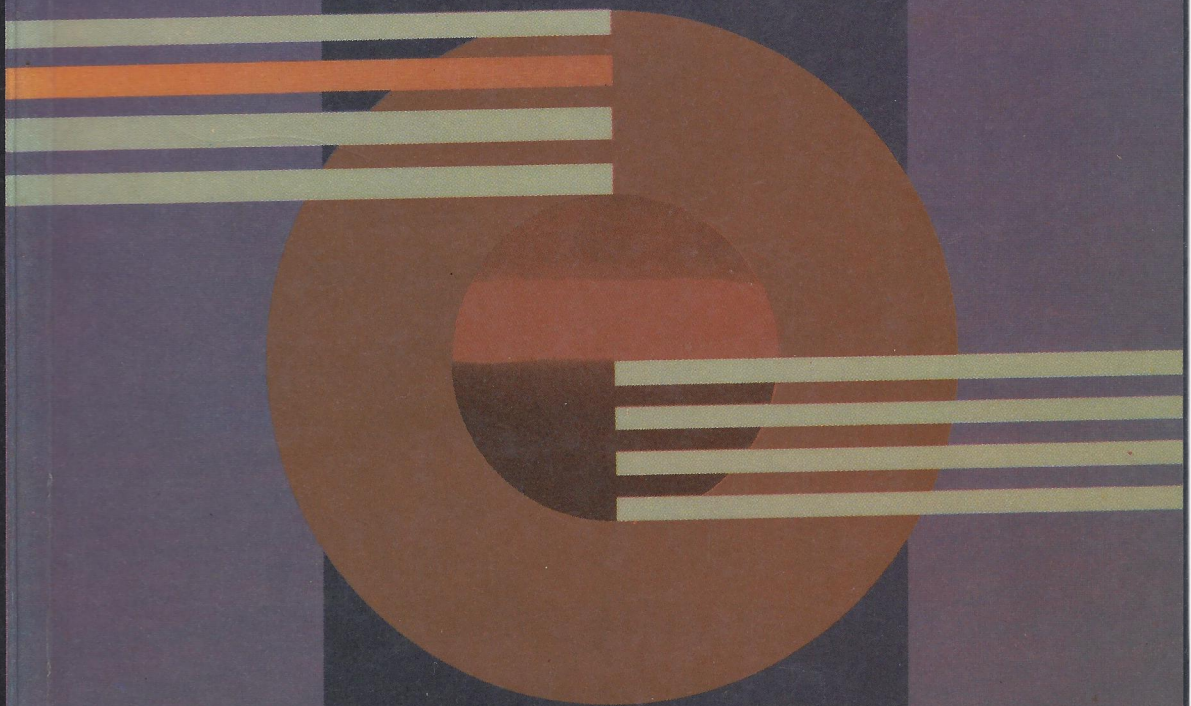
5ª Reimpressão

A CONSTRUÇÃO DE UM COMPILADOR

Valdemar W. Setzer
Inês S. Homem de Melo



EDITORA CAMPUS

Setzer/Melo
A CONSTRUÇÃO DE UM COMPILADOR

TÍTULOS DE INTERESSE CORRELATO

SISTEMAS OPERACIONAIS E COMPILADORES

A CONSTRUÇÃO DE UM COMPILADOR — *V.W. Setzer, I.S.H. Melo*
SISTEMAS OPERACIONAIS: Uma Visão Sistemática — *W.S. Davis*
O MANUAL DE CP/M INCLUINDO MP/M — *R. Zacks*
PC DOS: Como Usar o DOS com Inteligência — *P. Norton*
PRINCÍPIOS DE SISTEMAS OPERACIONAIS — *C.C. Guimarães*
PROJETO DE SISTEMA OPERACIONAL: O Enfoque Xinu — *D. Comer*
SISTEMAS OPERACIONAIS PARA MICROCOMPUTADORES — *M. Dahmke*
UNIX VOL. I: Entendendo o Sistema UNIX — *K. Christian*
UNIX VOL. II: Tópicos Avançados do Sistema UNIX — *K. Christian*
O GUIA PAUL MACE DE RECUPERAÇÃO DE DADOS — *P. Mace*
SISTEMAS OPERACIONAIS DISTRIBUÍDOS — *C. Kirner, S.B.T. Mendes*
EXPLORANDO O SISTEMA UNIX — *S.G. Kochan, P.H. Wood*
EXPLORANDO MS-DOS — *V. Wolverton*

Solicite nosso catálogo completo.

Procure nossas publicações nas boas livrarias ou comunique-se diretamente com:

EDITORA CAMPUS LTDA.

Qualidade internacional a serviço do autor e do leitor nacional.
Rua Barão de Itapagipe, 55 Rio Comprido
Tel. PABX (021) 293-6443 Telex (021) 32606 EDCP BR
FAX (021) 293-5683 20261 Rio de Janeiro RJ Brasil
Endereço Telegráfico: CAMPUSRIO

A CONSTRUÇÃO DE UM COMPILADOR

A CONSTRUÇÃO DE UM COMPILADOR

Valdemar W. Setzer
Inês S. Homem de Melo
Instituto de Matemática e Estatística USP

5.ª Reimpressão

Editora Campus Ltda.

Capa
Otavio Studart

Projeto Gráfico, Composição e Revisão
Editora Campus Ltda.
Qualidade internacional a serviço do autor e do leitor nacional.
Rua Barão de Itapagipe 55 Rio Comprido
Telefone: (021) 293 6443 Telex: (021) 32606 EDCP BR
FAX (021) 293-5683
20261 Rio de Janeiro RJ Brasil
Endereço Telegráfico: CAMPUSRIO

ISBN 85-7001-334-5

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

Setzer, Valdemar W.
S522c A construção de um compilador / Waldemar W. Setzer, Inês S. Homem de Melo. — Rio de Janeiro: Campus, 1989.

Apêndices.
Bibliografia.
ISBN 85-7001-334-5

1. Compilação (Computadores eletrônicos). I. Melo, Inês S. Homem de II. Título.

86-0291

CDD — 001.6425
CDU — 681.3:31

93 92 91 90 9 8 7 6 5

SUMÁRIO

CAPÍTULO 1

INTRODUÇÃO

O presente livro baseia-se no texto de mesmo nome elaborado para a 2ª Escola de Computação realizada em Campinas em janeiro de 1981. Foram feitas várias correções e algumas adições tendo-se conservado a ordem e o formato daquele original. Trata-se de obra com duas finalidades: didática e informativa. Assim, tanto pode ser empregada como livro-texto de disciplinas de teoria e construção de compiladores, quanto como texto para profissionais de processamento de dados que queiram informar-se sobre esse campo da Ciência de Computação ou Informática.

Este texto divide-se em duas partes: a primeira, até o capítulo 5 inclusive, trata de análise léxica e análise sintática; o restante do texto trata da análise de contexto, geração de código e detalhes de implementação. A essa divisão estrutural correspondem enfoques diversos na abordagem dos assuntos tratados. De fato, a primeira parte é absolutamente geral, podendo ser aplicada em qualquer linguagem de programação para a qual se construa uma gramática que atenda às restrições descritas. Normalmente encontram-se nos textos sobre compilação a descrição de vários métodos de análise sintática. Em contraposição, é aqui apresentado apenas um método, desenvolvido por V. W. Setzer. Os autores consideram esse método muito prático, sob todos os pontos de vista: é simples, eficiente tanto em velocidade de processamento como em espaço ocupado e contém tratamento automático de erros sintáticos; é suficientemente geral para que as restrições impostas às gramáticas não prejudiquem o projeto de linguagens de programação; essas restrições podem ser verificadas com muita simplicidade, durante o projeto de uma gramática, pelo próprio projetista que tem a escolha de usar eventualmente uma representação desta em forma de diagrama. O leitor que quiser obter informações sobre outros métodos de análise sintática pode consultar as inúmeras obras existentes sobre o assunto, inclusive a de T.Kowaltowski /Kow 83/. Com o tratamento em detalhe de um só método, pudemos abordar em profundidade todas as características de sua implementação, apresentando mesmo os vários algoritmos envolvidos. Por outro lado, cremos que com o estudo desta primeira parte o leitor adquirirá uma introdução razoável ao campo de análise sintática, facilitando seus estudos posteriores nessa área.

Na segunda parte do livro decidimos abordar novamente um só caso concreto: em lugar de expormos os problemas de implementação de compiladores em geral, ou desenvolver uma linguagem específica para exemplificar os vários tópicos, decidimos pela escolha de uma linguagem concreta, qual seja a PASCAL desenvolvida por N. Wirth. Essa escolha teve dois objetivos: permitir a abordagem de uma gama muito típica e variada de questões ligadas à

construção de compiladores, e ao mesmo tempo tratar de uma linguagem cujo sucesso mundial demonstra sua aplicação e interesse. Abraçamos aqui uma tarefa temerária: cobrir com alguma profundidade praticamente todas as características dessa linguagem. Infelizmente, com isso alguns itens ficaram bastante áridos, como é o caso da construção da Tabela de Símbolos do compilador. No entanto, cremos que o esforço valeu a pena: se alguém interessar-se pelos detalhes de implementação, pode aqui obter inúmeras informações a esse respeito, as quais provavelmente ajudá-lo-ão a encontrar suas próprias soluções em seu caso particular. Chamamos a atenção, no entanto, para o fato de que em vários tópicos a abordagem é geral, como por exemplo a implementação de variáveis indexadas e de procedimentos.

Em lugar de darmos uma lista de exercícios para cada tópico abordado, resolvemos sugerir como trabalho prático para o leitor: a execução de um projeto completo de um compilador. Nesse sentido, damos indicações das várias partes desse projeto, que se constitui também em um exemplo de modularização de projetos de sistemas de programação. Recomendamos aos leitores o uso de alguns dos inúmeros livros sobre compilação para obterem enunciados de exercícios relevantes a essa área.

Os autores têm experiência de vários anos na utilização da metodologia aqui descrita, tanto na implementação de compiladores, como no ensino dessa matéria, em disciplinas de um semestre. É possível exigir dos alunos a implementação de um compilador para um subconjunto relativamente grande na linguagem PASCAL, em apenas um semestre letivo (4 meses). Para isso recomendamos os seguintes prazos onde a superposição é intencional, pois a depuração de uma parte pode ser feita simultaneamente com a programação da parte seguinte:

PARTE	SEMANAS
1 — Rotina de Símbolos Reservados	1 e 2
2 — Analisador Léxico	3 e 4
3 — Carregador Sintático	5 e 6
4 — Analisador Sintático	6 e 7
5 — Tratamento de Erros Sintáticos	8 a 10
6 — Tabela de Símbolos	10 a 12
7 — Geração de Código	12 a 16

Uma forma que também temos empregado é a de dividir o projeto em dois semestres; no primeiro semestre seriam abordados os itens 1 a 5, o que é uma necessidade quando os alunos não têm grande prática de programação, como é muitas vezes o caso em disciplinas de pós-graduação.

Evidentemente, essa divisão do projeto implica em uma divisão paralela dos tópicos abordados em aula. Para isso recomendamos uma abordagem inicial essencialmente prática do assunto, tratando-se das bases teóricas posteriormente. Por exemplo, nada impede de se descrever o funcionamento do Analisador Sintático com a parte de recuperação de erros, para mais tarde se definirem rigorosamente as gramáticas aceitas pelo algoritmo.

CAPÍTULO 2

O PROCESSO DE COMPILAÇÃO

2.1 FUNCIONAMENTO DOS COMPILADORES

Um compilador C é um programa de computador que tem a finalidade de traduzir ou converter um programa escrito em uma linguagem L_f para um programa escrito em uma outra linguagem L_b. L_f é denominada *Linguagem-Fonte*, e o programa P_f nela escrito, a ser traduzido, é denominado *Programa-Fonte*. Analogamente temos uma *Linguagem-Objeto* (L_b) em que é escrito o *Programa-Objeto* (P_b), resultado da tradução.

Graficamente, esse processo pode ser representado por:

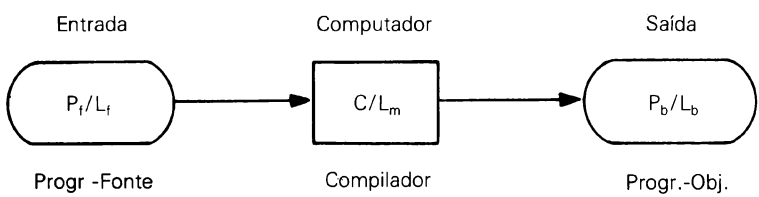
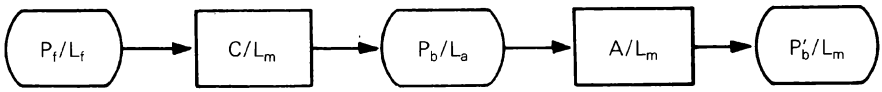


Fig. 2.1

onde P/L representa um programa P escrito na linguagem L. Note-se que o próprio compilador C é um programa, escrito em uma linguagem L_m, que em geral é a linguagem de máquina do computador onde ele é processado. Na maior parte dos casos, C não é programado em L_m, sendo convertido para essa linguagem por meio de uma compilação.

Em geral, entende-se por compilador o programa C que aceita como entrada um P_f escrito em uma linguagem L_f de *alto nível*, como por exemplo ALGOL, COBOL, PL/I, PASCAL, FORTRAN. L_b não é necessariamente uma linguagem de máquina. Por exemplo, L_b pode ser uma Linguagem de Montagem ("Assembler") L_a. Nesse caso é necessário ter-se mais uma *fase* de tradução de L_a para a linguagem de máquina L_m do computador a ser utilizado para processar o programa-objeto:



Progr.-Fonte Compilador Progr.-Obj. do Compilador Montador Progr.-Obj.

Fig. 2.2

O próprio compilador C pode ser composto de várias fases, denominadas neste caso de *passos* do compilador. Em cada passo, é usada uma linguagem-fonte e uma linguagem-objeto, próprias desse passo. No 1º passo, temos a linguagem-fonte L_f original, e no último passo a linguagem-objeto final L_b desejada. As outras linguagens envolvidas são denominadas *linguagens intermediárias*. Uma compilação de n passos pode ser representada por:

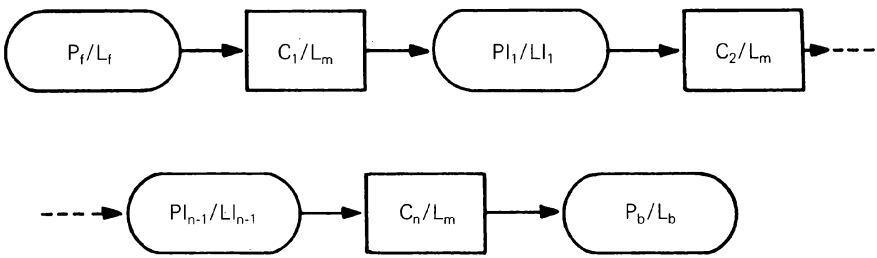


Fig. 2.3

As principais vantagens de se construir compiladores de vários passos são as seguintes: a) Menor utilização de memória do computador, já que cada passo exerce apenas uma parte das funções de todo o compilador; b) Maior possibilidade de se efetuar otimizações levando a um programa-objeto mais eficiente quanto ao espaço ocupado ou ao tempo de processamento; c) Os projetos e implementações das várias partes do compilador são mais independentes.

As principais desvantagens são: a) Maior volume de entrada/saída, quando os programas intermediários e os passos do compilador não ficam residentes na memória, o que é em geral o caso; b) Normalmente, aumento do tempo de compilação; c) Aumento do projeto total, com a necessidade de introdução das linguagens intermediárias.

Existem linguagens para as quais há necessidade absoluta de compilação em mais de um passo, como o ALGOL-68/WIJ 75/ Neste caso, referências feitas no programa-fonte a objetos declarados posteriormente impedem a compilação em um só passo.

Um caso interessante foi o do compilador FORTRAN para o IBM-1401 do início da década de 60, em que havia 64 passos. Os programas-fonte e intermediários ficavam residentes na pequena memória (4K ou 8K caracteres), e os passos, de tamanho muito reduzido, eram lidos e executados seqüencialmente.

Neste texto, será descrito um compilador para a linguagem PASCAL/J-W 74/ de um só passo. Será usada como linguagem-objeto a linguagem de montagem HAL (v. apêndice III); no capítulo 10 serão dadas indicações das mudanças necessárias para gerar-se diretamente linguagem de máquina, usando o computador hipotético HIPO (v. apêndice III).

2.2 ESTRUTURA GERAL DO COMPILADOR

A grosso modo, o compilador pode ser dividido estruturalmente nas partes da fig. 2.4.

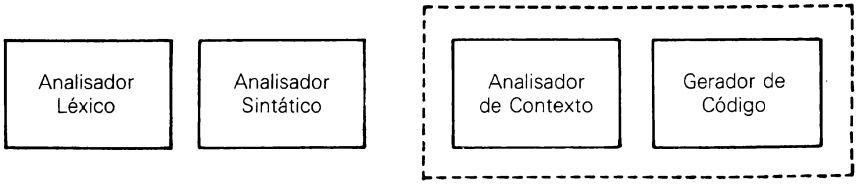


Fig. 2.4

No programa fonte podem ser encontradas *unidades sintáticas*, que especificam a estrutura gramatical do programa. Não é possível, a esta altura, definirmos o que se entende por essa denominacão, já que essa estrutura será definida através de uma gramática, a ser estudada no capítulo 4. Faremos aqui uma caracterização de várias unidades sintáticas da linguagem PASCAL, através de exemplos.

Tomemos um trecho de programa em PASCAL, um bloco ("block") com dois comandos ("statements") de atribuição. Na fig. 2.5, mostramos esse trecho, com as suas unidades sintáticas representadas por o .

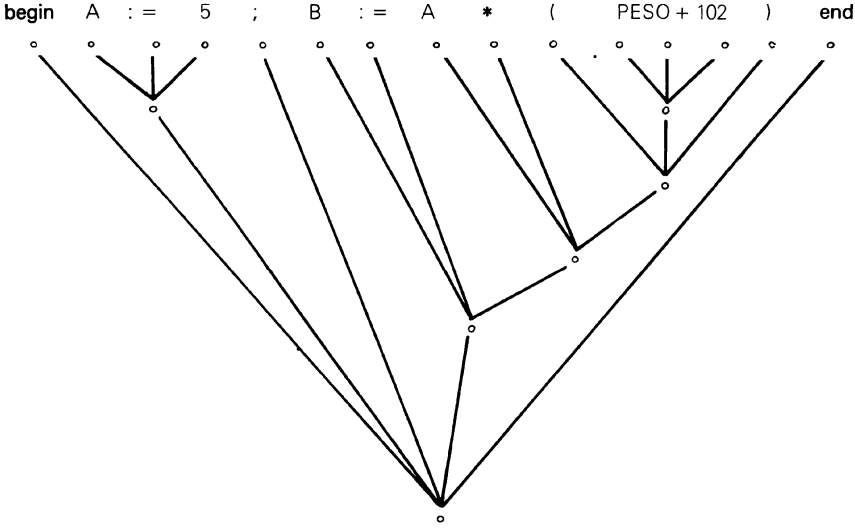


Fig. 2.5

Temos aí representada uma *árvore sintática*, que mostra a estrutura gramatical desse trecho de programa. Cada um de seus nós representa uma unidade sintática; assim **begin**,

A, ':=', 5, ';', ...')', **end**, são unidades sintáticas. Os três símbolos do subtrecho A:=5 compõem, num segundo nível, uma única unidade sintática. De fato, eles constituem um "statement" do PASCAL. Num terceiro nível, temos (PESO+102); num quarto nível, A * (PESO+102) que constitui a unidade sintática "expression" do PASCAL etc.

Pode-se notar que há unidades sintáticas compostas de outras unidades (as que estão do segundo nível para baixo), como (PESO+102), A*(PESO+102) etc., e outras que são *unidades elementares*, não compostas de outras, e que estão no primeiro nível, como **begin**, A, ':=' etc. Dá-se o nome de *item léxico* a cada uma dessas unidades elementares.

O *Analisador Léxico* (fig. 2.4) é a parte do compilador que lê o programa fonte e reconhece dentro deste cada item léxico. Na fig. 2.5 seriam reconhecidos os itens léxicos **begin**, A, ':=', 5 etc. Esses itens léxicos são fornecidos ao *Analisador Sintático* (fig. 2.4), que os agrupa nas diversas unidades sintáticas, construindo a árvore sintática (fig. 2.5). Para isso, ele utiliza uma série de regras de sintaxe, que constituem a *gramática* da linguagem fonte. Como já vimos, a gramática da linguagem fonte é que define, em última instância, as unidades sintáticas e portanto a estrutura sintática do programa-fonte. O reconhecimento dessa estrutura é essencial para o processo de compilação. Assim, na fig. 2.5 vemos que PESO + 102 deve ser reconhecido como uma unidade, antes de se reconhecer a unidade contendo a multiplicação ('*'). Isto é, o analisador sintático "descobre" que a soma é analisada (e portanto o código correspondente é gerado) antes da multiplicação. Da mesma maneira, as expressões do lado direito das atribuições (':=') são analisadas (e geradas) antes da atribuição propriamente dita.

Note-se que o analisador léxico trabalha com os caracteres do programa-fonte, agrupando-os no item léxico conveniente. Assim, as letras 'e', 'n', 'd' são agrupadas na palavra **end**. Mais ainda, é ele que reconhece o fato de que **end** é uma *palavra reservada* da linguagem fonte. No caso da variável PESO, as letras 'P', 'E', 'S', 'O' são agrupadas adequadamente, e a palavra assim formada é reconhecida como sendo um "variable identifier". Do mesmo modo, a seqüência de caracteres '1', 'O', '2' é reconhecida como o número 102. A seqüência ':', '=', como ':=' etc.

O analisador sintático tem também por tarefa o reconhecimento de *erros sintáticos*, que são construções do programa fonte que não estão de acordo com as regras de formação de unidades sintáticas, como especificado pela gramática. Assim, na seqüência A + * B, deve ser detectado um operador aritmético a mais. Após reconhecer um erro de sintaxe, o analisador deve emitir mensagem de erro adequada e *tratar* ("recover") esse erro, isto é, continuar a análise do resto do programa. Assim, o erro ocorrido deve influenciar o mínimo possível a análise desse resto.

O analisador léxico deve detectar erros léxicos, que no nosso caso podem ser, por exemplo, números inteiros com grandeza maior do que aquela que pode ser representada no computador-objeto, ou um símbolo especial inválido.

Toda a análise efetuada pelo compilador além da análise sintática é denominada comumente por *Análise "Semântica"* (fig. 2.4). (Usamos a palavra "semântica" entre aspas pois ela abarca uma parte muito pequena do que vem a ser realmente a semântica de um programa.) Ela engloba, além de outras, duas partes principais: a *Análise de Contexto* e a *Geração de Código*. Vamos caracterizar a primeira por meio de um exemplo: no comando A: = 5, é necessário fazer a seguinte análise:

1. A foi declarado? Se não o foi, há um erro de contexto.
2. Aonde A foi declarado (**procedure** mais interna englobando A, ou programa principal)?
3. Qual o tipo de A? A é um parâmetro?
4. O tipo de A é inteiro (compatível com o lado direito da atribuição)?
   Se não o for, há um erro de contexto.
5. Qual o endereço de A no código objeto?

Note-se que nenhum desses itens tem a ver diretamente com a análise sintática, que se limita a reconhecer um comando de atribuição e informar quais são os lados direito e esquer-

do do mesmo. Na PASCAL, no caso de um comando **go to** 5 onde 5 não foi declarado como rótulo (**label**), há um erro de contexto. No entanto a sintaxe é perfeita.

Um exemplo clássico de erro de contexto é o seguinte:

```
  :
var B: Boolean;
  :
... B + 5 ...
  :
```

Temos aí um trecho sintaticamente correto, mas com mistura não admissível de tipos.

A geração de código é a parte do compilador que gera o programa objeto. O código é gerado sempre para determinadas unidades sintáticas, sendo utilizadas informações fornecidas pelo analisador de contexto. Por exemplo A: = 5 poderia gerar o seguinte código em HAL (v. apêndice III):

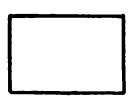
```
LDA =5
STA B1+12
```

onde B1 + 12 é o endereço de A (como veremos no capítulo 7, B1 + 12 significa que A é a 13ª variável declarada no programa principal).

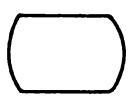
É interessante observar que em muitos compiladores uma parte da análise de contexto é efetuada pelo analisador léxico, que neste caso constrói e manipula as tabelas com as variáveis, rótulos, constantes, nomes de procedimentos, funções e tipos, em lugar do analisador de contextos.

2.3 ESTRUTURA FUNCIONAL DO COMPILADOR

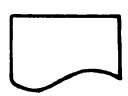
Neste item apresentaremos a estrutura funcional detalhada de nosso compilador. A esta altura, muitos pontos não ficarão claros para o leitor. Posteriormente, quando abordarmos cada parte do compilador, esses pontos serão devidamente esclarecidos. Essa estrutura é apresentada por meio do diagrama da fig. 2.6, que será referenciada em vários itens do resto deste trabalho. São usadas as seguintes convenções:



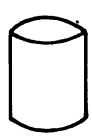
procedimento (rotina)



dados de entrada



listagem

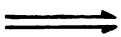


armazenamento intermediário

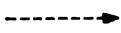
tabela residente na memória

fluxo de dados

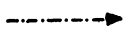
chamada de procedimento



conversão manual



execução opcional



consulta a tabela

Passemos a uma breve explanação desse diagrama.

A rotina central do compilador é, claramente, o *analisador sintático* (abreviado por AS, bloco **n**). Quando a análise sintática necessitar de um novo item léxico, é chamado (ligação **10**) o *analisador léxico* (abreviado por AL, bloco **f**) que retorna parâmetros de saída com as informações sobre esse item. Para isso, o AL lê o programa fonte (**e**) produzindo a listagem do mesmo (**g**), bem como de eventuais erros léxicos (**i**). Durante o teste do compilador, é interessante em certas fases da implementação conhecer-se explicitamente cada item léxico; o programador pode especificar opcionalmente que ele deseja uma listagem desses itens (**h**). Certos itens léxicos serão símbolos reservados (p.ex. **begin**, **if**, ':=', '*' etc.); essa informação deve ser dada pelo AL ao AS. Para decidir se um item léxico é um símbolo reservado, o AL chama a rotina de símbolos reservados (**b**), que consulta (**3**) uma tabela onde estes estão armazenados (**c**). No início da compilação, essa tabela é lida (**1**) e montada (**2**), a partir de dados fornecidos ao programa (**a**); uma listagem opcional dessa tabela (**d**) pode ser especificada.

O AS a ser usado neste texto segue um método particular. Ele aceita uma determinada classe de gramáticas, que podem ser especificadas por meio de um *grafo sintático* (**j**), o qual nada mais é do que um diagrama da sintaxe da linguagem fonte. Esse grafo será convertido manualmente (**11**) para uma tabela (**k**), cujo formato é adequado para a sua codificação como dados. Essa tabela será lida por um *carregador sintático* (**l**), o qual a converte numa segunda tabela (que denominaremos de *estrutura sintática* (**m**)); essa tabela segue uma estrutura de dados mais adequada à sua utilização pelo AS. A estrutura sintática especifica quais as ações que devem ser tomadas pelo AS em cada momento da análise: se deve ser lido novo item léxico, quais itens léxicos estão sendo procurados, ou se uma nova unidade sintática (v. 2.2) deve ser procurada ou acabou de ser reconhecida. O AS é bem geral, pois alterando-se a tabela sintática (**k**) muda-se a linguagem fonte a ser analisada. Nesse sentido, diz-se que ele é do tipo *controlado por tabela* ("table driven").

O AS utiliza para seu funcionamento uma estrutura de dados em forma de pilha (**o**). Isto se deve ao fato de que, se uma unidade sintática U é composta de outras unidades, estas devem ser analisadas primeiramente, para em seguida voltar-se à análise de U. Ao interromper a análise de U, é "empilhada" informação de como retornar para o ponto de interrupção. Além disso, é empilhada também informação sobre o número de unidades sintáticas que devem ser reconhecidas como a próxima unidade na árvore sintática (v. fig. 2.5). Mensagens especificando os erros sintáticos detectados pelo AS são produzidas pelo mesmo (**p**). Opcionalmente, pode ser obtida uma listagem da análise sintática (**q**); neste caso é necessário utilizar uma estrutura de dados em forma de pilha (*pilha sintática*, **r**). Na verdade, não se obtém a ár-

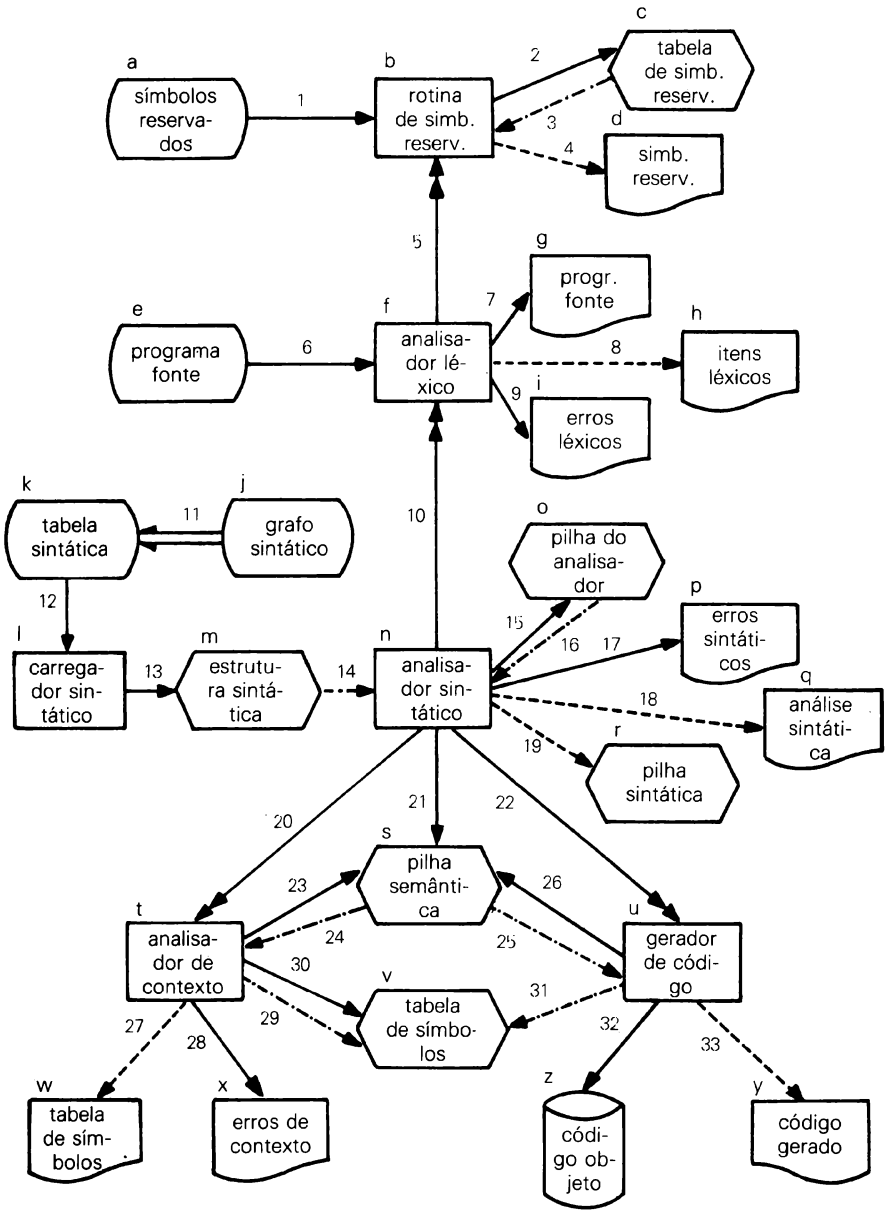

a
símbolos reservados
1
b
rotina de simb. reserv.
2
c
tabela de simb. reserv.
3
4
d
simb. reserv.
5
e
programa fonte
6
f
analisador léxico
7
g
progr. fonte
8
h
itens léxicos
9
i
erros léxicos
10
k
tabela sintática
11
j
grafo sintático
12
l
carregador sintático
13
m
estrutura sintática
14
n
analisador sintático
15
16
17
o
pilha do analisador
p
erros sintáticos
18
q
análise sintática
19
r
pilha sintática
20
21
22
s
pilha semântica
23
24
25
26
t
analisador de contexto
u
gerador de código
27
28
29
30
31
32
33
v
tabela de símbolos
w
tabela de símbolos
x
erros de contexto
z
código objeto
y
código gerado

Fig. 2.6

vore sintática completa, pois isso sobrecarregaria demasiado o AS; são apenas listadas as unidades sintáticas na ordem em que elas são reconhecidas e como aparecem na pilha sintática. A partir dessa listagem pode-se facilmente construir, manualmente, aquela árvore.

Informações "semânticas" sobre cada novo item léxico reconhecido pelo AL são colocadas pelo AS na *pilha semântica* (**s**). Essas informações são usadas e eventualmente modificadas tanto pelo *analisador de contexto* (**t**) como pelo *gerador de código* (**u**), abreviados por AC e GC respectivamente. Por exemplo, na declaração de uma variável, o AS coloca o seu identificador (nome) na pilha semântica e chama o AC para determinar o seu tipo e colocar nome, tipo, endereço no programa objeto e outras informações na *tabela de símbolos* (**v**); na geração de código objeto para uma operação aritmética qualquer o GC consulta (**25**) a pilha semântica para saber qual a operação a ser gerada e quais os operandos a serem usados (e cujos endereços são obtidos da tabela de símbolos, (**31**)).

Durante a construção do AC, é interessante poderem se obter listagens da tabela de símbolos (**w**). Erros de contexto são listados pelo AC (**x**). Da mesma maneira é interessante obter-se eventualmente uma listagem do programa objeto gerado pelo GC (**y**).

O código gerado pelo GC é gravado em um arquivo (**z**). Em nosso caso, a linguagem objeto será a linguagem de montagem HAL para o computador hipotético HIPO. Assim sendo, devemos acrescentar os passos adicionais da fig. 2.7 ao processo de compilação para se chegar à execução do programa objeto, caso desejada.

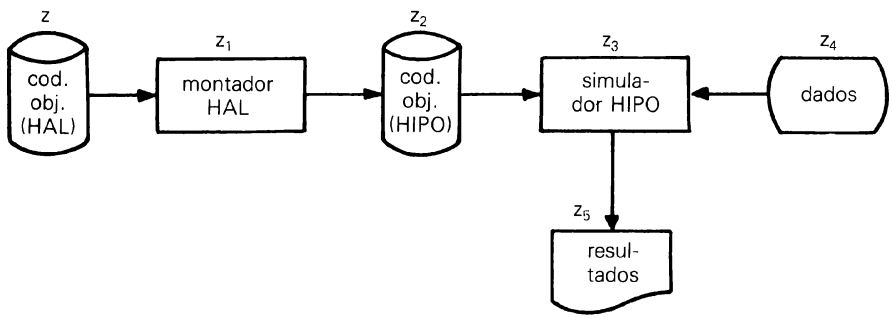


Fig. 2.7

O montador HAL (z_1) converte o código objeto escrito em HAL para a linguagem de máquina do computador HIPO (z_2). A seguir, o simulador dessa máquina (z_3) executa o código objeto, lendo dados (z_4) e produzindo finalmente os resultados supostamente esperados (z_5).

2.4 PROJETO — PARTE I: ROTINA DE SÍMBOLOS RESERVADOS

Nesta 1ª parte deverá ser programado o procedimento que lê a tabela de símbolos reservados e consulta-a, indicando se determinada seqüência de caracteres pertence ou não a ela. Suporemos que o computador a ser usado na implementação do compilador tenha uma palavra de 6 caracteres. Desta forma, deve ser executado o seguinte roteiro:

a) Leia os símbolos reservados, construindo com eles uma tabela, empregando uma técnica de "espalhamento" ("Hashing")/KNU 73, cap. 6.4/. Por exemplo, construa uma tabela em que cada entrada é composta de dois campos, que denominaremos de SÍMBOLO e COLISÃO. Essa tabela é dividida em duas partes consecutivas, que denominaremos de "Tabela Primária" e de "Tabela de Colisões". Suponhamos que os índices das entradas da primeira parte variem de 0 a m-1, e os da segunda de m a m+n. O número m deve ser um nú-

mero primo. Em geral toma-se m um pouco maior do que o número de símbolos a serem guardados.

Para introduzir um símbolo 'x' na tabela, tome a representação inteira de 'x' ou de uma parte mais à esquerda de 'x' (por exemplo, só os primeiros quatro caracteres). Divida essa representação por m, obtendo um resto r. Esse r será o índice da entrada da Tabela Primária onde 'x' deve ser introduzido. Existem três casos a considerar. i) Se a entrada de índice r estiver vazia, coloque 'x' no campo SÍMBOLO dessa posição e zero no seu campo COLISÃO. Se essa entrada não estiver vazia, isto é, contém algum símbolo 'y' introduzido anteriormente, há uma colisão entre 'x' e 'y', havendo duas possibilidades: ii) O campo COLISÃO de 'y' contém zero ou iii) um índice c_1 diferente de zero. No caso (ii), introduza 'x' na próxima entrada vazia da Tabela de Colisões; seja c o seu índice; coloque zero no campo COLISÃO dessa entrada de índice c; introduza c no campo COLISÃO da entrada r. Dessa maneira criamos uma "Cadeia de Colisões" para 'y'. No caso (iii) percorre-se a cadeia de colisões de 'y', que começa na entrada da Tabela de Colisões de índice c_1, até achar uma entrada dessa tabela com campo COLISÃO igual a zero; seja r o índice dessa entrada Repete-se o procedimento do caso (ii), colocando-se 'x' na próxima entrada vazia dessa tabela. Com isso, incrementa-se a cadeia de colisões de 'y' de mais um elemento.

b) Programe uma função do tipo "Boolean" que recebe como parâmetro uma palavra contendo uma seqüência de até 6 caracteres, alinhada à esquerda e preenchida com brancos à direita. A função, usando a técnica de "Hashing", testa se esse parâmetro pertence à tabela, devolvendo o valor true em caso positivo, e false em caso contrário.

O compilador deve ler inicialmente um registro de controle, onde é especificado se se deseja ou não a impressão da tabela de símbolos reservados e/ou a impressão de cada símbolo consultado juntamente com o valor retornado pela função. Além disso, nesse registro de controle devem ser fornecidos os tamanhos m e n das tabelas como descrito acima.

A lista de símbolos reservados da PASCAL que usaremos no projeto é a seguinte:

a) Símbolos especiais:

\+ − * / := . , ; : ' < > < <= >= > ( ) [ ] { } .. ↑

Além desses, será empregado o símbolo $, que não consta da especificação da linguagem.

b) Palavras reservadas:

and, array, begin, case, const, div, do, downto, else, end, file, for, func, goto, if, in, label, mod, nil, not, of, or, packed, proc, progr, record, repeat, set, then, to, type, until, var, while, with.

Se a cadeia de impressão do computador a ser usado não contém alguns dos símbolos, troque-os por outros ou substitua-os por palavras reservadas, por exemplo:

↑ por @
{ por /*
} por */
<> por **neq**

É interessante observar que essa rotina será alterada posteriormente, eliminando-se a leitura da tabela, cujos elementos serão obtidos do carregador sintático a ser visto no capítulo 5. Em termos da fig. 2.5 isso significa desenhar-se um arco do carregador sintático (**f**) para a rotina de símbolos reservados (**b**) eliminando-se a entrada (**a**).

CAPÍTULO 3

O ANALISADOR LÉXICO

3.1 INTRODUÇÃO

Examinando-se o processo de análise léxica, verifica-se que este segue certa estrutura particular. De fato, a função do analisador léxico (AL) resume-se a varrer o programa fonte da esquerda para a direita, agrupando os símbolos de cada item léxico e determinando sua classe. Exemplos dessas classes são os identificadores (SONIA, PAULO, X10), as palavras reservadas (**begin, if**) e os números inteiros (10, 040). Essa estrutura simples pode ser representada por meio de uma máquina abstrata, chamada Autômato de Estados Finitos ("finite state automata", abreviado aqui por AF). Essa representação não é mero formalismo: ela pode ser usada como metodologia de projeto de ALs, como veremos neste capítulo. Não entraremos a fundo na teoria de AFs, que e bastante extensa (v. p. ex. /VEL 79/); usaremos apenas o necessário para a descrição do AL e de maneira quase informal. Além disso, usaremos AFs como introdução a gramáticas formais.

3.2 BREVE INTRODUÇÃO AOS AUTÔMATOS DE ESTADOS FINITOS

Seja o diagrama da fig. 3.1. Podemos considerá-lo como uma máquina que assume um de dois estados distintos: s_0 ou s_1, representados por círculos. Inicialmente a máquina encontra-se em s_0, o que é indicado pela flecha "in", denominado *estado inicial*.

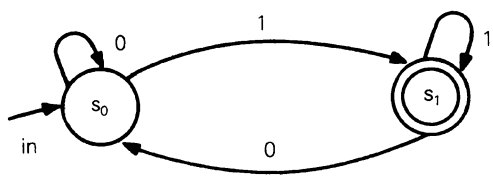


Fig. 3.1

O estado s_1 é especial, representado por dois círculos: é um *estado final*.
A máquina pode mudar de estado fazendo uma transição, isto é, estando em um esta-

do s_i, ela pode mudar para s_j (no nosso exemplo, i, j = 0,1). Isto é feito por meio da *leitura* de um símbolo de entrada, representado no arco orientado, que se dirije de s_i para s_j. Assim, vemos na fig. 3.1 que, estando em s_0, se a máquina ler um '1' ela passará para o estado s_1; por outro lado, se ela ler um '0', passará para s_0, isto é, continuará no mesmo estado em que estava antes dessa *transição*. Por outro lado, temos duas transições em s_1: lendo um '1', continua em s_1, lendo um '0', passa para s_0. Podemos imaginar que essa máquina tem uma unidade de fita ligada a ela (fig. 3.2); nessa fita temos gravada uma *cadeia de entrada*. Partindo do estado inicial, a máquina lê os símbolos seqüencialmente, efetuando uma transição para cada símbolo lido, segundo o *diagrama de estados* da fig. 3.1. Depois de ler cada símbolo, a máquina comanda a fita, que é movida por uma célula para a esquerda, de modo que a cabeça de leitura fica posicionada no símbolo seguinte. O *controle finito* consiste, em síntese, do diagrama de estados e do mecanismo que comanda o movimento da fita e lê o símbolo apontado pela cabeça.

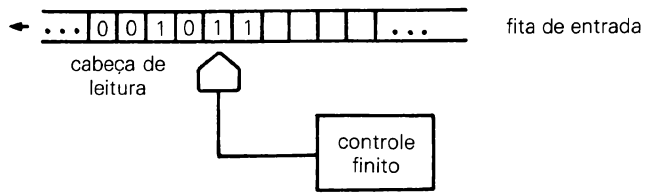


Fig. 3.2

Dizemos que uma cadeia de n símbolos $K = k_1\ k_2 \ldots k_n$, $n \geq 0$ é *aceita* ou *reconhecida* por um AF quando, partindo-se do estudo inicial desse AF, forem lidos todos os símbolos de K e efetuadas as correspondentes transições, de modo que, ao ler-se k_n, o AF passa a um estado final. Assim, no AF da fig. 3.1 a cadeia de entrada 001011 produz as seguintes transições:

$$s_0 \xrightarrow{'0'} s_0 \xrightarrow{'0'} s_0 \xrightarrow{'1'} s_1 \xrightarrow{'0'} s_0 \xrightarrow{'1'} s_1 \xrightarrow{'1'} s_1$$

Como s_1 é o estado final, dizemos que essa cadeia é aceita pelo autômato. Por outro lado, a cadeia 001010 produz as transições:

$$s_0 \xrightarrow{'0'} s_0 \xrightarrow{'0'} s_0 \xrightarrow{'1'} s_1 \xrightarrow{'0'} s_0 \xrightarrow{'1'} s_1 \xrightarrow{'0'} s_0$$

Ela *não* é aceita, pois o estado atingido na última transição, s_0, não é um estado final.

O conjunto contendo todas as cadeias aceitas por um AF, e apenas estas, é denominado de *linguagem aceita* por esse AF. No caso do AF da fig. 3.1, a linguagem aceita é o conjunto de todas as cadeias de '0' e '1' terminadas em '1', isto é, o conjunto dos números binários com correspondente decimal ímpar.

Como um segundo exemplo, seja o AF da fig. 3.3. Ele aceita todas as cadeias com zero ou mais 'a' e 'b' de modo que haja um número par (inclusive zero) de 'a's.

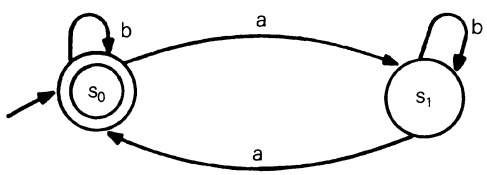


Fig. 3.3.

A cadeia com nenhum 'a' e nenhum 'b', ou melhor, sem símbolo algum, é denominada de *cadeia vazia*, e será representada por λ. Ela é aceita por esse AF pois o estado inicial é, também, um estado final. Assim, o AF começa no estado inicial s_0; se a fita de entrada contiver apenas a cadeia vazia (isto é, não há nada gravado nela), o AF permanece em s_0 sem efetuar nenhuma transição. Ora, já que a cadeia de entrada é vazia, podemos considerar que ela foi totalmente "lida"; após essa "leitura", o AF encontra-se em um estado final (s_0), portanto essa cadeia foi "aceita"

Para compreender-se ainda um pouco melhor o significado de λ, definamos o *comprimento* de uma cadeia $K = k_1 k_2 \ldots k_n$, $k_i \neq \lambda$, $i = 1, \ldots, n$ como o número de símbolos de K, no caso, n. Podemos definir a cadeia vazia como sendo a cadeia de comprimento nulo. No caso dos AF, o comprimento da cadeia lida corresponde ao número de transições efetuadas pelo autômato. Evidentemente, a "leitura" da cadeia vazia corresponde a zero transições.

A linguagem aceita pelo AF da fig. 3.3 é portanto: {λ, b, aa, bb, aab, aba, baa, bbb, aaaa, aabb, abab, ...}. Para completar essa introdução passemos a definir formalmente um AF. Note-se que necessitamos especificar quais os estados do autômato, qual desses é o estado inicial e quais são os estados finais (podem ser mais do que um); qual o conjunto de símbolos que podem ser gravados na fita de entrada e a especificação das transições. Esta assumirá a forma de uma aplicação no sentido matemático, levando um par (estado, símbolo de entrada) a um estado.

Definição: um autômato finito M é uma quíntupla ordenada

$$M = (S, s_0, F, A, g) \text{ onde}$$

S é um conjunto finito, $S \neq \Phi$, denominado *conjunto de estados*;
$s_0 \in S$, denominado *estado inicial*;
$F \subset S$, não-vazio, denominado de *conjunto de estados finais*;
A é um conjunto finito, denominado *alfabeto de entrada*;
$g: S \times A \rightarrow S$, denominada *aplicação de transição*.

Por exemplo, o autômato da fig. 3.1 é definido por:

$$(\{s_0, s_1\}, s_0, \{s_1\}, \{0,1\}, g) \text{ onde}$$

$$g(s_0, 0) = s_0,\ g(s_0, 1) = s_1,\ g(s_1, 0) = s_0,\ g(s_1, 1) = s_1.$$

Note-se que g não precisa estar necessariamente definida para todos os possíveis pares (estado, símbolo de entrada), isto é, para um estado podemos ter transições apenas para alguns símbolos de entrada.

3.3 IMPLEMENTAÇÃO DE AUTÔMATOS FINITOS EM COMPUTADORES

Um AF pode ser facilmente implementado em um computador, construindo-se um programa que execute exatamente as transições por ele especificadas. Por exemplo, na fig. 3.4 temos um programa em PASCAL para o autômato da fig. 3.1.

Os estados s_0 e s_1 são implementados por meio dos rótulos '0' e '1', respectivamente; os rótulos '2' e '3' são auxiliares, indicando os comandos onde são produzidos a mensagem de não-aceitação da cadeia de entrada e o término do programa, respectivamente. As transições são implementadas pela leitura do próximo símbolo e desvio para o rótulo correspondente ao estado desejado. A função EOF (input) deve assumir o valor true se for detetado o fim do arquivo de entrada ("end of file"), isto é, se todos os símbolos da cadeia de entrada já tiverem sido lidos. No estado s_0, um "end of file" indica que a cadeia terminou mas não se está em um estado final, não sendo a mesma aceita; por outro lado, em s_1 essa mesma situação indica a aceitação da cadeia, pois esse estado é final.

Esse programa é particular para o AF da fig. 3.1. Cada autômato pode ser dessa maneira simulado pela execução de um particular programa escrito de maneira análoga. Esse processo de implementação de programas específicos para cada autômato pode ser generalizado através de um simulador geral, que aceita como entrada inicial a descrição de um AF sob forma de uma tabela de transições e uma lista de estados finais, passando em seguida à leitura

```
program AF31 (input, output);
  label 0,1,2,3;
  var entrada: char;
  begin
    0: read (entrada);
       if EOF (input) then goto 2;
       if entrada = '0' then goto 0;
       if entrada = '1' then goto 1;
       goto 2;
    1: read (entrada);
       if EOF (input) then begin write ('cadeia aceita');
                                 goto 3 end;
       if entrada = '1' then goto 1;
       if entrada = '0' then goto 0;
    2: write ('cadeia não aceita');
    3: end.
```

Fig. 3.4

da cadeia de entrada, e executando as transições por meio de uma *interpretação* dessa tabela. Exemplificando, para o AF da fig. 3.1 teríamos a tabela da fig. 3.5.

TRANSIÇÃO	ESTADO ATUAL	SÍMBOLO LIDO	PRÓXIMO ESTADO	PRÓXIMA TRANSIÇÃO	
1	s_0	0	s_0	1	ESTADOS FINAIS: s_1
2	s_0	1	s_1	3	
3	s_1	1	s_1	3	
4	s_1	0	s_0	1	

Fig. 3.5

Nesta figura, a parte interior tracejada representa as transições do AF propriamente dito. Para simplificar o processo de interpretação, podemos agrupar todas as transições de um mesmo estado em linhas consecutivas e supor que o estado inicial seja o primeiro da tabela. Além disso, adicionamos duas colunas extras, que indicam uma numeração das transições, e uma conversão do "próximo estado" no número da primeira transição (de cima para baixo) correspondente ao seu grupo. Finalmente, é fornecida a lista de estados finais.

O funcionamento do interpretador é basicamente o seguinte. Suponhamos que L seja um indicador, contendo o número da linha sendo interpretada e que será aqui usado como ín-

dice para os elementos das colunas de uma tabela como a da fig. 3.5. Suponhamos ainda que E contenha o símbolo de entrada atual e S o estado atual do autômato. Inicialmente fazemos L:= 1. Ao interpretar uma linha L qualquer, verificamos se E = SÍMBOLOLIDO[L]. Se verdadeiro, fazemos S: = PRÓXIMOESTADO[L];a seguir L:= PRÓXIMATRANSIÇÃO[L] e em seguida lê-se novo E, passando-se à interpretação dessa nova linha L; com isso, efetuamos uma transição. Se E ≠ SÍMBOLOLIDO[L], fazemos L:= L + 1, verificando em seguida se S = ESTADOATUAL[L]; em caso afirmativo, ainda há pelo menos uma transição desse estado S, que deve ser tentada, interpretando-se a nova linha L como descrito acima. Se S ≠ ESTADOATUAL[L], E não corresponde a nenhuma transição para S, e portanto a cadeia não é aceita. Sempre que se lê novo E, é necessário testar se foi atingido o fim do arquivo de entrada (EOF). Em caso afirmativo e se S é um estado final, a cadeia é aceita, caso contrário ela não é aceita.

Observe-se que essa interpretação é mais lenta do que a execução do programa da fig. 3.4. No entanto, o programa interpretador mais a tabela da fig. 3.5 podem requerer muito menos espaço de memória, principalmente se o autômato tiver muitas transições. O maior ganho, no entanto, reside no fato de que a programação de cada autômato consiste unicamente na construção da tabela correspondente: o interpretador é universal.

3.4 O ANALISADOR LÉXICO COMO UM AUTÔMATO FINITO

Inicialmente, daremos um detalhamento das funções do AL, para em seguida passarmos à sua representação como AF.

Na versão da linguagem PASCAL aceita pelo nosso compilador, teremos as seguintes classes de itens léxicos:

C_1: identificadores. São seqüências de letras e dígitos, começando à esquerda com uma letra, não pertencentes à tabela de símbolos reservados. Qualquer caractere diferente de letra e dígito delimita um identificador à direita.

C_2: palavras reservadas. Como os identificadores, pertencendo a seqüência a tabela de símbolos reservados.

C_3: números inteiros. Seqüência de dígitos. Qualquer caractere diferente de dígito delimita um número à direita.

C_4: símbolos especiais. Qualquer seqüência de um ou dois caracteres diferentes de letra, dígito e '␣' (branco), pertencente à tabela de símbolos reservados.

C_5: fim de arquivo (EOF). Detetado automaticamente pela unidade de entrada do AL; não se trata de um caractere externo gravado após o último item léxico.

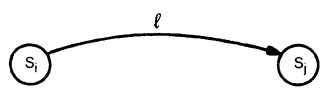
Deixamos de incluir a classe de cadeias de caracteres que ficará como exercício (v. 3.6).

Note-se que as classes C_1 e C_2 são idênticas do ponto de vista da seqüência de caracteres do item léxico. Para simplificar a estrutura do autômato, este não fará distinção direta entre elas, ficando a mesma por conta de mecanismo "semântico" a ser descrito posteriormente. Na fig. 3.6 apresentamos o AF em questão, que reconhece um único item léxico; ele dará origem a um procedimento que será chamado pelo analisador sintático (v. fig. 2.6) para cada novo item léxico requerido por este. Para simplificar o diagrama de estados agrupamos várias transições em uma só. Temos as seguintes denotações:

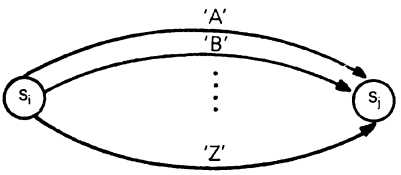
ℓ = {'A', 'B', ..., 'Z'}
d = {'0', '1', ..., '9'}
c = conjunto de todos os caracteres

A notação c-d-ℓ indica o complemento relativo do conjunto c, isto é, todos os caracteres que não são dígitos nem letras.

Assim, a transição:



representa na realidade o conjunto de transições:



Na fig. 3.6 indicamos em certas transições, entre parênteses, uma *ação "semântica"* a_i, que será explicada no item 3.5. Note-se que o estado s_1 corresponde ao reconhecimento de um identificador ou palavras reservadas, s_2 ao de um número, s_3 a um símbolo especial e s_b a um comentário, que é qualquer serqüência de caracteres entre '/*' e '*/', que usaremos para ilustração em lugar de '{'e'}'.

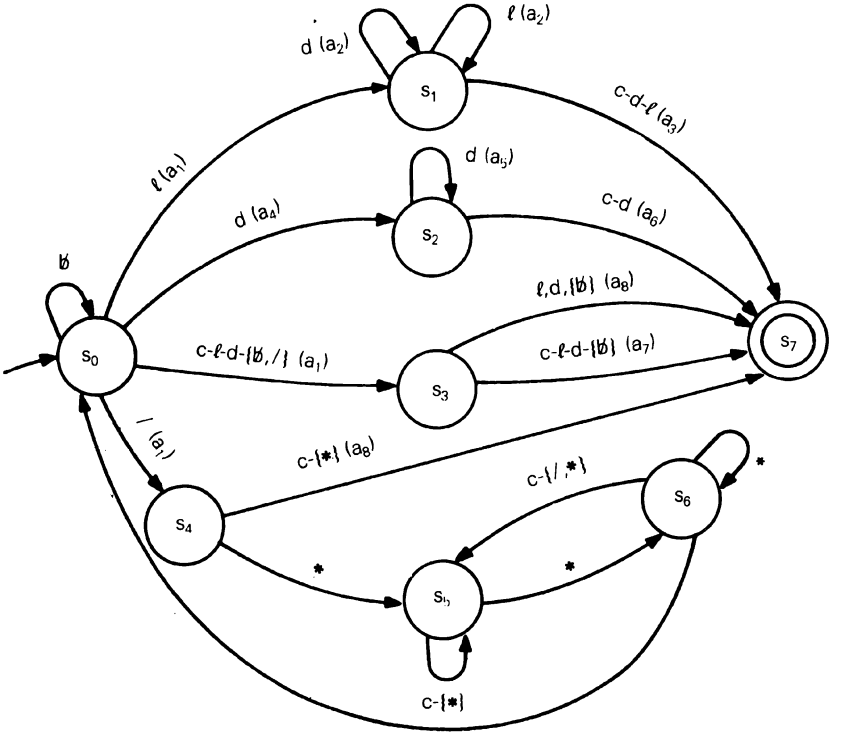


Fig. 3.6

Como dissemos anteriormente, o AF da fig. 3.6 dará origem a um procedimento, que pode ser implementado segundo as diretrizes indicadas em 3.3. Esse procedimento retorna

três parâmetros p_1, p_2 e p_3, cujo conteúdo é especificado na tabela da fig. 3.7; suporemos que o computador usado na implementação do compilador tem uma palavra de 6 caracteres. Os parâmetros que são seqüências de caracteres são sempre ajustados à esquerda, e completados com 'ϐ' à direita quando necessário.

classe	p_1	p_2	p_3
C_1 — identificador	'IDEN'	6 primeiros caracteres do identificador	—
C_2 — palavra reservada	caracteres da palavra reservada	igual a p_1	—
C_3 — número inteiro, sem sinal de até 10 algarismos	'NUMB'	—	o número na representação interna (binária)
C_4 — símbolo especial reservado	caracteres do símbolo especial	igual a p_1	—
C_5 — fim de arquivo	'$'	'$'	—

Fig. 3.7

Note-se que na classe C_1 são usados apenas os 6 primeiros caracteres, de modo que os caracteres restantes devem ser ignorados, podendo servir para compor um nome representativo do significado do identificador. Por outro lado, restringiremos a classe C_2 apenas a palavras reservadas de até 6 caracteres; com isso serão usadas as seguintes abreviaturas em nossa versão de PASCAL: **program** — **progr**; **procedure** — **proc**; **function** — **func.** Na classe C_3 impomos uma restrição ao tamanho dos números inteiros, compatível com o tamanho da palavra do computador HIPO da linguagem objeto. Quando um número for maior do que 10^{10}-1, deve haver impressão de mensagem de erro. No caso da classe C_5, importa notar que "fim de arquivo" é uma característica do computador onde é processado o compilador; não se trata do símbolo **$** que é usado como parâmetro de saída do AL. Assim, não se deve entender que o programa fonte precisa ser encerrado com um **$**.

O parâmetro p_1 será usado pelo analisador sintático, contendo o item léxico propriamente dito; p_2 e p_3 serão usados pelo analisador de contexto (v. fig. 2.6.).

3.5 AÇÕES "SEMÂNTICAS" DO ANALISADOR LÉXICO

Em alguns arcos do AF da fig. 3.6 estão indicadas ações "semânticas" como por exemplo (a_1). Essas ações podem ser entendidas como procedimentos que completam a atuação do AF, construindo os parâmetros da saída p_1, p_2 e p_3 do AL. Para a descrição dessas ações usaremos a linguagem PASCAL. Supomos que haja um tipo declarado globalmente, da seguinte maneira:

type alpha = **record** simb: **packed array** [1..6] **of** char **end**;

que será usado como tipo dos parâmetros p_1 e p_2. Além disso, são globais duas funções

func RES (p: alpha): Boolean; **begin**/* esta função busca a tabela de símbolos reservados e assume o valor true se p pertence a essa tabela, ou false caso contrário*/ **end**

func ALPHINT (p: char): integer; **begin**/* esta função converte um dígito na representação externa char para uma palavra do tipo integer (binária)*/ **end**

A seguir apresentamos o procedimento do analisador léxico com as declarações das ações semânticas. O simulador do AF da fig. 3.6 foi omitido; note-se que cada ação semântica deve ser chamada após a simulação da respectiva transição, antes da mudança de estado. Supomos que a cadeia de entrada é lida pelo AF e colocada no **array** 'entrada' de 80 caracteres declarado no início do procedimento, e apontado pelo indexador i. Este deve ser atualizado pelo AF, após a leitura de cada caractere. Note-se que no caso de a_3, a_6 e a_8 é necessário voltar-se um símbolo de leitura para trás, pois foi lido um símbolo a mais, de modo que na próxima chamada o AL inicie a análise do item léxico seguinte em seu primeiro caractere. Isto também acontece em um caso de a_7. Observe-se ainda que nesse procedimento não incluímos as detecções de erros, quais sejam a de um número inteiro exceder o valor limite (a_5) e a de um símbolo especial não pertencer à tabela de símbolos reservados (a_7 e a_8).

```
proc ANALEX (var p1, p2: alpha; var p3: integer);
  var entrada: array [1..80] of char; i,j: integer;
  proc a1; begin
             p2:= 'ббббббб'; j:= 1; p2. simb [j]:= entrada [i]
           end;
  proc a2; begin
             j:= j+1; if j ≤ 6 then p2. simb [j]:= entrada [i]
           end;
  proc a3; begin
             if RES (p2) then p1:= p2 else p1:= 'IDEN';
             i:= i-1
           end;
  proc a4; begin p3:= ALPHINT (entrada [i]) end;
  proc a5; begin p3:=p3 * 10+ ALPHINT (entrada [i]) end;
  proc a6; begin p1:= 'NUMB'; i:= i-1 end;
  proc a7; begin
             p2. simb[2]:= entrada [i];
             if not RES (p2)
               then begin p2. simb [2]:= 'б'; i:= i-1 end;
             p1:= p2
           end;
  proc a8; begin p1:= p2; i:= i-1 end;
  proc a9; begin p1:= '$'; p2:= '$' end;
  begin
    /*simulador do AF*/
  end;
```

3.6 PROJETO — PARTE II: ANALISADOR LÉXICO

Nesta 2ª Parte deverá ser programado o procedimento que efetua a análise léxica do programa fonte a ser compilado. A cada chamada do AL, este deve retornar o próximo item léxico ainda não reconhecido, bem como as informações "semânticas" adicionais, em três parâmetros de saída. Os itens léxicos e os respectivos delimitadores são os descritos em 3.4. Acrescente a esses itens duas classes adicionais, ampliando o AF da fig. 3.6, a tabela da fig. 3.7 e as ações semânticas do item 3.5. A primeira é a classe "cadeia de caracteres", que é

uma cadeia delimitada à esquerda e à direita por um apóstrofo ('), em que brancos não são ignorados; entre esses delimitadores, um apóstrofo deve ser escrito com dois (''). Os parâmetros de saída devem ser os seguintes, para esta classe: p_1: 'STRING'; p_2: 6 primeiros caracteres da cadeia (sem os delimitadores). A segunda é a classe "comandos de controle do compilador". Esta classe consiste de uma cadeia de caracteres delimitada tanto à esquerda como à direita pelo símbolo '$'. A cadeia consiste de uma lista de comandos, separados por vírgulas. A cada comando corresponde uma ação que deve ser executada imediatamente pelo compilador. Implemente os seguintes comandos que dizem respeito às partes I e II do compilador:

1. SIMRES — produz a impressão da tabela de símbolos reservados.
2. LISTON — produz a impressão dos registros do programa fonte sendo compilado, a partir do próximo item léxico.
3. LISTOF — interrompe a impressão dos registros do programa fonte, até que seja novamente reconhecido o comando LISTON.
4. ITLEON — imprime os três parâmetros de saída do analisador léxico para cada item léxico reconhecido, a partir do próximo item léxico.
5. ITLEOF — interrompe a impressão dos parâmetros de saída, até que seja novamente reconhecido o comando ITLEON.

Exemplo de um item léxico dessa classe: $SIMRES, LISTON$. Note-se que, ao terminar o reconhecimento de um item dessa classe, o analisador léxico deve buscar o próximo item léxico do programa fonte, como se se tivesse tratado de um comentário. A leitura do registro de controle descrito no item 2.4b deve ser eliminada do compilador após a implementação dessa classe.

O formato dos registros de entrada pode ser o seguinte:

- cada registro tem 80 caracteres (colunas)
- somente os primeiros 72 caracteres contêm os itens léxicos
- o formato do programa-fonte é livre, sendo que um item léxico pode ser interrompido na 72ª coluna, continuando na 1ª coluna do registro seguinte.

Os parâmetros de saída do procedimento seguem as especificações da fig. 3.7.

CAPÍTULO 4

EXPRESSÕES REGULARES E GRAMÁTICAS

4.1 INTRODUÇÃO

No capítulo anterior examinamos a descrição do analisador léxico (AL) como um autômato finito (AF). Neste capítulo, faremos uma transição entre o AL e o analisador sintático (AS). De fato, este último não pode ser descrito como um AF, pois seu nível de complexidade ultrapassa a capacidade deste. Por exemplo, como vimos, o AF reconhece os símbolos da cadeia de entrada seqüencialmente, da esquerda para a direita, daí ter-se prestado tão bem para representar as funções do AL, que segue essa característica. Na fig. 2.5 vimos como o AS deve reconhecer inicialmente certas partes da cadeia de entrada, para posteriormente reconhecer outros trechos, que eventualmente vieram antes (isto é, mais à esquerda). Esse retorno para símbolos já lidos não pode ser descrito por meio de um AF.

Para atingirmos esses objetivos, introduziremos a noção de *gramática* que, por motivos didáticos, será apresentada inicialmente como uma notação especial, não-gráfica, da aplicação de transição do AF (v. 3.2). Posteriormente, as gramáticas serão estendidas transcendendo o nível dos AFs.

4.2 EXPRESSÕES REGULARES

O conjunto das cadeias aceitas por um AF foi denominado de *linguagem aceita* (ou *reconhecida*) por esse AF (v. 3.2). Esse conjunto foi caracterizado por meio de uma descrição informal ou por meio de uma enumeração exaustiva dos seus elementos. Uma outra maneira de apresentar aquela linguagem consiste na dedução de uma fórmula denominada de *expressão regular.* Um componente essencial dessas fórmulas é a representação de concatenações sucessivas de uma cadeia.

Assim, se x é uma cadeia qualquer,

$$x^* = \{\lambda, x, xx, xxx, \ldots\}$$

que se denomina *fechamento transitivo de concatenação.*
Por exemplo,

$$(01)^* = \{\lambda, 01, 0101, 010101, \ldots\}$$

Esse fechamento pode ser concatenado com outras cadeias:

$$a\ b\ c^*\ d = \{abd, abcd, abccd, abcccd, \ldots\}$$

Com essa notação, podemos exprimir a linguagem reconhecida pelo AF da fig. 3.1 por meio da seguinte fórmula: (0*1*)*1 que equivale a 0*1*0*1* ... 0*1*1. Escolhendo-se convenientemente os fechamentos de concatenação, pode-se exprimir qualquer número binário terminado em "1", que constituía uma cadeia aceita pelo autômato.

O AF da fig. 3.3 reconhece a linguagem expressa por (b*ab*a)*b*, isto é, qualquer cadeia de a's e b's com um número par, inclusive zero, de a's.

Essas fórmulas são denominadas de *expressões regulares* (ER), que passamos a definir formalmente, de maneira indutiva:

Definição:

a) Um símbolo qualquer é uma expressão regular;
b) Se x e y são expressões regulares, a sua concatenação xy é uma expressão regular;
c) Se x é uma expressão regular, x* é uma expressão regular;
d) A união de duas expressões regulares é uma expressão regular;
e) A expressão obtida por um número finito de aplicações de (b) a (d) é regular.

Deixaremos neste texto de empregar ainda certas formas de ERs, pois não serão aqui utilizadas.

Pode-se provar que ERs são *equivalentes* a AFs /SAL 69/, isto é, dado um AF existe uma ER que exprime a linguagem aceita por esse autômato; dada uma ER, existe um AF que aceita todas as cadeias representadas por ela, e apenas estas.

4.3 GRAMÁTICAS

Tomemos o AF da fig. 3.1. A transição de s_0 para s_1, que é feita pela leitura de um '1', pode ser denotada por $s_0 \rightarrow$ '1' s_1, denominado de *produção*. Observe-se que essa notação corresponde diretamente à aplicação de transição para o par $(s_0, 1)$, isto é, $g(s_0, 1) = s_1$. Damos abaixo a transcrição de todas as aplicações de transição do AF mencionado nessa notação (compare-se com essa aplicação conforme vista no item 3.2):

$s_0 \rightarrow 0s_0$
$s_0 \rightarrow 1s_1$
$s_1 \rightarrow 1s_1$
$s_1 \rightarrow 0s_0$

Para que possamos descrever todas as características de um AF nessa notação, falta especificarmos o estado inicial, no caso s_0, e os estados finais. Estes serão denotados por uma produção especial para os mesmos, com o símbolo 'λ' no lado direito. Assim, em nosso caso, indicaremos que s_1 é um estado final pela produção $s_1 \rightarrow \lambda$. Podemos agrupar as produções com um mesmo estado no lado esquerdo da flecha separando os lados direitos por meio de uma barra vertical, obtendo:

$s_0 \rightarrow 0s_0 \mid 1s_1$
$s_1 \rightarrow 1s_1 \mid 0s_0 \mid \lambda$ $\quad (P_1)$

Chamamos a atenção para o fato de que essa é simplesmente uma maneira de descrever um AF; especificando-se que s_0 é o estado inicial, temos nessa descrição todas as informações do diagrama de estados da fig. 3.1.

Como um exemplo adicional, teremos as seguintes produções correspondentes ao AF da fig. 3.3 (o estado inicial é novamente s_0):

$s_0 \rightarrow bs_0 \mid as_1 \mid \lambda$
$s_1 \rightarrow bs_1 \mid as_0$ $\quad (P_2)$

Os diagramas de estado dos AFs representam muito bem esses autômatos como reconhecedores de cadeias. A notação que acabamos de introduzir adapta-se melhor, por outro lado, à representação do processo oposto, isto é, de *geração* de cadeias. Para isso, podemos encarar as produções como regras de formação de cadeias por meio de substituições sucessivas de símbolos. Assim, por exemplo, no conjunto de produções P1 anterior, deve-se entender por $s_1 \rightarrow 0s_0$ que o símbolo s_1 pode ser substituído por $0s_0$, formando nova cadeia. A cadeia inicial é sempre constituída pelo estado inicial do AF. Cada passo da geração será repre-

sentado por $x \stackrel{*}{\rightarrow} y$; y é obtido a partir da cadeia x por meio da substituição de algum dos símbolos desta, segundo alguma produção. Assim, a partir de P_1 podemos gerar

$$s_0 \stackrel{*}{\rightarrow} 0s_0 \stackrel{*}{\rightarrow} 00s_0 \stackrel{*}{\rightarrow} 001s_1 \stackrel{*}{\rightarrow} 0010s_0 \stackrel{*}{\rightarrow} 00101s_1 \stackrel{*}{\rightarrow} 001011s_1 \stackrel{*}{\rightarrow} 001011$$

Note-se que em cada etapa gera-se uma nova cadeia, tendo-se aplicado uma produção de P_1; assim, na 3ª etapa substitui-se s_0 por $1s_1$, tendo-se aplicado a produção $s_0 \rightarrow 1s_1$. No último passo, o símbolo s_1 é substituído por λ, segundo $s_1 \rightarrow \lambda$. Recordemos que λ é a cadeia vazia (v. 3.2), que concatenada a qualquer cadeia dá origem a esta própria cadeia. Com essa última etapa, encerramos o processo de geração, pois não há mais nenhum símbolo a ser substituído. Assim, a produção $s_1 \rightarrow \lambda$ de P_1, que foi introduzida apenas para indicar que s_1 é um estado final, adquire um outro significado (que não deixa de ser análogo ao primeiro), isto é, indica o encerramento do processo de geração. O AF podia ter sido usado como "máquina de geração", em lugar de "máquina de reconhecimento". Para isso, parte-se do estado inicial e de uma cadeia inicial λ; para cada transição acrescenta-se o símbolo desta à direita da cadeia gerada até esse momento. Ao atingir-se o estado final, pode-se encerrar o processo de geração; a cadeia assim obtida é evidentemente aceita pelo mesmo AF, pois foram seguidos exatamente os passos de reconhecimento. Fica claro, portanto, que qualquer cadeia aceita pelo AF da fig. 3.1 pode ser gerada por P_1, e vice-versa. O mesmo se passa com o AF da fig. 3.3 e P_2.

Analogamente, os conjuntos de produções também podem ser considerados como regras de reconhecimento de cadeias.

Para denotarmos zero ou mais passos de gerações, usaremos o símbolo $\stackrel{*}{\rightarrow}$. Assim, temos, por exemplo, para P_1: $s_0 \stackrel{*}{\rightarrow} 0s_0$; $s_0 \stackrel{*}{\rightarrow} 00s_0$; ...; $s_0 \stackrel{*}{\rightarrow} 001011$. Mas temos também: $001s_1 \stackrel{*}{\rightarrow} 00101s_1$ etc.

Neste ponto, podemos usar as produções para introduzir a noção e definição de gramáticas. A parte principal de uma gramática será um conjunto de regras de substituição de cadeias (produções); a ele devemos acrescentar a especificação de um *símbolo inicial*, que constituirá a primeira cadeia dos processos de geração, como foi o caso de s_0 no exemplo de 001011 para P_1. Além desses elementos, serão especificados ainda dois conjuntos: o de símbolos auxiliares ou *não-terminais*, que são usados apenas para descrever as regras de substituição, e que não devem fazer parte das cadeias finais geradas; o segundo conjunto, contendo os símbolos das cadeias finais, ou símbolos *terminais*. Formalmente, temos:

Definição: uma *gramática* G é uma quádrupla ordenada (V_N, V_T, S, P), onde:

V_N é um alfabeto (isto é, um conjunto finito) de símbolos *não-terminais*;

V_T é um alfabeto de símbolos *terminais*, com $V_N \cap V_T = \Phi$;

$S \in V_N$ é o *símbolo inicial*;

P é um conjunto finito de *produções*, da forma $\alpha \rightarrow \beta$, tal que $\alpha \in (V_N \cup V_T)^* - \{\lambda\}$ e $\beta \in (V_N \cup V_T)^*$

A notação X^*, onde X é um conjunto de cadeias, indica o conjunto de cadeias obtidas pela concatenação sucessiva de elementos de X, unido com $\{\lambda\}$. Assim,

$$\{a,b\}^* = \{\lambda, a, b, aa, ab, ba, bb, aaa, aab, aba, \ldots\}$$

Como um exemplo, definimos a gramática G_1 que tem P_1 como conjunto de produções:

$$G_1 = (\{s_0, s_1\}, \{0,1,\lambda\}, s_0, \{s_0 \rightarrow 0s_0, s_0 \rightarrow 1s_1, s_1 \rightarrow 1s_1, s_1 \rightarrow 0s_0, s_1 \rightarrow \lambda\})$$

Como outro exemplo, o conjunto de produções P_2 visto anteriormente pode ser considerado como pertencendo a uma gramática

$$G_2 = (V_N, V_T, s_0, P_2)$$

onde $V_N = \{s_0, s_1\}$ e $V_T = \{a,b,\lambda\}$.

Formalizemos agora como gerar cadeias a partir de gramáticas.

Definição: Data uma gramática $G = (V_N, V_T, S, P)$, dizemos que uma cadeia $\alpha\gamma\beta$ *gera diretamente* uma cadeia $\alpha\delta\beta$, com $\alpha, \beta \in (V_N \cup V_T)^*$ se $\gamma \rightarrow \delta$ é uma produção de P. Notação:

$$\alpha\gamma\beta \stackrel{*}{\rightarrow} \alpha\delta\beta$$

No exemplo para P_1 já visto, temos:

$$001s_1 \stackrel{*}{\rightarrow} 0010s_0$$

Neste caso temos $\alpha = 001$, $\beta = \lambda$, $\gamma = s_1$ e $\delta = 0s_0$

Definição: Dada uma gramática $G = (V_N, V_T, S, P)$, dizemos que uma cadeia α *gera* uma cadeia β se existir uma seqüência de gerações diretas para G, da forma

$$\alpha = \alpha_1 \stackrel{*}{\rightarrow} \alpha_2 \stackrel{*}{\rightarrow} \ldots \stackrel{*}{\rightarrow} \alpha_n = \beta,\ n \geq 1$$

Notação: $\alpha \stackrel{*}{\rightarrow} \beta$

No mesmo exemplo para P_1 temos:

$$001s_1 \stackrel{*}{\rightarrow} 001011s_1$$

Definição: Dizemos que uma cadeia $\alpha \in (V_N \cup V_T)^*$ é uma *forma sentencial* de uma gramática $G = (V_N, V_T, S, P)$ se $S \stackrel{*}{\rightarrow} \alpha$.

Por exemplo, para P_1 temos as formas sentenciais:

$$s_0, 0s_0, 00s_0, \ldots, 001011$$

Definição: A *linguagem gerada* por uma gramática $G = (V_N, V_T, S, P)$, denotada por L(G), é o conjunto de todas as cadeias que possuem apenas símbolos terminais, e que são geradas a partir de S, ou seja,

$$L(G) = \{\omega \in V_T^* : S \stackrel{*}{\rightarrow} \omega\}$$

Exemplo: a linguagem gerada por G_1 é:

$$L(G_1) = \{1, 01, 11, 001, 011, 101, 111, 0001, \ldots\}$$

Definição: — Duas gramáticas G' e G'' são *equivalentes* se e somente se L(G') = L(G''), isto é, se elas geram a mesma linguagem.

4.4 GRAMÁTICAS REGULARES

No item anterior, vimos como se pode deduzir um conjunto de produções de uma gramática, correspondendo às transições de um AF. Vamos estabelecer a correspondência de todo AF com uma gramática.

Seja um AF $M = (E, e_0, F, A, g)$ (v. item 3.2). A ele faremos corresponder uma gramática $G = (V_N, V_T, S, P)$, da seguinte maneira: $V_N = E$; $V_T = A \cup \{\lambda\}$; $S = e_0$, e P é construído como visto em 4.3, isto é, introduzimos em P uma produção

$N \rightarrow tQ$ para cada transição $g(N,t) = Q$ e

$N \rightarrow \lambda$ para cada $N \in F$.

Note-se que os estados finais são representados por produções especiais, donde não se ter em G um conjunto explícito correspondendo a F de M.

Observando-se a definição de gramática dada em 4.3, vê-se que ela é mais geral do que as gramáticas deduzidas a partir das AFs. De fato, ela permite produções que não seguem as formas acima (do lado direito, um terminal seguido de um não-terminal, ou apenas λ). Por exemplo, pela definição podemos ter produções da forma $N \rightarrow QRt$, $tNR \rightarrow Q$ etc., com $N,Q,R \in V_N$ e $t \in V_T$. Damos, a seguir, uma definição de um tipo restrito de gramáticas, mais "próximas" às formas de produções correspondentes a transições (e estados finais) de AFs.

Definição: uma gramática $G = (V_N, V_T, S, P)$ diz-se *regular* (GR) ou do *tipo* 3, se as produções de P são da forma

$$N \rightarrow tM \quad \text{ou} \quad N \rightarrow t'$$

onde $N, M \in V_N$ e $t, t' \in V_T$ e $t \neq \lambda$.

Por exemplo, seja $G_3 = (V_N, V_T, S, P_3)$ onde

$V_N = \{S,M\}$, $V_T = \{a,b,c\}$

e

$$\begin{aligned} S &\rightarrow bS \mid aM \mid c \\ M &\rightarrow bS \end{aligned} \qquad (P_3)$$

Essa gramática é regular e gera cadeias com 'a's e 'b's sempre terminando com 'c', de tal modo que cada 'a' sempre é seguido de um 'b'. No entanto, ela não corresponde a um AF, da maneira como foi descrito no início deste item, pois na conversão das transições desses autômatos em produções, nunca aparece uma produção do tipo $N \rightarrow t$, $N \in V_N$, $t \in V_T$ e $t \neq \lambda$. No entanto, em P_3 temos uma produção $S \rightarrow c$. Entretanto, é muito simples mudar-se P_3 de tal modo que se obtenha um novo conjunto de produções P'_3 de uma gramática G'_3, tal que

$L(G_3) = L(G_3')$, correspondendo a um AF: basta trocar $S \rightarrow c$ por $S \rightarrow cN$ e adicionar $N \rightarrow \lambda$, obtendo-se

$S \rightarrow bS \mid aM \mid cN$
$M \rightarrow bS$ (P_3')
$N \rightarrow \lambda$

O diagrama de estados do AF seria:

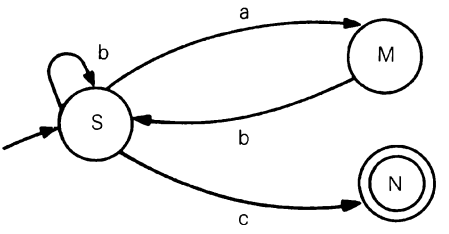


Fig. 4.1

Evidentemente, esse autômato reconhece a mesma linguagem gerada por G_3' e, portanto, por G_3.

Vê-se, portanto, que dada uma GR qualquer, pode-se construir uma gramática equivalente, de modo que ela corresponda a um AF. Podemos assim enunciar o seguinte

Teorema: dada uma gramática regular G, existe um AF M equivalente a ela, isto é, tal que L(G) é igual à linguagem aceita por M.

Evidentemente, o contrário também é verdade, isto é, dado um AF, existe uma GR equivalente.

Vejamos agora um AF equivalente à seguinte gramática G_4 (deste ponto em diante, usaremos letras maiúsculas para os não-terminais e minúsculas para os terminais, e S como símbolo inicial, menos em casos explicitamente mencionados quando a gramática tiver mais de um não-terminal; assim, omitiremos a descrição da quádrupla ordenada, bem como os conjuntos de símbolos, já que podem ser imediatamente deduzidos dos conjuntos de produções):

$S \rightarrow aM \mid aN$
$M \rightarrow b$ (G_4)
$N \rightarrow c$

O AF equivalente seria, seguindo o método exposto,

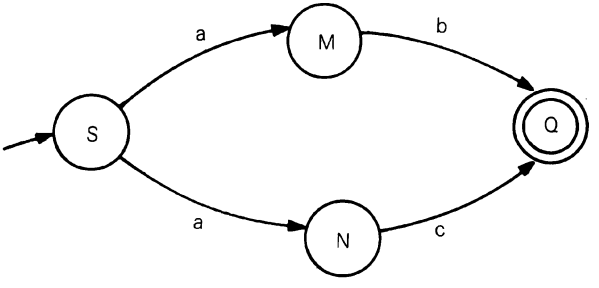


Fig. 4.2

Note-se que, estando no estado S, o AF, ao ler um 'a', não pode decidir-se pela transição para M ou para N. Dizemos que esse autômato é *não-determinístico*. Note-se que a definição de AFs do item 3.2 engloba esse tipo de autômato. De fato, temos g(s,a) = M e g(s,a) = N; daí termos chamado g de *aplicação* de transição, e não *função* de transição.

Definição: um AF $M = (E,e_0,F,A,g)$ é dito *não-determinístico* se existirem $e_1,e_2,e_3 \in E$ e $a \in A$ tal que $g(e_1,a) = e_2$ e $g(e_1,a) = e_3$, com $e_2 \neq e_3$; caso contrário ele é dito *determinístico*.

Pode-se demonstrar /H-U 69/ que, dado um AF não-determinístico, existe um AF determinístico equivalente, isto é, que aceita a mesma linguagem. No caso do AF da fig. 4.2, teríamos o equivalente:

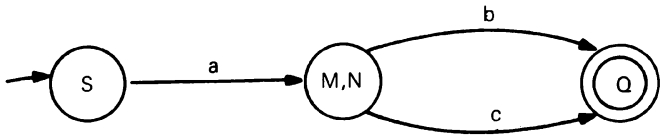


Fig. 4.3

O estado M,N substitui os estados M e N. Essa substituição é, em síntese, a transformação sistemática que se faz para obter o outro tipo de AF.

Por analogia, chamaremos G_4 de *gramática regular não-determinística*.

A equivalência de GRs e AFs aplica-se também aos casos não-determinísticos.

Recordando a equivalência de expressões regulares com AFs (v. 4.2), podemos também enunciar a equivalência de GRs com expressões regulares. Assim, dada uma expressão regular E, existe uma gramática regular G equivalente, isto é, L(G) = E, e vice-versa.

4.5 GRAMÁTICAS LIVRES DE CONTEXTO E ÁRVORES SINTÁTICAS

Como vimos no item anterior, a definição de gramáticas abrange tipos destas que não se conformam às restrições impostas às produções das GRs. Vejamos neste item um tipo de gramáticas cujas produções são menos restritas do que as das GRs:

Definição: uma gramática $G = (V_N, V_T, S, P)$ diz-se *livre de contexto* (GLC) ou do *tipo 2* se as produções de P são da forma $N \rightarrow \alpha$ onde $\alpha \in (V_N \cup V_T)^*$ e $N \in V_N$.

Por exemplo,

$E \rightarrow T \mid E+T \mid E-T$
$T \rightarrow F \mid T*F \mid T/F$
$F \rightarrow i \mid F**i$ $\qquad (G_5)$

com símbolo inicial E, é uma GLC. Ela gera expressões aritméticas com a variável 'i' contendo somas, subtrações, multiplicações (*), divisões e exponenciações (**). Por exemplo, temos $E \stackrel{*}{\rightarrow} i-i*i+i**i$. Essa geração de ser representada passo a passo por meio de uma *árvore sintática*, apresentada na fig. 4.4.

Note-se a correspondência com o que já dissemos sobre árvores sintáticas, na introdução em 2.2, isto é, elas fornecem a *estrutura gramatical* da cadeia analisada. Vemos agora que essa estrutura é baseada na gramática que gera a cadeia. Nesta árvore temos a representação da geração de todas as formas sentenciais, até a cadeia terminal:

$$E \stackrel{*}{\rightarrow} E+T \stackrel{*}{\rightarrow} E-T+T \stackrel{*}{\rightarrow} T \stackrel{*}{\rightarrow} T+T \stackrel{*}{\rightarrow} F-T+T \stackrel{*}{\rightarrow} i-T+T \stackrel{*}{\rightarrow} i-T*F+T \text{ etc.}$$

Na árvore, não temos a ordem explícita em que foi feita a derivação.

Formalmente, temos:

Definição: dada uma GLC $G = (V_N, V_T, S, P)$,

a) A árvore constituída inicialmente pelo nó rotulado S é uma árvore sintática.

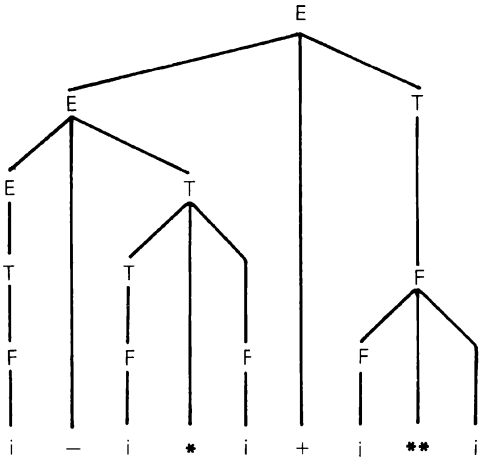
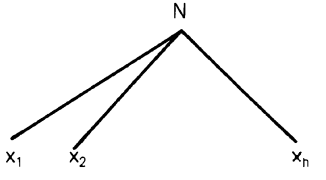


Fig. 4.4

b) A árvore obtida pela substituição de uma folha (nó final) rotulada N de uma árvore sintática, pela árvore



onde $N \rightarrow x_1x_2...x_n \in P$, é uma árvore sintática.

Essa definição indutiva corresponde exatamente à construção de uma árvore sintática para uma determinada forma sentencial da gramática G.

As seguintes observações sobre essas árvores são pertinentes:

a) A construção da árvore termina quando todas as folhas forem símbolos terminais.

b) A cada árvore sintática corresponde pelo menos uma geração (v. 4.3).

c) A cada geração corresponde apenas uma única árvore sintática (embora várias gerações possam ter a mesma árvore).

d) Qualquer nó não-folha é rotulado com um não-terminal de G.

e) Um nó não-terminal N e seus nós descendentes imediatos $x_1, x_2, \ldots, x_n$ correspondem a uma geração direta $N \stackrel{\bullet}{\rightarrow} x_1 x_2...x_n$.

A observação (c) acima sugere a introdução de um importante conceito:

Definição: Dada uma gramática $G = (V_N, V_T, S, P)$, ela se diz *ambígua* se existir uma cadeia $x \in L(G)$ para a qual podem ser construídas duas árvores sintáticas diferentes.

Exemplo: seja

$E \rightarrow E+E \mid E*E \mid i \qquad (G_6)$

para a cadeia $i+i*i$, podemos ter as duas árvores distintas seguintes:

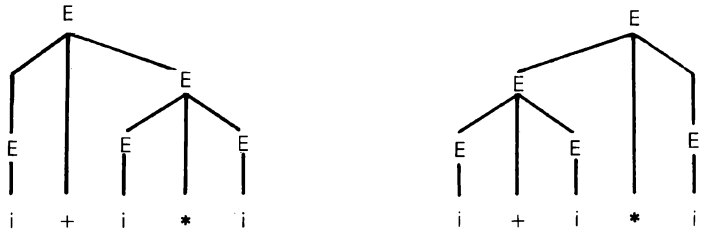


Fig. 4.5

Assim, G_6 é ambígua.

Como contra-exemplo, seja

$S \rightarrow MN$

$M \rightarrow a$ $\quad (G_7)$

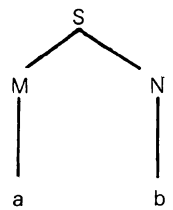
$N \rightarrow b$

Para esta gramática temos 2 derivações diferentes da única cadeia gerada por ela, ab:

$S \dot{\rightarrow} MN \dot{\rightarrow} aN \dot{\rightarrow} ab$ e

$S \dot{\rightarrow} MN \dot{\rightarrow} Mb \dot{\rightarrow} ab$

No entanto, há uma só árvore sintática correspondente à geração:



Devido a esse fato é que se utilizam árvores sintáticas como critério de ambigüidade, e não gerações. Pode-se particularizar a maneira de fazer a geração, obtendo-se uma outra caracterização para ambigüidade. Para isso, a cada geração direta devemos substituir sempre o símbolo não-terminal mais à direita. Esse tipo de geração chama-se geração canônica.

Definição: Uma geração direta $xNy \dot{\rightarrow} xuy$ com $N \epsilon V_N$ e $x,y,u \epsilon (V_N \cup V_T)^*$ é *canônica* se y contém apenas símbolos terminais.

Usando essa definição podemos dizer que uma gramática G é ambígua se, e somente se, existir uma cadeia $x \epsilon L(G)$ para a qual existe mais do que uma geração canônica. Essa definição é equivalente à anterior.

Como exemplo, seja

$S \rightarrow S+S \mid i$ $\quad (G_8)$

A cadeia i+i+i tem duas gerações canônicas:

$S \dot{\rightarrow} S+S \dot{\rightarrow} S+S+S \dot{\rightarrow} S+S+i \dot{\rightarrow} S+i+i \dot{\rightarrow} i+i+i$

$S \dot{\rightarrow} S+S \dot{\rightarrow} S+i \dot{\rightarrow} S+S+i \dot{\rightarrow} S+i+i \dot{\rightarrow} i+i+i$

Portanto, G_8 é ambígua. Note-se que G_7 tem uma só geração canônica:

$S \dot{\rightarrow} MN \dot{\rightarrow} Mb \dot{\rightarrow} ab$

4.6 COMPARAÇÃO ENTRE GRAMÁTICAS DE TIPO 2 E 3

Ao introduzir-se GLC, pode ter surgido uma dúvida: qual a sua vantagem frente às GRs? Pela definição, pode-se observar que todas as GRs são GLC. As produções daquelas são, co-

mo vimos, mais simples, e pode parecer estranho querer-se empregar gramáticas mais complicadas. A resposta a essa pergunta reside justamente nessa maior complexidade. De fato, podemos justificar o uso de GLCs com dois argumentos:

a) As GLCs permitem a construção de árvores sintáticas mais "complexas". A seguinte gramática regular com símbolo inicial E é equivalente a G_5 (v. 4.5):

$E \rightarrow iM \mid i$
$M \rightarrow +E \mid -E \mid *E \mid /E \mid **E$ (G_9)

A árvore sintática correspondente à cadeia i-i*i+i**i é:

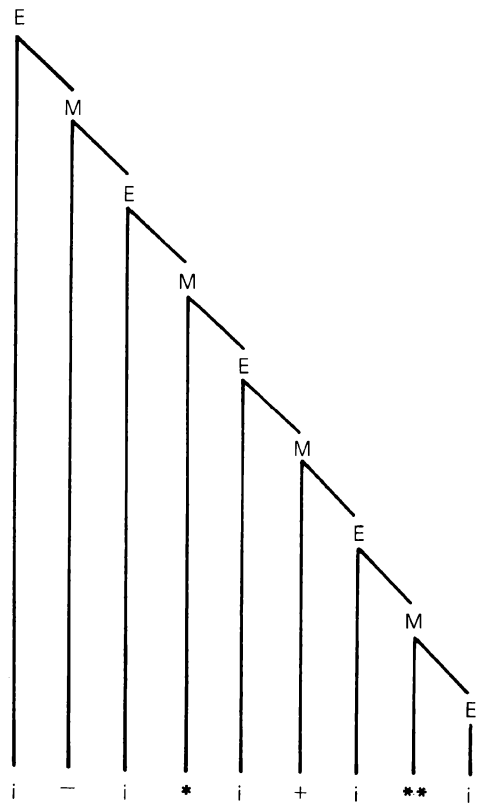


Fig. 4.6

Todas as árvores sintáticas de GRs têm essa forma característica, isto é, são árvores binárias onde cada nó não-terminal tem necessariamente um descendente terminal e eventualmente um descendente não-terminal à direita do primeiro. Há como que uma forma "linear" nessas árvores. As gerações para GRs são sempre canônicas: existe sempre um só não-terminal em cada forma sentencial. Por outro lado examinando-se a árvore sintática da fig. 4.4 (v. 4.5), verificamos que esta revela uma estrutura diferente para a mesma cadeia. Note-se como a multiplicação (*) com seus operandos constitui uma unidade sintática (T), que posteriormente deve ser encarada como um operando da subtração, reconhecida como outra unidade sintática (E). Dessa maneira, através de G_5, temos a possibilidade de exprimir uma *precedência* entre unidades sintáticas. Considere-se o processo inverso ao da geração, isto é, reconheci-

mento de cadeias através de gramáticas (v. 4.3); note-se que a árvore sintática de reconhecimento é idêntica à de geração. Verificamos que na fig. 4.4 o reconhecimento da multiplicação e seus operandos deve necessariamente preceder o da subtração ou da soma adjacentes. Essa possibilidade de exprimir precedências é altamente desejável em linguagens de programação, pelo auxílio que pode fornecer ao processo de compilação: o analisador sintático, se seguir a gramática G_5, analisará (reconhecerá) automaticamente uma multiplicação sempre antes de somas ou subtrações adjacentes. Por outro lado, segundo a GR G_9, o reconhecimento é feito linearmente, um operador após o outro, independente de suas precedências. Note-se que, segundo a G_5, o reconhecimento de dois operadores consecutivos de mesma precedência é feito da esquerda para a direita, inclusive para a exponenciação(**), contrariando esta última a interpretação usual da notação matemática.

b) O segundo argumento da necessidade de GLCs reside na maior "complexidade" das linguagens geradas (ou reconhecidas) por estas. Por exemplo, a gramática G_{10} com símbolo inicial E corresponde às expressões aritméticas de G_5, acrescidas de parênteses e de operadores aditivos ('+','−') unários:

$$\begin{aligned} &E \rightarrow T \mid +T \mid -T \mid E+T \mid E-T \\ &T \rightarrow F \mid T*F \mid T/F \\ &F \rightarrow P \mid F**P \\ &P \rightarrow i \mid (E) \end{aligned} \qquad (G_{10})$$

Esses parênteses vêm aos pares, como se espera de uma expressão bem-formada. É impossível obter-se uma GR que gere a mesma linguagem. Intuitivamente, pode-se compreender esse fato devido à característica das GRs de gerarem cadeias linearmente, isto é, sempre acrescentando um novo símbolo terminal à direita da cadeia já gerada. Com isso, "perde-se" totalmente o controle do número de parênteses esquerdos já gerados, se não há um limite para esse número. Um exemplo clássico de linguagem para a qual não existe nenhuma GR é $L_p = \{a^n b^n \mid n \geq 1\}$. A prova desse fato pode ser vista em /H-U 69/. No entanto a GLC G_{11} abaixo gera essa linguagem:

$$S \rightarrow aSb \mid ab \qquad (G_{11})$$

Note-se a analogia parcial com os parênteses de G_{10}.

Da mesma maneira como não existe uma GR que gera L_p, não existe um AF que reconhece L_p. Outro tipo de autômato deve ser introduzido para reconhecer linguagens geradas por GLC e não geradas por GR: trata-se do *autômato com pilha*, cujo esquema, análogo ao AF da fig. 3.2, é dado na fig. 4.7.

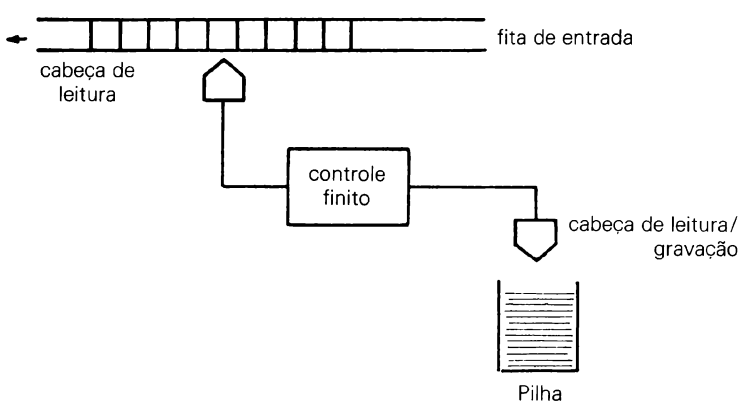


Fig. 4.7

A cabeça da pilha só pode ler o símbolo superior desta e substituir esse símbolo por uma cadeia (eventualmente vazia). Não daremos aqui a definição formal de autômatos com pilha (v. p. ex. /H-U 69/). Podemos informalmente descrever um desses autômatos que reconhece L_p: para cada 'a' lido, este é colocado na pilha; ao ler o primeiro 'b', desempilha-se um 'a', repetindo este processo para os restantes 'b's e não aceitando-se nenhum outro símbolo. Ao terminar a cadeia de entrada, se a pilha estiver vazia entra-se em um estado final.

4.7 GRAMÁTICAS LIVRES DE CONTEXTO COM EXPRESSÕES REGULARES

Para permitir a simplificação das produções de uma GLC, introduziremos a possibilidade de que o lado direito de cada produção contenha expressões regulares, envolvendo terminais e não-terminais.

Assim, por exemplo, a gramática

$S \rightarrow aS \mid b \qquad (G_{12})$

é evidentemente equivalente a

$S \rightarrow a^*b$

pois $L(G_{12}) = a^*b$

Se um não-terminal qualquer N é definido recursivamente pelas produções:

$N \rightarrow \alpha N \mid \beta,$

onde α,β ou pertençem a $(V_N \cup V_T)^*$ ou são expressões regulares construídas com símbolos de V_N e V_T, podemos substituir essas produções por

$N \rightarrow \alpha^*\beta$

Por outro lado,

$N \rightarrow N\alpha \mid \beta$

pode ser substituída por

$N \rightarrow \beta\alpha^*$

No restante deste texto, usaremos a notação $\{\alpha\}^*$ em lugar de α^*.

Podemos, além disso, introduzir fatoração de produções. Assim, as produções

$N \rightarrow \alpha\beta\delta \mid \alpha\gamma\delta$

podem ser reescritas como

$N \rightarrow \alpha(\beta \mid \gamma)\delta$

Usemos essa fatoração na gramática G_{10} (V. 4.6). A primeira linha das produções era

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

Usando a notação introduzida acima podemos escrever:

$E \rightarrow (+ \mid - \mid \lambda)\ T \mid E(+\ T \mid -\ T)$

Podemos agora substituir as recursividades, obtendo:

$E \rightarrow (+ \mid - \mid \lambda)T\{+T \mid -T\}^*$

Note-se a extensão do fechamento de concatenação aplicado a alternativas. Estendendo ainda a notação, podemos introduzir a seguinte forma: $[\alpha]$ indicando que α é opcional, isto é, pode ser substituído pela cadeia vazia λ. Este é um caso muito comum em linguagens de programação, onde freqüentemente uma certa cadeia é opcional, isto é, ocorre zero ou uma vez. As produções acima simplificam-se portanto para:

$E \rightarrow [+ \mid -]T\{+T \mid -T\}^*$

A gramática G_{10} pode, assim, ser totalmente reescrita. Para não haver confusão entre os metassímbolos, isto é, os símbolos usados na notação das produções, como (,[,{ etc., com os símbolos terminais das gramáticas, escrevemos estes últimos entre apóstrofos. Isto será feito, no entanto, somente quando houver possibilidade de confusão, como é o caso da G_{10}.

$E \rightarrow ['+' \mid '-']T\{'+'T \mid '-'T\}^*$

$T \rightarrow F\{'*'F \mid '/'F\}^*$

$F \rightarrow P\{'**'P\}^*$

$P \rightarrow 'i' \mid '('E')'$

Finalmente, introduziremos uma nova notação para outro caso muito comum em linguagens de programação. Trata-se de uma cadeia α que deve ocorrer pelo menos uma vez; se ela for repetida, deve sê-lo intercalando-se entre duas de suas ocorrências adjacentes a cadeia β; isto é, gera-se o conjunto $\{\alpha, \alpha\beta\alpha, \alpha\beta\alpha\beta\alpha, \ldots\}$.
Essa notação será
$\{\alpha \parallel \beta\}^+$
onde α e β podem conter qualquer das construções anteriores ou mesmo desta que está sendo definida.

Com essa notação, a gramática G_{10} pode ser agora reescrita em sua forma final:

$E \rightarrow$ ['+'|'−'] {T ‖ ('+'|'−')}$^+$
$T \rightarrow$ [F ‖ ('*'|'/')}$^+$ (G_{13})
$F \rightarrow$ {P ‖ '**'}$^+$
$P \rightarrow$ 'i' | '(' E ')'

As expressões assim obtidas nos lados direitos das produções serão por nós denominadas de *expressões regulares estendidas* (ERE). Façamos uma definição formal indutiva da formação dessas expressões:

Definição:

a) Uma cadeia de símbolos α é uma ERE;
b) Se α e β são EREs, $\alpha\beta$ e $\alpha|\beta$ são EREs;
c) Se α é uma ERE, então (α), $[\alpha]$, $\{\alpha\}^*$ são EREs;
d) Se α e β são EREs, $\{\alpha \parallel \beta\}^+$ é uma ERE;
e) A expressão obtida pela aplicação de (b), (c) e (d) um número finito de vezes é uma ERE.

A interpretação dessas expressões é a seguinte:

$(\alpha) = \alpha$
$[\alpha] = (\alpha|\lambda)$
$\{\alpha\}^* = \lambda|\alpha|\alpha\alpha|\ldots$
$\{\alpha \parallel \beta\}^+ = \alpha\{\beta\alpha\}^*$

As GLC em cujas produções o lado direito é uma ERE, serão por nós denotadas por *ERE-gramáticas*.

Antes de deixarmos este item, é interessante mencionar que existe uma notação especial para GLC denominada de "Backus Normal Form" ou "Backus-Naur Form", abreviada por "BNF", onde os não-terminais são delimitados por colchetes angulares, como por exemplo <statement> e o sinal '→' que separa o lado direito do lado esquerdo das produções é representado por '::='. Usaremos, quando for interessante, essa notação, empregando-a nas ERE-gramáticas.

Assim, G_{13} poderia ser escrita também como
<expressão> ::= ['+'|'−']{<termo> ‖ ('+'|'−')}$^+$ etc.

Note-se que em BNF não se costuma usar os terminais entre apóstrofos. Um exemplo clássico do uso da notação BNF é o relatório da linguagem ALGOL-60 /NAU 63/.

4.8 GRAFOS SINTÁTICOS

As ERE-gramáticas podem ser representadas diagramaticamente por meio dos assim chamados *Grafos Sintáticos*. Com isso obtém-se uma interessante visualização das gramáticas, que pode facilitar a dedução da linguagem gerada, bem como a compreensão das estruturas representadas pelas mesmas. Como veremos no próximo capítulo, esses grafos podem também facilitar a compreensão de certos métodos de análise sintática. Usaremos aqui a notação empregada por Wirth na descrição da linguagem PASCAL /J-W 74/, estendendo-a para as EREs como foram introduzidas no item anterior.

As regras de construção desses grafos são as seguintes:

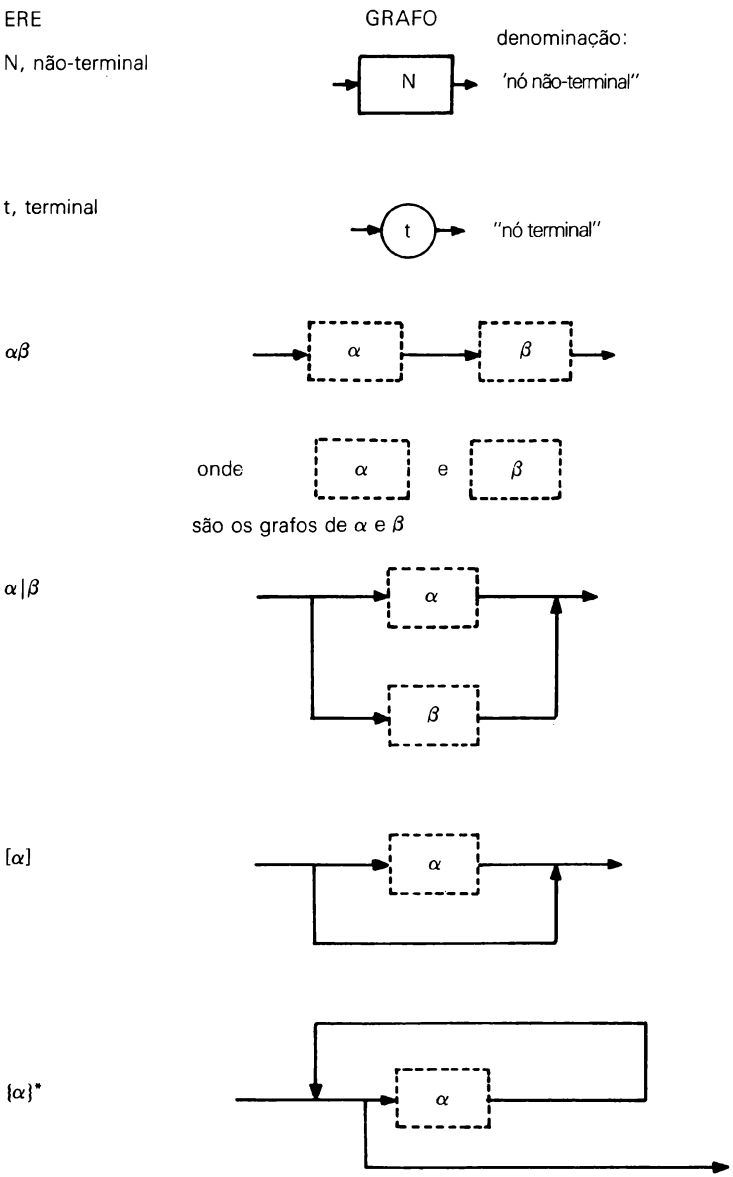


Fig. 4.8 (continua)

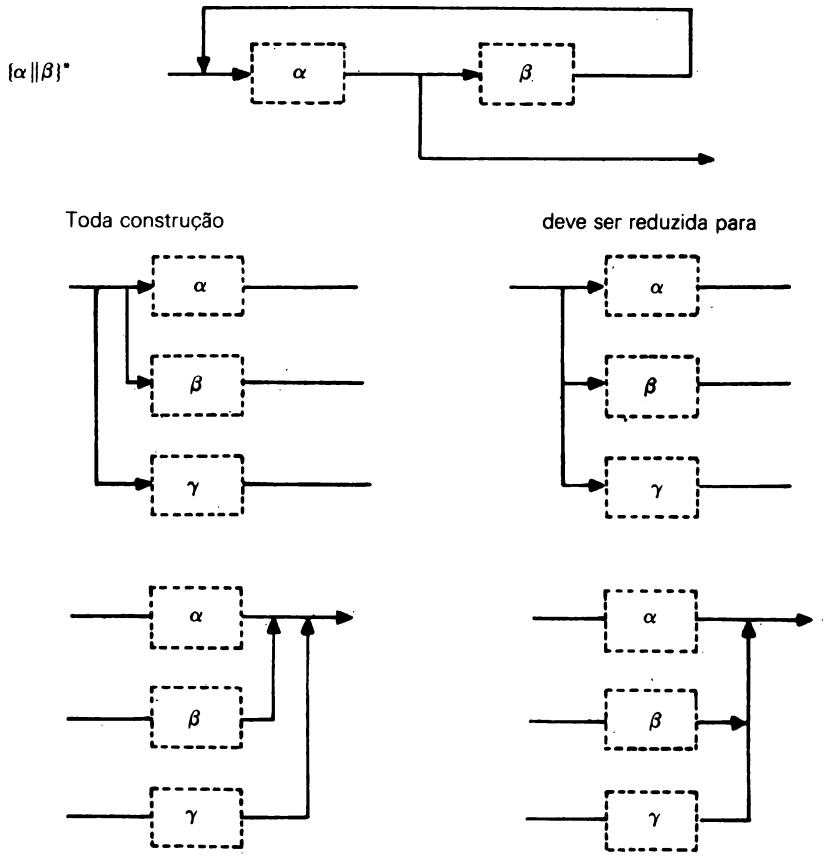


Fig. 4.8 (conclusão)

Na fig. 4.9 apresentamos duas formas de grafos que são encontrados na literatura (p. ex. /J-W 74/) e que se poderia supor como correspondentes às expressões anotadas à esquerda dos mesmos. Essas construções não devem, no entanto, ser empregadas, tomando-se em seu lugar as da fig. 4.8, onde se pode criar uma ordenação dos nós, que será necessária para a definição das gramáticas que são aceitas pelo analisador sintático a ser estudado no capítulo 5.

{α}*

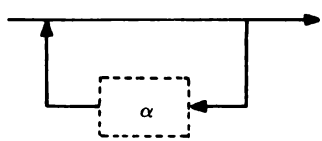


Fig. 4.9 (continua)

$\{\alpha \| \beta\}^+$

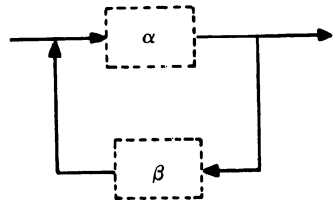
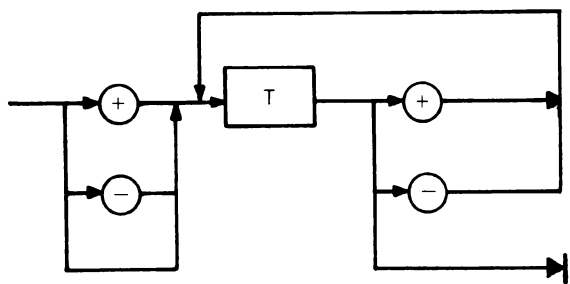


Fig. 4.9 (conclusão)

Cada produção de uma ERE-gramática será representada pelo nome do não-terminal à esquerda da mesma, seguido do subgrafo que representa o seu lado direito. Como exemplo, apresentamos na fig. 4.10 o grafo para a gramática G_{13} (v. 4.7).

Note-se que o término das produções é indicado por uma flecha que aponta para uma barra vertical.

E



T

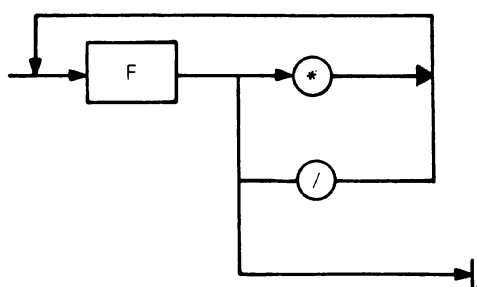


Fig. 4.10 (continua)

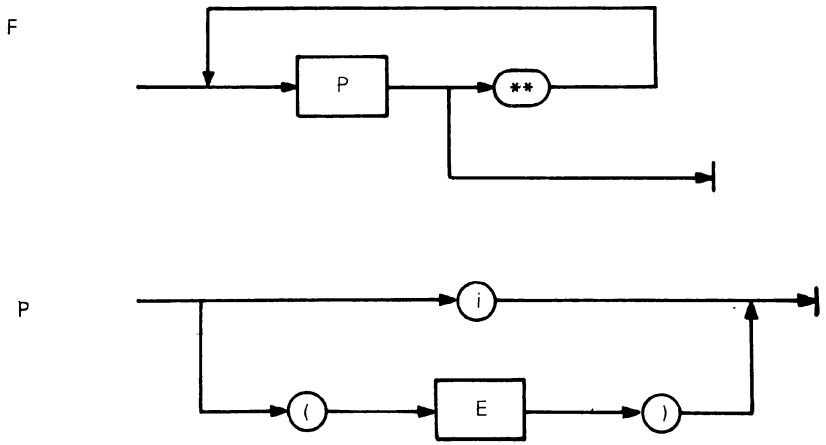


Fig. 4.10 (conclusão)

CAPÍTULO 5

O ANALISADOR SINTÁTICO

5.1 INTRODUÇÃO

No capítulo 2 fizemos uma introdução às características do AS. No item 4.5 introduzimos a noção de árvore sintática. Neste capítulo abordaremos em detalhe o processo de funcionamento de um AS, que pode construir a árvore sintática. Não trataremos de dar uma introdução a vários tipos de ASs e de suas características; procuraremos concentrar-nos em um único método que consideramos muito bom, tanto do ponto de vista didático como prático. Deste último, podemos destacar a simplicidade do método, sua generalidade, eficiência e possibilidades quanto à recuperação de erros sintáticos (v. 2.2). A concentração em um só método tem a desvantagem de não fazer uma abordagem geral sobre esse interessante campo da Ciência da Computação mas, por outro lado, tem a vantagem de não tornar o texto dispersivo.

5.2 O PROBLEMA DA ANÁLISE SINTÁTICA

Dada uma gramática $G = (V_N, V_T, S, P)$ e uma cadeia $x = t_1 t_2 \ldots t_n \in L(G)$, o AS que reconhece essa cadeia segundo a gramática G deve ter as seguintes características:

a) Ler x da esquerda para a direita, sem reler partes já lidas. Essa restrição é natural, pois os programas que serão compilados são estruturados da esquerda para a direita, como por exemplo as declarações que vêm antes dos comandos. A imposição de não reler partes já lidas deve-se a uma questão de eficiência.

b) O reconhecimento deve ser canônico, isto é, seguir os passos exatamente contrários aos de uma geração canônica (v. 4.5). Inicialmente, x constitui a forma sentencial a ser analisada; em algum ponto da análise, temos uma forma sentencial σ' proveniente da substituição de uma subcadeia α, da forma sentencial anterior σ, por um não-terminal N onde $N \rightarrow \alpha \in P$. O AS deve decidir quais símbolos de σ constituem o lado direito α da produção a ser aplicada, e substituí-lo pelo lado esquerdo N. A análise termina quando a forma sentencial sendo analisada reduzir-se a S.

c) Qualquer forma sentencial σ da análise é constituída de duas subcadeias, isto é, uma à direita, contendo a parte x_d de x ainda não lida, e outra σ_e à esquerda que contém eventualmente símbolos não-terminais, isto é, $\sigma = \sigma_e x_d$. O AS deve decidir quando o símbolo mais à esquerda de x_d deve ser lido, no caso em que não deve (ou não pode) ser feito nenhum reconhecimento dos símbolos de σ_e.

Assim, podemos resumir o problema fundamental de um algoritmo de análise sintática, como sendo o problema de decidir qual subcadeia de uma forma sentencial deve ser reconhecida como o lado direito de uma produção da gramática, fazendo em seguida a substituição desse lado direito pelo não-terminal do lado esquerdo, lendo novos símbolos da cadeia de entrada quando necessário.

Exemplificando, tomemos o caso da gramática G_5 (v. 4.5). Na fig. 5.1 vemos duas tentativas de análises para a cadeia i + i*i, onde numeramos cada passo do reconhecimento canônico colocando um número de ordem no não-terminal que foi reconhecido nesse passo.

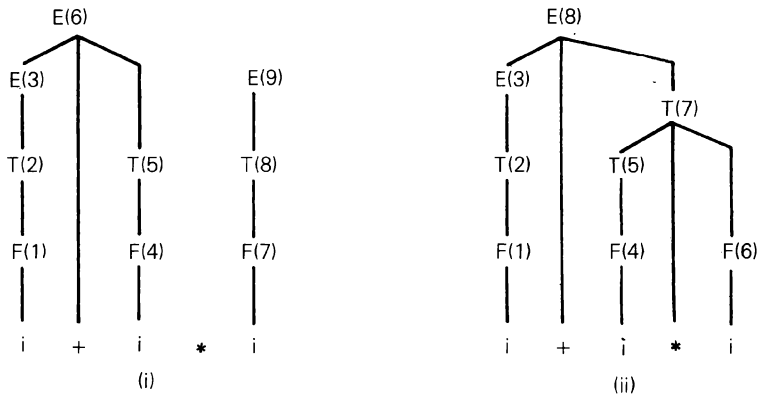


Fig. 5.1

No primeiro caso, a análise não pôde prosseguir, pois não há lados direitos de produções de C_5 contendo uma das cadeias E * i, E * F, E * T, E * E; de fato, isso poderia ter sido deduzido ao aparecer E(6), pois não há lado direito contendo a subcadeia E* ou o símbolo E isoladamente. No segundo caso, a análise foi mais feliz; todas as cadeias intermediárias continham pelo menos um lado direito de alguma produção de G_5, e conseguiu-se chegar à forma sentencial contendo apenas o símbolo inicial E. Note-se que se pode facilmente distinguir nessas árvores as cadeias intermediárias do reconhecimento, assumindo-se que em cada substituição eliminam-se da árvore os símbolos substituídos (e que constituem um lado direito de alguma produção). Somente no segundo caso todas essas cadeias são formas sentenciais.

Nesse exemplo, vê-se a dificuldade que deve enfrentar o AS, de decidir o que fazer em passos com mais do que uma possibilidade de reconhecimento, e como no reconhecimento canônico é às vezes necessário evitar alguma substituição mais à esquerda. Note-se que o reconhecimento canônico deve reconhecer sempre as subcadeias mais à esquerda levando ao símbolo inicial (contrário da geração canônica, v. 4.5).

5.3 ANÁLISE SINTÁTICA ASCENDENTE E DESCENDENTE

Existem duas importantes classes de estratégias gerais dos algoritmos que solucionam o problema da análise sintática. Seus nomes, estratégias *ascendente* e *descendente* (do inglês "bottom-up" e "top-down" conforme tradução de T. Kowaltowski /KOW 83/), indicam sua principal característica: na primeira, a árvore sintática é construída partindo-se da cadeia a ser analisada, "subindo-se" até atingir o símbolo inicial da gramática; na segunda, parte-se do símbolo inicial e vai-se "descendo" até atingir todos os símbolos da cadeia. Na fig. 5.1 (ii) temos um exemplo de análise ascendente; na fig. 5.2 a mesma cadeia é analisada com a estratégia descendente. Neste caso, a numeração dos símbolos não é feita somente com os não-

terminais como no caso ascendente, mas também com os terminais. Números iguais indicam um lado direito de uma produção, cujos símbolos ligados ao não-terminal de cima passam a fazer parte da árvore. Em qualquer ponto do reconhecimento, os nós-folhas constituem uma forma sentencial, notando-se que alguns símbolos da cadeia podem ainda não estar conectados à árvore.

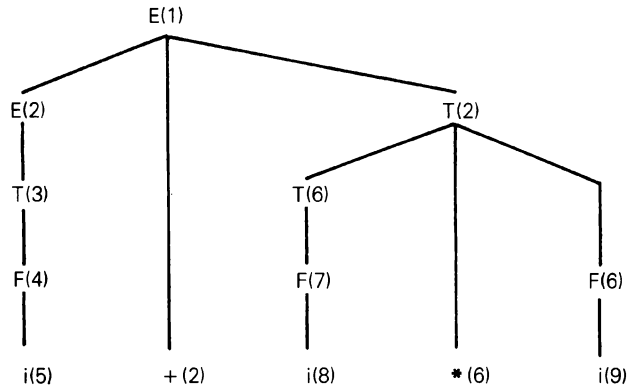


Fig. 5.2

As estratégias descendente e ascendente podem ser representadas esquematicamente como nas figs. 5.3 e 5.4, respectivamente. Na fig. 5.3, temos em (i) o reconhecimento até um certo trecho, e a árvore já foi construída em toda a sua parte tracejada; neste ponto temos como objetivo o não-terminal N, cujo nó deve ser substituído pela subárvore correspondente à produção $N \rightarrow \alpha$. Em (ii), esse reconhecimento já foi efetuado, tendo-se expandido a parte construída da árvore; α pode eventualmente conter símbolos terminais.

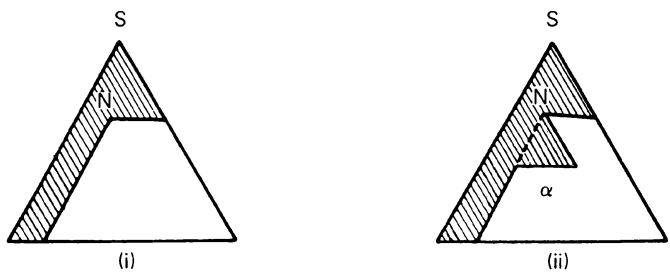


Fig. 5.3

Na fig. 5.4, temos os mesmos passos, para a análise ascendente. Note-se que neste caso não é conhecido o objetivo N. Este deve ser deduzido pelo fato de se encontrar o lado direito α da produção $N \rightarrow \alpha$ na parte "superior" da árvore já construída. Novamente, α pode conter alguns símbolos da cadeia terminal.

Na implementação do compilador que será tratado neste texto, usaremos um método particular de análise sintática descendente. Nos próximos itens introduziremos as restrições às gramáticas que as tornam analisáveis pelo método mencionado.

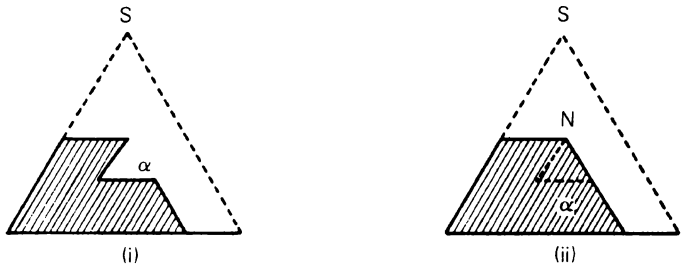


Fig. 5.4

5.4 GRAMÁTICAS LL(k)

A análise descendente canônica pode ser descrita formalmente como se segue:

Seja uma GLC G = (V_N, V_T, S, P) não-ambígua e $t_1t_2...t_it_{i+1}...t_n \epsilon L(G)$. Se a subcadeia $t_1t_2...t_i$ já foi reconhecida como a subcadeia σ da forma sentencial $\xi = \sigma t_{i+1}...t_n$ e sabe-se que $N \epsilon V_N$ é o próximo não-terminal que deve ser reconhecido nessa forma sentencial, é necessário localizar qual produção de N deve ser aplicada. Assim, se $N \rightarrow \alpha_1 \mid \alpha_2 \mid ... \mid \alpha_m$ são as produções de N, é preciso descobrir qual α_j é idêntico a alguma subcadeia de ξ, de modo que se tenha um reconhecimento canônico. Esse α_j é, evidentemente, único, pois G é não-ambígua.

Em 1968, Lewis e Stearns /L-S 68/ definiram a classe de gramáticas *LL(k)* que permitem uma escolha realizável do lado direito adequado. LL(k) é abreviatura de "left-to-right, leftmost derivation with k look-ahead symbols". Para essas gramáticas, os conhecimentos de N, de σ e dos próximos k símbolos da cadeia terminal $t_{i+1}t_{i+2}...t_{i+k}$ determinam univocamente α_j, j = 1,2,...,m. Formalmente, se $\sigma \epsilon (V_N \cup V_T)^*$, $\gamma, \gamma_p, \gamma_q \epsilon V_T^*$,
p,q, = 1,2,...m, e
$S \overset{*}{\Rightarrow} \sigma N\gamma \overset{*}{\Rightarrow} \sigma\alpha_p\gamma = \sigma\gamma_p$
$S \overset{*}{\Rightarrow} \sigma N\gamma \overset{*}{\Rightarrow} \sigma\alpha_q\gamma = \sigma\gamma_q$
então G é LL(k), $k \geq 0$ sse $\alpha_p \neq \alpha_q \Rightarrow \gamma_p^{(k)} \neq \gamma_q^{(k)}$ onde $\gamma^{(k)}$ indica os k símbolos mais à esquerda de γ.

Note-se que nossa definição difere um pouco da de Lewis e Stearns que usaram $\sigma, \gamma \epsilon V_T^*$, o que é sempre possível pois G é GLC.

Como exemplo, seja a gramática G_{14} com as produções:

$S \rightarrow aNbc \mid dNc$
$N \rightarrow cb \mid c$ $\quad (G_{14})$

Esta gramática gera quatro cadeias, cujas análises descendentes estão representadas na fig. 5.5. Usamos nessas árvores sintáticas uma notação especial, para o caso de terminais que são gerados mas ainda não reconhecidos na cadeia. Quando chega o momento de reconhecê-los, este fato é representado por uma conexão tracejada entre os respectivos símbolos; assim terminais ligados por esse tipo de conexão constituem na realidade um só nó da árvore. Para representar os passos rigorosos da análise da esquerda para a direita, a mesma notação deveria ter sido empregada na fig. 5.2.

Comparando-se na fig. 5.5 (i) com (ii), verifica-se que N deve ser reconhecido como 'c' ou 'cb', baseado em c(5) no caso (i) e em b(4) no caso (ii). Assim é necessário nesse caso examinarem-se os três próximos símbolos de entrada, cbc e cbb respectivamente. Nunca é necessário examinarem-se quatro símbolos, de modo que G_{14} é LL(3). Por outro lado, G_{14} é um exemplo interessante de um caso em que é necessário usar a subcadeia σ da definição vista acima. De fato em (i) e (iii), tendo-se N como objetivo, deve-se decidir entre usar $N \rightarrow cb$ ou

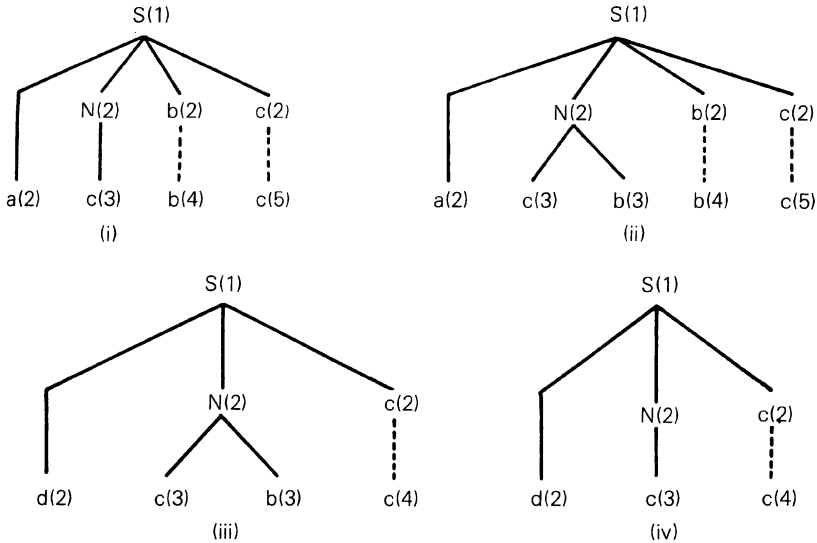


Fig. 5.5

N→c. A parte da cadeia ainda não reconhecida é, em ambos os casos, 'cbc'. A decisão de qual produção empregar é feita com base no símbolo já reconhecido, isto é, 'a' em (i) e 'd' em (iii).

Como contra-exemplo, vejamos a gramática G_{15}:

$S \rightarrow Sa \mid b \qquad (G_{15})$

Essa gramática não é LL(k), qualquer que seja k. De fato, suponhamos que se tome k=1. A cadeia 'ba' não pode ser analisada, pois se k=1 só podemos tomar um símbolo à frente para decidir qual das duas produções deve ser aplicada. Ora, no início da análise teremos apenas a informação de que o primeiro símbolo é 'b', não podendo tomar conhecimento de que o segundo é 'a', o que exigiria a aplicação de S→Sa e não de S→b. Assim, G_{15} não é LL(1). Para um certo k >1, se a cadeia de entrada for ba...a com k+1 símbolos, podemos inicialmente decidir que S→Sa deve ser aplicado; só que depois desta aplicação não reconhecemos ainda nenhum símbolo da cadeia de entrada, pois não sabemos qual é o último 'a', que deve corresponder ao terminal gerado por essa produção, já que temos só k símbolos à disposição.

Nenhuma GLC que contenha uma recursão à esquerda como G_{15} ou, em geral, nenhuma GLC para a qual um dos seus não-terminais N seja tal que $N \overset{+}{\rightarrow} N\alpha$, com qualquer α, é LL(k). No entanto qualquer gramática com recursão à esquerda pode ser transformada em uma gramática equivalente (v. 4.3), sem recursão à esquerda. Por exemplo, G'_{15} é obviamente equivalente a G_{15}:

$S \rightarrow bM$
$M \rightarrow aM \mid \lambda \qquad (G'_{15})$

Nesse caso substituímos G_{15} por uma gramática com recursão à direita. Podemos eliminar totalmente a recursão usando uma ERE-gramática (v. 4.7).

$S \rightarrow ba^* \qquad (G''_{15})$

A gramática G''_{15} é do tipo LL(k). De fato ela é LL(1). Ela não é LL(0) pois, após a leitura de 'b', é necessário conhecer-se o próximo símbolo para decidir se a cadeia de 'a's já terminou.

A transformação da gramática G_{10} na equivalente G_{13} (v. 4.7) teve, em parte, a intenção de obter outra gramática equivalente, agora do tipo LL(k), já que G_{10} não é desse tipo. G_{13} é do tipo LL(1); note-se que é necessário considerar, para isso, todas as construções que envolvem alternativas e fechamento de concatenação.

Em 1970, Rosenkrantz e Stearns/R-S 70/ definiram a classe de gramáticas *Strong LL(k)*, ou LL(k)-fortes, para as quais em um ponto qualquer da análise canônica, se N está sendo procurado, α_j é determinado exclusivamente pelos k próximos símbolos ainda não reconhecidos, independentemente de σ, isto é, independentemente do histórico passado do reconhecimento (compare-se com a definição de gramáticas LL(k)). Formalmente, se β', β'' $\epsilon(V_N \cup V_T)^*$, σ', σ'', γ, γ_p, $\gamma_q \epsilon V_T^*$
p,q = 1,2,...,m, e
$S \overset{*}{\Rightarrow} \beta' N\gamma \Rightarrow \beta'\alpha_p\gamma \overset{*}{\Rightarrow} \sigma'\gamma_p$
$S \overset{*}{\Rightarrow} \beta'' N\gamma \Rightarrow \beta''\alpha_q\gamma \overset{*}{\Rightarrow} \sigma''\gamma_q$
então G é LL(k)-forte se $\alpha_p \neq \alpha_q \Rightarrow \gamma_p^{(k)} \neq \gamma_q^{(k)}$.

A gramática G_{14} é apenas LL(3), não sendo LL(3)-forte, como mostramos nos casos (i) e (iii) da fig. 5.5. De fato, nesses casos N deve ser reconhecido como 'c' ou 'cb'. A decisão sobre um desses lados direitos de G_{14} não pode ser baseada nos símbolos que ainda não foram reconhecidos, pois estes formam duas cadeias idênticas ('cbc'). A decisão entre essas duas produções pode, no entanto, ser baseada no símbolo já reconhecido 'a' ou 'd'. Assim, é necessário usar o passado da análise, no caso, diretamente os símbolos já reconhecidos 'a' ou 'd', para poder continuá-la. Veremos mais adiante como as gramáticas LL(k)-fortes simplificam o processo de análise sintática. Todas as gramáticas LL(1) são fortes e não-ambíguas.

Para maiores detalhes sobre gramáticas LL(k), veja o livro de Aho e Ullman/A-U 72/.

Em 1973, Rechenberg /REC 73/ definiu as gramáticas PLL(k), que são LL(k)-fortes, estendendo-se a definição destas para gramáticas com lados direitos na forma:
N → E ou
N → E{E}*
onde E contém alternativas e/ou alternativas fatoradas. Uma definição precisa pode ser encontrada em /REC 73/. A gramática G_{16} é PLL(1):

$$\begin{aligned} &S \rightarrow M \mid N \\ &M \rightarrow a\{b \mid Nc\}^* \\ &N \rightarrow d(S \mid ef \mid \lambda) \mid g \end{aligned} \qquad (G_{16})$$

As gramáticas PLL(1) podem ser analisadas com certa eficiência. Para distinguir-se entre várias alternativas de um lado direito, é necessário em geral deduzirem-se os conjuntos de símbolos terminais que podem ser gerados mais à esquerda a partir de cada não-terminal. No caso de G_{16}, para distinguir-se entre S→M ou S→N é preciso comparar se o próximo símbolo de entrada pertence ao conjunto {'a'} ou ao conjunto {'d', 'g'}, que contém os símbolos gerados mais à esquerda por M e N, respectivamente. Em alguns casos é preciso também deduzir os símbolos que podem seguir um determinado não-terminal em qualquer forma sentencial.

No próximo item definiremos um subconjunto das ERE-gramáticas que são extensões das gramáticas LL(1). Os lados direitos das produções serão mais gerais do que os da gramática PLL(k); no entanto certas restrições serão impostas para permitir uma análise mais eficiente do que a das gramáticas PLL(1). Elas constituirão as gramáticas aceitas pelo algoritmo de análise sintática a ser usado neste texto.

5.5 GRAMÁTICAS ESLL(1)

Neste item definiremos as gramáticas, que denominaremos de ESLL(1), de "*extended simple* LL(1)" que formam a classe de ERE-gramáticas (v. 4.7) aceitas pelo analisador sintático a ser descrito neste capítulo. Essa definição será dada em termos de grafos sintáticos (v.

4.8) e para isso deveremos introduzir algumas noções adicionais que se aplicam a esses grafos. Note-se que estes devem ser originados de ERE-gramáticas e devem seguir estritamente as regras de construção da fig. 4.8.

Definição: Dado um nó m de um grafo sintático, dizemos que n é um *nó alternativo* (ou simplesmente é *alternativa*) de m se, e somente se, o arco orientado (flecha) que chega em m tem uma bifurcação *para baixo* antes de m, e n é o nó apontado por essa bifurcação. Se essa bifurcação apontar para mais do que um nó, o nó alternativo de m é o que é apontado pela bifurcação que está imediatamente abaixo dele.

Na fig. 5.6 ilustramos essa situação com uma parte genérica de algum grafo. Os nós m e n podem ser terminais.

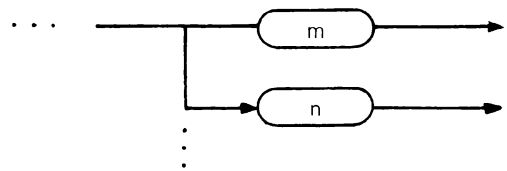


fig. 5.6

Por exemplo, para o trecho de grafo da fig. 5.7 (i) temos: o nó com 'b' é o nó alternativo do nó com 'a' e tem como alternativa o nó com N. O nó com N não possui alternativa, e o nó com 'a' não é alternativa de nenhum nó.

Na fig. 5.7 (ii) temos o grafo correspondente à ERE ... [a] b ... Neste caso a alternativa de 'a' é 'b'.

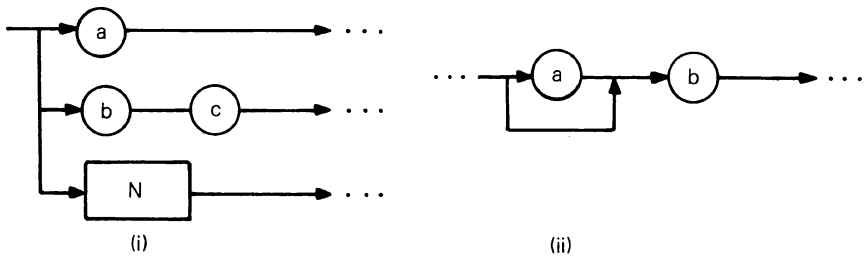


Fig. 5.7

Na fig. 4.10, para o subgrafo do não-terminal T, temos '/' como alternativa de '*'. O nó com '/' tem uma alternativa especial, constituída por um arco que aponta para uma barra vertical. Chamaremos essas alternativas "vazias" de λ-*alternativas*. Note-se, nessa figura, o subgrafo de P não tem λ-alternativa.

Na fig. 5.8 (i) temos o grafo correspondente à gramática S → a|λ e em (ii) o correspondente à gramática S→ [a]. No primeiro caso, temos explicitamente um λ-*nó*, isto é, um nó terminal contendo λ; a alternativa de 'a' é esse λ-nó. No segundo caso, 'a' tem uma λ-alternativa. Veremos mais tarde (v. 5.6.2) que a implementação dessas duas gramáticas em estruturas de dados será idêntica; de fato, para cada λ-alternativa será gerado implicitamente um λ-nó.

Nada impede que se use o símbolo λ em qualquer lugar das produções, como por exemplo N → a λ b. Nesse caso coloca-se um λ-nó como sucessor de 'a', tendo como sucessor o nó 'b', conforme a definição de sucessor dada logo adiante.

(i) (ii)

Fig. 5.8

Se os nós $n_1, n_2, \ldots, n_m$, são tais que se $m > 1$, n_i é alternativa de n_{i-1}, com $i = 2,3,\ldots,m$, diremos que esses nós constituem uma *seqüência de nós alternativos* (ou simplesmente *seqüência de alternativas*). Note-se que qualquer nó da seqüência pode ter uma λ-alternativa que evidentemente não é considerada como um nó da seqüência.

Definição: Dado um nó m de um grafo sintático, dizemos que n é um *nó sucessor* (ou simplesmente *sucessor*) de m se, e somente se, o arco que sai de m aponta diretamente para n; se esse arco leva a uma seqüência de alternativas, o nó sucessor de m é o primeiro nó da seqüência.

Por exemplo, na fig. 5.7 (i) o nó 'c' é sucessor do nó 'b'. Na fig. 5.7 (ii), 'b' é sucessor de 'a' (e também sua alternativa). Na fig. 4.10, para o subgrafo de T, '*' é sucessor de F e (atenção!) F é sucessor de '*' e '/'.

Com essas noções, podemos agora definir as gramáticas do tipo ESLL(1).

Definição: Uma ERE-gramática $G = (V_N, V_T, S, P)$ é uma gramática do tipo ESLL(1) se o grafo g de G construído a partir de P segundo as regras da fig. 4.8 satisfaz as seguintes propriedades:

1) Para cada nó n_1 de g que pertence a uma seqüência de alternativas $n_1, n_2, \ldots, n_m$, $m > 1$, não existe n_i, $i = 2,3,\ldots,m$, tal que n_i é o próprio n_1.
2) Para cada não-terminal $N \epsilon V_N$, $FIRST(N) \neq \phi$ onde $FIRST(N) = \{t \epsilon V_T \mid N \overset{*}{\rightarrow} t\alpha, N \epsilon V_N, \alpha \epsilon (V_N \cup V_T)^*\}$.
3) Se n for um nó isolado do grafo, isto é, n não pertence a uma seqüência de alternativas e n tem uma λ − alternativa, então n deve conter $t \epsilon V_T$, $t \neq \lambda$.
4) Para cada não-terminal $N \epsilon V_N$, cada uma das seqüências de alternativas $n_1, n_2, \ldots, n_m$, $m > 1$ do subgrafo de N deve satisfazer às seguintes condições, onde $i,j = 1,2,\ldots,m-1$:

4.1) n_i deve ser nó terminal, isto é, n_i contém $t_i \epsilon V_T$, $t_i \neq \lambda$ e $t_i \neq t_j$ para $i \neq j$;

4.2) n_i não tem uma λ-alternativa;

Seja $T = \{t_1, t_2, \ldots, t_{m-1}\}$:

4.3) se n_m não tem uma λ-alternativa e

4.3.1) se n_m contém $t_m \epsilon V_T$ e $t_m \neq \lambda$ então $t_m \notin T$

4.3.2) se n_m contém $M \epsilon V_N$ e não existir geração $M \overset{*}{\rightarrow} \lambda$ então $T \cap FIRST(M) = \phi$;

4.3.3) se n_m contém λ e n_m não tem sucessor então $T \cap FOLLOW(N) = \phi$ onde $FOLLOW(x) = \{t \epsilon V_T \mid S \overset{*}{\rightarrow} \alpha x t \beta,\ x \epsilon (V_N \cup V_T),\ t \neq \lambda,\ \alpha, \beta \epsilon (V_N \cup V_T)^*\}$;

4.3.4) se n_m contém $M \epsilon V_N$ e $M \overset{*}{\rightarrow} \lambda$ então $T \cap FIRST(M)\ \ T \cap FIRST(M) = \phi$;

4.3.4.1) se não existe sucessor de n_m então $T \cup FOLLOW(N) = \phi$

4.3.4.2) se existe um sucessor n_{m+1} de n_m, seja $n_{m+1}, n_{m+2}, \ldots, n_{m+k}$ uma seqüência de alternativas. A seqüência $n_1, n_2, \ldots, n_{m-1}, n_{m+1}, \ldots, n_{m+k}$ deve satisfazer às condições 4.

4.4) se n_m tem uma λ-alternativa então n_m contém $t_m \epsilon V_T, t_m \neq \lambda$, $t_m \notin T$ e $(T \cup \{t_m\}) \cap FOLLOW(N) = \phi$;

4.5) se n_m contém λ e n_m tem um nó sucessor n_{m+1} com alternativas $n_{m+1}, \ldots, n_{m+k}$ então a seqüência de alternativas $n_1, n_2, \ldots, n_{m-1}, n_{m+1}, n_{m+2}, \ldots, n_{m+k}$ deve satisfazer às condições 4.

No próximo item descreveremos o AS para as ESLL(1)-gramáticas. Com isso, a definição acima ficará clara, podendo-se compreender melhor a necessidade das restrições descritas.

Usamos a denominação "ESLL(1)" devido ao fato dessas gramáticas serem extensões das gramáticas denominadas "simple-LL(1)", em que não se tem expressões regulares nos lados direitos das produções. Além disso, nas gramáticas "simple-LL(1)", para cada não-terminal N, as produções de suas alternativas devem todas começar mais à esquerda com um terminal distinto; nenhuma alternativa pode constituir-se exclusivamente de λ /A-U 72/.

No apêndice I apresentamos uma gramática-ESLL(1) e o correspondente grafo sintático, para a linguagem PASCAL completa /J-W 74/. Note-se que o grafo sintático da linguagem PASCAL introduzido por Wirth /J-W 74/ não se conforma às restrições dadas acima. Por exemplo, nas figs. 5.9 e 5.10 temos transcrições exatas de trechos daquele grafo dentro dessas condições. No primeiro caso, identifier é sucessor de ';', e **type** deveria ser alternativa de identifier; no entanto, neste caso teríamos a seqüência ...**const type**... que não é permitida. Não há uma ERE que exprima esse trecho de grafo; é necessário repetir identifier para resolver o problema, conforme o apêndice I. No segundo caso, "constant" é um não-terminal e, no entanto, tem uma alternativa, não obedecendo a algumas das restrições vistas. Nossa solução no apêndice I foi a de substituir as produções de "constant" em lugar deste não-terminal.

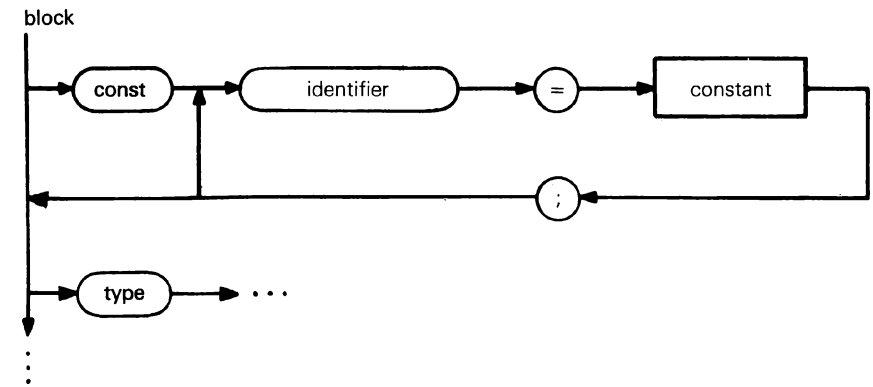


fig. 5.9

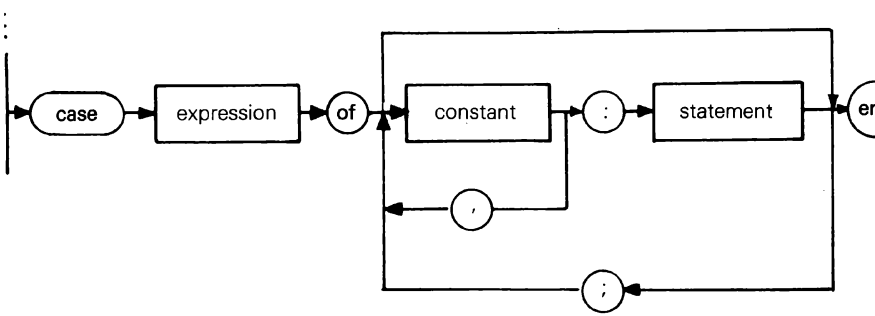


fig. 5.10

Gramáticas que não são do tipo ESLL(1) podem ser eventualmente transformadas em outras equivalentes, desse tipo. A gramática G_{10} do item 4.6 não é do tipo ESLL(1); em 4.7 fize-

mos transformações sistemáticas dessa gramática até obter a gramática G_{13} equivalente, que é do tipo ESLL(1).

Um contra-exemplo para a condição 1 da definição dada é a gramática S → {a*}*b. Do mesmo modo S→{a*b*}*c tere essa condição. Por outro lado a equivalente a esta última (cf. 4.3) S→{a|b}*c é do tipo ESLL(1).

Note-se que as condições 2, 4.1 e 4.3.2 excluem gramáticas com recursividade à esquerda.

5.6 O PROCEDIMENTO DO ANALISADOR SINTÁTICO

Neste item faremos uma introdução ao AS que será empregado no compilador descrito neste texto, mostrando intuitivamente o seu funcionamento através de exemplos. Entretanto não se trata de um analisador particular, podendo ser utilizado em uma gama muito grande de linguagens. A seguir daremos uma representação do grafo como estruturas de dados e, finalmente, descreveremos um procedimento em PASCAL para o AS. É interessante acompanhar-se a explicação consultando a fig. 2.6, para que seja localizado cada novo elemento sendo introduzido.

5.6.1 EXEMPLO DO FUNCIONAMENTO

Tomemos como exemplo a gramática G_{17} com as seguintes produções:

S → a(b|Sc) | dM | e $\quad (G_{17})$

M → {fS}*

O grafo de G_{17} é apresentado na fig. 5.11, tendo-se numerado os nós para que se possa fazer referência aos mesmos posteriormente. Cada alternativa vazia recebe também um número.

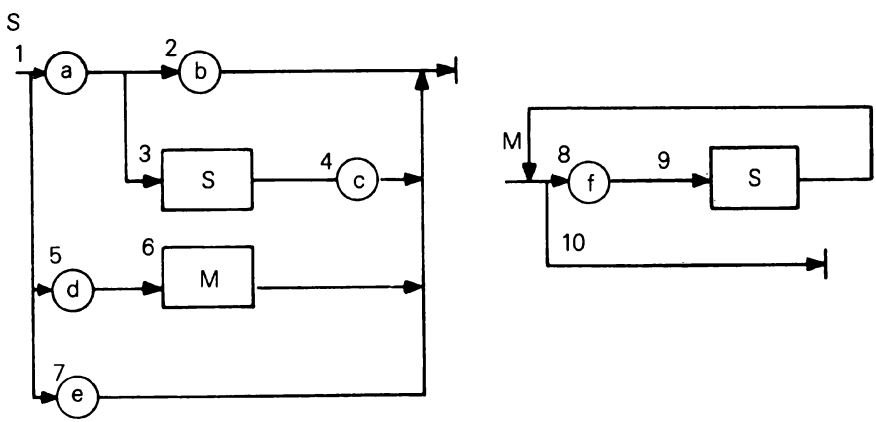


fig. 5.11

Seja a cadeia de entrada 'aabc' gerada por G_{17}. O AS será do tipo "descendente" (v. 5.3) e o reconhecimento canônico será o seguinte: aabc ← aSc ← S. Inicialmente, o objetivo é encontrar o símbolo inicial S. Suponhamos que o AL, sempre que chamado, leia um símbolo da cadeia, colocando-o em ent, e que o primeiro símbolo 'a' já tenha sido lido.

Como o objetivo atual é S, passamos a examinar o subgrafo de S. Para isso, começamos no seu primeiro nó, 1 (isto é, o nó mais acima e mais à esquerda). Como o nó 1 é um nó

terminal, comparamos seu conteúdo com o conteúdo de ent. Se estes fossem diferentes, passaríamos à alternativa (nó 5) do nó 1. No caso eles são iguais, e podemos dizer que o primeiro símbolo ('a') da cadeia de entrada e o nó 1 foram *reconhecidos*. Devido a isso, chamamos o AL que lê o próximo símbolo ('a') e o coloca em ent, e passamos ao *sucessor* do nó reconhecido, ou seja, passamos ao nó 2. O nó 2 é um nó terminal, mas o seu conteúdo é diferente do conteúdo de ent ('b' ≠ ent). Neste caso, tomamos como próximo nó a alternativa do nó 2, qual seja o nó 3. Note-se que o segundo 'a' da cadeia ainda não foi reconhecido, e portanto o AL não é chamado. O nó 3 corresponde ao não-terminal S. Nesse caso interrompemos o fluxo da análise, guardando o número desse nó em K para podermos retornar a ele depois de termos reconhecido um S na cadeia. Nosso novo objetivo é S (a identidade com o objetivo inicial é mera coincidência), portanto passamos a examinar o subgrafo de S a partir de seu primeiro nó (nó 1). Tudo se passa da mesma maneira, até se chegar ao nó 2, com 'b' lido em ent. Agora, o conteúdo do nó 2 é igual ao conteúdo de ent, de modo que reconhecemos 'b' e o nó 2. Chamamos o AL, que coloca o próximo símbolo 'c' em ent, e passamos ao sucessor do nó 2. Esse sucessor não existe. Quando um nó (terminal ou não-terminal) é reconhecido, e não há sucessor para o mesmo, atingiu-se o fim do lado direito de uma produção, ou seja, o último objetivo foi alcançado, isto é, o não-terminal correspondente foi reconhecido. Neste caso devemos retornar ao nó onde o fluxo da análise foi interrompido, isto é, em nosso caso, ao nó cujo número foi guardado em K, ou seja, 3; limpamos K, que passa a não conter nenhum número de nó. Na verdade, esse nó 3 (ou um S) acabou de ser reconhecido; como se trata de um não-terminal, o AL não é chamado. Passamos ao seu sucessor, que é o nó 4. Este é reconhecido, pois ent = 'c'; chamamos o AL, que encontra um fim-de-arquivo, pois a cadeia acabou, colocando '$' em ent (v. 3.4). A seguir, passamos ao sucessor do nó 4. Esse sucessor não existe, o que significa o fim de uma produção. Consultamos K que, não contendo agora nenhum número de nó, indica que todas as interrupções da análise ocorridas anteriormente, como a acima com S, já foram tratadas, não havendo nenhuma pendente; em outras palavras, não há nenhum objetivo pendente além do objetivo inicial. Nesse ponto, o AS verifica se o AL encontrou um fim-de-arquivo, isto é, se ent = '$'. Neste caso, acusa o fim da análise com sucesso, ou seja, a cadeia foi reconhecida.

Examinemos agora o caso da cadeia de entrada 'dfaec'. Seguindo-se passos análogos aos do exemplo anterior, o símbolo 'd' será reconhecido no nó 5, interrompendo-se a análise no nó 6, a fim de procurar-se um M; colocamos 6 em K. Passando ao subgrafo de M, 'f' é reconhecido no nó 8, e devemos interromper novamente a análise no nó 9; se colocássemos simplesmente 9 em K, "apagaríamos" o seu conteúdo anterior 6, que não pode ser perdido. A solução para esse problema é considerar-se K como uma estrutura de pilha, empilhando-se sempre o número do nó não-terminal onde ocorreu a interrupção e foi estabelecido novo objetivo. Implementando essa pilha como um vetor, teremos K[1] = 6, K[2] = 9. Continuando, 'a' é reconhecido em 1; o teste de 'b' no nó 2 falha (pois ent = 'e'), tomando-se a alternativa 3. Empilha-se este número fazendo K[3]: = 3 e desviando-se para o grafo de S. Os testes com 'a' e 'd' falham; o teste com 'e' no nó 7 tem sucesso; como se trata de nó sem sucessor, 7 indica o fim de uma produção de S, que acaba portanto de ser reconhecido. Neste momento, é preciso voltar para o nó onde houve a última interrupção, no caso o nó apontado pelo topo K[3] da pilha, isto é, o nó 3. Retiramos o topo da pilha, que passa a ser K[2]. Voltamos ao nó 3, que acabou de ser reconhecido (isto é, foi reconhecido um S), e passamos ao seu sucessor, o nó 4. O símbolo 'c' é reconhecido, chegando-se mais uma vez ao fim de uma produção. Desempilhamos o topo da pilha K[2], obtendo o número do nó ao qual devemos voltar, que é o 9; o topo da pilha passa a ser K[1]; o nó 9 indica que reconhecemos um S; tomamos seu sucessor, que é o 8. O conteúdo deste não é reconhecido (pois temos nestas alturas ent = '$'); passamos à alternativa do nó 8, que é uma λ-alternativa, indicada por 10. Esta nos mostra um fim de produção. Aqui, tudo se passa como se um nó terminal de número 10 contendo λ estivesse no lugar da λ-alternativa. Esse nó é sempre reconhecido independentemente do conteúdo de ent, não provocando nova chamada do AL. Ele tem como sucessor um arco para o vazio, co-

mo a λ-alternativa. Podemos reescrever o grafo de M explicitando esse nó λ, como na fig. 5.12.

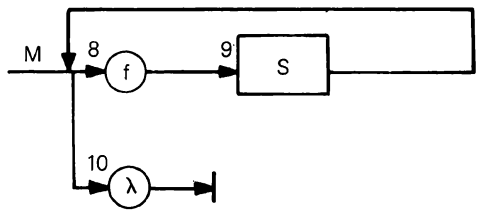


Fig. 5.12

Como chegamos ao final de uma produção, desempilhamos novamente o número do nó para onde devemos retornar, no caso o nó 6,cujo número estava em K[1]; a pilha passa a ficar vazia. Com isso reconhecemos um M; passando ao sucessor de 6, chegamos ao fim de uma produção. Como a pilha está vazia e ent = '$', a análise termina com sucesso. Note-se o reconhecimento canônico efetuado:
dfaec ≐ dfaSc ≐ dfS ≐ dM ≐ S

Note-se que em ambos os exemplos o reconhecimento canônico mostra a substituição de partes das formas sentenciais por não-terminais. O AS como descrito acima apenas reconhece os símbolos terminais, não construindo as formas sentenciais. No item 5.7 veremos como o AS pode ser usado para construir essas formas, fazendo realmente uma análise sintática. Da maneira como o intruduzimos, o AS pode ser considerado como um simples reconhecedor.

Observe-se ainda que o funcionamento do AS como descrito corresponde a uma *interpretação*, isto é, um percurso do grafo sintático para cada cadeia. Trata-se em síntese do mesmo processo pelo qual uma cadeia é reconhecida por um autômato finito. De fato, existe uma analogia entre um AF e o subgrafo de um não-terminal: pode-se considerar cada nó terminal ou não-terminal como uma transição ligando estados, sob a leitura do conteúdo do nó. Cada nó do subgrafo dá origem a um único estado correspondente ao arco que sai desse nó, antes de bifurcações, que por sua vez não dão origem a novos estados. É necessário considerar um estado inicial correspondendo ao arco de entrada no subgrafo; cada arco do subgrafo que se dirige para o vazio dá origem a um estado final do AF. Na fig. 5.13 mostramos os AFs correspondentes ao grafo da fig. 5.11.

Note-se que na fig. 5.13 temos algumas transições com símbolos terminais e outras com símbolos não-terminais. Uma transição $g(s_i,N) = s_j$ de um AF P (v. 3.2) onde N é um não-terminal, deve ser processada interrompendo-se o funcionamento de P e iniciando-se o processamento do AF de N a partir do primeiro símbolo da cadeia de entrada ainda não reconhecido por P. Ao atingir-se o estado final do AF de N, retorna-se para s do AF de P. Uma pilha pode conter os nomes dos estados para onde deve-se retornar.

Podemos agora dar uma justificativa para as restrições dadas às gramáticas ESLL(1) em 5.5. Considerando-se uma ordenação das transições dos AFs correspondente à ordenação dos nós, assumindo-se que essas transições são testadas na ordem correspondente aos nós das seqüências de alternativas (isto é, de cima para baixo), as citadas restrições implicam em que o autômato tenha um "funcionamento determinístico". Não podemos denominar o AF de determinístico, pois nem ao menos temos, a rigor, um AF, já que em algumas transições há um AF em lugar de um símbolo de entrada, podendo haver recursões. De fato, na fig. 5.13, se substituíssemos o AF M na transição de s_2 para s_7, fazendo $s'_0 \equiv s_2$ e $s'_1 \equiv s_7$, o AF S teria uma transição que utiliza o próprio S.

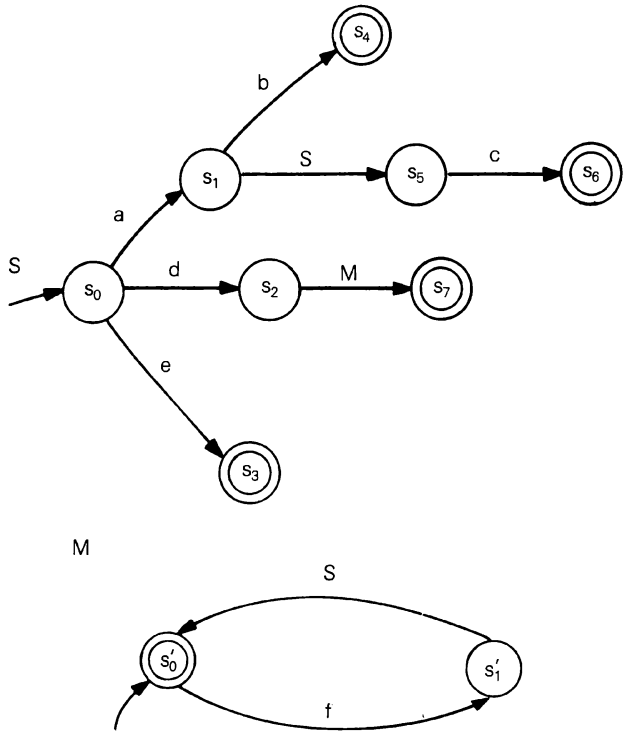


Fig. 5.13

5.6.2 ESTRUTURA DE DADOS DO GRAFO SINTÁTICO

Para programarmos o AS cujo funcionamento foi exemplificado em 5.6.1, devemos armazenar o grafo sintático na "memória" de um computador sob a forma de uma tabela, que denominaremos de TABGRAFO. Para isso, é necessário introduzir estruturas de dados que possam representar os grafos, com seus nós e arcos.

Cada nó do grafo será representado por um elemento de TABGRAFO que segue a estrutura de dados representada graficamente na fig. 5.14.

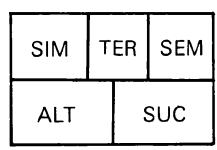


Fig. 5.14

Os símbolos terminais e não-terminais da gramática são guardados em duas tabelas, que denominaremos de TABT e TABNT, respectivamente, cujas primeiras entradas têm índi-

ce 1 (um). A TABNT é uma tabela com dois campos (NOME e PRIM) por entrada: NOME contém uma cadeia de até 6 caracteres com o nome do não-terminal, ajustado à esquerda; PRIM contém um índice de uma entrada para a TABGRAFO e aponta para o elemento desta correspondente ao primeiro nó do grafo das produções desse não-terminal. A TABT é a própria tabela de símbolos reservados, descrita em 2.3 e cuja construção descrevemos em 2.4.

Na fig. 5.14, o campo SIM de cada elemento n_e da TABGRAFO contém um índice para a TABT ou para TABNT. Se o nó n_g correspondente do grafo contiver um símbolo terminal, o campo TER contém o valor true e SIM contém o índice da entrada da TABT onde está armazenado o terminal correspondente. Se o nó n_g contiver um não-terminal, o campo TER de n_e contém o valor false e SIM contém o índice da entrada da TABNT onde está armazenado o não-terminal correspondente. O campo ALT contém o índice para a TABGRAFO onde se encontra o elemento correspondente à alternativa do nó n_g; se n_g não tem alternativa, ALT deve conter o número 0 (zero). Analogamente, SUC contém o índice do elemento da TABGRAFO correspondente ao sucessor de n_g. O campo SEM contém um número inteiro, indicando uma rotina "semântica" que deverá ser executada após o reconhecimento do nó, como veremos no item 6.2.

Se o nó n_g for um λ-nó, é gerado um elemento da TABGRAFO com TER = true, SIM = 0. O mesmo se passa para cada λ-alternativa, a qual deve gerar um elemento da TABGRAFO com os campos adicionais ALT = 0 e SUC = 0, já que pela definição de ESLL(1)-gramáticas, cada λ-alternativa não tem alternativa e aponta para o vazio, isto é, não tem sucessor.

Na fig. 5.15, apresentamos um diagrama da estrutura de dados correspondente ao grafo da fig. 5.11. Nesse diagrama simplificamos os elementos como introduzidos na fig. 5.14; se o nó correspondente é terminal, o próprio símbolo terminal é colocado no elemento; se é nó não-terminal, colocamos um apontador para um elemento da TABNT, que é representado por uma estrutura à parte. Omitimos, neste exemplo, o campo SEM. Desta maneira, recaímos nos diagramas usados por Wirth em /WIR 76/.

Na fig. 5.16 encontram-se as tabelas correspondentes ao grafo da fig. 5.11 e ao diagrama da fig. 5.15. Note-se que não é necessário construir-se esse diagrama como passo intermediário entre o grafo e a tabela, já que os grafos construídos a partir das ERE-gramáticas, segundo as regras dadas em 4.8, contêm diretamente todas as informações para a tabela, o que, como vimos, não acontece com o grafo sintático da PASCAL de /J-W 74/.

5.6.3 O CARREGADOR SINTÁTICO

Neste item descrevemos um procedimento que lê registros onde se dão informações sobre o grafo sintático, e que produz as três tabelas do item anterior. Esses registros não contêm a numeração absoluta dos nós, como exemplificado na fig. 5.11, mas sim um número relativo ao primeiro nó de cada subgrafo de um não-terminal, isto é, em cada um desses subgrafos iniciamos novamente a numeração dos nós a partir de 1 (um). Esses números relativos são usados também em ALT e SUC. Dessa maneira consegue-se uma grande flexibilidade na codificação do grafo. O carregador sintático deve transformar esses números relativos em absolutos. Cada subgrafo de um não-terminal é precedido de um registro especial, contendo o código 'C' (de "cabeça"), e que é usado para se introduzir o valor de PRIM em TABNT, já que, ao ser lido, o carregador pode deduzir a numeração absoluta de seu primeiro nó. Os nós dos grafos são introduzidos por registros contendo o código 'N' para não-terminais e 'T' para terminais. Na fig. 5.17 apresentamos os registros de entrada para o carregador, correspondentes ao grafo da fig. 5.11. O campo NUMNO contém o número relativo de cada nó. A numeração relativa dos nós 8, 9 e 10 do subgrafo de M é 1, 2 e 3, respectivamente. Novamente, deixamos de colocar os números das rotinas "semânticas". O λ-nó é representado por um registro cujo campo NOMER contém valor branco.

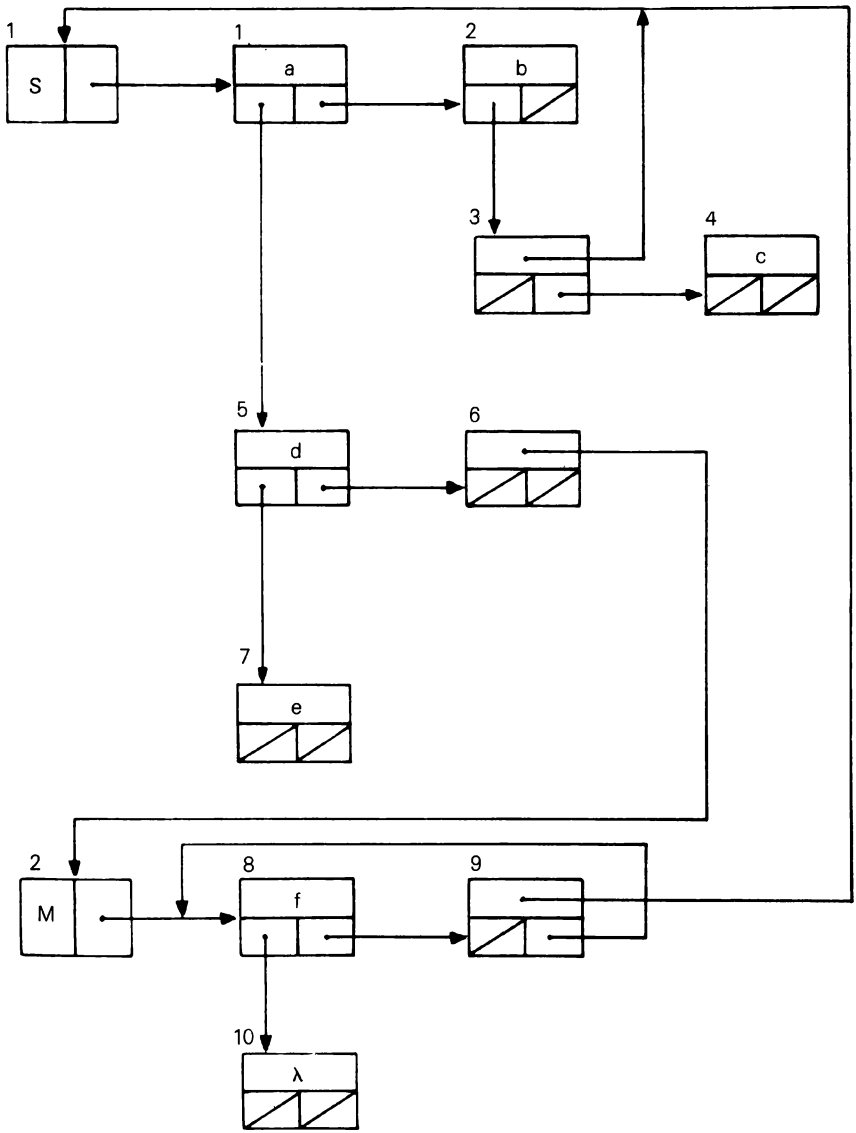


Fig. 5.15

Damos a seguir, em notação informal, uma descrição do carregador. Os significados de algumas variáveis são os seguintes: MAXT — índice do último elemento da TABT; MAXNT idem para a TABNT; INDPRIM — índice do elemento correspondente ao primeiro nó de um

TABGRAFO

	TER	SIM	ALT	SUC	SEM
1	true	1	5	2	
2	true	2	3	0	
3	false	1	0	4	
4	true	3	0	0	
5	true	4	7	6	
6	false	2	0	0	
7	true	5	0	0	
8	true	6	10	9	
9	false	1	0	8	
10	true	0	0	0	

TABT

1	a
2	b
3	c
4	d
5	e
6	f

TABNT

	NOME	PRIM
1	S	1
2	M	8

Fig. 5.16

TIPO	NOMER	NUMNO	ALTR	SUCR	SEMR
C	S				
T	A	1	5	2	
T	B	2	3	0	
N	S	3	0	4	
T	C	4	0	0	
T	D	5	7	6	
N	M	6	0	0	
T	E	7	0	0	
C	M				
T	F	1	3	2	
N	S	2	0	1	
T		3	0	0	

Fig. 5.17

subgrafo para um certo não-terminal; NOMAX — número do nó de maior número relativo de um subgrafo. Note-se que supomos não haver nenhum registro que não seja do tipo 'C', 'T' ou 'N'. Como se pode observar, a ordem dos registros que se seguem a um registro do tipo cabeça não é relevante. Indicaremos comandos compostos por ⌜,⌟

É interessante observar que a TABT, contendo os símbolos terminais, é na verdade a própria tabela de símbolos reservados (v. 2.3 e 2.4). Vemos assim que o carregador deve substituir a rotina programada para carregar essa tabela e descrita em 2.4. Para simplificar a descrição, usaremos no carregador uma construção linear de TABT, e não um método de "Hashing" como proposto em 2.4.

```
MAXT:=0; MAXNT:=0; INDPRIM:=1; NOMAX:=0;
enquanto houverem registros para serem lidos
    ⌜leia um registro com TIPO, NOMER, NUMNO, ALTR, SUCR, SEMR;
     se TIPO='C'
        então ⌜INDPRIM:=INDPRIM+NOMAX; NOMAX:=0;
              se NOMER não se encontra na TABNT
                 então ⌜MAXNT:=MAXNT+1;
                        TABNT[MAXNT].NOME:=NOMER;
                        TABNT[MAXNT].PRIM:=INDPRIM⌟;
```

```
        se existe E tal que 1≤E≤ MAXNT e TABNT[E]. NOME=NOMER
           então se TABNT[E]. PRIM=0
                    então TABNT[E].PRIM:= INDPRIM
                    senão ERRO (dois cabeças para um mesmo não-terminal) ];
senão[I:=INDPRIM+NUMO-1;
        se TIPO='T' e NOMER≠'ø' (é terminal, diferente de λ-nó)
           então [se NOMER não se encontra na TABT
                    então [MAXT:= MAXT+1;
                           TABT[MAXT]:= NOMER ];
                  seja E tal que 1≤E≤ MAXT e TABT[E]=NOMER
                    faça TABGRAFO[I].TER:=true ] ;
        se TIPO='N'
           então [se NOMER não se encontra na TABNT
                    então [MAXNT:=MAXNT+1;
                           TABNT[MAXNT].NOME:= NOMER;
                           TABNT[MAXNT].PRIM:= 0 ];
                  seja E tal que 1≤E≤ MAXNT e TABNT[E].NOME=NOMER
                    faça TABGRAFO[I].TER:= false ];
        se NOMER='ø' (λ-nó)
           então TABGRAFO[I].SIM:= 0
           senão TABGRAFO[I].SIM:= E;
        se ALTR≠0
           então TABGRAFO[I].ALT:= INDPRIM+ALTR-1
           senão TABGRAFO[I].ALT:=0;
        se SUCR≠0
           então TABGRAFO[I].SUC:= INDPRIM+SUCR-1
           senão TABGRAFO[I].SUC:= 0;
        TABGRAFO[I]. SEM:= SEMR;
        se NOMAX < NUMNO então NOMAX:= NUMNO ]
```

5.6.4 O PROCEDIMENTO ANSIN

Neste item damos, em PASCAL, o procedimento do AS /SET 79/, que terá o nome ANSIN. Comentários inseridos entre os comandos devem tornar compreensível o funcionamento do procedimento, principalmente seguindo-se as indicações dadas em 5.6.1. Algumas observações breves sobre esse procedimento:

– As constantes MAXG, MAXNT e MAXT, que indicam o índice máximo das tabelas TABGRAFO, TABNT e TABT, foram declaradas com valores apropriados para o grafo da linguagem PASCAL dado no apêndice I e devem ser alteradas para outras linguagens. MAXK dá o tamanho máximo da pilha do analisador; esse tamanho deve ser baseado em testes de programas típicos, para se obter um limite razoável. Note-se que o procedimento EMPILHA não testa a ultrapassagem desse limite, já que TOPO foi declarada do tipo 0..MAXK; teoricamente, esse intervalo ("range") deveria ser testado pelos programas-objeto produzidos pelos compiladores PASCAL, o que em geral não é o caso.

– Os procedimentos EMPILHA e DESEMPILHA manipulam a pilha do analisador. ANALEX devolve apenas o 1º parâmetro do AL conforme descrito em 3.4. CARREGADOR carrega as tabelas do grafo; não é necessário que ele siga os passos da rotina descrita em 5.6.3; ele poderia ler os registros de entrada, produzir as tabelas e gravá-las em um arquivo intermediário e deduzir o valor de MAXG, MAXNT e MAXT; o CARREGADOR leria essas tabelas diretamente do arquivo intermediário; infelizmente esse não é um processo simples em PASCAL, pela falta de limites variáveis de "arrays". Também não existe em PASCAL iniciali-

zação de "arrays" ou a declaração destes como constantes, o que poderia eliminar a necessidade da carga das tabelas.

— O procedimento TRATAERRO, que faz o tratamento de erros sintáticos, será descrito no item 5.8.

— Alguns pontos do procedimento ANSIN foram marcados com (+) e (++). Eles indicam os pontos onde deve ser manipulada a pilha sintática (ver menção à mesma em 2.3), o que será descrito em 5.7.

— O programa principal, que chama os procedimentos CARREGADOR e ANSIN, não é apresentado. O parâmetro OBJETIVO é o índice do símbolo inicial da gramática (ou do grafo) na TABNT. *Atenção*: para que ANSIN funcione perfeitamente, é necessário que se introduza em TABGRAFO[0] um nó inicial contendo, pela ordem, os seguintes campos: (false,m,0,0,n) onde m é o índice de OBJETIVO na TABNT e n é o número da rotina semântica a ser executada no fim da compilação; foi com esse nó em mente que inicializamos a primeira célula da pilha com 0 (zero).

```
type alpha = record SIMB: packed array [1:6] of char end;
const MAXG = 244; MAXNT = 13; MAXT = 66; MAXK = 50;
var TABGRAFO: array [0..MAXG] of
                    record TER: Boolean;
                           SIM, ALT, SUC: 0..MAXG;
                           SEM: integer  end;
    TABNT: array [1..MAXNT] of
                    record NOME: alpha;
                           PRIM: 1..MAXG end;
    TABT: array [1..MAXT] of alpha;
    K: array [1..MAXK] of 0..MAXG;/*pilha do analisador */
    TOPO: 0..MAXG;/* índice do topo de K */
procedure DESEMPILHA (var p: 0..MAXG);
  begin   p:= K[TOPO]; TOPO: = TOPO-1 end;
procedure EMPILHA (p: 0..MAXG);
  begin   TOPO:= TOPO + 1; K[TOPO]:= p end;
procedure ANALEX (var PROX: alpha)/* analisador léxico;retorna o próximo símbolo */
procedure CARREGADOR;/* carrega as tabelas */
procedure ANSIN (OBJETIVO: 1..MAXNT; var SUCESSO: Boolean);
var CONTINUE: Boolean;/* controle da continuação da interpretação */
  I: 0..MAXG;/* índice para nós do grafo; indica o nó sendo interpretado */
  IU: 0..MAXG;/* índice do primeiro nó não encontrado;usado pela rotina de erro */
  ENT: alpha;/* próximo símbolo de entrada */
procedure TRATAERRO (IU: 1..MAXG);/* tratamento de erro; recebe o índice do primeiro nó
                                                    não encontrado */
begin
  ANALEX (ENT);/* lê o primeiro símbolo */
  /* (+) */
  TOPO:   1; K[1]:=0;/* inicialização da pilha */
  I:= TABNT[OBJETIVO].PRIM;/* aponta para o primeiro nó do símbolo inicial */
  IU:   I;
  CONTINUE: = true;
  while CONTINUE do
   if I ≠ 0 /* não é o fim de uma produção? */
     then if TABGRAFO[I].TER /* é um terminal? */
            then if TABGRAFO[I].SIM = 0 /* é um λ-nó? */
                   then begin /* vai para o sucessor */
                          I:= TABGRAFO[I] SUC;
```

```
                    IU: = I /* início da lista de erro */
                  end
            else if TABT[TABGRAFO[I].SIM] = ENT /* reconheceu o terminal? */

                  then begin /* (++) */
                        ANALEX(ENT); /* lê o próximo */
                        I: = TABGRAFO[I].SUC; /* vai para o sucessor */
                        IU: = I /* início da lista de erro */
                       end
                  else if TABGRAFO[I]. ALT≠ 0 /* há alternativa? */
                        then /* tome a alternativa */
                          I: = TABGRAFO[I].ALT
                        else TRATAERRO(IU)
      else begin /* é um não-terminal */
             EMPILHA(I);
             /* vai para o primeiro nó do não-terminal */
             I: = TABNT[TABGRAFO[I].SIM].PRIM
           end
else /  é o fim de um lado direito de uma produção */
  if TOPO≠0 /* pilha não está vazia? */
    then begin
           DESEMPILHA(I);
           /* vai para o sucessor do não-terminal */
           I: = TABGRAFO[I].SUC;
           IU: = I /* início da lista de erro */
         end
    else begin
           if ENT = '$' /* fim de arquivo? */
             then SUCESSO: = true
             else SUCESSO: = false;/*estão sobrando símbolos */
           CONTINUE: = false /* encerra a análise */
         end
end;
```

Antes de deixarmos este item, é interessante observar que as gramáticas ESLL(1), com as restrições vistas (v. 5.5), foram definidas baseando-se no algoritmo acima. Essas restrições é que permitem que o algoritmo analise corretamente a cadeia de entrada. Por exemplo, a gramática S→ab|ac geraria um grafo (e uma tabela) o qual, se interpretado pelo analisador, não seria capaz de reconhecer a cadeia 'ac'. De fato, o primeiro 'a' forçaria sempre a escolha da primeira alternativa da gramática. As gramáticas ESLL(1) são tais que o próximo símbolo de entrada determina univocamente o caminho a ser seguido no grafo correspondente. Lembramos que isso implica no fato de elas serem do tipo "forte" (v. 5.4): não é necessário usar-se o passado da análise (isto é, o trecho da forma sentencial já analisado) para tomar a decisão de qual alternativa do grafo deve ser seguida.

É importante notar também que existem grafos que seguem as restrições dadas em 5.5, sendo portanto adequados para o algoritmo descrito, mas que não provêm de gramáticas ESLL(1), segundo as regras de construção dos grafos dadas em 4.8. Na fig. 5.18 apresentamos um trecho de grafo que se encaixa dentro dessa situação. Ele corresponde a comandos separados por um número qualquer de ';'. Esse efeito é conseguido na PASCAL através da existência do comando vazio. Note-se que a produção block::= **begin** { statm||;{;}*}+ **end** é ESLL(1) e a ela corresponde um grafo como o da fig. 5.18 trocando-se as posições do **end** e dos ';'.

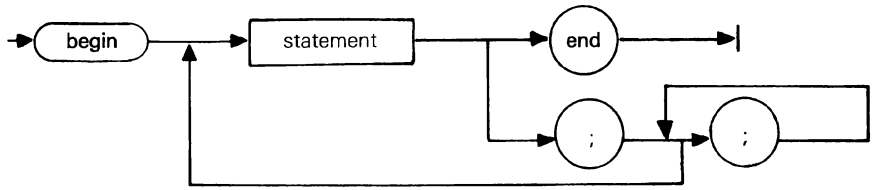


Fig. 5.18

Casos como o da fig. 5.18 sugerem a técnica de projetar grafos diretamente, satisfazendo às restrições de 5.5, sem que se passe pela fase de escrever uma gramática ESLL(1). O uso dessa classe de gramáticas neste texto deveu-se primordialmente a motivos didáticos, tendo-se com isso feito uma introdução às gramáticas formais e à geração de analisadores sintáticos a partir das mesmas.

5.7 A PILHA SINTÁTICA

Como já mencionamos anteriormente (v. 5.6.1), o AS aqui introduzido não efetua, a rigor, uma análise sintática. Para cada novo símbolo de entrada, o AS percorre simplesmente um trecho do grafo sintático, ignorando inclusive todos os símbolos já reconhecidos. Assim, não são construídas as formas sentenciais intermediárias entre a cadeia de entrada e o símbolo inicial da gramática, o que constituiria realmente uma análise sintática. Vejamos como se deve introduzir algumas modificações a fim de que o procedimento ANSIN de 5.6.4 construa as formas sentenciais.

Seja uma gramática ESLL(1) $G = (V_N, V_T, S, P)$, e $F_1 = e_1 \ldots e_{i-1} e_i \ldots e_n \in L(G)$. Sejam as formas sentenciais $S \overset{*}{\Rightarrow} F_{q+1} \overset{*}{\Rightarrow} F_q \overset{*}{\Rightarrow} F_1$. Vejamos como se pode encarar a passagem de F_q para F_{q+1}, durante o reconhecimento efetuado pelo AS. Seja $F_q = \gamma_1 \gamma_2 \ldots \gamma_m e_i \ldots e_n$, com $\gamma_1, \ldots, \gamma_m \in V_N \cup V_T$.

Suponhamos que no próximo reconhecimento direto uma parte de F_q seja reconhecida como o objetivo atual, o não-terminal N, isto é, queremos representar o passo $\gamma_j \ldots \gamma_m e_i \ldots e_{i+p} \overset{*}{\Rightarrow} N$,
onde $1 \leq j \leq m+1$ e $i-1 \leq p \leq n$; $j = m+1$ e $p = i-1$ indicam, respectivamente, a ausência de símbolos de $\gamma_j \ldots \gamma_m$ e de $e_i \ldots e_{i+p}$.

Nesse caso, N está em um nó apontado necessariamente pelo topo da pilha K do analisador. Após esse passo, teremos $F_{q+1} = \gamma_1 \gamma_2 \ldots \gamma_{j-1} N e_{i+p+1} \ldots e_n$.

Vê-se portanto que, para se obter as formas sentenciais, é preciso efetuar o seguinte:

a) Guardar toda a parte $\gamma_1 \ldots \gamma_m$ já reconhecida.

b) O reconhecimento direto de um não-terminal N pode englobar os símbolos mais à direita de $\gamma \ldots \gamma_m$ e símbolos mais à esquerda da cadeia de entrada $e_i \ldots e_n$. Todos esses símbolos devem ser substituídos por N.

c) Depois dessa substituição, N torna-se o símbolo mais à direita da parte já reconhecida.

Essas considerações sugerem uma estrutura de pilha que denominaremos de *pilha sintática*, abreviada por PS, para armazenar a parte já reconhecida $\gamma_1 \ldots \gamma_m$ das formas sentenciais, já que os reconhecimentos implicam em substituição, sempre, dos símbolos mais à direita dessa subcadeia, por um não-terminal N que é colocado na posição mais à direita da parte já reconhecida. Pode-se também ver que cada novo símbolo terminal e reconhecido durante o percurso do grafo deve ser empilhado na PS. Por outro lado, sempre que se chegar ao fim do lado direito de uma produção (isto é, quando um nó reconhecido não tiver sucessor ou, em termos do ANSIN, $I = 0$ no começo da malha **while**), devemos desempilhar da PS todo esse lado direito e, em seguida, empilhar o não-terminal N do lado esquerdo da produção. Es-

te N está no nó apontado pelo topo da pilha K. Aqui surge um interessante problema: qual o tamanho do lado direito da produção de N que devemos reconhecer, isto é, quantos símbolos de uma forma sentencial devem ser substituídos por N? Note-se que as gramáticas ESLL(1) podem ter expressões regulares (estendidas) nos lados direitos das produções. Com isso, alguns lados direitos deixam de ter tamanho fixo. Por exemplo, ao querermos aplicar a produção $N \rightarrow ab^*c$ em um reconhecimento direto, podemos ter um número qualquer de 'b's na forma sentencial. Esse problema pode ser resolvido com muita simplicidade em nosso método de análise sintática. Para isso, basta introduzir um campo adicional em cada célula da pilha K do analisador. Se atingirmos um nó n do grafo, com um não-terminal N, n é empilhado em K, como anteriormente, agora junto com um número r que é o índice da próxima célula livre da PS. Isso é compreensível: a partir dessa célula da PS é que serão colocados os símbolos que formarão a subcadeia $\gamma_j \ldots \gamma_m e_i \ldots e_{i+p}$, como já vimos. Quando chegamos ao fim de um lado direito de N, desempilhamos de K o par (n,r,); desempilhamos todas as células da PS desde a célula r inclusive, até o topo; vamos ao nó n e lá encontramos o apontador para a tabela de não-terminais, indicando N; empilhamos N (na célula r da PS). A inicialização de K deve ser, agora, feita com o par (0,1), se a primeira céulula de PS tem índice 1.

Na fig. 5.19 apresentamos um exemplo de funcionamento do AS, agora com a PS, para a gramática G_{17} do item 5.6.1, cujo grafo está na fig. 5.11. Usamos a cadeia 'dfaec', cuja análise já foi exemplificada naquele item. A ordem das colunas indica a ordem de execução das respectivas ações. Elementos em branco indicam a repetição do elemento anterior na mesma coluna.

Note-se que esse método funciona perfeitamente quando é reconhecida uma produção com lado direito igual a λ. Este seria o caso da cadeia 'dfadc', que deixamos para o leitor verificar.

As alterações a serem feitas no programa do AS são as seguintes:

a) Declarações adicionais e mudanças nas declarações já feitas.

```
type...
const...; MAXPS = 100;
var...
   K: array [1..MAXK] of record NO: 0..MAXG; R:1..MAXPS end;
   TOPO...

   PS: array [1.. MAXPS] of alpha; /* pilha sintática */
   TOPPS: 0..MAXPS; /* índice do topo da PS */
procedure DESEMPILHA (var p: 0..MAXG);
   begin p: = K[TOPO].NO; TOPPS: = K[TOPO].R;TOPO: = TOPO - 1;
      /* empilha o não-terminal na PS */
      PS[TOPPS]: = TABNT[TABGRAFO[p].SIM].NOME end;
procedure EMPILHA (p: 0..MAXG);
   begin TOPO: = TOPO + 1; K[TOPO].NO: = p; K[TOPO].R: = TOPPS + 1 end;
```

b) Alterações no procedimento ANSIN. Os pontos onde essas alterações devem ser feitas são aqueles marcados em ANSIN com (+) e (++).

```
(+):
   begin
      TOPO: = 1; K[1].NO: = 0; K[1].R: = 1; /* inicialização da pilha do AS */
      TOPPS: = 0;
      I: = TABNT...
(++):
   begin
      TOPPS: = TOPPS + 1; PS[TOPPS]: = ENT; /* empilha em PS o símbolo reconhecido */
      ANALEX...
```

símbolo lido	cadeia restante	nó visitado	pilha do analisador	pilha sintática 1 2 3 4 5
d	faec	0	(0,1)	
		1		
		5		d
f	aec	6	(0,1) (6,2)	
		8		d f
a	ec	9	(0,1) (6,2) (9,3)	
		1		d f a
e	c	2		
		3	(0,1) (6,2) (9,3) (3,4)	
		1		
		5		
		7		d f a e
c		0	(0,1) (6,2) (9,3)	d f a S
		4		d f a S c
$		0	(0,1) (6,2)	d f S
		8		
		10		
		0	(0,1)	d M
		0		S

Fig. 5.19

Um exercício interessante, que deixamos para o leitor, é a construção de árvores sintáticas a partir das informações da pilha sintática e da pilha do analisador.

5.8 TRATAMENTO AUTOMÁTICO DE ERROS SINTÁTICOS

Como vimos no item 2.2 o tratamento de erros sintáticos é uma parte fundamental do algoritmo de análise sintática. (Usaremos a denominação "tratamento" em lugar de "recuperação" como é por vezes empregada a tradução literal de "error recovery".) Trata-se, em síntese, de dois processos distintos: a *detecção* do erro e a *correção sintática* da cadeia de entrada, de modo que a análise sintática possa prosseguir até o fim dessa cadeia. Este último aspecto é essencial para os compiladores de uso prático: o usuário quer conhecer o maior número de erros que podem ser detectados no programa-fonte, para que ele possa posteriormente corrigi-los todos a um só tempo. Seria altamente indesejável se o compilador parasse a análise sintática no primeiro erro sintático detectado, ignorando o restante do programa.

Uma noção importante nessa área é a do *tratamento automático de erros sintáticos*. Essa expressão é usada quando os processos de tratamento independem do analisador, sendo dependentes exclusivamente da gramática analisada. Isto é, no caso do analisador ser fixo e receber uma gramática (ou tabela que a representa, como em nosso caso) como entrada não deve ser necessário reprogramar algumas de suas partes para tratar os erros de gramática diferentes. Assim, uma vez programado o processo de tratamento, ele se aplica a qualquer gramática aceita pelo analisador. Este será o caso dos algoritmos a serem descritos neste item.

5.8.1 DETECÇÃO DE ERROS SINTÁTICOS

Há duas possibilidades de ocorrerem erros sintáticos:

a) O símbolo inicial da gramática foi encontrado, e ainda sobram símbolos da cadeia de entrada, ainda não analisados. Essa situação é deduzida pelo procedimento ANSIN nos últimos comandos deste; a pilha do analisador está vazia e não foi detectado um "fim de arquivo". Note-se que a pilha fica vazia somente quando o símbolo inicial é reconhecido, devido à inicialização do analisador, a qual coloca na pilha o apontador para o nó inicial contendo

esse símbolo (v. 5.6.4). O programa principal que chama ANSIN pode saber que esse erro ocorreu, pelo valor "false" do parâmetro de saída SUCESSO.

b) O nó sendo visitado, de índice I, é do tipo terminal, não é λ-nó, mas seu conteúdo não coincide com o símbolo lido, e não há alternativa para esse nó, isto é:
TABGRAFO[I].TER = true, TABGRAFO[I].SIM ≠ 0,
TABT[TABGRAFO[I].SIM] ≠ ENT e TABGRAFO[I]. ALT = 0
Essa situação ocorre exatamente onde se dá a chamada de TRATAERRO(IU) no procedimento ANSIN. Vejamos em detalhes as ações que devem ser tomadas nesse caso.

Logo após a detecção de um erro sintático do tipo (b), é necessário emitir uma mensagem de erro, mostrando ao usuário exatamente em que símbolo t da cadeia de entrada deu-se o erro. Isso pode ser conseguido emitindo-se um caractere como '|' no registro de saída imediatamente abaixo do registro onde saiu t, e exatamente embaixo deste símbolo, como mostramos em exemplos no fim deste item. O AL tem um apontador para o registro de entrada, que pode ser usado para esse fim. Em seguida ao caractere '|' deve ser emitida, no mesmo registro, mensagem indicando qual o tipo de erro que ocorreu. Em nosso caso, podemos listar todos os símbolos terminais $t_1, t_2, \ldots, t_n$ esperados, com $n \geq 1$, nenhum dos quais coincide com t. Note-se que o nó com t_n não tem nó alternativo, como já foi dito; o nó de t_{n-1} pode ser um nó cuja alternativa é t_n; eventualmente, algum t_i pode ter um não-terminal N como alternativa; neste caso t_{i+1} deve ser o primeiro terminal do subgrafo de N, com $1 \leq i \leq n$, se o primeiro nó de N não for não-terminal. É necessário percorrer a seqüência de alternativas $t_1, t_2, \ldots, t_i$, N continuando com a seqüência $t_{i+1}, t_{i+2}, \ldots, t_n$, emitindo todos os terminais que forem aparecendo. Evidentemente, $\{t_{i+1}, t_{i+2}, \ldots, t_n\}$ = FIRST(N) (v. 5.5). Se algum desses terminais coincidisse com t, ou fosse λ, não teria ocorrido a condição de erro. Naturalmente, a segunda seqüência pode também ser interrompida por um não-terminal N', e assim por diante. Conhecendo-se o nó inicial da primeira seqüência não é difícil percorrer as seqüências de alternativas, pulando para o primeiro nó de cada não-terminal que ocorrer no meio do percurso. Este se encerra com um nó terminal que não tem alternativa, e que será obrigatoriamente t_n. Denominaremos essa seqüência dos nós $t_1, \ldots, t_n$ de *percurso de alternativas*.

Formalmente podemos definir os símbolos terminais desse percurso para o sucessor de um nó m contendo o símbolo $x \epsilon V_N \cup V_T$, da produção $M \rightarrow \alpha x \beta \in P$ $\alpha, \beta \epsilon (V_N \cup V_T)^*$, como sendo o conjunto

$$PERALT(x, M \rightarrow \alpha x \beta) = \{t \epsilon V_T \mid S \overset{*}{\Rightarrow} \alpha' M \beta' \overset{*}{\Rightarrow} \alpha' \alpha x \beta \beta' \overset{*}{\Rightarrow} \alpha' \alpha x t \beta'' \beta', \alpha', \beta', \beta'' \epsilon (V_N \cup V_T)^*\}$$

O procedimento ANSIN já foi programado guardando-se o índice IU do nó de t_1, que é o nó inicial do percurso. Como se pode ver em ANSIN, IU assume um novo valor nos seguintes casos, pela ordem: primeiro nó do símbolo inicial da gramática; sucessor de um λ-nó; sucessor de um terminal que acabou de ser reconhecido; sucessor de um não-terminal que acabou de ser reconhecido. Damos a seguir o trecho do procedimento TRATAERRO que emite a mensagem dos terminais esperados e não encontrados. Suponhamos que o AL guarde na variável global IP a posição do primeiro caractere do símbolo t onde ocorreu o erro. Se no percurso de alternativas a partir de t_1 for encontrado um nó não-terminal, seu índice é desempilhado da pilha do analisador onde ele foi erroneamente colocado.

```
procedure TRATAERRO (IU: 1..MAXG);
  var IX: 0..MAXG; /* percorre a lista de alternativas */
      IT: 1..80; /* contador para IP */
  begin
    IX:= IU; /* início do percurso */
    writeln('ERRO␣␣␣'); /* muda de registro e emite início da mensagem */
    for IT:= 1 to IP−1 do write ('␣');
    write ('|␣');
    while IX≠0 do /* percorre ti */
      if TABGRAFO[IX].TER /* é terminal? */
        then begin /* emite terminal esperado */
               write ('''', TABT[TABGRAFO[IX].SIM], ''',␣');
```

```
                /* vai para a próxima alternativa */
                IX: = TABGRAFO[IX].ALT
              end
        else begin /* nó não-terminal */
                IX: = TABNT[TABGRAFO[IX].SIM].PRIM;
                TOPO: = TOPO - 1 /* desempilha um não-terminal */
             end;
    write ('ESPERADO(S)');
```

Como exemplo, tomemos a gramática G_{17} do item 5.6.1 cujo grafo está na fig. 5.11. A cadeia 'aadfgecc' (o 'g' foi colocado a mais) provocaria a seguinte mensagem de erro:

```
                A A D F G E C C
ERRO                    |: 'A', 'D', 'E', ESPERADO(S)
```

No caso da cadeia de entrada 'ag' teríamos

```
                A G
ERRO              |: 'B', 'A', 'D', 'E', ESPERADO(S)
```

Note-se que no exemplo seguinte a cadeia de entrada 'adgec', onde 'g' foi escrito em lugar de 'f', provocará a mensagem:

```
              A D G E C
ERRO                |: 'C', ESPERADO(S)
```

Essa mensagem é devida ao fato de que 't' tem uma λ-alternativa, portanto M é sempre reconhecido quando procurado. Esse mesmo problema ocorre com o comando (errado) em PASCAL

```
IF A = B THEN THEN C:= D ELSE C:= E.
```

Neste caso, um comando vazio será reconhecido pelo analisador entre os doi then (v. grafo no apêndice I). Em seguida, será emitida a mensagem

```
                |: ';', 'END', ESPERADO(S)
```

Se o erro for corrigido sintaticamente assumindo-se ';' em lugar do segundo THEN haverá detecção de novo erro no ELSE

```
                          |: ';', 'END' ESPERADO(S)
```

Vemos que o projeto de uma gramática deveria levar em conta o tratamento de erros feito pelo analisador sintático; de maneira geral, podemos recomendar que não se utilize λ-nós, alternativas vazias ou lados direitos consistindo exclusivamente da cadeia vazia. Um exemplo do primeiro caso é alternativa de ELSE e, do terceiro, o caso de "statement" vazio, ambos na gramática da linguagem PASCAL /J-W 74/ e conservados no grafo apresentado no apêndice I. No item 5.8.2e propomos uma detecção e correção de erros causados pelo reconhecimento indevido de alternativas vazias, como nos exemplos vistos acima.

5.8.2 CORREÇÃO SINTÁTICA DE ERROS

É importante notar o título deste item: a correção a ser tratada aqui é exclusivamente sintática. Supomos que ao encontrar o primeiro erro (sintático ou "semântico") o compilador deixe de gerar código-objeto. Tanto a análise sintática como a de contexto devem continuar, para que sejam detectados erros dessas naturezas no restante do texto, sendo o usuário informado dos mesmos.

Seja uma ESLL(1)-gramática $G = (V_N, V_T, S, P)$ e uma cadeia de entrada $e_1 \ldots e_j e_{j+1} \ldots e_n$. Suponhamos que a subcadeia $e_1 \ldots e_{j-1}$ já tenha sido reconhecida pelo AS, e que houve uma detecção de erro (cf. 5.8.1.) no símbolo e_j. Lembremos que foi tentado o percurso de alternativas partindo de nó apontado por IU, passando pelos nós contendo os terminais $t_1, \ldots, t_n$:

A correção será feita através de uma de quatro estratégias, que passaremos a descrever a seguir. Essas estratégias são testadas seqüencialmente; se nenhuma conseguir corrigir o erro, ignora-se o símbolo e_j e passa-se ao próximo símbolo e_{j+1}, voltando a testar as quatro es-

tratégias novamente. Se mais do que uma tiver sucesso, usa-se a primeira estratégia da seqüência que ignorar o menor número de símbolos de entrada.

a) Estratégia E: eliminação de um símbolo

Nesta estratégia supomos que o usuário cometeu o engano de escrever a cadeia de entrada com um símbolo a mais. Isto é, e_j é supérfluo e deve ser eliminado. Para testarmos se essa hipótese pode ser assumida, basta compararmos e_{j+1} com os terminais $t_1, t_2, \ldots, t_n$ mencionados acima. Se $e_{j+1} = t_i$, devemos emitir mensagem de correção avisando que e_j foi ignorado e continuar a análise a partir do nó que contém t_i.

Para testarmos essa igualdade, basta seguirmos procedimento exatamente análogo à rotina de detecção vista no item anterior. É preciso, no entanto, tomar um cuidado especial: é necessário empilhar na pilha K do analisador qualquer nó não-terminal que aparecer no meio do percurso, para que se possa corrigir essa pilha adequadamente. É necessário guardar o apontador para o topo da pilha K, tal como este se apresentava antes do início da execução da estratégia. Se a estratégia não tiver sucesso, isto é, $e_{j+1} \neq t_i$, $i = 1,2,\ldots,n$, é necessário voltar ao topo original, desempilhando-se eventuais não-terminais que ocorreram durante o percurso. Neste caso passa-se à estratégia seguinte.

No primeiro exemplo do item anterior, teríamos agora as seguintes mensagens correspondentes ao tratamento do erro:

```
          A A D F G E C C
ERRO                |: 'A', 'D', 'E', ESPERADO(S)
CORREÇÃO            |: IGNORADO
```

b) Estratégia I: inserção de um símbolo

Nesta estratégia supomos que o usuário cometeu o engano de omitir um símbolo que deve ser inserido. Para testarmos se essa hipótese pode ser assumida basta supormos que entre e_{j-1} e e_j havia um símbolo que é idêntico a algum t_i, $i = 1,\ldots,n$. Seja s_1^i o terminal do nó sucessor do que contém t_i. Neste caso, a estratégia terá sucesso com t_i e este deve ser inserido, se e_j for idêntico a um dos terminais do percurso de alternativas $s_1^i, s_2^i, \ldots, s_{n_i}^i$ construído a partir de s_1^i como na detecção de erro ou na estratégia E. Note-se que nenhum dos elementos dessa seqüência será um λ-nó, pois se durante o percurso ocorrer um λ-nó deve-se tomar seu nó sucessor como o próximo. Se durante o percurso de alternativas a partir do sucessor de t_i for atingido algum nó sem alternativa (ALT = 0), a estratégia não terá tido sucesso para t_i; passa-se então a testá-la para t_{i+1}. Se algum t_i não tiver sucessor, passa-se para t_{i+1}; note-se que, assim, $s_k^i \in$ FOLLOW(t_i) (v. 5.5), $k = 1,\ldots,n_i$, mas este conjunto pode eventualmente conter símbolos diferentes de qualquer s_k^i. O processamento da estratégia termina com sucesso logo que for encontrado o primeiro s_k^i tal que $e_j = s_k^i$.

Essa estratégia resume-se em achar-se o s_k^i com os menores índices i e k, tal que $s_k^i = e_j$ e $s_k^i \in$ PERALT (t_i, $M \rightarrow \alpha t_i \beta$), onde t_i é o conteúdo de um dos nós do percurso de alternativa $t_1,\ldots,t_n$ descrito acima, correspondendo ao t_i de $M \rightarrow \alpha t_i \beta$.

Como exemplo, tomemos a gramática G_{17} e seu grafo da fig. 5.11 (v. 5.6.1). A cadeia de entrada 'dfb' produziria as seguintes mensagens de tratamento de erros, pois não se aplica a estratégia anterior:

```
            D F B
ERRO            |: 'A', 'D', 'E', ESPERADO(S)
CORREÇÃO        |: 'A' INSERIDO ANTES DESTE SÍMBOLO
```

No caso da linguagem PASCAL (v. apêndice I), teríamos, por exemplo,

```
          VAR A B;
ERRO            |: ',', ':' ESPERADO(S)
CORREÇÃO        |: ',', INSERIDO ANTES DESTE SÍMBOLO
```

Damos, a seguir, a codificação dessa estratégia em PASCAL. Supomos que ela seja inserida no procedimento TRATAERRO, de 5.8.1, não sendo declaradas as variáveis que já constam do mesmo ou de ANSIN. Note-se que o percurso de $t_1,\ldots,t_n$ e de $s^i_1..s^i_{ni}$ são feitos de maneira similar à da detecção do erro. Do mesmo modo que na estratégia E, é necessário empilhar na pilha do analisador eventuais nós não-terminais que forem encontrados nesses dois percursos. A variável I recebe o índice do nó contendo $s^i_k = e_j$, para continuação da análise sintática pelo procedimento ANSIN.

```
var ACHOU: Boolean; /* indica se esta estratégia teve sucesso */
    TOPOAUX1, TOPOAUX2: 1..MAXK; /* novos topos de K para t1,...,tn e s1i,...,sni*/
    IY: 0..MAXG; /* índice para s1i,...,sni */
ACHOU:=false;
TOPOAUX1:=TOPO;
IX:=IU;
while (IX≠0) and not ACHOU do /* percorre t1,...,tn */
  if TABGRAFO[IX].TER /* símbolo esperado é terminal? */
    then begin /* ti */
           TOPOAUX2:=TOPOAUX1;
           IY:=TABGRAFO[IX].SUC;
           while (IY≠0) and not ACHOU do /* percorre s1i,...,sni */
             if TABGRAFO[IY].TER /* é terminal? */
               then if TABGRAFO[IY].SIM=0 /* ski é λ-nó? */
                      then IY:=TABGRAFO[IY].SUC
                      else if TABT[TABGRAFO[IY].SIM]=ENT /* achou? */
                             then begin
                                    TOPPS:=TOPPS+1;
                                    /* empilha ti na PS */
                                    PS[TOPPS]:=TABT[TABGRAFO[IX].SIM];
                                    ACHOU:=true;
                                    I:=IY /* novo nó para ANSIN */
                                  end
                             else IY:=TABGRAFO[IY].ALT
               else /* sucessor é não-terminal */
                 begin /* empilha e vai para o subgrafo */
                   TOPOAUX2:=TOPOAUX2+1;
                   K[TOPOAUX2].NO:=IY;
                   /* reserva uma célula na PS para o símbolo a ser inserido */
                   K[TOPOAUX2].R:=TOPPS+2;
                   IY:=TABNT[TABGRAFO[IY].SIM].PRIM
                 end;
           if not ACHOU then IX:=TABGRAFO[IX].ALT
         end
    else begin / não-terminal na seqüência dos esperados */
           TOPOAUX1:=TOPOAUX1+1;
           K[TOPOAUX1].NO:=IX;
           K[TOPOAUX1].R:=TOPPS+1;
           IX:=TABNT[TABGRAFO[IX].SIM].PRIM;
         end;
if ACHOU then
  begin
    TOPO:=TOPOAUX2; /* novo topo */
    writeln('CORREÇÃO'); /* muda de registro */
```

```
for IT:=1 to IP-1 do write ('ß');
write ('|:ß', TABT[TABGRAFO[IX].SIM], 'INSERIDO ANTES DESTE SÍMBOLO')
end;
```

c) Estratégia T: um símbolo trocado

Nesta estratégia supomos que o usuário escreveu um símbolo erradamente. Isto é, um símbolo correto e_j^i foi trocado pelo símbolo incorreto e_j. Para testarmos se essa hipótese pode ser assumida, basta aplicarmos o algoritmo da estratégia I para o símbolo e_{j+1} em lugar de e_j como foi o caso.

Por exemplo, para a gramática G_{17}, cujo grafo está na fig. 5.11 (v. 5.6.1), a cadeia de entrada 'agfec' produziria as seguintes mensagens, pois não se aplicam as estratégias anteriores:

```
                A  G  F  E  C
ERRO              |: 'B', 'A', 'D', 'E', ESPERADO(S)
CORREÇÃO          |: TROCADO POR 'D'
```

O exemplo de PASCAL visto em 5.8.1 encaixa-se nesta estratégia:

```
IF A=B THEN THEN C:=D ELSE C:=E
ERRO              |: ';', 'END', ESPERADO(S)
CORREÇÃO          |: TROCADO POR ';'
ERRO                          |: ';', 'END', ESPERADO(S)
CORREÇÃO                      |: TROCADO POR ';'
```

Essas correções são devidas ao fato de que um identificador (no caso, C) é um sucessor possível para o ';' através de "statement".

d) Estratégia D: busca de delimitador

Nesta estratégia, testamos se e_j é idêntico a algum dos símbolos que são sucessores dos não-terminais sendo procurados, ou pertencem a percursos de alternativas desses sucessores. Isto é, suponhamos que a pilha K contenha (no campo NO) apontadores para os nós $m_1, m_2, \ldots, m_n$, que contém respectivamente os não-terminais $N_1, N_2, \ldots, N_n$, onde o índice de m_1 está no topo de K. Seja s_1^i o nó sucessor de m_i, $i = 1, \ldots, n$, e $s_1^i, s_2^i, \ldots, s_{n_i}^i$ o percurso de alternativas cujo nó inicial é s_1^i. Procuramos o terminal s_k^i, $k = 1, \ldots, n_i$, com os menores índices i e k, tal que $e_j = s_k^i$. Para isso, desempilhamos inicialmente o índice m_1 e procuramos e_j em $s_1^1, s_2^1, \ldots, s_{n_1}^1$; se não for encontrado, desempilhamos m_2 e procuramos e_j em $s_1^2, s_2^2, \ldots, s_{n_2}^2$, e assim por diante, até termos desempilhado m_n e testado o percurso de alternativas correspondente. Como se vê estamos buscando símbolos terminais que possam funcionar como delimitadores dos não-terminais que estão sendo procurados.

Como exemplo, tomemos novamente o caso da gramática G_{17} e seu grafo da fig. 5.11 (v. 5.6.1). A cadeia de entrada 'ac' produz as seguintes mensagens correspondentes ao tratamento do erro segundo esta estratégia, pois não se aplicam as estratégias anteriores:

```
                A  C
ERRO              |: 'B', 'A', 'D', 'E', ESPERADO(S)
CORREÇÃO          |: ASSUMIDO COMO DELIMITADOR
```

Em PASCAL, podemos ter o seguinte caso (v. apêndice I):

```
                IF A=B THEN GOTO ELSE A:=B
ERRO                             |: 'NUMB', ESPERADO(S)
CORREÇÃO                         |: ASSUMIDO COMO DELIMITADOR
```

Damos, a seguir, a codificação dessa estratégia em PASCAL; supomos, como nos outros casos, que ela seja inserida no procedimento TRATAERRO de 5.8.1. Note-se a necessidade de guardar em KAUX a parte "superior" da pilha K do analisador ao se tentar um percurso, já que o desempilhamento de um nó m_i provocaria uma alteração de K se algum nó não-terminal do percurso de alternativas do sucessor de m_i fosse empilhado no lugar de m_i.

```
var /* pilha de trabalho */
    KAUX array [1..MAXK] of record NO: 0..MAXG; R: 1..MAXPS end;
    TOPPSAUX: 0..MAXPS; /* cópia de trabalho de TOPPS */
    J: integer;
ACHOU:= false;
TOPOAUX1:= TOPO; /* topo para cada novo percurso */
while (TOPOAUX1≠0) and not ACHOU do
   begin /* desempilha e vai para o percurso */
      IX:=K[TOPOAUX1].NO; /* índice de m_i */
      TOPPSAUX:= K[TOPOAUX1].R; /* índice de N_i na PS */
      TOPOAUX1:= TOPOAUX1−1; /* desempilha m_i */
      TOPOAUX2:= TOPOAUX1; /* auxiliar para o percurso */
      IY:=TABGRAFO[IX].SUC; /* início do percurso */
      while (IY≠0) and not ACHOU do /* percurso */
        if TABGRAFO[IY].TER /* é terminal? */
         then if TABGRAFO[IY].SIM=0 /* é λ-nó? */
                 then IY:=TABGRAFO[IY].SUC
                 else if TABT[TABGRAFO[IY].SIM]=ENT /* achou? */
                         then begin
                                 TOPPS:=TOPPSAUX;
                                 /* empilha N_i na PS */
                                 PS[TOPPS]:=TABNT[TABGRAFO[IX].SIM].NOME;
                                 ACHOU:=true;
                                 I:=IY /* novo nó para ANSIN */
                              end
                         else IY:=TABGRAFO[IY].ALT
        else /* é não-terminal */
            begin /* empilha e vai para o subgrafo */
              TOPOAUX2:= TOPOAUX2+1;
              KAUX[TOPOAUX2].NO:= IY;
              KAUX[TOPOAUX2].R:= TOPPSAUX;
              IY:=TABNT[TABGRAFO[IY].SIM].PRIM
            end
    end;
if ACHOU
   then begin
           TOPO:=TOPOAUX2;
           /* copia pilha de trabalho em K */
           for J:=TOPOAUX1+1 to TOPO do K[J]:=KAUX[J];
           writeln ('CORREÇÃO');  /* muda de registro */
           for IT:=1 to IP-1 do write(' ');
           write('↑', 'ASSUMIDO COMO DELIMITADOR')
        end;
```

Se nenhuma das quatro estratégias tiver sucesso, deve-se ler o próximo símbolo de entrada e_{j+1} e tentá-las novamente com esse novo símbolo. Como se pode observar, não há nesse caso sentido em se testar a estratégia I, pois ela é idêntica à T, usando o símbolo atual em lugar do próximo. Assim, para todos os efeitos, I já foi testada com e_{j+1} ao se ter testado T com esse símbolo. Além disso, a estratégia E também busca e_{j+1} para erro em e_j. Isso sugere que se apliquem todas as estratégias logo de início para e_j e e_{j+1} simultaneamente; o tempo gasto a mais é desprezível, pois o que demora é percorrer o grafo. Após testarem-se as quatro

estratégias para e_j e e_{j+1}, lemos os dois próximos símbolos e_{j+2} e e_{j+3}, voltando a testar novamente toda a seqüência (a) a (d). Se mais de uma estratégia aplicar-se a um ou mais símbolos de entrada, pode-se tomar como norma usar-se o símbolo mais à esquerda e a primeira estratégia na ordem apresentada. O primeiro critério é importante: deve-se ignorar o menor número possível de símbolos de entrada, isto é, a correção deve ser feita o mais cedo que se puder.

Vejamos alguns exemplos de teste das estratégias e mensagens para mais do que um símbolo. Para a gramática G_{17}, fig. 5.11 (v. 5.6.1), temos as seguintes extensões aos exemplos já vistos para as diferentes estratégias:

```
            A A D F G G G E C E
ERRO                  |: 'A', 'D', 'E', ESPERADO(S)
CORREÇÃO                    |: IGNORADO ATÉ AQUI INCLUSIVE
          D F G G F E
ERRO            |: 'A', 'D', 'E',ESPERADO(S)
CORREÇÃO              |: IGNORADO ATÉ AQUI INCLUSIVE E TROCADO POR 'D'
          A G G G C
ERRO            |: 'B', 'A', 'D', 'E' ESPERADO(S)
CORREÇÃO              |: IGNORADO ATÉ AQUI INCLUSIVE
CORREÇÃO                |: ASSUMIDO COMO DELIMITADOR
```

Note-se a necessidade de um registro de saída à mais no último caso, para estratégia D.

e) Estratégia V: validação de λ-alternativas

Como vimos em 5.8.1, e segundo o grafo do apêndice I, um comando como

if A = B **then then** C: = D **else** C: = E

produz o reconhecimento de **if** A = B **then** como um comando, pois, ao percorrer a última alternativa do nó inicial do subgrafo de "statm", a cadeia vazia é reconhecida (erroneamente) entre os dois **then** como sendo um comando vazio, completando a seqüência **if** expr **then** statm. Observando-se esse exemplo e outros semelhantes podemos concluir que o problema reside no reconhecimento indevido de λ-alternativas. Vejamos como se pode evitar esse reconhecimento indevido, através da verificação do contexto à direita do não-terminal cujo reconhecimento é encerrado pela λ-alternativa.

Suponhamos que no topo da pilha K do analisador se tenha, num dado momento da análise, um apontador para o nó n do subgrafo do não-terminal M, e que n contenha o não-terminal N. No exemplo, teríamos M = "statm" e n seria o nó contendo N = "statm" no ramo **if** depois do nó **then.** Durante o reconhecimento do subgrafo de N aparece uma λ-alternativa (no exemplo, o ramo inferior de "statm"), e o próximo símbolo lido é o terminal t (no caso, o segundo **then**). Nessa situação, o analisador poderia verificar se o reconhecimento da λ-alternativa é válido. Para isso, bastaria verificar se t é um sucessor admissível para n, isto é, t ocorre como conteúdo de um dos nós que pertencem à seqüência de alternativas que se inicia com o nó sucessor de n. No nosso exemplo, essa seqüência contém **else** e uma λ-alternativa. Como o nó **else** não é reconhecido, a nova λ-alternativa deve provocar a repetição de todo o processo, testando-se os sucessores de M, depois de desempilhar o apontador para n. Suponhamos que o novo topo da pilha do analisador contenha agora um apontador para o nó m contendo M e de onde se desviou para o subgrafo desse não-terminal. As seguintes condições provocam a parada desse processo: i) algum nó da seqüência de alternativas do sucessor de um dos nós desempilhados contém t; nesse caso deve-se aceitar a λ-alternativa de N e o reconhecimento prossegue normalmente; ii) atinge-se o último nó de uma seqüência de alternativas, nó esse que contém um terminal diferente de t; nesse caso não se deve aceitar a λ-alternativa de N; deve-se imprimir mensagem de erro (listando-se todos os terminais que são sucessores dos não-terminais procurados) e iniciar o processamento das estratégias vistas anteriormente, a partir do último nó reconhecido, com a seguinte diferença fundamental: as λ-alternativas devem ser ignoradas, como se não existissem; iii) no processo de desempilha-

mento da pilha do analisador, atinge-se o fundo da mesma; deve-se proceder como no caso (ii).

Note-se que a distinção entre um λ-nó comum e um λ-nó que é uma λ-alternativa reside no fato de, ao contrário do segundo, o primeiro poder ter um sucessor.

A implementação dessa estratégia exige uma alteração do analisador sintático (a detecção de uma λ-alternativa inicia o processo de validação e de correção do erro, se necessário) e das rotinas das estratégias (que nesse caso devem passar a ignorar as λ-alternativas).

O resultado dessa estratégia no exemplo do comando "if" apresentado anteriormente poderia ser o seguinte, já que a estratégia E teria sucesso, e supondo-se que o "if" tenha ocorrido dentro de um "repeat":

```
                IF A = B THEN THEN C: = D ELSE C: = E
ERRO                          |: 'ELSE', ';', 'UNTIL', ESPERADO(S)
CORREÇÃO                      |: IGNORADO
```

Vejamos mais um exemplo, agora para a gramática G_{17} (v. 5.6.1) cujo grafo está na fig. 5.11:

```
          A A D E C C
ERRO              |: 'F', 'C', ESPERADO(S)
CORREÇÃO          |: 'F' INSERIDO ANTES DESTE SÍMBOLO
```

Note-se que se a cadeia correta deveria conter uma cadeia vazia, esta estratégia poderá não dar o resultado esperado, como no seguinte exemplo, em que o segundo **then** é supérfluo:

```
          IF A>B THEN THEN ELSE C: = A;
ERRO                    |: 'ELSE', ';', 'UNTIL', ESPERADO(S)
CORREÇÃO                     |: IGNORADO ATÉ AQUI INCLUSIVE
```

Note-se ainda que a mensagem de erro proposta não é muito adequada, pois a correção pode provocar a continuação da análise em um símbolo diferente daqueles listados. Pode-se, por exemplo, usar a mensagem "... ESPERADO(S) AO ASSUMIR UMA CADEIA VAZIA". No entanto, ela exigiria do programador um conhecimento do que é uma "cadeia vazja".

Pode-se levantar a dúvida se realmente valeria a pena implementar esta estratégia, já que ela implica em aumento no tempo de processamento do analisador, devido à validação necessária para todas as λ-alternativas. Ocorre, porém, que o tempo gasto na fase de análise sintática é em geral pequeno em comparação com o tempo restante (análises léxica e de contexto e geração de código). Assim sendo, o ganho na maior clareza da correção torna-se justificativa suficiente para a adoção desta estratégia.

É interessante observar que a estratégia D é em geral usada nos compiladores cujo AS usa uma técnica "top-down" porém com esse tratamento de erros programado "manualmente". Neste caso o programador deduz, ele mesmo, quais são os possíveis delimitadores que podem ser usados em cada caso de erro e força o AL a procurar um desses símbolos na cadeia de entrada. Essa é a técnica apresentada por Wirth no capítulo sobre compilação de /WIR 76/. Comparando, na prática, as mensagens e correção de erros das estratégias aqui apresentadas, com as mensagens de erros de compiladores que seguem o padrão para a PASCAL /J-W 74/, podemos constatar que aquelas são em geral mais freqüentes e mais claras, apesar de serem "automáticas", isto é, não houve nenhuma programação adicional além da gramática ESLL(1) para a PASCAL.

Vale a pena também citar que existem trabalhos de levantamento dos erros mais freqüentes feitos por programadores, como o de Ripley e Druseikis /R-D 76/. Neste trabalho, os autores chegam à conclusão de que a maior parte das vezes um só símbolo está errado em cada trecho do programa-fonte. Por meio das estratégias de eliminação, inserção e troca, deve-se conseguir tratar a maior parte desses erros. Por outro lado, é a estratégia do delimitador que garante

o fato de não se eliminar um número exagerado de símbolos de entrada. De fato, se fosse necessário escolher apenas uma das quatro estratégias, recomendaríamos essa última.

Finalmente, note-se que o tratamento automático de erros permite que se instale um sistema de testes de gramáticas, levando em conta não somente a estrutura sintática, mas também uma clara detecção e conveniente correção de erros.

5.9 EFICIÊNCIA DO ANALISADOR SINTÁTICO

Neste item faremos uma breve incursão na complexidade do AS introduzido neste capítulo, incluindo as rotinas de tratamento de erro. Veremos que ele pode ser considerado eficiente, tanto em termos de espaço quanto de tempo.

5.9.1 ESPAÇO OCUPADO

O espaço fixo usado pelo AS é bastante reduzido: como vimos, as rotinas são razoavelmente simples e concisas.

A parte variável do AS diz respeito às tabelas TABGRAFO, TABNT e TABT bem como às pilhas K e KAUX. Não é necessário levar em conta a PS pois, como vimos, esta pode ser dispensada depois dos testes da gramática.

Em geral, a TABNT é desprezível em comparação com a TABGRAFO; a TABT pode ser razoavelmente grande, principalmente se se levantar a restrição que impusemos de no máximo 6 caracteres por nome. No entanto, ela é em geral bem menor do que a TABGRAFO. Esta, como vimos, tem uma entrada para cada símbolo do lado direito de cada produção da gramática; ou, em outras palavras, uma entrada para cada nó do grafo sintático correspondente, acrescido de uma entrada para cada λ-alternativa. Uma gramática para a PASCAL exige cerca de 240 entradas. Essas entradas podem ser bastante compactadas, por exemplo usando a seguinte declaração:

```
var TABGRAFO: array [1..MAXG] of
                packed record TER: Boolean; /* 1 bit */
                              SIM: 0..255 ;  /* 8 bits */
                              ALT: 0..127 ;  /* 7 bits */
                              SUC:0 ..127 ;  /* 7 bits */
                              SEM:0 ..511 ;  /* 9 bits */
                             end
```

Com isso, temos uma única palavra de 32 bits, ou uma subdivisão exata em duas de 16 bits. Pode parecer estranho que se permitam apenas 128 nós de alternativas ou de sucessores. Porém, esses índices podem ser conservados na TABGRAFO tal qual constam dos registros sugeridos para entrada do CARREGADOR (v. 5.6.3), isto é, índices relativos ao primeiro nó do subgrafo de cada não-terminal. Assim, teríamos um máximo de 128 nós por subgrafo, o que é bem razoável. Compactações adicionais podem ser obtidas reduzindo-se o campo ALT a apenas 1 bit, que indica a existência de uma alternativa; esta seria o nó seguinte. Além disso, em lugar de SEM pode-se tomar o número do nó como sendo o número da rotina semântica. Com isso reduzimos o espaço de cada nó para apenas 16 bits, o que dá um sistema extremamente eficiente, comparando-se com outros.

Note-se que a compactação aumenta o tempo de interpretação do grafo, mas este não é um fator que pesa muito no tempo geral de compilação. Por outro lado, essa compactação pode ser muito importante em computadores pequenos.

5.9.2 TEMPO REQUERIDO

Examinemos independentemente o AS ANSIN e cada estratégia de detecção e correção de erro. O tempo requerido será dado por um limite no número de nós percorridos da TABGRAFO, isto é, o tempo para o pior caso.

Suponhamos que se tenha uma ESLL(1)-gramática G em que o percurso de alternativas (segundo 5.8.1.) contendo o maior número de nós, contando-se também os nós não-terminais do percurso, contenha p_{max} nós. Como G é do tipo ESLL(1), não há recursões de não-terminais à esquerda ($N \overset{*}{\rightarrow} N\alpha$); portanto p_{max} é finito. Suponhamos que a cadeia de entrada a ser analisada contenha n símbolos.

a) ANSIN

Para cada símbolo t de entrada, o procedimento ANSIN faz um percurso de no máximo p_{max} nós, comparando t com o conteúdo de cada nó terminal e tomando a sua alternativa ou desviando para o primeiro nó do subgrafo de um não-terminal. Portanto, o número de nós percorridos será, no pior caso, de $p_{max}.n$, isto é, ANSIN tem complexidade linear no tempo.

b) Detecção de erro e estratégia E

Considerando-se um símbolo de entrada errado, temos em cada uma dessas duas rotinas um percurso máximo de p_{max} nós. Portanto, no pior caso temos $p_{max}.n$ nós percorridos para toda a cadeia.

c) Estratégias I e T

Nestes casos, usando a nomenclatura de 5.8.2, estratégia I, para cada símbolo de entrada teremos um percurso máximo dos nós de $t_1,\ldots,t_{p_{max}}$; por outro lado, para cada t_i destes temos um percurso máximo de $s^i_1,\ldots,s^i_{p_{max}}$. Portanto, teremos no pior caso um percurso de $p^2_{max}.n$ nós.

d) Estratégias D e V

Nestes casos, podemos observar que para cada símbolo de entrada errado temos que desempilhar de K consecutivamente os índices dos nós não-terminais, e para cada um desses nós fazer um percurso de no máximo p_{max} nós. Suponhamos que se esteja no i-ésimo símbolo da cadeia de entrada. Nessa situação, só se pode ter no máximo MAXNT.i não-terminais na pilha K, onde MAXNT é o número de não-terminais da gramática. Isso se deve ao fato de que não há recursividades de não-terminais à esquerda ($N \overset{*}{\rightarrow} N\alpha$); portanto, depois de se empilhar MAXNT não-terminais distintos, é necessariamente obrigatório reconhecer um símbolo terminal. Portanto, no i-ésimo símbolo de entrada temos que percorrer no máximo MAXNT. $p_{max}.i$ nós. Portanto, para os n símbolos de entrada temos

$$\sum_{i=1}^{n} \text{MAXNT}.p_{max}.i = \text{MAXNT}.p_{max}.n(n+1)/2$$

Portanto, estas estratégias percorrem um número de nós da ordem de n^2, no pior caso.

Com essas considerações, podemos verificar que o algoritmo de análise sintática com tratamento de erros é eficiente. Excluindo-se as estratégias D e V, temos até mesmo um algoritmo altamente eficiente; considerando-se que o número de erros é em geral pequeno comparado com o tamanho das cadeias de entrada, que as estratégias D e V nem sempre chegam a ser aplicadas (e, mesmo se forem, o número de células de K é em geral bem menor que MAXNT) e que a grande maioria das seqüências de alternativas tem comprimento muito menor do que p_{max}, podemos dizer que o comportamento do AS é bastante satisfatório.

5.10 O GRAFO SINTÁTICO DA LINGUAGEM PASCAL

No apêndice I apresentamos a gramática da linguagem PASCAL sob a forma de ESLL(1)-gramática e o grafo sintático correspondente construído segundo as regras dadas em 4.8. As seguintes observações são pertinentes a esse grafo:

a) A linguagem PASCAL conforme /J-W 74/ foi estritamente observada;
b) Empregamos dois tipos de terminais:

— os que correm no programa-fonte tal qual aparecem na gramática e no grafo, e que não são alterados pelo AL: são os símbolos reservados, isto é, símbolos especiais (';', ':=', etc.)' e as palavras reservadas (**begin, nil,** etc.). Estas últimas são escritas em negrito.

— os que são reconhecidos pelo AL como identificadores (IDEN), constantes numéricas sem sinal (NUMB) e cadeias de caracteres (STRING). Os identificadores são classificados pela rotina da tabela de símbolos (v. 6.2) nas seguintes classes, dependendo de sua declaração: COIDEN, FIIDEN. FUIDEN, PRIDEN, TYIDEN e VAIDEN correspondendo, respectivamente, aos identificadores de constantes, campos de registros ("fields"), funções, procedimentos, tipos e variáveis. Todos esses são escritos com letras maiúsculas. Note-se que esses identificadores são representados no grafo de /J-W 74/ como não-terminais (blocos retangulares).

c) Foram eliminados da gramática e do grafo aqui apresentados os não-terminais do grafo de /J-W 74/ "identifier", "unsigned integer", "unsigned number" e "unsigned constant". Note-se que não fazemos distinção entre constante numérica inteira ou "real" (ponto flutuante), ficando a distinção por conta do AL (neste texto, tratamos apenas do caso inteiro).

d) Em alguns casos, foi necessário substituir o uso de alguns não-terminais do grafo de /J-W 74/ por suas produções, como por exemplo "constant" de "field list", para se obter uma ESLL (1)-gramática. Pelo mesmo motivo, foi necessário duplicar alguns terminais, como por exemplo vários casos de IDEN em "block".

e) O não-terminal "variable" /J-W 74/ foi alterado, retirando-se os terminais "variable identifier" e "field identifier" e denominando-se o restante do subgrafo de "infipo" ("index, field, pointer").

f) Alguns nós do grafo contêm um número de uma rotina "semântica" no campo SEM de TABGRAFO (v. 5.6.2) que deve ser chamada pelo AS logo após o reconhecimento do nó. Esses números são colocados no lado direito dos nós, sobre o arco que sai dos mesmos. No restante do texto quando fizermos referência à rotina "semântica" de número n, usaremos a abreviatura RS, correspondendo à chamada RS(n) do procedimento contendo essas rotinas. No apêndice II damos uma lista de localização das rotinas "semânticas" no grafo sintático do apêndice I.

g) Em certos casos foi necessário introduzir um λ-nó, como por exemplo em "filist", para que uma rotina semântica fosse executada antes do nó sucessor.

5.11 PROJETO — PARTES III, IV E V: CARREGADOR SINTÁTICO, ANALISADOR SINTÁTICO E TRATAMENTO DE ERROS SINTÁTICOS

a) Parte III — Carregador sintático

Nesta 3ª parte do projeto deve ser programado o carregador sintático descrito em 5.6.3. O programa lê registros de entrada que descrevem os nós do grafo. Os registros são de dois tipos, onde os nomes dos campos são os vistos em 5.6.3.

1. Cabeça. Corresponde ao não-terminal de um subgrafo. O seu formato pode ser o seguinte onde os números indicam as colunas dos campos:

'C'	NOMER
1	3-8

2. Nó. Corresponde a um nó no subgrafo do não-terminal especificado no registro cabeça precedente. O seu formato pode ser:

TIPO	NOMER	NUMNO	ALTR	SUCR	SEMR
1	3-8	10-12	14-16	18-20	22-24

TIPO pode conter os valores 'N' (não-terminal), 'T' (terminal) e 'I' (classe de identificadores). Este último é uma extensão do que já foi descrito no item 5.6.3, para linguagens como a PASCAL, como descrito em 5.10; NUMNO contém o índice do nó relativo ao primeiro nó do subgrafo (v. fig. 5.17).

O carregador lê os registros de entrada e monta três tabelas: TABNT e TABT, com os símbolos não-terminais e terminais respectivamente; TABGRAFO com os nós do grafo.

A TABNT é uma tabela com dois campos por entrada: o primeiro contém o identificador de um não-terminal e o segundo o índice da TABGRAFO onde se encontra o primeiro nó do subgrafo desse não-terminal. Esse índice deve ser deduzido pelo carregador, como no programa visto em 5.6.3.

A TABT contém os símbolos terminais, ou símbolos reservados, da gramática. Substitua a 1ª parte do projeto (rotina de símbolos reservados), de modo que o carregador sintático carregue ele próprio a TABT, usando a técnica de espalhamento proposta em 2.4. No caso de nó terminal com tipo 'I' (classe de identificadores), o conteúdo do campo NOMER deve ser carregado na Tabela de Colisões, sem pertencer a nenhuma cadeia de colisões. Assim, poderá ser declarado no programa-fonte, por exemplo, uma variável de nome COIDEN; esse identificador não será reconhecido como palavra reservada, apesar de pertencer à TABT, podendo portanto ser apontado por nós do grafo.

A TABGRAFO deve conter os campos vistos em 5.6.3, já com índices absolutos nos campos ALT e SUC.

Estenda a lista de comandos de controle do compilador, descritos em 3.6, introduzindo os seguintes comandos: TABT, TABNT e TABGRAFO. Eles especificam que se deseja a impressão (ou exibição) das tabelas correspondentes, montadas pelo carregador.

Como teste, e para uso futuro do compilador, codifique o grafo sintático da linguagem PASCAL dado no apêndice I.

Opcional: programe uma rotina que lê uma ESLL(1)-gramática e que gera a TABGRAFO. Esta tarefa não é simples. Sugerimos que se desenvolva uma metagramática que descreve a linguagem das ESLL(1)-gramáticas. As tabelas do analisador sintático para essa metagramática são codificadas manualmente e o analisador sintático ANSIN a ser programado na próxima parte do projeto pode ser usado para analisar as gramáticas que se quer converter nas tabelas TABGRAFO. Para facilitar a construção dessas tabelas, use uma pilha "semântica" como a que será introduzida no capítulo 6. Essa técnica de usar uma metagramática é semelhante à de auto-implementação de compiladores ("bootstrapping") a ser examinada no item 10.9.

b) Parte IV — Analisador sintático

Nesta parte do projeto deve ser programado e testado o analisador sintático AS segundo o procedimento ANSIN de 5.6.4. Inicialmente é chamado o carregador sintático, que lê a descrição do grafo e monta as tabelas TABGRAFO, TABNT e TABT. Pode-se também supor que essas tabelas sejam produzidas pelo carregador em um processamento independente e constituam arquivos guardados em alguma unidade de armazenamento, como por exemplo discos magnéticos; nesse caso, deve-se ler o conteúdo das tabelas — que podem ser declaradas como um só "record" compactado em PASCAL, aumentando a eficiência da leitura. Uma terceira solução seria declarar as tabelas com valores iniciais ou como constantes, o que seria a forma mais eficiente, mas que infelizmente não pode ser aplicada em PASCAL, que não conta com inicialização de matrizes e registros ou declaração de elementos destes como constantes.

Introduza os comandos de controle do compilador PSINON e PSINOF. O primeiro especifica que a pilha sintática PS (v. 5.7) deve começar a ser impressa (exibida). O segundo interrompe a impressão da pilha. Essa impressão é fundamental para o teste de gramáticas. Ela deve ser feita da seguinte maneira: Todas as vezes que houver uma alteração na PS, esta é impressa, mostrando o seu novo estado. Para economizar saída, pode-se imprimir de cada vez no máximo os 10 elementos mais próximos do topo da PS, começando sempre em um

elemento de índice $10n + 1$ e indo no máximo até $10n + 10$, $n \geq 0$. À esquerda desses elementos deve-se imprimir o índice do primeiro elemento da linha, isto é, $10n + 1$.

O AS deve ser testado com o grafo da PASCAL, como apresentado no apêndice I. Note que, da maneira como o carregador sintático foi especificado, as classes de identificadores COIDEN, FIIDEN, FUIDEN, PRIDEN, TYIDEN e VAIDEN (v. 5.10) serão carregadas na TABT, apesar de não funcionarem como palavras reservadas. Posteriormente, quando a Tabela de Identificadores e Rótulos (capítulo 6) for implementada, poderá ocorrer a troca de um IDEN por sua classe. No momento, teste o AS colocando os próprios identificadores dessas classes no programa-fonte a ser analisado, em lugar de identificadores de constantes, de campos, etc. Assim, por exemplo, coloque VAIDEN := VAIDEN como um comando de simples atribuição. Além disso, e até a implementação da Tabela de Identificadores e Rótulos, carregue na TABT as classes COIDEN, FILDEN,etc. como se fossem palavras reservadas, isto é, com tipo 'T' no registro de entrada do carregador (um registro para cada classe é suficiente) ou então altere a rotina de símbolos reservados, produzindo também a busca na área de colisões onde estão as classes de identificadores.

c) Parte V — Tratamento de erros sintáticos

Nesta parte, devem ser implementadas as rotinas de tratamento de erros sintáticos descritas em 5.8.2, onde foram indicados os formatos de saída das mensagens de erro e da correção efetuada. Após serem feitas correções na pilha sintática deve ser produzida uma exibição desta, como vimos em (b), se o comando de controle do compilador PSINON tiver sido lido e desde a sua última leitura não tiver sido lido um comando PSINOF

Se, por alguma razão, for escolhida para implementação somente uma das quatro estratégias vistas, deve-se empregar a D — busca de delimitador, pelas razões expostas no fim do item 5.8.2.

Havendo um compilador PASCAL disponível, é interessante fazer-se uma comparação do tratamento de erros do mesmo com os resultados das estratégias aqui apresentadas.

CAPÍTULO 6

AS TABELAS DE SÍMBOLOS

6.1 INTRODUÇÃO

Neste capítulo abordaremos alguns problemas da construção e consulta das tabelas de símbolos do compilador. Na fig. 2.6 podemos ver que elas são construídas pelo analisador de contexto (AC), por chamada do analisador sintático (AS); o mesmo AC consulta as tabelas de símbolos, o que também é feito pelo gerador de código (GC). As tabelas de símbolos são: tabelas de identificadores e de rótulos — "labels" (TIR), tabela de constantes numéricas (TCN) e tabela de constantes alfanuméricas — cadeias de caracteres (TCA). Somente a primeira apresenta problemas quanto à decisão do tipo de acesso que deve ser feito: inserção de um novo símbolo ou consulta para obter as informações referentes a um símbolo já inserido previamente. Isso se deve ao fato de um determinado identificador ou rótulo não poder ser declarado mais do que uma vez em um programa principal ou procedimento. Já com a TCN e a TCA seus símbolos não têm declaração explícita; pode-se considerar como declaração a primeira ocorrência do símbolo em todo o programa, ocasião em que ele é inserido na tabela correspondente. Daí para a frente, qualquer nova ocorrência desse mesmo símbolo deve provocar uma simples busca, sem nova inserção.

As tabelas TCN e TCA são muito mais simples do que a TIR; nas primeiras temos como argumento o código interno do símbolo e como valor retornado apenas o endereço no programa-objeto. Eventualmente, poderíamos ter, no caso da primeira, uma distinção entre constantes de ponto fixo ou ponto flutuante, mas como não trataremos desse caso aqui, o texto será um pouco mais simples. Já na TIR temos o problema do escopo ("scope"), isto é, região de validade dos identificadores: ao se chegar ao fim de um procedimento, todos os símbolos da TIR nele declarados devem ser retirados da tabela. Além disso, temos identificadores de várias classes e tipos, com eventuais estruturas no caso de tipos estruturados, que são as matrizes ("arrays") e os registros ("records").

6.2 CLASSES DE IDENTIFICADORES E INTRODUÇÃO ÀS ROTINAS "SEMÂNTICAS"

Os identificadores são detectados no programa-fonte pelo AL, que retorna ao AS, que o chama, parâmetros p_1 com a cadeia IDEN e p_2 com a cadeia de caracteres que foi reconhecida como um identificador (v. 3.4) restringida em nosso caso apenas aos 6 primeiros caracte-

res. Examinando-se, no entanto, o grafo sintático da PASCAL (v. apêndice I), observamos que ele contém não só o terminal IDEN correspondendo a identificador, mas também outros terminais como COIDEN, FIIDEN etc. para casos especiais de identificadores. Como vimos no item 5.10, esses identificadores correspondem a uma classificação que depende da maneira como eles foram declarados. Assim, o AL retorna um IDEN, que eventualmente deve ser substituído, como unidade sintática, por exemplo por COIDEN, se ele tiver sido declarado como uma constante. Essa substituição não se dá sempre: ela depende da fase da análise. Se esta corresponder a uma fase de declarações, então é necessário deixar o código IDEN intacto, pois é assim que ele aparece no grafo sintático. Por outro lado, se o momento da análise corresponde ao emprego de um identificador já declarado, por exemplo em uma expressão, então teremos que substituir IDEN pela classe correspondente. Damos, a seguir, as classes de identificadores e as respectivas declarações:
COIDEN — declarado em "block" precedido por **const**, e em "sitype" (ver explicação deste último abaixo);
FIIDEN — declarado em "filist";
FUIDEN — declarado precedido por **func** em "block" e em "palist";
PRIDEN — declarado precedido por **proc** em "block" e em "palist";
TYIDEN — declarado precedido de **type** em "block";
VAIDEN — declarado precedido por **var** em "block" e em "palist".

A classe COIDEN contém também os identificadores declarados em "sitype" entre '('e')' que é a declaração do que denominaremos de uma *seqüência de constantes simbólicas*. Por exemplo, as declarações:

```
type COR = (AZUL, VERDE, ROSA);
var CÉU: COR;
```

definem as constantes simbólicas AZUL, VERDE e ROSA, e a variável CÉU que pode assumir um desses valores. Na atribuição CÉU: = AZUL, AZUL deve ser reconhecido como COIDEN. O tipo COR é portanto uma seqüência de constantes simbólicas. A denominação de "seqüência" é devida à ordem estabelecida na declaração. Por exemplo, as expressões AZUL < VERDE e ROSA < VERDE fornecem os valores true e false respectivamente.

Podemos declarar a TIR como um vetor (matriz unidimensional) de registros; estes devem ter um campo IDROT com a cadeia de caracteres do identificador como fornecida pelo AL em seu parâmetro p_2, e que funciona como o argumento da tabela. No caso de ser um identificador (pode ainda ser um rótulo), temos em um campo adicional CLASSE a classe do mesmo. Vamos usar também esse campo para guardar a informação de que o conteúdo de IDROT é um rótulo, colocando o valor "LABEL' nesse campo. A declaração da TIR seria, até aqui, a seguinte:

```
var TIR: array [1..MAXTIR] of
    record IDROT: alpha; /*V. alpha em 3.5*/
           CLASSE: (COIDEN, FIIDEN, FUIDEN, LABEL, PRIDEN, TYIDEN, VAIDEN);


    end;
```

Quando um identificador é introduzido na TIR, seu campo CLASSE deve receber o valor correspondente à sua classe. Isso é conseguido por meio da execução de uma rotina "semântica", chamada pelo AS logo após o reconhecimento de um nó com IDEN no grafo. Assim, ao se reconhecer um IDEN que foi precedido de **const** em "block", a RS_{10} é executada. Esta atribui o valor do parâmetro p_2 do AL ao campo IDROT da entrada correspondente a esse identificador e coloca COIDEN no campo CLASSE. Se ocorrer um outro identificador ainda nesta declaração **const**, a mesma rotina RS_{10} será executada, também introduzindo-o adequadamente na TIR.

Continuemos com esse último exemplo de entrada de identificadores de constantes na TIR. Após a execução da rotina RS_{10}, o controle volta para o AS, que reconhece um '=' e a

seguir desvia para o subgrafo do não-terminal "const". Suponhamos, por exemplo, que se tenha no programa-fonte sendo analisado a declaração **const** NAIPE1 = COPAS. Nesse caso, COPAS é um COIDEN. NAIPE1 foi retornado pelo AL como IDEN, e tudo correu bem até aí com o subgrafo de "block"; mas o AL também reconhece COPAS como um IDEN; se este último símbolo for usado para a comparação com o conteúdo do nó onde está COIDEN do subgrafo de "const", este não será reconhecido. Alguma ação deve ser executada para transformar o IDEN de COPAS em COIDEN logo após esse símbolo ser lido, depois do reconhecimento do '='. Essa mudança de IDEN para COIDEN deve ser feita pelo próprio AS, chamando uma rotina de busca na TIR. Colocaremos essa chamada imediatamente após a chamada do AL, controlada por um teste se o parâmetro p_1 do AL contém IDEN e se é necessário fazer uma busca e a conseqüente troca, no caso desse exemplo, para COIDEN. Para isso, introduzimos uma variável global ao compilador,

var BUSCATIR: Boolean;

A seqüência de comandos do AS ANSIN (v. 5.6.4) será agora, em lugar da 2ª chamada ANALEX (ENT) (observe-se que no programa do ANSIN não entramos nos detalhes dos parâmetros p_1, p_2 e p_3 do AL):

```
ANALEX (ENT,p2,p3);
if (ENT = 'IDEN') and BUSCATIR
    then /* chama uma rotina que busca na TIR uma entrada de índice E com
            TIR[E].IDROT = p2 e TIR[E]. CLASSE ≠ FIIDEN; se não encontrar emite erro de
            contexto; se encontrar faz ENT: = TIR[E].CLASSE */
```

Note-se que a rotina de busca ignora as entradas com classe FIIDEN; isso se deve ao fato de os identificadores de campos de registros poderem coincidir com outros identificadores. Os primeiros serão sempre procurados diretamente a partir dos identificadores dos registros. Voltaremos a esse problema em 7.10.

Agora surge a questão: quais rotinas alteram o valor de BUSCATIR? A resposta é: algumas das rotinas "semânticas" (daí termos colocado BUSCATIR como variável global). Na realidade, consideramos como "estado" normal de BUSCATIR o valor true; sempre que aparece um nó com IDEN no grafo, deve ter sido chamada a rotina RS_1, que faz BUSCATIR: = false. Logo após esse nó ou um fechamento transitivo em que ele aparece, é executada a RS_2, que faz BUSCATIR: = true. Note-se que antes de haver um desvio para "block" deve ser chamada a RS_1, o que acontece em "progrm" e em "block" — neste último caso a RS_{141} de ";" no ramo **proc** termina a sua execução com a chamada de RS_1; o mesmo deve se passar antes de desvios para "filist"; com isso diminuímos o número de chamadas a essas duas rotinas.

A inserção de um IDEN na TIR, com a sua classe, é feita por rotinas semânticas cujos números vão de 10 a 65, tendo dezenas distintas para classes distintas, como se pode ver no grafo. (O apêndice II facilita a localização das rotinas semânticas no grafo.) Assim, por exemplo, ao ser executada, a RS_{50} sempre insere o identificador reconhecido pelo AL com classe TYIDEN. Note-se a distinção entre as rotinas RS_{40} e RS_{45}: ambas colocam a classe PRIDEN no identificador, mas são diferentes pois no primeiro caso estamos definindo o nome de um procedimento, e no outro um parâmetro tipo procedimento. Nesses casos, outras informações a serem colocadas nas respectivas entradas da TIR devem diferir.

O caso da RS_5 em "palist" é especial, pois ela se aplica a duas classes diferentes de identificadores: FUIDEN se se passou pelo nó **func** e VAIDEN em caso contrário, com ou sem especificação **var**. Um parâmetro de procedimento ou função (denominado de "parâmetro formal") do tipo **var** indica que uma alteração no valor desse parâmetro dentro do procedimento provoca uma alteração no valor do parâmetro de chamada (denominado de "parâmetro atual"); um parâmetro formal sem nenhuma especificação indica que uma alteração de

seu valor no procedimento não deve provocar uma alteração do valor do parâmetro atual. A distinção entre esses dois tipos será feita através dos valores PARAM e PARAV, respectivamente, do campo TIPO da TIR, como veremos em 6.3k.

Algumas palavras sobre a chamada das rotinas "semânticas". Observando o grafo sintático e o processo de análise sintática, vemos que elas devem ser chamadas pelo AS imediatamente após o reconhecimento de um nó, antes de se ler o próximo símbolo se esse nó for terminal, e antes de se passar para o sucessor, em qualquer tipo de nó. Com isso, podemos determinar na rotina ANSIN os seguintes pontos de chamada da rotina "semântica":

a) O ponto marcado com (+ +); considerando as alterações de ANSIN descritas em 5.7, teríamos:

```
begin
   TOPPS: = TOPPS + 1; PS[TOPPS]: = ENT;
   RS(TABGRAFO[I].SEM); /* chamada da RS referenciada pelo nó terminal reconhecido */
   ANALEX (ENT,p2,p3);
```

b) Imediatamente após a chamada da rotina DESEMPILHA em ANSIN

```
   DESEMPILHA(I);
   RS(TABGRAFO[I].SEM); /* chamada da RS referenciada pelo nó não-terminal reconhecido* /
   I: = TABGRAFO[I].SUC;
```

6.3 ESTRUTURA DA TABELA DE IDENTIFICADORES E RÓTULOS

Vamos completar a estrutura da TIR, adicionando àquela introduzida no item anterior os campos necessários para se armazenar todas as informações sobre cada elemento da tabela. Damos, abaixo, a nova declaração da TIR, explicando a seguir o significado de cada campo.

```
var TIR: array [1..MAXTIR] of
         record IDROT: alpha;
           CLASSE: (COINDEN, FIIDEN, FUIDEN, LABEL, NOIDEN, PRIDEN, TYIDEN,
                    VAIDEN);
           TIPO:   (ARRAYT, FILET, INTCTT, LABELT, PARAM, PARAV, POINTT,
                    RECT, SEQT, SCIDT, SETT, STANDT, STRCTT, SUBRGT, TYIDT);
           INDTT:  integer;
           ENDOBJ: integer
         end
```

- IDROT: contém o identificador ou rótulo (parâmetro p_2 do AL);
- CLASSE: como visto em 6.2, acrescida da classe fictícia NOIDEN, indicando que não há identificador;
- TIPO: contém um escalar indicando o tipo do elemento; descreveremos em detalhe o significado de cada um, e como afetam as informações da TIR; para cada um dos tipos SUBRGT, ARRAYT e RECT é declarada uma tabela adicional, que denominaremos de *tabela descritora do tipo;*
- INDTT: contém um índice para um elemento da TIR, das tabelas TCN e TCA (v. 6.1) ou das tabelas descritoras, conforme o caso; pode ainda conter os números de ordem de uma constante simbólica, cf. (d) adiante;
- ENDOBJ: contém informação relativa ao endereço do elemento no programa-objeto ou espaço ocupado por ele.

Tratemos inicialmente de ENDOBJ. Esse campo contém para as classes:

- COIDEN, LABEL e VAIDEN, o endereço do elemento no programa-objeto;
- FIIDEN, o endereço relativo ao início do registro;
- TYIDEN, o espaço total da memória ocupado por um objeto desse tipo;

— FUIDEN e PRIDEN, o endereço da função ou procedimento no programa-objeto gerado, no caso do campo TIPO conter TYIDT, e endereço do parâmetro relativo à base da função ou procedimento no caso de PARAM, cf. (b) adiante.

Vejamos agora, para cada tipo, o conteúdo dos campos da TIR e a organização da tabela descritora do tipo, quando ela existir. Os campos não-mencionados em cada caso são ignorados.

a) STANDT — tipo básico ("standard type", cf. /J-W 74/). Aplica-se à classe TYIDEN. Corresponde à declaração fictícia dos tipos int e char, com **type** em "block" (v. gramática ou grafo no apêndice I). No início da compilação deve-se inicializar a TIR com as seguintes entradas:

TIR[1]: 'INT', TYIDEN, STANDT, 0, 1
TIR[2]: 'CHAR', TYIDEN, STANDT, 0, 1

O compilador, ao encontrar o conteúdo STANDT no campo TIPO, deve produzir um tratamento especial, com geração das instruções adequadas ao tipo sendo compilado. Note-se que na PASCAL existe ainda o tipo básico "real" que não será abordado neste texto.

b) TYIDT — identificador de tipo ("type identifier"), declarado em "block" com **type** a menos de procedimentos e funções. Aplica-se às classes FIIDEN, FUIDEN, PRIDEN, TYIDEN, VAIDEN. Neste caso INDTT contém o índice para a TIR, correspondendo à entrada onde o identificador de tipo foi declarado. Por exemplo a declaração **var** V2: alpha gera uma entrada na TIR com os seguintes campos, pela ordem:

'V2' VAIDEN, TYIDT, m, n.

Aqui, m é o índice da entrada da TIR com 'ALPHA' no campo IDROT. No caso de classe FUIDEN, o campo INDTT contém o índice da entrada da TIR onde se encontra o tipo do resultado da função. No caso PRIDEN, ocorrem os tipos TYIDT e PARAM, usados para indicar que se trata de declaração de procedimento, ou de declaração de parâmetro do tipo procedimento, respectivamente.

c) POINTT — tipo apontador ("pointer type"), declarado em "type" com '↑'. Aplica-se às classes FIIDEN, NOIDEN, TYIDEN e VAIDEN. Neste caso INDTT contém o índice para a TIR correspondendo à entrada onde o identificador de tipo foi declarado. Por exemplo, **var** V3:↑PESSOA gera uma entrada na TIR com os seguintes campos, pela ordem:

'V3', VAIDEN, POINTT, m, n

Aqui, m é o índice da entrada da TIR com PESSOA no campo IDROT.

A TIR deve ser inicializada com a constante **nil**, correspondente a um apontador para o vazio, da seguinte maneira:

TIR[3]: 'NIL', COIDEN, POINTT, 0, n

onde n é o seu endereço no programa-objeto. Como no caso de false e true, o compilador deve tratar essa constante de maneira especial; seguindo sugestão de /WIR 71/ pode-se associar a essa constante um valor correspondente a um endereço maior do que o último endereço permitido pelo computador, onde será processado o programa-objeto, isto é, fora do espaço de endereçamento deste. Assim, um erro de endereçamento ocorrerá se houver uma tentativa de acesso a uma localização de memória apontada por **nil.**

d) SEQT e SCIDT — tipo seqüência de constantes simbólicas e tipo constante simbólica respectivamente, declarado em "sitype" entre '(' e ')'. O primeiro aplica-se às classes FIIDEN, NOIDEN, TYIDEN e VAIDEN e o segundo à classe COIDEN. As constantes simbólicas são colocadas seqüencialmente a partir da entrada seguinte na TIR, tendo em TIPO o valor SCIDT (identificador de constante simbólica, "symbolic constant identifier") e o seu número de ordem na seqüência em INDTT. Por exemplo, a declaração

type COR = (AZUL,VERDE, ROSA)

gera quatro entradas na TIR, com os seguintes campos pela ordem:

TIR[j] — 'COR', TYIDEN, SEQT, 3, 1
TIR[j+1] — 'AZUL', COIDEN, SCIDT, 0, 0
TIR[j+2] — 'VERDE', COIDEN, SCIDT, 1, 0
TIR[j+3] — 'ROSA', COIDEN, SCIDT, 2, 0

Supusemos que cada objeto de tipo SEQT ocupe uma palavra de memória donde o '1' do campo ENDOBJ de TIR[j]. No campo INDTT dessa entrada coloca-se o número de elementos da seqüência. As constantes simbólicas não são armazenadas no programa-objeto. Uma variável que é declarada com esse tipo contém simplesmente, durante a execução do programa-objeto, o número de ordem de uma das constantes simbólicas da seqüência. O compilador deve gerar no programa-objeto as constantes. 0,1,...,n, onde n é igual ao maior número de ordem atribuído a uma constante simbólica durante toda a compilação. Essas constantes serão usadas nos comandos que envolvem variáveis tipo seqüência de constantes simbólicas (comumente denominado "tipo enumeração") tais como atribuições, comparações etc.

A TIR deve ser inicializada com entradas resultantes da declaração
type Bool = (false, true), gerando:
TIR[4]: 'BOOL', TYIDEN, SEQT, 2, 1
TIR[5]: 'FALSE', COIDEN, SCIDT, 0, 0
TIR[6]: 'TRUE', COIDEN, SCIDT, 1, 0

e) SETT — tipo conjunto ("set type"), declarado em "type" com **set.** Aplica-se às classes FIIDEN, NOIDEN, TYIDEN, VAIDEN. A declaração de um objeto com esse tipo gera uma entrada na TIR, onde o campo INDTT contém o índice para a TIR, onde se encontra o tipo correspondente a "sitype" depois de **of** no subgrafo de "type". Por exemplo, considerando COR como declarado em (d) acima, a declaração
type CJCOR = **set of** COR
geraria na TIR uma entrada com os seguintes valores de campos pela ordem:
'CJCOR', TYIDEN, SETT, j, 1

Cada variável do tipo **set** ocupará uma palavra de memória, donde o '1' de ENDOBJ. Por outro lado, a declaração
var CJNAIP = **set of** (ESPAD, COPAS, OUROS, PAUS)
geraria na TIR as entradas
TIR[K]: 'CJNAIP', VAIDEN, SETT, K+1, p
TIR[K+1]: '', NOIDEN, SEQT, 4, 1
TIR[K+2]: 'ESPAD', COIDEN, SCIDT, 0, 0
TIR[K+3]: 'COPAS', COIDEN, SCIDT, 1, 0
TIR[K+4]: 'OUROS', COIDEN, SCIDT, 2, 0
TIR[K+5]: 'PAUS', COIDEN, SCIDT, 3, 0

Note-se que a segunda entrada, correspondente ao tipo seqüência de constantes simbólicas não tem identificador de tipo, donde ter-se usado NOIDEN.

f) INTCTT e STRCTT– tipo de constante ("constant type"), inteira ou cadeia de caracteres ("string"), respectivamente, declarado em "block" com **const.** Aplicam-se à classe COIDEN. Distinguem-se do tipo SCIDT — cf. (d) acima — pelo fato de que neste último caso tem-se a declaração de uma constante simbólica (AZUL, p. ex.) e no presente caso tem-se uma constante de tipo "integer", "string" ou o uso de uma constante simbólica já declarada. (O tipo "char" será aqui englobado em "string".) Assim, na entrada da TIR com tipo contendo INTCTT e STRCTT haverá um índice para as tabelas TCN e TCA (v. 6.1) contendo respectivamente as constantes numéricas e alfanuméricas que vão aparecendo no programa-fonte, ao lado de seus endereços no programa-objeto. Essas tabelas poderiam ser declaradas com:

```
var TCN: array [1..MAXTCN] of
           record VALOR: integer;
                  ENDOBJ: integer
           end;
   TCA: array [1..MAXTCA] of
          record VALOR: alpha;
                 ENDOBJ: integer
          end;
```

g) SUBRGT — tipo de intervalo ("subrange type"), declarado em "sitype" com '..'. Aplica-se às classes FIIDEN, NOIDEN, TYIDEN, VAIDEN. A este tipo está associada uma tabela descritora TABSRG, declarada como segue:

```
var TABSRG: array [1..MAXSRG] of
            record TIPO: (INTSR, CHARSR, SCIDSR);
                   INDINF, INDSUP: integer
            end;
```

onde TIPO contém um indicador de a qual tipo correspondem os limites especificados: "integer", "char" ou constante simbólica. No primeiro caso, INDINF e INDSUP apontam para as entradas na TCN correspondentes aos limites inferior e superior, respectivamente; no segundo caso, para a TCA; no terceiro caso, para entradas da TIR com campo TIPO contendo SCIDT proveniente de algum tipo de seqüência de constantes simbólicas cf. (d) acima. Por exemplo, levando-se em conta a declaração de COR em (d),

```
var V3: AZUL..ROSA
```

geraria uma entrada na TIR com os seguintes campos, pela ordem:
'V3', VAIDEN, SUBRGT, p, n.
Em TABSRG[p] seria gerada uma entrada com os campos, pela ordem:
SCIDSR, j+1, j+3

h) ARRAYT — tipo matriz ("array type"), declarado em "type" com **array.** Aplica-se às classes FIIDEN, NOIDEN, TYIDEN, VAIDEN. Associada a esse tipo há a tabela descritora TABARR, declarada como segue:

```
var TABARR: array [1..MAXARR] of
            record NUMDIM: 1..MAXDIM;
                   INDTIR: array [1..MAXDIM] of 1..MAXTIR;
                   INDTIP: 1..MAXTIR;
                   ESPTOT: integer
            end;
```

onde NUNDIM indica o número de dimensões da matriz, podendo atingir o valor máximo de MAXDIM; INDTIR contém uma seqüência de índices para a TIR, localizando as entradas onde estão declarados os tipos dos índices. Conforme a definição da PASCAL /J-W 74/, estes podem ser dos tipos: seqüência de constantes simbólicas, intervalo ou identificador de tipo definido como um desses dois últimos tipos. Se não se tratar de identificador de tipo, os tipos dos índices são inseridos na TIR. INDTIP é o índice da TIR onde se encontra o tipo dos elementos da matriz; ESPTOT deve conter o espaço total que a matriz ocupará no programa-objeto.

Neste caso, o campo ENDOBJ da entrada da TIR correspondente à matriz sendo declarada deve conter o endereço da origem virtual da matriz (v. 7.9), no caso de ela aplicar-se a uma variável e não a uma declaração de tipo. Esse endereço, os limites dos índices e o tamanho dos elementos dado pelo tipo dos mesmos servem para gerar o código de cálculo do endereço de cada elemento da matriz referenciado no programa-objeto.

Note-se que a TABARR é muito ineficiente no uso do espaço. De fato, cada um de seus elementos contém um espaço vazio se o número de dimensões do mesmo não chegar a MAXDIM. Uma solução para contornar esse problema seria declarar simplesmente um vetor de inteiros; cada matriz seria descrita por um grupo de m + 3 elementos desse vetor; o primeiro elemento de cada grupo conteria o número m de índices (dimensões); a seguir viriam m elementos com os índices para a TIR, o índice do tipo na TIR e, finalmente, o espaço total requerido.

Se o tipo dos elementos da matriz não existir ainda na TIR, como por exemplo um novo tipo "array", ele é colocado a partir do elemento seguinte na TIR com branco no campo IDROT e NOIDEN no campo CLASSE desta tabela, quando for o caso.

Por exemplo a declaração

type T1: **array** [−1..10] **of array** [(FRACO, FORTE)] **of** COR

geraria as entradas seguintes nas tabelas afetadas (consideraremos a TABARR comprimida):

TIR[q]: 'T1', TYIDEN, ARRAYT, r, 24
TIR[q+1]: 'b', NOIDEN, SUBRGT, t, 1
TIR[q+2]: 'b', NOIDEN, ARRAYT, r+4, 2
TIR[q+3]: 'b', NOIDEN, SEQT, 2, 1
TIR[q+4]: FRACO, COIDEN, SCIDT, 0, 0
TIR[q+5]: FORTE, COIDEN, SCIDT, 1, 0
TABARR[r]: 1, q+1, q+2, 24
TABARR[r+4]: 1, q+3, j, 2
TABSRG[t]: INTSR, u, v
TCN[u]: −1, n_1
TCN[v]: 10, n_2

Nesse exemplo, consideramos já existente a declaração de COR vista em (d) acima; n_1 e n_2 indicam os endereços das constantes −1 e 10 no programa-objeto.

Note-se que as informações sobre o tipo **array** armazenadas na TIR e na TABARR constituem uma estrutura de dados em forma de árvore. A raiz desta está na TIR com tipo ARRAYT e contém n+1 "filhos", onde n é o número de dimensões, todas armazenadas na TIR, apontadas indiretamente através da TABARR. Os n primeiros "filhos" indicam os tipos de cada índice, em geral apenas um intervalo de valores (SUBRGT). O último "filho" aponta para o tipo dos elementos da matriz. Se este for do tipo **array** ele constituirá a raiz de uma subárvore da primeira, com as mesmas características, como no exemplo visto.

i) RECT — tipo registro ("record type"), declarado em "type" com **record**. Aplica-se às classes FIIDEN, NOIDEN, TYIDEN, VAIDEN. Associada a esse tipo há a tabela descritora TABREC, com a declaração

```
var TABREC: array [1..MAXREC] of
                record NUNCAM: 1..MAXCAM;
                       INDTIR: array [1..MAXCAM] of 1..MAXTIR;
                       ESPTOT: integer
                end;
```

onde NUMCAM indica o número de campos do registro; INDTIR contém uma seqüência de índices para a TIR, localizando as entradas onde estão o identificador e o tipo de cada campo, na ordem de seu aparecimento na declaração do registro; em ESPTOT colocamos o espaço de memória total necessário para armazenar um registro desse tipo.

Nas entradas da TIR correspondentes a cada campo do registro, colocamos em ENDOBJ o endereço do campo relativo ao início do registro.

Por motivos didáticos, essa tabela foi, como a TABARR de (h), declarada de modo ineficiente; sua compressão é um pouco complexa, pois deve levar em conta o fato de que um campo pode ser do tipo **record**; neste caso, INDTIR deve também assinalar esse fato (p. ex., com um valor negativo), e a próxima entrada de INDTIR deve apontar para a entrada do campo seguinte, pulando as entradas do **record** embutido.

Por exemplo, a declaração:

```
type T2: record C1, C2: COR;
                C3: T1;
                C4: Bool
         end;
```

poderia gerar as seguintes entradas na TIR e na TABREC:

TIR[v]: 'T2', TYIDEN, RECT, w, 27
TIR[v+1]: 'C1', FIIDEN, TYIDT, j, 0
TIR[v+2]: 'C2', FIIDEN, TYIDT, j, 1
TIR[v+3]: 'C3', FIIDEN, TYIDT, q, 2

TIR[v+4]: 'C4', FIIDEN, TYIDT, 4, 26
TABREC[w]: 4, v+1, v+2, v+3, v+4, 27

Nesse exemplo supusemos COR e T1 como declarados em (d) e (h); lembremos que Bool está na entrada 4 da TIR — cf. (d) acima.

Note-se que há necessidade da TABREC, pois os campos de registro podem não ocorrer contiguamente na TIR, o que acontece se, por exemplo, um dos campos for do tipo registro. Como no caso do tipo **array**, temos também aqui uma estrutura em forma de árvore.

A declaração **record** pode ter uma "parte variante" ("variant part"). Por meio dela um registro pode ter formato variável, ou melhor, vários formatos diferentes. Segundo a sintaxe da PASCAL /J-W 74/, essa parte que varia é precedida da palavra reservada **case** e sempre vem no fim do registro, conforme podemos ver no grafo do apêndice I, em "filist". Depois de **case** segue-se um campo com um identificador que chamaremos de 'pseudo-seletor' (e que pode ser omitido). Depois de **of** seguem-se os campos variantes, precedidos de uma constante seletora. Essa constante é totalmente supérflua, servindo apenas como comentário ou, no melhor dos casos, para uma verificação pelo compilador, testando se seu tipo concorda com o tipo do identificador do pseudo-seletor. O não-terminal "filist" depois da constante seletora mostra que podemos tratar os campos variantes como compondo um registro entre '(' e ')'. Na verdade consideraremos essa parte como uma continuação do registro anterior. Qualquer referência aos campos variantes no programa-fonte é feita através de seus próprios identificadores, e não através das constantes seletoras. Assim, estas podem ser desprezadas, não havendo necessidade de colocá-las na TIR se não se deseja verificar se o seu tipo combina com o do pseudo-seletor. Trataremos, portanto, os campos de pseudo-seletor e de "filist" entre '(' e ')' como continuação dos campos da parte não-variante. A única diferença é que o endereço no programa-objeto do primeiro campo de cada alternativa da parte variante deve ser o mesmo. Dessa maneira, as alternativas compartilham a mesma área de memória em tempo-objeto; deve-se reservar uma área total para o registro correspondendo à parte fixa acrescida do tamanho da maior alternativa da parte variante.

Por exemplo, a declaração

```
var A: record
         A1:  P1;
         case A2: P2 of
                  C21: (A3:P3;
                        A4:P4);
                  C22: (A5:P5)
       end;
```

geraria as seguintes entradas na TIR e na TABREC:
TIR[i]: 'A', VAIDEN, RECT, j, e_1
TIR[i+1]: 'A1', FIIDEN, TYIDT, i_{p1}, 0
TIR[i+2]: 'A2', FIIDEN, TYIDT, i_{p2}, t_{p1}
TIR[i+3]: 'A3', FIIDEN, TYIDT, i_{p3}, $t_{p1}+t_{p2}$
TIR[i+4]: 'A4', FIIDEN, TYIDT, i_{p4}, $t_{p1}+t_{p2}+t_{p3}$
TIR[i+5]: 'A5', FIIDEN, TYIDT, i_{p5}, $t_{p1}+t_{p2}$
TABREC[j]: 5, i+1, i+2, i+3, i+4, i+5, t
onde t_j é o espaço ocupado pelo tipo p_j; $t = t_{p1}+t_{p2}+\max(t_{p3}+t_{p4}, t_{p5})$.

j) FILET — tipo arquivo ("file type"), declarado em "type" com **file.** Aplica-se às classes TYIDEN e VAIDEN. Este caso é tratado de maneira análoga à SETT conforme (e) acima. A diferença é que agora depois de **of** temos um "type" e não um "sitype", isto é, qualquer dos tipos já vistos pode ser apontado pelo conteúdo do campo INDTT da entrada gerada na TIR. A TIR deve ser inicializada com os arquivos padrões INPUT e OUTPUT /J-W 74/.

k) PARAM e PARAV — PARAM indica parâmetro de procedimentos e funções, declarado como IDEN em "palist", não precedido de **func, proc** e **var.** Aplica-se às classes FUIDEN, VAI-

DEN e PRIDEN, respectivamente. PARAV indica parâmetro de procedimentos e funções declarado como IDEN em "palist", *não* precedido das declarações **func, proc** e **var.** Aplica-se à classe VAIDEN.

Essa distinção de tipos de identificadores de parâmetros de classe VAIDEN é essencial para a geração de código para procedimentos. De fato, um parâmetro formal VAIDEN com tipo PARAM deve ser armazenado no programa-objeto de tal modo, que qualquer alteração de seu valor dentro do procedimento deve provocar uma alteração do parâmetro atual, isto é, o da seqüência de chamada. Por outro lado, segundo a definição da PASCAL dada em seu manual /J-W 74/, dentro de um procedimento podem haver alterações de valor de seus parâmetros aqui classificados como VAIDEN e de tipo PARAV, mas essas alterações não devem refletir-se nos parâmetros atuais correspondentes, cujo valor deve permanecer inalterado. Daí termos introduzido essa definição, que será usada no capítulo 9 para a geração de código dos vários tipos de parâmetro. Note-se que não pudemos considerar esses últimos parâmetros como tendo classe COIDEN, pois nesse caso eles não poderiam aparecer no lado esquerdo de atribuições, que começam com VAIDEN — como se pode ver no subgrafo de "statm", ramo VAIDEN.

Examinando o grafo sintático da PASCAL (apêndice I), podemos notar que os parâmetros formais (isto é, os que se encontram entre '(' e ')' na declaração do procedimento, em "palist") são declarados sempre como sendo do tipo TYIDEN, isto é, um identificador de tipo — cf (b) acima. Com isso o armazenamento dos parâmetros e seus tipos na TIR fica muito simplificado. Colocaremos o identificador do procedimento ou função na próxima entrada disponível da TIR, e cada parâmetro numa entrada seguinte na ordem de aparecimento; todos têm **tipo PARAM ou PARAV contendo em INDTT um apontador para a entrada da TIR onde está o identificador de tipo referenciado pelo parâmetro. No caso de função, a primeira entrada, correspondente ao identificador da função, deve conter TYIDT em TIPO e em INDTT um apontador para o tipo do resultado, como declarado após ':' de func em "block".**

Por exemplo, a declaração

func F (**func** F1: T1; **var** V1,V2:T2; C:T3; **proc** P): T4

geraria as seguintes entradas:

TIR[i]: 'F', FUIDEN, TYIDT, i_{t4}, n
TIR[i+1]: 'F1', FUIDEN, PARAM, i_{t1}, 0
TIR[i+2]: 'V1', VAIDEN, PARAM, i_{t2}, 1
TIR[i+3]: 'V2', VAIDEN, PARAM, i_{t2}, 2
TIR[i+4]: 'C', VAIDEN, PARAV, i_{t3}, 3
TIR[i+5]: 'P', PRIDEN, PARAM, 0, 4

onde n é o endereço do início do código da função no programa-objeto. Como veremos no capítulo 9, cada parâmetro ocupa uma palavra no programa-objeto. Seu endereço é relativo à base dos dados da função (ou procedimento).

Na TIR a lista de parâmetros é encerrada pela primeira entrada que não é parâmetro, isto é, não contém PARAM ou PARAV em TIPO.

ℓ) LABELT — rótulo, declarado com **label** em "block". Aplica-se à classe LABEL. No campo IDROT ocorre o código interno (binário) do inteiro correspondente ao rótulo; em ENDOBJ, o endereço da primeira instrução gerada pelo comando ("statm") do programa-fonte que é precedido pelo rótulo.

Note-se que as tabelas descritoras dos tipos SUBRGT, ARRAYT e RECT podem ser todas reunidas em uma só tabela, desde que se codifique em inteiros os tipos referidos pela TABSRG.

Para não sobrecarregar o texto, deixamos de examinar a declaração **packed** de "type", pois esta corresponde simplesmente a uma marca adicional que indica se o tipo deve ser compactado; essa marca pode ser implementada por meio de mais um campo na TIR ou por meio de tipos suplementares, como PARRAY, PFILE, PSET, PREC.

Até aqui vimos como as informações sobre as declarações de variáveis, tipos etc. estão armazenadas nas tabelas de símbolos. Devemos, agora, descrever como o compilador introduz essas informações nas tabelas. Para isso, introduziremos uma estrutura adicional no compilador, que ocorre em geral subdividida em várias outras estruturas; a unificação aqui exposta permite uma compreensão mais simples de todo o processo de análise de contexto e de geração de código. Trata-se da "pilha semântica" Em seguida, abordaremos o problema da inserção das informações e, finalmente, descreveremos como manipular a TIR levando em conta a "estrutura de blocos" da linguagem PASCAL.

6.4 A PILHA "SEMÂNTICA"

Examinando-se a gramática ou grafo da linguagem PASCAL vemos que várias de suas estruturas são definidas recursivamente, como nos exemplos "type", em que ocorre o próprio não-terminal "type" nos ramos de **array, file** e, indiretamente, em "filist" de **record**; "statm", onde este não-terminal ocorre nos ramos de **begin, if** etc. Isso significa que, no caso de declaração de um tipo, é necessário interromper esse processo para gerar-se as informações referentes a um outro tipo interior ao primeiro, como no caso de registros que contêm registros como campos. Esses fatos sugerem o uso de uma estrutura de pilha para guardar certas informações quando houver uma interrupção na geração das informações de um determinado objeto. Como essa interrupção é devida à estrutura sintática reconhecida pelo analisador sintático, vamos introduzir uma pilha, que denominaremos de *pilha "semântica"*, abreviada por PSE, e que é "paralela" à pilha sintática (PS) — já que esta exprime justamente as interrupções sintáticas do reconhecimento de um objeto. Esse paralelismo significará que, para cada célula da PS, existirá uma célula da PSE e, sendo cada pilha implementada por meio de um vetor, um único índice apontará para o topo de ambas: TOPPS, como introduzido em 5.7. Note-se que, sendo o analisador do tipo ESLL(1), a PS é totalmente dispensável. Independentemente de sua implementação ou não, suporemos que ela exista, pois seus elementos servirão de referência para os elementos da PSE. Esta é manipulada tanto pelas rotinas "semânticas" (v. 6.2) como pelo AS ANSIN (v. 5.6.4 e 5.7) através de seu procedimento DESEMPILHA na atualização de TOPPS e nos pontos marcados com (+) e (++), analogamente à manipulação da PS. Quando um símbolo terminal diferente de NUMB é reconhecido pelo ANSIN, o parâmetro p_1 do analisador léxico (AL) (v. 3.4) é empilhado na PS; neste momento, são empilhadas duas informações na PSE, cujas células devem ter dois campos, TAB e IND como declarado adiante. Em TAB é colocado o nome da tabela onde é ou será armazenado o valor de p_2 do AL; em IND é colocado um índice para a entrada dessa tabela onde se encontra armazenado o símbolo reconhecido pelo AL. No caso de se ter um identificador de constante, por exemplo MAX declarado como **const** MAX = 50, e reconhecido como COIDEN, deve-se empilhar na PSE o nome e a entrada da tabela onde se encontra a constante final, isto é, TCN ou TCA; essa informação é dada pelo campo TIPO da TIR, que contém nesses casos INTCTT ou STRCTT. Temos as seguintes possibilidades para o campo TAB: TIR (no caso de p_1 conter IDEN, VAIDEN, TYIDEN etc.); TCN (no caso de NUMB); TCA (no caso de STRING); TABT — tabela de símbolos reservados (no caso de um desses símbolos — v. 5.6.2); TEMP — tabela de valores temporários a ser descrita mais adiante sob as formas ITEMP, BTEMP, PTEMP, STEMP. O valor de IND é obtido quando da inserção ou busca do símbolo lido na tabela correspondente, de maneira análoga à inserção de um IDEN na TIR, como descrito em 6.2. Além dessas tabelas teremos índices para a própria PSE, indicados por SIPSE, e endereços do programa-objeto, SOBJ.

A declaração da PSE pode ser então

```
var PSE: array [1..MAXPS] of
    record TAB: (STIR, STCN, STCA, STABT, ITEMP, BTEMP, PTEMP, STEMP, SIPSE,
                                                                    SOBJ);
           IND: integer end;
```

Note-se que na realidade não é necessário declarar cada célula da PSE com dois campos, o que adotamos por motivos didáticos. Pode-se comprimir ambos em um só campo numérico adicionando-se a IND uma constante diferente para cada tabela, por exemplo de 1000 a 10000.

O caso de NUMB é tratado por rotinas "semânticas" que são chamadas em cada nó desse terminal para diferenciar-se rótulos ("labels") de constantes numéricas.

Ao ser reconhecido um nó não-terminal, a rotina "semântica" correspondente só pode ser chamada após a execução da rotina DESEMPILHA (v. 5.7 e 6.2). Esta só altera a PS e o valor de TOPPS. Qualquer alteração da PSE deve ser feita pela referida rotina semântica, devendo-se tomar cuidado com a posição da pilha onde será feita a alteração. Esta ocorre em geral apenas na célula "paralela" à do não-terminal que acabou de ser reconhecido.

Por exemplo, suponhamos uma certa situação dessas pilhas representadas na fig. 6.1 (os índices das tabelas são arbitrários). Na fig. 6.2 damos a situação das pilhas após o reconhecimento de "siexpr".

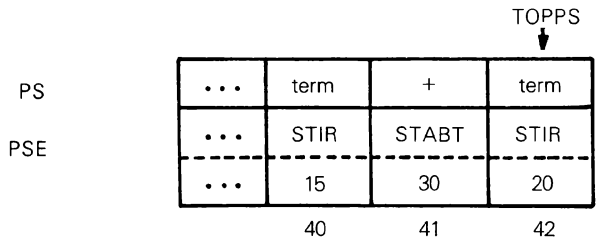


Fig. 6.1

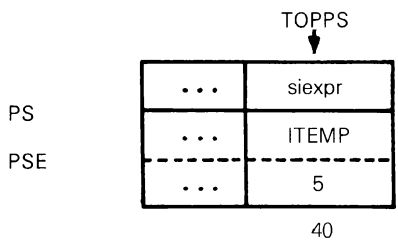


Fig. 6.2

O resultado da adição é colocado em um espaço de memória usado para valores temporários; o endereço será, nesse exemplo, o do sexto temporário do programa-objeto, e o tipo será inteiro, como indicado por ITEMP; daí os conteúdos de PSE[40] da fig. 6.2.

A rotina "semântica" 152 do nó "term" no subgrafo de "siexpr" (v. apêndice I) é que coloca em PSE[40] os valores ITEMP e 5, substituindo os valores STIR e 15. Na verdade, deve ter sido gerado código para essa soma, o que foi também feito por essa rotina. Maiores detalhes serão vistos no capítulo 7. Posteriormente, ao concluir o reconhecimento de "siexpr", o AS altera o conteúdo de PS[40], obtendo-se a situação da fig. 6.2.

6.5 INTRODUÇÃO DE INFORMAÇÕES NAS TABELAS DE SÍMBOLOS

Vamos aqui descrever, sucintamente, o processo de inserção de informações nas tabelas de símbolos para cada classe e tipo. Inicialmente daremos um exemplo ilustrativo, para depois abordarmos cada declaração em separado.

Seja a declaração

var A,B,C: (AZUL,VERDE,ROSA)

Estamos em "block" (v. grafo no apêndice I) e entramos no ramo **var**, reconhecendo este símbolo. Vamos descrever as ações executadas pelas rotinas "semânticas" correspondentes a cada nó onde existe uma referência explícita a uma delas:

- Primeiro nó IDEN, RS_{60}. Como vimos em 6.2, o AS, ao receber um IDEN do AL (que corresponde aqui a'A'), produz a introdução desse identificador na TIR, na próxima entrada disponível, que suporemos ter índice j; além disso coloca o par STIR, j no topo da PSE, paralelo a IDEN da PS. Ao ser executada, a RS_{60} também introduz VAIDEN no campo CLASSE, e não coloca nada nos campos TIPO, INDTT e ENDOBJ. Em seguida, ela guarda em uma variável PRIID o valor de TOPPS, indicando a posição de PSE onde foi colocado o primeiro identificador na lista sendo declarada.
- Nó ','. São colocados os valores apropriados na PS e na PSE.
- Segundo nó IDEN, RS_{65}. Idem ao primeiro nó IDEN, agora colocando 'B' em TIR[j + 1] e sem alterar PRIID.
- Nó ','.
- Segundo nó IDEN, RS_{65}. Idem à passagem pelo segundo nó IDEN anterior, agora colocando 'C' em TIR[j + 2].
- Nó ':', RS_{102}. Coloca em PSE[TOPPS] o par SIPSE, v, onde v é o valor de PRIID. Em seguida chama RS_2 como visto em 6.2.

Na fig. 6.3 mostramos a configuração das pilhas nesse momento; as entradas de **var** e ',' na TABT são indicadas por i_{var} e $i_,$.

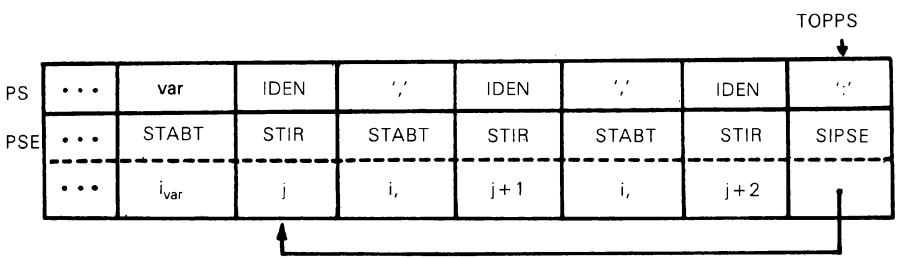


Fig. 6.3.

Nesse ponto, atinge-se o sucessor de ':', devendo-se desviar para o subgrafo de "type". Neste, acontece um novo desvio, para o subgrafo de "sitype", onde são reconhecidos os seguintes nós:

- Nó '(', RS_{101}. Essa rotina introduz uma entrada na próxima posição disponível da TIR, da seguinte maneira:
  TIR[j + 3]: 'b', NOIDEN, SEQT, 0, 1
  Em seguida, coloca no topo da PSE o par STIR, j + 3 indicando o índice do tipo da TIR; uma variável IT é inicializada com o valor j + 3.
- Nó IDEN, RS_{11}. Introduz na TIR
  TIR[j + 4]: 'AZUL', COIDEN, SCIDT, 0, 0
  A seguir faz TIR[IT].INDTT: = TIR[IT].INDTT + 1
- Nó ','
- Nó IDEN, RS_{11}. Introduz na TIR
  TIR[j + 5]: 'VERDE', COIDEN, SCIDT, 1, 0
  e faz o incremento em TIR[IT].INDTT como acima.
- Nó ','.
- Nó IDEN, R_{11}. Introduz na TIR

TIR[j+6]: 'ROSA', COIDEN, SCIDT, 2, 0
e faz o incremento mencionado; TIR[j+3].INDTT fica portanto com o valor final 3, indicando o número de elementos da seqüência.

— Nó ')', RS_2. A execução dessa rotina (v. 6.2) não altera a PSE.

Na fig. 6.4 mostramos as 7 células superiores das pilhas nessa situação:

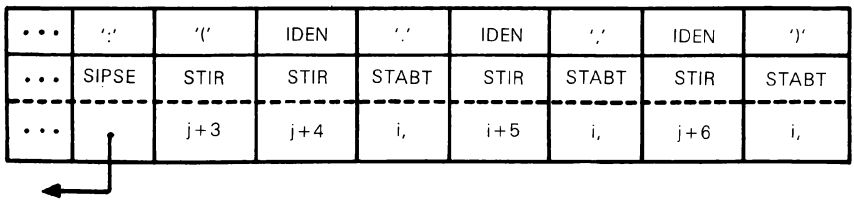


Fig. 6.4

Neste momento, um "sitype" acabou de ser reconhecido. Todo o seu lado direito será substituído nas células superiores da PS pelo não-terminal "sitype". A configuração das 3 células superiores será a da fig. 6.5.

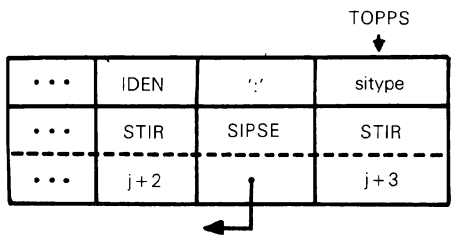


Fig. 6.5

Note-se que, na fig. 6.5, em paralelo a "sitype" temos na PSE sua localização na TIR, indicada por j+3. Podemos dizer que essa célula da PSE dá o significado, isto é, a "semântica" do não-terminal "sitype", mostrando sua definição na TIR.

Em seguida, retorna-se ao subgrafo de "type" tendo-se reconhecido o seu último nó, "sitype", onde ocorrera o desvio, e onde não há rotina "semântica" a ser executada. Como esse nó não tem sucessor, reconhecemos um "type"; a configuração das pilhas é a mesma da fig. 6.5, trocando-se apenas "sitype" por "type", retornando-se ao subgrafo de "block", ramo **var**, nó "type".

— Nó "type", RS_{69}. Essa rotina preenche os campos TIPO e INDTT de TIR[j], TIR[j+1] e TIR[j+2]. Nesse ponto da análise isso já pode ser feito pois já conhecemos o tipo desses três campos, que é especificado pelo par STIR,j+3 do topo da PSE (e que dá significado "semântico" ao "type" do topo da PS). A RS_{69} toma o conteúdo da PSE na posição TOPPS-1, onde se encontra o par SIPSE,p; p é o índice da PSE onde se encontra o índice j do primeiro identificador da lista declarada (v. fig. 6.3). Obtido o j, faz-se

TIR[j].TIPO: = TIR[j+3].TIPO;
TIR[j].INDTT: = TIR[j+3].INDTT;
TIR[j].ENDOBJ: = q;
q: = q+ TIR[j+3].ENDOBJ

onde q contém o endereço da próxima palavra a ser gerada no programa-objeto na área de dados.

Essa seqüência de comandos é repetida para todos os identificadores da lista. Suas localizações na TIR são obtidas no campo IND da PSE, tomando-se duas células na pilha após a célula do último identificador tratado. Esse processo termina quando se atinge a célula da PSE de índice igual ao conteúdo de TOPPS, exclusive.

Uma outra maneira de guardar-se um apontador para o primeiro identificador da lista sendo declarado seria a de guardar em PRIID, em lugar de TOPPS, o índice para a entrada da TIR correspondente ao primeiro identificador da lista. Os outros identificadores seriam localizados nas posições crescentes da TIR, até que se chegue ao topo dessa tabela ou a uma entrada com campo TIPO de valor já definido.

Note-se que, na implementação desse exemplo, a entrada TIR[j + 3] fica posteriormente supérflua. Poderíamos ter colocado as informações de seus campos TIPO, INDTT e ENDOBJ nos campos correspondentes da última variável sendo declarada, evitando assim a entrada supérflua. Adotamos a solução aqui apresentada por motivos didáticos.

A seguir, abordaremos sucintamente as ações que devem ser realizadas para se inserir nas tabelas de símbolos as entradas correspondentes às várias declarações. Vamos examinar cada caso a partir dos subgrafos dos não-terminais envolvidos (apêndice I).

a) "factor"

Como vimos em 6.4, logo depois de reconhecido um item léxico pelo AL, é empilhado na PSE o par ST,j correspondendo à tabela T e o índice da entrada de T onde se encontra o símbolo, com uma exceção: NUMB. Quando o AL reconhece um NUMB, não pode distinguir entre sua utilização como constante numérica ou como rótulo ("label"). O nó contendo NUMB em "factor" indica a utilização desse item léxico como constante numérica. Assim, a RS_{73} busca na TCN uma entrada com valor igual ao do parâmetro p_3 retornado pelo AL (v. 3.4), inserindo-o na mesma se não for encontrado; a seguir, empilha o par STCN,j na PSE, em paralelo a NUMB. No caso STRING, COIDEN, VAIDEN e FUIDEN não há nada a fazer; o par correspondente já deve estar na PSE no devido lugar, sendo que em COIDEN deve-se já ter o par ST,j correspondendo à tabela T onde se encontra o valor da constante. A constante tipo apontador **nil** é tratada em /J-W 74/ e conseqüentemente neste texto como palavra reservada. A RS_{110} troca o par STABT,j do topo da PSE por STIR,3 indicando a localização de **nil** na TIR (v. 6.3c). Dessa maneira uniformizamos o tratamento de variáveis, constantes e funções dentro de expressões.

b) "const"

As mesmas considerações feitas em (a) valem também neste caso. Além disso, é preciso tratar do caso dos sinais '+' e '−'. Para simplificar o problema, vamos introduzir na TCN novas constantes numéricas se aparecerem com sinal negativo. Assim, a RS_{72} verifica se ocorreu um sinal '−', testando a célula TOPPS-1 da PSE (TOPPS contém o índice da célula do topo); se ela contiver o par STABT,i verifica-se se TABT[i] = '−'. Neste caso, troca-se o sinal da constante que ainda está no parâmetro p_3 do analisador léxico (v. 3.4) e somente depois disso é que a constante é procurada na TCN e aí inserida se não for encontrada. O par STCN,i onde i é o índice da entrada da constante na TCN, é colocado na PSE em paralelo a '+' ou '−', isto é, em PSE[TOPPS-1] se um desses símbolos tiver ocorrido, caso contrário é colocado em paralelo a NUMB em PSE[TOPPS]. No caso de COIDEN, a RS_{71} verifica se se trata de constante numérica, testando se PSE[TOPPS].TAB = STCN. Neste caso, é chamada em seguida a RS_{72}. Na verdade, é necessário eventualmente percorrer outros identificadores de constantes da TIR, até se chegar em tipo INTCTT, STRCTT (v. 6.3f) ou SCIDT (v. 6.3d). Note-se que ao terminar o reconhecimento de "const", em paralelo a esse não-terminal estará na PSE um apontador para a tabela que contém a constante.

c) "sitype"

c_1) No caso de TYIDEN, já temos na PSE, em paralelo a esse símbolo, o par STIR,i com a localização i do tipo da TIR. Assim, nada há a fazer, pois ao se reconhecer "sitype" o par passa a ficar paralelo a este não-terminal, dando o significado "semântico" a ele.

c_2) No caso de seqüência de constantes simbólicas, as rotinas semânticas executam as funções já vistas no exemplo abordado no começo deste item. Suponhamos que se gere para cada seqüência uma nova entrada i na TIR, mesmo se ela for supérflua, já que com isso simplificamos o tratamento. a eliminação dessa entrada supérflua é muito simples de ser conseguida, nessa fase ou posteriormente. A RS_{101} gera portanto

TIR[i]:'b', NOIDEN, SEQT, 0, 1

supondo-se que os objetos desse tipo ocupem sempre uma palavra de memória, e guarda em PSE[TOPPS], portanto na célula paralela a '(', o par STIR,i. Em seguida, a RS_{101} chama a RS_1 (v. 6.2). O índice i é guardado para incrementar-se o campo INDTT para cada novo identificador, sob controle da RS_{11}. Esta, a cada chamada, introduz um novo identificador na TIR com classe COIDEN e tipo SCIDT, e incrementa de 1 o campo TIR[i].INDTT, como vimos no exemplo mencionado. Ao terminar o reconhecimento de "sitype", teremos o par STIR,i na célula paralela a este não-terminal na PSE.

c_3) No caso de intervalo ("subrange"), a RS_{103} deve verificar inicialmente se o tipo das duas constantes é o mesmo, o que é verdade se PSE[TOPPS-2].TAB = PSE[TOPPS].TAB. Isso vale também para casos como 1..MAX pois, como vimos em 6.4, em paralelo ao COIDEN de MAX é colocada na PSE a entrada para a TCN onde se encontra o valor numérico da constante. No entanto, se os dois campos TAB contiverem o valor STIR e tratar-se de elementos de seqüências de constantes simbólicas, é necessário verificar se a seqüência é a mesma. Isso pode ser feito localizando-se na TIR a entrada de tipo SEQT, que encabeça a seqüência onde está cada constante. Para isso, basta subtrair do índice de cada constante o conteúdo de seu campo INDTT, acrescido de uma unidade. Depois da verificação, é gerada uma entrada j na TABSRG com a indicação da tabela onde estão as constantes-limites e os índices para a mesma. É gerada então, uma entrada na TIR,

TIR[i]:'b', NOIDEN,SUBRGT, j, 1

e coloca-se o par STIR,i em PSE[TOPPS-2] (isto é, na célula paralela à primeira constante do intervalo na PS). Esse será posteriormente o conteúdo da célula paralela a "sitype", quando este acabar de ser reconhecido.

Vemos, portanto, que o conteúdo "semântico" de "sitype" será sempre um par STIR,i indicando a entrada na TIR do tipo que acabou de ser declarado.

Note-se que, novamente, gerou-se uma entrada na TIR que posteriormente poderá ser supérflua.

d) "type"

Ao fim da compilação de um "type" as suas informações estarão armazenadas nas tabelas de símbolos e, em paralelo a esse não-terminal, ter-se-á sempre na PSE um par STIR,i indicando o índice i da entrada do tipo na TIR.

d_1) No caso de "↑", temos um tipo apontador ("pointer"). É criada pela RS_{111} uma entrada na TIR da seguinte maneira:

TIR[i]: 'b', NOIDEN, POINTT, j, 1

onde j é o conteúdo de PSE[TOPPS].IND, apontando para o identificador de classe TYIDEN da TIR. A seguir, é colocado em PSE[TOPPS-1] o par STIR,i que posteriormente ficará paralelo ao não-terminal "type" sendo reconhecido.

d_2) No caso de **set**, ao se executar a RS_{112}, tem-se em PSE[TOPPS], paralelamente a "sitype", o par STIR,j proveniente da declaração desse tipo. Assim, basta a RS_{112} introduzir na TIR uma nova entrada.

TIR[i]:'b', NOIDEN, SETT, j, 1

Para dar significado "semântico" ao "type", o par STIR,i é atribuído à célula PSE[TOPPS-3] no caso de ter sido reconhecido um **packed** ou à célula PSE[TOPPS-2] em ca-

so contrário. Essa decisão pode ser baseada em consulta à PSE[TOPPS-3] testando-se se aí há um par STABT,j apontando ou não para **packed** em TABT.

d_3) No caso de **file**, procede-se da mesma maneira que para **set**.

d_4) No caso "sitype", nada há a fazer; "type" "herdará" o par STIR,i de "sitype".

d_5) No caso de **array**, a RS_{114} gera as entradas

TIR [i]: 'b', NOIDEN, ARRAYT, r, 0 e

TABARR[r]: 0 (vamos supor essa tabela comprimida, conforme 6.4h) guardando r na variável auxiliar RT.

A RS_{115} é executada para cada tipo de índices "sitype"; em PSE[TOPPS] temos o par STIR,j indicando onde se encontra a definição do "sitype". Se TIR[j].TIPO = TYIDT, fazemos j: = TIR[j].INDTT, e passamos a verificar novamente TIR[j], até que TIR[j].TIPO seja igual a SEQT ou SUBRGT. Nesses casos fazemos

TABARR[RT]: = TABARR[RT] + 1; r: = r + 1; TABARR[r]: = j

com isso incrementando o número de dimensões da matriz e colocando em TABARR a localização da entrada, na TIR, do tipo desse índice.

Esse processo é repetido para cada novo índice da matriz, isto é, para cada chamada de RS_{115}.

A RS_{116} corresponde ao símbolo **of**, sua ação é fazer r: = r + 1, guardar o par STIR,i em PSE[TOPPS-1] e o par SIPSE,r em PSE[TOPPS] (SIPSE é aqui arbitrário). Com isso, empilhamos a localização da declaração desse **array** na TIR e a posição da TABARR na qual deverá ser colocado o índice para a TIR onde ficará o tipo dos elementos da matriz, ainda não reconhecido; em TABARR[r + 1], ficará ainda o espaço total de memória a ser ocupado pelo **array**, o que também depende do tipo dos elementos. (Poderíamos ter calculado o número total de elementos da matriz, à medida que fosse conhecida a declaração de cada índice da mesma através da RS_{115}.) A RS_{116} ainda faz r: = r + 2, atualizando r para o caso do tipo dos elementos da matriz ser também do tipo **array**, como no exemplo da 6.3h.

A seguir, desviamos para "type"; ao retornar, será executada a RS_{117}. A esta altura, em paralelo com o "type" do topo da PS, temos na PSE o par STIR,j indicando a localização j do tipo da TIR. A RS_{117} retira r de PSE[TOPPS-1] e i de PSE[TOPPS-2], fazendo

TABARR[r]: = j; TABARR[r + 1]: = s; TIR[i].ENDOBJ: = s.

onde s é o espaço total ocupado pela matriz, o que pode ser calculado percorrendo-se os índices e o espaço requerido pelo tipo dos elementos, que está em TIR[j].

Note-se a necessidade do empilhamento de r e de i e como foi empregada a PSE para isso. É interessante observar que essa construção é recursiva, pois dentro do "type" dos elementos da matriz podemos ter um tipo qualquer; o importante é que, depois deste ser reconhecido, seu índice na TIR seja guardado na PSE.

d_6) No caso de **record**, a RS_{121} gera

TIR[i]: 'b', NOIDEN, RECT, w, 0

e

TABREC[w]:0 (vamos supor que a TABREC é comprimida, conforme 6.3i) e é colocado o par STIR,i em PSE[TOPPS] (paralelo a **record**). Em seguida a RS_{121} chama a RS_1 (v. 6.2).

Quando a RS_{122} for executada, todos os campos, declarados em "filist", já foram introduzidos na TIR, tendo-se completado as entradas na TABREC. Resta agora preencher o campo do espaço total ocupado pelo tipo, em TIR[i].ENDOBJ, que é encontrado em TABREC[w + n + 1] onde n é o número de campos. Portanto, devemos programar as rotinas "semânticas" de "filist" de tal modo que n seja colocado em PSE[TOPPS].IND em paralelo a esse não-terminal, e que o espaço total seja colocado adequadamente na TABREC. Note-se que em PSE[TOPPS-1], isto é, em paralelo a **record**, já encontra-se o par STIR,i que indica a localização do tipo na TIR.

e) "filist"

Há dois nós que produzem um desvio para "filist", um em "type" e outro no próprio

"filist". O primeiro corresponde ao início de um registro; o segundo à parte variante de um registro (v. 6.3i).

e_1) Parte fixa do registro. Nesta parte a RS_{30} inclui cada novo identificador na TIR com classe FIIDEN e em TABREC o índice de sua entrada na TIR.

O campo TIPO de cada IDEN fica vazio, sendo preenchido posteriormente pela RS_{130}, colocando-se nele TYIDIT e o índice do tipo de "type" em INDIT. Esse preenchimento pode ser feito como no caso de declaração de variáveis visto no exemplo do início deste item. Vamos aqui ilustrar uma outra solução, que poderia também ser empregada naquele caso. A RS_{130} deve conhecer o índice da TIR do primeiro IDEN da lista; com essa informação ela pode preencher o campo TIPO dessa entrada e das seguintes, até encontrar o fim da TIR ou até que apareça uma entrada com campo TIPO diferente de vazio (caso de "type" ter introduzido novas entradas na TIR). O índice do primeiro IDEN da "filist" pode ser guardado em uma variável global, pela RS_{121} de **record** em "type" ou pela RS_{133} de ':' na parte variante (**case**). Ao preencher o tipo de cada IDEN, a RS_{130} também vai preenchendo o campo de endereço relativo ao programa-objeto, a partir do espaço ocupado pelo tipo do campo anterior.

Após o encerramento da parte fixa, se não houver parte variante (que inicia com **case**) será atingido o λ-nó da RS_{131}. (Esse λ-nó foi colocado no subgrafo somente para execução da RS_{131}.) Esta rotina encerra a declaração do registro incluindo na TIR e na TABREC as informações que especificamos em (d_6) acima.

e_2) Parte variante do registro. Neste caso é eventualmente executada a RS_{30} para IDEN, e depois a RS_{132} para TYIDEN. Esta última rotina deve descobrir se a RS_{30} foi ou não executada nesse ramo do subgrafo; isso é fácil de realizar: basta verificar se em PSE[TOPPS-1] está um par STABT,p, onde p é a entrada de ':' na TABT. A outra possibilidade é p apontar em TABT para **case**, se não tiver ocorrido o identificador depois deste símbolo. Como vimos em 6.3i, esse IDEN é tratado como um campo qualquer da parte fixa; assim, a RS_{132} deve incluir o índice da TIR, onde está o seu tipo, no campo INDTT do identificador, colocar TYDIT no campo TIPO, colocar em TABREC o índice do IDEN na TIR, e ainda colocar seu endereço relativo ao início do registro no campo ENDOBJ. Em seguida entramos na parte variante propriamente dita. Como vimos em 6.3i, as constantes seletoras que ocorrem em seguida são totalmente supérfluas, podendo-se eventualmente verificar se são do mesmo tipo que o TYIDEN reconhecido antes de **of**. Aqui, nem nos demos ao trabalho de especificar a execução de rotinas "semânticas" para as mesmas. Interessa-nos a geração de informações para "filist" que vem em seguida. A diferença com relação ao tratamento que fizemos para a "filist" proveniente da ocorrência desse não-terminal em "type" seguindo **record** é o fato de que agora cada primeiro campo de todos os "filist" entre '(' e ')' deve começar no mesmo endereço do programa-objeto. Portanto, cada vez que se atinge esse "filist" é necessário inicializar o endereço onde começa seu primeiro campo. Esse endereço precisa ser empilhado, pois dentro de partes variantes podem ocorrer registros com partes fixas e novas variantes, etc. Suponhamos que a variável EPROC contenha o endereço do próximo campo de qualquer registro. A RS_{133} de ':' na parte variante pode guardar o valor de EPROC em PSE[TOPPS].IND, isto é, na célula paralela a esse terminal, colocando um valor arbitrário no campo TAB, por exemplo SOBJ (v. 6.4). A RS_{134} deve comparar o espaço total necessário para o "filist" que acabou de ser compilado com o espaço necessário para o maior "filist" já reconhecido nessa parte variante, guardando o maior dos dois. Com isso, a RS_{135}, executada na saída final do registro, pode colocar na TABREC o valor do espaço necessário a fim de armazenar corretamente no programa-objeto o maior registro. Além disso, a RS_{134} inicializa a variável EPROC com o valor que havia sido guardado na célula da PSE paralela a ':', isto é, em PSE[TOPPS-2]. IND, forçando assim a nova "filist" a começar no mesmo endereço da anterior. A RS_{135} deve ainda introduzir na TIR e na TABREC as informações finais, como a RS_{131} no outro λ-nó, alternativa de **case**.

f) "palist"

Vejamos como são introduzidas as informações sobre parâmetros na TIR, conforme o exposto em 6.3k. A RS_8 do nó '(' inicializa duas variáveis, por exemplo CLAPAR e TIPPAR

com os valores VAIDEN e PARAV, respectivamente. As RS_6 e RS_7 alteram esses valores para FUIDEN, PARAM e VAIDEN, PARAM respectivamente. A RS_5 em IDEN simplesmente copia esses valores nos campos CLASSE e TIPO do identificador que acabou de ser introduzido na TIR, conforme o que vimos em 6.2. Dessa maneira estabelece-se a distinção necessária nesses campos. No caso de **proc**, a RS_{45} introduz a classe PRIDEN e o tipo PARAM. A RS_5 ainda coloca no campo INDTT o valor 0 (zero).

No caso de **func, var** ou sem declaração, há ainda o tipo do parâmetro que deve ser adicionado; esse tipo é sempre um identificador de tipo, como se pode ver no nó TYIDEN. A RS_9 toma o par STIR,i de PSE[TOPPS], e copia o valor i nos campos INDTT de todos os últimos identificadores da TIR, varrendo esta tabela no sentido da última entrada para a primeira, até que apareça uma entrada com campo INDTT diferente de 0 (zero) ou uma entrada com classe PRIDEN; nessa entrada não se deve copiar o valor i.

g) "block"

Em "block" são declarados os identificadores e rótulos, a menos de identificadores de constantes simbólicas e de parâmetros.

g_1) **label**

Rótulos são sempre números inteiros em PASCAL. Devido à sua possível confusão com constantes propriamente ditas, a distinção entre estas e rótulos é feita por meio de rotinas "semânticas". O AS, ao receber do AL um item léxico NUMB, não produz uma busca na TCN, como o faz com IDEN na TIR (v. 6.2) não empilhando na PSE as informações correspondentes como nesse último caso (v. 6.4). Assim, a RS_{70} deve buscar a seção da TIR correspondente ao bloco sendo compilado, procurando uma entrada com classe LABEL e com valor igual ao do parâmetro p_3, do AL (v. 3.4); se ela for encontrada, há um erro de declaração duplicada de rótulo em um mesmo bloco; caso contrário, o rótulo é inserido na TIR da seguinte maneira:

TIR[i]: m, LABEL, LABELT, 0, n

onde m é o conteúdo de p_3 do AL e n o número do rótulo (endereço) no programa-objeto. Como geraremos código simbólico em linguagem de montagem do computador HIPO (v. apêndice III), não é necessário conhecer o endereço do comando a ser rotulado com esse rótulo m. O rótulo a ser gerado será simplesmente a cadeia 'RE_n', correspondendo ao *rótulo externo* (isto é, declarado no programa-fonte) de número n. Esse n deve ser o valor de um contador, incrementado para cada novo rótulo inserido na TIR pela RS_{70}.

g_2) **const**

Como vimos em 6.2, o IDEN de **const** entra na TIR, por exemplo na entrada k, com classe COIDEN por ação da RS_{10}. A RS_{12} consulta PSE[TOPPS], para saber em que tabela se encontra a constante. Essa informação foi inserida na PSE pela RS_{71} de COIDEN em "const", vista em (b) acima. Se na PSE encontrar-se o par STIR, j trata-se de "const" declarado anteriormente como uma constante, por exemplo na declaração de A2 em **const** A1 = 5; A2 = A1. A constante correspondente a "const", A1 no caso, terá por construção uma entrada na TIR com classe COIDEN e um certo tipo no campo TIPO. Nessa situação, inserimos os valores de TIR[j]. TIPO e TIR[j]. INDTT nos campos correspondentes de TIR[k] onde se encontra o IDEN sendo declarado. O valor de ENDOBJ é retirado das tabelas TCN ou TCA.

g_3) **type**

A RS_{50}, depois de introduzir o valor do parâmetro p_2 do AL em TIR[i].IDROT e fazer TIR[i].CLASSE: = TYIDEN, empilha na PSE, em paralelo a IDEN, o par STIR,i. Com isso, se forem definidas constantes simbólicas dentro de "type", produzindo novas entradas na TIR, pode-se voltar à entrada i da TIR. A RS_{51}, ao ser executada, encontra em PSE[TOPPS] o par STIR,j indicando que o tipo associado ao "type" que acabou de ser reconhecido está na entrada j da TIR. Se TIR[j].TIPO = TYIDT (o que aconteceria com T2, por exemplo nas declarações **type** T1 = integer; T2 = T1), faz-se j: = TIR[j].INDTT. Nesse caso, copiam-se em TIR[i] os

campos TIPO, INDTT e ENDOBJ de TIR[j]. O índice i é obtido em PSE[TOPPS-2].IND, onde foi empilhado como explicado acima.

g_4) **var**

Já vimos o funcionamento das rotinas "semânticas" nesse caso, no exemplo apresentado no começo deste item. Note-se a diferença com (g_3) no sentido de que aqui pode haver uma lista de identificadores.

g_5) **proc**

A RS_{40} produz a entrada na TIR do identificador (nome) do procedimento com classe PRIDEN e tipo TYIDT, e faz o tratamento da abertura de um novo bloco, como veremos em 6.6. O nó seguinte desvia para "palist", onde são colocadas as informações dos parâmetros, como visto em (f). A RS_{141} faz o tratamento do encerramento de um bloco, como veremos em 6.6, chamando em seguida a RS_1 (v. 6.2).

g_6) **func**

A RS_{20} atua de maneira idêntica à RS_{40}, a menos da entrada da classe FUIDEN. A RS_{74} usa o índice i do par STIR,i que está em PSE[TOPPS-3], em paralelo a IDEN, para colocar em TIR[i].INDTT o índice da entrada da TIR onde foi declarado o TYIDEN. A mesma indexação composta descrita em (g_2) e (g_3) pode ser usada, para apontar-se para o tipo final.

Note-se que essa otimização da eliminação de indexações compostas de tipos pode ser empregada em todos os nós onde ocorre um TYIDEN, em "palist", "filist", "type" e "sitype".

h) "progrm"

Não entraremos nos detalhes do nome do programa (primeiro IDEN) e dos parâmetros de programa (segundo IDEN), pois eles têm a ver com a ativação do programa e os arquivos de entrada e saída /J-W 74/, o que exigiria detalhes de ligação com o sistema operacional que controla o compilador e a execução do programa-objeto.

6.6 ESTRUTURA DE BLOCOS

Na linguagem ALGOL-60 /NAU 63/ foram concretizados dois conceitos muito importantes nas linguagens de programação: o de *estrutura de blocos* e o de trecho de validez de identificadores, ou *escopo.* Naquela linguagem, podem haver declarações dentro de qualquer trecho delimitado por **begin** e **end**, antes do aparecimento de qualquer comando que não esteja dentro de um procedimento declarado nesse trecho. Havendo pelo menos uma declaração em um desses trechos, ele passa a ser denominado *bloco* ("block"), caso contrário ele é simplesmente um "comando composto" ("compound statement"). O *escopo* de um identificador é todo o trecho do bloco no qual ele foi declarado. Um mesmo identificador pode ser declarado em um bloco e, depois que este foi terminado por seu **end**, novamente em um bloco posterior, como por exemplo, usando a sintaxe do ALGOL-60 (tipos antes dos identificadores), podemos ter:

...**begin integer** A;...**end**...**begin Boolean** A;...**end**...

Não há confusão entre esses identificadores devido aos escopos diferentes de cada um. Por outro lado, é possível declarar a mesma variável em blocos encaixados:

...**begin integer** A,...**begin Boolean** A;...**end**...**end**...

Novamente, não há confusão entre as duas variáveis, pois se aparecer uma referência a A no bloco mais interior, ela diz respeito à declaração **Boolean** A; fora desse bloco mais interior, qualquer referência a A diz respeito ao bloco mais exterior (se estiver dentro de seus limites, é lógico). Em particular, A poderia ter exatamente o mesmo tipo nos dois blocos.

Se em um bloco é feita referência a um identificador A declarado em um bloco que engloba o primeiro, dizemos que A é *global* ao bloco interior. Por outro lado, um identificador declarado em um bloco diz-se local a esse bloco. Assim, em

...**begin procedure** P;...**begin integer** I;...P;...I...**end**...P;...**end**

temos: P é local ao bloco externo e global ao interno; I é local ao bloco interno.

Vale a pena ainda darmos um contra-exemplo. No trecho abaixo, A é declarada somente no bloco interno:

...**begin**...**begin integer** A;...**end**...A...**end**...

A referência a A no bloco externo é inválida; seu escopo limita-se ao bloco interno, onde foi declarada.

Na linguagem PASCAL só existem blocos, no sentido mencionado acima, i) no programa principal, isto é "block" de "progrm" (v. grafo no apêndice I) e ii) dentro de um procedimento ou função englobando os seus parâmetros formais (declarados em "palist" no subgrafo de "block", ramos **proc** e **func**) e "block" (no mesmo subgrafo e mesmos ramos) e excluindo o identificador do procedimento ou função. Assim, para abrir-se dentro de um programa um novo bloco com suas declarações é necessário declarar neste bloco um procedimento ou função. (Deste ponto em diante, mencionaremos apenas procedimentos quando funções tiverem o mesmo tratamento.) Para abrir-se um bloco dentro de um procedimento, é necessário declarar no bloco deste um outro procedimento. Note-se como em "block", no ramo **begin**, temos um comando ("statm") e não um "block"; em "statm" não há declarações a não ser constantes sob a forma de números inteiros ou cadeias de caracteres, que são sempre consideradas globais. Como veremos mais adiante, essa restrição à estrutura de blocos da PASCAL não precisaria ter sido feita, podendo-se implementar uma estrutura de blocos geral como em ALGOL sem perder a eficiência do programa-objeto e praticamente sem introduzir dificuldades adicionais ao compilador. Vejamos como introduzir a estrutura de blocos no compilador. Chamamos a atenção para o fato de que, segundo nossa denominação, o *bloco* de um procedimento engloba os seus parâmetros formais e seu "block", conforme a sintaxe da PASCAL. Já no programa principal o bloco engloba os identificadores que precedem "block".

Façamos a seguinte definição indutiva de um termo muito importante:

i) O *nível de encaixamento* do bloco mais externo (o de "progrm") é 0 (zero);

ii) Se o nível de encaixamento de um bloco é n, o *nível de encaixamento* de um bloco interior ao mesmo, sem que haja nenhum outro englobando o segundo e interno ao primeiro, é $n+1$.

O mecanismo do compilador que trata da estrutura de blocos deve implementar as seguintes características:

a) A TIR é a única tabela que é afetada pela estrutura de blocos; tanto a TCN como a TCA funcionam globalmente a todo programa e a seus procedimentos;

b) A busca de um identificador A (ou um rótulo n) na TIR deve começar sempre pela comparação com os identificadores (rótulos) do bloco mais interior que engloba o ponto que se está compilando; a seguir, devem ser comparados com A os identificadores (rótulos) do bloco que engloba imediatamente aquele mais interior, e assim sucessivamente até o bloco mais externo do programa; se A (n) não for encontrado na TIR, há um erro de identificador (rótulo) não declarado.

c) Quando se atinge um **end** de um bloco (não de um comando composto, no sentido do ALGOL-60!), deve-se retirar da TIR todos identificadores e rótulos que foram declarados dentro desse bloco, conservando na TIR os de todos os blocos que o englobam.

d) Quando é declarado um novo identificador (rótulo) dentro de um certo bloco, deve-se buscar a TIR somente na seção correspondente a esse bloco, para verificar se está havendo declaração duplicada de identificadores (rótulos), já que pode haver duplicação em blocos encaixados. Esse identificador (rótulo) deve ser colocado na seção correspondente ao bloco sendo declarado.

e) É necessário conhecer o nível de encaixamento de cada bloco, para se poder colocar informações a esse respeito nos procedimentos, bem como compilar corretamente os endereços dos identificadores locais.

f) O bloco de um procedimento começa com os parâmetros do mesmo que são considerados como variáveis locais. O identificador do procedimento faz parte dos identificadores do bloco que engloba imediatamente o bloco do procedimento.

g) Ao encerrar-se um bloco de um procedimento deve-se manter todos os parâmetros do mesmo na TIR; esses parâmetros passam a fazer parte do bloco do identificador do procedimento, permitindo assim que seja verificado se os parâmetros atuais em alguma chamada concordam, nos seus tipos, com os parâmetros formais.

Essas características sugerem a seguinte organização da TIR:

i) A TIR deve ser composta de seções bem delimitadas, uma para cada bloco.

ii) Essas seções devem constituir uma estrutura de pilha, podendo-se considerar cada seção como uma de suas células.

iii) A seção do bloco que está sendo compilado no momento corresponde ao topo da pilha.

iv) Quando se atinge o **end** de um bloco, a sua seção deve ser desempilhada; se esse bloco pertencer a um procedimento com parâmetros, estes últimos não devem ser desempilhados, sendo assim incorporados à seção anterior. Para não haver duplicação com identificadores dessa seção anterior, o conteúdo do campo IDROT de cada parâmetro deve ser substituído por 'b'.

v) A busca de um identificador ou rótulo deve começar na seção do topo da pilha, continuando com a seção seguinte, até atingir o fundo.

vi) A inserção de um identificador ou rótulo deve ser feita na seção que está no topo da pilha; o teste de duplicação da declaração de um mesmo identificador ou rótulo é feito somente na seção do topo.

Para implementar-se essa pilha e as seções descritas, vamos introduzir: a) duas variáveis INSEC e FIMSEC cujos valores são índices de entradas da TIR, correspondendo respectivamente ao primeiro e ao último identificador (rótulo) da seção do topo; por meio dessas variáveis podemos inserir novos identificadores (incrementando FIMSEC) e efetuar buscas somente na seção do topo (isto é, no bloco sendo declarado), quando necessário; b) uma pilha auxiliar de índices para cada entrada da TIR onde se encontra o primeiro símbolo de cada seção; essa pilha será implementada usando-se células da pilha semântica PSE. As RS_{40} e RS_{20} do nó IDEN em **proc** e **func** do subgrafo de "block" colocam na célula da PSE paralela a **proc** ou **func** o índice da entrada da TIR onde se encontra o primeiro símbolo do bloco anterior, ou melhor, do bloco que engloba o novo bloco que começa a ser declarado; elas também atribuem novos valores para INSEC e FIMSEC. A RS_{141}, no nó "block", desempilha uma seção da TIR, pois um bloco acabou de ser encerrado, atualizando os valores de INSEC e FIMSEC com os índices da seção (bloco) anterior, que foi interrompido pela declaração de um procedimento; FIMSEC é ainda ajustado para incluir os parâmetros desse procedimento, cujos identificadores são apagados em IDROT.

Vejamos um exemplo da organização da TIR como pilha.
Seja o trecho de programa:

```
progr X1 (output);
  /* PDX1: primeiras declarações de X1 */
proc P1;
     /* PDP1: primeiras declarações de P1 */
   proc P2(A,B:...);
      /* DP2: declarações de P2*/
      begin /* comandos de P2 */ end;
   /* UDP1: últimas declarações de P1 */
    begin /* comandos de P1 */ end;
/* UDX1: últimas declarações de X1 */
begin /* comandos de X1 */ end.
```

Na fig. 6.6 apresentamos as seguintes situações da TIR e das variáveis INSEC e FIMSEC:

i) Nos comandos de P2; ii) Nos comandos de P1.

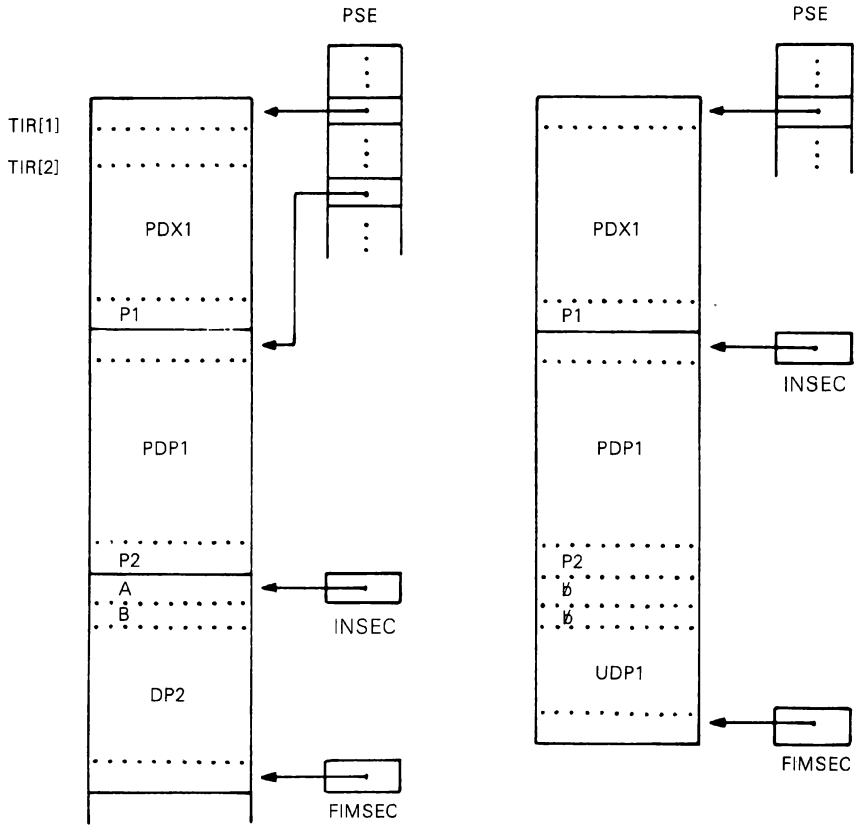


Fig. 6.6

As rotinas semânticas que empilham e desempilham os índices são as seguintes: (NIVAT é uma variável cujo valor indica o nível de encaixamento do bloco sendo compilado):

RS_{40}, RS_{20}: PSE[TOPPS-1].IND: = INSEC;
INSEC: = FIMSEC + 1;
NIVAT: = NIVAT + 1;

RS_{141}: FIMSEC: = INSEC-1;
if PSE[TOPPS-3] tem referência a ':' na TABT
then INSEC: = PSE[TOPPS-6].IND /* **func*** /
else INSEC: = PSE[TOPPS-4].IND; /* **proc** */
while TIR[FIMSEC + 1].TIPO = PARAM or TIR[FIMSEC + 1] TIPO = PARAV
/* varre os parâmetros */
do begin FIMSEC: = FIMSEC + 1;
TIR[FIMSEC].IDROT: = 'b' **end**;
NIVAT: = NIVAT-1;

A inicialização da TIR, com os tipos e constantes predefinidos, e das variáveis, NIVAT, INSEC e FIMSEC, é feita pela RS_{100}, de '(' em "progrm".

Quando alguma rotina semântica introduz um novo símbolo na TIR, deve inicialmente verificar se esse símbolo já se encontra na última seção, isto é, entre INSEC e FIMSEC; sendo esse o caso, há um erro, caso contrário FIMSEC é incrementado e o novo símbolo é introduzido em TIR[FIMSEC]. A busca de um símbolo na TIR é feita de FIMSEC até TIR[1], nesse sentido, encontrando-se portanto a entrada correspondente à declaração desse símbolo mais interior ao ponto em que ele está sendo referenciado. Estamos supondo aqui uma busca linear, que é a mais simples.

Pode-se organizar a TIR com um sistema de espalhamento ("Hashing", v. 2.4); nesse caso pode-se usar uma tabela em forma de pilha como a introduzida aqui apenas com o campo IDROT e uma outra adicional usando o sistema de "Hashing", novamente com esse campo e agora com os campos restantes da TIR. As inserções são feitas em ambas; as buscas são feitas somente na de "Hashing"; para isso, símbolos iguais devem ser ordenados nas cadeias de colisões segundo a estrutura de blocos a que eles correspondem, isto é, blocos mais externos no fim das cadeias de colisão; as eliminações ao fim dos blocos são feitas percorrendo-se a seção do topo da tabela em forma de pilha, eliminando-se para cada um de seus símbolos a entrada correspondente na tabela de "Hashing" (/GRI 71/).

A busca linear não é tão ineficiente quanto pode parecer à primeira vista. Isso se deve ao fato de que a probabilidade de se utilizar um símbolo de um bloco mais interno é em geral maior do que a probabilidade de se usar um símbolo de bloco mais externo, isto é, a tendência é de se empregarem símbolos locais, e não globais. A tabela em forma de pilha favorece justamente essa característica.

Pode-se ainda melhorar um pouco o desempenho da busca no caso da PASCAL, já que nessa linguagem a ordem das declarações é prefixada: primeiramente **label**, depois **const** etc. Assim, ao procurar-se um rótulo, pode-se começar na primeira célula de cada seção, passando-se à seção seguinte logo que aparecer a primeira entrada com classe diferente de LABEL. Por outro lado, as classes de **proc** e **func** são sempre as últimas de cada seção; isso sugere uma busca do fim para o começo na seção até aparecer uma outra classe diferente dessa quando se pode já pular para o fim da seção anterior. Essas buscas selecionadas deveriam ser ativadas a partir de pontos específicos do grafo. Por exemplo, antes de qualquer nó TYIDEN poder-se-ia fazer a busca somente nas áreas onde estão declarados os tipos.

Note-se a vantagem de colocar rótulos junto com identificadores na mesma tabela; isso se deve ao fato dos primeiros seguirem as mesmas regras de escopo que os segundos; assim, a manipulação da TIR como pilha é feita para ambos simultaneamente.

A introdução da estrutura de blocos produz um pequeno problema, qual seja o de eliminar das tabelas descritoras dos tipos intervalo, matriz e registro (TABSRG, TABARR e TABREC) as entradas correspondentes à seção da TIR sendo desempilhada. Para isso, antes de desempilhar-se uma seção da TIR pode-se varrê-la de INSEC até FIMSEC, neste sentido, buscando as primeiras entradas correspondentes a cada uma daquelas tabelas, o que se dá com a primeira ocorrência de cada um dos tipos SEQT, ARRAYT e RECT. Toma-se o conteúdo de INDTT, por exemplo j_1, j_2 e j_3, respectivamente; o último elemento ativo de cada tabela passa a ser j_1-1, j_2-1 e j_3-1.

É interessante observar que as rotinas semânticas que dizem respeito às tabelas de símbolos constituem aquilo que denominamos de "Analisador de Contexto" (v. fig. 2.6).

6.7 PROJETO — PARTE VI: TABELAS DE SÍMBOLOS

Nesta parte do projeto devem ser programadas as rotinas semânticas que inserem os símbolos do programa-fonte nas tabelas TIR, TCA e TCN, as que produzem a busca nessas tabelas e as que manipulam a TIR em relação à estrutura de blocos. Sugerimos a seguinte seqüência de implementação:

a) Rotinas que introduzem constantes numéricas na TCN e fazem a busca da mesma. Idem, TCA.

b) Rótulos.

c) Considerar apenas o tipo integer.

c_1) Somente programas principais, isto é, a TIR com um só bloco. Rotinas para inserção de rótulos e variáveis de tipo "integer";

c_2) Inserção de identificadores constantes (declaração **const**);

c_3) Busca de identificadores;

c_4) Estrutura de blocos.

d) Tipo "Boolean".

e) Declaração de tipos que se reduzem a "integer" e "Boolean" (p. ex., **type** A = int; B = A).

f) Registros sem matrizes.

g) Apontadores.

h) Seqüências de constantes simbólicas.

i) Intervalos.

j) Matrizes.

k) Registros com matrizes.

l) Conjuntos.

Para testar as rotinas estenda a lista de comandos de controle do compilador descritos em 3.6 e 5.11 introduzindo os comandos TBSION e TBSIOF. Se o primeiro é lido deve ser exibido cada novo elemento que acabou de entrar na TIR, com seu índice; ao abandonar-se um bloco devem ser exibidos todos os elementos do bloco anterior que foram alterados (isto é, parâmetros de procedimentos) ou o índice do novo topo da TIR, se não tiver ocorrido tal alteração. O comando TBSIOF interrompe essas exibições até que ocorra novamente a leitura do comando TBSION.

CAPÍTULO 7

GERAÇÃO DE CÓDIGO-OBJETO — I

7.1 INTRODUÇÃO

A geração de código baseia-se nas informações contidas na pilha semântica (PSE) e, indiretamente, nas tabelas de símbolos. Em geral, as rotinas semânticas utilizam os índices guardados na PSE, que apontam para elementos das várias tabelas; a partir desses elementos essas rotinas geram o código-objeto esperado e modificam as informações da PSE para uso posterior. A posição das informações na PSE é dada supondo, como já vimos, que há uma célula desta para cada nó reconhecido a menos de desempilhamentos ocorridos no reconhecimento de não-terminais.

O código-objeto será gerado na linguagem de montagem ("Assembler") do computador hipotético HIPO (v. apêndice III). O uso de uma linguagem de montagem simplifica muitas construções, principalmente as que fazem referência a partes do programa-objeto ainda não geradas.

Inicialmente não consideraremos procedimentos e funções, isto é, trataremos de mostrar como gerar código somente para elementos do programa principal. Consideraremos todos os endereços da área de dados desse programa relativos a uma base de endereços B1. No capítulo 9 introduziremos as modificações necessárias para tratar de programas com procedimentos e funções, com outras bases de endereços.

O código gerado não será otimizado, para simplificar a exposição.

Abreviaremos "programa-objeto" por PO.

7.2 RÓTULOS E DESVIOS

Rótulos são introduzidos na tabela de identificadores e rótulos (TIR) por meio da RS_{70}, do ramo **label** no subgrafo de "block", como vimos em $6.5g_1$. No momento dessa introdução atribui-se o número do rótulo como será usado no PO, incrementando o contador de rótulos cujo valor é sempre crescente, independendo da estrutura de blocos.

A RS_{75} no início do subgrafo de "statm" deve buscar a entrada da TIR com campo IDROT igual ao valor do parâmetro p_3 devolvido pelo analisador léxico (AL); essa busca, como vimos em 6.6, deve ser feita da seção do topo da TIR até o fundo dessa tabela, nesse sentido. Se o rótulo não for achado, há um erro de contexto; se for achado, ele foi declarado; no PO é gerada a instrução

```
REn     EQ      *
```

onde n é o valor de ENDOBJ da entrada da TIR onde foi encontrado o rotulo; com isso, associamos o rótulo externo REn à próxima instrução a ser gerada no PO.

Note-se que essa solução engloba o caso de rótulos múltiplos, que podem ser usados em PASCAL, por exemplo, como

...R1:;R2:;R3:A:=B...

que geraria no PO as instruções:

```
RE5     EQ      *
RE2     EQ      *
RE7     EQ      *
        LDA     B1+13
        STA     B1+5
```

onde 5, 2 e 7 são os conteúdos dos campos ENDOBJ dos rótulos R1, R2 e R3 respectivamente; como vimos em 7.1, B1 indica a base de endereços do programa principal; supusemos que os endereços de A e de B relativos a essa base sejam 5 e 13.

A RS_{76}, correspondente ao **goto**, busca a entrada da TIR com campo IDROT igual ao valor do parâmetro p_3 devolvido pelo AL; se o rótulo não for achado, há um erro (de contexto); se for achado, gera no PO uma instrução

```
BRN     REn
```

onde n é o conteúdo de campo ENDOBJ daquela entrada da TIR.

7.3 TEMPORÁRIOS

Como já vimos no exemplo dado em 6.4, o resultado de qualquer operação de uma expressão será uma variável *temporária*, isto é, uma variável gerada internamente pelo compilador para armazenar resultados parciais, que serão eventualmente usados em outras operações, em atribuição de valor a uma variável, ou como parâmetros atuais de procedimentos e funções.

No caso da PASCAL, existem operações sobre operandos de quatro tipos: inteiros, booleanos, conjuntos e apontadores ("pointers"). (Lembramos que não tratamos neste texto do tipo "real", isto é, ponto flutuante.) Os resultados de operações envolvendo esses tipos serão, para simplificar, colocados sempre em temporários. Para colocar na PSE informações sobre estes últimos, usaremos no campo TAB os valores ITEMP, BTEMP, STEMP e PTEMP, respectivamente. Como em nossa implementação usamos uma palavra do computador HIPO (v. apêndice III) para cada valor desses tipos, não é necessário ter mais do que uma área comum para todos os temporários. Uma palavra dessa área poderá, portanto, ser empregada ora para um, ora para outro tipo. Essa área começará no PO na localização ARTEMP+0, indo até ARTEMP+TMPMAX, onde TMPMAX+1 é o número máximo de temporários necessários em qualquer ponto do PO. Indiquemos esses temporários por $T_0, T_1, \ldots, T_{TMPMAX}$.

Suponhamos que em um certo ponto do PO, por exemplo em uma expressão aritmética, estão ativados n+1 temporários ($T_0, \ldots, T_n$). Usaremos as seguintes regras para estabelecer o número do temporário no qual será colocado o resultado de uma operação:

caso	1º operando	2º operando	temporário do resultado
(1)	não-temporário	não-temporário	T_{n+1}
(2)	temporário	não-temporário	T_n
(3)	não-temporário	temporário	T_n
(4)	temporário	temporário	T_{n-1}.

Usando-se essas regras, pode-se concluir que nos casos (2) e (3) o operando temporário é obrigatoriamente T_n; no caso (4), os operandos são, da esquerda para a direita, T_{n-1} e T_n.

Com essas regras, conclui-se que a rigor não é necessário colocar na PSE o índice dos temporários indicados em suas células; as operações são sempre feitas com o último temporário ativado, ou com os dois últimos, como se tivéssemos uma pilha. Assim, basta usar

uma variável do compilador, TMPATV, cujo valor é o índice do último temporário ativo (topo da pilha). Logicamente, TMPMAX é o maior valor assumido por TMPATV.

No caso (1) acima, incrementa-se de 1 o valor de TMPATV, que passará a conter o índice do temporário do resultado; nos casos (2) e (3), o valor de TMPATV deve permanecer o mesmo, reaproveitando-se o temporário usado; no caso (4), deve ser decrementado de 1: um temporário acabou de ser liberado.

Se o operador for unário, incrementa-se TMPATV se o operando for não-temporário, caso contrário mantém-se o seu valor, reaproveitando-se o temporário já usado.

No fim do programa-objeto, ao se executar a RS_{180}, de '.' no subgrafo de "progrm", é necessário reservar a área para os temporários. Isso é feito gerando-se a pseudo-instrução

ARTEMP DS n

onde n é um número inteiro, cujo valor é o conteúdo de TMPMAX, adicionado de 1.

7.4 EXPRESSÕES ARITMÉTICAS

Neste item mostraremos como gerar código para expressões aritméticas somente no caso de variáveis simples de tipo inteiro, isto é, que não sejam indexadas e nem apontadores ("pointers"). Pode-se ter campos de registros, pois o que importa neste caso é o seu endereço no PO, que é obtido a partir da TIR. Posteriormente daremos os códigos gerados para os tipos booleano e conjunto ("set"), bem como para as relações ('=', '≠', etc.).

As operações das expressões aritméticas são reconhecidas nos subgrafos de "factor", "term", "siexpr" e "expr" (v. apêndice I).

a) "term"

Para simplificar a exposição, suponhamos que se tenha uma expressão consistindo numa seqüência de multiplicações como A*B*C e que a análise sintática tenha atingido o subgrafo de "term". Ao completar-se o reconhecimento do nó com "factor", teremos no topo da pilha semântica um par STIR, i onde i é o índice de A na TIR. A RS_{153} é chamada; inicialmente, ela deve verificar se é a primeira vez que é chamada nesse "term" ou não, pois se não for a primeira vez o "factor" que acabou de ser reconhecido constitui um operador que está à direita de uma operação para a qual deve ser gerado código. Se for a primeira chamada, como no caso de A na expressão do exemplo, nada há a fazer. A RS_{153} pode detectar se está sendo chamada pela primeira vez nesse "term" verificando se o apontador para as pilhas sintática e semântica, que está empilhando no topo da pilha K do analisador, tem o mesmo valor que o apontador do topo daquelas pilhas (v. 5.7), isto é, se K[TOPO].R = TOPPS, então trata-se da primeira chamada da RS_{153} nesse "term". Suponhamos que a variável B do exemplo tenha acabado de ser analisada como um "factor". A configuração da pilha semântica será, nesse momento, a seguinte:

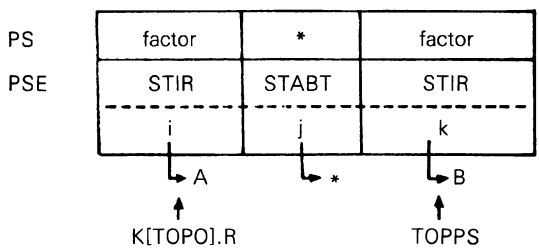


Fig. 7.1

Nessa situação a RS_{153} executa os seguintes passos:

i) Verifica que *não* se trata da primeira vez que é chamada para esse "term";

ii) Verifica qual a operação que deve ser gerada no PO, consultando para isso o conteú-

do de PSE[TOPPS-1], aí encontrando nesse caso uma referência a '*'; tratando-se de operandos aritméticos verifica o tipo dos operandos que estão em PSE[TOPPS] e PSE[K[TOPO].R] (na verdade a conclusão de que '*' é multiplicação e não interseção de conjuntos (v. 7.8) depende do tipo dos operandos).

iii) Gera as seguintes instruções no PO:

LDA E_A
MPY E_B
STA T_p

onde T_p é próximo temporário disponível e E_A e E_B são os endereços de A e B, respectivamente. Estes dois últimos são obtidos através de PSE[K[TOPO].R] e de PSE[TOPPS]; a razão de se ter usado K[TOPO].R e não TOPPS-2 ficará clara mais adiante;

iv) Coloca na célula PSE[K[TOPO].R] o par ITEMP,p.

Através do passo (iv) garante-se que uma referência ao resultado T_p da operação é sempre guardada na primeira célula da PSE correspondente ao "term" sendo analisado. Esse fato é de importância fundamental para que se possa localizar na PSE o operando da esquerda das próximas operações e também para localizar adequadamente na PSE o resultado de toda a expressão ao final desta.

Ao se completar a análise de C como um "factor", as pilhas terão agora a seguinte configuração:

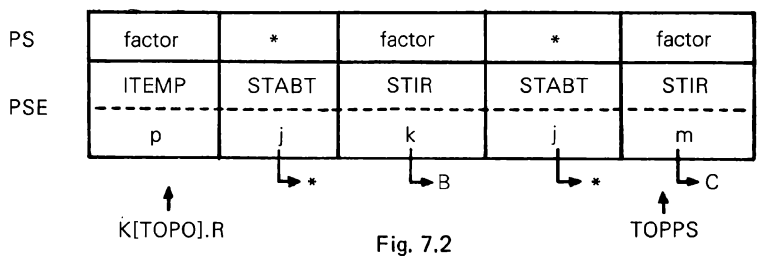


Fig. 7.2

Nessa situação, é a RS_{153} que executa novamente os quatro passos já descritos, gerando agora o código

LDA T_p
MPY E_C
STA T_p

e a carga (redundante) de T_p em PSE [K[TOPO].R].

Ao ser encerrada a análise desse "term", as pilhas sintática e semântica são reduzidas, desempilhando-se os elementos desde seu topo até a célula de índice K[TOPO].R (v. 5.7) que passa a ser o novo valor de TOPPS. Na pilha sintática (se existir) ainda é carregado o não-terminal encontrado, "term". Assim, teremos agora no topo das duas pilhas "term" e T_p respectivamente. Note-se que todo o esquema funciona para um número qualquer de operações.

As operações **div** e **mod** têm processamento idêntico a menos da geração das instruções DIV e MOD respectivamente. Já a operação '/' é usada com operandos reais (ponto flutuante) ou inteiros, dando resultado real, de modo que não será tratada aqui.

b) "siexpr"

No subgrafo de "siexpr" são gerados os códigos para o '−' unário (o '+' unário é ignorado do ponto de vista de geração de código) e para operações binárias de somas e subtração. Para isso, a RS_{152} executa os seguintes passos:

i) Verifica se se trata do primeiro "term" desse "siexpr", e se há operador unário. Para isso, testa se K[TOPO].R = TOPPS; nesse caso trata-se do primeiro "term" e não há operador

unário, encerrando a execução da RS_{152}. Se K[TOPO].R = TOPPS-1, trata-se do primeiro "term" e há operador unário; desvia para (ii) Se nenhum dos testes for verdadeiro, desvia para (iii).

ii) Há um operador unário. Se PSE[TOPPS-1] faz referência a '+' na tabela de símbolos reservados, então faz PSE[TOPPS-1]: = PSE[TOPPS], deslocando assim o resultado do "term" para a primeira célula desse "siexpr". Se a referência for para '−', gera as seguintes instruções:

```
LDA     E
RVS
STA     Tp
```

onde E é o endereço do resultado do "term", obtido através de PSE[TOPPS], e T_p é o próximo temporário disponível. A seguir, coloca o par ITEMP, p em PSE[TOPPS-1]. Tanto no caso de '+' quanto no de '−' a execução da RS_{152} é encerrada nesse ponto.

iii) Não se trata do primeiro "term" desse "siexpr". Nesse caso, são executados os passos (i) a (iv) da RS_{153} do subgrafo de "term" visto em (a) acima, trocando-se naturalmente o código da operação gerada.

c) "expr"

Esse caso é bem mais simples, pois trata-se de uma só operação, e será visto no item 7.6.

d) "factor"

Trataremos aqui exclusivamente do caso de expressões entre parênteses, isto é, do ramo '(' "expr" ')', pois os outros casos são vistos em outros itens. Da maneira como compilamos "term", "siexpr" e "expr", ao se iniciar a execução da RS_{154} o resultado de "expr" estará referenciado na célula PSE[TOPPS], em "paralelo" a esse não-terminal. Devemos simplesmente fazer PSE[TOPPS-1]: = PSE[TOPPS] passando a referência ao "expr" para a célula paralela a '('. Com isso, o "factor", ao ser reconhecido após o ')', ficará com seu significado semântico adequado.

7.5 EXPRESSÕES BOOLEANAS

Não há nenhuma diferença entre a compilação de expressões booleanas e a de expressões aritméticas a menos, é lógico, das instruções geradas. Nas primeiras, usam-se os operadores **or** de "siexpr", **and** de "term" e **not** de "factor". Basta, portanto, examinarmos qual o código que será gerado para essas operações. Esse código depende de qual representação no programa-objeto será empregada para "true" e "false", as constantes booleanas que, como mencionamos em 6.3d, devem ser tratadas de maneira especial pelo compilador. Vamos supor que "false" seja implementada como 0 (zero) e "true" como 1. Essas constantes devem fazer parte obrigatória do PO.

A fig. 7.3 mostra a tabela de valores para as operações booleanas. Vejamos quais instruções devem ser geradas no PO; vamos representar os endereços de X e Y no PO por E_x e E_y, respectivamente; X e Y podem ser temporários, constantes ou variáveis tipo booleano.

X	Y	X **or** Y	X **and** Y	**not** Y
0	0	0	0	1
0	1	1	0	0
1	0	1	0	1
1	1	1	1	0

Fig. 7.3

a) X **or** Y

```
        LDA     Ex
        BNZ     *+2             /* sai se X = "true" */
        LDA     Ey              /* X era "false"; resultado = Y */
        STA     ARTEMP+n
```

b) X **and** Y

```
        LDA     Ex
        MPY     Ey
        STA     ARTEMP+n
```

c) **not** Y

```
        LDA     Ey
        SUB     =1
        BZR     *+2             /* sai se 0 (era 1) */
        RVS                     /* -1→1 (era 0) */
        STA     ARTEMP+n
```

Nesses casos estamos considerando que o resultado fique no temporário T_n.

Note-se que, se tivéssemos usado os valores −1 e 1 para "false" e "true", as instruções geradas para **not** resumir-se-iam a LDA,RVS,STA.

7.6 EXPRESSÕES DE RELAÇÃO

Trata-se de operacões entre dois não-terminais "siexpr", como se pode ver no subgrafo de "expr". O resultado é sempre do tipo booleano, isto é, "false" ou "true". Os operandos devem ser ambos de tipo inteiro, booleano, cadeia ("string"), constantes simbólicas, apontador ("pointer") ou conjunto ("set"), sendo este último abordado no item 7.8; na definição da PASCAL /J-W 74/, há o tipo "char" em lugar de "string" e ainda o tipo "real". Para o tipo conjunto, há ainda o operador de pertinência **in**.

A RS_{156} na segunda "siexpr" de "expr" gera o código-objeto para as relações. Nessa situação as pilhas sintática e semântica têm a configuração da fig. 7.4. Em paralelo aos dois "siexpr" na PSE temos o par T,i e T,j correspondentes às localizações dos temporários, variáveis ou constantes que podem ter dado origem aos "siexpr"; representamos por "op" um dos operadores possíveis; paralelo a ele temos o par STABT,k indicando sua localização na TABT.

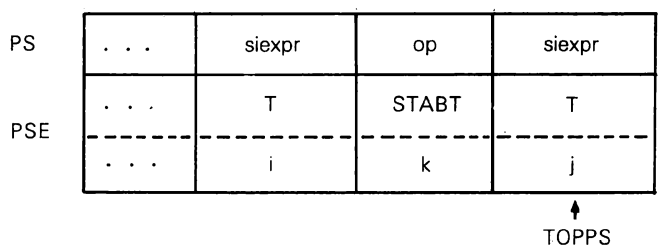


Fig. 7.4

O resultado dessa expressão deverá ser um temporário booleano, colocado pela RS_{156} em paralelo ao "siexpr" mais à esquerda; sua localização na PSE é, evidentemente, TOPPS-2. A RS_{156} verifica se os dois operandos têm tipos iguais e gera o código adequado para esse tipo e para o operador. Como trataremos do tipo "set" no item 7.8 e o tipo "real" não será tratado, as instruções geradas são as mesmas para todos os tipos aqui abordados. Suponhamos uma comparação X op Y, onde X e Y podem ser variáveis, constantes ou temporários.

a) inteiros, booleanos e cadeia de caracteres

```
LDA     E_x
SUB     E_y
Bzz     *+3
LDA     =0              /* insucesso */
BRN     *+2
LDA     =1              /* sucesso */
STA     ARTEMP+n
```

O desvio Bzz segue a seguinte tabela:

op:	=	<	>	<=	>=	< >
Bzz:	BZR	BNG	BPS	BNP	BNN	BNZ

No caso de tipos booleanos, '=' corresponde à equivalência, '≤' à implicação lógica e ≠ ao "ou exclusivo". Essas correspondências resultam como conseqüência direta da ordem das constantes, isto é, "false" < "true", o que é respeitado pelos valores 0 e 1 que adotamos.

Vamos restringir as cadeias de caracteres a no máximo 5 caracteres, para que possam ser representadas no computador-objeto HIPO (v. apêndice III) em uma só palavra, alinhadas à esquerda com preenchimento de brancos à direita. Se assim não fosse, poderíamos seguir a recomendação do manual da PASCAL /J-W 74/ e implementá-las como matriz unidimensional de caracteres; nesse caso, o comprimento a cada cadeia seria dado pelas informações da TIR e da TABARR (v. 6.3h). As palavras com cadeias têm sempre sinal positivo; assim, a comparação de inteiros empregada pelas instruções acima segue exatamente a seqüência lexicográfica, no caso de cadeias de letras.

b) constantes simbólicas

Como definimos em 6.3d, constantes simbólicas são identificadores declarados em seqüências das mesmas, em "sitype" entre '(' e ')'.

A sua declaração estabelece uma ordenação entre os vários elementos da seqüência. Assim, (AZUL,VERDE,ROSA) estabelece por exemplo
AZUL < VERDE.

Seguindo a sugestão de implementação de seqüência de constantes simbólicas dada em /WIR 71/, vamos supor que essas constantes (p. ex., AZUL, VERDE etc.) não sejam armazenadas no programa-objeto. Uma variável que assume um valor de uma dessas seqüências, isto é, de uma constante simbólica, recebe no PO um número inteiro, que indica a ordem dessa constante na seqüência da declaração. A correspondência entre a constante simbólica e seu número de ordem na seqüência é conhecida somente em tempo de compilação, e dada no campo INDTT da TIR (v. 6.3d). O compilador deve converter cada constante simbólica em seu número de ordem para uso no PO. Não foram definidas em PASCAL construções que necessitassem das próprias constantes simbólicas no PO; por exemplo, na indicação de entrada/saída de /J-W 74/ só se faz menção a variáveis de tipos "integer", "Boolean", "char" e "real". Em termos de expressões, somente são usados os operadores de comparação. A sua implementação é idêntica ao caso (a), já que durante a execução do PO os valores de variáveis e constantes do tipo em questão são na realidade números inteiros.

c) apontador ("pointer")

Para esse tipo estão definidas apenas as comparações '=' e '≠'; após a detecção do tipo a RS_{156} verifica se o operador é um desses dois permitidos e gera as instruções correspondentes como no caso (a), fazendo-se assim a comparação entre os endereços dos apontadores. A comparação com a constante nil é efetuada normalmente, usando-se seu valor como visto em 6.3c. No item seguinte examinamos expressões com apontadores.

7.7 EXPRESSÕES COM APONTADORES

O tipo apontador ("pointer") é declarado em "type" no ramo de '↑', como vimos em 6.5d_1. Uma variável P declarada com esse tipo contém o endereço de uma palavra da memória do PO, onde começa um objeto de um tipo T qualquer. Por exemplo, as declarações seguintes definem P como apontador para um registro:

```
type PESSOA: record NOME: alpha;
                    PAI: ↑PESSOA;
                    MÃE: ↑PESSOA;
                    IDADE: 0..110
             end;
var P: ↑PESSOA;
```

A criação de uma nova área na memória comportando um registro do tipo PESSOA e um apontador para a mesma é feita através da função new (P). Com isso, P fica apontando para um registro que ainda está vazio, e cujos campos ainda devem ser preenchidos. No exemplo acima, devemos criar obviamente dois registros, por exemplo com nomes ADÃO e EVA, cujos campos PAI e MÃE contêm **nil**.

Dado um certo registro apontado por P, podemos saber o nome do avô do mesmo por meio de P↑.PAI↑.NOME. Se houver aniversário, podemos fazer P↑.IDADE: = P↑.IDADE + 1.

No caso de expressões, interessa-nos no momento uma construção mais simples do que a do lado direito dessa atribuição, pois aí temos um caso de campo de registro, que será abordado em 7.10. Por exemplo, as declarações

```
type T1: ↑integer;
     T2: ↑T1;
var  P1: T1; P2: T2;
```

podem ocasionar expressões como P1↑ + 1 + P2↑↑.

Vejamos uma maneira de compilar essas variáveis com apontadores. A RS_{160} em '↑' do subgrafo de "infipo" pode consultar a célula PSE[TOPPS-1]; se aí houver um VAIDEN (como P1 acima), ela não faz nada. A RS_{167}, em "infipo" de "factor", consulta PSE[TOPPS] e aí encontra o par STABT,i apontando para a entrada de '↑' em TABT. Nesse caso, ela gera a seguinte seqüência de instruções armazenando o conteúdo da palavra apontada pelo VAIDEN no próximo temporário disponível:

```
LDA  I  E
STA     ARTEMP+n
```

onde E é o endereço do apontador correspondente ao VAIDEN, endereço esse obtido da sua entrada na TIR, indicada no par STIR,j em PSE[TOPPS-1]; a instrução LDA usa um endereçamento indireto para carregar o conteúdo da palavra apontada por E. Evidentemente, usamos uma só palavra por tratar-se de apontador para inteiro. Essa ação de se tomar o conteúdo de uma palavra apontada por um apontador P é denominada de "derreferenciação".

Em seguida, a RS_{167} coloca em PSE[TOPPS-1] o par ITEMP,n indicando o resultado, que ficará sendo a "semântica" de "factor".

Se a RS_{160} encontrar o par STABT,i em PSE[TOPPS-1], ocorrem dois '↑' em seguida, como em P2 no exemplo acima. Nesse caso é necessário fazer uma "derreferenciação" inicial; essa rotina produz então a mesma seqüência de instruções como acima, consultando agora a entrada da TIR apontada por PSE[TOPPS-2], célula paralela ao VAIDEN; após essa geração, o par PTEMP,n é colocado em PSE[TOPPS-1), em paralelo ao primeiro '↑', recaindo-se no caso anterior. Portanto, cada '↑', a menos do último, produz uma derreferenciação.

7.8 EXPRESSÕES COM CONJUNTOS

As expressões com conjuntos ("sets") envolvem os seguintes operadores /J-W 74/:

'+' – união
'*' – interseção

'−' – complemento relativo (p. ex., A-B indica o conjunto com os elementos de A que não pertencem a B)
'=', '<>' – igualdade e desigualdade dos conjuntos
'<=' e '>=' – inclusão, correspondentes a $\subseteq$ e $\supseteq$
in – pertinência; nesse caso o primeiro operando é de tipo escalar e o segundo de tipo conjunto.

Conjuntos são associados sempre a seqüências de escalares, como por exemplo:

```
var ARCOIR: set of (VERM,LARANJ,AMAR,VERDE,AZUL,ANIL,VIOL);
    LETRAS: set of 'A'..'J';
```

A maneira de armazenar as informações desses tipos foi vista em 6.3l e 6.3g; note-se no item 6.3g o tratamento especial para o caso de intervalo de inteiros ou caracteres.

No PO o tipo "set" gera uma palavra onde existe um indicador para cada elemento do conjunto, da esquerda para a direita, na ordem da seqüência. Em computadores binários, um "bit" serve como indicador. No caso do HIPO (v. apêndice III), usaremos o dígito 1 como indicador de presença, e 0 como ausência; podemos, portanto, ter conjuntos de até 10 elementos, já que empregaremos apenas uma palavra para cada conjunto. Assim depois da atribuição

```
ARCOIR:=[LARANJ,VERDE,AZUL]
```

teremos na palavra correspondente a ARCOIR o conteúdo +0101100000

Fazendo-se

```
ARCOIR:=[ ]  /* conjunto vazio */
```

teremos o conteúdo +0000000000.

Note-se que o tipo conjunto é na verdade o conjunto de potências, pois seus valores indicam todos os possíveis subconjuntos do conjunto declarado.

O resultado de expressão envolvendo conjuntos é sempre uma variável (se não houver operadores) ou um temporário, ambos do tipo conjunto. Associado a esse tipo existe a constante que representa conjunto vazio '[ ]', cujo conteúdo é +0000000000; na PSE uma referência a ela será indicada pelo par STEMP,-1; no PO ela será armazenada na palavra de rótulo 'VAZIO'. As rotinas "semânticas" que manipulam variáveis do tipo conjunto devem verificar sempre se estão tratando com essa constante.

Vejamos as rotinas "semânticas" para esse tipo.

a) "factor"
a1) [ ]
A RS_{161} coloca em PSE[TOPPS-1] o par STEMP,-1.
a2) ["expr" ...]

Aqui começa a especificação de um subconjunto de algum conjunto declarado. A RS_{162} pode reconhecer que se trata de fato da primeira especificação do subconjunto, isto é, que se trata da primeira passagem pelo nó contendo "expr" à esquerda de '..', quando ocorrer um par STABT,i em PSE[TOPPS-1], onde i é o índice da entrada da TABT onde se encontra '['; na volta a esse nó teremos obrigatoriamente uma referência a ',', em PSE[TOPPS-1].

Tendo detectado a primeira ocorrência de "expr", a RS_{162} deve gerar instruções para produzir a palavra que representa o subconjunto correspondente ao elemento indicado pela "expr". Os próximos elementos são introduzidos no subconjunto por meio de uniões sucessivas. A RS_{162} ainda coloca em PSE[TOPPS-1], isto é, em paralelo ao '[', o par STEMP,j.

Para representar um subconjunto somente com um elemento, a fim de usá-lo na inicialização e nas uniões seguintes, é preciso montar uma palavra com um dígito 1 na posição desse elemento e 0 nas outras posições.

A posição do dígito 1 dentro da palavra que indica o elemento do conjunto pode ser determinada pela posição desse elemento dentro da seqüência. Assim, nos exemplos dados, VERM tem posição 0; LARANJ, 1; A, 0; D, 3 etc. Recordemos que no programa-objeto as constantes simbólicas e elementos de intervalos (como por exemplo elementos de 4..9) são representados pela sua ordem na seqüência e não por seus valores usados no programa-fonte

(v. 6.2d e 6.2e). Suponhamos que essa ordem esteja guardada na constante do PO de rótulo P. As seguintes instruções geram a palavra que representa o elemento do conjunto; a instrução SRA 1 0 provoca um deslocamento dígito a dígito do acumulador, n dígitos para a direita onde n é o conteúdo do indexador 1 (na verdade, deve ser usado o próximo indexador livre).

```
LDA     P               /* armazena o número de ordem */
STA     1               /* no indexador 1 */
LDA     =1000000000     /* carrega o primeiro elemento do conjunto */
SRA     1 0             /* calcula o elemento procurado */
STA     T_j             /* temporário contendo o resultado */
```

Quando a "expr" da RS_{162} é uma constante, sua ordem P na seqüência pode ser calculada pelo compilador. Se for uma variável, o seu valor conterá, por construção, o número de ordem do elemento da seqüência. Se for uma expressão envolvendo constantes simbólicas, ela se constituirá necessariamente de aplicações das funções succ e pred (como em succ(succ(VERDE))). Nesse caso, o compilador gera simplesmente a carga da ordem da constante, e instruções de incremento ou decremento de 1 para cada chamada da função succ ou pred, respectivamente (o que pode ser feito pela RS_{79} do ramo FUIDEN do subgrafo do "factor", no apêndice I).

A RS_{163} deve gerar instruções no PO para calcular a palavra representando todo o subconjunto indicado pelo intervalo. Se esse intervalo é de constantes, como por exemplo D..H, o próprio compilador deve calcular essa palavra. Se forem expressões, é necessário gerar a palavra que represente cada elemento e, através da instrução LDG, que funciona como instrução OR para os dígitos 1 e 0, ir montando a palavra final. Para isso, a RS_{163} deve gerar uma malha de instruções que executará tantas SRA e LDG quantos forem os elementos do intervalo, menos um (correspondente à SRA do elemento inicial).

Se não for a primeira vez que as RS_{162} e RS_{163} estão sendo executadas, devem ser geradas instruções LDG que fazem a união dos novos elementos ao conjunto já calculado em suas execuções anteriores.

O resultado, ao ser reconhecido um "factor", será um par STEMP,j colocado na PSE em paralelo a esse não-terminal.

Não abordamos o problema da detecção de erros devido à incompatibilidade de tipos dos elementos do subconjunto, que deve evidentemente ser feita.

b) "term"

A RS_{153} deve detectar que há um operador antes de "factor", isto é, em PSE[TOPPS-1]; nesse caso, se for '*' (interseção) verifica os tipos dos dois operandos, que estão em PSE[K[TOPO].R] e PSE[TOPPS] como nas expressões aritméticas (v. 7.4); sendo os dois de tipo "set", ela produz:

```
LDA     E_1
LZR     E_2             /* E ∩ E_2 */
STA     ARTEMP+j
```

onde E_1 e E_2 são os endereços obtidos a partir das informações das células da PSE mencionadas e T_j é o temporário gerado segundo 7.3; a instrução LZR funciona em nosso caso (zeros e uns) como **and** ("e" lógico): o par STEMP,j é carregado em PSE[K[TOPO].R].

c) "siexpr"

A RS_{152} é processada como a RS_{153}; ao contrário do caso de tipos inteiros (v. 7.4) não há operador unário para conjuntos.

No caso de '+' temos uma união; em lugar de LZR em (b) acima, é gerada LDG.

No caso de '−', temos uma complementação relativa. É gerada a seqüência de instruções

```
LDA     E_1
LZR     E_2             /* E_1 ∩ E_2 */
STA     ARTEMP+j
```

```
LDA     E1              /* E1 - E1∩E2 */
SUB     ARTEMP+j
STA     ARTEMP+j
```

Por exemplo se os subconjuntos fossem representados respectivamente por 11011 e 11100 teríamos como resultado intermediário 11000 e como resultado final 00011.

d) "expr"

A RS_{156} produz, nos casos '=' e '≠', código igual às relações vistas em 7.6.

No caso de '≥', inclusão ($\supseteq$), a RS_{156} gera o código

```
LDA     E1
LZR     E2              /* E1∩E2 */
SUB     E2              /* E1∩E2 = E2? */
BNZ     *+3
LDA     =1
BRN     *+2
LDA     =0              /* insucesso */
STA     ARTEMP+j
```

onde E_1 e E_2 correspondem às "siexpr" à esquerda (em PSE[TOPPS-2]) e à direita (em PSE[TOPPS]) do operador, respectivamente. No caso de '≤' ($\subseteq$) a RS_{156} gera a mesma seqüência, trocando a ordem dos operandos, E_1 e E_2, ou simplesmente trocando E_2 da instrução SUB por E_1.

No caso de **in** (pertinência), a RS_{156} usa a constante simbólica, o inteiro ou o caractere referenciado pela "siexpr" da esquerda e converte-o para uma constante representando o subconjunto unitário do conjunto do tipo do "siexpr" à direita, como vimos em (a) acima, colocando-a em E_1.

A seguir gera:

```
LDA     E1
LZR     E2              /* E1∩E2 */
BZR     *+2             /* E1∩E2 = Φ? */
LDA     =1              /* E1∩E2 ≠ Φ */
STA     ARTEMP+j
```

7.9 VARIÁVEIS INDEXADAS

Denominaremos de *matriz* a estrutura declarada em um "array", cujas informações são guardadas na TIR como vimos em 6.3h. Cada elemento de uma matriz especificado através de seus índices será denominado de *variável indexada*.

Uma variável indexada é detectada no ramo VAIDEN do subgrafo de "factor" nos comandos de atribuição, **for** e **with** de "statement". Por outro lado, campos de registros podem também ser indexados, o que é detectado por um FIIDEN seguido de índices no subgrafo de "infipo"; finalmente podem seguir-se a apontadores, como é o caso de um '↑' seguido de índices também em "infipo". Além disso, os índices podem ser de vários tipos: intervalos de inteiros, de caracteres e de constantes simbólicas.

Neste item abordaremos a geração de código para variáveis indexadas. Não nos preocuparemos em gerar código eficiente, pois isso complicaria demasiado a exposição.

a) Índices inteiros — o "dope vector"

Suponhamos uma declaração de matriz

var M: **array** $[i_1..s_1, i_2..s_2, \ldots, i_n..s_n]$ **of** T

onde i_q,s_q indicam os limites inferior e superior da q-ésima dimensão. Como vimos em 6.3h, esses limites estão armazenados na TIR, através de referência à TABARR, que localiza os tipos dos índices. Consideremos aqui apenas índices numéricos.

Suponhamos que essa matriz seja armazenada na memória do PO com a seguinte ordenação dos seus elementos, em posições de memória de endereços crescentes:

$M[i_1,...,i_n]$, $M[i_1,...,i_n+1]$,..., $M[i_1,...,s_n]$,
$M[i_1,...,i_{n-1}+1,i_n]$, $M[i_1,...,i_{n-1}+1,i_n+1]$,..., $M[i_1,...,i_{n-1}+1,s_n]$
...
$M[i_1,...,s_{n-1},i_n]$, $M[i_1,...,s_{n-1},i_n+1]$,..., $M[i_1,...,s_{n-1},s_n]$
⋮
$M[s_1,...,s_{n-1},i_n]$, $M[s_1,...,s_{n-1},i_n+1]$,..., $M[s_1,...,s_{n-1},s_n]$

Com essa disposição na memória, podemos agora localizar um elemento qualquer, supondo que t seja o número de palavras necessário para armazenar cada elemento, isto é, o espaço necessário no PO para um objeto do tipo T, com a seguinte fórmula, onde END(x) representa o endereço de x.

$$End(M[j_1,j_2,...,j_n]) = End(M[i_1,i_2,...,i_n]) + (\{\sum_{p=1}^{n-1} \prod_{q=p+1}^{n} (s_q - i_q + 1)\ (j_p - i_p)\} + j_n - i_n)t$$

Se n = 1, a parte da somatória não é usada.

Pode-se separar da somatória uma parcela com os elementos constantes desta fórmula, e subtrair-se do endereço do primeiro elemento o espaço de memória resultante dessa parcela, obtendo-se

$$End(M[j_1,j_2,...,j_n]) = End(M[0,0,...,0]) + (\{\sum_{p=1}^{n-1} \prod_{q=p+1}^{n} (s_q - i_q + 1)j_p\} + j_n)t$$

onde M[0,0,...,0] é a denominada *origem virtual* da matriz, isto é, onde "estaria" o elemento com todos os índices nulos e cuja fórmula damos adiante; ela é virtual pois eventualmente esse elemento pode não existir.

As produtórias podem também ser representadas pela seguinte definição indutiva:

$d_n = 1$
$d_p = d_{p+1}\ (s_{p+1} - i_{p+1} + 1)$, $p = n-1, n-2, ..., 1, 0$.

A somatória pode agora ser reescrita como:

$\sum_{p=1}^{n-1} d_p j_p$. Note-se que d_0 dá o número total de elementos da matriz.

O compilador pode calcular as produtórias $d_0, d_1, ..., d_{n-1}$ logo após a declaração da matriz, pois seus fatores dependem exclusivamente dos limites inferior e superior de cada dimensão, que em PASCAL devem ser constantes. Ele também pode calcular a origem virtual, da seguinte maneira:

$$End(M[0,0,...,0]) = End(M[i_1,i_2,...,i_n]) - (\sum_{p=1}^{n} d_p i_p)t$$

Para gerar código-objeto para o cálculo do endereço de um elemento $M[j_1,j_2,...,j_n]$ é portanto necessário conhecer exclusivamente a seqüência $(n+1, M[0,0,...,0], d_1, d_2, ..., d_{n-1})$, a qual é denominada de *"dope vector"* (que abreviaremos por DV) e os índices $j_1, j_2, ..., j_n$. Conhecido o "dope vector", o cálculo do endereço de um elemento qualquer da matriz é feito gerando-se no PO as instruções de multiplicação e soma para calcular

$$End(M[j_1,j_2,...,j_n]) = End(M[0,0,...,0]) + d_1j_1 + d_2j_2 + ... + d_{n-1}j_{n-1} + j_n \quad (1)$$

Quando introduzimos as informações da TIR para matrizes (v. 6.3h), foram armazenados apenas os limites i_q e s_q. A partir deles podem ser calculados os elementos do DV que poderiam substituir os elementos da TABARR.

Se a declaracão for do tipo
array [...] **of array** [...] **of** ...
ao atingir-se o segundo **array** pode-se continuar o primeiro **array** como se as dimensões do segundo fossem concatenadas às do primeiro, continuando a montagem do DV que será um só para as duas matrizes. Isso pode ser generalizado, evidentemente, para casos de concatenação de mais de duas matrizes. Essa estratégia está de acordo com o manual da PASCAL /J-W 74/ onde se afirma que M[i] [j] pode ser representado como M[i,j].

Vejamos como as rotinas semânticas geram còdigo para variáveis indexadas usando o DV em um caso simples. Suponhamos que os índices sejam todos inteiros: expressões, variáveis (inclusive indexadas) ou constantes. Suponhamos também que se trate de uma variável indexada que não é campò de registro (como B em A.B[j]) e que os índices não são precedidos de apontadores (como P1[j]). Nesse caso certamente passamos por VAIDEN do subgrafo de "factor" (v. apêndice I), tendo em seguida desviado para "infipo". Seja M o identificador desse VAIDEN. Na PSE, em paralelo a VAIDEN, temos o índice da TIR onde se encontra a entrada de M; suponhamos que no campo INDTT haja um índice para uma entrada de uma matriz unidimensional, TABDOP, onde são guardados seqüencialmente os DVs no compilador; TABDOP substitui portanto TABARR.

O endereço resultante da fórmula (1) acima será calculado passo a passo, com os resultados intermediários e final armazenados em um temporário, que é o próximo disponível, por exemplo T_k. Se em algum dos índices de M ocorrer uma variável indexada, esta usará outro temporário; pelas regras de uso de temporários vistas em 7.3, não é necessário guardar K, pois no inícic da compilação de cada índice ter-se-á liberado os temporários a partir de K + 1 inclusive. No entanto, é necessário guardar o índice p do elemento d_p do DV sendo usado. Esse índice será guardado em paralelo ao primeiro '[' reconhecido em "infipo"; com isso produzimos seu empilhamento, o que é necessário no caso de haver variáveis indexadas dentro dos índices de M. Lembramos que o índice do início do DV de M na TABDOP está armazenado em paralelo a VAIDEN.

A RS_{165} de '[' em "infipo" deve testar se se trata da primeira execução para M; isso pode ser detectado verificando-se se em PSE[TOPPS-1] há uma entrada com um par STIR,i e em TIR[i] há um VAIDEN com tipo ARRAYT. Se não se tratar da primeira execução de RS_{165}, nada é feito: trata-se de caso como $M[j_1]\ [j_2]$ e os próximos índices serão tratados como continuação da seqüência dos primeiros. Se se tratar de primeira execução, são feitas as seguintes ações:

i) Inicialização do temporário T_k. É obtido o índice k do próximo temporário disponível (v. 7.3) e em seguida é feita a inicialização de T_k a partir da origem virtual obtida do DV de M, que está em

TABDOP[TIR[PSE[TOPPS-1].IND].INDTT + 1] por meio de

```
LDA     = M0
STA     Tk
```

onde M_0 é uma constante numérica igual a M[0,0,...0]; para facilitar a compreensão, todos os elementos do DV de M serão gerados como constantes no PO.

ii) Empilhamento de índice p para o DV. Faz-se PSE[TOPPS].IND: = 2, empilhando assim o valor de p em paralelo a "["; note-se que o último elemento do DV usado foi o de índice p = 2.

A RS_{166} usa o resultado da "expr" de seu nó, que será uma variável, uma constante ou um temporário (neste caso será certamente o T_{k+1}); suponhamos que o endereço desse resultado no PO seja E. O conteúdo de p armazenado em paralelo a "[" é retirado, usando-se para isso o índice para a PSE armazenado no topo da pilha K do analisador (v. 5.7), que aponta para o primeiro símbolo de "infipo", isto é, faz-se p: = PSE[K[TOPO].R].IND. Deve-se agora obter o conteúdo do primeiro elemento do DV de M, isto é, n + 1, apontado pelo INDTT da variável M da TIR; a entrada de M na TIR está em paralelo a VAIDEN, isto é, uma célula antes do "[" onde se encontrava p, de modo que o valor de n + 1 é encontrado em TABDOP[p_0] onde

p_0 é o conteúdo de TIR[PSE[K[TOPO].R-1]IND].INDTT. Faz-se p:=p+1. Se $p<p_0+n$, usando o último temporário ocupado, que será automaticamente T_k, gera-se

```
LDA     =d_p
MPY     E           /* d_p,j_p */
ADD     T_k
STA     T_k
```

e a seguir guarda-se p em paralelo a "[", isto é, em PSE[K[TOPO].R].IND. Se $p=p_0+n$, chegamos ao último índice; gera-se

```
LDA     E           /* j_n */
ADD     T_k
STA     T_k
```

Se $p>p_0+n$, há um erro: mais índices na variável indexada do que foram declarados na matriz.

Como resultado, teremos em T_k o endereço do elemento da matriz referenciado pela variável indexada. Acabado o reconhecimento de "infipo", retorna-se a "factor", onde a RS_{167} reconhece que o VAIDEN corresponde a uma variável indexada; isso é feito consultando-se o tipo (que no caso será ARRAYT) da entrada da TIR indicada na célula da PSE paralela ao VAIDEN, isto é, em PSE[TOPPS-1]. Verificando ser uma variável indexada, A RS_{167} gera

```
LDA     I T_k
STA     T_k
```

onde o 'I' da LDA indica endereçamento indireto. Com isso, o valor do elemento da matriz é colocado em T_k. Logicamente isso só vale se cada elemento ocupar uma só palavra; se o tipo T dos elementos da matriz ocupar t palavras, seria necessário usar t temporários. O tipo T estava guardado na TABARR, e deve ser guardado no DV se este for usado em lugar daquela.

Uma outra possibilidade teria sido a de inicializar T_k com zero em lugar de M_0, através da RS_{165}; a RS_{167} geraria então

```
LDA     T_k
STA     1           /* T_k → indexador 1 */
LDA   1 M_0
STA     T_k
```

tendo-se agora usado o indexador 1.

A verificação de que houve um número insuficiente de índices na variável indexada pode ser feita também pela RS_{167}; ao ser executada, o conteúdo da PSE que estava em paralelo a '[', isto é, o valor de p, estará agora em paralelo a "infipo" de "factor". Se $p=p_0+n$ o número de índices foi correto, caso contrário houve índices a menos (o excesso de índices é detectado pela RS_{166} como mostrado anteriormente).

O esquema que mostramos aqui teve caráter didático; pode-se modificá-lo de várias maneiras a fim de gerar código mais eficiente, por exemplo eliminando-se LDA's e STA's supérfluos, usando-se mais indexadores etc.

b) Índices não-inteiros

Quando os índices não são inteiros são necessariamente elementos de uma seqüência ("subrange") de constantes simbólicas ou de caracteres; é necessário convertê-los para índices inteiros para se poder calcular tanto o DV durante a compilação como os índices no PO. Isso é feito varrendo-se a seqüência de constantes simbólicas e contando-se o número de elementos até encontrar-se o procurado ou fazendo uma conversão direta no caso de caracteres, como já indicamos em $7.8a_2$.

A RS_{166} é que deve produzir o código-objeto adequado para a conversão, no caso de detectar que o tipo de "expr" é SCIDT ou STRCTT (v. 6.3d e 6.3f).

c) Campos de registros

Examinemos um caso como A.M[j_1,j_2]. Aqui, A é o VAIDEN reconhecido no subgrafo de "factor"; M é reconhecido em "infipo" como FIIDEN; a seguir começa o reconhecimento

dos índices, pela volta ao início de "infipo". Nesse exemplo, o compilador pode calcular o DV da matriz sem problemas, durante a declaração da mesma; a origem virtual agora deve ser calculada em relação ao início do campo M, que é conhecido pelo compilador. A diferença é que agora a RS_{165} encontra à esquerda de '[' um FIIDEN e não um VAIDEN. Mas o FIIDEN tem as informações sobre o DV como se fosse um VAIDEN, inclusive seu tipo na TIR será ARRAYT, de modo que o tratamento é de todo idêntico aos casos vistos anteriormente. O mesmo se passa com A.B.M[...], e assim por diante.

A situação se altera quando temos algo como $A[j_1,j_2].M[j_3,j_4]$, isto é, uma matriz de registros que por sua vez tem como um de seus campos uma matriz. Nesse caso, a origem real de M, isto é, o endereço de seu primeiro elemento, é o resultado do cálculo do endereço de $A[j_1,j_2]$. Observando-se a definição de origem virtual, dada anteriormente, vemos que ela tem duas partes, a saber:

$End(M[i_1,\ldots,i_n])$ e $(\sum_{p=1}^{n} d_p i_p)t$; a primeira é a origem real; a segunda, o deslocamento em relação a essa origem, isto é, o número de palavras que se deve mover na ordem decrescente dos endereços até atingir-se o ponto da origem virtual. Esse deslocamento não depende da origem real. É ele, portanto, que deve ser colocado no DV no lugar da origem virtual. Ao ser executada, a RS_{168} examina o tipo do elemento anterior do registro (A, no exemplo); é fácil saber se ele era uma variável indexada: em PSE[TOPPS-2], isto é, na célula imediatamente anterior ao '.' deve haver um par STABT,i apontando para ']'. Nesse caso, o endereço do campo está no último temporário ocupado, T_k. São produzidas as instruções no PO

```
LDA     Tk
SUB     d
STA     Tk
```

onde d é o deslocamento da origem virtual, obtida do DV. Daí para frente, tudo se passa como no caso (a) acima.

Em um caso como $M[j_1,j_2].A$, basta gerar no PO uma instrução para adicionar ao endereço de $M[j_1,j_2]$ o deslocamento de A relativo ao início de cada elemento de M. Aquele deslocamento está no campo ENDOBJ da TIR do FIIDEN 'A' (v. 6.3i).

d) Apontadores ("pointers")

Este é o caso de, por exemplo, $P\uparrow[j_1,j_2]$, proveniente das declarações

```
type T: array [1..10,1..20] of integer;
var  P: ↑T;
```

Neste caso é necessário partir-se do endereço da origem real, que é a apontada por P. Esse endereço pode inicializar o temporário T_k usado para o cálculo do endereço do elemento da matriz, como em (a) acima. Por construção (v. 7.7), a RS_{160} de '↑' no subgrafo de "infipo" não produz nenhuma "derreferenciação" quando ná um só '↑'. Portanto, a RS_{165} em '[' é que deve tratar de inicializar o temporário T_k. Ela produz, portanto,

```
LDA     Ep
SUB     d
STA     Tk
```

onde E_p é o endereço de P, do deslocamento da origem virtual como descrito em (a) e T_k o próximo temporário disponível.

Se houver mais do que um '↑', haverá derreferenciações anteriores, como descrevemos em 7.7; nesse caso E_p será o próprio T_k.

7.10 CAMPOS DE REGISTROS

Vejamos como compilar campos de registros, que podem ser usados em expressões ou à esquerda do sinal ':=' (comando de atribuição ou **for**).

No caso de expressões, a RS_{167} do nó "infipo" de "factor" reconhece que se trata de um registro consultando PSE[TOPPS-1], em paralelo a VAIDEN, onde se encontra um par STIR,i indicando a entrada da TIR onde está esse VAIDEN. No caso de ser um registro, o campo TIPO desse VAIDEN conterá RECT (v. 6.3i).

Nos campos do registro, isto é, entradas com classe FIIDEN na TIR, tem-se em ENDOBJ o endereço relativo ao início do registro, que também podemos chamar de "deslocamento". Devido a isso, ao final do reconhecimento de um registro, deve-se ter em paralelo à célula de "infipo" em "factor" o total do deslocamento em relação ao início do registro, independente do número de campos usados, por exemplo em A.B.C. Nesse exemplo, o reconhecimento de B.C em "infipo" deve produzir o deslocamento de C em relação ao início de A. Para isso, a RS_{169} no nó '.' deve verificar se se trata do primeiro campo (como B, no exemplo); isso pode ser feito verificando-se se a célula PSE[TOPPS-1] contém uma referência à TIR onde se encontra um VAIDEN, pois se for um segundo campo (como C) essa célula faz referência a um FIIDEN ou a outros símbolos nos casos de campos indexados ou com apontadores ('↑'). No caso de ser o primeiro campo (como B), a RS_{169} coloca o valor 0 (zero) em PSE[TOPPS].IND, caso contrário ela não faz nada. Posteriormente, a RS_{168} deve incrementar esse valor para cada novo campo reconhecido; a célula da PSE que contém o valor do deslocamento até esse momento é a de índice K[TOPO].R (v. 5.7), isto é, a primeira que foi reconhecida ho subgrafo de "infipo". Note-se que se em FIIDEN houver uma referência a um tipo matriz (tipo ARRAYT), será dado o tratamento visto no item anterior.

Em PASCAL, pode-se ter campos de registros com mesmo identificador que outros objetos, como por exemplo variáveis simples, ou outros campos. Assim, é permitido declarar

var A: **record** B: T_1;
C: T_2 **end**;
B: **record** A: T_3;
B: T_4;
C: **record** A: T_5; B: T_6 **end**
end;
C: T_7

Teremos assim A e C ocorrendo três vezes na TIR e B quatro vezes. Ao se buscar um campo de registro é necessário, portanto, percorrer a "árvore" de definição do registro, usando a TABREC em lugar de buscar indiscriminadamente o seu identificador na TIR (v. 6.2). Isso é feito através da RS_{169}, no nó '.' de "infipo". Essa rotina deve forçar a busca do próximo IDEN reconhecido pelo analisador léxico apenas entre as entradas da TIR apontadas pelos elementos da entrada da TABREC correspondentes ao campo anterior ou ao identificador do registro, que precederam o '.'. Assim, ao iniciar-se o reconhecimento de B.C.B proveniente da declaração acima, o primeiro B é encontrado normalmente como VAIDEN na TIR (sendo que em sua busca todas as entradas com FIIDEN devem ser desprezadas). O campo INDTT desse B é o índice para a TABREC, e deve ser guardado em uma variável do compilador, por exemplo em INTARE, pela RS_{170} no nó VAIDEN de "factor"; a RS_{169} usa o conteúdo de INTARE, que aponta para a TABREC onde estão índices das entradas da TIR dos identificadores de campos A, B e C. A RS_{168}, no nó FIIDEN de "infipo", guarda por sua vez em INTARE o apontador INDTT para a TABREC que está naquele C. Na passagem seguinte, a RS_{169} força a busca somente nas entradas da TIR cujos índices são dados pelos elementos da TABREC apontados por INTARE, isto é, somente em A e B, encontrando-se assim o último campo de B.C.B.

Ao se encerrar o reconhecimento de "infipo", a célula do topo da PSE conterá no campo IND o deslocamento total d correspondente ao último campo referenciado. Retornando-se a "factor" a RS_{167} detecta que se trata de um registro verificando o tipo da entrada da TIR para o VAIDEN que está em PSE[TOPPS-1]. Neste caso, são geradas as instruções

```
LDA     E+d
STA     Tk
```

onde E é o endereço do início do registro, obtido no campo ENDOBJ da entrada do VAIDEN e T_k o próximo temporário disponível. Essas instruções só valem para o caso do tipo do campo

conter uma só palavra; em caso contrário é necessário produzir uma seqüência de pares LDA, STA ou uma malha varrendo todas as palavras envolvidas, usando-se um indexador, como será visto em 8.2.

CAPÍTULO 8

GERAÇÃO DE CÓDIGO-OBJETO — II

8.1 INTRODUÇÃO

Neste capítulo abordaremos a geração de código para os comandos do subgrafo de "statm", isto é, as estruturas de controle da linguagem PASCAL. Não serão abordados os casos do **go to**, que já foi tratado no capítulo anterior (7.2), de chamada de procedimentos e de comandos de atribuição a identificadores de função que serão abordados no próximo capítulo.

No capítulo anterior vimos que na geração de código-objeto para expressões são usadas informações obtidas através de dados armazenados na pilha "semântica". Novas informações terão que ser armazenadas nessa pilha, para se poder compilar adequadamente os comandos de definição recursiva. Isso se passa com todos os que contêm o não-terminal "statm" Devido a esse fato, será necessário empilhar algumas informações para não perdê-las ao se interromper a análise para compilar o "statm" interior; para isso será empregada a PSE. Evidentemente, a cada comando poder-se-ia associar uma pilha diferente, como por exemplo uma pilha para o comando "while", uma outra para o "if" etc. Manteremos a PSE em "paralelo" à pilha sintática PS, que não precisa ser implementada, servindo apenas como referência para a localização dos elementos da PSE.

Não nos preocuparemos em gerar o código mais curto e mais eficiente; a preocupação geral é de mostrar didaticamente como podem ser gerados os códigos para os diversos comandos.

8.2 COMANDO DE ATRIBUIÇÃO

O código-objeto para comandos de atribuição é gerado pela RS_{80}, no nó "expr" dos ramos VAIDEN e FUIDEN. Há duas espécies de atribuição: para uma variável, ou para o identificador de uma função; neste último caso a RS_{80} detecta em PSE[TOPPS-2] um par STIR,i que aponta para uma entrada na TIR com classe FUIDEN. O lado direito sendo uma expressão, é obrigatoriamente i) constante ou variável simples, casos em que o conteúdo de PSE[TOPPS] será do tipo STIR,j apontando para entradas na TIR de classes COIDEN ou VAIDEN respectivamente; ii) variável indexada, campo de registro, apontador ou expressão com operadores, casos em que o conteúdo de PSE[TOPPS] indicará um temporário T_k onde terá sido armazenado o valor desses elementos, como vimos no capítulo anterior. Se o lado esquerdo da atri-

buição contiver indexação ou mais do que uma derreferenciação ('1'), então k = 1, caso contrário k = 0. O tipo do lado direito é dado pela sua entrada na TIR (v. 6.3) ou pelo tipo do temporário (v. 7.3).

Se o lado esquerdo for uma variável simples como A, em PSE[TOPPS-3] a RS_{80} encontra a indicação de uma entrada na TIR com classe VAIDEN, tipo TYIDT (v. 6.3b) e o tipo apontado por INDTT. É preciso verificar se seu tipo é o mesmo do que o do lado direito, e em seguida gerar simplesmente LDA E_2; STA E_1, onde E_1 e E_2 são os endereços dos objetos do lado esquerdo (obtido na TIR) e direito (obtido da TIR, ou temporário).

Se o lado esquerdo for uma variável indexada como $M[j_1,j_2]$, em paralelo a "infipo", isto é, em PSE[TOPPS-2] teremos um temporário T_n com seu endereço. Nesse caso, a RS_{80} gera

```
LDA     E
STA  I  ARTEMP + N
```

onde E é o endereço do lado direito, e a STA emprega endereçamento indireto.

Se o lado direito for uma variável indexada, a LDA deve usar endereçamento indireto sobre o temporário onde foi colocado o endereço dessa variável.

Estas considerações valem somente no caso em que o tipo dos dois lados necessitar de uma única palavra para seu armazenamento no PO. Caso contrário, é necessário gerar uma seqüência de LDA's e STA's, ou uma malha de repetição dessas instruções usando endereços modificados por algum registro indexador, que é incrementado a cada nova execução da malha.

Por exemplo, para um tipo que necessita n palavras, poderiam ser geradas as seguintes instruções, que usam o indexador 1 (v. apêndice III):

```
      LDA       =n          /* número de palavras a serem atribuídas */
      STA       1           /* inicializa o indexador 1 com n */
RIm   LDA   1   E2-1
      STA   1   E1-1
      MNX   1   1
      BRN       RIm
```

onde E_1 e E_2 são os endereços das primeiras palavras indicadas pelos lados esquerdo e direito da atribuição, respectivamente.

Se o lado direito ou o esquerdo tiverem usado um temporário, este deve ser liberado.

O caso do lado esquerdo ser um identificador de função FUIDEN será examinado no item sobre compilação de procedimentos e funções.

8.3 COMANDO "WHILE"

Neste comando, como em outros que veremos adiante, será gerado um "rótulo interno" RIm, onde m é o conteúdo de uma variável inteira do compilador, o contador de rótulos internos ROTINT. Inicialmente, ROTINT deve ser inicializado com o valor 0 (zero).

A execução do comando "while" no PO deve testar o resultado de uma expressão ("expr") booleana entre **while** e **do** (v. grafo sintático no apêndice I). Se ela tiver valor "true", é executado o comando ("statm") depois do **do**, caso contrário o comando "while" é abandonado; após a execução do "statm" é feito um desvio para calcular e testar novamente a expressão. Na fig. 8.1 damos o código gerado por esse comando com a indicação das rotinas "semânticas" que geram as diferentes instruções. Representamos por E o endereço do resultado da expressão (variável, temporário etc.).

Vejamos sucintamente as ações das rotinas "semânticas".

- RS_{81}: i) faz ROTINT: = ROTINT + 1; ii) armazena em PSE[TOPPS], isto é, em paralelo a **while**, o par SOBJ,m, onde m é o conteúdo de ROTINT; iii) gera a instrução do PO.
- RS_{82}: i) verifica se o resultado da "expr" é de tipo booleano, emitindo mensagem de erro em caso contrário; ii) faz ROTINT: = ROTTINT + 1; iii) armazena em PSE[TOPPS], isto é, em paralelo ao **do**, o par SOBJ,n, onde n é o conteúdo de ROTINT (como expressões não

usam rótulos internos, n = m + 1); iv) gera as instruções; v) se o resultado da "expr" for um temporário, ele é liberado, isto é, é feito TMPATV: = TMPATV-1 (v. 7.3).

— RS_{83}: i) obtém m de PSE[TOPPS-3], isto é, em paralelo ao **while**; ii) obtém n de PSE[TOPPS-1], isto é, em paralelo ao **do**; iii) gera as instruções.

```
Rlm  EQ      *      } RS81
     "expr"                    /* código da expressão */
     LDA     E
     BZR     Rln    } RS82     /* sai se "false" */
     "statm"                   /* código do comando */
     BRN     Rlm
Rln  EQ      *      } RS83
```

Fig. 8.1

É interessante notar a necessidade de empilhamento dos números m e n dos rótulos internos, o que foi feito usando-se a PSE. Essa necessidade advém do fato de que dentro do "statm" que segue o **do** podem haver outros comandos "while"; naturalmente, poderíamos ter usado uma pilha especial só para os comandos "while", mas dessa maneira aproveita-se a pilha semântica, que também será usada em outros comandos.

8.4 COMANDO "REPEAT"

A execução deste comando é análoga à do "while", com a diferença de que o teste é feito após a seqüência de comandos, a qual, ao contrário do "while", pode conter mais do que um comando.

Na fig. 8.2 apresentamos as instruções geradas pelas rotinas "semânticas"; novamente E representa o endereço do resultado de "expr"

```
Rlm  EQ      *      } RS84
     "statm"
     :
     "statm"
     "expr"
     LDA     E
     BZR     Rlm    } RS85
```

Fig. 8.2

O funcionamento das rotinas "semânticas" é o seguinte:

— RS_{84}: i) faz ROTINT: = ROTINT + 1; ii) armazena SOBJ, m em PSE[TOPPS], isto é, em paralelo a **repeat**, onde m é o conteúdo de ROTTINT; iii) gera a instrução.

— RS_{85}: i) verifica se a "expr" é do tipo booleano, emitindo mensagem de erro em caso contrário; ii) obtém m de PSE[K[TOPO].R] ou de PSE[K[TOPO].R + 2], isto é, em paralelo a **repeat**; iii) gera as instruções; iv) se o resultado da "expr" for um temporário, ele é liberado, isto é, faz-se TMPATV: = TMPATV-1 (v. 7.3).

Na RS_{85} foi necessário recorrer ao apontador do topo da pilha K do analisador sintático, que localiza o primeiro elemento do não-terminal sendo analisado, na PS ou na PSE (v. 5.7). Isso se deve ao fato de que um número qualquer de "statm"s pode ter sido reconhecido dentro do comando "repeat". Se esse comando tiver sido precedido de um rótulo, deve-se empregar o segundo índice da PSE visto acima no item (ii) da RS_{85}, caso contrário (isto é, sem rótulo), o primeiro índice. Pode-se distinguir um caso do outro verificando se ocorreu um ':' em PSE[K[TOPO].R + 1].

8.5 COMANDO "IF"

Esse comando tem duas formas: com ou sem parte **else**, isto é, diferenciando os comandos

if "expr" **then** "statm"$_1$

e

if "expr" **then** "statm"$_1$ **else** "statm"$_2$

O código gerado em cada caso difere na medida em que no segundo caso é necessário gerar um desvio incondicional logo depois de "statm"$_1$, para que "statm"$_2$ não seja também executado. Para distinguir os dois casos introduzimos no grafo um λ-nó logo depois do segundo "statm", de modo a poder executar uma rotina semântica nesse ponto. Vejamos como tratar do segundo caso e, em seguida, do primeiro.

a) **if**...**then**...**else**...

O código gerado é o da fig. 8.3, onde E representa o endereço do resultado da "expr". Vejamos, sucintamente, as rotinas semânticas desse caso:

- RS_{86}: i) verifica se o tipo de "expr" (em PSE[TOPPS-1] é booleano: ii) faz ROTINT: = ROTINT + 1; iii) armazena o par SOBJ,m em PSE[TOPPS], isto é, em paralelo a **then**, onde m é o conteúdo de ROTINT: iv) gera as instruções no PO; v) se o resultado da "expr" é um temporário, ele é liberado, isto é, faz-se TMPATV: = TMPATV-1 (v. 7.3).

```
        "expr"                          /* código da expressão */
        LDA     E       }
        BZR     Rlm     }  RS86
        "statm"1                        /* código do 1º comando */
        BRN     Rln     }  RS87
Rlm     EQ      *
        "statm"2                        /* código do 2º comando */
Rln     EQ      *       }  RS88
```

Fig. 8.3

- RS_{87}: i) faz ROTINT: = ROTINT + 1; ii) armazena o par SOBJ,n em PSE[TOPPS], isto é, em paralelo a **else**, onde n é o conteúdo de ROTINT; iii) obtém o valor m de PSE[TOPPS-2], isto é, em paralelo a **then**; iv) gera as instruções.
- RS_{88}: i) obtém n de PSE[TOPPS-1], isto é, em paralelo a **else**; ii) gera a instrução.

b) **if**...**then**...

O código gerado é o da fig. 8.4. A única diferença reside na não-execução da RS_{87}. A RS_{88} é executada exatamente como no caso anterior, sendo que neste caso ela obtém m de PSE[TOPPS-1], em paralelo a "then".

```
        "expr"
        LDA     E       }  RS86
        BZR     Rlm     }
        "statm"1
Rlm     EQ      *       }  RS88
```

Fig. 8.4

Note-se que os empilhamentos dos números dos rótulos garantem a compilação correta no caso de ocorrerem outros comandos "if" em "statm"$_1$ ou "statm"$_2$.

8.6 COMANDO "FOR"

O comando "for" da PASCAL é uma grande simplificação do correspondente introduzido no ALGOL-60 /NAU 63/. Na PASCAL o incremento e decremento são sempre "unitários", e a distinção entre um caso e outro é explícita, feita por meio das palavras reservadas **to**

e **downto**, respectivamente. Mas a maior simplificação reside no fato de que o valor da variável de controle (VAIDEN com "infipo", no grafo), o valor inicial (primeiro "expr") e o valor final (segundo "expr") não devem ser modificados durante a execução do "statm" sendo repetido. Em ALGOL-60 isso não ocorre, e há necessidade de se prever essas alterações, inclusive durante o próprio teste de parada da repetição, devido a possíveis efeitos colaterais de funções que podem ocorrer nas expressões.

Citamos acima que os incrementos e decrementos são unitários; no caso da variável de controle ser do tipo inteiro, o efeito é evidente; se a variável de controle for do tipo seqüência de constantes simbólicas, é necessário percorrer a seqüência desde o elemento correspondente ao valor inicial até o elemento do valor final.

a) Variável de controle de tipo inteiro

Para tornar a explicação mais compreensível, damos na fig. 8.5 a seqüência de instruções geradas em cada rotina "semântica". Apresentamos o caso **to**; as modificações das instruções BNG e ADD para o caso **downto** são dadas como comentário. E_1 e E_2 representam os endereços dos resultados das expressões dos valores inicial e final cujo código é representado por "expr"$_1$ e "expr"$_2$ respectivamente. O endereço da variável de controle é representado por V; T_k é o temporário (próximo disponível) onde é colocado o valor de "expr"$_2$, isto é, o valor final para teste. Por motivos didáticos não apresentaremos um código otimizado, isto é, com um número mínimo de instruções.

Vejamos sucintamente o funcionamento das rotinas "semânticas".

— RS_{89}: i) testa a consistência dos tipos de "expr"$_1$ e VAIDEN (PSE[TOPPS] e PSE[TOPPS-3], respectivamente); ii) gera as instruções; iii) se o resultado de "expr"$_1$ é um temporário, ele é liberado, isto é, faz-se TMPATV: = TMPATV-1 (v. 7.3).

```
        "expr"1
        LDA     E1          } RS89      /* valor inicial */
        STA     V           }
        "expr"2
        LDA     E2          }           /* valor final */
        STA     Tk          }
        LDA     V           }
RIm     EQ      *           } RS90      /* comparação */
        SUB     Tk          }
        BPS     RIn         }           /* downto: BNG */
        "statm"             }
        LDA     V           }           /* incremento */
        ADD     =1          }           /* downto: SUB */
        STA     V           } RS91      }
        BRN     RIm         }           }
RIn     EQ      *           }           }
```

Fig. 8.5

— RS_{90}: i) testa a consistência do tipo de "expr"$_2$ com o de "expr"$_1$; ii) se o resultado de "expr"$_2$ é um temporário, ele é liberado; iii) incrementa o contador de temporários, obtendo k; iv) faz ROTINT: = ROTINT + 1; v) armazena o par SOBJ,m em PSE[TOPPS-3], isto é, em paralelo a "expr"$_1$, onde m é o conteúdo de ROTINT; vi) faz ROTINT: = ROTINT + 1; vii) armazena o par SOBJ,n em PSE[TOPPS], isto é, em paralelo a **do**, onde n é o conteúdo de ROTINT (n = m + 1); viii) gera as instruções (o endereço de V é obtido a partir de VAIDEN e "infipo", isto é, de PSE[TOPPS-6] e PSE[TOPPS-5]), verificando se em PSE[TOPPS-2], isto é, em paralelo a **to** ou **downto** há uma referência a **to** na TABT, caso em que a última instrução gerada é uma BPS, caso contrário é uma BNG.

— RS_{91}: i) obtém o endereço de V através de VAIDEN e "infipo" (PSE[TOPPS-7] e PSE[TOPPS-6]); ii) obtém m em PSE[TOPPS-4], isto é, em paralelo a "expr"$_1$; iii) obtém n

em PSE[TOPPS-1], isto é, em paralelo a **do**; iv) gera as instruções, usando ADD se encontrar em PSE[TOPPS-3], isto é, em paralelo a **to** ou **downto**, uma referência a **to** na TABT, caso contrário usa SUB; v) se a VAIDEN for uma variável indexada libera-se o temporário usado no cálculo do endereço da mesma; vi) libera-se mais um temporário (T_K).

Note-se a necessidade de empilharem-se os elementos V, m e n, já que em "statm" podem ocorrer outros comandos "for"

b) Variável de controle de tipo seqüência de constantes simbólicas

Nesse caso é necessário varrer a seqüência de constantes simbólicas armazenada no PO, tomando-se em cada iteração o sucessor (predecessor) do último elemento usado no caso de **to** (**downto**). Como vimos em 7.6b, uma variável que assume valores dessas seqüências contém no PO um número inteiro, correspondente ao número de ordem de alguma constante simbólica da seqüência Se as expressões do valor inicial e final do "for" forem constantes, o compilador fará a busca das mesmas na TIR, usando o valor de INDTT (que é o seu número de ordem) no PO. Dessa maneira, o comando "for", com esse tipo de variável de controle, reduz-se ao caso (a) visto acima.

8.7 COMANDO "WITH"

As variáveis que se seguem à palavra reservada **with** devem ser todas identificadores de registros, isto é, entradas na TIR com classe VAIDEN e tipo RECT. (Lembremos que, por construção, se uma variável é declarada com um tipo T, onde T é identificador de tipo, a sua entrada na TIR conterá nos campos TIPO e INDTT a cópia dos mesmos campos de T, conforme vimos em 6.5 g_3.) Dentro do "statm" do "with", não é necessário especificar aqueles identificadores de registros, podendo-se usar diretamente os identificadores de campos.
Por exemplo, pode-se ter

```
var A:T1;
    B:record C:T2; A:T3; D:record E:T4; F:T5 end end;
begin
    with B do ... A:= ... /* equivale a B.A:= ... */ ...;
```

Por outro lado, **with** B,D **do** ... E:= ... /* equivale a B.D.E:= ... */...;

Vejamos como compilar esse comando. Temos dois casos a considerar: o identificador de registro não é indexado, como nos casos acima, ou é indexado, como no seguinte exemplo.

```
var G:array.[1..10] of record H:T6; I:T7 end;
begin
    ...
    with G[j] do H:= ...
```

A atribuição é equivalente a G[j].H:= ... Note-se que j não deve variar dentro do comando que se segue ao **do**.

a) Identificadores simples de registros

Este caso é detectado pela RS_{92}, verificando que VAIDEN apontado pelo par STIR,i em PSE[TOPPS-1] não tem tipo ARRAYT.

Neste caso, é necessário *inibir* a busca de variáveis com o mesmo nome dos campos do registro cujo identificador consta da lista entre **with** e **do**. No primeiro exemplo acima, deveríamos inibir a busca da variável A, de modo que toda busca de 'A' no campo IDROT da TIR encontrasse o identificador de campo A do registro B. O endereço desse campo relativo ao início do registro já consta da sua entrada na TIR, por construção (v. 6.3i).

Para se conseguir essa inibição, basta fazermos o seguinte: i) Introduzir na TIR o campo adicional INIB de tipo booleano; ele tem dupla função: se uma entrada com classe diferente de FIIDEN tem campo INIB = true, então tudo se passa como se ela não estivesse presente na TIR; se uma entrada com classe FIIDEN tem INIB = false, então ela passa a ser buscada como

uma entrada qualquer. Lembremos que as entradas com FIIDEN não devem ser buscadas em condições normais, pois sua busca se dá sempre através dos apontadores de TABREC (v. 7.10); nessas condições, essas entradas devem ter INIB = true. ii) Cada identificador I simples de registro, reconhecido na lista de identificadores de um **with**, produz as seguintes ações executadas pela RS_{92}:

São localizados os campos mais externos de I (no primeiro exemplo acima, seriam, para B, os campos C, A e D) através da entrada da TABREC correspondente, isto é, apontada por INDTT de I; cada um desses campos tem o valor de seu campo INIB mudado para "false"; ao ser encontrado o identificador Q de um desses campos, a TIR é varrida desde a entrada do mesmo até o topo, nesse sentido, e todos os identificadores com classe VAIDEN com nome (isto é, conteúdo de IDROT) igual a Q tem o valor de seu campo INIB alterado para "true".

A RS_{93} é executada ao chegar-se ao fim do comando onde vale a atuação do "with". Ela deve alterar todos os inibidores que foram posicionados pela execução da RS_{92}. Para saber quais são as entradas da TIR que foram inibidas, e as de classe FIIDEN que foram desinibidas, basta percorrer a PSE, onde ficaram armazenados, em paralelo aos VAIDENs, os pares STIR,i de cada elemento da lista entre **with** e **do**. As células da PSE são as de índice TOPPS-3, TOPPS-6 etc., até que apareça um par STABT,j apontando para **with** na TABT, e TOPPS-4n , $n \geq 1$.

b) Identificadores indexados de registro

Como vimos em 7.9, as variáveis indexadas têm seu endereço calculado por meio de instruções geradas no PO, e que o colocam em um temporário T_k. Vimos, outrossim, em 7.10 que campos de registros indexados, como H em G[j].H produzem apenas uma adição de seu deslocamento ao endereço calculado para G[j] e que foi armazenado em um temporário T_k. Assim sendo, a RS_{92}, após detectar que se trata deste caso, inibe os identificadores com mesmo nome e desinibe os campos do identificador de registro reconhecido (aos quais tem acesso através da TABREC) como no caso (a) acima. Em seguida ela deve adicionar, às informações desses campos na TIR, o número k do temporário onde "infipo" deixou o endereço da variável indexada. Para isso, é necessário acrescentar ainda um campo extra à TIR, onde o compilador coloca o índice k do temporário. Se o identificador de registro de um campo não é indexado, esse campo conterá -1 no lugar de k; portanto, um $k \geq 0$ indica que deve ser adicionado o deslocamento em ENDOBJ ao endereço já calculado e armazenado em T_k. A RS_{93} deve também liberar nesse caso os temporários usados no cálculo do endereço das variáveis indexadas.

Note-se que esse esquema vale também no caso de indexações consecutivas como por exemplo em $X[j_i].Y[j_2].Z[j_3]$, com **with** $X[j_1], Y[j_2]$ **do** $Z[j_3]$: = ... Neste caso, ao se atingir $Y[j_2]$, a entrada de Y já terá índice do temporário onde foi calculado o endereço de $X[j_1]$; esse temporário deve ser usado novamente em todos os campos de Y, isto é, no caso, Z.

8.8 COMANDO "CASE"

Na execução deste comando, a expressão entre **case** e **of**, que denominaremos de "expressão seletora", é calculada e seu valor comparado com as constantes que rotulam cada elemento da lista de comandos e que denominaremos de "constantes de seleção"; é executado o comando para o qual o valor de sua constante seletora coincide com o valor da expressão seletora; depois da execução desse comando, é feito um desvio por sobre todos os outros comandos da lista, isto é, para o fim do comando "case"

Apresentaremos aqui uma solução para compilação deste comando; no fim do item damos uma sugestão sobre outra possível implementação.

O código gerado tem a forma dada na fig. 8.6.

Vejamos, sucintamente, o funcionamento das rotinas "semânticas".

— RS_{94}: i) faz ROTINT: = ROTINT + 1; ii) armazena o par SOBJ,s em PSE[TOPPS], isto é, em

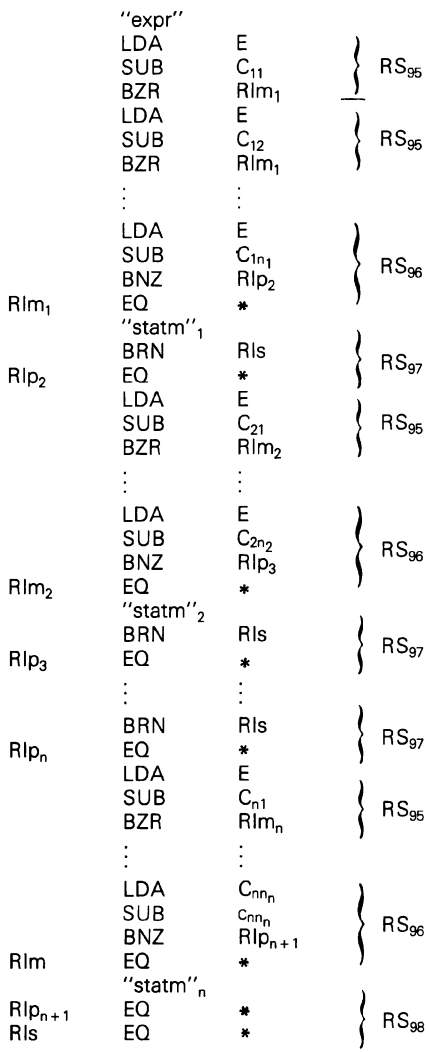


Fig. 8.6

paralelo a **case** — com isso foi empilhado o rótulo do fim do "case"; iii) faz ROTINT := ROTINT + 1 — conteúdo de ROTINT será o valor de m_1.

— RS_{95}: estamos na j-ésima constante do i-ésimo comando; i) obtém o endereço E de "expr" que está em PSE[K[TOPO].R + 1], isto é, em paralelo a "expr" — lembramos que K[TOPO].R aponta para a célula paralela a **case** no caso do "case" não ser precedido de um rótulo, o que pode ser verificado testando se o conteúdo de PSE[K[TOPO].R + 1] não contém uma referência a ':'. Se contiver essa referência, deve-se tomar a célula de índice

K[TOPO].R + 3 (v. 5.7); ii) obtém o endereço C_{ij}, da constante que foi reconhecida, de PSE[TOPPS-2] se PSE[TOPPS-3] contiver uma referência a **of**, ',' ou ';', ou de PSE[TOPPS-1] em caso contrário; iii) gera as instruções, onde m_i do rótulo RIm_i é o conteúdo de ROTINT.

- RS_{96}: estamos no i-ésimo comando; i) obtém E e C_{ij} onde $j = n_i$ como a RS_{95}; ii) gera as instruções, onde m_i é o conteúdo atual de ROTINT, e p_{i+1} é o conteúdo de ROTINT, acrescido de 1; iii) faz ROTINT: = ROTINT + 1; iv) coloca em PSE[TOPPS], isto é, em paralelo a ':', o valor p_{i+1} de ROTINT, para gerar o rótulo que precede o (i + 1)-ésimo comando.
- RS_{97}: i) verifica se não houve um comando antes desse ';', testando se PSE[TOPPS-1] contém uma referência a ';' ou **of**; se esse for o caso, gera RIm_i EQ * e vai para (v), caso contrário acabou de ser gerado o código para o i-ésimo comando; ii) obtém s de PSE[K[TOPO].R], isto é, em paralelo a **case**, aí colocado pela RS_{94}; iii) obtém p_{i+1} em PSE[TOPPS-2], isto é, em paralelo a ':'; iv) gera as instruções; v) faz ROTINT: = ROTINT + 1, cujo conteúdo passa a ser m_{i+1}, do rótulo do próximo comando.
- RS_{98}: executa os passos (i) a (iii) de RS_{97}; iv) gera as instruções; v) se o resultado de "expr" que está em PSE[K[TOPO].R + 1] for um temporário, libera esse temporário, isto é, faz TMPATV: = TMPATV-1.

Note-se que houve necessidade de se empilhar s e p_i, já que nos comandos da lista de um "case" pode ocorrer qualquer comando que gere rótulos internos, em particular um novo "case". Por outro lado, não foi necessário empilhar m_i, pois entre as constantes seletoras de um comando e este último, não pode ocorrer um outro comando.

A implementação que descrevemos acima consiste, em síntese, na geração de uma seqüência de comandos **if**...**then**...**else if**. Uma outra possibilidade de implementação é a de se produzir uma tabela em que cada entrada contém dois campos: um, o endereço de uma constante seletora, e o outro, o endereço do comando que ela rotula. Calculada a expressão seletora, varre-se a tabela em busca da constante seletora com mesmo valor; localizada esta última, toma-se o campo de endereço de comando, fazendo-se um desvio indireto para o comando que foi selecionado. O problema dessa solução é que não se sabe *a priori*, em uma compilação de um só passo, quantas constantes seletoras serão usadas, de modo que não se pode reservar de antemão o espaço necessário para a tabela; por outro lado, outros comandos "case" dentro dos comandos selecionados podem exigir a interrupção da construção de uma tabela, a fim de uma outra ser construída. Uma solução para esse problema é fixar-se de antemão um tamanho para as tabelas; no entanto essa solução tende a prejudicar a grande vantagem deste método, que é uma eventual economia de espaço. Nas linguagens em que o comando "case" funciona com constantes seletoras de tipo inteiro e de valores obrigatoriamente consecutivos (por exemplo, o comando "computed go to" do FORTRAN) o método da tabela deve ser empregado; neste caso o próprio valor da expressão seletora serve como índice para a tabela, cujas entradas reduzem-se ao campo de endereços de comandos; dessa maneira obtém-se uma economia no tempo de processamento.

8.9 PROJETO — PARTE VII: GERAÇÃO DE CÓDIGO SEM PROCEDIMENTOS

Nesta parte do projeto devem ser programadas as rotinas semânticas de geração de código-objeto para as expressões e os comandos vistos neste capítulo e no anterior. Sugerimos a seguinte seqüência de implementação:

a) Rótulos e "goto"
b) Comando de atribuição com lado direito constando de uma variável simples
c) Expressões com variáveis simples de tipo inteiro e booleano; temporários
d) Comandos "if", "while" e "repeat"
e) Comandos "for" e "case"
f) Variáveis indexadas
g) Campos de registros. Registros

h) Apontadores
i) Conjuntos
j) Comando "with"

Devem ser introduzidos dois novos comandos de controle do compilador PSEON e PSEOFF, especificando o trecho em que deve ser exibida a pilha semântica, de maneira análoga à pilha sintática (v. 5.11 parte V), para a fase de testes.

CAPÍTULO 9

COMPILAÇÃO DE PROCEDIMENTOS

9.1 INTRODUÇÃO

Procedimentos ("procedures") constituem uma característica essencial das linguagens de programação. De um lado, permitem a programação modular, onde cada módulo pode ser desenvolvido com um grande grau de independência; esse grau cresce na razão inversa do número de variáveis globais empregadas dentro do procedimento, isto é, variáveis declaradas no programa principal ou em procedimentos dentro dos quais o módulo sendo programado está declarado. Os parâmetros do procedimento deveriam, idealmente, ser o único meio de comunicação do procedimento com outros módulos ou com o programa principal; os programadores deveriam, portanto, receber apenas as informações sobre o tipo e a ordem dos parâmetros, podendo assim desenvolver seu módulo sem se preocuparem com o restante do programa. Por outro lado, em PASCAL os procedimentos são a única maneira de introduzir estrutura de blocos no uso dos identificadores (v. 6.6), permitindo assim que blocos (ou procedimentos) de mesmo nível de encaixamento repartam mesmas partes da memória do PO (programa-objeto) dedicada à área de dados. Procedimentos podem ainda facilitar a gestão de memória não só na área de dados, mas também na área de instruções do PO. Trata-se do que é conhecido em Sistemas Operacionais por "Segmentação". Os procedimentos estabelecem limites naturais para a segmentação; admitindo-se que o processamento do PO permanece um certo período de tempo em cada procedimento, pode-se repartir áreas de código entre procedimentos de mesmo nível de encaixamento.

No que segue, usaremos a palavra "procedimento" em todas as considerações que se aplicam tanto a procedimentos como a funções no sentido da PASCAL. Os casos de aplicação distinta serão mencionados explicitamente.

9.2 CONCEITOS BÁSICOS

Vejamos alguns conceitos básicos sobre procedimentos e funções e seus parâmetros. Faremos menção apenas a procedimentos, mas tudo o que será exposto vale também para funções.

Parâmetros subdividem-se quanto ao que chamaremos de "espécie" em dois grupos distintos: *parâmetro atual*, aquele que aparece na chamada de um procedimento e *parâmetro formal*, aquele que aparece na declaração do procedimento, isto é, dentro de "palist" no sub-

grafo de "block" (v. apêndice I). Na fig. 9.1 apresentamos exemplo de trechos de um programa onde anotamos a espécie dos parâmetros; aproveitamos para recordar a diferença entre variável local e global (v. 6.6).

Note-se que a cada parâmetro atual corresponde um parâmetro formal; essa correspondência é dada pela posição dos mesmos nas respectivas listas de parâmetros.

```
progr...
  var X: int; Y: Bool;
  proc P(q:int; var r: Bool);                    /* parâmetros formais */
    var U:int; V:Bool;
    begin
        V:=false;                          }     /* variáveis locais */
        U:=q;                              }
        X:=2;                                    /* variável global */
        :
        P(U*X+q,V);                              /* parâmetros atuais */
        :
     end;
  begin
    X:=2;                                  }     /* variáveis locais */
    Y:=true;                               }
    P(X,Y);                                      /* parâmetros atuais */
    :
  end
```

Fig. 9.1

Na fig. 9.1 temos ainda o exemplo de uma chamada recursiva do procedimento P. Um procedimento P contém uma *chamada recursiva direta* se em seu bloco aparece uma chamada a ele próprio; contém uma *chamada recursiva indireta* se existir uma seqüência de procedimentos $Q_1, Q_2, \ldots, Q_n$, $n \geq 3$, tal que $Q_i \neq Q_{i+1}$ e Q_i contém em seu bloco uma chamada a Q_{i+1}, com $P = Q_1 = Q_n$, $1 \leq i < n$.

Os parâmetros formais podem ser de quatro classes distintas: procedimentos declarados com **proc** no subgrafo de "palist"; funções, declaradas com **func**; variáveis com declaração precedida ou não de **var**. Os identificadores desses parâmetros são introduzidos na tabela de símbolos TIR com campo TIPO contendo o valor PARAM para as três primeiras classes e PARAV para a última; os campos CLASSE contêm os seguintes valores, respectivamente: PRIDEN, FUIDEN, COIDEN e VAIDEN (v. 6.2, 6.3k e 6.5f). Na fig. 9.1, q e r são de classe VAIDEN e tipo PARAV e PARAM, respectivamente.

Em PASCAL, um parâmetro atual, correspondente a um formal declarado com classe **var**, deve ser obrigatoriamente uma variável; isto é, não pode ser uma constante, uma função ou uma expressão com operadores. A correspondência entre um parâmetro atual e um formal dessa classe pode ser denominada *passagem por endereço* ("call by address", /GRI 71/). Tudo se passa como se o endereço do parâmetro atual substituísse o do formal em todas as referências a este parâmetro, no bloco do procedimento. Assim, qualquer atribuição de valor ao parâmetro formal durante a execução do procedimento produz na realidade a atribuição desse valor ao parâmetro atual, de modo que o primeiro pode ser considerado como uma variável global. Portanto, na chamada de um procedimento P, cada parâmetro atual do mesmo, correspondente a um formal com classe **var**, deve produzir a passagem do endereço do primeiro para o procedimento.

Por outro lado um parâmetro atual, correspondente a um formal sem especificação de classe, deve ser uma expressão, com ou sem operadores. A correspondência entre um parâmetro atual e um formal dessa classe pode ser denominada de *passagem por valor* ("call by

value"). Tudo se passa como se, no momento da chamada, o valor do parâmetro atual fosse atribuído ao parâmetro formal, que funciona como variável local ao procedimento. Se houver uma atribuição de um valor a um parâmetro formal dessa classe, o valor de parâmetro atual não se altera. Note-se que no manual da PASCAL /J-W 74/ menciona-se explicitamente que pode haver alteração, dentro do procedimento, do valor de um parâmetro passado por valor. Assim, o compilador deve permitir atribuições aos identificadores desses parâmetros; daí termos usado classe VAIDEN e não COIDEN.

Na fig. 9.2 damos um exemplo do efeito distinto dessas duas classes. O resultado impresso por esse programa será:

4 5 0 2

Note-se que os parâmetros atuais i e A[i] são passados por endereço, e portanto as atribuições aos parâmetros formais correspondentes feitas no procedimento alteram seus valores; no entanto, não houve alteração do índice de A, isto é, foi passado o endereço de A[i] e qualquer alteração de i dentro do procedimento não muda esse endereço; como j foi passado por valor, a atribuição dentro do procedimento não o altera.

```
progr...
   var i,j: int;
      A: array [1..10] of integer;
   proc P(var q,r:int;t:int):
      begin
            q:=4; r:=5; t:=6
         end;
   begin
        i:=1; j:=2;
        A[1]:=3; A[4]:=0;
        P(i, A[i], j);
        write(i, A[1], A[4], j)

    end
```

Fig. 9.2

Suponhamos que um procedimento P seja chamado; diremos que os identificadores de P, isto é, os parâmetros e variáveis locais de P, passam a ser *ativos*. Eles devem ser desativados ao atingir-se o **end** do bloco de P durante a execução dessa chamada. As variáveis de um programa principal estão sempre ativas durante seu processamento e o de procedimentos declarados dentro dele. Dois procedimentos P1 e P2 declarados encaixadamente, isto é, P2 dentro de P1, têm ambos seus parâmetros e variáveis locais ativados durante o processamento de P2. Por exemplo, isso acontece no rótulo 3 do programa da fig. 9.3, onde estão ativos X0, Q1, X1, Q2 e X2; ao retornar dessa ativação de P2, o processamento atinge o rótulo 2; nesse momento Q2 e X2 não estão mais ativos. Note-se que a estrutura de blocos (v. 6.6) indica quais as variáveis e parâmetros que estão ativos em qualquer ponto do programa. Assim, no comando de rótulo 2 só podem estar ativos X0, Q1 e X1.

```
progr...
   label 1,2,3;
   var X0:...
   proc P1(Q1: int);
        var X1: int;
        proc P2(Q2: int);
             var X2: int;
             begin
                  3:...
                  .
                  .
                  .
               end;
```

Fig. 9.3 (continua)

```
        begin
            X1: = X0 + 1;
            P2(X1);
            2:...
            ⋮
        end;
begin
    X0: = 1;
    P1(X0);
    1:...
    ⋮
end
```

Fig. 9.3 (conclusão)

Suponhamos agora que dois procedimentos P1 e P2 tenham o mesmo nível de encaixamento, conforme a definição de 6.6, e que P1 chame P2, como na fig. 9.4. Durante a execução de P1 estão ativos os seus parâmetros formais e suas variáveis locais (Q1 e V1). No momento em que ocorrer a chamada de P2 e se desviar o processamento para esse procedimento, as suas variáveis passarão a ficar ativas; não podemos dizer que as variáveis de P1 foram desativadas, pois ao se retornar de P2 elas passarão novamente a atuar, como no momento da chamada de P2. Para caracterizar esse caso, diremos que dentro de P2, por exemplo no rótulo 3, as variáveis de P1 estão ativas mas não estão *disponíveis* /MEL 78/. Por outro lado, no rótulo 2, os parâmetros e variáveis de P2 estão desativados e os de P1 estão não só ativados como também disponíveis.

```
progr P...
    label 1, 2, 3, 4, 5;
    var V0: int;
    proc P2 (Q2:int);
        var V2:int;
        begin
            ⋮
            3:...;
            ⋮
        end;
    proc P1(Q1:int);
        var V1:int;
        begin
            ⋮
            2:...;
            P2(V1);
            4:...;
            ⋮
        end;
    begin
        1:...;
        P1(V0);
        5:...;
    end.
```

Fig. 9.4

A tabela da fig. 9.5 mostra o estado "ativo" e/ou disponível de cada identificador durante a execução do programa objeto, nos pontos correspondentes aos rótulos do programa-fonte da fig. 9.4. Na fig. 9.6 damos uma representação em forma de "diagrama de execução" do programa-objeto, mostrando os estados de cada identificador. Um identificador está ativo dentro de todo o retângulo em que ele aparece na parte superior, inclusive nos retângulos que ele intersecciona. Assim, Q1 e V1 estão ativos nos pontos 2, 3 e 4. Para se saber qual a região de disponibilidade de um identificador, basta percorrer o lado vertical esquerdo do retângulo onde ele foi declarado; se houver algum seccionamento ou interrupção desse contorno por interseção com outro retângulo deve-se percorrer os lados superior, direito e inferior deste último, até retornar-se ao lado vertical, seccionado ou interrompido, do primeiro retângulo, continuando-se daí para a frente da mesma maneira. Assim Q1 e V1 estão disponíveis apenas nas regiões 2 e 4, pois o retângulo de P2 secciona o retângulo de P1; Q2 e V2 estão disponíveis somente na região 3; V0 está disponível em todas as regiões.

rótulo	identificadores	ativos	disponíveis
1:	V0	S	S
2:	V0,Q1,V1	S	S
3:	V0,Q2,V2	S	S
	Q1,V1	S	N
4:	V0,Q1,V1	S	S
	Q2,V2	N	N
5:	V0	S	S
	Q1,V1,Q2,V2	N	N

Fig. 9.5

Para se saber em qualquer ponto do diagrama de execução quais são os identificadores ativos naquele momento, deve-se traçar uma horizontal para a direita; estão ativos os identificadores de cada retângulo cujo lado vertical direito foi seccionado por essa horizontal. Os identificadores disponíveis são obtidos da mesma maneira traçando-se uma horizontal para a esquerda. Note-se que somente identificadores ativos podem estar eventualmente disponíveis.

Nos diagramas de execução os retângulos encaixam-se segundo os níveis de encaixamento dos blocos correspondentes, como definimos em 6.6. Assim, na fig. 9.6 vemos que o nível de encaixamento de P é zero, ocupando portanto o retângulo mais externo; o nível de P1 e P2 é 1 (um), de modo que seus retângulos encaixam-se, ambos, no retângulo de P; o lado vertical esquerdo é que estabelece o nível de encaixamento. Na fig. 9.7a temos um programa com um procedimento, R, de nível 2; ele pode chamar qualquer procedimento de nível 1, como é o caso de S. O diagrama de execução é o apresentado na fig. 9.7b. Nas regiões do rótulo 40 estão ativos os identificadores de P, Q, R e S, mas apenas os de P e S estão disponíveis, pois os retângulos de Q e R foram interrompidos.

Vejamos agora um caso com recursividade; aqui o diagrama de execução auxilia não somente a compreensão dos conceitos de ativação e disponibilidade, mas também da seqüência de chamadas e retornos. Seja, por exemplo, o programa da fig. 9.8a que calcula através do procedimento F o fatorial de um número natural N e a soma da seqüência dos naturais de 1 (um) até N. Esse procedimento é meramente ilustrativo; não apresenta a melhor solução para esses problemas. Em cada ativação de F estão disponíveis apenas os identificadores locais (no caso, exclusivamente o parâmetro formal) e os do programa principal P; permanecem ativos, mas não disponíveis, os identificadores das chamadas anteriores de F. Note-se que as

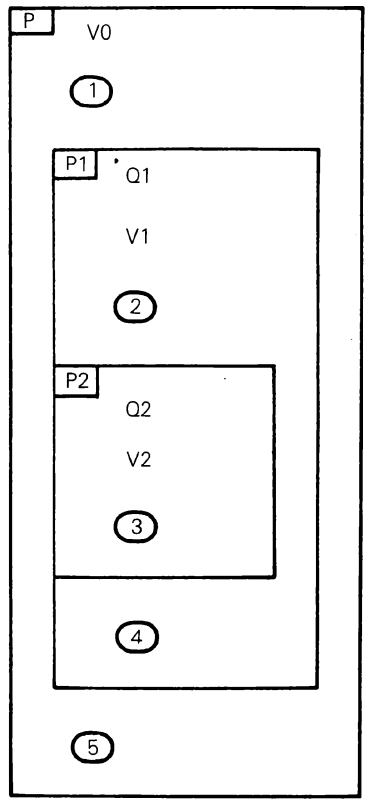
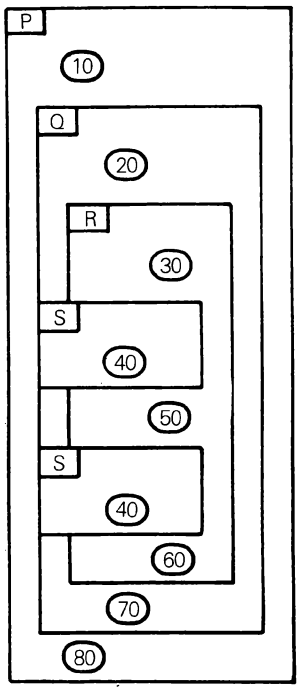


Fig. 9.6

```
progr P...;
    label 10,20,30,40,50,60,70,80;
    :
    proc S;
        begin
            :
            40:...;
            :
        end;
    proc Q;
        proc R;
            begin
                :
                30:...;
                S;
                50:...;
                S;
                60:...;
                :
            end;
        begin
            :
            20:...;
            R;
            70:...;
            :
        end;
    begin
        :
        10:...;
        Q;
        80:...;
        :
    end
```



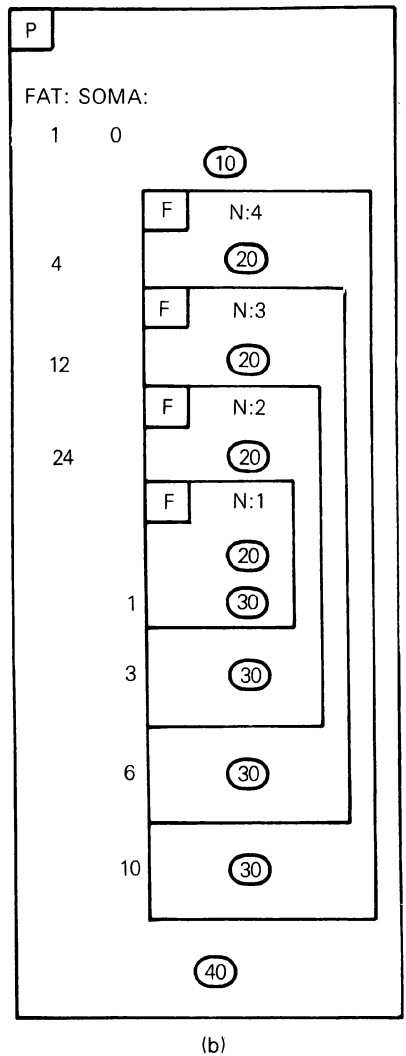
(a) (b)

Fig. 9.7

multiplicações (cálculo do fatorial) são feitas na parte superior dos retângulos de F, e a soma na parte inferior, e que cada retângulo de F contém o parâmetro N. Na figura 9.8b apresentamos o diagrama de execução com os valores das variáveis FAT e SOMA ao atingir-se os comandos especificados pelos rótulos indicados — mas antes desses comandos serem processados. Valores deixados em branco indicam que não houve alteração desde o valor imediatamente superior.

```
proc P...
   label 10,20,30,40;
   var  FAT, SOMA: int;
   proc F(N: int);
      begin
         FAT:= FAT*N;
         20:if N>1 then F(N-1);
         SOMA:= SOMA+N;
         30:;
      end;
   begin
         FAT:= 1;
         SOMA:= 0;
         10:F(4);
         40:write (FAT,SOMA);
   end
```

(a)



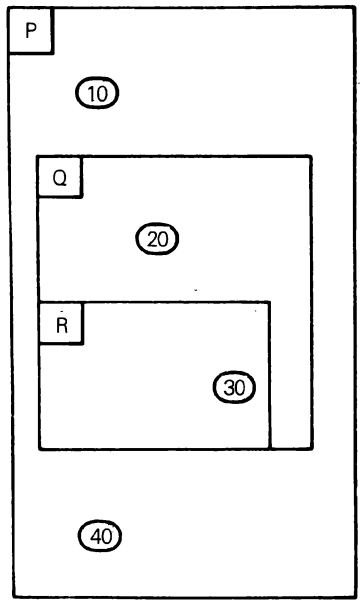
(b)

Fig. 9.8

Em todos os exemplos anteriores, encerrou-se o processamento de um procedimento atingindo-se o fim do seu bloco, isto é, seu **end.** Na fig. 9.9a apresentamos um caso em que esse processamento é encerrado pela execução de um desvio para fora do procedimento. Note-se o efeito correspondente no diagrama de execução da fig. 9.9b. Desvios são sempre feitos para um procedimento cujas variáveis locais estão disponíveis, ou para o programa principal (cujas variáveis estão sempre disponíveis).

```
progr P...;
  label 10,20,30,40;
  var...;
  proc Q;
      proc R;
          begin
              :
              30:...;
              goto 40;
              :
          end;
      begin
          :
          20:...;
          R;
          :
      end;
  begin
      :
      10:...;
      Q;
      40:...;
      :
  end
```

(a)



(b)

Fig. 9.9

9.3 SISTEMA DE EXECUÇÃO

Seguindo a nomenclatura introduzida no vernáculo em /KOW 83/, chamaremos de "sistema de execução" a estrutura de dados que permite a execução dos POs em relação à estrutura de blocos. O original usado em inglês é "run-time system"

O sistema de execução deve implementar os conceitos de identificadores ativados e disponíveis que definimos no item anterior. Observando-se atentamente os diagramas de execução, podem-se sugerir as seguintes características de implementação (a fig. 9.7 é a mais representativa, e pode ser examinada como ilustração):

a) A parte superior de cada retângulo pode ser considerada como a área de memória associada ao bloco correspondente, durante a execução do PO, onde são colocadas suas variáveis.

b) A manipulação dessas áreas de memória deve seguir uma organização de dados em forma de pilha. De fato, os próprios diagramas de execução podem ser encarados como representando os vários estados dessa "pilha de execução" (invertida, isto é, com o fundo na parte superior). Como veremos em seguida, cada área da pilha é denominada de *registro de ativação* ("activation record").

c) A abertura do retângulo mais externo, em sua parte superior, corresponde à ativação do programa principal; a abertura de um outro retângulo qualquer, em sua parte superior, corresponde à chamada — ou ativação — de um procedimento. Quando um retângulo for

aberto, deve-se empilhar na pilha de execução o registro de ativação contendo seus dados; isto é, parâmetros formais e variáveis locais. Assim, os registros de ativação aparecem na pilha na ordem de chamada dos respectivos procedimentos.

d) O fechamento de um retângulo (lado inferior) corresponde a abandonar-se um bloco, o que pode ocorrer se se atingir seu **end** ou desviar-se para fora dele por um **goto**; nesse momento, deve-se desempilhar o registro de ativação onde estão seus dados. Se se tratou de um desvio para um ponto de um outro retângulo, as variáveis destes devem estar disponíveis, e é necessário eventualmente desempilhar vários registros de ativação; alguns destes podem não estar disponíveis, como é o caso, por exemplo, de se desviar para fora de um procedimento que teve chamadas recursivas.

e) O estado da pilha em qualquer ponto da execução do PO pode ser obtido tomando-se o ponto correspondente no diagrama de execução e traçando-se uma horizontal por esse ponto; a pilha contém os registros de ativação dos lados direitos de retângulos seccionados por essa horizontal, e que constituem as áreas de identificadores ativos.

f) Todos os registros de ativação da pilha correspondem a identificadores ativos; é necessário indicar em quais desses registros estão os identificadores disponíveis (diremos "registros ativos" e "registros disponíveis"). Essa indicação deve mudar quando da ativação de cada bloco e do término de sua execução. O topo e o fundo da pilha de execução correspondem sempre a registros de ativação disponíveis.

g) Os registros de ativação disponíveis correspondem a uma seqüência densa (isto é, sem faltar nenhum elemento) de níveis de encaixamento dos blocos correspondentes (v. 6.6). Assim, se no topo da pilha tivermos um registro disponível de um bloco com nível de encaixamento n, existem obrigatoriamente na pilha registros disponíveis de níveis n-1,...,1,0, com $n \geq 0$ e exatamente um para cada nível.

h) Se o registro de ativação disponível do topo da pilha tem nível de encaixamento n, podem haver na pilha registros com nível de encaixamento igual ou superior a n, mas estes estarão obrigatoriamente não-disponíveis.

Essas características sugerem que se faça a implementação da pilha de execução e dos endereços das variáveis da seguinte maneira:

i) Para cada registro de ativação r introduzimos na pilha uma entrada denominada de *base*, a qual encabeça r contendo três campos: 1) o mais à esquerda, que denominaremos de *apontador estático*, contém um apontador para a pilha, indicando qual a base do registro de ativação r_d disponível imediatamente anterior; isto é, considerando-se r no topo da pilha, com nível de encaixamento n, r_d tem nível n-1. 2) o campo central contém o nível de encaixamento de r. 3) o mais à direita, que denominaremos de *apontador dinâmico*, contém um apontador para o registro de ativação imediatamente anterior a r na pilha.

Os apontadores estáticos formam a *cadeia estática* ("static chain") de bases e dos respectivos registros; se o registro de ativação r está no topo da pilha, a cadeia estática que começa na base de r passa por todos os registros disponíveis. Os apontadores dinâmicos formam a *cadeia dinâmica* ("dinamic chain") de bases dos respectivos registros; a cadeia dinâmica que começa em r passa por todos os registros de ativação da pilha, na seqüência inversa das chamadas dos respectivos procedimentos. Essas denominações provêm do fato de que em tempo de compilação pode-se determinar qual registro de ativação será apontado por um apontador estático de uma base; por outro lado, o compilador não pode determinar a ordem de chamada dos procedimentos, e portanto as informações dos apontadores dinâmicos.

Na fig. 9.10 apresentamos a situação da pilha de execução PE do programa da fig. 9.7 ao atingir-se a primeira vez o rótulo 40. Os registros de ativação são indicados pelos nomes dos respectivos procedimentos. Note-se a presença de uma base para o programa principal, com apontadores contendo **nil**. Na fig. 9.10 encontram-se ainda um apontador TOPPE para o topo da PE, bem como duas estruturas D e MND, sobre as quais falaremos adiante.

Na fig. 9.11 mostramos a mesma pilha ao atingir-se o rótulo 50. Nessa situação as cadeias estática e dinâmica contêm os mesmos apontadores, pois as chamadas dos procedi-

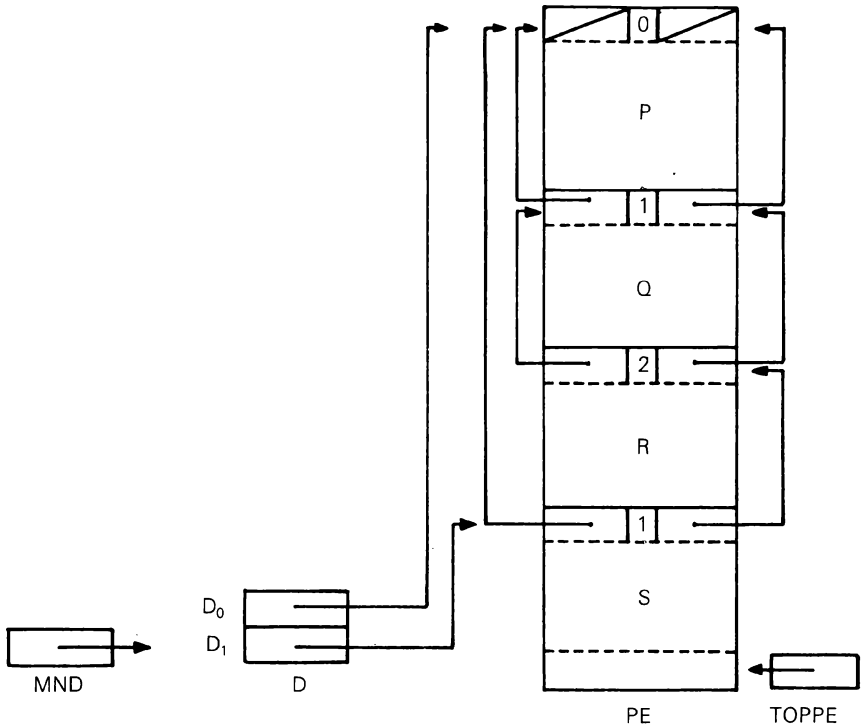


Fig. 9.10

mentos se deram sempre para níveis de encaixamento superiores (isto é, com uma unidade a mais do que os níveis dos procedimentos ou programa principal) onde ocorreram as chamadas.

j) Em um ponto qualquer do programa-objeto estão disponíveis as variáveis dos registros de ativação pertencentes à cadeia estática que começa no topo da pilha de execução, isto é, a "cadeia estática disponível". Para facilitar o acesso a esses registros disponíveis, vamos introduzir uma outra estrutura, com apontadores para cada base da cadeia estática disponível; essa estrutura também funciona parcialmente como uma pilha, a "pilha de apontadores para os registros disponíveis" (em inglês "display"), indicada nas figs. 9.10 e 9.11 por D. Ao lado de cada elemento de D colocamos seu identificador D_i; i indica o nível de encaixamento do registro de ativação apontado, isto é, o nível de encaixamento do respectivo procedimento (ou do programa principal, no caso de D_0). Para se saber qual o registro do topo da PE, isto é, qual o maior valor de i, adicionamos ainda um apontador para o topo da D indicando qual o "maior nível disponível" MND.

Na fig. 9.12 mostramos como se pode declarar PE e D como matrizes unidimensionais, para simplificar a sua representação. EST e DIN são os índices para PE das cadeias estática e dinâmica; NIV contém o nível de encaixamento; PAL é a palavra de memória onde serão armazenadas as variáveis do programa principal e dos procedimentos.

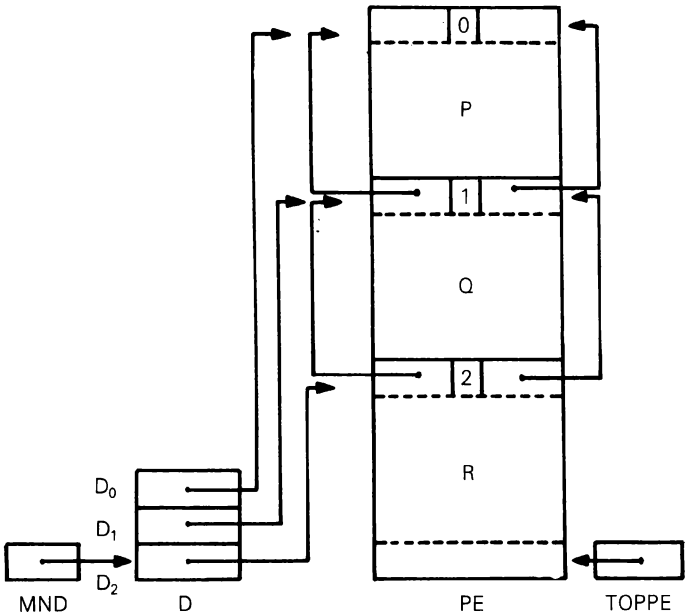


Fig. 9.11

```
var PE: array [1..MAXPE] of
        packed record case BASE: Boolean of
                                  true: (EST: 1..MAXPE;
                                         NIV: 0..MAXNIV;
                                         DIN: 1..MAXPE);
                                  false: (PAL: integer);
                end;
    TOPPE: 1..MAXPE;
    D: array [0..MAXNIV] of 1..MAXPE;
    MND: 0..MAXNIV;
```

Fig. 9.12

Com a declaração **packed** quisemos indicar que o conteúdo do registro deve ser compactado; vamos supor que se consiga compactar os campos EST, NIV e DIN em uma só palavra.

k) As variáveis do programa-objeto serão então endereçadas não mais por meio de endereços absolutos (que seriam no caso representados por um índice da PE), mas sim por meio de endereços relativos às bases dos seus registros de ativação e por uma menção de qual base da cadeia estática deve ser usada. Em outras palavras, o endereço de cada variável V passa a ser um par (n,d), onde n é um índice para D, cujo valor é o nível de encaixamento do procedimento (ou programa principal) onde V foi declarada, e d o início da entrada de V na PE relativa ao valor de D[n] denominado de "deslocamento". Assim, V encontra-se na entrada PE[D[n] + d]. As bases têm sempre um endereço (n,0). Lembremos que, dessa maneira, só se

tem acesso aos registros de ativação disponíveis, pois os apontadores D apontam apenas para essa classe de registros.

Vejamos em seguida como manipular PE, D e MND nos casos de chamada, encerramento (pelo **end**) e abandono (**goto**) de um procedimento.

l) Chamada de procedimento. Suponhamos que se chame um procedimento de nível de encaixamento n (deduzido pelo compilador, conforme 6.6). O PO deve empilhar na PE os parâmetros, adicionar a TOPPE o número de palavras necessárias a fim de reservar a área para as variáveis locais do procedimento e outras informações e finalmente executar as instruções cuja descrição em PASCAL damos na fig. 9.13. Note-se que essas instruções servem tanto para casos de se estar no momento da chamada em um nível de encaixamento n-1, quanto em um nível superior a n, caso em que D deve ser desempilhada de uma ou mais células; essa situação é ilustrada pela passagem da configuração da fig. 9.11 para a da fig. 9.10.

```
TOPPE: = TOPPE + 1;
with PE[TOPPE] do
  begin  BASE: = true;
         NIV: = n;
         DIN: = D[MND];          /* registro anterior da pilha */
         EST: = D[n-1];          /* registro disponível anterior */
         MND: = n;
         D[MND]: = TOPPE
  end
```

Fig. 9.13

m) Saída de procedimento por **end**. Neste caso, basta desempilhar um registro de ativação da PE, usando para isso a cadeia dinâmica, e atualizar a pilha D percorrendo a cadeia estática que começa no novo topo. A passagem da fig. 9.10 para a 9.11 ilustra um caso destes. O código a ser executado pelo PO está na fig. 9.14; AUX é uma variável de tipo 1..MAXPE que é usada para conter o índice das entradas das bases que vão sendo encontradas no percurso da cadeia estática. Note-se que, por simplicidade, refazemos toda a pilha D, a menos de D_0; na verdade, pode-se parar esse processo no maior nível i tal que $i \leq j$ onde j é o nível do procedimento que acabou de ser encerrado, e tal que o conteúdo de D[i] seja igual ao apontador estático para o registro de ativação de nível i que continuará a estar disponível. Isto é devido ao fato de todos os registros de ativação da cadeia estática (que está sendo percorrida), de níveis igual e inferior a i, continuarem disponíveis.

```
TOPPE: = D[MND]-1;          /* novo índice do topo da PE: = índice da base atual -1 */
AUX: = PE[D[MND]].DIN;      /* índice da base anterior e que se tornará a base do topo */
MND: = PE[AUX].NIV;         /* novo índice do topo da pilha D */
while PE[AUX].NIV ≠ 0 do    /* percorre a cadeia estática refazendo D */
  begin
    D[PE[AUX].NIV]: = AUX;  /* refaz célula de D */
    AUX: = PE[AUX].EST      /* próxima base */
  end
```

Fig. 9.14

n) Saída de procedimento por **goto**. Neste caso, é eventualmente necessário desempilhar da PE vários registros de ativação. Como o desvio se faz sempre para um bloco cujo registro de ativação está disponível, cf. (d) acima, é suficiente conhecer o nível do procedimento (ou programa principal) para o qual se está desviando, e refazer os topos das pilhas PE e D. Um exemplo deste caso seria nas figs. 9.7 e 9.11 um desvio dentro do procedimento R para um rótulo no procedimento Q. Na fig. 9.15 apresentamos as instruções que o PO deve executar para efetuar o rearranjo das pilhas quando ocorre um tal desvio. Supomos que o desvio se dê para um procedimento (ou programa principal) de nível n, $MND > n \geq 0$, onde o conteúdo

de MND indica o valor do nível de encaixamento do procedimento em que ocorreu o **goto**. Note-se que tanto n como o valor de MND podem ser conhecidos pelo compilador; o primeiro, por haver declarações de rótulos (**label**...) e o segundo através do nível de encaixamento do bloco sendo compilado, indicado por NIVAT no item 6.6. É trivial refazer o topo de D passando-se a conhecer a base do novo registro de ativação que constituirá o registro do topo da PE. O problema é determinar o topo desta pilha. Para isso são geradas instruções que percorrem a cadeia dinâmica até achar-se o registro de ativação que foi empilhado sobre o registro procurado; a base do primeiro permite estabelecer o topo do segundo.

```
MND: = n;                              /* novo índice do topo da D */
TOPPE: = D[n + 1];
while PE[TOPPE].DIN ≠ D[MND] do        /* percorre a cadeia dinâmica */
      TOPPE: = PE[TOPPE].DIN;
TOPPE: = TOPPE-1;
```

Fig. 9.15

As instruções apresentadas nas figs. 9.13 a 9.15 podem ser geradas no PO em cada chamada, **end** ou **goto** para fora de um procedimento. Isso acarretaria uma duplicação desses trechos de instruções. Isso pode ser evitado fazendo-se com que essas instruções formem *sub-rotinas* do PO, que podem ser chamadas pela instrução BST do HIPO (v. apêndice III); essa instrução só transmite como parâmetro o endereço de volta, de modo que n e TOPPE devem ser variáveis globais a essas sub-rotinas. Para um processamento realmente eficiente dessas rotinas, elas deveriam ser implementadas no "hardware" do computador-objeto, como é o caso do Burroughs B-6900. Nesse caso uma só instrução pode ser suficiente para executar todas as ações de cada seqüência.

9.4 GERAÇÃO DO CÓDIGO-OBJETO

a) Implementação da pilha de execução

Para uma implementação da pilha de execução no computador hipotético HIPO (v. apêndice III), poderíamos supor que este tivesse instruções de manipulação de pilhas, o que reduziria o código gerado. Não o faremos, no entanto, para podermos mostrar em detalhe as ações que devem ser seguidas. Evidentemente, muitas seqüências de instruções poderiam ser substituídas por uma só, implementando no "hardware" do computador-objeto instruções especiais que executam as seqüências aqui apresentadas.

Implementaremos a pilha PE como se fosse uma matriz unidimensional; o índice do topo será sempre guardado no indexador número 9 (X9), que fará o papel do apontador TOPPE de 9.3; tomemos a base de PE com endereço B1 no PO. Assim, por exemplo, para empilhar-se o conteúdo do acumulador na PE pode-se executar a seqüência

```
MDX  9  1          /* incrementa TOPPE */
STA  9  B1
```

A pilha de apontadores para os registros disponíveis D (v. 9.3j) será implementada no HIPO nos indexadores 2 a 8 (X2 a X8), permitindo-se assim 7 níveis de encaixamento no programa-fonte. (O indexador 1 fica reservado para cálculo de endereços de variáveis indexadas, cf. 7.9.) As variáveis do programa principal (nível de encaixamento zero) serão endereçadas por meio de endereços absolutos, em relação a B1. Por exemplo, a 7ª variável do programa principal terá endereço B1 + 6 no código-objeto. Já o acesso a uma variável de um procedimento de nível de encaixamento 4 será feito por meio de uma instrução de PO como

```
op   5  E
```

onde E é o seu deslocamento em relação à base (no caso, apontada pelo X5) do procedimento.

b) Níveis de encaixamento

Como vimos em 9.3k, os parâmetros e variáveis locais aos procedimentos são endereçados por meio de um par (n,d), onde n é o nível de encaixamento do procedimento e d o deslocamento da variável em relação à base do procedimento na pilha de execução PE. Em nosso caso, como vimos, será usado o indexador do HIPO de número n + 1 para os procedimentos, e um endereço absoluto para as variáveis do programa principal. Vemos, portanto, que o conhecimento do nível de encaixamento de cada variável é fundamental para a compilação de procedimentos. Como supusemos uma busca linear na tabela de símbolos TIR do compilador (v. 6.6), esse nível pode ser deduzido conhecendo-se o nível n do procedimento sendo compilado e decrementando-se n de 1 todas as vezes que, na busca de um identificador ou rótulo na TIR, passa-se por uma entrada de classe PRIDEN ou FUIDEN e tipo TYIDT (v. 6.3b). Com isso a busca linear fica prejudicada em sua eficiência e uma busca por método de espalhamento ("Hashing") não daria mais a informação do nível. A maneira mais simples de contornar esses problemas é a de implementar um campo adicional nos elementos da TIR, contendo o nível de encaixamento de cada identificador ou rótulo. As rotinas RS_{40} e RS_{20} dos ramos **proc** e **func** no subgrafo de "block" devem incrementar um "contador de nível de encaixamento" e a RS_{141}, executada ao fim do reconhecimento do procedimento (nó "block") deve decrementá-lo; o conteúdo desse contador é então copiado no campo indicador do nível de encaixamento de cada novo símbolo inserido na TIR.

Cada variável local a um procedimento declarada na seção **var** de seu bloco deve ser introduzida na TIR com o seu deslocamento d em relação à base do procedimento, colocado no campo ENDOBJ. Para isso, implementa-se uma variável DES cujo valor, ao iniciar-se a inclusão de uma nova variável na TIR (tanto no programa principal como nos procedimentos) deve ser o do deslocamento dessa variável, isto é, o espaço total ocupado por todas as variáveis do procedimento até esse momento. Depois de ser incluída na TIR a última variável local de um procedimento, o valor de DES indicará o espaço total necessário para armazenarem-se todas as variáveis locais na PE, bem como parâmetros, a base do procedimento e outras entradas necessárias, como veremos adiante; veremos também que DES deverá ser empilhada para uso posterior.

c) Temporários locais

Devido ao fato de que em expressões podem ocorrer chamadas de funções e devido a comandos como "for", é preciso que os temporários empregados em procedimentos e funções sejam locais, isto é, armazenados na PE. O próprio contador de temporários TMPATV (v. 7.3) pode ser empregado, mas deve conter agora o deslocamento do temporário na PE em relação à base do procedimento; ele deve ser inicializado com o valor de DES pela RS_{142} do nó **begin** em "block". Cada vez que se necessita de um novo temporário dentro do procedimento, TMPATV deve ser incrementado, e seu valor comparado com DES, a qual é atualizada se seu valor for menor. Assim DES dá o espaço necessário não só para as variáveis locais, e outras informações da PE, mas também para os temporários. Além de atualizar DES, a cada novo temporário reservado na PE deve-se gerar a instrução

```
MDX  9  1
```

que incrementa o índice TOPPE; isso se dá pelo fato de não se poder saber de antemão quantos temporários serão necessários para cada procedimento.

Note-se que o endereço de variáveis indexadas deve ser colocado em temporários locais, e não em temporários globais como vimos, para o programa principal, em 7.9.

d) Chamada de procedimento e retorno

Suponhamos que o código gerado para um procedimento P comece no endereço E_p. Uma seqüência de instruções geradas para implementar uma chamada de P envolve duas partes: tratamento dos parâmetros atuais e desvio para o procedimento. Na realidade, é necessário também armazenar o estado da computação, como conteúdo de registradores, indicado-

res etc.; não abordaremos esses casos aqui. O desvio para o procedimento seria feito por uma instrução

C BRN E_p

onde C foi colocado apenas como referência para o texto abaixo.

Essa instrução seria gerada pela RS_{79}, executada ao fim da seqüência de chamada de P, isto é, em ')' do ramo PRIDEN no subgrafo de "statm", em ')' do ramo FUIDEN de "factor" no caso de P ser uma função, e nos λ-nós desses dois ramos.

Ao atingir-se o **end** de P, é necessário produzir um desvio para a posição C + 1, que é o ponto para onde se deve retornar. Para isso, o PO deve armazenar o valor de C antes de executar a BRN acima. Obviamente, C deve ser guardado em uma pilha, pois P pode conter chamadas de procedimentos com seus próprios desvios. Vamos empilhar C na pilha de execução PE introduzida no item anterior, na célula imediatamente seguinte à base do procedimento. Conhecendo o nível de encaixamento n do procedimento sendo chamado, armazenado na TIR na entrada do seu identificador como vimos em (b), o compilador gera as instruções da fig. 9.13 (ou gera uma chamada para uma sub-rotina do PO com essas instruções) e a seguir gera a seguinte seqüência em lugar da instrução dada acima

```
MDX  9 1        /* incrementa TOPPE */
LAD    *+3      /* carga do endereço de retorno */
STA  9 1        /* empilha o endereço de retorno */
BRN    Ep
```

Lembremos que X9 contém um apontador para o topo da PE. No fim do código do procedimento P, deve ser gerado o desvio para o ponto de retorno. Para isso, a RS_{141}, do nó "block" no ramo **proc** do subgrafo de "block", além de manipular a tabela de símbolos TIR do compilador (v. 6.6) deve gerar as instruções da fig. 9.14 (ou uma chamada para uma sub-rotina do PO que executa essas instruções) e em seguida deve gerar

BRN 9I 2

que é um desvio indireto sobre o conteúdo do índice da base de P na PE acrescido de uma unidade. Essa é exatamente a posição onde foi empilhado o endereço de retorno, já que ao terminar a execução das instruções da fig. 9.14 o topo da PE coincide com o topo do novo registro de ativação, isto é, uma célula anterior à base de P (v. fig. 9.16).

No caso de desvio para fora do procedimento (**goto** no programa-fonte) a RS_{76}, executada após o reconhecimento do rótulo do **goto** no subgrafo de "statm", deve inicialmente reconhecer que se trata realmente de um desses desvios. Isso é feito comparando-se o nível de encaixamento do rótulo para onde está sendo feito o desvio, com o nível do procedimento atual; em caso positivo a RS_{76} gera as instruções da fig. 9.15 (ou chamada de sub-rotina do PO que executa essas instruções) e em seguida gera um desvio BRN para o rótulo.

e) Parâmetros

Em 9.2, vimos que há quatro classes de parâmetros. Em todos os casos, serão empilhadas informações a respeito dos mesmos na PE (v. fig. 9.16 no fim deste capítulo); essas informações serão compactadas de modo que se tenha apenas uma palavra do HIPO para cada parâmetro. Durante a seqüência de chamada, as informações sobre os parâmetros atuais vão sendo empilhadas na PE, como se fossem entradas adicionais do registro de ativação atual, isto é, do procedimento (ou programa principal) onde está ocorrendo a chamada. Esse empilhamento é feito pela RS_{78} no ramo PRIDEN do subgrafo de "statm" e no ramo FUIDEN de "factor", gerando inicialmente uma instrução para incrementar o valor de TOPPE. Isto é necessário pois podem ocorrer funções dentro dos parâmetros atuais. A RS_{79}, executada após o reconhecimento de todos os parâmetros, deve gerar uma instrução que subtrai de TOPPE o número de células da PE utilizadas para armazená-los. Com isso, corrige-se o valor de TOPPE antes do ajuste das cadeias estática e dinâmica e do desvio para o procedimento (v. 9.3l e 9.4d).

A RS_{78} deve também verificar se o tipo e classe de cada parâmetro atual corresponde ao tipo e classe dos parâmetros formais que, como vimos em 6.3k, estão armazenados na tabela

de símbolos TIR nas posições seguintes à do identificador do procedimento ou função. Para isso, as rotinas RS_{77} dos nós de PRIDEN no subgrafo de "statm" e FUIDEN em "factor" inicializam uma variável IP com o índice na TIR do primeiro parâmetro. A RS_{78}, além das ações já descritas, incrementa IP para cada novo parâmetro atual reconhecido, podendo assim comparar seu tipo, que é obtido a partir das informações do topo da pilha semântica (índice da TIR, temporário, índice para TCA ou TCN), com o do parâmetro formal. Devido ao fato de parâmetros atuais poderem ser expressões envolvendo chamadas de funções, o valor de IP deve ser empilhado; podemos usar para isso a PSE, colocando esse valor em paralelo a '(' depois de PRIDEN e FUIDEN; neste último caso, a localização da célula paralela a '(' é feita a partir do topo da pilha do analisador sintático, K[TOPO].R (v. 5.7) que aponta para FUIDEN; no primeiro caso, pode haver ou não um rótulo antes do PRIDEN, o que pode ser reconhecido pela presença de um indicador de ':' em K[TOPO].R + 1.

Vejamos quais informações são armazenadas na PE para cada classe de parâmetro formal e como essas informações são tratadas no bloco do procedimento.

e_1) Classe VAIDEN e tipo PARAM

Como vimos em 9.2, trata-se aqui de parâmetros formais declarados com **var** na lista de parâmetros, e a correspondência com o parâmetro atual é feita pela "passagem por endereço". De fato, na PE empilhamos o *endereço* do parâmetro atual, que deve ser obrigatoriamente uma variável. Esse endereço deve ser um par (n,d) indicando o nível de encaixamento e o deslocamento da variável, como vimos em 9.3k. No caso de variável indexada, o par indica um temporário local.

Dentro do bloco do procedimento, um comando que envolve um parâmetro atual dessa classe deve sempre gerar instruções que usam o indexador correspondente ao nível de encaixamento do procedimento sendo compilado, indicador de endereçamento indireto e na parte de endereço o deslocamento do parâmetro em relação à base. O endereçamento indireto garante o cálculo correto do endereço; pode haver uma dupla indexação se o parâmetro atual for parâmetro formal de outro procedimento.

e_2) Classe VAIDEN e tipo PARAV

Essa classe é usada na "passagem por valor". Nesse caso na seqüência de chamada pode-se colocar o conteúdo do parâmetro atual na PE, e não seu endereço. Na geração de código usando o parâmetro formal correspondente são produzidas instruções sem endereçamento indireto, de modo que o próprio conteúdo da palavra da PE onde está o parâmetro é usado ou modificado.

Note-se que, ao se compilar uma chamada, sabe-se exatamente qual o parâmetro formal da TIR correspondente a cada parâmetro atual. Assim é possível gerar-se a seqüência adequada a cada classe de parâmetros.

e_3) Classe PRIDEN

Suponhamos que um parâmetro formal de um procedimento P tenha sido declarado como **proc** f. Neste caso, nas seqüências de chamada de P é necessário empilhar na PE o endereço do procedimento do parâmetro atual, compactado ainda com o seu nível de encaixamento. Nas ocorrências de f no bloco de P são empilhados na PE os parâmetros, todos por valor conforme a especificação no manual da PASCAL /J-W 74/. A seguir é extraído o nível de encaixamento da palavra do parâmetro f na PE, e são executadas as instruções descritas na fig. 9.13. Finalmente, é produzido um desvio indireto

```
BRN  nI  m
```

onde n-1 é o nível de encaixamento do bloco do procedimento P e m o deslocamento de f na PE em relação à base.

Note-se que a RS_{78} reconhece que o parâmetro é de classe PRIDEN consultando a entrada da TIR indicada no topo da PSE.

e_4) Classe FUIDEN

Aqui tudo se passa como na classe anterior, a menos das peculiaridades de chamadas de função que examinaremos em seguida.

f) Funções

As chamadas de funções dão-se em expressões, no subgrafo de "factor". Assim, podemos compilar as funções de tal modo que seu resultado seja sempre armazenado em um dos temporários do procedimento chamador. Estes, como vimos em (c), têm endereços relativos no caso de procedimentos e absolutos no caso do programa principal. No primeiro caso, o endereço conterá um indexador. Apenas um temporário basta para conter o resultado, já que este é sempre dos tipos básicos ("integer" ou "Boolean", em nosso caso), intervalo de seqüência de constantes simbólicas ou apontador ("pointer"), conforme a definição da PASCAL.

Uma palavra na PE deve ser reservada para conter o endereço do temporário; tomemos como norma que ela será a terceira palavra do registro de ativação da função, logo após a base e o endereço de retorno (v. fig. 9.16). Assim, no bloco desta, quando houver uma atribuição ao seu identificador, a RS_{80} no comando de atribuição do subgrafo de "statm" gera a seqüência

```
LDA      E
STA  nl  2
```

onde E é o endereço de "expr" do lado direito da atribuição, e n-1 o nível de encaixamento da função que se está compilando. Note-se que a RS_{80} reconhece que se trata de um desses casos pois em PSE[TOPPS-2] haverá um par STIR,i e TIR[i].CLASSE = FUIDEN.

Na seqüência de chamada, é necessário inicialmente carregar na PE o endereço do próximo temporário disponível, o que é feito pela RS_{77} no nó FUIDEN de "factor", que incrementa o conteúdo de TMPATV (v. 7.3) e gera as seguintes instruções no caso da chamada ocorrer no programa principal:

```
LDA      ARTEMP+m
STA  9   3
```

pois o indexador 9 contém o topo da PE, e onde m é o conteúdo de TMPATV. Se a chamada ocorrer em um procedimento, deve-se utilizar o indexador correspondente ao seu nível.

g) Reserva de área para as variáveis locais

Como vimos em 9.3*ℓ*, depois de se gerar as instruções da fig. 9.13 — ou uma chamada para uma sub-rotina do PO que as executa — devem ser empilhados na PE as informações sobre os parâmetros atuais, e em seguida deve ser ajustado o índice do topo da PE reservando-se com isso uma área para as variáveis locais. Em lugar de colocar esse ajuste na seqüência de chamada, vamos colocá-lo como primeira instrução a ser executada pelo procedimento; assim ele será incorporado uma só vez ao PO. Ele será da forma

```
MDX  9   n
```

onde n = q + k, onde q é a área total da PE necessária para armazenar os parâmetros e as variáveis locais, e k contém o valor 2 no caso de procedimentos (para incluir a base e o endereço de retorno) ou 3 no caso de funções (para incluir ainda o endereço do temporário do resultado). O valor de n é o conteúdo da variável DES, introduzida em (b) acima, quando for terminada a inclusão da última variável local na TIR.

A instrução acima será gerada pela RS_{142}, do nó **begin** de "block", pois nesse ponto tem-se certeza de que terminaram as declarações, e será iniciada a geração de código para os comandos do procedimento sendo declarado. Como pode ter ocorrido a declaração de outros procedimentos no bloco do procedimento sendo compilado, é necessário empilhar o valor de DES. Usaremos para isso a primeira célula de "block", isto é, a apontada por K[TOPO].R (v. 5.7). O conteúdo dessa célula, da forma SOBJ,n deve ser atualizado por todas as rotinas semânticas que incluem na TIR parâmetros formais (RS_{45} e RS_5 de "palist") e variáveis locais RS_{60} e RS_{65}) no ramo **var** de "block"; no primeiro caso, é necessário usar o índice da PSE dado por K[TOPO-1].R, já que se entrou em "palist" vindo de "block". Assim, a RS_{142} deve retirar o valor de n de K[TOPO].R como indicado.

Na fig. 9.16 apresentamos um esquema da PE para as informações de uma função, durante sua execução. Um procedimento não teria a entrada "endereço do temporário do resultado".

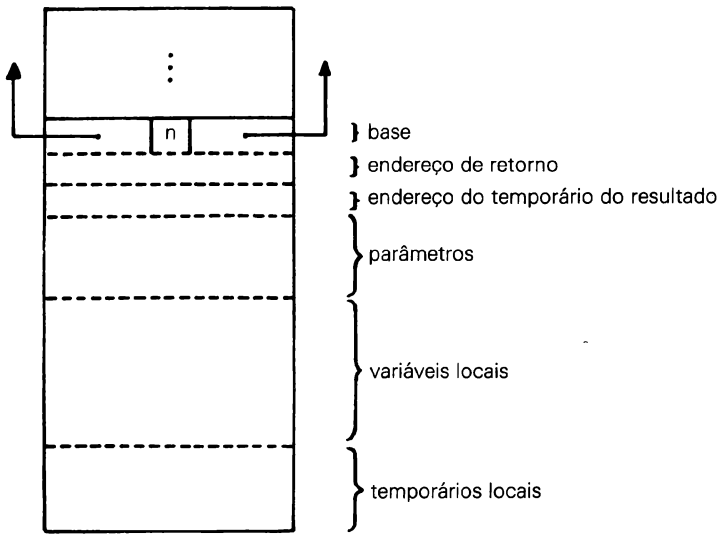


Fig. 9.16

h) Procedimentos "forward"

Na linguagem PASCAL como em muitas linguagens de programação não é possível utilizar-se um identificador sem que se o tenha declarado. Esse fato tem as conseqüências salutares de se melhorar a documentação do programa e de se diminuir a quantidade de erros de programação. Essa característica aplica-se também a chamada de procedimentos, isto é, não se pode chamar um procedimento P sem que este tenha sido previamente declarado no bloco em que está sendo feita a chamada ou em um bloco mais externo que engloba o primeiro. Se não há nenhuma chamada recursiva indireta, pode-se sempre ordenar os procedimentos de tal modo que qualquer chamada a um deles apareça no programa-fonte em um ponto posterior à sua declaração. Mas este não é o caso com recursões indiretas. Por exemplo, se P1 chama P2, que por sua vez chama P1, é impossível colocar todas as declarações antes das chamadas. Para isso foi introduzido na PASCAL /J-W 74/ um meio de se declarar somente o cabeçalho do procedimento ("Heading"), sem o seu "block": trata-se de substituir este último pela palavra "forward". Por exemplo, podemos ter as declarações

```
⋮
proc P (x:T1); forward;
proc Q (y:T2); begin P(a) end;
proc P; begin Q(b) end;
⋮
```

Note-se que na segunda ocorrência de P, onde é definido seu "block", os parâmetros devem ser omitidos.

Para compilar-se um procedimento P com essa característica adicional, é necessário modificar-se o grafo sintático, colocando um ramo alternativo a **begin** no subgrafo de "block", com a palavra reservada **forward** como único nó. Não colocamo-la na lista de palavras reservadas pois ela não é tratada como tal no manual da PASCAL. Uma rotina "semântica" executada ao se reconhecer esse nó deve colocar uma marca na entrada da tabela de símbolos TIR onde se encontra 'P', por exemplo por meio do valor '-1' no campo INDTT. Os parâmetros de P também devem ser marcados, pois estes não devem ser usados na busca de identificadores de parâmetros ou variáveis locais de outros procedimentos; eles não podem ter seu campo IDROT apagado, como descrevemos em 6.6g, pois serão necessários quando ocorrer a segunda declaração de P, onde os parâmetros devem ser omitidos. As rotinas que inserem novos identificadores na TIR, descritas em 6.2, devem também ignorar a presença na TIR de identificadores com essas marcas; já as RS_{40} e RS_{20}, dos ramos **proc** e **func** do subgrafo de "block", devem verificar se o identificador que acabou de ser inserido não ocorre em uma entrada anterior no mesmo nível de encaixamento, isto é, na mesma seção do topo da TIR. Se esse for o caso, a última entrada é eliminada, continuando-se normalmente a compilação. A ausência obrigatória de parâmetros pode ser testada por uma rotina "semântica" no nó ';' dos ramos citados de "block".

9.5 PROJETO — PARTE VIII: GERAÇÃO DE CÓDIGO-OBJETO PARA PROCEDIMENTOS

Nesta parte do projeto deve-se programar as rotinas de geração de código para procedimentos, como visto neste capítulo. Sugerimos a seguinte seqüência de implementação:

a) Procedimentos sem parâmetros e sem variáveis locais.
b) Idem, funções.
c) Variáveis locais.
d) Parâmetros tipo variável, por valor.
e) Idem, por endereço.
f) Parâmetro tipo matriz. Este tópico não foi tratado neste capítulo. Note-se que todos os elementos do "dope vector" são conhecidos, devido à declaração do parâmetro, mesmo a origem virtual, que deve ser passada.
g) Parâmetros tipo **proc** e **func**.

Se o computador-objeto contar apenas com um indexador, pode-se reduzir a pilha D a apenas 1 elemento; o programa principal pode ter endereçamento absoluto. As variáveis que não sejam locais ao procedimento sendo executado e nem declaradas no programa principal são localizadas na PE percorrendo-se a cadeia estática em tempo de execução. Surpreendentemente, pode-se até obter maior eficiência na execução do programa-objeto organizado dessa maneira, devido a eventual redução nas seqüências de instruções, como relatado por Wirth /WIR 72/

CAPÍTULO 10

CONSIDERAÇÕES SOBRE A IMPLEMENTAÇÃO

Neste capítulo abordaremos, brevemente, alguns tópicos importantes de implementação; alguns complementam assuntos já abordados, outros introduzem novos conceitos.

10.1 REDUÇÃO DA PILHA "SEMÂNTICA"

Como se pôde observar nos capítulos 6 a 9, a pilha semântica (PSE) introduzida em 6.4 constitui um meio extremamente prático para armazenar, durante a compilação, informações que devem ser empilhadas, devido à recursividade dos elementos sintáticos. Esse aspecto advém do fato de que ela pode ser considerada como caminhando em "paralelo" à pilha sintática (PS), maneira como a encaramos naqueles capítulos. Assim, o próprio grafo sintático orientou-nos quanto à posição em que estavam os elementos da PSE em relação a seu topo ou, em alguns casos, em relação ao primeiro símbolo reconhecido no subgrafo de um não-terminal. No entanto, essa forma de implementar a PSE em paralelo à PS, a qual deve ser omitida por ser desnecessária, apresenta um grande inconveniente: inúmeras células da PSE não contêm nenhuma informação necessária para a compilação, ou sua informação já foi usada e não o será mais. Em ambos os casos, essas células poderiam ser retiradas da PSE, diminuindo sobremaneira seu tamanho. Um exemplo do primeiro caso é uma seqüência de comandos. Se a PSE caminhar em paralelo à PS, teremos duas células ("statm" e ';') para cada comando reconhecido entre **begin** e **end**. No entanto, não é necessário guardar nenhuma informação a respeito de um comando cujos reconhecimento e geração de código já terminaram. De fato, não foi necessário empilhar nenhuma informação em paralelo a "statm". Aliás, a ausência de uma rotina semântica nos nós "statm" e ';' tanto no subgrafo de "block" como no de "statm" entre **begin** e **end** já mostra que não são utilizadas informações semânticas desses nós. Um exemplo do outro caso é o de "block". Imagine-se a quantidade de células da PSE que são reservadas para armazenar eventuais informações semânticas, tanto de terminais como de não-terminais, de todas as declarações de cada bloco! Se um identificador já foi inserido na tabela de símbolos TIR, suas informações "semânticas" na PSE não são posteriormente empregadas; se o conteúdo da PSE para um não-terminal "type" é essencial para introduzir-se na TIR o índice do mesmo nos identificadores que foram reconhecidos, depois dessa introdução ele torna-se supérfluo.

Essas considerações sugerem que se elimine da PSE células supérfluas; isso pode ser feito pela execução de rotinas "semânticas" que produzem essa eliminação, diminuindo o va-

lor do índice do topo ou, ao contrário, somente inserindo informações na PSE que sejam posteriormente utilizadas. Com isso, perdemos o paralelismo estrito entre a PSE e a PS. Será necessário agora saber precisamente quais células estão faltando, para que se possa localizar as informações em relação ao topo da PSE. De qualquer modo, não se perderá totalmente a ajuda do grafo sintático, que é um auxiliar precioso na estruturação das informações "semânticas" Isso aconteceria se fossem empregadas múltiplas pilhas, por exemplo uma para cada comando, tipo etc., como acontece em muitos compiladores.

10.2 GERAÇÃO DE CÓDIGO EM LINGUAGEM DE MÁQUINA

Nos capítulos 7 a 9 geramos o código-objeto em uma linguagem de montagem ("Assembler"). Com isso evitamos o problema das referências a objetos que ainda não foram encontrados no programa-fonte, como desvios (**goto**) para a frente, que geraram instruções de desvio para endereços simbólicos, cujos rótulos foram posteriormente incorporados ao PO. No entanto, não é difícil gerar-se código diretamente em linguagem de máquina, acelerando assim o processo de compilação. Vamos aqui indicar sucintamente algumas técnicas para se gerar as referências a objetos ainda não compilados.

a) Geração de código residente

O código do PO é gerado na memória do computador em que o compilador está sendo processado. O campo de endereço de uma instrução que faz referência a um objeto ainda não reconhecido é gerado de tal forma a constituir uma lista simplesmente ligada, existindo uma lista para cada um desses objetos. Ao encontrar esse objeto, o compilador percorre essa lista, inserindo o endereço do mesmo. Esse processo pode ser feito para cada procedimento; desvios para fora do procedimento são gerados indiretamente, para entradas de tabelas que contêm os endereços dos rótulos ainda não reconhecidos.

b) Relocação

O código-objeto é gerado em um formato especial (código intermediário) e armazenado em uma unidade intermediária, como por exemplo discos magnéticos. Posteriormente, um carregador (ou "relocador") lê o código intermediário, produzindo o formato final. Nesse sistema, campos de endereços de instruções que fazem referência a objetos ainda não reconhecidos são gerados no código intermediário com índices para uma "tabela de relocação"; nesta são colocados os endereços dos referidos objetos, quando estes são reconhecidos. A tabela é incorporada ao programa-objeto, e lida pelo relocador, que a utiliza para alterar os endereços dos campos das instruções referidas acima. Em particular, relocadores são usados em computadores que não contam com registradores-base ("display" de 9.3) para alterarem os campos de endereços de todas as instruções sempre que o PO for carregado em outra área (eventualmente uma "partição") da memória.

c) Compilação em dois passos

Se o compilador tem mais do que um passo, o primeiro passo pode ser usado para deduzirem-se os endereços de todas as variáveis e rótulos. Nesse passo o programa-fonte é convertido para um código em uma linguagem intermediária simples (v. 2.1). O segundo passo lê esse código e o converte para linguagem de máquina. Os endereços que não podem ser resolvidos imediatamente durante o primeiro passo são transformados em índices para tabelas, as quais são preenchidas quando ocorrerem posteriormente as referências que permitem a resolução daqueles endereços. O segundo passo deve ter acesso a essas tabelas, podendo então gerar o código-objeto definitivo em linguagem de máquina.

10.3 COMPUTADOR-OBJETO COM ESTRUTURA DE PILHA

Existem computadores cuja estrutura ("arquitetura") é tal que as operações, por exemplo, aritméticas são efetuadas com operandos colocados previamente no topo de uma pilha.

Este é o caso por exemplo dos computadores Burroughs B-6900 e Hewlett-Packard HP-3000. Em ambos os casos, existem dois registradores rápidos que constituem a célula do topo e a imediatamente abaixo do topo. O restante das células é armazenado na memória. Uma instrução como "adicione" não usa endereços; ela automaticamente emprega as duas células do topo da pilha como operandos; estes são desempilhados e o resultado da soma é empilhado posteriormente. Neste caso, apenas um dos registradores do topo está ocupado; se nova operação binária for executada logo em seguida à primeira, um dos operandos será o resultado anterior, e o outro será o da célula vizinha ao topo, que neste caso estará no prolongamento da pilha, na memória do computador. Este segundo operando é trazido para um dos registradores, e a operação é feita normalmente. Este seria o caso de, por exemplo, (A+B)*(C+D), para a qual seriam executadas as seguintes operações, onde E_I indica o endereço da variável I, AD e MP são as instruções de adição e multiplicação, e LD a instrução que empilha o valor do seu operando no topo de pilha:

```
LD    E_A
LD    E_B
AD
LD    E_C
LD    E_D
AD
MP
```

Note-se que desta maneira não houve necessidade de se armazenar o resultado de A+B em um temporário; aliás, isso realmente se deu, pois ao se carregar C e D na pilha, o resultado de A+B teve que sair de um registrador e ser guardado na memória, na extensão da pilha, de onde foi trazido de volta para o processamento da instrução MP.

A compilação de expressões fica levemente simplificada, mas não o suficiente para justificar a introdução dessa estrutura. Note-se que nos dois computadores mencionados o fato de se ter somente dois registradores nas células do topo da pilha provoca um grande número de acessos à memória para armazenamento de temporários. A grande vantagem da estrutura de pilha é a possibilidade de se implementar instruções em linguagem de máquina para chamada e saída de procedimentos, como é justamente o caso do B-6900 e HP-3000. De fato, as seqüências de instruções das figs. 9.13, 9.14 e 9.15 reduzem-se a uma única instrução nesses computadores. Isso é essencial para a execução eficiente da chamada e saída de procedimentos, se se quiser implementar uma estrutura de blocos e recursividade. No entanto, parecenos uma mistura de dois aspectos diferentes o implementar-se por "hardware" uma pilha para o armazenamento dos dados, isto é, a "pilha de execução" PE introduzida no capítulo 9, e ao mesmo tempo utilizar-se essa pilha para os operandos das várias instruções.

10.4 O TIPO "REAL"

Esse tipo corresponde à representação interna de "ponto flutuante", isto é, uma fração decimal e uma potência de 2 ou de 10 (computadores binários ou decimais). A grande alteração que deve ser introduzida no compilador aqui estudado é a do Analisador Léxico. Ele deve ler as constantes de tipo "real", calcular seu valor e produzir uma distinção entre estas e as inteiras, de modo que elas sejam inseridas ou encontradas em uma tabela de constantes flutuantes. Todas as operações aritméticas devem fazer a verificação dos tipos para a conversão dos mesmos quando necessária, já que em PASCAL é permitida a mistura de tipos "real" e "integer" em operações aritméticas.

No grafo da PASCAL do apêndice I não fizemos distinção entre os tipos "real" e "integer" em relação às constantes numéricas. Essa distinção pode ser feita por meio de rotinas semânticas, as quais podem, por exemplo, verificar que um rótulo deve ser um inteiro.

10.5 IDENTIFICADORES COM COMPRIMENTO LIVRE

Na implementação descrita nos capítulos anteriores limitamos os identificadores — e palavras reservadas — a no máximo 6 caracteres. Para levantar-se essa restrição pode-se introduzir uma tabela ID em forma de matriz unidimensional de caracteres, e uma segunda tabela INDID contendo dois inteiros por entrada: o primeiro aponta para o início de um identificador na ID e o segundo o número de caracteres do identificador. Por exemplo, podemos ter

```
var ID: array [1..MAXID] of char;
    INDID: array (1..MAXINDID] of record IND:1..MAXIND;
                                         TAM:1..MAXTAM
                                  end;
```

As entradas da Tabela de Identificadores e Rótulos (TIR) teriam no campo IDROT um índice para o INDID. O mesmo pode-se aplicar para a tabela de símbolos reservados TABT. É preciso alterar o analisador léxico, para este dar como resultado parâmetros p_1 (no caso de palavras reservadas) e p_2 que admitam o reconhecimento de unidades léxicas com um número qualquer de caracteres. Nesse sentido, o AL pode dar como resultado um parâmetro que aponta para o início da unidade léxica na cadeia de entrada, e um parâmetro que dá seu comprimento.

10.6 GENERALIZAÇÃO DA ESTRUTURA DE BLOCOS EM PASCAL

Ao nosso ver, um dos grandes defeitos da linguagem PASCAL é a restrição da estrutura de blocos somente ao nível de procedimentos, como vimos em 6.6. Não é difícil estender-se a sintaxe para comportar blocos no sentido do ALGOL-60, isto é, permitir declarações na primeira parte de qualquer construção **begin**...**end**. Para isso, bastaria definir

<block>::= **begin** <declaration list> <statement list> **end** | λ

em que <declaration list> pode ser vazio, tendo-se neste último caso um comando composto como em PASCAL. Deve-se eliminar o ramo **begin** de "statm" e substituir a sua λ-alternativa por "block" (v. apêndice I).

Do ponto de vista da análise de contexto, o tratamento da tabela de símbolos do compilador seguiria exatamente as indicações dadas para o caso da PASCAL. Na geração de código, é necessário reservar para o programa principal e cada procedimento uma área na pilha de execução correspondente à *área máxima* requerida pelas variáveis locais de cada um deles. Conservando-se as matrizes com limites constantes nos índices, esse máximo pode ser facilmente deduzido pelo compilador. Para isso, pode-se usar a variável DES introduzida em 9.4b que dá o deslocamento, em relação à base, da próxima variável do programa principal ou do procedimento sendo compilado. Todas as vezes que se reconhece a declaração de uma nova variável, incrementa-se o valor de DES com o espaço necessário para a mesma; nesse momento, o valor de DES é comparado com uma variável como DESMAX, que dá o valor máximo atingido por DES dentro de um procedimento ou programa principal. Faz-se

```
if DES> DESMAX then DESMAX:= DES;
```

Quando se atinge o fim de um bloco, DES deve ser decrementado do espaço ocupado na pilha de execução por esse bloco — o que pode ser deduzido a partir do deslocamento da primeira variável declarada no bloco.

A técnica descrita acima foi empregada com sucesso na implementação da linguagem LAPA /MEL 78/ desenvolvida por Setzer e Bressan /BRE 77/.

10.7 ENTRADA/SAÍDA

Devido a problemas de espaço, deixamos de abordar a questão de como compilar os procedimentos de entrada/saída. Os procedimentos reset, rewrite, get e put e a função eof /J-W 74/ não apresentam maiores problemas, já que sua sintaxe segue exatamente a de pro-

cedimentos e funções, eles devem apenas ser pré-declarados, isto é, constarem da inicialização da tabela de identificadores e rótulos. Já os procedimentos write, read, writeln, readln devem ser tratados de maneira especial pelo compilador, já que podem ter um número variável de parâmetros; além disso, write e writeln podem ter especificações de formatação que fogem totalmente à sintaxe de parâmetros, como por exemplo write (output, PESO :10:3). O número variável de parâmetros não é difícil de ser tratado, bastando simplesmente inibir o teste de número de parâmetros e a verificação de seu tipo. A verificação da sintaxe da especificação de formato pode ser feita dentro de uma rotina semântica adequada.

10.8 OTIMIZAÇÃO DO CÓDIGO-OBJETO

A otimização do código-objeto constitui-se por si só um campo de pesquisa na área de compilação. Existem otimizações locais, isto é, que dizem respeito a cada unidade do programa-fonte — em geral um comando —, e globais, que envolvem a análise do contexto de um comando em relação e outros, em geral adjacentes a ele. Nessa última categoria, inclui-se por exemplo a movimentação de comandos "invariantes" para fora de malhas ("while", "repeat", "for"). Assim, por exemplo, se o comando A:= B encontra-se dentro de uma malha em que o valor de B não é alterado e A não ocorre em nenhum dos comandos da malha, então A: = B pode ser removido para fora da malha. Em geral otimizações globais exigem mais do que um passo de compilação.

Uma otimização local simples de ser obtida em nosso caso é a da eliminação de LDA's e STA's supérfluos. Da maneira como geramos o código-objeto, um comando como A = B + C + D geraria o seguinte código, onde E_I é o endereço da variável I e T indica um temporário:

```
LDA   E_B
ADD   E_C
STA   T_0   }
LDA   T_0   }   /* supérfluas */
ADD   E_d
STA   T_0   }
LDA   T_0   }   /* supérfluas */
STA   E_A
```

Há várias técnicas para eliminar-se a geração das instruções supérfluas. Citemos duas delas:

a) A geração de uma instrução STA T_k, onde T_k é um temporário, é adiada até a geração da próxima instrução. Se esta for LDA T_m com m = k, então as duas são eliminadas, não sendo introduzidas no programa-objeto. Essa técnica não elimina a geração supérflua de instruções como em A: = B*(C + D), pois tudo se passa como se a expressão fosse reduzida para $B*T_0$, o que provocaria a geração de LDA E_B em primeiro lugar. Esse problema é resolvido testando-se a ordem inversa dos operandos para o caso de operadores comutativos como '+' e '*'.

b) Em uma expressão, a geração de uma instrução STA T_k que envolve um temporário T_k só é feita quando T_k atinge a terceira posição de operando na pilha semântica PSE, do topo para o fundo. Nesse caso, gera-se STA T_k e marca-se o temporário na PSE, que indicaremos aqui por $\dot{T}_k$. Se as instruções correspondentes a um operador tiverem que ser geradas, verifica-se se um dos seus operandos é um temporário marcado; se ele for marcado, deve-se gerar a instrução LDA para o mesmo. Se ele não for marcado, significa que seu valor ainda se encontra no acumulador, e não deve ser gerado um LDA para o mesmo.

Essa técnica pode ser empregada também para computadores-objeto com acumuladores múltiplos. Por exemplo, com dois acumuladores somente é armazenado na memória o temporário que atinge a 4ª posição de operando na PSE.

10.9 AUTO-IMPLEMENTAÇÃO

Traduziremos por "auto-implementação" a técnica denominada de "bootstrap" ("fazer-se — ou subir na vida — por si próprio"). Vamos abordar sucintamente uma versão particular dessas técnicas, apenas como ilustração. Para uma resenha das mesmas no vernáculo, consulte-se a dissertação de Sanches /SAN 79/.

Vamos introduzir as seguintes anotações: a) Um programa P escrito numa linguagem L_1 é denotado por $P[L_1]$. Note-se que "escrito" pode resultar de um processo manual de programação ou de uma conversão automática, isto é, uma compilação. b) Um compilador C escrito numa linguagem L_1 que converte programas-fonte escritos em L_2 para programas-objeto escritos em L_3 é denotado por $C[L_1]:L_2 \rightarrow L_3$. c) Dado um programa $P[L_2]$, o resultado de sua compilação por $C[L_1]:L_2 \rightarrow L_3$ é um programa $P[L_3]$; esse processo é denotado por $C(P[L_2]) = P[L_3]$.

Suponhamos agora que exista um compilador PASCAL C1 que processa em um computador M1, produzindo programa-objeto na linguagem de máquina L_{M1} do computador M1, isto é, $C1[L_{M1}]:PASCAL \rightarrow L_{M1}$. Queremos implementar um compilador PASCAL para uma máquina M2. Nesse caso: i) Escrevemos manualmente em PASCAL um compilador C2 que converte PASCAL em L_{M2}, isto é, $C2[PASCAL]:PASCAL \rightarrow L_{M2}$. A construção desse compilador é facilitada pelo fato de ele ser escrito (manualmente) em linguagem de alto nível. ii) Carrega-se C1 em M1 e compila-se C2, obtendo-se C3, isto é, $C1(C2[PASCAL]) = C3[L_{M1}]$, onde $C3[L_{M1}]:PASCAL \rightarrow L_{M2}$. iii) Carrega-se C3 em M1 e compila-se o programa C2, obtendo C4, isto é, $C3(C2[PASCAL]) = C4(L_{M2})$, onde $C4[L_{M2}]:PASCAL \rightarrow L_{M2}$. C4 é o compilador procurado, que processa na máquina M2.

Note-se que, nesse processo de auto-implementação, C1 não necessita ser obrigatoriamente escrito em L_{M1}; ele pode ser escrito por exemplo em ALGOL e ser processado por meio de um interpretador que processa na máquina M1.

BIBLIOGRAFIA

/A-U 72/ Aho, A. V. & Ullman, J. D. — *The theory of Parsing, translation and compiling*, vol. 1: Parsing, Prentice-Hall, Englewood Cliffs, 1972.

/BRE 77/ Bressan, G. — *Linguagens de Implementação de Sistemas e a Linguagem LAPA*, Dissertação de Mestrado, IME-USP, São Paulo, dez. 1977.

/GRI 71/ Gries, D. — *Compiler Construction for Digital Computers*, Wiley, New York, 1971.

/H-U 69/ Hopcroft, J. E. & Ullman, J. D. — *Formal languages and their relation to automata*, Addison-Wesley, Reading, 1969.

/J-W 74/ Jensen, K. & Wirth, N. — *PASCAL user manual and report*, Springer-Verlag. New York, 1974.

/KNU 73/ Knuth, D. E. — *The art of computer programming*, vol. 3: Sorting and Searching, Addison-Wesley, Reading, 1973.

/KOW 83/ Kowaltowski, T. — *Implementação de linguagens de Programação*, Guanabara Dois, Rio de Janeiro, 1983.

/L-S 68/ Lewis, P. M. & Stearns, R. E — Syntax-directed transduction, *Journal of the ACM*, 15,3, 1968, pp. 464-468.

/MEL 78/ Melo, I. S. H. de — *Alguns tópicos de compilação e uma implementação da linguagem LAPA para o computador PADE*, Diss. Mestrado, Instituto de Matemática e Estatística da USP, São Paulo, set. 1978.

/NAU 63/ Naur, P. (Ed.) — Revised Report on the algorithmic language ALGOL 60, *Comm. ACM*, jan. 1963, pp. 1-17.

/REC 73/ Rechenberg, P. — Sackgassenfreie Syntaxanalyse, *Elektronische Rechenanlagen*, 15, 3,4 1973, pp. 119-125, 170-176.

/R-D 76/ Ripley, G. D. & Druseikis, F. C. — *Towards a Compiler Error Recovery Effecctiveness Rating*, Technical Report, Computer Science Department, The University of Arizona, Tucson, Arizona, April 1976.

/R-S 70/ Rosenkrantz, D. J. & Stearns, R. E. — Properties of deterministic top-down grammars, *Information and Control*, 17, 1970, pp. 226-256.

/SAL 69/ Salomaa, A. — *Theory of automata*, Pergamon Press, Oxford, 1969.

/SAN 79/ Sanches, M. M. — *Portabilidade de Compiladores*, Dissertação de Mestrado, Instituto de Matemática e Estatística da USP, São Paulo, mar. 1979.

/SET 79/ Setzer, V. W. — Non-recursive Top-down Syntax Analysis, *Software-Practice and Experience*, vol. 9, 1979, pp. 237-245.

/VEL 79/ Veloso, P. A. S. — *Máquinas e linguagens: uma introdução à teoria de autômatos*, Escola de Computação, Instituto de Matemática e Estatística da UPS, São Paulo, 1979.

/WIJ 75/ van Wijngaarden et alii — Revised-Report on the Algorithmic language ALGOL 68, *Acta informatica*, 5, 1-3, 1975, pp. 1-236.

/WIR 71/ Wirth, N. — Design of a PASCAL compiler, *Software-Practice and Experience*, vol. 1, 1971, pp. 309-333.

/WIR 72/ Wirth, N. — *On "PASCAL", code generation, and the CDC 6000 Computer*, STAN-CS-72-257, Computer Science Dept., Stanford University. Feb. 1972.

/WIR 76/ Wirth, N. — *Algorithms + Data Structures = Programs*, Prentice-Hall, Englewood Cliffs, New York, 1976.

APÊNDICE I

GRAMÁTICA ESLL(1) E GRAFO SINTÁTICO DA LINGUAGEM PASCAL

Na gramática e no grafo sintático dados a seguir, foram empregadas as seguintes abreviaturas:

não-terminais

const — constant
sitype — simple type
filist — field list
infipo — index, field, pointer
siexpr — simple expression
expr — expression
palist — parameter list
statm — statement
progrm — program

terminais

NUMB — number
STRING — cadeia de caracteres
IDEN — identifier
COIDEN — constant identifier
FIIDEN — field identifier
VAIDEN — variable identifier
FUIDEN — function identifier
TYIDEN — type identifier
PRIDEN — procedure identifier

const ::= STRING | ['+' | '−'] (COIDEN | NUMB)

sitype ::= TYIDEN | '(' {IDEN ‖ ',' }+ ')' | const '..' const

type ::= '↑' TYIDEN | [**packed**] (**array** '[' {sitype ‖ ',' }+ ']' **of** type |
file of type | **set of** sitype | **record** filist **end** | sitype)

filist ::= {[{IDEN ‖ ',' }+ ':' type] ‖ ';'}+
[**CASE** [IDEN ':'] TYIDEN **of**
{[{STRING | ('+' | '−') (COIDEN | NUMB) | COIDEN | NUMB ‖ ',' }+ ':'
'(' filist ')'] ‖ ';'}+]

infipo ::= { '[' {expr ‖ ','}+ ']' | '.' FIIDEN | '↑' }*

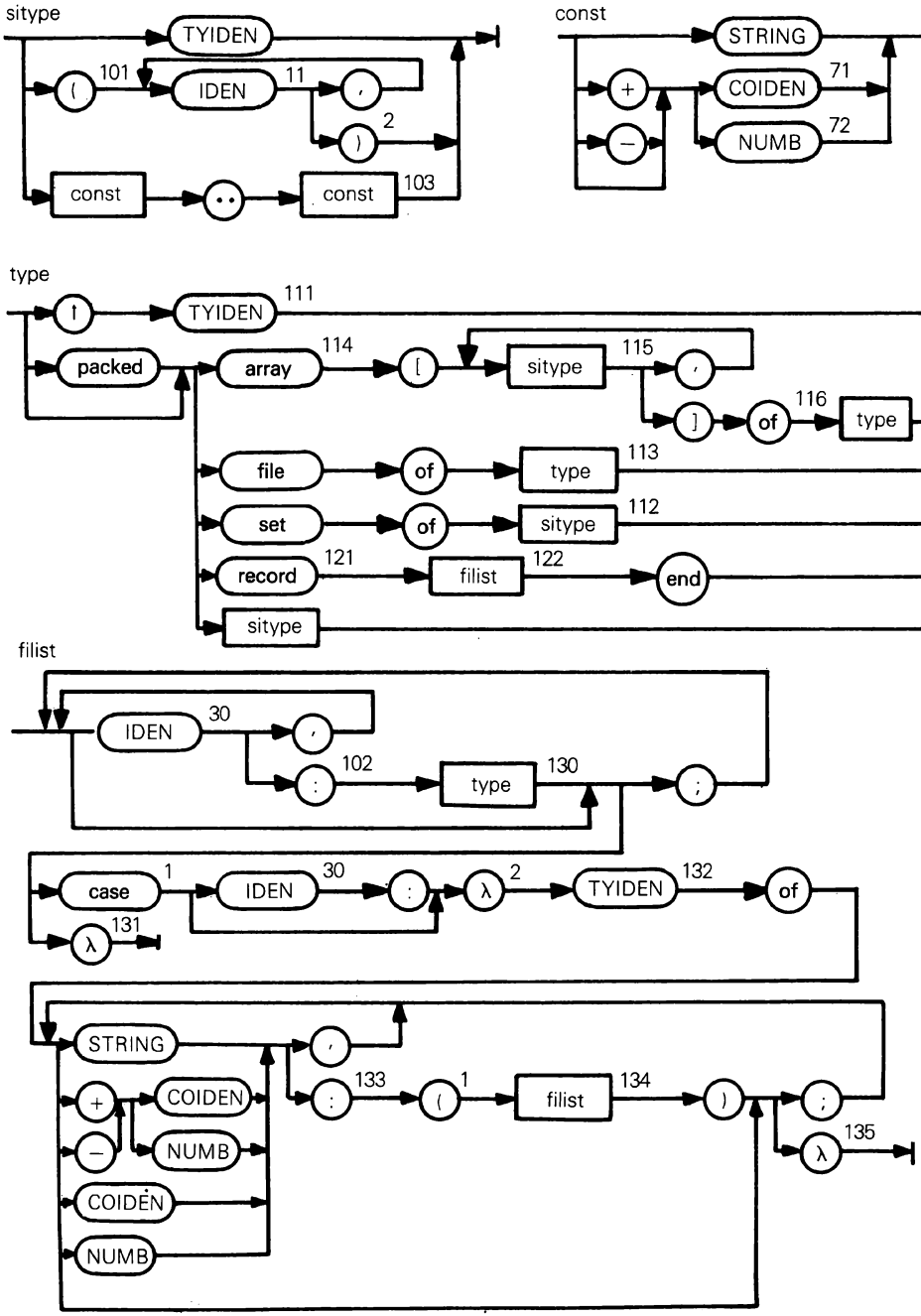
factor ::= COIDEN | NUMB | **nil** | STRING | VAIDEN infipo |
FUIDEN ['(' {expr ‖ ','}+ ')'] | '(' expr ')' | **not** factor |
'[' (']' | {expr ['..' expr] ‖ ','}+ ']')

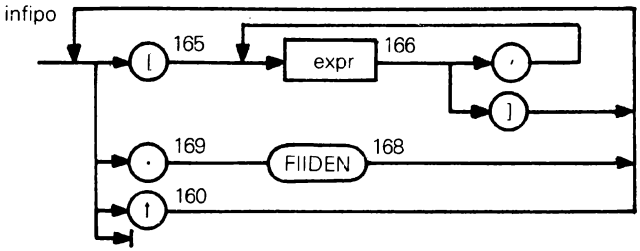
term ::= {factor ‖ '*' | '/' | **div** | **mod** | **and**}+

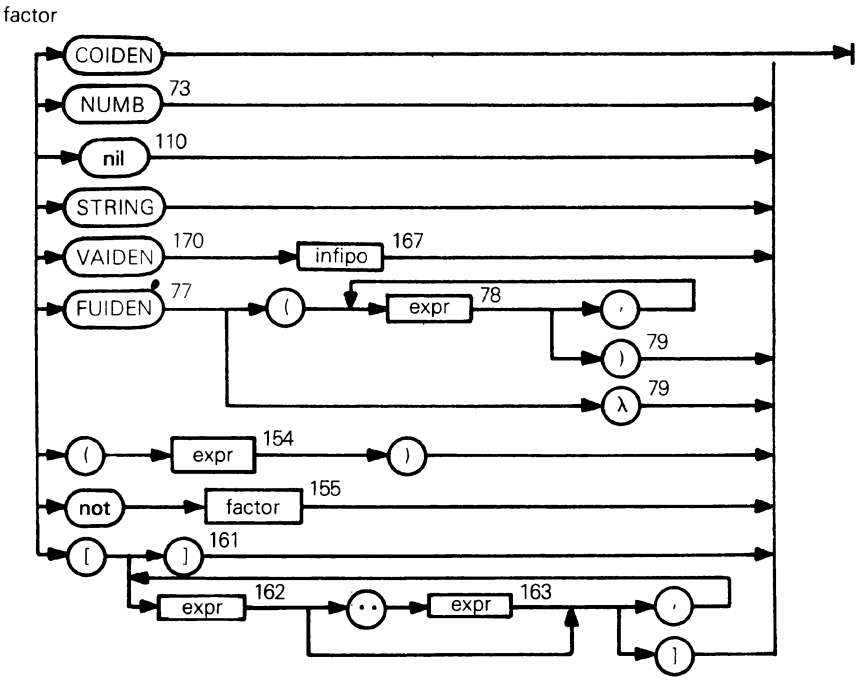
siexpr ::= ['+' | '−'] {term ‖ '+' | '−' | **or**}+

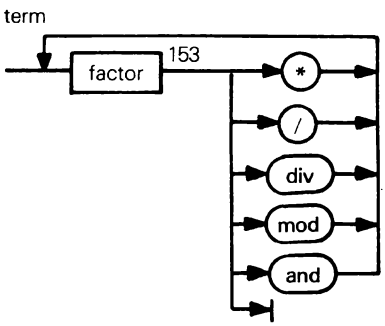
```
expr    ::= siexpr [('=' | '<' | '>' | '<>' | '>=' | '<=' | in) siexpr]
palist  ::= [ '(' {proc {IDEN ‖ ','}+ | [func | var] {IDEN ‖ ','}+':' TYIDEN ‖ ';' }+')' ]
block   ::= [label {NUMB ‖ ','}+';']
            [const IDEN {'=' const ';' ‖ IDEN}+]
            [type IDEN {'=' type ';' ‖ IDEN}+]
            [var IDEN {{ ',' IDEN}* ':' type ';' ‖ IDEN}+]
            [{(proc IDEN palist | func IDEN palist ':' TYIDEN) ';' block ';'}*]
            begin {statm ‖ ';'}+ end
statm   ::= [NUMB ':'] [(VAIDEN infipo | FUIDEN) ':=' expr |
            PRIDEN [ '(' {expr | PRIDEN ‖ ','}+')' ] |
            begin {statm ‖ ';'}+ end |
            if expr then statm [else statm] |
            case expr of { [ {(STRING | (+ | -) (COIDEN | NUMB)
                              | COIDEN | NUMB) ‖ ','}+ ':' statm] ‖ ';'}+ end |

            while expr do statm |
            repeat {statm ‖ ';'}+ until expr |
            for VAIDEN infipo ':=' expr (to | downto) expr do statm |
            with {VAIDEN infipo ‖ ','}+ do statm |
            goto NUMB]
progrm  ::= progr IDEN '(' {IDEN ‖ ','}+')' ';' block '.'
```

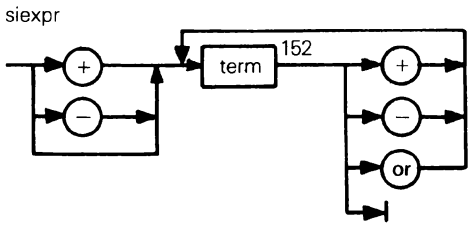
Alguns nós do grafo contêm uma menção a uma rotina semântica que deve ser chamada após o reconhecimento sintático do nó. Essa menção é feita colocando-se o número da rotina à direita do respectivo nó.

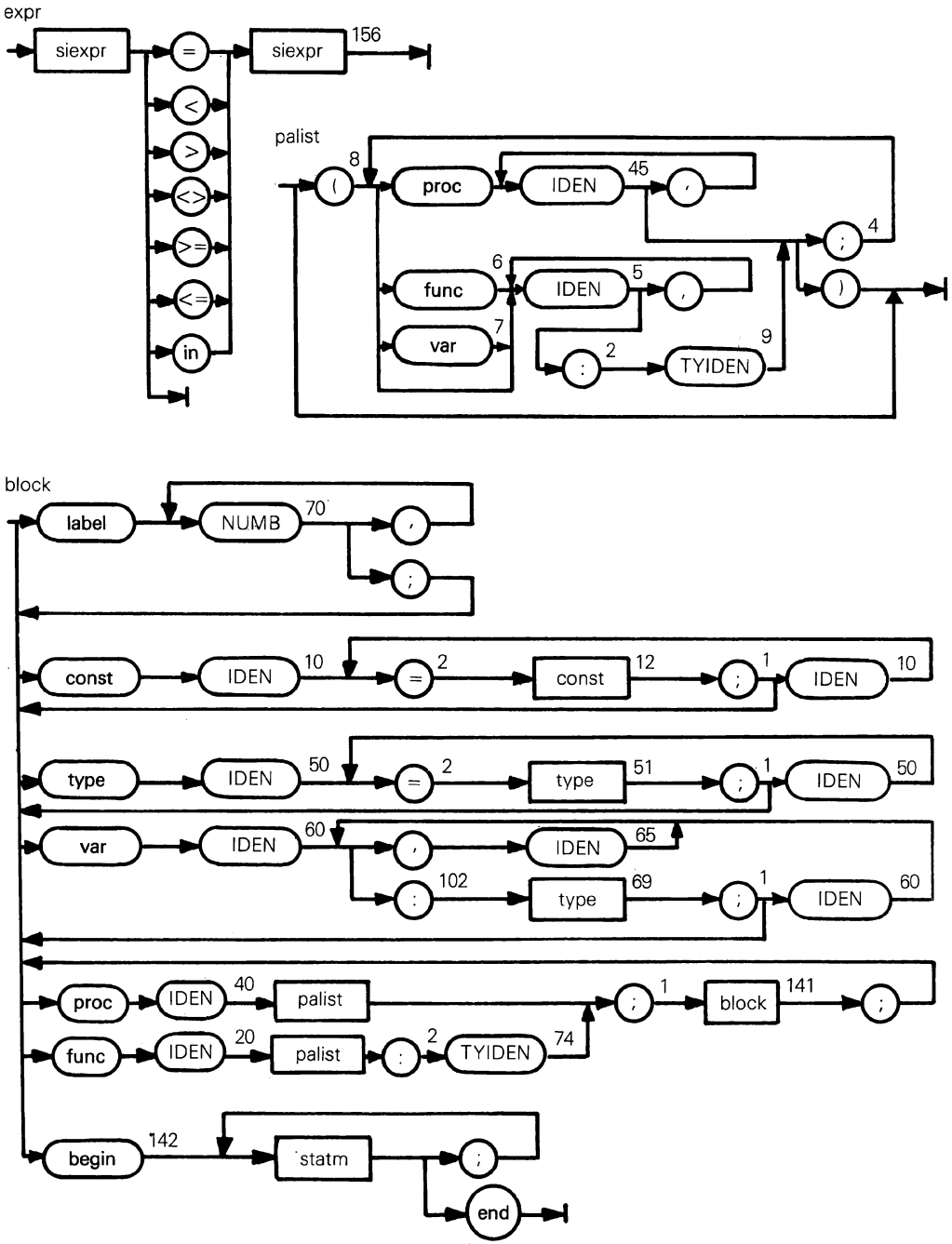

sitype
TYIDEN
(
101
IDEN
11
,
)
2
const
..
const
103
const
STRING
+
−
COIDEN
71
NUMB
72
type
↑
TYIDEN
111
packed
array
114
[
sitype
115
,
]
of
116
type
117
file
of
type
113
set
of
sitype
112
record
121
filist
122
end
sitype
filist
IDEN
30
,
:
102
type
130
;
case
1
IDEN
30
:
λ
2
TYIDEN
132
of
λ
131
STRING
+
−
COIDEN
NUMB
COIDEN
NUMB
,
:
133
(
1
filist
134
)
;
λ
135


infipo
(
165
expr
166
,
)
.
169
FIIDEN
168
↑
160

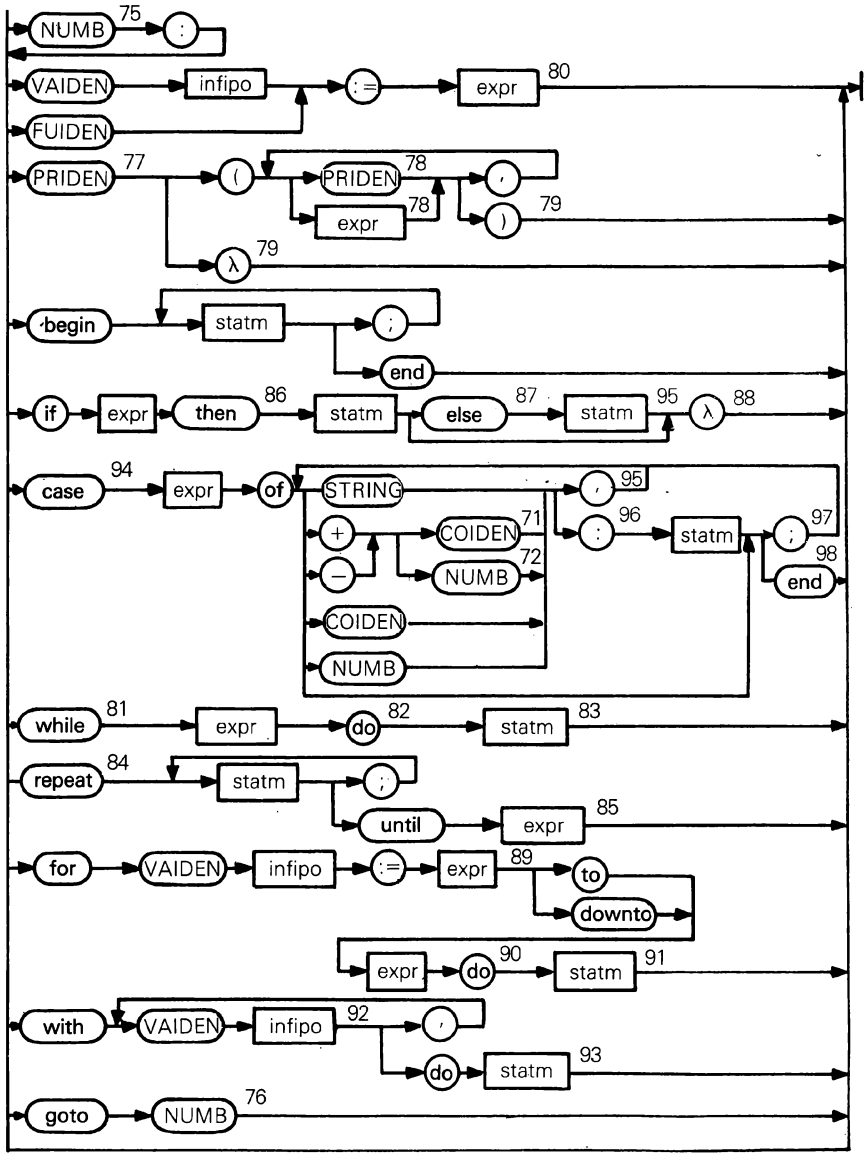

factor
COIDEN
NUMB
73
nil
110
STRING
VAIDEN
170
infipo
167
FUIDEN
77
(
expr
78
,
)
79
λ
79
(
expr
154
)
not
factor
155
[
]
161
expr
162
..
expr
163
,
]


term
factor
153
*
/
div
mod
and

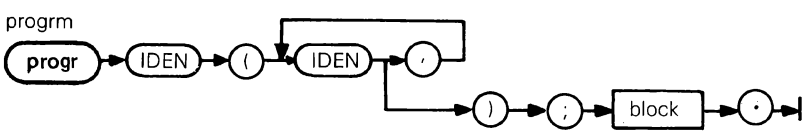

siexpr
+
−
term
152
+
−
or


expr
siexpr
=
<
>
<>
>=
<=
in
siexpr
156
palist
8
(
proc
IDEN
45
,
;
4
)
func
6
IDEN
5
,
var
7
:
2
TYIDEN
9
block
label
NUMB
70
,
;
const
IDEN
10
=
2
const
12
;
1
IDEN
10
type
IDEN
50
=
2
type
51
;
1
IDEN
50
var
IDEN
60
,
IDEN
65
:
102
type
69
;
1
IDEN
60
proc
IDEN
40
palist
;
1
block
141
;
func
IDEN
20
palist
:
2
TYIDEN
74
begin
142
statm
;
end

statm



progrm



APÊNDICE II

ÍNDICE DE ROTINAS "SEMÂNTICAS"

As rotinas abaixo referem-se às rotinas semânticas RS_n do grafo sintático da linguagem PASCAL apresentado no apêndice I. Para facilitar a localização de cada rotina no grafo, são dadas, juntamente com o número da mesma, indicações sobre a localização do nó em que ela se encontra, constando de: subgrafo (isto é, não-terminal à esquerda da produção), ramo do subgrafo e o nó propriamente dito. As rotinas marcadas com + chamam a RS_1 depois de concluídas e com * chamam a RS_2.

n	subgrafo	ramo	nó	n	subgrafo	ramo	nó
1	progrm	**progr**	**progr**	30	filist	**case**	IDEN
"	block	**const**	;	40	block	**proc**	IDEN
"	"	**type**	;	45	palist	**proc**	IDEN
"	"	**var**	;	50*	block	**type**	IDEN
"	"	**proc**	;	"	"	"	"
"	filist	**case**	**case**	51	"	"	type
"	"	**string**	(	60	"	**var**	IDEN
2	palist	**func**	:	"	"	"	"
"	block	**const**	=	65	"	"	"
"	"	**type**	=	69	"	"	type
"	sitype	(	)	70	"	**label**	NUMB
"	filist	**case**	λ	71	const	+	COIDEN
4+	palist	**proc**	;	"	statm	**case**	"
5	"	**func**	IDEN	72	"	"	NUMB
6	"	"	**func**	"	const	+	"
7	"	**var**	**var**	73	factor	NUMB	NUMB
8	"	(	(	74	block	**func**	TYIDEN
9	"	**func**	TYIDEN	75	statm	NUMB	NUMB
10*	block	**const**	IDEN	76	"	**goto**	NUMB
10*	"	"	"	77	statm	PRIDEM	PRIDEN
11	sitype	(	IDEN	"	factor	FUIDEN	FUIDEN
12	block	**const**	const	78	statm	PRIDEN	expr
20	block	**func**	IDEN	"	"	"	PRIDEN
30	filist	IDEN	IDEN	"	factor	FUIDEN	expr

n	subgrafo	ramo	nó	n	subgrafo	ramo	nó
79	factor	FUIDEN	)	114	type	**array**	**array**
"	"	"	λ	115	"	"	sitype
"	statm	PRIDEN	"	116	"	"	**of**
"	"	"	)	117	"	**array**	**type**
80	"	VAIDEN	expr	121+	"	**record**	**record**
81	"	**while**	**while**	122	"	"	filist
82	"	"	**do**	130+	filist	IDEN	type
83	"	"	statm	131	"	λ	λ
84	"	**repeat**	**repeat**	132	"	**case**	TYIDEN
85	"	"	expr	133	"	STRING	:
86	"	**if**	**then**	134	"	"	filist
87	"	"	**else**	135	"	"	λ
88	"	"	λ	141+	block	**proc**	block
89	"	**for**	expr	142*	"	**begin**	**begin**
90	"	"	**do**	150	siexpr	+	+
91	"	"	statm	151	"	−	−
92	"	**with**	infipo	152	"	+	term
93	"	"	statm	153	term	factor	factor
94	"	**case**	**case**	154	factor	(	expr
95	"	"	,	155	factor	**not**	factor
96	"	"	:	156	expr	siexpr	siexpr
97	"	"	;	160	infipo	↑	↑
98	"	"	**end**	161	factor	[	[
100	progrm	**progr**	(	162	factor	[	expr
101+	sitype	(	(	163	"	"	"
102*	block	**var**	:	165	infipo	[	[
" *	filist	IDEN	:	166	"	[	expr
103	sitype	const	const	167	factor	VAIDEN	infipo
110	factor	**nil**	**nil**	168	infipo	.	FIIDEN
111	type	↑	TYIDEN	169	"	"	.
112	"	**set**	sitype	170	factor	VAIDEN	VAIDEN
113	"	**file**	type	180	progrm	**progr**	

ROTINAS "SEMÂNTICAS" REFERIDAS NO TEXTO

RS	págs.	RS	págs.
1	83, 96, 97	50	83, 99
2	83, 93, 94	51	99
5	83, 99, 150	60	93, 150
6	99	65	93, 150
7	99	69	94
9	82, 99	70	99, 106
10	99	71	95, 99
11	93, 96	72	95
12	99	73	95
20	100, 102, 103, 147, 152	74	100
30	98	75	106
40	83, 99, 102, 103, 147, 152	76	107, 148
45	83, 99, 150	77	149, 150

APÊNDICE III

RESUMO DO COMPUTADOR-OBJETO HIPO

O HIPO é um computador hipotético simulado no sistema B-6900, desenvolvido pelo Departamento de Matemática Aplicada do Instituto de Matemática e Estatística da USP. Ele foi implementado com finalidades didáticas. Além do simulador da máquina HIPO, que aceita programas na sua linguagem de máquina, existe um montador ("Assembler") que converte a linguagem de montagem HAL para a linguagem de máquina. Veremos aqui, sucintamente, apenas a parte das especificações do HIPO e da linguagem HAL que foram utilizadas neste texto.

a) Memória

O HIPO conta com memória de 10 000 palavras endereçadas de 0000 a 9999, e usadas indistintamente para instruções e dados. Cada palavra tem o formato

SDDDDDDDDDD

onde $S \in \{+, -\}$ e $D \in \{0,1,\ldots,9\}$.

Numa palavra pode ser representado um número inteiro com 10 algarismos ou 5 caracteres alfanuméricos codificados conforme tabela dada adiante em (h). Não daremos aqui a representação em ponto flutuante.

As palavras de endereço 0001 a 0009 podem ser usadas como indexadores.

b) Instruções do HIPO

As instruções mencionadas neste texto têm o formato

+00ccxieeee

onde cc é o código de operação da instrução; x indica o indexador a ser usado, isto é, deve ser usado o indexador que está na palavra de endereço 000x; i indica o uso ($i = 1$) ou não ($i = 0$) de endereçamento indireto; eeee é o endereço referido pela instrução, sujeito a modificações devido ao uso de indexador e endereçamento indireto, produzindo o endereço efetivo E que é dado pela seguinte fórmula, onde [P] indica o conteúdo da palavra de endereço P:

$E = eeee + [x]$ se $i = 0$

$E = [eeee + [x]]$ se $i = 1$; se na palavra de endereço $eeee + [x]$ houver $x \neq 0$ ou $i \neq 0$, então toma-se E como eeee e repete-se as duas fórmulas.

c) Operações do HAL

O montador HAL utiliza operações no seguinte formato, nos registros de entrada:

Coluna 1: rótulo de até 6 letras e dígitos, encabeçado por uma letra, ajustado à esquerda
" 7: ⍽
" 8: código mnemônico da operação,
" 12: ⍽
" 13: número do indexador
" 14: I, indicando o uso de endereçamento indireto, ou ⍽ em caso contrário
" 15: ⍽
" 16: operando

d) Acumulador

Algumas instruções usam um registrador especial, o Acumulador, que não é endereçável, representado no que segue por A; [A] representa o conteúdo de A.

e) Instruções do HIPO e operações correspondentes em HAL

Instrução em HIPO	Operação em HAL	execução
11	LDA	A ← [E]
12	STA	E ← [A]
15	LZR	carrega em A, nas respectivas posições, os zeros de [E], mantendo inalteradas as demais posições.
16	LDG	carrega em A, nas respectivas posições, os dígitos não-nulos de [E], mantendo inalteradas as demais posições.
21	ADD	A ← [A] + [E]
22	SUB	A ← [A] − [E]
23	MPY	A ← [A] × [E]
24	DIV	A ← [A] ÷ [E]
25	MOD	A ← resto ([A] ÷ [E])
29	RVS	A ← −[A]
31	RNW	transfere numericamente as colunas 1 a 11 de um registro de entrada para E.
35	RAW	transfere alfanumericamente as colunas 1 a 5 de um registro de entrada para E.
41	PNW	impressão numérica de uma palavra.
45	PAW	impressão alfanumérica de uma palavra.
51	BRN	Desvia para E
52	BNP	Se [A] $\leq$ 0, desvia para E
53	BNZ	Se [A] $\neq$ 0, " " "
54	BPS	Se [A] > 0, " " "
55	BZR	Se [A] = 0, " " "
56	BNG	Se [A] < 0, " " "
57	BNN	Se [A] $\geq$ 0, " " "
59	BST	Desvia para E+1 e guarda o endereço seguinte ao desta instrução BST, em E.
61	SLA	Desloca [A] para a esquerda, preenchendo à direita com zeros (*).
62	SRA	Desloca [A] para a direita, preenchendo à esquerda com zeros (*). (*) O número de posições deslocadas é dado pelas duas últimas casas de eeee, se x=0, ou por |[x]|, se x≠0.
65	MDX	1) x ← [x] + eeee; 2) se, após (1), [x] mudou de sinal ou ficou nulo, pula a próxima instrução.
66	MNX	1) x ← [x] − eeee;

2) se, após (1), x mudou de sinal ou ficou nulo, pula a próxima instrução.

70	STP	Pare
93	LAD	A ← E

f) Pseudo-operações do HAL

f_1) DS (defina símbolo). Esta instrução reserva espaços para uma ou mais palavras de memória. O número de palavras a serem reservadas é indicado no campo de operando e a sua ausência indica a reserva de uma palavra. Se o comando tiver um rótulo, o endereço da primeira destas palavras reservadas é associado ao rótulo.

f_2) DC (defina constante). Reserva uma palavra na qual será carregada uma constante especificada no campo de operando. A constante pode ser numérica ou alfanumérica. Neste último caso, a constante deve ser colocada entre aspas, não devendo ultrapassar 5 caracteres. Se o comando possuir um rótulo, o endereço da palavra é associado ao mesmo.

f_3) EQ (equivalência de rótulos). O uso de um símbolo no campo de rótulo é obrigatório. Esta pseudo-operação associa ao símbolo do rótulo o mesmo enderèço associado ao operando, que pode ser de qualquer natureza, exceto literal.

f_4) END (encerra o programa-fonte). Esta pseudo-operação encerra o programa em HAL.

g) Operandos em HAL

O campo de operando em HAL pode conter: 1) Um número inteiro, que nas operações representa o valor de eeee. 2) Um rótulo, cujo endereço representa o valor de eeee. 3) Uma expressão do tipo R + C ou R − C, onde R é um rótulo e C é um número inteiro; neste caso o valor de eeee será a soma ou subtração do endereço associado ao rótulo com a constante. 4) Um literal numérico ou alfanumérico, obrigatoriamente precedido do sinal '='. O literal gera uma constante em HIPO e o endereço dessa constante torna-se o valor de eeee. 5) Asterisco '*'; o valor de eeee é o da palavra que está sendo montada. 6) * + C ou * − C, onde C é um número inteiro; análogo a (3) acima, usando a interpretação de '*' de (5).

h) Tabela de códigos alfanuméricos

00	⍽	30	)	41	'	71	H	82	S	90	0
20	[	31	;	42	=	72	I	83	T	91	1
21	.	32	¬	43	"	73	J	84	U	92	2
22	<	33	/	44	!	74	K	85	V	93	3
23	(	34	,	64	A	75	L	86	W	94	4
24	+	35	%	65	B	76	M	87	X	95	5
25	|	36	–	66	C	77	N	88	Y	96	6
26	&	37	>	67	D	78	O	89	Z	97	7
27	]	38	:	68	E	79	P			98	8
28	$	39	#	69	F	80	Q			99	9
29	*	40	@	70	G	81	R				

APÊNDICE IV

ABREVIATURAS

ABREV	SIGNIFICADO	PÁG
AC	— Analisador de contexto	19
AF	— Autômato de estados finitos	21
AL	— Analisador léxico	17
AS	— Analisador sintático	17
DV	— "Dope vector"	117
ER	— Expressão regular	31
ERE	— Expressão regular estendida	41
ERE-Gramática	— GLC de produções com ERE à direita	41
ESLL(1)	— "Extended simple LL(1)"	51
GC	— Gerador de código	19
GLC	— Gramática livre de contexto	35
GR	— Gramática regular	33
PO	— Programa-objeto	106
PS	— Pilha sintática	65
PSE	— Pilha semântica	91
RS_n	— Rotina "semântica" de número n	78
TABNT	— Tabela de não-terminais	58
TABT	— Tabela de terminais	58
TCA	— Tabela de constantes alfanuméricas	81
TCN	— Tabela de constantes numéricas	81
TIR	— Tabela de identificadores e de rótulos	81

Índice Analítico

As páginas referidas em negrito dizem respeito às entradas principais dos itens no texto.

Impressão e acabamento
(com filmes fornecidos):
EDITORA SANTUÁRIO
Fone (0125) 36-2140
APARECIDA - SP

VAMOS NOS CONHECER MELHOR!

☐ **SIM** por favor enviem mais informações sobre os livros de computação da **Editora Campus.**

Livro Adquirido: .....

Nome: .....

Endereço: .....

Estado: ..... Cidade ..... CEP: .....

Profissão: ..... Função: .....

Instituição: .....

Conhecer melhor nossos leitores nos ajuda a produzir nossos futuros livros de uma forma mais precisa. Pedimos-lhe que reserve um momento para responder as seguintes perguntas:

1. Onde você comprou este livro?
   - ☐ Livraria .....
   - ☐ Estado .....
   - ☐ Mala Direta Campus
   - ☐ Outros: Local .....
2. Quantos livros de computação você compra por ano?
   - ☐ 1-2
   - ☐ 3-5
   - ☐ 6-10
   - ☐ 11-15
   - ☐ Mais de 15
3. Onde você usa mais o computador?
   - ☐ Na empresa
   - ☐ Em casa
4. Quando você compra um livro de computação, o que influencia mais a sua decisão?
   - ☐ Prestígio do autor
   - ☐ Recomendações de amigos/colegas
   - ☐ Folhear o livro
   - ☐ Nome da editora
   - ☐ Preço do livro
5. Qual o seu nível de formação em informática?
   - ☐ Iniciante
   - ☐ Intermediário
   - ☐ Avançado

NOSSOS LIVROS ENCONTRAM-SE
EM TODAS AS BOAS LIVRARIAS

DOBRE AQUI

SELO

Editora Campus Ltda.

LIVROS CIENTÍFICOS E TÉCNICOS
CAIXA POSTAL 13007 CEP 20262
RIO DE JANEIRO RJ BRASIL
Endereço Telegráfico: CAMPUSRIO

A CONSTRUÇÃO DE UM COMPILADOR

A grande experiência dos autores, tanto acadêmica como profissional, está na origem deste livro.

Nele, o leitor descobrirá um método essencialmente prático que o levará rapidamente à execução de um projeto completo de compilador.

ISBN 85-7001-334-5