

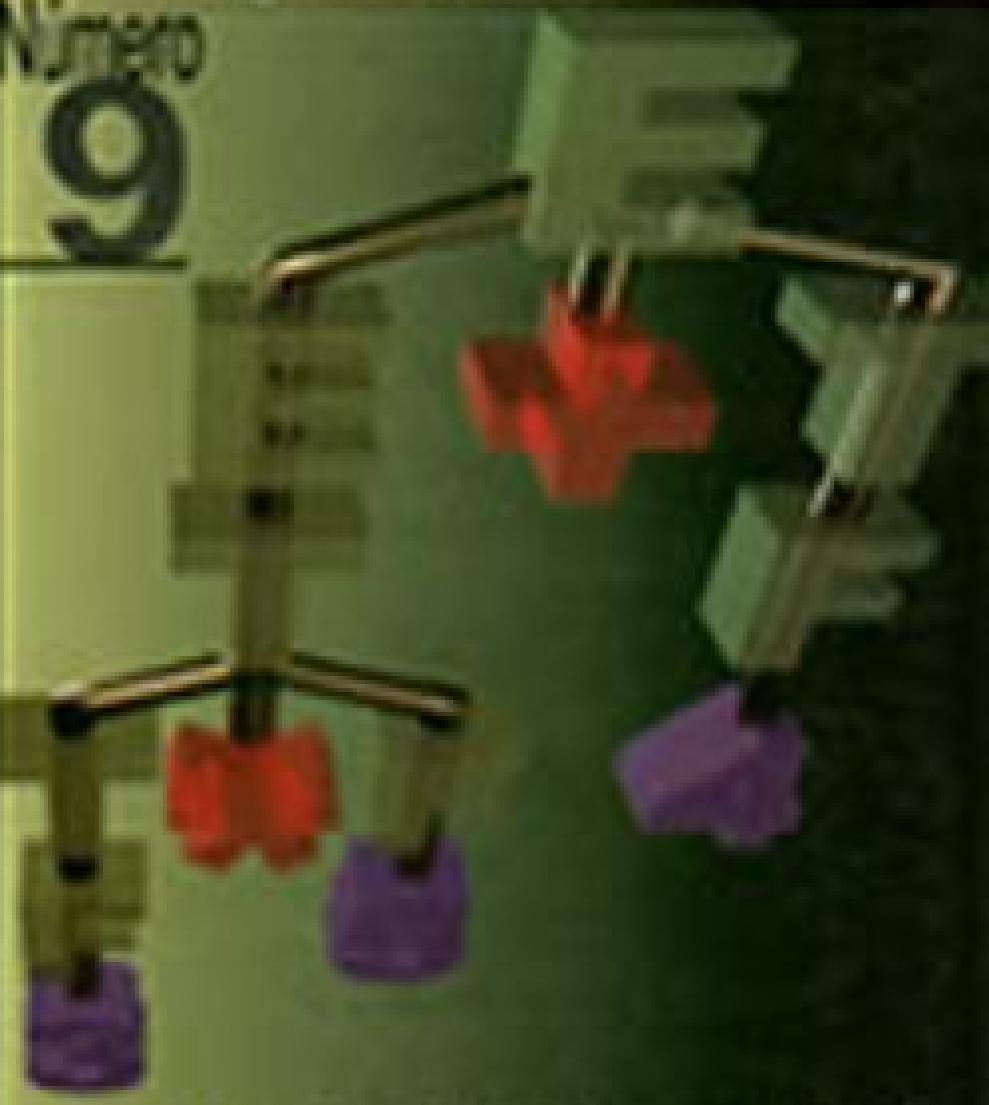
Implementação de Linguagens de Programação: Compiladores

Série
Livros Didáticos

VOLUME
9



Instituto de Informática
da UFRGS



Ana Maria de Alencar Price
Simão Sirineo Toscani



Diretor
Prof. Philippe Olivier Alexandre Navaux

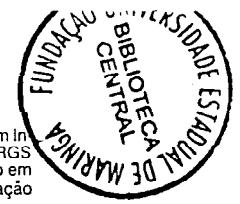
Vice-Diretor
Prof. Ótacílio Jose Carollo de Souza

Comissão Editorial
Prof. Tiarajú Asmuz Diverio
Prof. Clesio Saraiva dos Santos
Prof. Ricardo Augusto da Luz Reis
Profª Carla Maria Dal Sasso Freitas

Endereço
UFRGS – Instituto de Informática
Av Bento Gonçalves, 9500 Bloco IV Bairro Agronomia
Caixa Postal 15064 91501-970 Porto Alegre, RS
Fone 00 55 (051) 3316 6165 Fax 00 55 (051) 3319 7308
e-mail: informatica@ufrgs.br
<http://www.inf.ufrgs.br>

Ana Maria de Alencar Price

Doutora em Computer Science pela University of Sussex, UK (1985) | Mestre em Informática pela PUCIRJ (1976) | Graduada em Engenharia Química pela UFRGS (1972) | Professora da UFRGS desde 1975, atuando no curso de Bacharelado em Ciência da Computação, em Cursos de Especialização em Sistemas de Informação e nos programas de mestrado e doutorado do Instituto de Informática



Simão Sirineo Toscani

Doutor em Informática pela Universidade Nova de Lisboa, Portugal (1993) | Mestre em Informática pela PUCIRJ (1969) | Engenheiro Eletricista pela UFRGS (1967) | Professor do Departamento de Informática da PUC/RJ (1969 - 1974) e do Instituto de Informática da UFRGS (1975 - 1998) | Atualmente é coordenador e professor do Curso de Ciência da Computação da Universidade de Cruz Alta (UNICRUZ) e professor orientador do Programa de Pós-Graduação em Computação da UFRGS

implementação de Linguagens de Programação: Compiladores

Série Livros Didáticos • Número 9
Instituto de Informática da UFRGS



Prefácio da Segunda Edição

Agradecemos à turma da UNISC, em especial, à Professora Alessandra Dahmer pelas modificações sugeridas. Somos gratos, também, ao Professor Giovani Librelotto, da UNICRÚZ, por sua colaboração nessa segunda edição.

Ana Price e Simão Toscani
Porto Alegre, Julho de 2001

Sumário

| | |
|--|-----------|
| 1 Tradução de Linguagens de Programação | 1 |
| 1.1 Evolução das Linguagens de Programação | 1 |
| 1.2 Tradutores de Linguagens de Programação | 4 |
| 1.3 Estrutura de um Tradutor | 7 |
| 1.3.1 Análise Léxica | 7 |
| 1.3.2 Análise Sintática e Semântica | 9 |
| 1.3.3 Geração de Código Intermediário | 11 |
| 1.3.4 Ottimização de Código | 12 |
| 1.3.5 Geração de Código Objeto | 13 |
| 1.3.6 Gerência de Tabelas | 13 |
| 1.3.7 Atendimento a Erros | 14 |
| 1.4 Geradores de Compiladores | 14 |
| Exercícios | 16 |
| 2 Análise Léxica | 17 |
| 2.1 Gramáticas e Linguagens Regulares | 18 |
| 2.2 Tokens | 22 |
| 2.3 Especificação | 24 |
| 2.4 Implementação | 25 |
| 2.5 Tabela de Símbolos | 26 |
| Exercícios | 28 |
| 3 Análise Sintática..... | 29 |
| 3.1 Revisão de Gramáticas Livres-do-Contexto | 30 |
| 3.1.1 Definições e exemplos | 30 |
| 3.1.2 Transformações de GLC's | 34 |
| 3.2 Análise Descendente (Top-down)..... | 38 |
| 3.2.1 Análise Recursiva com Retorno | 38 |
| 3.2.2 Análise Recursiva Preditiva | 41 |
| 3.2.3 Análise Preditiva Tabular | 45 |
| 3.3 Análise Redutiva (Bottom-up) | 53 |
| 3.3.1 Analisadores de Precedência de Operadores | 55 |
| 3.3.2 Funções de Precedência | 64 |
| 3.3.3 Analisadores LR | 66 |
| 3.4 Recuperação de Erros | 74 |
| 3.4.1 Recuperação de Erros na Análise LL | 75 |
| 3.4.2 Recuperação de Erros na Análise de Precedência de Operadores | 77 |
| 3.4.3 Recuperação de Erros na Análise LR | 79 |
| Exercícios | 80 |

| | |
|--|------|
| 4 Tradução Dirigida por Sintaxe | 85 |
| 4.1 Esquemas de Tradução | 86 |
| 4.1.1 Estratégia depth-first | 88 |
| 4.1.2 Atributos sintetizados e atributos herdados | 89 |
| 4.1.3 Gramática de atributos | 89 |
| 4.1.4 Tipos de Esquemas de Tradução | 92 |
| 4.2 Grafos de dependências | 94 |
| 4.3 Árvore de Sintaxe | 96 |
| 4.4 Implementação de Esquemas S-Atribuídos | 98 |
| 4.5 Esquemas de Tradução L-Atribuídos | 101 |
| 4.6 Implementação de Esquemas L-atribuídos | 104 |
| 4.7 Projeto de um Tradutor Preditivo..... | 110 |
| Exercícios | 112 |
| 5 Geração de Código Intermediário | i 15 |
| 5.1 Linguagens Intermediárias..... | 115 |
| 5.1.1 Árvore e Grafo de Sintaxe | 115 |
| 5.1.2 Notações Pós-fixada e Pré-fixada | 116 |
| 5.1.3 Código de Três-Endereços | 117 |
| 5.2 Ações Semânticas para a Construção de Tabelas de Símbolos..... | 120 |
| 5.3 Geração de Código para Comando de Atribuição | 123 |
| 5.3.1 Conversão de Tipos em Expressões Aritméticas | 124 |
| 5.3.2 Endereçamento de Elementos de Matrizes | 125 |
| 5.4 Expressões Lógicas e Comandos de Controle | 129 |
| 5.4.1 Representação Numérica de Expressões Lógicas | 129 |
| 5.4.2 Representação por Fluxode Controle | 132 |
| 5.5 Backpatching | 136 |
| 5.5.1 Backpatching em Expressões Lógicas | 137 |
| 5.5.2 Backpatching em Comandos de Controle | 138 |
| Exercícios | 141 |
| 6 Otimização de Código..... | 145 |
| 6.1 Otimização de Código Intermediário | 146 |
| 6.1.1 Representação de Blocos Básicos Através de Grafos | 146 |
| 6.1.2 Algoritmo para Construir o GAD de um Bloco | 147 |
| 6.1.3 Algoritmo para Gerar uma Sequência Otimizada de Código..... | 149 |
| 6.2 Otimização de Código para Expressões Aritméticas..... | 150 |
| 6.2.1 Algoritmo para Obter o Número de Acumuladores | 153 |
| 6.2.2 Geração de Código para Máquina com N Acumuladores | 158 |
| Exercícios | 168 |

| | |
|--|-----|
| 7 Gerência de Memória | 169 |
| 7.1 Estratégias para Alocação de Memória..... | |
| 7.2 Alocação em Memória de Pilha | 169 |
| 7.3 Acesso à Variáveis Não-locais | 171 |
| 7.4 Passagem de Parâmetros | |
| 7.5 Alocação Dinâmica de Memória | 183 |
| Exercícios | 185 |
| 8 Geração de Código Objeto | 187 |
| 8.1 Considerações no Projeto de um Gerador de Código | 187 |
| 8.2 A Máquina Objeto | 189 |
| 8.3 Gerador de Código Simplificado | 190 |
| 8.3.1 Informação de Próximo-Uso | 191 |
| 8.3.2 Descritores de Registradores e de Endereços | 192 |
| 8.3.3 Algoritmo de Geração de Código | 192 |
| Exercícios | 195 |
| Referências Bibliográficas | 197 |

1 Tradução de Linguagens de Programação

O meio mais eficaz de comunicação entre pessoas é a *linguagem* (língua ou idioma). Na programação de computadores, uma *linguagem de programação* serve como meio de comunicação entre o indivíduo que deseja resolver um determinado problema e o **computador escolhido** para ajudá-lo na solução. A linguagem de programação deve fazer a **ligação** entre o pensamento humano (muitas vezes, de natureza não estruturada) e a precisão requerida para o processamento pela máquina.

O desenvolvimento de um programa torna-se mais fácil se a linguagem de programação em uso estiver próxima ao problema a ser resolvido. Isto é, se a linguagem incluir construções que refletem a terminologia e/ou os elementos usados na descrição do problema. Tal tipo de linguagem de programação é considerada como uma linguagem de **alto nível**. Os computadores digitais, por sua vez, aceitam e entendem somente sua própria linguagem de máquina (dita de baixo nível), a qual consiste tipicamente de sequências de zeros e uns. Esse tipo de linguagem é bem diferente da linguagem de alto nível usada para descrever um problema numa dada área de aplicação.

As linguagens de programação mais utilizadas hoje são aquelas classificadas como de alto nível, consideradas mais próximas às linguagens naturais ou ao domínio da aplicação em questão (linguagens procedimentais e linguagens de 4ª geração). Para que se tomem operacionais, os programas escritos em linguagens de alto nível devem ser traduzidos para *linguagem de máquina*. Essa conversão é realizada através de sistemas especializados – *compiladores* ou *interpretadores* – que aceitam (como entrada) uma representação textual da solução de um problema, expresso em uma *linguagem fonte*, e produzem uma representação do mesmo algoritmo expresso em outra linguagem, dita *linguagem objeto*.

1.1 Evolução das Linguagens de Programação

Cronologicamente, as linguagens de programação podem ser classificadas em cinco gerações: (1^a) linguagens de máquina, (2^a) linguagens simbólicas (Assembly), (3^a) linguagens orientadas ao usuário, (4^a) linguagens orientadas à aplicação e (5^a) linguagens de conhecimento. As linguagens de 1^a e 2^a gerações são consideradas linguagens de baixo nível; as demais são classificadas como linguagens de alto nível.

Os primeiros computadores eram programados em linguagem de máquina, em notação binária. Cada instrução de máquina é, em geral, formada por um código de operação e um ou dois endereços de registradores ou de memória. As linguagens de máquina permitem a comunicação direta com o computador em termos de "bits", registradores e operações de máquina bastante primitivas. Como um programa em linguagem de máquina nada mais é que uma sequência de zeros e uns, a programação de um algoritmo complexo usando esse tipo de linguagem é complexa, cansativa e fortemente sujeita a erros.

A 2ª geração compreende as linguagens simbólicas ou de montagem (Assembly), projetadas para minimizar as dificuldades da programação em notação binária. Códigos de operação e endereços binários foram substituídos por mnemônicos. Nas linguagens de montagem, a maioria das instruções são representações simbólicas de instruções de máquina. O processamento de um programa em linguagem simbólica requer tradução para linguagem de máquina antes de ele ser executado.

As linguagens de 3ª geração surgiram na década de 60. Algumas delas são orientadas à solução de problemas científicos, tais como FORTRAN, PASCAL e ALGOL; outras, tal como COBOL, são usadas para aplicações comerciais. Linguagens como PL/I e ADA contêm facilidades para ambos os tipos de computações (científica e comercial).

As linguagens de 3ª geração podem também ser classificadas em *linguagens procedimentais* (também chamadas "procedurais" ou imperativas) e *linguagens declarativas*. Nas linguagens procedimentais, um programa especifica um procedimento, isto é, uma sequência de passos a serem seguidos para solucionar um problema. O programa descreve, de forma direta, "como" será resolvido o problema em questão. As instruções oferecidas por essas linguagens pertencem, em geral, a três classes: instruções de *entrada/saída*, instruções de cálculos aritméticos ou lógicos, e instruções de controle do fluxo de execução (desvios condicionais, incondicionais e processamento iterativo). Como exemplos de linguagens procedimentais, tem-se BASIC, ALGOL, PL/I, PASCAL, ADA, C, etc.

As linguagens declarativas dividem-se, basicamente, em duas classes: *funcionais*, as quais se baseiam na teoria das funções recursivas, e *lógicas*, cuja base é a lógica matemática. A programação funcional envolve, essencialmente, a definição e a chamada de funções. LISP é o exemplo mais difundido de linguagem funcional. Nas linguagens lógicas, um programa declara *fatos* (dados e relações entre eles) e *cláusulas lógicas* (regras de dedução), que permitirão deduzir novas verdades a partir dos fatos conhecidos. O programador "define" regras de dedução sem detalhar os passos a serem seguidos para chegar a uma conclusão.

ambiente de execução provê um "sistema dedutor" que aceita questões do usuário e usa as regras de dedução para responder a essas questões. PROLOG constitui o primeiro exemplo de linguagem de programação lógica. Embora seja uma linguagem declarativa, cronologicamente PROLOG é considerada de 5ª geração.

As linguagens de 3ª geração foram projetadas para profissionais de processamento de dados e não para usuários finais. A depuração de programas escritos nessas linguagens consome tempo, e a modificação de sistemas complexos é relativamente difícil. As linguagens de 4ª geração foram projetadas em resposta a esses problemas.

Os principais objetivos das linguagens de 4ª geração são: (1) facilitar a programação de computadores de maneira tal que usuários finais possam resolver seus problemas; (2) apressar o processo de desenvolvimento de aplicações; (3) facilitar e reduzir o custo de manutenção de aplicações; (4) minimizar problemas de depuração; e (5) gerar código sem erros a partir de requisitos de expressões de alto nível.

Os programas escritos em linguagens de 3ª geração necessitam de menor número de linhas de código do que os programas correspondentes codificados em linguagens de programação convencionais. Adicionalmente ao emprego de comandos sequenciais, tal como nas linguagens de 3ª geração, as de 4ª geração empregam diversos outros mecanismos, como por exemplo, preenchimento de formulários, interação via vídeo (menus), e auxílio para a construção de gráficos. Muitas linguagens de 4ª geração são dependentes de um banco de dados e de um dicionário associado. Em alguns casos, o dicionário de dados evoluiu para conter formatos de telas e de relatórios, estruturas de diálogos, mecanismos de verificação, autorizações para ler e modificar dados, relações lógicas entre valores de dados, etc. As linguagens de 4ª geração variam bastante no número de facilidades oferecidas ao usuário. Algumas são, meramente, geradores de relatórios ou pacotes gráficos; outras são capazes de gerar aplicações completas. Em geral, essas linguagens são projetadas para atender a classes específicas de aplicações.

Usualmente, as linguagens de 4ª geração permitem a combinação de ambos, comandos procedimentais e comandos não-procedimentais, num mesmo segmento de código. Essa característica é desejável porque operações não-procedimentais aceleram e simplificam o uso da linguagem, enquanto o código procedural estende o espaço de aplicações, dando maior flexibilidade de manipulação lógica. Exemplos de linguagens de 4ª geração são: LOTUS 1-2-3, EXCEL, SQL, SUPERCALC, VISICALC, DATATRLEVE, FRAMEWORK.

As linguagens de 5^a geração são usadas principalmente na área de Inteligência Artificial. Tais linguagens facilitam a representação do conhecimento que é essencial para a simulação de comportamentos inteligentes.

1.2 Tradutores de Linguagens de Programação

Tradutor, no contexto de linguagens de programação, é um sistema que aceita como entrada um programa escrito em uma linguagem de programação (*linguagem fonte*) e produz como resultado um programa equivalente em outra linguagem (*linguagem objeto*).

Os tradutores de linguagens de programação podem ser classificados em:

- *Montadores (assemblers)* – são aqueles tradutores que mapeiam instruções em *linguagem simbólica (assembly)* para instruções de linguagem de máquina, geralmente, numa relação de uma–para–uma, isto é, uma instrução de linguagem simbólica para uma instrução de máquina.
- *Macro-assemblers* - são tradutores que mapeiam instruções em linguagem simbólica para linguagem de máquina, geralmente, numa relação de uma–para–várias. Muitas linguagens simbólicas possuem facilidades de definição de "macros" que são, na realidade, facilidades de expansão de texto em linguagem mnemônica. Um comando macro é traduzido para uma sequência de comandos simbólicos antes de ser procedida a tradução efetiva para linguagem de máquina.
- *Compiladores* - são tradutores que mapeiam programas escritos em linguagem de alto nível para programas equivalentes em linguagem simbólica ou linguagem de máquina. A execução de um programa escrito em linguagem de alto nível é, basicamente, um processo de dois passos, conforme é mostrado na figura 1.1.

O intervalo de tempo no qual ocorre a conversão de um programa fonte para um programa objeto é chamado de *tempo de compilação*. O programa objeto é executado no intervalo de tempo chamado *tempo de execução*. Observa-se que o programa fonte e os dados são processados em momentos distintos, respectivamente, tempo de compilação e tempo de execução.

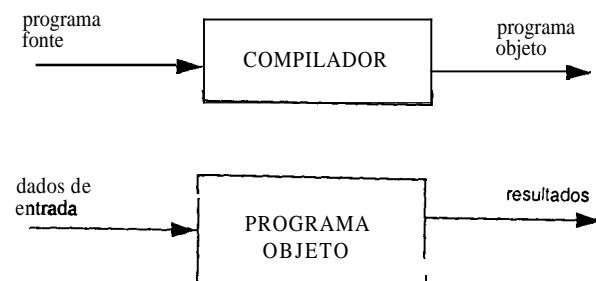


Figura 1.1 Execução de um programa fonte

- *Pré-compiladores, Pré-processadores ou Filtros* - são aqueles processadores que mapeiam instruções escritas numa linguagem de alto nível estendida para instruções da linguagem de programação original, ou seja, são tradutores que efetuam conversões entre duas linguagens de alto nível.



Figura 1.2 Pré-compilador ou filtro

Os pré-compiladores surgiram para facilitar a extensão de linguagens de alto nível existentes. As extensões são usadas, na maioria das vezes, para atender aplicações específicas, cujo objetivo é aprimorar o projeto e a escrita de algoritmos. Por exemplo, existem pré-processadores FORTRAN, BASIC e COBOL estruturados que mapeiam programas em versões estruturadas dessas linguagens para programas em FORTRAN, BASIC e COBOL padrões.

- *Interpretadores* - são processadores que aceitam como entrada o código intermediário de um programa anteriormente traduzido e produzem o "efeito de execução" do algoritmo original sem, porém, mapeá-lo para linguagem de máquina. Os interpretadores processam uma forma intermediária do programa fonte e dados ao mesmo tempo. Isto é, a interpretação da forma interna do fonte ocorre em tempo de execução, não sendo gerado um programa objeto. A figura que segue ilustra o processo de interpretação.



Figura 1.3. Interpretação de código

Alguns interpretadores não utilizam um código intermediário, trabalhando diretamente sobre o programa fonte, analisando um comando fonte cada vez que esse deve ser executado. Tal enfoque consome muito tempo e é, raramente, utilizado. Uma estratégia mais eficiente envolve a aplicação de técnicas de compilação para traduzir o programa fonte para uma forma intermediária, que é, então, interpretada. Neste texto, o termo interpretador referir-se-á não só ao programa que simula a execução do código intermediário, mas a todo o processo interpretativo, desde a tradução do código fonte comando-a-comando até a sua execução.

Os interpretadores são, geralmente, menores que os compiladores e facilitam a implementação de constmções complexas de linguagens de programação. A principal desvantagem dos interpretadores é que o tempo de execução de um programa interpretado é maior que o tempo necessário para executar um programa objeto (compilado) equivalente. Isso porque a "execução" do código intermediário tem embutido o custo do processamento de uma tradução virtual para código de máquina cada vez que uma instrução em código intermediário deve ser operada. Pode-se pensar no programa em código **intermediário** como o código de máquina de um computador virtual (hipotético).

O tradutor **BASIC**, residente em muitos microcomputadores, é um sistema interpretativo que, praticamente, não utiliza código intermediário, pois "compila" um comando **BASIC** a cada vez, utilizando o enfoque traduz-exectua.

Os interpretadores são usualmente mais amigáveis (*user-friendly*) do que os compiladores, pois estão bem mais próximos do código fonte que programas já completamente traduzidos. As mensagens de erro e as facilidades de teste e depuração em relação à interpretação são mais claras e objetivas e, principalmente, certeiras ao referirem-se ao código fonte.

Uma grande vantagem dos sistemas interpretativos está na implementação de novas linguagens para diferentes equipamentos de computação. A utilização do sistema interpretativo permite uma implementação relativamente fácil, uma vez que é produzido um

código intermediário padrão, independente de máquina, para o qual é programado um interpretador para cada máquina diferente. Esse método foi utilizado para a linguagem Pascal, o que permitiu a rápida aceitação dessa linguagem em todo o mundo.

Tradutores que geram código para máquinas hospedeiras, isto é, máquinas nas quais eles próprios executam, são denominados auto-residentes (*self-resident translators*). Aqueles que geram código objeto para outras máquinas, que não as hospedeiras, são denominados compiladores cruzados (*cross-translators*).

1.3 Estrutura de um Tradutor

Em geral, os tradutores de linguagens de programação (compiladores, interpretadores) são programas bastante complexos. Porém, devido à experiência acumulada ao longo dos anos e, principalmente, ao desenvolvimento de teorias relacionadas às tarefas de análise e síntese de programas, existe um consenso sobre a estrutura básica desses processadores.

Independentemente da linguagem a ser traduzida ou do programa objeto a ser gerado, os tradutores, de um modo geral, compõem-se de funções padronizadas, que compreendem a análise do programa fonte e a posterior síntese para a derivação do código objeto.

O processo de tradução é, comumente, estmturado em fases, conforme a Figura 1.4, no qual cada fase se comunica com a seguinte através de uma linguagem intermediária adequada. Na prática (ponto de vista de implementação), seguidamente, a distinção entre as fases não é muito clara. Isto é, as funções básicas para a tradução podem não estar individualizadas em módulos específicos e podem se apresentar distribuídas em módulos distintos.

1.3.1 Análise Léxica

O objetivo principal desta fase é identificar sequências de caracteres que constituem unidades léxicas ("tokens"). O analisador léxico lê, caractere a caractere, o texto fonte, verificando se os caracteres lidos pertencem ao alfabeto da linguagem, identificando **tokens**, e desprezando comentários e brancos desnecessários. Os **tokens** constituem classes de símbolos tais como palavras reservadas, delimitadores, identificadores, etc., e podem ser representados internamente, através do próprio símbolo (como no caso dos delimitadores e das palavras reservadas) ou por um par ordenado, no qual o primeiro elemento indica a classe do símbolo, e o segundo, um índice para uma área onde o próprio símbolo foi armazenado (por exemplo, um identificador e sua entrada numa tabela de identificadores). Além da identificação de

tokens, o analisador léxico, em geral, inicia a construção da Tabela de Símbolos (Seção 2.4) e envia mensagens de erro caso identifique unidades léxicas não aceitas pela linguagem em questão.

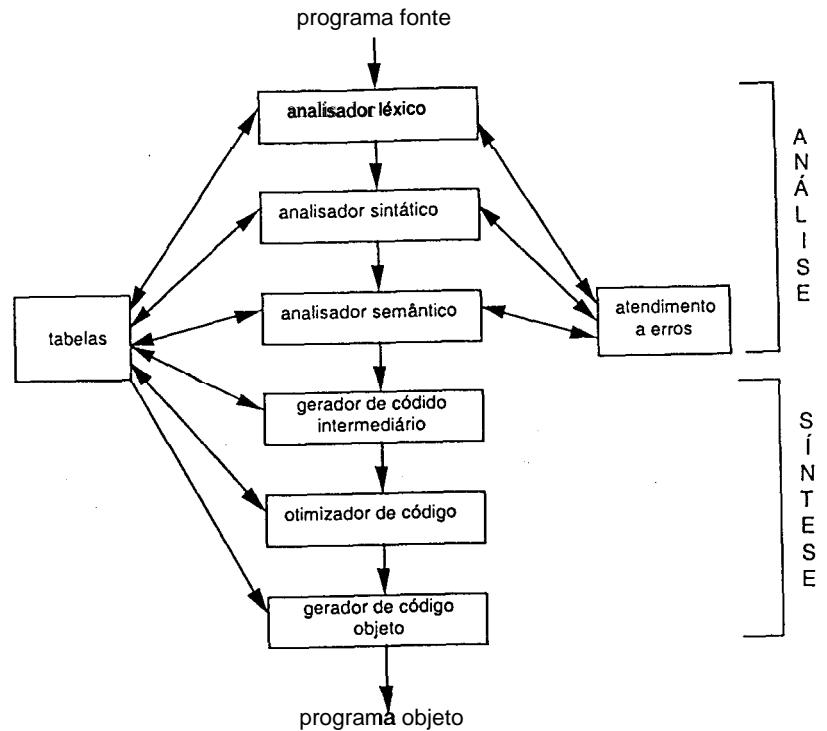


Figura 1.4. Estrutura de um compilador

A saída do analisador Léxico é uma cadeia de *tokens* que é passada para a próxima fase, a Análise Sintática. Em geral, o Analisador Léxico é implementado como uma subrotina que funciona sob o comando do Analisador Sintático.

EXEMPLO 1.1 Saída do analisador léxico.

Seja o seguinte texto fonte em Pascal:

while I < 100 do I := J + I ;

Após a Análise Léxica da instrução acima, ter-se-ia a Seguinte cadeia de *tokens*:

[while, I [id, 7] [<, I [cte, 13] [do,] [id, 7] [=,] [id, 12] [+ ,] [id, 7] [; ,] na qual, palavras reservadas, operadores e delimitadores são representados pelos próprios símbolos, e identificadores de variáveis e constantes numéricas são representados por um par [classe do símbolo, índice de tabela].

1.3.2 Analises Sintática e Semântica

A fase de análise sintática tem por função verificar se a estrutura gramatical do programa está correta (isto é, se essa estrutura foi formada usando as regras gramaticais da linguagem). A análise semântica tem por função verificar se as estruturas do programa irão fazer sentido durante a execução.

O Analisador Sintático identifica sequências de símbolos que constituem estruturas sintáticas (por exemplo, expressões, comandos), através de uma varredura ou "parsing" da representação interna (cadeia de *tokens*) do programa fonte. O Analisador Sintático produz (explícita ou implicitamente) uma estrutura em árvore, chamada *árvore de derivação*, que exibe a estrutura sintática do texto fonte, resultante da aplicação das regras gramaticais da linguagem. Em geral, a árvore de derivação não é produzida explicitamente, mas sua construção está implícita nas chamadas das rotinas recursivas que executam a análise sintática. Em muitos compiladores, a representação interna do programa resultante da análise sintática não é a árvore de derivação completa do texto fonte, mas uma árvore compactada (árvore de sintaxe) que visa a eliminar redundâncias e elementos supérfluos. Essa estrutura objetiva facilitar a geração do código que é a fase seguinte à análise.

Outra função dos reconhecedores sintáticos é a detecção de erros de sintaxe identificando clara e objetivamente a posição e o tipo de erro ocorrido. Mesmo que erros tenham sido encontrados, o Analisador Sintático deve tentar recuperá-los prosseguindo a análise do texto restante.

Muitas vezes, o Analisador Sintático opera conjuntamente com o Analisador Semântico, cuja principal atividade é determinar se as estruturas sintáticas analisadas fazem sentido, ou seja, verificar se um identificador declarado como variável é usado como tal; se existe compatibilidade entre operandos e operadores em expressões; etc. Por exemplo, em Pascal, o comando while tem a seguinte sintaxe:

while <expressão> do <comando>;

a estrutura <expressão> deve apresentar-se sintaticamente correta, e sua avaliação deve retomar um valor do tipo lógico. Isto é, a aplicação de operadores (relacionais/lógicos) sobre os **operandos** (constantes/variáveis) deve resultar num valor do tipo lógico (verdadeiro/falso).

As regras gramaticais que definem as construções da linguagem podem ser descritas através de produções (regras que produzem, geram) cujos elementos incluem *símbolos terminais* (aqueles que fazem parte do código fonte) e símbolos *não-terminais* (aqueles que geram outras regras). No exemplo que segue, as produções são apresentadas na *Forma Normal de Backus* (Backus Naur Form ou BNF).

EXEMPLO 1.2 Produções BNF.

Este exemplo mostra produções que geram comandos de atribuição e comandos iterativos. Os terminais aparecem em negrito e os símbolos não-terminais aparecem delimitados por “<” e “>”. Os comandos while e de atribuição podem ser definidos (parcialmente) pelas seguintes produções:

$$\begin{aligned}
 <\text{comando}> &\rightarrow <\text{while}> \mid <\text{atrib}> \mid \dots \\
 <\text{while}> &\rightarrow \text{while } <\text{expr_bool}> \text{ do } <\text{comando}> \\
 <\text{atrib}> &\rightarrow <\text{variável}> := <\text{expr_arit}> \\
 <\text{expr_bool}> &\rightarrow <\text{expr_arit}> < <\text{expr_arit}> \\
 <\text{expr_arit}> &\rightarrow <\text{expr_arit}> + <\text{termo}> \mid <\text{termo}> \\
 <\text{termo}> &\rightarrow <\text{número}> \mid <\text{variável}> \\
 <\text{variável}> &\rightarrow \text{I} \mid \text{J} \\
 <\text{número}> &\rightarrow 100
 \end{aligned}$$

EXEMPLO 1.3 Árvore de derivação.

Considerando o comando while do exemplo 1.1, o Analisador Sintático produziria a árvore de derivação mostrada na Figura 1.5 (a partir da sequência de tokens liberada pelo Analisador Léxico).

R

As fases até aqui descritas constituem módulos que executam tarefas analíticas. As fases seguintes trabalham para construir o código objeto: geração de código intermediário, otimização e geração de código objeto.

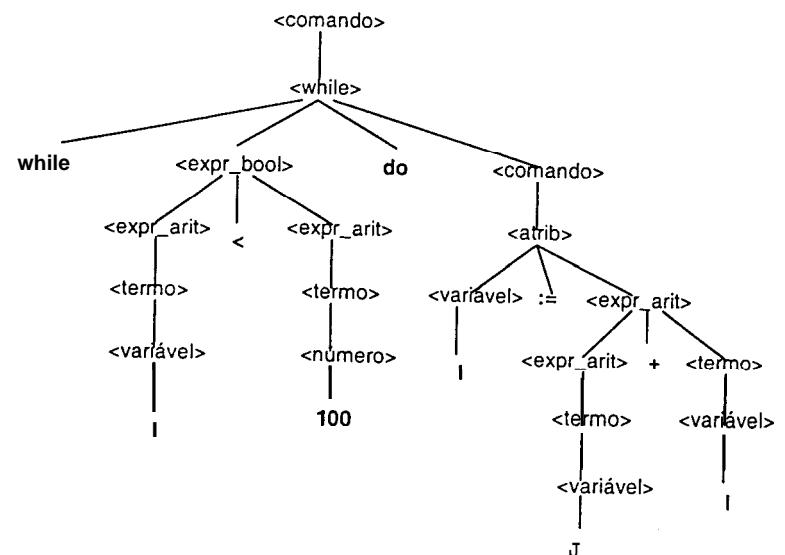


Figura 1.5 Árvore de derivação

1.3.3 Geração de Código Intermediário

Esta fase utiliza a representação interna produzida pelo Analisador Sintático e gera como saída uma sequência de código. Esse código pode, eventualmente, ser o código objeto final mas, na maioria das vezes, constitui-se num código intermediário, pois a tradução de código fonte para objeto em mais de um passo apresenta algumas vantagens:

- possibilita a otimização de código intermediário, de modo a obter-se o código objeto final mais eficiente;
- resolve, gradualmente, as dificuldades da passagem de código fonte para código objeto (alto nível para baixo nível), já que o código fonte pode ser visto como um texto condensado que "explode" em inúmeras instruções elementares de baixo nível.

A geração de código intermediário pode estar estruturalmente distribuída nas fases anteriores (análise sintática e semântica) ou mesmo não existir (tradução direta para código objeto), no caso de tradutores bem simples.

A grande diferença entre o código intermediário e o código objeto final é que o intermediário não especifica detalhes tais como quais registradores serão usados, quais endereços de memória serão referenciados, etc.

EXEMPLO 1.4 Código intermediário.

Para o comando while apresentado anteriormente, o gerador de código intermediário, recebendo a árvore de derivação mostrada na figura 1.5, poderia produzir a seguinte sequência de instruções:

```

L0  if I < 100 goto L1
      goto L2
L1  TEMP := J + I
      I := TEMP
      goto L0
L2  ...

```

□

Há vários tipos de código intermediário: quádruplas, triplas, notação polonesa pós-fixada, etc. A linguagem intermediária do exemplo acima é chamada "código de três endereços", pois cada instrução tem no máximo três operandos.

1.3.4 Otimização de Código

Esta fase tem por objetivo otimizar o código intermediário em termos de velocidade de execução e espaço de memória.

EXEMPLO 1.5 Código otimizado.

Considerando o código intermediário do exemplo anterior, o seguinte código otimizado poderia ser obtido:

```

L0  if I ≥ 100 goto L2
      I := J + I
      goto L0
L2  ...

```

1.3.5 Geração de Código Objeto

Esta fase tem como objetivos: produção de código objeto, reserva de memória para constantes e variáveis, seleção de registradores, etc. É a fase mais difícil, pois requer uma seleção cuidadosa das instruções e dos registradores da máquina alvo a fim de produzir código objeto eficiente. Existem tradutores que possuem mais uma fase para realizar a otimização do código objeto, isto é, otimização do código dependente de máquina.

EXEMPLO 1.6 Código de máquina.

A partir do código intermediário otimizado mostrado no exemplo anterior, obter-se-ia o código objeto final abaixo, código este baseado na linguagem simbólica de um microcomputador PC **8086**.

```

L0  MOV AX, I
    CMP AX, 100
    JGE L2
    MOV AX, J
    MOV BX, I
    ADD BX
    MOV I, AX
    JMP L0
L2  ...

```

1.3.6 Gerência de Tabelas

Este módulo não constitui uma fase no sentido das anteriores, mas compreende um conjunto de tabelas e rotinas associadas que são utilizadas por quase todas as fases do tradutor.

Algumas das tabelas usadas são fixas para cada linguagem, por exemplo, a tabela de palavras reservadas, tabelas de delimitadores, etc. Entretanto, a estrutura que possui importância fundamental é aquela que é montada durante a análise do programa fonte, com informações sobre:

- declarações de variáveis;
- declarações dos procedimentos ou subrotinas;
- parâmetros de subrotinas; etc.

Essas informações são armazenadas na Tabela de Símbolos (às vezes chamada de tabela de nomes ou lista de identificadores). A cada ocorrência de um identificador no programa fonte, a tabela é acessada, e o identificador é procurado na tabela. Quando encontrado, as informações associadas a ele são comparadas com as informações obtidas no programa fonte, sendo que qualquer nova informação é inserida na tabela.

Os dados a serem coletados e armazenados na tabela de símbolos dependem da linguagem, do projeto do tradutor, do programa objeto a ser gerado. Entretanto, os atributos mais comumente registrados são:

- para variáveis: classe (var), tipo, endereço no texto, precisão, tamanho;
- parâmetros formais: classe (par), tipo, mecanismo de passagem;
- procedimentos/subrotinas: classe (proc), número de parâmetros;

A Tabela de Símbolos deve ser estruturada de uma forma tal que permita rápida inserção e extração de informações, porém deve ser tão compacta quanto possível.

1.3.7 Atendimento a Erros

Este módulo tem por objetivo "tratar os erros" que são detectados em todas as fases de análise do programa fonte. Qualquer fase analítica deve prosseguir em sua análise, ainda que erros tenham sido detectados. Isso pode ser realizado através de mecanismos de recuperação de erros, encarregados de re-sincronizar a fase com o ponto do texto em análise. A perda desse sincronismo faria a análise prosseguir de forma errada, propagando o efeito do erro.

É fundamental que o tradutor prossiga na tradução. **após** a detecção de erros, de modo que o texto seja totalmente analisado.'

1.4 Geradores de Compiladores

Atualmente, a implementação de linguagens de programação é apoiada por sistemas **geradores de compiladores** (*compiler-compilers*, *compiler generators*, *translator-writing systems*). Esses sistemas classificam-se em três grupos:

- a) geradores de analisadores léxicos - geram automaticamente reconhecedores para os símbolos léxicos (palavras-chave, identificadores, operadores, etc.) a partir de especificações de gramáticas ou expressões regulares.

- b) geradores de analisadores sintáticos - produzem reconhecedores sintáticos a partir de gramáticas livres do contexto. Inicialmente, a implementação da análise sintática consumia grande esforço na construção de compiladores. Hoje, essa fase é considerada uma das mais fáceis de implementar.
- c) geradores de geradores de código - recebem como entrada regras que definem a tradução de cada operação da linguagem intermediária para a linguagem de máquina. As regras devem incluir detalhes suficientes para possibilitar a manipulação de diferentes métodos de acesso a dados (por exemplo, uma variável pode estar em registradores, em memória ou na pilha da máquina). Em geral, instruções intermediárias são mapeadas para esqueletos que representam sequências de instruções de máquina.

EXERCÍCIOS

- 1) No contexto de implementação de linguagens de programação, dê o significado dos seguintes termos: compilador, interpretador, montador, pré-compilador.
- 2) Aponte as vantagens e desvantagens dos interpretadores em relação aos compiladores.
- 3) Explique o processo de compilação: fases e seu inter-relacionamento.
- 4) Qual o significado de "passo" no processo de compilação? Quais as vantagens e desvantagens de implementar um compilador em vários passos?

2 Análise Léxica

A análise léxica é a primeira fase do compilador. A função do analisador léxico, também denominado *scanner*, é:

Fazer a leitura do programa fonte, caractere a caractere, e traduzi-lo para uma sequência de *símbolos* léxicos, também chamados *tokens*.

Exemplos de símbolos léxicos são as palavras reservadas, os identificadores, as constantes e os operadores da linguagem. Durante o processo de análise léxica, são desprezados caracteres não significativos como espaços em branco e comentários. Além de reconhecer os símbolos léxicos, o analisador também realiza outras funções, como armazenar alguns desses símbolos (tipicamente identificadores e constantes) em tabelas internas e indicar a ocorrência de erros léxicos. A sequência de *tokens* produzida (reconhecida) pelo analisador léxico é utilizada como entrada pelo módulo seguinte do compilador, o analisador sintático. É interessante observar que o mesmo *programa* fonte é visto pelos analisadores léxico e sintático como sentenças de linguagens diferentes. Para o analisador léxico, o *programa* fonte é uma sequência de palavras de uma *linguagem* regular. Para o analisador sintático, essa sequência de tokens constitui uma sentença de uma *linguagem livre do contexto*. Os analisadores sintáticos, a serem estudados no próximo capítulo, trabalham com gramáticas livres do contexto, as quais permitem expressar construções mais sofisticadas (sentenças formadas por expressões e blocos aninhados, por exemplo).

Uma *linguagem* é um conjunto de palavras formadas por símbolos de um determinado alfabeto. Os símbolos léxicos (ou *tokens*) de uma linguagem de programação constituem uma *linguagem* regular. Estas linguagens são as mais simples, segundo a classificação proposta por Chomsky¹, dentro da Teoria das Linguagens Formais. No contexto da tradução de linguagens de programação, as linguagens são usualmente apresentadas através de gramáticas ou de algoritmos (autômatos) que as reconhecem.

¹ Segundo a classificação de Chomsky, em ordem crescente de complexidade e de generalidade, os tipos de linguagens são: linguagens regulares (tipo 3), linguagens livres do contexto (tipo 2), linguagens sensíveis ao contexto (tipo 1) e linguagens recursivamente enumeráveis (tipo 0).

2.1 Gramáticas e Linguagens Regulares

Esta seção apresenta uma breve revisão dos conceitos relacionados a gramáticas em geral e a gramáticas regulares. Um estudo mais aprofundado sobre esse assunto pode ser encontrado no livro *Linguagens Formais e Autômatos*, de P. B. Menezes.

Definição 2.1 Gramática.

Uma **gramática** G é um mecanismo para gerar as sentenças (ou palavras) de uma linguagem e é definida pela quádrupla:

$$(S, T, P, S)$$

onde

N é um conjunto de símbolos não-terminais (variáveis)

T é um conjunto de símbolos terminais (constantes), e $T \cap N = \emptyset$

P é um conjunto de regras de produção (regras sintáticas)

S é o símbolo inicial da gramática ($S \in N$).

As regras de produção são representadas por $a \rightarrow \beta$ e definem o mecanismo de geração das sentenças da linguagem. Uma sequência de regras da forma $a \rightarrow \beta_1, a \rightarrow \beta_2, \dots, a \rightarrow \beta_n$ (com o mesmo lado esquerdo) pode ser representada de forma simplificada como segue:

$$a \rightarrow \beta_1 | \beta_2 | \dots | \beta_n$$

A aplicação sucessiva de regras de produção, a partir do símbolo inicial da gramática, permite **derivar** as sentenças válidas da linguagem representada pela gramática.

Definição 2.2 Derivação.

Uma **derivação** é um par da relação denotada por \Rightarrow , com domínio em $(N \cup T)^*$ e contradomínio em $(N \cup T)^*$, e é representada de forma infixada como $a \xrightarrow{*} \beta$. O conjunto $(N \cup T)^*$ é o conjunto de sentenças que podem ser formadas por símbolos de N e T , incluindo a **palavra vazia** ϵ (palavra com comprimento zero), e $(N \cup T)^* = (N \cup T)^* - \{ \epsilon \}$.

As sequências de símbolos a e β são chamadas **formas sentenciais**. A relação \Rightarrow é indutivamente definida como segue:

- para produções da forma $S \rightarrow \beta$ (sendo S o símbolo inicial de G) tem-se que

$$S \Rightarrow \beta$$
- para todo par $a \Rightarrow \beta$, onde $\beta = \beta_u \beta_v \beta_w$, se $\beta_v \rightarrow \beta_t$ é uma regra de P , então

$$\beta \Rightarrow \beta_u \beta_t \beta_w$$

Portanto, uma derivação é a substituição de uma sequência de símbolos numa forma sentencial de acordo com uma regra de produção.

Derivações sucessivas são definidas como segue:

- \Rightarrow^+ fecho transitivo da relação \Rightarrow , ou seja, uma ou mais derivações sucessivas;
- \Rightarrow^* fecho transitivo e reflexivo da relação \Rightarrow , ou seja, zero ou mais derivações sucessivas.

Definição 2.3 Linguagem Gerada.

A **linguagem gerada** por G , denotada por $L(G)$, é o conjunto formado por todas as sentenças de símbolos terminais deriváveis a partir do símbolo inicial S , ou seja

$$L(G) = \{ s \mid s \text{ é um elemento de } T^* \text{ e } S \xrightarrow{*} s \}$$

Convenções:

As seguintes convenções são usadas para representar os símbolos **gramaticais** (terminais e não-terminais) e as formas sentenciais (sequências de símbolos terminais e não-terminais).

1- Símbolos que representam terminais:

- letras minúsculas do inicio do alfabeto tais como: a, b, c, ... (as letras finais v, w, x, ... são usadas para representar cadeias de terminais)
- operadores em geral, tais como: +, -, *, :=, ...
- símbolos que representam pontuação, a saber, vírgula, ponto+–vírgula, ponto e os delimitadores tais como: (,), {, }, [].
- dígitos: 0, 1, 2, 3, 4, ...
- cadeias de caracteres em **negrito**, como if, then, begin, ...

2- Símbolos que representam não-terminais:

- letras maiúsculas do alfabeto: A, B, C, D, ... (em geral, S representa o símbolo inicial)
- qualquer string delimitado por "<" e ">"

3- Símbolos que representam formas sentenciais (sequências de terminais e não-terminais):

- as letras minúsculas do alfabeto grego, como $\alpha, \beta, \gamma, \delta, \dots$

Definição 2.4 Gramática Regular.

Uma gramática que tenha produções exclusivamente da forma $A \rightarrow wB$ ou $A \rightarrow tw$, onde $w \in T^*$ e $A, B \in N$, é classificada como uma *gramática regular*.

Uma gramática tal como definida acima é dita *gramática linear à direita*. As gramáticas lineares à esquerda, cujas produções são da forma $A \rightarrow Bw$ ou $A \rightarrow w$, também são regulares. Para qualquer gramática regular, se $|w| \leq 1$, então a gramática é dita *unitária*.

EXEMPLO 2.1 Gramática regular.

A gramática abaixo gera identificadores que iniciam por letra (I), podendo esta ser seguida por qualquer número de letras e/ou dígitos (d). G é uma gramática linear à direita unitária

$$\begin{aligned} G &= (N, T, P, I) \\ N &= \{ I, R \} \\ T &= \{ I, d \} \\ P &= \{ I \rightarrow I | I R, R \rightarrow I R | d R | I | d \} \end{aligned}$$

As *linguagens regulares* podem ser reconhecidas por máquinas de transição de estados chamadas *autômatos finitos*. Fornecendo-se uma sentença a um autômato, este é capaz de responder se a mesma pertence ou não à linguagem que ele representa.

Os autômatos compõem-se basicamente de três partes:

- a fita de entrada, a qual contém a sentença a ser analisada;
- uma unidade de controle, que reflete o estado corrente e realiza os movimentos da máquina (leitura e mudança de estado);
- a função de transição que comanda a leitura de símbolos da fita de entrada e define o estado da máquina.

Definição 2.5 Autômato Finito.

Um *autômato finito* M sobre um alfabeto Σ é uma 5-upla $(K, L, 6, e_0, F)$, onde

K é um conjunto finito de estados

Σ é o alfabeto dos símbolos da linguagem

$6 : K \times \Sigma \rightarrow K$ é a função de transição de estados

F é o conjunto de estados finais.

6 é uma função parcial, pois não precisa estar definida para todos os pares $K \times \Sigma$.

EXEMPLO 2.2 Autômato Finito

O autômato M abaixo reconhece números inteiros e reais.

$$\begin{array}{ll} M = (K, \Sigma, \delta, e_0, F) & \delta(e_0, d) = e_1 \\ \Sigma = (d, .) & \delta(e_1, d) = e_1 \\ K = (e_0, e_1, e_2, e_3) & \delta(e_1, .) = e_2 \\ F = (e_2, e_3) & \delta(e_2, d) = e_1 \\ & \delta(e_2, .) = e_1 \end{array}$$

Uma segunda maneira de representar um autômato finito é através de uma tabela de transição de estados, contendo os estados nas linhas e os símbolos do alfabeto nas colunas. A tabela a seguir é a tabela de transição para o autômato M definido acima.

| | d | . |
|-------|-------|-------|
| e_0 | e_1 | |
| e_1 | e_1 | |
| e_2 | | e_2 |
| e_3 | e_3 | |

Diz-se que uma sentença é aceita por um autômato (pertence à linguagem que ele representa) se, após seu processamento, o autômato pára em um estado final. Se, durante o processamento da sentença, ocorre uma situação de indefinição, o autômato pára e a sentença não é aceita. No exemplo anterior, as transições $\delta(e_0, .)$, $\delta(e_2, .)$ e $\delta(e_3, .)$ são situações indefinidas.

Um autômato finito pode também ser representado através de um grafo, no qual os nodos representam os estados, e os arcos representam as transições entre estados. O estado inicial é indicado por uma seta, e os estados finais são distinguidos por circunferências concêntricas. A Figura 2.1 ilustra um autômato finito que reconhece identificadores e constantes numéricas reais e inteiros.

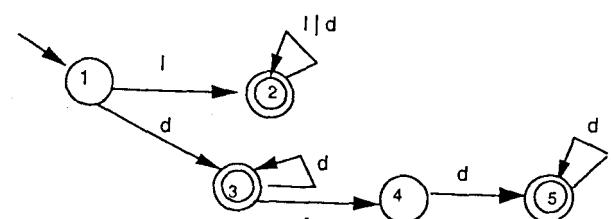


Figura 2.1 Exemplo de Autômato Finito

Além da representação através de gramáticas regulares ou autômatos finitos, a estrutura léxica dos *tokens* pode ser descrita por *expressões regulares*. A especificação dos símbolos léxicos em uma dessas notações é utilizada na construção do analisador léxico. Uma vantagem em usar expressões regulares para especificar os *iokens* é que essa notação é bastante clara e concisa. Além disso, a partir de uma expressão regular, pode-se utilizar um gerador de analisadores léxicos, tal como o LEX, para construir automaticamente um reconhecedor para os *tokens* gerados pela expressão.

Definição 2.6 Expressão regular.

Uma *expressão regular* r , sobre um conjunto de símbolos T , representa uma linguagem $L(r)$, a qual pode ser definida indutivamente a partir de *expressões básicas*, como segue:

1. \emptyset representa a linguagem vazia (conjunto contendo zero palavras);
2. $\{ E \}$ representa a linguagem cuja única palavra é a palavra vazia;
3. $(x | x \in T)$ representa a linguagem cuja única palavra é x ;
4. se r_1 e r_2 são expressões regulares definindo as linguagens $L(r_1)$ e $L(r_2)$, tem-se que:
 - $r_1 | r_2$ é a linguagem cujas palavras constituem o conjunto $L(r_1) \cup L(r_2)$;
 - $r_1 r_2$ é a linguagem $\{ vw \mid v \in L(r_1) \text{ e } w \in L(r_2) \}$, isto é, a linguagem cujas palavras são formadas pela **concatenação** de uma palavra de $L(r_1)$ com uma palavra de $L(r_2)$, nesta ordem;
 - r_1^* representa o conjunto $L^*(r_1)$, isto é, o conjunto de palavras que podem ser formadas concatenando-se zero ou mais palavras de $L(r_1)$.

EXEMPLO 2.3 Expressões regulares.

- | | |
|------------|--|
| $d(d)^*$ | representa o conjunto dos inteiros (assumido que d representa dígito) |
| $1(1 d)^*$ | representa identificadores que iniciam por letra seguida opcionalmente por letras e dígitos |

2.2 Tokens

Conforme mencionado anteriormente, a função do analisador léxico é ler uma sequência de caracteres que constitui um programa fonte e coletar, dessa sequência, os *tokens* (palavras de uma linguagem regular) que constituem o programa. Os *tokens* ou símbolos léxicos são as unidades básicas do texto do programa. Cada *token* é representado internamente por três informações:

- **classe do token**, que representa o tipo do *token* reconhecido. Exemplos de classes são: identificadores, constantes numéricas, cadeias de caracteres, palavras reservadas, operadores e separadores.
- **valor do token**, o qual depende da classe. Para *iokens* da classe constante inteira, por exemplo, o valor do *token* pode ser o número inteiro representado pela constante. Para tokens da classe identificador, o valor pode ser a sequência de caracteres, lida no programa fonte, que representa o identificador, ou um apontador para a entrada de uma tabela que contém essa sequência de caracteres. Essa tabela, chamada Tabela de Símbolos, é discutida na Seção 2.4. Algumas classes de tokens, como as palavras reservadas, não têm valor associado.
- **posição do token**, a qual indica o local do texto fonte (linha e coluna) onde ocorreu o token. Essa informação é utilizada, principalmente, para indicar o local de erros.

Em função do campo **valor**, os tokens podem ser divididos em dois **grupos**:

- **tokens simples**, que são os tokens que não têm um valor associado (como as palavras reservadas, operadores e delimitadores) porque a classe do *token* descreve-o completamente. Esses *tokens* correspondem a elementos fixos da linguagem.
- **iokens com argumento**, que são os *tokens* que têm um valor associado (como identificadores e constantes). Correspondem aos elementos da linguagem definidos pelo programador como, por exemplo, identificadores, constantes numéricas e cadeias de caracteres.

A seguir é repetida a sequência de *tokens* do Exemplo 1.1, a qual é obtida para o trecho de programa "while I < 100 do I := J + I ;". A análise léxica desse segmento produz a seguinte sequência:

[whi, I [id, 7] [<, I [cte, 13] [do,] [id, 7] [=,] [id, 12] [+ ,] [id, 7] [;, , I

Para simplificar, os *tokens* estão representados por pares (omitiu-se a posição). Identificadores e constantes numéricas estão representados pelo par [classe-token, índice-tabela]. As classes para palavras reservadas constituem-se em abreviações dessas, não sendo necessário passar seus valores para o analisador sintático. Para delimitadores e operadores, a classe é o próprio valor do *token*. Usualmente, os compiladores representam a classe de um *token* por um número inteiro para tomar a representação mais compacta. Neste texto, empregou-se uma representação simbólica para ajudar a compreensão.

2.3 Especificação

Conforme já referido, os analisadores Iéxicos são, usualmente, especificados através de notações para a descrição de linguagens regulares tais como autômatos finitos, expressões regulares ou gramáticas regulares. A especificação de um analisador Iéxico descreve o conjunto dos *tokens* que formam a linguagem. Também fazem parte dessa especificação sequências de caracteres que podem aparecer entre *tokens* e devem ser ignoradas tais como espaços em branco e comentários.

Na maioria das linguagens de programação, as palavras reservadas da linguagem são casos particulares dos identificadores e seguem a mesma notação desses. Logo, as especificações de analisadores Iéxicos, usualmente, não expressam, explicitamente, o reconhecimento de palavras reservadas. Essas palavras são armazenadas em uma tabela interna, que é examinada cada vez que um identificador é reconhecido. Se o identificador ocorre na tabela, então trata-se de uma palavra reservada. Caso contrário, trata-se de um identificador.

EXEMPLO 2.4 Reconhecimento de tokens.

Considere uma linguagem simples cujos *tokens* são os seguintes:

- identificadores formados por uma letra seguida, opcionalmente, por uma ou mais letras ou dígitos;
- números inteiros formados por um ou mais dígitos;
- comentários delimitados por (*) e (*).

Espaços em branco são ignorados. A Figura 2.2 representa um autômato finito que especifica o analisador Iéxico para essa linguagem. Nesse autômato, o símbolo B representa um espaço, os símbolos L e D representam, respectivamente, letra e dígito, e C e C' representam, respectivamente, qualquer caractere diferente de * e de * ou). Observe que há um estado terminal para cada classe de *token*.



O analisador Iéxico pode constituir um passo individual do compilador. Porém, em geral, o analisador Iéxico e o analisador sintático formam um único passo. Nesse caso, o analisador Iéxico atua como uma subrotina que é chamada pelo analisador sintático sempre que este necessita de mais um *token*. Os motivos que levam a dividir (conceitualmente) a análise do programa em análise Iéxica e sintática são os seguintes:

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

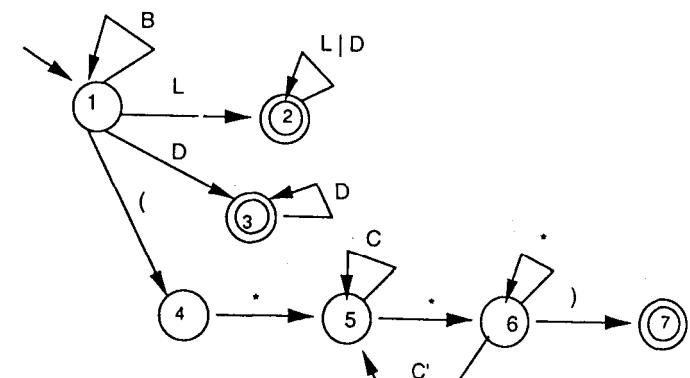


Figura 2.2. Autômato do analisador Iéxico

2.4 Implementação

A implementação de analisadores Iéxicos é feita, em geral, através de uma tabela de transição, a qual indica a passagem de um estado a outro pela leitura de um determinado caractere. Essa tabela e o correspondente programa de controle podem ser gerados automaticamente com o uso de um gerador de analisadores Iéxicos, tal como o LEX. No caso desse gerador, o projetista especifica os *tokens* a serem reconhecidos através de expressões regulares e, a partir dessas, o LEX gera um programa que implementa o analisador Iéxico correspondente.

Outra alternativa para implementação de um analisador Iéxico é a construção de um programa que simula o funcionamento do autômato correspondente, conforme é mostrado a seguir.

EXEMPLO 2.5 Programação de um autômato.

O trecho de código a seguir simula o funcionamento do **autômato finito** apresentado no Exemplo 2.4. Nesse código, a função *getchar* controla a leitura do texto fonte, retomando o caractere do programa fonte indicado pelo ponteiro de leitura e posicionando esse ponteiro sobre o próximo caractere a ser lido.

```

letter: set of (a .. z);
digit: set of (0 .. 9);
ident := null; /* inicialização */
number := null;
begin
    car := getchar;
    while car = '' do car := getchar;
    if car in letter
        then while car in (letter or digit) do
            begin ident := ident || car;
            car := getchar
            end
        else if car in digit
            then while car in digit do
                begin number := number || car;
                car := getchar
                end
            else if car in delimiter
                then ....
    ...
end

```

2.5 Tabela de Símbolos

A Tabela de Símbolos é uma estrutura de dados gerada pelo compilador com o objetivo de armazenar informações sobre os **nomes** (identificadores de **variáveis**, de **parâmetros**, de **funções**, de procedimentos, etc.) definidos no programa fonte. A Tabela de Símbolos associa atributos (tais como tipo, escopo, limites no caso de vetores e número de **parâmetros** no caso de funções) aos nomes definidos pelo programador. Em geral, a Tabela de Símbolos a ser construída durante a análise léxica, quando os identificadores são reconhecidos. Na primeira vez que um identificador é encontrado, o analisador léxico armazena-o na tabela,

mas talvez não tenha ainda condições de associar atributos a esse identificador. Às vezes, essa tarefa só pode ser executada durante as fases de análise sintática e **semântica**. Toda vez que um identificador é reconhecido no programa fonte, a Tabela de Símbolos é consultada, a fim de verificar se o nome já está registrado; caso não esteja, é feita sua inserção na tabela. É importante que o compilador possa variar dinamicamente o tamanho da Tabela de Símbolos. Se o tamanho for fixo, este deve ser escolhido grande suficiente para permitir a análise de qualquer programa fonte que se apresente.

Existem vários **modos** de organizar e acessar tabelas de símbolos. Os mais comuns são através de listas lineares, árvores binárias e tabelas *hash*. Lista linear é o mecanismo mais simples, mas seu desempenho é pobre quando o número de consultas é elevado. Tabelas *hash* têm melhor desempenho, mas exigem mais memória e esforço de programação.

Cada entrada na Tabela de Símbolos está relacionada com a declaração de um nome. As entradas podem não ser **uniformes** para classes distintas de identificadores. Por exemplo, entradas para identificadores de funções requerem registro do número de **parâmetros**, enquanto entradas para identificadores de matrizes requerem registro dos limites inferior e superior de cada **dimensão**. Nesses casos, pode-se ter parte da entrada uniforme e usar ponteiros para **registros** com informações adicionais.

O armazenamento dos nomes pode ser feito diretamente na tabela ou em uma **área** distinta. No primeiro caso, temos um desperdício de memória pela diversidade de tamanho dos identificadores; no segundo, a recuperação de nomes é ligeiramente mais demorada.

EXERCÍCIOS

- 1) Defina uma gramática regular que gere comentários em Pascal: sequências de caracteres delimitadas por (* e *), não contendo pares do tipo *).
- 2) Defina um autômato finito que reconheça sentenças em {0,1}*, as quais não contenham sequências do tipo 101.
- 3) Construa um autômato finito e a gramática regular equivalente à seguinte expressão regular

$$a(a|bc)^*b(a|b)^*$$
- 4) Defina um autômato finito e a gramática regular equivalente que reconheça/gere constantes numéricas em Pascal, cujo formato é o seguinte:

[+ | -] n [. n] [E [+ | -] n]

onde n é uma sequência de um ou mais dígitos,

[x] significa que x é opcional, e

| significa alternativa.

- 5) Defina um autômato finito que aceite sentenças delimitadas por apóstrofes, podendo conter quaisquer caracteres, inclusive a sentença vazia. Caso uma sentença venha a incluir outros apóstrofes, esses devem vir aos pares.
- 6) Defina uma gramática regular que gere identificadores segundo o formato:

$L(L|D)^*(_L(L|D)(L|D)^*)^*$

- 7) Escreva um programa que reconheça identificadores, strings e constantes numéricas válidos na linguagem Pascal.

3 Análise Sintática

A análise sintática constitui a segunda fase de um tradutor. Sua função é verificar se as construções usadas no programa estão gramaticalmente corretas. Normalmente, as estruturas sintáticas válidas são especificadas através de uma gramática livre do contexto¹ (GLC).

Dada uma gramática livre do contexto G e uma sentença (programa fonte) s, o objetivo do analisador sintático é verificar se a sentença s pertence à linguagem gerada por G. O analisador (ou reconhecedor) sintático, também chamado *parser*, recebe do analisador léxico a sequência de tokens que constitui a sentença s e produz como resultado uma árvore de derivação para s. Se a sentença é válida, ou emite uma mensagem de erro, caso contrário.

A árvore de derivação para s pode ser construída explicitamente (representada através de uma estrutura de dados) ou ficar implícita nas chamadas das rotinas que aplicam as regras de produção da gramática durante o reconhecimento. Os analisadores sintáticos devem ser projetados de modo que possam prosseguir na análise, até o fim do programa, mesmo que encontrem erros no texto fonte. Há duas estratégias básicas para a análise sintática:

- Estratégia TOP-DOWN ou DESCENDENTE
- Estratégia BOTTOM-UP ou REDUTIVA

Os métodos de análise baseados na estratégia *top-down* (descendente) constróem a árvore de derivação a partir do símbolo inicial da gramática (raiz da árvore), fazendo a árvore crescer até atingir suas folhas. A estratégia *bottom-up* (redutiva) realiza a análise no sentido inverso, isto é, a partir dos tokens do texto fonte (folhas da árvore de derivação) constrói a árvore até o símbolo inicial da gramática. Na estratégia *top-down*, em cada passo, um lado esquerdo de produção é substituído por um lado direito (expansão); na estratégia *bottom-up*, em cada passo, um lado direito de produção é substituído por um símbolo não-terminal (redução).

Recapitulando, a análise sintática do texto fonte de um programa é realizada por dois módulos especializados, os quais permitem simplificar e tornar mais eficiente o processo de

¹ Gramáticas mais simples, como as regulares, não permitem especificar construções do tipo de expressões aritméticas e comandos aninhados.

análise. O analisador léxico é especializado no reconhecimento do texto fonte original, fazendo a leitura efetiva dos caracteres e obtendo as unidades léxicas (*tokens*) do programa. Esse analisador vê o texto fonte como uma sequência de palavras de uma linguagem regular e o reconhece através de um autômato finito. Por seu lado, o analisador sintático vê o mesmo texto como uma sentença de uma linguagem livre do contexto, isto é, como uma sentença que deve satisfazer às regras gramaticais de uma **GLC**. Os *roteiros* coletados pelo analisador léxico são os símbolos terminais da **GLC**.

3.1 Revisão de Gramáticas Livres do Contexto

Nesta seção, é apresentada uma revisão de conceitos relacionados às gramáticas livres do contexto. Esta revisão baseou-se no livro *Linguagens Formais e Autômatos* de P. B. Menezes.

As gramáticas livres do contexto, popularizadas pela notação BNF (Backus Naur Form), formam a base para a análise sintática das linguagens de programação, pois permitem descrever a maioria das linguagens de programação usadas atualmente. É interessante observar a seguinte relação entre linguagens regulares e linguagens livres do contexto: uma linguagem regular pode ser reconhecida por um autômato simples e para a mesma é possível, a partir da expressão regular que a descreve, construir automaticamente um analisador léxico (por exemplo, usando o gerador de analisadores léxicos LEX). Semelhantemente, uma linguagem livre do contexto pode ser reconhecida por um autômato a pilha e para ela é possível, a partir da gramática livre do contexto que a descreve, construir automaticamente um analisador sintático (por exemplo, usando o gerador de compiladores YACC).

3.1.1 Definições e Exemplos

Definição 3.1 Gramática livre do contexto.

Gramática livre do contexto (GLC) é qualquer gramática cujas produções são da forma:

$$A \rightarrow a$$

onde **A** é um símbolo não-terminal e **a** é um elemento de $(N \cup T)^*$.

A denominação *livre do contexto* deriva do fato de o não-terminal **A** poder ser substituído por **a** em qualquer contexto, isto é, sem depender ("livre") de qualquer análise dos símbolos que sucedem ou antecedem **A** na forma sentencial em questão. Isso não acontece nas *gramáticas sensíveis ao contexto*, que têm produções da forma $a \rightarrow \beta$, com $\alpha \in (N \cup T)^+$ e

$$\beta \in (N \cup T)^*$$

EXEMPLO 3.1 Gramática livre de contexto

A gramática **G** a seguir gera expressões aritméticas:

$$G = (\{ E \}, \{ +, -, *, /, (,), x \}, P, E) \text{ sendo}$$

$$P = \{ E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid (E) \mid x \}$$

Definição 3.2 Árvore de derivação.

Árvore de derivação é a representação gráfica de uma derivação de sentença. Essa representação apresenta, de forma explícita, a estrutura hierárquica que originou a sentença.

Dada uma **GLC**, a árvore de derivação para uma sentença é obtida como segue:

- a) A raiz da árvore é o símbolo inicial da gramática;
- b) Os vértices interiores, obrigatoriamente, são não-terminais. Se $A \rightarrow X_1X_2\dots X_n$ é uma produção da gramática, então **A** será um vértice interior, e X_1, X_2, \dots, X_n serão os seus filhos (da esquerda para a direita);
- c) Símbolos terminais e a palavra vazia são vértices folha.

EXEMPLO 3.2 Árvore de derivação.

Seja a gramática $G = (\{<\text{número}>, <\text{num}>, <\text{digit}>\}, \{0, 1, 2, \dots, 9\}, P, <\text{número}>)$, onde as regras do conjunto P são:

$$<\text{número}> \rightarrow <\text{num}>$$

$$<\text{num}> \rightarrow <\text{num}> <\text{digit}> \mid <\text{digit}>$$

$$<\text{digit}> \rightarrow 0 \mid 1 \mid \dots \mid 9$$

A árvore de derivação correspondente à sentença "4 5" é a mostrada na Figura 3.1(a).

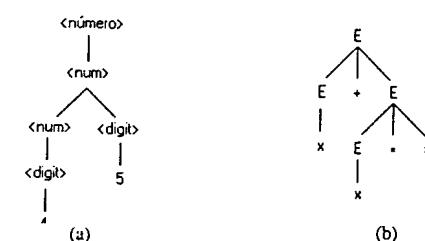


Figura 3.1 Árvores de derivação

EXEMPLO 3.3 Árvore de derivação x Derivações.

Seja a gramática $G = ((E), (+, *, (,), x), P, E)$, onde $P = \{ E \rightarrow E+E \mid E*E \mid (E) \mid x \}$, a qual gera expressões aritméticas com parênteses balanceados. Uma árvore de derivação correspondente à sentença $x+x*x$ é mostrada na Figura 3.1(b). Essa árvore pode ser obtida a partir de derivações distintas, todas gerando a mesma expressão, por exemplo:

- $E \Rightarrow E+E \Rightarrow x+E \Rightarrow x+E*E \Rightarrow x+x*E \Rightarrow x+x*x$
- $E \Rightarrow E+E \Rightarrow E+E*E \Rightarrow E+E*x \Rightarrow E+x*x \Rightarrow x+x*x$
- $E \Rightarrow E+E \Rightarrow E+E*E \Rightarrow x+E*E \Rightarrow x+x*E \Rightarrow x+x*x$

Ponanto, uma única árvore de derivação pode representar mais de uma derivação para uma mesma sentença.

Definição 3.3 Derivações mais à esquerda e mais à direita.

Derivação mais à esquerda de uma sentença é a sequência de formas sentenciais que se obtém derivando sempre o símbolo não-terminal mais à esquerda. Uma **derivação mais à direita** aplica as produções sempre ao não-terminal mais à direita.

No Exemplo 3.3, a **sequência (a)** é uma derivação mais à esquerda, e a sequência **(b)** é uma derivação mais à direita.

Definição 3.4 Gramática ambígua.

Gramática ambígua é uma gramática que permite construir mais de uma árvore de derivação para uma mesma sentença. A Figura 3.2 mostra duas árvores para a sentença $x+x*x$ considerada anteriormente.



Figura 3.2 Ambigüidade: árvores distintas para uma mesma expressão

A seguinte definição é equivalente à Definição 3.4: uma GLC é ambígua se existe alguma sentença com mais de uma derivação mais à esquerda (ou à direita). Isso porque, dada uma árvore, existe uma e somente uma derivação mais à esquerda correspondente à mesma. Árvores de derivação distintas originam derivações mais à esquerda distintas. Se a gramática admite construir mais de uma árvore para uma mesma sentença, é evidente que ela vai admitir mais de uma derivação mais à esquerda para essa sentença.

A existência de gramáticas ambíguas torna-se um problema quando os reconhecedores exigem derivações unívocas para obter um bom desempenho, ou mesmo para concluir a análise sintática. Se o uso da gramática fosse limitado a determinar se uma sequência de tokens pertence ou não à linguagem, a ambigüidade não seria tão problemática. Em algumas linguagens de programação, parte do significado dos comandos está especificada em sua estrutura sintática (i.e., existe semântica embutida na estrutura do programa). Nesses casos, a ambigüidade precisa ser eliminada.

No exemplo anterior, as duas árvores de derivação correspondem a diferentes sequências de avaliação para a expressão. Na árvore da esquerda, a operação **+** é realizada em primeiro lugar, e, na da direita, essa operação é executada em segundo lugar, o que concorda com a regra padrão da aritmética. Para essa gramática, a ambigüidade pode ser eliminada fazendo com que a multiplicação tenha precedência sobre a soma.

Infelizmente, não existe um procedimento geral para eliminar a ambigüidade de uma gramática. Aliás, existem gramáticas para as quais é impossível eliminar produções ambíguas. Uma linguagem é dita **inerentemente ambígua** se qualquer GLC que a descreva é ambígua. Por exemplo, a seguinte linguagem é inerentemente ambígua:

$$\{w \mid w = a^n b^n c^m d^m \text{ ou } a^n b^m c^m d^n, n \geq 1, m \geq 1\}$$

Definição 3.5 Gramática sem ciclos.

Gramática seni ciclos é uma GLC que não possui derivação da forma:

$$A \xrightarrow{*} A \text{ para algum } A \in N$$

Definição 3.6 Gramática **ϵ -livre**.

Gramática ϵ -livre é uma GLC que não possui produção do tipo $A \rightarrow \epsilon$, exceto, possivelmente, uma produção $S \rightarrow \epsilon$ (onde S é o símbolo inicial).

Definição 3.7 Gramática fatorada à esquerda.

Gramática fatorada à esquerda é uma GLC que não apresenta produções do tipo $A \rightarrow \alpha\beta_1 | \alpha\beta_2$ para alguma forma sentencial α .

Definição 3.8 Gramática recursiva à esquerda.

Gramática recursiva à esquerda é uma GLC que permite a derivação:

$$A \Rightarrow^* A\alpha \text{ para algum } A \in N$$

ou seja, um não-terminal deriva ele mesmo, de forma direta ou indireta, como o símbolo mais à esquerda de uma subpalavra gerada.

Os reconhecedores *top-down* exigem que a gramática não apresente recursividade à esquerda. Quando a recursão é direta, a eliminação é simples (ver Exemplo 3.4). Quando a recursão apresenta-se de forma indireta, a eliminação requer que a gramática seja inicialmente simplificada (ver Exemplo 3.5).

Definição 3.9 Gramática simplificada.

Gramática simplificada é uma GLC que não apresenta símbolos inúteis, produções vazias, nem produções do tipo $A \rightarrow B$.

3.1.2 Transformações de GLC's

Uma vez que existem diferentes métodos de análise, cada qual exigindo gramáticas com características específicas, é importante que uma gramática possa ser transformada, **porém**, sem perder a qualidade de gerar a mesma linguagem. As **gramáticas** que, mesmo tendo conjuntos diferentes de produções, geram a mesma linguagem são ditas **gramáticas equivalentes**.

Os métodos de reconhecimento **sintático** podem exigir uma **ou** mais das seguintes **transformações**: (a) eliminação de produções vazias, (b) retirada de recursividade à esquerda e (c) fatoração de produções. Os **algoritmos** para esses três tipos de **transformações** são apresentados nesta seção.

Transformação 1: Eliminação de produções vazias

O objetivo é eliminar produções da forma $A \rightarrow \epsilon$. Se ϵ faz parte da linguagem, então deverá ser incluída a produção $S \rightarrow \epsilon$, especialmente para gerar a palavra vazia (**nesse caso**, S não pode aparecer em nenhum lado direito de produção). A exclusão de produções vazias pode

causar modificações diversas nas produções da gramática. O algoritmo é dividido em três etapas, como segue.

Algoritmo para eliminação de produções vazias:

Seja $G = (N, T, P, S)$ uma gramática livre do contexto.

a) **Etapa 1:** Construir N_ϵ conjunto de não-terminalis que geram a palavra vazia:

$$N_\epsilon = \{ A \mid A \rightarrow \epsilon \}; \\ \text{repita}$$

$$N_\epsilon = N_\epsilon \cup \{ X \mid X \rightarrow X_1...X_n \in P \text{ tq } X_1,...,X_n \in N_\epsilon \}$$

até que o cardinal de N_ϵ não aumente.

b) **Etapa 2:** Construir conjunto de produções sem produções vazias. A gramática resultante desta etapa é $G_1 = (N, T, P_1, S)$, onde P_1 é construído como segue:

$$P_1 = \{ A \rightarrow \alpha \mid \alpha \neq \epsilon \}; \\ \text{repita} \\ \text{para toda } A \rightarrow a \in P_1 \text{ e } X \in N_\epsilon \text{ tq } a = \alpha_1 X \alpha_2 \text{ e } \alpha_1 \alpha_2 \neq \epsilon \\ \text{faça } P_1 = P_1 \cup \{ A \rightarrow \alpha_1 \alpha_2 \} \\ \text{até que o cardinal de } P_1 \text{ não aumente.}$$

c) **Etapa 3:** Incluir geração da palavra vazia, se necessário. Se a palavra vazia pertence à linguagem, então a gramática resultante é $G_2 = (N, T, P_2, S)$, onde $P_2 = P_1 \cup \{ S \rightarrow \epsilon \}$.

Transformação 2: Eliminação de recursividade à esquerda

Os reconhecedores descendentes (*top-down*) podem processar somente gramáticas que não apresentem **recursividade à esquerda**. Para os casos de recursividade direta, a recursividade pode ser eliminada manualmente, conforme é mostrado no Exemplo 3.4. Para recursões indiretas, a transformação é mais trabalhosa e exige que a gramática seja primeiramente simplificada.

EXEMPLO 3.4 Eliminação de recursividade direta.

A linguagem cujas palavras são formadas por um b seguido de zero ou mais a's (b, ba, haa, baaa, ...) pode ser gerada pelas seguintes regras de produção:

$$A \rightarrow Aa \mid b$$

ou, alternativamente, pelas seguintes, sem recursividade à esquerda:

$$\begin{aligned} A &\rightarrow bX \\ X &\rightarrow aX \mid \epsilon \end{aligned}$$

ou ainda pelas produções abaixo, se a produção vazia não é permitida:

$$\begin{aligned} A &\rightarrow b \mid bX \\ X &\rightarrow a \mid aX \end{aligned}$$

Eliminação de recursões diretas

A eliminação de recursões diretas é realizada pelo algoritmo a seguir, que generaliza o procedimento simples do Exemplo 3.4.

Substituir cada regra da forma:

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_n \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_m$$

onde nenhum β_i começa por A, por

$$\begin{aligned} A &\rightarrow \beta_1X \mid \beta_2X \mid \dots \mid \beta_mX \\ X &\rightarrow \alpha_1X \mid \alpha_2X \mid \dots \mid \alpha_nX \mid \epsilon \end{aligned}$$

Se produções vazias não são permitidas, a substituição pode ser:

$$\begin{aligned} A &\rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_m \mid \beta_1X \mid \beta_2X \mid \dots \mid \beta_mX \\ X &\rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_r \mid \alpha_1X \mid \alpha_2X \mid \dots \mid \alpha_nX \end{aligned}$$

Eliminação de recursões indiretas

A eliminação de recursões indiretas é realizada pelo algoritmo a seguir, em 3 etapas.

Seja $G = (N, T, P, S)$ uma gramática livre do contexto simplificada.

a) *Etapa 1: Renomeação dos não-terminais em uma ordem crescente se qualquer.*

Sendo n a cardinalidade de N, renomear os não-terminais para: A_1, A_2, \dots, A_n , e fazer as correspondentes renomeações nas regras de P.

b) *Etapa 2: Transformação das produções para o forma $A_r \rightarrow A_s\alpha$, onde $r \leq s$.*

Para r variando de 1 até n faça
 para s variando de 1 até $r-1$ faça $I^* \leq r \leq *$ /
 para toda $A_r \rightarrow A_s\alpha$ E P faça
 remova $A_r \rightarrow A_s\alpha$ de P;
 para toda $A_r \rightarrow \beta$ E P faça
 $P = P \cup \{A_r \rightarrow \beta\}$

c) *Etapa 3: Exclusão das recursões diretas.*

Substituir cada regra da forma:

$$A_r \rightarrow A_r\alpha_1 \mid A_r\alpha_2 \mid \dots \mid A_r\alpha_t \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_j$$

por

$$A_r \rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_j \mid \beta_1X \mid \beta_2X \mid \dots \mid \beta_jX$$

$$X \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_i \mid \alpha_1X \mid \alpha_2X \mid \dots \mid \alpha_iX$$

EXEMPLO 3.5 Eliminação de recursividade indireta.

Considere a seguinte GLC:

$$G = (\{S, A\}, \{a, b\}, P, S), \text{ onde:}$$

$$P = \{S \rightarrow AA \mid a, A \rightarrow SS \mid b\}$$

A exclusão da recursividade à esquerda é realizada como segue:

a) *Renomeação dos não-terminais em ordem crescente.* Os não-terminais S e A são renomeados para A_1 e A_2 , respectivamente. As produções ficam

$$\begin{aligned} A_1 &\rightarrow A_2A_2 \mid a \\ A_2 &\rightarrow A_1A_1 \mid b \end{aligned}$$

b) *Transformação das produções para a forma $A_r \rightarrow A_s\alpha$, com $r \leq s$.*

A produção $A_2 \rightarrow A_1A_1$ precisa ser modificada:

$$A_1 \rightarrow A_2A_2 \mid a$$

$$A_2 \rightarrow A_2A_2A_1 \mid aA_1 \mid b$$

c) *Etapa 3: Exclusão das recursões diretas.*

As regras de produção ficam como segue:

$$A_1 \rightarrow A_2A_2 \mid a$$

$$A_2 \rightarrow aA_1 \mid b \mid aA_1X \mid bX$$

$$X \rightarrow A_2A_1 \mid A_2A_1X$$

Transformação 3: Fatoração de uma gramática

Fatorar à esquerda a produção $A \rightarrow \alpha_1\alpha_2\dots\alpha_n$ é introduzir um novo não-terminal X e, para algum i, substitui-la por $A \rightarrow \alpha_1\alpha_2\dots\alpha_iX$ e $X \rightarrow \alpha_{i+1}\dots\alpha_n$.² A fatoração à esquerda permite eliminar a indecisão sobre qual produção aplicar quando duas ou mais produções iniciam com a mesma forma sentencial. Por exemplo, a indecisão referente às produções

² A fatoração à direita substituiria a produção original por $A \rightarrow X\alpha_{i+1}\dots\alpha_n$ e $X \rightarrow \alpha_1\alpha_2\dots\alpha_i$.

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$$

seria eliminada fatorando as mesmas para

$$A \rightarrow \alpha X \quad \text{e} \quad X \rightarrow \beta_1 \mid \beta_2$$

Para a análise descendente preditiva, é necessário que a gramática esteja fatorada à esquerda.

3.2 Análise Descendente (Top-down)

A análise descendente de uma sentença (ou programa) pode ser vista como uma tentativa de construir uma árvore de derivação em pré-ordem (da esquerda para a direita) para a sentença em questão: cria a raiz e, a seguir, cria as subárvore filhas, da esquerda para a direita. Esse processo produz uma derivação mais à esquerda da sentença em análise.

Nesta seção, serão estudados três tipos de analisadores sintáticos descendentes:

- recursivo com retrocesso (*backtracking*)
- recursivo preditivo
- tabular preditivo

Nos dois primeiros, cada símbolo não-terminal é implementado por um procedimento que efetua o reconhecimento do(s) lado(s) direito(s) das produções que definem o símbolo. O terceiro tipo, tabular preditivo, é implementado através de um autômato de pilha controlado por uma tabela de análise, a qual indica a regra de produção a ser aplicada relativa ao símbolo não-terminal que está no topo da pilha.

3.2.1 Análise Recursiva com Retrocesso

Esta análise faz a expansão da árvore de derivação a partir da raiz, expandindo sempre o não-terminal mais à esquerda. Quando existe mais de uma regra de produção para o não-terminal a ser expandido, a opção escolhida é função do símbolo corrente na fita de entrada (*token* sob o cabeçote de leitura). Se o *token* de entrada não define univocamente a produção a ser usada, então todas as alternativas vão ser tentadas até que se obtenha sucesso (ou até que a análise falhe irremediavelmente).

EXEMPLO 3.6 Funcionamento de um analisador descendente com retrocesso.

Considere a sentença `[a]` derivada a partir da gramática abaixo, a qual gera listas:

$$S \rightarrow a \mid [L]$$

$$L \rightarrow S ; L \mid S$$

A análise descendente dessa sentença começa com o símbolo inicial **S** na raiz da árvore de derivação e com o cabeçote de leitura sobre o primeiro caractere da sentença. Como tal caractere é um colchete, a primeira produção a ser aplicada deve ser $S \rightarrow [L]$ (Figura 3.3.a).

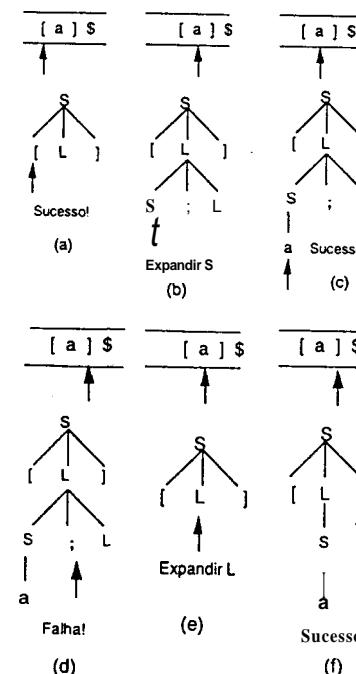


Figura 3.3 Reconhecimento da sentença `[a]`

O reconhecimento de `[` é bem sucedido, e a análise prossegue com a derivação de `L`, que pode ser efetuada usando um dos lados direitos alternativos: `S ; L` ou `S`. No exemplo, é escolhida a primeira alternativa `S ; L` (Figura 3.3.b), e `S` é expandido novamente (Figura 3.3.c), obtendo-se sucesso. Porém a comparação seguinte (`]` com `;`) falha (Figura 3.3.d), e o analisador deve, então, retroceder na fita de entrada para o ponto em que esta estava por ocasião da opção pela primeira alternativa de `L` (Figura 3.3.e). É aplicada, então, a segunda alternativa, $L \rightarrow S$. A derivação final é obtida aplicando-se a produção $S \rightarrow a$ (Figura 3.3.f).

No reconhecimento da sentença de entrada, a sequência de substituições de símbolos não-terminais pelos respectivos lados direitos de produções define a árvore de derivação correspondente à sentença.

EXEMPLO 3.7 Analisador recursivo com retrocesso.

O código de um analisador recursivo com retrocesso é apresentado a seguir. A gramática é a mesma do exemplo anterior (gramática que gera listas). Para cada símbolo não-terminal da gramática, é desenvolvida uma função que faz o reconhecimento das produções alternativas que definem esse símbolo. A função LETOKEN retorna um roken lido a partir da sentença de entrada; MARCA–PONTO marca, na sentença de entrada, um ponto de possível reinício da análise; e RETROCEDE volta o ponteiro de leitura para o último ponto marcado.

```

begin /* programa principal */
token:= LETOKEN;
if S
then if token = '$' then write('SUCESSO') else write('ERRO')
else write('ERRO')
end

function S ;
if token = 'a'
then { token := LETOKEN; return true}
else if token = 't'
then {token := LETOKEN;
if L
then if token = ']'
then {token := LETOKEN; return true}
else return false
else return false}
else return false

function L ;
MARA–PONTO ;
if S
then if token = ','
then {token := LETOKEN ;
if L
then return true
else return false)
else (RETROCEDE ; I* retrocede apenas o cabeçote de leitura. */
if S
then return true
else return false)
else return false

```

Ao processo de voltar atrás no reconhecimento e tentar produções alternativas dá-se o nome de retrocesso ou *backtracking*. Tal processo é inefficiente, pois leva à repetição da leitura de partes da sentença de entrada e, por isso, em geral, não é usado no reconhecimento de linguagens de programação. Na seção seguinte, é apresentado um analisador descendente que evita o retrocesso. Como o reconhecimento é, geralmente, acompanhado da execução de ações semânticas (por exemplo, armazenamento de identificadores na Tabela de Símbolos), a ocorrência de retrocesso pode levar o analisador sintático a ter que desfazer essas ações. Outra desvantagem dessa classe de analisadores é que, quando ocorre um erro, fica difícil indicar com precisão o local do erro, devido à tentativa de aplicação de produções alternativas.

Na verdade, para o caso particular da gramática do Exemplo 3.7, o reconhecimento não requer *backtracking*, pois a função L pode também ser programada como segue:

```

function L ;
if S
then if token = ';'
then {token := LETOKEN ;
if L
ihen return true
else return false]
else return tme
else return false

```

Embora não seja o caso do exemplo anterior, a presença de recursividade à esquerda em uma gramática ocasiona problemas para analisadores descendentes. Como a expansão é sempre feita para o não-terminal mais à esquerda, o analisador irá entrar num ciclo infinito se houver esse tipo de recursividade.

3.2.2 Análise Recursiva Preditiva

É possível implementar analisadores recursivos sem retrocesso. Esses analisadores são chamados **recursivos preditivos** e, para eles, o símbolo sob o cabeçote de leitura determina exatamente qual produção deve ser aplicada na expansão de cada não-terminal. Esses analisadores exigem: (1) que a gramática não tenha recursividade à esquerda, (2) que a gramática esteja fatorada à esquerda e (3) que, para os não-terminais com mais de uma regra de produção, os primeiros terminais deriváveis sejam capazes de identificar, univocamente, a produção que deve ser aplicada a cada instante da análise. Portanto, para construir-se um

analisador preditivo, deve ser possível determinar, dado o símbolo a sob o cabeçote de leitura e o não-terminal **A** a ser derivado, qual das produções alternativas $A \rightarrow \beta_1 | \beta_2 | \dots | \beta_n$ é a que deriva, unicamente, a sequência que inicia por a .

Por exemplo, considere as produções abaixo, que definem os comandos if–then, while–do, repeat–until e atribuição:

$$\begin{aligned} \text{COMANDO} \rightarrow & \text{ if EXPR then COMANDO } | \\ & \text{ while EXPR do COMANDO } | \\ & \text{ repeat LISTA until EXPR } | \\ & \text{ id } \coloneqq \text{EXPR} \end{aligned}$$

Para essas produções, os primeiros terminais dos lados direitos (if, while, repeat e id) determinam, diretamente, a produção a ser aplicada.

Se as produções correspondentes a COMANDO fossem:

$$\begin{aligned} \text{COMANDO} \rightarrow & \text{ CONDICIONAL } | \\ & \text{ ITERATIVO } | \\ & \text{ ATRIBUIÇÃO} \\ \text{CONDICIONAL} \rightarrow & \text{ if EXPR then COMANDO} \\ \text{ITERATIVO} \rightarrow & \text{ repeat LISTA until EXPR } | \\ & \text{ while EXPR do COMANDO} \\ \text{ATRIBUIÇÃO} \rightarrow & \text{ id } \coloneqq \text{EXPR} \end{aligned}$$

ainda continuaria sendo possível determinar univocamente a produção a ser usada, porém seria necessário determinar quais terminais iniciam sentenças deriváveis a partir de CONDICIONAL, ITERATIVO e ATRIBUIÇÃO.

Os terminais que iniciam sentenças deriváveis a partir de uma forma sentencial β constituem o conjunto $\text{FIRST}(\beta)$. As regras abaixo definem esse conjunto:

- 1) se $\beta \Rightarrow^* E$ então E é um elemento de $\text{FIRST}(\beta)$;
- 2) se $\beta \Rightarrow^* a\delta$ então a é um elemento de $\text{FIRST}(\beta)$, sendo a um símbolo terminal e δ uma forma sentencial qualquer, podendo ser vazia.

Dado um símbolo não-terminal **A** definido por várias alternativas:

$$A \rightarrow \beta_1 | \beta_2 | \dots | \beta_n$$

a implementação de um reconhecedor recursivo preditivo para **A** exige que os conjuntos FIRST de β_1, β_2, \dots e β_n sejam disjuntos dois a dois. Por exemplo, para as produções acima, que definem COMANDO, tem-se os seguintes conjuntos:

$$\begin{aligned} \text{FIRST(CONDICIONAL)} &= \{ \text{if} \} \\ \text{FIRST(ITERATIVO)} &= (\text{while}, \text{repeat}) \\ \text{FIRST(ATRIBUIÇÃO)} &= \{ \text{id} \} \end{aligned}$$

os quais são disjuntos dois a dois. Para os analisadores preditivos tabulares, a serem estudados na próxima seção, uma segunda função, chamada FOLLOW, desempenha papel importante. Os algoritmos para calcular os conjuntos FIRST e FOLLOW serão vistos na seção 3.2.3.

EXEMPLO 3.8 Analisador recursivo preditivo.

A função correspondente ao não-terminal COMANDO seria programada como segue:

```
function COMANDO ;
  if token = 'if'
    then if CONDICIONAL
      then return true
      else return false
    else if token = 'while' or token = 'repeat'
      then if ITERATIVO
        then return true
        else return false
      else if token = 'id'
        then if ATRIBUIÇÃO
          then return true
          else return false
        else return false
      else return false
```

EXEMPLO 3.9 Analisador recursivo preditivo com subrotinas.

No exemplo a seguir, os não-terminais são reconhecidos por procedimentos tipo subrotina. O exemplo é um analisador recursivo preditivo para uma gramática que gera declarações de variáveis, de acordo com as seguintes regras:

$$\begin{aligned} \text{DECL} &\rightarrow \text{LISTA-ID} : \text{TIPO} \\ \text{LISTA-ID} &\rightarrow \text{id} | \text{LISTA-ID}, \text{id} \\ \text{TIPO} &\rightarrow \text{SIMPLES} | \text{AGREGADO} \\ \text{SIMPLES} &\rightarrow \text{int} | \text{real} \\ \text{AGREGADO} &\rightarrow \text{mat} \text{ DIMENSÃO SIMPLES} | \\ &\quad \text{conj SIMPLES} \\ \text{DIMENSÃO} &\rightarrow [\text{num}] \end{aligned}$$

Eliminando-se a recursividade à esquerda das produções acima, tem-se:

$$\begin{array}{lcl}
 \text{DECL} & \rightarrow & \text{LISTA_ID} : \text{TIPO} \\
 \text{LISTA-ID} & \rightarrow & \text{id L-ID} \\
 \text{L-ID} & \rightarrow & , \text{id L-ID} \mid \epsilon \\
 \text{TIPO} & \rightarrow & \text{SIMPLES} \mid \text{AGREGADO} \\
 \text{SIMPLES} & \rightarrow & \text{int} \mid \text{real} \\
 \text{AGREGADO} & \rightarrow & \text{mat DIMENSÃO SIMPLES} \mid \text{conj SIMPLES} \\
 \text{DIMENSÃO} & \rightarrow & [\text{num}]
 \end{array}$$

Na programação a seguir, a variável global *token* é atualizada pelo procedimento LETOKEN. O procedimento ERRO informa o tipo de erro e trata da sua recuperação

```

begin      I* programa principal *I
    LETOKEN
    DECL
end

procedure DECL ;
    LISTA-ID ;
    if token = ':' then { LETOKEN ;
        TIPO }
    else ERRO

procedure L-ID ;
    if token = ',' then ( LETOKEN ;
        if token = 'id' then { LETOKEN ;
            L-ID }
        else ERRO
    else return;

procedure SIMPLES ;
    if token = 'int' then LETOKEN
    else if token= 'real' then LETOKEN
    else ERRO

procedure AGREGADO ;
    if token = 'mat' then { LETOKEN;
        DIMENSÃO;
        SIMPLES }
    else { LETOKEN; /*conjunto*/
        SIMPLES }

procedure LISTA-ID;
    if token = 'id'
    then { LETOKEN ;
        L-ID }
    else ERRO

procedure TIPO ;
    if token = 'int' or token = 'real' then SIMPLES
    else if token='mat' or token='conj' then AGREGADO
    else ERRO
  
```

É importante notar que, para as produções que derivam a palavra vazia, não é escrito código. Isso ocorre no exemplo anterior, na subrotina que implementa o símbolo **L-ID**. Se o restante da sentença a ser analisada inicia por ',', o analisador tenta reconhecer **id L-ID**;

senão, sai da subrotina **L-ID** sem chamar a rotina de ERRO, significando o **reconhecimento** de **L-ID → E**.

3.2.3 Análise Preditiva Tabular

É possível construir analisadores preditivos *não recursivos* que utilizam uma pilha explícita ao invés de chamadas recursivas (pilha implícita). Esse tipo de analisador implementa um autômato de pilha controlado por uma tabela de análise. O princípio do reconhecimento preditivo é a determinação da produção a ser aplicada, cujo lado direito irá substituir o símbolo não-terminal que se encontra no topo da pilha. O analisador busca a produção a ser aplicada na tabela de análise, levando em conta o não-terminal no topo da pilha e o *token* sob o cabeçote de leitura.

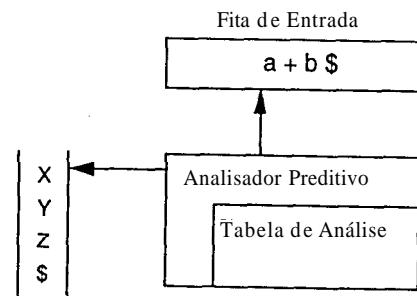


Figura 3.4 Analisador preditivo tabular

Um analisador preditivo orientado por tabela comprehende uma fita de entrada, uma pilha e uma tabela de análise, conforme é mostrado na Figura 3.4. A fita de entrada contém a sentença a ser analisada seguida de \$, símbolo que marca o fim da sentença. Inicialmente, a pilha contém \$, que marca a sua base, seguido do símbolo inicial da gramática. A tabela de análise é uma matriz M com n linhas e t+1 colunas, onde n é o número de símbolos não-terminais. e t é o número de símbolos terminais (a coluna extra corresponde ao símbolo \$).

O analisador é controlado por um programa que se comporta conforme descrito a seguir. Considerando X como o símbolo no topo da pilha e a como terminal da fita de entrada sob o cabeçote de leitura, o analisador executa uma de três ações possíveis:

- 1) se X = a = \$, o analisador pára, aceitando a sentença;

- 2) se $X = a \neq \$$, o analisador desempilha a e avança o cabeçote de leitura para o próximo símbolo na fita de entrada;
- 3) se X é um símbolo não-terminal, o analisador consulta a entrada $M[X, a]$ da tabela de análise. Essa entrada poderá conter uma produção da gramática ou ser vazia. Supondo $M[X, a] = (X \rightarrow UVW)$, o analisador substitui X (que está no topo da pilha) por UVW (ficando U no topo) e retoma a produção aplicada. Se $M[X, a]$ é vazia, isso corresponde a uma situação de erro; nesse caso, o analisador chama uma rotina de tratamento de erro.

O comportamento do analisador pode ser descrito através de uma tabela que mostra, a cada passo, o conteúdo da pilha e o restante da sentença a ser lida, conforme é exemplificado a seguir.

EXEMPLO 3.10 Movimentos de um analisador tabular preditivo.

Considere a gramática não-ambígua abaixo que gera expressões lógicas:

$$\begin{aligned} E &\rightarrow E \vee T \mid T \\ T &\rightarrow T \& F \mid F \\ F &\rightarrow \neg F \quad (i\ d) \end{aligned}$$

Eliminando-se a recursividade à esquerda das produções que definem E e T , obtém-se:

$$\begin{aligned} E &\rightarrow TE' \\ E' &\rightarrow \vee TE' \mid \epsilon \\ T &\rightarrow FT' \\ T' &\rightarrow \& FT' \mid \epsilon \\ F &\rightarrow \neg F \mid id \end{aligned}$$

A tabela de análise preditiva para essa gramática é mostrada a seguir:

| | id | \vee | $\&$ | \neg | \$ |
|------|---------------------|---------------------------|-------------------------|------------------------|---------------------------|
| E | $E \rightarrow TE'$ | | | $E \rightarrow TE'$ | |
| E' | | $E' \rightarrow \vee TE'$ | | | $E' \rightarrow \epsilon$ |
| T | $T \rightarrow FT'$ | | | $T \rightarrow FT'$ | |
| T' | | $T' \rightarrow \epsilon$ | $T' \rightarrow \& FT'$ | | $T' \rightarrow \epsilon$ |
| F | $F \rightarrow id$ | | | $F \rightarrow \neg F$ | |

Com a sentença $id \& id$, o reconhecedor preditivo realiza os movimentos mostrados na Figura 3.5

Inicialmente, o cabeçote aponta para o símbolo mais à esquerda da sentença de entrada. Observando as ações produzidas pelo reconhecedor, pode-se notar que as produções usadas na análise constituem uma derivação mais à esquerda da sentença.

| Pilha | Entrada | Ação |
|---------------|----------------------|---------------------------|
| \$E | id \vee id & id \$ | $E \rightarrow TE'$ |
| \$E' T | id \vee id & id \$ | $T \rightarrow FT'$ |
| \$E' T' F | id \vee id & id \$ | $F \rightarrow id$ |
| \$E' T' id | id \vee id & id \$ | desempilha e lê símbolo |
| \$E' T' | \vee id & id \$ | $T' \rightarrow \epsilon$ |
| \$E' | \vee id & id \$ | $E' \rightarrow \vee TE'$ |
| \$E' T \vee | \vee id & id \$ | desempilha e lê símbolo |
| \$E' T | id & id \$ | $T \rightarrow FT'$ |
| \$E' T' F | id & id \$ | $F \rightarrow id$ |
| \$E' T' id | id & id \$ | desempilha e lê símbolo |
| \$E' T' | & id \$ | $T' \rightarrow \& FT'$ |
| \$E' T' F & | & id \$ | desempilha e lê símbolo |
| \$E' T' F | id \$ | $F \rightarrow id$ |
| \$E' T' id | id \$ | desempilha e lê símbolo |
| \$E' T' | \$ | $T' \rightarrow \epsilon$ |
| \$E' | \$ | $E' \rightarrow \epsilon$ |
| \$ | \$ | Aceita a sentença! |

Figura 3.5 Movimentos de um analisador tabular preditivo

O algoritmo que guia os movimentos de um analisador preditivo não-recursivo é apresentado a seguir:

Algoritmo do Analisador Preditivo Tabular:

Entrada: Uma sentença S e a tabela de análise M para a gramática G .

Resultado: Uma derivação mais à esquerda de S , se S está em $L(G)$, ou uma indicação de erro, caso contrário.

Método: Inicialmente, o analisador está numa configuração na qual a pilha contém $\$S$ (com S no topo), e a fita de entrada contém $\$S$. O programa utiliza a tabela de análise preditiva M e comporta-se do seguinte modo:

Posiciona o cabeçote sobre o primeiro símbolo de $\$S$;

Seja X o símbolo do topo da pilha e a o símbolo sob o cabeçote.

```

Repete
  se X é um terminal
  então se X=a
    então desempilha X e avança o cabeçote
    senão ERRO
  senão /* X é um símbolo não-terminal */
    se M[X,a] = X → Y, Y, ... Yk
    então { desempilha X;
            empilha Y1Y2...Yk, com Yi no topo;
            imprime a produção X → Y1Y2...Yk }
    senão ERRO
até que X = $ I* pilha vazia */

```

□

Na implementação de um analisador preditivo tabular, a maior dificuldade está na constmção da tabela de análise. Para constmir essa tabela, é necessário computar duas funções associadas à gramática: as funções FIRST e FOLLOW.

Definição 3.10 FIRST(α).

Se α é um3 forma sentencial (sequência de símbolos da gramática), então FIRST(α) é o conjunto de terminais que iniciam formas sentenciais derivadas a partir de α . Se $\alpha \Rightarrow^* \epsilon$, então a palavra vazia também faz parte do conjunto.

Definição 3.11 FOLLOW(A).

A função FOLLOW é definida para símbolos não-terminais. Sendo A um não-terminal, FOLLOW(A) é o conjunto de terminais a que podem aparecer imediatamente à direita de A em alguma forma sentencial. Isto é, o conjunto de terminais a, tal que existe uma derivação da forma $S \Rightarrow^* \alpha A \beta$ para α e β quaisquer.

Cálculo das funções FIRST e FOLLOW

Algoritmo para calcular FIRST(X):

Para computar FIRST(X) para um símbolo X da gramática, aplicam-se as regras abaixo, até que não se possa adicionar mais terminais ou ϵ ao conjunto em questão.

- 1) Se a é terminal, então $FIRST(a) = \{ a \}$.
- 2) Se $X \rightarrow \epsilon$ é uma produção, então adicione ϵ a $FIRST(X)$

- 3) Se $X \rightarrow Y_1Y_2...Y_k$ é uma produção e, para algum i, todos $Y_1, Y_2, ..., Y_{i-1}$ derivam ϵ , então $FIRST(Y_i)$ está em $FIRST(X)$. Se todo Y_j ($j = 1, 2, ..., k$) deriva ϵ , então ϵ está em $FIRST(X)$.

Algoritmo para calcular FOLLOW(X):

Para computar FOLLOW(X), aplicam-se as regras abaixo até que não se possa adicionar mais símbolos ao conjunto.

- 1) Se S é o símbolo inicial da gramática e $\$$ é o marcador de fim da sentença, então $\$$ está em FOLLOW(S).
- 2) Se existe produção do tipo $A \rightarrow \alpha X \beta$, então todos os terminais de $FIRST(\beta)$ fazem parte de FOLLOW(X).
- 3) Se existe produção do tipo $A \rightarrow \alpha X$, ou $A \rightarrow \alpha X \beta$, sendo que $\beta \Rightarrow^* \epsilon$, então todos os terminais que estiverem em FOLLOW(A) fazem parte de FOLLOW(X).

EXEMPLO 3.11 Determinação das funções FIRST e FOLLOW.

Considere novamente a gramática da expressão lógica:

$$\begin{aligned} E &\rightarrow TE' \\ E' &+ \vee TE' \mid \epsilon \\ T &\rightarrow FT' \\ T' &\rightarrow \& FT' \mid \epsilon \\ F &\rightarrow \neg F \mid id \end{aligned}$$

Conjuntos FIRST:

Convém iniciar pelos não-terminais para os quais a obtenção do conjunto FIRST é trivial. Isso ocorre para F, que deriva apenas formas sentenciais que iniciam por terminais:

$$FIRST(F) = \{ \neg, id \}$$

T' deriva no terminal $\&$ e na palavra vazia; logo:

$$FIRST(T') = \{ \&, \epsilon \}$$

Similarmente, E' deriva no terminal \vee e na palavra vazia:

$$FIRST(E') = \{ \vee, \epsilon \}$$

Como T deriva apenas em FT', e F não deriva a palavra vazia, então $FIRST(T) = FIRST(F)$.

Portanto:

$$FIRST(T) = \{ \neg, id \}$$

Como E deriva apenas em $E \rightarrow TE'$ e T não deriva ϵ , tem-se $FIRST(E) = FIRST(T)$:

$$FIRST(E) = \{\neg, id\}$$

Conjuntos FOLLOW:

$FOLLOW(E)$ contém $\$$ pela regra 1:

$$FOLLOW(E) = \{ \$ \}$$

Tem-se que $FOLLOW(E') = FOLLOW(E)$, pois E' é o último símbolo do lado direito da produção $E \rightarrow TE'$ (regra 3):

$$FOLLOW(E') = \{ \$ \}$$

A partir da análise da produção $E' \rightarrow vTE'$, $FOLLOW(T)$ é obtido pela união de $FIRST(E')$, pela regra 2, com $FOLLOW(E')$, pela regra 3, pois $E' \rightarrow E$.

$$FOLLOW(T) = \{ v, \$ \}$$

$FOLLOW(T') = FOLLOW(T)$, pois T' é último na produção $T \rightarrow FT'$:

$$FOLLOW(T') = \{ v, \$ \}$$

$FOLLOW(F)$ é obtido pela união de $FIRST(T')$, pela regra 2, com $FOLLOW(T')$, pela regra 3:

$$FOLLOW(F) = \{ v, \&, \$ \}$$

□

O algoritmo para a construção da tabela de análise é apresentado a seguir.

Algoritmo para construir uma tabela de análise preditiva:

Entrada: gramática G

Resultado: Tabela de Análise M

Método:

- 1) Para cada produção $A \rightarrow a$ de G, execute os passos 2 e 3 (para criar a linha A da tabela M).
- 2) Para cada terminal a de $FIRST(\alpha)$, adicione a produção $A \rightarrow \alpha$ a $M[A, a]$.
- 3) Se $FIRST(\alpha)$ inclui a palavra vazia, então adicione $A \rightarrow a$ a $M[A, b]$ para cada b em $FOLLOW(A)$.

Aplicando o algoritmo acima à gramática que gera expressões lógicas, obtém-se as seguintes entradas para a tabela de análise. (A tabela é reprezentada abaixo.)

| | | | | |
|------|---------------------------|--------|-----------------------------|--|
| Para | $E \rightarrow TE'$ | tem-se | $FIRST(TE') = \{\neg, id\}$ | $M[E, \neg] = M[E, id] = E \rightarrow TE'$ |
| Para | $E' \rightarrow vTE'$ | tem-se | $FIRST(vTE') = \{v\}$ | $M[E', v] = E \rightarrow vTE'$ |
| Para | $E' \rightarrow \epsilon$ | tem-se | $FOLLOW(E') = \{ \$ \}$ | $M[E', \$] = E' \rightarrow \epsilon$ |
| Para | $T \rightarrow FT'$ | tem-se | $FIRST(FT') = \{\neg, id\}$ | $M[T, \neg] = M[T, id] = T \rightarrow FT'$ |
| Para | $T' \rightarrow \&FT'$ | tem-se | $FIRST(\&FT') = \{\&\}$ | $M[T', \&] = T' \rightarrow \&FT'$ |
| Para | $T' \rightarrow \epsilon$ | tem-se | $FOLLOW(T') = \{v, \$\}$ | $M[T', v] = M[T', \$] = T' \rightarrow \epsilon$ |
| Para | $F \rightarrow \neg F$ | tem-se | $FIRST(\neg F) = \{\neg\}$ | $M[F, \neg] = F \rightarrow \neg F$ |
| Para | $F \rightarrow id$ | tem-se | $FIRST(id) = \{id\}$ | $M[F, id] = F \rightarrow id$ |

| | id | v | & | \neg | $\$$ |
|------|---------------------|---------------------------|------------------------|------------------------|---------------------------|
| E | $E \rightarrow TE'$ | | | $E \rightarrow TE'$ | |
| E' | | $E' \rightarrow vTE'$ | | | $E' \rightarrow \epsilon$ |
| T | $T \rightarrow FT'$ | | | $T \rightarrow FT'$ | |
| T' | | $T' \rightarrow \epsilon$ | $T' \rightarrow \&FT'$ | | $T' \rightarrow \epsilon$ |
| F | $F \rightarrow id$ | | | $F \rightarrow \neg F$ | |

Se, em cada entrada da Tabela de Análise, existe apenas uma produção, então a gramática que originou a tabela é dita ser do tipo LL(1). Isso significa que as sentenças geradas pela gramática são passíveis de serem analisadas da esquerda para a direita (Left to right), produzindo uma derivação mais à esquerda (Leftmost derivation), levando em conta apenas um (1)símbolo da entrada. Essas gramáticas são mais bem caracterizadas adiante, na Definição 3.12.

O algoritmo anterior pode ser aplicado a qualquer gramática para produzir a tabela de análise. Contudo, para certas gramáticas, a tabela resultante poderá ter entradas multiplamente definidas. Isso ocorre para as gramáticas ambíguas, consequentemente, essas gramáticas não são do tipo LL(1).

EXEMPLO 3.12 Tabela sintática para uma gramática ambígua.

Seja G a gramática abaixo:

$$\begin{aligned} S &\rightarrow \text{if } C \text{ then } S \ S' \mid a \\ S' &\rightarrow \text{else } S \mid \epsilon \\ C &\rightarrow b \end{aligned}$$

A gramática G é ambígua, pois permite obter duas árvores de derivação distintas para a sentença `if b then if b then a else a.` conforme mostra a Figura 3.6.

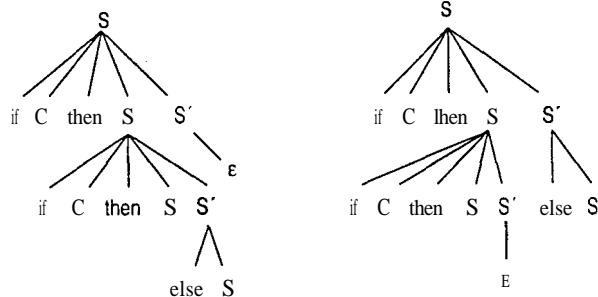


Figura 3.6 Árvore de derivação

A Tabela para G é apresentada abaixo. Vale observar que a regra $S' \rightarrow \epsilon$ aparece nas colunas "else" e "\$". Isso porque $\text{FOLLOW}(S') = \{\text{else}, \$\}$, pois $\text{FOLLOW}(S')$ inclui $\text{FOLLOW}(S)$, o qual inclui $\text{FIRST}(S')$.

| | a | b | else | if | then | \$ |
|------|-------------------|-------------------|--|---|------|---------------------------|
| S | $S \rightarrow a$ | | | $S \rightarrow \text{if } C \text{ then } S S'$ | | |
| S' | | | $S' \rightarrow \epsilon$ $S' \rightarrow \text{else } S$ | | | $S' \rightarrow \epsilon$ |
| C | | $C \rightarrow b$ | | | | |

Quando S' estiver no topo da pilha e else sob o cabeçote de leitura, o analisador terá duas opções: (1) apenas desempilhar S' , ou (2) desempilhar S' e empilhar else S. A primeira significa o reconhecimento do comando `if-then`; a segunda, o reconhecimento de `if-then-else`. Como as linguagens de programação em geral assumem que a cláusula `else` está associada ao `then` mais próximo, então, eliminando-se a produção $S' \rightarrow \epsilon$ da entrada $[S', \text{else}]$, o analisador sintático passa a ter o comportamento desejado.

Os movimentos que o analisador preditivo faz no reconhecimento da sentença `if b then if b then a else a` são mostrados na Figura 3.7.

□

Definição 3.12 Gramática LL(1).

As seguintes regras permitem identificar gramáticas LL(1). Uma gramática não recursiva à esquerda é LL(1) se e somente se, sempre que $A \rightarrow a$ e $A \rightarrow \beta$ são produções, ocorre que:

- 1) a interseção dos conjuntos $\text{FIRST}(\alpha)$ e $\text{FIRST}(\beta)$ é vazia;
- 2) no máximo um dos dois, α ou β , deriva a palavra vazia;
- 3) se $\beta \Rightarrow^* \epsilon$, então a interseção de $\text{FIRST}(\alpha)$ com $\text{FOLLOW}(A)$ é vazia.

Observe que $A \rightarrow a$ e $A \rightarrow \beta$ serão usadas para preencher a linha A da matriz de análise. Se as interseções referidas acima não fossem vazias, então haveria entradas multiplamente definidas na linha A.

| PILHA | ENTRADA | DERIVAÇÃO |
|--|--|---|
| $\$ S$ | $\text{if } b \text{ then if } b \text{ then } a \text{ else } a \$$ | $S \rightarrow \text{if } C \text{ then } S S'$ |
| $\$ S' S \text{ then } C \text{ if }$ | $\text{if } b \text{ then if } b \text{ then } a \text{ else } a \$$ | |
| $\$ S' S \text{ then } C$ | $b \text{ then if } b \text{ then } a \text{ else } a \$$ | $C \rightarrow b$ |
| $\$ S' S \text{ then } b$ | $b \text{ then if } b \text{ then } a \text{ else } a \$$ | |
| $\$ S' S \text{ then }$ | $\text{then if } b \text{ then } a \text{ else } a \$$ | |
| $\$ S' S$ | $\text{if } b \text{ then } a \text{ else } a \$$ | $S \rightarrow \text{if } C \text{ then } S S'$ |
| $\$ S' S' S \text{ then } C \text{ if }$ | $\text{if } b \text{ then } a \text{ else } a \$$ | |
| $\$ S' S' S \text{ then } C$ | $b \text{ then } a \text{ else } a \$$ | $C \rightarrow b$ |
| $\$ S' S' S \text{ then } b$ | $b \text{ then } a \text{ else } a \$$ | |
| $\$ S' S' S \text{ then }$ | $\text{then } a \text{ else } a \$$ | |
| $\$ S' S' S$ | $a \text{ else } a \$$ | $S \rightarrow a$ |
| $\$ S' S' a$ | $a \text{ else } a \$$ | |
| $\$ S' S'$ | $\text{else } a \$$ | $S' \rightarrow \text{else } S$ |
| $\$ S' S \text{ else }$ | $\text{else } a \$$ | |
| $\$ S' S$ | $a \$$ | $S \rightarrow a$ |
| $\$ S' a$ | $a \$$ | |
| $\$ S'$ | $\$$ | $S \rightarrow \epsilon$ |
| $\$$ | $\$$ | Aceita! |

Figura 3.7 Passos de um analisador preditivo tabular

3.3 Análise Redutiva (Bottom-up)

A análise redutiva de uma sentença (ou programa) pode ser vista como a tentativa de construir uma árvore de derivação a partir das folhas, produzindo uma derivação mais à direita ao reverso. A denominação redutiva refere-se ao processo que sofre a sentença de entrada, a qual é reduzida até ser atingido o símbolo inicial da gramática (raiz da árvore de derivação). Dá-se o nome de *redução* à operação de substituição do lado direito de uma produção pelo não-terminal correspondente (lado esquerdo).

Os analisadores redutivos, também chamados *empilha-reduz* são normalmente implementados por autômatos de pilha com controle dirigido por uma tabela de análise. Na configuração inicial do analisador, a fita de entrada contém a sentença a ser analisada seguida de um \$ (marcador de fim), e a pilha contém apenas o marcador de base \$.

O processo de reconhecimento consiste em transferir símbolos da fita de entrada para a pilha até que se tenha na pilha um lado direito de produção. Quando isso ocorre, esse lado direito é substituído (reduzido) pelo símbolo do lado esquerdo da produção. O processo segue adiante com esses movimentos (*empilhamento* e *redução*) até que a sentença de entrada seja completamente lida, e a pilha fique reduzida ao símbolo inicial da gramática.

EXEMPLO 3.13 Movimentos de um Analisador Redutivo.

Considere, novamente, a gramática que gera listas:

$$\begin{aligned} S &\rightarrow [L] \mid a \\ L &\rightarrow L; S \mid S \end{aligned}$$

No reconhecimento da sentença [a ; a], um analisador redutivo faria os movimentos mostrados na Figura 3.8. Observando as reduções indicadas na coluna Ação, nota-se que a análise da sentença [a ; a] produziu uma derivação mais à direita ao reverso:

$$S \Rightarrow [L] \Rightarrow [L; S] \Rightarrow [L; a] \Rightarrow [S; a] \Rightarrow [a; a]$$

○

Definição de handle

O analisador redutivo empilha símbolos da sentença de entrada até ter na pilha uma sequência de símbolos que corresponde à definição de algum não-terminal. Informalmente, é essa sequência de símbolos (que corresponde ao lado direito de uma produção) que define o *handle*. Embora o *handle* seja, na verdade, a produção cujo lado direito está no topo da pilha, neste texto, essa denominação será usada para referir-se apenas ao lado direito da produção. A *operação de redução* consiste em substituir, na pilha, o *handle* pelo lado esquerdo da produção. O uso da sequência correta de *handles* no processo de redução deverá levar, mais cedo ou mais tarde, ao símbolo inicial da gramática. Se coletada a sequência de *handles* utilizados na geração da árvore de derivação, essa sequência observada ao reverso (de trás para a frente) corresponderá a uma derivação mais à direita, a partir do símbolo inicial.

| Pilha | Entrada | Ação |
|------------|--------------|-----------------|
| \$ | [a ; a] \$ | empilha [|
| \$ [| a ; a] \$ | empilha a |
| \$ [a | ; a] \$ | reduz S → a |
| \$ [S | ; a] \$ | reduz L → S |
| \$ [L | ; a] \$ | empilha ; |
| \$ [L ; | a] \$ | empilha a |
| \$ [L ; a |] \$ | reduz S → a |
| \$ [L ; S |] \$ | reduz L → L ; S |
| \$ [L |] \$ | empilha] |
| \$ [L] | \$ | reduz S → [L] |
| \$ S | \$ | aceita ! |

Figura 3.8 Movimentos de um Analisador Redutivo

Definição 3.13 Handle.

No processo de análise, os *handles* são as sequências de símbolos que são lados direitos de produção, tais que suas reduções levam, no final, à redução para o símbolo inicial da gramática, através do reverso de uma derivação mais à direita. Se uma gramática G é não-ambígua, então toda forma sentencial gerada por G tem exatamente um *handle*.

As ações que podem ser realizadas por um reconhecedor empilha-reduz são as seguintes:

- **empilha:** coloca no topo da pilha o símbolo que está sendo lido e avança o cabeçote de leitura;
- **reduz:** substitui o *handle* do topo da pilha pelo não-terminal correspondente;
- **aceita:** reconhece que a sentença de entrada foi gerada pela gramática;
- **erro:** ocorrendo erro de sintaxe, chama uma subrotina de atendimento a erros.

Nesta seção, serão estudadas duas classes de analisadores do tipo empilha-reduz:

- 1) **analisadores de precedência de operadores**, muito eficientes no reconhecimento de expressões aritméticas e lógicas;
- 2) **analisadores LR**, que reconhecem a maior parte das linguagens livres do contexto.

3.3.1 Analisadores de Precedência de Operadores

Esses analisadores operam sobre a classe das *gramáticas de operadores*. Nessas gramáticas, os não-terminais aparecem sempre separados por símbolos terminais (isto é, nunca aparecem

dois não-terminais adjacentes) e, além disso, as produções não derivam a palavra vazia (i.e., nenhum lado direito de produção é " E^* "). Por exemplo, a gramática

$$\begin{aligned} E &\rightarrow E \circ E \mid (E) \mid id \\ O &\rightarrow + \mid - \mid * \mid / \mid ^* \end{aligned}$$

não é de precedência de operadores porque o lado direito " $E \circ E$ " contém três não-terminais consecutivos. Substituindo-se " O " por suas alternativas, a gramática passa a ser de operadores.

A análise de precedência de operadores é bastante eficiente e é aplicada, principalmente, no reconhecimento de expressões⁴. Dentre as desvantagens desses analisadores, estão a dificuldade em lidar com operadores iguais que tenham significados distintos (por exemplo, o operador " $-$ ", que pode ser binário ou unário) e o fato de eles se aplicarem a uma classe restrita de gramáticas.

Para identificar o *handle*, os analisadores de precedência de operadores baseiam-se em relações de precedência existentes entre os *tokens* (operandos e operadores). São três as relações de precedência entre os terminais: $<$, $>$ e $=$.

Sendo a e b símbolos terminais, tem-se que:

- $a < b$ significa que a tem precedência menor que b ;
- $a = b$ significa que a e b têm a mesma precedência; e
- $a > b$ significa que a tem precedência sobre b .

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

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

Esses analisadores são guiados por uma tabela de precedência, cujas relações definem o movimento que o analisador deve fazer: empilhar, reduzir, aceitar ou chamar uma rotina de atendimento a erro.

EXEMPLO 3.14 Tabela de precedência para expressões lógicas.

Para a gramática abaixo, que gera expressões lógicas, a tabela de precedência de operadores é a mostrada na Figura 3.9.

³ Vale observar que muitas gramáticas podem ser transformadas de modo a serem colocadas na forma de gramáticas de operadores. Nesse caso, grande parte dos símbolos terminais passam a ser "operadores" da gramática.

$$\begin{aligned} E &\rightarrow E \vee T \mid T \\ T &\rightarrow T \& F \mid F \\ F &\rightarrow (E) \mid id \end{aligned}$$

| | id | \vee | $\&$ | (|) | \$ |
|--------|----|--------|------|---|---|----|
| id | | > | > | | > | > |
| \vee | < | > | < | < | > | > |
| $\&$ | < | > | > | < | > | > |
| (| < | < | < | < | = | |
|) | | > | > | | > | > |
| \$ | < | < | < | < | | |

Figura 3.9 Tabela de precedência para expressões simplificadas

Observe que a tabela é uma matriz quadrada que relaciona todos os terminais da gramática mais o marcador \$. Desses terminais, poucos são realmente operadores (neste exemplo, os operadores reais são apenas \vee e $\&$). □

É importante saber que, na tabela, os terminais nas linhas representam terminais no topo da pilha, e os terminais nas colunas representam terminais sob o cabeçote de leitura. Basicamente, um analisador de precedência funciona da seguinte maneira. Seja a o terminal mais ao topo⁴ da pilha e b o terminal sob o cabeçote de leitura:

- 1) se $a < b$ ou $a = b$ então empilha;
- 2) se $a > b$ procura o *handle* na pilha (o qual deve estar delimitado pelas relações $<$ e $>$) e o substitui pelo nã-terminal correspondente.

No pilha, o *handle* vai desde o topo até (inclusive) o primeiro terminal x que tem abaixo de si um terminal y tal que $y < x$.

EXEMPLO 3.15 Movimentos de um Analisador de Precedência de Operadores.

A Figura 3.10 mostra os movimentos de um analisador para reconhecer a sentença *idvid&id*, considerando a gramática que gera expressões lógicas do Exemplo 3.14.

⁴ "Mais ao topo" porque no topo pode estar um nã-terminal. Na verdade, os analisadores de precedência desconsideram os não-terminais da gramática, levando em conta apenas a presença dos mesmos (suas identidades não interessam).

| Pilha | Relação | Entrada | Ação | Handle |
|--------------------|---------|----------------------|----------------|------------|
| \$ | < | id \vee id & id \$ | empilha id | |
| \$ id | > | \vee id & id \$ | reduz | id |
| \$ E | < | \vee id & id \$ | empilha \vee | |
| \$ E \vee | < | id & id \$ | empilha id | |
| \$ E \vee id | > | & id \$ | reduz | id |
| \$ E \vee E | < | & id \$ | empilha & | |
| \$ E \vee E & | < | id \$ | empilha id | |
| \$ E \vee E & id | > | \$ | reduz | id |
| \$ E \vee E & E | > | \$ | reduz | E & E |
| \$ E \vee E | > | \$ | reduz | E \vee E |
| \$ E | | \$ | aceita | |

Figura 3.10 Movimentos de um analisador de precedência de operadores

O

O algoritmo abaixo define os passos de um analisador de precedência para o reconhecimento de uma sentença S .

Algoritmo do Analisador de Precedência de Operadores

Entrada: matriz de relações de precedência e a sentença a ser analisada $s\$$.

Resultado: a sequência de produções aplicadas no reconhecimento de S , caso S pertença a $L(G)$, caso contrário, uma indicação de erro.

Método:

repita

se $\$S$ é o topo da pilha e $\$$ está sob o cabeçote de leitura

então aceita e pára

senão I^* seja a o terminal mais ao topo da pilha e b o terminal sob o cabeçote $*/$

se $a < b$ ou $a = b$

então empilha b e avança \mathbf{o} cabeçote

senão se $a > b$

então repita I^* reduzir $*/$

desempilha

até encontrar a **relação** < entre o terminal do topo e o último desempilhado;

empilha o **não-terminal** correspondente

imprime a produção aplicada

senão chama rotina de erro.

Construção da Tabela de Precedência de Operadores

Na análise de precedência de operadores, a maior dificuldade é a determinação da tabela a ser usada. Existem dois métodos para computar as relações de precedência entre os símbolos terminais de uma gramática de operadores:

- 1) **método intuitivo:** baseado no conhecimento da precedência e associatividade dos operadores. Por exemplo, se $*$ tem precedência sobre $+$, então faz-se:

$+$ (na pilha) $<$ * (na entrada) e

* (na pilha) $>$ + (na entrada)

- 2) **método mecânico:** obtém a tabela diretamente a partir da **gramática** de operadores. desde que as produções **reflitem** a associatividade e a precedência dos operadores.

Método intuitivo:

Este método obtém as relações de precedência a partir do conhecimento da associatividade e da precedência dos operadores da gramática. Considere dois operadores θ_1 e θ_2 .

- 1) Se o operador θ_1 tem maior precedência que o operador θ_2 , então θ_1 (na pilha) $>$ θ_2 (na entrada), e θ_2 (pilha) $<$ θ_1 (entrada). Por exemplo, o operador de multiplicação $*$ tem maior precedência que o operador $+$, logo, $* > +$, e $+ < *$.

- 2) Se θ_1 e θ_2 têm igual precedência (ou são iguais) e são associativos à esquerda, então $\theta_1 > \theta_2$, e $\theta_2 > \theta_1$; se são associativos à direita, então $\theta_1 < \theta_2$, e $\theta_2 > \theta_1$. Por exemplo, os operadores $*$ e $/$ têm a mesma precedência e são associativos à esquerda, portanto, $* > /$, e $/ > *$. Já o operador de exponenciação $**$ é associativo à direita, logo $** > **$.

- 3) As relações entre os operadores e os demais **tokens** (operandos e delimitadores) são fixas. Para todos os operadores θ , tem-se:

$\theta < id$ $\theta < ($ $\theta >)$ $\theta > \$$

$id > \theta$ $(< \theta$ $) > \theta$ $\$ < \theta$

- 4) As relações entre os **tokens** que não são operadores também são fixas:

$(< ($ $) >)$ $id >)$ $\$ < ($

$(=)$ $) > \$$ $id > \$$ $\$ < id$

$(< id$

EXEMPLO 3.16 Obtenção intuitiva da tabela de precedência de operadores

Dada a gramática:

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

tem-se os seguintes níveis de precedência e associatividade entre os operadores:

- ** tem maior precedência e é associativo à direita;
- * tem precedência intermediária e é associativo à esquerda;
- + tem menor precedência e é associativo à esquerda.

A tabela obtida, segundo as regras apresentadas anteriormente, é mostrada na Figura 3.11.

| | + | * | ** | (|) | id | \$ |
|----|---|---|----|---|---|----|--------|
| + | > | < | < | < | > | < | > |
| * | > | > | < | < | > | < | > |
| ** | > | > | < | < | > | < | > |
| (| < | < | < | < | = | < | |
|) | > | > | > | | > | | > |
| id | > | > | > | | > | | > |
| \$ | < | < | < | < | | < | Aceita |

Figura 3.11 Tabela de precedência obtida pelo método intuitivo

□

Para o caso de expressões lógicas, a tabela de precedência é obtida de maneira similar. Nesse caso, tem-se dois operadores binários (\vee e \wedge) e um operador unário prefixado (\neg), sendo que o operador \wedge tem precedência sobre o operador \vee . O caso dos operadores unários é considerado a seguir.

Operadores unários:

O operador lógico \neg apresenta maior precedência que o \vee e o \wedge . Para introduzi-lo, basta fazer “ $\theta < \neg$ ” e “ $\neg > 0$ ”, para qualquer operador 0, quer binário ou unário. Para o “ $-$ ”, a complicação adicional é o fato de que ele pode ser tanto binário como unário. Uma solução simples seria o analisador léxico distinguir entre os dois tipos de *tokens*, por exemplo, representando o operador unário por “ $-u$ ”. Nesse caso, a solução seria a mesma do “ \neg ”. Essa solução é usada pelo analisador léxico do Fortran, que devolve “ $-u$ ” se o “ $-$ ” é precedido de outro operador, de abre parênteses, de vírgula ou de símbolo de atribuição. Outra solução, não muito elegante, seria proibir o uso de “ $-$ ” unário: por exemplo, ao invés de escrever “ $-e$ ”, o programador seria obrigado a escrever “ $(0-e)$ ”.

Método mecânico:

Este método obtém as relações de precedência diretamente a partir da gramática de operadores, a qual precisa ser não-ambígua (observe que, por ser de operadores, a gramática já é ϵ -livre e não apresenta não-terminais adjacentes). A eliminação da ambigüidade será considerada no Exemplo 3.17.

As regras abaixo permitem determinar as relações de precedência entre os terminais da gramática. Para cada dois terminais a e b , diz-se que:

- 1) $a = b$ se $\alpha a \beta b \delta$ é lado direito de produção e β é E ou um (único) NT (símbolo não-terminal);
- 2) $a < b$ se $\alpha a X \beta$ é lado direito de produção e $X \Rightarrow^* \gamma b \delta$ onde γ é ϵ ou NT. Também, $\$ < b$ se $S \Rightarrow^* \gamma b \delta$, onde γ é E ou NT.
- 3) $a > b$ se $\alpha X b \beta$ é lado direito de produção e $A \Rightarrow^* \gamma a \delta$ onde δ é E ou NT. Também, $a > \$$ se $S \Rightarrow^* \gamma a \delta$ e δ é ϵ ou NT.

A regra 2 ($\alpha a X \beta$) diz: um terminal a seguido imediatamente de um não-terminal X tem precedência menor que os primeiros terminais deriváveis a partir de X (estes terminais aparecem em formas sentenciais precedidos de E ou NT). Estes terminais estarão num *handle* que será reduzido antes do *handle* que contém a .

A regra 3 ($\alpha X b \beta$) diz: todos os últimos terminais que podem ser derivados a partir de um não-terminal X (terminais que aparecem bem à direita, sucedidos de E ou NT) têm precedência maior que um terminal que segue imediatamente a X . Estes terminais estarão num *handle* que será reduzido antes do *handle* que contém b .

EXEMPLO 3.17 Eliminação da ambigüidade de uma gramática.

Seja G a seguinte gramática:

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

G é de operadores, porém é ambígua. É possível tomar G não-ambígua, transformando as produções para expressar a precedência e a associatividade dos operadores. Consegue-se isto introduzindo um símbolo não-terminal, para cada nível de precedência, conforme será visto neste exemplo.

Pelas regras normais da aritmética, uma expressão pode ser vista como um somatório de um ou mais termos (parcelas), cada termo sendo um produtório de um ou mais fatores. Isto porque as expressões são avaliadas fazendo primeiro as multiplicações (e divisões) e depois as somas (e subtrações). Isto significa que multiplicação e divisão tem precedência sobre soma e subtração. ou, mais informalmente, que os operadores $*$ e $/$ "agarram mais forte" que os operadores $+$ e $-$. Por exemplo, a expressão $3+2*4$ vale 11 (e não 20) porque a multiplicação "agarra mais" os seus operandos que a soma.

A associatividade dos operadores define a forma de avaliar as expressões quando ocorrem operadores com a mesma precedência. No caso de se ter associatividade à esquerda, a avaliação será feita da esquerda para à direita. Para os operadores aritméticos, com exceção do operador de potenciação $**$, que é associativo à direita⁵, todos os demais são associativos à esquerda.

A gramática não-ambígua que expressa as regras de precedência e de associatividade é a que segue:

$$\begin{array}{ll} E \rightarrow E + T \mid T & \text{(operador de menor precedência, associativo à esquerda)} \\ T \rightarrow T * F \mid F & \text{(operador de precedência média associativo à esquerda)} \\ F \rightarrow P^{**} F \mid P & \text{(operador de maior precedência, associativo à direita)} \\ P \rightarrow id \mid (E) & \text{(operandos)} \end{array}$$

Observe que o não-terminal P (operando) pode ser uma expressão entre parênteses. Dessa maneira, pode-se construir expressões com qualquer nível de aninhamento.

EXEMPLO 3.18 Obtenção da tabela de precedência pelo método mecânico.

Considere a gramática G a seguir, para a qual já foi obtida a tabela de precedência de operadores pelo método intuitivo:

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

Eliminando a ambigüidade (conforme o exemplo anterior), tem-se a seguinte gramática:

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

⁵ Na expressão $A^{**}I^{**}J$, é lógico avaliar da direita para a esquerda. isto é, calcular primeiro $K:=I^{**}J$ e depois calcular $A^{**}K$, porque se fosse usado o sentido esquerda→direita (isto é, calcular $B=A^{**}I$ e depois $B^{**}J$) o que obteríamos de fato seria $A^{**}(I^{**}J)$, e não é isso o que se quer (convença-se de que $(A^{**}I)^{**}J = A^{**}(I^{**}J)$).

Para facilitar a construção da matriz de precedência, é conveniente seguir os seguintes passos:

- Determinar, para cada não-terminal, os terminais que podem ocorrer como primeiro e último em alguma sentença derivada a partir daquele não-terminal.

| | primeiros | últimos |
|---|---------------|---------------|
| E | $+ * ** (id$ | $+ * **) id$ |
| T | $* ** (id$ | $* **) id$ |
| F | $** (id$ | $**) id$ |
| P | $id ($ | $id)$ |

- Para computar $<$, procurar pares aX (terminal, não-terminal) nos lados direitos de produção. Tem-se que a tem menor precedência do que qualquer "primeiro terminal" derivado a partir de X .

$$\begin{array}{lll} \text{Pares: } & + T & * F & ** F & (E \\ \text{Relações: } & + & < & (* ** (id) \\ & * & < & [* * (id) \\ & ** & < & (** (id) \\ & (& < & [+ * ** (id) \end{array}$$

- Para computar $>$, procurar pares Xb (não-terminal, terminal) nos lados direitos de produção. Qualquer "Último terminal" derivado de X tem precedência maior do que b .

$$\begin{array}{lll} \text{Pares: } & E + & T * & P^{**} & E) \\ \text{Relações: } & (+ * **) id] & > + \\ & { * **) id } & > * \\ & {) id } & > ** \\ & (+ * **) id) & >) \end{array}$$

- Para computar $=$, examinar os lados direitos de produção procurando por formas $a\beta b$, onde a e b são terminais, e β é NT, e fazer $a = b$. A única ocorrência dessa forma é na produção $P \rightarrow (E)$; portanto: $(=)$
- \$ tem precedência menor do que todos os "primeiros" terminais derivados a partir do símbolo inicial:

$$\$ < (+ * ** (id)$$

6) Todos os "Últimos" derivados a partir do símbolo inicial são $>$ do que $\$$:

$$(+ * **) id } > \$$$

A matriz de precedência completa é aquela mostrada na Figura 3.11

3.3.2 Funções de Precedência

Para uma classe grande de gramáticas, a matriz de precedência pode ser substituída pelas funções de precedência f e g que mapeiam símbolos terminais para inteiros. As funções de precedência permitem uma representação mais eficiente, pois ocupam um espaço $O(2^n)$, sendo n o número de terminais da gramática, enquanto a matriz de precedência ocupa um espaço $O(n^2)$. Infelizmente, nem toda tabela de precedência pode ser substituída por funções de precedência como pode ser visto no algoritmo apresentado mais adiante.

Sejam a e b símbolos terminais. A determinação de f e g é tal que:

$$f(a) < g(b) \text{ sempre que } a < b$$

$$f(a) = g(b) \text{ sempre que } a = b$$

$$f(a) > g(b) \text{ sempre que } a > b$$

Na comparação de dois símbolos a e b usa-se a função f para o símbolo da pilha e a função g para o da entrada. Assim, a relação de precedência entre a e b é equivalente à relação numérica entre $f(a)$ e $g(b)$. Como sempre existirá uma relação $<$, $>$ ou $=$ entre $f(a)$ e $g(b)$, quaisquer que sejam a e b , então as entradas de erro da matriz de precedência não terão representação. Os erros serão detectados quando, em reduções, os *handles* não forem encontrados na pilha.

Algoritmo para encontrar as Funções de Precedência

Entrada: matriz de precedência.

Saída: funções de precedência, se existirem.

- I) Criar símbolos fa e ga para cada terminal a e para o $\$$.
- 2) Distribuir os símbolos criados em grupos. Se $a=b$, então fa e gb ficam no mesmo grupo. Se $a=b$ e $c=b$, então fa e fc ficam no mesmo grupo que gb . Se, ainda, $c=d$, então fa , fc , gb e gd ficam no mesmo grupo, mesmo que $a=d$ não ocorra.

- 3) Gerar um grafo dirigido cujos nodos são os grupos anteriormente formados. Para quaisquer a e b , se $a > b$, construa um arco do grupo fa para o grupo gb e, se $a < b$, construa um arco do grupo gb para o grupo fa . (Observe que o grafo é *bipartido*, isto é, não existem ligações entre dois nodos f ou dois nodos g .)
- 4) Se o grafo contém ciclos, as funções de precedência não existem. Se não houver ciclos, $f(a)$ é igual ao comprimento do caminho mais longo iniciando em fa ; $g(a)$ é igual ao comprimento do caminho mais longo iniciando em ga .

EXEMPLO 3.19 Funções de precedência.

Seja a matriz de precedência para gramática que gera expressões lógicas simplificadas:

$$\begin{aligned} E &\rightarrow E \vee T \quad | \quad T \\ T &\rightarrow T \& F \quad | \quad F \\ F &\rightarrow id \end{aligned}$$

| | id | \vee | $\&$ | $\$$ |
|--------|-----|--------|------|--------|
| id | | $>$ | $>$ | $>$ |
| \vee | $<$ | $>$ | $<$ | $>$ |
| $\&$ | $<$ | $>$ | $>$ | $>$ |
| $\$$ | $<$ | $<$ | $<$ | Aceita |

Seguindo o algoritmo proposto, obtém-se o grafo da Figura 3.12. As funções f e g resultam os seguintes valores:

| | id | \vee | $\&$ | $\$$ |
|---|----|--------|------|------|
| f | 4 | 2 | 4 | 0 |
| g | 5 | 1 | 3 | 0 |

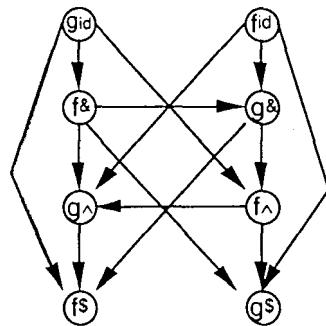


Figura 3.12. Grafo para obter as funções de precedência.

3.3.3 Analisadores LR(k)

Os analisadores LR (Level to Right with Rightmost derivation) são analisadores redutores eficientes que **lêem a sentença** em análise da esquerda para a direita e produzem uma derivação mais à direita ao reverso, considerando k símbolos sob o cabeçote de leitura. Dentre as vantagens **identificadas** nesses analisadores, destacam-se:

- 1) são capazes de reconhecer, praticamente, todas as estruturas sintáticas definidas por gramáticas livres do contexto;
- 2) o método de reconhecimento LR é mais geral que o de precedência de operadores e que qualquer outro do tipo empilha–reduz e pode ser implementado com o mesmo grau de eficiência;
- 3) analisadores LR são capazes de descobrir erros sintáticos no momento mais cedo. isto é, já na leitura **da** sentença em análise.

A principal desvantagem desses analisadores é a dificuldade de implementação dos mesmos, sendo necessário **utilizar** ferramentas automatizadas na construção da tabela de análise. Há, basicamente, três tipos de analisadores LR:

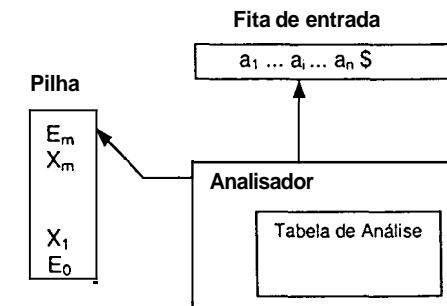
- 1) SLR (Simple LR), fáceis de **implementar**, porém aplicáveis a uma classe restrita de gramáticas;
- 2) LR Canônicos, mais poderosos, podendo ser aplicados a um **grande** número de linguagens livres do contexto; e

- 3) LALR (Look Ahead LR), de nível intermediário e implementação eficiente, que funciona para a maioria das linguagens de programação. O YACC gera esse tipo de analisador.

Nesta seção, será apresentada em detalhe a **construção** de um analisador SLR(1).

Funcionamento dos Analisadores LR

A estrutura genérica de um analisador LR(1) é mostrada na Figura 3.13. A **fita de entrada** mostra a sentença $(a_1 \dots a_i \dots a_n \$)$ a ser analisada, e a pilha armazena símbolos da gramática (X) intercalados com estados (E) do analisador. O símbolo da base da pilha é E_0 , estado inicial do analisador. O analisador é dirigido pela Tabela de Análise, cuja estrutura é mostrada na Figura 3.14.



X_j - símbolo da gramática

E_j - estado

Figura 3.13 Estrutura dos analisadores LR

A Tabela de Análise é uma tabela de transição de estados formada por duas partes: a parte **AÇÃO** contém ações (empilhar, reduzir, aceitar, ou condição de erro) associadas às transições de estados; e a parte **TRANSIÇÃO** contém transições de estados com relação aos símbolos não-terminais.

| | AÇÃO | | TERMINAIS | NÃO-TERMINAIS |
|---|---------|-------|-----------|---------------|
| | empilha | reduz | | |
| E | | | | estados |
| S | | | | |
| T | | | | |
| A | | | | |
| D | | | | |
| O | | | | |
| S | aceita | erro | | |

Figura 3.14 Estrutura da Tabela de Análise

O analisador funciona basicamente como segue. Seja E_m o estado do topo da pilha e a_i o token sob o caheçote de leitura. O analisador consulta a tabela $AÇÃO[E_m, a_i]$, que pode assumir um dos valores:

- a) empilha E : causa o empilhamento de "a, E ";
- b) reduz n (onde n é o número da produção $A \rightarrow \beta$): causa o desempilhamento de $2r$ símbolos, onde $r = |\beta|$, e o empilhamento de " AE_y " onde E_y resulta da consulta à tabela de $TRANSIÇÃO[E_{m-r}, A]$;⁶
- c) aceita: o analisador reconhece a sentença como válida;
- d) erro: o analisador pára a execução, identificando um erro sintático

O funcionamento do analisador pode ser entendido, considerando as transformações que ocorrem na pilha e na fita de entrada para cada ação. No texto que segue, consideram-se as configurações da pilha e da fita representadas por pares da forma (\dots pilha \dots , \dots fita \dots).

A configuração inicial de um analisador LR é:

($<E_0>, <a_1 a_2 \dots a_n \$>$)

Considerando que a configuração atual é como segue:

($<E_0 X_1 E_1 X_2 E_2 \dots X_m E_m>, <a_1 a_{i+1} \dots a_n \$>$)

tem-se que a configuração resultante após cada ação é:

$AÇÃO[E_m, a_i] = \text{empilha } X :$

($<E_0 X_1 E_1 X_2 E_2 \dots X_m E_m a_i X>, <a_{i+1} \dots a_n \$>$)

$AÇÃO[E_m, a_i] = \text{reduz } A \rightarrow \beta :$

($<E_0 X_1 E_1 X_2 E_2 \dots X_{m-r} E_{m-r} A E_y>, <a_i a_{i+1} \dots a_n \$>$)

⁶ E_{m-r} é o estado que está no topo da pilha logo após a operação de redução; após a transição, as 3 posições mais ao topo irão conter $E_{m-r} A E_y$.

sendo $|\beta| = r$ e $TRANSIÇÃO[E_{m-r}, A] = E_y$. Nesse caso, são desempilhados $2r$ símbolos, para depois ser empilhado " $A E_y$ ".

EXEMPLO 3.20 Movimentos de um analisador SLR(1).

Considerando a gramática abaixo, que gera expressões lógicas, a Figura 3.15 mostra a tabela de análise SLR(1) e os passos do analisador para reconhecer a sentença $\text{id} \& \text{id} \text{vid}$.

- | | |
|-----------------------------|------------------------------|
| 1) $E \rightarrow E \vee T$ | 4) $T \rightarrow F$ |
| 2) $E \rightarrow T$ | 5) $F \rightarrow (E)$ |
| 3) $T \rightarrow T \& F$ | 6) $F \rightarrow \text{id}$ |

| | AÇÃO | | | | | TRANSIÇÃO | | | |
|----|------|----|----|----|-----|-----------|----|---|----|
| | id | v | & | (|) | S | E | T | F |
| 0 | e5 | | | e4 | | | 1 | 2 | 3 |
| 1 | | e6 | | | | AC | | | |
| 2 | | r2 | e7 | | | r2 | r2 | | |
| 3 | | r4 | r4 | | | r4 | r4 | | |
| 4 | e5 | | | e4 | | | 8 | 2 | 3 |
| 5 | | r6 | r6 | | | r6 | r6 | | |
| 6 | e5 | | | e4 | | | | 9 | 3 |
| 7 | e5 | | | e4 | | | | | 10 |
| 8 | | e6 | | | e11 | | | | |
| 9 | | r1 | e7 | | | r1 | r1 | | |
| 10 | | r3 | r3 | | | r3 | r3 | | |
| 11 | | r5 | r5 | | | r5 | r5 | | |

| Pilha | Entrada | Ação/Transição |
|----------------|-----------------|--------------------------------|
| 0 | id & id v id \$ | e5: empilha id 5 |
| 0 id 5 | & id v id \$ | r6: reduz F→id TRANSIÇÃO[0 F] |
| OF3 | & id v id \$ | r4: reduz T→F TRANSIÇÃO[0 T] |
| 0 T 2 | & id v id \$ | e7: empilha & 7 |
| 0 T 2 & 7 | id v id \$ | e5: empilha id 5 |
| 0 T 2 & 7 id 5 | v id \$ | r6: reduz F-tid TRANSIÇÃO[7 F] |
| 0 T 2 & 7 F 10 | vid \$ | r3: reduz T→T&F TRANSIÇÃO[0 T] |
| OT 2 | vid \$ | r2: reduz E→T TRANSIÇÃO[0 E] |
| OE 1 | vid \$ | e6: empilha v 6 |
| OE 1 v 6 | id \$ | e5: empilha id 5 |

(continuação)

| Pilha | Entrada | Ação/Transição |
|----------------|---------|-------------------------------------|
| 0 E 1 v 6 id 5 | \$ | r6: reduz F→id TRANSIÇÃO[6 F] |
| 0 E 1 v 6 F 3 | \$ | r4: reduz T→F TRANSIÇÃO[6 T] |
| 0E 1 v 6 T 9 | \$ | r2: reduz E→EvT TRANSIÇÃO[0 E] |
| 0E 1 | \$ | ACEITA! |

Figura 3.15 Tabela de análise e passos do analisador

Construção de Analisadores SLR

A construção da tabela de controle para analisadores SLR baseia-se no que se denomina *Conjunto Canônico de Itens LR(0)*.

Um *item LR(0)*, para uma gramática G, é uma produção com um ponto em alguma posição do lado direito. O ponto é uma indicação de até onde uma produção já foi analisada no processo de reconhecimento. Por exemplo, a produção $A \rightarrow X Y Z$ origina quatro itens:

$$\begin{aligned} A &\rightarrow . X Y Z \\ A &\rightarrow X . Y Z \\ A &\rightarrow X Y . Z \\ A &\rightarrow X Y Z . \end{aligned}$$

enquanto a produção $A \rightarrow \epsilon$ gera apenas um item:

$$A \rightarrow .$$

A construção do Conjunto Canônico de Itens LR(0) requer duas operações:

- 1) acrescentar à gramática a produção $S' \rightarrow S$ (onde S é o símbolo inicial da gramática),
- 2) computar as funções *closirre* e *goto* para a nova gramática.

Cálculo da função closure(I)

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

- 1) Todo item em I pertence a *closure(I)*;
- 2) Se $A \rightarrow a.X\beta$ está no conjunto *closure(I)* e $X \rightarrow y$ é uma produção, então adicione $X \rightarrow .y$ ao conjunto.

A regra 2 aumenta o conjunto com as produções dos não-terminais que aparecem com um ponto no lado esquerdo.

EXEMPLO 3.21 Cálculo da função *closure(I)*.

Para as produções:

$$S \rightarrow a | [L]$$

$$L \rightarrow L ; S | S$$

e para $I = \{ S \rightarrow [. L] \}$, o conjunto *closure(I)* é o seguinte:

$$closure(\{ S \rightarrow [. L] \}) = \{ S \rightarrow [. L], L \rightarrow . L ; S, L \rightarrow . S, S \rightarrow . a, S \rightarrow [L] \}$$

Cálculo da função *goto(I,X)*

Informalmente, *goto(I,X)*, "avanço do ponto sobre X em I", consiste em coletar as produções com ponto no lado esquerdo de X, passar o ponto para a direita de X, e obter a função *closure* desse conjunto.

Formalmente, para X símbolo terminal ou não-terminal da gramática, *goto(I,X)* é a função *closirre* do conjunto dos itens $A \rightarrow \alpha X . \beta$, tais que $A \rightarrow a.X \beta$ pertence a I.

EXEMPLO 3.22 Cálculo da função *goto(I,X)*.

Considerando o conjunto $I = \{ S \rightarrow [L .] , L \rightarrow L ; S \}$, o cálculo de *goto(I,.)* resulta em:

$$goto(I,.) = \{ L \rightarrow L ; . S, S \rightarrow . a, S \rightarrow [L] \}$$

□

Algoritmo para obter o Conjunto Canônico de Itens LR(0)

Para uma gramática G, o Conjunto Canônico de Itens LR(0), referido por C, é obtido como segue.

Algoritmo:

Inicialização :

$$C = \{ I_0 = closure(\{ S' \rightarrow . S \}) \} \quad I^* \text{ os elementos de } C \text{ serão conjuntos */}$$

Repita :

Para cada conjunto I em C e X símbolo de G, tal que $goto(I,X) \neq \emptyset$
adicione $goto(I,X)$ a C
até que todos os conjuntos tenham sido adicionados a C.

Construção da Tabela de Análise SLR

Dada uma gramática G , obtém-se G' , aumentando G com a produção $S' \rightarrow S$, onde S é o símbolo inicial de G . A partir de G' , determina-se o conjunto canônico C . Finalmente, constróem-se as tabelas **AÇÃO** e **TRANSIÇÃO** conforme o algoritmo abaixo.

Algoritmo para construir a tabela SLR para G

Entrada: O conjunto C para G' .

Resultado: A tabela de análise SLR para G' (se ela existir).

Método: Seja $C = \{ I_0, I_1, \dots, I_n \}$. Os estados do analisador são $0, 1, \dots, n$ (0 é o estado inicial). A linha i da tabela é construída a partir do conjunto I_i , como segue.

As **ações** do analisador para o estado i são determinadas usando as regras:

- 1) se $\text{goto}(I_i, a) = I_j$, então faça $\text{AÇÃO}[i, a] = \text{empilha } j$;
- 2) se $A \rightarrow a.$ está em I_i , então para todo a em $\text{FOLLOW}(A)$, faça $\text{AÇÃO}[i, a] =$ reduz n , sendo n o número da produção $A \rightarrow a$.
- 3) se $S' \rightarrow S.$ está em I_i , então faça $\text{AÇÃO}[i, \$] = \text{aceita}$.

Se ações conflitantes são geradas pelas regras acima, então a gramática não é $\text{SLR}(1)$.

As **transições** para o estado i são construídas usando a regra:

- 4) se $\text{goto}(I_i, A) = I_j$, então $\text{TRANSIÇÃO}(i, A) = j$;

As entradas não definidas pelas regras acima correspondem a situações de erro. □

Definição 3.14 Gramática $\text{SLR}(1)$.

Uma gramática é dita $\text{SLR}(1)$ se, a partir dela, pode-se construir uma tabela de análise $\text{SLR}(1)$. Toda gramática $\text{SLR}(1)$ é não ambígua, porém existem gramáticas não-ambíguas que não são $\text{SLR}(1)$.

EXEMPLO 3.23 Cálculo da tabela SLR.

A seguir, é calculado o conjunto canônico de itens para a gramática que gera listas:

$S' \rightarrow S$ (produção adicionada)

1) $S \rightarrow a$

3) $L \rightarrow L ; S$

2) $S \rightarrow [L]$

4) $L \rightarrow S$

Cálculo dos Conjuntos $\text{LR}(0)$:

$$\begin{aligned} I_0 &= \{ S' \rightarrow .S, S \rightarrow .a, S \rightarrow .[L] \} \\ \text{goto}(I_0, S) &= I_1 = \{ S' \rightarrow S. \} \\ \text{goto}(I_0, a) &= I_2 = \{ S \rightarrow a. \} \\ \text{goto}(I_0, [) &= I_3 = \{ S \rightarrow [.L], L \rightarrow .L ; S, L \rightarrow .S, S \rightarrow .a, S \rightarrow .[L] \} \\ \text{goto}(I_3, L) &= I_4 = \{ S \rightarrow [L.], L \rightarrow L.; S \} \\ \text{goto}(I_3, S) &= I_5 = \{ L \rightarrow S. \} \\ \text{goto}(I_3, a) &= I_2 \\ \text{goto}(I_3, [) &= I_3 \\ \text{goto}(I_4, .) &= I_6 = \{ S \rightarrow [L]. \} \\ \text{goto}(I_4, ;) &= I_7 = \{ L \rightarrow L.; S, S \rightarrow .a, S \rightarrow .[L] \} \\ \text{goto}(I_7, S) &= I_8 = \{ L \rightarrow L; S. \} \end{aligned}$$

$C = \{ I_0, I_1, \dots, I_8 \}$, e a tabela SLR resultante é a apresentada na Figura 3.16.

| | AÇÃO | | | | TRANSIÇÃO | |
|---|------|----|----|----|-----------|-----|
| | a | [| ; | \$ | S | L |
| 0 | e2 | e3 | | | | 1 |
| 1 | | | | | AC | |
| 2 | | | r1 | r1 | r1 | |
| 3 | e2 | e3 | | | | 5 4 |
| 4 | | | e6 | e7 | | |
| 5 | | | r4 | r4 | | |
| 6 | | | r2 | r2 | r2 | |
| 7 | e2 | e3 | | | | 8 |
| 8 | | | r3 | r3 | | |

Figura 3.16 Tabela SLR

Convém observar que a tabela SLR é uma representação eficiente do autômato de pilha que reconhece a linguagem. O topo da pilha contém sempre o estado atual do autômato. Dado o estado atual e o token de entrada, a tabela indica a ação a ser executada. No caso da ação ser uma redução, a tabela também indica o próximo estado a ser assumido pelo autômato. As entradas em branco correspondem a situações de erro.

3.4 Recuperação de Erros

Quando um compilador detecta um erro de sintaxe, é desejável que ele tente continuar o processo de análise de modo a detectar outros erros que possam existir no código ainda não analisado. Isso envolve realizar o que se chama de recuperação (ou reparação) de erros.

Na recuperação de erros, tenta-se colocar o analisador em um estado tal que o restante da sentença de entrada possa ser analisada. Esse processo pode envolver modificação da pilha ou do restante da sentença em análise. Dependendo de como o processo é executado, erros falsos podem ser subsequentemente gerados a partir de um erro reparado, introduzindo erros em cascata. Por exemplo, no comando de atribuição

```
a := b c + d ;
```

um algoritmo de recuperação pode tentar reiniciar a análise, inserindo um ";" após o identificador b, assumindo que o identificador c inicia o próximo comando. Isso induziria um falso erro de sintaxe na posição do operador +.

A qualidade de uma rotina de recuperação de erros é medida pela precisão com que ela re-sincroniza a análise e é inversamente proporcional à sua capacidade de gerar falsos erros. Após a recuperação do primeiro erro na sentença em análise, as rotinas de verificação semântica devem continuar operacionais, enquanto as rotinas de geração de código devem ser desativadas, pois não existe motivo para executar o código objeto, dado que grande parte das reparações no código fonte requerem descarte de parte da sentença em análise.

A descoberta de erros pode ocorrer em dois momentos da análise:

- tempo mais cedo – na leitura da sentença de entrada, quando o *token* lido não é o esperado; acontece nos reconhecedores LL, LR e de precedência;
- tempo mais tarde – em operações sobre a pilha quando, numa redução, o *handle* não se encontra na pilha; acontece apenas na análise de precedência de operadores.

A forma mais simples de recuperação chama-se "modo pânico", no qual o analisador lê e descarta símbolos da sentença de entrada, até encontrar um *token* de sincronização (em geral, um delimitador ou palavra reservada), e remove, eventualmente, símbolos da pilha até o ponto a partir do qual consiga restabelecer o processo de análise. Esse tipo de recuperação não é plenamente satisfatório quando usado individualmente, mas em várias situações, é o único aplicável. A vantagem desse método é a simplicidade de implementação e o fato de ele nunca levar a um "loop" infinito.

3.4.1 Recuperação de Erros na Análise LL

Na tabela LL, as lacunas representam situações de erro e devem ser usadas para chamar rotinas de recuperação. Pode-se alterar a tabela de análise para recuperar erros segundo dois modos distintos:

- *modo pânico* - na ocorrência de um erro, o analisador despreza símbolos da entrada até encontrar um *token* de sincronização;
- *recuperação local* - o analisador tenta recuperar o erro, fazendo alterações sobre um símbolo apenas: desprezando o *token* da entrada, ou substituindo-o por outro, ou inserindo um novo *token*, ou ainda removendo um símbolo da pilha.

Modo Pânico

O conjunto de *tokens* de sincronização para um símbolo não-terminal A é formado pelos terminais em FOLLOW(A). Assim, quando o símbolo A estiver no topo da pilha e o símbolo da entrada for um *token* não esperado, mas pertencente ao conjunto FOLLOW(A), então a ação do analisador deverá ser "desempilha A".

Considerando o analisador preditivo tabular para a gramática a seguir, na qual FOLLOW(E) = { }, \$ }, FOLLOW(T) = { v, . }, \$ }, e FOLLOW(F) = { v, &, . }, \$ }. A tabela de análise ficaria tal como mostrada na Figura 3.17, onde *sinc* representa a ação de desempilhar o símbolo não-terminal do topo da pilha de análise.

$$\begin{array}{l} E \rightarrow TE' \\ E' + \quad v \, TE' \mid E \\ T \rightarrow FT' \\ T' \rightarrow \& \, FT' \mid \epsilon \\ F \rightarrow \neg F \mid (E) \mid id \end{array}$$

| | id | v | & | - | (|) | \$ |
|----|---------------------|---------------------------|----------------------------|------------------------|---------------------|---------------------------|---------------------------|
| E | $E \rightarrow TE'$ | | | $E \rightarrow TE'$ | $E \rightarrow TE'$ | sinc | sinc |
| E' | | $E' \rightarrow v \, TE'$ | | | | $E' \rightarrow \epsilon$ | $E' \rightarrow \epsilon$ |
| T | $T \rightarrow FT'$ | sinc | | $T \rightarrow FT'$ | $T \rightarrow FT'$ | sinc | sinc |
| T' | | $T' \rightarrow \epsilon$ | $T' \rightarrow \& \, FT'$ | | | $T' \rightarrow \epsilon$ | $T' \rightarrow \epsilon$ |
| F | $F \rightarrow id$ | sinc | sinc | $F \rightarrow \neg F$ | $F \rightarrow (E)$ | sinc | sinc |

Figura 3.17 Tabela de um reconhecedor LL estendida para tratamento de erros

A extensão do reconhecedor para incluir o tratamento de erros deve ser a seguinte:

- ao encontrar um *token* inesperado na sentença em análise,
 - (I) emitir mensagem de erro;
 - (2) tomar uma das seguintes atitudes:
 - se a entrada na tabela estiver vazia. ler o próximo *rokeri* (significa descarte do *rokeri* lido);
 - se a entrada é *sinc*, desempilhar o não-terminal do topo;
 - se o *rokeri* do topo não é igual ao símbolo da entrada. desempilhar o *rokeri*.

Recuperação Local

As rotinas de atendimento a erros podem fazer descarte, substituição ou inserção de apenas um símbolo a cada erro descoberto, tendo o cuidado de, no caso de inserção, não provocar um ciclo infinito no analisador.

Na recuperação local, a tabela LL(1) deve ser expandida para incluir as situações em que ocorre discrepância entre o *token* do topo da pilha e o da fita de entrada. A Figura 3.18 mostra a tabela anterior aumentada com linhas para os símbolos terminais. Nas linhas da tabela original, onde existiam produções vazias, as lacunas foram preenchidas com essas produções. As lacunas restantes foram preenchidas com nomes de rotinas de tratamento de erro.

| | <i>id</i> | <i>v</i> | & | \neg | (|) | \$ |
|--------|---------------------------|---------------------------|---------------------------|---------------------------|---------------------------|---------------------------|---------------------------|
| E | $E \rightarrow T E'$ | Errol | Errol | $E \rightarrow T E'$ | $E \rightarrow T E'$ | Errol | Errol |
| E' | $E' \rightarrow \epsilon$ | $E' \rightarrow v T E'$ | $E' \rightarrow \epsilon$ |
| T | $T \rightarrow F T'$ | Errol | Errol | $T \rightarrow F T'$ | $T \rightarrow F T'$ | Errol | Errol |
| T' | $T' \rightarrow \epsilon$ | $T' \rightarrow \epsilon$ | $T' \rightarrow \& F T'$ | $T' \rightarrow \epsilon$ | $T' \rightarrow \epsilon$ | $T' \rightarrow \epsilon$ | $T' \rightarrow \epsilon$ |
| F | $F \rightarrow id$ | Errol | Errol | $F \rightarrow \neg F$ | Errol | Errol | Errol |
| id | desempilha | | | | | | |
| v | | desempilha | | | | | |
| & | | | desempilha | | | | |
| \neg | | | | desempilha | | | |
|) | Erro2 | Erro2 | Erro2 | Erro2 | desempilha | Erro2 | Erro2 |
| \$ | Erro3 | Erro3 | Erro3 | Erro3 | Erro3 | Erro3 | AC |

Figura 3.18 Tabela LL(1) estendida para o tratamento de erros

As lacunas que permaneceram vazias (nas linhas dos *tokens* id, (, v, &, e \neg) representam situações que jamais ocorrerão durante a análise. Isso porque qualquer um desses *tokens* somente irá para o topo da pilha quando o mesmo for lido na fita de entrada (a pilha recebe um lado direito de produção que tem esse *token* como primeiro símbolo). Por exemplo, nunca acontecerá a situação em que o *token* do topo seja id. e o *token* sob o cabeçote de leitura seja &.

As rotinas de erro poderiam ser as seguintes:

Errol: insere o *rokeri* id na entrada e emite: "operando esperado";

Erro?: desempilha o símbolo ")" e emite: "parêntese direito esperado";

Erro3: descarta o *rokeri* da entrada e emite: "fim de arquivo encontrado"

3.4.2 Erros na Análise de Precedência de Operadores

Na análise de precedência de operadores, existem dois momentos nos quais o analisador pode descobrir erros sintáticos:

- na consulta à matriz de precedência. quando não existe relação de precedência entre o terminal mais ao topo da pilha e o símbolo da entrada (isto é, a entrada da matriz está vazia);
- quando o analisador supõe a existência de um *handle* no topo da pilha, mas não existe produção com o lado direito correspondente (isto é, não existe *handle* na pilha).

Erro na comparação do token na pilha com o token na entrada

As lacunas na tabela de precedência evidenciam condições de erro. Deve-se examinar caso a caso, para definir a mensagem e a recuperação apropriadas. Em geral, identificam-se classes de erros, tendo-se que implementar uma rotina para cada classe. Na tabela de precedência da Figura 3.19, as lacunas foram preenchidas com chamadas a rotinas de tratamento de erros. Nessa tabela, para quatro casos de erro, foi possível aplicar a mesma rotina (Erro?) de recuperação.

| | id | v | & | \neg | (|) | \$ |
|--------|-------|---|---|--------|-------|-------|-------|
| id | Erro2 | > | > | Erro2 | Erro2 | > | > |
| v | < | > | < | < | < | > | > |
| & | < | > | > | < | < | > | > |
| \neg | < | > | > | < | < | > | > |
| (| < | < | < | < | < | = | Erro1 |
|) | Erro2 | > | > | Erro2 | Erro2 | > | > |
| \$ | < | < | < | < | < | Erro3 | AC |

Figura 3.19 Tabela de precedência estendida para tratamento de erros

As rotinas de erro podem ser implementadas como segue:

Errol: empilha ")", e emite: "falta parêntese à direita";

Erro?: insere "v" na entrada e emite: "operador esperado";

Erro3: descarta ")" da entrada e emite: "parêntese direito ilegal".

Erro no momento de efetuar uma redução

Embora os símbolos não-terminais sejam transparentes na identificação do *handle*, a redução deve acontecer se realmente houver um *handle* (lado direito de produção) nas posições mais ao topo da pilha. Por exemplo, para a gramática que gera expressões lógicas (tabela apresentada na Figura 3.19):

$$E \rightarrow E \vee E \mid E \& E \mid (E) \mid \neg E \mid id$$

as seguintes verificações devem ser realizadas:

- se v ou & define um *handle*, verificar se existem não-terminais em ambos os lados do operador. Caso negativo, executar a redução e emitir a mensagem: "falta expressão";
- se o par "()" deve ser reduzido, verificar se existe um símbolo não-terminal entre os parênteses. Caso negativo, reduzir e emitir: "expressão nula entre parênteses";
- se \neg define o *handle*, verificar se existe um símbolo não-terminal acima dele na pilha. Caso negativo, executar a redução e emitir a mensagem: "falta expressão".

3.4.3 Recuperação de Erros na Análise LR

Na análise LR, os erros são identificados sempre no momento mais cedo, isto é, na leitura de *tokens*. Nesse tipo de analisadores, a cadeia de símbolos já empilhada está, com certeza, sintaticamente correta. As lacunas na tabela de AÇÃO representam situações de erro e, como tal, devem acionar rotinas de recuperação. A Figura 3.20 é a tabela de análise já vista no Exemplo 3.13, porém preenchida para o tratamento de erros.

Nas linhas em que houverem reduções, essas podem ser propagadas para as lacunas existentes, pois, de qualquer forma, os erros serão detectados nos passos subsequentes (na leitura do próximo *token*).

Nas linhas em que existem apenas ações de empilhar, as lacunas devem ser preenchidas com chamadas a rotinas de atendimento apropriadas. Por exemplo, na linha correspondente ao estado 7 (resultante de $I_7 = goto(I_4, :) = (L \rightarrow L ; .S, S \rightarrow .a, S \rightarrow .[L])$), é esperado um elemento atômico "a" ou uma sublistas iniciando por "["

| | AÇÃO | | | | TRANSIÇÃO | | |
|---|-------|-------|-------|-------|-----------|---|---|
| | a | [|] | ; | \$ | S | L |
| 0 | e2 | e3 | Erro2 | Erro1 | Erro1 | I | |
| 1 | Erro5 | Erro5 | Erro5 | Erro5 | AC | | |
| 2 | r1 | r1 | r1 | r1 | r1 | | |
| 3 | e2 | e3 | Erro1 | Erro1 | Erro1 | 5 | 4 |
| 4 | Erro3 | Erro3 | e6 | e7 | Erro4 | | |
| 5 | r4 | r4 | r4 | r4 | r4 | | |
| 6 | r2 | r2 | r2 | r2 | r2 | | |
| 7 | e2 | e3 | Erro1 | Erro1 | Erro1 | 8 | |
| 8 | r3 | r3 | r3 | r3 | r3 | | |

Figura 3.20 Tabela SLR estendida para tratamento de erros

As rotinas de erro poderiam ser as seguintes:

Errol: empilha o *token* a, cobre com estado 2 e emite: "elemento esperado";

Erro2: descarta *token* da entrada e emite: "colchete direito excedente";

Erro3: empilha o *token* ";", cobre com estado 7 e emite: ";" esperado";

Erro4: empilha], cobre com estado 6 e emite: "colchete direito esperado".

Erro5: descarta entrada e emite: "fim de arquivo esperado".

EXERCÍCIOS

1) Dada a gramática G abaixo:

$$\begin{aligned} G = (N, T, P, S) \quad N = \{S, L, E\} \quad T = \{\text{begin, end, id, ;, :=, v, -}\} \\ P = \{S \rightarrow \text{begin } L \text{ end} \mid \text{id} := E \\ L \rightarrow L; S \mid S \\ E \rightarrow E \vee E \mid \neg E \mid \text{id}\} \end{aligned}$$

a) A gramática G é ambígua? Justifique. Caso positivo, defina uma G', não ambígua, equivalente a G.

b) Elimine a recursividade à esquerda das produções de G.

2) Dada a gramática G abaixo, que gera expressões do cálculo de predicados:

$$\begin{aligned} G = (\{S\}, \{p, q, \neg, (,), \Rightarrow\}, P, S) \\ P = \{S \rightarrow \neg S \mid (S) \mid S \Rightarrow S \mid p \mid q\} \end{aligned}$$

a) G é ambígua? Justifique. Caso afirmativo, defina G', não ambígua, tal que $L(G) = L(G')$.

b) Escreva um analisador descendente recursivo preditivo que reconheça $L(G)$.

c) G é LL(1)? Justifique. Construa uma Tabela LL(1) para reconhecer $L(G)$.

3) Quais das 4 gramáticas abaixo são LL(1)?

$$\begin{array}{llll} S \rightarrow A B c & S \rightarrow A b & S \rightarrow A B B A & S \rightarrow a S e \mid B \\ A \rightarrow a \mid \epsilon & A \rightarrow a \mid B \mid \epsilon & A \rightarrow a \mid \epsilon & B \rightarrow b B e \mid C \\ B \rightarrow b \mid \epsilon & B \rightarrow b \mid \epsilon & B \rightarrow b \mid \epsilon & C \rightarrow c B e \mid d \end{array}$$

4) Considere a seguinte gramática, a qual gera sequências de dígitos.

$$S \rightarrow Sd \mid d$$

(a) Obtenha uma gramática equivalente não recursiva à esquerda.

(b) A gramática não recursiva obtida em (a) é LL(1)?

(c) A gramática equivalente $S \rightarrow dS \mid d$ é LL(1)?

(d) Fatorando a gramática de (c), ela passa a ser LL(1)?

(e) Se você tivesse que caracterizar uma gramática LL(2), como você faria?

(f) De acordo com a sua caracterização, a gramática de (c) é LL(2)?

5) Construa uma tabela LL(1) para a seguinte gramática:

$$\begin{aligned} E &\rightarrow - E \mid (E) \mid V L \\ L &\rightarrow - E \mid E \\ V &\rightarrow \text{id } S \\ S &\rightarrow (E) \mid \epsilon \end{aligned}$$

6) Transforme a gramática seguinte em LL(1):

$$\begin{aligned} \text{ListDecl} &\rightarrow \text{ListDecl} ; \text{Decl} \mid \text{Decl} \\ \text{Decl} &\rightarrow \text{Listald : Tipo} \\ \text{Listald} &\rightarrow \text{id} \mid \text{Listald , id} \\ \text{Tipo} &\rightarrow \text{Escalar} \mid \text{array (Dimens) of Tipo} \\ \text{Escalar} &\rightarrow \text{id} \mid (\text{Dimens}) \mid \text{int} \\ \text{Dimens} &\rightarrow \text{Limite .. Limite} \\ \text{Limite} &\rightarrow \text{int} \mid \text{id} \end{aligned}$$

7) Seja a gramática G abaixo, que gera expressões regulares sobre {a, b}:

$$S \rightarrow (S \vee S) \mid (SS) \mid (S)^* \mid a \mid b$$

a) Verifique se G é LL(1). Justifique.

b) Caso afirmativo, construa uma Tabela Preditiva para reconhecer $L(G)$; caso contrário, construa G' equivalente a G, tal que G' seja LL(1), e sua Tabela Preditiva.

c) Escreva um analisador descendente recursivo preditivo que reconheça $L(G)$.

d) Mostre os movimentos que um analisador LL(1) faria ao reconhecer a sentença $(a \vee (b(a)^*))$.

e) Escreva derivações mais à direita e mais à esquerda para $(a \vee (b(a)^*))$.

8) Quais das gramáticas abaixo são gramáticas de operadores?

$$\begin{array}{lll} S \rightarrow A b S & S \rightarrow A B c B A & S \rightarrow a S e \mid B \\ A \rightarrow a \mid B & A \rightarrow a A \mid a & B \rightarrow b B e \mid C \\ B \rightarrow b B \mid aa & B \rightarrow b B \mid b & C \rightarrow c B e \mid \epsilon \end{array}$$

9) Transforme a gramática abaixo em uma gramática de operadores:

$$\begin{aligned} E &\rightarrow - E \mid (E) \mid V L \\ L &\rightarrow - E \mid E \\ V &\rightarrow \text{id } S \\ S &\rightarrow (E) \mid \epsilon \end{aligned}$$

10) Construa uma Tabela de Precedência de Operadores para a gramática abaixo:

```

ListDecl → ListDecl ; Decl | Decl
Decl → Listald : Tipo
Listald → id | Listald , id
Tipo → Escalar | array ( Dimens ) of Tipo
Escalar → id | ( Dimens ) | int
Dimens → Limite .. Limite
Limite → int | id
  
```

11) Dadas as gramáticas G1 e G2 abaixo:

```

G1: S → ¬ S | ( S ) | S ⇒ S | p | q
G2: S → ( S ∨ S ) | ( S S ) | ( S ) * | a | b
  
```

- a) G1 e G2 são gramáticas de operadores? Justifique.
- b) Caso afirmativo, construa as tabelas de precedência correspondentes.

12) Dada a gramática $G = (N, T, P, S)$, onde

```

N = { S, L, E }
T = { repeat, until, id, ;, :=, v, ¬ }
P = ( S → repeat L until E | id := E
      L → L ; S | S
      E → E ∨ E | ¬ E | id )
  
```

- a) G é uma gramática de precedência de operadores? Justifique.
- b) Construa, pelo método mecânico, uma Tabela de Precedência que possibilite o reconhecimento de $L(G)$.
- c) Calcule as funções de precedência para analisar $L(G)$.
- d) Mostre os movimentos que um analisador de precedência faria ao reconhecer a sentença:

repeat id := id ; id := ¬ id until id ∨ id

13) Dada a gramática $G = (N, T, P, S)$, onde

```

N = { S L E }
T = { while do begin end a ; b }
P = { S → while E do S | begin L end | a
      L → L ; S | S
      E + b }
  
```

- a) G é uma gramática de precedência de operadores?
- b) G é SLR(1)?
- c) Defina uma gramática G' , equivalente a G, tal que G' seja LL(1).
(Justifique suas respostas)

14) Dada a gramática $G = (N, T, P, S)$, onde

```

N = { S E T }
T = { if then else id := v → ( ) }
P = { S → if E then S else S | id := E
      E → E ∨ T | T
      T → ¬ id | id | ( E ) }
  
```

- a) Escreva um analisador descendente recursivo preditivo que reconheça $L(G)$.
- b) Construa uma tabela LL(1) para reconhecer $L(G)$.
- c) Construa uma tabela SLR(1) para G, fazendo a seguinte simplificação nas produções derivadas de E:

$$E \rightarrow E \vee i d \mid \neg id \mid id$$
- d) Para as questões b) e c), especifique três procedimentos para tratamento de erros.
- e) Mostre os movimentos que os analisadores LL e LR fariam ao reconhecer a sentença:

if id → id then id := v id else ¬ id

15) Seja a gramática G abaixo:

```

S → L = R | R
L + * R | id
R → L
  
```

- a) G é ambígua? Justifique.
- b) G é uma gramática de precedência de operadores? Justifique.
- c) G é uma gramática SLR (1)? Justifique.
- d) Caso b) ou c) acima tenham respostas positivas, mostre os movimentos que um analisador de precedência ou SLR faria ao reconhecer a sentença

id * id = * id

e especifique procedimentos para tratamento de erros.

16) Dado o seguinte exemplo de comando de atribuição condicional

x := if a < b or b > c then x + y else x * z

- a) Defina uma gramática LL(1) que gere comandos desse tipo. construa a tabela LL(1) correspondente e especifique procedimentos para tratamento de erros;
- b) Mostre os movimentos que o analisador LL(1) faria ao reconhecer o comando exemplo acima;
- c) Escreva um analisador descendente recursivo que reconheça comandos desse tipo.

4 Tradução Dirigida por Sintaxe

Tradução Dirigida por Sintaxe é uma técnica que permite realizar tradução (geração de código) concomitantemente com a análise sintática. Ações semânticas são associadas às regras de produção da gramática de modo que, quando uma dada produção é processada (por derivação ou redução de uma forma sentencial no processo de reconhecimento), essas ações são executadas. A execução dessas ações pode gerar ou interpretar código, armazenar informações na tabela de símbolos, emitir mensagens de erro, etc.

Para tornar as ações semânticas mais efetivas, pode-se associar variáveis aos símbolos (terminais e não-terminais) da gramática. Assim, os símbolos gramaticais passam a conter atributos (ou parâmetros) capazes de armazenar valores durante o processo de reconhecimento. Toda vez que uma regra de produção é usada no processo de reconhecimento de uma sentença, os símbolos gramaticais dessa regra são "alocados" juntamente com seus atributos. Isto é, cada referência a um símbolo grammatical, numa regra, faz com que uma cópia desse símbolo seja criada, juntamente com seus atributos. Pensando na árvore de derivação da sentença sob análise, é como se a cada nó da árvore (símbolo grammatical) correspondesse uma instânciação de um símbolo e de suas variáveis. É como se o nó contivesse campos para armazenar os valores correspondentes ao símbolo. Daqui para a frente, os valores associados a um símbolo grammatical serão denominados *atributos do símbolo*.

EXEMPLO 4.1 Produções e ações semânticas.

Este exemplo mostra as produções usadas para especificar uma declaração simples e as ações semânticas associadas.

- 1) $D \rightarrow \text{var} : T \quad \{ \text{adTabSimb(var.nome, T.tipo)} \}$
- 2) $T \rightarrow \text{real} \quad \{ \text{T.tipo} := "r" \}$
- 3) $T \rightarrow \text{integer} \quad \{ \text{T.tipo} := "i" \}$

Durante o Processo de reconhecimento, quando a produção (2) ou (3) é reduzida, o atributo *tipo* associado ao não-terminal T recebe o valor do tipo reconhecido (caracter "r" ou "i"). A redução da produção (1) adiciona na tabela de símbolos o nome da variável e seu respectivo tipo. O atributo *nome* de var é inicializado quando esse terminal é reconhecido durante o processo de análise léxica.

4.1 Esquemas de Tradução

Um **Esquema de Tradução** é uma extensão de uma gramática livre do contexto. Esta extensão é realizada através da associação de atributos aos símbolos gramaticais e de ações semânticas às regras de produção. Os esquemas de tradução constituem uma notação apropriada para a especificação de operações que devem ser realizadas por um compilador durante a fase de análise sintática. Um atributo de um símbolo pode conter um valor numérico, uma cadeia de caracteres, um tipo de dado, um endereço de memória, etc. Ações semânticas podem ser avaliações de atributos ou chamadas a procedimentos e funções. Com exceção dos atributos dos terminais, que são "calculados" pelo analisador léxico, os valores dos demais atributos e variáveis são calculados durante a execução das ações semânticas.

Os atributos podem ser **sintetizados** ou **herdados**. Sendo **S** um símbolo, o valor de um atributo de **S** é dito sintetizado se ele é computado a partir, exclusivamente, dos valores dos atributos dos filhos de **S** (na árvore de derivação); o valor de um atributo de **S** é dito herdado se ele é computado a partir dos valores dos atributos dos irmãos ou do pai de **S**.

Dada uma sentença de entrada, a ela irá corresponder uma árvore de derivação, a qual será construída durante o processo de análise. A execução das ações semânticas irá definir os valores dos atributos dos nós da árvore para essa sentença de entrada. Deve ser observado que todas as informações referentes a uma sentença (programa fonte, no caso de um compilador) estão contidas na sentença original, isto é, na sequência de **tokens** da cadeia de entrada. Como os **tokens** ficam nas folhas da árvore de derivação, isso significa que todas as informações para um analisador-tradutor estão originalmente nessas folhas. Como os **tokens** (folhas da árvore) são coletados pelo analisador léxico, é natural que esse analisador obtenha os atributos dos mesmos. Conforme visto, um **token** é uma tupla que contém todas as informações referentes a um símbolo **terminal**. Por exemplo, para uma constante numérica, o **token** contém o código "cte" e o valor da constante; para um **identificador**, o token contém o código "id", e o valor do **token** pode ser o índice da tabela de símbolos em que o identificador foi instalado. Por convenção, o valor de um símbolo terminal **t** devolvido pelo analisador léxico será denotado por **t.lexval**. Por exemplo, **num.lexval** denotará o valor do símbolo **terminal num**.

As regras semânticas computam valores de atributos a partir de outros atributos, e isso estabelece uma dependência entre as regras, pois um atributo só pode ser calculado depois que todos os atributos dos quais ele depende tiverem sido calculados. Portanto, a ordem em que as regras de produção são usadas no processo de reconhecimento pode não ser necessariamente a

mesma ordem em que as ações semânticas correspondentes são executadas. A ordenação correta para realizar os cálculos dos atributos é representada através de um grafo dirigido, chamado **grafo de dependência**. Esse grafo determina uma sequência de avaliação (ou escalonamento) para as regras semânticas.

Um esquema de tradução sempre pode ser implementado através dos seguintes três passos: análise da cadeia de **tokens** acompanhada da construção da árvore de derivação; construção do grafo de dependência; finalmente, travessia do grafo de dependência para avaliação das regras semânticas associadas aos nós da árvore. Felizmente, os esquemas de tradução de interesse prático podem ser implementados num único passo, com a execução das ações semânticas durante a análise sintática, sem a necessidade de construir explicitamente a árvore de derivação, nem o grafo de dependência. Conforme será visto adiante, os esquemas S-atribuídos e os esquemas L-atribuídos possuem essa característica, já que os atributos dos símbolos podem ser calculados na mesma ordem em que as regras de produção são aplicadas no processo de reconhecimento da sentença.

A árvore de derivação que mostra os valores dos atributos associados a cada nó é chamada **árvore de derivação anotada**. Ao processo de computar os valores dos atributos dos nós denomina-se "anotar" ou "decorar" a árvore de derivação. Como a cada nó da árvore, com exceção das folhas, corresponde um símbolo não-terminal e uma regra de produção, então a cada nó irá corresponder zero ou mais ações.

Para cada ocorrência de um símbolo na árvore de derivação (o que corresponde ao uso de uma regra de produção no processo de análise), vai haver uma instância desse símbolo e de suas variáveis (seus atributos). Como um símbolo pode derivar mais de uma regra de produção, o mesmo símbolo pode originar diferentes ramificações na árvore de derivação (importante observar que o uso da regra correta é garantido pelo analisador, desde que a gramática seja não-ambígua). Isso significa que diferentes ações semânticas serão executadas, dependendo da regra de produção usada. Contudo, mesmo que o cálculo dos atributos de um símbolo possa variar em função da localização do símbolo na árvore, cada instância desse símbolo terá sempre a mesma configuração (mesmo conjunto de atributos). Isto é, o número de atributos e os seus tipos não variam em função do ponto em que o símbolo aparece na árvore de derivação.

O Exemplo 3.1 apresentou um esquema de tradução cujas ações semânticas estão especificadas à direita de cada produção. Conforme será visto adiante, isto significa que a ação vai ser executada após o reconhecimento da produção correspondente. O Exemplo 4.2,

abaixo, apresenta um esquema no qual as ações estão entremeadas com os símbolos gramaticais no lado direito das produções.

EXEMPLO 4.2 Esquema para traduzir expressões infixadas para pós-furadas.

$$\begin{aligned} E &\rightarrow T \ R \\ R &\rightarrow \text{Op } T \ (\text{print(Op.symbol)} \} \ R \mid \epsilon \\ T &\rightarrow \text{num } (\text{print(num.lexval)}) \end{aligned}$$

Supõe-se que os atributos "symbol" de "Op" (símbolo do operador) e "lexval" de "num" (valor léxico do número) tenham sido computados previamente.

4.1.1 Ordem de execução das ações semânticas

Pode-se visualizar a ordem em que as ações semânticas são executadas, desenhando-se a árvore de derivação e agregando-se as ações como se fossem folhas da árvore. Isso vale tanto para ações especificadas à direita das regras de produção, como para ações entremeadas com os símbolos gramaticais. A Figura 4.1 ilustra uma aplicação do esquema do exemplo anterior para a sentença de entrada $7 - 5 + 3$. A regra de ordenação é: "as ações são executadas quando as folhas correspondentes são visitadas usando a estratégia *depth-first*" (o algoritmo de caminhamento é apresentado a seguir). Quando a árvore é percorrida na ordem *depth-first*, as ações imprimem a expressão pós-fixada $7\ 5\ 3\ +$.

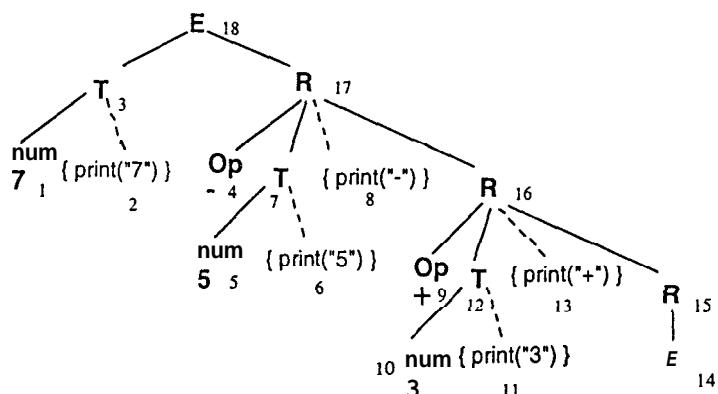


Figura 4.1 Árvore de derivação com numeração *depth-first* dos nós.

Estratégia *depth-first*:

A estratégia *depth-first* define uma forma de percorrer (visitar) os nós de uma árvore. Para percorrer uma árvore segundo essa estratégia, usa-se o seguinte procedimento:

```
DEPTH_FIRST(n: nodo);
  Para cada filho m de n, da esquerda para a direita, faça:
    DEPTH_FIRST(m);
    VISITE(n);
```

Observe que a visita a um nodo só acontece depois de todos os seus filhos terem sido visitados. A numeração dos nós na Figura 4.1 mostra como eles são visitados pelo procedimento DEPTH_FIRST.

4.1.2 Atributos sintetizados e atributos herdados

Num esquema de tradução, para cada produção $A \rightarrow a$, podem existir várias regras semânticas da forma $b := f(c_1, c_2, \dots, c_k)$, onde f é uma função, e b, c_1, c_2, \dots, c_k são atributos. Diz-se que:

- 1) b é um **atributo sintetizado** se ele é um atributo de A (símbolo do lado esquerdo da produção) e c_1, c_2, \dots, c_k são atributos associados aos símbolos do lado direito da produção (filhos de A);
- 2) b é um **atributo herdado** se ele é atributo de um dos símbolos do lado direito da produção. Nesse caso, c_1, c_2, \dots, c_k podem pertencer a quaisquer dos demais símbolos da produção (pai ou irmãos).

Em ambos os casos, diz-se que b depende dos atributos c_1, c_2, \dots, c_k .

4.1.3 Gramática de atributos

As ações semânticas podem produzir efeitos colaterais tais como imprimir um valor, armazenar um literal em uma tabela ou em um arquivo, atualizar uma variável global, etc. Quando o esquema de tradução não produz efeitos colaterais, ele é dito ser uma **gramática de atributos**. Nesse caso, as ações semânticas são atribuições ou funções envolvendo, **unicamente**, os atributos do esquema (não são usadas variáveis globais, arquivos, etc).

EXEMPLO 4.3 Gramática de atributos.

O exemplo a seguir é uma gramática de atributos que gera (ou reconhece) sentenças formadas por sequências de dígitos e obtém, no atributo *val* do símbolo raiz, o valor do somatório desses dígitos.

$$\begin{aligned} A &\rightarrow A_1 \text{ digit } \{ A.\text{val} := A_1.\text{val} + \text{digit.lexval} \} \\ A &\rightarrow \text{digit } \{ A.\text{val} := \text{digit.lexval} \} \end{aligned}$$

A árvore de derivação correspondente à sequência de *tokens* 3 4 5 é mostrada na Figura 4.2. Assume-se que *digit.lexval* contenha o valor numérico correspondente ao símbolo terminal *digit*.

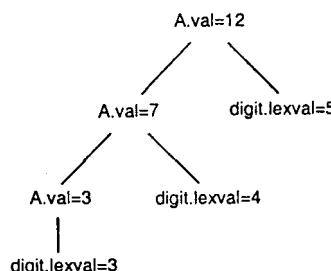


Figura 4.2 Árvore de derivação anotada para a sentença 3 4 5.

Observação:

Quando o mesmo símbolo ocorre mais de uma vez numa mesma regra de produção, é necessário usar índices para distinguir essas ocorrências. Isso permite identificar as instâncias dos símbolos gramaticais a que se referem os atributos. Por exemplo, na ação correspondente à primeira regra do exemplo anterior, *A.val* representa o atributo *val* da instância pai enquanto *A₁.val* representa o atributo *val* da instância filho (o atributo da instância pai é calculado em função do atributo da instância filho).

EXEMPLO 3.4 Esquema de tradução para um calculadora aritmética.

Este exemplo especifica um interpretador para uma calculadora, o qual reconhece uma expressão aritmética e imprime o seu valor. A expressão é formada por dígitos decimais, parênteses, operadores "+" e "*", e é seguida pelo token "=" (por exemplo, 2*(3+4)=).

| <u>Produções</u> | <u>Regras Semânticas</u> |
|------------------------------|--|
| $L \rightarrow E =$ | { print (E.val) } |
| $E \rightarrow E_1 + T$ | { E.val := E ₁ .val + T.val } |
| $E \rightarrow T$ | { E.val := T.val } |
| $T \rightarrow T_1 * F$ | { T.val := T ₁ .val * F.val } |
| $T \rightarrow F$ | { T.val := F.val } |
| $F \rightarrow (E)$ | { F.val := E.val } |
| $F \rightarrow \text{digit}$ | { F.val := digit.lexval } |

O esquema associa um atributo sintetizado chamado *val* a cada símbolo não-terminal E, T, e F. O símbolo terminal *digit* tem um atributo sintetizado *lexval* cujo valor é fornecido pelo analisador léxico. A regra semântica associada à produção "L → E =" é uma chamada de subrotina que imprime o valor da expressão aritmética reconhecida.

Este exemplo deixa de ser uma gramática de atributos porque, na primeira regra de produção, é feita uma chamada à função "print(E.val)", a qual produz efeitos colaterais. Todas as demais ações envolvem apenas atributos de símbolos gramaticais.

A Figura 4.3 mostra a árvore de derivação correspondente à sequência de *tokens* 2*3+4=. O resultado que é impresso, quando é feita a redução para o símbolo inicial da gramática, é o valor E.val do filho à esquerda da raiz.

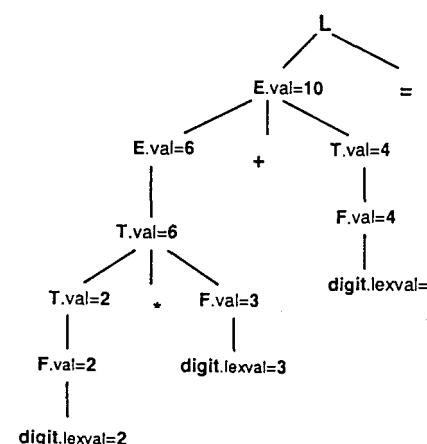


Figura 4.3 Árvore de derivação anotada para a expressão 2*3+4=.

Observação:

Num esquema de tradução, os símbolos terminais possuem apenas atributos sintetizados, os quais são calculados pelo analisador léxico. No outro extremo da árvore (na raiz da árvore), ocorre algo semelhante, pois o símbolo inicial da gramática não pode ter atributos herdados (ele não tem de quem herdar).

4.1.4 Tipos de esquemas de tradução

Todos os exemplos considerados até o momento, com exceção do Exemplo 4.2, operam apenas com atributos sintetizados, pois os atributos de cada nó dependem apenas dos atributos dos filhos desse nó. No Exemplo 4.2, a ação `{print(Op.simb)}` usa um atributo de `Op` que é "irmão" dessa ação. Uma árvore de derivação para um esquema que usa apenas atributos sintetizados pode sempre ser anotada (ou decorada) pela avaliação bottom-up dos atributos de cada nó, num único passo. Isso é equivalente a dizer que as ações semânticas podem ser executadas concomitantemente com a análise bottom-up, num único passo.

Esquemas S-atribuídos

Um esquema de tradução é *S-atribuído* se ele opera apenas com atributos sintetizados. Conforme será visto adiante, esses esquemas são implementados através de extensões de analisadores redutivos (*bottom-up*).

Os esquemas S-atribuídos apresentam suas ações semânticas à direita das regras de produção (isto é, as ações são colocadas após a forma sentencial que constitui o lado direito da produção). Um esquema deixa de ser S-atribuído quando ele inclui pelo menos um atributo herdado.

Conforme já vimos, um atributo de um símbolo X é herdado quando ele é avaliado em termos dos atributos do símbolo-pai e/ou dos símbolos-irmãos de X. O uso de atributos deste tipo permite passar informações adiante, de pai para filho e de irmão para irmão, na árvore de derivação. Desse modo, é possível especificar a dependência de um símbolo em relação às estruturas que o rodeiam (isto é, estruturas que estão no mesmo nível ou em um nível superior da árvore de derivação), estruturas estas que correspondem a símbolos irmãos ou ancestrais. Portanto, os atributos herdados são convenientes para expressar a dependência entre uma construção de linguagem e o contexto em que ela aparece. Esses atributos são usados quando é preciso passar adiante (no sentido da raiz para as folhas) os atributos de algum símbolo.

EXEMPLO 4.5 Esquema com atributos herdados.

O esquema de tradução a seguir usa atributos herdados para passar adiante, para cada elemento de uma lista de identificadores, o *tipo* declarado imediatamente antes (à esquerda) dessa lista.

| Producões | Regras Semânticas |
|------------------------------------|---|
| (1) $D \rightarrow T\ L$ | $\{ L.t := T.tipo \}$ |
| (2) $T \leftarrow \text{int}$ | $(T.tipo := \text{inteiro})$ |
| (3) $T \rightarrow \text{real}$ | $(T.tipo := \text{real})$ |
| (4) $L \rightarrow \text{id}, L_1$ | $\{ \text{adtipo(id.\$índice, L.t); } \\ L_1.t := L.t \}$ |
| (5) $L \rightarrow \text{id}$ | $\{ \text{adtipo(id.\$índice, L.t)} \}$ |

O não-terminal `T` tem um atributo sintetizado *tipo*, cujo valor é determinado a partir das palavras chaves `int` ou `real` nas produções 2 e 3. O valor desse atributo é passado para o atributo herdado `t` de `L` através da produção 1. A partir daí, as regras passam esse tipo para baixo, na árvore de derivação, através do atributo herdado `t` (cada filho `L1` recebe o valor do tipo diretamente do seu pai `L`). Regras associadas às produções de `L` chamam a rotina "adtipo" para incluir o tipo de cada identificador na tabela de símbolos. O terminal `id` possui o atributo `\$índice` que contém o índice da tabela de símbolos em que está armazenado o identificador. A Figura 4.4 mostra uma árvore anotada para a sentença `real a, b, c`.

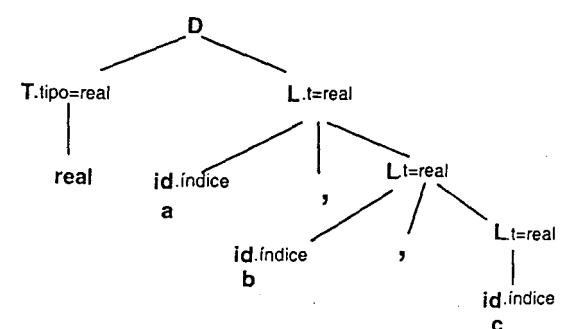


Figura 4.4 Árvore de derivação anotada para a sentença `real a, b, c`.

Esquemas com atributos herdados são naturalmente associados à analisadores *top-down*. Contudo, assim como é possível transformar uma gramática, de modo a torná-la apropriada para um determinado tipo de análise (por exemplo, transformar uma gramática LR

numa gramática equivalente LL), sempre é possível reescrever um esquema de tradução de modo a usar apenas atributos sintetizados.

O uso de atributos herdados, infelizmente, pode fazer com que um esquema de tradução deixe de poder ser implementado num único passo. Para um esquema de tradução com atributos herdados poder ser implementado num único passo (juntamente com um analisador *top-down*), é necessário que as ações semânticas satisfaçam algumas restrições.

Esquemas L-atribuídos

Os esquemas L-atribuídos restringem o uso de atributos herdados, de modo a permitir que as ações semânticas possam ser executadas durante a análise sintática, num único passo. Basicamente, a restrição imposta é: “para um símbolo X no lado direito de uma regra de produção, a ação que calcula um atributo herdado de X deve aparecer à esquerda de X”. Por exemplo, as regras 1 e 4 do esquema de tradução anterior fazem com que o esquema deixe de ser L-atribuído. Para satisfazer a restrição, a regra 1 teria que ser:

$$D \rightarrow T \quad \{ L.t := T.tipo \} \quad L$$

Os esquemas de tradução L-atribuídos serão estudados na seção 4.7. Para os esquemas com atributos herdados que não são L-atribuídos, é necessário obter a relação de dependência entre os atributos dos símbolos gramaticais. Isso é conseguido através da construção do grafo de dependência, assunto que será tratado a seguir.

4.2 Grafos de dependência

As dependências entre os valores dos atributos herdados e/ou sintetizados dos nós de uma árvore de derivação podem ser representadas num grafo denominado *grafo de dependência*.

Antes de construir o grafo de dependência, é necessário modificar as chamadas de rotinas, para que elas passem a ter a forma de chamadas de função. Assim, uma ação semântica do tipo $f(c_1, c_2, \dots, c_k)$ associada a uma produção $A \rightarrow \alpha$ passa a ter a forma:

$$b := f(c_1, c_2, \dots, c_k)$$

onde b é um atributo sintetizado (criado artificialmente) associado ao símbolo A (lado esquerdo da produção).

O grafo de dependência deve conter um nó para cada atributo e terá um arco dirigido de c para b se o atributo b depender do atributo c . O grafo é construído como segue:

Para cada nó N da árvore de derivação faça:

para cada atributo a de N faça: /“atributos artificiais já foram criados, se necessário”/
construa um nó para representar o atributo a .

Para cada nó N da árvore de derivação faça:

para cada regra semântica $b := f(c_1, c_2, \dots, c_k)$ associada ao nó N faça:
para $i := 1$ até k faça:
construa um arco do nó c_i para o nó b .

EXEMPLO 4.6 Atributo sintetizado e atributo herdado num grafo de dependência.

A Figura 4.5 mostra o grafo de dependência para o atributo sintetizado s de A , e para o atributo herdado h de V , considerando as regras de produção abaixo.

| <u>Produção</u> | <u>Regra Semântica</u> |
|-----------------------|----------------------------|
| $A \rightarrow X \ Y$ | $\{ A.s := f(X.x, Y.y) \}$ |
| $B \rightarrow V \ W$ | $\{ V.h := g(B.b, W.w) \}$ |

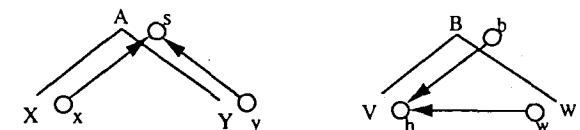


Figura 4.5 Grafos de dependência

EXEMPLO 4.7 Grafo de dependência para um esquema de tradução.

O grafo de dependência correspondente ao Exemplo 4.5 (cuja árvore de derivação anotada é mostrada na Figura 4.4) é mostrado na Figura 4.6, na qual os nós do grafo estão identificados por números. O arco 4→5 indica que o atributo t de L depende do atributo $tipo$ de T (de acordo com a regra semântica $L.t := T.tipo$ associada à produção $D \rightarrow T \ L$). A regra semântica $adtipo(id.indice, L.i)$, associada às produções de L , origina os atributos virtuais b representados pelos nós 6, 8 e 10.

A avaliação de um atributo qualquer $a := f(c_1, c_2, \dots, c_k)$ de um nó pode ser feita somente depois de terem sido computados os atributos c_1, c_2, \dots, c_k dos quais a depende. Uma *ordenação topológica* de um grafo dirigido e acíclico com k nós é qualquer ordenação n_1, n_2, \dots, n_k dos nós do grafo tal que o sentido dos arcos é sempre de nós que aparecem mais cedo

para nós que aparecem mais tarde na ordenação. Isto é, se $n_i \rightarrow n_j$ é um arco, então n_i aparece à esquerda de n_j na ordenação.

Qualquer ordenação topológica de um grafo de dependência serve como ordem de avaliação para as ações semânticas. Isto é, numa ordenação topológica, os atributos c_1, c_2, \dots, c_k de uma regra semântica $b := f(c_1, c_2, \dots, c_k)$ são sempre avaliados antes da função f . Para o grafo de dependência da Figura 4.6, a listagem dos nós em ordem crescente de numeração é uma ordenação topológica possível para avaliação dos atributos.

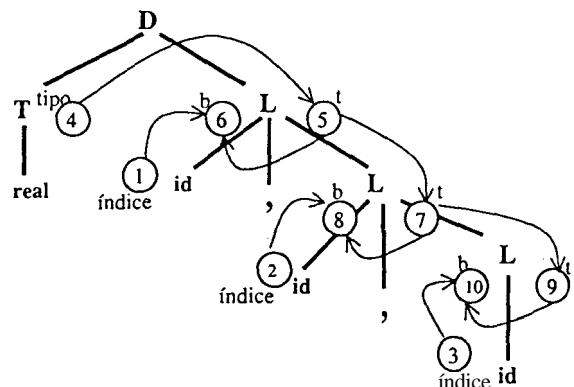


Figura 4.6 Grafo de dependência para a sentença **real id, id, id**.

4.3 Árvores de Sintaxe

A árvore de **sintaxe** é uma forma condensada da árvore de derivação, na qual somente os operandos da linguagem aparecem como folhas; os operadores passam a ser nós interiores da árvore. Outra vantagem dessa representação intermediária é que desaparecem as cadeias de produções simples tais como $A \rightarrow B$, $B \rightarrow C$.

A Figura 4.7 mostra a árvore de derivação e a árvore de sintaxe para a expressão **2*3+4**. Para a construção de uma árvore de sintaxe, um nó de operador deve possuir três campos: um para identificar o operador e outros dois para indicar os nós de **operandos**. Podem existir campos adicionais para outros atributos associados aos nós. se for o caso.

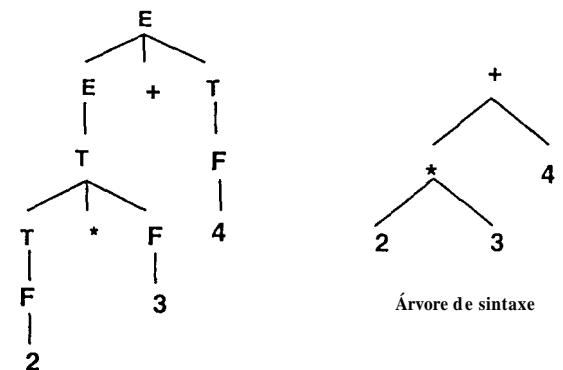


Figura 4.7 Árvore de derivação e árvore de sintaxe para a sentença **2*3+4**.

O uso de árvores de sintaxe como representação intermediária da sentença de entrada permite que a tradução para código objeto seja desligada da análise.

EXEMPLO 4.8 Cotisiruão da árvore de **sintaxe**.

Este exemplo ilustra um esquema de tradução S-atribuído que gera a árvore de sintaxe para uma expressão aritmética simplificada.

| <u>Produções</u> | <u>Reeras Semânticas</u> |
|-----------------------------|--|
| $E \rightarrow E_1 + T$ | { $E.\text{ptr} := \text{geranodo}("+", E_1.\text{ptr}, T.\text{ptr})$ } |
| $E \rightarrow E_1 \cdot T$ | { $E.\text{ptr} := \text{geranodo}("\cdot", E_1.\text{ptr}, T.\text{ptr})$ } |
| $E + T$ | { $E.\text{ptr} := T.\text{ptr}$ } |
| $T \rightarrow (E)$ | { $T.\text{ptr} := E.\text{ptr}$ } |
| $T \rightarrow \text{id}$ | { $T.\text{ptr} := \text{gerafolha}("id", id.\text{nome})$ } |
| $T \rightarrow \text{num}$ | { $T.\text{ptr} := \text{gerafolha}("num", num.\text{lexval})$ } |

Os atributos **ptr** associados aos não-terminais **E** e **T** guardam os ponteiros retomados pelas chamadas das funções. As funções que geram as folhas de operandos e os nós de operadores são as seguintes:

- 1) **gerafolha("id", nome)** – cria um nó de identificador com dois campos: rótulo “id” e nome do identificador.
- 2) **gerafolha("num", valor)** – cria um nó de número com dois campos: rótulo “num” e valor do número.

- 3) geranodo(*op*, esq, dir) - cria um nó de operador com rótulo “*op*” e dois campos contendo ponteiros para as subárvores esquerda e direita.

EXEMPLO 4.9 Árvore de derivação e árvore de sintaxe.

A Figura 4.8 mostra a árvore de derivação anotada e a correspondente árvore de sintaxe para a expressão $x + 5 - y$.

A árvore de derivação é construída no sentido *bottom-up*. Considerando a árvore já pronta e seus nós sendo visitados na ordem *depth-first*, então a ordem em que os nós serão visitados é a mesma ordem em que eles foram construídos durante a análise. Vale observar que a árvore de sintaxe é realmente construída, enquanto a árvore de derivação pode não ser explicitamente construída.

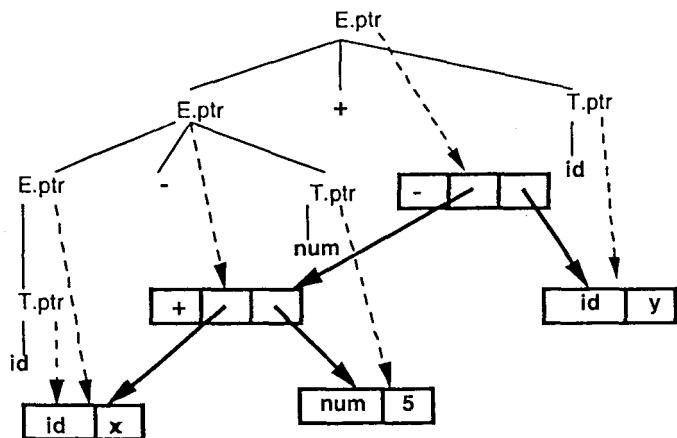


Figura 4.8 Árvore de derivação anotada e árvore de sintaxe para a expressão $x + 5 - y$.

4.4 Implementação de Esquemas S-Atribuídos

Os esquemas de tradução que usam apenas atributos sintetizados são adequados para processamento *bottom-up*. Esses esquemas podem ser implementados facilmente sobre qualquer analisador reduutivo.

Os atributos sintetizados podem ser avaliados à medida que a cadeia de entrada é reconhecida pelo analisador, conforme será mostrado a seguir. O analisador consegue manter os valores dos atributos associados aos símbolos da gramática na sua própria pilha. Sempre

que ocorre uma redução, são computados os atributos do símbolo do lado esquerdo da produção. a partir dos atributos que estão na pilha.

Por exemplo. seja o esquema de tradução:

$$A \rightarrow X Y Z \quad (A.a := f(X.x, Y.y, Z.z))$$

Quando o lado direito da produção $X Y Z$ estiver na pilha, ocasião em que será feita a redução para A . também estarão na pilha os valores de $X.x, Y.y, Z.z$. Nesse momento, o atributo $A.a$ poderá ser computado. A implementação pode ser feita, aumentando a pilha com uma nova coluna para armazenar os valores dos atributos, conforme é mostrado na Figura 4.9. Por ocasião da redução de $X Y Z$, os valores de x, y e z estarão na pilha **Atributo**, nas posições [topo], [topo-1] e [topo-2], respectivamente. Se um símbolo não tem atributos. então a entrada correspondente na pilha **Atributo** é vazia. Após a redução, o apontador topo é decrementado de 7. o símbolo A é armazenado na pilha em **Análise[topo]**, e o valor do atributo sintetizado $A.a$ é armazenado em **Atributo[topo]**.

| PILHA | |
|---------|----------|
| Análise | Atributo |
| Z | z |
| Y | Y |
| X | x |
| ... | ... |

↔ topo

Figura 4.9 Duplicação da pilha do analisador reduutivo

EXEMPLO 4.10 Analisador reduutivo estendido com ações semânticas.

Considere a definição da calculadora do Exemplo 4.4. Os atributos sintetizados mostrados na árvore da Figura 3.3 podem ser avaliados facilmente por um analisador tipo empilha-reduz. Para isso, o analisador deve ser modificado para executar o código abaixo por ocasião das operações de redução. O código foi obtido a partir das regras semânticas, substituindo cada atributo por uma posição no vetor **Atributo**.

| Produções | Código |
|------------------------------|--|
| $L \rightarrow E =$ | <code>print(Atributo[topo])</code> |
| $E \rightarrow E_1 + T$ | $Atributo[ntopo] := Atributo[topo-2] + Atributo[topo]$ |
| $E \rightarrow T$ | |
| $T \rightarrow T_1 * F$ | $Atributo[ntopo] := Atributo[topo-2] * Atributo[topo]$ |
| $T \rightarrow F$ | |
| $F \rightarrow (E)$ | $Atributo[ntopo] := Atributo[topo-1]$ |
| $F \rightarrow \text{digit}$ | $Atributo[topo] := \text{digit.lexval}$ |

Quando o analisador faz a redução $F \leftarrow \text{digit}$, o valor do *token digit* é armazenado em $Atributo[topo]$. O código não mostra como as variáveis *topo* e *ntopo* são operadas. Quando uma produção com n símbolos do lado direito é reduzida, o valor de *ntopo* é atualizado para $\text{topo}-(n-1)$. Após a execução de cada ação, *topo* é atualizado para *ntopo*. A tabela abaixo mostra a sequência de movimentos realizada na análise da sentença $2*3+4 =$.

| Entrada | Análise | Atributo | Produção |
|---------------|---------|----------|------------------------------|
| $2 * 3 + 4 =$ | \$ | | |
| $* 3 + 4 =$ | \$2 | | |
| $* 3 + 4 =$ | \$F | 2 | $F \rightarrow \text{digit}$ |
| $* 3 + 4 =$ | ST | 2 | $T \rightarrow F$ |
| $3 + 4 =$ | ST * | 2 _ | |
| $+ 4 =$ | ST * 3 | 2 _ | |
| $+ 4 =$ | ST * F | 2 _ 3 | $F \rightarrow \text{digit}$ |
| $+ 4 =$ | ST | 6 | $T \rightarrow T * F$ |
| $+ 4 =$ | SE | 6 | $E \rightarrow T$ |
| $4 =$ | SE + | 6 _ | |
| $=$ | SE + 4 | 6 _ | |
| $=$ | SE + F | 6 _ 4 | $F \rightarrow \text{digit}$ |
| $=$ | SE + T | 6 _ 4 | $T \rightarrow F$ |
| $=$ | SE | 10 | $E \rightarrow E + T$ |
| | SL | 10 | $L \rightarrow E =$ |

4.5 Esquemas de Tradução L–Atribuídos

Um esquema de tradução é dito *L–atribuído* se cada atributo de X_j , $1 \leq j \leq n$, no lado direito de $A \rightarrow X_1 X_2 \dots X_n$ (portanto atributo herdado) depende somente:

- 1) dos atributos dos símbolos X_1, X_2, \dots, X_{j-1} à esquerda de X_j na produção, e
- 2) dos atributos herdados de A .

Note que todo esquema S–atribuído é necessariamente L–atribuído, pois as restrições acima aplicam-se somente a atributos herdados.

EXEMPLO 4.11 Esquema não L–atribuído.

O esquema abaixo não é L–atribuído porque o atributo herdado $D.h$ depende do atributo $E.s$, e E está à direita de D .

| Produção | Remas Semânticas |
|---------------------|---|
| $A \rightarrow B C$ | { $B.h := f_1(A.h)$; $C.h := f_2(B.s)$; $A.s := f_3(C.s)$ } |
| $A \rightarrow D E$ | { $E.h := f_4(A.h)$; $D.h := f_5(E.s)$; $A.s := f_6(D.s)$ } |

Os esquemas de tradução L–atribuídos são mais flexíveis que os S–atribuídos, pois permitem usar atributos herdados e sintetizados e, além disso, permitem a avaliação dos atributos simultaneamente com a obtenção da árvore de derivação. Na Seção 4.6, será apresentado um método para implementar esquemas L–atribuídos através da extensão de um analisador descendente recursivo preditivo.

Conforme visto anteriormente, os reconhecedores descendentes (*top-down*) e os redutivos (*bottom-up*) constroem a árvore de derivação de forma diferente. Os descendentes constroem da raiz para as folhas e os redutivos, das folhas para a raiz. Embora os esquemas L–atribuídos sejam implementados sobre reconhecedores descendentes, e os S–atribuídos, sobre reconhecedores redutivos, é interessante notar que, em ambos os tipos de esquema, as ações semânticas são executadas (e os atributos são avaliados) na ordem *depth-first*. Isto é, se olharmos a árvore de derivação já pronta (com as ações semânticas agregadas à mesma) e caminharmos sobre a árvore, usando a estratégia *depth-first*, veremos que as ações terão sido executadas nos momentos em que essas ações são "visitadas".

Para a avaliação de atributos de um esquema de tradução L-atribuído (ou S-atribuído), usa-se o seguinte algoritmo:

Algoritmo para avaliação de atributos:

DEPTH FIRST(n ; node);

Para cada filho m de n , da esquerda para a direita, faça:

Calcule os atributos herdados de **m**:

DEPTH_FIRST(m);

Calcule os atributos sintetizados de n .

Neste momento, é interessante que o leitor revise o Exemplo 4.2 e a Figura 4.1. O exemplo apresenta um esquema de tradução que transforma expressões infixadas em pós-fixadas. A Figura 4.1 mostra a árvore de derivação anotada e a ordenação das ações semânticas, na qual se vê que as ações são executadas na ordem *depth-first*. Mais exemplos de esquemas L-atribuídos serão vistos adiante. A seguir, são apresentados os cuidados a serem tomados na especificação de ações semânticas, para que as mesmas possam ser executadas simultaneamente à construção da árvore de derivação, num único passo.

Cuidados na ordenação das ações semânticas

Na definição de um esquema de tradução, deve-se assegurar que os valores dos atributos já tenham sido **calculados** quando as ações referirem-se a eles. Quando são usados apenas atributos **sintetizados**, consegue-se isso colocando as ações semânticas **após** o lado direito da produção. Se existem ambos, atributos herdados e sintetizados, deve-se proceder da seguinte forma:

- 1) atributo herdado associado a símbolo do lado direito da produção deve ser computado em ação semântica especificada antes (à esquerda) desse símbolo;
 - 2) uma ação semântica só pode se referir a atributo sintetizado de símbolo que apareça à esquerda dessa ação (considerando a árvore de derivação e o caminhamento *depth-first*, é fácil constatar que esse atributo já estará calculado por ocasião da execução da ação);
 - 3) atributo sintetizado, associado ao não-terminal do lado esquerdo da produção, deve ser computado somente depois de serem computados todos os atributos que ele **referencia** (isso é garantido quando se coloca, no final da produção, a ação que computa **esse** atributo).

As definições L-atribuídas satisfazem automaticamente os três requisitos acima.

EXEMPLO 4.12. Esquema de tradução não implementável num único passo.

O esquema de tradução a seguir não satisfaz o primeiro dos requisitos acima, pois as ações que calculam os atributos herdados de A1 e A2 estão à direita dos símbolos correspondentes.

| | |
|------------------------|----------------------------|
| $S \rightarrow A1\ A2$ | { $A1.h := 1; A2.h := 2$ } |
| $A \rightarrow a$ | { $\text{print}(A.h)$ } |

O atributo herdado $A.h$ na segunda produção ainda não estará definido quando uma tentativa de imprimir seu valor for feita durante uma travessia *depth-first*. Isso fica claro na Figura 4.10, que é a árvore de derivação para a sentença "aa". Nessa árvore, a travessia visitará as subárvore A_1 e A_2 antes que os valores de $A_1.h$ e $A_2.h$ tenham sido atribuídos.

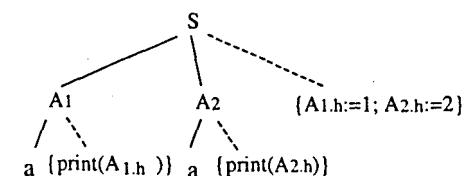


Figura 4.10 Árvore de derivação e ações para a sentença aa.

EXEMPLO 4.13 *Esquema* corrigido para execução em um passo.

Não haveria problema para executar as ações numa travessia *depth-first* se as produções de S, no exemplo anterior, fossem as seguintes:

```

S → {A1.h:=1} A1 {A2.h:=2} A2
A → a { print(A.h) }

```

Nesse caso, a árvore de derivação seria a mostrada na Figura 4.11.

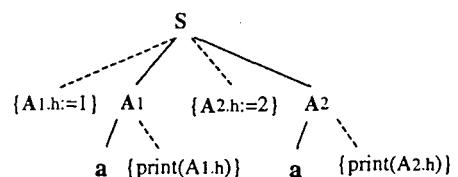


Figura 4.11 Nova árvore de derivação com ações para a sentença aa.

4.6 Implementação de Esquemas L–Atribuídos

Esta seção aborda a implementação de esquemas de tradução que atuam durante a análise descendente (*top-down*). Como os analisadores *top-down* não admitem recursividade à esquerda, ver-se-á inicialmente como um esquema de tradução com recursividade à esquerda e atributos sintetizados pode ser adaptado para a análise *top-down*. A transformação das regras de produção obriga a fazer transformações nas ações semânticas associadas às regras.

EXEMPLO 4.14 Transformação de esquema de tradução "bottom-up" para "top-down".

Seja a gramática de atributos abaixo, a qual reconhece sequências de dígitos e obtém o somatório desses dígitos no atributo *val* do símbolo raiz:

$$\begin{aligned} A &\rightarrow A_1 \text{ digit} & \{ A.\text{val} := A_1.\text{val} + \text{digit.lexval} \} \\ A &\rightarrow \text{digit} & (A.\text{val} := \text{digit.lexval}) \end{aligned}$$

A árvore de derivação anotada correspondente à sequência de tokens **3 4 5**, para a análise *bottom-up*, é mostrada na Figura 4.12, à esquerda. A transformação da gramática acima para eliminar a recursividade à esquerda implica no novo conjunto de regras:

$$\begin{aligned} A &\rightarrow \text{digit } (X.h := \text{digit.lexval}) \text{ } X \text{ } (A.\text{val} := X.s) \\ X &\rightarrow \text{digit } \{ X_1.h := X.h + \text{digit.lexval} \} \text{ } X_1 \text{ } \{ X.s := X_1.s \} \\ X &\rightarrow \epsilon \text{ } (X.s := X.h) \end{aligned}$$

A árvore de derivação anotada correspondente à sequência de tokens **3 4 5**, para o novo esquema de tradução, é mostrada na Figura 4.12, à direita.

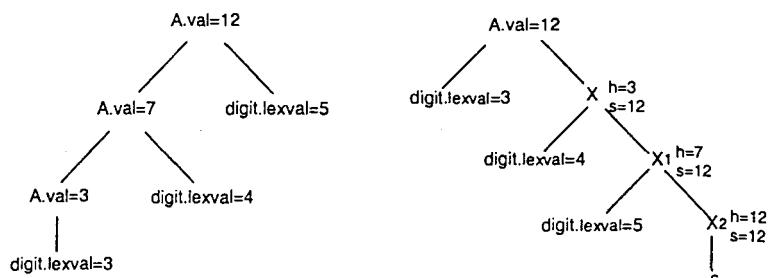


Figura 4.12 Árvores de derivação anotadas para a sentença 3 4 5.

Para deixar clara a ordem de avaliação dos atributos no esquema L–atribuído, a árvore de derivação com as ações agregadas é mostrada na Figura 4.13.

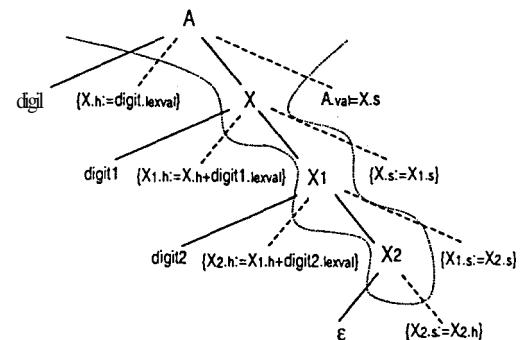


Figura 4.13 Árvore de derivação com ações agregadas

Considerando somente as folhas visitadas durante o percurso *depth-first*, tem-se a seguinte sequência:

$$\text{digit}; \{X.h := \text{digit.lexval}\}; \text{digit1}; \{X_1.h := X.h + \text{digit1.lexval}\}; \text{digit2}; \{X_2.h := X_1.h + \text{digit2.lexval}\}; \{X_2.s := X_2.h\}; \{X_1.s := X_2.s\}; \{X.s := X_1.s\}; \{A.\text{val} := X.s\}$$

na qual a primeira linha corresponde à "descida" na árvore, e a segunda linha corresponde à "subida". Observe que, na descida, são computados os atributos herdados. e. na subida. são computados os atributos sintetizados. É no nível mais baixo da árvore que o atributo herdado é passado para o atributo sintetizado.

Também é interessante considerar as configurações que a pilha assume durante o processo de análise. Inicialmente, a pilha contém apenas A, o símbolo inicial da gramática. A configuração $[\$ \ A ; 3 \ 4 \ 5 \ \$]$, isto é, A na pilha e 3 4 5 na fita de entrada (3 sob o cabeçote), faz com que o símbolo A seja substituído pelo lado direito da primeira regra de produção. A configuração do analisador passa a ser $[\$ \ \{A.val:=X.s\} \ X \ \{X.h:=digit.lexval\} digit ; 3 \ 4 \ 5 \ \$]$.

Neste momento tem-se digit no topo da pilha e também o token $digit=3$ na fita de entrada. Isso faz com que digit seja desempilhado e que o cabeçote de leitura avance. O topo da pilha passa a ser $\{X.h:=digit.lexval\}$; esta ação é executada e desempilhada, resultando o valor 3 para $X.h$. Fica no topo da pilha o símbolo X, para o qual o atributo h já foi calculado.

Considere $X^{(h=3)}$ representando X com atributo h igual a 3. A situação atual é X (já com atributo h calculado) no topo da pilha e $digit=4$ na fita de entrada. Isso faz com que X, na pilha, seja substituído pelo lado direito da segunda regra de produção, resultando a seguinte configuração: $[\$ \ \{A.val:=X.s\} \ \{X.s:=X1.s\} \ X1 \ \{X1.h:=X.h+digit.lexval\} digit ; 4 \ 5 \ \$]$.

Nesse ponto, tem-se digit no topo da pilha e $digit=4$ na fita de entrada, fazendo com que digit seja desempilhado e que o cabeçote de leitura avance. O topo da pilha, neste momento, é a ação $\{X1.h:=X.h+digit.lexval\}$, a qual é executada e desempilhada, resultando o valor 7 para $X1.h$. Isso significa que a segunda instância do símbolo X tem o atributo h valendo 7 (o que é representado por $X1^{(h=7)}$). Fica no topo da pilha a segunda instância do símbolo X.

Como a fita de entrada contém $digit=5$, isso faz com que X seja substituído, usando novamente a **segunda** regra de produção, o que resulta na seguinte configuração: $[\$ \ \{A.val:=X.s\} \ \{X.s:=X1.s\} \ \{X1.s:=X2.s\} \ X2 \ \{X2.h:=X1.h+digit.lexval\} digit ; 5 \ \$]$.

Nesse ponto, tem-se digit no topo da pilha e $digit=5$ na fita de entrada, fazendo com que digit seja desempilhado e que o cabeçote avance. Fica no topo da pilha a ação $\{X2.h:=X1.h+digit.lexval\}$, a qual é executada e desempilhada, resultando o valor 12 para $X2.h$. Isto é, a terceira instância do símbolo X tem o atributo h valendo 12 ($X2^{(h=12)}$). Agora o topo da pilha contém a terceira instância do símbolo X e a fita de entrada contém \$. Isso faz com que X seja substituído, usando a terceira regra de produção, resultando a seguinte configuração: $[\$ \ \{A.val:=X.s\} \ \{X.s:=X1.s\} \ \{X1.s:=X2.s\} \ \{X2.s:=X2.h\} ; \$]$. A partir daí, a pilha conterá apenas ações, as quais serão executadas e desempilhadas, urna a uma. O resultado final será a atribuição do valor 12 a A.val.

EXEMPLO 4.15 Generalização da transformação “bottom-up” para “top-down”.

Seja o seguinte esquema de tradução S-atribuído, no qual cada símbolo da gramática possui um atributo sintetizado:

$$\begin{array}{ll} A \rightarrow A_1 Y & \{ A.a := g(A_1.a, Y.y) \} \\ A \rightarrow X & \{ A.a := f(X.x) \} \end{array}$$

O algoritmo que elimina a recursividade à esquerda altera as regras de produção para:

$$\begin{array}{l} A \rightarrow X R \\ R \rightarrow Y R \mid \epsilon \end{array}$$

e, como consequência, as ações semânticas precisam ser modificadas. A transformação completa resulta em:

$$\begin{array}{l} A \rightarrow X \quad \{R.h := f(X.x)\} \quad R \quad \{A.a := R.s\} \\ R \rightarrow Y \quad \{R1.h := g(R.h, Y.y)\} \quad R1 \quad \{R.s := R1.s\} \\ R \rightarrow \epsilon \quad \{R.s := R.h\} \end{array}$$

O símbolo não-terminal introduzido contém um atributo herdado e um sintetizado ($R.h$ e $R.s$, respectivamente). A Figura 4.14 ilustra a transformação da forma sentencial XYR nos dois esquemas de tradução.

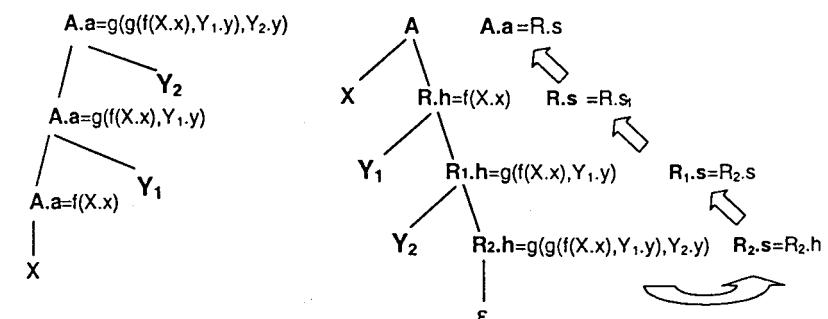


Figura 4.14 Transformação de esquema S-atribuído para L-atribuído

EXEMPLO 4.16 Esquema de tradução para avaliar expressões.

Seja o seguinte esquema de tradução S-atribuído que avalia adições:

$$\begin{array}{ll}
 E \rightarrow E_1 + T & \{E.val := E_1.val + T.val\} \\
 E \rightarrow E_1 - T & \{E.val := E_1.val - T.val\} \\
 E \rightarrow T & \{E.val := T.val\} \\
 T \rightarrow (E) & \{T.val := E.val\} \\
 T \rightarrow \text{num} & \{T.val := \text{num}.val\}
 \end{array}$$

Eliminada a recursividade à esquerda e realizadas as transformações correspondentes para a avaliação dos atributos, obtém-se o seguinte esquema:

$$\begin{array}{ll}
 E \rightarrow T \quad \{R.h := T.val\} \quad R \quad \{E.val := R.s\} \\
 R \rightarrow + T \quad \{R_1.h := R.h + T.val\} \quad R_1 \quad \{R.s := R_1.s\} \\
 R \rightarrow - T \quad \{R_1.h := R.h - T.val\} \quad R_1 \quad \{R.s := R_1.s\} \\
 R \rightarrow \epsilon \quad \{R.s := R.h\} \\
 T \rightarrow (E) \quad \{T.val := E.val\} \\
 T \rightarrow \text{num} \quad \{T.val := \text{num}.val\}
 \end{array}$$

Para a expressão $7-5+3$, o esquema transformado produz a árvore de derivação anotada mostrada na Figura 4.15. As setas mostram o caminho de avaliação da expressão, até ser atingido o nível mais baixo da árvore. Nesse nível, o valor do atributo herdado vai ser passado para o atributo sintetizado, e esse valor será **retornado** até o atributo *val* de *E*.

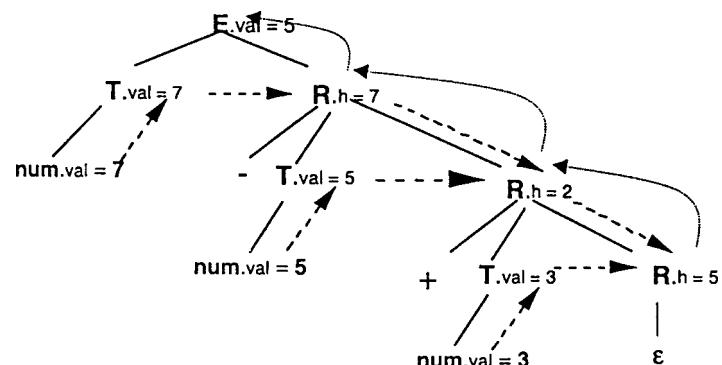


Figura 4.15 Árvore de derivação anotada para a expressão $7-5+3$.

Num esquema de tradução com análise *top-down*, uma ação pode ser executada imediatamente antes que um símbolo na mesma posição seja expandido. Assim, na segunda produção ($R \rightarrow + T \{R_1.h := R.h + T.val\} R_1, \{R.s := R_1.s\}$), a atribuição $\{R_1.h := R.h + T.val\}$ é executada antes de R_1 ser totalmente expandido para terminais, e a segunda ação $\{R.s := R_1.s\}$

é executada depois de R_1 ser expandido. Conforme já visto, para um esquema ser L-atribuído, um atributo herdado de um símbolo deve ser computado por uma ação que apareça antes desse símbolo, e um atributo sintetizado do não-terminal à esquerda deve ser computado depois que todos os atributos dos quais ele depende tenham sido computados.

EXEMPLO 4.17 Obtenção da árvore sintática com processamento top-down.

Este exemplo mostra a transformação do esquema de tradução definido no Exemplo 4.8 para processamento *top-down*.

Eliminando a recursividade à esquerda, o esquema transforma-se no seguinte:

$$\begin{array}{ll}
 E \rightarrow T \quad \{R.h := T.ptr\} \quad R \quad \{E.ptr := R.s\} \\
 R \rightarrow + T \quad \{R_1.h := \text{geranodo}(+, R.h, T.ptr)\} \quad R_1 \quad \{R.s := R_1.s\} \\
 R \rightarrow - T \quad \{R_1.h := \text{geranodo}(-, R.h, T.ptr)\} \quad R_1 \quad \{R.s := R_1.s\} \\
 R \rightarrow \epsilon \quad \{R.s := R.h\} \\
 T \rightarrow (E) \quad \{T.ptr := E.ptr\} \\
 T \rightarrow \text{id} \quad \{T.ptr := \text{gerafolha(id, id.indice)}\} \\
 T \rightarrow \text{num} \quad \{T.ptr := \text{gerafolha(num, num.val)}\}
 \end{array}$$

A Figura 4.16 mostra a árvore de sintaxe para a entrada $x-2+y$.

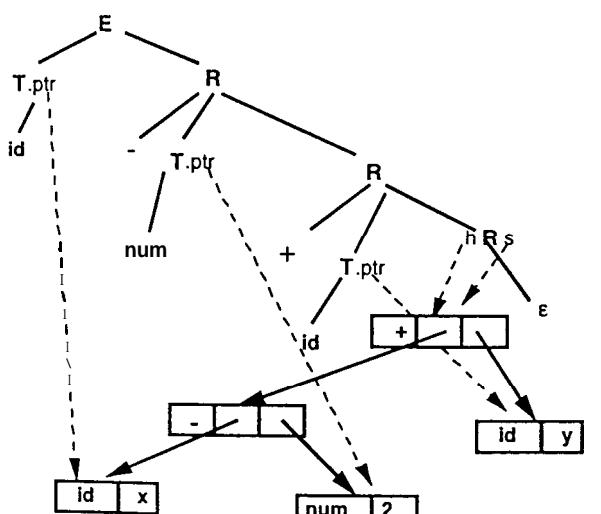


Figura 4.16 Árvore de sintaxe para a entrada $x-2+y$.

4.7 Projeto de um Tradutor Preditivo

Dada uma gramática livre do contexto apropriada para reconhecimento *top-down* e seu reconhecedor recursivo preditivo, pode-se estender facilmente as funções do reconhecedor de modo a incluir ações de um esquema de tradução.

Algoritmo: Construção de um tradutor preditivo dirigido por sintaxe.

Entrada: Um esquema de tradução (o esquema deve ser baseado numa gramática apropriada para análise preditiva).

Resultado: Código de um tradutor dirigido por sintaxe.

Método:

- 1) Para cada não-terminal A, construa uma função que tenha um parâmetro formal para cada atributo herdado de A e que **retorne** os valores dos atributos sintetizados de A. Para simplificar, assume-se que cada não-terminal tenha apenas um atributo sintetizado (dessa forma, o valor sintetizado pode retomar no nome da função). Além disso, a função para A deve ter uma variável local para cada atributo de símbolo gramatical que aparece nas produções de A (ver Exemplo 4.18).
- 2) O código para o não-terminal A deve decidir qual produção usar, baseado no símbolo corrente da entrada. Considerando o lado direito da produção a ser usada, da esquerda para a direita, o código associado deve **implementar** o seguinte (ver Exemplo 4.19):
 - i) para cada *token* X, verificar se X é o *token* lido e avançar a leitura;
 - ii) para cada não-terminal B, gerar uma atribuição $c := B(b_1, b_2, \dots, b_k)$, sendo b_1, b_2, \dots, b_k variáveis para os atributos herdados de B, e c a variável para o atributo sintetizado de B;
 - iii) para cada ação semântica, copiar o código correspondente, substituindo cada referência a atributo pela variável desse atributo.

EXEMPLO 4.18 Parâmetros e variáveis locais de uma função do analisador.

Considerando a produção:

$$E \rightarrow T \quad (R.h := T.ptr) \quad R \quad (E.ptr := R.s)$$

tem-se que E possui um atributo sintetizado, *ptr*, e nenhum atributo herdado. Tem-se ainda os seguintes atributos de símbolos gramaticais: *h* e *s* (atributos de R), e *ptr* (atributo de T). Portanto, a função E não terá parâmetros, retomando (no seu próprio nome) o valor de *ptr* e usará as seguintes variáveis locais: *h*, *ptr* (valor a ser retomado), *ptr1* (relativa a T) e *s*.

EXEMPLO 4.19 Construção de um tradutor preditivo dirigido por sintaxe.

Apresenta-se, a seguir, a programação do esquema de tradução do Exemplo 4.17. O analisador será formado por três funções: E, R e T. A partir dos atributos desses não-terminais, definem-se os seguintes tipos para os argumentos e resultados das funções E, R e T.

```
function E: ^nodo;
function R( h: ^nodo): ^nodo;
function T: ^nodo;
```

Os símbolos E e T possuem apenas um atributo sintetizado, *ptr*, que será o valor de retorno das funções correspondentes. O símbolo R tem dois atributos: *h* (herdado), declarado como parâmetro da função R, e *s* (sintetizado), que será o valor de retorno da função.

Considerando as produções de R (onde *op-a* representa operador de adição):

$$R \rightarrow op-a T \quad \{ R_1.h := geranodo(op_a.lexval, R.h, T.ptr) \} \quad R_1 \quad (R.s := R_1.s) \\ | \quad \epsilon \quad \{ R.s := R.h \}$$

tem-se que R terá um argumento *h*, retomará o valor de *s* e **usará** as seguintes variáveis locais (todas do tipo pointer): *hi*, *ptr*, *s*, *si*. Observe que *hi* é o atributo herdado *R.h*, *ptr* é o atributo sintetizado *T.ptr*, *s* é o atributo sintetizado de retorno, e *si* é o atributo sintetizado *R.s*.

Sabendo-se que o atributo *lexval* do token *op-a* pode ser “+” ou “-”, a função correspondente a R, **construída** de acordo com o algoritmo anterior, é a seguinte:

```
function R( h: ^ nodo ): ^ nodo;
var h1, ptr, s, si: ^ nodo;
begin
  if token = op-a
    then begin /* produção R → op-a T */
      token := LETOKEN;
      ptr := T();
      h1 := geranodo (op_a.lexval,h.ptr);
      si := R (h1);
      s := si
    end
    else s := h; /* produção R → ε */
    return s
  end;
```

As funções E e T podem ser construídas de forma similar e são deixadas como exercício.

EXERCÍCIOS

- 1) O que você entende por árvore de derivação, grafo de dependência e gramática de atributos? Dê exemplos das construções acima.
- 2) Dado o esquema de tradução abaixo, aplique-o à sentença **a+b*c** e apresente a árvore de derivação decorada (com os atributos avaliados) correspondente.

```

E → E1 + T { E.code := E1.code || T.code || "+" }
E → T { E.code := T.code }
T → T1 * F { T.code := T1.code || F.code || "*" }
T → F { T.code := F.code }
F → id { F.code = id.nome }

```

- 3) A gramática abaixo gera adições de constantes inteiras e reais. Quando dois inteiros são somados, o tipo resultante é inteiro; caso contrário, é real. Defina um esquema de tradução para determinar o tipo de cada subexpressão.

```

E → E + T | T
T → nreal | nint

```

- 4) Dado o esquema de tradução abaixo, aplique-o à sentença **3+4-5**, construa o grafo de dependência e identifique os atributos herdados e sintetizados.

```

E → T { R.e = T.val } R { E.val := R.c }
R → + T { R1.e := R.e + T.val } R1 { R.c := R1.c }
R → - T { R1.e := R.e - T.val } R1 { R.c := R1.c }
R → ε { R.c = R.e }
T → ( E ) { T.val := E.val }
T → num { T.val := num.val }

```

- 5) Em quais situações (dependência de atributos) é possível fazer análise sintática e avaliar atributos num único passo?
- 6) Dado o esquema de tradução abaixo, transforme-o num esquema de tradução que funcione conjuntamente com um analisador *top-down* (veja os Exemplos 4.14, 4.15, 4.16 e 4.17).

```

E → E1 + T { E.code := E1.code || T.code || "+" }
E → T { E.code := T.code }
T → T1 * F { T.code := T1.code || F.code || "*" }
T → F { T.code := F.code }
F → id { F.code = id.nome }

```

- 7) Construa árvores de derivação e de sintaxe para a expressão **a · (b + c)**:
 - a) de acordo com o esquema de tradução definido no Exemplo 4.8;
 - b) com base no esquema de tradução do Exemplo 4.17.
- 8) Escreva um tradutor descendente recursivo preditivo dirigido por sintaxe para o esquema de tradução definido no Exemplo 4.16 (use o modelo do Exemplo 4.19).
- 9) Complete o Exemplo 4.19.

5 Geração de Código Intermediário

A geração de código intermediário é a transformação da árvore de derivação em um segmento de código. Esse código pode, eventualmente, ser o código objeto final, mas, na maioria das vezes, constitui-se num código intermediário, pois a tradução de código fonte para objeto em mais de um passo apresenta algumas vantagens:

- possibilita a otimização do código intermediário, de modo a obter-se o código objeto final mais eficiente;
- simplifica a implementação do compilador, resolvendo, gradativamente, as dificuldades da passagem de código fonte para objeto (alto-nível para baixo-nível), já que o código fonte pode ser visto como um texto condensado que "explode" em inúmeras instruções elementares de baixo nível;
- possibilita a tradução de código intermediário para diversas máquinas.

A desvantagem de gerar código intermediário é que o compilador requer um passo a mais. A tradução direta do código fonte para objeto leva a uma compilação mais rápida.

A grande diferença entre o código intermediário e o código objeto final é que o intermediário não especifica detalhes da máquina alvo, tais como quais registradores serão usados, quais endereços de memória serão referenciados, etc.

5.1 Linguagens Intermediárias

Os vários tipos de código intermediário fazem parte de uma das seguintes categorias:

- representações gráficas: árvore e grafo de sintaxe;
- notação pós-fixada ou pré-fixada;
- código de três-endereços: quádruplas e triplas.

5.1.1 Árvore e Grafo de Sintaxe

Uma árvore de sintaxe é uma forma condensada de árvore de derivação na qual somente os operandos da linguagem aparecem como folhas; os operadores constituem nós interiores da árvore. Outra simplificação da árvore de sintaxe é que cadeias de produções simples (por exemplo, $A \rightarrow B, B \rightarrow C$) são eliminadas.

Um grafo de sintaxe, além de incluir as simplificações da árvore de sintaxe, faz a fatoração das subexpressões comuns, eliminando-as, conforme é ilustrado na Figura 5.1.

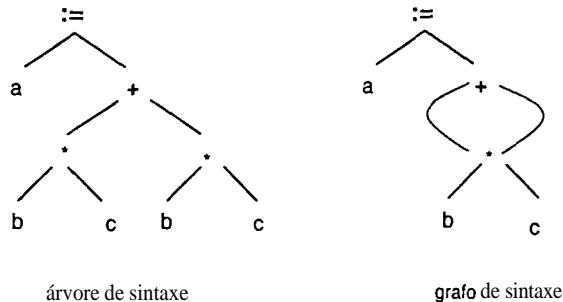


Figura 5.1 Árvore de sintaxe e grafo de sintaxe para $a := (b * c) + (b * c)$.

No Exemplo 4.8, foi visto um esquema de tradução que obtém a árvore de sintaxe para uma expressão. Esse mesmo esquema pode ser usado para obter o grafo de sintaxe. Basta modificar as funções que geram nós e folhas para que elas verifiquem, antes de construir um nó, se já existe algum nó idêntico a esse. Por exemplo, antes de construir um novo nó com rótulo *op* e ponteiros *ptr1* e *ptr2* para subárvores que representam subexpressões, a função *geranodo* verifica se já existe algum nó com rótulo *op* e ponteiros que apontem para árvores idênticas às apontadas por *ptr1* e *ptr2*. Caso positivo, a função apenas retoma um ponteiro para o nó previamente construído. A função *gerafolha* age de forma similar.

5.1.2 Notações Pós-fixada e Pré-fixada

Se E_1 e E_2 são expressões pós-fixadas e q é um operador binário, então, " $E_1 E_2 q$ " é a representação pós-fixada para a expressão " $E_1 q E_2$ ". Se, por outro lado, E_1 e E_2 são expressões pré-fixadas, então, " $q E_1 E_2$ " é a representação pré-fixada para a expressão " $E_1 q E_2$ ". Alguns exemplos são mostrados na tabela da Figura 5.2.

Notações pós e pré-fixadas podem ser generalizadas para operadores *n*-ários. Para a avaliação de expressões pós-fixadas, pode-se utilizar uma pilha e um processo que age do seguinte modo: lê a expressão da esquerda para a direita, empilhando cada operando até encontrar um operador. Encontrando um operador *n*-ário, aplica o operador aos *n* operandos do topo da pilha. Processamento semelhante pode ser aplicado para avaliação de expressões pré-fixadas; nesse caso, a expressão é lida da direita para a esquerda.

| Notação | | |
|-----------|------------|------------|
| infixada | pós-fixada | pré-fixada |
| $(a+b)*c$ | $ab+c^*$ | $*+abc$ |
| $a*(b+c)$ | $abc+^*$ | $*a+bc$ |
| $a+b*c$ | abc^*+ | $+a*bc$ |

Figura 5.2 Notações pós e pré fixadas

EXEMPLO 5.1 Esquema de tradução para gerar representação pós-fixada.

$$\begin{array}{ll} E \rightarrow E_1 + T & \{ E.cod = E_1.cod \parallel T.cod \parallel "+" \} \\ E \rightarrow T & (E.cod = T.cod) \\ T \rightarrow T_1 * F & \{ T.cod = T_1.cod \parallel F.cod \parallel "*" \} \\ T \rightarrow F & (T.cod = F.cod) \\ F \rightarrow id & (F.cod = id.nome) \end{array}$$

5.1.3 Código de Três-Endereços

No código intermediário de três-endereços, cada instrução faz referência, no máximo, a três variáveis (endereços de memória). As instruções dessa linguagem intermediária são as seguintes:

```
A := B op C
A := op B
A := B
goto L
if A oprel B goto L
```

onde A , B e C representam endereços de variáveis, *op* representa operador (binário ou unário), *oprel* representa operador relacional, e L representa o rótulo de uma instrução intermediária.

*EXEMPLO 5.2 Código de três-endereços para o comando $A := X + Y * Z$*

O comando de atribuição acima traduzido para código de três-endereços é o seguinte:

```
T1 := Y * Z
T2 := X + T1
A := T2
```

onde $T1$ e $T2$ são variáveis temporárias.

Um código de três-endereços pode ser implementado através de quádruplas ou triplas. As quádruplas são constituídas de quatro campos: um operador, dois operandos e o resultado. A Figura 5.3 mostra a representação do comando $A := B * (-C + D)$ em quádruplas.

| | oper | arg1 | arg2 | result |
|-----|------|------|------|--------|
| (0) | -u | C | | T1 |
| (1) | + | T1 | D | T2 |
| (2) | * | B | T2 | T3 |
| (3) | := | T3 | | A |

Figura 5.3 Representação em quádruplas

As triplas são formadas por: um operador e dois operandos. A representação do mesmo comando através de triplas é mostrada na Figura 5.4. Essa representação utiliza ponteiros para a própria estrutura, evitando a nomeação explícita de temporários.

| | oper | arg1 | arg2 |
|-----|------|------|------|
| (0) | -u | C | |
| (1) | + | (0) | D |
| (2) | * | B | (1) |
| (3) | := | A | (2) |

Figura 5.4 Representação em triplas

O esquema de tradução do exemplo a seguir produz código de três-endereços para comandos de atribuição de expressões aritméticas simplificadas.

EXEMPLO 5.3 Esquema de tradução para gerar código de três-endereços.

A função *geratemp* gera o nome de uma variável temporária. O atributo *nome* associado ao símbolo E armazena o nome de uma variável (se resultou da redução $E \leftarrow id$), ou o nome de um temporário. A função *geracod* gera um *string* (texto) correspondente a uma *instrução* de código intermediário a partir dos argumentos indicados entre parênteses.

$$\begin{aligned}
 S &\rightarrow id := E & (S.cod = E.cod \parallel geracod(id.nome := E.nome)) \\
 E &\rightarrow E1 + E2 & (E.nome = geratemp; E.cod = E1.cod \parallel E2.cod \parallel geracod(E.nome := E1.nome + E2.nome)) \\
 E &\rightarrow E1 * E2 & \{E.nome = geratemp; E.cod = E1.cod \parallel E2.cod \parallel geracod(E.nome := E1.nome * E2.nome)\} \\
 E &\rightarrow (E1) & (E.nome = E1.nome; E.cod = E1.cod) \\
 E &\rightarrow id & \{E.nome = id.nome; E.cod = " "\}
 \end{aligned}$$

O atributo *cod* de S contém o código final gerado para o comando. O atributo *cod* de E contém o código para a expressão correspondente. Observe que esse esquema de tradução é uma gramática de atributos. A Figura 5.5 mostra a árvore de derivação anotada correspondente ao comando $A := X + Y * Z$.

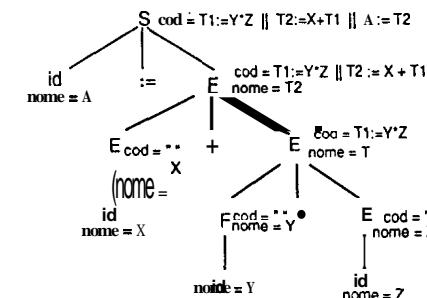


Figura 5.5 Árvore de derivação anotada para a expressão $A := X + Y * Z$

Reutilização de Temporários

Após uma variável temporária ser referenciada (aparecer no lado direito de uma *atribuição*), ela pode ser descartada. Para reutilização de temporários, usa-se um contador C, com valor inicial 1. Sempre que um novo temporário é gerado, usa-se TC no código e *incrementa-se* C de 1. Sempre que um temporário é usado como operando, *decrementa-se* C de 1.

EXEMPLO 5.4 Reutilização de temporários.

Para o comando de atribuição:

$$X := A * B + C * D - E * F$$

seria gerada a seguinte sequência de quádruplas:

$$\begin{array}{ll}
 T1 := A * B & (\text{incrementa } C \text{ de 1}) \\
 T2 := C * D & (\text{incrementa } C \text{ de 1}) \\
 T1 := T1 + T2 & (\text{decrementa } C \text{ de 1, duas vezes}) \\
 T2 := E * F & (\text{incrementa } C \text{ de 1}) \\
 T1 := T1 - T2 & (\text{decrementa } C \text{ de 1, duas vezes}) \\
 X := T1 & (\text{fim})
 \end{array}$$

5.2 Ações Semânticas para a Construção de Tabelas de Símbolos

Esta seção apresenta esquemas de tradução que reconhecem declarações de variáveis e geram tabelas de símbolos. Inicialmente, é apresentado um esquema que gera tabelas de símbolos para programas monolíticos, isto é, formados por um único bloco. Posteriormente, esse esquema é estendido para permitir a geração de tabelas para programas bloco-estruturados com procedimentos aninhados.

EXEMPLO 5.5 Esquema de tradução para gerar tabela de símbolos para bloco unitário.

O esquema de tradução a seguir constrói a tabela de símbolos para uma lista de declarações

| | |
|-----------------------------------|---|
| $P \rightarrow M \ D$ | |
| $M \rightarrow \epsilon$ | { desloc = 0 } |
| $D \rightarrow D ; D$ | |
| $D \rightarrow id : T$ | { adSimb (id.nome, T.tipo, desloc); desloc = desloc + T.tam } |
| $T \rightarrow int$ | (T.tipo = int; T.tam = 4) |
| $T \rightarrow real$ | (T.tipo = real; T.tam = 8) |
| $T \rightarrow array [num] of T1$ | (T.tipo = matriz (num.val, T1.tipo); T.tam = num.val * T1.tam) |
| $T \rightarrow ^T1$ | { T.tipo = ponteiro (T1.tipo); T.tam = 4 } |

O não-terminal M é empilhado logo no início do processo para atribuir o valor zero à variável $desloc$. Essa variável contém o próximo endereço disponível na área de dados do procedimento. Observe que $desloc$ não é atributo de símbolo da gramática. A rotina $adSimb$ adiciona um novo identificador à Tabela de Símbolos, com seu tipo e endereço. O atributo tam contém o tamanho em bytes para os diferentes tipos de dados aceitos nas declarações.

Observe que a redução $M \leftarrow \epsilon$ poderia ser realizada em qualquer momento do processo de construção da tabela, pois sempre se tem a palavra vazia no topo da pilha. Contudo, ao Programar o analisador, deve-se ter em mente que essa redução só deve ser efetuada no **início da análise**. Esse artifício de introduzir na gramática do esquema de tradução um não terminal artificial (caso de M) para forçar uma ação especial é uma técnica **comumente** usada.

A Figura 5.6 mostra a árvore de derivação e as ações semânticas para a declaração $a: int ; b: real$.

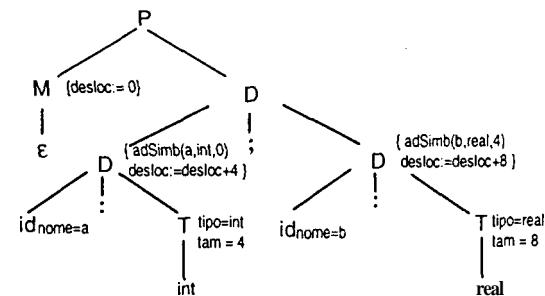


Figura 5.6 Árvore de derivação para a declaração $a: int ; b: real$.

O esquema de tradução a seguir estende o esquema anterior, incluindo uma regra de produção para a declaração de procedimentos. Isso permitirá a geração de tabelas de símbolos para procedimentos embutidos.

EXEMPLO 5.6 Esquema de tradução para gerar uma árvore de tabelas de símbolos.

Este esquema gera uma árvore de tabelas de símbolos para programas bloco-estruturados, obtendo-se uma tabela de símbolos para cada procedimento.

São usadas duas pilhas, $tabPtr$ e $desloc$: a primeira contém um ponteiro para a tabela de símbolos de cada procedimento, e a segunda contém o próximo endereço (de memória local) disponível na área de dados do procedimento. Durante a compilação, o ponteiro no topo de $tabPtr$ aponta para a tabela de símbolos do procedimento em análise. Cada tabela tem um ponteiro para a tabela do procedimento envolvente. Os símbolos não-terminais M e H servem para gerar, respectivamente, a tabela raiz (tabela de símbolos relativa ao programa principal, que contém as variáveis globais) e as tabelas referentes aos demais procedimentos. Observe que a redução $N \leftarrow \epsilon$ (que poderia ser efetuada a qualquer momento) deverá ser efetuada logo após ter sido empilhada a sequência de tokens "proc id ;".

As seguintes funções fazem parte do esquema de tradução:

- $geraTab(ptr)$ - gera uma tabela de símbolos (filha da tabela apontada por ptr) e retorna um ponteiro para a tabela gerada;
- $defTam(ptr, tam)$ - armazena na tabela de símbolos apontada por ptr o tamanho da área de dados local do procedimento correspondente;

- `adProc(ptr, nome, pt)` - insere na tabela de símbolos apontada por `ptr` o nome do procedimento e o ponteiro para a tabela de símbolos desse procedimento;
- `adSimb(ptr, nome, tipo, end)` - insere na tabela de símbolos apontada por `ptr` um novo símbolo, seu tipo e seu endereço na área de dados local.

| | |
|------------------------------------|--|
| <code>P → M D</code> | { <code>defTam (top (tabPtr), top (desloc));</code> <code>pop (tabPtr); pop (desloc)</code> } |
| <code>M → E</code> | { <code>t = geraTab (nil); push (t, tabPtr);</code> <code>push (0, desloc)</code> } |
| <code>D → D ; D</code> | |
| <code>D → id : T</code> | { <code>adSimb (top (tabPtr), id.nome, T.tipo, top (desloc));</code> <code>top (desloc) = top (desloc) + T.tam</code> } |
| <code>D → proc id; ND; S</code> | { <code>t = top (tabPtr); defTam (t, top (desloc));</code> <code>pop (tabPtr); pop (desloc);</code> <code>adProc (top (tabPtr), id.nome, t)</code> } |
| <code>N → ε</code> | { <code>t = geraTab (top (tabPtr));</code> <code>push (t, tabPtr); push (0, desloc)</code> } |
| <code>T → int</code> | { <code>T.tipo = int; T.tam = 4</code> } |
| <code>T → real</code> | { <code>T.tipo = real; T.tam = 8</code> } |
| <code>T → array [num] of T1</code> | { <code>T.tipo = matriz (num.val, T1.tipo);</code> <code>T.tam = num.val * T1.tam</code> } |
| <code>T → ^ T1</code> | { <code>T.tipo = ponteiro (T1.tipo); T.tam = 4</code> } |

A Figura 5.7 mostra as tabelas de símbolos que seriam geradas para o seguinte programa bloco-estruturado:

```
a : real;
b : int;
proc p1;
  c : real;

end p1;
proc p2;
  d : array[5] of int;
  proc p3;
    e, f: real;

  end p3;
  ---
end p2;
---
```

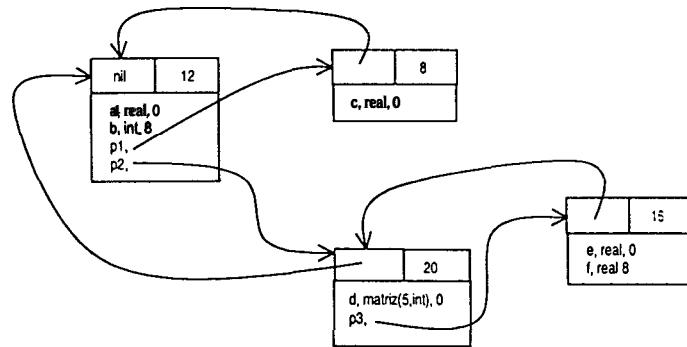


Figura 5.7 Tabela de símbolos para um programa bloco-estruturado

5.3 Geração de Código para Comandos de Atribuição

Esta seção trata da geração de código para comandos de atribuição. São apresentados, também, esquemas de tradução correlatos, para conversão de tipos e para endereçamento de elementos de matrizes.

EXEMPLO 5.7 Esquema de tradução para comandos de atribuição.

Este esquema de tradução gera código de três-endereços para comandos de atribuição. A função `lookup(id.nome)` procura o identificador armazenado `id.nome` na tabela de símbolos. Se existe uma entrada na tabela para esse identificador, a função retoma o índice correspondente ao mesmo; caso contrário, retoma `nil`. A ação semântica `geracod` grava comandos de três-endereços num arquivo de saída.

A diferença entre este esquema e o apresentado no Exemplo 5.3 é que aquele implementa uma gramática de atributos. Lá as ações envolvem apenas o cálculo de atributos (não há efeitos colaterais) e, no final, o código gerado fica armazenado no atributo `ccd` de `S`.

| | |
|--------------------------|---|
| <code>S → id := E</code> | { <code>p = lookup (id.nome);</code> <code>if p ≠ nil then geracod (p " := " E.ptr) else erro</code> } |
| <code>E → E1 + E2</code> | { <code>E.ptr = geratemp;</code> <code>geracod (E.ptr " := " E1.ptr " + " E2.ptr)</code> } |
| <code>E → E1 * E2</code> | { <code>E.ptr = geratemp;</code> <code>geracod (E.ptr " := " E1.ptr " * " E2.ptr)</code> } |
| <code>E → - E1</code> | { <code>E.ptr = geratemp;</code> <code>geracod (E.ptr " := " "-u" E1.ptr)</code> } |
| <code>E → (E1)</code> | { <code>E.ptr = E1.ptr</code> } |
| <code>E → id</code> | { <code>p = lookup (id.nome); if p ≠ nil then E.ptr = p else erro</code> } |

Quando aplicada a um identificador, a função *lookup* deve primeiro verificar se o identificador está na tabela de símbolos corrente, apontada por *top(tabPtr)*. Caso negativo, *lookup* usa o ponteiro dessa tabela para encontrar a tabela de símbolos do procedimento envolvente. Se o nome não é encontrado em nenhum dos escopos envolventes, então *lookup* retoma *nil*.

Como a função *lookup* retoma um ponteiro para a posição de memória correspondente a um identificador, então, para manter coerência, a função *geratemp* também deve retomar um ponteiro para a área **alocada** a um temporário (e não o nome de um iemporário).

5.3.1 Conversão de Tipos em Expressões Aritméticas

Na geração de código intermediário para expressões aritméticas, o tipo dos operandos (p. ex.. inteiro ou real) determina a natureza da operação (adição de inteiros ou de ponto flutuante) que deve ser efetuada. As ações **semânticas** abaixo, associadas à operação de soma, fazem a verificação do tipo dos operandos para **determinar** o tipo de operação a ser aplicado.

EXEMPLO 5.8 Verificação de tipo em expressões aritméticas.

```
E → E1 + E2      { E.nome = geratemp;
                     if E1.tipo = inteiro and E2.tipo = inteiro
                     then geracod( E.nome := "E1.nome "+int" E2.nome );
                           E.tipo =inteiro
                     else if E1.tipo = real and E2.tipo = real
                           then geracod( E.nome := "E1.nome "+real" E2.nome );
                                 E.tipo =real
                     else if E1.tipo = inteiro and E2.tipo = real
                           then u = geratemp;
                                 geraccd ( u := "convreal" E1.nome );
                                 geraccd ( E.nome := u "+real" E2.nome );
                                 E.tipo = real
                     else if E1.tipo = real and E2.tipo = inteiro
                           then u = geratemp;
                                 geracod ( u := "convreal" E2.nome );
                                 geracod ( E.nome := E1.nome "+real" u );
                                 E.tipo = real
                     else E.tipo = tipo_erro }
```

Ações semânticas análogas à anterior podem ser anexadas às produções que reconhecem as **operações aritméticas** de subtração, multiplicação, etc. Para o comando

$$X := Y + I * J$$

sendo X e Y do tipo real, e I e J do tipo inteiro, seriam geradas as seguintes quádruplas:

$$\begin{aligned} T1 &:= I * \text{int} J \\ T3 &:= \text{convreal } T1 \\ T2 &:= Y + \text{real } T3 \\ X &:= T2 \end{aligned}$$

5.3.2 Endereçamento de Elementos de Matrizes

Esta **seção** apresenta um esquema de tradução que gera código para **acessar** elementos de matrizes. Primeiramente, far-se-á um estudo sobre **endereçamento de vetores**, o qual **será** estendido para matrizes multidimensionais.

O endereço do i-ésimo elemento de um vetor A é obtido por:

$$\text{base} + (i - \text{low}) * w$$

onde w = comprimento de cada elemento;
 low = limite inferior;
 base = endereço inicial da memória reservada para o vetor, ou seja, é o endereço de A[low].

A expressão acima pode ser parcialmente avaliada em tempo de compilação se reescrita como:

$$i * w + (\text{base} - \text{low} * w)$$

A subexpressão

$$c = \text{base} - \text{low} * w$$

pode ser avaliada quando a declaração do vetor é analisada. O valor de c pode ser **armazenado** na tabela de símbolos, na entrada para o vetor A, tal que o endereço relativo de A[i] é obtido, somando-se $(i * w) + c$. O mesmo tipo de cálculo pode ser aplicado a matrizes **bidimensionais**, **tridimensionais**, etc.

No caso de uma matriz bidimensional, ela pode ser **armazenada** por linha ou por **coluna**. Por linha, a matriz A[2x3] seria armazenada da seguinte maneira:

| | | | | | |
|--------|--------|--------|--------|--------|--------|
| A[1,1] | A[1,2] | A[1,3] | A[2,1] | A[2,2] | A[2,3] |
|--------|--------|--------|--------|--------|--------|

Por coluna, o armazenamento seria como segue:

| | | | | | |
|--------|--------|--------|--------|--------|--------|
| A[1,1] | A[2,1] | A[1,2] | A[2,2] | A[1,3] | A[2,3] |
|--------|--------|--------|--------|--------|--------|

No caso de a matriz ser armazenada por linha, o endereço relativo A[i1,i2] pode ser calculado pela fórmula:

$$\text{base} + ((\text{ii} - \text{lowi}) * n_2 + \text{iz} - \text{lowz}) * w$$

onde lowi e lowz são os limites inferiores;

n_2 é o número de elementos da segunda dimensão, isto é, $n_2 = \text{high}_2 - \text{low}_2 + 1$.

Assumindo que ii e iz são os únicos valores não conhecidos em tempo de compilação, a expressão acima pode ser reescrita da seguinte forma:

$$((\text{ii} * n_2) + \text{iz}) * w + c$$

onde $c = (\text{base} - ((\text{lowi} * n_2) + \text{lowz}) * w)$

sendo que o valor de c pode ser calculado em tempo de compilação.

Generalizando, para uma matriz multidimensional armazenada por linha, o endereço relativo do elemento A[i1, i2, ..., ik] é dado por

$$((\dots((\text{ii} * n_2 + \text{iz}) * n_3 + \text{iz}) \dots) * n_k + \text{ik}) * w + c$$

onde $c = (\text{base} - ((\dots(\text{lowi} * n_2 + \text{lowz}) * n_3 + \text{lowz}) \dots) * n_k + \text{lowk}) * w$

Como, para qualquer j , $n_j = \text{high}_j - \text{low}_j + 1$ é fixo, então o valor de c pode ser computado pelo compilador e salvo na entrada da tabela de símbolos para A.

Geração de Código para Referências a Elementos de Matrizes

Seja a gramática que gera referências a elementos de matrizes:

$$\begin{aligned}\text{Var} &\rightarrow \text{id} [\text{Elist}] \mid \text{id} \\ \text{Elist} &\rightarrow \text{Elist}, \text{E} \quad \mid \text{E}\end{aligned}$$

Reescrevendo a gramática para permitir que o ponteiro para id (na tabela de símbolos) seja passado como um atributo sintetizado, tem-se:

$$\begin{aligned}\text{Var} &\rightarrow \text{Elist}] \mid \text{id} \\ \text{Elist} &\rightarrow \text{Elist}, \text{E} \mid \text{id} [\text{E}\end{aligned}$$

O código de três-endereços a ser gerado para computar, em tempo de execução, a expressão:

$$((\dots((\text{ii} * n_2 + \text{iz}) * n_3 + \text{iz}) \dots) * n_k + \text{ik}) * w$$

baseia-se nas fórmulas de recorrência:

$$\begin{aligned}\text{ei} &= \text{ii} \\ \text{em} &= \text{em} - 1 * \text{nm} + \text{im}, \quad m = 2, 3, \dots, k\end{aligned}$$

EXEMPLO 5.9 Endereçamento de elementos de matrizes.

O esquema de tradução a seguir gera código para comandos de atribuição que permitem referenciar elementos de matrizes:

$$\begin{aligned}\text{S} \rightarrow \text{Var} := \text{E} &\quad \{ \text{if Var.desloc} = \text{null} \quad /* \text{Var é variável simples */} \\ &\quad \text{then geracod} (\text{Var.nome} " := " \text{E.nome}) \\ &\quad \text{else geracod} (\text{Var.nome} "[\text{Var.desloc}]" " := " \text{E.nome}) \} \\ \text{E} \rightarrow \text{E1} + \text{E2} &\quad \{ \text{E.nome} := \text{geratemp}; \\ &\quad \text{geracod} (\text{E.nome} " := " \text{E1.nome} " + " \text{E2.nome}) \} \\ \text{E} \rightarrow (\text{E1}) &\quad \{ \text{E.nome} := \text{E1.nome} \} \\ \text{E} \rightarrow \text{Var} &\quad \{ \text{if Var.desloc} = \text{null} \quad /* \text{Var é variável simples */} \\ &\quad \text{then E.nome} := \text{Var.nome} \\ &\quad \text{else E.nome} := \text{geratemp}; \\ &\quad \text{geracod} (\text{E.nome} " := " \text{Var.nome} "[\text{Var.desloc}]") \} \\ \text{Var} \rightarrow \text{Elist}] &\quad \{ \text{Var.nome} := \text{geratemp}; \\ &\quad \text{Var.desloc} := \text{geratemp}; \\ &\quad \text{geracod} (\text{Var.nome} " := " \text{w} " * " \text{Elist.nome}) \} \\ \text{Var} \rightarrow \text{id} &\quad \{ \text{Var.nome} := \text{id.nome}; \\ &\quad \text{Var.desloc} := \text{null} \} \\ \text{Elist} \rightarrow \text{Elist}, \text{E} &\quad \{ \text{t} := \text{geratemp}; \text{m} := \text{Elist.ndim} + 1; \\ &\quad \text{geracod} (\text{t} " := " \text{Elist1.nome} " * " \text{limit} (\text{Elist1.array}, \text{m})); \\ &\quad \text{geracod} (\text{t} " := " \text{t} " + " \text{E.nome}); \\ &\quad \text{Elist.array} := \text{Elist1.array}; \\ &\quad \text{Elist.nome} := \text{t}; \text{Elist.ndim} := \text{m} \} \\ \text{Elist} \rightarrow \text{id} [\text{E} &\quad \{ \text{Elist.array} := \text{id.nome}; \\ &\quad \text{Elist.nome} := \text{E.nome}; \\ &\quad \text{Elist.ndim} := 1 \}\end{aligned}$$

onde

- Elist.ndim** - contém o número de dimensões da matriz;
- limit(array, j)** - função que retorna o número de elementos na dimensão j ;
- Elist.nome** - temporário que contém o valor computado a partir da expressão Elist;
- Var.desloc** - temporário contendo o valor da expressão de endereçamento computada em tempo de execução; quando igual a null, indica variável simples.

Seja M uma matriz 5×10 , na qual o tamanho de cada elemento é $w=8$. A árvore anotada para o comando $X := M [I, J + K]$ é apresentada na Figura 5.8. O comando é traduzido na seguinte sequência de quádruplas:

$$\begin{array}{ll} T1 := J + K & T4 := 8 * T2 \\ T2 := I * 10 & T5 := T3[T4] \\ T2 := T2 + T1 & X := T5 \\ T3 := \text{endM} - 88 & \end{array}$$

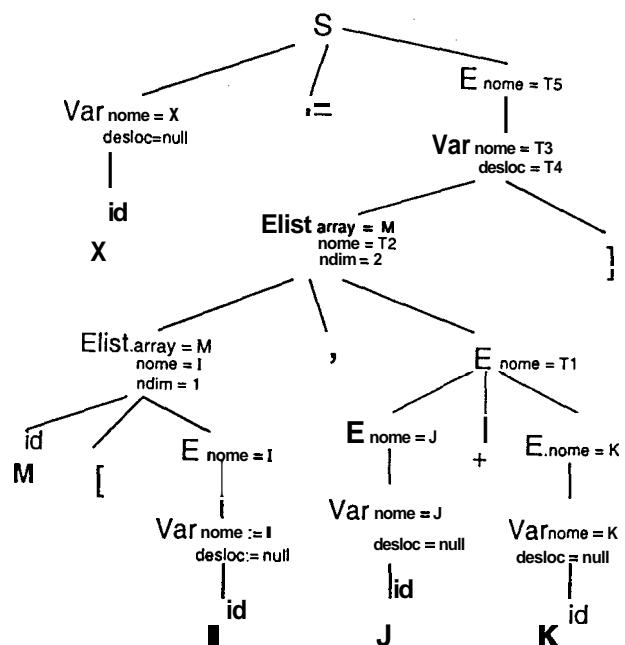


Figura 5.8 Árvore de derivação decorada para $X := M [I, J + K]$.

5.4 Expressões Lógicas e Comandos de Controle

Expressões lógicas são usadas como expressões condicionais em comandos de controle (*if*, *while*, etc) e em comandos de atribuição lógica. Nesta seção, serão apresentados dois métodos de tradução para expressões lógicas:

- 1) **Representação numérica:** este método codifica numericamente as constantes `true` e `false` (por exemplo, `true=1` e `false=0`) e avalia as expressões lógicas de forma numérica, ficando o resultado da avaliação numa variável temporária.
- 2) Representação por fluxo de controle: este método traduz expressões lógicas para instruções *if* e *goto* que desviam a execução do programa para pontos distintos, caso o resultado da avaliação seja `true` ou `false`. Para tanto, são dados os atributos `E.true` e `E.false` que contêm rótulos para onde a execução deve ser transferida em caso de avaliação `true` ou `false`, respectivamente.

5.4.1 Representação Numérica de Expressões Lógicas

Neste método, as operações lógicas *and*, *or* e *not* são avaliadas numericamente para 1 quando resultarem em `true`, e para 0, quando resultarem em `false`.

EXEMPLO 5.10 Código para avaliar *expressões* lógicas de *forma numérica*.

Supondo que o código gerado seja armazenado a partir da quádrupla número 100, o comando de atribuição “ $X := A \text{ or } B \text{ and not } C$ ” (onde A , B e C são variáveis lógicas) seria traduzido para:

$$\begin{array}{l} 100: T1 := \text{not } C \\ 101: T2 := B \text{ and } T1 \\ 102: T3 := A \text{ or } T2 \\ 103: X := T3 \end{array}$$

A expressão relacional $A < B$ (onde A e B são variáveis numéricas) seria traduzida para:

$$\begin{array}{l} 100: \text{if } A < B \text{ goto } 103 \\ 101: T1 := 0 \\ 102: \text{goto } 104 \\ 103: T1 := 1 \\ 104: \end{array}$$

Observe que o valor da expressão fica no **último** temporário gerado no processo de avaliação.

EXEMPLO 5.11 Esquema de tradução para avaliação numérica de expressões lógicas.

O esquema de tradução abaixo gera código para expressões lógicas, supondo que as instruções geradas são armazenadas num vetor de quâdmpias. A função *geracod* utiliza a variável *proxq* para indicar o índice da próxima quâdrupla disponível. Após gravar uma quâdmpia, a função *geracod* incrementa *proxq*.

| | |
|--|---|
| $E \rightarrow E_1 \text{ or } E_2$ | { E.nome = geratemp; geracod(E.nome ":= E1.nome "or" E2.nome) } |
| $E \rightarrow E_1 \text{ and } E_2$ | { E.nome = geratemp; geracod(E.nome ":= E1.nome "and" E2.nome) } |
| $E \rightarrow \text{not } E_1$ | { E.nome = geratemp; geracod(E.nome ":= "not" E1.nome) } |
| $E \rightarrow (E_1)$ | { E.nome = E1.nome } |
| $E \rightarrow \text{id}_1 \text{ oprel } \text{id}_2$ | { E.nome = geratemp; geracod("if" id1.nome oprel.simb id2.nome "goto" proxq + 3) geracod(E.nome ":= 0"); geracod("goto" proxq + 2); geracod(E.nome ":= 1") } |
| $E \rightarrow \text{tme}$ | { E.nome = geratemp; geracod(E.nome ":= 1") } |
| $E \rightarrow \text{false}$ | { E.nome = geratemp; geracod(E.nome ":= 0") } |

Observe que o atributo nome de E contém o nome do temporário que armazena o resultado da avaliação da expressão E. O código gerado (código que avalia a expressão) é colocado no vetor de quâdmpias.

EXEMPLO 5.12 Código gerado para a expressão $A < B \text{ or } C < D \text{ and } E < F$.

O esquema anterior gera a seguinte sequência de quâdmpias para a expressão acima:

| | |
|------------------------|------------------------|
| 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 := R and T3 |
| 106: goto 108 | 113: T5 := T1 or T4 |

Conforme já mencionado, o valor final da expressão fica no último temporário gerado no processo de avaliação (T5, neste exemplo). O nome desse temporário é armazenado no atributo *nome* do não-terminal que representa a expressão, e o código gerado é armazenado no vetor de quâdmpias. A Figura 5.9 mostra os momentos da análise em que os temporários são criados.

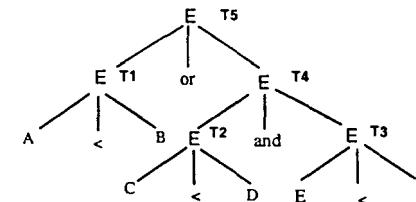


Figura 5.9 Ordem de criação dos temporários.

EXEMPLO 5.13 Geração de código com gramática de atributos.

O Exemplo 5.11 gera efeitos colaterais por gravar o código no vetor de quâdmpias (isso é feito pela função *geracod*). Pode-se transformar aquele esquema de tradução numa gramática de atributos que armazene o código gerado num novo atributo. Essa transformação é indicada a seguir.

Basta introduzir um novo atributo *cod* e substituir os comandos que geram código, conforme ilustrado a seguir:

| | |
|--------------------------------------|---|
| $E \rightarrow E_1 \text{ and } E_2$ | { E.nome = geratemp; geracod(E.nome ":= E1.nome "and" E2.nome) } |
|--------------------------------------|---|

é substituído por:

| | |
|--|---|
| $E \rightarrow E_1 \text{ and } E_2 ;$ | { E.nome = geratemp; E.cod = E1.cod E2.cod geracod(E.nome ":= E1.nome "and" E2.nome) } |
|--|---|

Agora a função *geracod* retoma um *string* que é a linha de código gerado.

EXEMPLO 5.14 Geração de código para contando while.

O esquema de tradução abaixo é uma gramática de atributos que gera código intermediário para comandos tipo *while-do* conforme o diagrama da Figura 5.10. O código gerado é um *string* de

instruções e rótulos que fica armazenado no atributo **cod** de S. Os atributos **início** e **prox** identificam, respectivamente, o início da iteração e o início do comando seguinte ao **while**.

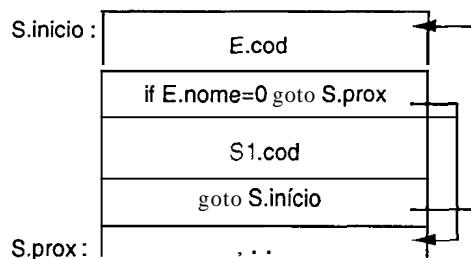


Figura 5.10 Diagrama do comando **while-do**

$S \rightarrow \text{while } E \text{ do } S_1 \quad \{S.\text{inicio} = \text{gerarótulo}; S.\text{prox} = \text{gerarótulo};$
 $S.\text{cod} = \text{geracod}(S.\text{inicio} ":") \parallel E.\text{cod} \parallel$
 $\quad \quad \quad \text{geracod}(\text{"if"} E.\text{nome} = 0 \text{ goto"} S.\text{prox}) \parallel S_1.\text{cod} \parallel$
 $\quad \quad \quad \text{geracod}(\text{"goto"} S.\text{inicio}) \parallel \text{geracod}(S.\text{prox} ":")\}$

□

A avaliação numérica de expressões lógicas adapta-se bem a esquemas S-atribuídos (**bottom-up**), pois são usados apenas atributos sintetizados. A avaliação por fluxo de controle (a seguir) adapta-se a esquemas L-atribuídos, pois usa atributos herdados e sintetizados. A vantagem da avaliação por fluxo de controle é que o código gerado é mais eficiente.

5.4.2 Representação por Fluxo de Controle

Este método traduz expressões lógicas para um código formado por instruções **if** e **goto**. São gerados os rótulos **E.true** e **E.false** para onde a execução deve ser transfenda em caso de avaliação **true** e **false**, respectivamente.

EXEMPLO 5.15 Tradução de expressões lógicas para fluxo de controle.

O esquema de tradução abaixo traduz expressões lógicas para um fluxo de controle, usando instruções **if** e **goto**. Os atributos herdados **true** e **false** de E especificam os rótulos de desvio. Esses rótulos são criados (através da função **gerarótulo**) durante o reconhecimento do comando que contém a expressão lógica E (os locais dos rótulos dependem, portanto, do contexto).

O atributo sintetizado **cod** de E conterá, no final da tradução, o código gerado para a expressão analisada.

| | |
|--------------------------------------|--|
| $E \rightarrow E_1 \text{ or } E_2$ | { $E_1.\text{true} = E.\text{true}; E_1.\text{false} = \text{gerarótulo};$ $E_2.\text{true} = E.\text{true}; E_2.\text{false} = E.\text{false};$ $E.\text{cod} = E_1.\text{cod} \parallel \text{geracod}(E_1.\text{false} ":") \parallel E_2.\text{cod} \}$ |
| $E \rightarrow E_1 \text{ and } E_2$ | { $E_1.\text{true} = \text{gerarótulo}; E_1.\text{false} = E.\text{false};$ $E_2.\text{true} = E.\text{true}; E_2.\text{false} = E.\text{false};$ $E.\text{cod} = E_1.\text{cod} \parallel \text{geracod}(E_1.\text{true} ":") \parallel E_2.\text{cod} \)$ |
| $E \rightarrow \text{not } E_1$ | { $E_1.\text{true} = E.\text{false}; E_1.\text{false} = E.\text{true}; E.\text{cod} = E_1.\text{cod} \)$ |
| $E \rightarrow (\ E_1 \)$ | { $E_1.\text{true} = E.\text{true}; E_1.\text{false} = E.\text{false}; E.\text{cod} = E_1.\text{cod} \)$ |
| $E \rightarrow \text{idl oprel id2}$ | { $E.\text{cod} = \text{geracod}(\text{"if"} \ "idl.\text{nome oprel.simb id2.\text{nome} "goto"} \ E.\text{true}); \text{geracod}(\text{"goto"} E.\text{false}) \)$ |
| $E \rightarrow \text{true}$ | { $E.\text{cod} = \text{geracod}(\text{"goto"} E.\text{true}) \)$ |
| $E \rightarrow \text{false}$ | { $E.\text{cod} = \text{geracod}(\text{"goto"} E.\text{false}) \ }$ |

EXEMPLO 5.16 Geração de código para expressão lógica.

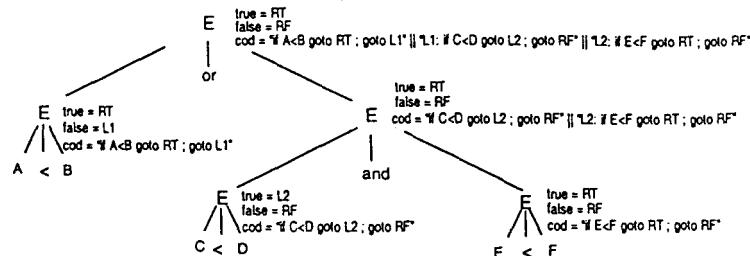
Supondo que os atributos **irue** e **ifalse** da expressão **A < B or C < D and E < F** tenham recebido os rótulos RT e RF, o esquema anterior gera a seguinte sequência de quádruplas:

```

if A < B goto RT
goto L1
L1: if C < D goto L2
goto RF
L2: if E < F goto RT
goto RF
  
```

□

Vale observar que o esquema de tradução anterior (que é uma gramática de atributos) não é nem S-atribuído, nem L-atribuído. Como tal, sua implementação requer a execução de três passos. O primeiro passo constrói a árvore de derivação (constmção bottom-up, pois existe recursividade à esquerda). O segundo percorre a árvore no sentido top-down para decorá-la com os atributos **E.true** e **E.false**. O terceiro passo percorre novamente a árvore no sentido bottom-up, para calcular os atributos **cod**. A árvore decorada final correspondente ao exemplo **A < B or C < D and E < F** é mostrada na Figura 5.11.

Figura 5.11 Árvore anotada para a expressão $A < B \text{ or } C < D \text{ and } E < F$.**EXEMPLO 5.17 Esquema de tradução para cornaridos de controle.**

O esquema a seguir traduz comandos de controle tipo *if-then*, *if-then-else* e *while-do*, segundo os diagramas apresentados na Figura 5.12. O atributo herdado *prox* de *S* identifica o comando que segue a *S* (é um atributo definido, portanto, pelo contexto de *S*).

```

S → if E then S1      { E.true = gerarótulo: E.false = S.prox; S1.prox = S.prox;
                           S.cod = E.cod || geracod(E.true ":") || S1.cod }

S → if E then S1 else S2 { E.true = == gerarótulo: E.false == gerarótulo;
                           S1.prox = S.prox; S2.prox = S.prox;
                           S.cod = E.cod || geracod(E.true ":") || S1.cod ||
                           geracod("goto" S.prox) ||
                           geracod(E.false ":") || S2.cod }

S → while E do S1     { S.início = gerarótulo:
                           E.true = gerarótulo; E.false = S.prox;
                           S1.prox = S.início;
                           S.cod = geracod(S.início ":") || E.cod || geracod(E.true ":") || S1.cod ||
                           geracod("goto" S.início) }

```

Conforme já mencionado, esse esquema **não é S-atribuído**, nem L-atribuído. Na próxima seção, será mostrado como esse esquema pode ser modificado para um esquema S-atribuído, permitindo que a análise sintática (construção da árvore de derivação) e a execução das ações semânticas sejam realizadas num único passo.

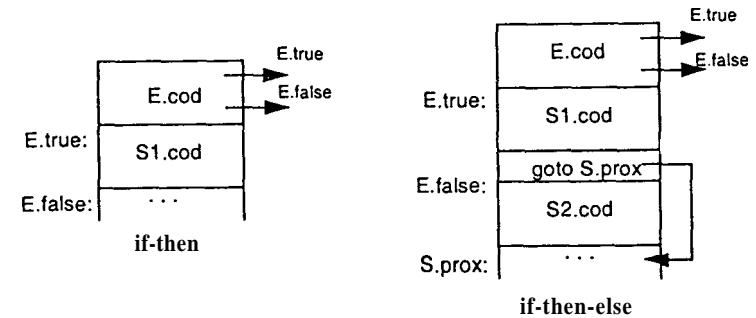


Figura 5.12 Diagramas dos comandos de controle

EXEMPLO 5.18 Geração de código para comando *while* e *if*.

Considere o comando

while $A < B$ do if $C < D$ then $X := Y + Z$ else $X := Y - Z$

Os esquemas de tradução definidos para comandos de atribuição, expressões lógicas e comandos de controle geram a seguinte sequência de quádruplas (supondo que o atributo herdado *prox* do comando *while* valha PRX):

```

L1: if A < B goto L2
      goto PRX
L2: if C < D goto L3
      goto L4
L3: T1 := Y + Z
      X := T1
      goto L1
L4: T2 := Y - Z
      X := T2
      goto L1
PRX:
  
```

5.5 Backpatching

O principal problema, na geração de código para expressões lógicas e comandos de controle, é que o código gerado deve incluir comandos de desvio para endereços que, em geral, ainda não são conhecidos. Isso inviabiliza a geração de código num único passo. Esta seção apresenta uma solução para esse problema, através da criação de listas de comandos de desvio incompletos (*cujos* endereços ainda não estão resolvidos) que serão devidamente completados quando o endereço destino for conhecido. Ao preenchimento dos endereços não resolvidos dá-se o nome de *backpatching*.

Esta seção mostra como *backpatching* pode ser adicionado a um reconhecedor *bottom-up* para gerar código, num único passo, para expressões lógicas, comandos de controle e demais comandos de uma linguagem de programação. O código gerado será armazenado num vetor de **quádruplas**, e os rótulos serão índices desse vetor.

São necessárias três funções para o processo de backpatching:

- 1) ***makelist(i)*** – cria uma lista contendo *i* (um único elemento) e retoma um ponteiro para a lista criada; o elemento *i* é um índice do vetor de **quádruplas**;
- 2) ***merge(p1, p2)*** – concatena as listas apontadas por *p1* e *p2*, e retoma um ponteiro para a lista resultante;
- 3) ***backpatch(p, i)*** – insere *i* (rótulo destino) no campo de endereço de cada uma das quádruplas da lista apontada por *p*.

5.5.1 Backpatching em Expressões Lógicas

O esquema de tradução abaixo produz quádruplas para expressões lógicas, usando um analisador redutivo. Foi acrescentado o símbolo não-terminal *M*, que tem como objetivo guardar o endereço da próxima quádrupla disponível no momento em que *M* é empilhado. Observe que *M* é empilhado logo após ter sido empilhado um *roken* “or” ou “and”.

EXEMPLO 5.19 Esquema de tradução com backpatching para expressões lógicas.

Neste esquema, **E.true** e **E.false** são atributos sintetizados, os quais são listas que indicam quádruplas com comandos de desvio incompletos. Os endereços são preenchidos, usando o valor armazenado em **M.quad**, quando ocorre um *backpatching* na lista.

| | |
|--|--|
| $E \rightarrow E_1 \text{ or } M \ E_2$ | { backpatch($E_1.\text{false}$, $M.\text{quad}$); $E.\text{true} = \text{merge}(E_1.\text{true}, E_2.\text{true})$; $E.\text{false} = E_2.\text{false}$ } |
| $E \rightarrow E_1 \text{ and } M \ E_2$ | { backpatch($E_1.\text{true}$, $M.\text{quad}$); $E.\text{true} = E_2.\text{true}$; $E.\text{false} = \text{merge}(E_1.\text{false}, E_2.\text{false})$ } |
| $E \rightarrow \text{not } E_1$ | { $E.\text{true} = E_1.\text{false}$; $E.\text{false} = E_1.\text{true}$ } |
| $E \rightarrow (E_1)$ | { $E.\text{true} = E_1.\text{true}$; $E.\text{false} = E_1.\text{false}$ } |
| $E \rightarrow id_1 \text{ oprel } id_2$ | ($E.\text{true} = \text{makelist}(\text{proxq})$; $E.\text{false} = \text{makelist}(\text{proxq} + 1)$; $\text{geracod}(\text{"if " } id_1.\text{name oprel.op id}_2.\text{name " goto " } _)$ $\text{geracod}(\text{"goto " } _)$) |
| $E \rightarrow id$ | { $E.\text{true} = \text{makelist}(\text{proxq})$; $E.\text{false} = \text{makelist}(\text{proxq} + 1)$; $\text{geracod}(\text{"if " } id.\text{name " goto " } _)$ $\text{geracod}(\text{"goto " } _)$ } |
| $M \rightarrow \epsilon$ | { $M.\text{quad} = \text{proxq}$ } |

O identificador ***id*** pode representar tanto uma variável numérica como uma variável lógica. Em ambos os casos, o valor da variável é referido usando o atributo ***nome*** de ***id***. Observe que identificadores numéricos só podem aparecer em expressão do tipo ***idl oprel id2***.

Quando for feita a redução $E \leftarrow E_1 \text{ or } M \ E_2$, os endereços em $E_1.\text{false}$ serão preenchidos com o endereço da quádrupla onde inicia o código que avalia E_2 (esse endereço está no atributo

quad de M). No final, estarão **armazenados** nos atributos **E.true** e **E.false** do símbolo raiz, as listas de quádruplas a serem completadas com os endereços dependentes **do contexto**. Essas listas serão resolvidas quando for reconhecido o comando que contém a expressão lógica.

EXEMPLO 5.20 Geração de código para $A < B$ or $C < D$ and $E < F$.

O Exemplo 5.16 mostrou o código para essa expressão, supondo que os atributos **true** e **false** valessem RT e RF. O esquema com **backpatching** irá gerar a seguinte sequência de quádruplas, considerando o código resultante carregado a partir da quádrupla 100:

```

100: if A < B goto __
101: goto 102
102: if C < D goto 104
103: goto __
104: if E < F goto __
105: goto __

```

A Figura 5.13 mostra a árvore de derivação anotada para a expressão, onde se pode ver as listas geradas e os valores do atributo **quad** de M. Os campos de endereço das quádruplas 102 e 104 já estão preenchidos no código gerado. As quádruplas 100, 104, 103 e 105 só serão completadas quando ficarem conhecidos os endereços para onde a execução deve desviar. Esses endereços são dependentes do comando que contém a expressão lógica.

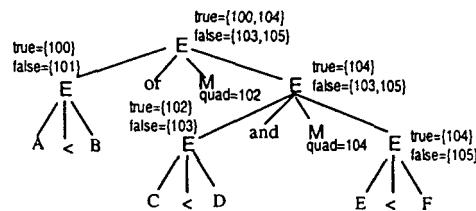


Figura 5.13 Árvore de derivação anotada para $A < B$ or $C < D$ and $E < F$.

5.5.2 Backpatching para Comandos de Controle

A seguir, o esquema de tradução anterior é estendido para tratar os comandos de controle e os demais comandos das linguagens de programação. É introduzido um novo não-terminal N, que serve para marcar o local da instrução "**goto —**" no fim do bloco **then**, em um comando **if-then-else**. Esta instrução deve ocasionar um salto sobre o bloco **else**. O atributo **go** de N

contém o número da quádrupla correspondente a esse **goro**. O endereço deixado em branco será preenchido quando for conhecido o endereço do comando seguinte ao **if**. Observe que o atributo **prox**, associado a S (comando) e a L (lista de comandos), é uma lista de desvios (lista de quádruplas com comandos **goro**) que devem ser completados com o endereço do comando que segue a S ou a L.

Observe também que o símbolo M é empilhado logo após ter sido empilhado um **token** "then", "else", "while", "do" ou ";", e que o símbolo N é empilhado imediatamente antes de ser empilhado o token "else".

EXEMPLO 5.21 Esquema de tradução com backpatching para uma linguagem simplificada

Na redução de cada comando, a primeira ação é preencher os campos de endereço das listas criadas durante a análise do lado direito da produção. Com exceção do comando **while**, cuja lista **S1.prox** pode ser tratada durante o próprio comando, as listas **prox** dos demais comandos (a serem preenchidas com o endereço do comando seguinte) precisam ser copiadas para serem tratadas mais tarde, quando o endereço do comando sequencial seguinte fica conhecido (isso só acontece quando o **token** ";" é encontrado).

```

S → if E then M Si           { backpatch(E.true, M.quad);
                                S.prox = merge(E.false, S1.prox) }

S → if E then Mi Si N else M2 S2 { backpatch(E.true, M1.quad);
                                      backpatch(E.false, M2.quad);
                                      S.prox=merge(S1.prox,merge(N.go,S2.prox)) }

N → E                         { N.go = makelist(proxq);
                                geracod(goto __) }

S → while M1 E do M2 Si      { backpatch(S1.prox, Mi.quad);
                                backpatch(E.true, M2.quad);
                                S.prox = E.false ;
                                geracod(goto Mi.quad) }

S → begin L end             { S.prox = L.prox }

S → A                         { S.prox = makelist(nil) }

A → id := E                   { ações semânticas para o comando de atribuição }

L → L1 ; M S                 { backpatch(L1.prox, M.quad);
                                L.prox = S.prox }

L → S                         { L.prox = S.prox }

P → S .                        { backpatch(S.prox, 9999) }

```

Observe que, quando for feita a redução final de "S." para P (símbolo inicial da gramática), os endereços da lista `S.prox` serão preenchidos com o valor 9999. Uma instrução de transferência para o endereço 9999 corresponde a uma transferência para o sistema operacional. Quando acaba a execução de um programa, o controle deve passar para o SO para que ele desterro o processo correspondente ao programa e libere seus recursos (memória, arquivos, etc).

EXEMPLO 5.22 Geração de código para um programa completo

Considere o seguinte programa:

```
while A < B do
    if C < D
        then X := Y + Z
        else X := Y - Z .
```

O esquema de tradução anterior gera a seguinte sequência de quádruplas:

```
000: if A < B goto 002
001: goto 9999
002: if C < D goto 004
003: goto 007
004: T1 := Y + Z
005: X := T1
006: goto 000
007: T2 := Y - Z
008: X := T2
009: goto 000
```

□

Para melhorar o entendimento do esquema de tradução apresentado nesta seção, o qual **corresponde** a um compilador completo para uma linguagem simplificada, é interessante que o leitor simule o funcionamento da pilha do analisador, bem como o preenchimento do **vetor** de quádruplas, observando como os atributos vão sendo calculados e como o código vai sendo gerado.

EXERCÍCIOS

- 1) Dado o comando de atribuição abaixo. traduza-o para uma árvore de sintaxe, para um grafo de sintaxe e derive código intermediário pós-fixado e de três-endereços (o último a partir de cada uma das representações gráficas):

$$x := (a + b) * c - (a + b) / d$$

- 2) Defina ações semânticas que gerem código intermediário para o comando **REPEAT-UNTIL** tomando como modelo o Exemplo 5.14.
- 3) Dado o programa abaixo e o esquema de tradução do Exemplo 5.6, mostre as tabelas de símbolos e as estruturas auxiliares que serão construídas até o ponto assinalado no programa.

```
x : real;
y : int;
proc P1;
    A : real;
    proc P2;
        B : int;
        S;
    proc P3;
        C : array [S] of real;
        D : int;
        S;
    => S;
    z : real;
```

- 4) Gere código intermediário para a expressão lógica " $A < B \text{ and } \neg C > D$ " segundo o esquema de tradução definido no Exemplo 5.11.
- 5) Gere código intermediário para o comando $X := A[X+Y,Z]$ segundo o esquema de tradução do Exemplo 5.9.
- 6) Dados os esquemas de tradução para comandos de controle (Exemplo 5.17). transforme-os de modo que eles possam ser processados no modo **top-down**.
- 7) Qual a finalidade dos símbolos não-terminais M e N na definição do comando **if-then-else** no esquema de tradução com **backpatching**.

- 8) Considerando os esquemas dos Exemplos 5.15 e 5.17, construa um grafo de dependência para a sentença:

```
if a < b then x := y else y = y + x
```

É possível fazer análise sintática e avaliar os atributos num mesmo passo? Justifique sua resposta.

- 9) Dada a sentença abaixo

```
if a < b then x := x + y else while c < d do c := c + x
```

e o esquema de tradução que gera código intermediário com rótulos para comandos de controle (Exemplo 5.17).

- Construa o grafo de dependência;
- Identifique a natureza dos atributos envolvidos;
- Gere código intermediário.

- 10) Modifique o esquema de tradução do Exemplo 5.15 para permitir a inclusão de expressões aritméticas em expressões relacionais.

- 11) Estenda o esquema de tradução do Exemplo 5.17 para implementar geração de código para o comando repeat-until, segundo a produção $S \rightarrow \text{repeat } S \text{ until } E$.

- 12) Estenda o esquema de tradução do Exemplo 5.21 para implementar geração de código com backpatching para o comando repeat-until, segundo a produção $S \rightarrow \text{repeat } L \text{ until } E$.

- 13) Para a sentença abaixo, gere código intermediário segundo os esquemas de tradução com *backpatching*.

```
while a < b do begin x := x + y; if x < z then a := x end
```

- 14) Qual a função do símbolo não-terminal M na produção $L \rightarrow L ; M S$? Explique as ações semânticas especificadas para essa produção.

- 15) Defina um esquema de tradução que implemente geração de código para o comando:

```
S → if (E) S1; S2; S3
```

cuja semântica é a seguinte: se o resultado da expressão aritmética E é menor que zero executa $S1$; se o resultado é igual a zero, executa $S2$; senão, executa $S3$.

Sugestão: Baseie-se no esquema de tradução para o comando WHILE do Exemplo 5.14.

- 16) Escreva um procedimento descendente recursivo que implemente o esquema de tradução a seguir:

```
S → if ( E.true = geralabel; E.false = geralabel ) E
    then ( S1.prox = S.prox ) S1
    else ( S2.prox = S.prox ) S2
    ( S.cod = E.cod || geracod(E.true ":") || S1.cod ||
      geracod("goto" S.prox) || geracod(E.false ":") || S2.cod )
```

6 Otimização de Código

Este capítulo trata do problema da geração de código eficiente, o qual envolve aspectos de uso de memória e de velocidade de execução. Esses aspectos são muitas vezes conflitantes, pois, em geral, ganhos no espaço utilizado implicam perdas no tempo de execução, e vice-versa.

Sabe-se que a geração de código ótimo é um problema indecidível. Na prática, o que se faz é utilizar heurísticas (técnicas *ad hoc* ou empíricas) para tentar otimizar o código ao máximo. Isso faz com que, em geral, a solução encontrada não seja a melhor possível.

Como a otimização de código consome tempo de compilação, ela somente deve ser implementada se a utilização do compilador realmente exige um código objeto eficiente. Por exemplo, num ambiente acadêmico, os estudantes compilam seus programas inúmeras vezes, mas os executam apenas algumas vezes, a fim de mostrar que o algoritmo implementado funciona. Logo, nesse ambiente, o código gerado pelos compiladores não necessita ser muito eficiente. Os compiladores, por sua vez, precisam ser bastante eficientes (com pequeno tempo de execução). Por outro lado, num ambiente de produção, no qual os programas são executados inúmeras vezes ao dia, deseja-se obter código objeto eficiente mesmo às custas de compilações mais demoradas. As opções de otimização tornam o compilador mais lento e encarecem o mesmo.

Normalmente, o processo de otimização desenvolve-se em duas fases: otimização do código intermediário e otimização do código objeto. A primeira inclui técnicas para eliminar atribuições redundantes, suprimir subexpressões comuns, eliminar temporários desnecessários, trocar instruções de lugar, etc.. de modo a obter um código intermediário menor. A otimização de código objeto é realizada através da troca de instruções de máquina por instruções mais rápidas e da melhor utilização de registradores.

Neste capítulo, serão vistas duas técnicas de otimização de código:

- otimização de blocos sequenciais de código intermediário por redução do número de instruções através de grafos acíclicos orientados; e
- transformações de expressões aritméticas visando ao uso otimizado de registradores.

6.1 Otimização de Código Intermediário

A fase de otimização do código intermediário vem logo após a fase de geração desse código e tem o objetivo de tomar o código **intermediário** mais apropriado para a produção de **código objeto** (código de máquina) eficiente, tanto em relação ao tamanho como ao tempo de execução.

A técnica de otimização de código intermediário apresentada nesta seção consiste em identificar segmentos sequenciais do programa, chamados **blocos básicos**, representá-los através de grafos dirigidos e submetê-los a um processo de otimização.

Definição 6.1 Bloco básico.

Um **bloco básico** é um trecho de programa que inicia por um comando “líder” e que não apresenta comandos de desvio, a não ser eventualmente o último, o qual pode ser um comando de desvio. Um comando é **líder** quando ele é referido por um **goto** ou quando ele segue a um **goro** (caso em que deverá possuir um rótulo). O primeiro comando de um programa também é um “líder”.

6.1.1 Representação de Blocos Básicos Através de Grafos

Todo bloco básico pode ser representado através de um grafo acíclico dirigido (GAD), também denominado **grafo de fluxo**. A representação de blocos básicos através de GADs permite:

- a identificação de subexpressões comuns num bloco;
- a identificação de variáveis usadas dentro do bloco mas que foram computadas fora (variáveis de entrada);
- a identificação de variáveis cujos valores são computados dentro do bloco e usadas fora do bloco (variáveis de saída).

Um GAD para um bloco básico é um grafo acíclico orientado (dirigido) tal que:

- as folhas (sumidouros) representam variáveis ou constantes;
- os **nodos internos** representam operadores e valores computados.

EXEMPLO 6.1 Bloco básico e GAD.

A Figura 6.1 mostra o bloco básico gerado para o comando de atribuição $x = (a + b) * (a - b)$, ao ser traduzido para código de três endereços, e o grafo dirigido correspondente. As variáveis

a e **b** são consideradas variáveis de entrada para o bloco, e a variável **x** é uma variável de saída calculada dentro do bloco.

$$\begin{aligned} t1 &= a + b \\ t2 &= a - b \\ t3 &= t1 * t2 \\ x &= t3 \end{aligned}$$

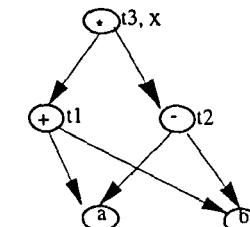


Figura 6.1 Bloco básico e GAD

6.1.2 Algoritmo para Construir o GAD de um Bloco

O algoritmo supõe que cada instrução (de três endereços) do código **intermediário** segue um dos seguintes três formatos: (i) $x = y \ op \ z$; (ii) $x = op \ y$; (iii) $x = y$. Instruções do tipo **if-goto** são tratadas como no caso (i).

Para cada instrução do bloco básico, execute os passos (1) e (2).

- 1) Se o **nodo y** ainda não existe no grafo, crie uma folha para **y**. Tratando-se do caso (i), faça o mesmo para **z**.
- 2) No caso (i), verifique se existe um **nodo op** com filhos **y** e **z** (nessa ordem). Caso exista, chame-o, também, de **x**; senão, crie um **nodo op** com nome **x** e dois arcos dirigidos do **nodo op** para **y** e para **z**.

No caso (ii), verifique se existe um **nodo op** com um único filho **y**. Se não existir, crie tal **nodo** e um arco orientado desse **nodo** para **y**. Chame de **x** o **nodo** criado ou encontrado.

No caso (iii), chame também de **x** o **nodo y**.

EXEMPLO 6.2 Bloco básico com subexpressões comuns e GAD correspondente.

A Figura 6.2 mostra o bloco básico e o GAD gerados para o comando de atribuição

$$y = ((a + b) * (a - b)) + ((a + b) * (a - c))$$

A subexpressão **(a + b)**, a qual aparece duas vezes na expressão sendo atribuída e no código de três endereços gerado, resulta em apenas uma subárvore no grafo dirigido.

$t1 = a + b$
 $t2 = a - b$
 $t3 = t1 * t2$
 $t4 = a + b$
 $t5 = a - c$
 $t6 = t4 * t5$
 $t7 = t3 * t6$
 $y = t7$

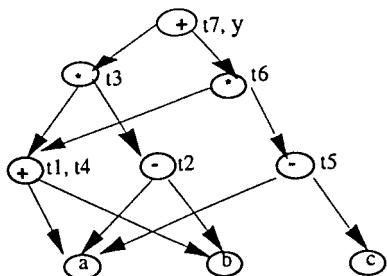


Figura 6.2 Bloco básico e GAD para $y = ((a+b) * (a-b)) + ((a+b) * (a-c))$

□

Pode-se evidenciar a **otimização** obtida através dos **GADs**, traduzindo-se o código de três endereços para código objeto e calculando-se a diferença de custo, medida em número de instruções, entre o código original e o código obtido a partir de um GAD. Para tal, usa-se um modelo de máquina simples, com apenas um acumulador, e **instruções** de carga (LOAD), de armazenamento (STORE), e de operações aritméticas (ADD, SUB, MULT e DIV).

EXEMPLO 6.3 Custo de código objeto.

Conforme referido acima, o custo de um segmento de código é o número de instruções de máquina desse segmento.

Seja o código de três endereços abaixo:

$$\begin{array}{ll}
 t1 = a + b & t5 = b - c \\
 t2 = a - b & t6 = t2 * t4 \\
 t3 = t1 * t2 & t7 = t6 * t5 \\
 t4 = a - c &
 \end{array}$$

O código objeto, não otimizado, gerado para uma máquina de um acumulador é o seguinte:

| | | | |
|---------|---------|---------|-----------------------|
| LOAD a | MULT t2 | SUB c | |
| ADD b | STO t3 | STO t5 | |
| STO t1 | LOAD a | LOAD t2 | Custo = 19 instruções |
| LOAD a | SUB c | MULT t4 | |
| SUB b | STO t4 | MULT t5 | |
| STO t2 | LOAD b | STO t7 | |
| LOAD t1 | | | |

6.1.3 Algoritmo para Gerar uma Sequência Otimizada de Código

O algoritmo abaixo produz uma ordenação dos nodos de um GAD, a qual corresponde ao código otimizado. A sequência ótima de instruções corresponde ao inverso da ordenação obtida.

- 1) Faça $L = \emptyset$ (lista vazia).
- 2) Escolha um **nodo** n que não esteja em L tal que, se existem arestas incidentes a n , essas se originam em **nodos** que já estão em L . Adicione n a L e vá para o passo 3. Se não existe tal n , encerra.
- 3) Se: (i) n é o último **nodo** adicionado a L e
 - (ii) a aresta mais à esquerda que se origina em n incide em um **nodo interno** m que não está em L e
 - (iii) todos os predecessores diretos de m estão em L
 Então adicione m a L e repita o passo (3).
- Senão, volte para o passo (2).

EXEMPLO 6.4 Código otimizado a partir de um GAD.

A Figura 6.3 mostra o grafo acíclico dirigido e a lista de **nodos** que representa a sequência (ao reverso) de código otimizado para o seguinte bloco básico, já considerado no Exemplo 6.1.

$$\begin{array}{ll}
 t1 = a + b & t5 = b - c \\
 t2 = a - b & t6 = t2 * t4 \\
 t3 = t1 * t2 & t7 = t6 * t5 \\
 t4 = a - c &
 \end{array}$$

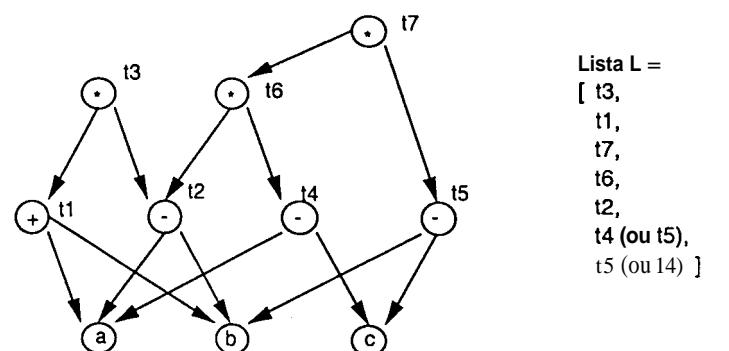


Figura 6.3 GAD e sequência otimizada (ao reverso)

Neste exemplo, pode-se iniciar a lista com um dos **nodos** t_3 ou t_7 . Iniciando com $L = [t_3]$, obtém-se a lista mostrada na Figura 6.3. O correspondente código intermediário otimizado é o seguinte:

$$\begin{array}{ll} t_5 = b - c & t_7 = t_6 * t_5 \\ t_4 = a - c & t_1 = a + b \\ t_2 = a - b & t_3 = t_1 * t_2 \\ t_6 = t_2 * t_4 \end{array}$$

O código objeto para o bloco básico otimizado é o seguinte:

| | | | | | | |
|------|-------|------|-------|------|-------|------------|
| LOAD | b | LOAD | a | LOAD | a | |
| SUB | c | SUB | b | ADD | b | |
| STO | t_5 | STO | t_2 | MULT | t_2 | Custo = 16 |
| LOAD | a | MULT | t_4 | STO | t_3 | |
| SUB | c | MULT | t_5 | | | |
| STO | t_4 | STO | t_7 | | | |

Observa-se que o número de instruções do código intermediário permaneceu o mesmo, porém o número de instruções do código objeto diminuiu de 19 para 16 pela simples troca de posição de instruções no **código intermediário**.

6.2 Otimização de Código para Expressões Aritméticas

Esta seção apresenta uma técnica que visa otimizar o código gerado a partir de expressões aritméticas levando em conta as 'propriedades comutativa e **associativa** dos operadores aritméticos.

Hipóteses simplificatórias:

A aplicação da técnica exige que as expressões aritméticas satisfaçam as seguintes restrições:

- (i) não há operandos repetidos na expressão;
- (ii) todos os operadores são binários.

A condição (ii) pode ser contornada com certa facilidade. Entretanto, a **otimização do código** para expressões que contêm dois ou mais **operandos iguais** torna-se muito difícil. Para a classe de blocos básicos que satisfazem as restrições acima, serão apresentados **algoritmos**

que produzem código otimizado em relação ao tamanho e ao número de acumuladores usados.

Deve-se notar que a restrição (i) implica que o grafo correspondente a um bloco seja **uma árvore** e que a restrição (ii) implica que essa árvore seja binária. Por exemplo, para o bloco correspondente à expressão $Z^*(X+Y)$, tem-se a árvore mostrada na Figura 6.4.

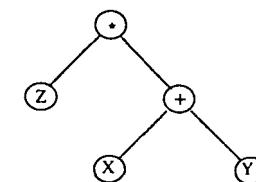


Figura 6.4 Árvore correspondente à expressão $Z^*(X+Y)$

A otimização obtida será evidenciada comparando-se os custos (número de instruções) do código original com os do código resultante da aplicação da técnica. Para tal, gerar-se-á código para um modelo de máquina com N acumuladores, $N \geq 1$.

Máquina alvo

O modelo de máquina aqui considerado opera com quatro instruções:

| | |
|---------------------|---|
| LOAD M, A | carrega o conteúdo da posição de memória M no acumulador A; |
| STORE A, M | armazena o conteúdo do acumulador A na posição de memória M; |
| OP θ A, M, B | aplica o operador θ aos conteúdos do acumulador A e da posição M e armazena o resultado no acumulador B. |
| OP θ A, B, C | aplica o operador θ aos conteúdos dos acumuladores A e B e armazena o resultado no acumulador C. |

Árvore de sintaxe

A técnica requer que **as expressões aritméticas** sejam representadas através de árvores de sintaxe. **Algoritmos** para obtenção de árvores de sintaxe já foram apresentados nas seções 4.3 e 4.6.

Determinação de valores dos **nodos** da árvore de sintaxe

O valor de um **nodo** de uma árvore de sintaxe, denotado por V , é obtido como segue:

- (a) se um **nodo** n refere-se a um operando A , então $V(n) = A$;
- (b) se um **nodo** n refere-se a um operador θ , então $V(n) = \theta V_1 V_2$,
onde V_1 e V_2 são os valores dos descendentes diretos de n .

O **valor de um árvore** é o valor do seu **nodo** raiz. Para a árvore da Figura 6.4, o valor da árvore é $*Z+XY$.

O algoritmo de geração de código a ser apresentado depende de um processo de colocação de rótulos nos **nodos** da árvore de sintaxe. Esses rótulos são números inteiros que indicam, para cada **nodo** não-terminal n , o número de acumuladores necessários para a avaliação da subárvore gerada por n . O algoritmo apresentado na seção seguinte coloca esses rótulos em uma árvore de sintaxe.

6.2.1 Algoritmo para Obter o Número de Acumuladores

O algoritmo a seguir rotula cada **nodo** interior de uma árvore de sintaxe com um valor que indica o número mínimo de acumuladores requeridos para avaliar, sem a necessidade de salvar resultados intermediários, a subárvore que tem esse **nodo** como raiz. O processo de atribuir rótulos à árvore de sintaxe inicia nas folhas seguindo em direção à raiz.

- (1) Se o **nodo** considerado é terminal (corresponde a operando) e
 - (i) se é o descendente direto esquerdo de seu predecessor ou é a raiz (caso em que a árvore contém apenas um **nodo**), então atribuir o rótulo 1 a esse **nodo**;
 - (ii) se é o descendente direto direito, então atribuir o rótulo 0 (zero) a esse **nodo**;
- (2) Considere um **nodo** n tendo descendentes diretos n_1 e n_2 , com rótulos l_1 e l_2 . Se $l_1 \neq l_2$, então o rótulo de n é o maior entre l_1 e l_2 . Se $l_1 = l_2$, o rótulo de n é $l_1 + 1$.

EXEMPLO 6.5 Árvore de sintaxe e número de acumuladores

A árvore de sintaxe correspondente à expressão $A*(B-C)/(D*(E-F))$, anotada com os rótulos é mostrada na Figura 6.5. Nessa representação intermediária, o rótulo de um **nodo** interior indica o número mínimo de acumuladores requeridos para avaliar, sem a necessidade de salvar resultados intermediários, a subárvore que tem esse **nodo** como raiz.

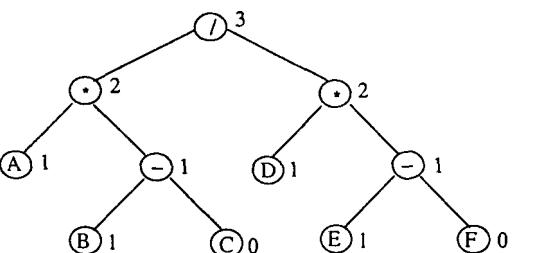


Figura 6.5 Árvore de sintaxe rotulada com número de acumuladores

6.2.2 Geração de Código para Máquina com N Acumuladores

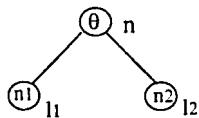
O algoritmo a seguir transforma uma árvore de sintaxe com rótulos em um programa Assembler para a máquina de 4 instruções e N acumuladores descrita anteriormente. O algoritmo consiste em aplicar a rotina recursiva $COD(n, i)$ onde n é um **nodo** e i é um número entre 1 e N . O número i indica que os acumuladores A_i, A_{i+1}, \dots, A_N estão disponíveis no momento para calcular a expressão correspondente ao **nodo** n . A rotina $COD(n, i)$ gera uma sequência de instruções Assembler que calcula o valor do **nodo** n , deixando o resultado no acumulador A_i .

Inicialmente, COD é chamada com os parâmetros n_0 e I , onde n_0 é o **nodo** raiz. O algoritmo executado pela rotina COD é o seguinte:

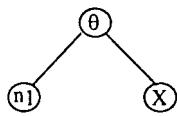
$COD(n, i)$:

- 1) Se n é um **nodo** terminal (folha da árvore), então execute o passo 2; senão, execute o passo 3.
- 2) Nesse caso, n é terminal e é um descendente esquerdo direto (ou é a raiz, no caso de uma árvore trivial). Se o nome do **nodo** é X , então:
 $COD(n, i) = "LOAD X, A_i"$

- 3) Caso em que n é um **nodo interior** apresentando a seguinte configuração:



- a) Se $l_2=0$ (nodo n_2 é um terminal), então, neste caso, tem-se a configuração:



onde X é o nome de n_2 , e n_1 pode ser ou terminal ($l_1=1$), ou a raiz de uma subárvore ($l_1 \geq 1$). Como n_2 é diretamente acessível a partir de θ , precisa-se, somente, gerar código relativo à subárvore de n_1 e aplicar a operação θ ao resultado dessa geração (em um acumulador) e a X :

$$\text{COD}(n, i) = \text{COD}(n_1, i) "OP\theta Ai,X,Ai"$$

O código de n é o código de n_1 (cujo resultado é colocado em A_i) seguido da operação θ aplicada a A_i e X , cujo resultado é colocado em A_i (agora disponível).

- b) Se $l_1 < l_2$ e $l_1 < N$, então:

$$\begin{aligned} \text{COD}(n, i) &= \text{COD}(n_2, i) \\ &\quad \text{COD}(n_1, i+1) \\ &\quad "OPB Ai+1,Ai,Ai" \end{aligned}$$

- c) Se $l_2 \leq l_1$ e $l_2 < N$, então:

$$\begin{aligned} \text{COD}(n, i) &= \text{COD}(n_1, i) \\ &\quad \text{COD}(n_2, i+1) \\ &\quad "OP\theta Ai,Ai+1,Ai" \end{aligned}$$

- d) Se $N \leq l_1$ e $N \leq l_2$, então:

$$\begin{aligned} \text{COD}(n, i) &= \text{COD}(n_2, i) \\ T &= \text{NOVOTEMP} \\ &"STORE Ai,T" \\ \text{COD}(n_1, i) \\ &"OPB Ai,T,Ai" \end{aligned}$$

Nos passos 3.b e 3.c, gera-se código primeiro para os ramos que necessitam mais acumuladores. Dessa forma otimiza-se o uso de acumuladores. No passo 3.d não há mais acumuladores disponíveis, portanto, tem-se que utilizar temporários.

EXEMPLO 6.6 Código para a expressão $Z^*(X+Y)$ usando 2 acumuladores ($N=2$).

A árvore de sintaxe anotada é mostrada na Figura 6.6. A sequência de chamadas de COD, os passos executados e o código gerado são os seguintes:

| <i>Chamada de COD</i> | <i>Passo executado</i> |
|-----------------------|------------------------|
| (*,1) | 3.c |
| (Z,1) | 2 |
| (+,2) | 3.a |
| (X,2) | 2 |

Código gerado:

```

LOAD Z, A1
LOAD X, A2
ADD A2, Y, A2
MULT A1, A2, A1
  
```

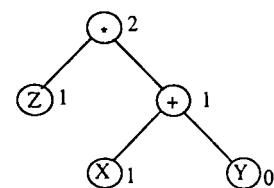


Figura 6.6 Árvore de sintaxe rotulada com o número de acumuladores

EXEMPLO 6.7 Código para a expressão $A^*(B-C)/(D^*(E-F))$ com $N=2$.

A árvore anotada correspondente a esta expressão é mostrada na Figura 6.5. A tabela que mostra as chamadas de COD e o código gerado, para o caso de 2 acumuladores, é apresentada a seguir. O custo associado ao código (número de instruções) é 10.

| Chamada de COD | l_1 | l_2 | Passo | Código gerado |
|----------------|----------|----------|-------|---|
| (/,1) | 2 | 0 | 3.d | |
| (*,1) | 1 | 1 | 3.c | |
| (D,1) | terminal | terminal | 2 | LOAD D, A1 |
| (-,2) | 1 | 0 | 3.a | |
| (E,2) | terminal | terminal | 2 | LOAD E, A2 SUB A2, F, A2 MULT A1, A2, A1 STORE A1, T |
| (*,1) | 1 | 1 | 3.c | |
| (A,1) | terminal | terminal | 2 | LOAD A, AI |
| (-,2) | 1 | 0 | 3.a | |
| (B,2) | terminal | terminal | | LOAD B, A2 SUB A2, C, A2 MULT A1, A2, A1 DIV A1, T, A1 |

EXEMPLO 6.8 Código para a expressão $A^*(B-C)/(D^*(E-F))$ com $N=1$.

Trata-se da mesma árvore do exemplo anterior, porém usando um único acumulador ($N=1$).

| Chamada de COD | l_1 | l_2 | Passo | Código Gerado |
|----------------|----------|----------|-------|--|
| (/,1) | 2 | 2 | 3.d | |
| (*,1) | 1 | 1 | 3.d | |
| (-,1) | 1 | 0 | 3.a | |
| (E,1) | terminal | terminal | 2 | LOAD E, A1 SUB A1, F, A1 STORE A1, T1 LOAD D, A1 MULT A1, T1, A1 STORE A1, T2 |
| (*,1) | 1 | 1 | 3.d | |
| (-,1) | 1 | 0 | 4 | |
| (B,1) | terminal | terminal | 2 | LOAD B, A1 SUB A1, C, A1 STORE A1, T3 LOAD A, A1 MULT A1, T3, A1 DIV A1, T2, A1 |

Neste caso, o custo do código resultou em 12, por terem sido necessárias mais duas instruções STORE.

Pode-se provar que, para uma árvore de sintaxe cujo rótulo da raiz é k, e havendo N acumuladores disponíveis. se $k \leq N$. então o código correspondente gerado não utiliza instruções STORE.

Isso é consequência direta do algoritmo de geração de código, no qual instruções STORE são introduzidas somente no passo 3.d, o qual. se $k \leq N$, nunca é executado. Se $k > N$, pode-se calcular a priori o número de instruções LOAD e STORE necessárias para a avaliação da expressão que a árvore representa.

Definição 6.2 Nodos tipo 1 e tipo 2.

Seja T uma árvore de sintaxe e N o número de acumuladores disponíveis. Um nodo de T é do tipo 1 se cada um de seus descendentes diretos tem rótulo igual ou maior que N. Um nodo é do tipo 2 se é terminal e tem rótulo 1.

Pode-se provar que, para uma árvore de sintaxe T e N acumuladores:

- número de nodos tipo 2 = número de LOAD's
- número de nodos tipo 1 = número de STORE's
- soma do número de nodos tipo 1, tipo 2 e de nodos interiores = número total de instruções.

Definição 6.3 Custo de uma árvore de sintaxe.

O custo de uma árvore de sintaxe pode ser definido como a soma do número de nodos interiores com o número de nodos do tipo 1 e com o número de nodos do tipo 2.

EXEMPLO 6.9 Custo de uma árvore de sintaxe.

A árvore da Figura 6.5 tem 5 nodos interiores (operadores) e 4 nodos do tipo 2 (LOAD's). O número de nodos tipo 1 (STORE's) varia com o número de acumuladores a serem considerados.

Para $N = 1 \Rightarrow 3$ nodos tipo 1 $\Rightarrow 12$ instruções
(3 STORE's, 4 LOAD's, 5 OP's)

Para $N = 2 \Rightarrow 1$ nodo tipo 1 $\Rightarrow 10$ instruções
(1 STORE, 4 LOAD's, 5 OP's)

Para $N = 3 \Rightarrow 0$ nodo tipo 1 $\Rightarrow 9$ instruções
(0 STORE's, 4 LOAD's, 5 OP's)

Muitas vezes, o custo de uma árvore de sintaxe pode ser diminuído mediante o uso de certas propriedades algébricas, como a comutatividade e a associatividade. Por exemplo, se n é um **nodo** interior correspondendo a um operador $+$, é possível trocar a ordem dos descendentes de n, aplicando-se a lei da comutatividade. A Figura 6.7 ilustra as **transformações** que podem ser feitas numa árvore quando o operador é comutativo e associativo.

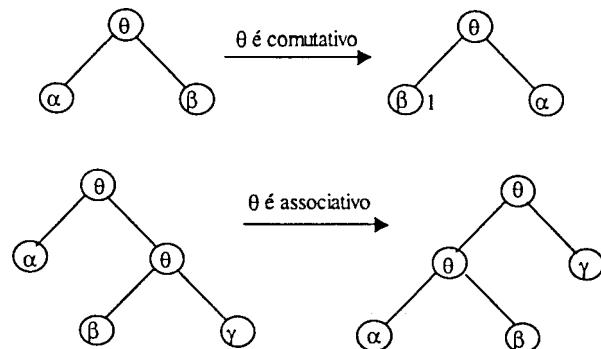


Figura 6.7 Transformações por comutatividade e associatividade

Definição 6.4 Equivalência de árvores.

Duas árvores são ditas **equivalentes** se uma pode ser obtida a partir da outra, aplicando-se as propriedades associativa e comutativa.

Assim, dada uma árvore de sintaxe, é interessante iniciar pela procura de uma árvore equivalente de menor custo, usando as propriedades ilustradas na Figura 6.7 e aplicando-se a função COD sobre a árvore resultante, para obter um código otimizado.

Como as leis acima preservam o número de operadores (e portanto o número de **nodos** internos), deve-se buscar uma árvore equivalente na qual a soma dos **nodos** tipo 1 e 2 seja minimizada.

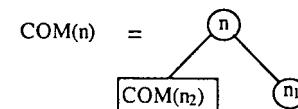
A seguir, são apresentados algoritmos para tal. Primeiramente, para o caso em que só há operadores comutativos e depois para o caso em que operadores comutativos podem também ser associativos.

Algoritmo para obter a árvore de menor custo em relação a operações comutativas

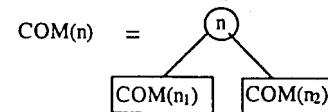
Dada uma árvore T, o algoritmo a seguir obtém uma árvore equivalente de custo mínimo, fazendo **transformações** sobre os operadores comutativos de T. A árvore de custo mínimo é obtida pela função **COM(n)**, onde n é um **nodo** da árvore. Inicialmente, COM é chamada com o parâmetro n_0 , raiz da árvore a ser transformada.

COM(n):

- (1) se n é **nodo** terminal, $\text{COM}(n) = n$;
- (2) se n é **nodo** interior:
 - (2.1) se o operador de n é comutativo, e n tem descendentes n_1 e n_2 nessa ordem. Se n_1 é terminal e n_2 é **nodo** interior, então:



- (2.2) em todos os outros casos:



EXEMPLO 6.10 Árvore de custo mínimo em relação a operadores comutativos

A árvore da Figura 6.5 tem somente o operador $*$ comutativo. A aplicação do procedimento COM a essa árvore resulta na árvore da Figura 6.8.

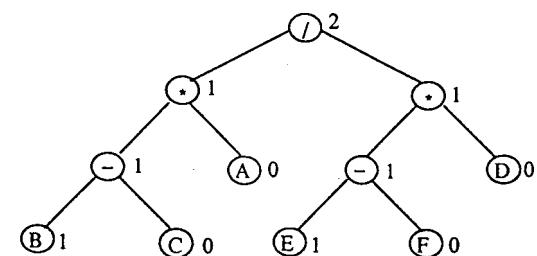


Figura 6.8 Árvore de custo mínimo em relação a operadores comutativos

Para $N = 2$, a árvore original (Figura 6.5) possui um **nodo tipo 1**, 4 **nodos tipo 2** e 5 **nodos interiores** (ver Exemplo 6.9). o que resultava num custo de 10 **instruções**. A árvore otimizada em relação às operações comutativas (Figura 6.8) não possui **nodos tipo 1** e tem somente dois **nodos tipo 2**, o que resulta num custo de apenas 7 **instruções** (ganho de 30%, portanto).

□

Quando alguns operadores são, ao mesmo tempo, comutativos e associativos, é possível aplicar outras transformações à árvore de sintaxe para reduzir seu custo.

Definição 6.5 CLUSTER.

Seja T uma árvore de sintaxe. Um conjunto S de nodos de T forma um **cluster** se:

- cada **nodo** de S é um **nodo** interior com o mesmo operador associativo e comutativo;
- os **nodos** de S , com suas arestas de conexão, formam uma árvore;
- S é maximal em relação às condições a) e b).

A raiz de um **cluster** é a raiz da árvore formada por seus **nodos** (condição b). Os descendentes diretos de um **cluster** S são **nodos** não pertencentes a S , mas são descendentes diretos de um **nodo** de S .

EXEMPLO 6.11 Clusters de uma árvore de sintaxe.

Como os operadores $+$ e $*$ são comutativos e associativos, a árvore da Figura 6.9 possui os 3 **clusters** tal como indicados.

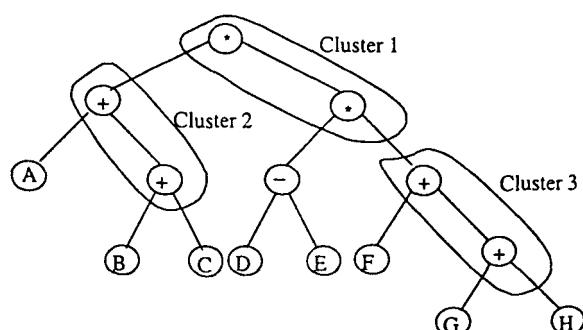


Figura 6.9 **Clusters** de uma árvore de sintaxe

Considere o **cluster** 2 acima com seus descendentes diretos (nodos A, B e C). Como o operador $+$ é comutativo e associativo, a ordem dos operandos A, B e C pode ser qualquer, que o valor do **cluster** não é alterado.

Definição 6.6 Árvore associativa.

Seja T uma árvore de sintaxe. A **árvore associativa** de T , referida por T' , é formada pela substituição de cada **cluster** de T por um único **nodo** n tendo o mesmo operador (comutativo e associativo). Os descendentes diretos de n em T' são os descendentes diretos do **cluster** correspondente em T .

EXEMPLO 6.12 Árvore associativa.

A Figura 6.10 é uma árvore de sintaxe com seus **clusters** evidenciados, e a Figura 6.11, apresenta a árvore associativa correspondente. Convém observar que a árvore associativa deixa de ser binária se a árvore de sintaxe possui **clusters**.

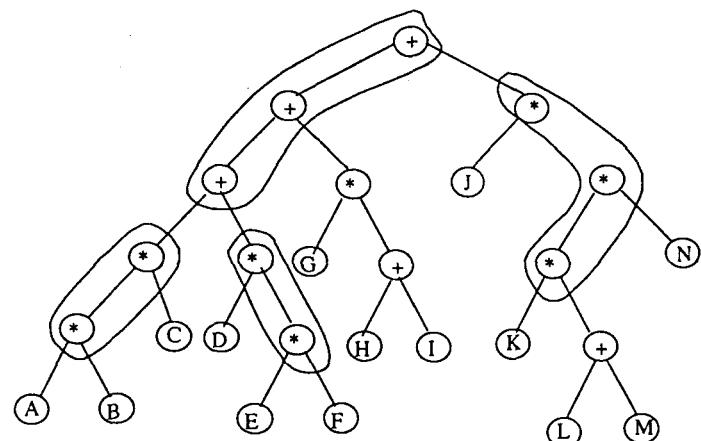
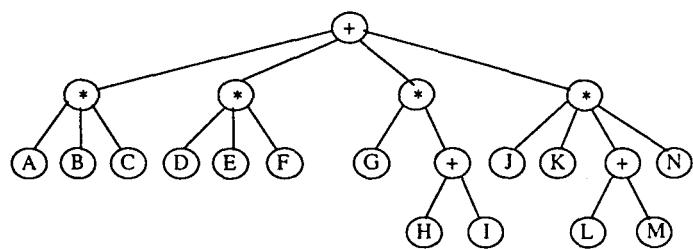


Figura 6.10 Árvore de sintaxe com **clusters**

Os próximos passos no processo de otimização são: associar rótulos aos **nodos** da árvore associativa e restaurar a árvore associativa numa árvore binária.

Figura 6.11 Árvore associativa sem *clusters***Algoritmo** para atribuir rótulos aos **nodos** de uma árvore associativa

Este algoritmo deve ser aplicado a partir das folhas em direção à raiz da árvore.

- (1) Se o **nodo** é terminal e é descendente esquerdo, então recebe o rótulo 1; todos os outros **nodos** terminais recebem o rótulo 0;
- (2) Se n é um **nodo** interior, com descendentes n_1, n_2, \dots, n_m rotulados de l_1, l_2, \dots, l_m , $m \geq 2$:
 - a) se um dos l_1, l_2, \dots, l_m é maior que os outros, esse será o rótulo de n,
 - b) se o operador de n é comutativo e se n é um **nodo** interior com $l_i = 1$ e $n_1, \dots, n_{i-1}, n_{i+1}, \dots, n_m$ são terminais, então o rótulo de n será 1.
 - c) Se o caso b não se aplica, e se $l_i = l_j$ para $i \neq j$ e l_i é maior ou igual a todos os outros l_k 's, então o rótulo de n será $l_i + 1$.

EXEMPLO 6.13 Árvore associativa rotulada.

A Figura 6.12 é a árvore associativa do exemplo anterior rotulada de acordo com o **algoritmo** apresentado.

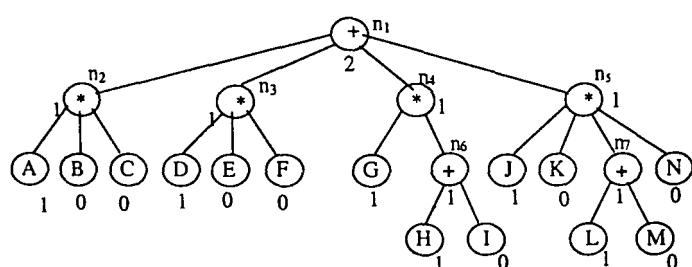
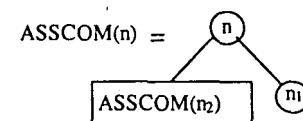


Figura 6.12 Árvore associativa rotulada

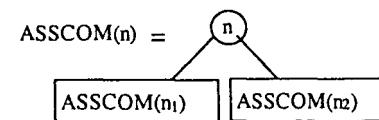
Algoritmo para obter a árvore de sintaxe de custo mínimo

O algoritmo abaixo obtém a árvore de menor custo equivalente a uma árvore de sintaxe T dada.

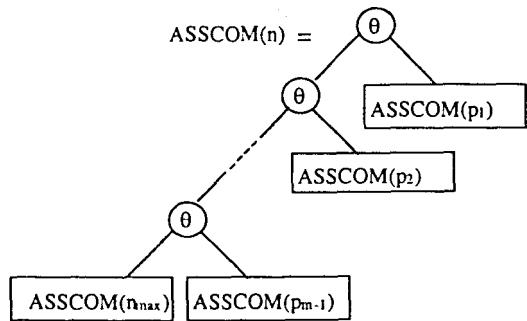
- (1) Construir a árvore associativa T' e rotular os **nodos**.
- (2) Aplicar a função **ASSCOM(n)**, definida abaixo, iniciando com n = raiz de T':
 - (a) Se n é terminal, **ASSCOM(n) = n**.
 - (b) Se n é **nodo** interior:
 - (b.1) Se n tem dois descendentes diretos n_1 e n_2 (nessa ordem) e o operador de n é comutativo (e talvez associativo).
 - (i) Se n_1 é terminal e n_2 é **nodo** interior, então **ASSCOM(n) = ASSCOM(n₂)**
 - (ii) Nos outros casos, **ASSCOM(n) = ASSCOM(n₁)**



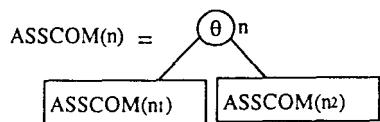
- (ii) Nos outros casos, **ASSCOM(n) = ASSCOM(n₁)**



- (b.2) Se θ, operador de n, é comutativo e associativo, e se n tem descendentes diretos n_1, n_2, \dots, n_m , $m \geq 3$, seja n_{\max} o nodo n_i com o maior rótulo. Se dois ou mais nodos têm o mesmo rótulo maior, escolha n_{\max} um nodo interior. Sejam p_1, p_2, \dots, p_{m-1} , os nodos restantes em $\{n_1, \dots, n_m\} - \{n_{\max}\}$ (em qualquer ordem). Nesse caso, resulta em:



(b.3) Se o operador θ de n não é comutativo, nem associativo, então:

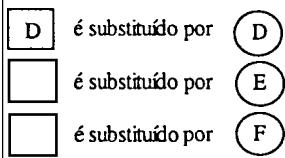
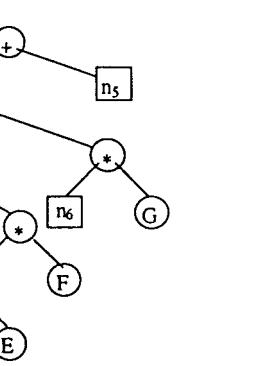


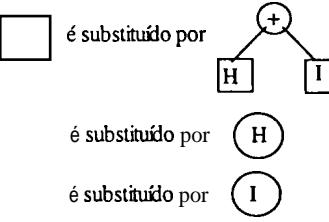
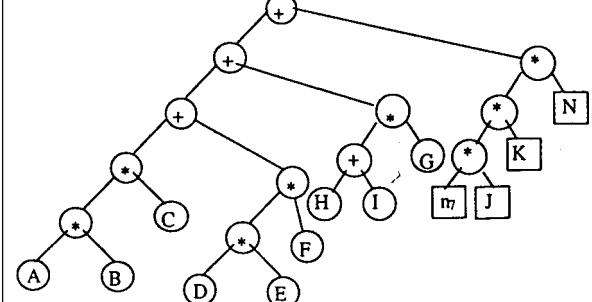
EXEMPLO 6.14 Árvore de sintaxe de custo mínimo.

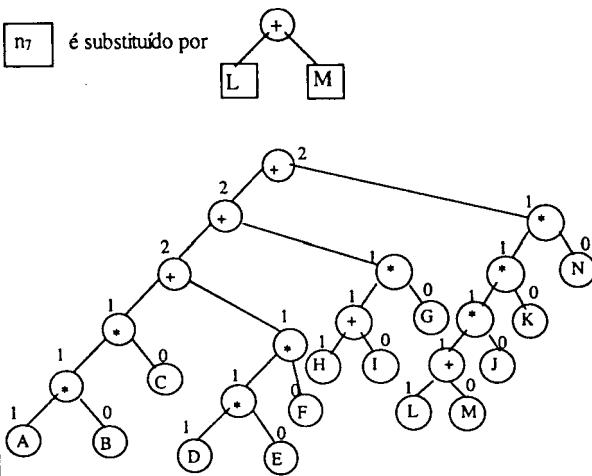
Este exemplo mostra a aplicação do algoritmo anterior à árvore associativa da Figura 6.12.

| Arg. | Label | Passo | Árvore |
|-------|-------|----------------------------|---|
| n_1 | 2 | 2b.2 $n_2 \in n_{\max}$ | <pre> graph TD root1[+] --- l1[+] l1 --- n2[n2] l1 --- n3[n3] r1[+] --- n5[n5] </pre> |

| Arg. | Label | Passo | Árvore |
|-------|-------|--------------------------|--|
| n_2 | 1 | 2b.2 $A \in n_{\max}$ | <pre> graph TD root2[+] --- l2[*] l2 --- A[A] l2 --- B[B] r2[+] --- n3[n3] r2 --- C[C] </pre> |
| A | 1 | 2a | $A \rightarrow A$ |
| B | 0 | 2a | $B \rightarrow B$ |
| C | 0 | 2a | $C \rightarrow C$ |
| n_3 | 1 | 2b.2 $D \in n_{\max}$ | <pre> graph TD root3[+] --- l3[*] l3 --- A1[A] l3 --- B1[B] r3[+] --- C1[C] r3 --- D1[*] D1 --- D[D] D1 --- E[E] r3 --- F1[F] </pre> |

| | | | |
|----------------|-------------|----------------|---|
| D E F | 1 0 0 | 2a 2a 2a |  |
| n ₄ | 1 | 2b.1(i) |  |

| | | | |
|----------------|-------------|-----------------------------|---|
| H I | 1 1 0 | 2b.1(ii) 2a 2a |  |
| n ₅ | 1 | b.2 $n_7 \in \Pi_{\max}$ |  |

| | | | |
|---|-------------------------------|--------|---|
| n ₇ L M J K N | 1 | b.1(i) |  |
| | | | Árvore binária de custo mínimo |

Rótulo da raiz da árvore inicial = 3

Rótulo da raiz da árvore obtida = 2

Para N = 2

Número de **nodos** do tipo 1 da árvore inicial = 2

Número de **nodos** do tipo 1 da árvore obtida = 0

Número de **nodos** do tipo 2 da árvore inicial = 8

Número de **nodos** do tipo 2 da árvore obtida = 4

Número de **nodos** interiores em ambas as árvores = 13

Custo da árvore inicial = 23

Custo da árvore obtida = 17

EXERCÍCIOS

1) Considere a seguinte sequência de comandos:

```
A := B + C * D;
B := A * (C * D);
C := (C * D) * 2;
if B > C * D goto L;
```

Gere código intermediário (de três endereços) otimizado pela aplicação do método de construção de GAD's para blocos básicos.

2) Dada a expressão

$$(a + (b * c * d)) / (e + f * g),$$

mostre a geração de código, passo a passo, para uma máquina com 2 registradores,

- a) aplicando diretamente o algoritmo COD (n, i);
- b) aplicando inicialmente o algoritmo COM (n);
- c) aplicando inicialmente o algoritmo ASCOM (n).

7 Gerência de Memória

No contexto da compilação, o termo gerência de memória refere-se às atividades de alocação e liberação de memória para dados (variáveis e constantes), em tempo de compilação e em tempo de execução. A alocação e a liberação de memória para dados são gerenciadas por um conjunto de rotinas (*run-time support package*) carregado junto com o código objeto gerado. Um programa objeto compõe-se de código e área de dados, conforme ilustra a Figura 7.1. O código é produzido pelo gerador de código objeto, e a área para dados é reservada pelas funções do gerente de memória.

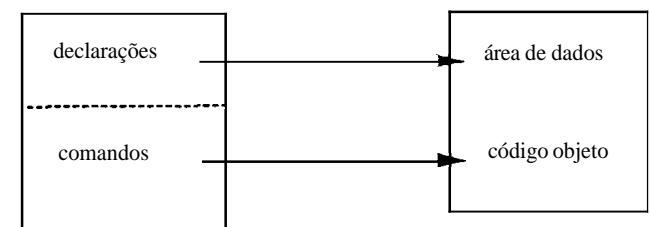


Figura 7.1 Composição de um programa objeto

O **armazenamento** de um dado na memória em tempo de execução depende do tipo desse dado. Em geral, tipos de dados elementares tais como caracteres, inteiros e reais ocupam, cada um, uma palavra de memória. Entretanto, agregados tais como **vetores**, cadeias de caracteres e registros vão ocupar várias posições contíguas de memória.

Se o tipo e o tamanho das variáveis são conhecidos em tempo de compilação, a reserva de espaço de memória é decidida na compilação; caso contrário, o espaço será alocado durante a execução do programa objeto.

7.1 Estratégias de Alocação de Memória

A memória para o programa compilado deve conter:

- o código objeto gerado;
- o espaço para as variáveis globais (área estática);

- a pilha para ativação de procedimentos;
- o espaço para memória dinâmica (heap).

Normalmente, esses componentes são organizados conforme mostra a Figura 7.2.

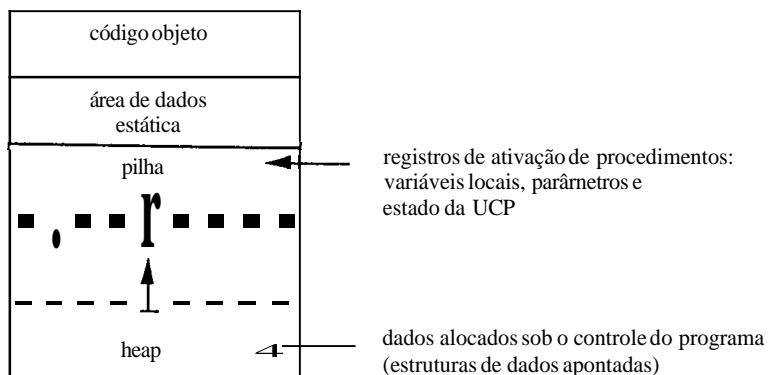


Figura 7.2 Organização da memória ocupada por um programa

A reserva de memória para dados pode ser feita de 3 maneiras:

- **Alocação estática.** Se o tipo (ou comprimento) do dado é conhecido em tempo de compilação e não é modificado durante a execução do programa, a reserva de memória é feita durante a compilação, de forma estática.
- **Alocação em pilha.** Áreas para dados locais de procedimentos (subrotinas ou funções) devem ser alocadas dinamicamente. Embora o tamanho seja, muitas vezes, conhecido em tempo de compilação, a alocação de espaço de memória somente pode ser realizada em tempo de execução, porque a ordem de chamadas é determinada pela execução do programa. Essas áreas são, em geral, alocadas numa estrutura em pilha (pilha de ativação de procedimentos), a qual pode ser a própria pilha de execução da máquina.
- **Alocação dinâmica.** Para as estruturas de dados referenciadas através de ponteiros, as áreas também são reservadas dinamicamente. Essas áreas são alocadas e liberadas, sob o controle do programa, por comandos específicos (por exemplo, "new" e "dispose", em

Pascal). Normalmente, essas estruturas são alocadas na área denominada "heap", que cresce no sentido contrário ao da pilha.

7.2 Alocação em Memória Pilha

A memória de pilha é usada para armazenamento das informações relativas às chamadas de procedimentos. Para cada chamada, é empilhado um registro de ativação (RA), o qual contém informações sobre o estado da UCP no momento da chamada, uma área de dados para as variáveis locais, uma área para parâmetros e alguns ponteiros de controle da própria pilha. Em cada retorno de procedimento, é desempilhado um registro de ativação. A composição de um registro de ativação é mostrada na Figura 7.3.

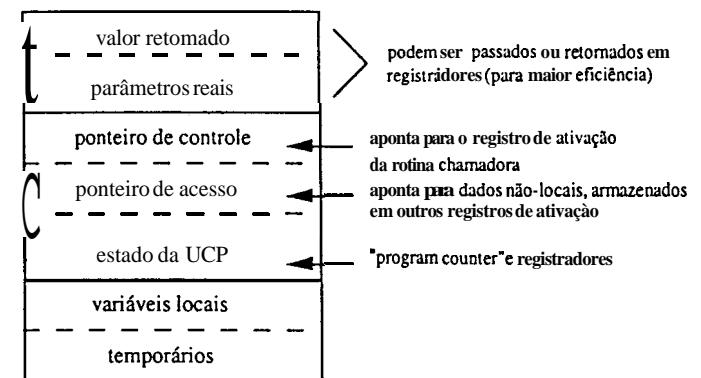


Figura 7.3 Registro de ativação de um procedimento

EXEMPLO 7.1 Pilha de ativação

Considere o programa que calcula o valor de e^x através da série:

$$e^x := x^0 + x^1 / 1! + x^2 / 2! + x^3 / 3! + x^4 / 4! +$$

com limite de erro tolerado. Os valores de x e do erro tolerado são dados do programa. O programa escreve o valor de x e o valor de e^x calculado através da série. O programa contém dois procedimentos: *fatorial* e *somatorio*.

```

program expoX;

var x, erro, expX: real;

function factorial(n: integer): integer;
begin
  if (n = 0) or (n = 1)
  then factorial := 1
  else factorial := n * factorial(n - 1)
end;

function somatorio(x: real; tol: real): real;
var termo, soma: real;
n: integer;
begin
  n := 0; soma := 0;
  repeat
    termo := (x ** n) * factorial(n);
    soma := soma + termo;
    n := n + 1;
  until termo <= tol;
  somatorio := soma;
end;

begin
  x := 2.5;
  erro := 0.0001;
  expX := somatorio(x, erro);
  writeln('Valor de X = ', x, ' Valor de expX = ', expX);
end.

```

A pilha de ativação, em alguns momentos da execução do programa *expoX*, é mostrada na Figura 7.4.

O

O código gerado para uma chamada de procedimento deve conter **instruções** para criar (empilhar) o registro de ativação correspondente. O código de retomo deve **destruir** (desempilhar) o registro de ativação, deixando na pilha apenas o valor de retomo e os **parâmetros** (essa parte é destruída pelo procedimento chamador).

Os códigos de chamada e de retomo **estão divididos** entre o procedimento **chamador** e o procedimento **chamado**. Não existe **uma divisão pré-estabelecida** sobre o que cabe ao **chamador** e o que cabe ao **chamado**, pois essa decisão depende da linguagem fonte, do sistema **operacional** e da máquina em questão.

O acesso às informações na pilha é efetuado através de dois ponteiros: **topo_RA** e **topo-pilha** (ver Figura 7.5). O primeiro, **topo_RA**, refere-se ao registro de ativação que está no topo da pilha. O segundo, topo-pilha, indica a posição a partir da qual será empilhado o registro de ativação da próxima rotina a ser chamada. O código para computar topo-RA e topo-pilha é gerado em tempo de compilação, tendo por base os tamanhos dos campos reservados para construir o registro de ativação da rotina chamada.

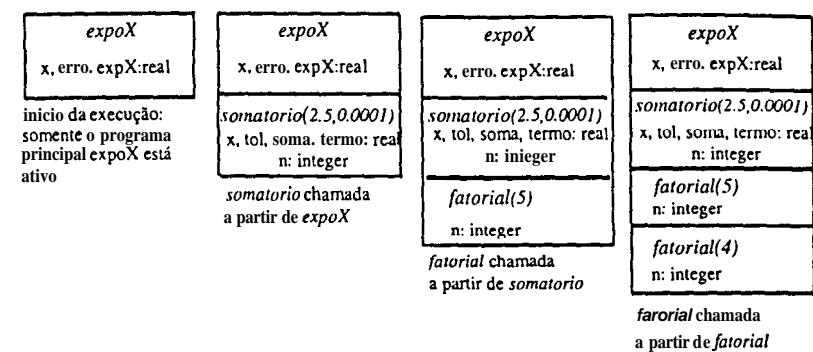


Figura 7.4 Pilha de ativação

Em geral, o código de chamada inclui as seguintes tarefas (ver Figura 7.6):

- 1) a rotina chamadora avalia os **parâmetros** reais e os coloca na pilha;
- 2) a rotina chamadora armazena o endereço de retomo e o valor antigo do apontador **topo_RA** no registro de ativação da rotina chamada e atualiza o valor do ponteiro topo-RA;
- 3) a rotina chamada salva valores de registradores e outras informações do estado da máquina;
- 4) a rotina chamada inicializa variáveis locais e começa sua execução.

O código de retomo deve incluir as seguintes tarefas:

- 1) a rotina chamada armazena o valor de retomo logo após o registro de ativação da rotina chamadora;
- 2) a rotina chamada restaura o apontador **topo_RA** e os registradores da máquina e desvia para o endereço de retomo dentro da rotina chamadora;
- 3) a rotina chamadora copia o valor retomado no seu próprio registro de ativação.

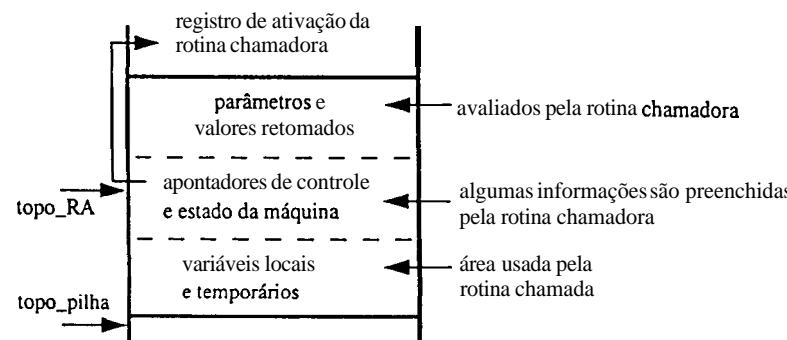


Figura 7.5 Ponteiros de acesso

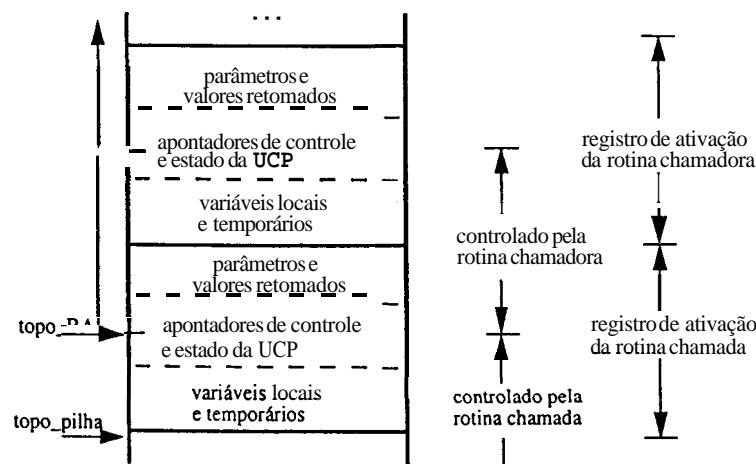


Figura 7.6 Distribuição de controle entre as rotinas chamadora e chamada

Para **matrizes** cujo tamanho somente é conhecido em tempo de execução, não **são** reservados espaços no registro de ativação, mas são criados apontadores que conterão os endereços dessas quando do conhecimento de seus tamanhos (ver Figura 7.7).

7.3 Acesso a Variáveis Não-Locais

As regras de escopo da linguagem determinam o tratamento de referências a **identificadores não-locais** ao procedimento. Linguagens de programação tais como Pascal e C implementam

a regra de "escopo léxico" (ou "escopo estático"), que permite conhecer o escopo de um identificador pelo exame do programa fonte. Lisp, APL e Snobol aplicam o "escopo dinâmico", pelo qual o escopo de um identificador somente é conhecido em tempo de execução.

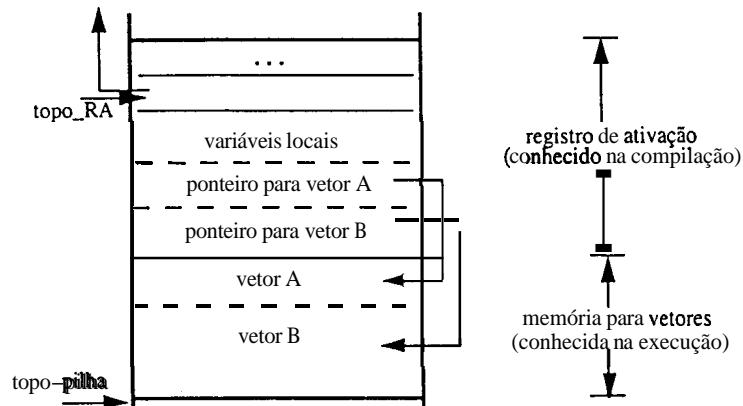


Figura 7.7 Alocação de espaço para matrizes

Escopo estático e rotinas não-aninhadas

Nas linguagens de programação que não permitem **embutimento** de rotinas, que é o caso das linguagens C e Fortran, a memória para as **variáveis declaradas fora das rotinas** (variáveis globais) pode ser **alocada** estaticamente. A posição relativa dessa área de memória é conhecida em tempo de compilação. Se um identificador é declarado externamente ao corpo da rotina, seu endereço estático é utilizado. Qualquer outro nome terá sido declarado localmente e **terá** seu endereço no registro de ativação que está no topo da pilha.

Uma vantagem importante desse esquema é que rotinas podem ser livremente passadas como **parâmetros** e retomadas como resultados, pois seus endereços estáticos **são** conhecidos. Qualquer nome não-local a uma rotina é não-local a todas as outras, e seu endereço estático pode ser usado por todas as funções independentemente de como **são** ativadas.

EXEMPLO 7.2 Escopo estático com procedimentos não aninhados

No programa abaixo as funções *soma* e *multiplica* são passadas como **parâmetro** para a subrotina *print*. A variável *glob* (sublinhada) é global. Seu endereço, que é determinado em tempo de compilação, é conhecido pelas duas funções e pela subrotina. A execução do programa resulta na impressão dos valores 5 e 0.

```
program P;
var glob: integer;
function soma(n: integer): integer;
begin soma := glob + n end;
function multiplica(n: integer): integer;
begin multiplica := glob * n end;
procedure print(function func(n: integer): integer);
begin write(func(5)) end;
begin
  glob:=0;
  print(soma); print(multiplica);
end.
```

Escopo estático e rotinas aninhadas

Nas linguagens de programação que permitem embutimento de rotinas e implementam o escopo estático (caso da linguagem Pascal), quando é encontrado um identificador, a declaração que vale para ele é a do maior nível de embutimento igual **ou menor** do que o escopo no qual o identificador está sendo referenciado.

O **nível de embutimento** é definido da seguinte forma. O programa principal tem nível de embutimento 1; a partir daí, adiciona-se 1 ao nível de embutimento à medida que se passa de um procedimento englobante para um procedimento englobado. Cada variável do programa tem o mesmo nível de embutimento do procedimento em que ela é declarada. O Exemplo 7.3 ilustra esses conceitos.

EXEMPLO 7.3 Escopo estático e rotinas aninhadas.

No programa abaixo, a função f1 tem nível de embutimento 2, e a função f2 tem nível de embutimento 3. A execução do programa resultará na impressão dos valores “6 5 2”, nessa ordem.

```
program P;
var v1, v2: integer;
function f1(m, n: integer): integer;
var v1: integer;
function f2(n: integer): integer;
var v2: integer;
begin v2 := 3;
  write(v1*v2);
  i2 := m+n+v2
end;
begin v1 := 2;
  write(f2(v1+v2));
  i2 := m+n+v2
end;
begin
  v1:=0; v2 := 1;
  write ( f1(v1, v2));
end.
```

O mecanismo de busca de variáveis não-locais pode ser **implementado** de dois modos:

(1) através de ponteiros de acesso; (2) através de ponteiros de níveis.

Ponteiros de acesso

Para acessar nomes não-locais, cada **registro de ativação** tem um **ponteiro** de acesso que **indica** o **registro de ativação** da rotina imediatamente **envolvente** (de nível de embutimento **uma** unidade menor), independente da ordem de ativação.

EXEMPLO 7.4 Pilha de ativação com ponteiros de acesso.

O programa abaixo é o mesmo programa do Exemplo 7.1, com a diferença da função *fatorial* estar agora embutida na função *somatorio*. A Figura 7.8 mostra a pilha de ativação com ponteiros de acesso a nomes não-locais, em diversos momentos da execução deste programa.

```
program expoX;
var x, erro, expX: real;
function somatorio(x: real, tol: real): real;
var termo, soma: real;
n: integer;
function fatorial(n: integer): integer;
begin
  if (n = 0) or (n = 1)
    then fatorial := 1
    else fatorial := n * fatorial(n - 1)
  end;
begin
  n := 0; soma := 0;
  repeat
    termo := (x**n)fatorial(n);
    soma := soma + termo;
    n := n + 1;
  until termo <= tol;
  sornatono := soma;
end;
begin
  x := 2.5;
  erro := 0.0001;
  expX := somatorio(x, erro);
  writeln ('Valor de X = ', x, ' Valor de expX = ', expX);
end.
```

□

Endereços de nomes não-locais são encontrados conforme descrito a seguir. Suponha um procedimento P, com nível de embutimento nP , contendo referência à variável 1150-local A, cujo nível de embutimento é nA , menor ou igual a nP . O endereço de A é encontrado como segue:

- 1) percorre-se $nP-nA$ ponteiros de acesso, chegando ao registro de ativação para o procedimento que contém a declaração de A;
- 2) a memória para A está localizada a um deslocamento fixo relativo ao início da área reservada para dados locais no registro de ativação em questão.

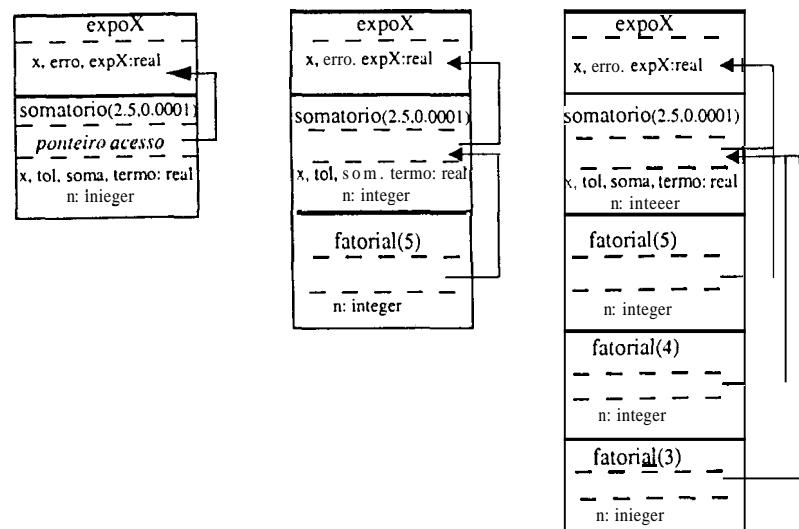


Figura 7.8 Pilha de ativação com ponteiros de acesso

O endereço de uma variável A global a P é dado pelo **seguinte** par de informações, computado em tempo de compilação e armazenado na Tabela de Símbolos:

($nP - nA$, deslocamento de A no registro de ativação)

O código para determinar o valor dos ponteiros de acesso faz parte do código de chamada dos procedimentos. Suponha que um procedimento P ao nível de embutimento nP chama um procedimento X ao nível de embutimento nX . O código para calcular o ponteiro de acesso do procedimento chamado vai depender do embutimento desse procedimento.

- $n_P < n_X$ (X está mais embutida que P): isso significa que X está declarada dentro de P (ou então não seria acessível a partir de P). Nesse caso, o ponteiro de acesso do registro de ativação da rotina chamada deve apontar para o ponteiro de acesso do registro de ativação da rotina chamadora, logo abaixo na pilha.
- $n_P \geq n_X$: as rotinas embutidas em P e X , de níveis de embutimento menores que n_X (ou seja, de níveis 1, 2, ..., n_X-1), devem ser as mesmas. Percorrendo $n_P - n_X + 1$ ponteiros de acesso a partir da rotina chamadora, encontra-se o registro de ativação mais recente da rotina que, estaticamente, envolve diretamente ambas (a chamada e a chamadora). A expressão $(n_P - n_X + 1)$ pode ser computada em tempo de compilação.

Ponteiros de níveis

O segundo mecanismo para acessar nomes não definidos localmente utiliza um vetor de ponteiros para registros de ativação. A área de memória reservada para uma variável não-local A , declarada no nível de embutimento i , estará localizada no registro de ativação apontado pelo elemento $V[i]$ do vetor de ponteiros.

Suponha que o controle de execução está com um procedimento P , cujo nível de embutimento é j . Os primeiros $j-i$ elementos do vetor apontam para as ativações mais recentes dos procedimentos que estaticamente incluem P , e $V[j]$ aponta para o registro de ativação de P . O vetor de níveis permite agilizar o acesso aos nomes não-locais e pode ser usado ao invés de ponteiros de acesso.

EXEMPLO 7.5 Pilha de ativação com ponteiros de níveis.

A Figura 7.9 mostra a pilha de ativação do programa *sort* com ponteiros de níveis em diversos momentos de sua execução.



Quando um novo registro de ativação, para um procedimento de nível i , é empilhado, as seguintes operações são executadas:

- o valor de $V[i]$ é salvo no novo registro de ativação;
- $V[i]$ passa a apontar para o novo registro de ativação.

Imediatamente antes de o controle retomar ao procedimento chamador, $V[i]$ é atualizado para o valor salvo.

As operações acima são justificadas de acordo com o que segue. Supondo que um procedimento de nível j chama um procedimento de nível i , deve-se considerar dois casos:

- se $j < i$, então $i = j+1$, isto é, o procedimento chamado está embutido no procedimento chamador. Os primeiros j elementos do vetor de níveis não necessitam ser modificados, e $V[i]$ aponta para o novo registro de ativação.
- se $j \geq i$, então os procedimentos envolvidos de níveis 1, 2, ..., $i-1$ dos procedimentos chamado e chamador devem ser os mesmos. Logo, salva-se $V[i]$ no novo registro de ativação e faz-se $V[i]$ apontar para esse registro. Os $i-1$ elementos do vetor de níveis permanecem iguais.

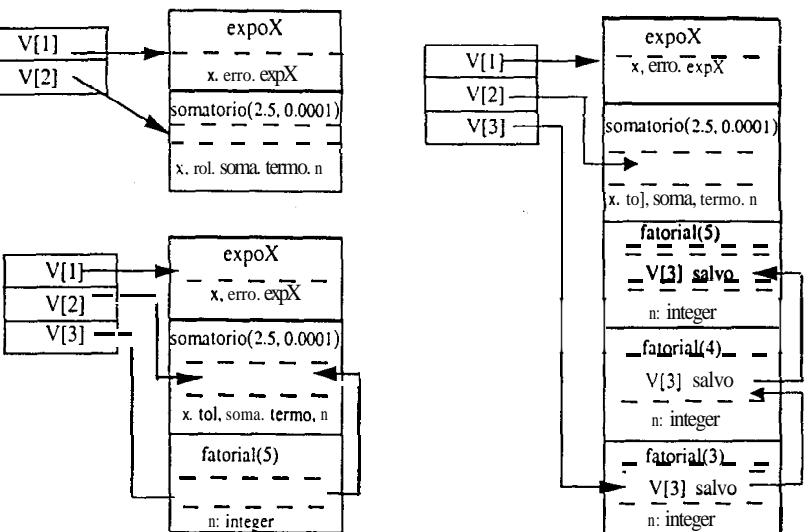


Figura 7.9 Pilha de ativação com ponteiros de níveis

Cada procedimento enxerga o vetor correto desde o nível 1 até o seu próprio nível. O tamanho máximo do vetor é determinado pelo nível máximo de embutimento de procedimentos no programa, em tempo de compilação. O vetor de níveis pode ser armazenado em uma área de memória reservada estaticamente ou na própria pilha de ativação. Neste último caso, uma cópia do vetor é feita a cada nova ativação.

7.4 Passagem de Parâmetros

Usualmente, a comunicação de dados entre procedimentos é feita através de variáveis globais ou de **parâmetros**. Dos mecanismos de passagem de **parâmetros**, os mais conhecidos são: passagem-por-valor (**call-by-value**), passagem-por-endereço (**call-by-address** ou **call-by-reference**), passagem-por-cópia-restauração (**call-by-copy-restore**) e passagem-por-nome (**call-by-name**).

Passagem de parâmetros por valor

É o método mais simples de passagem de parâmetros. Os parâmetros são avaliados, e seus valores são passados para o procedimento chamado. Esse mecanismo, usado nas linguagens C e Pascal, pode ser implementado como segue:

- um parâmetro formal é tratado exatamente como um nome local, de maneira que a memória para os parâmetros formais é reservada no registro de ativação do procedimento chamado;
- o procedimento chamador avalia os parâmetros reais e armazena seus valores na memória reservada para os parâmetros formais.

Passagem de parâmetros por endereço

Quando parâmetros são passados por endereço, o procedimento chamador passa o endereço de cada parâmetro real.

- se um parâmetro real é um identificador, então seu endereço é passado;
- se o parâmetro real é uma expressão, então a expressão é avaliada num temporário, e o endereço desse temporário é passado;

Referências a parâmetros formais, no procedimento chamado, são feitas de forma indireta. A passagem-por-endereço é usada em várias linguagens, inclusive em Pascal.

Passagem de parâmetros por cópia-restauração

É um meio-termo entre passagem por valor e passagem por referência e também é conhecido como passagem por dor-resultado (**call-by-value-result**):

- Antes da transferência para o procedimento chamado, os parâmetros reais são avaliados. Os argumentos que correspondem a valores são passados por valor, e os argumentos que correspondem a endereços são determinados.

- No retorno do procedimento, os valores dos argumentos computados pelo procedimento são copiados nos endereços anteriormente determinados.

O primeiro passo copia os valores dos **parâmetros reais** no registro de ativação do procedimento chamado (no espaço correspondente aos parâmetros formais). O segundo passo copia de volta os valores finais dos parâmetros formais no registro de ativação do procedimento chamador (nos endereços computados a partir dos parâmetros reais, antes da chamada).

Passagem por cópia-restauração é usada em algumas implementações da linguagem Fortran, outras usam a passagem por endereço.

Passagem de parâmetros por nome

Este é um tipo de passagem tradicional da linguagem Algol, cujo maior interesse é teórico, já que é pouco usado na prática.

- O procedimento chamado é tratado como se fosse uma macro, isto é, a sua chamada é substituída pelo corpo do procedimento, e os parâmetros formais são literalmente substituídos pelos parâmetros de chamada (cada parâmetro formal no corpo do procedimento é substituído pela expressão que foi usada como parâmetro real).
- Os nomes locais do procedimento chamado são mantidos distintos dos nomes do procedimento chamador. Pode-se considerar cada nome local como sendo sistematicamente renomeado antes de ser realizada a expansão da macro.

7.5 Alocação Dinâmica de Memória

Estruturas de dados alocadas diretamente sob o controle programa (variáveis apontadas por ponteiros) são armazenadas em memória *heap*. A alocação de memória pode ser

- explícita, através de instruções da linguagem específicas para tal como, por exemplo, o comando **new** de Pascal; ou
- implícita, resultante de operações de construção de estruturas mais complexas como, por exemplo, a aplicação da primitiva **cons** da linguagem Lisp.

Nas linguagens de programação em que a alocação dinâmica é realizada explicitamente, o programador deve preocupar-se em liberar a memória quando o dado não for mais necessário.

Essas linguagens provêm comandos específicos para liberação de memória, tal como o comando *dispose* de Pascal. Nas linguagens em que a alocação dinâmica é implícita, as rotinas de suporte à execução do programa provêm um coletor de lixo automático que libera a memória ocupada por informações que tornaram-se descartáveis.

EXERCÍCIOS

- 1) Mostre a pilha de registros de ativação com ponteiros de acesso a nomes não-locais e com *displays*, quando a função *r* do programa abaixo estiver executando.

```
procedure main
    function m(i: integer): integer;
    function p(i: integer): integer;
        begin return m(i * 2 + 1);
    end;
    function q(i: integer): integer;
        begin return m(i + 111);
    end;
    function r(i: integer): integer;
        begin return i * 3;
    end;
    begin case i mod 3 of
        0: return p(i13);
        1: return q(i13);
        2: return r(i13);
    end
    begin write (m(157));
    end.
```

- 2) Responda quais os valores impressos pelo programa abaixo, supondo passagem de parâmetros:

- (a) by value;
- (b) by reference;
- (c) by copy-restore;
- (d) by name.

```

Program main;
procedure p(x, y, z);
begin  y := y + 1;
        z := z + x
end;
begin  a := 2;
        b := 3;
        p(a + b, a, a);
        write(a)
end.

```

8 Geração de Código Objeto

Os principais requisitos impostos a geradores de código objeto são os seguintes:

- o código gerado deve ser correto e de alta qualidade;
- o código deve fazer uso efetivo dos recursos da máquina; e
- o código gerado deve executar eficientemente.

O problema de gerar código ótimo é insolúvel (**indecidível**) como tantos outros. Na prática, devemos contentar-nos com técnicas heurísticas que geram "bom" código (**não necessariamente Ótimo**).

8.1 Considerações no projeto de um gerador de código

Basicamente, quatro aspectos devem ser considerados no projeto de geradores de código:

- forma do código objeto a ser gerado: linguagem absoluta, relocável ou **assembly**;
- seleção das instruções de máquina: a escolha da sequência apropriada pode resultar num código mais curto e mais rápido;
- **alocação** de registradores;
- escolha da ordem de avaliação: a determinação da melhor ordem para execução das instruções é um problema insolúvel. Algumas computações requerem menos registradores para resultados intermediários.

Forma do código objeto

A geração de um programa em linguagem absoluta de máquina tem a vantagem de que o **programa objeto** pode ser armazenado numa área de memória fixa e ser imediatamente executado. Compiladores deste tipo são utilizados em ambientes universitários, onde é altamente conveniente diminuir o custo de **compilação**. Os compiladores que geram código absoluto e executam-no imediatamente são conhecidos como **load and go compilers**.

A geração de código em linguagem de máquina relocável permite a compilação separada de subprogramas. Módulos e objetos relocáveis podem ser ligados e carregados por

um Ligador-Carregador. Essa estratégia dá flexibilidade para compilar subrotinas separadamente e para chamar outros programas previamente compilados.

A tradução para linguagem **assembly** facilita o processo de geração de código. São geradas instruções simbólicas e podem ser usadas as facilidades de macro instruções. O preço pago é um passo adicional: tradução para linguagem de máquina. É uma estratégia razoável, especialmente, para máquinas com pouca memória, nas quais o compilador deve desenvolver-se em vários passos.

Alocação de Registradores

Instruções com registradores são mais curtas e mais rápidas do que instruções envolvendo memória. Portanto, o uso eficiente de registradores é muito importante. A atribuição ótima de registradores a variáveis é muito difícil, e muito problemática quando a máquina trabalha com registradores aos pares (para instruções de divisão e multiplicação), ou provê registradores específicos para endereçamento e para dados.

EXEMPLO 8.1 Divisão nos computadores IBM/360 e /370.

A operação de divisão é como segue:

D r, m o dividendo de 64 bits deve estar armazenado em um par de registradores par-ímpar, representado por r;
 m contém o divisor; após a divisão, o registrador par contém o resto e o ímpar contém o quociente.

| | | |
|-------------|--------|--------------------------------------|
| $t = a + b$ | L | R0, a |
| $t = t + c$ | A | R0, b |
| $t = t / d$ | A | R0, c |
| SRDA | RO, 32 | desloca dividendo para RI e limpa R0 |
| D | RO, d | |
| St | R1, t | |

O problema do uso otimizado de registradores fica simples quando a máquina possui um único registrador para realizar operações aritméticas (por exemplo, um acumulador). O problema deixa de existir quando as operações aritméticas são realizadas sobre uma pilha. Neste caso, deixa de ser necessária, inclusive, a utilização de variáveis **temporárias**.

8.2 A Máquina Objeto

Familiaridade com a máquina e com o conjunto de instruções é pré-requisito para projetar um bom gerador de código. Neste capítulo, será utilizada uma máquina de registradores endereçada por byte, com 4 bytes por palavra, n registradores de propósito geral RO, R1, ..., Rn e instruções da forma:

op fonte, destino

A máquina permite os modos de endereçamento listados na Tabela 8.1. O custo de uma instrução é igual a uma unidade mais o custo do endereçamento dos operandos na instrução. o que corresponde ao tamanho da instrução em palavras.

| endereçamento | forma | endereço | custo adicional |
|-------------------|-------|---------------------------|-----------------|
| absoluto | M | M | 1 |
| regrador | R | R | 0 |
| indexado | d(R) | d + conteúdo(R) | 1 |
| regrador indireto | *R | conteúdo(R) | 0 |
| indexado indireto | *d(R) | conteúdo(d + conteúdo(R)) | 1 |
| literal | # | constante c | 1 |

Tabela 8.1 Modos de endereçamento e custo das instruções

EXEMPLO 8.2 Tipos de endereçamento da máquina alvo.

| | | |
|-----|-----------|---|
| STO | RO, M | armazena o conteúdo de R0 na posição de memória M |
| STO | 8(R1), M | armazena o valor conteúdo (8 + conteúdo (R1)) em M |
| STO | *4(R2), M | armazena conteúdo (conteúdo (4+conteúdo (R2))) em M |
| STO | #1, R1 | armazena a constante 1 no registrador R1 |

EXEMPLO 8.3 Custo das instruções da máquina alvo.

| | | |
|------|--------|---|
| LOAD | RO, R1 | custo = 1 |
| STO | R5, M | custo = 2 endereço de M ocupa a palavra seguinte |

```
ADD #1, R3           custo = 2   constante 1 está na palavra seguinte
SUB 4(R0), *12(R1)  custo = 3   deslocamentos 4 e 12 ocupam 2 palavras
```

EXEMPLO 8.4 Possíveis códigos para o comando $a := b + c$.

A seguir, são mostradas quatro possibilidades de geração de código para o comando de atribuição $a := b + c$:

- 1) **LOAD** b, R0
ADD c, R0
STO R0, a custo = 6
- 2) **STO** b, a
ADD c, a custo = 6

3) Supondo que R0, R1, e R2 já contêm os endereços de a, b, c:

```
LOAD *R1, *R0
ADD *R2, *R0    custo = 2
```

4) Supondo que R1 e R2 contêm os valores de b e c, e o valor de b não será necessário após a atribuição:

```
ADD R2, R1
STO R1, a      custo = 3
```

8.3 Gerador de Código Simplificado

Para utilização do gerador descrito abaixo, deve-se assumir o seguinte:

- para cada operador do código intermediário, existe um operador no código **objeto**;
- resultados computados podem ser deixados em registradores tanto tempo quanto desejado, salvando-os somente:
 - a) se o **registrar** é necessário;
 - b) imediatamente antes de uma chamada de **procedure**, desvio ou comando com rótulo.

EXEMPLO 8.5 Códigos ótimos para o comando $a := b + c$.

O código ótimo para a atribuição $a := b + c$, supondo b e c armazenados em Rj, e o resultado podendo ser deixado em Ri, é:

```
ADD Rj, Ri      custo = 1
```

Se c está em memória, e Ri contém b, o código ótimo é:

```
ADD c, Ri      custo = 2
```

Caso c seja usado a seguir (como operando), o código ótimo seria:

```
LOAD c, Rj
ADD Rj, Ri      custo = 3
```

8.3.1 Informação de próximo-uso

Objetivo de computar o próximo-uso de uma variável é procurar manter essa variável em registrador se ela vai ser usada em seguida. Para o algoritmo a seguir, é **necessário** definir inicialmente o que significa uma variável estar viva.

Variável viva

Considerando **um** bloco básico, uma variável está **viva** numa linha i desse bloco se ela vai ser subsequentemente usada nesse bloco (em uma linha maior que i). Caso **contrário**, a variável está morta na linha i. Pela análise do fluxo da execução de um programa, é possível determinar as variáveis que estarão vivas na saída de um bloco. Se essa análise não for feita, todas as variáveis não temporárias deverão ser consideradas vivas na saída do bloco.

Algoritmo para computar os próximos-usos de variáveis

Analisa-se o código de um bloco básico de trás para diante (do fim para o início). Encontrando-se o comando i: $x := y \text{ op } z$, faz-se:

- 1) registrar no comando i a informação encontrada na tabela de símbolos com relação ao próximo-uso e vida de x, y e z. Se x não está vivo, então o comando pode ser desprezado;
- 2) na tabela de símbolos, registrar x como não-vivo e não tendo próximo-uso;
- 3) na tabela de símbolos, registrar y e z como vivos e i = próximo-uso de y e z.

Se a instrução i tem a forma $x := y$ ou $x := op\ y$, os passos continuam os mesmos, porém ignorando z .

8.3.2 Descritores de Registradores e Endereços

O algoritmo de geração de código manipula descritores que informam o que contém cada registrador. Inicialmente, todos os registradores estão vazios. O descritor de registradores é consultado cada vez que um registrador é necessário. O descritor de endereços informa onde o valor da variável pode ser encontrado. A localização pode ser um registrador, um endereço de memória ou uma entrada na pilha. Essa informação pode estar armazenada na tabela de símbolos.

8.3.3 Algoritmo de Geração de Código

Entrada: Bloco básico de código intermediário de três endereços.

Para cada instrução $x := y op z$, faça:

- 1) Chame `buscaReg` para determinar a posição L onde o resultado deve ser armazenado. Normalmente, L será um registrador, mas poderá ser memória.
- 2) Consulte o descritor de endereços para y para determinar y' , uma localização corrente de y ; se y estiver em memória e registrador, prefira registrador; se o valor de y não está ainda em L (local do resultado), gere a instrução `LOAD/STO` $y' L$ (L é registrador ou memória) que armazena uma cópia de y em L .
- 3) Gere a instrução `OP z'`, L onde z' é a posição corrente de z (prefira registrador se z está em memória e registrador); atualize o descritor de endereço de x para indicar que seu valor se encontra em L . Se L é um registrador, atualize o descritor para indicar que esse registrador contém o valor de x , e remova x de todos os outros descritores de registradores.
- 4) Se os valores de y e/ou z não têm próximo-uso, não estão vivos na saída do bloco e estão em registradores, então altere o descritor de registradores para indicar que esses registradores não contêm mais y e z .

Um importante caso especial é a atribuição $x := y$. Se y está no registrador, simplesmente troque os descritores de registrador e endereço para registrar que agora o valor de x é encontrado apenas no registrador que contém o valor de y .

Função `buscaReg`

Esta função retoma a localização L para armazenar o valor de x na atribuição $x := y op z$.

- 1) Se y está num registrador que não contém outras variáveis e y não está vivo, nem tem próximo-uso após a execução de $x := y op z$, então retoma o registrador de y como L . Atualize o descritor de endereço de y , indicando que y não está mais em L ;
- 2) Falhando (1), retome um registrador vazio para L , se existir algum;
- 3) Falhando (2), se x tem um próximo-uso no bloco, ou op é um operador que requer um registrador (por exemplo, indexação), encontre um registrador ocupado R . Armazene o valor de R em memória (`STO R,M`) se ainda não estiver armazenado, atualize o descritor de M e retome R . Se R contém o valor de várias variáveis, uma instrução `STO` deve ser gerada para cada variável que necessite ser salva;
- 4) Se x não é usado no bloco, ou nenhum registrador ocupado pode ser liberado, escolha L como endereço de memória para x .

EXEMPLO 8.6 Código gerado para o comando $d := (a - b) + (a - c) + (a - c)$

O código intermediário para o comando em questão é:

$$\begin{aligned} t1 &= a - b \\ t2 &= a - c \\ t3 &= t1 + t2 \\ d &= t3 + t2 \end{aligned}$$

A Tabela 8.2 mostra o código gerado, bem como os conteúdos do descritor de registrador e do descritor de endereço.

| Código intermediário | Código gerado | Descriptor de registrador | Descriptor de endereço |
|----------------------|-------------------------|------------------------------------|--|
| $t_1 = a - b$ | LOAD a, R0 SUB b, R0 | R0 contém t_1 | t_1 está em R0 |
| $t_2 = a - c$ | LOAD a, R1 SUB c, R1 | R0 contém t_1 R1 contém t_2 | t_1 está em R0 t_2 está em R1 |
| $t_3 = t_1 + t_2$ | ADD R1, R0 | R0 contém t_3 R1 contém t_2 | t_2 está em R1 t_3 está em R0 |
| $d = t_3 + t_2$ | ADD R0, R1 STO R0, d | R0 contém d R1 contém t_2 | t_2 está em R1 d está em R0 e memória |

Tabela 8.2 Código gerado para o comando $d := (a - b) + (a - c) + (a - c)$

Exercícios

- 1) Traduza para código de três-endereços e gere código objeto para o comando de atribuição

$$x := (a + b) - (a + c) + ((a + b) - (c - d))$$

considerando o esquema de tradução do Exemplo 5.3 e a máquina objeto, definida na Seção 8.2, a qual dispõe de dois registradores para operações aritméticas.

- 2) Para o comando de atribuição do exercício acima, gere código objeto, para a mesma máquina, a partir do código de três-endereços otimizado obtido pela aplicação do método de construção de GAD's (ver Seção 6.1).

Referências Bibliográficas

- Aho, A. V., Sethi, R. and Ullman, J. D. *Compiler Principles Techniques and Tools*. Addison-Wesley, 1988.
- Hopcroft, J. and Ullman, J. D. *Formal Languages and Their Relation to Automata*. Addison-Wesley, 1979.
- Mason, T. *Lex & Yacc*. O'Reilly & Associates, 1990.
- Menezes, P. B. *Linguagens Formais e Autômatos*. Porto Alegre: Sagra-Luzzatto - Instituto de Informática da UFRGS, 2000. (2 Ed. Série Livros Didáticos, 3)
- Peny, P. D. *Programrning Language Translation*. Addison-Wesley, 1986.



Série
Livros Didáticos

Número

96 páginas • ISBN 85-241-0515-1

O livro Fundamentos da Matemática Intervalar, de autoria dos professores Paulo W. De Oliveira, Tiarajú Asmuz Diverio e Dalcidio Moraes Claudio, busca apresentar os conceitos básicos da matéria, servindo como material didático para cursos introdutórios de Intervalos e Análise Intervalar. Pretende-se com esta obra disseminar o uso da Matemática Intervalar para resolução de problemas das Ciências e Engenharias.



Série
Livros Didáticos

Número

96 páginas. ISBN 85-241-0516-x

O PASCAL-XSC é a linguagem que apresenta melhores condições a implementação de algoritmos sofisticados – onde a verificação é feita pelo consumidor. No Programando em PASCAL-XSC são discutidos, entre outros tópicos, as características do PASCAL-XSC, informações sobre a instalação desta linguagem e exemplos de programas desenvolvidos em PASCAL-XSC.