo de dados que resolvermos através da propagação na direção para frente. Para problemas como variáveis vivas, onde propagamos para trás, a mesma média de cinco passagens pode ser atingida se escolhermos o inverso da ordem em profundidade. Por conseguinte, podemos propagar o uso de uma variável no bloco 17 de volta ao longo do percurso

$$3 \rightarrow 5 \rightarrow 19 \rightarrow 35 \rightarrow 16 \rightarrow 23 \rightarrow 45 \rightarrow 4 \rightarrow 10 \rightarrow 17$$

em uma passagem para *entrada*[4], onde precisamos esperar pela próxima passagem a fim de atingir *saída*[45]. Na segunda passagem, o mesmo atinge *entrada*[16] e, na terceira, vai de *saída*[35] para *saída*[3]. Em geral, um número de passagens igual à profundidade mais um é suficiente para conduzir o uso de uma variável para trás, ao longo de qualquer percurso acíclico, se escolhermos o reverso da ordem em profundidade para visitar os nós numa passagem, porque, então, os usos se propagam ao longo de qualquer seqüência decrescente em uma única passagem.

Análise de Fluxo de Dados Baseada em Estruturas

Com um pouco mais de esforço, podemos implementar algoritmos que visitam os nós (e aplicam equações de fluxo de dados) um número de vezes não maior do que a profundidade de intervalo do grafo de fluxo e freqüentemente o nó médio será visitado um número de vezes ainda menor do que esse número. Se o esforço extra representa um verdadeiro ganho de tempo, ainda não foi firmemente estabelecido, mas uma técnica como esta, baseada na análise de intervalos, tem sido usada em diversos compiladores. Sobretudo, as idéias expostas aqui se aplicam aos algoritmos de fluxo de dados dirigidos pela sintaxe para todos os tipos de enunciados de controle estruturados, não somente o **if**...**then** e o **do**...**while**, discutidos na Seção 10.5, sendo que esses algoritmos também apareceram em vários compiladores.

Iremos basear nosso algoritmo na estrutura induzida nos grafos de fluxo pelas transformações T_1 e T_2. Como na Seção 10.5, estamos envolvidos com as definições que são geradas e mortas à medida que o controle flui através de uma região. Diferentemente das regiões definidas pelos enunciados *if* e *while*, uma região geral pode ter múltiplas saídas, de forma que para cada bloco *B* na região *R* iremos computar os conjuntos $geradas_{R,B}$ e $mortas_{R,B}$ das definições geradas e mortas, respectivamente, ao longo de percursos dentro da região, a partir do cabeçalho até o final do bloco *B*. Esses conjuntos serão usados para definir uma *função de transferência* $trans_{R,B}(S)$, que informa para qualquer conjunto *S* de definições, que conjunto atinge o final do bloco *B* viajando ao longo de percursos completamente dentro de *R*, dado que todas e somente as definições em *S* atingem o cabeçalho de *R*.

Como vimos nas Seções 10.5 e 10.6, as definições incidentes ao fim do bloco *B* caem em duas classes.

1. Aquelas que são geradas dentro de *R* e se propagam até o fim de *B*, independentemente de *S*.

2. Aquelas que não geradas em *R*, mas, também, não são mortas ao longo de algum percurso a partir do cabeçalho de *R* até o fim de *B* e, por conseguinte, estão em $trans_{R,B}(S)$ se e somente se estiverem em *S*.

[13] A definição *d* também atinge *saída*[17], mas isso é irrelevante para a discussão em questão.

Conseqüentemente, podemos escrever *trans* sob a forma:

$$trans_{R,B}(S) = geradas_{R,B} \cup (S - mortas_{R,B})$$

O coração do algoritmo é uma forma de se computar $trans_{R,B}$ para regiões progressivamente maiores definidas por alguma decomposição-(T_1, T_2) de um grafo de fluxo. Para o momento, assumimos que o grafo de fluxo seja redutível, apesar de uma simples modificação permitir que o algoritmo funcione para grafos não redutíveis, igualmente.

A base é uma região consistindo de um único bloco, B. Aqui, a função de transferência da região é a função de transferência do bloco em si, uma vez que uma definição atinge o fim de um bloco se e somente se ou é gerada pelo bloco ou está no conjunto S e não é morta. Isto é,

$$geradas_{B,B} = geradas[B]$$
$$mortas_{B,B} = mortas[B]$$

Vamos agora considerar a construção de uma região R por T_2; isto é, R é formada quando R_1 consome R_2, como sugerido na Fig. 10.53. Primeiro, notemos que dentro da região R não existem lados de R_2 de volta à R_1, já que qualquer lado partindo de R_2 para o cabeçalho de R_1 não é parte de R. Por conseguinte, qualquer percurso totalmente dentro de R passa (opcionalmente) através de R_1 primeiro e, então, através de R_2 (opcionalmente), mas não pode retornar a R_1. Notemos igualmente que o cabeçalho de R é o cabeçalho de R_1. Podemos concluir que, dentro de R, R_2 não afeta a função de transferência dos nós de R_1; isto é,

$$geradas_{R,B} = geradas_{R_1,B}$$
$$mortas_{R,B} = mortas_{R_1}$$

para todo B em R_1.

Para B em R_2, uma definição pode atingir o final de B se qualquer uma das condições seguintes estiver em vigor.

1. A definição é gerada dentro de R_2.

2. A definição é gerada dentro de R_1, atinge o fim de algum predecessor do cabeçalho de R_2 e não é morta ao ir do cabeçalho de R_2 para B.

3. A definição está no conjunto S disponível no cabeçalho de R_1, não sendo morta ao ir para algum predecessor do cabeçalho de R_2 e não sendo morta ao ir do cabeçalho de R_2 para B.

Por conseguinte, as definições que atingem o fim daqueles blocos em R_1 que sejam predecessores do cabeçalho de R_2 desempenham um papel especial. Em essência, vemos o que acontece a um conjunto S que entre no cabeçalho de R_1 à medida que suas definições tentem atingir o cabeçalho de R_2, através de um de seus predecessores. O conjunto de definições que atingem um dos predecessores do cabeçalho de R_2 se torna o conjunto de entrada para R_2 e aplicamos as funções de transferência para R_2 àquele conjunto.

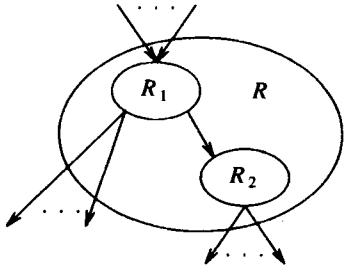


Fig. 10.53. Construção de regiões por T_2.

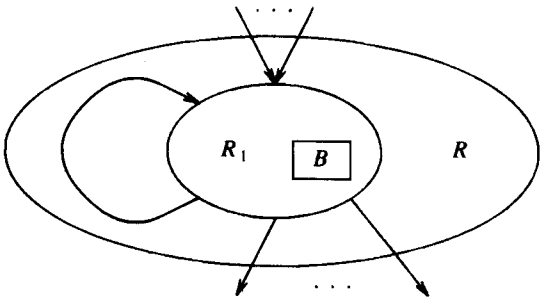


Fig. 10.54. Construção de regiões através de T_1.

Por conseguinte, seja G a união de $geradas_{R_1,P}$ para todos os predecessores P do cabeçalho de R_2 e seja M a interseção de $mortas_{R_1,P}$, para todos os predecessores P. Então, se S é o conjunto de definições que atingem o cabeçalho de R_1, o conjunto de definições que atingem o cabeçalho de R_2 ao longo de percursos localizados completamente dentro de R é $G \cup (S - M)$. Conseqüentemente, a função de transferência em R para aqueles blocos B em R_2 pode ser computada por

$$geradas_{R,B} = geradas_{R_2,B} \cup (G - mortas_{R_2,B})$$
$$mortas_{R,B} = mortas_{R_2,B} \cup (M - geradas_{R_2,B})$$

Consideremos, em seguida, o que acontece quando uma região R é construída a partir de uma região R_1 usando a transformação T_1. A situação geral é mostrada na Fig. 10.54; notemos que R consiste em R_1 mais alguns lados refluentes para o cabeçalho de R_1 (o qual é também o cabeçalho de R, naturalmente). Um percurso indo através do cabeçalho duas vezes deveria ser cíclico e, como argumentamos anteriormente, não precisa ser considerado. Por conseguinte, todas as definições geradas ao final do bloco B são geradas em uma de duas formas.

1. A definição é gerada dentro de R_1 e não precisa ter os lados refluentes incorporados em R a fim de atingir o final de B.

2. A definição é gerada em algum lugar dentro de R, atinge um predecessor do cabeçalho, segue um lado refluente e não é morta indo do cabeçalho para B.

Se fizermos G ser a união de $geradas_{R_1,P}$ para todos os predecessores do cabeçalho em R, então,

$$geradas_{R,B} = geradas_{R_1,B} \cup (G - mortas_{R_1,B})$$

Uma definição é morta indo do cabeçalho para B se e somente se for morta ao longo de todos os percursos acíclicos, de forma que os lados refluentes incorporados em R não fazem com que mais definições sejam mortas. Isto é,

$$mortas_{R,B} = mortas_{R_1,B}$$

Exemplo 10.40. Vamos reconsiderar o grafo de fluxo da Fig. 10.50, cuja decomposição-(T_1, T_2) é mostrada na Fig. 10.55, tendo as regiões da decomposição recebido nomes. Mostramos, também, na Fig. 10.56, alguns vetores de *bits* hipotéticos que representam três definições e se as mesmas são geradas ou mortas por cada um dos blocos da Fig. 10.55.

Começando de fora para dentro, notamos que, para regiões de um único nó, que chamamos A, B, C e D, *geradas* e *mortas* são dados pela tabela na Fig. 10.56. Podemos, então, prosseguir até a região R, que é formada quando C consome D através de T_2. Seguindo-se as regras para T_2 acima, notamos que *geradas* e *mortas* não mudam para C, isto é,

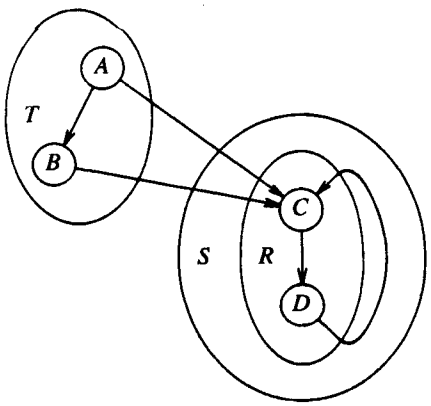


Fig. 10.55. Decomposição de um grafo de fluxo.

$$geradas_{R,C} = geradas_{C,C} = 000$$
$$mortas_{R,C} = mortas_{C,C} = 010$$

Para o nó D, temos que encontrar na região C a união de *geradas* para todos os predecessores do cabeçalho da região D. Naturalmente, o cabeçalho da região D é o nó D e existe somente um predecessor daquele nó na região C, nominalmente, o nó C. por conseguinte,

$$geradas_{R,D} = geradas_{D,D} \cup (geradas_{C,C} - mortas_{D,D}) = 001 + (000 - 000) = 001$$
$$mortas_{R,D} = mortas_{D,D} \cup (mortas_{C,C} - geradas_{D,D}) = 000 + (010 - 001) = 010$$

Construímos, agora, a região S a partir da região R através de T_1. Os conjuntos *mortas* não mudam, de forma que temos

$$mortas_{S,C} = mortas_{R,C} = 010$$
$$mortas_{S,C} = mortas_{R,D} = 010$$

Para computar os conjuntos *geradas* para S, notamos que o único lado refluente para o cabeçalho de S que é incorporado indo-se de R para S é o lado $D \rightarrow C$. Por conseguinte,

$$geradas_{S,C} = geradas_{R,C} \cup (geradas_{R,D} - mortas_{R,C}) = 000 + (001 - 010) = 001$$
$$geradas_{S,D} = geradas_{R,D} \cup (geradas_{R,D} - mortas_{R,D}) = 001 + (001 - 010) = 001$$

O cômputo para a região T é análogo àquele para a região R e obtemos

$$geradas_{T,A} = 100$$
$$mortas_{T,A} = 010$$
$$geradas_{T,B} = 010$$
$$mortas_{T,B} = 101$$

BLOCO	*GERADAS*	*MORTAS*
A	100	010
B	010	101
C	000	010
D	001	000

Fig. 10.56. Informações de *geradas* e *mortas* para os blocos da Fig. 10.55.

Finalmente, computamos *geradas* e *mortas* para a região U, todo o grafo de fluxo. Como U é construído quando T consome S pela transformação T_2, os valores de *geradas* e *mortas* para os nós A e B não mudam a partir do que já foi fornecido acima. Para C e D, notamos que o cabeçalho de S, nó C, possui dois predecessores na região T, nominalmente, A e B. Por conseguinte, computamos

$$G = geradas_{T,A} \cup geradas_{T,B} = 110$$
$$M = mortas_{T,A} \cap mortas_{T,B} = 000$$

Podemos, então, computar

$$geradas_{U,C} = geradas_{S,C} \cup (G - mortas_{S,C}) = 101$$
$$mortas_{U,C} = mortas_{S,C} \cup (M - geradas_{S,C}) = 010$$
$$geradas_{U,D} = geradas_{S,D} \cup (G - mortas_{S,D}) = 101$$
$$mortas_{U,D} = mortas_{S,D} \cup (M - geradas_{S,D}) = 010$$ □

Tendo computado $geradas_{U,B}$ e $mortas_{U,B}$ para cada bloco B, onde U é a região que consiste em todo o grafo de fluxo, computamos essencialmente *saída*[B] para cada bloco B. Isto é, se examinarmos a definição de $trans_{U,B}(S) = geradas_{U,B} \cup (S - mortas_{U,B})$, notaremos que $trans_{U,B}(\varnothing)$ é exatamente *saída*[B]. Mas $trans_{U,B}(\varnothing) = geradas_{U,B}$. Por conseguinte, a feitura do algoritmo de definições incidentes baseado em estruturas consiste em usar os conjuntos *geradas* e *saída* e computar os conjuntos *entrada* tomando a união dos conjuntos *saída* dos predecessores. Esses passos são sumarizados no algoritmo seguinte.

Algoritmo 10.17 Definições incidentes baseadas em estruturas.

Entrada. Um grafo de fluxo redutível G e conjuntos de definições *geradas*[B] e *mortas*[B] para cada bloco B de G.

Saída. Entrada[B] para cada bloco B.

Método.

1. Encontrar a decomposição-(T_1, T_2) para G.

2. Para cada região R na decomposição, de dentro para fora, computar $geradas_{R,B}$ e $mortas_{R,B}$ para cada bloco B em R.

3. Se U é o nome da região que consiste em todo o grafo, então, para cada bloco B, fazer *entrada*[B] igual à união, sobre todos os predecessores P do bloco B, de $geradas_{U,P}$. □

Algumas Acelerações do Algoritmo Baseado em Estruturas

Primeiro, note-se que se tivermos uma função de transferência $G \cup (S - M)$, a função não é modificada se removermos de M alguns dos membros de G. Conseqüentemente, ao aplicarmos T_2, em vez de usar as fórmulas

$$geradas_{R,B} = geradas_{R_2,B} \cup (G - mortas_{R_2,B})$$
$$mortas_{R,B} = mortas_{R_2,B} \cup M$$

podemos substituir a segunda por

$$mortas_{R,B} = mortas_{R_2,B} \cup M$$

economizando uma operação para cada bloco da região R_2.

Outra idéia útil é notar que o único instante em que aplicamos T_1 é após termos primeiro consumido alguma região R_2 por R_1 e existirem alguns lados refluentes partindo de R_2 para o cabeçalho de R_1. Em lugar de primeiro fazer as mudanças em R_1 e R_2 devido à operação T_1, podemos combinar os dois conjuntos de mudanças se fizermos o seguinte.

1. Usando a regra T_2, computamos a nova função de transferência para aqueles nós em R_2 que sejam predecessores do cabeçalho de R_1.

2. Usando a regra T_1, computamos a nova função de transferência para todos os nós de R_1.

3. Usando a regra T_2, computamos a nova função de transferência para todos os nós de R_2. Notemos que a realimentação devida à aplicação de T_1 atingiu os predecessores de R_2 e é passada a todos os nós de R_2 pela regra T_2; não há necessidade de se aplicar a regra T_1 a R_2.

Tratando Grafos de Fluxo Não-Redutíveis

Se a redução (T_1, T_2) de um grafo pára a um grafo de fluxo limite que não seja um único nó, precisamos realizar uma divisão de nós. Dividir um nó de um grafo de fluxo limite corresponde a duplicar a região inteira representada por aquele nó. Por exemplo, na Fig. 10.57, sugerimos o efeito que a divisão de nós poderia ter sobre o grafo de fluxo original de nove nós que foi particionado por T_1 e T_2 em três regiões conectadas por alguns lados.

Como mencionado na seção anterior, através da alternância das divisões com as seqüências de reduções, estamos garantidos em reduzir o grafo de fluxo a um único nó. O resultado das divisões é que alguns nós do grafo original terão mais de uma cópia na região representada pelo grafo de um único nó. Podemos aplicar o Algoritmo 10.17 a esta região com pouca mudança. A única diferença está em que, quando dividimos um nó, os conjuntos *geradas* e *mortas*, para os nós do grafo original na região representada pelo nó dividido, precisam ser duplicados. Por exemplo, o que quer que sejam os valores de *geradas* e *mortas* para os nós na região de dois nós da Fig. 10.57 à esquerda, se torna *geradas* e *mortas* para cada um dos nós correspondentes em ambas as regiões de dois nós à direita. No passo final, quando computamos os conjuntos *entrada* para todos os nós, aqueles nós do grafo original que têm vários representantes na região final possuem seus conjuntos *entrada* computados tomando-se a união dos conjuntos *entrada* de todos os seus representantes.

No pior caso, a divisão de nós poderia exponenciar o número total de nós representados por todas as regiões. Por conseguinte, se esperamos que muitos grafos de fluxo sejam não-redutíveis, provavelmente não deveríamos usar métodos baseados em estruturas. Felizmente, os grafos de fluxo não-redutíveis são suficientemente raros de forma que podemos geralmente ignorar o custo da divisão de nós.

10.11 UMA FERRAMENTA PARA A ANÁLISE DE FLUXO DE DADOS

Como apontamos anteriormente, existem fortes similaridades entre os vários problemas de fluxo de dados estudados. As equações de fluxo de dados da Seção 10.6 foram vistas serem distinguidas por:

1. A função de transferência usada, a qual em cada caso estudado era da forma $f(X) = A \cup (X - B)$. Por exemplo, A = *mortas* e B = *geradas* para as definições incidentes.

2. O operador de confluência, o qual em todos os casos até então tem sido ou a união ou a interseção.

3. A direção de propagação da informação: para frente ou para trás.

Como essas distinções não são grandes, não deveria ser surpreendente que todos esses problemas pudessem ser tratados de forma unificada. Uma tal abordagem foi descrita em Kildall [1973] e uma ferramenta para simplificar a implementação de problemas de fluxo de dados foi implementada por Kildall e usada pelo mesmo em vários projetos de compiladores. Não foi vista em uso disseminado provavelmente porque a quantidade de trabalho economizada pelo sistema não era tão grande quanto a economizada por ferramentas como um gerador de analisadores sintáticos. No entanto, devemos estar informados do que pode ser feito não somente porque isto sugere uma simplificação para implementadores de compiladores otimizantes, mas também porque auxilia a unificar as várias idéias que examinamos até então neste capítulo. Sobretudo, esta seção sugere como podem ser desenvolvidas estratégias mais poderosas de análise de fluxo de dados, aquelas que providenciam informações mais precisas do que os algoritmos mencionados até então.

Estruturas de Análise de Fluxo de Dados

Iremos descrever as estruturas que modelam problemas de propagação para adiante. Se considerarmos somente o tipo de solução iterativa para os problemas de análise de fluxo de dados, a direção não faz diferença; podemos reverter a direção dos lados e fazer alguns ajustes menores para levar em conta o nó inicial e então tratar o problema para trás como se fosse para adiante. Os algoritmos baseados em estruturas são um tanto diferentes; os problemas para adiante e para trás não são resolvidos exatamente da mesma forma porque o reverso de um grafo de fluxo redutível não precisa ser redutível. No entanto, o tratamento de problemas para trás será deixado como um exercício e iremos nos restringir somente aos problemas para adiante.

Uma *estrutura de análise de fluxo* de dados consiste em:

1. Um conjunto V de *valores* a serem propagados. Os valores de *entrada* e *saída* são membros de V.

2. Um conjunto F de *funções de transferência* de V para V.

3. Uma operação *binária de reunião* $\wedge$, sobre V, para representar o operador de confluência.

Exemplo 10.41. Para as definições incidentes, V consiste em todos os subconjuntos dos conjuntos de definições no programa. O conjunto F

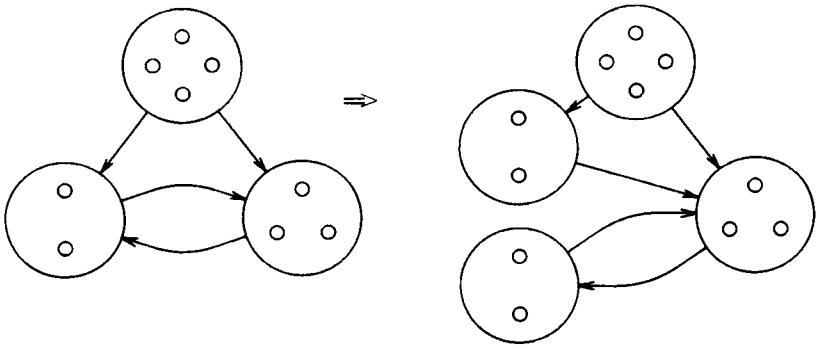


Fig. 10.57. Dividindo um grafo de fluxo não-redutível.

é o conjunto de todas as funções da forma $f(X) = A \cup (X - B)$, onde A e B são os conjuntos de definições, isto é, membros de V; A e B são o que chamamos de *geradas* e *mortas*, respectivamente. Finalmente, a operação $\wedge$ é a união.

Para as expressões disponíveis, V consiste em todos os subconjuntos de expressões computadas pelo programa e F é o conjunto de expressões da mesma forma que acima, mas onde A e B sejam agora conjuntos de expressões. A operação de reunião é a interseção, naturalmente. □

Exemplo 10.42. A abordagem de Kildall não está limitada a exemplos simples com os quais temos até então lidado, apesar da complexidade aumentar, tanto em termos do tempo de computação quanto da dificuldade intelectual. Os exercícios sugerem um exemplo muito poderoso, onde as informações computadas de fluxo de dados nos informam, em essência, todos os pares de expressões que têm mesmo valor a um ponto. No entanto, iremos sentir um pouco do sabor deste exemplo fornecendo um método para informar que variáveis têm valores constantes, de uma forma que captura mais informações do que as definições incidentes. Nossa nova estrutura compreende, por exemplo, que, quando x é definida através de d=x:=x+1 e que x tinha um valor constante antes da atribuição, a mesma a consideraria não constante somente dali para a frente.

Em contraste, usando as definições incidentes para a propagação de constantes, veríamos que o enunciado d era uma possível definição de x e assumiríamos, por conseguinte, que x não tinha um valor constante. Naturalmente, em uma passagem, o lado direito de d:x:=x+1 poderia ser substituído por uma constante e, então, outra rodada de propagação de constantes detectaria que os usos de x, definidos em, d eram, efetivamente, usos de uma constante.

Na nova estrutura, o conjunto V é o conjunto de todos os mapeamentos a partir das variáveis do programa num conjunto particular de valores. Esse conjunto de valores consiste em

1. Todas as constantes.

2. O valor *não-constante*, o qual significa que a variável em questão foi determinada como não tendo um valor constante. O valor *não-constante* deveria ser atribuído a uma variável x se, digamos, durante a análise de fluxo de dados descobríssemos dois percursos ao longo dos quais os valores 2 e 3, respectivamente, fossem atribuídos a x ou um percurso ao longo do qual a definição prévia de x fosse um enunciado de leitura.

3. O valor *indefinido*, que significa que nada pode ser afirmado a respeito da variável em questão, presumivelmente porque não foi descoberta antes na execução da análise de fluxo de dados nenhuma definição da variável que atingisse o ponto em questão.

Notemos que *não-constante* e *indefinido* não são os mesmos valores; são essencialmente opostos. O primeiro diz que vimos tantas formas sob as quais uma variável poderia ser definida que sabemos não se tratar de uma constante. A segunda diz que vimos tão pouco a respeito de uma variável que, de todo, não podemos dizer nada.

A operação de reunião é definida pela seguinte tabela. Fazemos μ e v serem dois membros de V; isto é, μ e v, ambas, mapeiam cada variável em uma constante, em *indefinido* ou em *não-constante*. A função é $\rho = \mu \wedge v$ é, então, definida na Fig. 10.58, onde damos o valor de $\rho(x)$ em termos dos valores de $\mu(x)$ e de $v(x)$ para cada variável x. Naquela tabela, c é uma constante arbitrária e d é uma outra constante, inquestionavelmente diferente de c. Por exemplo, se $\mu(x) = c$ e $v(x) = d$, uma constante diferente, aparentemente x toma os valores de c e d ao longo de dois diferentes percursos e, à confluência daqueles percursos, x não possui valor constante; daí a escolha $\rho(x)$ = *não-constante*. Como um outro exemplo, se, ao longo de qualquer percurso, nada é conhecido a respeito de x, refletido por $\mu(x)$ = *indefinido* e ao longo de um outro percurso, é creditado a x ter o valor c, então, após a confluência desses percursos, podemos somente afirmar que x possui o valor c. Naturalmente, uma descoberta posterior de outro percurso até o ponto de confluência, ao longo do qual x possua um valor além de c, irá mudar o valor atribuído a x, após a confluência, para *não-constante*.

Finalmente, precisamos projetar o conjunto de funções F que reflitam a transferência de informações do início para o final de um bloco. A descrição deste conjunto de funções é complicada, apesar das idéias serem diretas. Iremos, por conseguinte, fornecer uma base para o conjunto de funções através da descrição das funções que representam enunciados singelos e o conjunto inteiro de funções poderá ser então construído através da composição de funções, a partir deste conjunto base, de forma a refletir os blocos com mais de um enunciado de definição.

1. A função identidade está em F; esta função reflete (isto é, replica) qualquer bloco que não tenha enunciados de definição. Se I é a função identidade, e μ é um mapeamento qualquer a partir das variáveis para os valores, então $I(\mu) = \mu$. Notemos que μ por si só não precisa ser a identidade; isto é arbitrário.

2. Para cada variável x e constante c existe uma função f em F, tal que, para cada mapeamento μ em V, temos $f(\mu) = v$, onde $v(w)$, para todo w diferente de x e $v(x) = c$. Essas funções refletem a ação de um enunciado de atribuição x:=c.

3. Para cada três variáveis (não necessariamente distintas) x, y e z, existe uma função f em F tal que, para cada mapeamento μ em V, temos $f(\mu)= v$. O mapeamento v é definido por: para cada w, diferente de x, temos $v(w) = \mu(w)$ e $v(x) = \mu(y)+\mu(z)$. Se pelo menos um dentre $\mu(y)$ e $\mu(z)$ é *não-constante*, a soma é *não-constante*. Se um dentre $\mu(y)$ e $\mu(z)$ for *indefinido*, mas nenhum dos dois é *não-constante*, o resultado é, então, indefinido. Esta função expressa o efeito da atribuição x:=y+z. Como usual neste capítulo, + pode ser pensado como um operador genérico; aqui uma óbvia modificação é necessária se o operador for unário ternário ou com maior número de operandos e uma outra óbvia modificação é necessária para levar em conta o efeito de um enunciado de cópia x:=y.

4. Para cada variável x, existe uma função f em F, tal que, para cada μ, $f(x) = v$, onde $v(w) = \mu(w)$, para todo w que não x e $v(x)$ =

$v(x)$	$\mu(x)$			
	não-constante	c	d ($\neq c$)	*indefinido*
não-constante	*não-constante*	*não-constante*	*não-constante*	*não-constante*
c	*não-constante*	c	*não-constante*	c
indefinido	*não-constante*	c	d	*indefinido*

Fig. 10.58. $\rho(x)$ em termos de $\mu(x)$ e de $v(x)$.

não-constante. Esta função reflete uma definição através de uma leitura de x, já que, após um enunciado de leitura, x certamente precisa ser presumido como não contendo qualquer valor constante particular. □

Os Axiomas das Estruturas da Análise de Fluxo de Dados

A fim de fazer funcionar, para uma estrutura arbitrária, os tipos de algoritmos de fluxo de dados que vimos até então, precisamos fazer algumas suposições a respeito do conjunto V, do conjunto de funções F e do operador de reunião $\wedge$. Nossas suposições básicas são listadas abaixo, apesar de alguns algoritmos precisarem de suposições adicionais.

1. F possui função identidade I, tal que $I(\mu) = \mu$, para todo μ em V.

2. F é fechado sob a composição de funções; isto é, para quaisquer duas funções f e g em F, a função h definida por $h(\mu) = g(f(\mu))$ está em F.

3. $\wedge$ é uma operação associativa, comutativa e idempotente. Essas três propriedades são expressas algebricamente como

$$\mu \wedge (\nu \wedge \rho) = (\mu \wedge \nu) \wedge \rho$$
$$\mu \wedge \nu = \nu \wedge \mu$$
$$\mu \wedge \mu = \mu$$

para todos os μ, ν e ρ em V.

4. Existe um elemento de *topo* $\top$, satisfazendo a lei

$$\top \wedge \mu = \mu$$

para todo μ em V.

Exemplo 10.43. Vamos considerar as definições incidentes. F certamente possui a identidade, a função onde os conjuntos *geradas* e *mortas* são ambos o conjunto vazio. Para mostrar o fechamento sob a composição de funções, suponhamos ter duas funções

$$f_1(X) = G_1 \cup (X - M_1)$$
$$f_2(X) = G_2 \cup (X - M_2)$$

Então,

$$f_2(f_1(X)) = G_2 \cup ((G_1 \cup (X - M_1)) - M_2)$$

Podemos verificar que o lado direito da igualdade acima é algebricamente igual a

$$(G_2 \cup (G_1 - M_2)) \cup (X - (M_1 \cup M_2))$$

Se fizermos $M = M_1 \cup M_2$ e $G = (G_2 \cup (G_1 - M_2))$, mostramos, então, que a composição de f_1 e f_2, a qual é $f(X) = G \cup (X - M)$, é da forma que a torna um membro de F.

Para o operador de reunião, que é a união, é fácil verificar que a união é associativa, comutativa e idempotente. O elemento de "topo" vem a ser o cojunto vazio neste caso, uma vez que $\phi \cup X = X$ para todo conjunto X.

Quando consideramos as expressões disponíveis, encontramos que os mesmos argumentos usados para as definições incidentes também mostram que F possui uma identidade e é fechado sob a composição de funções. O operador de convergência é agora a interseção, mas esse operador é também associativo, comutativo e idempotente. O elemento de topo faz mais sentido do ponto de vista intuitivo desta vez; é o conjunto E de todas as expressões no programa, já que, para qualquer conjunto X de expressões, $E \cap X = X$. □

Exemplo 10.44. Vamos considerar a estrutura de cômputo de constantes, introduzida no Exemplo 10.42. O conjunto de funções F foi projetado para incluir a identidade e ser fechado sob a composição de funções. Para verificar as leis algébricas de $\wedge$, basta mostrar que as mesmas se aplicam a cada variável x. Como uma instância, iremos verificar a idempotência. Seja $\nu = \mu \wedge \mu$, isto é, para todo x, $\nu(x) = \mu(x) \wedge \mu(x)$. É simples verificar os casos em que $\nu(x) = \mu(x)$. Por exemplo, se $\mu(x) =$ *não-constante*, então, $\nu(x) =$ *não-constante*, já que o resultado do emparelhamento de *não-constante* consigo mesmo, na Fig. 10.58, é *não-constante*.

Finalmente, o elemento de topo é o mapeamento τ definido por $\tau(x) =$ *indefinido* para todas as variáveis x. Podemos verificar, a partir da Fig. 10.58, que, para qualquer mapeamento μ e qualquer variável x, se ν for a função $\tau \wedge \mu$, então, $\nu(x) = \mu(x)$, já que o resultado do emparelhamento de *indefinido* com qualquer valor, na Fig. 10.58, é aquele outro valor. □

Monotonicidade e Distributividade

Necessitamos de outra condição para fazer o algoritmo iterativo para a análise de fluxo de dados funcionar. Esta condição, chamada de monotonicidade, diz informalmente que, se tomarmos uma função qualquer f, proveniente do conjunto F, e a aplicarmos a dois membros de V, um "maior" do que o outro, o resultado da aplicação de f ao maior não é menor do que o que obteríamos aplicando f ao menor.

Para tornar a noção de "maior" precisa, definimos a relação $\leq$ em V por

$$\mu \leq \nu \text{ se e somente se } \mu \wedge \nu = \mu$$

Exemplo 10.45. Na estrutura das definições incidentes, onde a reunião é a união e os membros de V são conjuntos de definições, $X \leq Y$ significa $X \cup Y = X$, isto é, X é um superconjunto de Y. Por conseguinte, $\leq$ parece "para trás"; os menores elementos de V são superconjuntos dos maiores.

Para as expressões disponíveis, onde a convergência é a interseção, as coisas funcionam "direito" e $X \leq Y$ significa que $X \cap Y = X$, isto é, X é um subconjunto de Y. □

Notemos, a partir do Exemplo 10.45, que $\leq$, em nosso significado, não precisa ter todas as propriedades de $\leq$ sobre os inteiros. É verdadeiro que $\leq$ é transitiva; o leitor pode provar, como um exercício do uso dos axiomas de $\wedge$, que $\mu \leq \nu$ e $\nu \leq \rho$ implicam que $\mu \leq \rho$. No entanto, em nosso sentido, $\leq$ não é uma relação de ordem total. Por exemplo, na estrutura de expressões disponíveis, podemos ter dois conjuntos X e Y, nenhum dos quais seja um subconjunto do outro, caso em que nem $X \leq Y$ nem $Y \leq X$ é verdadeiro.

Freqüentemente auxilia desenhar o conjunto V num *diagrama reticulado*, o qual é um grafo cujos nós são os elementos de V e cujos lados são dirigidos para baixo, de X para Y, se $Y \leq X$. Por exemplo, a Fig. 10.59 mostra o conjunto V para o problema de fluxo de dados das definições incidentes, onde existem três definições, d_1, d_2 e d_3. Uma vez que $\leq$ é um "superconjunto de", um lado é dirigido para baixo a partir de qualquer subconjunto dessas três definições para cada um de seus superconjuntos. Já que $\leq$ é transitiva, convencionalmente omitimos o lado a partir de X para Y se existir um outro percurso de X para Y desenhado no diagrama. Por conseguinte, apesar de $\{d_1, d_2, d_3\} \leq \{d_1\}$, não desenhamos este lado uma vez que já é representado pelo percurso através de $\{d_1, d_2\}$, por exemplo.

É também útil notar que podemos extrair as operações de reunião a partir da leitura desse diagrama, já que $X \wedge Y$ é sempre o maior Z para o qual existam percursos para baixo, até Z, a partir de ambos, X e Y. Por exemplo, se X é $\{d_1\}$ e Y é $\{d_2\}$, então Z, na Fig. 10.59, é $\{d_1, d_2\}$, o que faz sentido, porque o operador de reunião é a união. É também verdadeiro que o elemento de topo irá aparecer ao topo do diagra-

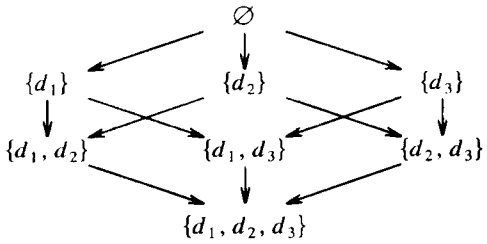


Fig. 10.59. Reticulado de subconjuntos de definições.

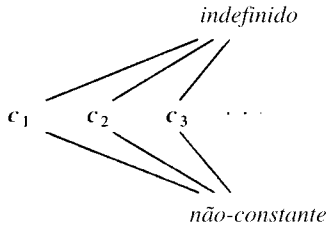


Fig. 10.60. Diagrama reticulado para valores de variáveis.

ma reticulado; isto é, existe um percurso para baixo de $\top$ para cada elemento.

Podemos, agora, definir uma estrutura $(F, V, \wedge)$ como *monótona* se

$$\mu \leq v \text{ implica que } f(\mu) \leq f(v) \qquad (10.15)$$

para todo μ e v em V e f em F.

Existe uma forma equivalente de se definir a monotonicidade:

$$f(\mu \wedge v) \leq f(\mu) \wedge f(v) \qquad (10.16)$$

para todo μ e v em V e f em F. É útil se ir e vir entre essas duas definições equivalentes, de forma que iremos delinear uma prova de suas equivalências, deixando para o leitor a verificação de algumas observações simples, usando a definição de $\leq$ e as leis associativa, comutativa e idempotente para $\wedge$.

Vamos assumir (10.15) e mostrar por que (10.16) é válida. Notemos, primeiro, que, para qualquer μ e v, $\mu \wedge v \leq \mu$ e $\mu \wedge v \leq v$ são ambas válidas; é uma prova simples, deixada para o leitor, mostrar esses fatos, provando que, para qualquer x e y, $(x \wedge y) \wedge y = x \wedge y$. Por conseguinte, por (10.15), $f(\mu \wedge v) \leq f(\mu)$ e $f(\mu \wedge v) \leq f(v)$. Deixamos para o leitor verificar a lei geral

$$x \leq y \text{ e } x \leq z \text{ implicam que } x \leq y \wedge z$$

Fazendo x ser igual a $f(\mu \wedge v)$, $y = f(\mu)$, e $z = f(v)$, temos (10.16).

Reciprocamente, vamos assumir (10.16) e provar (10.15). Supomos $\mu \leq v$ e usamos (10.16) para concluir $f(\mu) \leq f(v)$, provando, pois, (10.15). A equação (10.16) nos diz que $f(\mu \wedge v) \leq f(\mu) \wedge f(v)$ Mas, como é assumido que $\mu \leq v$, $\mu \wedge v = \mu$, por definição. Por conseguinte, (10.16) diz que $f(\mu) \leq f(\mu) \wedge f(v)$. Como uma regra geral, o leitor pode mostrar que

$$\text{se } x \leq y \wedge z, \text{ então } x \leq z$$

Conseqüentemente, (10.16) implica $f(\mu) \leq f(v)$, e havemos provado (10.15).

Freqüentemente, uma estrutura obedece a uma condição mais forte do que (10.16), que chamamos de *condição de distributividade*:

$$f(\mu \wedge v) = f(\mu) \wedge f(v)$$

para todo μ e v em V e f em F. Certamente, se $x = y$, então, $x \wedge y = x$ pela idempotência, e, dessa forma, $x \leq y$. Por conseguinte, a distributividade implica a monotonicidade.

Exemplo 10.46. Consideremos as estruturas de definições incidentes. Sejam X e Y conjuntos de definições e seja f uma função definida por $f(Z) = G \cup (Z - M)$ para alguns conjuntos de definições G e M. Podemos verificar, então, que a estrutura das definições incidentes satisfaz à condição de distributividade, através da verificação de que

$$G \cup ((X \cup Y) - M) = (G \cup (X - M) \cup (G \cup (Y - M))$$

O desenho de um diagrama de Venn torna transparente a prova do relacionamento acima, apesar do mesmo parecer complicado. □

Exemplo 10.47. Vamos mostrar que a estrutura de cômputo de constantes é monótona, mas não distributiva. Primeiro, é útil a aplicação da operação $\wedge$ e do relacionamento $\leq$ aos elementos que aparecem na Tabela da Fig. 10.58. Isto é, vamos definir

não-constante $\wedge$ c = *não-constante* para qualquer constante c

$c \wedge d$ = *não-constante* para $c \neq d$

$c \wedge$ *indefinido* = c para qualquer constante c

não-constante $\wedge$ *indefinido* = *não-constante*

$x \wedge x = x$ para qualquer valor x

A Fig. 10.58, então, pode ser interpretada como dizendo que $\rho(\mathtt{a}) = \mu(\mathtt{a}) \wedge v(\mathtt{a})$.

Podemos determinar que o relacionamento $\leq$ é sobre os valores, a partir da operação $\wedge$. Encontramos que

não-constante $\leq$ c para qualquer constante c

$c \leq$ *indefinido* para qualquer constante c

não-constante $\leq$ *indefinido*

Este relacionamento é mostrado no diagrama reticulado da Fig. 10.60, onde os c_i's estão destinados a sugerir todas as possíveis constantes. Notemos que aquela figura não é de $\leq$ sobre elementos de V; ao invés é uma relação sobre esses conjuntos de valores para $\mu(\mathtt{a})$ para valores individuais de a. Os elementos de V podem ser pensados como vetores de tais valores, um componente para cada variável e o diagrama reticulado para V pode ser extrapolado a partir da Fig. 10.60 se relembrarmos que $\mu \leq v$ é válida se e somente se $\mu(\mathtt{a}) \leq v(\mathtt{a})$ para todo a; isto é, os vetores representando μ e v têm cada componente relacionado por $\leq$ e o relacionamento é na mesma direção em cada componente.

Por conseguinte, dizer que $\mu \leq v$ é dizer que sempre que $\mu(\mathtt{a})$ for uma constante c, $v(\mathtt{a})$ ou é aquela constante ou *indefinido* e sempre que $\mu(\mathtt{a})$ for *indefinido*, também o será $v(\mathtt{a})$. Um exame cuidadoso das várias funções f que são associadas aos diferentes tipos de enunciados de definição nos capacita a verificar que se $\mu \leq v$, então $f(\mu) \leq f(v)$, provando, conseqüentemente, (10.15) e mostrando a monotonicidade. Por exemplo, se f está associada à atribuição a := b+c, somente $\mu(\mathtt{a})$ e $v(\mathtt{a})$ mudam, de forma que temos de verificar se $\mu \leq v$ — isto é, $\mu(\mathtt{x}) \leq v(\mathtt{x})$, para todo x —, então, $[f(\mu)](\mathtt{a}) \leq [f(v)](\mathtt{a})$.[14] Precisamos considerar todos os possíveis valores de $\mu(\mathtt{b})$, $\mu(\mathtt{c})$, $v(\mathtt{b})$ e $v(\mathtt{c})$, sujeitos às restrições de que $\mu(\mathtt{b}) \leq v(\mathtt{b})$ e $\mu(\mathtt{c}) \leq v(\mathtt{c})$, por exemplo, se

$\mu(\mathtt{b})$ = *não-constante*

$v(\mathtt{b})$ = 2

$\mu(\mathtt{c})$ = 3

$v(\mathtt{c})$ = *indefinido*

[14] Temos de ser cautelosos ao ler uma expressão como $[f(\mu)](\mathtt{a})$. A mesma diz que aplicamos f a μ para obter algum mapeamento $f(\mu)$, o qual chamamos μ'. Em seguida, aplicamos μ' a a e o resultado é um dos valores no diagrama 10.60.

então $[f(\mu)](\mathtt{a}) = \textit{não-constante}$ e $[f(v)](\mathtt{a}) = \textit{indefinido}$. Uma vez que *não-constante* $\leq$ *indefinido*, fizemos a verificação em um caso. Os outros casos são deixados para o leitor como exercício.

Precisamos, agora, verificar nossa afirmação de que a estrutura de cômputo de constantes não é distributiva. Para esta parte, seja f a função associada à atribuição `a:=b+c` e seja $\mu(\mathtt{b}) = 2$, $\mu(\mathtt{c}) = 3$, $v(\mathtt{b}) = 3$ e $v(\mathtt{c}) = 2$. Seja $\rho = \mu \wedge v$. Então, $\mu(\mathtt{b}) \wedge v(\mathtt{b}) = 2 \wedge 3 = \textit{não-constante}$. Semelhantemente, $\mu(\mathtt{c}) \wedge v(\mathtt{c}) = \textit{não-constante}$. Equivalentemente, $\rho(\mathtt{b}) = \rho(\mathtt{c}) = \textit{não-constante}$. Se segue que $[f(\rho)](\mathtt{a}) = \textit{não-constante}$, uma vez que a soma de dois valores não-constantes é presumida ser não-constante.

Por outro lado, $[f(\mu)](\mathtt{a}) = 5$, já que dado $\mathtt{b} = 2$ e $\mathtt{c} = 3$, a atribuição `a:=b+c` faz `a` igual a 5. Similarmente, $[f(v)](\mathtt{a}) = 5$. Por conseguinte, $(f(\mu) \wedge f(v))(\mathtt{a}) = 5$. Vemos agora que $\rho(\mathtt{a}) = [(f(\mu \wedge v)](\mathtt{a}) \neq [f(\mu) \wedge f(v)](\mathtt{a})$ e, dessa forma, a distributividade é violada.

Intuitivamente, a razão pela qual chegamos a uma violação da distributividade está em que a estrutura de cômputo de constantes não é poderosa o suficiente para se lembrar de todas as invariantes, em particular, o fato de que ao longo de percursos cujos efeitos sobre as variáveis são descritos ou por μ ou por v, a equação `b+c=5` está em vigor ainda que nem `b` nem `c` sejam, por si mesmas, constantes. Poderíamos divisar estruturas mais complicadas para evitar esse problema particular, apesar do retorno em o fazer não estar muito claro. Felizmente, a monotonicidade é adequada para um algoritmo de fluxo de dados iterativo "funcionar", como veremos em seguida. □

Soluções de Convergência de Percursos* para Problemas de Fluxo de Dados

Vamos imaginar que um grafo de fluxo tenha associado a cada um de seus nós uma função de transferência, uma das funções no conjunto F. Para cada bloco B, seja f_B a função de transferência para B.

Consideremos qualquer percurso $P = B_0 \rightarrow B_1 \ldots \rightarrow B_k$ a partir do nó inicial B_0 até algum bloco B_k. Podemos definir a *função de transferência para P* como sendo a composição de $f_{B_0}, f_{B_1}, \ldots, f_{B_{k-1}}$. Notemos que f_{B_k} não é parte da composição, refletindo o ponto de vista de que este percurso é tomado para atingir o início do bloco B_k, não o seu fim.

Assumimos que os valores em V representam informações a respeito de dados usados pelo programa e que o operador de confluência $\wedge$ nos diz como aquelas informações são combinadas quando os percursos convergem. Também faz sentido ver o elemento de topo como representando "nenhuma informação", uma vez que qualquer percurso que carregue o elemento de topo pode desaguar em qualquer outro percurso, dependendo de que informações sejam conduzidas após a confluência dos percursos. Por conseguinte, se B é um bloco no grafo de fluxo, as informações que entram em B deveriam ser computáveis através da consideração de cada possível percurso a partir do nó inicial até B e vendo-se o que acontece ao longo de cada um desses percursos, começando-se sem nenhuma informação. Isto é, para cada percurso P, de B_0 até B, computamos $f_P(\top)$ e tomamos a convergência de todos os elementos resultantes.

Em princípio, essa convergência poderia ser sobre um número infinito de diferentes valores, uma vez que existe um número infinito de diferentes percursos. Na prática, é freqüentemente adequado considerar somente os percursos acíclicos e mesmo quando não seja, como para a estrutura de cômputos de constantes, discutida acima, existem usualmente outras razões que podemos encontrar para tornar finita essa convergência infinita, para qualquer grafo de fluxo particular.

Formalmente, definimos a *solução de convergência de percursos* para um grafo de fluxo como sendo

$$scp(B) = \bigwedge_{\substack{\text{percursos } P \\ \text{de } B_0 \text{ até } B}} f_P(\top)$$

A solução *scp* para um grafo de fluxo faz sentido quando compreendemos que, na medida em que as informações que atingem o bloco B são consideradas, o grafo de fluxo pode igualmente ser aquele sugerido na Fig. 10.61, onde a cada função de transferência associada a cada um dos percursos $P_1, P_2, \ldots$ (o número de percursos é possivelmente infinito), no grafo de fluxo original, foi dado um percurso inteiramente distinto até B. Na Fig. 10.61, as informações que atingem B são dadas pela convergência de todos os percursos.

Soluções Conservativas para Problemas de Fluxo de Dados

Quando tentamos solucionar equações de fluxo de dados que vêm de uma estrutura arbitrária, podemos estar ou não capacitados a obter a solução *scp* facilmente. Felizmente, da mesma forma que com exemplos concretos das estruturas de fluxo de dados das Seções 10.5 e 10.6, existe uma direção segura na qual errarmos e o algoritmo de fluxo de dados que discutimos naquelas seções se presta sempre a nos providenciar uma solução segura. Dizemos que uma solução *entrada*[B] é uma solução segura, se *entrada*[B] $\leq$ *scp*[B] para todos os blocos B.

A despeito do que o leitor poderia imaginar, não pinçamos esta definição da "estratosfera". Lembremos que, em qualquer grafo de fluxo, o conjunto de percursos *aparentes* até um nó (aqueles que são percursos no grafo de fluxo) pode ser um subconjunto próprio dos percursos reais, aqueles que são seguidos em alguma execução do programa correspondente ao grafo de fluxo em questão. Para que o resultado da análise do fluxo de dados seja usável, seja lá qual for o propósito pretendido, os dados precisam estar seguros se modificarmos o grafo de fluxo removendo alguns dos percursos, uma vez que não podemos, em geral, distinguir os percursos reais dos percursos aparentes que não sejam reais.

Suponhamos que dentre o conjunto infinito de percursos sugerido pela Fig. 10.61, x seja a convergência de $f_P(\top)$ tomada sobre todos os percursos reais P que sejam seguidos em alguma execução. Igualmente, seja y a convergência de $f_P(\top)$ sobre todos os outros percursos P. Por conseguinte, $scp(B)$ é $x \wedge y$. Então, a resposta verdadeira para nosso problema de fluxo de dados ao nó B é x, mas a solução *scp* é $x \wedge y$. Relembremos que $x \wedge y \leq y$, uma vez que $(x \wedge y) \wedge y = x \wedge y$. Por conseguinte, "solução *scp* $\leq$ solução verdadeira".

Conquanto possamos preferir a solução "verdadeira" para o problema de fluxo de dados, quase certamente não teremos forma eficiente de dizer exatamente que percursos são reais e que percursos não o são, de forma que somos forçados a aceitar a solução *scp* como a solução viável mais próxima. Por conseguinte, qualquer que seja o uso que façamos das informações de fluxo de dados, o mesmo terá que ser consistente com a possibilidade da solução que obtemos ser $\leq$ que a solução verdadeira. Uma vez que aceitemos isso, devemos também ser capazes de aceitar uma solução que seja $\leq$ que a solução *scp* (e, por conseguinte, $\leq$ solução verdadeira). Tais soluções são mais fáceis de se obter do que a *scp* para aquelas estruturas que são monótonas mas não distributivas. Para estruturas distributivas, como aquelas da Seção 10.6, o algoritmo iterativo simples computa a solução *scp*.

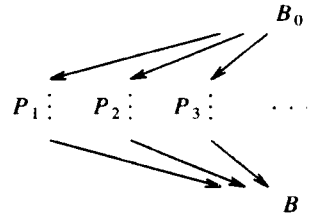


Fig. 10.61. Grafo mostrando o conjunto de todos os possíveis percursos até B.

*Do original em inglês: *Meet-over paths*. (N. do T.)

```
(1)  para cada nó B faça /* inicializar, assumindo que entrada[B] = ⊤ */
(2)      saída[B] := f_B[⊤];
(3)  enquanto ocorrerem mudanças a qualquer saída faça
(4)      para cada bloco B, em ordem de pesquisa em profundidade faça início
(5)          entrada[B] := ∧_{predecessores P de B} saída[P];
             /* acima, a convergência de um conjunto vazio é ⊤ */
(6)          saída[B] := f_B(entrada[B])
         fim
```

Fig. 10.62. Algoritmo iterativo para estruturas gerais.

O Algoritmo Iterativo para Estruturas Gerais

Existe uma generalização óbvia do Algoritmo 10.2, que funciona para uma ampla variedade de estruturas. O algoritmo iterativo requer que a estrutura seja monótona, e esta requer finitude, no sentido de que a convergência sobre o conjunto infinito de percursos, sugerida na Fig. 10.61, é equivalente à convergência sobre um subconjunto finito. Forneceremos o algoritmo e em seguida discutiremos as formas pelas quais poderíamos assegurar a finitude. No entanto, uma garantia comum de finitude é aquela que nos acompanhou o tempo todo: a propagação ao longo de percursos acíclicos é suficiente.

Algoritmo 10.18 Solução iterativa para estruturas gerais de fluxo de dados.

Entrada. Um grafo de fluxo de dados, um conjunto de "valores" V, um conjunto de funções F, uma operação de reunião $\wedge$ e uma atribuição de um membro de F a cada nó do grafo de fluxo.

Saída. Um valor *entrada*[B] em V para cada nó do grafo de fluxo.

Método. O algoritmo é dado pela Fig. 10.62. Da mesma forma que com os algoritmos iterativos de fluxo de dados, já familiares, computamos *entrada* e *saída* para cada nó através de aproximações sucessivas. Assumimos que f_B é a função em f associada ao bloco B; aquela função desempenha o papel de *geradas* e *mortas* na Seção 10.6.

Uma Ferramenta de Análise de Fluxo de Dados

Podemos ver agora como as idéias desta seção podem ser aplicadas a uma ferramenta para análise de fluxo de dados. O Algoritmo 10.18 depende, para seu funcionamento, das seguintes sub-rotinas.

1. Uma rotina para aplicar uma dada f_B em F a um dado valor em V. Esta rotina é usada nas linhas (2) e (6) da Fig. 10.62.

2. Uma rotina para aplicar o operador de reunião a dois valores em V; esta rotina é necessitada zero ou mais vezes à linha (5).

3. Uma rotina para informar se dois valores são iguais. Este teste não é feito explicitamente na Fig. 10.62, mas é implícito no teste para a mudança em quaisquer dos valores de *saída*.

Necessitamos, também, ter declarações de tipos de dados específicas para F e V, de forma a podermos transmitir argumentos às rotinas mencionadas acima. Os valores de *entrada* e *saída* , na Fig. 10.62, também são do tipo declarado para V. Finalmente, necessitamos de uma rotina que irá tomar a representação ordinária do conteúdo de um bloco básico, isto é, uma lista de enunciados, e produzir um elemento de F, ou seja, a função de transferência para aquele bloco.

Exemplo 10.48. Para a estrutura de definições incidentes, poderíamos, primeiro, construir uma tabela que identificasse cada elemento do grafo de fluxo dado com um único inteiro de 1 até algum máximo m. F poderia ser representado por pares de vetores de *bits* daquele tamanho, isto é, pelos conjuntos *geradas* e *mortas*. A rotina para construir os vetores de *bits* para *geradas* e *mortas*, dados os enunciados de um bloco e uma tabela que associa os enunciados de definição com posições no vetor de *bits*, é direta, como são as rotinas para computar as reuniões (ou lógico de vetores de *bits*), testar a igualdade de vetores de *bits* e aplicar funções definidas por um par *geradas-mortas* a vetores de *bits*. □

A ferramenta de análise de fluxo de dados é, por conseguinte, um pouco mais do que uma implementação da Fig. 10.62, com chamadas para as sub-rotinas dadas sempre que for necessitada uma reunião, aplicação de função, ou comparação. A ferramenta suportaria uma representação fixa de grafos de fluxo e, por conseguinte, estaria capacitada a realizar tarefas como encontrar todos os predecessores de um nó, encontrar o ordenamento em profundidade de um grafo de fluxo ou aplicar a cada bloco a rotina que computa a função em F associada àquele bloco. A vantagem de se usar uma tal ferramenta está em que a manipulação de grafos e os aspectos da verificação de convergência do Algoritmo 10.18 não têm que ser reescritos para cada análise de fluxo de dados que fazemos.

Propriedades do Algoritmo 10.18

Deveríamos tornar claras as suposições sob as quais o Algoritmo 10.18 efetivamente funciona e exatamente para o que o algoritmo converge, quando realmente converge. Primeiro, se a estrutura é monótona e converge, afirmamos, então, que o resultado para o algoritmo é que *entrada*[B] $\leq$ *scp*[B] para todos os blocos B. A razão intuitiva é que, ao longo de qualquer percurso $P = B_0, B_1, ..., B_i$, a partir do nó inicial até $B = B_i$, podemos mostrar por indução sobre i, que o efeito do percurso a partir de B_0 até B_i é sentido após o máximo de i iterações do laço **enquanto** da Fig. 10.62. Isto é, se P_i é o percurso $B_0, ..., B_i$, então, após i rodadas, *entrada*[B] $\leq f_P(\top)$. Por conseguinte, quando e se o algoritmo convergir, *entrada*[B] será $\leq f_P(\top)$ para cada percurso P de B_0 até B. Usando a regra que estabelece que se $x \leq y$ e $x \leq z$, então, $x \leq y \wedge z$[15], podemos mostrar que *entrada*[B] $\leq$ *scp*(B).

Quando uma estrutura é distributiva, podemos mostrar que o Algoritmo 10.18 converge, de fato, para a solução *scp*. A idéia essencial é provar que todas as vezes durante a execução do algoritmo, *entrada*[B] e *saída*[B] são, cada um, iguais à runião de $f_P(\top)$ para algum conjunto de percursos P até o início e final de B, respectivamente. No entanto, mostramos, no próximo exemplo, que este não precisa ser o caso quando a estrutura for monótona, mas não distributiva.

[15]Existe a tecnicalidade de que precisamos, em princípio, mostrar esta regra não somente para dois valores, y e z (a partir do que se segue a regra de que, se $x \leq y_i$, para qualquer conjunto finito de y_i's, então $x \leq \wedge y_i$), mas que a mesma regra seja válida para um número infinito de y_i's. No entanto, na prática, sempre que atingirmos a convergência do Algoritmo 10.18, iremos encontrar um número finito de percursos em que a reunião sobre todos os percursos é igual à reunião sobre este conjunto finito.

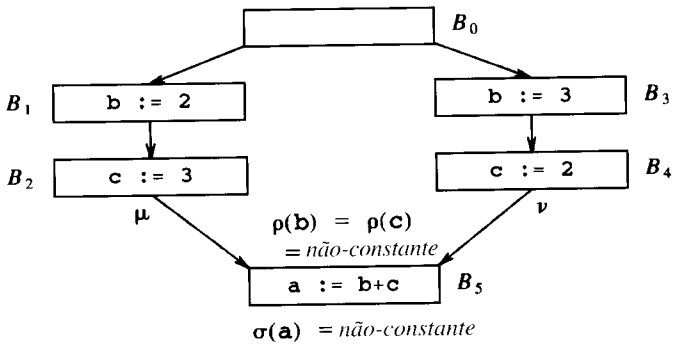


Fig. 10.63. Exemplo de solução menor do que a solução *scp*.

Exemplo 10.49. Vamos explorar o exemplo da não distributividade da estrutura de cômputo de constantes discutida no Exemplo 10.47; o grafo de fluxo relevante é mostrado na Fig. 10.63. Os mapeamentos μ e ν, saindo de B_2 e de B_4, são aqueles mostrados no Exemplo 10.47. O mapeamento ρ, entrando B_5, é $\mu \wedge \nu$ e σ é o mapeamento saindo de B_5, estabelecendo `a` em *não-constante*, ainda que cada percurso real (e cada percurso aparente) compute `a` = 5 após B_5.

O problema, intuitivamente, é que o Algoritmo 10.18, lidando com uma estrutura não distributiva, se comporta como se algumas seqüências de nós, que não são nem mesmo percursos aparentes (percursos no grafo de fluxo), fossem percursos reais. Por conseguinte, na Fig. 10.63, o algoritmo se comporta como se os percursos como $B_0 \rightarrow B_1 \rightarrow B_4 \rightarrow B_5$ ou $B_0 \rightarrow B_3 \rightarrow B_2 \rightarrow B_5$ fossem percursos reais, estabelecendo `b` e `c` com uma combinação de valores que não soma cinco.□

Convergência do Algoritmo 10.18

Existem várias formas pelas quais poderíamos provar que o Algoritmo 10.18 converge para uma estrutura particular. Provavelmente, o caso mais comum é quando somente percursos acíclicos sejam necessários, isto é, podemos mostrar que a reunião sobre percursos acíclicos é o mesmo que a solução *scp* sobre todos os percursos. Se este for o caso, não somente o algoritmo converge, mas normalmente o fará muito rapidamente, em duas passagens a mais do que a profundidade do grafo de fluxo, como discutimos na Seção 10.10.

Por outro lado, estruturas como nosso exemplo de cômputo de constantes, requerem a consideração de mais do que os percursos acíclicos. Por exemplo, a Fig. 10.64 mostra um simples grafo de fluxo, onde temos que considerar o percurso $B_1 \rightarrow B_2 \rightarrow B_2 \rightarrow B_3$, para compreender que `x` não tem um valor constante ao entrar em B_3.

No entanto, para computações constantes, podemos argumentar que o Algoritmo 10.18 converge como se segue. Primeiro, é fácil mostrar para uma estrutura monótona arbitrária que *entrada*[B] e *saída*[B], para qualquer bloco B, formam uma seqüência não crescente, no sentido de que o novo valor para uma dessas variáveis é sempre $\leq$ que o valor antigo. Se relembrarmos a Fig. 10.60, o diagrama reticulado para os valores de mapeamentos aplicados às variáveis, compreendemos que, para qualquer variável, o valor de *entrada*[B] ou de *saída*[B] pode cair apenas duas vezes, uma a partir de *indefinido* para uma constante e outra daquela constante para *não-constante*.

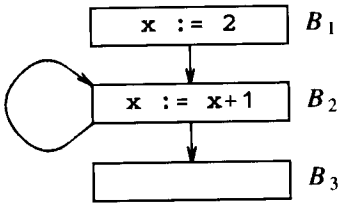


Fig. 10.64. Grafo de fluxo requerendo que percursos cíclicos sejam incluídos na *scp*.

Suponhamos que existam n nós e v variáveis. Então, a cada iteração do laço **enquanto** da Fig. 10.62, pelo menos uma variável precisará ter a queda em seu valor em algum conjunto *saída*[B], ou o algoritmo converge e, mesmo uma iteração infinita do laço **enquanto** não irá mudar os valores dos conjuntos *entrada* e *saída*. Por conseguinte, o número de iterações está limitado a $2nv$; se ocorrer tal número de mudanças, cada variável, então, terá que ter atingido o valor *não-constante* a cada bloco do grafo de fluxo.

Ajustando a Inicialização

Em alguns problemas de fluxo de dados existe uma discrepância entre o que o Algoritmo 10.18 nos dá como solução e o que desejamos intuitivamente. Relembremos que, para as expressões disponíveis, $\wedge$ é a interseção, de forma que $\top$ precisa ser o conjunto de todas as expressões. Uma vez que o Algoritmo 10.18 assume inicialmente que *entrada*[B] é $\top$ para cada bloco B, incluindo o nó inicial, a solução *scp* produzida pelo Algoritmo 10.18 é efetivamente o conjunto de expressões que, assumindo que as mesmas estejam disponíveis no nó inicial (o que não o estão), deveriam estar disponíveis à entrada do bloco B.

A diferença, naturalmente, é que poderiam existir percursos a partir do nó inicial até B ao longo dos quais uma expressão `x+y` não é nem gerada nem morta. O Algoritmo 10.19 diria que `x+y` está disponível quando, de fato, não o está, porque nenhuma variável ao longo daquele percurso pode ser achada guardando o seu valor. A correção é simples. Podemos ou modificar o Algoritmo 10.18 de forma que, para a estrutura de expressões disponíveis, *entrada*[B_0] seja estabelecido e mantido como o conjunto vazio, ou podemos modificar o grafo de fluxo introduzindo um nó inicial fictício, um predecessor do nó inicial real, que mate cada expressão.

10.12 ESTIMATIVA DE TIPOS

Chegamos agora a um problema de fluxo de dados que é mais desafiador do que as estruturas da seção anterior. Várias linguagens, indo de APL até SETL, aos muitos dialetos de Lisp, não requerem que os tipos das variáveis sejam declarados e permitem, inclusive, que a mesma variável abrigue valores de diferentes tipos em diferentes tempos. Tentativas sérias de compilar tais linguagens em código eficiente têm usado a análise de fluxo de dados para inferir os tipos de variáveis, já que, digamos, o código para adicionar dois inteiros é muito mais eficiente do que chamar uma rotina geral para adicionar dois objetos dentro de uma variedade de tipos possíveis (por exemplo, inteiros, reais, vetores).

Nossa primeira estimativa seria a de que o cômputo dos tipos das variáveis seria algo como computar as definições incidentes. Podemos associar um conjunto de possíveis tipos para cada variável a cada ponto. O operador de confluência é a união sobre os conjuntos de tipos, já que se a variável `x` possuir o conjunto de tipos possíveis S_1 em um percurso e o conjunto S_2 em outro, então, `x` possui qualquer um dos tipos em $S_1 \cup S_2$ após a confluência dos percursos. Na medida em que o percurso passe através de um enunciado, devemos estar capacitados a realizar algumas inferências a respeito dos tipos das variáveis, baseados nos operadores do enunciado, nos possíveis tipos de seus operandos e nos tipos que produzem como resultado. O Exemplo 6.6, que lidou com um operador que podia multiplicar tanto números inteiros quanto complexos, foi um exemplo desse tipo de inferência.

Infelizmente, existem, pelo menos, dois problemas com essa abordagem.

1. O conjunto de possíveis tipos para uma variável pode ser infinito.

2. A determinação de tipo usualmente requer tanto a propagação de informações para adiante quanto para trás, a fim de obter estimativas

precisas dos possíveis tipos. Por conseguinte, mesmo a estrutura da Seção 10.11 não é geral o suficiente para fazer justiça ao problema.

Antes de considerar o ponto (1), vamos examinar alguns tipos de inferências a respeito de tipos que podem ser feitas em algumas linguagens familiares.

Exemplo 10.50. Consideremos os enunciados

```
i := a[j]
k := a[i]
```

Suponhamos que, em primeiro lugar, não saibamos nada a respeito dos tipos das variáveis `a`, `i`, `j` e `k`. No entanto, vamos supor que o operador de acesso a *arrays* [] requeira um argumento inteiro. Pelo exame do primeiro enunciado, podemos inferir que `j` seja um inteiro àquele ponto e que `a` seja um *array* de elementos de algum tipo. Em seguida, o segundo enunciado nos diz que `i` é um inteiro.

Agora, devemos propagar as inferências para trás. Se `i` foi computado como sendo um inteiro no primeiro enunciado, então o tipo da expressão `a[i]` precisa ser inteiro, o que significa que `a` precisa ser um *array* de inteiros. Podemos, então, raciocinar para frente, de novo, para descobrir que o valor atribuído a `k` pelo segundo enunciado precisa, também, ser um inteiro. Note-se que é impossível descobrir que os elementos de `a` são inteiros raciocinando-se somente para a frente ou para trás somente. □

Lidando com Conjuntos Infinitos de Tipos

Existem numerosos exemplos de casos patológicos onde o conjunto de tipos possíveis para uma variável é realmente infinito. Por exemplo, SETL permite que um enunciado como

```
x := {x}
```

seja executado dentro de um laço. Se começarmos sabendo somente que `x` poderia ser um inteiro, após considerarmos uma iteração do laço, compreendemos que `x` poderia ser um inteiro ou um conjunto de inteiros. Após considerar uma segunda iteração, encontramos que `x` poderia ser um conjunto de um conjunto de inteiros e assim por diante.

Um problema similar poderia ocorrer numa versão sem tipo de uma linguagem convencional como C, onde o enunciado

```
x = &x
```

com a possibilidade inicial de que `x` seja um inteiro nos leva a descobrir que `x` pode ter qualquer tipo da forma

apontador para apontador para ... apontador para inteiro

A forma tradicional para se lidar com tais problemas é reduzir o conjunto de possíveis tipos a um número finito. A idéia geral é a de agrupar o número infinito de possíveis tipos num número finito de classes, geralmente mantendo os tipos mais simples isolados e agrupando os mais complicados, e desejavelmente mais raros, em classes maiores. Quando assim o fazemos, temos de exercitar o julgamento sobre como fazer as inferências a respeito das interações entre os tipos e os operadores. O seguinte exemplo sugere o que pode ser feito.

Exemplo 10.51. Vamos continuar o exemplo do Capítulo 6, onde usamos o operador $\rightarrow$ como um construtor de tipos para funções. Aqui, nosso conjunto de tipos irá incluir o tipo básico *int* e todos os tipos da forma $\tau \rightarrow \sigma$, representando o tipo de uma função com domínio de tipo τ e intervalo de tipo σ, onde τ e σ são tipos no nosso conjunto. Por conseguinte, o conjunto de tipos é infinito, incluindo tipos tais como

$$(int \rightarrow int) \rightarrow ((int \rightarrow int) \rightarrow int)$$

Para reduzir este conjunto a um número finito de classes, iremos restringir uma expressão de tipo, de forma que tenha somente um construtor do tipo função $\rightarrow$, substituindo subexpressões numa expressão de tipo contendo pelo menos uma ocorrência de $\rightarrow$ pelo nome *func*. Por conseguinte, existem cinco diferentes tipos:

$$\begin{array}{l} int \\ int \rightarrow int \\ int \rightarrow func \\ func \rightarrow int \\ func \rightarrow func \end{array}$$

Iremos representar os conjuntos de tipos como vetores de *bits* de comprimento cinco, com as posições correspondendo aos cinco tipos listados na ordem acima. Por conseguinte, 01111 representa o tipo para qualquer aplicação de função, isto é, para qualquer coisa que não *int*. Notemos que este é, num certo sentido, o tipo de *func*, uma vez que *func* pode não ser um inteiro.

O enunciado de atribuição básico para o nosso modelo é

```
x := f(y)
```

Sabendo os possíveis tipos de `f` e de `y`, podemos determinar os possíveis tipos de `x` procurando pelo tipo na tabela da Fig. 10.65. Se `f` puder ser qualquer tipo no conjunto S_1 e y qualquer tipo no conjunto S_2, tomamos cada par τ em S_1 e σ em S_2 e procuramos pela entrada na linha para τ e coluna para σ, o que chamamos $\tau(\sigma)$. Em seguida, tomamos a união dos resultados de todas essas buscas para obter o conjunto de possíveis tipos para `x`.

Por exemplo, se $\tau = int \rightarrow func$ e $\sigma = int$, então, $\tau(\sigma) = 01111$. Isto é, o resultado de se aplicar um mapeamento do tipo $int \rightarrow func$ a um *int* é uma *func*, o que significa um mapeamento de qualquer um dos quatro nós além de *int*. Não podemos dizer qual porque nossa simplificação de um número infinito de tipos em apenas cinco classes nos impede de saber.

Para um segundo exemplo, seja τ como antes e $\sigma = int \rightarrow int$. Então, $\tau(\sigma) = 00000$, porque o tipo de domínio de τ é definitivamen-

τ	σ				
	int	*int→int*	*int→func*	*func→int*	*func→func*
int	00000	00000	00000	00000	00000
int→int	10000	00000	00000	00000	00000
int→func	01111	00000	00000	00000	00000
func→int	00000	10000	10000	10000	10000
func→func	00000	01111	01111	01111	01111

Fig. 10.65. O valor de $\tau(\sigma)$.

te diferente do tipo de σ, e, por conseguinte, o mapeamento é inaplicável. □

Um Sistema Simples de Tipos

Para ilustrar as idéias por trás dos nossos algoritmos de inferência de tipos, introduzimos um sistema simples de tipos e uma linguagem baseada no Exemplo 10.51. Os tipos são os cinco ilustrados naquele exemplo. Os enunciados de nossa linguagem são de três tipos.

1. `read x`. Um valor de `x` é lido a partir da entrada e, presumivelmente, nada é conhecido a respeito de seu tipo.

2. `x := f(y)`. O valor de `x` é estabelecido para aquele obtido através da aplicação da função `f` ao valor `y`. O que sabemos a respeito do valor de `x` após a atribuição é sumarizado na Fig. 10.65.

3. `use x as` τ. Ao irmos através de um enunciado, devemos assumir que o programa está correto e, por conseguinte, o tipo de `x` pode ser somente τ, tanto antes quanto depois do enunciado. O valor e o tipo de `x` não são afetados pelo enunciado.

Inferimos os tipos realizando uma análise de fluxo de dados no grafo de fluxo de um programa consistindo de enunciados desses três tipos. Por uma questão de simplicidade, assumimos que todos os blocos consistam de um único enunciado. Os valores de *entrada* e *saída* para os blocos são mapeamentos de variáveis para conjuntos dos cinco tipos a partir do Exemplo 10.51.

Inicialmente, cada conjunto *entrada* e *saída* mapeia cada variável no conjunto de todos os cinco tipos. À medida que propagamos informações, reduzimos o conjunto de tipos associados a certas variáveis em certos pontos, até que em algum instante não poderemos mais reduzir quaisquer desses conjuntos. Os conjuntos resultantes serão assumidos para indicar os possíveis tipos de cada variável a cada ponto. A suposição é conservativa, uma vez que um tipo é eliminado somente se pudermos provar que (dado que o programa está correto) que o tipo é impossível. Normalmente, esperamos tirar vantagem do fato de que certos tipos sejam impossíveis, não que sejam possíveis, dessa forma, "o maior possível" é a direção correta para os erros.

Usamos o mesmo esquema para modificar os conjuntos *entrada* e *saída:* um esquema "para adiante" e um esquema "para trás". O esquema para adiante usa o enunciado num bloco B e o valor de *entrada* (B) para restringir *saída*[B],[16] e o esquema para trás realiza o oposto. Em cada esquema, o operador de confluência é a "união de variáveis", no sentido em que a confluência de dois mapeamentos α e β é o mapeamento γ tal que, para todas as variáveis `x`,

$$\gamma[x] = \alpha[x] \cup \beta[x]$$

O Esquema para Adiante

Vamos supor que tenhamos um bloco B com *entrada*[B] como o mapeamento μ e *saída*[B] como o mapeamento ν. O esquema para adiante nos permite restringir ν. As regras para restringir ν dependem de que instrução é encontrada no bloco B, naturalmente.

1. Se o enunciado for `read x`, então qualquer tipo poderia ser lido. Se já soubermos algumas coisas a respeito do tipo de `x` após a leitura, não poderemos esquecer isso durante a passagem para adiante, de forma que simplesmente não mudamos $\nu(x)$ na passagem para adiante. Para todas as outras variáveis `y`, fazemos

$$\nu(y) := \nu(y) \cap \mu(y)$$

2. Suponhamos agora que o enunciado é `use x as` τ. Após esse enunciado, τ é o único possível tipo para `x`. Se já soubermos que o tipo τ é impossível para `x`, então não há tipo possível para `x` após o enunciado. Essas observações podem ser sumarizadas por:

$$\nu(x) := \nu(x) \cap \{\tau\}$$
$$\nu(y) := \nu(y) \cap \mu(y) \text{ para } y \neq x$$

3. Consideremos agora o caso em que o enunciado é `x := f(y)`. Os únicos tipos possíveis para `x` são aqueles que

   i) são possíveis de acordo com o valor presente de ν e

   ii) são o resultado de aplicação do mapeamento de algum tipo τ ao tipo σ e τ e σ são tipos que `f` e `y`, respectivamente, poderiam ter antes do enunciado ser executado.

Formalmente,

$$\nu(x) := \nu(x) \cap \{\rho \mid \rho = \tau(\sigma), \tau \text{ está em } \mu(f) \text{ e } \sigma \text{ em } \mu(y)\}$$

Podemos também fazer algumas inferências a respeito dos tipos de `f` e de `y`, já que na suposição da correção do programa, `f` não pode ter um tipo que não se aplique a algum tipo de `y` e não pode ter um tipo que não possa servir como tipo de argumento para algum possível tipo de `f`. Isto é, se `f` $\neq$ `x`, então

$$\nu(f) := \nu(f) \cap \{\tau \text{ em } \mu(f) \mid \text{para algum } \sigma \text{ em } \mu(y), \tau(\sigma) \neq \emptyset\}$$

se `y` $\neq$ `x`, então

$$\nu(y) := \nu(y) \cap \{\sigma \text{ em } \mu(y) \mid \text{para algum } \tau \text{ em } \mu(f), \tau(\sigma) \neq \emptyset\}$$

para todos os outros `z`'s

$$\nu(z) := \nu(z) \cap \mu(z)$$

O Esquema para Trás

Vamos considerar como, numa passagem para trás, podemos restringir μ baseados no que ν e o enunciado nos dizem.

1. Se o enunciado é `read x`, é fácil ver que não podem ser feitas, antes do enunciado, novas inferências sobre os tipos impossíveis, de forma que $\mu(x)$ não muda. No entanto, para todos os `y` $\neq$ `x`, podemos propagar informações para trás fazendo $\mu(y) := \mu(y) \cap \nu(y)$.

2. Se temos um enunciado `use x as` τ então podemos fazer o mesmo tipo de inferência que fizemos na direção para adiante; `x` pode ter somente o tipo τ antes do enunciado e os tipos de quaisquer outras variáveis são aqueles creditados como possíveis tanto antes quanto após o enunciado. Isto é,

$$\mu(x) := \mu(x) \cap \{\tau\}$$
$$\mu(y) := \mu(y) \cap \nu(y) \text{ para } y \neq x$$

3. Como antes, o caso mais complexo é um enunciado da forma `x := f(y)`. Para começar, nada novo pode ser inferido a respeito de `x`

[16] É valioso notar que, nos esquemas para adiante de fluxo de dados tradicionais, não restringimos *saída*, mas, em lugar, o recomputamos, de novo, a partir de *entrada*, a cada vez. Podíamos fazer isso porque os conjuntos *entrada* e *saída* sempre mudavam em uma direção, quer crescendo ou encolhendo. No entanto, num problema como a inferência de tipos, onde realizamos alternativamente passagens para a frente e para trás, podemos ter uma situação onde a passagem para trás deixou *saída* muito menor do que poderíamos justificar pela aplicação das regras para adiante a *entrada*. Por conseguinte, não podemos aumentar acidentalmente numa passagem para adiante somente para tê-lo diminuído de novo (mas talvez nem tanto) numa passagem para trás. Um comentário similar se aplica à passagem para trás; precisamos restringir *entrada*, não recomputá-lo.

antes do enunciado, a menos que aconteça x ser f ou y. Por conseguinte, $\mu(x)$ não é mudado exceto pelas regras seguintes, relacionando x e y. Notemos, em seguida, que, como nas regras para adiante, podemos fazer inferências a partir do fato de que os tipos de f e de y precisam ser compatíveis antes do enunciado. No entanto, se f ≠ x, podemos também restringir $\mu(f)$ aos tipos em $\nu(f)$ e uma afirmação análoga é válida a respeito de y. Por outro lado, se f = x, então os tipos de f após o enunciado são irrelacionados aos tipos de f antes do enunciado, de forma que nenhuma restrição desse tipo é permitida. De novo, uma afirmativa análoga é válida se y = x. É útil definir-se um mapeamento especial, justamente para f e y, de forma a refletir-se esta decisão. Por conseguinte, definimos

se f = x então $\mu_1(f) := \mu(f)$ senão $\mu_1(f) := \mu(f) \cap \nu(f)$
se y = x então $\mu_1(y) := \mu(y)$ senão $\mu_1(y) := \mu(y) \cap \nu(y)$

Agora, podemos restringir f e y àqueles tipos que são compatíveis com os conjuntos de tipos das demais. Ao mesmo tempo, podemos restringir os tipos de f e de y baseados no fato de que não precisam ser compatíveis, mas precisam produzir um tipo que ν diz que x possa ter. Por conseguinte, definimos:

$\mu(f) := \{\tau \text{ em } \mu_1(f) \mid \text{para algum } \sigma \text{ em } \mu_1(y), \tau(\sigma) \cap \nu(x) \neq \emptyset\}$
$\mu(y) := \{\sigma \text{ em } \mu_1(y) \mid \text{para algum } \tau \text{ em } \mu_1(f), \tau(\sigma) \cap \nu(x) \neq \emptyset\}$
$\mu(z) := \mu(z) \cap \nu(z)$ para z diferente de x, y ou f

Antes de prosseguir para o algoritmo de determinação de tipos, vamos relembrar de nossa discussão das definições incidentes na Seção 10.5 que, se começarmos com a falsa suposição de que alguma definição *d* esteja disponível em algum ponto num laço, podemos propagar erroneamente esse fato ao longo do laço, o que nos deixaria com um conjunto de definições incidentes maior do que o necessário. Um problema similar pode ocorrer na determinação de tipos, onde a suposição de que uma variável possa ter um certo tipo "prova" a si mesma na medida em que avançarmos em redor de um laço. Por conseguinte, iremos introduzir um trigésimo terceiro valor, adicionalmente aos 32 conjuntos de tipos a partir do Exemplo 10.51, que um mapeamento μ pode atribuir a uma variável, o valor *indefinido*. Esse uso de *indefinido* é similar a seu uso na estrutura de propagação de constantes da seção prévia.

Durante a confluência, o valor *indefinido* produz qualquer outro valor, isto é, age como o tipo 00000. Por outro lado, ao interseccionarmos conjuntos de tipos, por exemplo, ao computarmos $\mu(x) \cap \nu(x)$, o valor *indefinido* também produz qualquer outro conjunto de tipos, isto é, funciona como 11111. Por conseguinte, ao lermos, por exemplo, o valor de uma variável x, o fato de que o "tipo" de x fosse pensado com *indefinido* após a leitura é desconsiderado e o tipo de x se torna 11111.

Algoritmo 10.19

Entrada. Um grafo de fluxo cujos blocos são enunciados singelos dos três tipos mencionados acima (leitura (*read*), atribuição (:=) e uso (*use*)).

Saída. Um conjunto de tipos para cada variável a cada ponto. O conjunto é conservativo, no sentido em que qualquer cômputo real terá de levar a um tipo no conjunto.

Método. Computamos o mapeamento *entrada*[*B*] e o mapeamento *saída*[*B*] para cada bloco *B*. Cada mapeamento remete as variáveis do programa a conjuntos de tipos no sistema de tipos introduzido no Exemplo 10.5. Inicialmente, todos os mapeamentos remetem cada variável para *indefinido*.

Realizamos, então, passagens alternadas para adiante e para trás através do grafo de fluxo, até que passagens consecutivas para adiante e para trás, ambas, falhem em realizar quaisquer mudanças. A passagem para adiante é realizada por:

para cada bloco *B*, examinado em ordem de profundidade
faça início
$entrada[B] := \bigcup_{\text{predecessor } P \text{ de } B} saída[P];$

saída [*B*] := função de *entrada*[*B*] e *saída*[*B*] como definida acima
fim

A passagem para trás é:

para cada bloco *B*, examinado em ordem reversa de profundidade
faça início
$saída[B] := \bigcup_{\text{sucessor } S \text{ de } B} entrada[S];$

entrada[*B*] := função *entrada*[*B*] e *saída*[*B*] como definida acima
fim □

Exemplo 10.52. Consideremos o programa simples, sem ramificações, mostrado na Fig. 10.66. Estamos interessados em quatro mapeamentos, que designamos μ_1 a μ_4. Cada μ_i é tanto *saída*[B_i] quanto *entrada* [B_{i+1}]. Tecnicamente, não é assumido que B_i tenha dois enunciados, porque assumimos que os blocos sejam enunciados singelos nesta seção. No entanto, não estamos preocupados com o que acontece antes do final de B_1, porque todas as variáveis podem ter qualquer tipo lá.

Segue-se que precisamos de cinco passagens antes que a convergência ocorra e outras duas para detectar que a mesma ocorreu. As passagens são sumarizadas na Fig. 10.67(a)–(e). A primeira passagem é para adiante. Quando consideramos B_2, descobrimos que b não pode ser do tipo inteiro, porque é usada como um mapeamento. Descobrimos, também, que a é usada como inteiro em B_3 e, conseqüentemente, pode somente ser mapeada para *inteiro* em μ_3 e μ_4. Essas observações são sumarizadas na Fig. 10.67(a).

A segunda passagem, mostrada na Fig. 10.67(b) é para trás. Nesta passagem, quando consideramos B_2, sabemos que a precisa ser um inteiro quando b for aplicada à mesma. Por conseguinte, o tipo de b poderia somente ser *int* → *int* ou *int* → *func*. À terceira passagem, a qual é para adiante, esta restrição de tipo de b propaga-se por todo o percurso abaixo no grafo de fluxo, como mostrado em 10.67(c).

A quarta passagem é para trás, como mostrado na Fig. 10.67(d). Aqui, o fato de que c é um argumento de b em B_3 nos diz que c pode ser somente um inteiro. Igualmente, quando consideramos B_2, encontramos que o resultado de b(a) pode somente ser do tipo de c, o qual é *int*. Estes fatos proscrevem a possibilidade de que b seja do tipo *int* → *func*. Finalmente, na Fig. 10.67(e), vemos como à quinta passagem, para adiante, esses fatos sobre b e c se propagam. Nos casos poste-

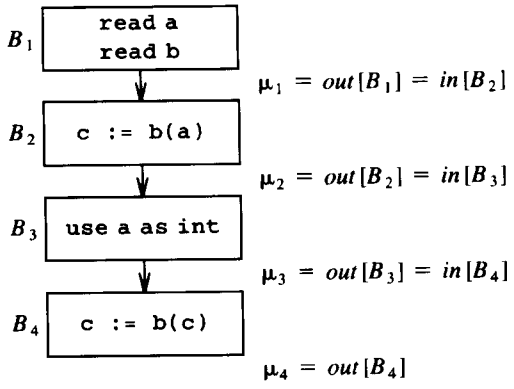


Fig. 10.66. Programa exemplo.

	a	b	c
μ_1	11111	11111	*undef*
μ_2	11111	01111	11111
μ_3	10000	01111	11111
μ_4	10000	01111	11111

(A) PARA ADIANTE

	a	b	c
μ_1	10000	01100	*undef*
μ_2	10000	01111	11111
μ_3	10000	01111	11111
μ_4	10000	01111	11111

(B) PARA TRÁS

	a	b	c
μ_1	10000	01100	*undef*
μ_2	10000	01100	11111
μ_3	10000	01100	11111
μ_4	10000	01100	11111

(C) PARA ADIANTE

	a	b	c
μ_1	10000	01000	*undef*
μ_2	10000	01100	10000
μ_3	10000	01100	10000
μ_4	10000	01100	11111

(D) PARA TRÁS

	a	b	c
μ_1	10000	01000	*undef*
μ_2	10000	01000	10000
μ_3	10000	01000	10000
μ_4	10000	01000	10000

(E) PARA ADIANTE

Fig. 10.67. Simulação do Algoritmo 10.19 no grafo de fluxo da Fig. 10.66.

riores, não podem ser feitas novas inferências. Neste caso, reduzimos os conjuntos de possíveis tipos a tipos singelos para cada variável a cada ponto; `a` e `c` são inteiros e `b` é um mapeamento de inteiros em inteiros. Em geral, poderíamos ser deixados com vários possíveis tipos para uma variável num ponto. □

10.13 DEPURAÇÃO SIMBÓLICA DE CÓDIGO OTIMIZADO

Um *depurador simbólico* é um sistema que nos permite examinar os dados do programa enquanto o mesmo estiver rodando. O depurador é usualmente chamado quando ocorre um erro de programa, tal como um estouro de capacidade, ou quando certos enunciados, indicados pelo programador no código-fonte, são atingidos. Uma vez invocado, o depurador simbólico permite que o programador examine, e possivelmente mude, quaisquer das variáveis que são correntemente acessíveis pelo programa em execução.

Para um comando do usuário, como "exiba o valor corrente de `a`", ser inteligível, para o depurador, o mesmo precisa ter disponível certas informações.

1. Deve haver uma forma de se associar um identificador como `a` à localização que o mesmo representa. Por conseguinte, a parte da tabela de símbolos que associa cada variável a uma localização, por exemplo, um local numa área de dados global ou num registro de ativação para algum procedimento, precisa ser registrada pelo compilador e preservada para uso do depurador. Essas informações, por exemplo, poderiam ser codificadas dentro do módulo de carga para o programa.

2. Deve haver informações de escopo, de forma a podermos tornar inambíguas as referências a um identificador que seja declarado mais de uma vez e de forma a que possamos dizer, dado que estejamos em algum procedimento `p`, quais dados de outros procedimentos são acessíveis e como encontramos os dados na pilha ou em outra estrutura em tempo de execucão. De novo, estas informações precisam ser obtidas a partir da tabela de símbolos do compilador e preservadas para uso futuro por parte do depurador.

3. Precisamos saber onde estamos no programa quando o depurador for invocado. Essas informações são inseridas pelo compilador na chamada para o depurador, quando o compilador trata a invocação para o depurador, declarada pelo usuário. É também obtida a partir do tratador de exceções quando um erro em tempo de execução faz com que o depurador seja chamado.

4. A fim de que as informações de localização do programa mencionadas em (3) façam sentido para o usuário, deverá haver uma tabela associando cada enunciado da linguagem de máquina ao enunciado-fonte do qual é oriundo. Esta tabela pode ser preparada pelo compilador à medida que gere o código.

Conquanto o projeto de um depurador simbólico seja interessante de *per si*, iremos considerar somente as dificuldades que ocorrem ao se tentar escrever um depurador simbólico para um compilador otimizante. À primeira vista, pode parecer que não haja necessidade de se depurar um programa otimizado. No ciclo normal de desenvolvimento, à medida que o usuário depura o programa, um compilador rápido, não otimizante, seria usado até que o usuário se certificasse de que o programa-fonte estivesse correto. Somente então um compilador otimizante seria usado.

Infelizmente, um programa pode rodar corretamente com um compilador não otimizante e falhar, com os mesmos dados de entrada, quando processado pelo compilador otimizante. Por exemplo, pode haver um erro no compilador otimizante ou, através do reordenamento das operações, um compilador otimizante pode introduzir um estouro de capacidade ou gerar um resultado irrepresentável na precisão da máquina. Inclusive, mesmo os compiladores "não otimizantes" podem realizar algumas transformações simples, como a eliminação de subexpressões comuns locais ou reordenar o código dentro de um bloco básico, o que faz uma grande diferença na dificuldade de se projetar um depurador simbólico. Por con-

seguinte, necessitamos considerar que algoritmos e estruturas de dados usar num depurador simbólico para um compilador otimizante que transforme os blocos básicos de formas arbitrárias.

Deduzindo Valores de Variáveis em Blocos Básicos

Por uma questão de simplicidade, vamos assumir que tanto a estrutura do código-fonte quanto a do objeto sejam seqüências de enunciados intermediários. O tratamento do código-fonte como código intermediário não apresenta problemas, uma vez que o último é mais geral do que o primeiro. Por exemplo, o usuário pode ser autorizado a somente colocar chamadas para o depurador entre os enunciados-fonte, mas aqui permitiremos que as chamadas sejam colocadas após qualquer enunciado intermediário. O tratamento do código objeto como código intermediário é questionável somente se o otimizador quebra um único enunciado intermediário em diversos enunciados de máquina que quedem separados. Por exemplo, por alguma razão, o compilador pode compilar os dois enunciados intermediários

```
u := v + w
x := y + x
```

em um código onde as duas adições sejam realizadas em registradores diferentes e entrelaçadas. Se esse for o caso, podemos tratar as cargas e armazenamentos de registradores como se os registradores fossem locais de armazenamento temporário no código intermediário, como, por exemplo:

```
r1 := v
r2 := y
r1 := r1 + w
r2 := r2 + z
 u := r1
 x := r2
```

Diversos problemas ocorrem ao interagirmos com o usuário a respeito de um bloco, onde o usuário pensa que o código-fonte está sendo executado, mas, de fato, uma versão otimizada daquele bloco está rodando:

1. Vamos supor que estejamos executando um programa que resulte da otimização de alguns blocos básicos do programa-fonte e, durante execução do enunciado `a:=b+c`, ocorra um estouro de capacidade. Precisamos informar ao usuário que um erro ocorreu em um dos enunciados-fonte. Uma vez que `b+c` pode ser uma subexpressão comum figurando em dois ou mais enunciados-fonte, para que enunciado indicaremos o erro?

2. Um problema mais difícil ocorre se o usuário do depurador deseja ver o valor "corrente" de alguma variável `d`. Num programa otimizado, `d` pode ter recebido uma atribuição por último em algum enunciado *s*. Mas no programa-fonte, *s* pode vir após o enunciado ao qual o depurador foi invocado, de forma que o valor de `d` que está disponível para o depurador não é aquele que o usuário pensa que seja o valor "corrente" de `d`, de acordo com a listagem do código-fonte. Similarmente, *s* pode preceder o enunciado que invoca o depurador, mas no código-fonte existe uma outra atribuição a `d` entre os mesmos, de forma que o valor de `d` disponível para o depurador estará desatualizado. É possível tornar disponível o valor correto para o usuário? Por exemplo, poderia ser o valor de alguma outra variável na versão otimizada ou poderia ser computado a partir dos valores de outras variáveis?

3. Finalmente, se o usuário coloca uma chamada para o depurador após algum enunciado do código-fonte, quando o controle deveria ser dado ao depurador durante a execução do código otimizado?

```
(1)  c := a+b      (1')  d := a+b
(2)  d := c        (2')  t := b*e
(3)  c := c-e      (3')  a := d-e
(4)  a := d-e      (4')  b := d/t
(5)  b := b*e      (5')  c := a
(6)  b := d/b
       (a)                (b)
```

Fig. 10.68. Código-fonte e otimizado.

Uma solução seria rodar a versão não otimizada do bloco juntamente com a versão otimizada, de forma a tornar disponível o valor correto de cada variável em todos os instantes. Rejeitamos esta "solução" porque os erros mais sutis, especialmente aqueles introduzidos pelo compilador, podem desaparecer quando as instruções que causaram o problema ficarem separadas uma das outras, no tempo ou no espaço.

A solução que adotamos é providenciar informações suficientes, a respeito de cada bloco, para o depurador, de forma que o mesmo possa pelo menos responder à questão: é possível providenciar o valor correto de uma variável `a` e, se assim o for, como? A estrutura que usamos para incorporar estas informações é o GDA do bloco básico, anotado com informações a respeito de que variáveis guardam o valor correspondente a um nó do GDA, e em que tempos, tanto no programa-fonte quanto no otimizado. A notação

$$\texttt{a} : i - j$$

atrelada a um nó significa que o valor representado por aquele nó é armazenado numa variável `a` a partir do início do enunciado *i*, através da parte do enunciado *j*, exatamente antes de sua atribuição ocorrer. Se $j = \infty$, então `a` abriga esse valor até o fim do bloco.

Exemplo 10.53. Na Fig. 10.68(a), vemos um bloco básico de um código-fonte e na Fig. 10.68(b) está uma possível versão "otimizada" daquele código. A Fig. 10.69 mostra o GDA para um bloco, com indicações dos intervalos nos quais as variáveis guardam esses valores, tanto no código-fonte quanto no otimizado. Os apóstrofos são usados para indicar que o intervalo de enunciados está no código otimizado. Por exemplo, o nó rotulado + é o valor de `c` no código-fonte, a partir do início do enunciado (2) até exatamente antes da atribuição no enunciado (3). É também o valor de `d` no código-fonte, a partir do início do enunciado (3) até o final. Adicionalmente, o mesmo nó é o valor de `d` no código otimizado, a partir do enunciado (2') até o final. □

Agora podemos responder à primeira questão levantada acima. Suponhamos que um erro, tal como um estouro de capacidade, ocorra enquanto se execute o enunciado *j'* do código otimizado. Uma vez que o mesmo valor seria computado por qualquer enunciado que compute

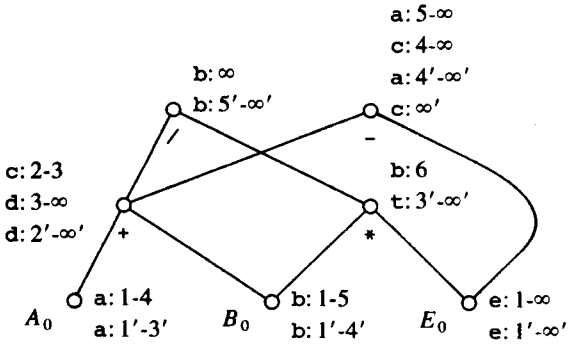


Fig. 10.69. GDA anotado.

o mesmo nó do GDA que o enunciado j', faz sentido reportar ao usuário que um erro ocorreu no primeiro enunciado-fonte a computar este nó. Por conseguinte, no Exemplo 10.53, se um erro ocorresse no enunciado 1', 2', 3' ou 4', reportaríamos que o mesmo teria ocorrido no enunciado 1, 5, 3 e 6, respectivamente. Nenhum erro pode ocorrer no enunciado 5', porque nenhum valor é computado. Postergamos os detalhes sobre como os enunciados correspondentes são computados até o Exemplo 10.54, abaixo.

Podemos, também, responder à segunda questão. Suponhamos que estejamos no enunciado j' do código otimizado e ao usuário é dito que o controle está no enunciado i do código-fonte, onde um erro ocorreu. Se o usuário solicita ver o valor da variável `x`, precisamos encontrar uma variável `y` (freqüentemente, mas nem sempre, `y` é `x`) tal que o valor de `x` no enunciado i do código-fonte esteja no mesmo nó do GDA que `y` no enunciado j' no código otimizado. Inspecionamos o GDA para verificar que nó representa o valor de `x` em i e podemos ler daquele nó todas as variáveis do programa objeto que alguma vez tiveram aquele valor, para ver se uma delas abriga este valor em j'.

Se assim o for, estamos feitos; senão, podemos ainda computar o valor de `x` em i a partir de outras variáveis em j'. Seja n o nó para `x` no tempo i. Podemos, então, considerar os filhos de n, digamos m e p, para ver ser ambos os nós representam o valor de alguma variável no tempo j. Se, digamos, existir uma variável para m, mas nenhuma para p, podemos considerar os filhos de p, recursivamente. Em última análise, ou encontramos uma forma de computar o valor de `x` no instante i ou concluímos que não existe uma tal forma de cômputo. Se encontrarmos uma forma de computar os valores de m e de p, então os computamos e em seguida aplicamos o operador a n para computar `x` no instante i.[17]

Exemplo 10.54. Suponhamos que durante a execução do código da Fig. 10.68(b), um erro ocorra no enunciado (2'). O enunciado estava computando o nó rotulado $*$ na Fig. 10.69 e o primeiro enunciado-fonte computando aquele valor é o enunciado 5. Reportamos, conseqüentemente, um erro no enunciado 5.

Na Fig. 10.70, tabulamos o nó do GDA ao início dos enunciados 5 e 2', respectivamente; os nós são indicados por seus rótulos, quer um símbolo de operador ou um símbolo de valor inicial como A_0. Indicamos, também, como computar o valor no tempo 5 a partir dos valores das variáveis no tempo 2'. Por exemplo, se o usuário solicita o valor de `a`, o valor do nó rotulado $-$ é fornecido. Nenhuma variável possui aquele valor no tempo 2', mas, felizmente, existem variáveis, `d` e `e`, que guardam o valor de cada um dos filhos do nó $-$ no instante 2', de forma que podemos imprimir o valor de `a` computando o valor de `d-e`. □

Agora, vamos responder à terceira questão: como tratar as chamadas do depurador inseridas pelo usuário. Num certo sentido, a resposta é trivial; se o usuário faz uma chamada ao depurador após o enunciado i no programa-fonte, podemos parar a execução do programa ao início do bloco. Se o usuário deseja ver o valor de alguma variável `x` após o enunciado i, consultamos o GDA anotado para ver que nó representa o valor desejado de `x` e computamos aquele valor a partir dos valores iniciais das variáveis para aquele bloco.

Por outro lado, podemos deixar menos trabalho para o depurador e também evitar algumas situações onde as tentativas para computar um valor levem a erros que precisem ser anunciados ao usuário, se postergarmos as chamadas para o depurador para um tempo tão tarde quanto o possível. É fácil computar o último enunciado j' no programa otimizado, de forma que chamamos o depurador após j' e encenamos para o usuário que a chamada foi feita após o enunciado i do programa-fonte. Para encontrar j', seja S o conjunto de nós do GDA que correspondem ao valor de alguma variável do programa-fonte imediatamente após o enunciado i. Podemos ser solicitados pelo usuário para computar qualquer valor em S. Por conseguinte, podemos chamar o depurador após o enunciado j' do código otimizado somente se, para cada nó n em S, existir algum $k' > j'$, tal que alguma variável esteja associada ao nó n no tempo k' no código otimizado. Para tanto, sabemos que o valor de n ou está disponível imediatamente após o enunciado j' ou será computado algum tempo após o enunciado j'. No primeiro caso, é trivial computar o valor de n se chamarmos o depurador após j', enquanto que, no último caso, sabemos que os valores disponíveis após j' são suficientes para computar n de alguma forma.

Variável	Valor		Obtido por
	Tempo 2'	Tempo 5	
a	A_0	-	d-e
b	B_0	B_0	b
c	indefinido	-	d-e
d	+	+	d
e	E_0	E_0	e
t	indefinido		

Fig. 10.70. Valores das variáveis nos tempos 2' e 5.

Exemplo 10.55. Consideremos de novo o código-fonte e o otimizado, da Fig. 10.68, e suponhamos que o usuário insira uma chamada ao depurador após o enunciado (3) do código-fonte. Para encontrar o conjunto S, inspecionamos o GDA da Fig. 10.69 e vemos que nós possuem variáveis do programa-fonte atreladas aos mesmos no tempo 4. Esses nós são aqueles rotulados A_0, B_0, E_0, + e $-$ naquela figura.

Em seguida, examinamos, de novo, o GDA com a finalidade de encontrar o maior j' tal que cada um dos nós em S tenha alguma variável do código otimizado atrelada a si num tempo estritamente superior a j'. Os nós rotulados +, $-$, e E_0 não apresentam problemas, uma vez que seus valores são carregados pelas variáveis `d`, `a` e `e`, respectivamente, num tempo ∞'. Os nós A_0 e B_0 efetivamente limitam o valor de j' e o que perde seu valor mais cedo, dentre os dois, é A_0, cujo valor é destruído pelo enunciado 3'. Por conseguinte, $j' = 2'$ é maior valor possível de j'; isto é, se o usuário solicita uma chamada ao depurador após o enunciado-fonte 3, podemos concedê-la após o enunciado 2'. □

O leitor deveria estar atento para uma sutileza no Exemplo 10.55, para a qual não há realmente uma boa solução. Se rodarmos o código otimizado através do enunciado 2' antes de chamarmos o depurador, um erro no cômputo de `b*e` ao enunciado 2' (por exemplo, um *underflow*) pode fazer com que o depurador seja chamado antes da chamada pretendida. No entanto, como o cômputo correspondente ao enunciado 2' não ocorre até o enunciado 5 no programa-fonte, iremos informar ao usuário que um erro ocorreu no enunciado 5. Será um tanto misterioso para o usuário a forma com que chegamos ao enunciado 5 sem chamar o depurador no enunciado 3. Provavelmente, a melhor solução para este problema é não permitir que j' seja tão grande que exista um enunciado k' no código otimizado, com $k' \leq j'$, tal que o código-fonte não compute o valor computado por k', até depois do enunciado i, no qual a chamada para o depurador foi colocada.

Efeitos da Otimização Global

Quando nosso compilador realiza otimizações globais, existem problemas mais difíceis para o depurador simbólico resolver e, freqüentemente, não existe nenhuma forma de encontrar o valor correto de uma va-

[17] Uma sutileza ocorre se o cômputo do valor do nó n causar um outro erro. Precisamos, então, reportar ao usuário de que o erro ocorreu realmente mais cedo, ao primeiro enunciado-fonte que compute o valor de n.

riável a um ponto. Duas importantes transformações que não acarretam problemas significativos são a eliminação de variáveis de indução e a eliminação global de subexpressões comuns; em cada caso, o problema pode ser confinado a uns poucos blocos e tratado da forma discutida abaixo.

Eliminação de Variáveis de Indução

Se eliminamos uma variável de indução *i*, do programa-fonte, em favor de algum membro da família de `i`, digamos `t`, existe, então, alguma função linear relacionando `i` e `t`. Sobretudo, se seguirmos os métodos da Seção 10.7, o código otimizado irá mudar `t` exatamente naqueles blocos nos quais `i` é mudado, de forma que o relacionamento linear entre `i` e `t` será sempre válido. Por conseguinte, após levar em consideração o reordenamento dos enunciados dentro de um bloco que atribua a `t` (e no fonte, atribua a `i`), podemos suprir o usuário com o valor "corrente" de `i` através de uma transformação linear sobre `t`.

Temos de ser cuidadosos caso `i` não seja definida antes do laço, uma vez que `t` receberá certamente uma atribuição antes da entrada do laço e poderíamos providenciar, certamente, um valor para `i` a um ponto do programa onde o usuário esperaria que a mesma estivesse indefinida. Felizmente, é usual para uma variável do programa-fonte, que seja uma variável de indução, que a mesma seja inicializada antes do laço e somente as variáveis geradas pelo compilador (cujos valores o usuário não pode interrogar) estarão indefinidas à entrada do laço. Se esse não for o caso para alguma variável de indução `i`, então temos um problema similar àquele da movimentação de código, a ser discutido abaixo.

Eliminação Global de Subexpressões Comuns

Se realizarmos uma eliminação global de subexpressões comuns para a expressão `a+b`, afetamos, também, um número limitado de blocos de uma maneira simples. Se `t` for a variável usada para guardar o valor de `a+b`, então, em certos blocos que computem `a+b`, poderíamos substituir

```
c := a+b
```

por

```
t := a+b
c := t
```

Este tipo de mudança pode ser manipulado pelos métodos para blocos básicos, já discutidos.

Em outros blocos, um uso, tal como `d:=a+b`, poderia ser substituído por `d:=t`. Para tratar esta situação pelos métodos anteriores, temos somente de anotar no GDA para esse bloco, que o valor para `t` permanece o tempo todo no valor do nó para `a+b` (que aparecerá no GDA para o código-fonte, mas que, contrariamente, jamais apareceria no GDA para o código otimizado).

Movimentação de Código

As outras transformações não são tão fáceis de se tratar. Por exemplo, suponhamos ter um enunciado de cópia

```
s: a:= b+c
```

fora do nosso laço porque o mesmo é laço-invariante. Se chamarmos o depurador dentro do laço, não saberemos se o enunciado *s* já teria sido executado no programa-fonte e, por conseguinte, não poderíamos saber se o valor corrente `a` é um daqueles que o usuário veria no programa-fonte.

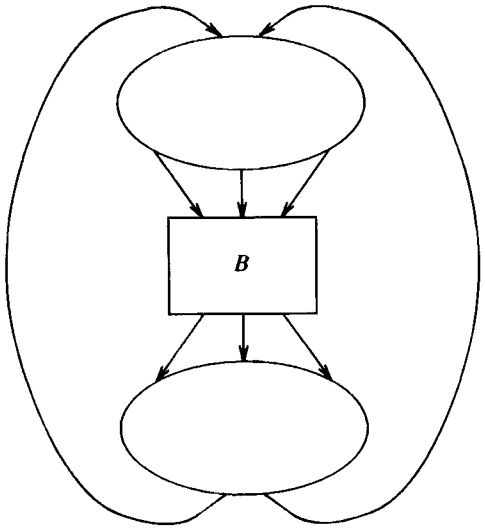


Fig. 10.71. Um bloco que divide um laço em duas partes.

Uma possibilidade seria inserir no código otimizado uma nova variável que, dentro do laço, indicasse se `a` já teria sido atribuído dentro do laço (o que pode somente ocorrer na antiga posição do enunciado *s*). No entanto, esta estratégia pode nem sempre ser adequada, porque, para absoluta confiabilidade, somente o código real, não uma versão do código construída especialmente para fins de depuração, deveria ser usada.

Existe um caso especial comum onde podemos fazer melhor, entretanto. Suponhamos que o bloco *B*, contendo o enunciado *s* no programa-fonte, divida o laço em dois conjuntos de nós: aqueles que dominam *B* e aqueles que são dominados pelo mesmo. Sobretudo, suponhamos que todos os predecessores do cabeçalho sejam dominados por *B*, como sugerido na Fig. 10.71. Podemos assumir, então, na primeira vez através dos blocos que dominam *B*, que `a` não recebeu atribuição no laço, enquanto que, na primeira através dos blocos que *B* domina, podemos assumir que `a` recebeu atribuição no enunciado *s*. Naturalmente, na segunda vez, e subseqüentes, ao longo do laço, `a` certamente foi atribuída em *s*.

Se a chamada para o depurador for causada por um erro em tempo de execução, há uma boa chance de que o erro seja exposto na primeira vez ao longo do laço. Se esse for o caso, temos, então, que saber somente se estamos acima ou abaixo de *B*, na Fig. 10.71. Saberemos, então, se o valor de `a` é aquele definido em *s*, caso em que simplesmente imprimimos o valor de `a` produzido pelo código otimizado ou se o valor de `a` é aquele detido à entrada do laço na versão fonte do programa. No último caso, há pouco a fazer, exceto se

1. o depurador atingiu as informações de definição disponíveis ao mesmo, tanto para o programa-fonte quanto para o otimizado.

2. há uma única definição de `a` que atinge o cabeçalho no programa-fonte, e

3. a definição é, também, a única definição de alguma variável `x` que atinge o ponto onde o depurador foi chamado.

Se todas essas condições forem válidas, podemos imprimir o valor de `x` e dizer que é o valor de `a`.

O leitor deve estar alerta para o fato de que esta linha de raciocínio não será válida se o depurador tiver sido chamado num ponto de quebra inserido pelo usuário, motivo pelo qual não teremos qualquer razão para suspeitar que estaremos indo em torno do laço pela primeira vez. No entanto, se os pontos de quebra inseridos pelo usuário estão

sendo usados, seria razoável, então, inserir código no programa otimizado, que auxiliasse o depurador a informar se esta era a primeira vez que seguíamos através do laço, uma solução que mencionamos mais cedo mas que sugerimos inadequada porque interferia com o código otimizado.

EXERCÍCIOS

10.1 Considere a multiplicação matricial na Fig. 10.72.
- a) Assumindo que `a`, `b` e `c` tenham a memória alocada estaticamente e que haja quatro *bytes* por palavra numa memória endereçada em *bytes*, produza os enunciados de três endereços para o programa da Fig. 10.72.
- b) Gere o código de máquina-alvo a partir dos enunciados de três endereços.
- c) Construa um grafo de fluxo a partir dos enunciados de três endereços.
- d) Elimine as subexpressões comuns a partir de cada bloco básico.
- e) Encontre os laços no grafo de fluxo.
- f) Mova as computações laço-invariantes para fora dos laços.
- g) Encontre as variáveis de indução para cada laço e as elimine onde possível.
- h) Gere o código da máquina-alvo a partir do grafo de fluxo em (g). Compare o código com aquele produzido em (b).

10.2 Compute as definições incidentes e cadeias-ud para o grafo de fluxo original do Exercício 10.1(c) e o grafo de fluxo final a partir de 10.1(g).

10.3 O programa da Fig. 10.73 conta os primos de `2` até `n` usando o método da "peneira" sobre um *array* adequadamente grande.
- a) Traduza o programa da Fig. 10.73 em enunciados de três endereços assumindo que `a` tenha a sua memória alocada estaticamente.
- b) Gere o código da máquina-alvo a partir dos enunciados de três endereços.
- c) Construa um grafo de fluxo para os enunciados de três endereços.
- d) Mostre a árvore de dominadores para o grafo de fluxo em (c).
- e) Para o grafo de fluxo em (c), indique os lados refluentes e seus laços naturais.
- f) Mova as computações invariantes para fora dos laços usando o Algoritmo 10.7.
- g) Elimine as variáveis de indução onde quer que seja possível.
- h) Propague para fora os enunciados de cópia sempre que possível.
- i) É possível uma quebra de laço? Se for, faça-a.
- j) Na suposição de que `n` seja sempre par, simplifique os laços mais internos um de cada vez. Que novas otimizações são agora possíveis?

```
begin
    for i := 1 to n do
        for j := 1 to n do
            c[i,j] := 0;
    for i := 1 to n do
        for j := 1 to n do
            for k := 1 to n do
                c[i,j] := c[i,j] + a[i,k] * b[k,j]
end
```

Fig. 10.72. Programa de multiplicação matricial.

```
begin
    read n;
    for i := 2 to n do
        a[i] := true;    /* initialize  */
    count := 0;
    for i := 2 to n ** .5 do
        if a[i] then    /* i is a prime */
            begin
                count := count + 1;
                for j := 2 * i to n by i do
                    a[j] := false
                    /* j is divisible by i */
            end;
    print count
end
```

Fig. 10.73. Programa para calcular primos.

10.4 Repita o Exercício 10.3 na suposição de que `a` tenha a memória alocada dinamicamente, com `ptr` sendo um apontador para a primeira palavra de `a`.

10.5 Para o grafo de fluxo da Fig. 10.74, compute:
- a) as cadeias-ud e du
- b) as variáveis vivas ao final de cada bloco
- c) as expressões disponíveis.

10.6 É possível alguma transposição para constante na Fig. 10.74? Se for, faça-o.

10.7 Há subexpressões comuns na Fig. 10.74? Se houver, elimine-as.

10.8 Uma expressão e é dita ser *muito ocupada* a um ponto p se, não importa que percurso seja tomado a partir de p, a expressão e será avaliada antes que qualquer um de seus operandos seja definido. Forneça um algoritmo, no estilo da Seção 10.6, para encontrar todas as expressões muito ocupadas. Informe que operador de confluência V. deverá usar e se a propagação irá se dar para a frente ou para trás. Aplique seu algoritmo ao grafo de fluxo da Fig. 10.74.

***10.9** Se a expressão e é muito ocupada a um ponto p, podemos *aliviar* e computando-a em p e preservando seu valor para uso subseqüente. (Nota: esta otimização usualmente não economiza tempo, mas pode produzir ganhos de espaço). Forneça um algoritmo para aliviar expressões muito ocupadas.

10.10 Existem expressões que podem ser aliviadas na Fig. 10.74? Se assim o for, alivie-as.

10.11 Onde possível, propague para fora quaisquer passos de cópia introduzidos nas modificações dos Exercícios 10.6, 10.7 e 10.10.

10.12 Um *bloco básico estendido* é uma seqüência de blocos $B_1, \ldots, B_k$ tal que, para $1 \leq i \leq k$, B_i é o único predecessor de B_{i+1} e B_1 não tem um único predecessor. Encontre o final do bloco estendido a cada nó
- a) da Fig. 10.39
- b) do grafo de fluxo construído no Exercício 10.1(c)
- c) da Fig. 10.74.

***10.13** Forneça um algoritmo que rode num tempo $O(n)$ (de ordem n) num grafo de fluxo para encontrar o final do bloco básico estendido a cada nó.

10.14 Podemos realizar alguma otimização interblocos sem qualquer análise de fluxo de dados, tratando cada bloco básico estendido como se fosse um bloco básico. Forneça algoritmos para realizar as seguintes otimizações dentro de um bloco básico estendido; em cada caso, indique que efeitos pode causar, sobre outros blocos básicos estendidos, uma mudança dentro de um bloco básico estendido.

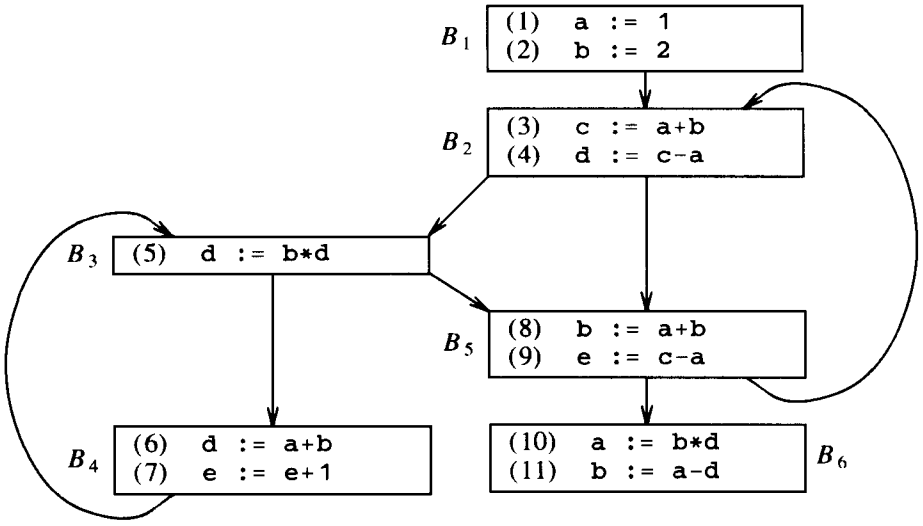


Fig. 10.74. Grafo de fluxo.

a) Eliminação de subexpressões comuns
b) Transposição para constantes
c) Propagação de cópias.

10.15 Para o grafo de fluxo do Exercício 10.1(c):
a) Encontre uma seqüência de reduções T_1 e T_2 para o grafo
b) Encontre a seqüência de grafos de intervalo
c) Qual é o grafo de fluxo limite? É o grafo de fluxo redutível?

10.16 Repita o Exercício 10.15 para o grafo de fluxo da Fig. 10.74.

****10.17** Mostre que as seguintes condições são equivalentes (são as definições alternativas de "grafo de fluxo redutível")
a) O limite de reduções $T_1 - T_2$ é um único nó
b) O limite da análise de intervalos é um único nó.
c) O grafo de fluxo pode ter seus lados divididos em duas classes; uma forma um grafo acíclico e a outra, os lados de "retorno", consiste nos lados cujas cabeças dominam suas caudas.
d) O grafo de fluxo não possui subgrafo da forma mostrada na Fig. 10.75. Aqui, n_0 é o nó inicial e n_0, a, b e c são todos distintos, com exceção que $a = n_0$ é possível. As setas representam os percursos de nós disjuntos (exceto para os pontos finais, naturalmente).

10.18 Forneça algoritmos para computar (a) as expressões disponíveis e (b) as variáveis vivas para a linguagem com apontadores discutida na Seção 10.8. Assegure-se de fazer suposições conservativas a respeito de *geradas, mortas, usos* e *definidas* em (b).

10.19 Dê um algoritmo para computar as definições interprocedimentalmente incidentes, usando o modelo da Seção 10.8. De novo, assegure-se de fazer aproximações conservativas da verdade.

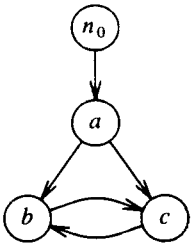


Fig. 10.75. Subgrafo proibido para grafos de fluxo redutíveis.

10.20 Suponha que a transmissão de parâmetros seja por valor e não por referência. Podem esses dois nomes ser sinônimos (pseudônimos) um do outro? E se for usada a ligação de cópia e restauração (transmissão por valor-resultado)?

10.21 Qual é a profundidade do grafo de fluxo do Exercício 10.1(c)?

****10.22** Prove que a profundidade de um grafo de fluxo redutível nunca é menor do que o número de vezes que análise de intervalos precisa ser realizada de forma a produzir um único nó.

***10.23** Generalize o algoritmo de análise de fluxo de dados baseado em estruturas da Seção 10.8 para uma estrutura geral de fluxo de dados no sentido da Seção 10.11. Que suposições a respeito de F e de $\wedge$ V. terá que fazer para assegurar que seu algoritmo funciona?

***10.24** Uma estrutura interessante e poderosa é obtida supondo-se que os "valores" a serem propagados sejam todas as possíveis partições das expressões, de tal forma que duas expressões estão na mesma classe se e somente se têm o mesmo valor. Para evitar ter que listar toda a infinidade de possíveis expressões, podemos representar tais valores listando somente as expressões mínimas que sejam equivalentes a alguma outra expressão. Por exemplo, se executarmos os enunciados

```
A := B
C := A + D
```

então teremos as seguintes equivalências minimais: `A ≡ B` e `C ≡ A + D`. A partir dessas seguem-se outras equivalências, tais como `C ≡ B + D` e `A + E ≡ B + E`, mas não há necessidade de listá-las explicitamente.
a) Qual é o operador apropriado de reunião, ou confluência, para esta estrutura?
b) Forneça uma estrutura de dados para representar valores e um algoritmo para implementar o operador de reunião.
c) Quais são as funções apropriadas para associar aos enunciados? Explique o efeito que a função associada às atribuições, tais como `A := B+C`, deveria ter numa partição.
d) A estrutura é distributiva? Monótona?

10.25 Como V. usaria os dados capturados pela estrutura do Exercício 10.24 para realizar
a) eliminação de subexpressões comuns?
b) propagação de cópias?
c) transposição para constantes?

***10.26** Forneça provas formais do seguinte a respeito da relação $\leq$ sobre reticulados.

a) $a \leq b$ e $a \leq c$ implica $a \leq (b \wedge c)$

b) $a \leq (b \wedge c)$ implica $a \leq b$

c) $a \leq b$ e $b \leq c$ implica $a \leq c$

d) $a \leq b$ e $b \leq a$ implica $a = b$

****10.27** Mostre que a seguinte condição é necessária e suficiente para o algoritmo iterativo de fluxo de dados, com ordenamento em profundidade, convergir em "2 mais a profundidade" passagens. Para todas as funções f e g e valor a:

$$f(g(a)) \geq f(a) \wedge g(a) \wedge a$$

10.28 Mostre que as estruturas de definições incidentes e de expressões disponíveis satisfazem a condição do Exercício 10.27. Nota: De fato, essas estruturas convergem em 1 mais a profundidade passagens.

****10.29** É a condição do Exercício 10.27 implicada pela monotonicidade? Pela distributividade? E vice-versa?

10.30 Vemos na Fig. 10.76 dois blocos básicos, o primeiro código "inicial" e o segundo, uma versão otimizada.

a) Construa GDAs para os blocos da Fig. 10.76(a) e (b). Verifique que, na suposição de somente J estar viva à saída, esses dois blocos são equivalentes.

b) Anote o GDA com os tempos aos quais o valor de cada variável é conhecido a cada nó.

c) Indique, para um erro que ocorra a cada um dos enunciados (1') a (4'), a que enunciado da Fig. 10.76(a) deveríamos dizer que o mesmo ocorreu.

d) Para cada um dos erros na parte (c), indique para que variáveis da Fig. 10.76(a) é possível computar um valor e como fazemos isso.

e) Vamos supor que nos permitíssemos usar leis algebricamente válidas como "se $a + b = c$ então $a = c - b$". A sua resposta ao item (d) mudaria?

10.31 Generalize o Exemplo 10.14 para que leve em conta um conjunto arbitrário de enunciados *break*. Igualmente, generalize para permitir enunciados de continuação (*continue*) que não encerram o laço mais interno mas fazem o controle prosseguir diretamente para a próxima iteração do laço. Sugestão: Use as técnicas desenvolvidas na Seção 10.10 para os grafos de fluxo redutíveis.

10.32 Mostre que no Algoritmo 10.3 os conjuntos de definições *entrada* e *saída* nunca decrescem. Semelhantemente, mostre que no Algoritmo 10.4 esses algoritmos nunca crescem.

10.33 Generalize o Algoritmo 10.9 para eliminação das variáveis de indução para o caso onde as constantes multiplicativas possam ser negativas.

10.34 Generalize o Algoritmo para determinar para o que os apontadores podem apontar, a partir da Seção 10.8, para o caso onde os apontadores são permitidos apontar para outros apontadores.

***10.35** Ao estimar cada um dos seguintes conjuntos, informe se as estimativas muito grandes ou muito pequenas são conservativas. Explique sua resposta em termos do uso pretendido das informações.

```
1)  E := A+B
2)  F := E-C
3)  G := F*D
4)  H := A+B
5)  I := I-C
6)  J := I+G
```

(a) INICIAL

```
1')  E := A+B
2')  E := E-C
3')  F := E*D
4')  J := E+F
```

(b) OTIMIZADO

Fig. 10.76. Código inicial e código otimizado.

a) Expressões disponíveis

b) Variáveis modificadas por um procedimento

c) Variáveis não mudadas por um procedimento

d) Variáveis de indução não modificadas por um procedimento

e) Variáveis de indução pertencentes a uma dada família

f) Enunciados de cópia que atingem a um dado ponto.

***10.36** Refine o Algoritmo 10.12 para que compute os pseudônimos de uma dada variável a um dado ponto.

***10.37** Modifique o Algoritmo 10.12 para os casos em que são passados parâmetros

a) por valor

b) por cópia e restauração (valor resultado)

***10.38** Prove que o Algoritmo 10.13 converge para um superconjunto (não necessariamente próprio) das variáveis verdadeiramente modificadas.

***10.39** Generalize o Algoritmo 10.13 para que determine as variáveis modificadas no caso em que as variáveis do tipo procedimento sejam permitidas.

***10.40** Prove que, em cada grafo de intervalos, cada nó representa uma região do grafo de fluxo original.

10.41 Prove que o Algoritmo 10.16 computa corretamente o conjunto de dominadores de cada nó.

***10.42** Modifique o Algoritmo 10.17 (definições incidentes baseadas em estruturas) para computar as definições incidentes para pequenas regiões designadas apenas, sem requerer que todo o grafo de fluxo esteja presente na memória de uma vez. Certifique-se de que seu resultado é conservativo. Adapte seu algoritmo para as expressões disponíveis. O que é mais propenso a fornecer informações úteis?

***10.43** Na Seção 10.10, propusemos um algoritmo de aceleração baseado na combinação de uma redução T_1 com uma redução T_2. Prove a correção da modificação.

10.44 Generalize o método iterativo da Seção 10.11 para problemas de fluxo para trás.

****10.45** Prove que, quando o Algoritmo 10.18 converge, a solução resultante é $\leq$ que a solução *scp*, mostrando que, para cada percurso P de comprimento i, após i iterações, $entrada[B_i] \leq f_p[\top]$.

10.46 Na Fig 10.77 está um grafo de fluxo de um programa na linguagem hipotética introduzida na Seção 10.12. Encontre a melhor estimativa dos tipos de cada variável usando o Algoritmo 10.19.

NOTAS BIBLIOGRÁFICAS

Informações adicionais a respeito da otimização de código podem ser encontradas em Cocke e Schwartz [1970], Abel e Bell [1972], Schaefer [1973], Hecht [1977] e Muchnick e Jones [1981]. Allen [1975] providencia uma bibliografia para a otimização de programas.

Muitos compiladores otimizantes são descritos na literatura. Ershov [1966] discute um compilador precursor que usava técnicas de otimização sofisticadas. Lowry e Medlock [1969] e Scarborough e Kolsky [1980] detalham a construção de compiladores otimizantes em Fortran. Busam e Englund [1969] e Metcalf [1982] descrevem técnicas adicionais para a otimização em Fortran. Wulf et al. [1975] discute o projeto de um influente compilador otimizante para Bliss.

Allen et al. [1980] descreve um sistema que foi construído para experimentar otimizações de programas. Cocke e Markstein [1980] reportam a respeito da efetividade de várias otimizações para uma linguagem semelhante a PL/I. Anklam, Cutler, Heinen e MacLaren [1982] descrevem a implementação de transformações otimizantes que foram usadas em compiladores para PL/I e C. Auslander e Hopkins [1982] reportam a respeito de um compilador para uma variante de PL/I que usa um algoritmo simples para produzir um código intermediário em baixo nível e é otimizado por transformações globais. Freudenberger,

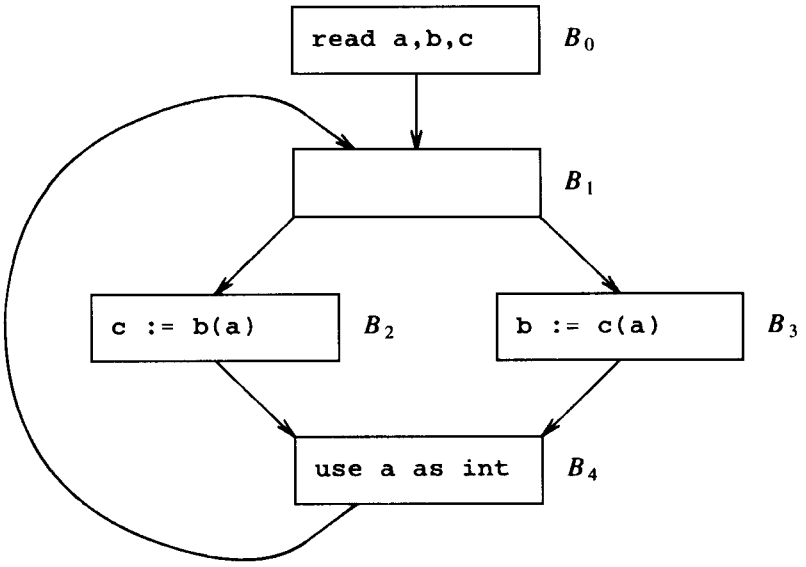


Fig. 10.77. Programa exemplo para a inferência de tipos.

Schwartz e Sharir [1983] descrevem experiências com um otimizador para SETL. Chow [1983] relata experimentos com um otimizador portável para Pascal, independente da máquina. Powell [1984] descreve um compilador otimizante, portável, independente da máquina, para Modula-2.

O estudo sistemático das técnicas de análise de dados começa com Allen [1970] e Cocke [1970], desde então publicados juntos como Allen e Cocke [1976], apesar dos vários métodos de análise de fluxo de dados estarem em uso antes dessa época.

A análise de fluxo de dados dirigida pela sintaxe, como introduzida na Seção 10.5, foi usada em Bliss (Wulf et al. [1975], Geschke [1972]), SIMPL (Zelkowitz e Bail [1974]) e Modula-2 (Powell [1984]). Discussões adicionais desta família de algoritmos aparecem em Hecht e Schaffer [1975], Hecht [1977] e Rosen [1977].

A abordagem iterativa à análise de fluxo de dados, discutida na Seção 10.6, foi rastreada até Vyssotsky (ver Vyssotsky e Wegner [1963]), que usou o método num compilador Fortran. O uso do ordenamento em profundidade para melhorar a eficiência é proveniente de Hecht e Ullman [1975].

A abordagem da análise de intervalos para a análise de fluxo de dados foi pioneirizada por Cocke [1970]. Kennedy [1971] originou o uso da análise de intervalos para problemas de fluxo para trás, como as variáveis vivas. Existe razão para se acreditar, seguindo-se Kennedy [1976], que os métodos baseados em intervalos são um tanto mais eficientes do que os iterativos, se a linguagem que está sendo otimizada tende a produzir poucos grafos de fluxo não redutíveis, se algum. A variante usada aqui, baseada em T_1 e T_2, é proveniente de Ullman [1973]. Uma versão um tanto mais rápida, que tira vantagem do fato de que a maioria das regiões tem uma única saída, foi fornecida em Graham e Wegman [1976].

A definição original de um grafo de fluxo redutível, aquele que se transforma num único nó sob a análise de intervalos iterada, é proveniente de Allen [1970]. Caracterizações equivalentes são encontradas em Hecht e Ullman [1972, 1974], Kasyanov [1973] e Tarjan [1974b]. A divisão de nós para grafos não redutíveis é de Cocke e Miller [1969].

A idéia de que o fluxo de controle estruturado é modelado por grafos de fluxo redutíveis é expressa em Kosaraju [1974], Kasami, Peterson e Tokura [1973] e Cherniavsky, Henderson e Keohane [1976]. Baker [1977] descreve seus usos num algoritmo de estruturação de programas.

A abordagem teorética dos reticulados à análise iterativa do fluxo de dados começou com Kildall [1973]. Tennenbaum [1974] e Wegbreit [1975] são formulações similares. A versão eficiente do Algoritmo de Kildall, na qual a ordem em profundidade é usada, é proveniente de Kam e Ullman [1976].

Enquanto Kildall assumiu a condição de distributividade (a qual suas estruturas, tais como a de cômputo de constantes do Exemplo 10.42, não atendem de fato), a adequação da monotonicidade foi percebida num número de *papers* que forneciam algoritmos de fluxo de dados, tais como Tennenbaum [1974], Schwartz [1975a,b], Graham e Wegman [1976], Jones e Muchnick [1976], Kam e Ullman [1977] e Cousot e Cousot [1977].

Uma vez que algoritmos diferentes requerem diferentes suposições a respeito dos dados, a teoria sobre que propriedades são necessitadas para certos algoritmos foi desenvolvida por Kam e Ullman [1977], Rosen [1980] e Tarjan [1981].

Outra direção que se seguiu a partir do *paper* de Kildall foi a melhoria de algoritmos para se lidar com problemas particulares de fluxo de dados (por exemplo, o Exemplo 10.42) que ele mesmo introduziu. Uma idéia-chave é a de que o reticulado de elementos não precisa ser tratado como atômico, mas que podemos explorar o fato de que são realmente mapeamentos de variáveis para valores. Ver Reif e Lewis [1977] e Wegman e Zadeck [1985]. Kou [1977] também explora a idéia para problemas mais convencionais.

Kennedy [1981] é um levantamento das técnicas de análise de fluxo de dados e Cousot [1981] faz uma pesquisa extensiva das idéias teoréticas a respeito dos reticulados.

Gear [1965] introduziu as otimizações básicas de laços de movimentação de código e uma forma limitada de eliminação de variáveis de indução. Allen [1969] é um *paper* fundamental na otimização de laços; Allen e Cocke [1972] e Waite [1976b] são pesquisas mais extensivas das técnicas na área. Morel e Renvoise [1979] descrevem um algoritmo que elimina simultaneamente cômputos redundantes e invariantes dos laços.

A discussão da eliminação das variáveis de indução na Seção 10.7 está baseada em Lowry e Medlock [1969]. Ver Allen, Cocke e Kennedy [1981] para algoritmos mais poderosos.

Um algoritmo para alguns dos problemas de laços não discutidos em detalhes aqui, tais como encontrar se existe um percurso de *a* até *b* que não passe por *c*, pode ser resolvidos por um algoritmo eficiente de Wegman [1983].

O uso dos dominadores, tanto para a descoberta de laços quanto para realizar a movimentação de código foi pioneirizado por Lowry e Medlock [1969], apesar de ambos atribuírem a idéia geral a Prosser [1959].

O Algoritmo 10.16 para se encontrar os dominadores foi descoberto independentemente por Purdom e Moore [1972] e Aho e Ullman [1973a]. O uso do ordenamento em profundidade para acelerar o algoritmo é proveniente de Hecht e Ullman [1975], enquanto que a forma assintoticamente mais eficiente de realizar a tarefa é proveniente de Tarjan [1974a]. Lengauer e Tarjan [1979] descrevem um algoritmo eficiente para encontrar os dominadores, o qual é adequado para o uso prático.

O estudo da polionomia e da análise interprocedimental começa com Spillman [1971] e Allen [1974]. Foram desenvolvidos alguns métodos mais poderosos do que aqueles da Seção 10.8. Em geral, lidam com a relação de pseudônimos a cada ponto do programa de forma a evitar alguns dos pares de pseudônimos impossíveis que nosso algoritmo simples "descobre". Esses trabalhos incluem Barth [1978], Banning [1979] e Weihl [1980]. Ver também Ryder [1979] na construção dos grafos de chamada.

Um tema similar à análise interprocedimental, o efeito das exceções na análise de fluxo de dados de programas é discutido por Hennessy [1981].

O *paper* fundamental sobre os tipos de análise de fluxo de dados é Tennenbaum [1974], no qual nossa discussão da Seção 10.12 está baseada. Kaplan e Ullman [1980] fornecem um algoritmo mais poderoso para a detecção de tipos.

A discussão da depuração simbólica de código otimizado na Seção 10.13 é proveniente de Hennessy [1982].

Tem havido um número de *papers* que tentam avaliar a melhoria devido às várias otimizações. O valor de uma otimização parece ser altamente dependente da linguagem que está sendo compilada. O leitor pode desejar consultar o estudo clássico da otimização de Fortran em Knuth [1971b] ou os *papers* por Gajewska [1975], Palm [1975], Cocke e Kennedy [1976] Cocke e Markstein [1980], Chow [1983], Chow e Hennessy [1984] e Powell [1984].

Outro tópico na otimização que não cobrimos aqui é a otimização das linguagens "em nível muito alto", tais como a linguagem teorética de conjuntos SETL, onde mudamos realmente os algoritmos subjacentes e as estruturas de dados. Uma otimização central nesta área é a eliminação generalizada de variáveis de indução, como em Earley [1975b], Fong e Ullman [1976], Paige e Schwartz [1977] e Fong [1979].

Outro passo-chave na otimização de linguagens de nível muito alto é a seleção das estruturas de dados; este tópico é coberto em Schwartz [1975a,b], Low e Rovner [1976] e Schonberg, Schwartz e Sharir [1981].

Não tocamos, também, nos temas da otimização incremental de código, onde pequenas modificações ao programa não requerem uma completa reotimização. Ryder [1983] discute a análise de fluxo de dados incremental, enquanto que Pollock e Soffa [1985] tentam realizar a otimização incremental de blocos básicos.

Finalmente, deveríamos mencionar algumas das muitas outras formas sob as quais as técnicas de análise de fluxo de dados têm sido usadas. Backhouse [1984] usa-as nos grafos de transição associados aos analisadores sintáticos para realizar a recuperação de erros. Harrison [1977] e Suzuki e Ishihata [1977] discutem seu uso na verificação dos limites de *arrays* em tempo de compilação.

Um dos usos mais significativos da análise de fluxo de dados, fora da otimização de código, está na área da verificação estática (tempo de compilação) de erros. Fosdick e Osterweil [1976] é um *paper* fundamental, enquanto que Osterweil [1981], Adrion, Bronstad e Cherniavsky [1982] e Freudenberger [1984] fornecem alguns dos desenvolvimentos mais recentes.

CAPÍTULO 11

DESEJA ESCREVER UM COMPILADOR?

Tendo examinado os princípios, técnicas e ferramentas para o projeto de compiladores, suponhamos que desejemos escrever um compilador. Com algum planejamento antecipado, a implementação pode prosseguir mais rápida e suavemente do que em caso contrário. Este breve capítulo levanta alguns temas de implementação que emergem na construção dos compiladores. Muito da discussão está focalizado na escrita de compiladores usando o sistema operacional UNIX e suas ferramentas.

11.1 PLANEJANDO UM COMPILADOR

Um novo compilador pode estar destinado a processar uma nova linguagem-fonte ou a produzir um novo código-alvo. Usando a estrutura adotada neste livro, obtemos, em última análise, um projeto para um compilador que consiste em um conjunto de módulos. Vários fatores distintos impactam o projeto e a implementação desses módulos.

Temas das Linguagens-Fonte

O "tamanho" de um compilador afeta o tamanho e número dos módulos. Apesar de não existir definição precisa para o tamanho de uma linguagem, é visível que um compilador para uma linguagem como Ada ou PL/I é maior e mais difícil de implementar do que um compilador para uma pequena linguagem como Ratfor (um pré-processador Fortran "racional", Kernighan [1975]) ou EQN (uma linguagem para a composição tipográfica de textos matemáticos).

Um outro importante fator é a extensão em que a linguagem-fonte irá mudar ao longo do curso da construção do compilador. Apesar da especificação da linguagem-fonte parecer imutável, poucas linguagens permanecem inalteradas durante a vida de um compilador. Mesmo as linguagens amadurecidas evoluem, apesar de lentamente. Fortran, por exemplo, mudou consideravelmente a partir da linguagem que era em 1957; os laços, os enunciados Hollerith e os comandos condicionais em Fortran 77 são bem diferentes daqueles na linguagem original. Rosler [1984] relata a evolução de C.

Por outro lado, uma linguagem nova, experimental, pode sofrer transformações dramáticas à medida que seja implementada. Uma forma de se criar uma nova linguagem é evoluir de um compilador de um protótipo de trabalho de uma linguagem para um que atenda às necessidades de certos grupos de usuários. Muitas dessas "pequenas" linguagens, desenvolvidas inicialmente no sistema UNIX, tais como AWK e EQN, foram criadas dessa forma.

Conseqüentemente, um escritor de compilador poderia antecipar pelo menos um certo número de mudanças na definição da linguagem-fonte, ao longo do tempo de vida do compilador. O projeto modular e o uso de ferramentas podem auxiliar o escritor do compilador a lidar com essas mudanças. Por exemplo, usando-se geradores para implementar os analisadores léxico e sintático de um compilador, permite-se que o autor do mesmo acomode as mudanças sintáticas na definição da linguagem mais prontamente do que quando ambos são escritos diretamente no código.

Os Temas das Linguagens-Alvo

A natureza e as limitações da linguagem-alvo e o ambiente em tempo de execução têm que ser considerados cuidadosamente, já que possuem uma forte influência no projeto do compilador e nas estratégias de geração de código que deveriam usar. Se a linguagem-alvo for nova, o escritor do compilador é aconselhado a assegurar que a mesma está correta e que as suas seqüências de temporização são bem compreendidas. Uma nova máquina ou um novo montador pode ter erros que um compilador é inclinado a desencobrir. Os erros na linguagem-alvo podem agravar grandemente a tarefa de depuração do compilador em si.

Uma linguagem-fonte de sucesso está inclinada a ser implementada em várias máquinas-alvo. Se a linguagem sobreviver, os compiladores para a mesma necessitarão gerar código para várias gerações de máquinas-alvo. A evolução posterior no *hardware* das máquinas parece certa, e, por conseguinte, compiladores reorientáveis estão inclinados a desempenhar algum papel. Conseqüentemente, o projeto de linguagens intermediárias é importante, na medida em que confina detalhes específicos de máquina para um pequeno número de módulos.

Critérios de Desempenho

Existem vários aspectos do desempenho do compilador: velocidade, qualidade do código, diagnósticos de erro e condições de manutenção. As barganhas entre esses critérios não são tão claramente divididas e a especificação do compilador pode deixar muitos desses parâmetros indefinidos. Por exemplo, é a velocidade de compilação mais importante do que a velocidade do código-alvo? Quão importantes são as mensagens e a recuperação de erros?

A velocidade do compilador pode ser melhorada através da redução do número de módulos e passagens, tanto quanto possível, talvez até o ponto de se gerar uma linguagem de máquina diretamente

numa única passagem. No entanto, esse enfoque pode produzir um compilador que não gere um código-alvo de alta qualidade, nem um que seja particularmente fácil de manter.

Existem dois aspectos da portabilidade: reorientabilidade e a reinstabilidade. Um compilador reorientável é aquele que pode ser modificado facilmente de forma a gerar código para uma nova linguagem-alvo. Um compilador reinstalável é o que pode ser facilmente movido de forma a que rode numa nova máquina. Um compilador portável pode não ser tão eficiente quanto um projetado para uma máquina específica, porque um compilador para uma única máquina pode seguramente realizar suposições específicas sobre a máquina-alvo que um compilador portável não pode.

11.2 ABORDAGENS PARA O DESENVOLVIMENTO DOS COMPILADORES

Existem vários enfoques diferentes que o autor de um compilador pode adotar para implementá-lo. A forma mais simples é reorientar ou reinstalar um compilador existente. Se não há um compilador adequado, o autor pode adotar a organização de um compilador conhecido para uma linguagem similar e implementar os componentes correspondentes, usando ferramentas de geração de componentes ou implementando-os à mão. É relativamente raro que uma organização de compilador completamente nova seja requerida.

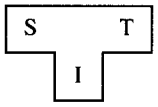
Não importa que enfoque seja adotado, escrever compiladores é um exercício em engenharia de *software*. As lições a partir de outros esforços na área de *software* (veja, por exemplo, Brooks [1975]) podem ser aplicadas para melhorar a confiabilidade e as condições de manutenção do produto final. Um projeto que acolha mudanças prontamente irá permitir que o compilador evolua junto com a linguagem. O uso de ferramentas de construção de compiladores pode ser um auxílio significativo nesse aspecto.

Bootstrapping

Um compilador é um programa complexo o suficiente para que desejemos escrevê-lo numa linguagem mais amigável do que uma linguagem de montagem. No ambiente de programação UNIX, os compiladores são usualmente escritos em C. Mesmo os compiladores C são escritos em C. Usar as facilidades oferecidas por uma linguagem para compilar a si mesma é a essência do *bootstrapping**. Aqui, iremos examinar o uso do *bootstrapping* para criar compiladores e migrá-los de uma máquina para outra através da modificação da interface de retaguarda. As idéias básicas sobre o *bootstrapping* são conhecidas desde meados dos anos 50. (Strong et al. [1958].)

O *bootstrapping* pode levantar a questão, "como foi compilado o primeiro compilador?", que soa muito como, "o que veio primeiro, a galinha ou o ovo?", mas é mais fácil de responder. Para uma resposta, consideramos como Lisp se tornou uma linguagem de programação. McCarthy [1981] nota que, ao final de 1958, Lisp foi usada como uma notação para escrever funções; eram traduzidas à mão em linguagem de montagem e executadas. A implementação de um interpretador para Lisp ocorreu inesperadamente. McCarthy desejava mostrar que Lisp era uma notação para descrever funções "mais ajustada do que as máquinas de Turing ou as definições recursivas usadas na teoria das funções recursivas"; e, por conseguinte, escreveu uma função denominada *eval*[*e, a*] em Lisp, que tomava uma expressão Lisp *e* como argumento. S. R. Russell observou que *eval* poderia servir como um interpretador para Lisp, codificado à mão, e, conseqüentemente, criou uma linguagem de programação com um interpretador. Como mencionado na Seção 1.1, ao invés de gerar um código-alvo, um interpretador efetivamente realiza as operações do programa-fonte.

Para os propósitos de *bootstrapping*, um compilador é caracterizado por três linguagens: a linguagem-fonte S que o mesmo compila, a linguagem-alvo T para a qual gera código e a linguagem de implementação I na qual é escrito. Representamos as três linguagens pelo seguinte diagrama, chamado de *diagrama-T*, por causa de sua conformação (Bratman [1961]).



Dentro do texto, abreviamos o diagrama-T acima como S_IT. As três linguagens S, I e T podem ser todas completamente diferentes. Por exemplo, um compilador pode rodar numa máquina e produzir código-alvo para uma outra máquina. Tal compilador é freqüentemente chamado de *compilador cruzado*.

Vamos supor que escrevemos um compilador cruzado para uma nova linguagem L, numa linguagem de implementação S, para gerar código para a máquina N; isto é, criamos L_SN. Se um compilador existente para S rodar na máquina M e gerar código para M é caracterizado como S_MM. Se L_SN é executado através de S_MM, obtemos o compilador L_MN, isto é, um compilador de L para N, que roda em M. Este processo é ilustrado na Fig. 11.1, colocando-se juntos os diagramas-T para esses compiladores.

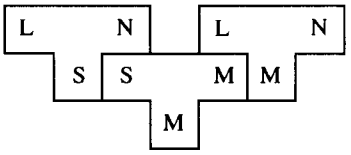


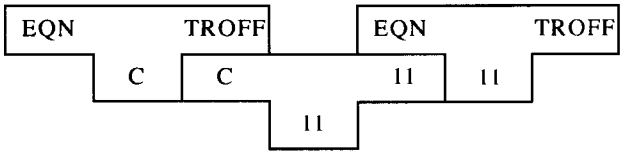
Fig. 11.1. Compilando um compilador.

Quando os diagramas-T são colocados juntos, como na Fig. 11.1, a linguagem de implementação S do compilador L_SN precisa ser a mesma que a linguagem-fonte do compilador existente S_MM e a linguagem-alvo M do compilador existente precisa ser a mesma que a linguagem de implementação da forma traduzida L_MN. Um trio de diagramas-T, tal como o da Fig. 11.1, pode ser pensado como a equação

$$L_SN + S_MM = L_MN$$

Exemplo 11.1. A primeira versão do compilador EQN (veja a Seção 12.1) tinha C como a linguagem de implementação e gerava comandos para o formatador de textos TROFF. Como mostrado no diagrama seguinte, um compilador cruzado para EQN, rodando num PDP-11, foi obtido rodando-se EQN_C TROFF através do compilador $C_{11}11$ no PDP-11.

Uma forma de *bootstrapping* constrói um compilador para subconjuntos cada vez maiores de uma linguagem. Suponhamos que uma

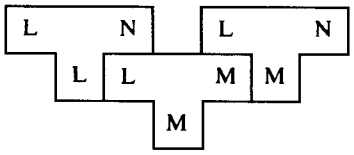


*O termo é originário de duas palavras: *boot* (bota, coturno) e *strap* (cadarço, barbante) e no jargão militar significa a seqüência de atividades que um soldado deve realizar para que, dado um toque de alerta, o mesmo esteja pronto para o combate, isto é, "com as botas amarradas e o uniforme vestido", uma vez que estivesse deitado e dormindo. No nosso caso, significa como "acordar" o compilador do zero. (N. do T.)

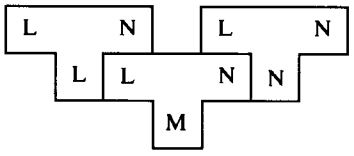
nova linguagem L deva ser implementada na máquina M. Como primeiro passo, poderíamos escrever um pequeno compilador que traduzisse um subconjunto S de L no código-alvo para M; isto é, um compilador $S_M M$. Em seguida, usamos o subconjunto S para escrever um compilador $L_S M$ para L. Quando $L_S M$ é rodado através de $S_M M$, obtemos uma implementação de L, $L_M M$. Neliac foi uma das primeiras linguagens a ser implementada em sua própria linguagem (Huskey, Halstead e McArthur [1960]).

Wirth [1971] nota que Pascal foi primeiro implementada escrevendo-se um compilador na própria linguagem Pascal. O compilador foi, então, traduzido "à mão" numa linguagem de baixo nível disponível sem qualquer tentativa de otimização. O compilador era para um subconjunto "(>60%)" de Pascal; vários estágios de *bootstrapping* mais tarde e um compilador todo de Pascal foi obtido. Lecarme e Peyrolle-Thomas [1978] sumariza métodos que foram usados para realizar o *bootstrapping* dos compiladores Pascal.

Para que as vantagens do *bootstrapping* sejam realizadas integralmente, um compilador precisa ser escrito na mesma linguagem que compila. Vamos supor que escrevemos um compilador $L_L N$, para a linguagem L, em L, de forma a gerar código para a máquina N. O desenvolvimento tem lugar na máquina M, onde um compilador existente $L_M M$, para L, roda e gera código para M. Compilando-se primeiro $L_L L$ com $L_M M$, obtemos um compilador cruzado $L_M N$ que roda em M, mas produz código para N:



O compilador $L_L N$ pode ser compilado uma segunda vez, desta vez usando o compilador cruzado gerado:



O resultado da segunda compilação é um compilador $L_N N$, que roda em N e gera código para N. Existe um número de aplicações úteis deste processo em duas etapas, de forma que o escreveremos como na Fig. 11.2.

Exemplo 11.2. Este exemplo é motivado pelo desenvolvimento do compilador Fortran H (veja a Seção 12.4). "O compilador foi, ele próprio, escrito em Fortran e passou por um processo de *bootstrapping* três vezes." A primeira vez foi para converter do IBM 7094 para o Sistema /360 — uma tarefa árdua. A segunda vez, para otimizar a si mesmo, o que reduziu o tamanho do compilador de aproximadamente 550K para algo em torno de 400K *bytes*" (Lowry e Medlock [1969]).

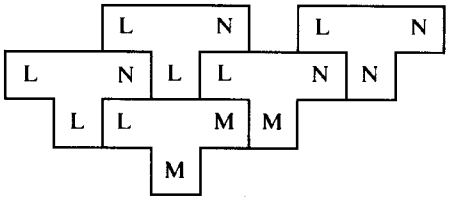
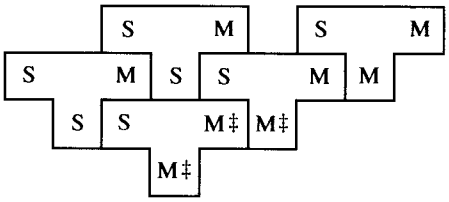


Fig. 11.2. Realizando o *bootstrap* de um compilador.

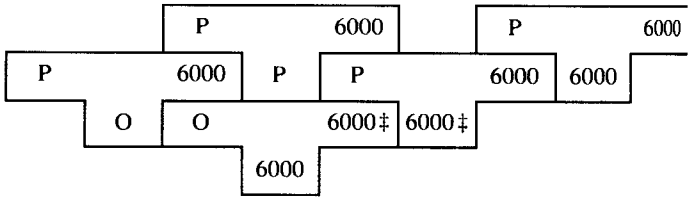
Usando as técnicas de *bootstrapping*, um compilador otimizante pode otimizar a si próprio. Suponhamos que todo o desenvolvimento seja feito na máquina M. Temos $S_S M$, um bom compilador otimizante para a linguagem S, escrito em S, e desejamos $S_M M$, um bom compilador otimizante para S, escrito em M.

Podemos criar $S_{M‡} M‡$, um rápido e "sujo" compilador para S em M que não só gera um código de baixa qualidade, mas também leva muito tempo para fazê-lo. (M‡ indica uma pobre implementação em M. $S_{M‡} M‡$ é uma implementação pobre de um compilador que gera código pobre.) No entanto, podemos usar o compilador indiferente $S_{M‡} M‡$ para obter um bom compilador para S em duas etapas:



Primeiro, o compilador otimizante $S_S M$ é traduzido pelo compilador rápido e "sujo" para produzir $S_{M‡} M$, uma pobre implementação do compilador otimizante, mas que produz código de boa qualidade. O bom compilador otimizante $S_M M$ é obtido recompilando-se $S_S M$ através de $S_{M‡} M$.

Exemplo 11.3. Ammann [1981] descreve como uma implementação "limpa" de Pascal foi obtida num processo similar àquela do Exemplo 11.2. As revisões de Pascal levaram à escrita de um compilador novo em 1972, para as máquinas da série CDC 6000. No diagrama seguinte, O representa Pascal "antigo", e P a linguagem revisada.



Um compilador para o Pascal revisado foi escrito no Pascal antigo e traduzido no $P_{6000‡}6000$. Como no Exemplo 11.2, o símbolo ‡ marca uma fonte de ineficiência. O compilador velho não gerava um código eficiente o bastante. "Por conseguinte, a velocidade do compilador [$P_{6000‡}6000$] era um tanto moderada e suas exigências de memória bastante altas (Ammann [1981])." As revisões em Pascal eram pequenas o suficiente para que o compilador $P_O 6000$ pudesse ser traduzido à mão com pouco esforço no Pp6000 e rodasse através do compilador ineficiente $P_{6000‡}6000$ para obter uma implementação "limpa".

11.3 O AMBIENTE DE DESENVOLVIMENTO DE COMPILADORES

Num sentido real, um compilador é simplesmente um programa. O ambiente no qual esse programa é desenvolvido pode afetar quão rápida e confiavelmente o mesmo é implementado. A linguagem na qual o compilador é implementado é igualmente importante. Apesar dos compiladores terem sido escritos em linguagens como Fortran, uma escolha mais esclarecida para a maioria dos compiladores é uma linguagem orientada para sistemas, tal como C.

Se a linguagem-fonte, por si só, for uma nova linguagem orientada para sistemas, faz muito sentido escrever um compilador na sua

própria linguagem. Usando as técnicas de *bootstrapping*, discutidas na seção anterior, compilar o compilador auxilia na depuração do mesmo.

As ferramentas de construção de *software* no ambiente de programação podem facilitar grandemente a criação de um compilador eficiente. Ao se escrever um compilador, é usual se particionar o programa todo em módulos, onde cada módulo pode ser processado de formas bastante distintas. Um programa que gerencie o processamento desses módulos é uma ajuda indispensável ao escritor do compilador. O sistema UNIX contém um comando chamado *make* (Feldman [1979a]), que gerencia e mantém os módulos que constituem um programa de computador; *make* controla os relacionamentos entre os módulos do programa e emite apenas aqueles comandos necessitados para tornar os módulos consistentes após as mudanças terem sido realizadas.

Exemplo 11.4. O comando *make* lê a especificação das tarefas que precisam ser realizadas a partir de um arquivo chamado `makefile`. Na Seção 2.9, construímos um tradutor através da compilação de sete arquivos com um compilador C, cada um deles dependendo de um arquivo de cabeçalho global `global.h`. Para mostrar como a tarefa de juntar as partes do compilador pode ser feita por *make*, vamos supor que chamamos o compilador resultante de `trans`. A especificação de `makefile` poderia se parecer com:

```
OBJS =  lexer.o parser.o emitter.o symbol.o\
        init.o error.o main.o
trans:  $(OBJS)
        cc $(OBJS) -o trans
lexer.o parser.o emitter.o symbol.o\
        init.o error.o main.o: global.h
```

O sinal de igual à primeira linha faz com que `OBJS` à esquerda figure no lugar dos sete arquivos objeto à direita. (As linhas longas podem ser partidas colocando-se uma barra invertida ao fim da parte interrompida.) Os dois pontos na segunda linha dizem que `trans`, à esquerda, depende de todos os arquivos em `OBJS`. Tal linha de dependências pode ser seguida por um comando para construir ("make") o arquivo à esquerda dos dois pontos. A terceira linha, por conseguinte, diz que o programa-alvo `trans` é criado ligando-se os arquivos objetos `lexer.o parser.o,...,main.o`. Entretanto, *make* sabe que primeiro precisará criar os arquivos-objeto; faz isso automaticamente olhando para os arquivos-fonte correspondentes `lexer.c, parser.c,...,main.c,` e compilando cada um com o compilador C, para criar os arquivos-objeto correspondentes. A última linha de `makefile` diz que todos os sete arquivos-objeto dependem do arquivo global de cabeçalho `global.h`.

O tradutor é criado digitando-se simplesmente o comando *make*, que causa a emissão dos seguintes comandos:

```
cc -c lexer.c
cc -c parser.c
cc -c emitter.c
cc -c symbol.c
cc -c init.c
cc -c error.c
cc -c main.c
cc lexer.o parser.o emitter.o symbol.o\
      init.o error.o main.o -o trans
```

Subseqüentemente, uma compilação será refeita somente se um arquivo dependente da entrada for modificado após a última compilação. Kernighan e Pike [1984] contém exemplos do uso de *make* para facilitar a construção de um compilador. □

Um gerador de perfis de comportamento é uma ferramenta útil para a escrita de compiladores. Uma vez que um compilador tenha sido escrito, um gerador de perfis pode ser usado para determinar onde o compilador está gastando o seu tempo na medida em que compila um programa-fonte. A identificação e modificação dos pontos críticos podem acelerar um compilador por um fator de dois ou três.

Adicionalmente às ferramentas de *software*, um número de ferramentas tem sido criado especificamente para o processo de desenvolvimento de compiladores. Na Seção 3.5, descrevemos o gerador Lex, que pode ser usado para produzir automaticamente um analisador léxico a partir de uma especificação sob a forma de expressões regulares; na Seção 4.9, descrevemos o gerador Yacc, que pode ser usado para produzir automaticamente um analisador sintático LR a partir de uma descrição gramatical da sintaxe da linguagem. O comando *make* descrito acima irá invocar automaticamente Lex e Yacc sempre que necessário. Adicionalmente aos geradores de analisadores léxicos e sintáticos, os geradores de gramáticas de atributos e os de código têm sido criados para auxiliar a construção dos componentes do compilador. Muitas dessas ferramentas de construção de compiladores possuem a desejável propriedade de que irão localizar falhas na especificação dos mesmos.

Tem havido algum debate a respeito da eficiência e conveniência de geradores de programas na construção de compiladores. (Waite e Carter [1985].) O fato observado é que geradores de programas bem implementados se constituem numa ajuda significativa na produção de componentes confiáveis de compiladores. É muito mais fácil produzir um analisador sintático correto usando uma descrição gramatical da linguagem e um gerador de analisadores sintáticos, do que implementar um analisador sintático diretamente à mão. Um tema importante, entretanto, é quão bem esses geradores interfaciam-se mutuamente e aos outros programas. Um erro comum no projeto de um gerador é assumir que o mesmo é o centro do projeto. Um projeto melhor destina o gerador a produzir sub-rotinas com interfaces inteligentes que possam ser chamadas por outros programas (Johnson e Lesk [1978]).

11.4 TESTE E MANUTENÇÃO

Um compilador tem que gerar código correto. Idealmente, gostaríamos que o computador verificasse mecanicamente que o compilador implementou fielmente sua especificação. Vários artigos realmente discutem a correção dos vários algoritmos de compilação mas, infelizmente, os compiladores raramente são especificados de tal forma que uma implementação arbitrária possa ser mecanicamente verificada em confronto com uma especificação formal. Como os compiladores são usualmente funções um tanto complexas, existe também o tema de verificar se a própria especificação também está correta.

Na prática, precisamos nos voltar para algum método sistemático de teste de um compilador, de forma que aumentemos a nossa confiança em que o mesmo irá funcionar satisfatoriamente no campo. Um enfoque usado com sucesso por muitos escritores de compiladores é o teste de "regressão". Aqui, mantemos um coquetel de programas de teste e, sempre que o compilador for modificado, os programas de teste são compilados usando tanto a versão nova do compilador quanto a antiga. Quaisquer diferenças nos programas-alvo produzidos pelos dois compiladores são reportadas ao escritor do compilador. O comando do sistema UNIX *make* pode ser usado para automatizar o teste.

A escolha dos programas para incluir no coquetel é um problema difícil. Como meta, gostaríamos que os programas de teste exercitassem cada comando dentro do compilador pelo menos uma vez. Usualmente, é requerida uma grande criatividade para encontrar um tal coquetel de teste. Coquetéis de teste exaustivos têm sido construídos para várias linguagens (Fortran, TEX, C, etc.). Muitos escritores de compiladores adicionam ao teste de regressão programas que expuseram falhas nas versões anteriores de seus compiladores; é frustrante ter um velho problema ressurgindo por causa de uma nova correção.

O teste de desempenho também é importante. Alguns escritores de compiladores verificam se as novas versões do compilador geram

um código que seja aproximadamente tão bom quanto o da versão prévia, através da realização de testes de medição de tempo como parte do teste de regressão.

A manutenção de um compilador é um outro importante problema, particularmente se o compilador vai ser rodado em ambientes diferentes ou o pessoal envolvido no projeto do compilador flutua. Um elemento crucial para estarmos aptos a manter um compilador é um bom estilo de programação e uma boa documentação. Os autores conhecem um compilador que foi escrito usando-se somente sete comentários, num dos quais se lia "Este código está doente". Desnecessário dizer que um tal programa é difícil de ser mantido por qualquer um que não seja o autor original.

Knuth [1984b] desenvolveu um sistema chamado WEB que endereça o problema de documentar grandes programas escritos em Pascal. WEB facilita a programação ilustre; a documentação é desenvolvida ao mesmo tempo que o código, não num processo posterior de levantamento. Muitas das idéias em WEB podem ser aplicadas igualmente bem a outras linguagens.

CAPÍTULO 12

UM RÁPIDO EXAME DE ALGUNS COMPILADORES

Este capítulo discute a estrutura de alguns compiladores existentes para uma linguagem de formatação de textos, Pascal, C, Fortran, Bliss e Modula 2. Nossa intenção não é advogar os projetos apresentados aqui em detrimento de outros, mas, ao invés, ilustrar a variedade que é possível na implementação de um compilador.

Os compiladores para Pascal foram escolhidos porque influenciaram o projeto da própria linguagem. Os compiladores para C foram escolhidos porque C é a linguagem primária de programação no sistema operacional UNIX. O compilador Fortran H foi escolhido porque influenciou significativamente o desenvolvimento das técnicas de otimização. BLISS/11 foi escolhida para ilustrar o projeto de um compilador cuja meta é otimizar o espaço. O compilador DEC para Modula 2 foi escolhido porque usa técnicas relativamente simples para produzir um excelente código e foi escrito por uma pessoa em poucos meses.

12.1 EQN, UM PRÉ-PROCESSADOR PARA COMPOSIÇÃO DE TIPOS MATEMÁTICOS

O conjunto de possíveis entradas para um número de programas de computador pode ser visto como uma pequena linguagem. A estrutura do conjunto pode ser descrita por uma gramática e uma tradução dirigida pela sintaxe pode ser usada para especificar precisamente o que o programa faz. A tecnologia de compiladores pode, então, ser aplicada para implementar o programa.

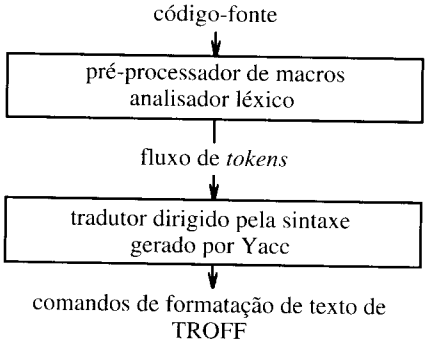


Fig. 12.1. Implementação de EQN.

Um dos primeiros compiladores para pequenas linguagens no ambiente de programação UNIX foi EQN, por Kernighan e Cherry [1975]. Como descrito brevemente na Seção 1.2, EQN toma uma entrada como "`E sub 1`" e gera comandos para o formatador de texto TROFF, produzindo uma saída da forma "E_1".

A implementação de EQN é delineada na Fig. 12.1. O pré-processamento de macros (ver a Seção 1.4) e a análise léxica são feitas concomitantemente. O fluxo de *tokens*, após a análise léxica, é traduzido, durante a análise sintática, nos comandos de formatação de texto. O tradutor é construído usando o gerador de analisadores sintáticos Yacc, descrito na Seção 4.9.

O enfoque de tratar a entrada para EQN como uma linguagem e aplicar a tecnologia de compiladores para construir um tradutor possui vários benefícios notados pelos autores.

1. *Facilidade de implementação.* "Construção de sistema operativo, suficiente para exercitar os exemplos significativos exigidos, talvez, por uma pessoa/mês".

2. *Evolução da linguagem.* Uma definição dirigida pela sintaxe facilita as mudanças na linguagem de entrada. Durante anos EQN evoluiu em resposta às necessidades dos usuários.

Os autores concluem observando que "definir uma linguagem e construir um compilador para a mesma usando um compilador de compiladores parece ser a única maneira sensível de realizar o negócio".

12.2 COMPILADORES PARA PASCAL

O projeto de Pascal e o desenvolvimento do primeiro compilador para o mesmo "foram interdependentes", como observa Wirth [1971]. É, por conseguinte, instrutivo examinarmos a estrutura de compiladores para a linguagem escrita por Wirth e seus colegas. O primeiro (Wirth [1971] e o segundo compiladores (Ammann [1981, 1977]) geravam código absoluto de máquina para os equipamentos máquinas da série CDC 6000. Experimentos de portabilidade com o segundo compilador levaram ao compilador Pascal-P, que gera um código, chamado P-code, para uma máquina de pilha abstrata (Nori *et al.* [1981]).

Cada um dos compiladores acima é de uma passagem, organizado em torno de um analisador sintático de descendência recursiva, como a interface de vanguarda "neném" do Capítulo 2. Wirth [1971] observa que "tornou-se relativamente fácil moldar uma linguagem de

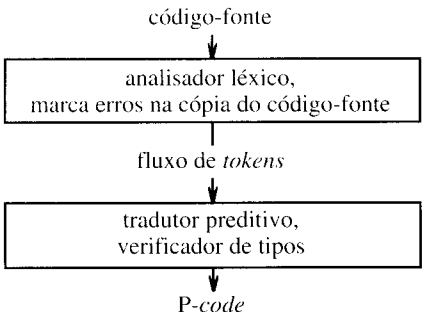


Fig. 12.2. Compilador Pascal-P.

acordo com [as restrições do método de análise sintática]". A organização do compilador Pascal-P é mostrada na Fig. 12.2.

As operações básicas da máquina abstrata de pilha usadas pelo compilador Pascal-P refletem as necessidades de Pascal. A memória da máquina é organizada em quatro áreas:

1. código para procedimentos,
2. constantes,
3. uma pilha para os registros de ativação e
4. um *heap* para os dados alocados através da aplicação do operador **new**

Como os procedimentos podem ser aninhados em Pascal, o registro de ativação para um procedimento contém tanto os elos de acesso quanto os de controle. Uma chamada de procedimento é traduzida numa instrução de "marcar pilha" para a máquina abstrata, com os elos de acesso e de controle como parâmetros. O código para o procedimento se refere à memória para um nome local usando um deslocamento a partir de uma das extremidades do registro de ativação. A memória para os não-locais é referenciada através de um par, consistindo em um número de elos de acesso a serem atravessados, e um deslocamento, como na Seção 7.4. O primeiro compilador usou um *display* para o acesso eficiente aos não-locais.

Ammann [1981] delincia as seguintes conclusões, a partir da experiência, ao escrever o segundo compilador. Por um lado, o compilador de uma passagem era fácil de implementar e gerava a mais modesta atividade de entrada e saída (o código para um corpo de procedimento é compilado na memória e escrito como uma unidade em memória secundária). Por outro lado, a organização de uma passagem "impõe severas restrições sobre a qualidade do código gerado e padece de exigências relativamente altas de memória".

12.3 OS COMPILADORES C

C é uma linguagem de programação de propósito geral projetada por D.M. Ritchie e é usada como a linguagem primária de programação do sistema operacional UNIX (Ritchie e Thompson [1974]). O próprio UNIX é escrito em C e tem sido movido para um número de máquinas que vão desde microprocessadores a amplos *mainframes*, instalando-se primeiro um compilador C. Esta seção descreve brevemente a estrutura geral do compilador para o PDP-11 por Ritchie [1979] e a família PCC de compiladores C portáteis por Johnson [1979]. Três quartos do código de PCC são independentes da máquina-alvo. Todos esses compiladores são essencialmente de duas passagens; o compilador do PDP-11 possui uma terceira passagem opcional que realiza a otimização da

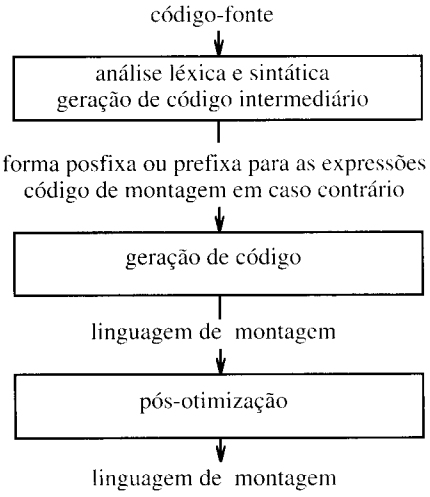


Fig. 12.3. Estrutura de passagens dos compiladores C.

saída em linguagem de máquina, como indicado na Fig. 12.3. Essa fase de otimização *peephole* elimina enunciados redundantes ou inacessíveis.

A passagem I de cada compilador realiza a análise léxica, sintática e a geração de código intermediário. O compilador PDP-11 usa a descendência recursiva para analisar sintaticamente tudo, exceto expressões, para as quais a precedência de operadores é usada. O código intermediário consiste em uma notação prefixa para as expressões e código de montagem para os enunciados de fluxo de controle. PCC usa um analisador sintático LALR(1) gerado por Yacc. Seu código intermediário consiste em uma notação prefixa para expressões e código de montagem para as outras construções. Em cada caso, a alocação de memória para os nomes não-locais é feita durante a primeira passagem e dessa forma os nomes podem ser referenciados usando-se os deslocamentos dentro do registro de ativação.

Dentro da interface de retaguarda, as expressões são representadas por árvores sintáticas. No compilador do PDP-11, a geração de código é implementada através de uma caminhada na árvore, usando-se de uma estratégia similar à do algoritmo de rotulação da Seção 9.10. As modificações naquele algoritmo foram feitas de modo a assegurar que pares de registradores estivessem disponíveis para as operações que deles necessitassem e também para tirar vantagem dos operandos que fossem constantes.

Johnson [1978] revê a influência da teoria sobre o PCC. Em ambos, PCC e PCC2, uma versão subseqüente do compilador, o código para as expressões é gerado através da reescrita de árvores. O gerador de código em PCC examina um enunciado da linguagem-fonte de cada vez, procurando repetidamente por subárvores máximas que possam ser computadas sem armazenamentos usando os registradores disponíveis. Os rótulos, computados como na Seção 9.10, identificam as subexpressões a serem computadas e armazenadas em temporários. O código para avaliar e armazenar os valores representados por essas subárvores é gerado pelo compilador à medida que as subárvores são selecionadas. A reescrita é mais evidente em PCC2, cujo gerador de código está baseado no algoritmo de programação dinâmica da Seção 9.11.

Johnson e Ritchie [1981] descrevem a influência da máquina-alvo sobre o projeto dos registros de ativação e da seqüência de chamada e retorno de procedimentos. A função de biblioteca padrão `printf` pode ter um número variável de argumentos, e o projeto da seqüência de chamada em algumas máquinas é dominado pela necessidade de permitir listas de tamanho variável para os argumentos .

12.4 OS COMPILADORES FORTRAN H

O compilador original de Fortran H, escrito por Lowry e Medlock [1969], era um abrangente e razoavelmente poderoso compilador oti-

[1]A operação de *bootstrapping* é facilitada pelo fato do compilador, escrito num subconjunto que o mesmo compila, usa o *heap* como uma pilha, de tal forma que um simples gerenciador de memória *heap* pode ser usado inicialmente.

mizante, construído utilizando-se de métodos que antecediam amplamente aqueles descritos neste livro. Várias tentativas com desempenho crescente foram feitas; uma versão estendida do compilador foi desenvolvida para o IBM/370 e uma versão "melhorada" foi desenvolvida por Scarborough e Kolsky [1980]. Fortran H oferece ao usuário a escolha de nenhuma otimização, otimização de registradores somente ou otimização completa. Um esboço do compilador no caso da otimização completa ser realizada aparece na Fig. 12.4.

O texto-fonte é tratado em quatro passagens. As duas primeiras realizam a análise léxica e a sintática, produzindo quádruplas. A passagem seguinte incorpora a otimização de código e de registradores e a passagem final gera código objeto a partir das quádruplas e atribuições de registradores.

A fase de análise léxica é um tanto incomum, uma vez que sua saída não é uma cadeia de *tokens* mas uma cadeia de "pares operador-operando", que são, grosso modo, equivalentes a um *token* operando juntamente com o *token* não-operando precedente. Deveria ser notado que, em Fortran, como na maioria das linguagens, nunca temos dois *tokens* operandos consecutivos tais como identificadores ou constantes; ao invés, tais *tokens* estão sempre separados por, pelo menos, um *token* de pontuação.

Por exemplo, o enunciado de atribuição

```
A = B(I) + C
```

seria traduzido na seqüência de pares

"enunciado de atribuição"	A
=	B
(s	I
)	–
+	C

A fase da análise léxica distingue entre um parênteses à esquerda, cuja tarefa é introduzir uma lista de parâmetros ou subscritos, daqueles cujo trabalho é o de agrupar os operandos. Por conseguinte, o símbolo "(s" pretende-se que represente um parênteses à esquerda usado como um operador de subscrição. Os parênteses à direita nunca têm um operando em seguida, e Fortran H não distingue os dois papéis para os parênteses à direita.

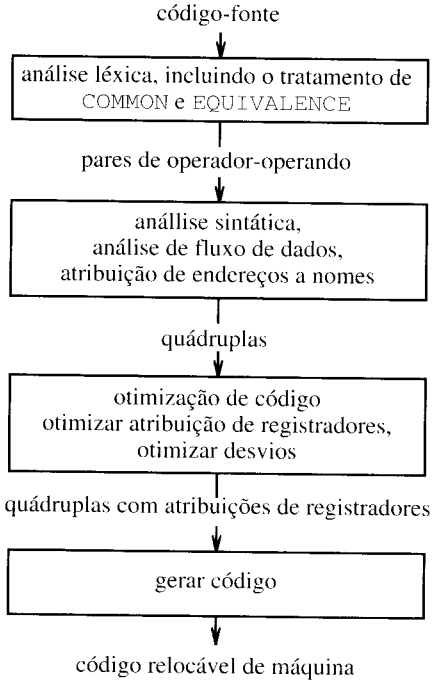


Fig. 12.4. Esboço do compilador Fortran H.

Associado à análise léxica está o processamento de enunciados COMMON e EQUIVALENCE. É possível neste estágio mapear cada bloco COMMON de memória, bem como os blocos de memória associados às sub-rotinas, e determinar a localização de cada variável mencionada pelo programa numa dessas áreas de memória estática.

Como Fortran não possui enunciados de controle estruturados, como comandos *while*, a análise sintática, exceto para expressões, é bastante direta e Fortran H simplesmente usa um analisador sintático de precedência de operadores para as expressões. Algumas otimizações locais muito simples são realizadas durante a geração de quádruplas; por exemplo, multiplicações por potências de 2 são substituídas por operações de deslocamento para a esquerda.

Otimização de Código em Fortran H

Cada sub-rotina é particionada em blocos básicos e a estrutura de laço é deduzida encontrando-se os lados do grafo de fluxo cujas cabeças dominem suas caudas, como descrito na Seção 10.4. O compilador realiza as seguintes otimizações:

1. *Eliminação de subexpressões comuns.* O compilador procura por subexpressões locais comuns e por expressões que sejam comuns a um bloco *B* e a um ou mais blocos que *B* domine. Outras instâncias de subexpressões comuns não são detectadas. Mais ainda, a detecção de subexpressões comuns é feita uma expressão a cada vez, ao invés de se usar o método do vetor de *bits*, descrito na Seção 10.6. Curiosamente, ao se desenvolver a versão "melhorada" do compilador, os autores encontraram que uma aceleração maior era possível através do uso de métodos como vetores de *bits*.

2. *Movimentação de código.* Os enunciados laço-invariantes são movidos dos laços essencialmente como descrito na Seção 10.7.

3. *Propagação de cópia.* De novo, isso é feito um enunciado de cópia de cada vez.

4. *Eliminação de variáveis de indução.* Essa otimização é realizada somente para variáveis que recebem atribuição uma vez no laço. Ao invés de usar a abordagem da "família", descrita na Seção 10.7, são feitas múltiplas passagens através do código com a finalidade de detectar variáveis de indução que pertençam à família de alguma outra variável de indução.

Apesar da análise do fluxo de dados ser feita num estilo uma de cada vez, os valores correspondentes ao que chamamos *entrada* e *saída* são armazenados como vetores de *bits*. Entretanto, no compilador original, o limite de tamanho 127 foi colocado nesses vetores, e, conseqüentemente, os grandes programas têm somente as suas variáveis mais freqüentemente usadas envolvidas em otimizações. As versões melhoradas aumentam o limite, mas não o removem.

Otimizações Algébricas

Como a medida em Fortran é freqüentemente usada para cômputos numéricos, a otimização algébrica é perigosa, uma vez que as transformações de expressões podem, na aritmética de computadores, introduzir estouros de capacidade ou perdas de precisão que não sejam visíveis se tivermos uma visão idealizada da simplificação algébrica. No entanto, as transformações algébricas envolvendo inteiros são geralmente seguras e a versão melhorada do compilador realiza algumas dessas otimizações somente no caso de referências a *arrays*.

Em geral, uma referência a *array*, como A(I,J,K) envolve um cálculo de deslocamento no qual uma expressão da forma $aI + bJ +$

$cK + d$ é computada; os valores exatos das constantes dependem da localização de A e das dimensões do *array*. Se, digamos, I e K forem constantes, quer constantes numéricas ou variáveis laço invariantes, o compilador aplica a lei comutativa e associativa para obter uma expressão de forma $bJ + e$, onde $e = aI + cK + d$.

Otimização de Registradores

Fortran H divide os registradores em três classes. Esses conjuntos de registradores são usados para a otimização local de registradores, otimização global de registradores e "otimização de desvios". O número exato de registradores em cada classe pode ser ajustado pelo compilador, dentro de limites.

Os registradores globais são alocados, numa base laço a laço, às variáveis mais freqüentemente referenciadas no laço em questão. Uma variável que se qualifica para um registrador em um laço *L*, mas não no laço que contenha imediatamente *L*, é carregada no pré-cabeçalho de *L* e armazenada à saída de *L*.

Os registradores locais são usados dentro de um bloco básico para guardar os resultados de um enunciado até que seja usado em um ou mais enunciados subseqüentes. Somente se não existirem registradores locais suficientes é que um valor temporário é armazenado. O compilador tenta computar novos valores no registrador que guarda um de seus operandos, se aquele operando for subseqüentemente eliminado, isto é, morto. Na versão melhorada, é feita uma tentativa para reconhecer a situação onde os registradores globais podem ser intercambiados com outros registradores para aumentar o número de vezes que uma operação possa ocorrer no registrador que abrigue um de seus operandos.

A otimização de desvios é um artifício do conjunto de instruções do IBM/370, que paga um prêmio significativo ao desviarmos somente para localizações que possam ser expressas como o conteúdo de algum registrador mais uma constante no intervalo de 0 a 4095. Por conseguinte, Fortran H aloca alguns registradores para guardar endereços no espaço de código, em intervalos de 4096* *bytes*, de forma a permitir desvios eficientes em todos os programas, menos os extremamente grandes.

12.5 O COMPILADOR BLISS/11

Esse compilador implementa a linguagem de programação de sistemas Bliss num PDP-11 (Wulf *et al.* [1975]). Num certo sentido, é um compilador otimizante de um mundo que deixou de existir, aquele em que o espaço de memória estava a prêmio numa escala suficiente para fazer sentido realizar otimizações cujo único propósito era o de reduzir espaço ao invés de tempo. No entanto, a maioria das otimizações realizadas pelo compilador ganhava tempo igualmente, e os descendentes desse compilador estão em uso hoje em dia.

O compilador merece nossa atenção por várias razões. Seu desempenho de otimização é forte e realiza transformações que não são encontradas em lugar nenhum. Sobretudo, foi o pioneiro na abordagem "dirigida pela sintaxe" para a otimização, como discutido na Seção 10.5. Isto é, a linguagem Bliss foi projetada para produzir somente grafos de fluxo redutíveis (não possuem desvios). Por conseguinte, foi possível que a análise de fluxo de dados fosse realizada numa árvore gramatical diretamente, ao invés de no grafo de fluxo.

O compilador opera numa única passagem, com um procedimento processado completamente antes do próximo ser lido. Os projetistas vêem o compilador como composto por cinco módulos, como mostrado na Fig. 12.5.

LEXSYNFLO realiza a análise léxica e sintática. Um analisador sintático recursivo-descendente é usado. Na medida em que BLISS

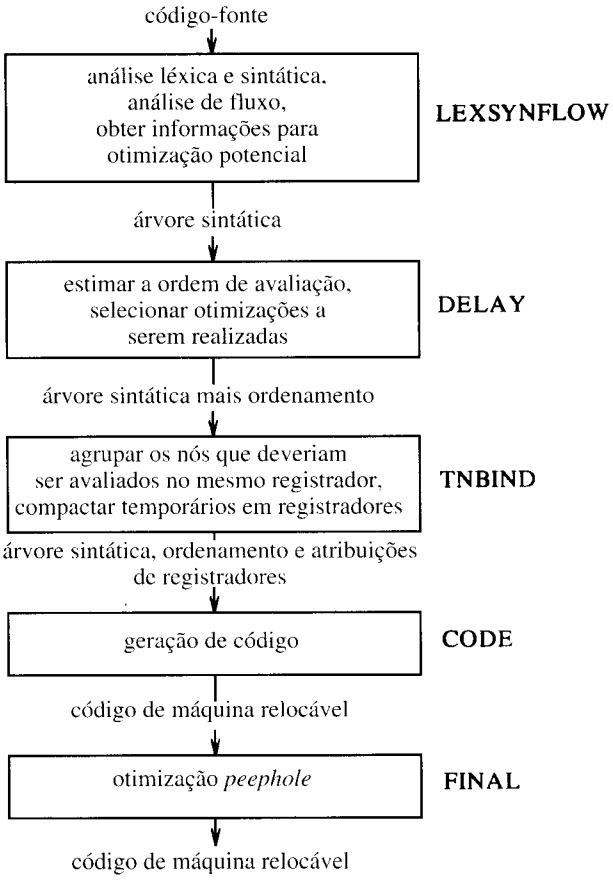


Fig. 12.5. O compilador BLISS/11.

não permite comandos de desvio, todos os grafos de fluxo de procedimentos BLISS são redutíveis. De fato, a sintaxe da linguagem nos habilita a construir os grafos de fluxo e determinar os laços e entradas de laços, na medida em que analisamos sintaticamente. LEXSYNFLO assim o faz e também determina as subexpressões comuns e uma variante das cadeias-ud e -du, tirando vantagem da estrutura dos grafos de fluxo redutíveis. Outra tarefa importante de LEXSYNFLO é a de detectar grupos de expressões similares. Existem candidatas para substituição por uma única sub-rotina. Note-se que essa substituição faz com que o programa rode mais lentamente, mas pode economizar espaço.

O módulo DELAY examina a árvore sintática para determinar que instâncias particulares das otimizações usuais, tais como a movimentação de código invariante e a eliminação de subexpressões comuns, estão realmente inclinadas a produzir lucro. A ordem de avaliações de expressões é determinada nesse tempo, baseada na estratégia de rotulação da Seção 9.10, modificada para levar em conta os registradores que não estão disponíveis por estarem sendo usados para preservar os valores das subexpressões comuns. As leis algébricas são usadas para determinar se o reordenamento das computações deveria ser feito. As expressões condicionais são avaliadas, numericamente ou através do fluxo de controle, como discutido na Seção 8.4, e DELAY decide que nó é mais barato em cada instância.

TNBIND considera que nomes temporários devem ser amarrados a registradores. Tanto os registradores quanto as localizações de memória são alocados. A estratégia usada é primeiro agrupar os nós da árvore sintática que deveriam estar associados ao mesmo registrador. Como discutido na Seção 9.6, há vantagem em se avaliar um nó no mesmo registrador que um de seus pais. Em seguida, a vantagem a ser ganha mantendo-se um temporário num registrador é estimada por um cálculo que favorece aqueles que são usados várias vezes sobre um pequeno intervalo. Os registradores são atribuídos até que usados,

*De fato, pode ser a soma de dois registradores, um registrador chamado de *base* e o outro de *índice*, mais o *deslocamento*, segundo a nomenclatura da IBM. (N. do T.)

compactando-se primeiro nos registradores, os nós mais vantajosos. CODE converte a árvore, com seu ordenamento e informações de atribuição de registradores, para código relocável de máquina.

Esse código é, em seguida, repetidamente examinado por FINAL que realiza a otimização *peephole* até que não resultem mais melhorias. As melhorias feitas incluem a eliminação de desvios para desvios (condicionais ou incondicionais) e a complementação das condicionais, como discutido na Seção 9.9.

As instruções redundantes ou inatingíveis são eliminadas (essas poderiam resultar de outras otimizações de FINAL). A combinação das seqüências de códigos similares nas duas ramificações de um desvio é tentada, como também o é a propagação de constantes. Um número de outras otimizações locais, algumas realmente dependentes da máquina, é tentado. Uma dessas otimizações, importante, é a substituição, onde for possível, de instruções de desvio por "ramificações" PDP-11, que requerem uma palavra, mas estão limitadas em 128 *bytes* em seu alcance.

12.6 O COMPILADOR OTIMIZANTE MODULA-2

Esse compilador, descrito em Powell [1984], foi desenvolvido com a finalidade de produzir código de boa qualidade, usando otimizações que providenciam um alto retorno para um pequeno esforço; o autor descreve sua estratégia como procurar pelas "melhores otimizações simples". Tal filosofia pode ser difícil de conduzir sem experimentação e medições, é difícil decidir quais são as "melhores otimizações simples" antecipadamente e algumas das decisões feitas no compilador de Modula-2 são provavelmente inadequadas para um compilador que providencie máxima otimização. Contudo, a estratégia atingiu a meta do autor em produzir um código excelente com um compilador que foi escrito em uns poucos meses por uma pessoa. Os cinco passos da vanguarda do compilador são delineados na Fig. 12.6.

O analisador sintático foi gerado usando o Yacc e o mesmo produz árvores sintáticas em duas passagens, já que as variáveis de Modula não têm que ser declaradas antes do uso. Foi feita uma tentativa de tornar esse compilador compatível com facilidades existentes. O código intermediário é P-*code*, para compatibilidade com muitos compiladores Pascal e C, que rodam sob Berkeley UNIX, e, por conseguinte, os procedimentos escritos nas três linguagens podem ser incorporados facilmente.

O compilador não realiza a análise de fluxo de dados. Ao invés, Modula-2, como Bliss, é uma linguagem que pode produzir somente grafos de fluxo redutíveis, e, dessa forma, a metodologia da Seção 10.5 pode igualmente ser usada aqui. De fato, o compilador de Modula-2 vai além do compilador Bliss-11 pela forma com que tira vantagem da sintaxe. Os laços são identificados por suas sintaxes; isto é, o compilador procura por construções *while* e *for*. As expressões invariantes são detectadas pelo fato de que nenhuma de suas variáveis são definidas no laço, sendo essas expressões removidas para o cabeçalho do mesmo. As únicas variáveis de indução que são detectadas são as que estão na família de um índice de laço *for*. As subexpressões globais comuns são detectadas quando uma está no bloco que domina o bloco da outra, mas essa análise é feita uma expressão de cada vez, ao invés de se usar vetores de *bits*.

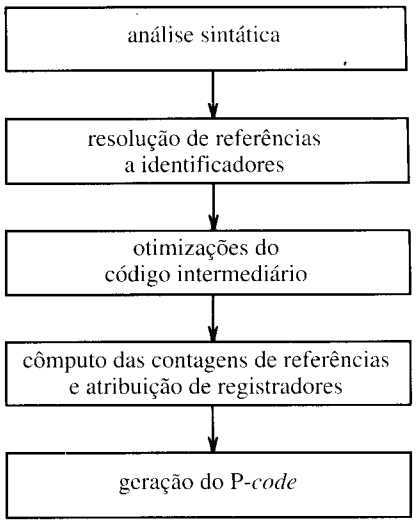


Fig. 12.6. Passagens do compilador Modula-2.

A estratégia de alocação de registradores é similarmente projetada para realizar coisas razoáveis, sem ser exaustiva. Em particular, são considerados como candidatos para alocação a um registrador somente:

1. os temporários usados durante a avaliação de uma expressão (esses recebem a primeira prioridade)
2. valores de subexpressões comuns,
3. índices e valores limite de laços *for*,
4. o endereço de *E* numa expressão da forma `with` *E* `do` e
5. variáveis simples (caracteres, inteiros e assim por diante) locais ao procedimento corrente.

É feita uma tentativa para estimar o valor de se manter cada variável nas classes (2)-(5) num registrador. É assumido que se um enunciado é executado 10^d se estiver aninhado em *d* laços. No entanto, as variáveis referenciadas não mais do que duas vezes não são consideradas elegíveis; outras são classificadas pela ordem de uso estimado e atribuídas a um registrador, se um estiver disponível, após atribuir aos temporários de expressões e às variáveis de maior *ranking*.

APÊNDICE

UM PROJETO DE PROGRAMAÇÃO

A.1 INTRODUÇÃO

Este apêndice sugere exercícios de programação que podem ser usados num laboratório de programação acompanhando um curso de projeto de compiladores baseado neste livro. Os exercícios consistem na implementação dos constituintes básicos de um compilador para um subconjunto de Pascal. O subconjunto é mínimo, mas permite que programas, tais como o procedimento recursivo de classificação da Seção 7.1, sejam expressos. Sendo um subconjunto de uma linguagem de programação existente, possui uma certa utilidade. O significado dos programas no subconjunto é determinado pela semântica de Pascal (Jensen e Wirth [1975]). Se um compilador Pascal estiver disponível, pode ser usado como um exercício para verificar o comportamento do compilador escrito. As construções no subconjunto aparecem na maioria das linguagens de programação, e, dessa forma, exercícios correspondentes podem ser formulados usando-se uma linguagem diferente, se um compilador Pascal não estiver disponível.

A.2 ESTRUTURA DE PROGRAMA

Um programa consiste em uma seqüência de declarações de dados globais, de declarações de procedimentos e funções e de um único enunciado composto, que é o "programa principal". Para os dados globais, deve ser alocada memória estática. Os dados locais aos procedimentos e funções têm a memória alocada numa pilha. A recursão é permitida e os parâmetros são passados por referência. Assume-se que os procedimentos `read` e `write` sejam fornecidos pelo compilador.

A Fig. A.1 mostra um programa exemplo. O nome do programa é `exemplo` e `input` e `output` são os nomes dos arquivos usados por `read` e `write`, respectivamente.

```
programe exemplo (input, output);
var x, y: integer;
function mdc (a, b: integer): integer;
begin
    if b = 0 then mdc := a
    else mdc := mdc (b, a mod b)
end ;

begin
    read (x, y);
    write (mdc (x, y))
end .
```

Fig. A.1/. Programa exemplo.

A.3 SINTAXE DE UM SUBCONJUNTO DE PASCAL

Listada abaixo está uma gramática LALR(1) para um subconjunto de Pascal. A gramática pode ser modificada para uma análise sintática de descendência recursiva através da eliminação da recursividade à esquerda, como descrito nas Seções 2.4 e 4.3. Um analisador sintático de precedência de operadores pode ser construído para expressões, através da substituição de **operador_relacional, operador_aditivo** e **operador_multiplicativo** e eliminando-se as produções-ϵ.

A adição da produção

enunciado → **if** *expressão* **then** *enunciado*

introduz a ambigüidade do "*else*-vazio", que pode ser eliminada conforme discutido na Seção 4.3 (ver também o Exemplo 4.19 se a análise sintática preditiva também estiver sendo usada).

Não existe distinção sintática entre uma variável simples e uma chamada de função sem parâmetros. Ambas são geradas pela produção

fator → **id**

Por conseguinte, a atribuição `a := b` estabelece `a` com o valor retornado pela função `b`, se `b` tiver sido declarado como uma função.

programa →
    program id (*lista_de_identificadores*);
    declarações
    declarações_de_subprogramas
    enunciado_composto

    •

lista_de_identificadores →
    id
    | *lista_de_identificadores*, **id**

declarações →
    declarações **var** *lista_de_identificadores : tipo* ;
    | ϵ

tipo →

    tipo_padrão
    | **array [num .. num] of** *tipo_padrão*

tipo_padrão →
    inteiro
    | **real**

declarações_de_subprogramas →
  declarações_de_subprogramas declaração_de_subprograma ;
  | ϵ

declaração_de_subprograma →
  cabeçalho_de_subprograma
  declarações
  enunciado_composto

cabeçalho_de_subprograma →
  function id *argumentos : tipo_padrão* ;
  | **procedure id** *argumentos* :

argumentos →
  (*lista_de_parâmetros*)
  | ϵ

lista_de_parâmetros →
  lista_de_identificadores : tipo
  | *lista_de_parâmetros ; lista_de_identificadores : tipo*

enunciado_composto →
  begin
  enunciados_opcionais
  end

enunciados_opcionais →
  lista_de_enunciados
  | ϵ

lista_de_enunciados →
  enunciado
  | *lista_de_enunciados ; enunciado*

enunciado →
  variável **operador-de-atribuição** *expressão*
  | *chamada_de_procedimento*
  | *enunciado_composto*
  | **if** *expressão* **then** *enunciado* **else** *enunciado*
  | **while** *expressão* **do** *enunciado*

variável →
  id
  | **id** [*expressão*]

chamada_de_procedimento →
  id
  | **id** (*lista_de_expressões*)

lista-de-expressões →
  expreessão
  | *lista_de_expressões , expressão*

expressão →
  expressão_simples
  | *expressão_simples*
  operador_relacional *expressão_simples*

expressão_simples →
  termo
  | *sinal termo*
  | *expressão_simples* **operador_aditivo** *termo*

termo →
  fator
  | *termo* **operador_multiplicativo** *fator*

fator →
  id
  | **id** (*lista_de_expressões*)
  | **num**
  | (*expressão*)
  | **not** *fator*

sinal →
  + | −

A.4 CONVENÇÕES LÉXICAS

A notação para a especificação de *tokens* é proveniente da Seção 3.3.

1. Os comentários são envolvidos por { e }, não podendo conter {. Os comentários podem aparecer após qualquer *token*.
2. Os espaços entre os *tokens* são opcionais, com exceção das palavras-chave, que têm que ser envolvidas por espaços ou avanços de linha, estar no início do programa ou serem seguidas pelo ponto final.
3. O *token* **id** para identificadores reconhece uma letra seguida por letras ou dígitos:

letra → `[a-zA-Z]`
dígito → `[0-9]`
id → **letra** (**letra** | **dígito**)*

O implementador pode desejar colocar um limite no comprimento do identificador.

4. O *token* **num** reconhece os inteiros sem signal (ver o Exemplo 3.5):

dígitos → **dígito dígito***
fração_opcional → · **dígitos** | ϵ
expoente_opcional → (E (+ | − | ϵ) **dígitos** | ϵ
num → **dígitos fração_opcional expoente_opcional**

5. As palavras-chave são reservadas e aparecem em negrito na gramática.
6. Os operadores relacionais (**operador_relacional**) são: =, <>, <, <=, >= e >. Note-se que <> denota ≠.
7. Os operadores aditivos (**operador_aditivo**) são: +, − e `or`.
8. Os operadores multiplicativos (**operador_multiplicativo**) são: *, /, `div`, `mod` e `and`.
9. O lexema para o *token* **operador_de_atribuição** é :=.

A. 5 EXERCÍCIOS SUGERIDOS

Um exercício de programação adequado para um curso de um semestre é o de escrever um interpretador para a linguagem definida acima ou para um subconjunto similar de outra linguagem de alto nível. O projeto envolve a tradução do programa-fonte numa representação intermediária, tal como quádruplas ou código de máquina de pilha, e a interpretação dessa representação. Iremos propor uma ordem para a construção dos módulos. A ordem é diferente daquela na qual os módulos são executados no compilador porque é conveniente ter um interpretador operativo para depurar os outros componentes do compilador.

1. *Projetar um mecanismo de tabela de símbolos.* Decidir a respeito da organização da tabela de símbolos. Permitir que sejam coletadas informações a respeito de nomes, mas manter flexível a estrutura de registro da tabela de símbolos por enquanto. Escrever rotinas para:
   i) Pesquisar a tabela de símbolos procurando por um determinado nome, criar uma nova entrada para o nome, se já não estiver nela e, num e noutro caso retornar um apontador para o registro para aquele nome.
   ii) Remover da tabela de símbolos todos os nomes locais a um dado procedimento.
2. *Escrever um interpretador para quádruplas.* O conjunto exato de quádruplas pode ser deixado em aberto por enquanto, mas deveria incluir os enunciados aritméticos e de desvio condicional correspondentes ao conjunto de operadores da linguagem. Incluir igualmente operações lógicas, se as condições forem avaliadas aritmeticamente ao invés de através da posição no programa. Adicionalmente, prever a necessidade de "quádruplas" para a conversão de inteiro para real, para marcar o início e o final de procedimentos e para a transmissão de parâmetros e chamadas de procedimentos.

É também necessário, a esse tempo, projetar a seqüência de chamada e a organização em tempo de execução para os programas que serão interpretados. A organização simples de pilha discutida na Seção 7.3 é adequada à linguagem-exemplo, porque nenhuma declaração aninhada de procedimento é permitida na mesma; isto é, as variáveis ou são globais (declaradas ao nível de todo o programa) ou locais a um procedimento simples.

Por uma questão de simplicidade, outra linguagem de alto nível pode ser usada em lugar do interpretador. Cada quádrupla pode ser um enunciado de uma linguagem de alto nível como C ou mesmo Pascal. A saída do compilador é, então, uma seqüência de enunciados C que podem ser compilados num compilador C existente. Essa abordagem permite que o implementador se concentre na organização em tempo de execução.

3. *Escrever o analisador léxico.* Selecionar os códigos internos para as *tokens*. Decidir como as constantes serão representadas no compilador. Contar as linhas para uso futuro por parte do tratador de mensagens de erro. Produzir uma listagem do programa-fonte se desejado. Escrever um programa para introduzir as palavras-chave reservadas na tabela de símbolos. Projetar seu analisador léxico para ser uma sub-rotina chamada pelo analisador sintático, retornando o par (*token*, valor de atributo). Presentemente, os erros detectados por seu analisador léxico podem ser tratados através da chamada de uma rotina de impressão de erros e encerramento.
4. *Escrever as ações semânticas.* Escrever as rotinas semânticas para gerar as quádruplas. A gramática necessitará ser modificada em alguns locais para tornar a tradução mais fácil. Consultar as Seções 5.5 e 5.6 para os exemplos sobre como modificar a gramática de forma útil. Realizar as ações semânticas a esse tempo, convertendo os inteiros para os reais quando necessário.
5. *Escrever o analisador sintático.* Se um gerador de analisadores sintáticos LALR estiver disponível, isso irá simplificar a tarefa consideravelmente. Se um gerador de analisadores sintáticos, que trate gramáticas ambíguas como Yacc, estiver disponível, os não-terminais que denotem expressões podem ser combinados. Sobretudo, a ambigüidade do "***else***-vazio " pode ser resolvida, empilhando-se sempre que um conflito de empilhar/reduzir ocorrer.
6. *Escrever as rotinas de tratamento de erros.* Estar preparado para se recuperar dos erros léxicos e sintáticos. Imprimir diagnósticos de erro para os erros léxicos, sintáticos e semânticos.
7. *Avaliação.* O programa da Fig. A.1 pode servir como uma simples rotina de teste. Um outro programa de teste pode ser baseado no programa da Fig. 7.1. O código para a função `partition` na figura corresponde ao fragmento de programa C marcado na Fig. 10.2. Rode seu compilador através de um gerador de perfis de comportamento, se um estiver disponível.* Determine as rotinas nas quais a maior parte do tempo está sendo gasto. Que módulos deveriam ser modificados para se aumentar a velocidade do seu compilador?

A.6 EVOLUÇÃO DO INTERPRETADOR

Uma abordagem alternativa para se construir um interpretador para linguagens é começar implementando uma calculadora de bolso, isto é, um interpretador para expressões. Gradualmente, adicionar construções à linguagem até que um interpretador para a linguagem completa seja obtido. Uma abordagem similar é usada em Kernighan e Pike [1984]. Uma ordem proposta para se adicionar as construções é:

1. *Traduzir expressões para a forma posfixa.* Usando ou uma análise sintática recursivo-descendente, como no Capítulo 2, ou um gerador de analisadores sintáticos, familiarizar-se com o ambiente de programação, escrevendo um tradutor de expressões aritméticas simples para a notação posfixa.
2. *Adicionar o analisador léxico.* Permitir que palavras-chave, identificadores e números figurem no tradutor construído acima. Reoriente o tradutor de forma a que produza ou código para uma máquina de pilha ou quádruplas.
3. *Escrever um interpretador para a representação intermediária.* Como discutido na Seção 5.7, uma linguagem de alto nível pode ser usada em lugar do interpretador. Para o momento, o interpretador necessita somente suportar operações aritméticas, atribuições e entrada e saída. Expandir a linguagem, permitindo declarações de variáveis globais, atribuições e chamadas aos procedimentos `read` e `write`. Essas construções permitem que o interpretador seja testado.
4. *Adicionar os enunciados.* Um programa na linguagem consiste agora em um programa principal com declarações de subprogramas. Testar ambos, tradutor e interpretador.
5. *Adicionar procedimentos e funções.* A tabela de símbolos precisa agora permitir que os escopos dos identificadores sejam limitados aos corpos dos procedimentos. Projetar uma seqüência de chamada. De novo, a organização simples de pilha da Seção 7.3 é adequada. Expandir o interpretador para suportar a seqüência de chamada.

A.7 EXTENSÕES

Existe um número de figurações que podem ser adicionadas à linguagem sem aumentar grandemente a complexidade da compilação. Dentre essas estão:

1. *arrays* multidimensionais
2. enunciado *for* e *case*
3. estrutura de bloco
4. estrutura de registro

Se o tempo permitir, adicionar uma ou mais dessas extensões ao seu compilador.

*Do original em inglês: *profiler*. (N. do T.)

BIBLIOGRAFIA

ABEL, N. E. AND J. R. BELL [1972]. "Global optimization in compilers," *Proc. First USA-Japan Computer Conf.*, AFIPS Press, Montvale, N. J.

ABELSON, H. AND G. J. SUSSMAN [1985]. *Structure and Interpretation of Computer Programs*, MIT Press, Cambridge, Mass.

ADRION, W. R., M. A. BRANSTAD, AND J. C. CHERNIAVSKY [1982]. "Validation, verification, and testing of computer software," *Computing Surveys* **14:2**, 159-192.

AHO, A. V. [1980]. "Pattern matching in strings," in Book [1980], pp. 325-347.

AHO, A. V. AND M. J. CORASICK [1975]. "Efficient string matching: an aid to bibliographic search," *Comm. ACM* **18:6**, 333-340.

AHO, A. V. AND M. GANAPATHI [1985]. "Efficient tree pattern matching: an aid to code generation," *Twelfth Annual ACM Symposium on Principles of Programming Languages*, 334-340.

AHO, A. V., J. E. HOPCROFT, AND J. D. ULLMAN [1974]. *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass.

AHO, A. V., J. E. HOPCROFT, AND J. D. ULLMAN [1983]. *Data Structures and Algorithms*, Addison-Wesley, Reading, Mass.

AHO, A. V. AND S. C. JOHNSON [1974]. "LR parsing," *Computing Surveys* **6:2**, 99-124.

AHO, A. V. AND S. C. JOHNSON [1976]. "Optimal code generation for expression trees," *J. ACM* **23:3**, 488-501.

AHO, A. V., S. C. JOHNSON, AND J. D. ULLMAN [1975]. "Deterministic parsing of ambiguous grammars," *Comm. ACM* **18:8**, 441-452.

AHO, A. V., S. C. JOHNSON, AND J. D. ULLMAN [1977a]. "Code generation for expressions with common subexpressions," *J. ACM* **24:1**, 146-160.

AHO, A.V., S. C. JOHNSON, AND J. D. ULLMAN [1977b]. "Code generation for machines with multiregister operations," *Fourth ACM Symposium on Principles of Programming Languages*, 21-28.

AHO, A. V., B. W. KERNIGHAN, AND P. J. WEINBERGER [1979]. "Awk — a pattern scanning and processing language," *Software — Practice and Experience* **9:4**, 267-280.

AHO, A. V. AND T. G. PETERSON [1972]. "A minimum distance error-correcting parser for context-free languages," *SIAM J. Computing* **1:4**, 305-312.

AHO, A. V. AND R. SETHI [1977]. "How hard is compiler code generation?" Lecture Notes in Computer Science **52**, Springer-Verlag, Berlin, 1-15.

AHO, A. V. AND J. D. ULLMAN [1972a]. "Optimization of straight line code," *SIAM J. Computing* **1:1**, 1-19.

AHO, A. V. AND J. D. ULLMAN [1972b]. *The Theory of Parsing, Translation and Compiling*, Vol. I: *Parsing*, Prentice-Hall, Englewood Cliffs, N. J.

AHO, A. V. AND J. D. ULLMAN [1973a]. *The Theory of Parsing, Translation and Compiling*, Vol. II: *Compiling*, Prentice-Hall, Englewood Cliffs, N. J.

AHO, A. V. AND J. D. ULLMAN [1973b]. "A technique for speeding up LR(k) parsers," *SIAM J. Computing* **2:2**, 106-127.

AHO, A. V. AND J. D. ULLMAN [1977]. *Principles of Compiler Design*, Addison-Wesley, Reading, Mass.

AIGRAIN, P., S. L. GRAHAM, R. R. HENRY, M. K. MCKUSICK, AND E. PELEGRI-LLOPART [1984]. "Experience with a Graham-Glanville style code generator," *ACM SIGPLAN Notices* **19:6**, 13-24.

ALLEN, F. E. [1969]. "Program optimization," *Annual Review in Automatic Programming* **5**, 239-307.

ALLEN, F. E. [1970]. "Control flow analysis," *ACM SIGPLAN Notices* **5:7**, 1-19.

ALLEN, P. E. [1974]. "Interprocedural data flow analysis," *Information Processing 74*, North-Holland, Amsterdam, 398-402.

ALLEN, F. E. [1975]. "Bibliography on program optimization," RC-5767, IBM T. J. Watson Research Center, Yorktown Heights, N. Y.

ALLEN, F. E., J. L. CARTER, J. FABRI, J. FERRANTE, W. H. HARRISON, P. G. LOEWNER, AND L. H. TREVILLYAN [1980]. "The experimental compiling system," *IBM. J. Research and Development* **24:6**, 695-715.

ALLEN, F. E. AND J. COCKE [1972]. "A catalogue of optimizing transformations," in Rustin [1972], pp. 1-30.

ALLEN, F. E. AND J. COCKE [1976]. "A program data flow analysis procedure," *Comm. ACM* **19:3**, 137-147.

ALLEN, F. E., J. COCKE, AND K. KENNEDY [1981]. "Reduction of operator strength," in Muchnick and Jones [1981], pp. 79-101.

AMMANN, U. [1977]. "On code generation in a Pascal compiler," *Software—Practice and Experience* **7:3**, 391-423.

AMMANN, U. [1981]. "The Zurich implementation," in Barron [1981], pp. 63-82.

ANDERSON, J. P. [1964]. "A note on some compiling algorithms," *Comm. ACM* **7:3**, 149-150.

ANDERSON, T., J. EVE, AND J. J. HORNING [1973]. "Efficient LR(1) parsers," *Acta Informatica* **2:1**, 12-39.

ANKLAN, P., D. CUTLER, R. HEINEN, JR., AND M. D. MACLAREN [1982]. *Engineering a Compiler*, Digital Press, Bedford, Mass.

ARDEN, B. W., B. A. GALLER, and R. M. GRAHAN [1961]. "An algorithm for equivalence declarations," *Comm. ACM* **4:7**, 310-314.

AUSLANDER, M. A. AND M. E. HOPKINS [1982]. "An overview of the PL.8 compiler," *ACM SIGPLAN Notices* **17:6**, 22-31.

BACKHOUSE, R. C. [1976]. "An alternative approach to the improvement of LR parsers," *Acta Informatica* **6:3**, 277-296.

BACKHOUSE, R. C. [1984]. "Global data flow analysis problems arising in locally least-cost error recovery," *TOPLAS* **6:2**, 192-214.

BACKUS, J. W. [1981]. "Transcript of presentation on the history of Fortran I, II, and III," in Wexelblat [1981], pp. 45-66.

BACKUS, J. W., R. J. BEEBER, S. BEST, R. GOLDBERG, L. M. HAIBT, H. L. HERRICK, R. A. NELSON, D. SAYRE, P. B. SHERIDAN, H. STERN, I. ZILLER, R. A. HUGHES, AND R. NUTT [1957]. "The Fortran automatic coding system," *Western Joint Computer Conference*, 188-198. Reprinted in Rosen [1967], pp. 29-47.

BAKER, B. S. [1977]. "An algorithm for structuring programs," *J. ACM* **24:1**, 98-120.

BAKER, T. P. [1982]. "A one-pass algorithm for overload resolution in Ada," *TOPLAS* **4:4**, 601-614.

BANNING, J. P. [1979]. "An efficient way to find the side effects of procedure calls and aliases of variables," *Sixth Annual ACM Symposium on Principles of Programming Languages*, 29-41.

BARRON, D. W. [1981]. *Pascal — The Language and its Implementation*, Wiley, Chichester.

BARTH, J. M. [1978]. "A practical interprocedural data flow analysis algorithm," *Comm. ACM* **21:9**, 724-736.

BATSON, A. [1965]. "The organization of symbol tables," *Comm. ACM* **8:2**, 111-112.

BAUER, A. M. AND H. J. SAAL [1974]. "Does APL really need run-time checking?" *Software—Practice and Experience* **4:2**, 129-138.

BAUER, F. L. [1976]. "Historical remarks on compiler construction," in Bauer and Eickel [1976], pp. 603-621. Addendum by A. P. Ershov, pp. 622-626.

BAUER, F. L. AND J. EICKEL [1976]. *Compiler Construction: An Advanced Course*, 2nd Ed., Lecture Notes in Computer Science **21**, Springer-Verlag, Berlin.

BAUER, F. L. AND H. WOSSNER [1972]. "The 'Plankankül' of Konrad Zuse: A forerunner of today's programming languages," *Comm. ACM* **15:7**, 678-685.

BEATTY, J. C. [1972]. "An axiomatic approach to code optimization for expressions," *J. ACM* **19:4**, 714-724. Errata **20** (1973), p. 180 and 538.

Beatty, J. C. [1974]. "Register assignment algorithm for generation of highly optimized object code," *IBM J. Research and Development* **5:2**, 20-39.

BELADY, L. A . [1966] . "A study of replacement algorithms for a virtual storage computer," *IBM Systems J.* **5:2**, 78-101.

BENTLEY, J. L. [1982]. *Writing Efficient Programs*, Prentice-Hall, Englewood Cliffs, N. J.

BENTLEY, J. L., W. S. CLEVELAND, AND R. SETHI [1985]. Empirical analysis of hash functions," manuscript, AT&T Bell Laboratories, Murray Hill, N. J.

BIRMAN, A. AND J. D. ULLMAN [1973]. "Parsing algorithms with backtrack," *Information and Control* **23:1**, 1-34.

BOCHMANN, G. V. [1976]. "Semantic evaluation from left to right," *Comm. ACM* **19:2**, 55-62.

BOCHMANN, G. V. AND P. WARD [1978]. Compiler writing system for attribute grammars," *Computer J.* **21:2**, 144-148.

BOOK, R. V. [1980]. *Formal Language Theory*, Academic Press, New York.

BOYER, R. S. AND J. S. MOORE [1977]. "A fast string searching algorithm," *Comm. ACM* **20:10**, 262-272.

BRANQUART, P., J.-P. CARDINAEL, J. LEWI, J.-P. DELESCAILLE, AND M. VANBEGIN [1976]. *An Optimized Translation Process and its Application to Algol 68*, Lecture Notes in Computer Science, **38**, Springer-Verlag, Berlin.

BRATMAN, H. [1961]. "An alternate form of the 'Uncol diagram'," *Comm. ACM* **4:3**, 142.

BROOKER, R. A. AND D. MORRIS [1962]. "A general translation program for phrase structure languages," *J. ACM* **9:1**, 1-10.

BROOKS, F. P., JR. [1975]. *The Mythical Man-Month*, Addison-Wesley, Reading, Mass.

BROSGOL, B. M. [1974]. *Deterministic Translation Grammars*, Ph. D. Thesis, TR 3-74, Harvard Univ., Cambridge, Mass.

BRUNO, J. AND T. LASSAGNE [1975]. "The generation of optimal code for stack machines," *J. ACM* **22:3**,382-396.

BRUNO, J. AND R. SETHI [1976]. "Code generation for a one-register machine," *J. ACM* **23:3**,502-510.

BURSTALL, R. M., D. B. MACQUEEN, AND D. T. SANNELLA [1980]. "Hope: an experimental applicative language," *Lisp Conference*, P.O. Box 487, Redwood Estates, Calif. 95044, 136-143.

BUSAM, V. A. AND D. E. ENGLUND [1969]. "Optimization of expressions in Fortran," *Comm. ACM* **12:12**, 666-674.

CARDELLI, L. [1984]. "Basic polymorphic typechecking," Computing Science Technical Report 112, AT&T Bell Laboratories, Murray Hill, N. J.

CARTER, L. R. [1982]. *An Analysis of Pascal Programs*, UMI Research Press, Ann Arbor, Michigan.

CARTWRIGHT, R. [1985]. "Types as intervals," *Twelfth Annual ACM Symposium on Principles of Programming Languages*, 22-36.

CATTELL, R. G. G. [1980]. "Automatic derivation of code generators from machine descriptions," *TOPLAS* **2:2**, 173-190.

CHAITIN, G. J. [1982]. "Register allocation and spilling via graph coloring," *ACM SIGPLAN Notices* **17:6**, 201-207.

CHAITIN, G. J., M. A. AUSLANDER, A. K. CHANDRA, J. COCKE, M. E. HOPKINS, AND P. W. MARKSTEIN [1981]. "Register allocation via coloring," *Computer Languages* **6**, 47-57.

CHERNIAVSKY, J. C., P. B. HENDERSON, AND J. KEOHANE [1976]. "On the equivalence of URE flow graphs and reducible flow graphs," *Proc. 1976 Conference on Information Sciences and Systems*, Johns Hopkins Univ., 423-429.

CHERRY, L. L. [1982]. "Writing tools," *IEEE Trans. on Communications* **COM-30:1**, 100-104.

CHOMSKY, Y, N. [1956]. "Three models for the description of language," *IRE Trans. on Information Theory* **IT-2:3**, 113-124.

CHOW, F. [1983]. *A Portable Machine-Independent Global Optimizer*, Ph. D. Thesis, Computer System Lab., Stanford Univ., Stanford, Calif.

CHOW, F. AND J. L. HENNESSY [1984]. "Register allocation by priority-based coloring," *ACM SIGPLAN Notices* **19:6**, 222-232.

CHURCH, A. [1941]. *The Calculi of Lambda Conversion*, Annals of Math. Studies, No. 6, Princeton University Press, Princeton, N. J.

CHURCH, A. [1956]. *Introduction to Mathematical Logic*, Vol. I, Princeton University Press, Princeton, N. J.

CIESINGER, J. [1979]. "A bibliography of error handling," *ACM SIGPLAN Notices* **14:1**, 16-26.

COCKE, J. [1970]. "Global common subexpression elimination," *ACM SIGPLAN Notices* **5:7**, 20-24.

COCKE, J. AND K. KENNEDY [1976]. "Profitability computations on program flow graphs," *Computers and Mathematics with Applications* **2:2**, 145-159.

COCKE, J. AND K. KENNEDY [1977]. An algorithm for reduction of operator strength," Comm. ACM **20:11,** 850-856.

COCKE, J. AND J. MARKSTEIN [1980]. "Measurement of code improvement algorithms," *Information Processing 80*, 221-228.

COCKE, J. AND J. MILLER [1969]."Some analysis techniques for optimizing computer programs," *Proc. 2nd Hawaii Intl. Conf. on Systems Sciences,* 143-146.

COCKE, J. AND J. T. SCHWARTZ [1970]. *Programming Languages and Their Compilers: Preliminary Notes, Second Revised Version,* Courant Institute of Mathematical Sciences, New York.

COFFMAN, E. G., JR. AND R. SETHI [1983]. "Instruction sets for evaluating arithmetic expressions," *J. ACM* **30:3**, 457-478.

COHEN, R. AND E. HARRY [1979]. "Automatic generation of near-optimal linear-time translators for non-circular attribute grammars," *Sixth ACM Symposium on Principles of Programming Languages,* 121-134.

CONWAY, M. E. [1963]. "Design of a separable transition diagram compiler," *Comm. ACM* **6:7**, 396-408.

CONWAY, R. W. AND W. L. MAXWELL [1963]. "CORC - the Cornell computing language," *Comm. ACM* **6:6**,317-321.

CONWAY, R. W. AND T. R. WILCOX [1973]. "Design and implementation of a diagnostic compiler for PL/I," *Comm. ACM* **16:3**, 169-179.

CORMACK, G. V. [1981]. "An algorithm for the selection of overloaded functions in Ada," *ACM SIGPLAN Notices* **16:2** (February) 48-52.

CORMACK, G. V., R. N. S. HORSPOOL, AND M. KAISERSWERTH [1985]. Practical perfect hashing," *Computer J.* **28:1**, 54-58.

COURCELLE, B. [1984]. "Attribute grammars: definitions, analysis of dependencies, proof methods," in Lorho [1984], pp. 81-102.

COUSOT, P. [1981]. "Semantic foundations of program analysis," in Muchnick and Jones [1981], pp. 303-342.

COUSOT, P. AND R. COUSOT [1977]. "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," *Fourth ACM Symposium on Principles of Programming Languages,* 238-252.

CURRY, H. B. AND R. FEYS [1958]. *Combinatory Logic*, Vol. 1, North-Holland, Amsterdam.

DATE, C. J. [1986]. *An Introduction to Database Systems*, 4th Ed., AddisonWesley, Reading, Mass.

DAVIDSON, J. W. AND C. W. FRASER [1980]. The design and application of a retargetable peephole optimizer," *TOPLAS* **2:2**, 191-202. Errata **3:1** (1981) 110.

DAVIDSON, J. W. AND C. W. FRASER [1984a]. Automatic generation of peephole optimizations," *ACM SIGPLAN Notices* **19:6**, 111-116.

DAVIDSON, J. W. AND C. W. FRASER [1984b]. Code selection through object code optimization," *TOPLAS* **6:4**, 505-526.

DEREMER, F. [1969]. *Practical Translators for LR(k) Languages*, Ph. D. Thesis, M.I.T., Cambridge, Mass.

DEREMER, F. [1971]. "Simple LR(*k*) grammars," *Comm. ACM* **14:7**, 453-460.

DEREMER, F. AND T. PENNELLO [1982]. Efficient computation of LALR(1) lookahead sets," *TOPLAS* **4:4**, 615-649.

DEMERS, A. J. [1975]. "Elimination of single productions and merging of nonterminal symbols in LR(1) grammars," *J. Computer Languages* **1:2**, 105-119.

DENCKER, P., K. DURRE, AND J. HEUFT [1984]. Optimization of parser tables for portable compilers," *TOPLAS* **6:4**, 546-572.

DERANSART, P., M. JOURDAN, AND B. LORHO [1984]. "Speeding up circularity tests for attribute grammars," *Acta Informatica* **21**, 375-391.

DESPEYROUX, T. [1984]. "Executable specifications of static semantics," in Kahn, MacQueen, and Plotkin [1984], pp. 215-233.

DIJKSTRA, E. W. [1960]. "Recursive programming," *Numerische Math.* **2**, 312-318. Reprinted in Rosen [1967], pp. 221-228.

DIJKSTRA, E. W. [1963]. "An Algol 60 translator for the X1," *Annual Review in Automatic Programming* **3**, Pergamon Press, New York, 329-345.

DITZEL, D. AND H. R. MCLELLAN [1982]. "Register allocation for free: the C machine stack cache," *Proc. ACM Symp. on Architectural Support for Programming Languages and Operating Systems*, 48-56.

DOWNEY, P. J. AND R. SETHI [1978]. "Assignment commands with array references," *J. ACM* **25:4**, 652-666.

DOWNEY, P. J., R. SETHI, AND R. E. TARJAN [1980]. Variations on the common subexpression problem," *J. ACM* **27:4**, 758-771.

EARLEY, J. [1970]. "An efficient context-free parsing algorithm," *Comm. ACM* **13:2**, 94-102.

EARLEY, J. [1975a]. "Ambiguity and precedence in syntax description," *Acta Informatica* **4:2**, 183-192.

EARLEY, J. [1975b]. "High level iterators and a method of data structure choice," *J. Computer Languages* **1:4**, 321-342.

ELSHOFF, J. L. [1976]. "An analysis of some commercial PL/I programs," *IEEE Trans. Software Engineering* **SE2:2**, 113-120.

ENGELFRIET, J. [1984]. "Attribute evaluation methods," in Lorho [1984], pp. 103-138.

ERSHOV, A. P. [1958]. "On programming of arithmetic operations," *Comm. ACM* **1:8** (August) 3-6. Figures 1-3 appear in **1:9** (September 1958), p. 16.

ERSHOV, A. P. [1966]. "Alpha — an automatic programming system of high efficiency," *J. ACM* **13:1**, 17-24.

ERSHOV, A. P. [1971]. *The Alpha Automatic Programming System,* Academic Press, New York.

ERSHOV, A. P. AND C. H. A. KOSTER [1977]. *Methods of Algorithmic Language Implementation,* Lecture Notes in Computer Science **47**, Springer-Verlag, Berlin.

FANG, I. [1972]. "FOLDS, a declarative formal language definition system," STAN-CS-72-329, Stanford Univ.

FARROW, R. [1984]. "Generating a production compiler from an attribute grammar," *IEEE Software* **1** (October) 77-93.

FARROW, R. AND D. YELLIN [1985]. A comparison of storage optimizations in automatically-generated compilers," manuscript, Columbia Univ.

FELDMAN, S. I. [1979a]. "Make — a program for maintaining computer programs," *Software—Practice and Experience* **9:4**, 255-265.

FELDMAN, S. I. [1979b]. "Implementation of a portable Fortran 77 compiler using modern tools," *ACM SIGPLAN Notices* **14:8**, 98-106.

FISCHER, M. J. [1972]. "Efficiency of equivalence algorithms," in Miller and Thatcher [1972], pp. 153-168.

FLECK, A. C. [1976]. "The impossibility of content exchange through the byname parameter transmission technique," *ACM SIGPLAN Notices* **11:11** (November) 38-41.

FLOYD, R. W. [1961]. "An algorithm for coding efficient arithmetic expressions," *Comm. ACM* **4:1**, 42-51.

FLOYD, R. W. [1963]. "Syntactic analysis and operator precedence," *J. ACM* **10:3**, 316-333.

FLOYD, R. W. [1964]. "Bounded context syntactic analysis," *Comm. ACM* **7:2**, 62-67.

FONG, A. C. [1979]. "Automatic improvement of programs in very high level languages," *Sixth Annual ACM Symposium on Principles of Programming Languages,* 21-28.

FONG, A. C. AND J. D. ULLMAN [1976]. "Induction variables in very high-level languages," *Third Annual ACM Symposium on Principles of Programming Languages,* 104-112.

FOSDICK, L. D. AND L. J. OSTERWEIL [1976]. "Data flow analysis in software reliability," *Computing Surveys* **8:3**, 305-330.

FOSTER, J. M. [1968]. "A syntax improving program," *Computer J.* **11:1**, 31-34.

FRASER, C. W. [1977]. *Automatic Generation of Gode Generators,* Ph. D. Thesis, Yale Univ., New Haven, Conn.

FRASER, C. W. [1979]. "A compact, machine-independent peephole optimizer," *Sixth Annual ACM Symposium on Principles of Programming Languages,* 1-6.

FRASER, C. W. AND D. R. HANSON [1982]. "A machine-independent linker," *Software—Practice and Experience* **12**, 351-366.

FREDMAN, M. L., J. KOMLOS, AND E. SZEMEREDI [1984]. "Storing a sparse table with *O*(l) worst case access time," *J. ACM* **31:3**, 538-544.

FREGE, G. [1879]. "Begriffsschrift, a formula language, modeled upon that of arithmetic, for pure thought," in Heijenoort [1967], 1-82.

FREIBURGHOUSE, R. A. [1969]. "The Multics PL/I compiler," *AFIPS Fall Joint Computer Conference* **35**, 187-208.

FREIBURGHOUSE, R. A. [1974]. "Register allocation via usage counts," *Comm. ACM* **17:11**, 638-642.

FREUDENBERGER, S. M. [1984]. "On the use of global optimization algorithms for the detection of semantic programming errors," NSO-24, New York Univ.

FREUDENBERGER, S. M., J. T. SCHWARTZ, AND M. SHARIR [1983]. "Experience with the SETL optimizer," *TOPLAS* **5:1**, 26-45.

GAJEWSKA, H. [1975]. "Some statistics on the usage of the C language," AT&T Bell Laboratories, Murray Hill, N. J.

GALLER, B. A. AND M. J. FISCHER [1964]. "An improved equivalence algorithm," *Comm. ACM* **7:5**, 301-303.

GANAPATHI, M. [1980]. *Retargetable Code Generation and Optimization using Attribute Grammars,* Ph. D. Thesis, Univ. of Wisconsin, Madison, Wis.

GANAPATHI, M. AND C. N. FISCHER [1982]. "Description-driven code generation using attribute grammars," *Ninth ACM Symposium on Principles of Programming Languages,* 108-119.

GANAPATHI, M., C. N. FISCHER, AND J. L. HENNESSY [1982]. "Retargetable compiler code generation," *Computing Surveys* **14:4**, 573-592.

GANNON, J. D. AND J. J. HORNING [1975]. "Language design for programming reliability," *IEEE Trans. Software Engineering* **SE-1:2**, 179-191.

GANZINGER, H., R. GIEGERICH, U. MONCKE, AND R. WILHELM [1982]. "A truly generative semantics-directed compiler generator," *ACM SIGPLAN Notices* **17:6** (June) 172-184.

GANZINGER, H. AND K. RIPKEN [1980]. "Operator identification in Ada," *ACM SIGPLAN Notices* **15:2** (February) 30-42.

GAREY, M. R. AND D. S. JOHNSON [1979]. *Computers and Intractability: A Guide to the Theory of NP-Completeness,* Freeman, San Francisco.

GEAR, C. W. [1965]. "High speed compilation of efficient object code," *Comm. ACM* **8:8**, 483-488.

GESCHKE, C. M. [1972]. *Global Program Optimizations,* Ph. D. Thesis, Dept. of Computer Science, Carnegie-Mellon Univ.

GIEGERICH, R. [1983]. "A formal framework for the derivation of machine-specific optimizers," *TOPLAS* **5:3**, 422-448.

GIEGERICH, R. AND R. WILHELM [1978]. "Counter-one-pass features in one-pass compilation: a formalization using attribute grammars," *Information Processing Letters* **7:6**, 279-284.

GLANVILLE, R. S. [1977]. *A Machine Independent Algorithm for Code Generation and its Use in Retargetable Compilers,* Ph. D. Thesis, Univ. of California, Berkeley.

GLANVILLE, R. S. AND S. L. GRAHAM [1978]. "A new method for compiler code generation," *Fifth ACM Symposium on Principles of Programming Languages,* 231-240.

GRAHAM, R. M. [1964]. "Bounded context translation," *AFIPS Spring Joint Computer Conference* **40**, 205-217. Reprinted in Rosen [1967], pp. 184-205.

GRAHAM, S. L. [1980]. "Table-driven code generation," *Computer* **13:8**, 25-34.

GRAHAM, S. L. [1984]. "Code generation and optimization," in Lorho [1984], pp. 251-288.

GRAHAM, S. L., C. B. HALEY, AND W. N. JOY [1979]. "Practical LR error recovery," *ACM SIGPLAN Notices* **14:8**, 168-175.

GRAHAM, S. L., M. A. HARRISON, AND W. L. RUZZO [1980]. "An improved context-free recognizer," *TOPLAS* **2:3**, 415-462.

GRAHAM, S. L. AND S. P. RHODES [1975]. "Practical syntactic error recovery," *Comm. ACM* **18:11**, 639-650.

GRAHAM, S. L. AND M. WEGMAN [1976]. "A fast and usually linear algorithm for global data flow analysis," *J. ACM* **23:1**, 172-202.

GRAU, A. A., U. HILL, AND H. LANGMAACK [1967]. *Translation of Algol 60,* Springer-Verlag, New York.

HANSON, D. R. [1981]. "Is block structure necessary?" *Software—Practice and Experience* **11**, 853-866.

HARRISON, M. C. [1971]. "Implementation of the substring test by hashing," *Comm. ACM* **14:12**, 777-779.

HARRISON, W. [1975]. "A class of register allocation algorithms," RC-5342, IBM T. J. Watson Research Center, Yorktown Heights, N. Y.

HARRISON, W. [1977]. "Compiler analysis of the value ranges for variables," *IEEE Trans. Software Engineering* **3:3**.

HECHT, M. S. [1977]. *Flow Analysis of Computer Programs,* North-Holland, New York.

HECHT, M. S. AND J. B. SHAFFER [1975]. "Ideas on the design of a 'quad improver' for SIMPL-T, part 1: overview and intersegment analysis," Dept. of Computer Science, Univ. of Maryland, College Park, Md.

HECHT, M . S. AND J. D. ULLMAN [1972]. "Flow graph reducibility", *SIAM J. Computing* **1**, 188-202.

HECHT, M. S. AND J. D. ULLMAN [1974]. "Characterizations of reducible flow graphs," *J. ACM* **21**, 367-375.

HECHT, M. S. AND J. D. ULLMAN [1975]. "A simple algorithm for global data flow analysis programs," *SIAM J. Computing* **4**, 519-532.

HEIJENOORT, J. VAN [1967]. *From Frege to Gödel,* Harvard Univ. Press, Cambridge, Mass.

HENNESSY, J. [1981]. "Program optimization and exception handling," *Eighth Annual ACM Symposium on Principles of Programming Languages,* 200-206.

HENNESSY, J. [1982]. "Symbolic debugging of optimized code," *TOPLAS* **4:3**, 323-344.

HENRY, R. R. [1984]. *Graham-Glanville Code Generators,* Ph. D. Thesis, Univ. of California, Berkeley.

HEXT, J. B. [1967]. "Compile time type-matching," *Computer J.* **9**, 365-369.

HINDLEY, R. [1969]. "The principal type-scheme of an object in combinatory logic," *Trans. AMS* **146**, 29-60.

HOARE, C. A. R. [1962a]. "Quicksort," *Computer J.* **5:1**, 10-15.

HOARE, C. A. R. [1962b]. "Report on the Elliott Algol translator," *Computer J.* **5:2**, 127-129.

HOFFMAN, C. M. AND M. J. O'DONNELL [1982]. "Pattern matching in trees," *J. ACM* **29:1**, 68-95.

HOPCROFT, J. E. AND R. M. KARP [1971]. "An algorithm for testing the equivalence of finite automata," TR-71-114, Dept. of Computer Science, Cornell Univ. See Aho, Hopcroft, and Ullman [1974], pp. 143-145.

HOPCROFT, J. E. AND J. D. ULLMAN [1969]. *Formal Languages and Their Relation to Automata,* Addison-Wesley, Reading, Mass.

HOPCROFT, J. E. AND J. D. ULLMAN [1973]. "Set merging algorithms," *SIAM J. Computing* **2:3**, 294-303.

HOPCROFT, J. E. AND J. D. ULLMAN [1979]. *Introduction to Automata Theory, Languages, and Computation,* Addison-Wesley, Reading, Mass.

HORNING, J. J. [1976]. "What the compiler should tell the user," in Bauer and Eickel [1976].

HORWITZ, L. P., R. M. KARP, R. E. MILLER, AND S. WINOGRAD [1966]. "Index register allocation," *J. ACM* **13:1**, 43-61.

HUET, G. AND G. KAHN (EDS.) [1975]. *Proving and Improving Programs,* Colloque IRIA, Arc-et-Senans, France.

HUET, G. AND J.-J. LEVY [1979]. "Call-by-need computations in nonambiguous linear term rewriting systems," Rapport de Recherche 359, INRIA Laboria, Rocquencourt.

HUFFMAN, D. A. [1954]. "The synthesis of sequential machines," *J. Franklin Inst.* **257**, 3-4, 161, 190, 275-303.

HUNT, J. W. AND M. D. MCILROY [1976]. "An algorithm for differential file comparison," Computing Science Technical Report 41, AT&T Bell Laboratories, Murray Hill, N. J.

HUNT, J. W. AND T. G. SZYMANSKI [1977]. "A fast algorithm for computing longest common subsequences," *Comm. ACM* **20:5**, 350-353.

HUSKEY, H. D., M. H. HALSTEAD, AND R. MCARTHUR [1960]. "Neliac — a dialect of Algol," *Comm. ACM* **3:8**, 463-468.

ICHBIAH, J. D. AND S. P. MORSE [1970]. "A technique for generating almost optimal Floyd-Evans productions for precedence grammars," *Comm. ACM* **13:8**, 501-508.

INGALLS, D. H. H. [1978]. "The Smalltalk-76 programming system design and implementation," *Fifth Annual ACM Symposium on Principles of Programming Languages,* 9-16.

INGERMAN, P. Z. [1967]. "Panini-Backus form suggested," Comm. ACM **10:3**, 137.

IRONS, E. T. [1961]. "A syntax directed compiler for Algol 60," *Comm. ACM* **4:1**, 51-55.

IRONS, E. T. [1963]. "An error correcting parse algorithm," *Comm. ACM* **6:11**, 669-673.

IVERSON, K. [1962]. *A Programming Language,* Wiley, New York.

JANAS, J. M. [1980]. "A comment on 'Operator identification in Ada' by Ganzinger and Ripken," *ACM SIGPLAN Notices* **15:9** (September) 39-43.

Jarvis, J. F. [1976]. "Feature recognition in line drawings using regular expressions," *Proc. 3rd Intl. Joint Conf. on Pattern Recognition,* 189-192.

JAZAYERI, M., W. F. OGDEN, AND W. C. ROUNDS [1975]. "The intrinsic exponential complexity of the circularity problem for attribute grammars," *Comm. ACM* **18:12**, 697-706.

JAZAYERI, M. AND. POZEFSKY [1981]. "Space-efficient storage management in an attribute grammar evaluator," *TOPLAS* **3:4**, 388-404.

JAZAYERI, M. AND K. G. WALTER [1975]. "Alternating semantic evaluator," *Proc. ACM Annual Conference*, 230-234.

JENSEN, K. AND N. WIRTH [1975]. *Pascal User Manual and Report*, Springer-Verlag, New York.

JOHNSON, S. C. [1975]. "Yacc — yet another compiler compiler," Computing Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, N. J.

JOHNSON, S. C. [1978]. "A portable compiler: theory and practice," *Fifth Annual ACM Symposium on Principles of Programming Languages,* 97-104.

JOHNSON, S. C. [1979]. "A tour through the portable C compiler," AT&T Bell Laboratories, Murray Hill, N. J.

JOHNSON, S. C. [1983]. "Code generation for silicon," *Tenth Annual ACM Symposium on Principles of Programming Languages,* 14-19.

JOHNSON, S. C. AND M. E. LESK [1978]. "Language development tools," *Bell System Technical J.* **57:6**, 2155-2175.

JOHNSON, S. C. AND D. M. RITCHIE [1981]. "The C language calling sequence," Computing Science Technical Report 102, AT&T Bell Laboratories, Murray Hill, N. J.

JOHNSON, W. L., J. H. PORTER, S. I. ACKLEY, AND D. T. ROSS [1968]. "Automatic generation of efficient lexical processors using finite state techniques," *Comm. ACM* **11:12**, 805-813.

JOHNSSON, R. K. [1975]. *An Approach to Global Register Allocation,* Ph. D. Thesis, Carnegie-Mellon Univ., Pittsburgh, Pa.

JOLIAT, M. L. [1976]. "A simple technique for partial elimination of unit productions from LR(k) parser tables," *IEEE Trans. on Computers* **C-25:7**, 763-764.

JONES, N. D. [1980]. *Semantics Directed Gompiler Generation,* Lecture Notes in Computer Science **94**, Springer-Verlag, Berlin.

JONES, N. D. AND C. M. MADSEN [1980]. "Attribute-influenced LR parsing," in Jones [1980], pp. 393-407.

JONES, N. D. AND S. S. MUCHNICK [1976]. "Binding time optimization in programming languages," *Third ACM Symposium on Principles of Programming Languages,* 77-94.

JOURDAN, M. [1984]. "Strongly noncircular attribute grammars and their recursive evaluation," *ACM SIGPLAN Notices* **19:6**, 81-93.

KAHN, G., D. B. MACQUEEN, AND G. PLOTKIN [1984]. *Semantics of Data Types,* Lecture Notes in Computer Science **173**, Springer-Verlag, Berlin.

KAM, J. B. AND J. D. ULLMAN [1976]. "Global data flow analysis and iterative algorithms," *J. ACM* **23:1**, 158-171.

KAM, J. B. AND J. D. ULLMAN [1977]. "Monotone data flow analysis frameworks," *Acta Informatica* **7:3**, 305-318.

KAPLAN, M. AND J. D. ULLMAN [1980]. "A general scheme for the automatic inference of variable types," *J. ACM* **27:1**, 128-145.

KASAMI, T. [1965]. "An efficient recognition and syntax analysis algorithm for context-free languages," AFCRL-65-758, Air Force Cambridge Research Laboratory, Bedford, Mass.

KASAMI, T., W. W. PETERSON, AND N. TOKURA [1973]. On the capabilities of while, repeat, and exit statements," *Comm. ACM* **16:8**, 503-512.

KASTENS, U. [1980]. "Ordered attribute grammars," *Acta Informatica* **13:3**, 229-256.

KASTENS, U., B. HUTT, AND E. ZIMMERMANN [1982]. *GAG: A Practical Compiler Generator,* Lecture Notes in Computer Science **141**, Springer-Verlag, Berlin.

KASYANOV, V. N. [1973]. "Some properties of fully reducible graphs," *Information Processing Letters* **2:4**, 113-117.

KATAYAMA, T. [1984]. "Translation of attribute grammars into procedures," *TOPLAS* **6:3**, 345-369.

KENNEDY, K. [1971]. "A global flow analysis algorithm," *Intern. J. Computer Math. Section A* **3**, 5-15.

KENNEDY, K. [1972]. "Index register allocation in straight line code and simple loops," in Rustin [1972], pp. 51-64.

KENNEDY, K. [1976]. "A comparison of two algorithms for global flow analysis," *SIAM J. Computing* **5:1**, 158-180.

KENNEDY, K . [1981]. "A survey of data flow analysis techniques, " in Muchnick and Jones [1981], pp. 5-54.

KENNEDY, K. AND J. RAMANATHAN [1979]. "A deterministic attribute grammar evaluator based on dynamic sequencing," *TOPLAS* **1:1**, 142-160.

KENNEDY, K. AND S. K. WARREN [1976]. "Automatic generation of efficient evaluators for attribute grammars," *Third ACM Symposium on Principles of Programming Languages*, 32-49.

KERNIGHAN, B. W. [1975]. "Ratfor — a preprocessor for a rational Fortran," *Software—Practice and Experience* **5:4**, 395-406.

KERNIGHAN, B. W. [1982]. "PIC — a language for typesetting graphics," *Software—Practice and Experience* **12:1**, 1-21.

KERNIGHAN, B. W. AND L. L. CHERRY [1975]. "A system for typesetting mathematics," *Comm. ACM* **18:3**, 151-157.

KERNIGHAN, B. W. AND R. PIKE [1984]. *The UNIX Programming Environment*, Prentice-Hall, Englewood Cliffs, N. J.

KERNIGHAN, B. W. AND D. M. RITCHIE [1978]. *The C Programming Language*, Prentice-Hall, Englewood Cliffs, N. J.

KILDALL, G. [1973]. "A unified approach to global program optimization," *ACM Symposium on Principles of Programming Languages*, 194-206.

KLEENE, S. C. [1956]. "Representation of events in nerve nets," in Shannon and McCarthy [1956], pp. 3-40.

KNUTH, D. E. [1962]. "A history of writing compilers," *Computers and Automation* (December) 8-18. Reprinted in Pollack [1972], pp. 38-56.

KNUTH, D. E. [1964]. "Backus Normal Form vs. Backus Naur Form," *Comm. ACM* **7:12**, 735-736.

KNUTH, D. E. [1965]. "On the translation of languages from left to right," *Information and Control* **8:6**, 607-639.

KNUTH, D. E. [1968]. "Semantics of context-free languages," *Mathematical Systems Theory* **2:2**, 127-145. Errata **5:1** (1971) 95-96.

KNUTH, D. E. [1971a]. "Top-down syntax analysis," *Acta Informatica* **1:2**, 79-110.

KNUTH, D. E. [1971b]. "An empirical study of FORTRAN programs," *Software—Practice and Experience* **1:2**, 105-133.

KNUTH, D. E. [1973a]. *The Art of Computer Programming:* Vol. 1, 2nd. Ed., *Fundamental Algorithms*, Addison-Wesley, Reading, Mass.

KNUTH, D. E. [1973b]. *The Art of Computer Programming*: Vol. 3, *Sorting and Searching*, Addison-Wesley, Reading, Mass.

KNUTH, D. E. [1977]. "A generalization of Dijkstra's algorithm," *Information Processing Letters* **6**, 1-5.

KNUTH, D. E. [1984a]. *The TEXbook*, Addison-Wesley, Reading, Mass.

KNUTH, D. E. [1984b]. "Literate programming," *Computer J.* **28:2**, 97-111.

KNUTH, D. E. [1985,1986]. *Computers and Typesetting*, Vol. 1: *TEX*, Addison-Wesley, Reading, Mass. A preliminary version has been published under the title, "*TEX: The Program.*"

KNUTH, D. E., J. H. MORRIS, AND V. R. PRATT [1977]. Fast pattern matching in strings," *SIAM J. Computing* **6:2**, 323-350.

KNUTH, D. E. AND L. TRABB PARDO [1977]. "Early development of programming languages," *Encyclopedia of Computer Science and Technology* **7**, Marcel Dekker, New York, 419-493.

KORENJAK, A. J. [1969]. "A practical method for constructing LR(k) processors," *Comm. ACM* **12:11**, 613-623.

KOSARAJU, S. R. [1974]. "Analysis of structured programs," *J. Computer and System Sciences* **9:3**, 232-255.

KOSKIMIES, K. AND K.-J. RÄIHÄ [1983]. "Modelling of space-efficient one-pass translation using attribute grammars," *Software—Practice and Experience* **13**, 119-129.

KOSTER, C. H. A. [1971]. "Affix grammars," in Peck [1971], pp. 95-109.

KOU, L. [1977]. "On live-dead analysis for global data flow problems," *J. ACM* **24:3**, 473-483.

KRISTENSEN, B. B. AND O. L. MADSEN [1981]. "Methods for computing LALR(k) lookahead," *TOPLAS* **3:1**, 60-82.

KRON, H. [1975]. *Tree Templates and Subtree Transformational Grammars*, Ph. D. Thesis, Univ. of California, Santa Cruz.

LALONDE, W. R. [1971]. "An efficient LALR parser generator," Tech. Rep. 2, Computer Systems Research Group, Univ. of Toronto.

LALONDE, W. R. [1976]. "On directly constructing LR(k) parsers without chain reductions," *Third ACM Symposium on Principles of Programming Languages*, 127-133.

LALONDE, W. R., E. S. LEE, AND J. J. HORNING [1971]. "An LALR(k) parser generator," *Proc. IFIP Congress 71* **TA-3**, North-Holland, Amsterdam, 153-157.

LAMB, D. A. [1981]. "Construction of a peephole optimizer," *Software— Practice and Experience* **11**, 638-647.

LAMPSON, B. W. [1982]. "Fast procedure calls," *ACM SIGPLAN Notices* **17:4** (April) 66-76.

LANDIN, P. J. [1964]. "The mechanical evaluation of expressions," *Computer J.* **6:4**, 308-320.

LECARME, O. AND M.-C. PEYROLLE-THOMAS [1978]. "Self-compiling compilers: an appraisal of their implementation and portability," *Software—Practice and Experience* **8**, 149-170.

LEDGARD, H. F. [1971]. "Ten mini-languages: a study of topical issues in programming languages," *Computing Surveys* **3:3**, 115-146.

LEINIUS, R . P. [1970]. *Error Detection and Recovery for Syntax Directed Compiler Systems*, Ph. D. Thesis, University of Wisconsin, Madison.

LENGAUER, T. AND R. E. TARJAN [1979]. "A fast algorithm for finding dominators in a flowgraph," *TOPLAS* **1**, 121-141.

LESK, M. E. [1975]. "Lex — a lexical analyzer generator," Computing Science Technical Report 39, AT&T Bell Laboratories, Murray Hill, N. J.

LEVERETT, B. W. [1982]. "Topics in code generation and register allocation," CMU CS-82-130, Computer Science Dept., Carnegie-Mellon Univ., Pittsburgh, Pennsylvania.

LEVERETT, B. W., R. G. G. CATTELL, S. O. HOBBS, J. M. NEWCOMER, A. H. REINER, B. R. SCHATZ, AND W. A. WULF [1980]. "An overview of the production-quality compiler-compiler project," *Computer* **13:8**, 38-40.

LEVERETT, B. W. AND T. G. SZYMANSKI [1980]. "Chaining span-dependent jump instructions," *TOPLAS* **2:3**, 274-289.

LEVY, J. P. [1975]. "Automatic correction of syntax errors in programming languages," *Acta Informatica* **4**, 271-292.

LEWIS, P. M., II, D. J. ROSENKRANTZ, AND R. E. STEARNS [1974]. "Attributed translations," *J. Computer and System Sciences* **9:3**, 279-307.

LEWIS, P. M., II, D. J. ROSENKRANTZ, AND R. E. STEARNS [1976]. *Compiler Design Theory*, Addison-Wesley, Reading, Mass.

LEWIS, P. M., II AND R. E. STEARNS [1968]. "Syntax-directed transduction," *J. ACM* **15:3**, 465-488.

LORHO, B. [1977]. "Semantic attribute processing in the system Delta," in Ershov and Koster [1977], pp. 21-40.

LORHO, B. [1984]. *Methods and Tools for Compiler Construction*, Cambridge Univ. Press.

LORHO, B. AND C. PAIR [1975]. "Algorithms for checking consistency of attribute grammars," in Huet and Kahn [1975], pp. 29-54.

LOW, J. AND P. ROVNER [1976]. "Techniques for the automatic selection of data structures," *Third ACM Symposium on Principles of Programming Languages*, 58-67.

LOWRY, E. S. AND C. W. MEDLOCK [1969]. "Object code optimization", *Comm. ACM* **12**, 13-22.

LUCAS, P. [1961]. "The structure of formula translators," *Elektronische Rechenanlagen* **3**, 159-166.

LUNDE, A. [1977]. "Empirical evaluation of some features of instruction set processor architectures," *Comm. ACM* **20:3**, 143-153.

LUNELL, H. [1983]. *Code Generator Writing Systems*, Ph. D. Thesis, Linköping University, Linkoping, Sweden.

MACQUEEN, D. B., G. P. PLOTKIN, AND R. SETHI [1984]. "An ideal model of recursive polymorphic types," *Eleventh Annual ACM Symposium on Principles of Programming Languages*, 165-174.

MADSEN, O. L. [1980]. "On defining semantics by means of extended attribute grammars," in Jones [1980], pp. 259-299.

MARILL, T. [1962]. "Computational chains and the simplification of computer programs," *IRE Trans. Electronic Computers* **EC-11:2**, 173-180.

MARTELLI, A. AND U. MONTANARI [1982]. "An efficient unification algorithm," *TOPLAS* **4:2**, 258-282.

MAUNEY, J. AND C. N. FISCHER [1982]. "A forward move algorithm for LL and LR parsers," *ACM SIGPLAN Notices* **17:4**, 79-87.

MAYOH, B. H. [1981]. "Attribute grammars and mathematical semantics," *SIAM J. Computing* **10:3**, 503-518.

MCCARTHY, J. [1963]. "Towards a mathematical science of computation," *Information Processing 1962*, North-Holland, Amsterdam, 21-28.

MCCARTHY, J. [1981]. "History of Lisp", in Wexelblat [1981], pp. 173-185.

MCCLURE, R. M. [1965]. "TMG — a syntax-directed compiler," *Proc. 20th ACM National Conf.*, 262-274.

MCCRACKEN, N. J. [1979]. *An Investigation of a Programming Language with a Polymorphic Type Structure*, Ph. D. Thesis, Syracuse University, Syracuse, N. Y.

MCCULLOUGH, W . S. AND W. PITTS [1943]. "A logical calculus of the ideas immanent in nervous activity," *Bulletin of Math. Biophysics* **5**, 115-133.

MCKEEMAN, W. M. [1965]. "Peephole optimization," *Comm. ACM* **8:7**, 443-444

MCKEEMAN, W. M. [1976]. "Symbol table access," in Bauer and Eickel [1976], pp. 253-301.

MCKEEMAN, W. M., J. J. HORNING, AND D. B. WORTMAN [1970]. *A Compiler Generator*, Prentice-Hall, Englewood Cliffs, N. J.

MCNAUGHTON, R. AND H. YAMADA [1960]. "Regular expressions and state graphs for automata," *IRE Trans. on Electronic Computers* **EC-9:1**, 38-47.

MEERTENS, L. [1983]. "Incremental polymorphic type checking in B," *Tenth ACM Symposium on Principles of Programming Languages*, 265-275.

METCALF, M. [1982]. *Fortran Optimization*, Academic Press, New York.

MILLER, R. E. AND J. W. THATCHER (EDS.) [1972]. *Complexity of Computer Computations*, Academic Press, New York.

MILNER, R. [1978]. "A theory of type polymorphism in programming," *J. Computer and System Sciences* **17:3**, 348-375.

MILNER, R. [1984]. "A proposal for standard ML," *ACM Symposium on Lisp and Functional Programming*, 184- 197.

MINKER, J. AND R. G. MINKER [1980]. "Optimization of boolean expressions—historical developments," *A. of the History of Computing* **2:3**, 227-238.

MITCHELL, J. C. [1984]. "Coercion and type inference," *Eleventh ACM Symposium on Principles of Programming Languages*, 175-185.

MOORE, E. F. [1956]. "Gedanken experiments in sequential machines," in Shannon and McCarthy [1956], pp. 129-153.

MOREL, E. AND C. RENVOISE [1979]. "Global optimization by suppression of partial redundancies," *Comm. ACM* **22**, 96-103.

MORRIS, J. H. [1968a]. *Lambda-Calculus Models of Programming Languages*, Ph. D. Thesis, MIT, Cambridge, Mass.

MORRIS, R. [1968b]. "Scatter storage techniques," *Comm. ACM* **11:1**, 38-43.

MOSES, J. [1970]. "The function of FUNCTION in Lisp," *SIGSAM Bulletin* **15** (July) 13-27.

MOULTON, P. G. AND M. E. MULLER [1967]. "DITRAN — a compiler emphasizing diagnostics," *Comm. ACM* **10:1**, 52-54.

MUCHNICK, S. S. AND N. D. JONES [1981]. *Program Flow Analysis: Theory and Applications*, Prentice-Hall, Englewood Cliffs, N. J.

NAKATA, I. [1967]. "On compiling algorithms for arithmetic expressions," *Comm. ACM* **10:8**, 492-494.

NAUR, P. (ED.) [1963]. "Revised report on the algorithmic language Algol 60," *Comm. ACM* **6:1**, 1-17.

NAUR, P. [1965]. "Checking of operand types in Algol compilers," *BIT* **5**, 151-163.

NAUR, P. [1981]. "The European side of the last phase of the development of Algol 60," in Wexelblat [1981], pp. 92-139, 147-161.

NEWEY, M. C., P. C. POOLE, AND W. M. WAITE [1972]. "Abstract machine modelling to produce portable software — a review and evaluation," *Software—Practice and Experience* **2:2**, 107-136.

NEWEY, M. C. AND W. M. WAITE [1985]. "The robust implementation of sequence-controlled iteration," *Software—Practice and Experience* **15:7**, 655-668.

NICHOLLS, J. E. [1975]. *The Structure and Design of Programming Languages*, Addison-Wesley, Reading, Mass.

NIEVERGELT, J. [1965]. "On the automatic simplification of computer code," *Comm. ACM* **8:6**, 366-370.

NORI, K. V., U. AMMANN, K. JENSEN, H. H. NÄGELI, AND CH. JACOBI [1981]. "Pascal P implementation notes," in Barron [1981], pp. 125-170.

OSTERWEIL, L. J. [1981]. "Using data flow tools in software engineering," in Muchnick and Jones [1981], pp. 237-263.

PAGER, D. [1977a]. "A practical general method for constructing LR(k) parsers," *Acta Informatica* **7**, 249-268.

PAGER, D. [1977b]. "Eliminating unit productions from LR(k) parsers," Acta Informatica **9**, 31-59.

PAI, A. B. AND R. B. KIEBURTZ [1980]. "Global context recovery: a new strategy for syntactic error recovery by table-driven parsers," *TOPLAS* **2:1**, 18-41.

PAIGE, R. AND J. T. SCHWARTZ [1977]. "Expression continuity and the formal differentiation of algorithms," *Fourth ACM Symposium on Principles of Programming Languages*, 58-71.

PALM, R. C., JR. [1975]. "A portable optimizer for the language C," M. Sc. Thesis, MIT, Cambridge, Mass.

PARK, J. C. H., K. M. CHOE, AND C. H. CHANG [1985]. A new analysis of LALR formalisms," *TOPLAS* 7:1, 159-175.

PATERSON, M. S. AND M. WEGMAN [1978]. "Linear unification," *J. Computer and System Sciences* **16:2**, 158-167.

PECK, J. E. L. [1971]. *Algol 68 Implementation*, North-Holland, Amsterdam.

PENNELLO, T. AND F. DEREMER [1978]. "A forward move algorithm for LR error recovery," *Fifth Annual ACM Symposium on Principles of Programming Languages*, 241-254.

PENNELLO, T., F. DEREMER, AND R. MEYERS [1980]. A simplified operator identification scheme for Ada," *ACM SIGPLAN Notices* **15:7** (July-August) 82-87.

PERSCH, G., G. WINTERSTEIN, M. DAUSSMANN, AND S. DROSSOPOULOU [1980]. "Overloading in preliminary Ada," *ACM SIGPLAN Notices* **15:11** (November) 47-56.

PETERSON, W. W. [1957]. "Addressing for random access storage," *IBM J. Research and Development* **1:2**, 130-146.

POLLACK, B. W. [1972]. *Compiler Techniques*, Auerbach Publishers, Princeton, N. J.

POLLOCK, L. L. AND M. L. SOFFA [1985]. "Incremental compilation of locally optimized code," *Twelfth Annual ACM Symposium on Principles of Programming Languages*, 152-164.

POWELL, M. L. [1984]. "A portable optimizing compiler for Modula-2," *ACM SIGPLAN Notices* **19:6**, 310-318.

PRATT, T. W. [1984]. *Programming Languages: Design and Implementation*, 2nd Ed., Prentice-Hall, Englewood Cliffs, N. J.

PRATT, V. R. [1973]. "Top down operator precedence," *ACM Symposium on Principles of Programming Languages*, 41-51.

PRICE, C. E. [1971]. "Table lookup techniques," *Computing Surveys* **3:2**, 49-65.

PROSSER, R. T. [1959]. "Applications of boolean matrices to the analysis of flow diagrams," *AFIPS Eastern Joint Computer Conf.*, Spartan Books, Baltimore, Md., 133-138.

PURDOM, P. AND C. A. BROWN [1980]. "Semantic routines and LR(k) parsers, " *Acta Informatica* **14:4**, 299-315.

PURDOM, P. W. AND E. F. MOORE [1972]. "Immediate predominators in a directed graph," *Comm. ACM* **15:8**, 777-778.

RABIN, M. O. AND D. SCOTT. [1959]. "Finite automata and their decision problems," *IBM J. Research and Development* **3:2**, 114-125.

RADIN, G. AND H. P. ROGOWAY [1965]. "NPL: Highlights of a new programming language," *Comm. ACM* **8:1**, 9-17.

RÄIHÄ, K.-J. [1981]. *A Space Management Technique for Multi-Pass Attribute Evaluators*, Ph. D. Thesis, Report A-1981-4, Dept. of Computer Science, University of Helsinki.

RÄIHÄ, K.-J. AND M. SAARINEN [1982]. "Testing attribute grammars for circularity," *Acta Informatica* **17**, 185-192.

RÄIHÄ, K.-J., M. SAARINEN, M. SARJAKOSKI, S. SIPPU, E. SOISALON-SOININEN, AND M. TIENARI [1983]. "Revised report on the compiler writing system HLP78," Report A-1983-1, Dept. of Computer Science, University of Helsinki.

RANDELL, B. AND L. J. RUSSELL [1964]. *Algol 60 Implementation*, Academic Press, New York.

REDZIEJOWSKI, R. R. [1969]. "On arithmetic expressions and trees," *Comm. ACM* **12:2**, 81-84.

REIF, J. H. AND H. R. LEWIS [1977]. "Symbolic evaluation and the global value graph," *Fourth ACM Symposium on Principles of Programming Languages*, 104-118.

REISS, S. P. [1983]. "Generation of compiler symbol processing mechanisms from specifications," *TOPLAS* **5:2**, 127-163.

REPS, T. W. [1984]. *Generating Language-Based Environments*, MIT Press, Cambridge, Mass.

REYNOLDS, J. C. [1985]. "Three approaches to type structure," *Mathematical Foundations of Software Development*, Lecture Notes in Computer Science **185**, Springer-Verlag, Berlin, 97-138.

RICHARDS, M. [1971]. "The portability of the BCPL compiler," *Software—Practice and Experience* **1:2**, 135-146.

RICHARDS, M. [1977]. "The implementation of the BCPL compiler," in P. J. Brown (ed.), *Software Portability: An Advanced Course*, Cambridge University Press.

RIPKEN, K. [1977]. "Formale beschreibun von maschinen, implementierungen und optimierender maschinen-codeerzeugung aus attributierten programmgraphe," TUM-INFO-7731, Institut für Informatik, Universität München, Munich.

RIPLEY, G. D. AND F. C. DRUSEIKIS [1978]. "A statistical analysis of syntax errors," *Computer Languages* **3**, 227-240.

RITCHIE, D. M. [1979]. "A tour through the UNIX C compiler," AT&T Bell Laboratories, Murray Hill, N. J.

RITCHIE, D. M. AND K. THOMPSON [1974]. "The UNIX time-sharing system," *Comm. ACM* **17:7**, 365-375.

ROBERTSON, E. L. [1979]. "Code generation and storage allocation for machines with span-dependent instructions," *TOPLAS* **1:1**, 71-83.

ROBINSON, J. A. [1965]. "A machine-oriented logic based on the resolution principle," *J. ACM* **12:1**, 23-41.

ROHL, J. S. [1975]. *An Introduction to Compiler Writing*, American Elsevier, New York.

ROHRICH, J. [1980]. "Methods for the automatic construction of error correcting parsers," *Acta Informatica* **13:2**, 115-139.

ROSEN, B. K. [1977]. "High-level data flow analysis," *Comm. ACM* **20**, 712-724.

ROSEN, B. K. [1980]. "Monoids for rapid data flow analysis," *SIAM J. Computing* **9:1**, 159-196.

ROSEN, S. [1967]. *Programming Systems and Languages*, McGraw-Hill, New York.

ROSENKRANTZ, D. J. AND R. E. STEARNS [1970]. Properties of deterministic top-down grammars," *Information and Control* **17:3**, 226-256.

ROSLER, L. [1984]. "The evolution of C—past and future," *AT&T Bell Labs Technical Journal* **63:8**, 1685-1699.

RUSTIN, R. [1972]. *Design and Optimization of Compilers*, Prentice-Hall, Englewood Cliffs, N.J.

RYDER, B. G. [1979]. "Constructing the call graph of a program," *IEEE Trans. Software Engineering* **SE-5:3**, 216-226.

RYDER, B. G. [1983]. "Incremental data flow analysis," *Tenth ACM Symposium on Principles of Programming Languages*, 167-176.

SAARINEN, M. [1978]. "On constructing efficient evaluators for attribute grammars," *Automata, Languages and Programming, Fifth Colloquium*, Lecture Notes in Computer Science **62**, Springer-Verlag, Berlin, 382-397.

SAMELSON, K. AND F. L. BAUER [1960]. "Sequential formula translation," *Comm. ACM* **3:2**, 76-83.

SANKOFF, D. AND J. B. KRUSKAL (EDS.) [1983]. *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*, Addison-Wesley, Reading, Mass.

SCARBOROUGH, R. G. AND H. G. KOLSKY [1980]. Improved optimization of Fortran object programs," *IBM J. Research and Development* **24:6**, 660-676.

SCHAEFER, M. [1973]. *A Mathematical Theory of Global Program Optimization*, Prentice Hall, Englewood Cliffs, N. J.

SCHONBERG, E., J. T. SCHWARTZ, AND M. SHARIR [1981]. "An automatic technique for selection of data representations in SETL Programs," *TOPLAS* **3:2**, 126-143.

SCHORRE, D. V. [1964]. "Meta-II: a syntax-oriented compiler writing language," *Proc. 19th ACM National Conf.*, D1.3-1 - D1.3-11.

SCHWARTZ, J. T. [1973]. *On Programming: An Interim Report on the SETL Project*, Courant Inst., New York.

SCHWARTZ, J. T. [1975a]. "Automatic data structure choice in a language of very high level," *Comm. ACM* **18:12**, 722-728.

SCHWARTZ, J. T. [1975b]. "Optimization of very high level languages," *Computer Languages*. Part I: "Value transmission and its corollaries," **1:2**, 161-194; part II: "Deducing relationships of inclusion and membership," **1:3**, 197-218.

SEDGEWICK, R. [1978]. "Implementing Quicksort programs," *Comm. ACM* **21**, 847-857.

SETHI, R. [1975]. "Complete register allocation problems," *SIAM J. Computing* **4:3**, 226-248.

Sethi, R. AND J. D. ULLMAN [1970]. "The generation of optimal code for arithmetic expressions," *J. ACM* **17:4**, 715-728.

SHANNON, C. AND J. MACARTHY [1956]. *Automata Studies*, Princeton University Press.

SHERIDAN, P. B. [1959]. "The arithmetic translator-compiler of the IBM Fortran automatic coding system," *Comm. ACM* **2:2**, 9-21.

SHIMASAKI, M., S. FUKAYA, K. IKEDA, AND T. KIYONO [1980]. "An analysis of Pascal programs in compiler writing," *Software—Practice and Experience* **10:2**, 149-157.

SHUSTEK, L. J. [1978]. "Analysis and performance of computer instruction sets," SLAC Report 205, Stanford Linear Accelerator Center, Stanford University, Stanford, California.

SIPPU, S. [1981]. "Syntax error handling in compilers," Rep. A-1981-1, Dept. of Computer Science, Univ. of Helsinki, Helsinki, Finland.

SIPPU, S. AND E. SOISALON-SOININEN [1983]. "A syntax-error-handling technique and its experimental analysis," *TOPLAS* **5:4**, 656-679.

SOISALON-SOININEN, E. [1980]. "On the space optimizing effect of eliminating single productions from LR parsers," *Acta Informatica* **14**, 157-174.

SOISALON-SOININEN, E. AND E. UKKONEN [1979]. "A method for transforming grammars into LL(k) form," *Acta Informatica* **12**, 339-369.

SPILLMAN, T. C. [1971]. "Exposing side effects in a PL/I optimizing compiler," *Information Processing 71*, North-Holland, Amsterdam, 376-381.

STEARNS, R. E. [1971]. "Deterministic top-down parsing," *Proc. 5th Annual Princeton Conf. on Information Sciences and Systems*, 182-188.

STEEL, T. B., JR. [1961]. "A first version of Uncol," *Western Joint Computer Conference*, 371-378.

STEELE, G. L., JR. [1984]. *Common LISP*, Digital Press, Burlington, Mass.

STOCKHAUSEN, P. F. [1973]. "Adapting optimal code generation for arithmetic expressions to the instruction sets available on present-day computers," *Comm. ACM* **16:6**, 353-354. Errata: **17:10** (1974) 591.

STONEBRAKER, M., E. WONG, P. KREPS, AND G. HELD [1976]. "The design and implementation of INGRES," *ACM Trans. Database Systems* **1:3**, 189-222.

STRONG, J., J. WEGSTEIN, A. TRITTER, J. OLSZTYN, O. MOCK, AND T. STEEL [1958]. "The problem of programming communication with changing machines: a proposed solution," *Comm. ACM* **1:8** (August) 12-18. Part 2: **1:9** (September) 9-15. Report of the Share Ad-Hoc committee on Universal Languages.

STROUSTRUP, B. [1986]. *The C++ Programming Language*, Addison-Wesley, Reading, Mass.

SUZUKI, N. [1981]. "Inferring types in Smalltalk," *Eighth ACM Symposium on Principles of Programming Languages*, 187- 199.

SUZUKI, N. AND K. ISHIHATA [1977]. "Implementation of array bound checker," *Fourth ACM Symposium on Principles of Programming Languages*, 132-143.

SZYMANSKI, T. G. [1978]. "Assembling code for machines with span-dependent instructions," *Comm. ACM* **21:4**, 300-308.

TAI, K. C. [1978]. "Syntactic error correction in programming languages," *IEEE Trans. Software Engineering* **SE-4:5**, 414-425.

TANENBAUM, A. S., H. VAN STAVEREN, E. G. KEIZER, AND J. W. STEVENSON [1983]. "A practical tool kit for making portable compilers," *Comm. ACM* **26:9**, 654-660.

TANENBAUM, A. S., H. VAN STAVEREN, AND J. W. STEVENSON [1982]. "Using peephole optimization on intermediate code," *TOPLAS* **4:1**, 21-36.

TANTZEN, R. G. [1963]. "Algorithm 199: Conversions between calendar date and Julian day number," *Comm. ACM* **6:8**, 443.

TARHIO, J. [1982]. "Attribute evaluation during LR parsing," Report A-1982-4, Dept. of Computer Science, University of Helsinki.

TARJAN, R. E. [1974a]. "Finding dominators in directed graphs," *SIAM J. Computing* **3:1**, 62-89.

TARJAN, R. E. [1974b]. "Testing flow graph reducibility," *J. Computer and System Sciences* **9:3**, 355-365.

TARJAN, R. E. [1975]. "Efficiency of a good but not linear set union algorithm," *JACM* **22:2**, 215-225.

TARJAN, R. E. [1981]. "A unified approach to path problems," *J. ACM* **28:3**, 577-593. And "Fast algorithms for solving path problems," *J. ACM* **28:3** 594-614.

TARJAN, R. E. AND A. C. YAO [1979]. "Storing a sparse table," *Comm. ACM* **22:11**, 606-611.

TENNENBAUM, A. M. [1974]. "Type determination in very high level languages," NSO-3, Courant Institute of Math. Sciences, New York Univ.

TENNENT, R. D. [1981]. *Principles of Programming Languages*, Prentice-Hall International, Englewood Cliffs, N. J.

THOMPSON, K. [1968]. "Regular expression search algorithm," *Comm. ACM* **11:6**, 419-422.

TJIANG, S. W. K. [1986]. "Twig language manual," Computing Science Technical Report 120, AT&T Bell Laboratories, Murray Hill, N. J.

TOKUDA, T. [1981]. "Eliminating unit reductions from LR(k) parsers using minimum contexts," *Acta Informatica* **15**, 447-470.

TRICKEY, H. W. [1985]. *Compiling Pascal Programs into Silicon*, Ph. D. Thesis, Stanford Univ.

ULLMAN, J. D. [1973]. "Fast algorithms for the elimination of common subexpressions," *Acta Informatica* **2**, 191-213.

ULLMAN, J. D. [1982]. *Principles of Database Systems*, 2nd Ed., Computer Science Press, Rockville, Md.

ULLMAN, J. D. [1984]. *Computational Aspects of VLSI*, Computer Science Press, Rockville, Md.

Vysotsky, V. AND P. WEGNER [1963]. "A graph theoretical Fortran source language analyzer," manuscript, AT&T Bell Laboratories, Murray Hill, N. J.

WAGNER, R. A. [1974]. "Order-*n* correction for regular languages," *Comm. ACM* **16:5**, 265-268.

WAGNER, R. A. AND M. J. FISCHER [1974]. The string-to-string correction problem," *J. ACM* **21:1**, 168-174.

WAITE, W. M. [1976a]. "Code generation," in Bauer and Eickel [1976], 302-332.

WAITE, W. M. [1976b]. "Optimization," in Bauer and Eickel [1976], 549-602.

WAITE, W. M. AND L. R. CARTER [1985]. "The cost of a generated parser, "*Software — Practice and Experience* **15:3**, 221-237.

WASILEW, S. G. [1971]. *A Compiler Writing System with Optimization Capabilities for Complex Order Structures*, Ph. D. Thesis, Northwestern Univ., Evanston, Ill.

WATT, D. A. [1977]. "The parsing problem for affix grammars," *Acta Informatica* **8**, 1-20.

WEGBREIT, B. [1974]. "The treatment of data types in EL1," *Comm. ACM* **17:5**, 251-264.

WEGBREIT, B. [1975]. "Property extraction in well-founded property sets," *IEEE Trans. on Software Engineering* **1:3**, 270-285.

WEGMAN, M. N. [1983]. "Summarizing graphs by regular expressions," *Tenth Annual ACM Symposium on Principles of Programming Languages*, 203-216.

WEGMAN, M. N. AND F. K. ZADECK [1985]. "Constant propagation with conditional branches," *Twelfth Annual ACM Symposium on Principles of Programming Languages*, 291-299.

WEGSTEIN, J. H. [1981]. "Notes on Algol 60" , in Wexelblat [1981], pp. 126-127.

WEIHL, W. E. [1980]. "Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables," *Seventh Annual ACM Symposium on Principles of Programming Languages*, 83-94.

WEINGART, S. W. [1973]. *An Efficient and Systematic Method of Code Generation*, Ph. D. Thesis, Yale University, New Haven, Connecticut.

WELSH, J., W. J. SNEERINGER, AND C. A. R. HOARE [1977]. "Ambiguities and insecurities in Pascal," *Software—Practice and Experience* **7:6**, 685-696.

WEXELBLAT, R. L. [1981]. *History of Programming Languages*, Academic Press, New York.

WIRTH, N. [1968]. "PL 360 — a programming language for the 360 computers," *J. ACM* **15:1**, 37-74.

WIRTH, N. [1971]. "The design of a Pascal compiler," *Software—Practice and Experience* **1:4**, 309-333.

WIRTH, N. [1981]. "Pascal-S: A subset and its implementation," in Barron [1981], pp. 199-259.

WIRTH, N. AND H. WEBER [1966]. "Euler: a generalization of Algol and its formal definition: Part I," *Comm. ACM* **9:1**, 13-23.

WOOD, D. [1969]. "The theory of left factored languages," *Computer J.* **12:4**, 349-356.

WULF, W. A., R. K. JOHNSSON, C. B. WEINSTOCK, S. O. HOBBS, AND C. M. GESCHKE [1975]. *The Design of an Optimizing Compiler*, American Elsevier, New York.

YANNAKAKIS, M. [1985]. Private communication.

YOUNGER, D. H. [1967]. "Recognition and parsing of context-free languages in time n^3," *Information and Control* **10:2**, 189-208.

ZELKOWITZ, M. V. AND W. G. BAIL [1974]. "Optimization of structured programs," *Software—Practice and Experience* **4:1**, 51-57.

ÍNDICE ALFABÉTICO

A

Abel, N. E., 313
Abelson, H.,199
Abordagem
- conservativa, 266, 267, 274, 285, 288, 301
- segura, *ver* Abordagem conservativa
Aceitação, 52-53, 88
Acesso
- profundo, 182
- superficial, 182
Ação semântica, 2, 4
Ada, 147, 154, 156, 177
Adrion, W. R., 315
AFD, *ver* Autômato finito determinístico
AFN, *ver* Autômato finito não determinístico
Aho, A.V., 90, 118, 125, 168, 192, 199, 246, 248, 253, 315
Aigrain, P., 253
Alfabeto, 42
- binário, 42
Algol, 11, 36, 37, 70, 118, 184, 198, 221, 243
Algol-68, 10, 39, 165, 221
Algoritmo
- CYK, *Ver* Algoritmo Cocker-Younger-Kasami
- de Cocker-Younger-Kasami, 72, 118
- de Earley, 72, 118
- KMP, 68
- Knuth-Morris_Pratt, 70
- - *ver também* Algoritmo KMP
Alias, *ver* Pseudônimos
Alinhamento, de dados, 171, 204
Aliviar código, 311
Allen, F.E., 313-315
Alocação
- de memória, 172-177, 186, 190-193
- de pilha, 172, 173-177, 226, 227-228
- de registradores, 224, 235-237, 243-244, 323-325
- dinâmica, 172, 190-191
- em memória *heap*, 172, 176-177, 190-192
- estática, 172-173, 226-227, 228
- explícita, 190, 191
- global de registradores, 235-236
- implícita, 190, 192
Alternativas, 75
Amarração, de nomes, 169
Ambiente, 197
- de ativação, 197
- de uma passagem, 197
- léxico, 197
Ambigüidade, 14, 77, 78-79, 82, 85, 89, 99, 107-113, 251
Ammann, U., 37, 253, 318, 321
Análise, 1-5
- de fluxo de dados, 254, 264-271
- - iterativa, 271-276, 293-294, 302-303
- de intervalo, 271, 288, 291, 314
- - *ver também* Análise T1-T2
- gramatical, 3, 7, 14, 18-21, 25, 31, 38, 72-119 (*ver* Análise sintática Parsing)
- - *bottom-up*, 18, 86, 125, 126-127, 132-136, 200
- - canônica LR, 100-103, 110
- - de contexto limitado, 118-119
- - de descendência recursiva, 20, 37, 81, 322, 324
- - de empilhar e reduzir, 88-89, 91
- - de precedência de operadores, 90-94, 118, 322
- - dirigida pela tabela, 83, 85-86, 95-96
- - LALR, 95, 103-105, 119
- - - *ver também* Yacc
- - *lookahead*, 95, 103-106, 109, 119
- - - *ver* Análise gramatical LALR
- - LR simples, *ver* Análise gramatical SLR
- - preditiva, 20-21, 81-84, 85-86, 94, 129
- - recursivo-descendente, *ver* Análise gramatical de descendência recursiva
- - SLR, 95, 97-100, 110, 119
- - *top-down*, 18-21, 79, 81-86, 130, 145, 200 (*ver* Análise gramatical preditiva)
- hierárquica, 3
- - *ver também* Análise gramatical, sintática
- léxica, 3, 6, 12, 25-26, 31, 38-71, 72-73, 77, 112, 114, 323
- linear, 2
- semântica, 2, 4
- sintática, *ver* Análise gramatical
- - *bottom-up*, *ver* Análise gramatical *bottom-up*
- - de contexto limitado, *ver* Análise gramatical de contexto limitado
- - de empilhar e reduzir, *ver* Análise gramatical de empilhar e reduzir
- - de precedência de operadores, *ver* Análise gramatical de precedência de operadores
- - LALR, *ver* Análise gramatical LALR
- - *Lookahead* LR, *ver* Análise gramatical LALR
- - LR, *ver* Análise gramatical LR
- T1-T2, 292, 294-297
Anderson, J.P., 253
Anderson, T., 119
Aninhamento, de ativações, 168 (*ver também* Estrutura em blocos)
Anklam, P., 313
APL, 2, 165, 177, 303
Apontador, 150, 176, 202, 234, 240, 252, 283-285
Arden, B.W., 199
Área de dados, 192, 196
Argumento fictício, *ver* Parâmetro formal
Armazenamento, *ver* Memória
- por linha, 208
Array, 148, 149, 184
Árvore(s), 17, 136-137
- de alcance em profundidade, 289
- de ativação, 168
- de dominadores, 262
- gramatical, 3, 13-14, 18-19, 22, 73, 76-77, 87, 120, 127 (*ver* Árvore sintática)
- - anotada, 15, 120
- sintática, 1, 4, 22, 123-124, 200-201, 204
- - abstrata, 22
- - - *ver também* Árvore sintática
- - concreta, 22
ASCII, 26
Assinatura, de um nó de um GDA, 125
Associatividade, 14, 15, 43-44, 91, 107-108, 113, 299
- à direita, 14, 91, 113
- à esquerda, 14, 91, 113
Ativação, 167
Atribuição de registradores, 7-8, 224, 232-234, 236
Atributo, 5, 15, 112, 120
- herdado, 16, 120, 121, 129, 132-136, 139, 145
- sintetizado, 15, 121, 128, 135, 139
- *ver também* Atributo herdado, Valor léxico, Definição dirigida pela sintaxe
Auslander, M.A., 253, 313
Autômato
- finito, 51-52
- - determinístico, 51, 52-54, 57-58, 60-61, 64-66, 67, 81, 95,97-99
- - não-determinístico, 51, 53-57, 58-60
- - *ver também* Diagrama de transições
Avaliação contígua, 247
AWK, 38, 70, 316

B

Backhouse, R.C., 119, 315
Backus, J.W., 1, 37, 70, 165
Bail, W.G., 314
Baker, B.S., 314
Banning, J., 315
Barron, D.W., 37
Barth, J.M., 315
Bauer, A.M., 165
Bauer, F.L., 37, 145, 165
Beatty, J.C, 253
Begriffsschrift, 199
Belady, L.A., 253
Bell, J.R., 313
Bentley, J.L., 154, 199, 255
Biblioteca, 2, 24
Birman, A., 118
Bit de relocação, 9
Bliss, 235, 243, 253, 264, 313, 324-325
Bloco
- básico, 229-231, 257, 261-262, 265, 307
- comum, 186, 192-193, 196
- *ver* Bloco básico, Estrutura de blocos, Blocos comuns
BNF, 12, 37, 72, 118
Bochmann, G.V., 145
Bootstrapping, 317-318
Boyer, R.S, 71
Branquart, P., 146, 221
Branstad, M.A., 315
Bratman, H., 317
Brooks, F.P., 317
Brosgol, B.M., 145
Bruno, J.L., 246, 253
Bucket, 187
- *ver também Hashing*
Buffer, 25, 28, 40-42, 58
Burstall, R.M., 164
Busam, V.A., 313
Busca
- em profundidade, 288-290
- em um grafo, 54 (*ver também* Busca em profundidade)
Byte, 171

C

C, 23, 47-48, 140, 153, 156, 170, 177, 178, 183, 190, 204, 220, 235, 243, 256
Cabeça, 263
Cabeçalho, 263, 266, 290
- do laço, *ver* Cabeçalho
Cadeia, 42, 75
- de definição-uso, *ver* Cadeia-du
- de Fibonacci, 68
- du, 276, 280
- *Hollerith*, 45
- literal, 39
- ud, 270, 280
- uso-definição, *ver* Cadeia-ud
- vazia, 13, 21, 42, 43, 44
Cálculo lambda, 165
Caminhamento, 17, 136-137
- em profundidade, 16-17, 139-140, 168
Campo, de um registro, 207, 211
Caractere de escape, 49
Cardelli, L., 165
Carregador, 9
Carter, J.P., 319
Carter, L.R., 253
Cartwright, R., 165
Cattell, R.G.G., 253
Cauda, 263
CDC 6600, 253
Chaitin, G.J., 237, 253
Chamada
- de procedimento, 89, 170, 171, 174-177, 202, 219-220, 226-227, 240, 283
- por endereço, *ver* Chamada por referência
- por localização, *ver* Chamada por referência
- por nome, 184-185
- por referência, 184
- por valor, 183, 185
- *ver* Chamada de procedimento
Chang, C.H., 119
Cherniavsky, J.C., 314, 315
Cherry, L.L., 4, 70, 109, 321
Choe, K.M., 119
Chomsky, N., 37
Chow, F., 253, 314, 315
Church, A., 199
Ciclo, 79
- em grafos de tipo, 167
Ciesinger, J., 119
Classe de caractere, 42, 44, 67
- *ver também* Alfabeto
Classificação topológica, 122, 239
Cleveland, W.S., 199
Coação, *ver* Coerção
Cocke, J., 72, 118, 253, 313-315
Codificação de tipos, 151-152
Código
- absoluto de máquina, 3, 9, 223
- - *ver também* Máquina de pilha
- de condição, 234
- de máquina, 3, 9, 241, 246
- - relocável, 3, 9, 223
- de montagem, 2, 7, 8-9, 223, 224-225
- de três endereços, 7, 201
- em curto-circuito, 212
- inatingível, *ver* Código morto
- intermediário, 7, 200-221, 222, 256, 308
- - *ver* Máquina abstrata, Expressão posfixa, Quádrupla, Código de três endereços
- morto, 259
- objeto, 308
- redundante, 240
Coerção, 154, 165
Coffman, E.G., 253
Cohen, R., 146
Coleção
- canônica de conjuntos de itens, 97, 98, 100-101
- de conjunto itens LALR, 103
Coleta de lixo, 190
Comando
- de atribuição, *ver* Enunciado de atribuição
- de cópia, *ver* Enunciado de cópia
Comentário, 38
Common Lisp, 199
Compactação, de memória, 192
Compilador
- cruzado, 317
- de compiladores, 10
- de silício, 2
- de uma passagem, *ver* Tradução em uma passagem
- otimizante, *ver* Otimização de código
- Portable C, *ver* PCC
Composição, 299
Compressão, *ver* codificação de tipos
- de tabela, 65-66, 68, 106-107
Computação laço-invariante, *ver* Movimentação de código
Comutatividade, 299
Concatenação, 42-44
Configuração, 95
Conflito, *ver* Regra de inambigüidade, Conflito reduzir-reduzir, empilhar-reduzir
- empilhar-reduzir, 89, 94, 103, 112, 251
- reduzir-reduzir, 89, 103, 112, 251
Conjunto
- não regular, 81 (*ver também* Conjunto regular)
- regular, 45
- vazio, 42
Constante manifesta, 49
Construção(ões)
- de estados "preguiçosos", 58, 70
- de subconjuntos, 53-54, 61
Construtor de tipo, 153-154, 210-211, *ver também* Coerção
Contadores de referência, 192
Contagem de utilização, 235-236, 253
Conversão
- explícita de tipo, 153
- implícita de tipos, 154
Conway, R.W., 74, 119
Corasick, M.J., 70
Cormack, G.V., 70, 165
Corpo de procedimento, 167
Correção
- de erros de distância mínima, 40
- global de erros, 74
Courcelle, B., 146
Cousot, 314
Cousot, R., 314
CPL, 165
Curry, H.B., 165
Cutler, D., 313

D

Dado(s)
- compactados, 171
- de tamanho variável, 175, 176, 178
Davidson, J.W., 221, 253
Declaração, 115, 204-207, 220
Decoração, *ver* Árvore gramatical anotada
Definição, 265, 275
- ambígua, 265
- circular dirigida pela sintaxe, 123, 142-143, 145, 146
- de inambígua, 265
- de procedimento, 167
- dirigida pela sintaxe, 15, 120-123 (*ver* Árvore gramatical anotada, Tradução)
- - fortemente não-circular, 142-143
- incidente, 265-270, 272-273, 285, 294-297, 299
- L-atribuída, 120, 127-137, 145
- regular, 44, 49
- S-atribuída, 121, 126-127
DELTA, 146
Demers, A.J., 119
Dencker, P., 170
Depuração, 240
- simbólica, 307-311
Depurador, 174-175
Deransart, P., 146
DeRemer, F., 119, 165
Derivação, 14, 75-76
- canônica, 76
- mais à direita, 76, 87
- mais à esquerda, 76
Descritor
- de endereço, 232
- de registradores, 233
Desempilhar, 29
Deslocamento, 170, 194, 204, 227
Despeyroux, T., 166
Desvio, de um conjunto de itens, 97, 98-101, 104
Diagnóstico, *ver* Mensagem de erro
Diagrama
- de transições, 45-47, 52, 82-83, 98-99 (*ver também* Autômato finito)
- - determinístico, 45
- - não-determinístico, 82
- T, 317
Display, 182
Dispositivo
- de fluxo de dados, 11
- de tradução dirigida pela sintaxe, 11 (*ver também* GAG, HLP, LINGUIST, MUG, NEATS)
Distância
- de edição, 69
- entre cadeias de caracteres, 69
Distributividade, 314
Ditzel, D., 253
Divisão de nós, 291, 296-297
Dominador, 262, 279, 293, 314
- imediato, 262
Downey, P.J., 166, 253
Druseikis, F.C., 73
Dump simbólico, 232
Durre, K., 70

E

Earley, J., 81, 118, 119
EBCDIC, 26
Editor
- de estruturas, 2
- de ligações, 9, 19
- de texto, 70
Efeito colateral, 120
Eficiência, 38, 40, 57-58, 65-66, 68, 104-105, 120, 154, 165, 187, 188, 194, 223
- *ver também* Otimização de código
Egrep, 70
EL 1, 165
Elemento de topo, 299
Elo
- de acesso, 171, 179-182
- de controle, 171, 174-176, 182
Else vazio, 78, 85, 89, 108-109, 112, 115
Elshoff, J.L., 253
Empilhar, 29
- um caractere de entrada, 88, 95
Enchimento, 171
Endereçamento
- indexado, 225, 234
- indireto, 225
Endereço
- de retorno, 175, 226-228
- relativo, *ver* Deslocamento
Engelfriet, J., 146
Englund, D.E., 313
Entrada, para um laço, 231
Enunciado, 12, 13, 15, 29, 151
- *break*, 270-271
- *case*, 215-216
- DATA, em Fortran, 172
- de atribuição, 29, 202, 207-211
- de cópia, 202, 239, 258
- de desvio, 219
- DO, 51, 39
- EQUIVALENCE, em Fortran, 186, 193-196
- *if*, 51, 212-213, 218
- Save, 173
- *switch*, *ver* Enunciado *case*
- *while*, 212, 218
Epílogo-recursividade, 23
EQN, 4-5, 109-110, 129, 316, 317, 321
Equação(ões)
- de fluxo de dados, 271, 297
- - para adiante, 271-272, 305-306
- - para trás, 271-272, 305-307
Equel, 8
Equivalência
- de autômatos finitos, 166
- de blocos básicos, 229
- de definições dirigidas pela sintaxe, 129-131
- de expressões
- - de tipo, 151-153 (*ver também* Unificação)
- - regulares, 43, 67-68
- de gramáticas, 166
- estrutural, de tipos de expressões, 151-152, 160, 163
- por nomes, de expressões de tipo, 152
- sob uma substituição, 159, 161-162
Erro
- léxico, 40, 73
- lógico, 73
- semântico, 73, 149
- sintático, 73-74, 85-86, 88, 91, 92-94, 95, 109-111, 114, 118
Ershov, A.P., 145, 253, 313

Escopo, 169, 177, 188-190, 198, 205-207
- dinâmico, 177, 182-183
- estático, *ver* Escopo léxico
- léxico, 177-182
Espaço em branco, 25, 38, 45
Esqueleto de árvore gramatical, 91
Esquema de tradução, 17-18, 128-129
- de árvores, 248
Estado, 45, 52, 69, 95, 126
- da memória, 169
- de aceitação, 52
- de máquina, 171, 175
- de partida, 45
- final, *ver* Estado de aceitação
- importante, 60
Estimativa de tipo, 303-307
Estocagem máxima, 251
Estrutura
- de análise de fluxo de dados, 297-303
- de blocos, 177, 188-190
- distributiva, 300-301, 302
- monótona, 300, 302
- *ver* registro de ativação
Eve, J., 278
Expansão em linha, 184
- *ver também* Macro
Expressão(ões), 3, 15, 75, 125-126, 150
- anulável, 61, 62-63
- booleana, 140, 211-215, 217-218
- condicional, *ver* Expressão booleana
- de modos mistos, 214-215
- de tipo, 148-149
- disponíveis, 273-275, 288, 299, 303
- infixa, 15
- muito ocupada, 311
- posfixa, 12, 15, 200, 201, 202, 220
- prefixa, 220
- regular, 38, 43-44, 49, 51, 55, 58, 61-64, 67, 77, 115

F

Falha no escopo de uma declaração, 178
Família, de uma variável de indução, 281
Fang, I., 146
Farrow, R., 146
Fase, 5 (*ver* Geração de código, Otimização de código, Tratamento de erros)
Fatoração à esquerda, 79-80
Fechamento, 42-44, 55-56
- de congruência, *ver* Nós congruentes
- de conjunto de itens, 97, 98, 100-101
- E, 53-54, 98
Feldman, S.I., 70, 221, 313
Ferramentas, 316 (*ver* Gerador automático de código, Compilador de compiladores)
Feys, R., 165
Fgrep, 70
Fila, 220
Filho, 14
Fischer, C.N., 119, 253
Fischer, M.J., 71, 199
Fleck, A.C., 184
Floyd, R.W., 118, 253
Fluxo
- de controle, 29, 202-203, 211-218, 241, 263, 266, 270, 301, 314
- do controle, *ver* Fluxo de controle
FNC, *ver* Forma normal de Chomsky
FOLDS, 146
Folha, 14
- à direita, 243
- à esquerda, 243
Fong, A.C., 315
Forma
- de Backus-Naur, *ver* BNF
- normal
- - de Chomsky, 118
- - de Greibach, 116
- setencial, 76
- - à direita, 76, 87
- - à esquerda, 76
Formatador de texto, 2, 4-5
Fortran, 1, 39, 51, 70, 92, 165, 170, 172, 184, 186, 192-196, 208, 262, 313, 316
- H, 235, 253, 318, 322-323
Fosdick, L.D., 315
Foster, J.M., 37, 119
Fragmentação, 191
Fraser, C.W., 221, 253
Fredman, M., 70
Frege, G., 199
Freiburghouse, R.A., 221, 253
Freudenberger, S.M., 313, 315
Função(ões)
- de falha, 68, 69
- de *hash*, 187-188
- de precedência, 92
- de transferência, 294, 297, 301
- de transição, 52, 69
- genérica, 156
- - *ver também* Função polimórfica
- identidade, 299
- polimórfica, 147, 156-160
- *ver* Procedimento

G

Gabarito de dados, 171, 204-205
GAG, 146
Gajewska, H., 315
Galler, B.A., 199
Ganapathi, M., 253
Gannon, J.D., 73
Ganzinger, H., 146, 165
Garey, M.R., 253
GDA, *ver* Grafo direcionado acíclico
Gear, C.W., 314
Geração
- de código, 7-8, 222-253, 322
- de uma cadeia de caracteres, 14
- espontânea, de *lookaheads*, 105
Geradas, 264, 266-267, 273, 277
Gerador
- automático de código, 11
- de analisadores sintáticos, 11, 319 (*ver também* Yacc)
- de *scanners*, 11 (*ver também* Lex)
Geschke, C.M., 235-236, 314
Giegerich, R., 146, 221, 253
Glanville, R.S., 251, 253
GLC, *ver* Gramática livre de contexto
GNF, *ver* Forma normal de Greibach
Grafo
- acíclico, *ver* Grafo direcionado acrílico
- colorimento de, 236-237
- de dependências, 120, 122, 141-142
- de dependências aumentado, 143
- de fluxo, 229, 230, 237, 257, 262
- - limite, 291, 292
- - não-redutível, 264, 297
- - redutível, 263-264, 290, 291, 292, 312, 324
- de interferência de registradores, 236
- de intervalos, 291
- de tipo, 149, 151, 153
- de transições, 52
- direcionado acíclico, 125-126, 148, 200-201, 204, 237-240, 242-243, 252, 253
Graham, R.M., 118, 199
Graham, S.L., 119, 253, 314
Gramática
- afixa, 145
- aumentada, 97
- de atributos, 121, 252
- de operadores, 90
- de precedência de operadores, 116
- E-livre, 116
- LALR, 103
- livre
- - de ciclos, 116
- - de contexto, 12-14, 18, 36-37, 74-77, 120
- - - *ver também* Gramática LL, LR e de operadores
- LL, 72, 73, 85, 97, 116, 117, 119, 132
- LR(1), 102
- LR, 94-107, 145, 251
- SLR, 99
Grau, A.A., 221
Grep, 70

H

Haley, C.B., 119
Halstead, M.H., 221, 318
Handle, 87-88, 91, 93, 98
Hanson, D.R., 221
Harrison, M.A., 119
Harrison, M.C., 71
Harrison, W.H., 253
Harry, E., 146
Hashing, 125-126, 187-190, 198
Hashpjw, 188
Heap, 170, 322
Heinen, R., 313
Helsinki Language Processor, *ver* HLP
Henderson, P.B., 314
Hennessy, J.L., 253, 315
Henry, R.R., 253
Hecht, M.S., 313, 315
Heuft, J. 70
Hext, J.B., 165
Hill, U., 221
Hindley, R., 165
HLP, 146
Hoare, C.A.R., 37, 165
Hoffman, C.M., 253
Hopcroft, J.E., 64, 70, 118, 166, 192, 199, 253
Hope, 164
Hopkins, M.E., 253
Horning, J.J., 37, 73, 118
Horspool, R. N.S.,70
Horwitz, L.P., 253
Huet, G., 253
Huffman, D.A., 70
Hunt, J.W., 71
Huskey, H.D., 318, 221
Hutt, B., 146

I

IBM-370, 224, 247, 253, 323, 324
IBM-7090, 253
Ichbiah, J.D., 118
Idempotência, 44
Identificação de operador, 154 (*ver também* Sobrecarga)
Identificador, 25, 39, 80
Indireção, 204
Inferência de tipo, 156, 159-160, 303
Ingalls, D.H.H., 165
Ingerman, P.Z., 37
Instância, de um tipo polimórfico, 158
Interface de retaguarda, 10, 78
Interpretador, 2
Irons, E.T., 37, 119, 145
Ishihata, K., 315
Item
- de núcleo, 97, 105
- LR(0), 97
- LR(1), 100
- válido, 98, 100
Iverson, K., 165

J

Janas, J.M., 165
Jarvis, J.F., 38, 70
Jazayeri, M., 146
Jensen, K., 326
Johnson, D., 253
Johnson, R.K., 253
Johnson, S. C., 2, 70, 111, 119, 145, 152, 199, 221, 246, 248, 253, 319, 322
Johnson, W.L., 70
Joliat, M., 119
Jones, N.D., 145, 165, 313, 314
Jourdan, M., 146
Joy, W.N, 119

K

Kaiserwerth, M., 70
Kam, J.B., 314
Kaplan, M.A., 165, 315
Karp, R.M., 166
Kasami, T., 72, 118, 314
Kastens, U., 146
Kasyanov, V.N., 314
Katayama, T., 146
Kennedy, K., 146, 253, 314
Keohane, J., 314
Kernighan, B.W., 4, 11, 37, 70, 109, 199, 316, 319, 321, 328
Kieburtz, R.B., 119
Kildall, G.A., 277, 297, 314
Kleene, S.C., 70
Knuth, D.E., 4, 11, 37, 70, 119, 145, 166, 192, 199, 253, 294, 315, 320
Kolsky, H.G., 313, 323
Komlos, J., 70

Korenjak, A.J., 119
Kosaraju, S. R., 314
Koskimies, K., 145
Koster, C.H.A., 145
Kou, L., 314
Kristensen, B.B., 119
Kron, H., 253
Kruskal, J.B., 71

L

Laço, 231, 236, 262-264, 268-269, 288
- mais interno, 231, 263
- natural, 163
Lado(s)
- adiante, 264
- cruzado, 290
- progressivo, 290
- refluente, 263, 264, 290
- retraentes, 290
LaLonde, W.R., 119
Lamb, D.A., 253
Lampson, B.W., 199
Landin, P.J., 199
Langmaack, H., 221
Lassagne, T., 253
Latim estilizado, 36
Lecarme, O., 318
Ledgard, H.F., 165
Leinius, R.P., 119
Lengauer, T., 315
Lesk, M.E., 70, 319
Leverett, B.W., 221, 253
Levy, J.J., 253
Levy, J.P., 119
Lewis, H.R., 314
Lewis, P.M., 118, 145
Lex, 38, 48-51, 58, 67, 70, 319
Lexema, 6, 25, 27, 39, 185-186
Líder, 229
Ligação de cópia e restauração, 184
Linguagem, 13, 42, 52, 75, 76, 90
- alvo, 1
- de programação, *ver* Ada, Algol, APL, BCPL, Bliss, C, Cobol, CPL, EL1, Fortran
- fonte, 1
- fortemente tipada, 149
- livre de contexto, 76, 77
- objeto, *ver* Linguagem alvo
LINGUIST, 146
Lint, 149
Lisp, 177, 190, 198, 303, 317
Lista
- de adjacências, 52
- ligada, 186-187, 189
Localizar, 162
Lookahead, 51, 60, 94, 100
Lorho, B., 146
Low, J., 315
Lowry, E.S., 235, 253, 313, 314, 318, 322
Lucas, P., 37
Lunde, A. 253
Lunnel, H. 253

M

MacLaren, M.D., 313
MacQueen, D.B., 164, 165
Macro, 8, 38, 196
Madsen, C.M., 145
Madsen, O.L., 119, 145
Make, comando do UNIX, 319
Manipulação de erros, *ver* Tratamento de erros
Mapa de memória, 192
Máquina
- abstrata, 28 (*ver também* Máquina de pilha)
- alvo, 316
- de pilha, 28-31, 200, 253
Marill, T., 146, 253
Markstein, J., 313, 315
Martelli, A., 166
Matar, 265, 260, 273, 277
Mauney, J., 119
Maxwell, W.L., 119
Mayoh, B.H., 145
McArthur, R., 221, 318
McCarthy, J., 37, 198, 317
McClure, R.M., 118
McCracken, N.J., 155
McCulloch, W.S., 70
McIlroy, M., 71
McKeeman, W.M., 37, 118, 199, 253
McLellan, H.R., 253
McNaughton, R., 170
Medlock, C.W., 235, 253, 313, 314, 318, 322
Meertens, L., 165
Memória, 169
Mensagem de erro, 86, 93-94, 110-111
META, 118
Metcalf, M., 313
Meyers, R., 165
Miller, R.E., 314
Milner, R., 156, 165
Minimização de estados, 64-65
Minker, J., 221
Minker, R.G., 221
Mitchell, J.C., 165
ML, 156, 159, 165
Modalidade pânico, 40, 74, 86, 110
Modo de endereçamento, 9, 225-226, 252
Modula, 264, 314, 325
Módulo emissor, 30, 31
Monotonicidade, 314
Montador de duas passagens, 9
Montanari, U., 166
Moore, E.F., 70, 315
Moore, J.S., 71
Morel, E., 314
Morris, D., 145
Morris, J.S., 70, 165
Morris, R., 199
Morse, S.P., 118
Moses, J., 199
Moulton, P.G., 119
Movimentação de código, 259, 279-280, 310, 314, 325
Muchnick, S.S., 165, 313
MUG, 146
Muller, M.E., 119

N

Nakata, I., 253
Naur, P., 37, 118, 165, 198
NEATS, 146
Neliac, 318
Newey, M.C., 221
Nievergelt, J., 253
Nó(s)
- compartilhado, 246
- congruentes, 164, 166
- de retorno, 285
- inicial, 230
Nome, 167
- de tipo, 148, 152
- global, 285
- - *ver também* Nome não-local
- local, 169, 171, 177
- não-local, 169, 177-183, 229
Nori, K.V., 221
Núcleo, de conjunto de itens, 103
Número de valor, 125, 277
Não-terminal, 13, 75, 90
- marcador, 133-135, 145

O

O'Donnell, M.J., 253
Objeto, 167, 169
Ogden, W.F., 146
Operador
- aritmético, 154
- de confluência, 272, 297, 303
- de reunião, 297
- unário, 92
Ordem
- de avaliação, para definições dirigidas pela sintaxe, 122-123, 128, 135-144
- parcial, 143
Ordenamento em profundidade, 127, 288, 293-294
Organização
- de memória, 170-172
- por coluna, de arrays, 208-209
Osterweil, L.J., 315
Otimização
- de código, 7, 200, 204, 208, 222, 229, 240-241, 254-315, 323
- de desvios, 324, 325
- de laço, 259 (*ver também* Movimentação de código, Variável de indução, Redução)
- global, 257, 276
- local, 257, 276
- *peephole*, 240-241, 253, 255

P

P-code, 321, 325
Pager, D., 119
Pai, A.B., 119
Paige, R., 315
Pair, C., 146
Palavra, 42
- chave, 25, 39, 185
- reservada, 25, 40
Palm, R.C., 315
Panini, 37
Par de registradores, 224, 245
Parâmetro
- atual, 167, 171
- de procedimento, 179, 180
- formal, 167
Parênteses, 43, 44, 78
Park, J.C.H., 119
Partição, 64
- de intervalo, 291
Pascal, 23, 39, 43, 44, 73, 149, 152, 156, 170, 177, 183, 184, 190, 204, 208
Passagem, 10
- de parâmetros, *ver* Transmissão de parâmetros
Paterson, M.S., 166
PCC, 224, 248, 253, 322
Pennello, T., 119, 165
Perfil de comportamento, 255
Período de uma cadeia, 68
Persch, G., 165
Pesquisa em profundidade, *ver* Busca em profundidade
Peterson, T.G., 119
Peterson, W.W., 199, 314
Peyrolle-Thomas, M.C., 318
Pic, 196
Pike, R., 37, 199, 319, 328
Pilha, 57, 83, 88, 95, 110, 118, 125-127, 133-135, 139-140, 168-170, 206-207,243
- de controle, 168-169, 170
Primeira-pos, 61-63
Pitts, W., 70
PI/C, 74, 223
PL/I, 10, 36, 40, 73, 164, 165, 211, 220, 221, 313
Plankalkul, 165
Plotkin, G., 165
Poda do handle, 87-88
Pollack, B.W., 11
Pollock, L.L., 315
Ponteiro, *ver* Apontador
Ponto, 265
Poole, P.C., 221
Pop, *ver* Desempilhar
Portabilidade, 39, 317
Pos-seguinte, 61, 62
Post, E., 37
Powell, M.L., 314, 315, 325
Pozefsky, D., 146
Pratt, T.W., 199
Pratt, V.R., 70, 118
Pré-cabeçalho, 263
Pré-processador, 2, 8
Precedência, 15, 43, 91, 107-108, 113 (*ver* Gramática de precedência de operador)
- de estratégia mista, 118
- de operadores, 14
- fraca, 118
- simples, 118
Predecessor, 230
Prefixo, 42
- viável, 89, 95, 98, 100
PRIMEIRO, 20, 83-84, 86
Procedimento, 167
- aninhados, 179-182, 205-206
Produção
- de erro, 74, 114-115
- E, 79, 84, 116
- singela, 107, 116
- unitária, *ver* Produção singela
Produto
- cartesiano, 148
- de uma árvore, 14
Profundidade

- de aninhamento, 179
- de intervalo, 294
- de um grafo de fluxo, 290, 293-294, 313
Programação dinâmica, 118, 246-248, 253
Projeto de programação, 326-328
Propagação
- de cópia, 257, 258-259, 277-278
- do caractere *lookahead*, 105
Prosser, R.T., 314
Pseudônimos, 283, 315
Purdom, P.W. 145, 315
Push, *ver* Empilhar

Q

Quádruplas, 203, 204
Quantificador universal, 157
Quicksort, 167, 255

R

Rabin, M.O., 70
Radin, G., 165
Raiha, K.J., 119, 146
Raiz, 14
Ramanathan, J. 146
Randell, B., 11, 37, 199, 221
Ratfor, 316
Reconhecedor, 51
Reconhecimento de padrões, 39, 58-60, 250, 253
Recuperação de erros a nível de frase, 74, 86, 110
Recursividade, *ver* Recursão
Recursão, 3, 74, 136, 141-142, 168
- à direita, 243
- à esquerda, 21-22, 31, 79-80, 81, 85, 129-131
- imediata à esquerda, 79
Redução, 86, 88, 93-94, 95, 110
- de capacidade, 241, 259-260, 261, 281
Redziejowski, R.R., 253
Reescrita de árvore, 248-252
Referência, *ver* Uso
- a *arrays*, 89, 201, 208-210, 239, 252, 255, 283
- externa, 9
- oca, 176, 190
Registrador simbólico, 236
Registro de ativação, 228
Região, 260, 292-293, 294-296
Regra
- de cópia, 138, 139, 184-185
- de inambigüidade, 165
- - *ver* também Sobrecarga
- de tradução, 49
- do aninhamento mais interno, 177, 179
- semântica, 15, 120-123
Reif, J.H., 314
Reinstalabilidade, 317
Reiss, S.P., 199
Relação(ões)
- de precedência de operadores, 90
Remoção, de objetos de dados locais, 173
Renomeação, de variáveis, 230
Renvoise, C., 314
Reorientabilidade, 316, 317
Reps, T.W., 145
Reserva implícita, *ver* Alocação implícita
Retenção, de variáveis locais, 172-173, 176-177
Reticulado, 165
Retrocesso, 81
Retrocorreção, 10, 216-219, 223
Reynolds, J.C., 165
Rhodes, S.P., 119
Richards, M., 221, 253
Ripken, K., 165, 253
Ripley, G.D., 73
Ritchie, D.M., 152, 199, 221, 322
Robinson, J.A., 165
Rogoway, H.P., 165
Rohl, J.S., 199
Rosen, S., 11
Rosenkrantz, D.J., 118
Rosler, L., 316
Rótulo, 29, 201, 219, 223
Rounds, W.C., 146
Rovner, P., 315
Russell, L.J., 11, 37, 199, 221
Russell, S.R., 317
Ruzzo, W.L., 119
Ryder, B.G., 315

S

Saal, H.J., 165
Saarinen, M, 146
Samelson, K., 145
Sankoff, D., 71
Sannella, D.T., 164
Scanner, 38
Scanning, *ver* Análise léxica
Scarborough, R.G., 313, 323
Schaefer, M., 313
Schonberg, E., 315
Schorre, D.V., 118
Schwartz, J.T., 165, 253, 313, 315
Scott, D., 70
Sedgewick, R., 255
SEGUINTE, 84, 86, 100
Seleção de instruções, 223
Semântica, 12
- denotacional, 145
Sentença, 42, 76
Sentinela, 42
Seqüência
- de chamada, 174-175, 219
- de retorno, 174-176
Sethi, R., 146, 165, 199, 246, 253
SETL, 165, 303, 314
Sharir, M., 314, 315
Shell, 67
Sheridan, P.B., 118, 165
Shimasaki, M., 253
Shustek, L.J., 253
Símbolo
- básico, 43, 55
- de entrada, 52
- de partida, 13, 75, 121
- inútil, 116
SIMPL, 314
Sintaxe, 12 (*ver também* Gramática livre de contexto)
Sippu, S., 119
Sistema
- de escrita de compiladores, *ver* Compilador de compiladores
- de tipos, 149, 305
- sonoro de tipos, 149
Sneeringer, W.J., 165
Snobol, 177
Sobrecarga, 141, 147, 154-156, 164, 165
Soffa, M.L., 315
Soisalon-Soininen, E., 119
Solução de convergência de percursos, 301, 302, 303
Spillman, T.C., 315
Staveren, H. van, 253
Stdio h., 26
Stearns, R.E., 118, 145
Steel, T.B., 221
Steele, G.L., 199
Stevenson, J.W., 253
Stockhausen, P.F., 253
Stonebraker, M., 8
Strong, J., 221, 317
Stroustrup, B., 188
Subcadeia, 42
Subexpressão comum, 125, 230, 237, 239, 246, 258, 261, 276-277, 310, 323, 325
- *ver também* Expressão disponível
Subseqüência, *ver também* Maior subexpressão comum
- comum mais longa, 69
Substituição, 158, 160-162
Sucessor, 230
Sufixo, 42
Suporte em tempo de execução, 167 (*ver também* Alocação de memória de pilha)
Sussman, G.J., 199
Suzuki, N., 165, 315
Szemeredi, E., 70
Szymanski, T.G., 71, 253

T

Tabela
- de ações sintáticas, 95
- de cadeias, 186
- de desvio, 95
- de *hash*, 187
- de símbolos, 4, 5, 27-28, 185-190, 203, 205-208, 307
- de transições, 52
- gramatical SLR, *ver* Tabela sintática SLR
- sintática
- - canônica LR, 100-103
- - LALR, 103-105
- - SLR, 99-100
Tai, K.C., 119
Tanenbaum, A.S., 221, 253
Tantzen, R.G., 37
Tarjan, R.E., 70, 166, 199, 314
Tempo de vida
- de um atributo, 138, 139-141
- de um temporário, 208
- de uma ativação, 168, 177
Temporário, 170-171, 202, 208, 232, 277, 279
Tennenbaum, A.M., 165, 314
Terminal, 13, 74-75, 121
Teste de regressão, 319
Teste, 319-320
TEX, 4, 8, 37, 319
Thompson, K., 55, 322
Thunk, 185
Tipo, 147-166
- apontador, 148
- básico, 148
- da função, 148-149, 151, 154-156
- - *ver também* Função polimórfica
- polimórfico, 157
- registro, 148, 153, 207
- vazio, 148, 151
- viável, 155
Tjiang, S., 253
TMG, 118
Token de sincronização, 86
Token, 2, 6, 13, 25, 39-40, 45, 75, 80
Tokuda, T., 119
Tokura, N., 314
Trabb Pardo, L., 11
Tradução
- em uma passagem, 120, 322
- simples dirigida pela sintaxe, 17-18, 128
Tradutor preditivo, 131-132
Transformação algébrica, 230, 241, 245, 261, 323
Transição-E, 52, 60
Transmissão
- de parâmetros, 179, 183-185, 285
- por valor-resultado, *ver* ligação da cópia e restauração
Transposição para constante, 257, 259, 261, 297-299, 300-301
Tratamento de Erros, 5, 40, 73-74
- *ver também* Erros léxico, lógico, semântico, sintático
Travessia pós-ordem, 243
Trickey, H.W., 2
Trie, 68, 69
Triplas, 203-204
- indiretas, 204
TROFF, comando do Unix, 317, 321

U

Ukkonen, E., 119
Ullman, J.D., 2, 64, 70, 81, 90, 118, 165, 192, 199, 246, 253, 255, 314
Última-pos, 61-63
UNCOL, 37, 221
Unificação, 158-159, 160-163, 165
Unificador mais geral, 158, 160
UNIX, 67, 71, 111, 317, 322
União, 43-44, 55, 162
Uso(s), 231, 276
- expostos acima, 276

V

Valor
- de retorno, 171, 175
- default, 215
- L, 29, 99, 169, 183-185
- léxico, 6, 50-51, 121
- r, 29, 99, 169, 183-185, 237
Van Staveren, *ver* Staveren, H. van
Vanguarda, do compilador, 9, 28
Variável, *ver* Identificador, Variável de tipo
- básica de indução, 281
- de indução, 259, 281-283, 310, 315, 323
- modificada, 287-288
- tipo, 156
- viva, 231, 235, 239, 259, 261, 275, 280, 285
Verificação
- de tipos, 4, 147, 149, 222
- de unicidade, 147
- dinâmica, 147, 149

- do fluxo de controle, 147
- estática, 2, 147, 149, 315
- relacionada pelo nome, 147
Vyssotsky, V., 314

W

Wagner, R.A., 71
Waite, W.M., 221, 253, 314, 319
Walter, K.G., 146
Ward, P., 145
Warren, S.K., 146
Wasilew, S.G., 253
WATFIV, 223
Watt, D.A., 145
WEB, 320
Weber, H., 118
Wegbreit, B., 165, 314
Wegman, M.N., 166, 314
Wegner, P., 314
Wegstein, J.H., 70
Weihl, W.E., 315
Weinberger, P.J., 70, 188
Weingart, S., 253
Welsh, J., 165
Wexelblat, R.L., 11, 37
Wilcox, T.R., 74
Wilhelm, R., 146, 221
Wirth, N., 37, 119, 199, 221, 318, 321, 326
Wood, D., 119
Wortman, D.B., 165
Wossner, H., 165
Wulf, W.A., 212, 253, 313, 324

Y

Yacc, 111-115, 319, 325
Yamada, H., 70
Yannakakis, M., 253
Yao, A.C., 70
Yellin, D., 146
Younger, D.H., 72, 118

Z

Zelkowitz, M.V., 314
Zimmermann, E., 146
Zuse, K., 165

41,00
16 11 99
Em
Computação
Vilma Pedro Vetro

Serviços de impressão e acabamento
executados, a partir de filmes fornecidos,
nas oficinas gráficas da EDITORA SANTUÁRIO
Fone: (12) 565-2140 - fax: (12) 565-2141
http://www.redemptor.com.br - Aparecida - SP