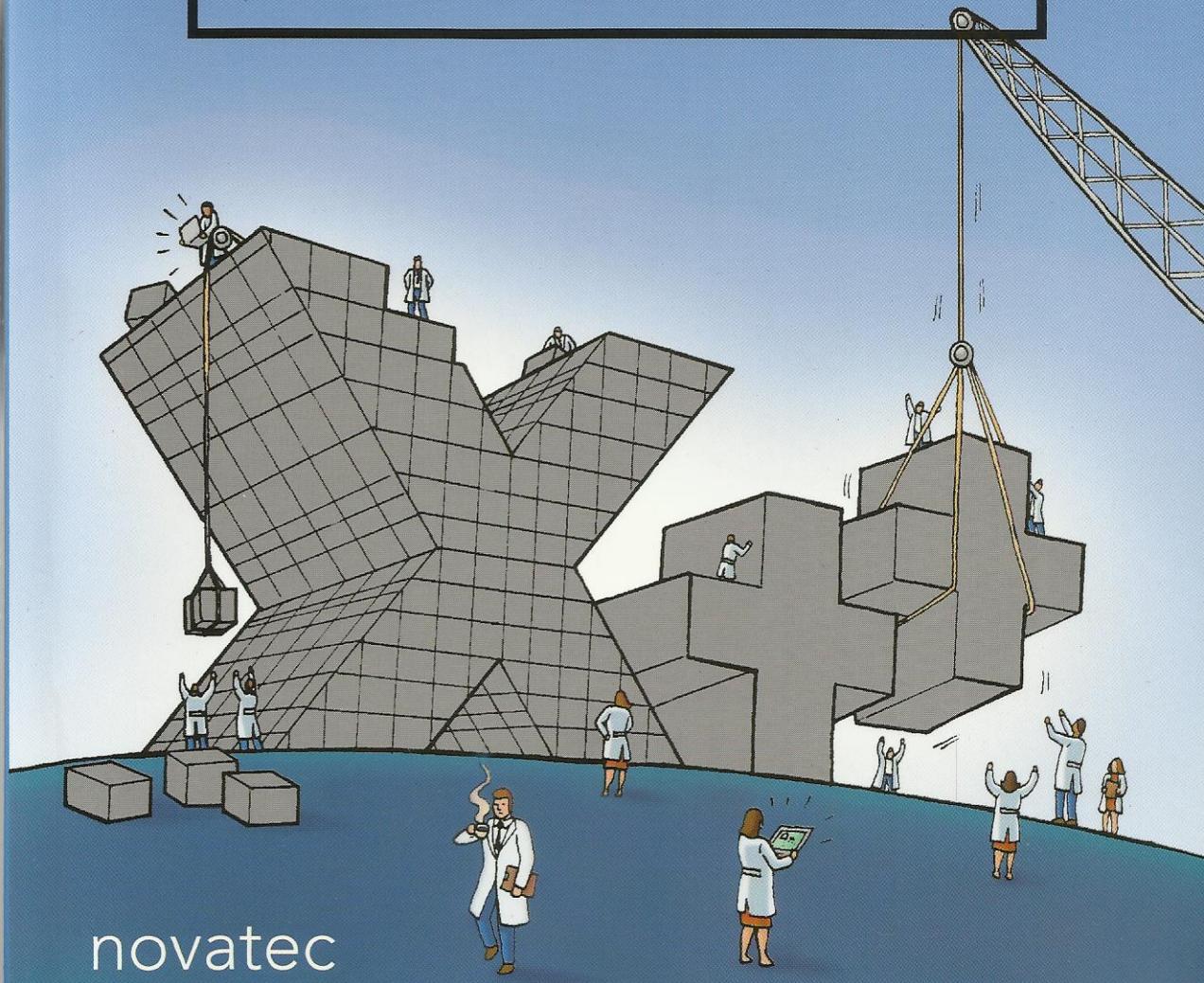


Como Construir um Compilador

Utilizando Ferramentas Java

Márcio Eduardo Delamaro



novatec

Sumário

1	Introdução	1
1.1	O que faz um compilador	1
1.2	Os componentes de um compilador	3
1.2.1	O analisador léxico	3
1.2.2	O analisador sintático	4
1.2.3	O analisador semântico	6
1.2.4	O gerador de código	6
1.3	O JavaCC e o Jasmin	7
2	A Linguagem X^{++}	9
2.1	Alfabetos, palavras e linguagens	9
2.2	Gramática livre de contexto	11
2.3	Forma de Backus-Naur	12
2.4	BNF para a linguagem X^{++}	15
2.5	Grafos sintáticos	24
2.6	Semântica da linguagem X^{++}	33
2.7	Um programa em X^{++}	33
3	Análise Léxica	37
3.1	Autômatos finitos	37
3.2	Expressões regulares	46
3.3	O analisador léxico da linguagem X^{++}	47
3.4	Comentários	54
3.5	Recuperação de erros léxicos	56
3.6	Arquivos-fonte do compilador	59

4 Análise Sintática	61
4.1 Análise sintática descendente recursiva	61
4.2 O analisador sintático da linguagem X^{++}	68
4.3 Arquivos-fonte do compilador	81
5 Tratamento de Erros Sintáticos	85
5.1 O método de ressincronização	85
5.2 Implementação da recuperação de erros	89
5.3 Algumas possíveis melhorias	107
5.4 Arquivos-fonte do compilador	117
6 Geração da Árvore Sintática	123
6.1 O que é a árvore sintática	123
6.2 Implementação da árvore sintática	125
6.3 Arquivos-fonte do compilador	167
7 Exibição da Árvore Sintática	169
7.1 Exibição da árvore sintática	169
7.1.1 Numeração dos nós	174
7.1.2 Exibição dos nós	180
7.2 Outras operações	181
7.3 Arquivos-fonte do compilador	182
8 Tabela de Símbolos	185
8.1 Para que serve	185
8.2 A tabela de símbolos para X^{++}	187
8.3 Implementação da tabela de símbolos	191
9 Análise Semântica – Primeira Parte	199
9.1 Análise semântica em fases	199
9.2 Análise da declaração de classes	202
9.3 Implementação	203
9.4 O programa principal	208
9.5 Arquivos-fonte do compilador	208

10 Análise Semântica – Segunda Parte	209
10.1 Análise da hierarquia de classes	209
10.2 Análise da declaração de variáveis e métodos	212
10.3 O programa principal	218
10.4 Arquivos-fonte do compilador	219
11 Análise Semântica – Parte Final	221
11.1 Checagem de tipos	221
11.2 Análise das declarações	226
11.3 Análise dos comandos	235
11.4 Análise de expressões	248
11.5 Arquivos-fonte do compilador	259
12 Geração de Código	261
12.1 A Máquina Virtual Java	261
12.2 O Assembler para Java	265
12.3 Implementação	267
12.3.1 Geração de código para os comandos	277
12.3.2 Geração de código para as expressões	283
12.3.3 Algumas restrições	288
12.4 O runtime X^{++}	289
12.5 Arquivos-fonte do compilador	291
12.6 Exemplos	292
A Exercícios	297
B JavaCC	299
C Jasmin	305

Prefácio

Um livro sobre compiladores deveria apresentar uma visão geral sobre técnicas de compilação, ferramentas, aplicações etc. Este livro é bem mais modesto, visto que mostra como construir um compilador, utiliza técnica e ferramentas específicas para tal intento.

Mesmo assim, será útil a cursos de graduação que se proponham também a uma abordagem prática, a incentivarem os estudantes a “colocarem as mãos na massa” e construírem seus próprios compiladores. Dessa forma, poderá servir como um guia a ser utilizado em cursos de compiladores, em paralelo a aulas que ensinem a teoria de forma mais ampla.

Também poderá auxiliar aqueles que, embora não sendo “da área” de compiladores, necessitem em algum momento da vida acadêmica ou profissional utilizar técnicas de compilação. Este é o caso do autor, cuja área principal de atuação é a de teste de software. Atuando nessa área, o desenvolvimento de ferramentas de análise de programas é essencial, assim como o domínio, mesmo que de forma básica, de técnicas para a construção de compiladores.

Com esses objetivos em mente, procurou-se manter o mais simples possível o estilo deste livro. Escolheu-se um método de análise sintática e ferramentas para sua implementação fáceis e que possam ser compreendidos com algum esforço mesmo por “principiantes”. É mostrada, de forma detalhada, toda a implementação de um compilador para uma linguagem, embora restrita, com características importantes. Em particular, uma linguagem orientada a objetos. Ao final do “projeto” terá o leitor a satisfação de desenvolver seu próprio compilador, que poderá ser utilizado para implementar programas reais, que executem em um ambiente real, que é a Máquina Virtual Java.

Pré-requisitos

Espera-se do leitor conhecimentos prévios sobre diversos assuntos. O primeiro tópico seria para que serve e para que desejamos construir um compilador.

O segundo item necessário à utilização deste livro é o domínio de alguma linguagem de programação, de preferência Java, pois, além de ser utilizada como linguagem para a implementação do compilador, também o é como referência e comparação para a linguagem-alvo do compilador. Conhecimentos básicos de linguagens formais

e autômatos finitos, embora não indispensáveis, certamente ajudarão na compreensão do texto.

Além de, é claro, muita paciência para seguir os intermináveis trechos da implementação incluídos no texto...

Agradecimentos

Agradeço a todos os que colaboraram de alguma forma na elaboração deste livro. Em particular, a meus alunos de compiladores Lú, Beto, Gustavo(s), Dani(s), Gisele, Andréia, Paulinha, Neves, Juliano, Juliana.

Este talvez seja o primeiro e último livro que escrevi. Assim, esta é uma oportunidade única de expressar minha gratidão ao prof. José Carlos Maldonado pela amizade, orientação e oportunidade de trabalharmos juntos. Se em cada uma das conquistas de minha carreira acadêmica posso sentir a influência do amigo Maldonado, neste livro não seria diferente. Por isso, muito obrigado.

Download dos programas

Todos os códigos-fonte dos programas que aparecem neste livro podem ser encontrados em <http://www.novateceditora.com.br/downloads.php>.

Capítulo 1

Introdução

O leitor que se interessa por construção de compiladores provavelmente já saiba o que é e para que serve um compilador. Por isso, vamos apenas brevemente discutir quais são as funções desempenhadas por um compilador. Também abordaremos como se estrutura internamente um compilador e comentaremos sobre a ferramenta JavaCC, um gerador de compiladores que será utilizado neste texto.

1.1 O que faz um compilador

Há uma frase famosa que diz: “Um programa de computador é uma seqüência de 0s e 1s armazenada na sua memória”. Essa seqüência de 0s e 1s pode representar dados tais como números inteiros, strings, registros etc., ou pode ser uma instrução, que indica como o computador deve se comportar. A memória é dividida em palavras, cada qual possuindo um endereço (por exemplo, 0, 1, 2, ...) que a identifica de maneira única.

Embora seja armazenado na memória, todo programa é executado na CPU do computador. A CPU possui registradores que guardam temporariamente dados sobre os quais se desejam realizar operações. Por exemplo, para somar um número armazenado na memória no endereço 100 com o número armazenado no endereço 101 e colocar o resultado no endereço 102, a CPU deveria executar operações como:

- copiar o conteúdo da posição de memória 100 para o registrador A;
- copiar o conteúdo da posição de memória 101 para o registrador B;
- somar o conteúdo de B em A;
- copiar o conteúdo de A para a posição de memória 102.

Com o aumento da complexidade dos programas de computador, tornou-se necessário desenvolvê-los num nível de abstração um pouco mais elevado, menos dependente das instruções de uma determinada máquina. Foram criadas, assim, as linguagens de

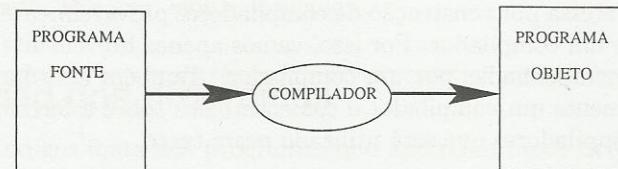
2 COMO CONSTRUIR UM COMPILADOR

alto nível, que substituem as instruções dos computadores por comandos cujas utilização e compreensão são mais fáceis. Por exemplo, o comando

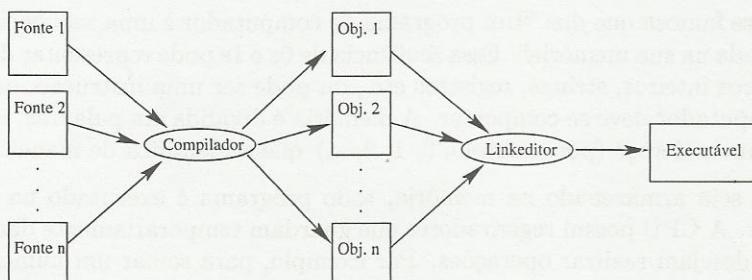
$$c = a + b;$$

indica que queremos somar o conteúdo de uma posição de memória que foi chamada pelo programador de *a* com o conteúdo da posição de memória chamada de *b* e colocar o resultado na posição de memória chamada de *c*.

Se as linguagens de programação mudaram radicalmente, os computadores num certo sentido permanecem inalterados e só conseguem “entender” instruções simples e que estejam no seu repertório. Por isso, programas escritos em linguagens de alto nível precisam ser convertidos nessas instruções antes de serem executados. Essa é a principal atribuição de um compilador: transformar um programa escrito numa linguagem de alto nível – que chamamos de linguagem fonte – em instruções executáveis por uma determinada máquina – que chamamos de código objeto.



(a)



(b)

Figura 1.1 – Processos de (a) compilação e (b) linkedição.

Isso é resumido na figura 1.1(a). Em geral, o compilador recebe como entrada um arquivo contendo o programa na linguagem fonte (o programa fonte) e produz como saída um outro arquivo com o código objeto (o programa objeto). Muitas vezes esse programa objeto é apenas parte do programa como um todo. A maioria das linguagens de programação e seus respectivos compiladores permitem que um programa seja dividido em diversos arquivos-fonte e respectivos objetos. Nesse caso é necessário mais um passo para preparar o programa para execução (Figura 1.1(b)). Um linkeditor junta todos os programas objetos, formando o programa executável, que está pronto para ser carregado na memória do computador e executado.

1.2 Os componentes de um compilador

Tradicionalmente, a construção de um compilador é dividida em partes, cada uma com uma função específica. Em geral, podemos identificar em um compilador:

- o analisador léxico;
- o analisador sintático;
- o analisador semântico;
- o gerador de código.

1.2.1 O analisador léxico

O analisador léxico (AL) encarrega-se de separar no programa fonte cada símbolo que tenha algum significado para a linguagem ou de avisar quando um símbolo que não faz parte da linguagem é encontrado. Por exemplo, vamos supor um programa fonte com a seguinte seqüência de símbolos:

```
123 x1 ; y2 true begin
```

Ao analisar tal entrada, o analisador léxico deveria identificar a ocorrência de 6 símbolos. Além disso, o analisador deve categorizar cada um deles, indicando de que tipo é aquele símbolo. No caso anterior, teríamos:

- 123 – constante inteira;
- x1 – nome de variável ou procedimento;
- ; – símbolo especial “ponto-e-vírgula”;
- y2 – nome de variável ou procedimento;
- true – constante booleana;
- begin – palavra reservada.

Obviamente essa classificação depende de qual linguagem fonte está sendo analisada. A classificação pode ser real para a linguagem Pascal, mas certamente não o é para Java, onde a palavra “begin” não tem nenhum significado especial e pode ser utilizada, por exemplo, como nome de variável. Assim, para determinar quais são os símbolos que devem ser reconhecidos pelo AL, devemos recorrer à descrição da linguagem.

O AL deve também avisar quando um símbolo inválido aparece no programa fonte. Por exemplo, se o símbolo ‘@’ aparecer num programa sendo analisado por um compilador Java, o AL deve avisar que um símbolo inválido foi encontrado, pois ‘@’ não faz parte de nenhuma categoria de símbolos aceita pelo AL.

O funcionamento do AL parece bastante simples. Tudo o que ele deve fazer é “quebrar” o programa fonte em símbolos e verificar a que categoria eles pertencem. A verdade não é bem essa. Existem diversos complicadores para essa tarefa. O primeiro deles é que a entrada nem sempre está tão bem arrumada como a que mostramos. Por exemplo, como seria analisada a seguinte entrada?

```
123x1begin{end
```

Ela deveria ser dividida (em Pascal):

123 – constante inteira;

x1begin – nome de variável ou procedimento;

{ – símbolo especial “abre chave”;

end – palavra reservada.

Outro fator que dificulta a construção do AL é que este pode encontrar-se em “estados” diferentes, de acordo com os caracteres que são encontrados no programa fonte. Por exemplo, dissemos há pouco que a presença de um ‘@’ no programa fonte causa um erro denominado erro léxico. Mas isso não é verdade sempre. Se aparecer no programa fonte

“Aqui @ temos uma arroba”

o AL não deve apontar um erro léxico em “Aqui @ temos uma arroba”, e, sim, reconhecer uma constante do tipo string. Algo semelhante acontece com os comentários. Ao encontrar, por exemplo, o símbolo (*) o AL entra num estado em que deve ignorar tudo que apareça até o próximo (*), até mesmo símbolos normalmente não válidos na linguagem.

No capítulo 3 veremos como identificar e classificar os símbolos de uma linguagem e como construir um AL.

1.2.2 O analisador sintático

O analisador sintático (AS) é o “coração” do compilador, responsável por verificar se a sequência de símbolos contida no programa fonte forma um programa válido ou não.

Note o leitor que o AL se encarrega de identificar os símbolos que aparecem no programa fonte, mas não se preocupa em verificar se a ordem em que eles aparecem é válida ou não. Essa é uma das atribuições do AS. Por exemplo, considere o seguinte trecho de programa:

```
if (a - 10 > b * 2)
    a = b;
```

O AS deve ser capaz de analisar esse programa e reconhecê-lo como válido. Para isso, o AS precisa saber que após a palavra reservada `if` deve vir um “(”, uma expressão e um “)”. Depois disso, deve vir um comando qualquer, por exemplo, uma atribuição como `a = b;`.

Para isso, o AS é construído sobre uma gramática que descreve a linguagem fonte. Essa gramática é composta de uma série de regras que descrevem quais são as construções válidas da linguagem. O AS deve aceitar aqueles programas que seguem essas regras e rejeitar – indicando a ocorrência de um erro sintático – aqueles que as violam. No capítulo 2 descreveremos por meio de uma gramática a linguagem X^{++} , que será utilizada neste texto para a construção de um compilador. Também comentaremos mais sobre a utilização de gramáticas para descrever linguagens.

Além de aceitar os programas sintaticamente corretos e rejeitar os incorretos, o AS desempenha ainda outra importante função que é a construção da árvore sintática do programa fonte. Uma árvore sintática é uma estrutura em forma de árvore que descreve as construções da linguagem reconhecidas pelo AS no programa fonte. Se o programa fonte possui um comando `if` como aquele visto há pouco, sua árvore sintática deve espelhar esse fato e descrever como esse comando é formado. Poderíamos ter para aquele comando a árvore sintática da figura 1.2, que indica que o comando `if` é formado por uma expressão e um comando de atribuição. A expressão, por sua vez, é também representada por uma subárvore cuja raiz é uma operação de comparação `>`, que tem como subárvore uma subexpressão de subtração e uma subexpressão de multiplicação.

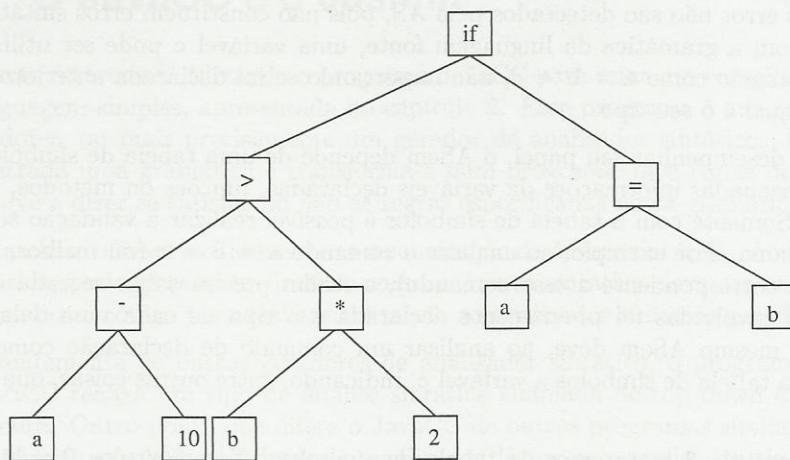


Figura 1.2 – Exemplo de árvore sintática.

Nos capítulos 4 e 5 estaremos tratando com detalhes do AS. Nos capítulos 6 e 7 trataremos de árvore sintática.

1.2.3 O analisador semântico

O analisador semântico (ASem) irá verificar se os aspectos semânticos do programa estão corretos, ou seja, se não existem incoerências quanto ao significado das construções utilizadas pelo programador.

O analisador semântico não utiliza mais o programa fonte para fazer tal verificação. Em vez disso, utiliza a árvore sintática como representação do programa. O analisador semântico deve procurar por incoerências como:

- tipos de operandos incompatíveis com operadores. Se tivermos o comando `a = b * c` e a variável `c` foi declarada do tipo `string`, então o analisador semântico deve apontar um erro semântico, pois esse tipo de operando não é compatível com o operador `*`;
- variáveis não declaradas;
- redeclaração de variáveis;
- chamadas de funções ou métodos com o número incorreto de parâmetros;
- comandos colocados fora de contexto. Por exemplo, a utilização de um comando `continue` fora de um comando de laço deve ser apontada como um erro semântico.

Esses erros não são detectados pelo AS, pois não constituem erros sintáticos. De acordo com a gramática da linguagem fonte, uma variável `c` pode ser utilizada em uma expressão como `a = b * c`, não importando se foi declarada anteriormente ou não, ou qual é o seu tipo.

Para desempenhar seu papel, o ASem depende de uma tabela de símbolos. Nela são armazenadas informações de variáveis declaradas, funções ou métodos, tipos ou classes. Somente com a tabela de símbolos é possível realizar a validação semântica do programa. Por exemplo, ao analisar o comando `a = b * c` (ou melhor, a árvore sintática correspondente a esse comando), o ASem precisa saber se cada uma das variáveis envolvidas foi previamente declarada e o tipo de cada uma delas. Para tanto, o mesmo ASem deve, ao analisar um comando de declaração como `int c`, incluir na tabela de símbolos a variável `c`, indicando, entre outras coisas, que seu tipo é `int`.

No capítulo 8 trataremos de tabela de símbolos. Nos capítulos 9 a 11, iremos estudar como funciona um analisador semântico.

1.2.4 O gerador de código

Uma vez verificado que não existem erros sintáticos ou semânticos, o compilador pode realizar sua tarefa, que é a criação do programa objeto. Em geral, o programa objeto é armazenado num arquivo que pode ser posteriormente linkeditado com outros programas objetos ou simplesmente “carregado” na memória do computador e executado pelo sistema operacional.

O programa objeto reflete, mediante instruções de baixo nível, os comandos do programa fonte. Como cada máquina ou cada plataforma possui um conjunto diferente de instruções e de meios de acesso ao sistema operacional, em geral é necessário que exista um gerador de código distinto para cada plataforma.

A otimização do código também pode fazer parte do processo de geração de código. Nela são aplicadas diversas técnicas para otimizar algumas características do programa objeto, como, por exemplo, seu tamanho ou sua velocidade.

A geração de código será tratada no capítulo 12. Em vez de nos concentrarmos em uma arquitetura particular, iremos utilizar a Máquina Virtual Java (JVM) como máquina-alvo do nosso compilador. A JVM é um programa capaz de executar (na verdade, interpretar) arquivos “.class” que são gerados por compiladores Java. Assim como qualquer outra máquina, possui um conjunto de instruções que serão utilizadas em nosso compilador para representar os comandos de alto nível da nossa linguagem fonte. O nosso compilador não gera diretamente um arquivo “.class”, mas, sim, um arquivo em que as instruções da JVM podem ser visualizadas, como numa linguagem de montagem. Tal arquivo pode ser posteriormente processado e transformado num “.class”, utilizando-se o programa Jasmin, descrito a seguir. Na verdade, essa abordagem de gerar-se um código intermediário utilizando uma linguagem de montagem, que depois é transformado no programa objeto, é também utilizada em compiladores reais.

1.3 O JavaCC e o Jasmin

Neste texto estaremos utilizando o programa JavaCC para criar um compilador para uma linguagem simples, apresentada no capítulo 2. Esse programa é um gerador de compiladores, ou mais precisamente um gerador de analisador sintático. Ele toma como entrada uma gramática e transforma-a num programa Java capaz de analisar um arquivo e dizer se satisfaz ou não as regras especificadas nessa gramática.

Ele também oferece facilidades para a construção da árvore sintática. Ao descrever a gramática, pode-se também indicar como a árvore sintática deve ser construída, incorporando-se código para realizar tal tarefa ao analisador sintático gerado, .

Diferentemente de outros geradores de analisador sintático, o programa gerado pelo JavaCC realiza um tipo de análise sintática chamada de top down ou análise descendente. Outro ponto que difere o JavaCC de outros programas similares é que aquele permite que sejam definidos o analisador léxico e o analisador sintático de uma só vez. Em geral, outros programas são utilizados aos pares (Lex e Yacc; Flex e Byson), um para gerar o analisador léxico e outro, para o analisador sintático.

O JavaCC é um produto de propriedade da Sun Microsystems e liberado sob a licença Berkeley Software Distribution (BSD). Pode ser obtido na Internet no endereço <https://javacc.dev.java.net/>. Neste texto trataremos dos aspectos do JavaCC que interessam à construção de nosso pequeno compilador. A cada etapa da construção do compilador serão dados mais detalhes do funcionamento do JavaCC. Não temos, porém, a intenção de abordar todos os detalhes de uso deste software. Para tal, o leitor deve verificar a documentação que acompanha o software e as indicações de outras fontes de informação que se encontram no Apêndice B.

O programa Jasmin é uma interface de montagem para Java (Java ASseMbler INterface), que toma como entrada um arquivo ASCII com instruções JVM e produz um arquivo executável JVM (arquivo ".class"). Esse programa pode também ser obtido na Internet, no endereço <http://www.cat.nyu.edu/~meyer/jasmin>. Ele será utilizado caso o leitor queira produzir um arquivo executável a partir do código gerado pelo compilador que iremos construir neste texto. Em outras palavras, nosso compilador gera um programa JVM que está no formato ASCII e que pode ser facilmente visualizado. Assim podemos verificar se o código gerado está correto ou não. Porém, para que possamos executar esse código, precisamos executar mais uma tarefa, que é transformá-lo num arquivo executável Java, utilizando o programa Jasmin. Jasmin é parte do material utilizado no livro *Java Virtual Machine* de Troy Drowsing e Jon Meyer, publicado pela O'Reilly e que descreve a Máquina Virtual Java. O Apêndice C apresenta também uma pequena introdução ao uso do Jasmin.

Capítulo 2

A Linguagem X^{++}

Antes de continuarmos com a construção do nosso compilador, estudando como funciona cada uma de suas partes, precisamos definir a linguagem-alvo dele. É o que faremos neste capítulo.

Poderíamos utilizar linguagens de programação como Java ou C++. Contudo, essas linguagens são complexas demais para os objetivos deste livro. Em vez disso, iremos definir uma linguagem que seja mais simples, mas que possa dar ao leitor uma visão completa de como um compilador funciona. A linguagem que utilizaremos – e chamaremos de X^{++} – é uma linguagem orientada a objetos semelhante à Java, porém ligeiramente mais simples.

Para definir a sintaxe da linguagem, iremos antes relembrar alguns conceitos sobre linguagens formais e como podemos defini-las. Iremos tratar de alfabetos, linguagens, gramáticas e outras formas comumente usadas para definir-se a sintaxe de linguagens de programação.

2.1 Alfabetos, palavras e linguagens

Inicialmente, definimos alfabeto como um conjunto finito e não vazio de símbolos. Assim, por exemplo, os seguintes conjuntos são alfabetos:

- $\{0,1\}$;
- $\{0,1,2,3,4,5,6,7,8,9,0\}$;
- $\{a,b,c,\dots,z\}$;
- $\{a, b, ab, abc\}$.

Cada elemento de um alfabeto é chamado de uma letra. Note-se que neste contexto o termo letra tem um significado diverso daquele que estamos acostumados a usar e que poderia ter como sinônimo algo como “caractere”. Assim, o quarto alfabeto possui 4 letras: *a*, *b*, *ab* e *abc* (essas duas últimas com três “caracteres”).

Uma palavra ou cadeia sobre um alfabeto Σ é uma tupla ordenada de letras de Σ . Por exemplo:

- $\langle 0, 1, 0, 1, 1, 0 \rangle$ é uma palavra sobre $\{0,1\}$;
- $\langle 2, 1, 0, 8 \rangle$ é uma palavra sobre $\{0,1,2,3,4,5,6,7,8,9\}$;
- $\langle c, o, m, p, i, l, e, r \rangle$ é uma palavra sobre $\{a,b,\dots, z\}$;
- $\langle a, ab, b, abc \rangle$ é uma palavra sobre $\{a, b, ab, abc\}$.

Em geral, podemos representar uma palavra apenas aglutinando, na ordem correta, as letras que a compõem. Por exemplo, podemos escrever

- $\langle 0, 1, 0, 1, 1, 0 \rangle$ como *010110*;
- $\langle 2, 1, 0, 8 \rangle$ como *2108*;
- $\langle c, o, m, p, i, l, e, r \rangle$ como *compiler*.

Porém, $\langle a, ab, b, abc \rangle$ não pode ser representada simplesmente por *aabbabc*, pois essa representação poderia indicar outras palavras além daquela que desejamos representar, como, por exemplo, $\langle a, a, b, abc \rangle$. Nesse caso, podemos utilizar espaços entre as letras para indicar como “separar” as letras da palavra. A palavra $\langle a, ab, b, abc \rangle$ seria, então, representada como *a ab b abc* e a palavra $\langle a, a, b, abc \rangle$, como *a a b abc*. Note-se, porém, que os espaços não fazem parte da palavra.

Define-se o tamanho de uma palavra x (denotado por $|x|$) como o número de letras de x . Nos exemplos anteriores, teríamos:

- $|010110| = 6$;
- $|2108| = 4$;
- $|compiler| = 8$;
- $|a ab b abc| = 4$.

Sobre qualquer alfabeto Σ , define-se uma única palavra de tamanho 0 que denotamos por λ . Definem-se, também, os conjuntos:

- $\Sigma^k = \{\text{palavras } x \text{ sobre } \Sigma \mid |x| = k\}$;
- $\Sigma^* = \bigcup_{k=0}^{\infty} \Sigma^k = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \dots$;
- $\Sigma^+ = \Sigma^* - \{\lambda\}$.

Chegamos, então, à definição do que é uma linguagem: dado o alfabeto Σ , uma linguagem L sobre Σ é um subconjunto qualquer de Σ^* . Vejamos os exemplos a seguir.

- $\{0, 1, 00, 01, 10, 11\}$ é uma linguagem sobre $\{0,1\}$ que contém seis palavras;
- $\{x \in \{0,1,2,3,4,5,6,7,8,9\}^* \mid x \text{ representa um número decimal ímpar}\}$ é uma linguagem sobre $\{0,1,2,3,4,5,6,7,8,9\}$ com um número infinito de palavras;
- o conjunto de todos os programas Java válidos (sintaticamente corretos) é uma linguagem sobre um alfabeto Σ composto de
 - palavras reservadas como **for**, **while**, **if** etc.
 - símbolos especiais como $\{ \}$, $*$, $,$, $+$ etc.
 - nomes de variáveis, métodos, classes etc.

Para definirmos formalmente uma linguagem de programação, devemos, então, definir qual o alfabeto sobre o qual essa linguagem é formada e definir quais são as palavras válidas para essa linguagem. Nas próximas seções veremos algumas maneiras de fazer isso.

2.2 Gramática livre de contexto

Em geral, uma linguagem de programação pertence a uma classe de linguagens chamadas de linguagens livres de contexto. Uma das maneiras de se definirem tais linguagens é por meio das gramáticas livres de contexto. Uma gramática livre de contexto (GLC) é uma quádrupla $\langle \Omega, \Sigma, S, P \rangle$, onde

Σ – é o alfabeto sobre o qual a linguagem é definida;

Ω – é um conjunto não vazio de símbolos não terminais;

P – é um conjunto de produções da forma $A \rightarrow \alpha$, onde $A \in \Omega$ e $\alpha \in (\Sigma \cup \Omega)^*$;

S – é o símbolo inicial da gramática, $S \in \Omega$.

O conjunto Σ de uma gramática é o alfabeto sobre o qual as palavras da linguagem desejada são formadas. No contexto de GLCs, costuma-se chamar os elementos de Σ de símbolos terminais. O conjunto Ω é disjunto de Σ . Seus elementos não entram na formação das cadeias da linguagem. São utilizados como símbolos auxiliares para definirem-se as produções da gramática.

As produções P são regras de substituição que têm do lado esquerdo do \rightarrow um símbolo não terminal e do lado direito, uma cadeia formada por símbolos terminais e símbolos não terminais ou λ . A partir do símbolo inicial S , essas regras são aplicadas substituindo-se o não-terminal que aparece do lado esquerdo de uma produção pela palavra que está do lado direito desta. Se a palavra obtida possui somente símbolos terminais, então essa palavra pertence à linguagem definida pela gramática. Se a cadeia obtida contém ainda símbolos não terminais, então nova(s) substituição(ões) deve(m) ser feita(s) até que se chegue a uma cadeia que tenha apenas símbolos terminais.

Vamos tomar como exemplo a gramática $G_1 = \langle \{S, A, B\}, \{a, b, c\}, S, P \rangle$, onde P é:

- (1) $S \rightarrow AB$
- (2) $A \rightarrow aAb$
- (3) $A \rightarrow \lambda$
- (4) $B \rightarrow cB$
- (5) $B \rightarrow \lambda$

Assim, a partir do símbolo inicial S podemos aplicar uma seqüência de produções de P até que uma cadeia x que possua apenas símbolos terminais seja alcançada. Então x pertence à linguagem definida por G_1 , o que podemos denotar como: $x \in L(G_1)$. Por exemplo:

Iniciamos com S	aplicamos (1)	obtemos AB
temos AB	aplicamos (3)	obtemos B
temos B	aplicamos (5)	obtemos $\lambda \in L(G_1)$

Iniciamos com S	aplicamos (1)	obtemos AB
temos AB	aplicamos (2)	obtemos $aAbB$
temos $aAbB$	aplicamos (2)	obtemos $aaAbbB$
temos $aaAbbB$	aplicamos (3)	obtemos $aabbB$
temos $aabbB$	aplicamos (4)	obtemos $aabbcB$
temos $aabbcB$	aplicamos (4)	obtemos $aabbccB$
temos $aabbccB$	aplicamos (5)	obtemos $aabbcc \in L(G_1)$

Pode-se facilmente notar que $|L(G_1)| = \infty$, pois dependendo do número de vezes que são aplicadas as produções (2) e (4), podem ser criadas palavras de tamanho tão grande quanto se deseje.

Cada uma das palavras intermediárias geradas até se chegar à palavra que só contém terminais é chamada de uma forma sentencial. Se uma cadeia β pode ser obtida a partir da forma sentencial α mediante a aplicação de uma única produção, dizemos que α deriva diretamente β , ou que β é derivada diretamente de α , o que denotamos por $\alpha \Rightarrow \beta$. Se β pode ser obtida mediante a aplicação de um número finito de produções, dizemos que α deriva β e denotamos $\alpha \xrightarrow{*} \beta$.

Como a linguagem $L(G)$, definida pela gramática livre de contexto $G = \langle \Omega, \Sigma, S, P \rangle$, é o conjunto de todas as palavras formadas apenas por elementos de Σ que podem ser derivadas a partir do símbolo inicial S , isso equivale dizer que $L(G) = \{x \in \Sigma^* \mid S \xrightarrow{*} x\}$.

2.3 Forma de Backus-Naur

A Forma de Backus-Naur (BNF) é uma outra maneira de se definir linguagens livres de contexto. Ela é semelhante a uma gramática livre de contexto, mas permite que o lado direito das produções possua alguns operadores. Esses operadores serão apresentados a seguir.

Seleção

$(\alpha \mid \beta)$ – Um dos elementos entre parênteses (α ou β) pode ser utilizado na aplicação da produção.

Por exemplo, se tivéssemos a produção:

$$S \rightarrow a(b \mid c \mid d)e$$

poderíamos ter as derivações:

$$\begin{aligned} S &\Rightarrow abe \\ S &\Rightarrow ace \\ S &\Rightarrow ade \end{aligned}$$

o que significa que a linguagem gerada é $\{abe, ace, ade\}$.

Uma GLC equivalente a essa BNF seria

$$\begin{aligned} S &\rightarrow abe \\ S &\rightarrow ace \\ S &\rightarrow ade \end{aligned}$$

Vejamos outro exemplo:

$$S \rightarrow (c(aSa \mid bSb)c \mid \lambda)$$

gera cadeias do tipo:

$$S \Rightarrow caSac \Rightarrow cacaSacac \Rightarrow cacacbSbcacac \Rightarrow cacacbbcacac$$

Nesse caso, a GLC equivalente seria

$$\begin{aligned} S &\rightarrow \lambda \\ S &\rightarrow caSac \\ S &\rightarrow cbSbc \end{aligned}$$

Opcional

$[\alpha]$ – O que estiver entre os colchetes pode ser utilizado ou não na aplicação da produção.

Por exemplo, se tivéssemos a produção:

$$S \rightarrow a[bcd]e$$

poderíamos ter as derivações:

$$\begin{aligned} S &\Rightarrow ae \\ S &\Rightarrow abcde \end{aligned}$$

o que significa que a linguagem gerada é $\{ae, abcde\}$.

Vejamos outro exemplo:

$$\begin{array}{lcl} S & \rightarrow & a[(A \mid B)c]d \\ A & \rightarrow & a[a] \\ B & \rightarrow & [b] \end{array}$$

gera as cadeias:

$$\begin{aligned} S &\Rightarrow ad \\ S &\Rightarrow aAcd \Rightarrow aacd \\ S &\Rightarrow aAcd \Rightarrow aaacd \\ S &\Rightarrow aBcd \Rightarrow abcd \\ S &\Rightarrow aBcd \Rightarrow acd \end{aligned}$$

ou seja, gera a linguagem $\{ad, aacd, aaacd, abcd, acd\}$. Uma GLC equivalente a essa BNF seria:

$$\begin{array}{lcl} S & \rightarrow & ad \\ S & \rightarrow & aAcd \\ S & \rightarrow & aBcd \\ A & \rightarrow & aa \\ A & \rightarrow & a \\ B & \rightarrow & b \\ B & \rightarrow & \lambda \end{array}$$

Note ainda que uma produção do tipo $\alpha \rightarrow \beta[\gamma]\delta$ é sempre equivalente à produção $\alpha \rightarrow \beta(\gamma \mid \lambda)\delta$.

Repetição 0 ou mais vezes

$(\alpha)^*$ – O que estiver entre parêntese pode ser usado um número qualquer de vezes na aplicação da produção e pode também não ser usado (repetido nenhuma vez).

Por exemplo, se tivéssemos a produção:

$$S \rightarrow a(b)^*c$$

teríamos como resultado a linguagem $\{ac, abc, abbc, abbdc, \dots\}$.

Como qualquer outra BNF, podemos encontrar uma GLC correspondente, como, por exemplo:

$$\begin{array}{lcl} S & \rightarrow & aZc \\ Z & \rightarrow & bZ \\ Z & \rightarrow & \lambda \end{array}$$

Como outro exemplo, podemos ter a BNF:

$$S \rightarrow a(b \mid c)^*d$$

que gera todas as cadeias que iniciam com a , terminam com d e têm entre estes o λ ou qualquer cadeia formada por b e c .

Repetição 1 ou mais vezes

$(\alpha)^+$ – O que estiver entre parêntese pode ser usado uma ou mais vezes na aplicação da produção. A produção $\alpha \rightarrow \beta(\gamma)^+\delta$ é equivalente a $\alpha \rightarrow \beta\gamma(\gamma)^*\delta$.

A seguir, utilizaremos a BNF para definir a nossa linguagem X^{++} . Será efetuada uma descrição de quais são as estruturas da linguagem representadas em cada uma das produções.

2.4 BNF para a linguagem X^{++}

Nossa linguagem é construída sobre uma alfabeto que possui símbolos que podem ser confundidos com os metassímbolos utilizados como operadores na notação BNF, como, por exemplo, [e]. Por isso, vamos adotar a seguinte convenção:

- $\langle symbol \rangle$ representa o não-terminal $symbol$;
- símbolos terminais são representados com “symbol” ;
- metassímbolos da notação BNF são representados grafados com [].

Iniciaremos a definição pelo símbolo inicial da BNF, que representa a estrutura de um programa escrito em X^{++} .

$$\langle program \rangle \rightarrow [\langle classlist \rangle]$$

Esse não-terminal indica que um programa em X^{++} é composto de uma lista de classes ou se trata de uma cadeia vazia. Assim, o nosso compilador, ao tentar compilar um programa armazenado num arquivo que esteja vazio, vai aceitá-lo como um programa válido, como ocorre com a maioria dos compiladores.

$$\langle classlist \rangle \rightarrow (\langle classdecl \rangle)^+$$

ou

$$\langle classlist \rangle \rightarrow \langle classdecl \rangle [\langle classlist \rangle]$$

O não-terminal $classlist$ representa uma repetição de declarações de classes, como num programa em Java, que basicamente é composto de uma seqüência de declarações de classes como:

```
class a {....}
class b {....}
...
class c {....}
```

Foram utilizados dois estilos diferentes para definir-se *classlist*, mas ambos geram as mesmas cadeias. O primeiro estilo utiliza o operador de repetição $()^+$ e o segundo, o próprio não-terminal *classlist* recursivamente para obter-se essa repetição. A nossa BNF irá utilizar ora um estilo, ora outro, para que o leitor possa avaliar quais são as implicações, quando formos implementar o analisador sintático, de se utilizar um ou outro estilo.

Continuando com a próxima produção,

$$\langle \text{classdecl} \rangle \rightarrow \text{"class" "ident" ["extends" "ident"] } \langle \text{classbody} \rangle$$

Esse não-terminal *classdecl* representa a declaração de uma classe. Ele indica que tal estrutura se inicia com a palavra reservada¹ *class* que vem seguida por um identificador (outro terminal) que irá dar nome à classe sendo declarada. Depois do nome da classe, pode ou não aparecer a palavra *extends* e o nome da superclasse da qual a classe sendo declarada descende. Seriam cadeias do tipo

~~class a ...~~

ou

~~class a extends b ...~~

O não-terminal *classbody* representa o corpo da declaração da classe:

$$\langle \text{classbody} \rangle \rightarrow \{ [\langle \text{classlist} \rangle] (\langle \text{vardecl} \rangle ;)^* (\langle \text{constructdecl} \rangle)^* (\langle \text{methoddecl} \rangle)^* \}$$

Ele se inicia com { que (opcionalmente) vem seguido por um não-terminal *classlist*, declarado anteriormente. Isso significa que nossa linguagem permite a declaração de classes aninhadas, como acontece na linguagem Java. Depois, seguem-se as declarações de variáveis, construtores e métodos, mediante a repetição (possivelmente nenhuma vez) dos não-terminais *vardecl*, *constructdecl* e *methoddecl*, terminando com um }. Note que nossa linguagem exige que as declarações sejam feitas exatamente nessa ordem. Ou seja, classes aninhadas, depois variáveis da classe, depois os construtores e, finalmente, os métodos.

Uma declaração de variáveis, representada pelo não-terminal *vardecl*, é semelhante à declaração de variáveis em Java. Por exemplo:

```
int a; ou
string a,b; ou
mytype a[], b[] [];
```

Temos então:

$$\langle \text{vardecl} \rangle \rightarrow (\text{"int" | "string" | "ident" }) \text{"ident" ("[" "]")^*} (";" \text{"ident" ("[" "]")^*})^*$$

¹ Costumam-se chamar esses símbolos como *class*, *for*, *while* de palavra reservada da linguagem. Porém, o leitor deve lembrar que cada um desses símbolos é, formalmente falando, uma letra do alfabeto de entrada.

Essa produção gera cadeias que começam com o tipo da variável a ser declarada, que pode ser *int*, *string*, ou um identificador que é o nome de uma classe declarada no próprio programa. Em seguida, vêm os nomes das variáveis sendo declaradas, que podem ser seguidos ou não por colchetes que indicam a dimensão de cada variável. Note que o “;” no final da lista de variáveis não foi incluído nesta produção, mas, sim, na produção do não-terminal *classbody*, após cada aparição de um *vardecl*.

De maneira semelhante, temos as declarações de construtores e métodos:

$$\begin{aligned} \langle\text{constructdecl}\rangle &\rightarrow \text{“constructor” } \langle\text{methodbody}\rangle \\ \langle\text{methoddecl}\rangle &\rightarrow (\text{“int”} \mid \text{“string”} \mid \text{“ident”}) ([“” “”])^* \\ &\quad \text{“ident” } \langle\text{methodbody}\rangle \end{aligned}$$

A declaração de um construtor começa com a palavra *constructor*. Um construtor não tem tipo de retorno ou um nome. Assim, depois de *constructor*, inicia-se o corpo do construtor, representado pelo não-terminal *methodbody*. Já a declaração de um método inicia-se com o tipo de retorno do método, que pode ou não ter várias dimensões, seguido pelo nome do método e pelo seu corpo.

O corpo de um método é dividido em duas partes, conforme mostra a seguinte produção:

$$\langle\text{methodbody}\rangle \rightarrow (“\langle\text{paramlist}\rangle”) \langle\text{statement}\rangle$$

A primeira parte é a lista de parâmetros formais do método, representada pelo não-terminal *paramlist*. A segunda parte são os comandos que compõem o método.

$$\langle\text{paramlist}\rangle \rightarrow [(\text{“int”} \mid \text{“string”} \mid \text{“ident”}) \text{“ident”} ([“” “”])^* \\ (\text{“,”} (\text{“int”} \mid \text{“string”} \mid \text{“ident”}) \text{“ident”} \\ ([“” “”])^*]$$

Essa lista de parâmetros, que é opcional, quando presente é formada por seqüências de declarações que têm o tipo do parâmetro e o nome da variável associada a ele, separadas por vírgulas. Por exemplo, as seguintes cadeias podem ser geradas por *paramlist*:

```
int a ou
int a, string b ou
string b[][], int a, MyType c
```

A segunda parte do corpo do método é um *statement*. Isso significa que essa parte é composta de um único comando, o que pode parecer estranho. Na verdade, como veremos na definição do não-terminal *statement* a seguir, uma construção do tipo

```
{
  “comando 1”
  “comando 2”
  ...
  “comando n”
}
```

é um comando composto e pode ser gerado a partir do não-terminal *statement*.

$$\begin{array}{lcl} \langle \text{statement} \rangle & \rightarrow & (\\ & & \langle \text{vardecl} \rangle ";" | \\ & & \langle \text{atribstat} \rangle ";" | \\ & & \langle \text{printstat} \rangle ";" | \\ & & \langle \text{readstat} \rangle ";" | \\ & & \langle \text{returnstat} \rangle ";" | \\ & & \langle \text{superstat} \rangle ";" | \\ & & \langle \text{ifstat} \rangle | \\ & & \langle \text{forstat} \rangle | \\ & & "{" \langle \text{statlist} \rangle "}" | \\ & & "break" ";" | \\ & & ";" \\ & &) \end{array}$$

Esse não-terminal gera os seguintes tipos de cadeias, que correspondem a comandos da nossa linguagem e que serão definidos a seguir:

- declaração de variáveis locais;
- comando de atribuição;
- comando de impressão;
- comando de leitura;
- comando de término do método e retorno de valor;
- comando de seleção;
- comando de repetição;
- comando de interrupção de laço;
- comando vazio.

O não-terminal *vardecl* foi definido anteriormente para declaração de variáveis de classe. Usaremos esse mesmo terminal para a declaração de variáveis locais, pois sintaticamente essas duas construções são iguais. Vejamos os outros comandos válidos:

$$\langle \text{atribstat} \rangle \rightarrow \langle lvalue \rangle "=" (\langle \text{expression} \rangle | \langle \text{aloceexpression} \rangle)$$

Essa é a definição de um comando de atribuição, que tem uma referência a uma posição de memória (representada pelo não-terminal *lvalue*), seguida por um = e, depois, uma expressão ou uma referência a um novo objeto, utilizando o operador *new*. Por exemplo:

a = 0 ou
a[10] = b + c.d
a[10].b = new MyType()

$$\langle printstat \rangle \rightarrow "print" \langle expression \rangle$$

Esse não-terminal gera as cadeias que representam comandos de impressão. Elas contêm a palavra reservada *print*, seguida de uma expressão qualquer. Por exemplo:

```
print 123
print a
print a[10].b * c.d[e]
```

$$\langle readstat \rangle \rightarrow "read" \langle lvalue \rangle$$

Essa produção corresponde ao comando de leitura *read*. Ele é semelhante ao *print*, mas a segunda parte da produção (*lvalue*) representa uma referência a uma posição de memória, para que o comando de leitura faça sentido. Não podemos utilizar o não-terminal *expression*, pois queremos que as seguintes cadeias possam ser geradas por essa produção:

```
read a
reas a.b
read a.b[c+2]
```

mas não queremos gerar

```
read a + b
read 123
```

$$\langle returnstat \rangle \rightarrow "return" [\langle expression \rangle]$$

Produz cadeias correspondentes aos comandos de retorno de uma chamada de método. A expressão que segue a palavra *return* é opcional, pois, embora todos os métodos tenham que ser declarados como retornando algum valor ou objeto (veja o não-terminal *methoddecl*), isso não se aplica aos construtores (veja *constructdecl*) que não retornam nada. Assim, comandos do tipo

```
return 0
return a + b.c
```

são usados dentro dos métodos e o comando *return* sem expressão de retorno é utilizado nos construtores.

Nos construtores pode ser utilizado também o comando *super* para chamada do construtor da superclasse. Sua sintaxe é:

$$\langle superstat \rangle \rightarrow "super" "(" \langle arglist \rangle ")"$$

O *ifstat* é o não-terminal que gera os comando de seleção. Como na maioria das linguagens, podemos ter ou não a parte *else* do comando.

$$\langle ifstat \rangle \rightarrow "if" "(" \langle expression \rangle ")" \langle statement \rangle ["else" \langle statement \rangle]$$

Como comentamos anteriormente, o não-terminal *statement* pode gerar comandos compostos (ou blocos de comandos). Por isso, são válidos ambos os comandos:

```
if (a > 0)
    read b;
```

ou

```
if (a + b == c)
{
    read d;
    print d;
    return 0;
}
else
    return 1;
```

O único comando repetitivo na linguagem *X⁺⁺* é o *for*, semelhante ao existente na linguagem Java. Sua descrição é:

$$\langle forstat \rangle \rightarrow "for" "(" [\langle atribstat \rangle] ";" [\langle expression \rangle] ";" [\langle atribstat \rangle] ")" \langle statement \rangle$$

Ele inicia com a palavra *for*, seguida de um “(” e os três elementos: um comando de inicialização, uma expressão de controle e um comando de incremento, todos opcionais e separados por “;”. O primeiro e o último correspondem a *atribstats*, enquanto o segundo é uma *expression*. O comando termina com um “)” e com o comando (*statement*) que deve ser executado. São comandos válidos:

```
for (;;) ;
```

ou

```
for (a = 0; ; ) read b[a];
```

ou

```
for (a = 0; a < b; a = a + 1)
{
    read c[a];
    print c[a];
}
```

O comando *break* pode ser usado no corpo do *for* para interromper sua execução. Ele foi definido diretamente dentro do não-terminal *statement*. Lá também foi definido o comando nulo, representado por um ponto-e-vírgula sozinho como em

```
if (a > b)
;
else
    read b;
```

Finalmente, o comando composto, definido dentro do não-terminal *statement*, inicia-se com um “{” que vem seguido de uma lista de comandos, representado pelo não-terminal *statlist*, e termina com um “}”. O *statlist* é:

$$\langle \text{statlist} \rangle \rightarrow \langle \text{statement} \rangle [\langle \text{statlist} \rangle]$$

Note que λ não é gerado por tal produção. Isso significa que um bloco de comando vazio como

```
int mymethod(int a)
{
}
```

não é válido. Porém o programador pode utilizar

```
int mymethod(int a)
{
    ;
}
```

ou

```
int mymethod(int a)
;
```

Continuando com os não-terminais que ainda faltam definir, temos:

$$\langle lvalue \rangle \rightarrow \text{"ident"} ("[" \langle \text{expression} \rangle "]" | \text{"."} \text{"ident"} ["(" \langle \text{arglist} \rangle ")"])^*$$

Esse não-terminal representa uma referência a uma posição de memória e foi usado em *atribstat* e *readstat*. Tal cadeia inicia-se sempre com um identificador que é o nome de uma variável ou um método. Esse identificador pode vir seguido várias vezes por um índice no caso de se estar referindo a uma variável indexada; uma referência a um campo, no caso de se querer referenciar um campo de objeto; ou um nome de um método seguido por uma lista de argumentos, no caso em que se está fazendo uma chamada de um método. Por exemplo:

- `read a`; contém somente a referência a uma variável simples;
- `read a[0][1]` é uma referência a uma variável indexada;
- `read a.b` é uma referência ao campo *b* da variável *a*;
- `read a.b[0][1]` é uma combinação dos dois tipos anteriores;
- `read a.b(12).c` é uma combinação dos três tipos. Nesse caso, *a* é uma variável que referencia um objeto. Com esse objeto estamos invocando o método *b*, passando 12 como argumento. Esse método deve retornar um outro objeto do qual estamos referenciando o campo *c*.

Note que utilizando este não-terminal num comando de atribuição poderíamos ter `a.b(12) = 10`, o que é ilegal pois estaríamos tentando atribuir valor a uma chamada de método. Sintaticamente isso, porém, será permitido na nossa linguagem. Tal erro será apontado em fases posteriores da análise.

Num comando de atribuição podemos ter uma *aloceexpression* que é uma referência a um novo objeto ou array. Sua definição é:

$$\langle \text{aloceexpression} \rangle \rightarrow \text{"new"} (\text{"ident"} ("(\text{arglist})") | (\text{"int"} | \text{"string"} | \text{"ident"}) (\text{["} (\text{expression}) \text{"]})^+)$$

Esse não-terminal produz expressões do tipo

```
new MyType(10, a * b)
new string[10][i][k]
new MyType(10, i - k)
```

O não-terminal *expression* é utilizado diversas vezes para representar as possíveis expressões que podem ser construídas na linguagem *X⁺⁺*. Vamos ver como são essas expressões.

$$\langle \text{expression} \rangle \rightarrow \langle \text{numexpr} \rangle [(< | > | <= | >= | == | !=) \langle \text{numexpr} \rangle]$$

As expressões na nossa gramática são definidas em diversos “níveis” diferentes. O nível mais alto é este, que define como é uma expressão que possui operadores relacionais. Ela é composta de uma subexpressão que corresponde ao primeiro *numexpr* seguida por um operador relacional e uma segunda subexpressão que corresponde ao segundo *numexpr*. Cada subexpressão, como veremos adiante, pode conter também outras subexpressões e outros operadores, por exemplo + ou *.

Assim, para a cadeia `a + b < c * d`, teríamos a seguinte derivação:

$$\text{expression} \Rightarrow \text{numexpr} < \text{numexpr} \xrightarrow{*} a + b < \text{numexpr} \xrightarrow{*} a + b < c * d$$

Note, porém, que a segunda parte da produção é opcional. Uma outra possível derivação a partir deste não-terminal seria

$$\text{expression} \Rightarrow \text{numexpr} \xrightarrow{*} a + b$$

o que significa que $a + b$ também é uma expressão válida, gerada a partir de *expression*, embora não utilize operadores relacionais. O nível seguinte é o não-terminal *numexpr*.

$$\langle \text{numexpr} \rangle \rightarrow \langle \text{term} \rangle (("+" | "-") \langle \text{term} \rangle)^*$$

Essa produção é semelhante à anterior. A diferença é que a segunda parte não é apenas opcional, mas pode ser repetida um número finito de vezes. Essa diferença baseia-se no fato de que queremos gerar expressões do tipo $a + b + c * d$, mas não cadeias do tipo $a < b < c$, pois, em geral, tal expressão não faz muito sentido. A derivação para $a + b + c * d$ seria:

$$\begin{aligned} \text{expression} &\Rightarrow \text{numexpr} \Rightarrow \text{term} + \text{term} + \text{term} \xrightarrow{*} a + \text{term} + \text{term} \xrightarrow{*} a + b + \\ &\text{term} \xrightarrow{*} a + b + c * d \end{aligned}$$

O não-terminal *term* segue o mesmo raciocínio para expressões com os operadores $*$, $/$ e $\%$ (resto da divisão).

$$\langle \text{term} \rangle \rightarrow \langle \text{unaryexpr} \rangle (("*" | "/" | "%") \langle \text{unaryexpr} \rangle)^*$$

Uma expressão que contenha um operador unário como $a + -1$ utiliza a seguinte produção:

$$\langle \text{unaryexpr} \rangle \rightarrow [("+" | "-")] \langle \text{factor} \rangle$$

Aqui, novamente, vemos que *unaryexpr* pode gerar cadeias que se iniciem com os operadores $+$ ou $-$ seguidos por uma cadeia gerada pelo não-terminal *factor*, e pode gerar também aquelas cadeias geradas por *factor* sem os operadores como prefixo. Por exemplo, para $a + -1$ teríamos

$$\begin{aligned} \text{expression} &\Rightarrow \text{numexpr} \Rightarrow \text{term} + \text{term} \xrightarrow{*} a + \text{term} \Rightarrow a + \text{unaryexpr} \Rightarrow \\ &a + -\text{factor} \xrightarrow{*} a + -1 \end{aligned}$$

O não-terminal *factor* representa as subexpressões mais simples que podemos ter, tais como constantes ou referências a variáveis.

$$\langle \text{factor} \rangle \rightarrow ("int-constant" | "string-constant" | "null" | \langle \text{lvalue} \rangle | \\ "(" \langle \text{expression} \rangle ")")$$

Assim, esse não-terminal produz constantes inteiras, constantes string, a constante *null*, referências a variáveis e chamadas de métodos (não-terminal *lvalue*) e, ainda, expressões entre parênteses. Esse último caso é necessário para que a nossa gramática

possa gerar cadeias como $a + (b + c)$. A derivação seria:

$$\begin{aligned} \text{expression} &\Rightarrow \text{numexpr} \Rightarrow \text{term} + \text{term} \xrightarrow{*} a + \text{term} \Rightarrow a + \text{unaryexpr} \Rightarrow a + \\ \text{factor} &\xrightarrow{*} a + (\text{expression}) \Rightarrow a + (\text{numexpr}) \Rightarrow a + (\text{term} + \text{term}) \xrightarrow{*} a + (b + \text{term}) \xrightarrow{*} \\ &a + (b + c) \end{aligned}$$

E, finalmente, o não-terminal que deriva cadeias que representam listas de argumentos, usadas em chamadas de métodos, é:

$$\langle \text{arglist} \rangle \rightarrow [\langle \text{expression} \rangle (, \langle \text{expression} \rangle)^*]$$

2.5 Grafos sintáticos

Veremos ainda, uma outra maneira de representar uma linguagem, chamada de grafo sintático. Essa representação facilita a visualização do tipo de cadeias ou formas sentenciais que cada não-terminal pode gerar. Para isso, define-se para cada símbolo não terminal um grafo direcionado com dois tipos de vértices: os que são rotulados com símbolos terminais e aqueles que são rotulados com símbolos não terminais.

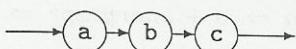
Cada um desses grafos corresponde ao lado direito de uma produção numa gramática na BNF. Uma aresta de um nó (vértice) com rótulo A para um nó com rótulo B indica que a subcadeia AB é parte da produção desse terminal, ou seja, tal aresta indica a concatenação da cadeia A com a cadeia B na produção desse não-terminal. Todo grafo possui um único nó inicial, sobre o qual não incide nenhuma aresta, e um único nó final de onde nenhuma aresta parte, ambos rotulados com a cadeia λ .

Assim, se tivermos uma produção:

$$A \rightarrow abc$$

teremos o grafo correspondente ao não-terminal A :

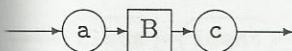
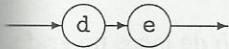
A



Note que os nós inicial e final não são representados no diagrama do grafo. Todas as arestas que não possuem um nó de origem (destino) no diagrama têm o nó inicial (final) como origem (destino). Os nós correspondentes a símbolos não terminais são representados no diagrama por retângulos. Por exemplo, para as produções

$$\begin{aligned} A &\rightarrow aBc \\ B &\rightarrow de \end{aligned}$$

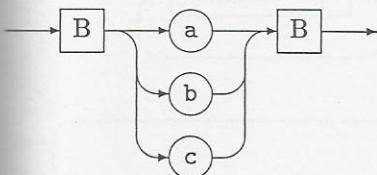
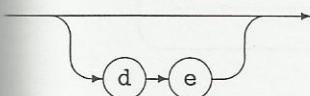
temos:

A*B*

Para sabermos quais são as formas sentenciais que podem ser geradas por um não-terminal, devemos percorrer todos os caminhos que vão do nó inicial ao nó final do grafo correspondente ao não-terminal. Se tivermos operadores de seleção na produção do não-terminal teremos caminhos alternativos no grafo, cada um passando por uma das cadeias que estão dentro do operador de seleção. Por exemplo,

$$\begin{array}{lcl} A & \rightarrow & B(a \mid b \mid c)B \\ B & \rightarrow & (de \mid \lambda) \end{array}$$

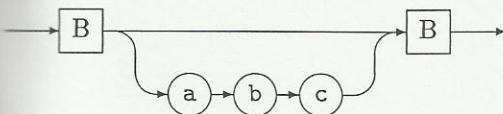
temos:

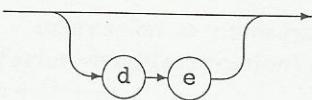
A*B*

Note que um nó rotulado com λ pode ser sempre evitado no grafo, como fizemos no não-terminal B . Dessa forma, sabemos também como representar uma produção que utilize cadeias opcionais, por meio do operador $[]$. Por exemplo, para

$$\begin{array}{lcl} A & \rightarrow & B[abc]B \\ B & \rightarrow & (de \mid \lambda) \end{array}$$

temos:

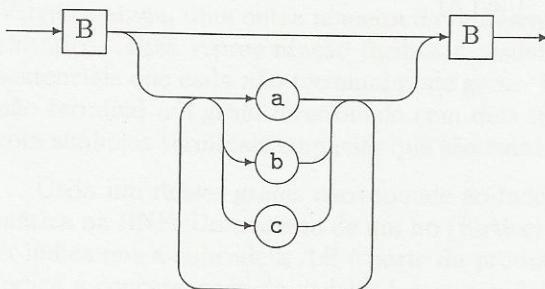
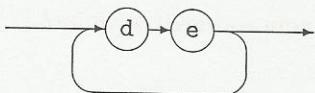
A

B

E os operadores de repetição podem ser representados por meio de laços no grafo. Por exemplo,

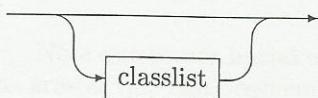
$$\begin{array}{lcl} A & \rightarrow & B(a \mid b \mid c)^*B \\ B & \rightarrow & (de)^+ \end{array}$$

pode ser representado como

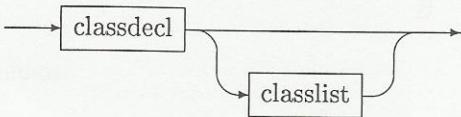
A*B*

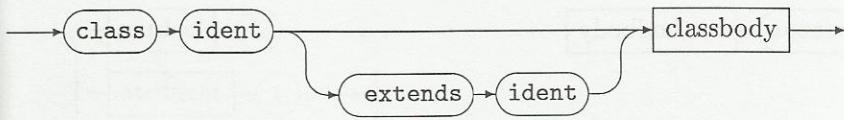
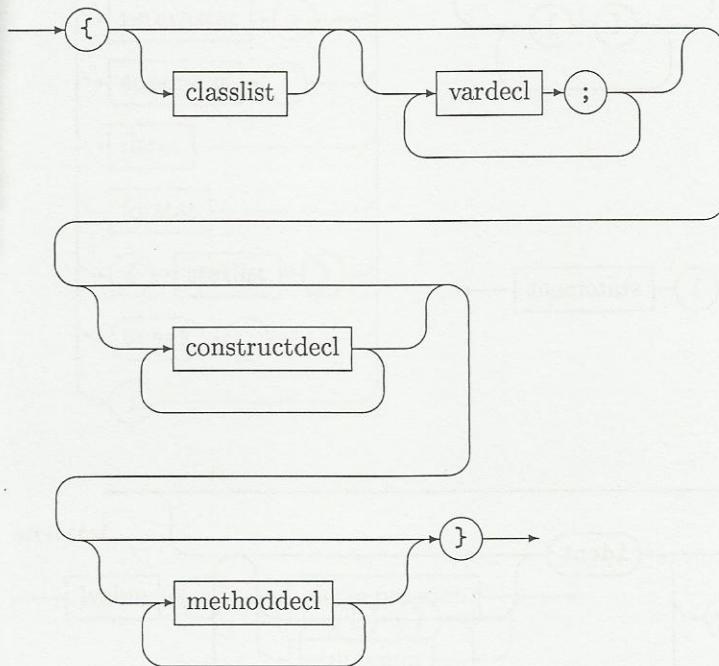
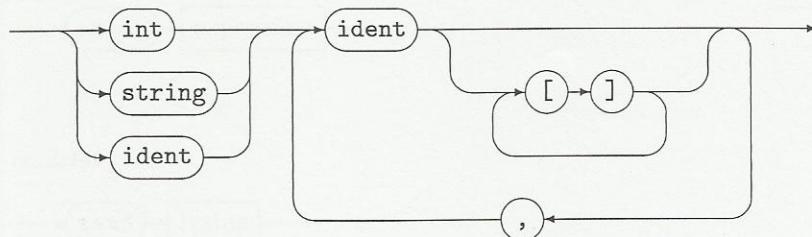
Veremo então como fica a gramática da linguagem X^{++} que definimos na seção anterior representada por meio de um grafo sintático.

program

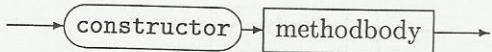


classlist

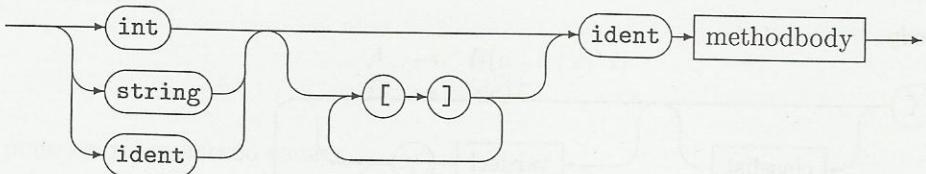


classdecl*classbody**vardecl*

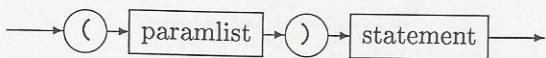
constructdecl



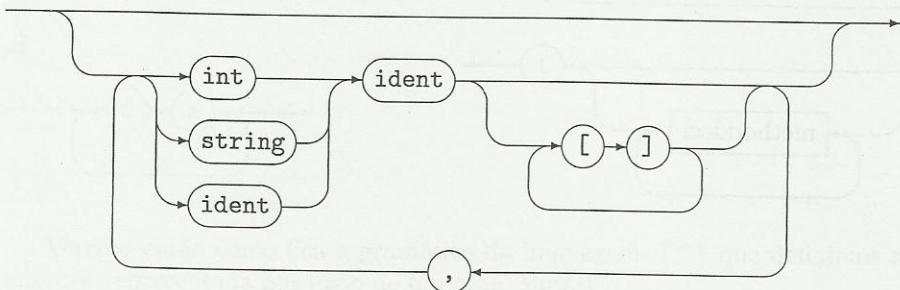
methoddecl



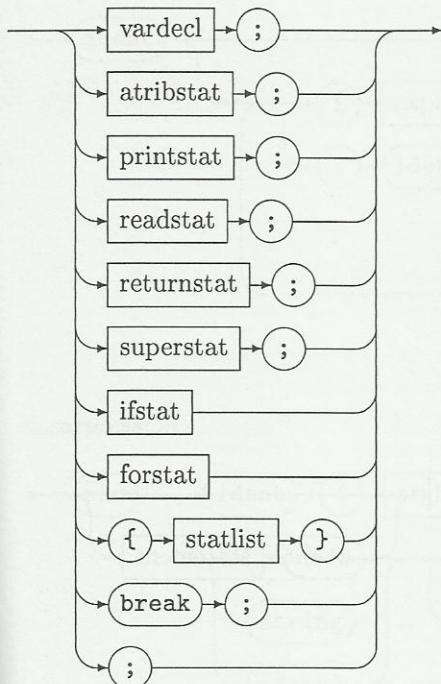
methodbody



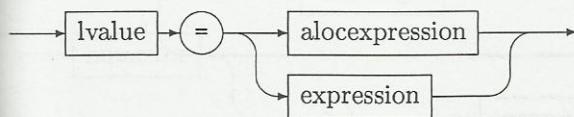
paramlist



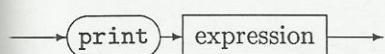
statement



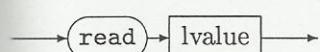
atribstat

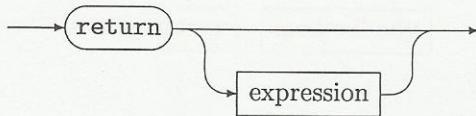
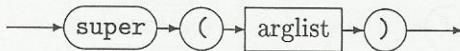
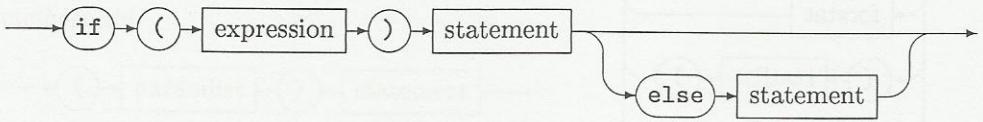
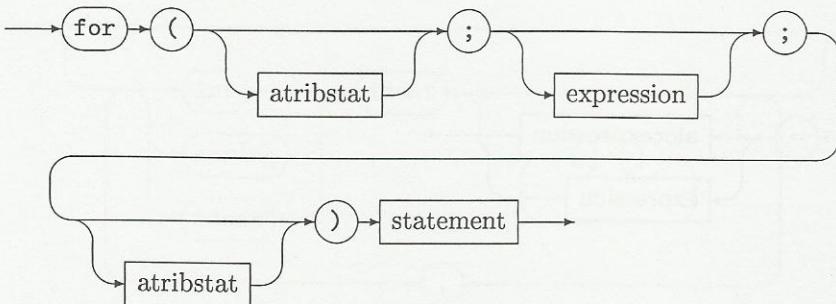
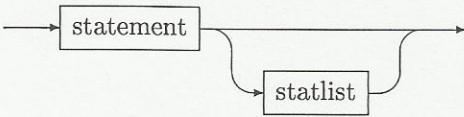


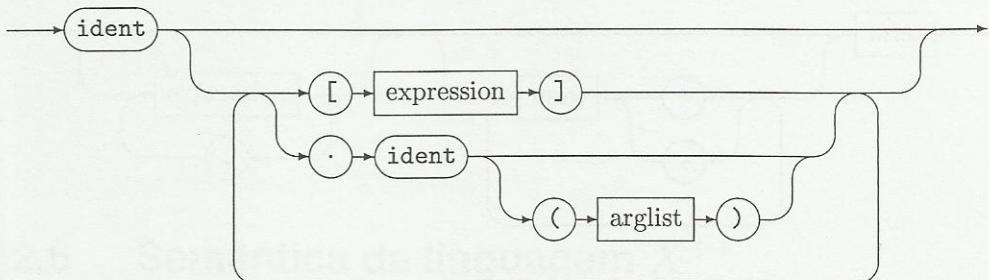
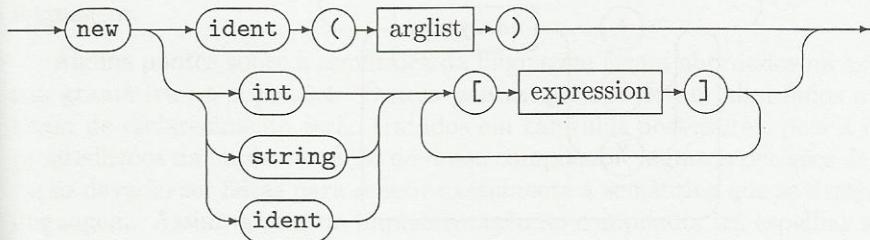
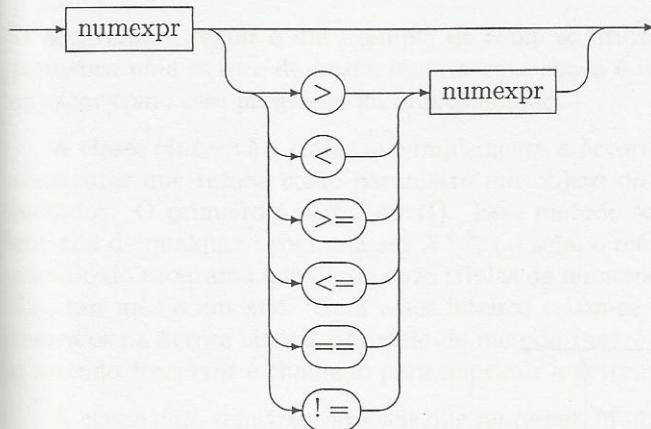
printstat



readstat

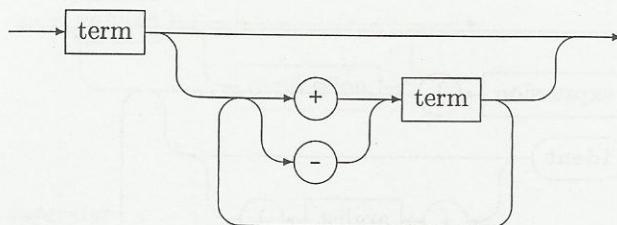


returnstat*superstat**ifstat**forstat**statlist*

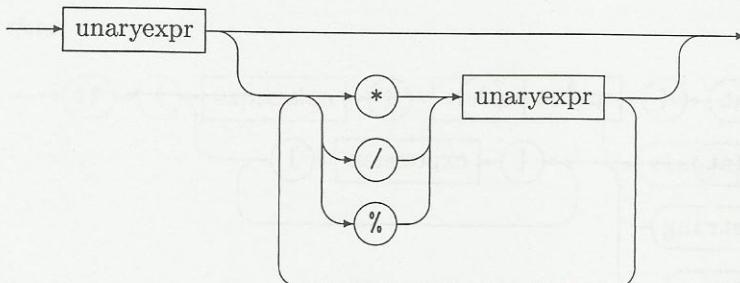
lvalue*alocexpression**expression*

32 COMO CONSTRUIR UM COMPILADOR

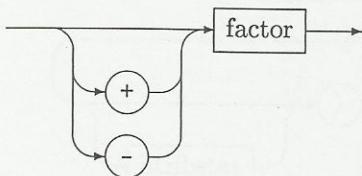
numexpr



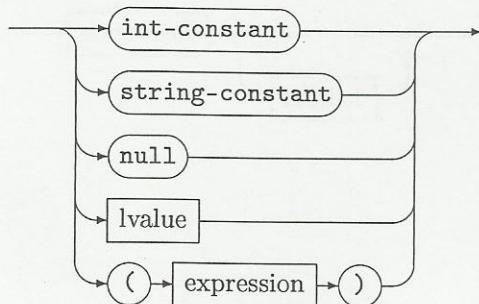
term



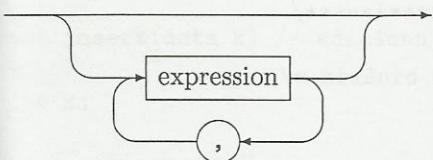
unaryexpr



factor



arglist



2.6 Semântica da linguagem X++

Não tentaremos aqui elaborar uma definição formal da semântica da linguagem X++ pois tal assunto não se restringe os objetivos deste livro. Porém, o leitor familiarizado com algumas linguagens de programação certamente será capaz, por meio da sintaxe definida, de identificar estruturas conhecidas que têm a mesma semântica na nossa linguagem.

Alguns pontos sobre a semântica da linguagem foram abordados na explicação da sua gramática na seção 2.4. Outros pontos que não foram abordados e que necessitam de esclarecimento serão tratados em capítulos posteriores, pois à medida que progredirmos na implementação do nosso compilador, algumas decisões de implementação deverão ser feitas para refletir exatamente a semântica que se deseja atribuir à linguagem. Assim, a própria implementação do compilador irá espelhar a semântica da linguagem.

2.7 Um programa em X++

O programa a seguir é um exemplo de como se utiliza a linguagem X++. Ele implementa uma árvore de busca binária cuja chave é uma data. Vamos brevemente explicar como esse programa foi implementado.

A classe *bintree* é a classe que implementa a árvore binária. Ela possui um único construtor que recebe como parâmetro um objeto do tipo *data*. Possui ainda três métodos. O primeiro é o *int start()*. Esse método será por convenção o ponto de entrada de qualquer programa em X++, ou seja, o método principal. Ele solicita ao usuário do programa que digite onze triplas de números inteiros que representam um dia, um mês e um ano. Com esses inteiros criam-se objetos do tipo *data* que são inseridos na árvore binária por meio do método *insert* desta classe *bintree*. No final, o método *treeprint* é chamado para imprimir a árvore criada.

A classe *data* tem três variáveis que representam um dia, um mês e um ano. Dois construtores são os responsáveis pela criação dos objetos dessa classe. Note-se que nenhuma verificação é feita quanto à validade da data que está sendo criada. O método *compara* dessa classe compara dois objetos do tipo *data* e é usado no método *insert* da classe *bintree* para encontrar o “lado” da árvore em que um determinado elemento deve ser inserido.

```

/***********************
Esse programa implementa uma árvore de busca binária
***********************/

class bintree { /* define o nó da árvore binária */

class data { // define um classe aninhada do tipo data (dia, mes, ano)
int dia, mes, ano;

constructor() // construtor 1, sem parâmetros
{
    ano = 1900; // inicializa em 1/1/1900
    mes = 1;
    dia = 1;
}

constructor(int d, int m, int a) // construtor 2 - dia, mês e ano como
{                               // parâmetros
    dia = d;
    mes = m;
    ano = a;
}

int compara(data x) // compara duas datas
{
    // retorna < 0 - menor > 0 maior 0 igual
    if (ano < x.ano) then return -1;
    if (ano > x.ano) then return 1;
    if (mes < x.mes) then return -1;
    if (mes > x.mes) then return 1;
    if (dia < x.dia) then return -1;
    if (dia > x.dia) then return 1;
    return 0;
}

} // final classe data

// variáveis da classe bintree

data key;      // chave de comparação
bintree left,right; // referência para os filhos

constructor(data x)
{
    key = x;
    left = null;
}

```

```
    right = null;
}

int insert(data k) // adiciona um elemento na árvore
{
int x;

x = k.compara(key);
if (x < 0) then
{
    if (left != null)
        then return left.insert(k);
    left = new bintree(k);
    return 1;
}
if (x > 0) then
{
    if (right != null)
        then return right.insert(k);
    right = new bintree(k);
    return 1;
}
return 0;
}

int treeprint(int x) // imprime a árvore
{
int i;

if (left != null)
    then i = left.treeprint(x+4);
for (i = 0; i < x; i = i + 1)
    print " ";
print key.dia+ "/" + key.mes + "/" + key.ano + "\n";
if (right != null)
    then i = right.treeprint(x+4);
}

int start()
{
bintree t;
int i, d, m, a;
data w;

print "Digite o dia: ";
read d;
```

```
print "Digite o mes: ";
read m;
print "Digite o ano: ";
read a;
w = new data(d, m, a);
t = new bintree(w);
for (i = 0; i < 10; i = i + 1)
{
    print "Digite o dia: ";
    read d;
    print "Digite o mes: ";
    read m;
    print "Digite o ano: ";
    read a;
    w = new data(d, m, a);
    if (t.insert(w) == 0)
        then print "Elemento ja existe\n";
}
i = t.treeprint(0);
return 0;
}
```

Capítulo 3

Análise Léxica

Neste capítulo veremos como construir o analisador léxico (AL) para a linguagem X^{++} . Como vimos, a tarefa do analisador léxico é quebrar a entrada – que, em geral, está armazenada num arquivo texto normal – em símbolos que façam sentido para a definição da linguagem e para o analisador sintático. Para identificarmos quais são esses símbolos, basta olharmos na definição da linguagem – sua gramática ou grafo sintático – e identificarmos quais são os seus símbolos terminais. No contexto da análise léxica, costumamos chamar esses símbolos de tokens.

Note que os símbolos não terminais compõem o alfabeto sobre o qual a nossa linguagem é definida. Por outro lado, esse alfabeto nada mais é que um conjunto de símbolos e, por isso, pode ser definido como uma linguagem, sobre um outro alfabeto. No caso, esse alfabeto será qualquer caractere ASCII que possa ser armazenado num arquivo texto.

Essa linguagem, formada pelos não-terminais, é, em geral, bem mais simples que a linguagem-alvo do compilador pois não inclui estruturas complexas como comandos aninhado ou pares de símbolos casados como parênteses, colchetes ou chaves. Por isso, podemos utilizar um modelo mais simples para representar tais linguagens, chamadas de linguagens regulares. Neste capítulo iremos rever os conceitos sobre autômatos finitos e expressões regulares, que são duas formas de representar linguagens regulares.

3.1 Autômatos finitos

Um autômato finito determinístico (AFD) é um modelo para definição de linguagens regulares composto de cinco elementos $\langle \Sigma, S, s_0, \delta, F \rangle$, onde:

Σ é o alfabeto sobre o qual a linguagem é definida;

S é um conjunto finito não vazio de estados;

s_0 é o estado inicial, $s_0 \in S$;

δ é a função de transição de estados, $\delta : S \times \Sigma \rightarrow S$;

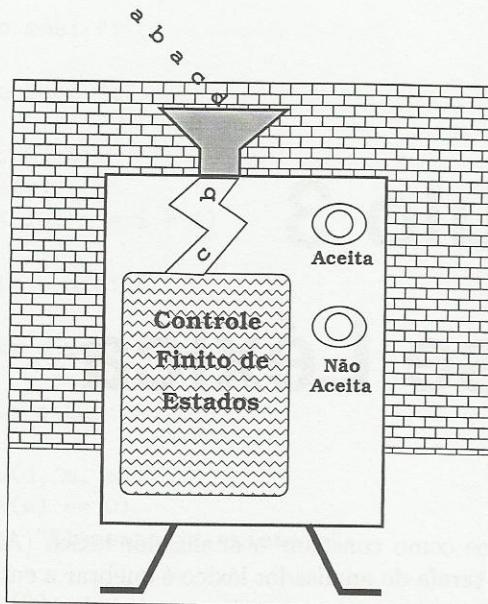


Figura 3.1 – Visão de um AFD como uma máquina.

F é o conjunto de estados finais, $F \subseteq S$.

Um AFD pode ser interpretado como uma “máquina de reconhecer cadeias”, como a mostrada na figura 3.1. Ele recebe como entrada uma cadeia e diz se ela pertence ou não à linguagem desejada. Para fazer isso, essa máquina possui um controle finito de estados (CFE) e um conjunto de estados S . O CFE coloca sempre a máquina em um estado pertencente ao conjunto S . A função δ diz como o AFD deve mudar de estado, à medida que as letras da cadeia de entrada vão sendo analisadas. Ao final da cadeia, isso é, depois de analisar uma por uma, todas as letras da entrada e realizar as mudanças de estados determinadas, o AFD aceita a cadeia – acendendo o indicador superior no painel – se o estado em que ele se encontra pertence ao subconjunto F . Caso contrário, a cadeia é rejeitada – acendendo a luz inferior do painel –, indicando que não pertence à linguagem definida pelo AFD.

Vamos tomar como exemplo o AFD A_1 que reconhece a linguagem $\{x \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}^* \mid x \text{ representa um número decimal divisível por } 3\}$. $A_1 = \langle \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}, \{s_0, s_1, s_2\}, s_0, \delta, \{s_0\} \rangle$, onde δ é:

$\delta(s_0, 0)$	=	s_0	$\delta(s_1, 0)$	=	s_1	$\delta(s_2, 0)$	=	s_2
$\delta(s_0, 1)$	=	s_1	$\delta(s_1, 1)$	=	s_2	$\delta(s_2, 1)$	=	s_0
$\delta(s_0, 2)$	=	s_2	$\delta(s_1, 2)$	=	s_0	$\delta(s_2, 2)$	=	s_1
$\delta(s_0, 3)$	=	s_0	$\delta(s_1, 3)$	=	s_1	$\delta(s_2, 3)$	=	s_2
$\delta(s_0, 4)$	=	s_1	$\delta(s_1, 4)$	=	s_2	$\delta(s_2, 4)$	=	s_0
$\delta(s_0, 5)$	=	s_2	$\delta(s_1, 5)$	=	s_0	$\delta(s_2, 5)$	=	s_1
$\delta(s_0, 6)$	=	s_0	$\delta(s_1, 6)$	=	s_1	$\delta(s_2, 6)$	=	s_2
$\delta(s_0, 7)$	=	s_1	$\delta(s_1, 7)$	=	s_2	$\delta(s_2, 7)$	=	s_0
$\delta(s_0, 8)$	=	s_2	$\delta(s_1, 8)$	=	s_0	$\delta(s_2, 8)$	=	s_1
$\delta(s_0, 9)$	=	s_0	$\delta(s_1, 9)$	=	s_1	$\delta(s_2, 9)$	=	s_2

Assim, por exemplo, quando executado com o string 01452, o AFD A_1 teria o seguinte comportamento:

Estado corrente	Letra lida	Próximo estado
s_0 (estado inicial)	0	s_0
s_0	1	s_1
s_1	4	s_2
s_2	5	s_1
s_1	2	$s_0 \in F$

Como o AFD termina sua execução no estado s_0 que pertence ao conjunto de estados finais, então essa cadeia pertence à linguagem definida por A_1 . Já para a cadeia 79612, teríamos:

Estado corrente	Letra lida	Próximo estado
s_0 (estado inicial)	7	s_1
s_1	9	s_1
s_1	6	s_1
s_1	1	s_2
s_2	2	$s_1 \notin F$

o que indica que tal cadeia não pertence à linguagem definida por A_1 .

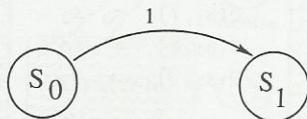
Um AFD pode ser representado por meio de uma tabela de transição de estados. Nessa tabela existe uma linha para cada estado e uma coluna para cada letra do alfabeto de entrada. Dados o estado s e a letra $a \in \Sigma$, coloca-se na posição da tabela $s \times a$ o valor de $\delta(s, a)$. Por exemplo, para A_1 , teríamos:

δ	0	1	2	3	4	5	6	7	8	9
s_0	s_0	s_1	s_2	s_0	s_1	s_2	s_0	s_1	s_2	s_0
s_1	s_1	s_2	s_0	s_1	s_2	s_0	s_1	s_2	s_0	s_1
s_2	s_2	s_0	s_1	s_2	s_0	s_1	s_2	s_0	s_1	s_2

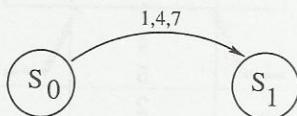
Uma das vantagens de se utilizar um AFD para definir quais são os tokens a serem reconhecidos pelo AL é que é fácil implementar um AL baseado na tabela de transição de estados. Tal analisador deve apenas ler uma letra da entrada e, baseado na tabela, fazer a mudança de estado. Se o estado em que o AFD se encontrar for um estado

final, então a cadeia lida até aquele ponto é um token válido. Antes de tratarmos de mais detalhes sobre esse tipo de AL vejamos uma outra forma de representar um AFD que é o diagrama de transição de estados.

Esse diagrama é um grafo direcionado cujos vértices, representados por círculos correspondem aos estados do AFD e cujas arestas correspondem às transições entre estados. Por exemplo, a transição $\delta(s_0, 1) = s_1$ é representada por:



No caso em que temos diversas transições definidas entre dois estados, podemos representá-las todas com uma única seta, rotulada com todas as letras associadas às transições. Por exemplo, se temos $\delta(s_0, 1) = \delta(s_0, 4) = \delta(s_0, 7) = s_1$, representamos:



Há uma representação especial para indicar o estado inicial. É uma seta chegando ao estado, sem nenhum estado de origem. Os estados finais são identificados por círculos duplos. O diagrama para o AFD A_1 é dado na figura 3.2

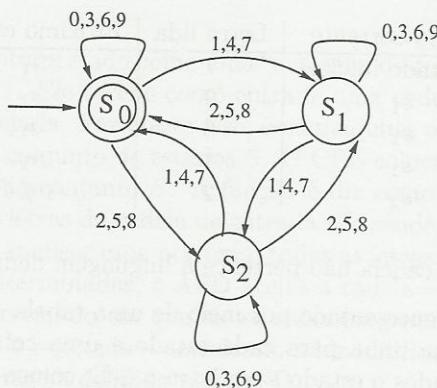
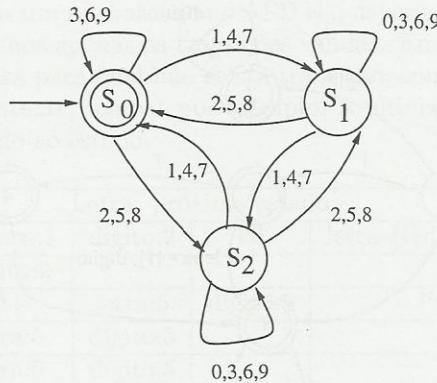


Figura 3.2 – Diagrama de transição de estado para A_1 .

Para utilizar um AFD como AL vamos faremos algumas pequenas modificações no comportamento desses autômatos. A primeira é que vamos permitir que a função de transição seja uma função parcial, ou seja, não definida para todos os pontos do domínio $S \times \Sigma$. Por exemplo, vamos tomar o AFD A_2 que é igual a A_1 , mas para o qual o valor de $\delta(s_0, 0)$ é indefinido. Teríamos, então, para esse AFD o diagrama da figura 3.3. Esse autômato reconhece todas as cadeias que representam múltiplos de 3

Figura 3.3 – Diagrama de transição de estado para A_2 .

menos algumas que têm o dígito 0 como 2409. Ao processar tal cadeia, o comportamento de A_2 seria:

Estado corrente	Letra lida	Próximo estado
s_0 (estado inicial)	2	s_2
s_2	4	s_0
s_0	0	???

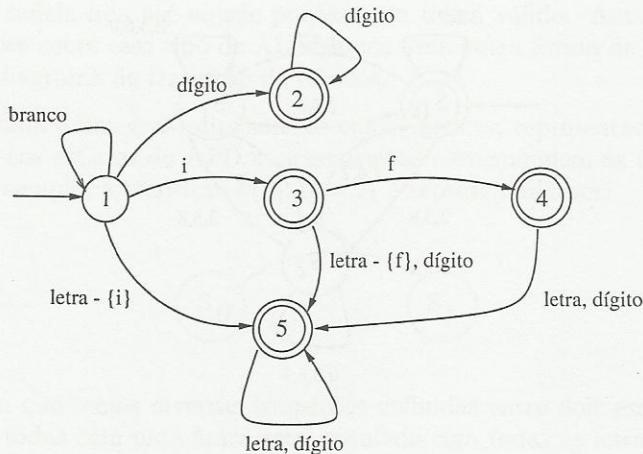
Isso significa que o AFD pára antes de chegar ao final da cadeia. No nosso AL, o AFD processa a cadeia de entrada até que não existam mais transições possíveis, então ele pára. Se o último estado em que o AFD estava quando parou for um estado final, então um token válido foi lido. Caso contrário, um erro léxico ocorreu.

Tomaremos como exemplo um AL que deve reconhecer apenas três tipos de tokens: números inteiros; identificadores, formados por letras e dígitos, sempre iniciados por uma letra; e a palavra reservada `if`. Podemos construir o AFD A_3 mostrado na figura 3.4.

Veja que o AFD da figura 3.4 consome todos os “brancos” como espaço, tab, etc, no estado 1. Depois, dependendo de qual é a letra seguinte, ele desvia para os estados 2, 3 ou 4 para reconhecer, respectivamente, um número, a palavra `if` ou o nome de um identificador. Ele então continua, consumindo letras até que nenhuma transição exista. Quando isso ocorre, um token foi identificado se o estado em que ele parou é final. Um erro léxico foi achado se o estado não é final. Portanto, cada vez que o AL é executado, um novo token é reconhecido. É exatamente assim que o analisador sintático utiliza o AL. A cada vez que o AS necessita de um token, ele executa o AL que analisa a entrada e lhe fornece um token novo.

Uma das características do AFD A_3 é que ele irá reconhecer como token a maior palavra que ele puder formar com as letras da entrada. Por exemplo, se tivermos na entrada:

i f 1234

Figura 3.4 – Diagrama de transição de estado para A_3 .

o AFD irá retornar na sua primeira execução o identificador *i*, na segunda, o identificador *f* e, na terceira, o número 1234. Se a entrada for

if 1234

a primeira chamada irá retornar a palavra reservada *if* e, depois, o número 1234. E se a entrada for

if1234

teremos como token retornado o identificador *if1234*.

Um outro ponto importante dessa implementação do AL é que, em virtude de sua execução parar quando não existem mais transições, algumas palavras da entrada não precisam ser separadas por espaços para que sejam corretamente reconhecidas. Por exemplo, se tivermos a entrada

1234if1234

iremos obter na primeira execução do AL o inteiro 1234 e, numa segunda execução, o identificador *if1234*.

Para completarmos o nosso AL baseado num AFD, precisamos ainda associar a cada nó qual o tipo de token associado a ele. Por exemplo, no AFD A_3 , sabemos que se sua execução termina no estado 2, um número foi reconhecido; se termina nos estados 3 ou 5, um identificador foi reconhecido; e se termina no estado 4, um *if* foi reconhecido. Além disso, precisamos saber mais algumas informações sobre a palavra que fez com que tal token fosse reconhecido. Principalmente em tokens como identificadores e números, precisamos saber qual foi a palavra que originou o token (por exemplo, se a cadeia é 10 ou 10000000). E precisamos saber também qual é a localização do token, por exemplo, em quais linha e coluna do arquivo de entrada essa palavra se encontra. Essa informação é útil para que o AS possa indicar – em caso de erro, por exemplo – o ponto onde está a palavra.

Assim, para construir um AL baseado no AFD A_3 , deveríamos construir a seguinte tabela, onde representamos apenas as transições válidas. Em qualquer estado, se for lida na entrada uma letra para qual não exista transição, considera-se como próximo estado um valor não existente, como 0, por exemplo. A última coluna da tabela indica o tipo do token associado ao estado.

Estado	Letra: próximo estado				Tipo
1	branco:1	digito:2	i:3	letra-{i}:5	ERRO
2	digito:2				NÚMERO
3	f:4	letra:5	digito:5		IDENT
4	letra:5	digito:5			IF
5	letra:5	digito:5			IDENT

E o programa para executá-lo seria parecido com o mostrado no programa 3.1.

Programa 3.1

```

1   enquanto c eh branco // c armazena a última letra lida
2   |   leia c           // ignora brancos
3   |   se c == fim de linha // controla linha e coluna correntes
4   |   |   coluna = 1
5   |   |   linha = linha + 1
6   |   senao
7   |   |   coluna = coluna + 1
8   k = 1           // estado inicial
9   s = ""          // palavra lida
10  linha0 = linha // linha onde se inicia o token
11  coluna0 = coluna // coluna onde se inicia o token
12  u = k           // inicializa último estado visitado
13  k = proximo estado // próximo estado na tabela (de acordo com c)
14  enquanto k > 0
15  |   s += c        // concatena c à palavra
16  |   leia c           // lê próxima letra da entrada
17  |   se c == fim de linha
18  |   |   coluna = 1
19  |   |   linha = linha + 1
20  |   senao
21  |   |   coluna = coluna + 1
22  |   u = k           // último estado visitado
23  |   k = proximo estado // próximo estado (de acordo com c)
24  se tabela[u].Token === ERRO // u não é estado final
25  |   Erro Lexico      // tratamento de erro léxico
26  senao
27  |   t.tipo = tabela[k].Token
28  |   t.palavra = s
29  |   t.linha = linha0
30  |   t.coluna = coluna0
31  retorno t          // retorna o token obtido

```

Nas linhas 1 e 2, a transição que consome os brancos iniciais é tratada de maneira especial, pois precisamos saber em que linha e que coluna inicia-se a palavra que originou o token. Esses brancos podem, na verdade, ser vistos como aqueles separadores

de palavras a que nos referimos no início do capítulo, em vez de letras que compõem as palavras. Não podemos, porém, ignorar todos os brancos que aparecerem na entrada, pois alguns deles podem fazer parte de tokens como numa constante do tipo string.

As variáveis *linha*, *coluna* e *c* devem ser inicializadas antes de se utilizar pela primeira vez o AL. Elas retêm seus valores entre duas execuções do AL. No início de cada execução, seus valores correspondem à próxima letra a ser utilizada.

Na linguagem X^{++} , que é bastante simples, temos nada menos que 39 símbolos terminais que teriam que ser reconhecidos pelo AL. Outras linguagens mais complexas podem ter um número ainda superior, o que dificulta bastante a construção do AFD correspondente. Uma das maneiras de diminuir essa dificuldade é utilizando um autômato finito não determinístico (AFND) que é um modelo ligeiramente diferente para representar a linguagem do AL.

Um AFND difere de um AFD em relação ao fato de ter diversos estados iniciais e que a função de transição de estados é definida $\delta : S \times \Sigma \rightarrow$ conjunto de subconjuntos finitos de $\rho(S)$, ou seja, pode existir mais de uma transição saindo de um estado para a mesma letra do alfabeto. Por exemplo, o AFND A_4 mostrado na figura 3.5 e o AFND A_5 da figura 3.6 reconhecem a mesma linguagem que o AFD A_3 mostrado na figura 3.4.

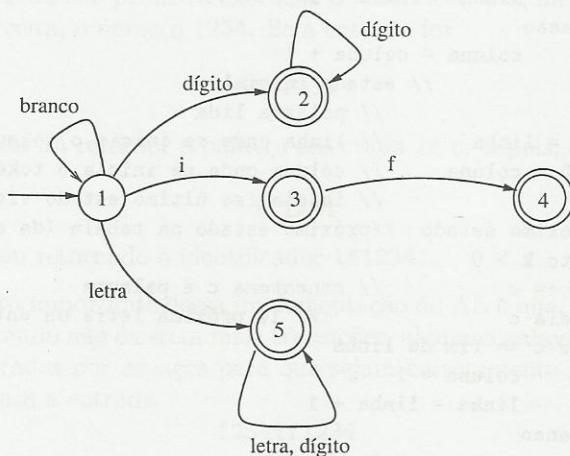


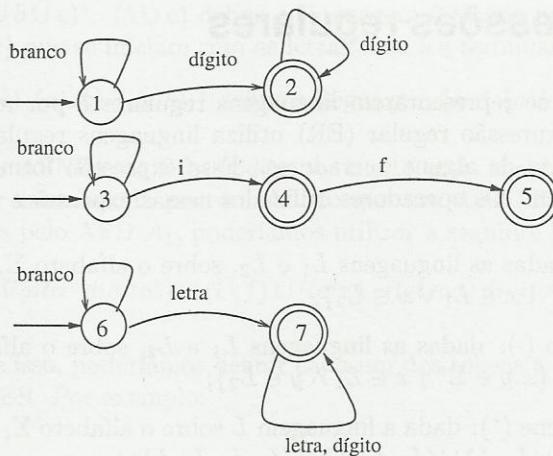
Figura 3.5 – Diagrama de transição de estado para A_4 .

Quando executado, um AFND pode ter mais de um estado corrente ao mesmo tempo. Por exemplo, ao processar as cadeias *if12* e *if* no AFND A_4 , a seqüência de estados ativos para cada um seria:

$$\text{if12 } \{1\} \xrightarrow{i} \{3, 5\} \xrightarrow{f} \{4, 5\} \xrightarrow{1} \{5\} \xrightarrow{2} \{5\}$$

$$\text{if } \{1\} \xrightarrow{i} \{3, 5\} \xrightarrow{f} \{4, 5\}$$

E no AFND A_5 , seria:

Figura 3.6 – Diagrama de transição de estado para A_5 .

if12 $\{1, 3, 6\} \xrightarrow{i} \{4, 7\} \xrightarrow{f} \{5, 7\} \xrightarrow{1} \{7\} \xrightarrow{2} \{7\}$

if $\{1, 3, 6\} \xrightarrow{i} \{4, 7\} \xrightarrow{f} \{5, 7\}$

Um AFND reconhece uma cadeia se, ao final da sua execução, algum dos estados correntes pertence ao conjunto de estados finais F . Executando A_4 com *if12*, teríamos o estado 5 ativo no final da execução do AFND e, portanto, *if12* deveria ser reconhecido como identificador. Já no segundo caso, ao final da cadeia *if*, teríamos dois estados correntes, ambos pertencentes ao conjunto F . Se fôssemos utilizar esse AFND para implementar no AL, teríamos um problema, pois não saberíamos qual o token correspondente à palavra lida, se seria um identificador ou um *if*. Teríamos, então, que estabelecer prioridades para decidir entre dois estados que terminaram ativos.

Apesar de facilitar a definição da linguagem desejada, um AFND não é tão facilmente implementado quanto um AFD. Tente, por exemplo, adaptar o algoritmo do AL que percorre a tabela de transições para um AFND. Por outro lado, as características de não-determinismo não adicionam aos AFNDs nenhum “poder” extra em relação aos AFDs. Isso significa que a classe de linguagens que podem ser definidas por meio de AFNDs é a mesma das que podem ser definidas por meio de AFDs.

Assim sendo, qualquer AFND pode ser transformado num AFD equivalente. Podemos, então, definir nossa linguagem por meio de um AFND, transformá-lo num AFD e, então, montar a tabela do AL. Ou melhor ainda, podemos escrever um programa que faça isso para nós (ou usar um que já exista). É exatamente essa a filosofia dos programas que geram ALs mediante a descrição da linguagem numa forma mais amigável. Esses programas acabam transformando essa descrição num AFD.

Na sessão 3.2 veremos outra forma de descrever linguagens regulares. São as expressões regulares, que, com os AFNDs, serão usadas na construção do AL para a linguagem X^{++} na seção 3.3.

3.2 Expressões regulares

Outra maneira de se representarem linguagens regulares é por meio de expressões regulares. Uma expressão regular (ER) utiliza linguagens regulares “primitivas” e combina-as por meio de alguns operadores. Essa expressão formada define, então, uma outra linguagem. Os operadores utilizados nessas expressões são os seguintes:

- União (\cup): dadas as linguagens L_1 e L_2 , sobre o alfabeto Σ , define-se $L_1 \cup L_2$ como $\{x \in \Sigma^* \mid x \in L_1 \vee x \in L_2\}$;
- Concatenação (\cdot): dadas as linguagens L_1 e L_2 , sobre o alfabeto Σ , define-se $L_1 \cdot L_2$ como $\{x.y \in \Sigma^* \mid x \in L_1 \wedge y \in L_2\}$;
- Fecho de Kleene (*): dada a linguagem L sobre o alfabeto Σ , define-se L^* como sendo $\lambda \cup L \cup (L \cdot L) \cup (L \cdot L \cdot L) \cup (L \cdot L \cdot L \cdot L) \cup \dots$.

Alguns exemplos da aplicação desses operadores são:

$$\begin{aligned} \{a, b\} \cup \{c, d\} &= \{a, b, c, d\} \\ \{100, 010, 110\} \cup \{00, 01, 11\} &= \{100, 010, 110, 00, 01, 11\} \\ \{101, 110\} \cdot \{00, 11\} &= \{10100, 10111, 11000, 11011\} \\ \{abc\}^* &= \{\lambda, abc, abcab, abcabcabc, abcababcabc, \dots\} \\ \{a, b\}^* &= \{\lambda, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, bab, bba, bbb, \dots\} \end{aligned}$$

E para definirmos o que é uma expressão regular, vamos partir das linguagens mais simples que existem e utilizar esses operadores. Temos, então, dado o alfabeto $\Sigma = \{a_1, a_2, \dots, a_n\}$:

- a_1, a_2, \dots, a_n são ERs que correspondem às linguagens $\{a_1\}, \{a_2\}, \dots, \{a_n\}$, respectivamente;
- ϵ é uma ER que corresponde à linguagem $\{\lambda\}$;
- \emptyset é uma ER que corresponde à linguagem $\{\}$;
- se e_1 e e_2 são ERs que correspondem às linguagens L_1 e L_2 , então $e_1 \cup e_2$ é uma expressão regular que corresponde à linguagem $L_1 \cup L_2$;
- se e_1 e e_2 são ERs que correspondem às linguagens L_1 e L_2 , então $e_1 \cdot e_2$ é uma expressão regular que corresponde à linguagem $L_1 \cdot L_2$;
- se e é uma ER que corresponde à linguagem L , então e^* é uma ER que corresponde à linguagem L^* .

Por exemplo, $(a \cup b \cup c \cup d)$ indica a união das linguagens $\{a\}, \{b\}, \{c\}, \{d\}$, ou seja, $\{a, b, c, d\}$. Para evitar confusão quando misturamos os operadores dentro de uma mesma expressão regular, devemos estabelecer uma prioridade entre os operadores. Temos, então, $*$ com a mais alta prioridade, depois \cdot e, finalmente, \cup . Podemos ainda utilizar parênteses para determinar a ordem de aplicação dos operadores. Vejamos alguns exemplos:

- $(a \cup b) \cdot (a \cup b \cup c)^* \cdot (b \cup c)$ define a linguagem formada por todas as cadeias sobre $\{a, b, c\}$ que se iniciam com as letras a ou b e terminam com b ou c ;
- $(a \cup b) \cdot ((a \cup b) \cdot (a \cup b) \cdot (a \cup b))^*$ define a linguagem $\{x \in \{a, b\}^* \mid |x| \bmod 3 = 1\}$.

Assim como os AFNDs, as ERs podem ser muito mais simples para representar a linguagem a ser reconhecida por um AL do que um AFD. Por exemplo, para a linguagem definida pelo AFD A_1 , poderíamos utilizar a seguinte ER:

$$(digito \cdot digito^*) \cup (i \cdot f) \cup (letra \cdot (letra \cup digito)^*)$$

Ou melhor que isso, poderíamos definir cada um dos tokens a serem reconhecidos por meio de uma ER. Por exemplo:

NUMERO: $digito \cdot digito^*$;

IF: $i \cdot f$;

IDENTIFICADOR: $letra \cdot (letra \cup digito)^*$.

Mas também como acontece com os AFNDs, é mais difícil implementar um programa que, dadas as ERs, consiga identificar cadeias que a elas correspondem. Por outro lado, é possível também transformar uma ER num AFD, a exemplo do que ocorre com os AFNDs. E melhor ainda, existem programas que já fazem isso. Um deles é o JavaCC, que utilizaremos a seguir para implementar o AL para a linguagem X^{++} . Esse programa permite que especifiquemos os tokens do nosso AL por meio de ERs e cria um programa Java capaz de ler e reconhecer esses tokens.

3.3 O analisador léxico da linguagem X^{++}

Nesta seção mostraremos como implementar o AL para a linguagem X^{++} usando o gerador de compiladores JavaCC. No Apêndice B é feita uma descrição de como funciona esse programa, incluindo alguns exemplos de sua utilização. Assim, recomenda-se que o leitor consulte esse apêndice antes de continuar a leitura desta seção.

O programa JavaCC aceita como entrada um arquivo que tem a descrição da linguagem a ser reconhecida e irá gerar um analisador sintático e um analisador léxico para essa linguagem. Assim, além da gramática da linguagem-alvo, deve-se descrever também qual é a linguagem, ou o conjunto de tokens a serem reconhecidos pelo AL. Esse arquivo pode ter qualquer nome, mas costuma-se utilizar uma extensão **.jj**. Vamos nos referir a esse arquivo como “arquivo **.jj**”.

O arquivo a ser tratado pelo JavaCC é dividido em diversas seções. A primeira delas determina alguns parâmetros para a geração do AL e do AS. Essa seção inicia-se com a palavra “options”, como mostrado a seguir. Cada um desses parâmetros possui um valor-padrão e só precisamos incluir na nossa definição aqueles parâmetros cujo valor desejamos alterar. No caso da linguagem X^{++} , iremos usar apenas a opção:

```
options {
    STATIC = false;
}
```

Essa opção afeta o modo que o AL e o AS são gerados. Ao gerar os analisadores, o JavaCC cria uma classe Java para o AL e uma para o AS. Se essa opção tiver o valor *true*, que é o valor-padrão, tanto a classe correspondente ao AS quanto ao AL serão classes cujos componentes (variáveis e métodos) são *static*. Como consequência, só um AS e um AL podem ser criados. Utilizando a opção *STATIC = false* tem-se maior flexibilidade, pois diversas cópias do AS (e, como consequência, do AL) podem ser criadas. Por outro lado, de acordo com a documentação do JavaCC, o desempenho de um compilador estático é superior ao de um não estático. Embora utilizemos apenas uma cópia do AS no nosso compilador, preferimos um não estático apenas para que o leitor possa ver quais são as implicações desta opção na implementação.

A segunda parte é delimitada pelas palavras *PARSER_BEGIN* e *PARSER_END* seguidas por um nome. Esse nome corresponde à classe que irá abrigar o AS. Por exemplo, um *PARSER_BEGIN(langX)* irá originar a uma classe chamada *langX*, que será a classe do AS, e um arquivo *langX.java* que contém essa classe. Todo código colocado nesse trecho do arquivo *.jj* será incluído nessa classe.

Além disso, será criada uma classe que corresponde ao AL do compilador. Essa classe (e o arquivo onde ela é armazenada) tem o nome do AS mais *TokenManager*. No nosso caso, por exemplo, a classe do AL seria chamada *langXTokenManager*. No jargão utilizado pelo JavaCC, o AL é chamado de Token Manager, por isso o nome da classe.

No programa 3.2, o código dentro dessa seção do arquivo *.jj* define o método *main* da classe *langX*. Ele é responsável por verificar quais são os parâmetros de entrada, criar o objeto do tipo *langX* e executar a análise sobre um arquivo cujo nome foi passado como parâmetro. Se o nome utilizado foi “-”, então a análise será feita sobre a entrada-padrão (*System.in*). Ele aceita também a opção *-short*, que modifica a saída produzida pelo compilador.

Programa 3.2

```

1  PARSER_BEGIN(langX)
2  package parser;
3  import java.io.*;
4
5
6  public class langX {
7      final static String Version = "X++ Compiler - Version 1.0 - 2004";
8      boolean Menosshort = false; // saída resumida = falso
9
10
11     // Define o método "main" da classe langX.
12     public static void main(String args[]) throws ParseException
13     {
14         String filename = ""; // nome do arquivo a ser analisado
15         langX parser;        // analisador léxico/sintático
16         int i;
```

```

17     boolean ms = false;
18
19     System.out.println(Version);
20     // lê os parâmetros passados para o compilador
21     for (i = 0; i < args.length - 1; i++)
22     {
23         if ( args[i].toLowerCase().equals("-short") )
24             ms = true;
25         else
26         {
27             System.out.println("Usage is: java langX [-short] inputfile");
28             System.exit(0);
29         }
30     }
31
32     if (args[i].equals("-"))
33     {      // lê da entrada-padrão
34         System.out.println("Reading from standard input . . .");
35         parser = new langX(System.in);
36     }
37     else
38     {      // lê do arquivo
39         filename = args[args.length-1];
40         System.out.println("Reading from file " + filename + " . . .");
41         try {
42             parser = new langX(new java.io.FileInputStream(filename));
43         }
44         catch (java.io.FileNotFoundException e) {
45             System.out.println("File " + filename + " not found.");
46             return;
47         }
48     }
49     parser.Menosshort = ms;
50     parser.program();    // chama o método que faz a análise
51
52     // verifica se houve erro léxico
53     if ( parser.token_source.foundLexError() != 0 )
54         System.out.println(parser.token_source.foundLexError() +
55                             " Lexical Errors found");
56     else
57         System.out.println("Program successfully analized.");
58 } // main
59
60 static public String im(int x) // método auxiliar
61 {
62     int k;
63     String s;
64     s = tokenImage[x];
65     k = s.lastIndexOf("\\");
66     try {s = s.substring(1,k);}
67     catch (StringIndexOutOfBoundsException e)
68     {}
69     return s;

```

```

70  }
71
72 } // langX
73
74 PARSER_END(langX)

```

O método estático *im* é utilizado apenas para relacionar um token reconhecido com o seu “nome”. Por exemplo, para um identificador, o nome será *IDENT*, para uma constante inteira, será *int_constant*, para a palavra reservada *class*, será *class*. Esses nomes são definidos no arquivo *.jj*, na seção que descreve os tokens a serem reconhecidos.

Em seguida, vem a seção demarcada pela palavra-chave *TOKEN_MGR_DECLS*. Essa seção serve para que possamos introduzir código Java na classe correspondente ao AL. No nosso AL, iremos introduzir uma variável não pública chamada *countLexError*. Essa variável é incrementada a cada erro léxico encontrado no arquivo sendo analisado. E introduzimos também um método que simplesmente retorna o valor dessa variável. Esse método será usado pelo AS, no final da análise para determinar se ocorreram erros léxicos ou não.

Programa 3.3

```

1 TOKEN_MGR_DECLS :
2 {
3     int countLexError = 0;
4
5     public int foundLexError()
6     {
7         return countLexError;
8     }
9
10 }

```

A seção seguinte é a mais importante para a definição do AL. Nela são definidos quais os tokens a serem reconhecidos, quais os caracteres a serem desprezados e outros comportamentos que o AL deva ter. Iniciaremos mencionando quais são os caracteres que devem ser desprezados antes de se iniciar o reconhecimento de um token, como fizemos quando utilizamos um AFD para definir o AL. Isso é feito da seguinte forma:

Programa 3.4

```

1 SKIP :
2 {
3     " "
4 | "\t"
5 | "\n"
6 | "\r"
7 | "\f"
8 }

```

Todas as definições nesta seção utilizam a representação de expressões regulares. A palavra *SKIP* indica ao JavaCC que desejamos definir quais são as cadeias que

devem ser ignoradas. No caso, estamos dizendo que um “branco” ou um *tab*, ou um final de linha, ou um *carriage return*, ou um *line feed*, devem ser ignorados e não entrar no reconhecimento de um token.

Além do *SKIP*, o JavaCC emprega também a palavra *TOKEN* que é utilizada para definir, por meio de expressões regulares, quais as cadeias a serem reconhecidas e quais os tipos de tokens que a elas correspondem. Vejamos o caso das palavras reservadas. Sua definição é bastante simples, pois a expressão regular que as define é a própria palavra. Temos:

Programa 3.5

```

1  /* Palavras reservadas */
2
3 TOKEN :
4 {
5     < BREAK: "break" >
6 | < CLASS: "class" >
7 | < CONSTRUCTOR: "constructor" >
8 | < ELSE: "else" >
9 | < EXTENDS: "extends" >
10 | < FOR: "for" >
11 | < IF: "if" >
12 | < INT: "int" >
13 | < NEW: "new" >
14 | < PRINT: "print" >
15 | < READ: "read" >
16 | < RETURN: "return" >
17 | < STRING: "string" >
18 | < SUPER: "super" >
19 }
```

Essa definição diz que os possíveis tokens a serem reconhecidos serão as cadeias *break*, *class*, *constructor* etc. A cada uma dessas cadeias é associado um tipo de token, que, no caso, chamamos de *BREAK*, *CLASS*, *CONSTRUCTOR* etc. Esses tipos de tokens irão corresponder, na implementação do AS, a constantes inteiras definidas no arquivo *langXConstants*, que é a superclasse imediata da classe *langX*. Assim, podemos nos referenciar a essas constantes como *langXConstants.BREAK* ou *langXConstants.CLASS*, ou *langXConstants.CONSTRUCTOR*.

Assim como descrito no início deste capítulo, o AL é construído de modo que a maior cadeia possível seja reconhecida. Por isso não há nenhum problema quanto a uma expressão regular poder gerar subcadeias de outra expressão regular. Sempre será considerada a maior cadeia da entrada que casar com alguma expressão regular. Isso acontece, por exemplo, a seguir com os tokens *GT* e *GE*. Se a entrada possuir um *string* \geq , este será identificado como um *GE*, mas se possuir um $>$ apenas, então *GT* será o casamento realizado.

Programa 3.6

```

1  /* Operadores */
2
3 TOKEN :
4 {
5     < ASSIGN: "=" >
6     | < GT: ">" >
7     | < LT: "<" >
8     | < EQ: "==" >
9     | < LE: "<=" >
10    | < GE: ">=" >
11    | < NEQ: "!=" >
12    | < PLUS: "+" >
13    | < MINUS: "-" >
14    | < STAR: "*" >
15    | < SLASH: "/" >
16    | < REM: "%" >
17 }
18
19 /* Símbolos especiais */
20
21 TOKEN :
22 {
23     < LPAREN: "(" >
24     | < RPAREN: ")" >
25     | < LBRACE: "{" >
26     | < RBRACE: "}" >
27     | < LBRACKET: "[" >
28     | < RBRACKET: "]" >
29     | < SEMICOLON: ";" >
30     | < COMMA: "," >
31     | < DOT: "." >
32 }
```

Note que o arquivo .jj pode possuir diversas seções *SKIP* ou *TOKEN*. Aqui dividimos conforme nossa conveniência, de acordo com a similaridade entre os tokens. A seguir descreveremos os tokens relacionados a constantes e nomes de identificadores.

Programa 3.7

```

1  /* constantes */
2
3 TOKEN :
4 {
5     < int_constant:( // números decimais, octais, hexadecimais ou binários
6         ("0"- "9"] ([ "0"- "9"])*) |
7         ("0"- "7"] ([ "0"- "7"])*) [ "o", "0" ] ) |
8         ("0"- "9"] ([ "0"- "7", "A"- "F", "a"- "f"])* [ "h", "H" ] ) |
9         ("0"- "1"] ([ "0"- "1"])*) [ "b", "B" ] )
10    ) >
11
12    | < string_constant: // constante string como "abcd bcda"
13        "\\" ( [ "\\", "\n", "\r"])* "\\" >
```

```

14  |
15  < null_constant: "null" > // constante null
16  }
17
18 /* Identificadores */
19
20 TOKEN :
21 {
22   < IDENT: <LETTER> (<LETTER>|<DIGIT>)* >
23   |
24   < #LETTER: ["A"-"Z", "a"-"z"] >
25   |
26   < #DIGIT: ["0"-"9"] >
27 }

```

Uma constante inteira é um número decimal, ou um número octal seguido pela letra *o* ou *O*, ou um número hexadecimal seguido por *h* ou *H*, ou, ainda, um número binário seguido por *b* ou *B*, todos sempre iniciados por um dígito (constantes hexadecimais não podem começar com *A-F* ou *a-f*). Aqui, novamente, aplica-se a regra de que o AL deve identificar como token a maior cadeia possível. Assim, por exemplo, vejamos alguns tokens “estranhos” que podem ser reconhecidos por essa definição:

123afhoje deve ser reconhecido como uma constante inteira *123afh* seguida de um identificador *oje*;

ObaCh também é uma constante hexadecimal;

76200O é uma constante octal seguida por um identificador *O*;

1011tb10b é uma constante decimal *1011* seguida pelo identificador *tb10b*.

Uma constante string, segundo a definição dada, inicia-se com um abre aspas que vem seguido por qualquer caractere, exceto um fim de linha (*\n* e *\r* são constantes reconhecidas pelo JavaCC), ou um fecha aspas, que serve para finalizar a constante. E a palavra *null* também é reconhecida como uma constante, no caso, uma referência a um objeto.

Os identificadores são definido como sendo iniciados por uma letra, seguida por letras ou dígitos. Foram utilizados dois tokens *#LETTER* e *#DIGIT* para definir *IDENT*. Esses tokens não são utilizados na gramática da linguagem *X⁺⁺*, mas servem como auxiliares na definição do próprio AL. Note também que palavras como *for*, *class*, etc, casam com a definição de identificadores. Ou seja, quando uma dessas palavras aparece na entrada, ela casa tanto com a definição mostrada anteriormente de palavra reservada quanto com a definição de identificador, gerando uma dupla interpretação. Esse é o caso em que precisamos definir uma prioridade e indicar qual das opções deve ser assumida. O JavaCC utiliza uma regra bastante simples: a definição que aparecer primeiro no arquivo *.jj* será utilizada no caso de dupla interpretação.

Quando há o casamento de uma cadeia da entrada com um dos tokens definidos no arquivo *.jj*, o AL reconhece essa cadeia como um token e produz um objeto do tipo *Token*, que é uma classe Java definida com as seguintes variáveis:

int kind; Contém o tipo do token reconhecido. Cada um dos tokens descritos no arquivo .jj como *IF* ou *IDENT* é definido na classe *langXConstants* como sendo uma constante inteira. Assim, supondo que *langXConstants.IDENT* foi definido com o valor 9, então ao reconhecer um identificador, o AL irá produzir um objeto Token cuja variável *kind* tem o valor 9;

int beginLine, beginColumn, endLine, endColumn; Essas variáveis indicam, respectivamente, a linha e a coluna dentro do arquivo de entrada, onde se inicia e onde termina o token reconhecido;

String image; É a cadeia que foi lida e reconhecida como token. Por exemplo, se a cadeia *func10* foi lida na entrada e reconhecida como um *IDENT*, então essa variável contém a cadeia lida, ou seja, *func10*;

Token next; Uma referência para o próximo token reconhecido após ele. Se o AL ainda não leu nenhum outro token ou se esse é o último token da entrada, então seu valor é *null*;

Token specialToken; É um apontador para o último token especial reconhecido antes deste. Veja mais adiante os comentários sobre o que são os tokens especiais.

Esses objetos são enviados para o AS, cada vez que ele necessita de um novo token para analisar sintaticamente a entrada. Os campos como linha e coluna em que aparecem os tokens são úteis para que se possa emitir mensagens no caso de erros sintáticos. Com eles, o AS pode indicar em que ponto do arquivo de entrada aparece o token que viola as regras sintáticas da linguagem.

3.4 Comentários

Também é uma das atribuições do AL reconhecer os comentários do programa e tomar a atitude correta em relação a eles, que pode ser, por exemplo, simplesmente ignorá-los. Note que um comentário não é um item léxico, ou seja, não deve ser enviado para o AS, uma vez que a gramática da linguagem nem sequer faz referência aos comentários. E nem poderia, pois na maioria das linguagens, até mesmo a nossa, um comentário pode aparecer em qualquer ponto do programa, como, por exemplo, no meio de um comando ou expressão:

```
a = b.myMethod(10, /* esse é um comentário */ c) + 2;
```

Na linguagem *X⁺⁺* utilizamos dois tipos de comentários. O primeiro é o comentário de uma única linha que se inicia com // e vai até o final da linha. E o segundo é o comentário que pode conter diversas linhas e que se inicia com /* e termina com */. Ambos os tipos são também encontrados em Java. Ao encontrar esses comentários, o nosso AL irá simplesmente ignorá-los.

Para implementar essa funcionalidade no AL, iremos utilizar uma outra característica do JavaCC que é o conceito de estado. A idéia é que ao encontrar uma cadeia que inicia um comentário, por exemplo um /*, o AL passa a operar em um estado

diferente. Nesse estado, as cadeias lidas não têm o mesmo significado que no estado normal do AL. Elas não são reconhecidas como tokens, mas, sim, são lidas e descartadas, até que se chegue à cadeia que fecha o comentário. Quando isso acontece, o AL é colocado de novo no seu estado normal, onde reconhece os tokens especificados.

Para cada estado que utilizarmos, devemos atribuir um nome. O estado “normal” do AL tem um nome predefinido que é *DEFAULT*. Vamos, então, alterar o nosso AL, fazendo com que, ao achar um /*, o AL mude para um estado chamado *multilinecomment*. Isso é feito com a seguinte definição:

Programa 3.8

```

1 SKIP :
2 {
3   "/*" : multilinecomment
4 }
```

Note que utilizamos um *SKIP* no estado *DEFAULT*, pois queremos que o início do comentário seja ignorado e que o AL passe para o estado *multilinecomment*. Porém, essa mudança de estado pode ser feita em qualquer casamento, por exemplo, na definição de um token como

TOKEN:
{
 < WHILE: "while" > : WHILEMODE
}

Nesse caso, se o AL estiver no estado *DEFAULT* e encontrar a cadeia *while*, esta é reconhecida como um token *WHILE* e o AL passa para o estado *WHILEMODE*.

No caso do nosso comentário, falta especificar o que o AL deve fazer no estado *multilinecomment*. Queremos que tudo que for lido nesse estado seja jogado fora pelo AL, até que seja encontrado um */ que termina o comentário e coloca o AL de volta no estado *DEFAULT*. Fazemos isso com:

Programa 3.9

```

1 <multilinecomment> SKIP:
2 {
3   "*/" : DEFAULT
4   | <~[]>
5 }
```

Definimos, nesse estado, o que deve ser ignorado, por meio de um *SKIP*. Poderíamos também definir um *TOKEN* em qualquer estado, como fazemos no estado *DEFAULT*. Queremos que o */ faça o AL voltar ao estado *DEFAULT* e que qualquer outra cadeia seja ignorada. A definição ~[] é utilizada para fazer o casamento com qualquer caractere que não esteja entre os colchetes. Por exemplo, ~[0-9] irá casar com qualquer caractere da entrada que não seja um dígito e (~["0"-"9"])* irá casar com qualquer cadeia que não possua dígitos. É importante notar que, na nossa

definição do comentário, as cadeias * e / também formam um casamento com o padrão ~[]. Porém, vale a regra de que sempre a maior cadeia possível é utilizada no casamento. Como o segundo padrão tem apenas um caractere, então ao aparecer a cadeia */, o casamento é sempre feito no primeiro padrão.

A definição de comentários de uma linha só é feita de maneira semelhante:

Programa 3.10

```

1 SKIP :
2 {
3     "//" : singlelinecomment
4 }
5 <singlelinecomment> SKIP:
6 {
7     <"\n", "\r"> : DEFAULT
8 |   <~[]>
9 }
```

Note que, nesse caso, um final de linha volta o AL ao estado *DEFAULT*. Aqui vale a regra de prioridade para os padrões que são definidos antes. Assim, as cadeias \n ou \r na entrada poderiam casar tanto com o primeiro quanto com o segundo padrão, mas, pela prioridade, o casamento é feito com o primeiro.

3.5 Recuperação de erros léxicos

Um erro léxico ocorre quando alguma cadeia que aparece na entrada não pode ser reconhecida como um token válido. Por exemplo, se tivermos na entrada a cadeia @@@##@@, veremos que o AL que definimos para a linguagem X^{++} irá apontar um erro. Nesse caso, o AL gerado pelo JavaCC toma uma atitude radical, que é lançar um erro do tipo *TokenMgrError*. Como não podemos tratar esse erro, o que ocorre é que ele é propagado até os níveis mais altos do compilador, que terá um término anormal. Esse comportamento muitas vezes não é adequado. Gostaríamos, por exemplo, de que o AL emitisse uma mensagem de erro, passasse por cima da cadeia inválida e continuasse com a análise.

Para isso, devemos evitar que o AL chegue a identificar um erro. Devemos especificar casamentos também para as cadeias que são inválidas e devemos fazer o tratamento de erro desejado para essas cadeias. Vamos, então, tentar identificar quais são os casos em que uma cadeia pode provocar um erro léxico.

A primeira delas é quando tentamos iniciar o reconhecimento de um token e deparamo-nos com um caractere que não casa com nenhuma das expressões de casamento definidas. Devemos, então, passar sobre os caracteres da entrada até que encontremos um que possa ser utilizado num casamento. Por exemplo, se tivermos a cadeia 1234@@@\#\#\#abcd" na entrada, o AL deve reconhecer o token 1234 e parar ao encontrar o primeiro @. Na próxima execução do AL, temos um erro, pois @ não inicia nenhum padrão de casamento. Devemos, então, pular toda a cadeia até que um caractere válido seja encontrado, no caso a aspa. A próxima execução retorna o token "abcd".

Para identificar esse tipo de erro faremos o casamento de todas as cadeias formadas por caracteres que não iniciam nenhuma expressão de casamento para cadeias válidas. Para isso, olhamos as expressões usadas na nossa definição e vemos que os tokens e SKIPs definidos anteriormente se iniciam sempre por:

- uma letra;
- um dígito;
- uma aspa;
- caracteres especiais como ([)] ; < > = ;
- um espaço ou delimitador de linha.

Assim, definiremos uma expressão que case com as cadeias inválidas. Podemos definir, então, o token que chamamos *INVALID_LEXICAL* como:

Programa 3.11

```

1 <INVALID_LEXICAL:
2 (~ ["a"-"z", "A"-"Z",
3   "0"-"9",
4   "\",
5   "(",
6   ")",
7   "[",
8   "]",
9   "{",
10  "}",
11  ";",
12  ",",
13  ".",
14  "=",
15  ">",
16  "<",
17  "!",
18  "+",
19  "-",
20  "*",
21  "/",
22  "%",
23  " ",
24  "\t",
25  "\n",
26  "\r",
27  "\f"
28 ]) +>

```

O operador *+* utilizado nessa expressão representa repetição pelo menos uma vez. E utilizamos o operador *~* para indicar que o casamento é feito quando o caractere da entrada não está no conjunto definido entre os colchetes. Há, ainda, um problema para ser resolvido. É que o token *INVALID_LEXICAL* será retornado normalmente para

o AS que terá que tratá-lo, provavelmente indicando um erro sintático, já que esse token não aparece na gramática. O que queremos é que o AL trate esse erro emitindo uma mensagem adequada e que a cadeia inválida seja ignorada, não chegando ao AS.

Para isso, definiremos *INVALID_LEXICAL* não como um *TOKEN*, mas, sim, como um *SPECIAL_TOKEN*. O AL gerado pelo JavaCC irá tratar esse tipo de token de maneira especial. Ele não é passado para o AS, mas é armazenado e irá aparecer na variável *specialToken* do próximo token. Por exemplo, no caso da cadeia "@@@##@\"abcd", o AL iria reconhecer um token especial @@@##@ que não é passado ao AS e, depois, o token "abcd" que tem uma referência ao token especial na sua variável *specialToken*.

Além disso, precisamos fazer com que o AL emita uma mensagem de erro e que controle o número total de erros léxicos encontrados. Para isso, o JavaCC permite que a cada expressão de casamento associemos um pedaço de código Java que é executado cada vez que ocorre um casamento. Teremos, então, a seguinte definição, que completa nosso AL:

Programa 3.12

```

1  /* Trata os erros léxicos */
2  SPECIAL_TOKEN :
3  {
4    <INVALID_LEXICAL:
5    (~ ["a"-"z", "A"-"Z",
6        "0"-"9",
7        "\",
8        "(",
9        ")",
10       "[" ,
11       "]",
12       "{",
13       "}",
14       ";",
15       ",",
16       ".",
17       "!=",
18       ">",
19       "<",
20       "!",
21       "+",
22       "-",
23       "*",
24       "/",
25       "%",
26       " ",
27       "\t",
28       "\n",
29       "\r",
30       "\f"
31   ]) +>
32   {
33     System.err.println("Line " + input_stream.getEndLine() +

```

```

34                     " - Invalid string found: " + image);
35             countLexError++;
36         }
37     |
38     <INVALID_CONST:
39     "\"\" ( ~ ["\n","\\r","\\"])* ["\n","\\r"]>
40     {
41         System.err.println("Line " + input_stream.getEndLine() +
42                         " - String constant has a \\n: " + image);
43         countLexError++;
44     }
45
46 }
```

O código associado ao token especial mostra a mensagem de erro e incrementa a variável *countLexError*, que foi definida na classe *langXTokenMgr*, por meio da seção *TOKEN_MGR_DECLS*, mostrada anteriormente.

A definição do token especial incorpora também o tratamento de um segundo tipo de erro léxico, identificado pelo token especial *INVALID_CONST*. Ele ocorre quando uma constante string é iniciada (uma cadeia iniciada por aspa), mas não termina antes do final da linha. Nesse caso, tem-se um erro léxico, pois não existe casamento válido possível. Esse casamento é definido, então, no token especial como sendo qualquer cadeia iniciada por aspa e terminada por um fim de linha.

3.6 Arquivos-fonte do compilador

Na página de downloads da editora (<http://www.novateceditora.com.br/downloads.php>), o leitor encontrará todos os arquivos-fonte relativos à implementação do compilador para X++. No final de cada capítulo, uma seção como esta descreve quais são os arquivos relativos àquele capítulo e como devem ser usados.

No subdiretório *cap03/parser*, encontra-se o arquivo *langX++.jj*, que contém a definição do AL estudado neste capítulo. Para gerar a primeira versão do nosso compilador, o leitor deve executar os seguintes comandos:

```

cap03/parser$ javacc langX++.jj
Java Compiler Compiler Version 2.1 (Parser Generator)
Copyright (c) 1996-2001 Sun Microsystems, Inc.
Copyright (c) 1997-2001 WebGain, Inc.
(type "javacc" with no arguments for help)
Reading from file langX++.jj . .
File "TokenMgrError.java" does not exist. Will create one.
File "ParseException.java" does not exist. Will create one.
File "Token.java" does not exist. Will create one.
File "SimpleCharStream.java" does not exist. Will create one.
Parser generated successfully.
```

```
cap03$ javac parser/langX.java
```

O primeiro processa o arquivo com a definição do nosso AL e gera os arquivos com o código Java do AS e do AL, além de alguns outros arquivos auxiliares. O segundo comando compila a classe *langX* e todas as classes que ela utiliza. Como resultado, obtém-se o arquivo *langX.class* que pode ser executado utilizando uma máquina virtual Java.

Essa primeira versão do compilador somente lê a entrada, identifica os tokens e mostra-os na saída-padrão. Ele aceita a opção *-short*, que reduz a quantidade de informação mostrada. O leitor pode experimentar o programa, utilizando o programa *bintree.x*, colocado no diretório *ssamples* da página de downloads ou, ainda, com o programa *bintree-erro-lexico.x* que contém alguns erros léxicos:

```
cap03$ java parser.langX -short ..//ssample/bintree.x
X++ Compiler - Version 1.0 - 2004
Reading from file ..//ssamples/bintree.x . .
class class 13
bintree <IDENT> 29
{ { 34
class class 13
data <IDENT> 29
{ { 34
int int 19
dia <IDENT> 29
, , 39
mes <IDENT> 29
, , 39
ano <IDENT> 29
...
treeprint <IDENT> 29
( ( 32
0 <int_constant> 26
) ) 33
; ; 38
return return 23
0 <int_constant> 26
; ; 38
} } 35
} } 35
<EOF> 0
Program successfully analyzed.

cap03$ java parser.langX ..//ssample/bintree-erro-lexico.x > out
Line 9 - Invalid string found: #
Line 13 - Invalid string found: ####
Line 103 - String constant has a \n: "Elemento ja existe ;
```

Capítulo 4

Análise Sintática

Neste capítulo começaremos a estudar o “coração” do compilador, que é o analisador sintático. É ele quem analisa o programa fonte e verifica se este pertence ou não à linguagem desejada. É também a base para as demais fases do compilador, como a análise semântica e a geração de código, mediante a construção da árvore sintática. Neste capítulo veremos somente os aspectos relacionados à análise sintática propriamente dita. A recuperação de erros e a construção da árvore sintática serão tratados nos capítulos seguintes.

Existem diversas técnicas para se implementar um analisador sintático. Vamos aqui nos concentrar numa técnica chamada de descendente recursiva. Essa técnica é bastante simples e flexível. É também eficiente no sentido de que um analisador sintático construído usando essa técnica tende a ser rápido ao analisar programas. Inicialmente iremos estudar com mais cuidado o que é a análise descendente recursiva. Depois veremos como implementar um analisador sintático descendente recursivo utilizando o JavaCC.

4.1 Análise sintática descendente recursiva

Dada uma GLC, a análise sintática baseada nessa GLC pode ser ascendente ou descendente. Um analisador ascendente agrupa os símbolos da entrada para formar os símbolos não terminais mais distantes do símbolo inicial da GLC ou que estão em níveis inferiores na estrutura da linguagem. Já na análise descendente, parte-se do símbolo inicial da GLC e busca-se identificar na entrada as construções que correspondem às produções da GLC.

Um analisador descendente recursivo utiliza essa última estratégia. Ele é chamado de recursivo, pois a cada símbolo não terminal da GLC corresponde uma função (ou procedimento, ou método) recursiva, responsável por verificar se a entrada “casa” com a estrutura daquele não-terminal.

Vamos tomar como exemplo a produção vista no capítulo 2:

$$\langle \text{classdecl} \rangle \rightarrow \text{"class" } \text{"ident" } [\text{"extends" } \text{"ident" }] \langle \text{classbody} \rangle$$

Uma entrada que case com essa produção deve iniciar-se com a cadeia *class*, que vem seguida de um identificador, pode ter um *extends* e um identificador e termina com o corpo da classe, representado na produção pelo não-terminal *classbody*. No analisador descendente recursivo, o método correspondente a essa produção deveria então:

- verificar se o próximo token da entrada é *class*. Se for, deve continuar com a análise. Se não for, um erro sintático ocorreu, pois a entrada não corresponde à produção esperada;
- verificar se o próximo token é *ident*. Se for, deve continuar com a análise. Se não for, um erro sintático ocorreu, pois a entrada não corresponde à produção esperada;
- verificar se o próximo token é *extends*. Se não for, deve continuar a análise, pois essa parte da produção é opcional. Se for, deve verificar se o token seguinte é um *ident*. Se for, deve continuar com a análise. Se não for, um erro sintático ocorreu, pois a entrada não corresponde à produção esperada;
- verificar se o resto da entrada corresponde ao não-terminal *classbody*. Isso não é feito diretamente pelo método, mas, sim, invocando o método correspondente àquele não-terminal.

A cada vez que o método necessita de um token da entrada, ele invoca o AL, que lhe fornece o token. Teríamos algo semelhante ao programa 4.1.

Programa 4.1

```

1 void classdecl()
2 {
3     if ( curToken.type == CLASS) // token corrente é "class"
4         analex();           // chama AL. curToken recebe novo token
5     else
6         SintaticError();    // ocorreu erro sintático
7     if ( curToken.type == IDENT) // token corrente é identificador
8         analex();
9     else
10        SintaticError();
11     if ( curToken.type == EXTENDS) // token corrente é "extends"
12     {
13         analex();
14         if ( curToken.type == IDENT) // token é identificador
15             analex();
16         else
17             SintaticError();
18     }
19     classbody();
20 }
```

A variável *curToken* contém sempre o último token devolvido pelo AL (*analex*). Ao se invocar o método *classdecl*, *curToken* já possui o token correspondente ao primeiro símbolo correspondente a esse não-terminal, no caso, o token que deve corresponder à palavra “*class*”. Essa política é mantida ao chamar-se o método *classbody*, pois *curToken* já possui o token que deve casar com o início da produção daquele não terminal.

É importante notar que o token lido na entrada serve para indicar qual o caminho a ser seguido no reconhecimento dos próximos tokens. Uma vez que um token foi lido da entrada e casado com algum símbolo de uma produção, não há como retroceder, caso algum token posterior não combine com a produção escolhida. Por exemplo, na produção

$$\langle \text{classdecl} \rangle \rightarrow \text{“class” “ident” } \langle \text{classbody} \rangle \mid \text{“class” “ident” “extends” “ident” } \langle \text{classbody} \rangle$$

se utilizarmos o token de entrada *class* para fazer o casamento com a primeira produção, então teremos escolhido seguir essa produção, e entradas que possuam *extends ident* não poderão ser reconhecidas. E analogamente, se escolhermos a segunda produção, não serão aceitas cadeias sem *extends ident*. Portanto, precisamos ter cuidado com as produções que utilizamos na construção de um analisador descendente recursivo.

As gramáticas que podem ser utilizadas na construção de um analisador descendente recursivo são chamadas de LL(1). O primeiro L significa left to right e indica que a leitura da entrada é feita da esquerda para a direita. O segundo L significa left linear e indica que sempre o símbolo não terminal que estiver mais à esquerda será reconhecido primeiro. E o número 1 indica que o número de símbolos de lookahead necessários é 1, ou seja, que se conhecendo apenas um símbolo da entrada é possível decidir que produção utilizar. Na produção citada seriam necessários três símbolos para se decidir qual produção utilizar.

Para auxiliar na construção de um analisador descendente recursivo, costuma-se construir uma tabela preditiva. Essa tabela possui uma linha para cada não-terminal da GLC e uma coluna para cada terminal. Se tivermos o não-terminal *A* e o terminal *a*, na posição *A* × *a* devemos colocar qual é a produção que devemos utilizar se, ao tentarmos reconhecer *A*, encontrarmos na entrada o símbolo *a*. Por exemplo, se tivermos

$$\begin{aligned} S &\rightarrow aSAb \mid bAa \\ A &\rightarrow bAb \mid c \end{aligned}$$

teremos a seguinte tabela preditiva:

Não-terminal	Terminais		
	<i>a</i>	<i>b</i>	<i>c</i>
<i>S</i>	<i>S</i> → <i>aSAb</i>	<i>S</i> → <i>bAa</i>	
<i>A</i>	<i>A</i> →	<i>S</i> → <i>bAb</i>	<i>S</i> → <i>c</i>

As posições vazias na tabela indicam que esse terminal não pode aparecer no início do não-terminal. Se tivermos mais do que uma produção em alguma posição da tabela, então a GLC não pode ser usada na construção de um analisador descendente recursivo com um único símbolo de lookahead. Muitas vezes, porém, podemos alterar a GLC de forma a eliminar esses problemas, sem alterar a linguagem gerada.

Um caso típico de problemas ao construir a tabela preditiva é relativo à a existência de recursão à esquerda. Isso acontece quando, para algum não-terminal B , temos $B \xrightarrow{*} Ba$, mediante a aplicação de uma ou mais produções. Isso significa que a partir de B podemos gerar uma forma sentencial em que o próprio B aparece como primeiro símbolo. Intuitivamente é fácil verificar que isso causa problemas. Ao tentar reconhecer B , o AS descendente recursivo invoca o método correspondente ao não-terminal. Olhando para o token da entrada, esse método pode escolher um caminho que leve ao aparecimento de outro B , sem que nenhum token tenha sido consumido – o que caracteriza a recursão à esquerda. Isso faz com que o mesmo caminho seja seguido, o que vai levar novamente a outro B , e assim por diante. Isso faz com que o AS permaneça numa recursão infinita, sem consumir nenhum token.

A recursão à esquerda pode aparecer direta ou indiretamente. Ela é direta quando temos $B \Rightarrow Ba$ e, indireta, quando $B \Rightarrow A\alpha \xrightarrow{*} B\beta$. A eliminação da recursão indireta pode ser bastante complicada e não será tratada aqui. A eliminação da recursão direta, por outro lado, é simples e pode ser resolvida da seguinte maneira:

- dividem-se as produções do não-terminal B em dois subconjuntos $N = \{\alpha_1, \dots, \alpha_n\}$, das produções que não possuem recursão à esquerda e $R = \{B\beta_1, \dots, B\beta_m\}$, das produções com recursão à esquerda;
- eliminam-se as produções de R ;
- adicionam-se as produções $B \rightarrow \alpha_1 B' | \dots | \alpha_n B'$, onde B' é um novo símbolo não terminal, que não pertence à gramática;
- adicionam-se as seguintes produções para B' : $B' \rightarrow \beta_1 | \dots | \beta_m | \beta_1 B' | \dots | \beta_m B'$.

Com esse procedimento, trocam-se as recursões à esquerda por recursões à direita, que não são problemáticas para a análise descendente recursiva. Um exemplo clássico de eliminação de recursão à esquerda é para a gramática:

$$\begin{array}{ll}
 \text{expression} & \rightarrow \text{expression + term} \mid \\
 & \text{expression - term} \mid \\
 & \text{term} \\
 \text{term} & \rightarrow \text{term * factor} \mid \\
 & \text{term/factor} \mid \\
 \text{factor} & \rightarrow \text{ident} \mid \text{constant} \mid (\text{expression})
 \end{array}$$

Aplicando a eliminação descrita, teríamos:

$$\begin{array}{ll}
 \text{expression} & \rightarrow \text{term} \mid \text{term expression}' \\
 \text{expression}' & \rightarrow -\text{term} \mid +\text{term} \mid -\text{term expression}' \mid +\text{term expression}' \\
 \text{term} & \rightarrow \text{factor} \mid \text{factor term}' \\
 \text{term}' & \rightarrow *fator \mid /fator \mid *fator term' \mid /fator term' \\
 \text{factor} & \rightarrow \text{ident} \mid \text{constant} \mid (\text{expression})
 \end{array}$$

Outro modo de modificar uma GLC para que possa ser usada num analisador descendente recursivo é fazendo a fatoração à esquerda daquelas produções de um mesmo não-terminal que têm um prefixo em comum. É o caso do exemplo visto anteriormente:

$$\langle \text{classdecl} \rangle \rightarrow \text{"class" "ident" } \langle \text{classbody} \rangle \mid \text{"class" "ident" "extends" "ident" } \langle \text{classbody} \rangle$$

Se fatorarmos à esquerda, teremos a produção que usamos na GLC da linguagem X^{++} :

$$\langle \text{classdecl} \rangle \rightarrow \text{"class" "ident" ["extends" "ident"] } \langle \text{classbody} \rangle$$

Ou na GLC das expressões, teríamos:

$$\begin{array}{lcl} \text{expression} & \rightarrow & \text{term expression}' \\ \text{expression}' & \rightarrow & -\text{term [expression']} \mid +\text{term [expression']} \mid \lambda \\ \text{term} & \rightarrow & \text{factor term}' \\ \text{term}' & \rightarrow & *\text{factor [term']} \mid / \text{factor [term']} \mid \lambda \\ \text{factor} & \rightarrow & \text{ident} \mid \text{constant} \mid (\text{expression}) \end{array}$$

À primeira vista, pode parecer fácil calcular a tabela preditiva de uma GLC. Basta olhar quais são os tokens que iniciam as produções do não-terminal B e colocar na linha de B , na coluna desses tokens, as produções que eles iniciam. Porém, existem diversas situações que podem complicar essa tarefa. Vejamos alguns desses casos:

- se temos $B \rightarrow A\alpha \mid C\beta$, então é preciso saber quais são os tokens que iniciam as produções de A e C para decidir qual produção utilizar. E essa situação pode se propagar, uma vez que A e C podem também se iniciar com não-terminais;
- se temos $B \rightarrow A_1A_2\dots A_n a \alpha$, precisamos saber se A_1, \dots, A_n podem gerar a cadeia vazia. Se isso acontecer, então o token a também deve ser utilizado para decidir qual produção de B utilizar;
- se temos $A \rightarrow aBb; B \rightarrow \alpha$ e $\alpha \xrightarrow{*} \lambda$, então b deve ser considerado como um token que deve ser utilizado para decidir pela produção $B \rightarrow \alpha$ ao tentar reconhecer B .

Vamos, então, definir exatamente como a tabela preditiva deve ser construída. Primeiro, adicionamos um novo símbolo terminal à GLC. Esse terminal denotado por $\$$ aparece sempre no final da cadeia de entrada. A seguir, vamos associar à GLC duas funções, $FIRST$ e $FOLLOW$.

Seja α uma forma sentencial da gramática. Então $FIRST(\alpha)$ é definido como o conjunto de todos os símbolos terminais a tal que $\alpha \xrightarrow{*} a\beta$, ou seja, o conjunto de terminais que iniciam alguma cadeia derivada a partir de α .

Para calcular o $FIRST$ de um string, devemos seguir as seguintes regras:

- para o terminal a , $FIRST(a) = \{a\}$;
- para o não-terminal B , tal que $B \rightarrow A_1 A_2 \dots A_n$, onde A_i são símbolos terminais ou não terminais, fazemos:
 - inicialmente $FIRST(B) = \{\}$
 - repetimos $FIRST(B) = FIRST(B) \cup FIRST(A_i)$, para $i = 1, 2, \dots$ até que encontremos algum i tal que A_i não deriva λ ;
- para um string $\alpha = A_1 A_2 \dots A_n$, onde A_i são símbolos terminais ou não terminais, fazemos:
 - repetimos $FIRST(\alpha) = FIRST(\alpha) \cup FIRST(A_i)$, para $i = 1, 2, \dots$ até que encontremos algum i tal que A_i não deriva λ .

Na GLC fatorada que mostramos anteriormente, teríamos os seguintes conjuntos, para os terminais e não-terminais:

$$FIRST(\$) = \{\$\}$$

$$FIRST(+) = \{+\}$$

$$FIRST(-) = \{-\}$$

$$FIRST(*) = \{*\}$$

$$FIRST(/) = \{/ \}$$

$$FIRST(ident) = \{ident\}$$

$$FIRST(constant) = \{constant\}$$

$$FIRST(() = \{()\}$$

$$FIRST(factor) = \{ident, constant, (\}$$

$$FIRST(term') = \{*, /\}$$

$$FIRST(term) = \{ident, constant, (\}$$

$$FIRST(expression') = \{+, -\}$$

$$FIRST(expression) = \{ident, constant, (\}$$

Seja B um não-terminal da gramática e S o seu símbolo inicial. O conjunto $FOLLOW(B)$ é formado pelos terminais a tal que $S \xrightarrow{*} \alpha Ba\beta$, ou seja, existe uma forma sentencial derivável a partir do símbolo inicial em que a aparece imediatamente à direita de B .

A importância do $FOLLOW$ pode não ser tão intuitiva como a do $FIRST$, mas se tivermos um símbolo não terminal B que possui uma produção do tipo $B \rightarrow \alpha$, e $\alpha \xrightarrow{*} \lambda$, então é importante que saibamos quais são os símbolos de $FOLLOW(B)$. Vamos tomar, por exemplo, o terminal a e supor que $a \in FOLLOW(B)$. Isso significa

que no analisador descendente recursivo, ao executarmos o método correspondente a B , iremos consumir um certo número de tokens e, se esses tokens casarem com a definição de B , podemos ter um a na entrada. Porém se $B \xrightarrow{*} \lambda$, isso significa que não consumindo nenhum token da entrada, a pode aparecer na entrada e será casado em algum ponto, após o reconhecimento (nulo) de B . Portanto, se no início do reconhecimento de B tivermos um a na entrada, devemos seguir a produção que faz com que B derive a cadeia vazia. Se tivermos outra produção como $B \rightarrow A\alpha$ e $a \in FIRST(A)$, então ocorrerá um problema na GLC, pois com o mesmo terminal temos duas produções a aplicar.

Para calcular os conjuntos $FOLLOW$ para os não-terminais da GLC, devemos:

- adicionar o indicador de fim de cadeia $\$$ ao conjunto $FOLLOW(S)$, onde S é o símbolo inicial da GLC;
- dada a produção $B \rightarrow \alpha A \beta$, fazemos:

$$FOLLOW(A) = FOLLOW(A) \cup FIRST(\beta)$$

se $\beta \xrightarrow{*} \lambda$, então fazemos $FOLLOW(A) = FOLLOW(A) \cup FOLLOW(B)$.

Calculando o $FOLLOW$ para o nosso exemplo, teríamos:

$$FOLLOW(expression) = \{\$\}, \{\}\}$$

$$FOLLOW(expression') = \{\$\}, \{\}\}$$

$$FOLLOW(term) = \{+, -, \$, \{\}\}$$

$$FOLLOW(term') = \{+, -, \$, \{\}\}$$

$$FOLLOW(factor) = \{*, /, +, -, \$, \{\}\}$$

Uma vez calculados os conjuntos $FIRST$ e $FOLLOW$, podemos, então, construir a tabela preditiva da GLC. Para isso, vamos examinar as produções da linguagem e tentar descobrir em que posições colocá-las na tabela. Devemos fazer:

- dada a produção $B \rightarrow \alpha$, para todo terminal $a \in FIRST(\alpha)$, devemos colocar essa produção na posição (B, a) da tabela;
- dada a produção $B \rightarrow \alpha$, onde $\alpha \xrightarrow{*} \lambda$, para todo terminal $a \in FOLLOW(B)$, devemos colocar essa produção na posição (B, a) da tabela.

Seguindo essas regras, construímos, então, a tabela preditiva para a GLC de expressões, dada na tabela 4.1. A tabela preditiva pode ser usada, além da verificação da gramática, para construir o AS. Para implementar o método de um não-terminal, podemos olhar a tabela e determinar para cada terminal válido quais são as seqüências de ações a tomar, como, por exemplo, consumir um terminal ou invocar os métodos de outros não-terminais.

Porém, se a linguagem é complexa, o cálculo da tabela preditiva e a construção do AS de forma manual são tarefas árduas. Ainda mais se estivermos trabalhando

Tabela 4.1 – Tabela preditiva para a GLC de expressões.

Não terminais	Terminais						
	ident	constant	+, -	*, /	()	\$
expression	1	1			1		
expression'			2			3	3
term	4	4			4		
term'			6	5		6	6
factor	7	8			9		
(1) $expression \rightarrow term \ expression'$ (2) $expression' \rightarrow +term \ expression'$ (3) $expression' \rightarrow \lambda$ (4) $term \rightarrow factor \ term'$ (5) $term' \rightarrow *factor \ term'$ (6) $term' \rightarrow \lambda$ (7) $factor \rightarrow ident$ (8) $factor \rightarrow constant$ (9) $factor \rightarrow (expression)$							

com uma gramática na BNF, pois, como vimos, o cálculo dos conjuntos *FIRST* e *FOLLOW* consideram apenas produções normais, sem os operadores da BNF. Nesse ponto entram os programas como o JavaCC, que, baseado na definição da linguagem em BNF, gera todos os métodos correspondentes aos não-terminais e verifica possíveis problemas, avisando ao implementador os pontos da gramática que os originaram.

Na seção seguinte abordaremos a implementação do AS para a linguagem X^{++} utilizando o JavaCC.

4.2 O analisador sintático da linguagem X^{++}

Veremos, a seguir, como implementar o AS, com base na gramática vista no capítulo 2, utilizando o JavaCC. Inicialmente, adicionaremos uma nova opção na primeira parte do arquivo `.jj`.

Programa 4.2

```

1 options {
2   STATIC = false;
3   DEBUG_LOOKAHEAD = true;
4 }
```

A opção `DEBUG_LOOKAHEAD = true` habilita o mecanismo de depuração do AS que será gerado pelo JavaCC. Isso faz com que o AS mostre na saída-padrão quais são os não-terminais que estão sendo “executados” e quais são os tokens que são consumidos em cada um deles. Além disso, mostra também as tentativas de casamentos feitas ao analisar *lookaheads*. No final deste capítulo, mostraremos um exemplo em que esse tipo de saída está habilitada.

Nem sempre queremos ver quais são as ações tomadas pelo AS. Em geral, estamos mais interessados em saber se existe um erro sintático e qual é esse erro. O JavaCC permite que o mecanismo de depuração do AS seja desabilitado em tempo de execução. Assim, nosso compilador aceita como parâmetro de entrada a opção `-debug_AS`. Se essa opção for fornecida quando o nosso compilador é executado, então as informações de depuração serão mostradas. Se essa opção não é utilizada, então o AS mostra apenas se existe e qual é o erro sintático. Para isso, alteramos também o método `main` do nosso AS. Ele verifica se a opção foi dada ou não e, em caso negativo, invoca o método `langX.disable_tracing()` do AS criado (variável `parser` do programa 4.3).

Programa 4.3

```

1  PARSE_BEGIN(langX)
2  package parser;
3  import java.io.*;
4
5
6  public class langX {
7      final static String Version = "X++ Compiler - Version 1.0 - 2004";
8      int contParseError = 0;           // contador de erros sintáticos
9
10
11 // Define o método "main" da classe langX.
12     public static void main(String args[]) throws ParseException
13     {
14         boolean debug = false;
15
16         String filename = ""; // nome do arquivo a ser analisado
17         langX parser;        // analisador léxico/sintático
18         int i;
19         boolean ms = false;
20
21         System.out.println(Version);
22         // lê os parâmetros passados para o compilador
23         for (i = 0; i < args.length - 1; i++)
24         {
25             if (args[i].equals("-debug_AS"))
26                 debug = true;
27             else
28             {
29                 System.out.println("Usage is: " +
30                         "java langX [-debug_AS] inputfile");
31                 System.exit(0);
32             }
33         }
34
35         if (args[i].equals("-"))
36         {           // lê da entrada-padrão
37             System.out.println("Reading from standard input . . .");
38             parser = new langX(System.in); // cria AS
39         }
40         else
41         {           // lê do arquivo

```

```

42         filename = args[args.length-1];
43         System.out.println("Reading from file " + filename + " . . .");
44         try { // cria AS
45             parser = new langX(new java.io.FileInputStream(filename));
46         }
47         catch (java.io.FileNotFoundException e) {
48             System.out.println("File " + filename + " not found.");
49             return;
50         }
51     }

52
53     if (! debug) parser.disable_tracing(); // desabilita verbose do AS
54
55     try {
56         parser.program(); // chama o método que faz a análise
57     }
58     catch (ParseException e)
59     {
60         System.err.println(e.getMessage());
61         parser.contParseError = 1; // não existe recuperação de erros
62     }
63     finally {
64         System.out.println(parser.token_source.foundLexError() +
65                             " Lexical Errors found");
66         System.out.println(parser.contParseError +
67                             " Syntactic Errors found");
68     }
69
70 } // main
71
72 static public String im(int x)
73 {
74     int k;
75     String s;
76     s = tokenImage[x];
77     k = s.lastIndexOf("\\\"");
78     try {s = s.substring(1,k);}
79     catch (StringIndexOutOfBoundsException e)
80     {}
81     return s;
82 }
83
84 } // langX
85
86 PARSER_END(langX)

```

No método *main*, temos também a chamada ao método *parser.program*. Como veremos a seguir, o JavaCC irá definir um método para cada símbolo não terminal da gramática. A partir das produções descritas no arquivo *.jj*, são criados os métodos que tentam fazer o casamento da entrada com os respectivos não-terminais. Assim, a chamada do método *parser.program* irá tentar reconhecer na entrada uma cadeia

que case com a descrição dada a seguir no arquivo .jj para o não-terminal *program*. Esse é o símbolo inicial da gramática. A entrada utilizada é a que está no arquivo passado como parâmetro na criação do objeto *parser*, no início do método *main* (linhas 38 e 45).

Uma das vantagens do AS gerado pelo JavaCC é que o reconhecimento não precisa ser feito necessariamente a partir de algum não-terminal específico. Uma chamada a qualquer método que corresponda a um não-terminal irá tentar reconhecer esse não-terminal na entrada. Por exemplo, se tivéssemos feito a chamada a *parser.expression*, o nosso compilador iria consumir os tokens da entrada procurando reconhecer uma expressão apenas, e não um programa.

A chamada a *parser.program* está dentro de um comando *try* porque a sua execução ou a de algum outro método correspondente a outro não-terminal pode lançar uma exceção do tipo *ParseException*. Isso acontece quando o AS detecta na entrada algum erro sintático. Por exemplo, o método correspondente ao não-terminal *classdecl* espera encontrar na entrada o token *class*. Caso esse método seja invocado e não encontre tal token na entrada, então ele lança uma exceção. Como nenhum método correspondente aos não-terminais trata essa exceção, ela se propaga até o método que primeiro chamou um desses métodos, no caso o método *main*. No método *main*, uma mensagem referente ao erro é emitida e a análise do programa deve terminar. Não há como continuar a análise, pois é impossível restaurar o estado do AS no momento que o erro foi detectado. Assim, ao encontrar um erro sintático, a execução do nosso compilador simplesmente termina. No capítulo 5 veremos como podemos implementar o que se chama de recuperação de erros, que permite que o AS continue sua execução mesmo que um erro sintático seja encontrado.

Vejamos, então, como definir os métodos que irão compor o nosso AS. Devemos definir cada não-terminal da nossa gramática mediante uma declaração que é uma mistura de código Java e de produções na BNF. Cada uma dessas declarações tem a seguinte forma:

$$\langle \text{não terminal} \rangle \rightarrow \langle \text{tipo} \rangle \text{ "nome" "(" } \langle \text{argumentos} \rangle \text{ ")" ":" } \\ \text{ "{" } \langle \text{decl. locais} \rangle \text{ "}" } \\ \text{ "{" } \langle \text{BNF} \rangle \text{ "}" }$$

Essa declaração inicia-se com um tipo, que é o tipo retornado pelo método correspondente a esse não-terminal. Pode ser qualquer tipo acessível dentro da classe do AS, no nosso caso, a classe *langX*. Depois vem o nome do não-terminal (e do método que será criado) e os parâmetros formais desse método. Em seguida, vêm um “:” e uma sessão de código Java colocada entre chaves, que corresponde às declarações locais desse método. As variáveis aí declaradas poderão ser utilizadas dentro do método. E depois vêm as produções desse não-terminal na BNF. A principal incumbência do JavaCC é transformar essas produções em código Java, que irá tentar reconhecer na entrada uma seqüência de tokens que casem com essa descrição. Vejamos, então, o símbolo inicial da nossa linguagem, que é o não-terminal *program*:

Programa 4.4

```

1 void program() :
2 {
3 }
4 {
5     [ classlist() ] <EOF>
6 }
```

Na descrição das produções podem aparecer os tokens definidos na seção do AL do arquivo .jj (o *EOF* é uma exceção) e “chamadas” a outros não-terminais como *classlist*.

Esse não-terminal *program* é utilizado aqui para indicar quando um *EOF* (fim de arquivo) é válido na nossa entrada. O token *EOF* não foi definido explicitamente nas declarações de tokens do AL. Trata-se de um token predefinido que é passado para o AS quando o arquivo de entrada foi completamente lido e não possui mais nenhum token disponível. Então, esse não-terminal *program* diz que um programa na nossa linguagem é formado por uma lista de classes, seguida pelo *EOF*. A definição de *classlist* é mostrada no programa 4.5. Ela simplesmente segue a definição da BNF vista no capítulo 2. Isto vale para *classdecl*. Sua produção inicia-se com *class*, vem seguida por um identificador, e assim por diante.

Programa 4.5

```

1 void classlist():
2 {
3 }
4 {
5     classdecl() [ classlist() ]
6 }
7
8
9 void classdecl():
10 {
11 }
12 {
13     <CLASS> <IDENT> [ <EXTENDS> <IDENT> ] classbody()
14 }
```

É bom observar que, em geral, o código Java gerado pelo JavaCC para os métodos que implementam os não-terminais não tem muita semelhança com as suas produções e pode ser difícil de ser compreendido. Por isso, não é uma política recomendável gerar o AS por meio do JavaCC e, depois, fazer alterações no código gerado. Além disso, se for necessária a alteração do arquivo .jj, um novo código Java deve ser gerado e as alterações feitas anteriormente diretamente nesse código são perdidas. Apenas como exemplo, apresentamos no programa 4.6 um trecho de código gerado pelo JavaCC para o método correspondente ao não-terminal *classdecl*.

Programa 4.6

```

1   final public void classdecl() throws ParseException {
2       trace_call("classdecl");
3       try {
4           jj_consume_token(CLASS);
5           jj_consume_token(IDENT);
6           switch ((jj_ntk===-1)?jj_ntk():jj_ntk) {
7               case EXTENDS:
8                   jj_consume_token(EXTENDS);
9                   jj_consume_token(IDENT);
10                  break;
11             default:
12                 jj_lai[1] = jj_gen;
13                 ;
14             }
15             classbody();
16         } finally {
17             trace_return("classdecl");
18         }
19     }

```

*

Continuando com a nossa implementação, temos o não-terminal *classbody*. De acordo com a nossa BNF, esse não-terminal deveria ser definido como:

Programa 4.7

```

1   void classbody():
2   {
3   }
4   {
5       <LBRACE>
6       [classlist()]
7       (vardecl() <SEMICOLON>)*
8       (constructdecl())*
9       (methoddecl())*
10      <RBRACE>
11  }

```

*

Porém, se tentarmos processar o nosso arquivo .jj contendo essa declaração, veremos que o JavaCC produz a seguinte advertência:

```

Warning: Choice conflict in (...)* construct at line 304, column 7.
Expansion nested within construct and expansion following construct
have common prefixes, one of which is: "int"
Consider using a lookahead of 2 or more for nested expansion.

```

Essa mensagem indica que existe um conflito na construção (*vardecl()* <*SEMICOLON*>)* da nossa produção. Diz, ainda, que dentro da construção e fora dela existem prefixos comuns, entre eles o terminal *int*. Isso acontece porque o *FIRST(vardecl)* inclui o terminal *int*, que também está no *FIRST(methoddecl)*, como mostra o programa 4.8.

Programa 4.8

```

1 void vardecl():
2 {
3 }
4 {
5     (<INT> | <STRING> | <IDENT> )
6     <IDENT> ( <LBRACKET> <RBRACKET>)*
7     (<COMMA> <IDENT> ( <RBRACKET> <LBRACKET>)* )*
8 }
9
10 void constructdecl():
11 {
12 }
13 {
14     <CONSTRUCTOR> methodbody()
15 }
16
17
18
19 void methoddecl():
20 {
21 }
22 {
23     (<INT> | <STRING> | <IDENT> ) (<RBRACKET> <LBRACKET>)*
24     <IDENT> methodbody()
25 }
```

*

Assim, o JavaCC não pode decidir se, ao ler o token *int*, o método *classbody* deve prosseguir reconhecendo um *vardecl* ou um *methoddecl*. É bom notar que *vardecl*, *constructdecl* e *methoddecl* podem ou não aparecer numa *classdecl*, o que gera esse conflito. Resumindo, essa advertência do JavaCC nos informa que nossa gramática não é LL(1) e, portanto, não é adequada para esse tipo de AS.

Porém, a última linha da mensagem nos dá uma indicação de como resolver esse problema, sem ter que alterar nossa gramática. O JavaCC não consegue decidir qual caminho seguir olhando apenas um símbolo da entrada, pois ele pode ser o mesmo para *vardecl* ou *methoddecl*. Mas talvez, utilizando mais alguns símbolos da entrada, seja capaz de tomar a decisão. Vamos verificar as construções que estão em conflito:

- ***vardecl*:** inicia-se com o tipo da variável que pode ser um *int*, um *string* ou um identificador. Depois tem um identificador, que é o nome de uma variável, e, depois, pode ter um abre colchete ou uma vírgula, ou um ponto-e-vírgula, e dependendo desse terceiro símbolo, pode ter um fecha colchete, ou um identificador, ou o fim da produção, respectivamente;
- ***methoddecl*:** inicia-se com o tipo do método, depois pode ou não ter uma sequência de abre e fecha colchetes e, depois, tem um identificador que é o nome do método e, depois, o corpo do método, que se inicia com um abre parênteses.

Assim, para saber se a entrada corresponde ou não a um *vardecl*, precisamos olhar três símbolos para frente em vez de apenas um. Se o primeiro símbolo for um *int*, um

string ou um identificador, ficamos em dúvida entre os dois não-terminais em questão. Olhamos, então, o segundo, que se for um identificador, ainda pode ser qualquer um dos dois (um método que retorna um tipo sem dimensão tem como segundo símbolo um identificador). Mas se olharmos o terceiro símbolo, veremos que para o caso da declaração de uma variável, ele deve ser um abre colchete, uma vírgula ou um ponto-e-vírgula, e que nenhum destes pode aparecer como terceiro símbolo da declaração de método. Para esta última, poderíamos ter um fecha colchete ou um abre parênteses do início do não-terminal *methodbody*.

Então, adicionaremos à nossa implementação o seguinte comando:

Programa 4.9

```

1 void classbody():
2 {
3 }
4 {
5     <LBRACE>
6     [classlist()]
7     (LOOKAHEAD(3) vardecl() <SEMICOLON>)*
8     (constructdecl())*
9     (methoddecl())*
10    <RBRACE>
11 }
```

O *LOOKAHEAD(3)* utilizado na frente do *vardecl* diz ao JavaCC que deve gerar um método que analisa três símbolos à frente para decidir se deve ou não tentar casar a entrada com um *vardecl*. É muito importante notar que o AS olha três símbolos adiante e verifica se eles podem ser casados com o que está após o *LOOKAHEAD*, neste caso, o não-terminal *vardecl*. Se esses três símbolos casarem, ele segue por esse caminho e se algum deles não casar, um caminho alternativo é tomado. Assim, o *LOOKAHEAD* deve ser utilizado sempre num ponto de decisão do AS. Além disso, devemos ser muito cuidadosos ao estipularmos o valor do *LOOKAHEAD*, pois se o número de símbolos estipulados for menor que o necessário, podemos ter entradas casadas com produções inadequadas, e nenhum aviso do JavaCC, pois o uso do *LOOKAHEAD* desliga a verificação de conflitos naquele ponto.

Por exemplo, se na produção anterior tivéssemos utilizado um *LOOKAHEAD(2)* ou mesmo um *LOOKAHEAD(1)*, e tivéssemos na entrada a seguinte cadeia:

int x() ...

teríamos gerado um AS que iria tentar reconhecer essa cadeia como um *vardecl*. Isso porque o AS iria olhar os próximos dois (ou um, se usássemos *LOOKAHEAD(1)*) símbolos que combinam com *vardecl* e isso seria o suficiente para decidir que esse seria o caminho correto para tentar reconhecer a entrada. Então, não se engane se introduzir um comando de *LOOKAHEAD* no arquivo .jj fizer com que uma advertência como aquela vista há pouco deixe de ocorrer. Isso não garante que a decisão que o implementador forçou o AS a tomar esteja correta.

O JavaCC possui uma opção global que é *LOOKAHEAD = n*, que determina que durante toda a análise o número de tokens consultados para decidir sobre que produção

aplicar é igual a n . O valor-padrão é 1, e quanto maior for esse valor, mais ineficiente será o AS. Assim, recomenda-se utilizar o valor 1 como lookahead global e utilizar comandos *LOOKAHEAD* nos pontos que forem necessários.

Os próximos não-terminais são simplesmente implementação do que já foi descrito na nossa gramática e não devem representar problema para o leitor.

Programa 4.10

```

1 void methodbody():
2 {
3 }
4 {
5     <LPAREN> paramlist() <RPAREN> statement()
6 }
7
8 void paramlist():
9 {
10 }
11 {
12     [
13         (<INT> | <STRING> | <IDENT>) <IDENT> (<RBRACKET> <LBRACKET>)*
14         (<COMMA> (<INT> | <STRING> | <IDENT>
15             <IDENT> (<RBRACKET> <LBRACKET>)*
16             )*
17     ]
18 }
```

No não-terminal *statement*, encontramos novamente um comando *LOOKAHEAD*, em frente de *vardecl*. Nesse caso, *vardecl* entra em conflito com *atribstat*, pois ambos podem iniciar-se com um identificador. Mas o *vardecl* sempre tem, depois da declaração do tipo, um outro identificador, o que não pode ocorrer no comando representado por *atribstat*. Assim, um lookahead de dois símbolos é suficiente para resolver o problema.

Programa 4.11

```

1 void statement():
2 {
3 }
4 {
5     LOOKAHEAD(2)
6     vardecl()
7     |
8     atribstat() <SEMICOLON>
9     |
10    printstat() <SEMICOLON>
11    |
12    readstat() <SEMICOLON>
13    |
14    returnstat() <SEMICOLON>
15    |
16    superstata() <SEMICOLON>
17    |
```

```

18     ifstat()
19   |
20     forstat()
21   |
22     <LBRACE> statlist() <RBRACE>
23   |
24     <BREAK> <SEMICOLON>
25   |
26     <SEMICOLON>
27 }

```

*

Os comandos da linguagem são reconhecidos pelos não-terminais seguintes:

Programa 4.12

```

1 void atribstat():
2 {
3 }
4 {
5   lvalue() <ASSIGN> ( alocexpression() | expression())
6 }
7
8 void printstat():
9 {
10 }
11 {
12   <PRINT> expression()
13 }
14
15 void readstat():
16 {
17 }
18 {
19   <READ> lvalue()
20 }
21
22
23 void returnstat():
24 {
25 }
26 {
27   <RETURN> [expression()]
28 }
29
30
31 void superstat():
32 {
33 }
34 {
35   <SUPER> <LPAREN> arglist() <RPAREN>
36 }
37
38 void ifstat():

```

```

39  {
40  }
41  {
42      <IF> <LPAREN> expression() <RPAREN> statement()
43      [LOOKAHEAD(1) <ELSE> statement()]
44  }
45
46
47
48 void forstat():
49 {
50 }
51 {
52     <FOR> <LPAREN> [atribstat()] <SEMICOLON>
53             [expression()] <SEMICOLON>
54             [atribstat()] <RPAREN>
55             statement()
56 }
57
58 void statlist() :
59 {
60 }
61 {
62     statement() [statlist()]
63 }
64
65 void lvalue() :
66 {
67 }
68 {
69     <IDENT> (
70         <LBRACKET> expression() <RBRACKET> |
71         <DOT> <IDENT> [<LPAREN> arglist() <RPAREN>]
72         )*
73 }
74
75 void aloceexpression() :
76 {
77 }
78 {
79     <NEW> (
80         LOOKAHEAD(2) <IDENT> <LPAREN> arglist() <RPAREN> |
81         ( <INT> | <STRING> | <IDENT> )
82         (<LBRACKET> expression() <RBRACKET>)+
83         )
84 }

```

O não-terminal *aloceexpression* possui um comando *LOOKAHEAD* cuja utilização o leitor pode facilmente identificar. Um caso diferente ocorre no não-terminal *ifstat*. Se não utilizarmos o comando *LOOKAHEAD* especificado naquele não-terminal, obteremos a seguinte advertência ao processar o arquivo .jj:

Warning: Choice conflict in [...] construct at line 425, column 5.
 Expansion nested within construct and expansion following construct
 have common prefixes, one of which is: "else"
 Consider using a lookahead of 2 or more for nested expansion.

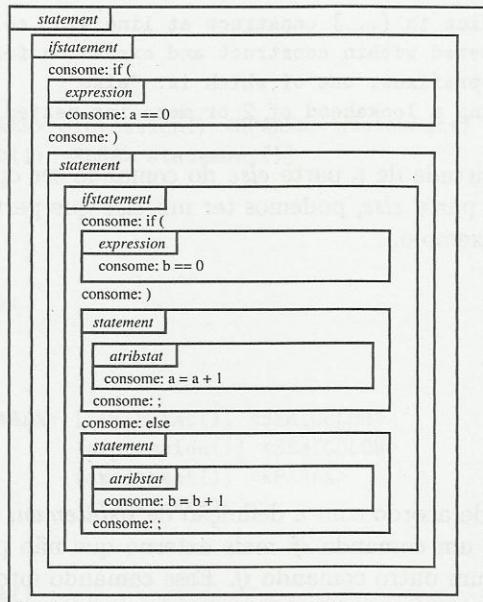
Isso acontece em virtude de a parte *else* do comando ser opcional. E depois de um comando *if* sem a parte *else*, podemos ter um *else* que pertence a um comando *if* mais externo. Por exemplo,

```
if ( a == 0 )
  if ( b == 0 )
    a = a + 1;
else
  b = b + 1;
```

Essa cadeia pode, de acordo com a definição de *ifstatement* (sem o *LOOKAHEAD*), ser interpretada como um comando *if* mais externo que não possui a parte *else* e que tem interno a ele um outro comando *if*. Esse comando interno possui, na parte “*then*”, uma atribuição *a = a + 1* e, na parte *else*, uma outra atribuição *b = b + 1*. Outra interpretação é um comando externo que tem um comando interno *if* no ramo verdadeiro e um comando de atribuição *b = b + 1* na parte *else*. O comando *if* interno possui apenas uma atribuição dentro dele, sem ter a parte *else*.

Para tentar resolver esse conflito, poderíamos tentar aumentar o número de tokens utilizados como lookahead. Porém, isso se mostraria inútil. O JavaCC não consegue decidir que caminho tomar quando o próximo token for um *else*, pois não sabe se deve consumir o símbolo *else* dentro do comando *if* ou pular toda a parte *else* do comando e casar a entrada com a estrutura que está mais externa ao *ifstatement*. Mas essa estrutura é a própria parte *else* de um outro *ifstat* mais externo. Assim, se tentarmos aumentar o número de lookaheads, iremos casar esses símbolos sempre com as duas construções que estão em conflito (na verdade, a mesma construção) ou com nenhuma delas. Isso significa que a decisão será a de utilizar sempre a produção do *ifstat* mais interno, pois o *LOOKAHEAD* induz a isso.

Felizmente, esse é o comportamento que desejamos. Ou seja, o *else* estará sempre associado com o *if* anterior a ele que estiver mais próximo. No exemplo, teríamos a situação mostrada na figura 4.1. Mais externamente, temos a chamada ao método correspondente ao *statement*, que, ao verificar o token corrente na cadeia de entrada, chama o método *ifstatement*. Este, por sua vez, consome os tokens *if* e *(*, depois chama *expression* e, ao retornar daquela chamada, consome *)*. Em seguida, chama recursivamente o *statement*, que, por sua vez, chama de novo o *ifstatement*. Essa segunda chamada consome *if* e *(*, chama *expression* e consome *)*. Chama, então, *statement* que, olhando o lookahead, chama *atribstat*. Ao retornar do *atribstat* e do *statement*, o método *ifstatement* verifica o token corrente, e como é um *else*, ele consome esse token (ou seja, decidiu por não pular a parte “*else*” da produção) e chama *statement* para a segunda atribuição. Esse é o comportamento desejado. Note, então, que o comando *LOOKAHEAD(1)* serve para forçar o AS a consumir o *else* dentro do método *ifstatement*.

Figura 4.1 – Chamadas do AS para *ifs* aninhados.

E, para concluir, temos as implementações dos não-terminais correspondentes às expressões da nossa linguagem. Aqui também não há nada de importante a ser comentado, pois esses não-terminais são apenas uma reprodução das produções mostradas no capítulo 2 e aí comentadas.

Programa 4.13

```

1 void expression() :
2 {
3 }
4 {
5     numexpr() [(<LT> | <GT> | <LE> | <GE> | <EQ> | <NEQ>) numexpr()]
6 }
7
8 void numexpr():
9 {
10 }
11 {
12     term() ((<PLUS> | <MINUS>) term())*
13 }
14
15 void term():
16 {
17 }
18 {
19     unaryexpr() ((<STAR> | <SLASH>| <REM>) unaryexpr())*
20 }
  
```

```

21
22 void unaryexpr() :
23 {
24 }
25 {
26   [(<PLUS> | <MINUS>)] factor()
27 }
28
29
30 void factor():
31 {
32 }
33 {
34
35
36   (
37     <int_constant> |
38     <string_constant> |
39     <null_constant> |
40     lvalue() |
41     <LPAREN> expression() <RPAREN>
42 }
43
44 void arglist():
45 {
46 }
47 {
48   [expression() (<COMMA> expression())*]
49 }
50

```

4.3 Arquivos-fonte do compilador

No diretório *cap04/parser* se encontra a nova versão do nosso arquivo *.jj* que implementa a análise sintática do nosso compilador. Os comandos para criar o AS são os mesmos mostrados no capítulo 3:

```

cap04/parser$ javacc langX++.jj
Java Compiler Compiler Version 2.1 (Parser Generator)
Copyright (c) 1996-2001 Sun Microsystems, Inc.
Copyright (c) 1997-2001 WebGain, Inc.
(type "javacc" with no arguments for help)
Reading from file langX++.jj . .
File "TokenMgrError.java" does not exist. Will create one.
File "ParseException.java" does not exist. Will create one.
File "Token.java" does not exist. Will create one.
File "SimpleCharStream.java" does not exist. Will create one.
Parser generated successfully.
cap04$ javac parser/langX.java

```

Incluiu-se, também, no diretório *ssample* um programa *bintree-erro-sintatico* que possui um versão do *bintree* com alguns erros sintáticos. Ao executarmos o nosso compilador para analisar esse arquivo, obtemos o seguinte resultado:

```
cap04$ java parser.langX ..//ssamples/bintree-erro-sintatico.x
X++ Compiler - Version 1.0 - 2004
Reading from file ..//ssamples/bintree-erro-sintatico.x . . .
Encountered "mes" at line 21, column 4.
Was expecting one of:
  "[" ...
  ";" ...
  ":" ...
  ">" ...
  "<" ...
  "==" ...
  "<=" ...
  ">=" ...
  "!=" ...
  "+" ...
  "_" ...
  "*" ...
  "/" ...
  "%" ...
0 Lexical Errors found
1 Syntactic Errors found
```

A mensagem mostra-nos quais seriam os possíveis tokens esperados após a atribuição *dia = d*. Esquecemos-nos de colocar o ponto-e-vírgula após esse comando, o que originou o erro. Note que, apesar de conter diversos erros sintáticos, somente o primeiro é mostrado ao analisarmos o arquivo *bintree-erro-sintatico*. Isto porque o nosso AS não se recupera do erro e termina sua execução. No capítulo 5 veremos como implementar recuperação de erro no nosso compilador.

Incluiu-se, também, no diretório *ssample* o arquivo *debugAS.x* que contém um programa extremamente simples:

```
class A {
    int a, b;

    int m()
    ;
}
```

Aqui poderia haver um conflito, pois a declaração de variáveis que inicia a classe poderia ser tomada com uma declaração de variáveis ou como uma declaração de método, se apenas os dois primeiros tokens fossem considerados para decidir qual produção utilizar. Algo semelhante ocorre com a declaração do método *m*. Mas

o AS foi gerado com um lookahead de três tokens nesse ponto, o que soluciona o conflito. Vamos executar o nosso compilador utilizando a opção `-debug_AS` para esse programa. A saída obtida é a seguinte:

```
cap04$ java parser.langX -debug_AS ../ssamples/debugAS.x
X++ Compiler - Version 1.0 - 2004
Reading from file ../ssamples/debugAS.x . . .
Call: program
Call: classlist
Call: classdecl
Consumed token: <"class">
Consumed token: <<IDENT>: "A">
Call: classbody
Consumed token: <"{">
Call: vardecl(LOOKING AHEAD...)
Visited token: <"int">; Expected token: <"int">
Visited token: <<IDENT>: "a">; Expected token: <<IDENT>>
Visited token: <", ">; Expected token: <"[">
Visited token: <", ">; Expected token: <", ">
Return: vardecl(LOOKAHEAD SUCCEEDED)
Call: vardecl
Consumed token: <"int">
Consumed token: <<IDENT>: "a">
Consumed token: <", ">
Consumed token: <<IDENT>: "b">
Return: vardecl
Consumed token: <", ">
Call: vardecl(LOOKING AHEAD...)
Visited token: <"int">; Expected token: <"int">
Visited token: <<IDENT>: "m">; Expected token: <<IDENT>>
Visited token: <"(">; Expected token: <"[">
Visited token: <"(">; Expected token: <", ">
Return: vardecl(LOOKAHEAD SUCCEEDED)
Visited token: <"(">; Expected token: <%;">
Call: methoddecl
Consumed token: <"int">
Consumed token: <<IDENT>: "m">
Call: methodbody
Consumed token: <"(">
Call: paramlist
Return: paramlist
Consumed token: <")">
Call: statement
Call: vardecl(LOOKING AHEAD...)
Visited token: <"%;">; Expected token: <"int">
Visited token: <"%;">; Expected token: <"string">
Visited token: <"%;">; Expected token: <<IDENT>>
Return: vardecl(LOOKAHEAD FAILED)
```

```

Consumed token: <";">
Return: statement
Return: methodbody
Return: methoddecl
Consumed token: <"}">
Return: classbody
Return: classdecl
Return: classlist
Consumed token: <<EOF>>
Return: program
0 Lexical Errors found
0 Syntactic Errors found

```

Veja que a saída mostra quais são os métodos chamados e quais são os tokens consumidos em cada um deles. Mostra, também, quando existe um comando *LOOKAHEAD*, quais são os tokens visitados na verificação dos lookaheads. É o caso do comando antes de *vardecl* no método *classdecl*. Na primeira tentativa, esse comando *LOOKAHEAD* termina com êxito e decide-se por tentar analisar a entrada por meio de um *vardecl*. Já no segundo caso, embora três símbolos tenham sido visitados, o terceiro não cessa com o não-terminal *vardecl* e, por isso, o casamento é tentado com a próxima possibilidade, que é *methoddecl*.

Capítulo 5

Tratamento de Erros Sintáticos

O compilador que temos até agora desempenha a simples tarefa de tentar reconhecer uma cadeia de acordo com uma gramática e avisar o usuário caso isso não seja possível. Se a cadeia de entrada não se enquadra em nenhuma produção da gramática, nosso AS mostra em que ponto da entrada não foi possível fazer o casamento e, então, termina sua execução. Em geral, essa indicação serve para que o usuário possa identificar o erro sintático ocorrido, corrigir a entrada e analisá-la novamente.

Esse processo pode ser entediante se o número de erros sintáticos é grande, pois a cada execução do compilador, o usuário deve editar o arquivo de entrada, localizar o erro, corrigi-lo e reiniciar o processo. Seria interessante se nosso compilador fosse capaz de mostrar todos ou, pelo menos, diversos erros sintáticos de uma só vez. Assim, o número de iterações compila-corrigir seria reduzido. Para que isso aconteça, o compilador deve ser capaz de recuperar-se do erro ocorrido e continuar na análise sintática. Isso nem sempre é fácil, pois enquanto a entrada casa com as produções da gramática, é simples para o AS decidir quais produções devem ser tomadas para reconhecer a cadeia. Porém, quando um erro sintático ocorre, torna-se mais difícil decidir como continuar no reconhecimento do restante da entrada.

Infelizmente, o JavaCC não oferece ao desenvolvedor muitos recursos para que seja implementada uma recuperação de erros eficiente. O método que utilizaremos e que estudaremos aqui é conhecido como método de ressincronização ou método do pânico. Veremos inicialmente qual o conceito desse método e, então, discutiremos sua implementação utilizando o JavaCC.

5.1 O método de ressincronização

Iniciaremos o estudo desse método com um exemplo. Vamos tomar a seguinte gramática:

$$\begin{array}{l} S \rightarrow aAcd \\ A \rightarrow gh \end{array}$$

Queremos analisar a entrada *agabcd*. Teríamos no nosso AS descendente recursivo:

- uma chamada ao método *S* que consome o token *a* da entrada;
- uma chamada ao método *A* que consome o *g*;
- o método *A* tenta achar na entrada um *h* que não está aí e, então, um erro sintático ocorre.

A idéia do método de ressincronização é fazer com que o método *S* não seja afetado por esse erro sintático e que possa continuar a analisar a entrada. Porém, se o método *A* simplesmente emitir uma mensagem de erro e retornar a execução para o método que o chamou, um outro erro sintático irá ocorrer, pois o método *S* espera que depois da chamada a *A* exista na entrada o token *c*, o que não ocorre, visto que ainda temos na entrada a cadeia *abcd*.

Por isso o método *A*, ao detectar um erro sintático, deve ressincronizar a entrada com o não-terminal que se espera reconhecer. Uma tentativa de se fazer isso é consumindo tokens da entrada até que apareça algum que possibilite a continuidade da análise. Mas quais seriam esses tokens? Uma boa idéia seria utilizar o conjunto *FOLLOW* de *A*. Assim, o método *A* deve consumir tokens da entrada até que apareça um símbolo pertencente ao seu conjunto *FOLLOW* e só depois retornar a execução para *S*. Em *S*, tudo se passa como se a execução de *A* tivesse sido bem-sucedida e a análise continua, com a certeza de que, pelo menos, o próximo token da entrada irá casar com a produção sendo utilizada. Com essas modificações, o AS descendente recursivo ficaria algo como o código mostrado no programa 5.1

Programa 5.1

```

1 void S()
2 {
3     if (curToken.type == 'a')
4     {
5         analex();
6         A();
7         if (curToken.type == 'c')
8         {
9             analex();
10            if (curToken.type == 'd')
11                analex();
12            else
13                SintaticError("Encontrado " + curToken.type() +
14                                " esperado: d");
15        }
16        else
17            SintaticError("Encontrado " + curToken.type() +
18                                " esperado: c");
19    }
20    else

```

```

21         SintaticError("Encontrado " + curToken.type() +
22                             " esperado: a");
23     }
24
25 void A()
26 {
27     try {
28         if (curToken.type == 'g')
29         {
30             analex();
31             if (curToken.type == 'h')
32                 analex();
33             else SintaticError("Encontrado " + curToken.type +
34                               " esperado: h");
35         }
36         else
37             SintaticError("Encontrado " + curToken.type() +
38                           "esperado: g");
39     }
40     catch (ASEException e)
41     {
42         System.err.println(e.getMessage());
43         consume_until('c');
44     }
45 }
46
47 void SintaticError(String s)
48 {
49     throw new ASEException(s);
50 }
51
52 void consume_until(int t)
53 {
54     while (curToken.type != t)
55         analex();
56 }

```

Nessa nossa implementação, o método *SintaticError* é responsável por lançar uma exceção do tipo *ASEException*. O método *consume_until* consome a entrada até que um determinado token seja encontrado. E o método *A* utiliza uma construção *try/catch* para tentar reconhecer a produção correspondente ao seu não-terminal *e*, em caso de erro, irá chamar o método *consume_until* que faz a resincronização desejada.

Mas podemos fazer ainda um pouco melhor do que isso. Vamos tomar como exemplo a seguinte gramática:

$$\begin{array}{lcl} S & \rightarrow & aAcd \mid bAef \\ A & \rightarrow & gh \end{array}$$

Nesse caso, $FOLLOW(A) = \{c, e\}$ e podemos utilizar esses dois tokens para efetuar a resincronização. Porém, se estivermos utilizando a primeira produção de *S*, dese-

jaremos utilizar o *c* para fazer a recuperação de erros, e não o *e*. E se estivermos utilizando a segunda produção de *S*, desejaremos utilizar o *e* para fazer a recuperação de erros, e não o *c*. Isso porque se fizermos a resincronização com o token errado, um novo erro sintático surgirá em *S*. Assim, podemos alterar o nosso AS, permitindo que a cada chamada de *A* seja informado, por meio de um parâmetro, qual é o token de sincronização a ser utilizado. Essa alteração é mostrada no programa 5.2

Programa 5.2

```

1 void S()
2 {
3     if (curToken.type == 'a')
4     {
5         analex();
6         A('c');
7         if (curToken.type == 'c')
8         {
9             analex();
10            if (curToken.type == 'd')
11                analex();
12            else
13                SintaticError("Encontrado " + curToken.type() +
14                                " esperado: d");
15        }
16        else
17            SintaticError("Encontrado " + curToken.type() +
18                                " esperado: c");
19    }
20    else
21    if (curToken.type == 'b')
22    {
23        analex();
24        A('e');
25        if (curToken.type == 'e')
26        {
27            analex();
28            if (curToken.type == 'f')
29                analex();
30            else
31                SintaticError("Encontrado " + curToken.type() +
32                                " esperado: f");
33        }
34        else
35            SintaticError("Encontrado " + curToken.type() +
36                                " esperado: e");
37    }
38    else
39        SintaticError("Encontrado " + curToken.type() +
40                                " esperado: b");
41 }
42
43 void A(int k)
44 {

```

```

45     try {
46         if (curToken.type == 'g')
47         {
48             analex();
49             if (curToken.type == 'h')
50                 analex();
51             else SintaticError("Encontrado " + curToken.type +
52                                 " esperado: h");
53         }
54         else
55             SintaticError("Encontrado " + curToken.type() +
56                           " esperado: g");
57     }
58     catch (ASEException e)
59     {
60         System.err.println(e.getMessage());
61         consume_until(k);
62     }
63 }
64
65 void SintaticError(String s)
66 {
67     throw new ASEException(s);
68 }
69
70 void consume_until(int t)
71 {
72     while (curToken.type != t)
73         analex();
74 }

```

A cada produção em S corresponde uma chamada distinta de A e, por isso, podemos passar parâmetros distintos para recuperação para cada uma delas. Note que, na verdade, ao tentarmos fazer a resincronização, nem sempre teremos uma única possibilidade, ou seja, nem sempre temos um único token que serve como sincronizador. Na maioria das vezes, utilizamos um conjunto de sincronização. Assim, os parâmetros que são passados para os métodos A e $consume_until$ deveriam ser conjuntos de tokens, e não um só token.

Esse exemplo simples apresenta o conceito básico da técnica de recuperação de erros que iremos utilizar com o JavaCC. Muitos pontos devem ainda ser abordados e procuraremos fazê-lo a seguir, quando discutiremos a implementação da recuperação de erros sintáticos para o AS da linguagem X^{++} .

5.2 Implementação da recuperação de erros

A facilidade que o JavaCC nos oferece para tratarmos erros sintáticos é bastante semelhante à abordagem mostrada nos exemplos. O JavaCC permite que as produções na BNF sejam colocadas entre construções `try/catch` e caso algum erro sintático seja detectado ao tentar casar a entrada com essas produções, podemos tratar esse erro.

Como vimos no capítulo 4 anterior, um erro sintático faz com que uma exceção do tipo *ParseException* seja lançada. Assim, um não-terminal com tratamento de erros fica semelhante a:

```
void A()
{
}
{
try {
    "g" "h"
}
catch (ParseException e)
{
    // tratamento de erro
}
}
```

Se o não-terminal possui chamada para outro não-terminal, não há mudança. Porém é importante notar que o não-terminal mais interno pode ou não tratar seus próprios erros. Se não tratar, um erro ocorrido na chamada interna é propagado por meio da exceção até o método que fez a chamada, que pode fazer o tratamento com a resincronização. Por exemplo, a seguir temos o método *A* que chama *B*, que não faz o tratamento de erros e pode, por isso, lançar uma *ParseException*, que é então tratada em *A*.

```
void A()
{
}
{
try {
    "g" B() "h"
}
catch (ParseException e)
{
    // tratamento de erros de A ou de B
}
}
```

Verificaremos, então, como implementar a recuperação de erros em nosso AS. Iniciamos com a inclusão de uma nova opção que pode ser utilizada pelo usuário para depurar a recuperação de erros. Essa opção *-debug_recovery* faz com que o AS emita mensagens de como está sendo processada a recuperação de erros, quando esta for aplicada. Com isso, a seção inicial do nosso analisador, que inclui o método *main*, deve ser alterada como mostrado no programa 5.3.

Programa 5.3

```

1 PARSER_BEGIN(langX)                               Programa 5.3
2 package parser;
3
4 import java.io.*;
5 import recovery.*; // importa as classes de recuperação de erros do AS
6
7
8 public class langX {
9     final static String Version = "X++ Compiler - Version 1.0 - 2004";
10    int contParseError = 0;           // contador de erros sintáticos
11    boolean debug_recovery; // controla verbose de recuperação de erros
12
13
14 // Define o método "main" da classe langX.
15    public static void main(String args[]) throws ParseException
16    {
17        boolean debug_as = false;
18        boolean debug_recovery = false;
19
20        String filename = ""; // nome do arquivo a ser analisado
21        langX parser; // analisador léxico/sintático
22        int i;
23        boolean ms = false;
24
25        System.out.println(Version);
26        // lê os parâmetros passados para o compilador
27        for (i = 0; i < args.length - 1; i++)
28        {
29            if (args[i].equals("-debug_AS"))
30                debug_as = true;
31            else
32                if (args[i].equals("-debug_recovery"))
33                    debug_recovery = true;
34            else
35            {
36                System.out.println("Usage is: java langX [-debug_AS] "
37                                "[-debug_recovery] inputfile");
38                System.exit(0);
39            }
40        }
41
42        if (args[i].equals("-"))
43        {           // lê da entrada-padrão
44            System.out.println("Reading from standard input . . .");
45            parser = new langX(System.in); // cria AS
46        }
47        else
48        {           // lê do arquivo
49            filename = args[args.length-1];
50            System.out.println("Reading from file " + filename + " . . .");
51            try { // cria AS
52                parser = new langX(new java.io.FileInputStream(filename));

```

```

53     }
54     catch (java.io.FileNotFoundException e) {
55         System.out.println("File " + filename + " not found.");
56     return;
57 }
58 }

59     parser.debug_recovery = debug_recovery;
60     if (! debug_as) parser.disable_tracing(); // desab. verbose do AS
61     try {
62         parser.program(); // chama o método que faz a análise
63     }
64     catch (ParseException e)
65     {
66         System.err.println(e.getMessage());
67     }
68     finally {
69         System.out.println(parser.token_source.foundLexError() +
70                             " Lexical Errors found");
71         System.out.println(parser.contParseError +
72                             " Syntactic Errors found");
73     }
74 }

75 }

76 } // main

77

78 static public String im(int x)
79 {
80     int k;
81     String s;
82     s = tokenImage[x];
83     k = s.lastIndexOf("\\\"");
84     try {s = s.substring(1,k);}
85     catch (StringIndexOutOfBoundsException e)
86     {}
87     return s;
88 }

89

90

91 boolean eof; // variável que indica se EOF foi alcançado
92 // o método abaixo consome tokens ate alcançar um que pertença
93 // ao conjunto de sincronização
94
95 void consumeUntil(RecoverySet g,
96                     ParseException e,
97                     String met) throws ParseException,
98                     ParseException
99 {
100     Token tok;
101
102     if ( debug_recovery) // informação sobre a recuperação
103     {
104         System.out.println();
105         System.out.println("/** " + met + " **");

```

```

106     System.out.println("      Syncronizing Set: " + g);
107 }
108
109 if (g == null) throw e; // se o conjunto é null, propaga a exceção
110
111 tok = getToken(1); // pega token corrente
112 while ( ! eof ) // se não chegou ao fim do arquivo
113 {
114     if ( g.contains(tok.kind) ) // achou um token no conjunto
115     {
116         if ( debug_recovery )
117             System.out.println("      Found syncronizing token: " +
118                             im(tok.kind));
119         break;
120     }
121     if (debug_recovery)
122         System.out.println("      Ignoring token: " + im(tok.kind));
123     getNextToken(); // pega próximo token
124     tok = getToken(1);
125     if (tok.kind == EOF && ! g.contains(EOF) ) // fim da entrada
126         eof = true;
127 }
128 System.out.println(e.getMessage());
129 contParseError++; // incrementa número de erros
130 if ( eof ) throw new ParseEOFException("EOF found prematurely.");
131 }
132
133 } // langX
134
135 PARSER_END(langX)

```

O leitor pode notar que, além da inclusão da nova opção, diversas outras alterações foram introduzidas. Elas serão explicadas à medida que formos estudando a nova implementação do AS.

Podemos fazer a recuperação de erros dos não-terminalis fazendo com que cada um conheça, a princípio, qual é o seu conjunto de sincronização (como no primeiro exemplo da seção 5.1) ou podemos passar esses conjuntos como parâmetros (segundo exemplo). Por considerá-la mais flexível, utilizaremos a segunda abordagem no nosso compilador. O primeiro não-terminal do nosso AS é mostrado no programa 5.4

Programa 5.4

```

1 void program() throws ParseEOFException :
2 {
3     RecoverySet g = new RecoverySet(EOF);
4 }
5 {
6     try {
7         [ classlist(g) ] <EOF>
8     }
9     catch (ParseException e)
10    {

```

```

11     consumeUntil(g, e, "program");
12 }
13 }
```

As alterações feitas são bastante significativas. Vamos estudá-las com calma, iniciando “de dentro para fora”. A produção foi colocada dentro da construção `try/catch` e, portanto, se alguma exceção for propagada pelo método `classlist` ou se não tivermos um `EOF` após a lista de classes, o código dentro do `catch` será executado. Importante: esse código é Java, e não produções na BNF. No caso, fazemos chamada ao método `consumeUntil` que foi incluído na seção inicial do arquivo `.jj` e, portanto, incluído como um método da classe `langX`. Esse método é mostrado no programa 5.3.

Vamos analisá-lo por partes. Inicialmente temos seus parâmetros, que são:

```

void consumeUntil(RecoverySet g,
                  ParseException e,
                  String met) throws ParseException,
                  ParseException
```

O parâmetro `g` é o conjunto de sincronização a ser utilizado. A classe `RecoverySet` que foi definida dentro do pacote `recovery`, é importada no início da classe `langX`. Essa classe é mostrada a seguir. Não há muito a comentar a respeito dela, apenas que implementa um conjunto em que cada elemento é um objeto do tipo `java.lang.Integer`, que representa um tipo de token (`CLASS` ou `IDENT`, etc). Os seus métodos representam as operações normais sobre conjuntos como união, pertinência de um elemento etc.

Programa 5.5

```

1 package recovery;
2
3 import java.util.*;
4 import langX;
5
6
7 public class RecoverySet extends HashSet {
8
9     public RecoverySet() // cria conjunto vazio
10    {
11        super();
12    }
13
14    public RecoverySet(int t) // cria conjunto com um tipo de token
15    {
16        this.add(new Integer(t));
17    }
18
19    public boolean contains(int t) // verifica se token pertence ao conjunto
20    {
21        return super.contains(new Integer(t));
22    }
23}
```

```

23
24 // faz a união de dois conjuntos
25 public RecoverySet union(RecoverySet s)
26 {
27 RecoverySet t = null;
28
29     if (s != null) // se s == null retorna null
30     {
31         t = (RecoverySet) this.clone();
32         t.addAll(s);
33     }
34     return t; // retorna um terceiro conjunto, sem destruir nenhum
35 }
36
37 public RecoverySet remove(int n) // retira um elemento do conjunto
38 {
39     RecoverySet t = (RecoverySet) this.clone();
40     t.remove(new Integer(n));
41     return t; // retorna um novo conjunto, sem um dos elementos
42 }
43
44 // cria string descrevendo os tokens que pertencem ao conjunto
45 public String toString()
46 {
47 Iterator it = this.iterator();
48 String s = "";
49 int k;
50
51     while ( it.hasNext() )
52     {
53         k = ((Integer) it.next()).intValue();
54         s += langX.im(k) + " ";
55     }
56     return s;
57 }
58 }

```

O segundo parâmetro do método *consumeUntil* é um objeto do tipo *ParseException*, que descreve qual foi o erro sintático que ocorreu. Esse parâmetro é extraído do catch no método que chama o *consumeUntil*. E o último parâmetro é o nome do não-terminal que está fazendo a recuperação de erros. No caso do não-terminal inicial da nossa gramática, temos o conjunto de sincronização formado por um único token, que é o *EOF*, e o terceiro parâmetro é obviamente um string com o nome do não-terminal "program".

A segunda parte do *consumeUntil* dá algumas informações sobre a recuperação sendo processada. Mostra qual é o não-terminal onde está sendo feita a recuperação e qual o conjunto de sincronização utilizado. A variável *debug_recovery* é uma variável da classe *langX* inicializada no método *main*, de acordo com a existência ou não da opção *-debug_reco*.

Em seguida, verifica-se se o conjunto de sincronização é `null`. Se for, isso indica que quem chamou o método desse não-terminal não deseja que este faça a recuperação de erros e, então, a exceção é simplesmente lançada novamente. Olhando a assinatura do método `consumeUntil`, o leitor irá reparar que, além de `ParseException`, ele pode também lançar uma exceção do tipo `ParseEOFException`. Isso irá ocorrer quando, ao tentar achar um token do conjunto de sincronização, o método chegar ao final da entrada. Nesse caso, nenhuma recuperação mais é possível e essa exceção não é tratada por nenhum dos métodos correspondentes aos não-terminais da gramática, sendo propagado até o método `main` que foi quem iniciou a análise sintática. Note que, no `main`, não temos mais o tratamento de uma `ParseException`, como na versão anterior do nosso compilador, mas, sim, de uma `ParseEOFException` (Programa 5.3, linha 65).

A parte seguinte do método é quem faz a busca de um token na entrada que case com algum token no conjunto de sincronização. Cada token consumido é mostrado, caso essa opção esteja habilitada. Para finalizar, a mensagem correspondente ao erro sintático é emitida e o número de erros, incrementado. A variável `eof` controla se o fim do arquivo já foi atingido. Caso afirmativo e caso o token `EOF` não faça parte do conjunto de sincronização, então a exceção `ParseEOFException` é lançada.

Voltando ao não-terminal `program`, vemos que existe a declaração da variável `g` que é utilizada como conjunto de sincronização para esse não-terminal e também para a chamada a `classlist`. Vemos, ainda, que a exceção `ParseEOFException` foi incluída na lista das exceções que esse método pode lançar. Não precisamos incluir nessa lista a `ParseException`, pois o JavaCC considera que todos os métodos dos não-terminais podem lançar essa exceção e já a inclui na lista.

Antes de prosseguirmos mostraremos um pequeno exemplo de como esse não-terminal se recupera de um erro sintático. Na verdade, nesse caso não podemos dizer exatamente que ele se recupere. Como seu conjunto de sincronização é um `EOF`, o que ele faz é, ao encontrar um erro, consumir tudo até o final do arquivo. Vamos utilizar o seguinte programa:

```
class test {
}

text that is not a "class"

class test2 {
```

Ao analisarmos esse programa teremos a primeira classe `test` reconhecida como uma `classdecl` e como uma `classlist`. Porém, ao encontrar o identificador `text`, o AS decide não procurar mais por uma `classlist` e retorna o controle para `program`, que deveria achar um `EOF`. Como este não é encontrado, caracteriza-se um erro sintático e a recuperação é posta em ação. Como dito, essa recuperação é simplesmente passar por cima de todos os demais tokens, até o fim do arquivo. Esse comporta-

mento é mostrado ao executarmos nosso compilador com esse programa e a opção `-debug_recovery`:

```
java parser.langX -debug_recovery ../ssamples/program.x
```

```
*** program ***
Synchronizing Set: <EOF>
Ignoring token: <IDENT>
Ignoring token: <string_constant>
Ignoring token: class
Ignoring token: <IDENT>
Ignoring token: {
Ignoring token: }
Found synchronizing token: <EOF>
Encountered "text" at line 5, column 1.
Was expecting one of:
<EOF>
"class" ...
```

0 Lexical Errors found
1 Syntactic Errors found

A parte inicial mostra como o não-terminal *program* tratou o erro. São apresentados o conjunto de sincronização e, depois, cada um dos tokens consumidos, até se encontrar um que pertencia ao conjunto. Em seguida é mostrada a mensagem correspondente ao erro encontrado. Na próxima seção, iremos discutir alguns problemas relacionados aos conjuntos de sincronização. Veremos como podemos tentar melhorar a recuperação do nosso AS para evitar tratamentos não muito apropriados, como o do não-terminal *program*. Antes disso, vamos prosseguir e mostrar como ficam os demais não-terminais.

O não-terminal *classlist* não muda muito em relação ao definido anteriormente:

Programa 5.6

```
1 void classlist(RecoverySet g) throws ParseEOFException :
2 {
3 RecoverySet f = First.classlist.union(g);
4 }
5 {
6   classdecl(f) [ classlist(g) ]
7 }
```

Ele chama o *classdecl* passando como conjunto de sincronização o conjunto *g*, que recebeu como parâmetro, e mais todos os terminais que estão no *FIRST(classlist)*. Isso

porque depois do *classdecl* podemos ter um *classlist* ou, então, o final da aplicação dessa produção. Nesse último caso, o conjunto de sincronização deve ser o mesmo do não-terminal corrente. Situação semelhante ocorre na chamada para *classlist*. Seu conjunto de sincronização é igual ao do método corrente. Isso se justifica, pois se ocorrer um erro dentro da chamada mais interna, o não-terminal que foi chamado deverá procurar um terminal que não está definido na produção corrente, mas, sim, numa produção do não-terminal que fez a chamada.

Utilizamos nesse não-terminal o conjunto *First.classlist*. Esse conjunto foi definido dentro do pacote *recovery*. A classe *first* é uma classe que possui diversas variáveis estáticas que representam alguns dos conjunto *FIRST* que usaremos para a recuperação de erros. No programa 5.7, mostramos a implementação dessa classe. Ela simplesmente define esses conjuntos que podem, então, ser utilizados pelo AS por meio do nome da classe e da variável, como o método *classlist* que utiliza *First.classlist*.

Programa 5.7

```

1 package recovery;
2
3 import java.util.*;
4 import langXConstants;
5
6 //implementa os conjuntos first p/ alguns n.terminais
7 public class first {
8     static public final RecoverySet methoddecl = new RecoverySet();
9     static public final RecoverySet vardecl = new RecoverySet();
10    static public final RecoverySet classlist = new RecoverySet();
11    static public final RecoverySet constructdecl = new RecoverySet();
12    static public final RecoverySet statlist = new RecoverySet();
13    static public final RecoverySet program = classlist;
14
15    static {
16        methoddecl.add(new Integer(langXConstants.INT));
17        methoddecl.add(new Integer(langXConstants.STRING));
18        methoddecl.add(new Integer(langXConstants.IDENT));
19
20        vardecl.add(new Integer(langXConstants.INT));
21        vardecl.add(new Integer(langXConstants.STRING));
22        vardecl.add(new Integer(langXConstants.IDENT));
23
24        classlist.add(new Integer(langXConstants.CLASS));
25
26        constructdecl.add(new Integer(langXConstants.CONSTRUCTOR));
27
28        statlist.addAll(vardecl);
29        statlist.add(new Integer(langXConstants.IDENT));
30        statlist.add(new Integer(langXConstants.PRINT));
31        statlist.add(new Integer(langXConstants.READ));
32        statlist.add(new Integer(langXConstants.RETURN));
33        statlist.add(new Integer(langXConstants.SUPER));
34        statlist.add(new Integer(langXConstants.IF));
35        statlist.add(new Integer(langXConstants.FOR));
36        statlist.add(new Integer(langXConstants.LBRACE));

```

```

37     statlist.add(new Integer(langXConstants.BREAK));
38     statlist.add(new Integer(langXConstants.SEMICOLON));
39
40
41 }
42
43 }

```

O leitor deve ter notado que no não-terminal *classlist* não existe recuperação de erro. Isso acontece porque não existe possibilidade de ocorrer erro sintático dentro desse não-terminal. A chamada ao método do não-terminal *classdecl* não propaga nenhuma exceção do tipo *ParseException*, pois esse método faz tratamento de erros sintáticos, como veremos a seguir. Ao retornar daquela chamada, o AS irá verificar se o token corrente está ou não no *FIRST(classlis)*. Em caso afirmativo, esse não-terminal será chamado e não deve também gerar nenhuma exceção. Em caso negativo, o método simplesmente retorna o controle para aquele que o chamou. Em ambos os casos, certamente não ocorre erro sintático.

Já o não-terminal *classdecl* foi alterado para fazer tratamento de erro sintático, como mostra o programa 5.8.

Programa 5.8

```

1 void classdecl(RecoverySet g) throws ParseEOFException :
2 {
3 }
4 {
5 try {
6     <CLASS> <IDENT> [ <EXTENDS> <IDENT> ] classbody(g)
7 }
8 catch (ParseException e)
9 {
10     consumeUntil(g, e, "classdecl");
11 }
12 }

```

Para esse não-terminal, não há muito o que comentar. No caso de erro, o tratamento é feito normalmente como já visto e a chamada a *classbody* utiliza o mesmo conjunto de sincronização, pois, por estar no final da produção, os não-terminais esperados após essa chamada são exatamente os mesmos do não-terminal corrente.

O não-terminal *classbody* é um pouco mais elaborado, em termos de recuperação de erros. Vejamos o programa 5.9.

Programa 5.9

```

1 void classbody(RecoverySet g) throws ParseEOFException :
2 {
3     RecoverySet f1 = new RecoverySet(RBRACE),
4         f2 = new RecoverySet(SEMICOLON),
5         f3 = First.methoddecl.union(f1),
6         f4 = First.constructdecl.union(f3),

```

```

7         f5 = First.vardecl.union(f4);
8     }
9     {
10    try {
11        <LBRACE>
12        [classlist(f5)]
13        (LOOKAHEAD(3) vardecl(f2) <SEMICOLON>)*
14        (constructdecl(f4))*  

15        (methoddecl(f3))*  

16    <RBRACE>
17  }
18  catch (ParseException e)
19  {
20      consumeUntil(g, e, "classbody");
21  }
22 }
```

Vamos analisá-lo iniciando pelo final da produção. Os tokens que podem vir após a chamada *methoddecl* são o fecha chave ou o início de uma nova definição de método, já que essa chamada pode ser repetida inúmeras vezes. Por isso, seu conjunto de sincronização é *f3*, variável à qual foi atribuído um conjunto formado por *First.methoddecl* unido com {*RBRACE*}.

Algo semelhante acontece com a chamada anterior, ao método *constructdecl*. Como essa chamada também pode ocorrer inúmeras vezes, *FIRST(constructdecl)* deve estar no seu conjunto de sincronização. Além disso, podemos ter após essa chamada uma declaração de método e, portanto, *FIRST(methoddecl)* deve fazer parte do conjunto de sincronização, assim como *RBRACE*, pois uma definição de método pode ou não aparecer. Assim, o conjunto de sincronização é a união de *f3* com *First.constructdecl*, atribuído à variável *f4*.

Já a chamada a *vardecl* é necessariamente seguida por um ponto-e-vírgula e, por isso, seu conjunto de sincronização é formado por esse único não-terminal. E, finalmente, a chamada a *classlist* usa como sincronizadores todos os símbolos que estão em *f4* mais os que estão em *First.vardecl*, que são os símbolos de *FIRST(vardecl)*.

No não-terminal *vardecl*, não há nenhuma chamada a métodos de outros não-terminais, apenas a recuperação de erros normal, mostrada no programa 5.10

Programa 5.10

```

1 void vardecl(RecoverySet g) throws ParseEOFException :
2 {
3 }
4 {
5     try {
6         (<INT> | <STRING> | <IDENT> )
7         <IDENT> ( <LBRACKET> <RBRACKET>)*
8         (<COMMA> <IDENT> ( <LBRACKET> <RBRACKET>)* )*
9     }
10    catch (ParseException e)
11    {
12        consumeUntil(g, e, "vardecl");
```

```
13  }
14 }
```

Para *constructdecl* e *methoddecl*, as únicas chamadas estão no final das produções e, portanto, somente os conjuntos de sincronização que foram recebidos como parâmetros nesses dois não-terminais são utilizados nas chamadas. É interessante notar que para a gramática que estamos utilizando não seria necessário que o método *constructdecl* fizesse recuperação de erros. Isso acontece porque esse método é chamado apenas dentro de *classbody*, num ponto de decisão, ou seja, que exige consulta ao lookahead para decidir-se se *constructdecl* deve ser chamado ou se outro caminho deve ser tomado. Assim, se a execução alcançou esse método é porque o token corrente é um *CONSTRUCTOR*. Dentro do método, esse token é consumido e, logo em seguida, uma chamada é feita a *methodbody*, que não propaga erros sintáticos. Assim, dentro de *constructdecl*, nenhum erro sintático precisa ser tratado. Por outro lado, manter a recuperação de erros não atrapalha em nada e permite que o não-terminal seja utilizado de maneira mais flexível, por exemplo sendo chamado diretamente de outros pontos do nosso programa, que não seja o não-terminal *classbody*.

O não-terminal *methodbody* é mostrado no programa 5.11. Possui duas chamadas a não-terminais. A primeira é feita a *paramlist* e utiliza o fecha parêntese como sincronizador. E a segunda, que vem no final da produção, utiliza o próprio parâmetro *g* como conjunto de sincronização.

```
1 void methodbody(RecoverySet g) throws ParseEOFException :
2 {
3     RecoverySet f = new RecoverySet(RPAREN);
4 }
5 {
6     try {
7         <LPAREN> paramlist(f) <RPAREN> statement(g)
8     }
9     catch (ParseException e)
10    {
11        consumeUntil(g, e, "methodbody");
12    }
13 }
```

O não-terminal *paramlist* não muda, a não ser pela inclusão do tratamento de erro (programa 5.12).

```
1 void paramlist(RecoverySet g) throws ParseEOFException :
2 {
3 }
4 {
5     try {
6         [
7             (<INT> | <STRING> | <IDENT>) <IDENT> (<LBRACKET> <RBRACKET>)*
```

```

8      ( <COMMA> (<INT> | <STRING> | <IDENT>) <IDENT>
9          (<LBRACKET> <RBRACKET>)*
10     )*
11   ]
12 }
13 catch (ParseException e)
14 {
15     consumeUntil(g, e, "paramlist");
16 }
17 }

```

No não-terminal *statement*, temos, na maioria dos casos, o token *SEMICOLON* como conjunto de sincronização. Nos casos em que a chamada a não-terminais aparece sozinha na produção, como em *ifstat*, utiliza-se o próprio conjunto de sincronização de *statement* como argumento da chamada. E no caso de *statlist*, utiliza-se o fecha chave, que é o delimitador da seqüência de comandos, como sincronizador. Temos, então, o código mostrado no programa 5.13

Programa 5.13

```

1 void statement(RecoverySet g) throws ParseEOFException :
2 {
3     RecoverySet f1 = new RecoverySet(SEMICOLON);
4     RecoverySet f2 = new RecoverySet(RBRACE);
5 }
6 {
7     try {
8         LOOKAHEAD(2)
9         vardecl(f1) <SEMICOLON>
10    |
11    atribstat(f1) <SEMICOLON>
12    |
13    printstat(f1) <SEMICOLON>
14    |
15    readstat(f1) <SEMICOLON>
16    |
17    returnstat(f1) <SEMICOLON>
18    |
19    superstat(f1) <SEMICOLON>
20    |
21    ifstat(g)
22    |
23    forstat(g)
24    |
25    <LBRACE> statlist(f2) <RBRACE>
26    |
27    <BREAK> <SEMICOLON>
28    |
29    <SEMICOLON>
30 }
31 catch (ParseException e)
32 {

```

```

33     consumeUntil(g, e, "statement");
34 }
35 }
```

Os demais não-terminais, cujas definições são apresentadas no programa 5.14 utilizam o mesmo raciocínio para determinar os conjuntos de sincronização das chamadas a outros não-terminais. Assim apenas apresentamos esses não-terminais, sem necessidade de nenhum comentário extra.

Programa 5.14

```

1 void atribstat(RecoverySet g) throws ParseEOFException :
2 {
3     RecoverySet f1 = new RecoverySet(ASSIGN);
4 }
5 {
6     try {
7         lvalue(f1) <ASSIGN> ( alocexpression(g) | expression(g))
8     }
9     catch (ParseException e)
10    {
11        consumeUntil(g, e, "atribstat");
12    }
13 }
14
15 void printstat(RecoverySet g) throws ParseEOFException :
16 {
17 }
18 {
19     try {
20         <PRINT> expression(g)
21     }
22     catch (ParseException e)
23    {
24        consumeUntil(g, e, "printstat");
25    }
26 }
27
28 void readstat(RecoverySet g) throws ParseEOFException :
29 {
30 }
31 {
32     try {
33         <READ> lvalue(g)
34     }
35     catch (ParseException e)
36    {
37        consumeUntil(g, e, "readstat");
38    }
39 }
40
41 void returnstat(RecoverySet g) throws ParseEOFException :
```

```

43 {
44 }
45 {
46 try {
47     <RETURN> [expression(g)]
48 }
49 catch (ParseException e)
50 {
51     consumeUntil(g, e, "returnstat");
52 }
53 }

54
55
56 void superstat(RecoverySet g) throws ParseEOFException :
57 {
58 RecoverySet f = new RecoverySet(RPAREN);
59 }
60 {
61 try {
62     <SUPER> <LPAREN> arglist(f) <RPAREN>
63 }
64 catch (ParseException e)
65 {
66     consumeUntil(g, e, "superstat");
67 }
68 }

69
70 void ifstat(RecoverySet g) throws ParseEOFException :
71 {
72 RecoverySet f1 = new RecoverySet(RPAREN),
73         f2 = new RecoverySet(ELSE).union(g);
74 }
75 {
76 try {
77     <IF> <LPAREN> expression(f1) <RPAREN> statement(f2)
78     [LOOKAHEAD(1) <ELSE> statement(g)]
79 }
80 catch (ParseException e)
81 {
82     consumeUntil(g, e, "ifstat");
83 }
84 }

85
86
87
88 void forstat(RecoverySet g) throws ParseEOFException :
89 {
90 RecoverySet f1 = new RecoverySet(SEMICOLON),
91         f2 = new RecoverySet(RPAREN);
92 }
93 {
94 try {
95     <FOR> <LPAREN> [atribstat(f1)] <SEMICOLON>

```

```

96      [expression(f1)] <SEMICOLON>
97      [atribstat(f2)] <RPAREN>
98      statement(g)
99  }
100 catch (ParseException e)
101 {
102     consumeUntil(g, e, "forstat");
103 }
104 }
105
106 void statlist(RecoverySet g) throws ParseEOFException :
107 {
108     RecoverySet f = First.statlist.union(g);
109 }
110 {
111     statement(f) [statlist(g)]
112 }

```

Já no não-terminal *lvalue*, programa 5.15, temos uma novidade. Note o leitor que os dois não-terminais que são chamados dentro de *lvalue* não utilizam conjuntos de sincronização, ou melhor, o valor passado como argumento nessa chamada é *null*. Isso significa que os métodos *expression* e *arglist* não devem fazer recuperação de erro e que caso ocorra um erro sintático na execução desses métodos, a exceção correspondente a esse erro deve ser propagada “para cima” até o *lvalue* que irá tratar o erro. Isso é feito porque *lvalue* aparece, na maioria das vezes, dentro de uma expressão ou de um comando e, em geral, corresponde a um número pequeno de tokens como *a[i]* ou *a.b.c[d+e]*.

Programa 5.15

```

1 void lvalue(RecoverySet g) throws ParseEOFException :
2 {
3 }
4 {
5     try {
6         <IDENT> (
7             <LBRACKET> expression(null) <RBRACKET> |
8             <DOT> <IDENT> [<LPAREN> arglist(null) <RPAREN>]
9         )*
10    }
11    catch (ParseException e)
12    {
13        consumeUntil(g, e, "lvalue");
14    }
15 }

```

Assim, podemos considerar que se um erro ocorre dentro da chamada ao método *expression* dentro de *lvalue*, todo o não-terminal *lvalue* deve ser considerado inválido e a recuperação deve ser feita a partir desse não-terminal, evitando uma granularidade muito fina na recuperação. Com efeito, se tivermos a seguinte entrada:

```
a.b.c[d + * e][i] = 10;
```

teremos uma chamada a *lvalue* com o conjunto de sincronização igual a *{ASSIGN}*, que faz uma chamada a *expression*. Se *expression* fizesse recuperação de erro, esta seria feita até o primeiro fecha colchete e a análise continuaria. Mas isso não é necessário. Vamos considerar que se um erro ocorreu na expressão entre colchetes, todo o lado esquerdo da atribuição está errado e a recuperação é feita apenas por *lvalue*, consumindo tudo até o *ASSIGN* e retornando o controle para *atribstat* que trata do resto do comando. Isso evita que dois erros apareçam muito perto um do outro, ou seja, dentro de uma mesma expressão.

Algo semelhante será feito para aqueles não-terminais que estão “abaixo” de *expression* como *numexpr*, *term* e *factor*. Esses não-terminais nunca fazem recuperação de erro sintático, ficando esta sempre por conta do método *expression*. Por isso, eles nem sequer possuem um parâmetro formal como os demais não-terminais. É o que mostra o programa 5.16

Programa 5.16

```

1 void aloceexpression(RecoverySet g) throws ParseEOFException :
2 {
3     RecoverySet f1 = new RecoverySet(RPAREN),
4                 f2 = new RecoverySet(RBRACKET);
5 }
6 {
7     <NEW> (
8         LOOKAHEAD(2) <IDENT> <LPAREN> arglist(f1) <RPAREN> |
9         (<INT> | <STRING> | <IDENT> )
10        (<LBRACKET> expression(f2) <RBRACKET>)+
11    )
12 }
13
14
15 void expression(RecoverySet g) throws ParseEOFException :
16 {
17 }
18 {
19     try {
20         numexpr() [(<LT> | <GT> | <LE> | <GE> | <EQ> | <NEQ>) numexpr()]
21     }
22     catch (ParseException e)
23     {
24         consumeUntil(g, e, "expression");
25     }
26 }
27
28
29 void numexpr() throws ParseEOFException :
30 {
31 }
32 {
33     term() ((<PLUS> | <MINUS>) term())*
34 }
```

```

35
36 void term() throws ParseEOFException :
37 {
38 }
39 {
40     unaryexpr() ((<STAR> | <SLASH>| <REM>) unaryexpr())*
41 }
42
43 void unaryexpr() throws ParseEOFException :
44 {
45 }
46 {
47     [<PLUS> | <MINUS>] factor()
48 }
49
50
51 void factor() throws ParseEOFException :
52 {
53 }
54 {
55     (
56         <int_constant> |
57         <string_constant> |
58         <null_constant> |
59         lvalue(null) |
60         <LPAREN> expression(null) <RPAREN>
61     )
62 }
63
64 void arglist(RecoverySet g) throws ParseEOFException :
65 {
66     RecoverySet f = new RecoverySet(COMMA).union(g);
67 }
68 {
69     [expression(f) (<COMMA> expression(f))*]
70 }


---



```

Assim, acabamos de ver a definição de todos os não-terminais, acrescidas da recuperação de erros. Na próxima seção, veremos alguns exemplos e tentaremos algumas formas de melhorar a recuperação de erros apresentada aqui.

5.3 Algumas possíveis melhorias

Vimos que é bastante simples calcular o conjunto de sincronização para uma chamada de um não-terminal da nossa gramática. Porém, o resultado obtido nem sempre é satisfatório. Iremos nesta seção apresentar algumas formas de melhorar esse método de recuperação, analisando com mais detalhes o conjunto de sincronização de alguns dos não-terminais. É preciso entender, porém, que a escolha dos conjuntos de sincronização depende bastante da gramática que se utiliza e que uma estratégia que pode

apresentar bons resultados para algumas gramáticas pode não apresentar os mesmos resultados para outras.

O primeiro problema da recuperação de erro foi visto na seção 5.2. É o caso do programa:

```
class test {
}

text that is not a "class"

class test2 {
}
```

onde um erro é apontado ao encontrar a palavra *text* e a recuperação de erros, do método *program*, ignora todos os tokens até encontrar um *EOF*. A primeira observação a ser feita é que não faz muito sentido utilizar o *EOF* como símbolo de sincronização, visto que estaremos sempre consumindo todo o restante da entrada, que não será analisada.

Vamos alterar o não-terminal *program* de modo que, após reconhecer um *classlist* na entrada, ele tente achar o *EOF*, mas se não conseguir, deverá fazer a resincronização e voltar a analisar o programa. Então, esse não-terminal fica como o apresentado no programa 5.17

Programa 5.17

```
1 void program() throws ParseEOFException :
2 {
3     RecoverySet g = First.program;
4 }
5 {
6     <EOF>
7 |
8     classlist(g)
9     try {
10         <EOF>
11     }
12     catch (ParseException e)
13     {
14         consumeUntil(g, e, "program");
15     }
16     [ program() ]
17 }
```

Agora esse método chama *classlist*, que, como já comentamos, nunca propaga uma exceção *ParseException* e, por isso, não precisa estar dentro de um *try/catch*. Em seguida, tenta achar na entrada um *EOF*. Se conseguir, então o método termina, pois *EOF* não pertence ao *FIRST(program)* e a chamada ao próprio não-terminal no final

do método não é feita. Se não conseguir, isso significa que algum token estranho está presente na entrada. Então, uma mensagem de erro é emitida e a resincronização é feita, nos mesmos moldes vistos anteriormente. Em seguida, o próprio não-terminal *program* é invocado para terminar a análise da entrada, isso se a resincronização foi realizada com sucesso. Se não foi bem-sucedida, então o final da entrada foi atingido durante o tratamento de erro e uma exceção *ParseEOFException* foi lançada e será propagada.

Nesse novo tratamento que demos ao não-terminal *program*, utilizamos o par *try/catch* do JavaCC não sobre a produção toda do não-terminal, mas apenas em uma parte dela. Usando esse artifício, o implementador pode saber exatamente em que ponto do método ocorreu o erro sintático. Por outro lado, podemos notar que a definição do não-terminal torna-se mais complicada e nem um pouco elegante. Cabe, portanto, ao implementador avaliar o quanto importante é a recuperação de erros para o seu compilador e em que pontos vale a pena empregar esse artifício.

Um outro exemplo de problemas que podem aparecer na nossa recuperação de erros é mostrada pelos exemplos a seguir. No primeiro, a recuperação é feita de maneira satisfatória, pelo método *statement*.

```
class test {
    int [] m(int a)
    {
        int b,c
    }
}

class test2 { }
```

A primeira declaração é reconhecida como uma *vardecl*, mas pela falta do ponto-e-vírgula um erro ocorre. A recuperação, dentro de *statement*, procura um símbolo que esteja no conjunto de sincronização, o qual inclui o fecha chave, e, então, retorna o controle para *statlist* que consome o fecha chave. Porém, se o programa fosse:

```
class test {
    int [] m(int a)
    {
        int b,c,
    }
}

class test2 { }
```

a recuperação não seria tão eficiente. Nesse caso, o erro ocorreu dentro de *vardecl* cujo conjunto de sincronização é apenas um ponto-e-vírgula. Ao tentar recuperar o erro, o método *vardecl* consome todos os tokens da entrada, sem encontrar o desejado. Isso certamente não é desejado. Para melhorar essa situação, podemos alterar *statement*

passando como parâmetro para *vardecl* não só o ponto-e-vírgula, que é esperado após a declaração de variável, mas também o próprio conjunto de sincronização passado como parâmetro para *statement*. Como resultado, *vardecl* iria parar a sincronização no fecha chave. Teríamos:

```
cap05$ java parser.langX -debug_recovery ..ssamples/statement2.x
X++ Compiler - Version 1.0 - 2004
Reading from file ..ssamples/statement2.x . . .

*** vardecl ***
Synchronizing Set: ; } { <IDENT> super string return read print
int if for break
Found synchronizing token: }
Encountered ")" at line 5, column 1.
Was expecting:
<IDENT> ...

*** statement ***
Synchronizing Set: ; } { <IDENT> super string return read print
int if for break
Found synchronizing token: }
Encountered ")" at line 5, column 1.
Was expecting:
";" ...

0 Lexical Errors found
2 Syntactic Errors found
```

Note que apesar da recuperação de erro realizada por *vardecl*, ao retornar o controle para *statement*, um outro erro ocorreu, pois o token esperado nesse não-terminal era um ponto-e-vírgula, mas a sincronização feita foi no fecha chave. Esse segundo erro é tratado em *statement* e a sincronização é feita no mesmo token, ou seja, no fecha chave. Então o controle volta para *statlist* que consome esse token. Em geral é preferível que tenhamos duas ou mais mensagens de erro, mas que consigamos uma recuperação adequada. Além disso, podemos alterar o nosso método *consumeUntil* de modo que não sejam emitidas mensagens distintas para um mesmo erro. Para isso, devemos registrar qual é o token corrente quando uma mensagem é emitida e só emitir uma nova mensagem se algum token for consumido. Teríamos, então, naquele método, o trecho de código

```
System.out.println(e.getMessage());
contParseError++; // incrementa número de erros
```

substituído por

```
if ( tok != lastError)
{
    System.out.println(e.getMessage());
    lastError = tok;
```

```

    contParseError++; // incrementa número de erros
}

```

Essa mesma regra de utilizar o conjunto corrente de sincronização do não-terminal *statement* nas chamadas dentro desse método serve também para os outros não-terminais, e iremos aplicá-la em seguida. Antes, analisaremos um outro caso. Existem alguns tokens que são bons candidatos a sincronizadores como as palavras-chave que iniciam comando, construtores ou classes. Devemos, sempre que possível e coerente, utilizá-los nos conjuntos de sincronização. Outros porém, não são tão bons, pois aparecem em diversas situações diferentes dentro do programa, e devemos evitá-los. É o caso do *IDENT*, que pode ser utilizado como nome de classe, de variável, como tipo de variável etc. Por exemplo, se tivermos um programa como

```

class test {
int [] m(int a)
{
    int a b,c;
}
}

```

teríamos uma má recuperação, visto que há um erro dentro de *statement*, pois o ponto-e-vírgula depois de *vardecl* não foi encontrado e a sincronização ocorreria ao achar o identificador *b*, e a partir desse token seria feita a tentativa de ser reconhecer outro comando no não-terminal *statlist*, o que conduziria a outros erros. Portanto, *IDENT* não é um bom token para sincronização em *statement*. Vamos deixá-lo de fora e confiar em outros tokens, como ponto-e-vírgula, *int*, *if* etc. Então, os não-terminais *statlist* e *statement* ficam como mostrado no programa 5.18

Programa 5.18

```

1 void statlist(RecoverySet g) throws ParseEOFException :
2 {
3     RecoverySet f = First.statlist.remove(IDENT).union(g);
4 }
5 {
6     statement(f) [statlist(g)]
7 }
8
9
10 void statement(RecoverySet g) throws ParseEOFException :
11 {
12     RecoverySet f1 = new RecoverySet(SEMICOLON).union(g).remove(IDENT);
13     RecoverySet f2 = new RecoverySet(RBRACE).union(g).remove(IDENT);}
14 {
15     try {
16         LOOKAHEAD(2)
17         vardecl(f1) <SEMICOLON>
18     |
19         atribstat(f1) <SEMICOLON>

```

```

20  |
21      printstat(f1) <SEMICOLON>
22  |
23      readstat(f1) <SEMICOLON>
24  |
25      returnstat(f1) <SEMICOLON>
26  |
27      superstat(f1) <SEMICOLON>
28  |
29      ifstat(g)
30  |
31      forstat(g)
32  |
33      <LBRACE> statlist(f2) <RBRACE>
34  |
35      <BREAK> <SEMICOLON>
36  |
37      <SEMICOLON>
38  }
39 catch (ParseException e)
40 {
41     consumeUntil(g, e, "statement");
42 }
43 }
```

*

Outro caso complicado para a recuperação de erros é o não-terminal *classbody*. Um dos problemas é a utilização do *LOOKAHEAD*. Esse comando faz com que qualquer erro na definição de variáveis, que ocorra nos três primeiros tokens dessa definição, leve o AS a seguir outro caminho, tentando reconhecer um *constructdecl* ou *methoddecl*. Assim, a recuperação de erro, que poderia ser feita em *vardecl*, não será feita, pois esse método não é sequer chamado. É o caso, por exemplo, do programa

```

class test {
    int a b;
    construct () ;
}
```

O erro na declaração *int a b* faz com que o AS, baseado no lookahead de 3 tokens, considere que o método a ser chamado é *methoddec*, e não *vardecl*. Com isso, um erro ocorre nessa declaração e na declaração posterior, pois um construtor não é esperado após a declaração de um método. Esse problema não apresenta solução simples e teremos que conviver com esse defeito no nosso AS. Porém, algumas outras melhorias podem ser feitas nesse não-terminal, de acordo com o que mostra o programa 5.19

```

1 void classbody(RecoverySet g) throws ParseEOFException :
2 {
3     RecoverySet f2 = new RecoverySet(SEMICOLON).union(g).remove(IDENT),
4         f3 = First.methoddecl.union(g).remove(IDENT),
5         f4 = First.constructdecl.union(f3).remove(IDENT),
6         f5 = First.vardecl.union(f4).remove(IDENT);
7 }
8 {
9     try {
10         <LBRACE>
11         [classlist(f5)]
12         (LOOKAHEAD(3) vardecl(f2) <SEMICOLON>)*
13         (constructdecl(f4))*  

14         (methoddecl(f3))*  

15     <RBRACE>
16 }
17 catch (ParseException e)
18 {
19     consumeUntil(g, e, "classbody");
20 }
21 }
```

*

Aqui retiramos o *IDENT* dos conjuntos de sincronização utilizados dentro de *classbody*. Além disso, retiramos o fecha chave do conjunto de sincronização da chamada a *methoddecl*, pois é muito comum que um método termine também com um fecha chave. Se isso ocorrer, esse fecha chave (do método) será encontrado na sincronização e, mesmo que depois da declaração do método venha um outro método, o AS considerará que o fecha chave encontrado terminou a definição da classe e não chamará novamente o método *methoddecl*. Isso ocasiona um desastre na recuperação de erros. É o exemplo de:

```

class teste {
    int m 1()
    {
    ;
}
int m2()
{
;
}
```

Eliminando o fecha chave do conjunto de sincronização de *methoddecl*, esse método se recupera ao encontrar o token *int* e, dentro de *classbody*, uma nova chamada é feita a *methoddecl*, que irá corretamente reconhecer o método *m2*. Se a sincronização fosse feita com o fecha chave, o AS decidiria que a declaração da classe terminou e todo o restante da entrada seria ignorado.

Aplicando agora a regra de utilizar o conjunto de sincronização do não-terminal nas chamadas dentro dele, ficamos com as definições mostradas no programa 5.20 para os demais não-terminais (só mostrados os não-terminais que foram modificados):

Programa 5.20

```

1 void methodbody(RecoverySet g) throws ParseEOFException :
2 {
3     RecoverySet f = new RecoverySet(RPAREN).union(g);
4 }
5 {
6     try {
7         <LPAREN> paramlist(f) <RPAREN> statement(g)
8     }
9     catch (ParseException e)
10    {
11        consumeUntil(g, e, "methodbody");
12    }
13 }

14
15 void atribstat(RecoverySet g) throws ParseEOFException :
16 {
17     RecoverySet f1 = new RecoverySet(ASSIGN).union(g);
18 }
19 {
20     try {
21         lvalue(f1) <ASSIGN> ( alocexpression(g) | expression(g))
22     }
23     catch (ParseException e)
24    {
25        consumeUntil(g, e, "atribstat");
26    }
27 }

28
29 void superstat(RecoverySet g) throws ParseEOFException :
30 {
31     RecoverySet f = new RecoverySet(RPAREN).union(g);
32 }
33 {
34     try {
35         <SUPER> <LPAREN> arglist(f) <RPAREN>
36     }
37     catch (ParseException e)
38    {
39        consumeUntil(g, e, "superstat");
40    }
41 }

42
43 void ifstat(RecoverySet g) throws ParseEOFException :
44 {
45     RecoverySet f1 = new RecoverySet(RPAREN).union(g),
46                 f2 = new RecoverySet(ELSE).union(g);
47 }
48 {

```

```

49 try {
50     <IF> <LPAREN> expression(f1) <RPAREN> statement(f2)
51     [LOOKAHEAD(1) <ELSE> statement(g)]
52 }
53 catch (ParseException e)
54 {
55     consumeUntil(g, e, "ifstat");
56 }
57 }
58
59
60 void forstat(RecoverySet g) throws ParseEOFException :
61 {
62     RecoverySet f1 = new RecoverySet(SEMICOLON).union(g),
63         f2 = new RecoverySet(RPAREN).union(g);
64 }
65 {
66     try {
67         <FOR> <LPAREN> [atribstat(f1)] <SEMICOLON>
68             [expression(f1)] <SEMICOLON>
69                 [atribstat(f2)] <RPAREN>
70                     statement(g)
71     }
72     catch (ParseException e)
73     {
74         consumeUntil(g, e, "forstat");
75     }
76 }
77
78 void alocexpression(RecoverySet g) throws ParseEOFException :
79 {
80     RecoverySet f1 = new RecoverySet(RPAREN).union(g),
81         f2 = new RecoverySet(RBRACKET).union(g);
82 }
83 {
84     <NEW> (
85         LOOKAHEAD(2) <IDENT> <LPAREN> arglist(f1) <RPAREN> |
86         ( <INT> | <STRING> | <IDENT> )
87             (<LBRACKET> expression(f2) <RBRACKET>)*
88     )
89 }

```

É importante observar que aqueles não-terminais que aceitam como argumento o valor *null* devem passar para os métodos que são chamados dentro deles também esse valor, para que esses métodos mais internos não façam também a recuperação de erros. Nesses não-terminais, temos na definição de variáveis declarações como:

Programa 5.21

```

1 RecoverySet f1 = new RecoverySet(RPAREN).union(g),
2         f2 = new RecoverySet(RBRACKET).union(g);

```

Nessa declaração, os valores das variáveis *f1* e *f2* são também *null*, pois elas serão utilizadas como argumentos para outras chamadas. Por isso, se o leitor olhar a classe *RecoverySet*, verá que o método *union* retorna *null* se o argumento de entrada também contiver esse valor.

E, por último, um problema grave, que surgiu com a recuperação de erros, ocorre quando um não-terminal é chamado repetidamente e, ao tratar de um erro sintático, não consome nenhum token. Com isso, a chamada continua a ser feita indefinidamente. Na nossa gramática, isso acontece no não-terminal *statlist* que chama *statement*. Este, por sua vez, pode provocar um erro sintético que será tratado sem nenhum token ser consumido. No retorno, *statlist* chama-o outra vez, e o processo se repete. Para solucionar esse problema, temos diversas opções. Vamos adotar a solução de fazer com que se pelo menos um token seja consumido na chamada de *statement*, então ele é consumido, o que não ocorre da forma com que ele foi definido por causa do *LOOKAHEAD*. Assim, modificamos mais uma vez o não-terminal *statement* e obtemos o programa 5.22.

Programa 5.22

```

1 void statement(RecoverySet g) throws ParseE0FException {
2 {
3 RecoverySet f1 = new RecoverySet(SEMICOLON).union(g).remove(IDENT);
4 RecoverySet f2 = new RecoverySet(RBRACE).union(g).remove(IDENT);
5 }
6 {
7 try {
8     LOOKAHEAD(<IDENT> <IDENT>
9     vardecl(f1) <SEMICOLON>
10    |
11    LOOKAHEAD(1)
12    atribstat(f1) <SEMICOLON>
13    |
14    vardecl(f1) <SEMICOLON>
15    |
16    printstat(f1) <SEMICOLON>
17    |
18    readstat(f1) <SEMICOLON>
19    |
20    returnstat(f1) <SEMICOLON>
21    |
22    superstat(f1) <SEMICOLON>
23    |
24    ifstat(g)
25    |
26    forstat(g)
27    |
28    <LBRACE> statlist(f2) <RBRACE>
29    |
30    <BREAK> <SEMICOLON>
31    |
32    <SEMICOLON>
33 }
34 catch (ParseException e)
```

```

35  {
36      consumeUntil(g, e, "statement");
37  }
38 }
```

*

5.4 Arquivos-fonte do compilador

Veremos, a seguir, alguns exemplos da recuperação de erros realizada pelo nosso AS. No diretório *ssamples*, o leitor pode encontrar os arquivos *program.x*, *classdecl.x*, *classbody.x*, *methoddecl.x*, *methodbody.x*, *vardecl.x*, *paramlist.x*, *statement.x*, *statement2.x*, *atribstat.x* e *expression.x*, cujos nomes indicam quais os métodos que fazem a recuperação dos erros sintáticos neles contidos. Destes, iremos analisar apenas três. O primeiro é a recuperação feita pelo método *program*. O programa a ser utilizado, já comentado na seção 5.3, é o seguinte(arquivo *program.x*):

```

class test {

}

text that is not a "class"

class test2 {
```

Executando o compilador *X⁺⁺* sobre esse programa, obtemos o seguinte resultado:

```
cap05$ java parser.langX -debug_recovery ../ssamples/program.x
X++ Compiler - Version 1.0 - 2004
Reading from file ../ssamples/program.x . . .
```

```

*** program ***
Synchronizing Set: class
Ignoring token: <IDENT>
Ignoring token: <string_constant>
Found synchronizing token: class
Encountered "text" at line 5, column 1.
Was expecting one of:
<EOF>
"class" ...
```

```
0 Lexical Errors found
1 Syntactic Errors found
```

Note que agora a recuperação é feita de maneira adequada, e todos aqueles tokens entre as definições de classes foram desprezados e as duas classes foram analisadas da maneira esperada. O próximo é o programa *atribstat.x*, em que a recuperação é feita dentro do método *atribstat*.

```
class test {
int [] m(int a)
{
int b,c;
int d;

b == c + d;
d[9] = 0;
d[9] == c + d;
}

}

class test2 {
```

Para esse programa, o AS produz o seguinte resultado:

```
cap05$ java parser.langX -debug_recovery ..//ssamples/atribstat.x
X++ Compiler - Version 1.0 - 2004
Reading from file ..//ssamples/atribstat.x . . .

*** atribstat ***
Synchronizing Set: ; } { super string return read print
int if for class break
Ignoring token: ==
Ignoring token: <IDENT>
Ignoring token: +
Ignoring token: <IDENT>
Found syncronizing token: ;
Encountered "==" at line 7, column 6.
Was expecting one of:
  "[" ...
  "." ...
  "=" ...
<IDENT> ...

*** lvalue ***
Synchronizing Set: = ; } { super string return read print
int if for class break
Ignoring token: )
```

```
    Found synchronizing token: =
Encountered ")" at line 8, column 7.
Was expecting one of:
```

```
"]" ...
">" ...
"<" ...
"==" ...
"<=" ...
">=" ...
"!=" ...
"+" ...
"-"
"*"
"/"
 "%" ...
```

***** lvalue *****

```
Synchronizing Set: = ; } { super string return read print
int if for class break
Ignoring token: )
Ignoring token: ==
Ignoring token: <IDENT>
Ignoring token: +
Ignoring token: <IDENT>
Found synchronizing token: ;
Encountered ")" at line 9, column 7.
```

Was expecting one of:

```
"]" ...
">" ...
"<" ...
"==" ...
"<=" ...
">=" ...
"!=" ...
"+" ...
"-"
"*"
"/"
 "%" ...
```

***** atribstat *****

```
Synchronizing Set: ; } { super string return read print
int if for class break
Found synchronizing token: ;
0 Lexical Errors found
3 Syntactic Errors found
```

Nesse caso, o primeiro erro, que é a substituição do sinal de atribuição = pelo de igualdade relacional ==, foi recuperado pelo *atribstat*, ignorando os tokens até o ponto-e-vírgula que delimita o comando. Note que se a recuperação tivesse utilizado um *IDENT* como sincronizador, pararia no identificador *c* e outro erro seria apontado. O segundo erro é encontrado dentro de *lvalue* e a recuperação é feita por esse mesmo não-terminal, consumindo os tokens até =. E o terceiro erro é encontrado dentro de *lvalue*, mas a recuperação tem que ser feita dentro de *lvalue* e depois em *atribstat*. Apesar disso, uma única mensagem é emitida para esse erro, graças à alteração feita no método *consumeUntil*.

E para o programa *expression.x*, que é mostrado a seguir,

```
class test {
int [] m(int a)
{
int b,c;
int d;

if ( a + b <> c)
;
c = a.g(10, a -> b, c -> d);
for ( ; a === 0; a = a + 0.)
;
}
}

class test2 {
```

temos a seguinte execução:

```
cap05$ java parser.langX -debug_recovery ..ssamples/expression.x
X++ Compiler - Version 1.0 - 2004
Reading from file ..ssamples/expression.x . .

*** expression ***
Synchronizing Set: ; } { ) super string return read print
int if for class break
Ignoring token: >
Ignoring token: <IDENT>
Found synchronizing token: )
Encountered ">" at line 7, column 16.
Was expecting one of:
<int_constant> ...
<string_constant> ...
"null" ...
```

```

<IDENT> ...
"(" ...
"+" ...
"-> ...

*** expression ***
Synchronizing Set: null

*** lvalue ***
Synchronizing Set: null

*** expression ***
Synchronizing Set: ; } { super string return read print
int if for class break
Ignoring token: >
Ignoring token: <IDENT>
Ignoring token: ,
Ignoring token: <IDENT>
Ignoring token: -
Ignoring token: >
Ignoring token: <IDENT>
Ignoring token: )
Found syncronizing token: ;
Encountered ">" at line 9, column 19.
Was expecting one of:
<int_constant> ...
<string_constant> ...
"null" ...
<IDENT> ...
"(" ...
"+" ...
"-> ...

*** expression ***
Synchronizing Set: ; } { super string return read print
int if for class break
Ignoring token: =
Ignoring token: <int_constant>
Found syncronizing token: ;
Encountered "=" at line 10, column 15.
Was expecting one of:
<int_constant> ...
<string_constant> ...
"null" ...
<IDENT> ...
"(" ...

```

```
"+" ...
"-+" ...
```

```
0 Lexical Errors found
3 Syntactic Errors found
```

No primeiro erro, a recuperação é feita normalmente por *expression*, ao encontrar um sincronizador, que é o fecha parênteses. Já o segundo erro é encontrado no método *expression*, chamado de dentro do *arglist*, que, por sua vez, está dentro de uma chamada de *lvalue* e outra de *expression*. Note que a chamada de *lvalue*, *arglist* e a chamada mais interna de *expression* são feitas com o conjunto de sincronização igual a *null*, o que indica que esses métodos não devem fazer a recuperação de erro, mas simplesmente propagar a exceção. É isso que mostra a saída do AS. Tanto *expression* quanto *lvalue* recebem como parâmetro o valor *null*, deixando para a chamada mais externa de *expression* a recuperação, que é feita sincronizando a entrada com o ponto-e-vírgula. Qualquer erro sintático dentro da lista de argumentos, como neste exemplo, somente seria tratado por essa chamada mais externa. No terceiro erro, a recuperação é feita também por *expression*, consumindo os tokens até achar o ponto-e-vírgula dentro do comando *for*.

E, por último, fica a sugestão ao leitor de aplicar o nosso AS com recuperação de erros ao programa *bintree-erro-sintático.x*, que também se encontra no diretório *ssamples* e que foi utilizado no capítulo 4 para demonstrar a utilização do AS.

Capítulo 6

Geração da Árvore Sintática

O AS de um compilador deve, além de identificar se a entrada fornecida pertence ou não à linguagem desejada e reportar os possíveis erros existentes, produzir como saída uma representação do programa que irá servir para que as fases sucessivas do compilador possam ser executadas. Essa representação é a árvore sintática ou árvore de derivação.

Neste capítulo iremos estudar essa forma de representação e como implementá-la. Inicialmente mostraremos o que é uma árvore sintática e como pode ser usada para representar a derivação de uma cadeia da linguagem. Depois veremos como a definição do AS da linguagem X^{++} deve ser alterada para que, ao analisar o programa de entrada, construa a árvore sintática correspondente.

6.1 O que é a árvore sintática

Dadas uma cadeia x que pertence à linguagem L e uma GLC G que define essa linguagem, para indicar quais são as produções que devem ser aplicadas para gerar essa cadeia, podemos mostrar a seqüência de derivações diretas e as formas sentenciais intermediárias usadas até alcançarmos a cadeia em questão. Por exemplo, utilizando a gramática da linguagem X^{++} e a cadeia

```
class test {  
}
```

```
class test2 {  
}
```

teríamos a seguinte seqüência de derivações diretas:

```

⟨program⟩ ⇒
⟨classlist⟩ EOF ⇒
⟨classdecl⟩ ⟨classlist⟩ EOF ⇒
class test ⟨classbody⟩ ⟨classlist⟩ EOF ⇒
class test { } ⟨classlist⟩ EOF ⇒
class test { } ⟨classdecl⟩ EOF ⇒
class test { } class test2 ⟨classbody⟩ EOF ⇒
class test { } class test2 { } EOF

```

Com isso, sabemos exatamente quais foram as produções aplicadas na formação da cadeia. Uma forma alternativa de realizar essa operação é por meio da árvore sintática. Dada a GLC, uma árvore sintática é uma árvore rotulada cuja raiz tem sempre o não-terminal inicial como rótulo. Numa derivação

$$S \Rightarrow \alpha_1 \alpha_2 \dots \alpha_n$$

o nó S tem como filhos n nós rotulados α_1 até α_n . Cada um desses nós que é não-terminal deverá eventualmente ser substituído e será raiz de uma subárvore que representa outra derivação, até que todos os nós folhas sejam compostos apenas de símbolos terminais. No caso da aplicação de uma produção vazia, do tipo $A \rightarrow \lambda$, a derivação é representada com A na raiz e o símbolo λ como rótulo do único nó filho. Para o exemplo, teríamos a árvore de derivação da figura 6.1.

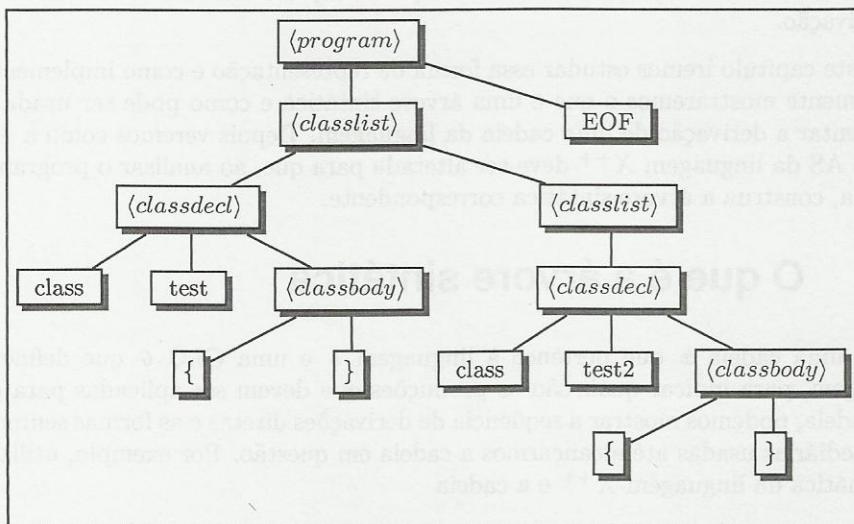


Figura 6.1 – Exemplo de árvore sintática.

Essa estrutura de árvore permite representar cada elemento do programa. Sua utilização pelo compilador permite que as demais tarefas a serem realizadas, como análise semântica e geração de código, sejam feitas de forma eficiente. Uma vez construída a árvore sintática, os demais passos do compilador consistem em visitar essa árvore numa determinada ordem. Essa tarefa pode ser feita de maneira muito

mais eficiente do que mediante a leitura do programa-fonte. Na seção 6.2 veremos como implementar a construção da árvore sintática no nosso compilador *X⁺⁺*.

A nossa utilização da árvore sintática não requer que todos os elementos de uma produção sejam colocados na árvore. Isso significa que muitos dos nós (em geral, nós folhas) podem ser desconsiderados no momento de construir a árvore. Por exemplo, quando tivermos um nó que representa o não-terminal *classdecl*, podemos eliminar os nós filhos correspondentes aos não-terminais *class* e *extends*, pois esses nós não acrescentam informação relevante para as demais fases do compilador. Esse tipo de árvore é chamada de árvore sintática abstrata. Para o exemplo anterior, teríamos, então, a árvore abstrata da figura 6.2

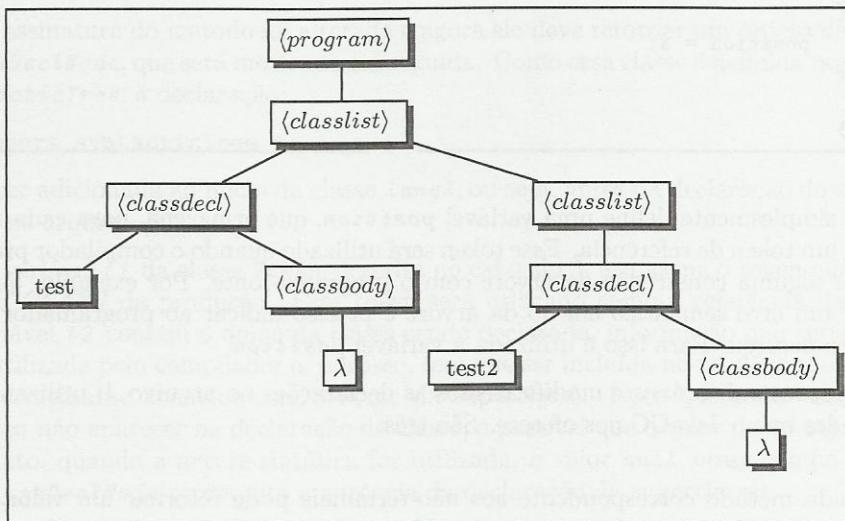


Figura 6.2 – Exemplo de árvore sintática abstrata.

6.2 Implementação da árvore sintática

A construção da árvore sintática no nosso compilador ocorre à medida que a análise sintática se processa. Cada execução de um dos métodos associados aos não-terminais cria um nó da árvore e “pendura” nele os nós filhos. Esses nós filhos são formados pelos tokens reconhecidos naquele método e pelos outros nós criados pelas chamadas a outros métodos de não-terminais. Por exemplo, uma chamada ao não-terminal *classdecl* irá criar um nó que tem como filhos cada um dos tokens que ele consumiu (como *IDENT*) e um outro nó criado pela chamada ao não-terminal *classbody*. Assim, embora a análise sintática seja descendente, a construção da árvore sintática é feita de baixo para cima, ou seja, dos nós mais profundos, em direção à raiz.

Cada nó da árvore será representado por um objeto Java que possui referências a outros objetos, que são seus filhos. Nós diferentes são representados por objetos de diferentes classes. Assim, teremos que definir uma classe para representar um nó do

tipo *program*, outra para um nó do tipo *classlist*, outra para *classdecl*, e assim por diante. Essas classes são declaradas dentro do pacote *syntacticTree*. Todas elas são subclasses de *GeneralNode*, mostrada no programa 6.1.

Programa 6.1

```

1 package syntacticTree;
2
3 import Token;
4
5 abstract public class GeneralNode {
6     public Token position;
7
8     public GeneralNode(Token x)
9     {
10         position = x;
11     }
12 }
13 }
```

Elá simplesmente define uma variável *position*, que armazena, para cada nó da árvore, um token de referência. Esse token será utilizado quando o compilador precisar associar alguma construção da árvore com o programa-fonte. Por exemplo, quando ocorrer um erro semântico em nó da árvore é preciso indicar ao programador onde esse erro ocorreu. Para isso é utilizada a variável *position*.

Para construir a árvore modificaremos as declarações no arquivo .jj utilizando as facilidades que o JavaCC nos oferece. São três:

- cada método correspondente aos não-terminais pode retornar um valor. Até agora, definimos todos como *void*. Vamos fazer com que cada um desses métodos retorne o nó que construiu, permitindo que o método que fez a chamada possa utilizar esse nó;
- cada token consumido pelo AS pode ser utilizado pelos métodos dos não-terminais para construir o nó da árvore. Para isso, basta “atribuir” o token que aparece na definição BNF para uma variável e, depois, utilizá-la na construção do nó;
- a cada token e a cada chamada de um método de um não-terminal que aparecem nas definições BNF é possível associar código Java que é executado, respectivamente, quando o token é consumido e quando o retorno de uma chamada ao método ocorre normalmente (sem que uma exceção seja lançada).

Por exemplo, para o não-terminal *classdecl*, podemos ter a definição mostrada no programa 6.2.

Programa 6.2

```

1 ClassDeclNode classdecl(RecoverySet g) throws ParseE0FException :
2 {
3     Token t1 = null, t2 = null, t3 = null;
4     ClassBodyNode c1 = null;
```

```

5  }
6  {
7  try {
8      t1 = <CLASS>  t2 = <IDENT> [ <EXTENDS> t3 = <IDENT> ] c1 = classbody(g)
9      { return new ClassDeclNode(t1, t2, t3, c1); }
10 }
11 catch (ParseException e)
12 {
13     consumeUntil(g, e, "classdecl");
14     return new ClassDeclNode(t1, t2, t3, c1);
15 }
16 }

```

A assinatura do método foi alterada e agora ele deve retornar um objeto da classe *ClassDeclNode*, que será mostrada em seguida. Como essa classe é definida no pacote *syntacticTree*, a declaração

```
import syntacticTree.*;
```

deve ser adicionada ao início da classe *langX*, ou seja, antes da declaração do método *main* no arquivo *.jj*.

A variável *t1* da classe *Token*, descrita no capítulo 3, armazena o token que casou com o *CLASS* da produção. Esse token será utilizado como a referência desse nó. A variável *t2* contém o nome da classe sendo declarada, informação que certamente será utilizada pelo compilador e, por isso, tem que ser incluída no nó *ClassDeclNode*. Algo semelhante acontece com o nome da superclasse. Note que como esse nome pode ou não aparecer na declaração da classe, é possível que o valor de *t3* seja *null*. Portanto, quando a árvore sintática for utilizada, o valor *null* nesse campo de um nó *ClassDeclNode* representa a ausência da declaração da superclasse.

E a variável *c1* corresponde a um *ClassBodyNode*, que é retornado pelo método *classbody*. Com esses elementos, constrói-se um nó *ClassDeclNode* no código Java que foi associado com a chamada de *classbody*. Como dissemos anteriormente, qualquer chamada a outro não-terminal ou qualquer token pode ter um trecho de código associado a si. Basta colocar o código entre chaves logo após qualquer um desses elementos.

No caso de erro sintático, além de chamar a rotina de recuperação, o método cria um *ClassDeclNode* “falso”, pois o método que chamou *classdecl* está esperando que um objeto desse tipo seja retornado. Nesse caso, a informação contida no nó estará incorreta. Por exemplo, a variável *t2* ou a variável *c1* podem ter valor *null* se o erro ocorrer antes dessas variáveis receberem o valor correto, o que não faz sentido, pois esses dois campos são sempre necessários na construção de um *ClassDeclNode*. Isso, porém, não tem importância, pois uma vez que algum erro sintático tenha ocorrido, sabe-se que a entrada analisada não corresponde a um programa válido e a árvore não poderá ser utilizada nas fases seguintes do compilador. Somente um programa sintaticamente correto origina uma árvore sintática válida. Em alguns casos, em vez de criarmos um nó falso, podemos retornar simplesmente o valor *null* no caso de erro sintático. Isso ocorre quando o não-terminal β pode derivar a cadeia vazia, o que obriga os métodos que fazem uso dele a estarem preparados para receber *null*,

indicando que nada foi reconhecido dentro do método de β . É o caso do não-terminal *paramlist*, mostrado no programa 6.18.

A classe *ClassDeclNode* é bastante simples, como em geral são todas as classes do pacote *syntacticTree*. Vejamos o programa 6.3.

Programa 6.3

```

1 package syntacticTree;
2
3 import Token;
4
5 public class ClassDeclNode extends GeneralNode {
6     public Token name;
7     public Token supertype;
8     public ClassBodyNode body;
9
10    public ClassDeclNode(Token t1, Token t2, Token t3, ClassBodyNode c)
11    {
12        super(t1); //passa token de referência para construtor da superclasse
13        name = t2;
14        supertype = t3;
15        body = c;
16    }
17
18 }
```

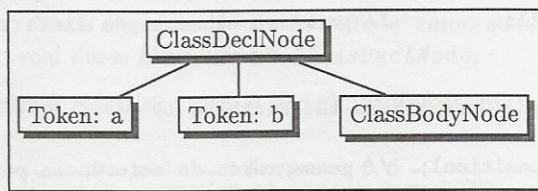
Essa classe possui apenas um construtor que recebe como parâmetros os dados passados pelo método *classdecl* e monta o nó da árvore. Um trecho de programa como

```

class a extends b {
    ...
}
```

fica, então, representado na árvore sintática como mostra a figura 6.3. Note que, além dos três filhos mostrados na figura, o nó *ClassDeclNode* possui também um token referente à palavra *class* lida na declaração da classe. Esse é o token de referência do nó e só serve para relacioná-lo ao programa-fonte. Por isso, não vamos representar esse token como um parte da árvore abstrata, a não ser para aqueles tipos de nós em que ele represente uma informação importante para a caracterização do programa analisado. Como no caso, por exemplo, do *AddNode* (Programa 6.47), que representa um expressão binária de soma ou subtração. Nesse caso, o token correspondente ao operador serve como referência, sendo também considerado um nó da árvore, pois é necessário para identificar qual é o operador que foi reconhecido na entrada.

Veremos então, como ficam os demais métodos dos não-terminais da gramática X^{++} com a inclusão do código para a construção da árvore sintática. Começaremos pelo símbolo inicial *program*. Para esse não-terminal, nada precisa ser feito, apenas retornar ao programa principal a lista de classes reconhecida pelo AS. Teremos, então, o programa 6.4.

Figura 6.3 – Exemplo de árvore sintática para *classdecl*.

Programa 6.4

```

1 ListNode program() throws ParseException :
2 {
3     RecoverySet g = First.program;
4
5     ListNode l = null, d = null;
6 }
7 {
8     <EOF>
9 }
10 (   l = classlist(g)
11     try {
12         <EOF> {return l;}
13     }
14     catch (ParseException e)
15     {
16         consumeUntil(g, e, "program");
17     }
18     [ d = program() ]
19 ) { return l;}
20 }
```

Esse não-terminal, então, retorna uma árvore sintática que foi montada dentro do método *classlist*. Tanto *program* quanto *classlist* retornam um nó do tipo *ListNode* que será utilizado sempre que precisarmos representar um grupo repetitivo de nós da árvore, como uma lista de classes ou uma lista de variáveis, ou uma lista de comandos etc. A definição dessa classe é dada no programa 6.5.

Programa 6.5

```

1 package syntacticTree;
2
3 public class ListNode extends GeneralNode {
4     public GeneralNode node;
5     public ListNode next;
6
7
8     public ListNode(GeneralNode t2)
9     {
10         super(t2.position); // passa token de referência para construtor da
11                         // superclasse. É o mesmo que o do seu filho
12     }
13 }
```

```

12     node = t2;
13     next = null;           // primeiro elemento da lista
14 }
15
16 public ListNode(GeneralNode t2, ListNode l)
17 {
18     super(t2.position);   // passa token de referência para construtor da
19                     // superclasse. É o mesmo que o do seu filho
20     node = t2;
21     next = l;             // primeiro elemento da lista
22 }
23
24 public void add(GeneralNode t2)
25 {
26     if (next == null) // verifica se é último da lista
27         next = new ListNode(t2); // insere no final
28     else
29         next.add(t2); // insere após o próximo
30 }
31
32 }

```

Além do construtor que monta um nó da árvore sintática colocando como filho um outro nó passado como parâmetro, essa classe possui o método *add*, que inclui um novo nó no final da lista. Por exemplo, se tivermos que construir uma lista com três classes, correspondente ao programa

```

class a extends b { ... }
class b { ... }
class c { ... }

```

teríamos como resultado a árvore mostrada na figura 6.4

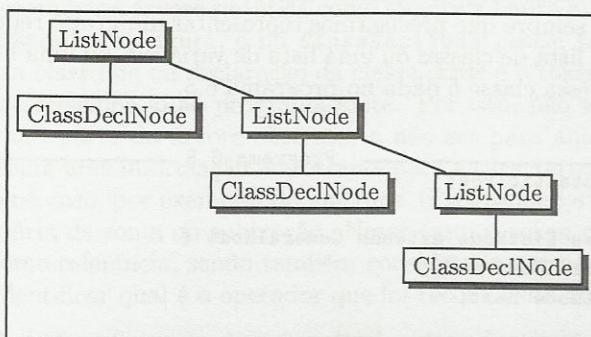


Figura 6.4 – Exemplo de árvore sintática para *classlist*.

Note que, sempre que estivermos lidando com uma lista de classes, embora tenhamos definido o filho “mais à esquerda” de *ListNode* como sendo um *GeneralNode*, teremos, como tipo real desse filho, um nó *ClassDeclNode*.

Para o não-terminal *classlist*, teremos a definição mostrada no programa 6.6

Programa 6.6

```

1  ListNode classlist(RecoverySet g) throws ParseEOFException :
2  {
3      ClassDeclNode c = null;
4      ListNode l = null;
5
6      RecoverySet f = First.classlist.union(g);
7  }
8  {
9      (
10         c = classdecl(f) [ l = classlist(g) ]
11     ) { return new ListNode(c, l);}
12
13 }
```

Aqui, a construção de uma lista de classes é feita de maneira bastante simples. O não-terminal *classdecl* retorna um nó do tipo *ClassDeclNode* e uma chamada recursiva a *classlist* retorna uma lista de classes. Basta, então, criar um novo nó do tipo *ListNode* para colocar o nó *ClassDeclNode* no início da lista. Note que a construção da lista é feita do final para o início. O primeiro nó *ListNode* é criado quando todas as classes já foram analisadas e a chamada ao método *classlist* não ocorre pois não existem mais classes a serem analisadas. Nesse caso, a variável *c* contém um nó que corresponde à definição da última classe analisada. O valor da variável *l* é *null*, pois como a chamada recursiva a *classlist* não foi executada, *l* continua com o valor com o qual foi inicializada no início do método. Então, ao criar o nó do tipo *ListNode*, temos como um filho o nó *ClassDeclNode*, correspondente à última classe analisada, e como outro filho, que seria o restante da lista, o valor *null*.

A figura 6.5 apresenta as etapas para construção daquele trecho de árvore mostrado anteriormente, para o programa

```

class a extends b { ... }
class b { ... }
class c { ... }
```

Inicialmente, o nó correspondente à classe *c* é incluído na lista, depois o nó correspondente à *b* e, finalmente, o nó da classe *a*.

Quando apresentamos a gramática de X^{++} , comentamos que existem diversas maneiras possíveis para definir as estruturas da linguagem. Em particular, comentamos que o não-terminal *classlist* poderia ser definido de duas maneiras e que a escolha de qual delas utilizar poderia ter consequências na implementação do AS. Naquele ponto, apresentamos as seguintes definições:

$$\langle \text{classlist} \rangle \rightarrow (\langle \text{classdecl} \rangle)^+$$

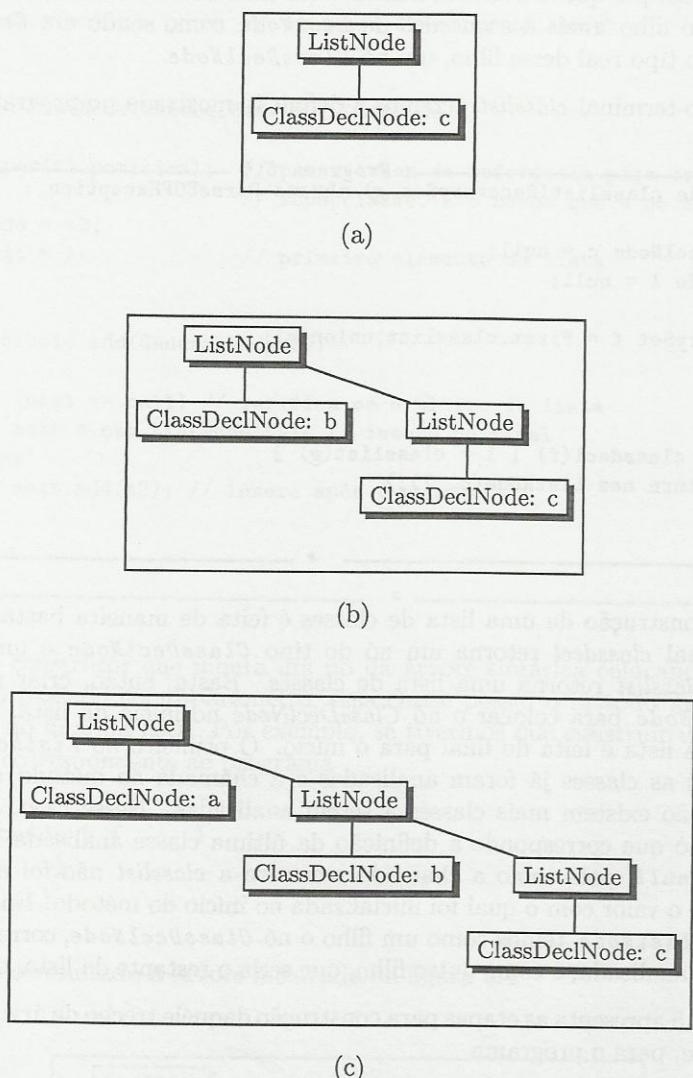


Figura 6.5 – Etapas da construção da árvore sintática.

ou

$$\langle \text{classlist} \rangle \rightarrow \langle \text{classdecl} \rangle [\langle \text{classlist} \rangle]$$

A diferença na implementação aparece justamente na maneira de construir a árvore sintática para esse não-terminal. Se tivéssemos escolhido a primeira forma de definir *classlist* teríamos que criar a lista do início para o fim. Seria algo como o código do programa 6.7.

Programa 6.7

```

1  ListNode classlist(RecoverySet g) throws ParseEOFException :
2  {
3      ClassDeclNode c = null;
4      ListNode l = null;
5
6      RecoverySet f = First.classlist.union(g);
7  }
8  {
9      (
10         c = classdecl(f)
11         { if (l == null)
12             l = new ListNode(c);
13         else
14             l.add(c);
15     }
16     )+ { return l; }
17
18 }
```

*

Nesse caso, cada classe reconhecida em *classdecl* tem que ser inserida na lista *l* por meio da chamada ao método *add* da classe *ListNode*. Essa inserção é feita na ordem em que as classes aparecem na entrada. Não pode ser diferente, pois a cada casamento feito dentro da construção *()+*, um novo valor é atribuído a *c* e o valor anterior é perdido. Em outros pontos da nossa gramática, como no não-terminal *classbody*, optou-se pela outra abordagem, ou seja, pela utilização dos operadores de repetição da BNF, para que o leitor possa comparar as duas formas de implementação. Em geral, a utilização de chamadas recursivas como em *classlist* torna mais fácil a construção da árvore sintática, mas, por outro lado, torna a gramática menos legível.

A definição do não-terminal *classdecl* já foi mostrada. Continuamos, então, com o próximo não-terminal que é *classbody*, no programa 6.8.

Programa 6.8

```

1  ClassBodyNode classbody(RecoverySet g) throws ParseEOFException :
2  {
3      ListNode c = null,
4          v = null,
5          ct = null,
6          m = null;
7      VarDeclNode vd;
8      ConstructDeclNode cd;
9      MethodDeclNode md;
10     Token t = null;
11
12     RecoverySet f2 = new RecoverySet(SEMICOLON).union(g).remove(IDENT),
13         f3 = First.methoddecl.union(g).remove(IDENT),
14         f4 = First.constructdecl.union(f3).remove(IDENT),
15         f5 = First.vardecl.union(f4).remove(IDENT);
16     }
17     {
18         try {
```

```

19     t = <LBRACE>
20     [c = classlist(f5)]
21     (LOOKAHEAD(3) vd = vardecl(f2) <SEMICOLON>
22     { if ( v == null)
23         v = new ListNode(vd);
24     else
25         v.add(vd);
26     }
27   )*
28   (cd = constructdecl(f4)
29   { if ( ct == null)
30       ct = new ListNode(cd);
31   else
32       ct.add(cd);
33   }
34   )*
35   (md = methoddecl(f3)
36   { if ( m == null)
37       m = new ListNode(md);
38   else
39       m.add(md);
40   }
41   )*
42   <RBRACE>
43   { return new ClassBodyNode(t, c, v, ct, m); }
44 }
45 catch (ParseException e)
46 {
47     consumeUntil(g, e, "classbody");
48     return new ClassBodyNode(t, c, v, ct, m);
49 }
50 }

```

O nó *ClassBodyNode* retornado por *classbody* possui quatro filhos, todos do tipo *ListNode*. O primeiro armazena um nó que corresponde à lista de classes aninhadas; o segundo, à lista de variáveis; o terceiro, à lista de construtores; e o quarto à lista de métodos da classe. Para a lista de classes, utilizamos a chamada recursiva ao método *classlist* descrito anteriormente, que já retorna a lista de *ClassDeclNodes* montada. Para as demais listas, utilizamos o operador de repetição da BNF e, por isso, precisamos montar a lista à medida que as variáveis, os construtores ou os métodos aparecem na entrada. A classe correspondente a esse nó e um exemplo de como fica a árvore montada por esse método são mostrados no programa 6.9 e na figura 6.6 respectivamente.

Programa 6.9

```

1 package syntacticTree;
2
3 import Token;
4
5 public class ClassBodyNode extends GeneralNode {
6     public ListNode clist;           // lista de classes aninhadas

```

```

7  public ListNode vlist;      // lista de variáveis da classe
8  public ListNode ctlist;     // lista de construtores
9  public ListNode mlist;      // lista de métodos
10
11 public ClassBodyNode(Token t1, ListNode c, ListNode v, ListNode ct,
12                      ListNode m)
13 {
14     super(t1); //passa token de referência para construtor da superclasse
15     ctlist = c;
16     vlist = v;
17     ctlist = ct;
18     mlist = m;
19 }
20
21 }

```

Para o seguinte trecho de programa, teríamos a árvore sintática da figura 6.6.

```

class teste {
class a { ... }
class b { ... }

int c, d;
string e, f;

constructor (int g)
{ ... }

int h()
{ ... }

}

```

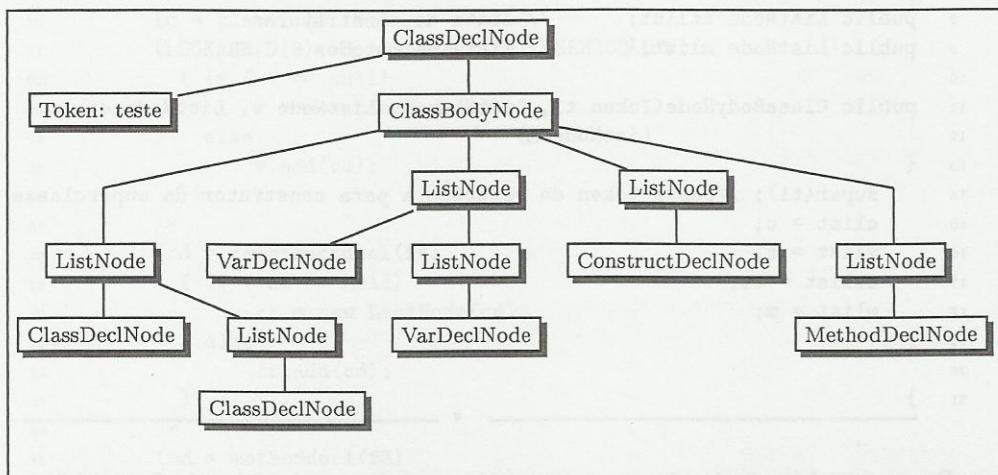
No não-terminal *vardecl* (Programa 6.10) há alguns comentários a serem feitos sobre a construção da árvore sintática.

Programa 6.10

```

1 VarDeclNode vardecl(RecoverySet g) throws ParseEOFException :
2 {
3     Token t1 = null, t2;
4     int k = 0;
5     ListNode l = null;
6 }
7 {
8     try {
9         ( t1 = <INT> | t1 = <STRING> | t1 = <IDENT> )
10        t2 = <IDENT> ( <LBRACKET> <RBRACKET> { k++; } )*
11        { l = new ListNode(new VarNode(t2, k)); }
12        (<COMMA> { k = 0; } t2 = <IDENT> ( <LBRACKET> <RBRACKET> { k++; } )*

```

Figura 6.6 – Exemplo de árvore sintática para *classbody*.

```

13
14     { l.add(new VarNode(t2, k)); }
15     */
16     { return new VarDeclNode(t1, l); }
17 }
18 catch (ParseException e)
19 {
20     consumeUntil(g, e, "vardecl");
21     return new VarDeclNode(t1, l);
22 }
23
  
```

Um nó *VarDeclNode* que é retornado por esse método representa uma declaração de uma ou mais variáveis de um mesmo tipo como nos comandos

`int a, b, c;`

ou

`string d, e;`

Essa declaração é representada no nó *VarDeclNode* que contém um filho que é o tipo da variável (um nó do tipo *Token*) e uma lista de variáveis, como sempre representada por um *ListNode*. A classe para esse tipo de nó é definida conforme mostra o programa 6.11.

Programa 6.11

```

1 package syntacticTree;
2
3 import Token;
4
5 public class VarDeclNode extends StatementNode {
6     public ListNode vars;
7
8     public VarDeclNode(Token t, ListNode p)
9     {
10         super(t);
11         vars = p;
12     }
13 }
```

Note que o nó do tipo *Token* não aparece na declaração da classe. Isso porque podemos utilizar a própria variável *position* que foi declarada na superclasse *GeneralNode*, ou seja, nesse caso, o token de referência faz parte da árvore sintática. Cada elemento da lista que aparece como filha de *VarDeclNode* é do tipo *VarNode*, que tem como filhos um *Token* e um número inteiro que indicam o nome da variável e a sua dimensão, respectivamente (Programa 6.12).

Programa 6.12

```

1 package syntacticTree;
2
3 import Token;
4
5 public class VarNode extends ExprNode {
6     public int dim;
7
8     public VarNode(Token t)
9     {
10         super(t);
11         dim = 0;
12     }
13
14     public VarNode(Token t, int k)
15     {
16         super(t);
17         dim = k;
18     }
19
20 }
```

Na construção da árvore sintática em *vardecl*, a dimensão da variável é registrada contando-se o número de vezes que aparece um fecha colchete na sua declaração (linhas 10 e 12 do Programa 6.10). A variável *k* é incrementada cada vez que ocorre um casamento com o fecha chaves, dando, assim, a dimensão da variável.

Na figura 6.7 vemos como seria a árvore sintática para as duas declarações de variáveis na classe *test*:

```
class test {
    int a, b[];
    string c[][][];
}
```

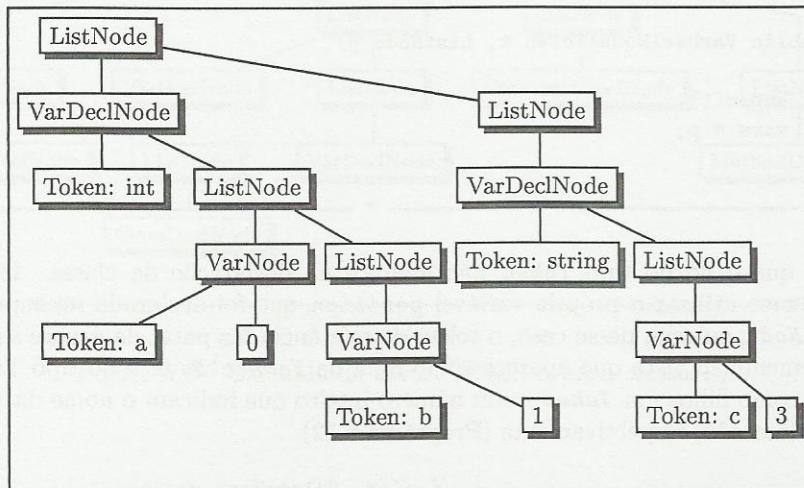


Figura 6.7 – Exemplo de árvore sintática para *vardecl*.

Os não-terminais *constructdecl* e *methoddecl* são mostrados em seguida. O primeiro cria e retorna um nó que possui apenas um filho, que é um *MethodBodyNode*. Já o segundo precisa acrescentar ao nó que retorna outras informações: o nome do método, o tipo e a dimensão do valor de retorno.

Programa 6.13

```

1 ConstructDeclNode constructdecl(RecoverySet g) throws ParseEOFException :
2 {
3     Token t = null;
4     MethodBodyNode m = null;
5 }
6 {
7     try {
8         t = <CONSTRUCTOR> m = methodbody(g)
9         { return new ConstructDeclNode(t, m);}
10 }
11 catch (ParseException e)
12 {
13     consumeUntil(g, e, "constructdecl");
14     return new ConstructDeclNode(t, m);
15 }
16 }
17
18 MethodDeclNode methoddecl(RecoverySet g) throws ParseEOFException :
19 {
```

```

20 Token t1 = null,
21     t2 = null;
22 int k = 0;
23 MethodBodyNode m = null;
24 }
25 {
26 try {
27     ( t1 = <INT> | t1 = <STRING> | t1 = <IDENT> )
28     (<LBRACKET> <RBRACKET> { k++; } )*
29     t2 = <IDENT> m = methodbody(g)
30     { return new MethodDeclNode(t1, k, t2, m); }
31 }
32 catch (ParseException e)
33 {
34     consumeUntil(g, e, "methoddecl");
35     return new MethodDeclNode(t1, k, t2, m);
36 }
37 }

```

Assim, as classes *constructDeclNode* e *MethodDeclNode* mostradas nos programas 6.14 e 6.15 refletem essas diferenças.

Programa 6.14

```

1 package syntacticTree;
2
3 import Token;
4
5 public class ConstructDeclNode extends GeneralNode {
6     public MethodBodyNode body;
7
8     public ConstructDeclNode(Token t, MethodBodyNode m)
9     {
10         super(t);
11         body = m;
12     }
13 }

```

Programa 6.15

```

1 package syntacticTree;
2
3 import Token;
4
5 public class MethodDeclNode extends GeneralNode {
6     public int dim;
7     public Token name;
8     public MethodBodyNode body;
9
10    public MethodDeclNode(Token t, int k, Token t2, MethodBodyNode b)
11    {
12        super(t);

```

```

13     dim = k;
14     name = t2;
15     body = b;
16 }
17 }

```

A figura 6.8 mostra um exemplo de utilização desses dois nós para o trecho de programa exemplificado a seguir. Note que as descrições dos parâmetros formais e do corpo do construtor ou método são feitas dentro do seu filho *MethodBodyNode*.

```

class teste {
constructor ()
{ ... }

int [] [] m()
{ ... }
}

```

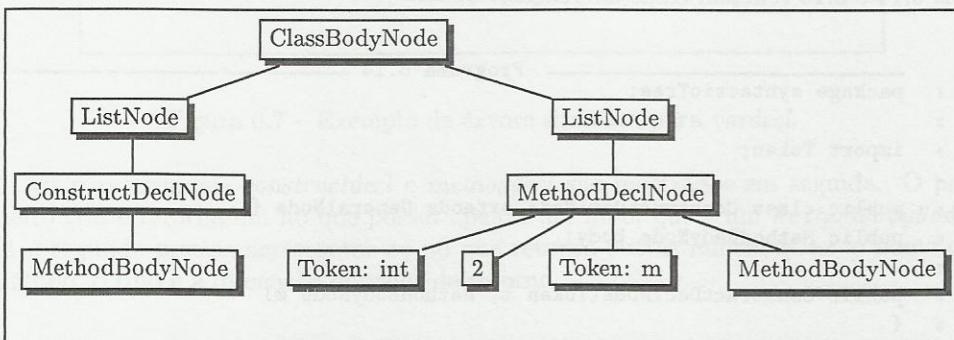


Figura 6.8 – Exemplo de árvore sintática para *constructdecl* e *methoddecl*.

Dentro do não-terminal *methodbody*, descrito no programa 6.16, é construído um nó do tipo *MethodBodyNode* que possui dois filhos, um do tipo *ListNode*, que descreve os parâmetros formais do método, e o segundo do tipo *statement*, que descreve o(s) comando(s) do método.

Programa 6.16

```

1 MethodBodyNode methodbody(RecoverySet g) throws ParseEOFException :
2 {
3     Token t1 = null;
4     ListNode l = null;
5     StatementNode s = null;
6
7     RecoverySet f = new RecoverySet(RPAREN).union(g);
8 }
9 {

```

```

10   try {
11     t1 = <LPAREN> l = paramlist(f) <RPAREN> s = statement(g)
12     { return new MethodBodyNode(t1, l, s); }
13   }
14   catch (ParseException e)
15   {
16     consumeUntil(g, e, "methodbody");
17     return new MethodBodyNode(t1, l, s);
18   }
19 }
```

A classe *MethodBodyNode* é implementada como mostra o programa 6.17.

Programa 6.17

```

1 package syntacticTree;
2
3 import Token;
4
5 public class MethodBodyNode extends GeneralNode {
6   public ListNode param;
7   public StatementNode stat;
8
9   public MethodBodyNode(Token t, ListNode l, StatementNode s)
10  {
11    super(t);
12    param = l;
13    stat = s;
14  }
15 }
```

A figura 6.9 mostra a árvore para o seguinte método:

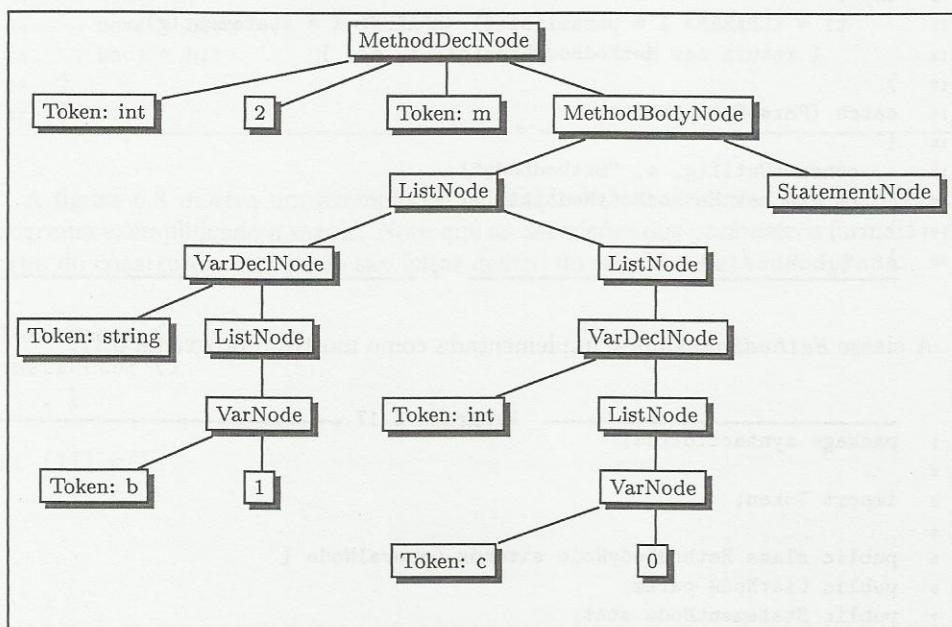
```
int[] [] m(string b[], int c)
;
```

Pela figura 6.9, podemos ver também que o método *paramlist* constrói um *ListNode* em que cada elemento é um nó do tipo *VarDeclNode*, que corresponde à declaração de um dos parâmetros formais do método. Sua definição é dada no programa 6.18. Note que para cada variável, o método cria um *VarNode* que é inserido num *ListNode* (linhas 13 e 19), que, por sua vez, é incluído como filho de um *VarDeclNode* (linhas 14 e 20). Esse último ainda faz parte do *ListNode* que é retornado pelo método (linha 23).

Programa 6.18

```

1 ListNode paramlist(RecoverySet g) throws ParseEOFException :
2 {
3   ListNode p = null, q = null;
4   int k = 0;
5   Token t1 = null, t2 = null;
```

Figura 6.9 – Exemplo de árvore sintática para *methodbody*.

```

6 }
7 {
8 try {
9 [
10   ( t1 = <INT> | t1 = <STRING> | t1 = <IDENT>) t2 = <IDENT>
11   (<LBRACKET> <RBRACKET> { k++; } )*
12   {
13     q = new ListNode(new VarNode(t2, k));
14     p = new ListNode(new VarDeclNode(t1, q));
15   }
16   ( <COMMA> {k = 0;} ( t1 = <INT> | t1 = <STRING> | t1 = <IDENT>)
17     t2= <IDENT> (<LBRACKET> <RBRACKET> {k ++;} )*
18   {
19     q = new ListNode(new VarNode(t2, k));
20     p.add(new VarDeclNode(t1, q));
21   }
22   )*
23 ] { return p;}
24 }
25 catch (ParseException e)
26 {
27   consumeUntil(g, e, "paramlist");
28   return null;
29 }
30 }
  
```

Esse é um método que, em caso de erro sintático, pode retornar simplesmente o valor `null`, pois esse não-terminal deriva a cadeia vazia e, portanto, o método que o chamou deve estar preparado para tratar um filho inexistente.

Na figura 6.9, vemos ainda que existe um único nó para representar o comando associado ao método. Porém é preciso que sejamos capazes de, ao analisarmos a árvore sintática, reconhecer que comando é esse. O nó deveria, de fato, representar ou uma atribuição, ou um `if` ou um `for`, ou um comando composto etc. Por isso, o nó do tipo `StatementNode` que foi declarado como filho do nó `MethodBodyNode` é, na verdade, um nó abstrato, ou melhor, definido por meio de uma classe abstrata. Para cada tipo de comando da linguagem X^{++} , teremos um tipo de nó distinto, subclasse de `StatementNode`. Se o leitor reparar na definição de `VarDeclNode` mostrada no programa 6.11, irá notar que aquela classe foi definida como subclasse de `StatementNode`. Isso porque uma declaração de variável é um dos possíveis comandos que podem aparecer no corpo de um método e utilizaremos esse mesmo tipo de nó `VarDeclNode` para representar tal comando.

O não-terminal `statement`, cuja implementação é mostrada no programa 6.19, trata de chamar os métodos correspondentes aos comandos da linguagem. Assim, na maioria das vezes, não precisa criar um nó que representa um comando, mas simplesmente retornar o nó criado dentro daqueles métodos. O tipo de nó que ele retorna tem, então, que ser `StatementNode` (Programa 6.20).

Programa 6.19

```

1 StatementNode statement(RecoverySet g) throws ParseEOFException :
2 {
3     StatementNode s = null;
4     ListNode l;
5     Token t1 = null;
6
7     RecoverySet f1 = new RecoverySet(SEMICOLON).union(g).remove(IDENT);
8     RecoverySet f2 = new RecoverySet(RBRACE).union(g).remove(IDENT);
9 }
10 {
11     try {
12
13         (
14             LOOKAHEAD(2)
15             s = vardecl(f1) <SEMICOLON>
16         |
17             s = atribstat(f1) <SEMICOLON>
18         |
19             s = printstat(f1) <SEMICOLON>
20         |
21             s = readstat(f1) <SEMICOLON>
22         |
23             s = returnstat(f1) <SEMICOLON>
24         |
25             s = superstat(f1) <SEMICOLON>
26         |
27             s = ifstat(g)
28     }

```

```

29     s = forstat(g)
30   |
31   t1 = <LBRAVE> l = statlist(f2) <RBRACE> { s = new BlockNode(t1, l); }
32   |
33   t1 = <BREAK> <SEMICOLON> { s = new BreakNode(t1);}
34   |
35   t1 = <SEMICOLON> { s = new NopNode(t1); }
36 } {return s;}
37 }
38 catch (ParseException e)
39 {
40   consumeUntil(g, e, "statement");
41   return new NopNode(t1);
42 }
43 }

```

Programa 6.20

```

1 package syntacticTree;
2
3 import Token;
4
5 abstract public class StatementNode extends GeneralNode {
6
7   public StatementNode(Token t)
8   {
9     super(t);
10 }
11 }

```

No caso de termos um comando composto, a chamada ao método *statlist* na linha 31 do método *statement* retorna uma lista de comandos, ou seja, um *ListNode* cujos componentes (filhos) são nós do tipo *StatementNode*. Nesse caso, dentro do método correspondente ao não-terminal *statement*, é criado um nó *BlockNode* cujo filho é o *ListNode* retornado por *statlist*.

O método *statlist* e a classe *BlockNode* são mostrados nos programas 6.21 e 6.22. A definição de *statlist* utiliza a abordagem de chamadas recursivas para montar a lista de comandos. Note que os nós que compõem a lista podem ser de qualquer subclasse de *StatementNode*, até mesmo outros *BlockNodes*.

Programa 6.21

```

1 ListNode statlist(RecoverySet g) throws ParseEOFException :
2 {
3   ListNode l = null;
4   StatementNode s = null;
5
6   RecoverySet f = First.statlist.remove(IDENT).union(g);
7 }
8 {
9   s = statement(f) [ l = statlist(g)]

```

```

10     { return new ListNode(s, 1); }
11 }
```

Programa 6.22

```

1 package syntacticTree;
2
3 import Token;
4
5 public class BlockNode extends StatementNode {
6     public ListNode stats;
7
8     public BlockNode(Token t, ListNode l)
9     {
10         super(t);
11         stats = l;
12     }
13 }
```

Também para o comando *break* e para o comando vazio, a construção do nó é feita dentro do método *statement*, nas linhas 33 e 35. Nesse caso, esses nós são sempre folhas e só possuem o token de referência. As classes *BreakNode* e *NopNode* são definidas de acordo com os programas 6.23 e 6.24.

Programa 6.23

```

1 package syntacticTree;
2
3 import Token;
4
5 public class BreakNode extends StatementNode {
6
7     public BreakNode(Token t)
8     {
9         super(t);
10    }
11 }
```

Programa 6.24

```

1 package syntacticTree;
2
3 import Token;
4
5 public class NopNode extends StatementNode {
6
7     public NopNode(Token t)
8     {
9         super(t);
10    }
11 }
```

Vamos tomar o exemplo visto anteriormente, ligeiramente modificado:

```
int[][] m(string b[], int c)
{
    ;
    c = 0;
}
```

Teríamos a árvore sintática da figura 6.10.

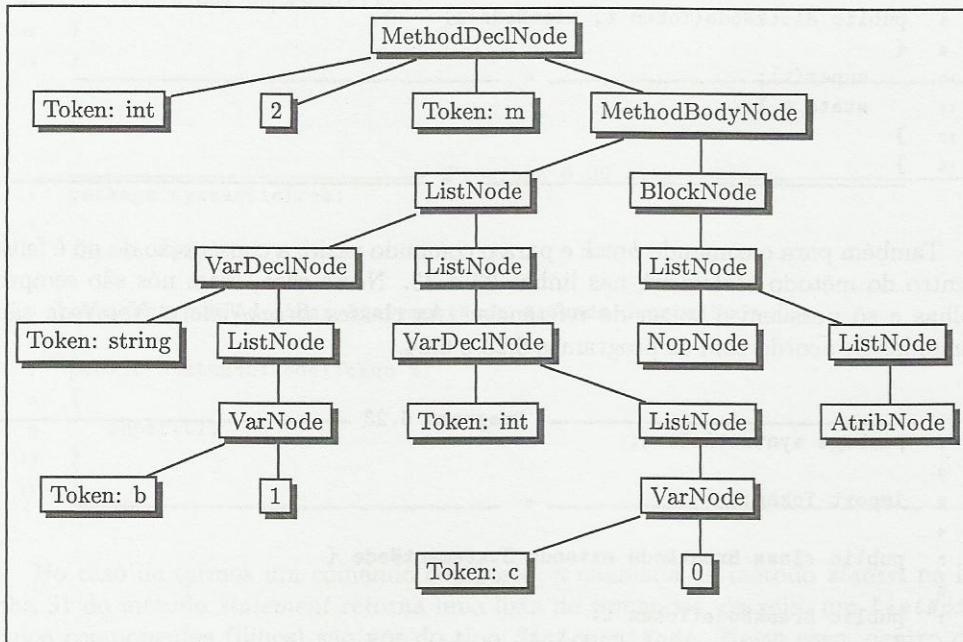


Figura 6.10 – Exemplo de árvore sintática para *statement* e *statlist*.

O método *atribstat* é o responsável pela criação de um nó do tipo *AtribNode*, cuja classe correspondente é mostrada no programa 6.25.

Programa 6.25

```
1 package syntacticTree;
2
3 import Token;
4
5 public class AtribNode extends StatementNode {
6     public ExpreNode expr1, expr2;
7
8     public AtribNode(Token t, ExpreNode e1, ExpreNode e2)
9     {
10         super(t);
```

```

11     expr1 = e1;
12     expr2 = e2;
13 }
14 }
```

O token de referência é o sinal de atribuição e as expressões *expr1* e *expr2* correspondem aos lados esquerdo e direito da atribuição, respectivamente. No método *atribstat* (Programa 6.26), o nó *AtribNode* é montado com a chamada a *lvalue* na linha 10, que devolve a primeira expressão, e de *expression* ou *aloceexpression*, que retorna a segunda. Note que, apesar de serem do mesmo tipo, a expressão retornada por *lvalue* possui algumas restrições sintáticas, já comentadas no capítulo 2.

Programa 6.26

```

1 AtribNode atribstat(RecoverySet g) throws ParseEOFException :
2 {
3     ExpreNode e1 = null, e2 = null;
4     Token t1 = null;
5
6     RecoverySet f1 = new RecoverySet(ASSIGN).union(g);
7 }
8 {
9     try {
10         e1 = lvalue(f1) t1 = <ASSIGN>
11         ( e2 = aloceexpression(g) | e2 = expression(g))
12         { return new AtribNode(t1, e1, e2); }
13     }
14     catch (ParseException e)
15     {
16         consumeUntil(g, e, "atribstat");
17         return new AtribNode(t1, e1, e2);
18     }
19 }
```

Note que os filhos do nó *AtribNode* podem ser diferentes tipos de expressão. Por exemplo, a segunda expressão pode ser uma soma binária ou uma multiplicação, ou simplesmente uma constante. Para cada um desses tipos de expressão, devemos ter um tipo diferenciado de nó para que possamos reconhecer como tratar aquela expressão. Assim como fizemos com *StatementNode*, definimos *ExpreNode* como uma classe abstrata da qual os demais nós que representam os tipos existentes de expressões são subclasses. O programa 6.27 apresenta a definição de *ExpreNode*.

Programa 6.27

```

1 package syntacticTree;
2
3 import Token;
4
5 abstract public class ExpreNode extends GeneralNode {
6
7     public ExpreNode(Token t)
8     {
```

```

9     super(t);
10    }
11 }

```

O não-terminal *lvalue* pode retornar quatro tipos distintos de nós, todos subclasses de *exprNode*. O primeiro é um *VarNode*, quando o não-terminal reconhece como expressão apenas a referência a uma variável como em *a = b - 10*. A construção desse nó é feita na linha 11 do programa 6.28. O nome da variável é utilizado para criar um *VarNode*, cuja classe foi mostrada no programa 6.12, e esse nó é retornado. Uma árvore sintática para a atribuição *a = b - 10* é mostrada na figura 6.11.

Programa 6.28

```

1 ExpreNode lvalue(RecoverySet g) throws ParseEOFException :
2 {
3     ExpreNode e1 = null,
4         e2 = null;
5     Token t1 = null,
6         t2 = null;
7     ListNode l = null;
8 }
9 {
10 try {
11     t1 = <IDENT> { e1 = new VarNode(t1); }
12     (
13         t1 = <LBRACKET> e2 = expression(null) <RBRACKET>
14         { e1 = new IndexNode(t1, e1, e2); }
15     |
16     LOOKAHEAD(3)
17     t1 = <DOT> t2 = <IDENT> <LPAREN> l = arglist(null) <RPAREN>
18     { e1 = new CallNode(t1, e1, t2, l); }
19     |
20     t1 = <DOT> t2 = <IDENT>
21     { e1 = new DotNode(t1, e1, t2); }
22     *)
23     { return e1; }
24 }
25 catch (ParseException e)
26 {
27     consumeUntil(g, e, "lvalue");
28     return new VarNode(t1);
29 }
30 }

```

Outro tipo de nó que pode ser retornado pelo não-terminal *lvalue* é um *IndexNode*, cuja classe é mostrada no programa 6.29, que representa a indexação de uma variável com dimensão maior que zero. Esse nó possui duas expressões como filhas.

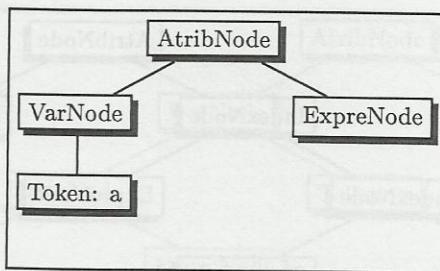


Figura 6.11 – Exemplo de árvore sintática para $a = b - 10$ (*lvalue*).

Programa 6.29

```

1 package syntacticTree;
2
3 import Token;
4
5 public class IndexNode extends ExpreNode {
6     public ExpreNode expr1, expr2;
7
8     public IndexNode(Token t, ExpreNode e1, ExpreNode e2)
9     {
10         super(t);
11         expr1 = e1;
12         expr2 = e2;
13     }
14 }
```

*

A primeira indica qual a variável que está sendo indexada e a segunda qual é o índice a ser utilizado. Note que a primeira expressão pode ser um nó do tipo *VarNode*, como no caso $a[i+1] = b - 10$, mas pode ser também de outros tipos, por exemplo, outro *IndexNode*, como em $a[j][i+1] = b - 10$. Esse tipo de nó é construído na linha 14 do programa 6.28. O nó é criado utilizando como filhos a variável *v1*, que contém o *VarNode* que foi criado na linha 11 representando a variável a ser indexada, e a expressão retornada na chamada a *expression*. O nó criado é atribuído à própria *v1*.

Se tivermos uma segunda indexação, então um novo *IndexNode* é criado tendo como filho *v1*, que é o *IndexNode* criado anteriormente, e a expressão retornada por *expression*. Essa seqüência pode ser repetida diversas vezes, uma para cada nível de indexação que for reconhecido no não-terminal *lvalue*. A figura 6.12 mostra a árvore sintática para a atribuição $a[j][i+1] = b - 10$.

No programa 6.30, mostramos a classe *DotNode* que também pode ser retornada por *lvalue* e representa o acesso a uma variável (campo) de um objeto como $a.x = b - 10$. Da mesma maneira que no caso da indexação, podemos ter múltiplos acessos a campos, como em $a.x.y = b - 10$. Podemos, ainda, combinar indexação e acesso a um campo, como em $a[i].x = b - 10$ ou $a.x[i] = b - 10$. O processo de construção do *DotNode* é similar. A cada casamento que ocorre na linha 21 de

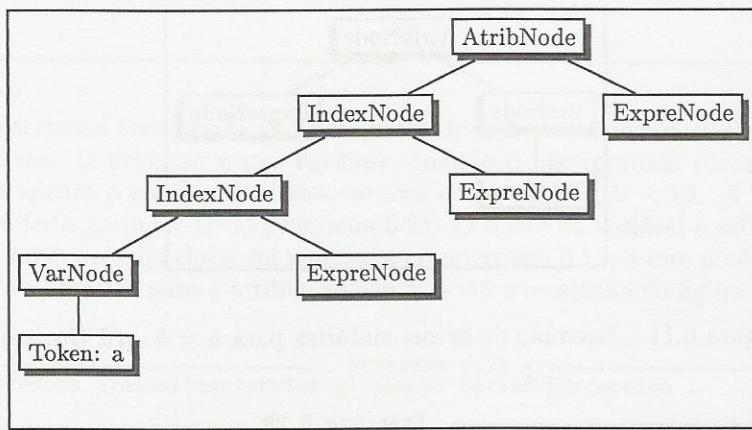


Figura 6.12 – Exemplo de árvore sintática para $a[j][i + 1] = b - 10$ (*lvalue*).

lvalue, um nó é montado tendo como primeiro filho o valor em *v1*, que, como já vimos, pode ser um *VarNode*, um *IndexNode* ou um outro *DotNode*, e como segundo filho um *Token* com o nome do campo a ser utilizado. Na figura 6.13, mostramos a árvore sintática para a atribuição $a[i].x = b - 10$.

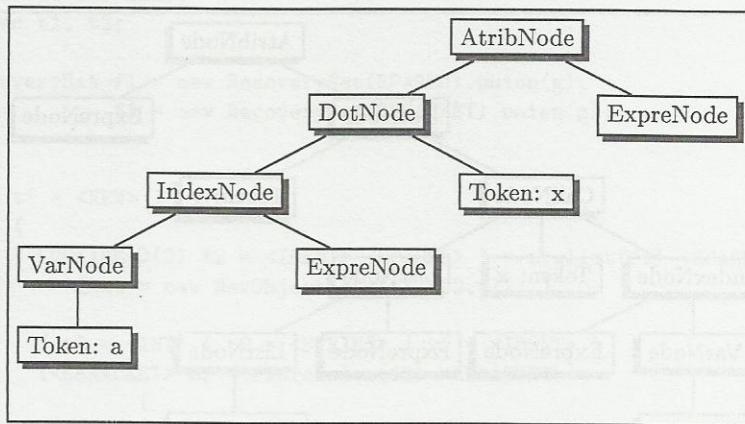
Programa 6.30

```

1 package syntacticTree;
2
3 import Token;
4
5 public class DotNode extends ExpreNode {
6     public ExpreNode expr;
7     public Token field;
8
9     public DotNode(Token t, ExpreNode e, Token t2)
10    {
11        super(t);
12        field = t2;
13        expr = e;
14    }
15}
  
```

E, por último, o nó *CallNode* mostrado no programa 6.31 também pode ser retornado por *lvalue*, representando uma chamada de método como em $a.x(i, j) = b - 10$.¹

¹Note que, semanticamente, esse comando não faz sentido, mas a nossa gramática permite que ele ocorra sem problemas. Esse erro deverá ser apontado pelo analisador semântico, que será tratado no capítulo 9.

Figura 6.13 – Exemplo de árvore sintática para $a[i].x = b - 10$ (*lvalue*).

Programa 6.31

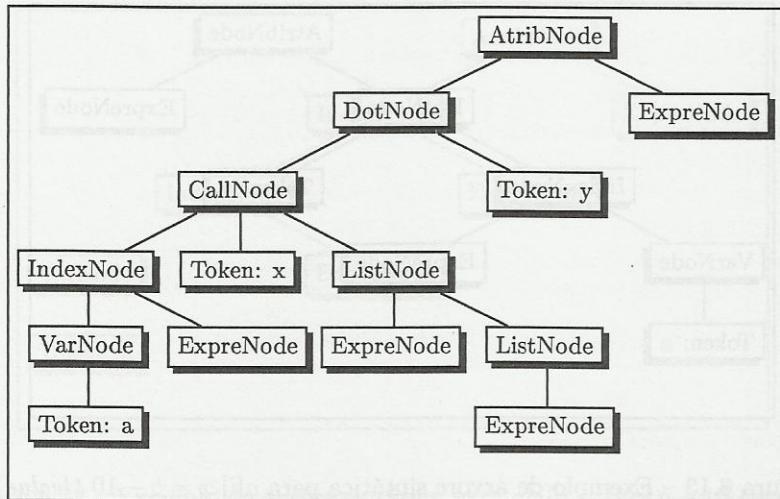
```

1 package syntacticTree;
2
3 import Token;
4
5 public class CallNode extends ExpreNode {
6     public ExpreNode expr;
7     public Token meth;
8     public ListNode args;
9
10    public CallNode(Token t, ExpreNode e, Token m, ListNode l)
11    {
12        super(t);
13        expr = e;
14        meth = m;
15        args = l;
16    }
17}
  
```

*

Esse nó é criado na linha 18 do não-terminal *lvalue* e possui como primeiro filho uma expressão que indica de qual objeto deseja-se utilizar um método, dado pela variável *v1*. O segundo filho é o nome do método, dado por um *Token*, e, por último, um nó *ListNode* que contém uma lista de expressões que são os argumentos da chamada. Esse tipo de nó pode ser utilizado com os demais vistos antes, dentro de *lvalue*. Por exemplo, podemos ter a atribuição $a[i].x(2, k).y = b - 10$, que possui um *VarNode*, um *IndexNode*, um *CallNode* e um *DotNode*, conforme mostrado na figura 6.14.

O não-terminal *arglist*, utilizado em *lvalue*, reconhece uma lista de expressões e constrói para elas um *ListNode*. Sua definição é dada no programa 6.32.

Figura 6.14 – Exemplo de árvore sintática para $a[i].x(2, k).y = b - 10$ (lvalue).

Programa 6.32

```

1  ListNode arglist(RecoverySet g) throws ParseEOFException :
2  {
3      ListNode l = null;
4      ExpreNode e = null;
5
6      RecoverySet f = new RecoverySet(COMMA).union(g);
7  }
8  {
9      [
10         e = expression(f)
11         { l = new ListNode(e); }
12         (<COMMA> e = expression(f)
13             { l.add(e); }
14             )*
15     ]
16     { return l; }
17 }
```

Do lado direito da atribuição, podemos ter uma expressão normal, representada pela chamada a *expression*, ou uma expressão de criação de um objeto ou de uma variável com dimensão. Esse tipo de expressão é reconhecida pelo não-terminal *allocexpression*, no programa 6.33.

Programa 6.33

```

1  ExpreNode allocexpression(RecoverySet g) throws ParseEOFException :
2  {
3      ExpreNode e1 = null,
4          e2 = null;
```

```

5  ListNode l = null;
6  Token t1, t2;
7
8  RecoverySet f1 = new RecoverySet(RPAREN).union(g),
9      f2 = new RecoverySet(RBRACKET).union(g);
10 }
11 {
12     t1 = <NEW>
13     (
14         LOOKAHEAD(2) t2 = <IDENT> <LPAREN> l = arglist(f1) <RPAREN>
15         { e1 = new NewObjectNode(t1, t2, l); }
16     |
17         ( t2 = <INT> | t2 = <STRING> | t2 = <IDENT> )
18         (<LBRACKET> e2 = expression(f2) <RBRACKET>
19             {
20                 if ( l == null )
21                     l = new ListNode(e2);
22                 else
23                     l.add(e2);
24             }
25         )+
26         { e1 = new NewArrayNode(t1, t2, l); }
27     )
28     { return e1; }
29 }
```

Nele, podemos identificar duas partes ou duas produções distintas. A primeira, da linha 13 até a linha 15, reconhece uma expressão que cria um objeto pertencente a uma determinada classe. Nesse caso, o não-terminal constrói e retorna um nó do tipo *NewObjectNode* mostrado no programa 6.34, que possui como filhos um *Token* com o nome da classe da qual deve ser criado o objeto e uma lista de expressões (um nó *ListNode*), que são os argumentos da chamada ao construtor da classe.

Programa 6.34

```

1 package syntacticTree;
2
3 import Token;
4
5 public class NewObjectNode extends ExpreNode {
6     public Token name;
7     public ListNode args;
8
9     public NewObjectNode(Token t, Token t2, ListNode l)
10    {
11        super(t);
12        name = t2;
13        args = l;
14    }
15}
```

O segundo tipo de expressão reconhecida em *aloexpression* é a criação de uma nova variável dimensionada (vetor ou matriz). Isso é feito nas linhas de 17 a 26. Nesse caso, o não-terminal constrói um nó do tipo *NewArrayNode*, que também possui como filhos um *Token* e um *ListNode*. O primeiro representa o tipo dos objetos que compõem o vetor a ser criado e o segundo, as expressões que dão as suas dimensões. A classe *NewArrayNode* é definida como mostra o programa 6.35.

Programa 6.35

```

1 package syntacticTree;
2
3 import Token;
4
5 public class NewArrayNode extends ExpreNode {
6     public Token name;
7     public ListNode dims;
8
9     public NewArrayNode(Token t, Token t2, ListNode d)
10    {
11        super(t);
12        name = t2;
13        dims = d;
14    }
15}

```

Depois do comando de atribuição, temos ainda não-terminais *printstat*, *readstat*, *returnstat* e *superstat* que retornam nós dos tipos *PrintNode*, *ReadNode*, *ReturnNode* e *SuperNode*, respectivamente. Nos programas que seguem, mostramos a definição desses não-terminais e as classes correspondentes aos nós que eles retornam. Não há muito a comentar a respeito deles. Os três primeiros possuem como filho um único nó *ExpreNode*, que, no caso de um *ReturnNode*, pode não existir (igual a *null*) quando o return é usado dentro de um construtor. E o *SuperNode* possui como filho um *ListNode*, com uma lista de expressões que são os argumentos da chamada ao construtor da superclasse, e também pode ser *null*.

Programa 6.36

```

1 PrintNode printstat(RecoverySet g) throws ParseEOFException :
2 {
3     ExpreNode e1 = null;
4     Token t = null;
5 }
6 {
7     try {
8         t = <PRINT> e1 = expression(g)
9         { return new PrintNode(t, e1); }
10    }
11    catch (ParseException e)
12    {
13        consumeUntil(g, e, "printstat");
14        return new PrintNode(t, e1);
15    }
16}

```

```
17
18 ReadNode readstat(RecoverySet g) throws ParseEOFException :
19 {
20     ExpreNode e1 = null;
21     Token t = null;
22 }
23 {
24     try {
25         t = <READ> e1 = lvalue(g)
26         { return new ReadNode(t, e1); }
27     }
28     catch (ParseException e)
29     {
30         consumeUntil(g, e, "readstat");
31         return new ReadNode(t, e1);
32     }
33 }
34
35
36 ReturnNode returnstat(RecoverySet g) throws ParseEOFException :
37 {
38     ExpreNode e1 = null;
39     Token t = null;
40 }
41 {
42     try {
43         t = <RETURN> [ e1 = expression(g)]
44         { return new ReturnNode(t, e1); }
45     }
46     catch (ParseException e)
47     {
48         consumeUntil(g, e, "returnstat");
49         return new ReturnNode(t, e1);
50     }
51 }
52
53
54 SuperNode superstat(RecoverySet g) throws ParseEOFException :
55 {
56     ListNode l = null;
57     Token t = null;
58
59     RecoverySet f = new RecoverySet(RPAREN).union(g);
60 }
61 {
62     try {
63         t = <SUPER> <LPAREN> l = arglist(f) <RPAREN>
64         { return new SuperNode(t, l); }
65     }
66     catch (ParseException e)
67     {
68         consumeUntil(g, e, "superstat");
69         return new SuperNode(t, l);
70 }
```

```
70 }
71 }
```

Programa 6.37

```
1 package syntacticTree;
2
3 import Token;
4
5 public class PrintNode extends StatementNode {
6     public ExpreNode expr;
7
8     public PrintNode(Token t, ExpreNode e)
9     {
10         super(t);
11         expr = e;
12     }
13 }
```

Programa 6.38

```
1 package syntacticTree;
2
3 import Token;
4
5 public class ReadNode extends StatementNode {
6     public ExpreNode expr;
7
8     public ReadNode(Token t, ExpreNode e )
9     {
10         super(t);
11         expr = e;
12     }
13 }
```

Programa 6.39

```
1 package syntacticTree;
2
3 import Token;
4
5 public class ReturnNode extends StatementNode {
6     public ExpreNode expr;
7
8     public ReturnNode(Token t, ExpreNode e)
9     {
10         super(t);
11         expr = e;
12     }
13 }
```

Programa 6.40

```

1 package syntacticTree;
2
3 import Token;
4
5 public class SuperNode extends StatementNode {
6     public ListNode args;
7
8     public SuperNode(Token t, ListNode p)
9     {
10         super(t);
11         args = p;
12     }
13 }
```

*

O não-terminal *ifstat* retorna um nó do tipo *IfNode*, como mostra o programa 6.41.

Programa 6.41

```

1 IfNode ifstat(RecoverySet g) throws ParseEOFException :
2 {
3     ExpreNode e1 = null;
4     StatementNode s1 = null,
5             s2 = null;
6     Token t = null;
7
8     RecoverySet f1 = new RecoverySet(RPAREN).union(g),
9                     f2 = new RecoverySet(ELSE).union(g);
10    }
11    {
12        try {
13            t = <IF> <LPAREN> e1 = expression(f1) <RPAREN> s1 = statement(f2)
14            [LOOKAHEAD(1) <ELSE> s2 = statement(g)]
15            { return new IfNode(t, e1, s1, s2); }
16        }
17        catch (ParseException e)
18        {
19            consumeUntil(g, e, "ifstat");
20            return new IfNode(t, e1, s1, s2);
21        }
22    }
```

*

Esse nó possui como filhos um *ExpreNode*, que representa a condição de teste do comando, e dois *StatementNodes*, que correspondem aos ramos verdadeiro e falso do comando *if*. O segundo filho *StatementNode* pode não existir quando não houver o ramo falso. Vemos, no programa 6.42, a classe *IfNode* e, na figura 6.15, um exemplo de árvore para o trecho:

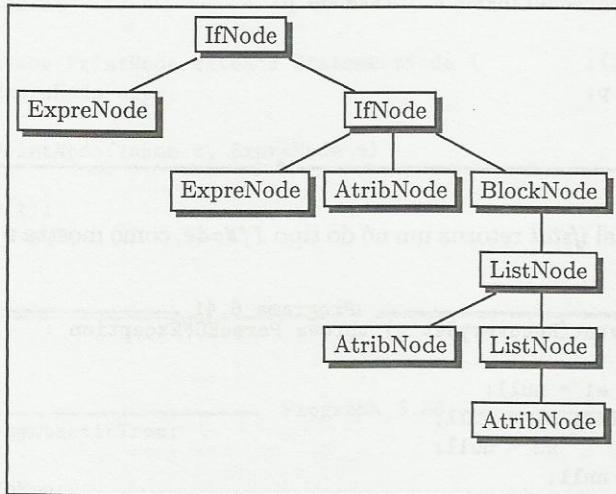
```

if ( a == b )
    if ( c < d )
        a = 10;
```

```

else
{
  b = 0;
  c = 1024;
}

```

Figura 6.15 – Exemplo de árvore sintática para *ifstat*.

Programa 6.42

```

1 package syntacticTree;
2
3 import Token;
4
5 public class IfNode extends StatementNode {
6   public StatementNode stat1, stat2;
7   public ExpreNode expr;
8
9   public IfNode(Token t, ExpreNode e, StatementNode s1, StatementNode s2)
10  {
11     super(t);
12     expr = e;
13     stat1 = s1;
14     stat2 = s2;
15   }
16 }

```

De maneira similar, o não-terminal *forstat* (Programa 6.43) constrói um nó *ForNode* (Programa 6.44), cujos filhos são dois *AtribNode*, que correspondem à inicialização e ao incremento do for, um *ExpreNode*, que corresponde à condição de controle, e um *StatementNode*, que é o corpo do comando. Note que aqui podemos definir os filhos

init e *incr* como *AtribNodes*, pois sabemos com certeza que temos esse tipo de nó retornado pela chamada a *atribstat* nas linhas 14 e 16 do não-terminal *forstat*.

Programa 6.43

```

1 package syntacticTree;
2
3 import Token;
4
5 public class ForNode extends StatementNode {
6     public AtribNode init, incr;
7     public StatementNode stat;
8     public ExpreNode expr;
9
10    public ForNode(Token t, ExpreNode e, AtribNode s1, AtribNode s2,
11                  StatementNode s3)
12    {
13        super(t);
14        expr = e;
15        init = s1;
16        incr = s2;
17        stat = s3;
18    }
19}

```

Programa 6.44

```

1 ForNode forstat(RecoverySet g) throws ParseEofException :
2 {
3     AtribNode s1 = null,
4             s2 = null;
5     StatementNode s3 = null;
6     ExpreNode e1 = null;
7     Token t = null;
8
9     RecoverySet f1 = new RecoverySet(SEMICOLON).union(g),
10                  f2 = new RecoverySet(RPAREN).union(g);
11 }
12 {
13     try {
14         t = <FOR> <LPAREN> [s1 = atribstat(f1)] <SEMICOLON>
15                         [e1 = expression(f1)] <SEMICOLON>
16                         [s2 = atribstat(f2)] <RPAREN>
17                         s3 = statement(g)
18         { return new ForNode(t, e1, s1, s2, s3); }
19     }
20     catch (ParseException e)
21     {
22         consumeUntil(g, e, "forstat");
23         return new ForNode(t, e1, s1, s2, s3);
24     }
25 }

```

A figura 6.16 apresenta um exemplo de árvore sintática que representa o trecho de programa

```
for ( i = 0 ; i < a; i = i + 1)
    for (j = i + 1; j < b; j = j + 1)
{
    b = 0;
    c = 1024;
}
```

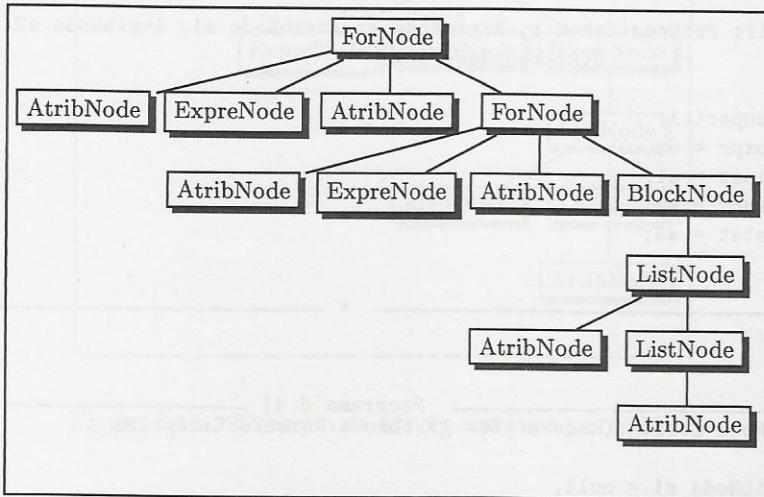


Figura 6.16 – Exemplo de árvore sintática para *forstat*.

Faltam, agora, os não-terminais que reconhecem e montam a árvore sintática para as expressões da linguagem *X⁺⁺*. Como dissemos, cada tipo de expressão é representado por um nó distinto. Todos esses nós são subclasses da classe *ExpreNode* mostrada no programa 6.27. O primeiro tipo de expressão é reconhecido pelo não-terminal *expression*, apresentado no programa 6.45 e que constrói um nó para uma expressão relacional.

Programa 6.45

```

1  ExpreNode expression(RecoverySet g) throws ParseE0FException :
2  {
3      ExpreNode e1 = null, e2 = null;
4      Token t = null;
5
6  }
7  {
8      try {
9          e1 = numexpr()
10         [

```

```

11     ( t = <LT> | t = <GT> | t = <LE> | t = <GE> | t = <EQ> | t = <NEQ>
12     e2 = numexpr()
13     { e1 = new RelationalNode(t, e1, e2); }
14   ]
15   { return e1; }
16 }
17 catch (ParseException e)
18 {
19   consumeUntil(g, e, "expression");
20   return new RelationalNode(t, e1, e2);
21 }
22 }

```

Na construção do nó correspondente a uma expressão relacional, esse não-terminal faz duas chamadas a *numexpr* que retornam dois *ExpreNodes*, que estarão envolvidos na expressão relacional. Para completar o nó, um *Token* que indica qual é o operador também é incluído no nó *RelationalNode*. Esse nó, portanto é definido como mostra o programa 6.46.

Programa 6.46

```

1 package syntacticTree;
2
3 import Token;
4
5 public class RelationalNode extends ExpreNode {
6   public ExpreNode expr1, expr2;
7
8   public RelationalNode(Token t, ExpreNode e1, ExpreNode e2)
9   {
10     super(t);
11     expr1 = e1;
12     expr2 = e2;
13   }
14 }

```

Note que o não-terminal *expression* retorna um *ExpreNode*, e não um *RelationalNode*. Isso porque nem sempre há casamento com a segunda parte do não-terminal, ou seja, o casamento com um operador relacional, e a segunda chamada a *numexpr* nem sempre acontece. Assim, pode ser que a expressão retornada pelo método seja aquela retornada pela primeira chamada *numexpr*. Nesse caso, o tipo da expressão não deve ser *RelationalNode*.

Em seguida, temos o não-terminal *numexpr*, que é bastante semelhante a *expression*. A diferença consiste em poder ter a segunda parte da produção repetida diversas vezes. Por isso, a cada vez que acontece um casamento com essa parte da produção, um novo não-terminal do tipo *AddNode* é criado.

Programa 6.47

```

1 package syntacticTree;
2
3 import Token;
4
5 public class AddNode extends ExpreNode {
6     public ExpreNode expr1, expr2;
7
8     public AddNode(Token t, ExpreNode e1, ExpreNode e2)
9     {
10         super(t);
11         expr1 = e1;
12         expr2 = e2;
13     }
14 }
```

*

Por exemplo, se tivermos a expressão $a + b + c$, teremos *numexpr* construindo dois nós do tipo *AddNode*: um nó *e1* para a primeira soma, que possui dois *VarNodes* como filhos, e um segundo que possui *e1* como primeiro filho, e um *VarNode* para *c* como segundo filho. Essa situação é apresentada na figura 6.17. É importante notar que a árvore sintática montada dessa forma respeita a semântica desejada para a expressão, ou seja, $a + b$ é tratada como uma subexpressão que forma um dos operadores para a segunda adição com *c*. Isso é muito importante, pois uma árvore construída de forma diversa iria alterar o significado e a forma de avaliação da expressão. Na verdade, a própria gramática tem que ser construída de forma a preservar a prioridade (por meio de diversos níveis de não-terminais como *expression*, *numexpr*, *term* etc) e a associatividade dos operadores.

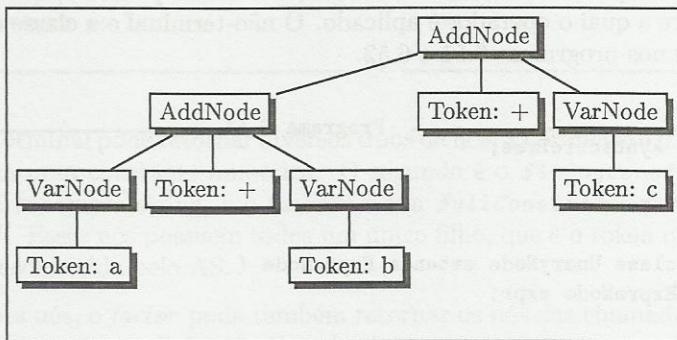
Programa 6.48

```

1 ExpreNode numexpr() throws ParseEUFException :
2 {
3     ExpreNode e1 = null, e2;
4     Token t;
5 }
6 {
7     e1 = term()
8     (
9         (t = <PLUS> | t = <MINUS>)
10        e2 = term()
11        { e1 = new AddNode(t, e1, e2); }
12    )*
13    {return e1; }
14 }
```

*

O não-terminal *term*, mostrado no programa 6.49, é análogo ao *numexpr*, porém trata de outros tipos de operadores aritméticos da linguagem *X⁺⁺*. Da mesma forma, o nó que ele constrói, que é o *MultNode* (Programa 6.50), é similar ao *AddNode*.

Figura 6.17 – Exemplo de árvore sintática para *numexpr*.

Programa 6.49

```

1  ExpreNode term() throws ParseEOFException :
2  {
3      ExpreNode e1 = null, e2;
4      Token t;
5  }
6  {
7      e1 = unaryexpr()
8      (
9          ( t = <STAR> | t = <SLASH>| t = <REM>)
10         e2 = unaryexpr()
11         { e1 = new MultNode(t, e1, e2); }
12     )*
13     { return e1; }
14 }
```

Programa 6.50

```

1  package syntacticTree;
2
3  import Token;
4
5  public class MultNode extends ExpreNode {
6      public ExpreNode expr1, expr2;
7
8      public MultNode(Token t, ExpreNode e1, ExpreNode e2)
9      {
10         super(t);
11         expr1 = e1;
12         expr2 = e2;
13     }
14 }
```

Uma operação unária é reconhecida no não-terminal *unaryexpr*, que monta um nó do tipo *UnaryNode* que possui dois filhos: um *Token* que contém o símbolo repre-

sentante da operação a ser efetuada e outro que é um *ExpreNode*, que representa a expressão sobre a qual o operador é aplicado. O não-terminal e a classe do nó gerado são mostrados nos programas 6.51 e 6.52.

Programa 6.51

```

1 package syntacticTree;
2
3 import Token;
4
5 public class UnaryNode extends ExpreNode {
6     public ExpreNode expr;
7
8     public UnaryNode(Token t, ExpreNode e)
9     {
10         super(t);
11         expr = e;
12     }
13 }
```

*

Programa 6.52

```

1 ExpreNode unaryexpr() throws ParseEOFException :
2 {
3     ExpreNode e;
4     Token t = null;
5 }
6 {
7     [( t = <PLUS> | t = <MINUS>)] e = factor()
8     { return ( (t == null) ? e : new UnaryNode(t, e));}
9 }
```

*

E, finalmente, temos os tipos mais básicos de expressões que são reconhecidos pelo não-terminal *factor*, no programa 6.53.

Programa 6.53

```

1 ExpreNode factor() throws ParseEOFException :
2 {
3     ExpreNode e = null;
4     Token t;
5 }
6 {
7     (
8         t = <int_constant> { e = new IntConstNode(t); }
9     |
10        t = <string_constant> { e = new StringConstNode(t); }
11    |
12        t = <null_constant> { e = new NullConstNode(t); }
13    |
14        e = lvalue(null)
15    |
16        <LPAREN> e = expression(null) <RPAREN>
```

```

17     )
18     { return e; }
19 }

```

*

Esse não-terminal pode retornar diversos tipos de nós. O primeiro é o *IntConstNode*, que representa uma constante numérica. O segundo é o *StringConstNode*, que representa uma constante string, e o terceiro é um *NullConstNode*, que representa a constante *null*. Esses nós possuem todos um único filho, que é o token que representa a constante reconhecida pelo AS.

Além desses nós, o *factor* pode também retornar os nós das chamadas a *lvalue* na linha 14 ou a *expression* na linha 16. No primeiro caso, como já vimos, podemos ter um *VarNode*, um *DotNode*, um *IndexNode* ou um *CallNode*, e, no segundo caso, qualquer tipo retornado por *expression*, que são todas as subclasses de *ExpreeNode*. A figura 6.18 apresenta a árvore sintática correspondente à expressão $a + b.x < -3 * c[2]$ que utiliza diversos dos nós descritos.

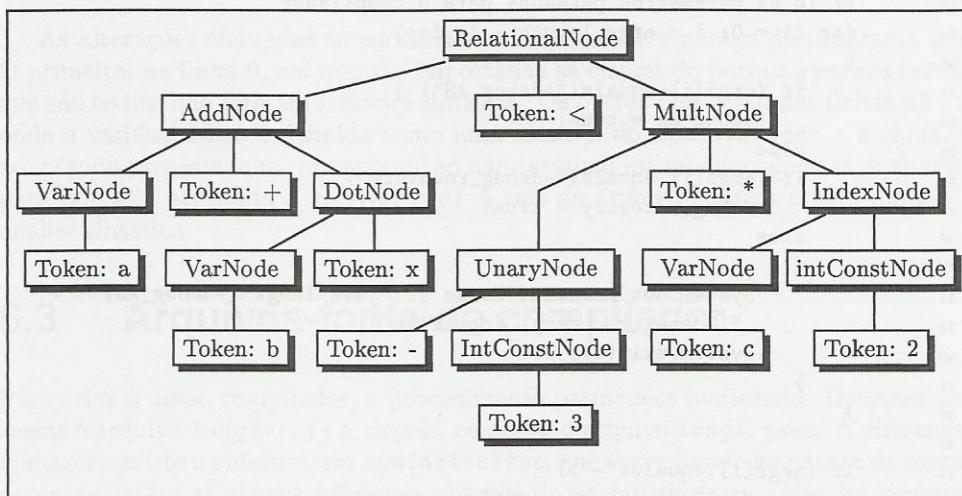


Figura 6.18 – Exemplo de árvore sintática para a expressão $a + b.x < -3 * c[2]$.

Por último, veremos quais são as modificações necessárias no método *main* do nosso AS para que ocorra a montagem da árvore sintática.

Programa 6.54

```

1 PARSER_BEGIN(langX)
2 package parser;
3
4 import java.io.*;
5 import recovery.*; // importa as classes de recuperação de erros do AS
6 import syntacticTree.*; // importa as classes dos nós da árvore sintática
7
8

```

```
9  public class langX {
10    final static String Version = "X++ Compiler - Version 1.0 - 2004";
11    int contParseError = 0;           // contador de erros sintáticos
12    boolean debug_recovery;        // controla verbose de recuperação de erros
13    Token lastError = null;
14
15
16    // Define o método "main" da classe langX.
17    public static void main(String args[]) throws ParseException
18    {
19        boolean debug_as = false;
20        boolean debug_recovery = false;
21
22        String filename = ""; // nome do arquivo a ser analisado
23        langX parser;       // analisador léxico/sintático
24        int i;
25        boolean ms = false;
26
27        System.out.println(Version);
28        // lê os parâmetros passados para o compilador
29        for (i = 0; i < args.length - 1; i++)
30        {
31            if (args[i].equals("-debug_AS") )
32                debug_as = true;
33            else
34                if (args[i].equals("-debug_recovery") )
35                    debug_recovery = true;
36            else
37            {
38                System.out.println("Usage is: java langX [-debug_AS] " +
39                               "[-debug_recovery] inputfile");
40                System.exit(0);
41            }
42        }
43
44        if (args[i].equals("-"))
45        {           // lê da entrada-padrão
46            System.out.println("Reading from standard input . . .");
47            parser = new langX(System.in); // cria AS
48        }
49        else
50        {           // lê do arquivo
51            filename = args[args.length-1];
52            System.out.println("Reading from file " + filename + " . . .");
53            try { // cria AS
54                parser = new langX(new java.io.FileInputStream(filename));
55            }
56            catch (java.io.FileNotFoundException e) {
57                System.out.println("File " + filename + " not found.");
58                return;
59            }
60        }
61    }
```

```

62     ListNode root = null;
63     parser.debug_recovery = debug_recovery;
64     if (! debug_as) parser.disable_tracing(); // desab. verbose do AS
65     try {
66         root = parser.program();    // chama o método que faz a análise
67     }
68     catch (ParseException e)
69     {
70         System.err.println(e.getMessage());
71     }
72     finally {
73         System.out.println(parser.token_source.foundLexError() +
74                         " Lexical Errors found");
75         System.out.println(parser.contParseError +
76                         " Syntactic Errors found");
77     }
78 }
79 } // main

```

As alterações efetuadas se encontram em apenas três pontos do programa 6.54. O primeiro, na linha 6, em que são importadas as classes do pacote *syntacticTree*, que são os nós que formam a árvore sintática. Os outros dois estão nas linhas 62 e 66 onde a variável *root* é definida como uma variável do tipo *ListNode* e a chamada ao método *program*, que corresponde ao não-terminal inicial da gramática, é alterada para que seja atribuída à variável *root* a raiz da árvore sintática criada durante a análise sintática.

6.3 Arquivos-fonte do compilador

Para criar o nosso compilador, o procedimento permanece inalterado. Devemos processar o arquivo *langX++.jj* e, depois, compilar o arquivo *langX.java*. A diferença é que agora existe o subdiretório *syntacticTree*, que corresponde ao pacote de mesmo nome e contém as classes referentes aos nós da árvore sintática. Não há nenhuma maneira de sabermos, por enquanto, se a árvore sintática construída corresponde ao que desejamos ou não. Isso será abordado no capítulo 7, onde veremos como imprimir a árvore sintática.

Capítulo 7

Exibição da Árvore Sintática

Neste capítulo veremos como visualizar a árvore sintática construída pelo nosso AS. Com isso, podemos verificar se nossa implementação da árvore sintática está correta. A partir de agora, não utilizaremos mais o arquivo .jj para fazermos nossas implementações. O trabalho do AS termina ao construir a árvore sintática e a partir deste momento, iremos sempre construir programas que atuam com base nela.

Além de permitir a verificação da árvore sintática, o código mostrado neste capítulo serve como uma introdução às demais fases do nosso compilador. A exibição da árvore sintática é efetuada por um conjunto de métodos que analisam os nós da árvore e adotam as ações necessárias para exibi-los; para cada tipo de nó, existe um ou mais métodos correspondentes. A análise semântica e a geração de código se processam exatamente da mesma forma. Cada nó da árvore sintática deve ser visitado utilizando os métodos correspondentes aos tipos de nós e que realizam a tarefa desejada, quer seja análise semântica, geração de código, quer seja simplesmente a impressão da árvore sintática.

Iremos inicialmente, estabelecer como queremos visualizar a árvore sintática e, depois, descrever as duas fases necessárias para que isso possa ser feito e como implementá-las. Essas duas fases são a numeração dos nós da árvore e a exibição propriamente dita.

7.1 Exibição da árvore sintática

Vamos utilizar uma forma bastante simples de visualizar a árvore sintática, ou seja, inicialmente numerar os seus nós e, depois, exibi-los na ordem em que foram numerados, mostrando o tipo de cada nó e quais os números dos seus filhos. A numeração deve iniciar-se pela raiz e ser feita primeiro em profundidade, ou seja, seguindo cada ramo da árvore até o final.

Vamos tomar como exemplo o programa a seguir:

```
class a {
    string b;

    constructor(int c)
    {
        b = c + "";
    }

    int m()
    {
        print "b: " + b;
    }
}
```

Seguindo a numeração descrita, obtemos a árvore sintática da figura 7.1. Uma vez feita a numeração, vamos exibir a árvore mostrando seqüencialmente cada um dos nós. Para cada nó, mostramos qual é o seu tipo e quais são os números de seus filhos. Como os nós que são *Tokens* não são numerados, exibimos direto seu conteúdo em vez de mostrar seu número.

Para o exemplo apenas visto, teríamos a seguinte saída:

```
1: ListNode (ClassDeclNode) ===> 2 null
2: ClassDeclNode ===> a null 3
3: ClassBodyNode ===> null 4 8 22
4: ListNode (VarDeclNode) ===> 5 null
5: VarDeclNode ===> string 6
6: ListNode (VarNode) ===> 7 null
7: VarNode ===> b
8: ListNode (ConstructDeclNode) ===> 9 null
9: ConstructDeclNode ===> 10
10: MethodBodyNode ===> 11 15
11: ListNode (VarDeclNode) ===> 12 null
12: VarDeclNode ===> int 13
13: ListNode (VarNode) ===> 14 null
14: VarNode ===> c
15: BlockNode ===> 16
16: ListNode (StatementNode) ===> 17 null
17: AtribNode ===> 18 19
18: VarNode ===> b
19: AddNode ===> 20 + 21
20: VarNode ===> c
21: StringConstNode ===> ""
22: ListNode (MethodDeclNode) ===> 23 null
```

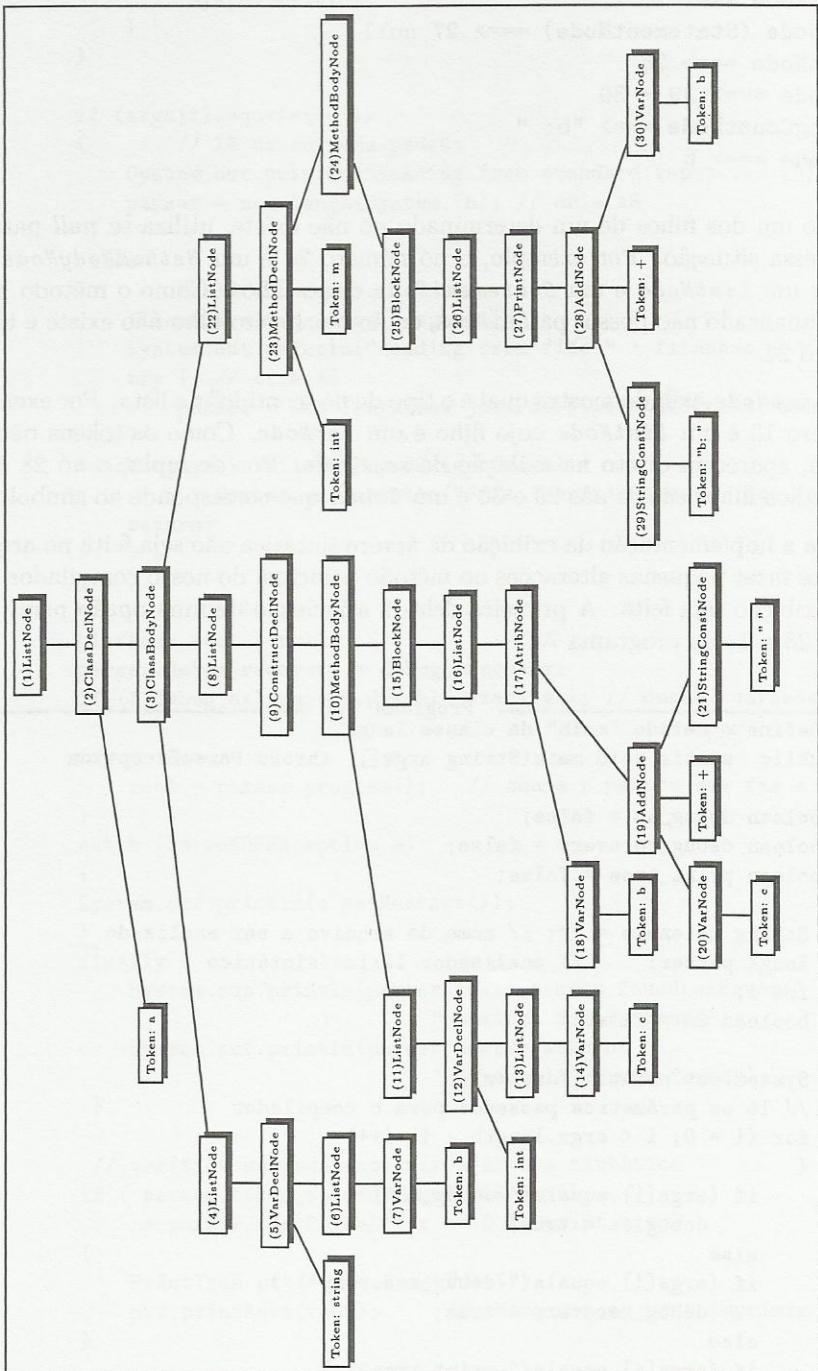


Figura 7.1 – Exemplo de árvore sintática numerada.

```

23: MethodDeclNode ===> int m 24
24: MethodBodyNode ===> null 25
25: BlockNode ===> 26
26: ListNode (StatementNode) ===> 27 null
27: PrintNode ===> 28
28: AddNode ===> 29 + 30
29: StringConstNode ===> "b: "
30: VarNode ===> b

```

Quando um dos filhos de um determinado nó não existe, utiliza-se *null* para representar essa situação. Por exemplo, o nó número 24 é um *MethodBodyNode* que deveria ter um *ListNode* e um *StatementNode* como filhos. Como o método *m* do programa analisado não possui parâmetros, então o primeiro filho não existe e temos *null* no nó 24.

Cada *ListNode* exibido mostra qual é o tipo de nó “contido” na lista. Por exemplo, o nó número 13 é um *ListNode* cujo filho é um *VarNode*. Como os tokens não são numerados, aparecem direto na exibição de seus pais. Por exemplo, o nó 28 é um *AddNode* cujos filhos são os nós 29 e 30 e um *Token* que corresponde ao símbolo *+*.

Embora a implementação da exibição da árvore sintática não seja feita no arquivo *.jj*, devemos fazer pequenas alterações no método principal do nosso compilador para que essa exibição seja feita. A primeira delas é a inclusão de uma opção *print_tree* nas linhas 23 a 24 do programa 7.1.

Programa 7.1

```

1 // Define o método "main" da classe langX.
2 public static void main(String args[]) throws ParseException
3 {
4     boolean debug_as = false;
5     boolean debug_recovery = false;
6     boolean print_tree = false;
7
8     String filename = ""; // nome do arquivo a ser analisado
9     langX parser;        // analisador léxico/sintático
10    int i;
11    boolean ms = false;
12
13    System.out.println(Version);
14    // lê os parâmetros passados para o compilador
15    for (i = 0; i < args.length - 1; i++)
16    {
17        if (args[i].equals("-debug_AS") )
18            debug_as = true;
19        else
20            if (args[i].equals("-debug_recovery") )
21                debug_recovery = true;
22        else
23            if (args[i].equals("-print_tree") )
24                print_tree = true;
25        else
26        {

```

```

27         System.out.println("Usage is: java langX [-debug_AS] " +
28             "[-debug_recovery] [-print_tree] inputfile");
29         System.exit(0);
30     }
31 }
32
33 if (args[i].equals("-"))
34 {
35     // lê da entrada-padrão
36     System.out.println("Reading from standard input . . .");
37     parser = new langX(System.in); // cria AS
38 }
39 else
40 {
41     // lê do arquivo
42     filename = args[args.length-1];
43     System.out.println("Reading from file " + filename + " . . .");
44     try { // cria AS
45         parser = new langX(new java.io.FileInputStream(filename));
46     }
47     catch (java.io.FileNotFoundException e) {
48         System.out.println("File " + filename + " not found.");
49         return;
50     }
51 }
52
53 ListNode root = null;
54 parser.debug_recovery = debug_recovery;
55 if (! debug_as) parser.disable_tracing(); // desab. verbose do AS
56
57 try {
58     root = parser.program(); // chama o método que faz a análise
59 }
60 catch (ParseException e)
61 {
62     System.err.println(e.getMessage());
63 }
64 finally {
65     System.out.println(parser.token_source.foundLexError() +
66                         " Lexical Errors found");
67     System.out.println(parser.contParseError +
68                         " Syntactic Errors found");
69 }
70
71 // verifica se pode imprimir a árvore sintática
72 if ( parser.token_source.foundLexError()
73     + parser.contParseError == 0 && print_tree)
74 {
75     PrintTree prt = new PrintTree();
76     prt.printRoot(root); // chama método para imprimir árvore
77 }
78 } // main

```

Para implementar a numeração e exibição da árvore sintática, existem diversas alternativas. As mais simples delas seriam incluir, para cada operação como numerar ou exibir, um método em cada uma das classes correspondentes aos nós da árvore; criar uma única classe e dentro desta incluir métodos que aceitam como argumento um nó e realizam as operações desejadas sobre esse nó.

A primeira é, sem dúvida, mais elegante, pois mantém a filosofia da programação orientada a objetos. Porém, causa um certo transtorno quando temos que alterar ou corrigir nossa implementação, pois esta está espalhada por diversos arquivos diferentes. A segunda, apesar de deselegante, diminui o esforço requerido para mantermos nossa implementação. Assim, vamos criar uma classe *PrintTree* dentro do pacote *syntacticTree*, que é responsável por exibir a árvore sintática. Nas linhas 70 a 75 do programa 7.1 é criado um objeto do tipo *PrintTree* e chamado o método *printRoot* passando-se como argumento o nó raiz da árvore sintática, retornado pelo AS. Note que essa operação só poderá ser realizada se o número de erros encontrados nas análises léxica e sintática for zero.

Nas próximas seções, mostraremos como a classe *PrintTree* é implementada.

7.1.1 Numeração dos nós

A classe *PrintTree* é definida dentro do pacote *syntacticTree*, com as classes que correspondem aos nós da árvore sintática. Primeiramente, ela é responsável por numerar cada um dos nós da árvore. Para isso, são definidos dentro dela métodos específicos para aplicar essa operação a cada tipo de nó da árvore. Por exemplo, temos um método para numerar um *ClassDeclNode*, um para *ClassBodyNode*, e assim por diante.

A estrutura da classe pode ser representada como mostra o programa 7.2

Programa 7.2

```

1 package syntacticTree;
2
3 import Token;
4
5 public class PrintTree {
6
7     int kk;
8
9     public PrintTree()
10    {
11         kk = 1;           // inicializa contador de nós
12    }
13
14
15    public void printRoot(ListNode x)
16    {
17        if ( x == null )
18            System.out.println("Empty syntatic tree. Nothing to be printed");
19        else
20        {

```

```

21     numberClassDeclListNode(x);
22     printClassDeclListNode(x);
23   }
24   System.out.println();
25 }
26
27 (MÉTODOS PARA NUMERAR OS NÓS)
28
29 (MÉTODOS PARA EXIBIR OS NÓS)
30
31 }
```

A variável *kk* é o contador do objeto *PrintTree*, que diz qual é o número do próximo nó a ser numerado. O construtor da classe simplesmente inicializa esse valor em 1. O método *printRoot* recebe como parâmetro o nó raiz da árvore e incumbe-se de chamar os métodos que vão numerar e exibir os nós da árvore sintática, que são *numberClassDeclListNode* e *printClassDeclListNode*. Esse método é chamado pelo compilador para executar a exibição da árvore, como vimos no programa 7.1.

O método que numera um determinado tipo de nó não precisa desempenhar muitas tarefas. Ele deve apenas:

- atribuir ao nó que lhe foi passado como argumento o valor da variável *kk*;
- incrementar *kk*;
- chamar os métodos para numeração, passando como argumento os filhos do nó que lhe foi passado como argumento.

Para poder atribuir um número ao nó, vamos modificar a classe *GeneralNode*, incluindo uma nova variável *number* para armazenar essa informação. A nova classe é mostrada no programa 7.3.

Programa 7.3

```

1 package syntacticTree;
2
3 import Token;
4
5 abstract public class GeneralNode {
6   public Token position;
7   public int number;
8
9   public GeneralNode(Token x)
10  {
11     position = x;
12     number = 0;
13   }
14 }
```

Vamos tomar como exemplo o método que numera um nó do tipo *ClassBodyNode*. Usou-se como padrão incluir o prefixo *number* mais o tipo do nó para denominar o método, obtendo-se, nesse caso, *numberClassBodyNode*. Ele é simplesmente definido como mostra o programa 7.4.

Programa 7.4

```

1 public void numberClassBodyNode(ClassBodyNode x)
2 {
3     if ( x == null ) return;
4     x.number = kk++;
5     numberClassDeclListNode(x.clist);
6     numberVarDeclListNode(x.vlist);
7     numberConstructDeclListNode(x.ctlist);
8     numberMethodDeclListNode(x.mlist);
9 }
```

Inicialmente, verifica se o valor passado como argumento não é *null*. Isso é necessário, pois alguns nós possuem filhos que podem ou não aparecer. Quando não aparecem, seu valor é *null*. Para evitarmos a verificação para toda chamada de método feita para os nós filhos, nós a incluímos no início do método, assim a chamada sempre é feita para os nós filhos. Se eles não existirem, o método simplesmente retorna, sem nada fazer.

Em seguida, ele atribui para a variável *number* do objeto que foi passado como argumento o valor de *kk* e incrementa *kk*. Em seguida, chama para cada um dos seus filhos os métodos respectivos que fazem a numeração.

Note que muitas vezes os filhos de um nó não possuem números consecutivos. No caso do método *numberClassBodyNode*, por exemplo, o filho *x.clist* corresponde à raiz de uma subárvore que será toda numerada antes que o filho *x.vlist* seja numerado. Se o nó *x* recebe o número *N* e a subárvore de *x.clist* possui *J* nós a serem numerados (excluindo-se os nós do tipo *Token*), então o filho *x.vlist* recebe o número *N + J + 1*.

Não vamos mostrar aqui os métodos para todos os tipos de nós, pois são todos muito simples e muito parecidos com o método *numberClassBodyNode* que descrevemos anteriormente. Vamos apenas comentar alguns casos especiais. O primeiro é referente aos nós do tipo *ListNode*.

Como já sabemos, um *ListNode* serve para agrupar diversos nós do mesmo tipo na forma de uma lista. Assim, ao visitarmos um nó desse tipo, queremos, na verdade, visitar cada elemento que compõe a lista. O método que numera um *ListNode* deveria, então, assemelhar-se com o programa 7.5.

Programa 7.5

```

1 public void numberListNode(ListNode x)
2 {
3     if ( x == null ) return;
4     // numera x.node
5     numberListNode(x.next);
6 }
```

O leitor pode perceber o problema na linha 4. Precisamos saber qual é o tipo de *x.node* para que possamos chamar o método correto para esse tipo de nó. E não adianta construirmos uma método *numberGeneralNode* que aceite como parâmetro um *GeneralNode*, ou seja, qualquer tipo de nó, porque dentro desse método não saberíamos como continuar a numerar os nós filhos.

Por isso, precisamos ter um método para numerar cada tipo de lista que possa aparecer na nossa árvore sintática. Por exemplo, o nó raiz da árvore é um *ListNode* cujos elementos são *ClassDeclNodes*. Assim, chamamos para esse *ListNode* o método *numberClassDeclListNode*, mostrado a seguir, no programa 7.6. Como sabemos que seus elementos são sempre *ClassDeclNodes*, podemos chamar o método correto para numerá-los, que é o *numberClassDeclNode*. Vale notar que o filho *node* da classe *ListNode* foi declarado como *GeneralNode* e, por isso, precisamos utilizar um operador de *cast* ao chamarmos *numberClassDeclNode* na linha 6.

Programa 7.6

```

1  public void numberClassDeclListNode(ListNode x)
2  {
3
4      if(x == null) return;
5      x.number = kk++;
6      numberClassDeclNode((ClassDeclNode) x.node);
7      numberClassDeclListNode(x.next);
8  }

```

Para o nosso AS, temos sete tipos distintos de listas e, portanto, sete métodos para numerar os *ListNode*s: *numberClassDeclListNode* (lista de *ClassDeclNode*), *numberVarDeclListNode* (lista de *VarDeclNode*), *numberVarListNode* (lista de *VarNode*), *numberConstructDeclListNode* (lista de *ConstructDeclNode*), *numberMethodDeclListNode* (lista de *MethodDeclNode*), *numberStatementListNode* (lista de *statementNode*) e *numberExpreListNode* (lista de *ExpreNode*).

Um outro ponto com o qual devemos ter cuidado são aqueles nós que têm como filho algum nó do tipo *StatementNode*. Por exemplo, para numerar um *MethodBodyNode*, temos o método do programa 7.7.

Programa 7.7

```

1  public void numberMethodBodyNode(MethodBodyNode x)
2  {
3      if (x == null) return;
4      x.number = kk++;
5      numberVarDeclListNode(x.param);
6      numberStatementNode(x.stat);
7  }

```

Ele chama *numberStatementNode* para numerar a parte da árvore sintática correspondente ao(s) comando(s) do método. Mas o que deve fazer esse método? Como vimos, deve numerar o nó que lhe foi passado como parâmetro e, então, chamar os métodos para numerarem os seus filhos. Porém, o nó que é apontado por *x.stat* não

é realmente um *StatementNode*, pois essa é uma classe abstrata. E cada uma das suas subclasses possui filhos diferentes, o que faz com que *numberStatementNode* não possa fazer a chamada correta para os filhos do nó que foi passado como argumento, a não ser que possa descobrir qual é o tipo real desse nó.

O que fazemos, então, é criar um *numberStatementNode* que simplesmente chama outros métodos específicos, dependendo do tipo do nó que lhe é passado como argumento. É o que é nos mostra o programa 7.8.

Programa 7.8

```

1  public void numberStatementNode(StatementNode x)
2  {
3      if (x instanceof BlockNode)
4          numberBlockNode( (BlockNode) x);
5      else
6          if (x instanceof VarDeclNode)
7              numberVarDeclNode( (VarDeclNode) x);
8          else
9              if (x instanceof AtribNode)
10                 numberAtribNode( (AtribNode) x);
11             else
12                 if (x instanceof IfNode)
13                     numberIfNode( (IfNode) x);
14                 else
15                     if (x instanceof ForNode)
16                         numberForNode( (ForNode) x);
17                     else
18                         if (x instanceof PrintNode)
19                             numberPrintNode( (PrintNode) x);
20                         else
21                             if (x instanceof NopNode)
22                                 numberNopNode( (NopNode) x);
23                             else
24                                 if (x instanceof ReadNode)
25                                     numberReadNode( (ReadNode) x);
26                                 else
27                                     if (x instanceof ReturnNode)
28                                         numberReturnNode( (ReturnNode) x);
29                                     else
30                                         if (x instanceof SuperNode)
31                                             numberSuperNode( (SuperNode) x);
32                                         else
33                                             if (x instanceof BreakNode)
34                                                 numberBreakNode( (BreakNode) x);
35 }

```

*

E para cada tipo específico de comando, temos um método como o *numberIfNode*, que numera um nó do tipo *IfNode* (Programa 7.9).

Programa 7.9

```

1 public void numberIfNode(IfNode x)
2 {
3     if (x == null) return;
4     x.number = kk++;
5     numberExpreNode(x.expr);
6     numberStatementNode(x.stat1);
7     numberStatementNode(x.stat2);
8 }

```

*

O mesmo problema que ocorre com o *StatementNode* ocorre também com o *ExpreNode*. Nós que têm como filho um *ExpreNode* podem, na realidade, ter diversos tipos de filhos, como *AddNode*, *UnaryNode* ou *intconstantNode*, por exemplo. Assim, tais nós fazem uma chamada a *numberExpreNode* que deve verificar qual é o tipo real do nó e chamar o método adequado. Temos, então, o programa 7.10 que mostra *numberExpreNode*.

Programa 7.10

```

1 public void numberExpreNode(ExpreNode x)
2 {
3     if (x instanceof NewObjectNode)
4         numberNewObjectNode( (NewObjectNode) x);
5     else
6         if (x instanceof NewArrayNode)
7             numberNewArrayNode( (NewArrayNode) x);
8     else
9         if (x instanceof RelationalNode)
10            numberRelationalNode( (RelationalNode) x);
11     else
12         if (x instanceof AddNode)
13             numberAddNode( (AddNode) x);
14     else
15         if (x instanceof MultNode)
16             numberMultNode( (MultNode) x);
17     else
18         if (x instanceof UnaryNode)
19             numberUnaryNode( (UnaryNode) x);
20     else
21         if (x instanceof CallNode)
22             numberCallNode( (CallNode) x);
23     else
24         if (x instanceof IntConstNode)
25             numberIntConstNode( (IntConstNode) x);
26     else
27         if (x instanceof StringConstNode)
28             numberStringConstNode( (StringConstNode) x);
29     else
30         if (x instanceof NullConstNode)
31             numberNullConstNode( (NullConstNode) x);
32     else
33         if (x instanceof IndexNode)
34             numberIndexNode( (IndexNode) x);

```

```

35     else
36         if (x instanceof DotNode)
37             numberDotNode( (DotNode) x );
38         else
39             if (x instanceof VarNode)
40                 numberVarNode( (VarNode) x );
41 }

```

E temos, para cada tipo de expressão, o seu método para numeração, como o programa 7.11 para o *AddNode*.

Programa 7.11

```

1 public void numberAddNode(AddNode x)
2 {
3     if (x == null) return;
4     x.number = kk++;
5     numberExpreNode(x.expr1);
6     numberExpreNode(x.expr2);
7 }

```

Na seção 7.1.2 veremos a segunda parte do processo de exibição da árvore sintática, que são os métodos que mostram cada um dos nós.

7.1.2 Exibição dos nós

Para a exibição dos nós da árvore, aplica-se quase tudo que comentamos em relação a sua numeração. Para cada tipo de nó, temos um, ou em alguns casos, alguns métodos que tratam desse tipo de nó. A diferença reside nas ações que são tomadas. Nesse caso, queremos mostrar qual é o tipo de nó e quais são os seus filhos. Vamos ver, como exemplo, no programa 7.12, o método que trata de um *ClassBodyNode*.

Programa 7.12

```

1 public void printClassBodyNode(ClassBodyNode x)
2 {
3     if ( x == null ) return;
4     System.out.println();
5     System.out.print(x.number + ": ClassBodyNode ===> " +
6         (x.clist == null ? "null" : String.valueOf(x.clist.number))
7         + " " +
8         (x.vlist == null ? "null" : String.valueOf(x.vlist.number))
9         + " " +
10        (x.ctlist == null ? "null" : String.valueOf(x.ctlist.number))
11        + " " +
12        (x.mlist == null ? "null" : String.valueOf(x.mlist.number)) );
13
14     printClassDeclListNode(x.clist);
15     printVarDeclListNode(x.vlist);
16     printConstructDeclListNode(x.ctlist);

```

```

17     printMethodDeclListNode(x.mlist);
18 }

```

Esse método exibe na saída-padrão uma mensagem indicando qual é o tipo do nó, no caso, *classbodyNode*. Em seguida, mostra qual é o número de cada um dos seus quatro nós filhos, tomando cuidado com aqueles que podem ou não aparecer. Se olharmos a gramática no arquivo .jj, veremos que ao criar-se um nó do tipo *ClassBodyNode*, temos todos os seus filhos como opcionais. Por isso, para cada um deles, devemos, antes, verificar se é ou não *null*. Se for, então, escrevemos “null” na saída, caso contrário escrevemos o número do nó filho. Depois de escrever o tipo do nó e número dos filhos, o método chama os outros métodos, responsáveis por tratar dos nós filhos.

Assim, não temos muito mais a acrescentar à descrição dos métodos que exibem a árvore sintática. São válidos os comentários sobre os nós do tipo *ListNode*, *ExprNode* e *StatementNode* que fizemos na seção 7.1.1. Contudo, seria interessante ressaltar o tratamento dado àqueles nós que possuem filhos do tipo *Token*. Como esses nós filhos não possuem numeração, exibimos diretamente seu conteúdo, ou seja, o conteúdo da sua variável *image* que mostra qual foi o símbolo consumido na entrada e que originou o token. Por exemplo, para o nó *ClassDeclNode*, teríamos o método do programa 7.13.

Programa 7.13

```

1 public void printClassDeclNode(ClassDeclNode x)
2 {
3     if ( x == null ) return;
4     System.out.println();
5     System.out.print(x.number + ": ClassDeclNode ===> " +
6                       x.name.image + " " +
7                       (x.supernode == null ? "null": x.supernode.image) + " " +
8                       (x.body == null ? "null": String.valueOf(x.body.number)) );
9
10    printClassBodyNode(x.body);
11 }

```

Esse método exibe o tipo, o nó, e logo em seguida, exibe o valor do seu primeiro filho, que é um nó do tipo *Token*. Esse valor, dado por *x.name.image* na linha 6 corresponde ao nome da classe que foi declarada no programa-fonte. Isto também se aplica ao segundo filho, apontado pela variável *supernode* e que corresponde ao nome da superclasse da classe declarada e que pode ou não estar presente.

7.2 Outras operações

Este capítulo pode ser utilizado como uma introdução a diversas outras operação que podem ser feitas sobre a árvore sintática. Entre elas há a análise semântica e geração de código, que serão tratadas nos próximos capítulos. Mas, além destas, podemos citar um grande número de operações que podem ser realizadas e cuja implementação segue o mesmo modelo visto neste capítulo. Por exemplo:

- programa de “pretty-print” que produz como saída o programa-fonte formatado de acordo com algumas regras de indentação;
- construção do grafo de fluxo de controle do programa. Esse tipo de grafo é muito útil para a atividade de teste, principalmente quando a este se associa informação sobre a utilização das variáveis;
- cálculo da complexidade do programa, como número de linhas de código, complexidade ciclomática, complexidade de Halstead etc.

7.3 Arquivos-fonte do compilador

Para este capítulo, incluímos no pacote *syntacticTree* a classe *PrintTree*, que foi apresentada nas seções anteriores. Essa nova classe e aquelas pequenas alterações no arquivo *.jj* foram as únicas mudanças realizadas no nosso compilador. Vamos, como ilustração, mostrar um trecho da saída produzida com a exibição da árvore sintática para o programa *bintree*, apresentado no capítulo 2.

```
cap07$ java parser.langX -print_tree ../ssamples/bintree.x
X++ Compiler - Version 1.0 - 2004
Reading from file ../ssamples/bintree.x . .
0 Lexical Errors found
0 Syntactic Errors found
```

```
1: ListNode (ClassDeclNode) ===> 2 null
2: ClassDeclNode ===> bintree null 3
3: ClassBodyNode ===> 4 121 131 151
4: ListNode (ClassDeclNode) ===> 5 null
5: ClassDeclNode ===> data null 6
6: ClassBodyNode ===> null 7 15 59
7: ListNode (VarDeclNode) ===> 8 null
8: VarDeclNode ===> int 9
9: ListNode (VarNode) ===> 10 11
10: VarNode ===> dia
11: ListNode (VarNode) ===> 12 13
12: VarNode ===> mes
13: ListNode (VarNode) ===> 14 null
14: VarNode ===> ano
15: ListNode (ConstructDeclNode) ===> 16 31
16: ConstructDeclNode ===> 17
17: MethodBodyNode ===> null 18
18: BlockNode ===> 19
19: ListNode (StatementNode) ===> 20 23
20: AtribNode ===> 21 22
21: VarNode ===> dia
22: IntConstNode ===> 1900
23: ListNode (StatementNode) ===> 24 27
```

```
24: AtribNode ===> 25 26
25: VarNode ===> mes
26: IntConstNode ===> 1
27: ListNode (StatementNode) ===> 28 null
28: AtribNode ===> 29 30
29: VarNode ===> dia
30: IntConstNode ===> 1
31: ListNode (ConstructDeclNode) ===> 32 null
32: ConstructDeclNode ===> 33
33: MethodBodyNode ===> 34 46
34: ListNode (VarDeclNode) ===> 35 38
35: VarDeclNode ===> int 36
36: ListNode (VarNode) ===> 37 null
37: VarNode ===> d
38: ListNode (VarDeclNode) ===> 39 42
39: VarDeclNode ===> int 40
40: ListNode (VarNode) ===> 41 null
...
335: VarNode ===> t
336: NewObjectNode ===> bintree 337
337: ListNode (ExpreNode) ===> 338 null
338: VarNode ===> w
339: ListNode (StatementNode) ===> 340 382
340: ForNode ===> 341 344 347 352
341: AtribNode ===> 342 343
342: VarNode ===> i
343: IntConstNode ===> 0
344: RelationalNode ===> 345 < 346
345: VarNode ===> i
346: IntConstNode ===> 10
347: AtribNode ===> 348 349
348: VarNode ===> i
349: AddNode ===> 350 + 351
350: VarNode ===> i
351: IntConstNode ===> 1
352: BlockNode ===> 353
353: ListNode (StatementNode) ===> 354 356
354: ReadNode ===> 355
355: VarNode ===> d
356: ListNode (StatementNode) ===> 357 359
357: ReadNode ===> 358
358: VarNode ===> m
359: ListNode (StatementNode) ===> 360 362
360: ReadNode ===> 361
361: VarNode ===> a
362: ListNode (StatementNode) ===> 363 372
363: AtribNode ===> 364 365
```

```
364: VarNode ===> w
365: NewObjectNode ===> data 366
366: ListNode (ExpreNode) ===> 367 368
367: VarNode ===> d
368: ListNode (ExpreNode) ===> 369 370
369: VarNode ===> m
370: ListNode (ExpreNode) ===> 371 null
371: VarNode ===> a
372: ListNode (StatementNode) ===> 373 null
373: IfNode ===> 374 380 null
374: RelationalNode ===> 375 == 379
375: CallNode ===> 376 insert 377
376: VarNode ===> t
377: ListNode (ExpreNode) ===> 378 null
378: VarNode ===> w
379: IntConstNode ===> 0
380: PrintNode ===> 381
381: StringConstNode ===> "Elemento ja existe\n"
382: ListNode (StatementNode) ===> 383 389
383: AtribNode ===> 384 385
384: VarNode ===> i
385: CallNode ===> 386 treeprint 387
386: VarNode ===> t
387: ListNode (ExpreNode) ===> 388 null
388: IntConstNode ===> 0
389: ListNode (StatementNode) ===> 390 null
390: ReturnNode ===> 391
391: IntConstNode ===> 0
```

Capítulo 8

Tabela de Símbolos

Neste capítulo trataremos de uma das mais importantes estruturas para que se possam realizar as próximas etapas da compilação. Essa estrutura é chamada de tabela de símbolos, visto que reúne os nomes dos diversos elementos utilizados no programa-fonte, como classes, variáveis e métodos.

Inicialmente, mostraremos alguns exemplos de como a tabela de símbolos é útil na análise semântica e na geração de código e, em seguida, mostraremos como iremos implementar a tabela de símbolos para o compilador X++.

8.1 Para que serve

A tabela de símbolos serve para armazenar informação sobre cada nome, ou identificador, utilizado no programa-fonte. Assim, ao analisar uma declaração do tipo

```
class firstclass extends secondclass
{
    ...
}
```

o compilador deve ser capaz de reconhecer que `firstclass` está sendo declarada como uma classe. Assim, quando uma outra declaração como

```
firstclass y;
```

aparecer, ele poderá reconhecê-la como válida, pois `firstclass` é um tipo de dado conhecido.

Além disso, devemos também armazenar algumas informações importantes, relacionadas aos identificadores. Para cada tipo de identificador, as informações a serem armazenadas diferem. Por exemplo, temos para

- classes: nome, qual é a superclasse, quais são as variáveis dessa classe, quais são as classes aninhadas, quais são os métodos e construtores;
- variáveis: nome, tipo, dimensão, se são locais ou não;
- métodos: nome, parâmetros, variáveis locais, tipo de retorno.

Com isso, a tabela de símbolos auxilia a análise semântica e a geração de código de diversas maneiras. Por exemplo:

- ao declarar a classe

```
class firstclass extends secondclass ...
```

é necessário saber se `firstclass` ainda não foi declarada, se `secondclass` existe e se pode ou não ser declarada como superclasse;

- ao declarar a variável

```
firstclass y;
```

é necessário saber se `firstclass` é um tipo ou classe válido e se `y` pode ser declarada nesse ponto;

- ao declarar o método

```
firstclass m(int k, secondclass s)
```

é necessário saber se `firstclass` é um tipo ou classe válido, se `m` pode ser declarado nesse ponto e se seus parâmetros são legais;

- ao utilizar o comando

```
x[i+9] = y.m(10, b)
```

é preciso saber se `x` é um array, se o índice utilizado tem o tipo esperado, se a classe de `y` possui o método `m` com dois parâmetros, qual é o tipo de retorno do método `m` e se os argumentos utilizados na sua chamada combinam com os parâmetros declarados e se os dois lados da atribuição possuem o mesmo tipo ou tipos compatíveis.

Esses são apenas alguns dos inúmeros casos em que precisamos das informações da tabela de símbolos. Ela auxilia também a controlar o escopo de utilização dos identificadores. Por exemplo, no seguinte programa

```
class classA {  
}  
class classB {  
}  
int i;
```

```

string m(classA x)
{
    if ( x == null )
    {
        int j;
    }
}

class classC {
}

```

vemos que cada identificador tem validade em certos trechos do programa, e não em outros. Em X^{++} , a classe `classA` pode ser utilizada em qualquer ponto dentro de si própria, incluindo em `classB` e dentro de `classC`. Já a classe `classB` não pode ser utilizada dentro de `classC`. A variável `i` pode ser utilizada dentro de `classA` (excluindo `classB`), mas não dentro de `classC`. Já a variável `x` é local a `m` e `j` é local ao comando `if`. Assim, a tabela de símbolos deve permitir que se possa saber quando um determinado identificador é válido, ou seja, quando pode ser usado e quando não.

Na seção 8.2 mostraremos alguns aspectos da utilização de identificadores na linguagem X^{++} e como implementar uma tabela de símbolos para satisfazer esses requisitos.

8.2 A tabela de símbolos para X^{++}

Um programa em X^{++} é composto de diversas classes. Assim, vamos começar a definir nossa tabela de símbolos simplesmente como uma lista em que cada elemento é uma dessas classes. Para o programa a seguir, teríamos algo como a tabela mostrada na figura 8.1.

```

class firstclass {
}

class secondclass {
}

class thirdclass {
}

```

Vamos chamar essa estrutura de *Syntable*. Não importa, por enquanto, como é implementada, podendo ser uma lista ligada, um vetor estático, uma tabela hash ou qualquer outra estrutura. Importa apenas o que deve conter e como é construída. Para entendermos como é criada, é melhor observarmos a representação do programa mostrada na figura 8.2. Assim como no programa que exibe a árvore sintática, temos

0	firstclass
1	secondclass
2	thirdclass

Figura 8.1 – Exemplo de uma estrutura do tipo *Syntable*.

um programa que analisa a árvore e vai montando a tabela de símbolos. Esse programa, ao analisar o nó número 2, deve inserir na tabela o identificador *firstclass*, ao analisar o nó 6, deve incluir *secondclass* e, ao analisar o nó 10, deve inserir *thirdclass*.

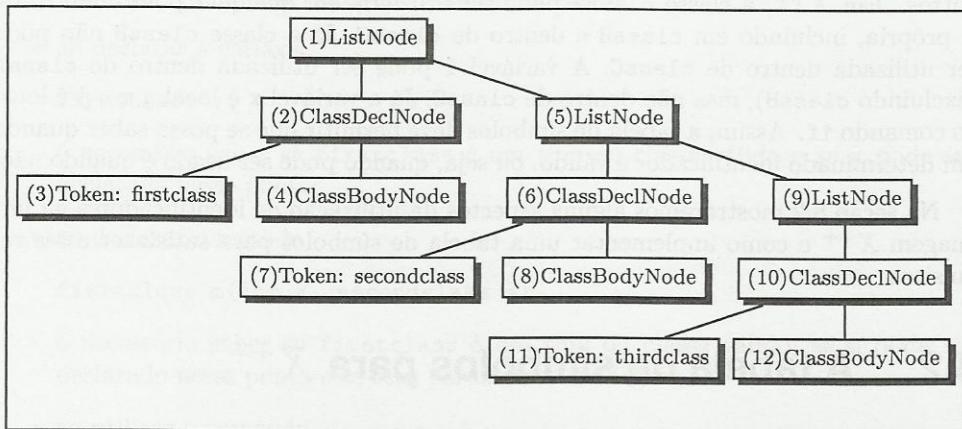


Figura 8.2 – Árvore sintática para construção da tabela de símbolos.

Cada entrada (ou descritor de classe) na *Syntable* deve possuir outras informações, como, por exemplo, o nome da superclasse. Deve possuir, também, informações sobre o conteúdo da classe, ou seja, suas variáveis, construtores, métodos e classes aninhadas. Para isso, associamos a cada descritor de classe uma outra estrutura do tipo *Syntable*. Vamos supor que *secondclass* fosse definida da seguinte maneira:

```

class secondclass {
    class sec2 {
        class sec3 {
        }
    }
    int secvar1, secvar2[];
    firstclass secvar3;
    thirdclass [] [] secmeth1();
}
  
```

Temos dentro de *secondclass* uma classe aninhada *sec2* e, aninhada a esta, uma outra classe *sec3*. Ainda em *secondclass* temos uma variável inteira *secvar1* e um vetor de inteiros *secvar2*, uma variável da classe *firstclass* e um método *secmeth1* que retorna uma matriz de objetos do tipo *thirdclass* e que não possui parâmetros formais. Para esse programa, teríamos, então, uma estrutura de tabela de símbolos como a mostrada na figura 8.3.

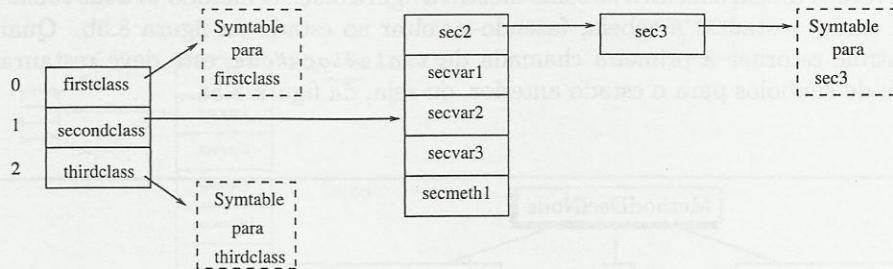


Figura 8.3 – Exemplo de uma tabela de símbolos.

Como foi dito, a tabela de símbolos deve permitir o controle do escopo de cada símbolo. Por exemplo, uma declaração de variável dentro da classe *sec2* pode utilizar as classes *sec2*, *sec3*, *secondclass*, *firstclass* ou *thirdclass*. Não pode, porém, utilizar uma classe declarada aninhada a *firstclass*, assim como *secondclass* não pode utilizar *sec3*, pois esta não está diretamente aninhada em *secondclass*.

O comportamento e escopo de classes, métodos, construtores e variáveis de classe são, de certa forma, estáticos. Uma vez declarados, esses identificadores mantêm as suas propriedades e podem ser utilizados de diversos pontos diferentes do programa. Já as variáveis locais dos métodos têm comportamento mais volátil, isto é, só podem ser usadas dentro do método em que foram definidas ou, ainda, dentro do bloco onde foram definidas. Para isso, ao iniciar a análise de um método ou um bloco, o analisador deve marcar quais as variáveis que foram definidas até aquele ponto. Ao terminar a análise desse bloco, ou seja, da subárvore ligada ao nó que inicia o bloco, como um *MethodDeclNode* ou um *BlockNode*, o analisador deve tirar da tabela aquelas variáveis declaradas dentro do bloco.

Vamos supor que o método *secmeth1* seja

```

thirdclass[][] secmeth1()
{
    int i,j;
    {
        string s;
    }
}
  
```

teríamos a árvore sintática mostrada na figura 8.4. Ao iniciar a análise de *secmeth1*, o analisador chama o método que trata do *MethodBodyNode*, que, por sua vez, chama

o método apropriado para o seu filho *BlockNode*, digamos *trataBlockNode*. Nesse ponto, o método *trataBlockNode* deve salvar o estado da tabela de símbolos, que é dado pela figura 8.5a. Prosseguindo a análise, chega-se ao segundo *BlockNode* com a tabela de símbolos contendo as variáveis locais *i* e *j*, como mostra a figura 8.5b. Novamente, o método *trataBlockNode* deve salvar o estado da tabela e continuar a análise da sua subárvore, que irá definir a variável *s*. Ao retornar da análise da subárvore, a tabela encontra-se como mostra a figura 8.5c. O método *trataBlockNode* deve, então, restaurar a tabela, fazendo-a voltar ao estado da figura 8.5b. Quando o controle retornar à primeira chamada de *trataBlockNode*, este deve restaurar a tabela de símbolos para o estado anterior, ou seja, da figura 8.5a.

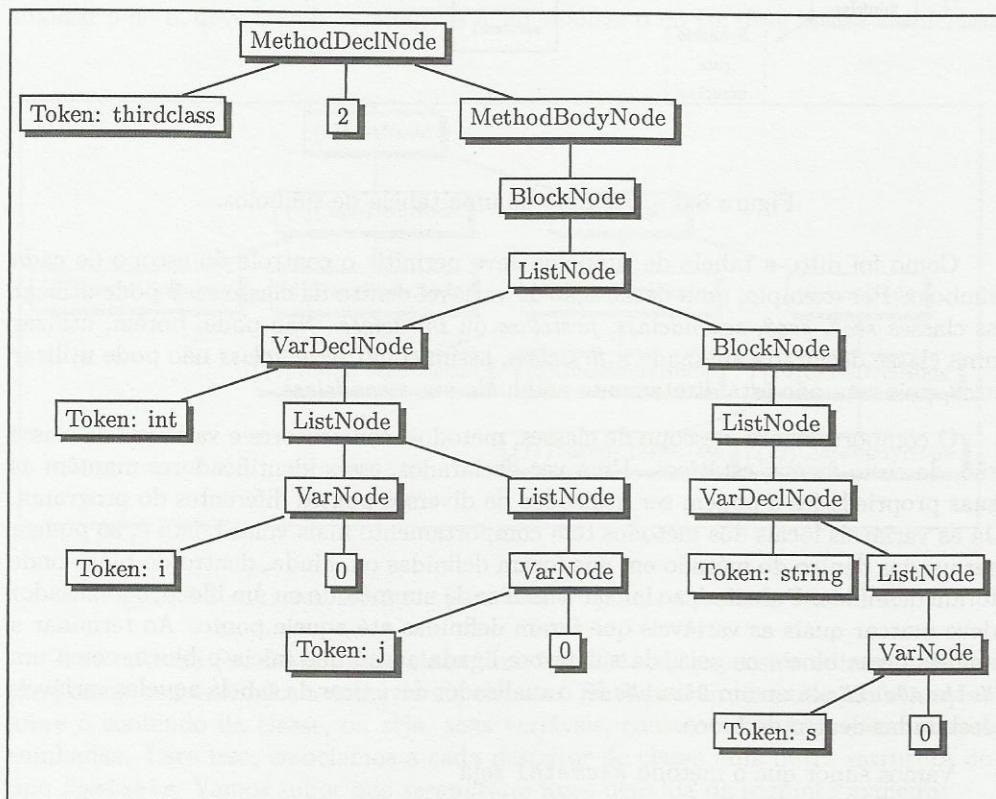


Figura 8.4 – Árvore sintática para *secmeth1*.

O leitor pode notar que cada entrada da *Symtable* pode representar tipos diferentes de identificadores, como um desritor de uma classe, de um construtor, de um método ou de uma variável. Para cada um desses desritores, o tipo de informação que se armazena é diferente. Na seção 8.3 mostraremos como é implementada a tabela de símbolos para o nosso compilador *X⁺⁺*, mostrando exatamente quais são as informações armazenadas para cada tipo desritor.

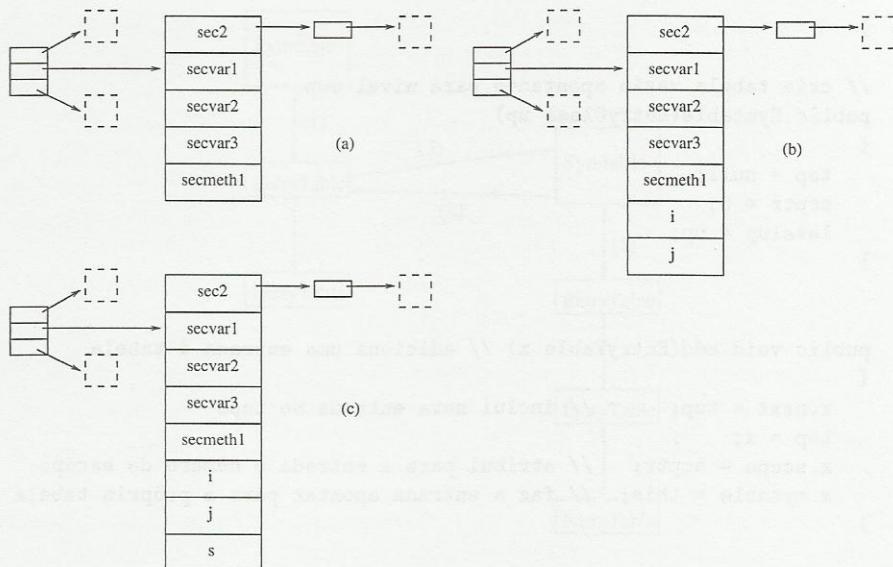


Figura 8.5 – Exemplo de tabela de símbolos com variáveis locais.

8.3 Implementação da tabela de símbolos

Nesta seção aprenderemos implementar a tabela de símbolos para o compilador X^{++} , começando pela *Symtable*. Escolhemos para ela uma estrutura bastante simples: uma lista ligada que funciona na forma de uma pilha, ou seja, a cabeça da lista aponta para o símbolo que foi mais recentemente inserido. Para isso, criamos dentro do compilador um pacote chamado *symtable* e, dentro dele, a classe *Symtable* mostrada no programa 8.1.

Programa 8.1

```

1 package symtable;
2
3 public class Symtable {
4     // apontador para o topo da tabela (mais recente)
5     public EntryTable top;
6
7     // número que controla o escopo (aninhamento) corrente
8     public int scptr;
9
10    // apontador para a entrada EntryClass de nível sup.
11    public EntryClass levelup;
12
13    public Symtable() // cria uma tabela vazia
14    {
15        top = null;
16        scptr = 0;
17        levelup = null;

```

```

18 }
19
20 // cria tabela vazia apontando para nível sup.
21 public Symtable(EntryClass up)
22 {
23     top = null;
24     scptr = 0;
25     levelup = up;
26 }
27
28
29 public void add(EntryTable x) // adiciona uma entrada à tabela
30 {
31     x.next = top;          // inclui nova entrada no topo
32     top = x;
33     x.scope = scptr;      // atribui para a entrada o número do escopo
34     x.mytable = this;    // faz a entrada apontar para a própria tabela
35 }
36
37
38 public void beginScope()
39 {
40     scptr++;             // inicia novo aninhamento de variáveis
41 }
42
43
44 public void endScope()
45 {
46     while (top != null && top.scope == scptr)
47         top = top.next;   // retira todas as vars do aninhamento corrente
48     scptr--;            // finaliza aninhamento corrente
49 }
50
51 }

```

Essa classe possui três variáveis. A primeira é *top*, que aponta para a cabeça da lista. A segunda, *scptr*, é um número inteiro que controla o aninhamento de blocos. Inicialmente seu valor é 0 e a cada novo bloco esse valor é incrementado. Cada variável guarda o valor de *scptr* quando inserida na tabela, assim é possível saber a qual bloco ela pertence. A terceira variável é *levelup*, que aponta para uma entrada em outra *Symtable*. Ela possibilita identificar a qual classe essa tabela pertence. A figura 8.6 mostra esquematicamente uma estrutura *Symtable*. A seta numerada (1) representa a variável *top* e a número (2), a variável *levelup*.

O primeiro construtor da classe *Symtable* simplesmente cria uma tabela vazia. O segundo também cria uma tabela vazia, porém recebe como parâmetro um objeto do tipo *EntryClass* que será descrito adiante. Esse objeto descreve a classe à que a *Symtable* pertence e seu valor é armazenado na variável *levelup*. Na figura 8.6, ela é a origem da seta número (3).

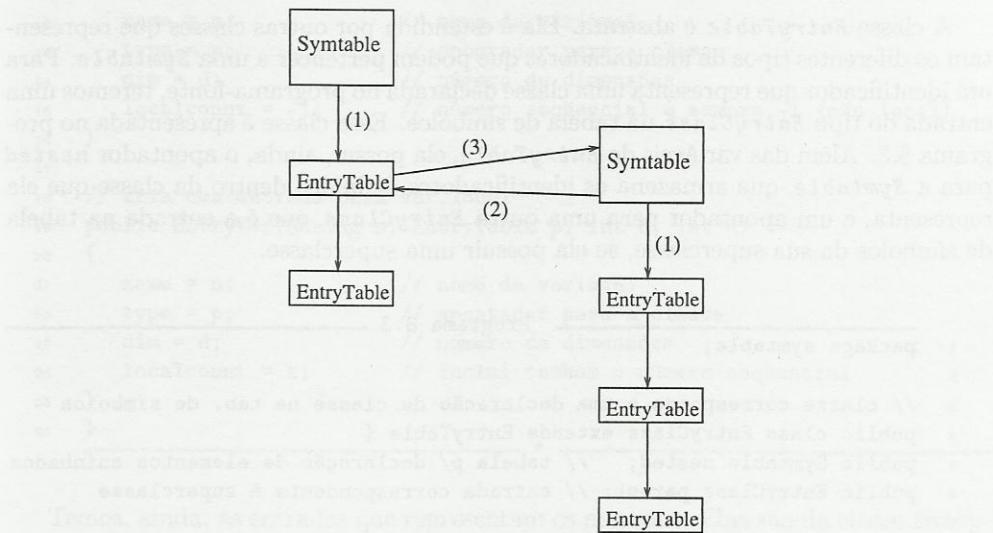


Figura 8.6 – Representação da implementação de *Symtable*.

O método `add` adiciona uma nova entrada na tabela, colocando-a no topo da pilha. Toda entrada na tabela é do tipo abstrato `EntryTable`, cuja definição é mostrada no programa 8.2.

Programa 8.2

```
1 package symtable;
2
3 // classe geral para as possíveis entradas na tabela de símbolo
4 abstract public class EntryTable {
5     public String name;          // nome do símbolo (var., método ou classe)
6     public EntryTable next;      // apontador para próximo dentro da tabela
7     public int scope;           // número do aninhamento corrente
8     public Symtable mytable;    // aponta para a tabela da qual ela é parte
9
10 }
```

Todo objeto desse tipo possui um nome que o identifica, armazenado na variável *name* e o apontador para o próximo elemento na tabela. Possui, ainda, *scope*, uma variável inteira que guarda o número do escopo ao qual ela pertence. O método *add* da classe *Symtable* atualiza o valor dessa variável quando a entrada é inserida na tabela. O número do escopo corrente da tabela é atualizado a cada vez que um novo método ou bloco é identificado na árvore sintática. Para isso, o programa que analisa a árvore deve chamar os métodos *beginScope* (quando um novo escopo é iniciado) e *endScope* (quando um escopo termina).

Uma entrada na tabela possui também a variável *mytable*, que indica a que tabela a entrada pertence. Essa informação é importante para os métodos de busca, conforme veremos posteriormente.

A classe *EntryTable* é abstrata. Ela é estendida por outras classes que representam os diferentes tipos de identificadores que podem pertencer a uma *Syntable*. Para um identificador que representa uma classe declarada no programa-fonte, teremos uma entrada do tipo *EntryClass* na tabela de símbolos. Essa classe é apresentada no programa 8.3. Além das variáveis de *EntryTable*, ela possui, ainda, o apontador *nested* para a *Syntable*, que armazena os identificadores definidos dentro da classe que ela representa, e um apontador para uma outra *EntryClass*, que é a entrada na tabela de símbolos da sua superclasse, se ela possuir uma superclasse.

Programa 8.3

```

1 package symtable;
2
3 // classe corresponde a uma declaração de classe na tab. de símbolos
4 public class EntryClass extends EntryTable {
5     public Syntable nested; // tabela p/ declaração de elementos aninhados
6     public EntryClass parent; // entrada correspondente à superclasse
7
8     public EntryClass(String n, Syntable t)
9     {
10         name = n; // nome da classe declarada
11         nested = new Syntable(this); // tabela onde inserir variáveis,
12                                     // métodos ou classes
13         parent = null; // sua superclasse
14     }
15 }
```

*

O segundo tipo de entrada que pode aparecer numa *Syntable* é a que representa uma variável. *EntryVar* possui um apontador *type* para uma outra entrada da tabela que indica qual o tipo da variável. Possui uma variável inteira *dim* que indica a dimensão com que a variável foi declarada. Um vetor unidimensional tem dimensão 1, uma matriz bidimensional possui valor 2, uma variável sem dimensão possui valor 0, e assim por diante. Uma *EntryVar* possui, ainda, uma variável inteira *localcount*, que diz qual o número sequencial dessa variável, caso se trate de variável local. Essa informação é utilizada na geração de código e será discutida com mais detalhes no capítulo 12. Essa classe é apresentada no programa 8.4. Ela possui dois construtores, um utilizado quando se trata de uma declaração de variáveis de classe e outro, para a declaração de variáveis locais.

Programa 8.4

```

1 package symtable;
2
3 // classe que abriga uma declaração de variável na tabela de símbolos
4 public class EntryVar extends EntryTable {
5     public EntryTable type; // apontador para o tipo da variável
6     public int dim; // número de dimensões da variável
7     public int localcount; // numeração sequencial para as vars. locais
8
9     // cria uma entrada para var. de classe
10    public EntryVar(String n, EntryTable p, int d)
11    {
```

```

12     name = n;           // nome da variável
13     type = p;          // apontador para a classe
14     dim = d;           // número de dimensões
15     localcount = -1;   // número seqüencial é sempre -1 (não local)
16 }
17
18 // cria uma entrada para var.local
19 public EntryVar(String n, EntryTable p, int d, int k)
20 {
21     name = n;           // nome da variável
22     type = p;          // apontador para a classe
23     dim = d;           // número de dimensões
24     localcount = k;    // inclui também o número seqüencial
25 }
26 }

```

Temos, ainda, as entradas que representam os métodos. Elas são da classe *EntryMethod* (Programa 8.5). Cada uma dessas entradas possui um apontador *type* para uma outra entrada na tabela de símbolos, que indica qual o tipo de retorno e um inteiro *dim*, que indica a sua dimensão. Dois outros inteiros *totallocals* e *totalstack* indicam, respectivamente, o número total de variáveis locais definidas no método e a quantidade de elementos utilizados na pilha da Máquina Virtual Java durante a execução do método. Esses dois valores são usados na geração de código. São inicializadas, no construtor da classe, com o valor 0. Esse valor é alterado para refletir as características do método durante a análise semântica.

Também utilizadas na análise semântica e na geração de código são as variáveis *fake* e *hassuper*. A primeira tem valor *false* para todos os métodos e construtores declarados no programa-fonte. Mas existe um caso em que o próprio compilador faz a declaração de um construtor. Nesse caso, o valor é *true*. A segunda indica se o corpo de um construtor possui uma chamada ao construtor da superclasse. Veremos esses casos durante a análise semântica e geração de código.

A variável *param* descreve quais são e como são os parâmetros formais do método. Deve indicar o tipo e a dimensão de cada um deles e, por isso, utiliza-se um objeto do tipo *EntryRec*, que é uma lista em que cada elemento representa um dos parâmetros.

Programa 8.5

```

1 package symtable;
2
3 // corresponde a uma declaração de método na tabela de símbolos
4
5 public class EntryMethod extends EntryTable {
6     public EntryTable type;           // tipo de retorno do método
7     public int      dim;            // número de dimensões do retorno
8     public EntryRec param;          // tipo dos parâmetros
9     public int      totallocals;    // número de variáveis locais
10    public int      totalstack;     // tamanho da pilha necessária
11    public boolean fake;           // true, se é um falso construtor
12    public boolean hassuper;        // true, se método possui chamada super
13

```

```

14 // cria elemento para inserir na tabela
15 public EntryMethod(String n, EntryTable p, int d, EntryRec r)
16 {
17     name = n;
18     type = p;
19     dim = d;
20     param = r;
21     totallocals = 0;
22     totalstack = 0;
23     fake = false;
24     hassuper = false;
25 }
26
27 public EntryMethod(String n, EntryTable p, boolean b)
28 {
29     name = n;
30     type = p;
31     dim = 0;
32     param = null;
33     totallocals = 0;
34     totalstack = 0;
35     fake = b;
36     hassuper = false;
37 }
38 }

```

A classe *EntryRec* (Programa 8.6) é usada para representar uma lista de tipos e dimensões, por exemplo, para descrever os parâmetros de um método. Cada objeto dessa classe possui uma variável *type*, que aponta uma entrada na tabela de símbolos, e a variável *dim*, que indica uma dimensão. A variável inteira *cont* indica quantos elementos existem na lista de tipos e a variável *next* aponta o próximo elemento da lista.

 Programa 8.6

```

1 package symtable;
2
3
4 // lista de EntryClass usada para representar os tipos de uma lista
5 // de parâmetros
6
7 public class EntryRec extends EntryTable {
8     public EntryTable type;          // tipo de um objeto
9     public int dim;                // dimensão
10    public EntryRec next;          // apontador para o resto da lista
11    public int cont;               // número de elementos a partir daquele elemento
12
13
14
15 // cria elemento
16 public EntryRec(EntryTable p, int d, int c)
17 {

```

```

18     type = p;
19     cont = c;
20     dim = d;
21     next = null;
22 }
23
24 // cria elemento e põe no início da lista
25 public EntryRec(EntryTable p, int d, int c, EntryRec t)
26 {
27     type = p;
28     cont = c;
29     dim = d;
30     next = t;
31 }
32 }

```

Assim, vamos supor que temos um método que possui três parâmetros: o primeiro do tipo *a*, o segundo do tipo *b* com dimensão 3 e o terceiro também do tipo *b*, mas com dimensão 1. Para criarmos um *EntryRec* que representa os tipos desses parâmetros, faríamos

```

EntryTable x = localiza a entrada do tipo a na tabela;
EntryTable y = localiza a entrada do tipo b na tabela;
EntryRec l = new EntryRec(y, 1, 1); // elemento do tipo b[]
l = new EntryRec(y, 3, 2, 1);      // elemento do tipo b[][][]
l = new EntryRec(x, 0, 3, 1);      // elemento do tipo a

```

Supondo que as variáveis *x* e *y* apontam as entradas na tabela para as classes *a* e *b*, respectivamente, esse exemplo utiliza o primeiro construtor da classe para criar um objeto do tipo *EntryRec*, para o último parâmetro do tipo *b* com dimensão 1. Depois, utilizando o segundo construtor, é criado um *EntryRec* para o segundo parâmetro, com o tipo *b* e dimensão 3, e que aponta para aquele elemento que representa o terceiro parâmetro. E, por fim, é criado um *EntryRec* para o primeiro parâmetro, com o tipo *a* e dimensão 0, e que aponta para aquele elemento que representa o segundo parâmetro.

Mais uma classe é utilizada como entrada numa *Syntable*. Ela descreve um tipo básico da linguagem, no nosso caso, os tipos *int* e *string*. Dessa maneira, podemos tratar as declaração que utilizam esses tipos da mesma forma que tratamos as declarações que utilizam classes definidas pelo programador. Por exemplo, num nó do tipo *VarDeclNode*, temos no filho *position* um *Token* que indica o tipo da variável declarada. Ao escrevermos um método que analisa esse nó, não é preciso verificar se esse filho contém um *int* ou *string* e tratá-los de forma especial. Para isso, devemos inserir na tabela de símbolos uma entrada para cada um desses tipos, como se fossem uma classe, e, assim, ao analisarmos o nó *VarDeclNode*, podemos simplesmente efetuar uma busca na tabela à procura do tipo representado no seu filho *position*, e o resultado da busca nos mostrará qual é o tipo da variável, seja ele uma classe ou um tipo básico.

A classe *EntrySimple* mostrada no programa 8.7 é utilizada para esse fim. Para esse tipo de entrada, nada é incluído no descritor, além do nome do identificador.

Programa 8.7

```

1 package symtable;
2
3 // entrada utilizada para declarar os tipos básicos da linguagem
4
5 public class EntrySimple extends EntryTable {
6
7
8     public EntrySimple(String n)
9     {
10         name = n;
11     }
12
13 }
```

Nos capítulos seguintes, veremos como essa estrutura de tabela de símbolos é utilizada para apoiar a análise semântica e a geração de código. As classes apresentadas neste capítulo – em particular, a classe *Symtable* – possuem diversos métodos que não foram mostrados aqui, como os métodos que fazem a busca de identificadores na tabela. Eles serão apresentados e comentados à medida que forem sendo utilizados nos programas que realizam a análise semântica ou a geração de código.

uma das aplicações mais comuns é sobre validação de expressões matemáticas e de programação. Um exemplo comum é a verificação se uma expressão é válida ou não levando em conta os operadores e tipos de variáveis que devem ser utilizadas em expressões matemáticas.

Capítulo 9

Análise Semântica – Primeira Parte

A análise semântica é responsável por verificar aspectos do programa relacionados ao “significado” de cada comando. Um programa pode estar correto de acordo com as regras de sintaxe definidas pela gramática, mas apresentar problemas relacionados com a semântica da linguagem. É o caso, por exemplo, do seguinte trecho de programa:

```
int a, b[];
String c;

a = b[c];
```

De acordo com as regras sintáticas da gramática de X^{++} , o trecho é perfeitamente aceitável. Porém, existe um erro semântico, pois, pelo menos em X^{++} , o valor usado para indexar um array deve ser um número inteiro. Não pode ser uma expressão cujo resultado é do tipo *string*.

A análise semântica é feita de maneira semelhante à visualização da árvore sintática mostrada no capítulo 7, ou seja, mediante um passeio pela árvore sintática. A análise semântica deve ser dividida em fases. Neste capítulo, inicialmente, será discutido o porquê dessa divisão e, em seguida, mostraremos a primeira dessas fases, em que as classes do programa-fonte são analisadas.

9.1 Análise semântica em fases

A análise semântica é realizada em diversas fases, cada uma delas objetivando a análise de aspectos específicos da semântica da linguagem. Um ponto a ser destacado é que cada uma dessas fases é feita por meio de um passeio pela árvore sintática. Isto significa que a cada fase a árvore sintática é percorrida e dados são levantados para serem utilizados na fase seguinte.

Seria mais interessante se pudéssemos realizar toda a análise semântica em uma única “passada” pela árvore sintática, uma vez que cada fase representa trabalho extra e acréscimo no tempo de compilação. Isto, porém, não é possível ou é, no mínimo, demasiado complicado. Para entendermos o porquê da necessidade dessa divisão vamos utilizar o exemplo do seguinte programa:

```
class A {
    B varB;
}

class B {
```

A árvore sintática correspondente é mostrada na figura 9.1. Uma tentativa de analisar este programa seria por meio de uma visita aos nós na seguinte ordem:

1. visita ao nó 1;
2. visita ao nó 2, inserindo a classe *A* na tabela de símbolos;
3. visita ao nó 3;
4. visita ao nó 4;
5. visita ao nó 5 e verifica se declaração de variável é válida;
6. visita ao nó 8;
7. visita ao nó 9 e insere classe *B* na tabela de símbolos;
8. visita ao nó 10.

O problema é que no item 5, verificar se a declaração da variável é legal, consiste em verificar – entre outras coisas – se o tipo da variável existe. Porém, note-se que a visita ao nó 5 é feita antes que o nó 9 tenha sido visitado e a classe *B* inserida na tabela de símbolos.

Para solucionar esse problema, efetuaremos a análise semântica em duas fases distintas, cada uma delas com um objetivo diferente. A primeira para colher informações sobre as classes declaradas e a segunda para realizar a verificação semântica propriamente dita. Então, a ordem do passeio seria:

Fase 1:

1. visita ao nó 1;
2. visita ao nó 2, inserindo a classe *A* na tabela de símbolos;
3. visita ao nó 3;

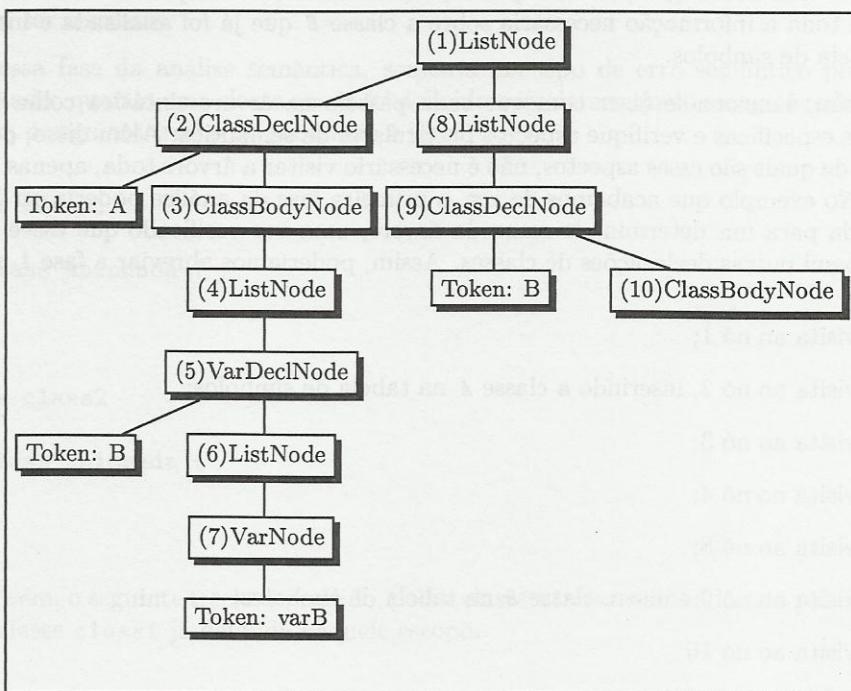


Figura 9.1 – Exemplo de árvore sintática.

4. visita ao nó 4;
5. visita ao nó 5;
6. visita ao nó 8;
7. visita ao nó 9 e insere classe *B* na tabela de símbolos;
8. visita ao nó 10.

Fase 2:

1. visita ao nó 1;
2. visita ao nó 2;
3. visita ao nó 3;
4. visita ao nó 4;
5. visita ao nó 5 e verifica se declaração de variável é válida;
6. visita ao nó 8;
7. visita ao nó 9;
8. visita ao nó 10.

Com essa abordagem, ao verificar-se a validade da declaração da variável *varB*, tem-se toda a informação necessária sobre a classe *B* que já foi analisada e inserida na tabela de símbolos.

Assim, tenciona-se fazer com que cada passeio na árvore sintática colha informações específicas e verifique aspectos particulares da semântica. Além disso, dependendo de quais são esses aspectos, não é necessário visitar a árvore toda, apenas parte dela. No exemplo que acabamos de ver, a primeira fase da análise poderia ser interrompida para um determinado ramo da árvore, uma vez verificado que esse ramo não possui outras declarações de classes. Assim, poderíamos abreviar a fase 1 para:

1. visita ao nó 1;
2. visita ao nó 2, inserindo a classe *A* na tabela de símbolos;
3. visita ao nó 3;
4. visita ao nó 4;
5. visita ao nó 8;
6. visita ao nó 9 e insere classe *B* na tabela de símbolos;
7. visita ao nó 10.

Ou seja, uma vez que o nó 4 não possui nenhum filho no qual possam existir declarações de classes aninhadas, não há necessidade de continuar a análise semântica abaixo desse nó.

Nas próximas seções será descrita a primeira fase da análise semântica para o compilador *X⁺⁺*, indicando-se quais são as informações coletadas e a sua implementação.

9.2 Análise da declaração de classes

Esta primeira fase da análise semântica é bastante simples. Seu objetivo é criar a tabela de símbolos e analisar a árvore sintática à procura das declarações de classes. Cada classe encontrada é inserida na tabela de símbolos.

A árvore sintática não precisa ser completamente percorrida, somente os seus níveis mais altos. O analisador semântico inicia a análise na raiz da árvore sintática, que sempre é um *ListNode*. Esse nó pode conter um filho *ClassDeclNode* que determina a declaração de uma classe. Ao encontrar tal nó, o analisador semântico deve incluir a classe correspondente na tabela de símbolos (no nível mais alto), o mesmo acontecendo com os demais nós *ClassDeclNode* que representam as classes de mais alto nível.

A análise semântica, porém, não deve parar por aí. Um nó *ClassDeclNode* possui como filho um nó *ClassBodyNode*, que, por sua vez, pode ter um outro *ClassDeclNode* correspondente a uma classe aninhada. Assim, caso esse nó exista, o analisador semântico deve prosseguir a análise para os níveis inferiores, até que não encontre uma classe aninhada. Deve também ter o cuidado de inserir as classes aninhadas na estrutura *Symtable* adequada, e não na de nível mais alto. Como visto no capítulo 8, cada

classe na tabela de símbolos possui sua própria *Syntable*, onde as classes aninhadas devem ser inseridas.

Nessa fase da análise semântica, somente um tipo de erro semântico pode ser detectado: quando uma classe que já foi definida num determinado escopo é redefinida. Assim, o seguinte trecho de programa é legal

```
class class1
{
    class aninhada {
    }
}

class class2
{
    class aninhada {
    }
}
```

Porém, o seguinte trecho é inválido, pois a classe *class1* aninhada está redefinindo uma classe *class1* já definida naquele escopo.

```
class class1
{
    class class1 {
    }
}
```

9.3 Implementação

A análise semântica neste nível é implementada pela classe *ClassCheck* no pacote *semanalysis*. Antes de iniciar a visita pelos nós da árvore, o compilador *X⁺⁺* deve criar um objeto dessa classe e, então, utilizá-lo na análise, como foi feito com a exibição da árvore sintática no capítulo 7.

No programa 9.1, vemos as declarações e o construtor da classe. O construtor é o responsável pela criação da tabela de símbolos criando um objeto de tipo *Syntable*. Além disso, ele inicializa o contador de erros semânticos e insere, na tabela de símbolos, dois símbolos que representam os tipos básicos da linguagem, ou seja, os tipos *int* e *string*.

Programa 9.1

```
1 package semanalysis;
2
3 import symtable.*;
4 import syntacticTree.*;
5
6 public class ClassCheck {
```

```

7  Symtable Mantable;           // tabela de mais alto nível
8  protected Symtable Curtable; // apontador para a tabela corrente
9  int foundSemanticError;
10
11 public ClassCheck()
12 {
13     EntrySimple k;
14
15     foundSemanticError = 0;
16     Mantable = new Symtable(); // cria tabela principal
17     k = new EntrySimple("int"); // insere tipos básicos da linguagem
18     Mantable.add(k);
19     k = new EntrySimple("string");
20     Mantable.add(k);
21 }

```

O compilador utiliza o método *ClassCheckRoot*, mostrado no programa 9.2, que inicia a visita chamando *ClassCheckClassDeclListNode* para a raiz da árvore sintática e verifica se algum erro semântico foi encontrado. Além deste, a classe possui um método para cada tipo de nó a ser analisado. O programa 9.2 mostra também o método que efetua o tratamento propriamente dito da raiz da árvore, que é um *ListNode* onde cada elemento da lista representa uma classe.

Programa 9.2

```

1  public void ClassCheckRoot(ListNode x) throws SemanticException
2  {
3      Curtable = Mantable;           // tabela corrente = principal
4      ClassCheckClassDeclListNode(x); // chama análise para raiz da árvore
5      if (foundSemanticError != 0) // se houve erro, lança exceção
6          throw new SemanticException(foundSemanticError +
7                  " Semantic Errors found (phase 1)");
8  }
9
10 public void ClassCheckClassDeclListNode(ListNode x)
11 {
12     if (x == null) return;
13     try {
14         ClassCheckClassDeclNode( (ClassDeclNode) x.node);
15     }
16     catch (SemanticException e)
17     { // se um erro ocorreu na análise da classe,
18         // dá a mensagem, mas faz a análise para próxima classe
19         System.out.println(e.getMessage());
20         foundSemanticError++;
21     }
22     ClassCheckClassDeclListNode(x.next);
23 }

```

O que *ClassCheckClassDeclListNode* faz é chamar o método que trata da classe e verificar se um erro foi encontrado. Caso tenha sido encontrado, o método emite uma

mensagem, incrementa o contador de erros e continua com a análise para as demais classes, chamando-se recursivamente, passando como argumento o resto (cauda) do *ListNode*.

O método *ClassCheckClassDeclNode* trata da declaração de uma classe. Sua função é verificar se no escopo corrente já existe uma classe com esse nome e, então, tomar a atitude devida: inseri-la na tabela de símbolos, em caso negativo, ou lançar uma exceção, em caso positivo. Este método é mostrado no programa 9.3.

A primeira ação nesse método é salvar o objeto *Symtable* corrente. A tabela corrente é responsável por armazenar os símbolos da classe corrente. Inicialmente, no nível mais alto, a tabela corrente é aquela criada no construtor dessa classe, onde são inseridos os tipos básicos e as classes de nível mais externo. Quando a declaração de uma classe é encontrada na árvore sintática, uma nova entrada do tipo *EntryClass* é inserida na tabela corrente. Lembre-se de que uma das variáveis dessa entrada é um outro objeto *Symtable* para armazenar os símbolos da recém-declarada classe. Além disso, como a análise semântica prossegue nos nós “dentro” da declaração da classe, o novo objeto *Symtable* deve passar a ser a tabela corrente, de modo que novos símbolos sejam inseridos nesta nova tabela que corresponde à classe sendo analisada. Ao final do método, o valor antigo da variável *Curtable* é recuperado, de forma que novas declarações serão efetuadas na tabela de nível superior.

Tomemos como exemplo a árvore sintática da figura 9.1. A ordem de visita dos nós seria a seguinte (os números indicam os nós sendo visitados):

- *ClassCheckRoot*(1)
- *ClassCheckClassDeclListNode*(1)
- *ClassCheckClassDeclNode*(2)

Nesse ponto, a tabela de símbolos apresenta a forma mostrada na figura 9.2a. Então, a entrada da classe *A* é inserida na tabela de símbolos, passando a ser esta a classe corrente, devendo sua tabela de símbolos ser aquela usada para inserir novos símbolos. É a situação mostrada na figura 9.2b que permite que a análise semântica possa prosseguir com a chamada a *ClassCheckClassBodyNode* de maneira correta. Ao retornar dessa chamada, a situação da tabela de símbolos é aquela mostrada na figura 9.1c. Dessa forma, ao continuar a análise semântica, por exemplo, com a visita ao nó 8, tem-se novamente a tabela de mais alto nível como corrente e a inserção da classe *B* no posto justo.

Programa 9.3

```

1  public void ClassCheckClassDeclNode(ClassDeclNode x)
2          throws SemanticException
3  {
4      Symtable temphold = Curtable; // salva apontador p/ tabela corrente
5      EntryClass nc;
6
7      if (x == null) return;
8
9      // procura classe na tabela

```

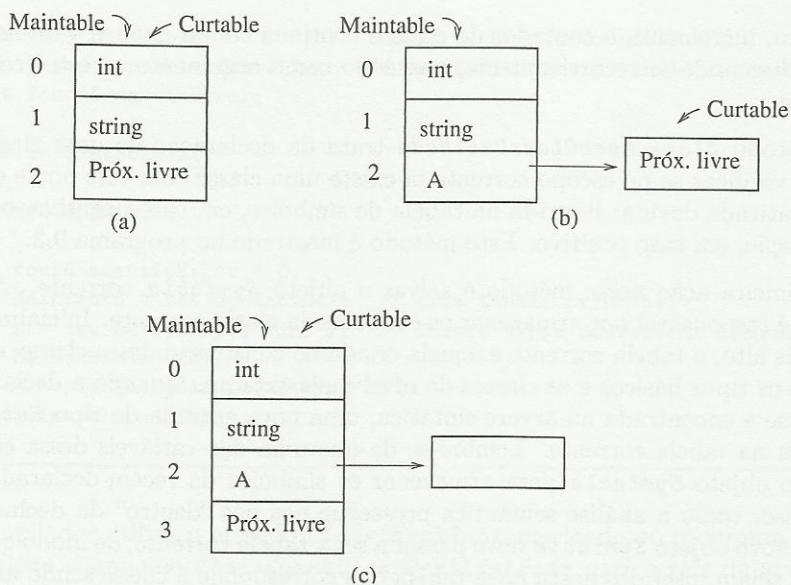


Figura 9.2 – Inserção de classe na tabela de símbolos.

```

10     nc = (EntryClass) Curtable.classFindUp(x.name.image);
11
12     if (nc != null) // já declarada, ERRO
13     {
14         throw new SemanticException(x.name,
15             "Class "+ x.name.image + " already declared");
16     }
17
18     // inclui classe na tabela corrente
19     Curtable.add(nc = new EntryClass(x.name.image, Curtable));
20     Curtable = nc.nested;    // tabela corrente = tabela da classe
21     ClassCheckClassBodyNode(x.body);
22     Curtable = tempHold;    // recupera apontador p/ tabela corrente
23 }
```

Em seguida, na linha 10, verifica-se se no escopo corrente já existe uma classe com o mesmo nome da classe sendo declarada. Essa procura é feita por meio do método *Symtable.classFindUp* mostrado no programa 9.4. Esse método deve respeitar as regras de escopo da linguagem. A busca funciona da seguinte forma:

1. inicialmente, o método procura a classe na tabela corrente (ou seja, verifica se na classe corrente existe uma classe com esse nome);
2. caso não tenha encontrado, chama recursivamente a pesquisa utilizando a tabela de nível superior, ou seja, aquela referente à classe um nível mais externo (caso ela exista).

Durante a busca na tabela de símbolos, somente entradas do tipo *EntryClass* e *EntrySimple* (classe e tipo simples, respectivamente) são consideradas. A implementação desse método, na realidade, define as regras de escopo para utilização das classes. Em um determinado ponto do programa, são acessíveis todas as classes definidas na classe corrente e todas definidas em classes de nível mais externo.

Programa 9.4

```

1  public EntryTable classFindUp(String x)
2  {
3      EntryTable p = top;
4
5      // para cada elemento da tabela corrente
6      while (p != null)
7      {
8          // verifica se é uma entrada de classe ou tipo simples
9          // e então compara o nome
10         if ((p instanceof EntryClass) || (p instanceof EntrySimple))
11             && p.name.equals(x))
12             return p;
13         p = p.next; // próxima entrada
14     }
15     if (levelup == null) // se não achou e é o nível mais externo
16         return null; // retorna null
17
18     // procura no nível mais externo
19     return levelup.mytable.classFindUp(x);
20 }
```

*

Voltando ao programa 9.2, vemos que se a classe é encontrada, então uma exceção é lançada. Caso contrário, cria-se uma nova entrada *EntryClass* para a classe e essa entrada é inserida na tabela corrente. Note que o construtor de *EntryClass* já cuida da criação de um novo objeto *Syntable*. Na linha 20, esta nova tabela torna-se a tabela corrente e a análise continua no nó *ClassBodyNode*, filho do nó corrente, pois é nele que podem aparecer outras declarações de classes.

No método que analisa um *ClassBodyNode* (o corpo da classe), existe somente uma chamada para o filho que contém a lista de classes aninhadas, conforme mostrado no programa 9.5.

Programa 9.5

```

1  public void ClassCheckClassBodyNode(ClassBodyNode x)
2  {
3      if (x == null) return;
4      ClassCheckClassDeclListNode(x.clist);
5 }
```

*

E esses são os únicos nós que precisam ser tratados nesta primeira fase da análise semântica. São eles que possuem ou estão em um caminho que leve a uma declaração de classe. Ao final dessa fase, a estrutura de tabela de símbolos, em termos de classes declaradas no programa, estará completamente montada.

9.4 O programa principal

Por fim, é preciso alterar o método principal do nosso compilador para que ele execute este nível da análise semântica. Isso é feito, como de costume, alterando o arquivo `.jj`. O programa 9.6 mostra o trecho do método `main` onde é criado o objeto `ClassCheck` (linha 10) e iniciada a primeira fase da análise semântica (linha 12).

```

1  // verifica se pode operar sobre a árvore sintática
2  if (parser.token_source.foundLexError()
3      + parser contParseError == 0)
4  {
5      if (print_tree) // exibir a árvore
6      {
7          PrintTree prt = new PrintTree();
8          prt.printRoot(root);      // chama método para imprimir árvore
9      }
10     ClassCheck tc = new ClassCheck();
11     try {
12         tc.ClassCheckRoot(root);
13         System.out.println("0 Semantic error found");
14     }
15     catch (SemanticException e)
16     {
17         System.out.println(e.getMessage());
18     }
19 }
```

9.5 Arquivos-fonte do compilador

Para este capítulo foram incluídos no diretório `cap09` os pacotes `symtable` e `semanalysis`. Além disso, o arquivo `.jj` foi alterado, como descrito na seção 9.4.

Colocamos, também, no diretório `ssamples` uma variação muito simples do programa `bintree`, no qual são declaradas algumas classes aninhadas que redefinem as classes `bintree` e `data`, gerando, assim, um erro semântico. O resultado é o seguinte:

```

cap09$ java parser.langX ..//ssamples/bintree-erro-semantico-fase1.x
X++ Compiler - Version 1.0 - 2004
Reading from file ..//ssamples/bintree-erro-semantico-fase1.x . .
0 Lexical Errors found
0 Syntactic Errors found
Line 12 column 19: Class bintree already declared
Line 17 column 23: Class data already declared
2 Semantic Errors found (phase 1)
```

Capítulo 10

Análise Semântica – Segunda Parte

Neste capítulo, apresentaremos a segunda parte, ou segunda fase, da análise semântica. Nela, serão analisados alguns aspectos referentes à herança entre as classes declaradas e também as declarações das variáveis, dos métodos e construtores das classes.

10.1 Análise da hierarquia de classes

No capítulo 9 vimos como a árvore sintática é utilizada para montar a estrutura básica da tabela de símbolos e verificar a presença de erros, como a redeclaração de classes em escopos não permitidos. Um aspecto da declaração de classes, contudo, não foi verificado: a coerência na utilização da cláusula *extends*.

Quando temos uma declaração de classe como

```
class A extends B
```

um dos aspectos ao qual devemos ter atenção é a existência da classe *B*. Como em *X++* não trabalhamos com classes “externas”, a classe *B* tem que estar sendo compilada com a classe *A*. Porém, pode aparecer antes ou depois de *A*. Por isso, na primeira fase da análise semântica não é possível, ou seria mais difícil, realizar tal verificação. Se *B* é declarada depois de *A*, ao analisar a declaração de *A*, não seria possível saber se *B* é declarada ou não. Já nesta segunda fase, quando todas as classes estão inseridas na tabela de símbolos, é possível efetuar tal verificação.

Um outro aspecto que deve ser verificado em relação à hierarquia de classes é a existência de declarações circulares como

```
class A extends B  
class B extends C  
class C extends A
```

que é ilegal. Esse tipo de problema não será tratado nesta fase na análise semântica. Isto porque a constatação do problema só poderia ser feita na declaração de *C*. Assim, preferimos adiar essa verificação para a fase seguinte e poder apontar o erro já na análise da declaração de *A*. Assim, nesta fase, ao tratar de uma declaração de classe, o analisador semântico simplesmente verifica se a superclasse existe e aponta um erro em caso negativo ou completa a entrada da classe na tabela de símbolos, indicando qual a sua superclasse.

Para esta segunda fase foi definida a classe *VarCheck*, que se inicia da seguinte maneira:

Programa 10.1

```

1 package semanalysis;
2
3 import symtable.*;
4 import syntacticTree.*;
5
6 public class VarCheck extends ClassCheck {
7
8     public VarCheck()
9     {
10         super();
11     }
12
13     public void VarCheckRoot(ListNode x) throws SemanticException
14     {
15         ClassCheckRoot(x);    // faz análise das classes
16         VarCheckClassDeclListNode(x);
17         if (foundSemanticError != 0) // se houve erro, lança exceção
18             throw new SemanticException(foundSemanticError +
19                   " Semantic Errors found (phase 2)");
20     }

```

Essa classe estende a classe *ClassCheck*, de forma que todas as variáveis aí definidas possam ser utilizadas aqui. O construtor simplesmente invoca o construtor da superclasse e o método *VarCheckRoot* é o método que deve ser chamado pelo compilador para disparar a análise semântica. O que esse método faz é invocar a primeira fase da análise semântica (método *ClassCheckRoot*) e, caso não ocorram erros, iniciar a segunda fase chamando *VarCheckClassDeclListNode*, passando como argumento a raiz da árvore sintática.

Como de costume, esse método percorre a lista de classes chamando o método que trata de cada uma delas. Se um erro ocorre em alguma das classes, o contador de erros é incrementado e a análise continua para o resto da lista de classes.

Programa 10.2

```

1     public void VarCheckClassDeclListNode(ListNode x)
2     {
3         if (x == null) return;
4         try {
5             VarCheckClassDeclNode( (ClassDeclNode) x.node);

```

```

6     }
7     catch (SemanticException e)
8     { // se um erro ocorreu na análise da classe,
9      // dá a mensagem, mas faz a análise para próxima classe
10     System.out.println(e.getMessage());
11     foundSemanticError++;
12   }
13   VarCheckClassDeclListNode(x.next);
14 }
```

Vejamos agora o que faz o método que trata da declaração de cada classe.

Programa 10.3

```

1  public void VarCheckClassDeclNode(ClassDeclNode x)
2          throws SemanticException
3  {
4     Symtable temphold = Curtable; // salva tabela corrente
5     EntryClass c = null;
6     EntryClass nc;
7
8     if (x == null) return;
9     if (x.supername != null)
10    { // verifica se superclasse foi definida
11       c = (EntryClass) Curtable.classFindUp(x.supername.image);
12       if (c == null) // Se não achou superclasse, ERRO
13       {
14           throw new SemanticException(x.position, "Superclass "+
15                                         x.supername.image + " not found");
16       }
17    }
18    nc = (EntryClass) Curtable.classFindUp(x.name.image);
19    nc.parent = c; // coloca na tabela o apontador p/ superclasse
20    Curtable = nc.nested; // tabela corrente = tabela da classe
21    VarCheckClassBodyNode(x.body);
22    Curtable = temphold; // recupera tabela corrente
23 }
```

Inicialmente, a referência à tabela corrente é salva na variável *temphold*. Em seguida, verifica-se se existe no nó corrente da árvore sintática o filho *supername*, que indica que a cláusula *extends* foi usada. Em caso positivo, a entrada correspondente à superclasse é procurada na tabela utilizando o método *Symtable.classFindUp*, que, como visto no capítulo 9 respeita as regras de escopo para as classes. Assim, se o programador tentar utilizar como superclasse uma classe que não foi declarada ou que não é acessível no contexto corrente, a pesquisa falha e uma exceção é lançada, interrompendo a análise semântica para toda essa classe.

Caso a superclasse seja encontrada, a variável *c* possui a correspondente entrada na tabela de símbolos. Em seguida, a entrada correspondente à classe sendo declarada é procurada na tabela e completada com a referência à superclasse. Este é basicamente todo o processamento realizado pelo método. Posteriormente, valor de *Curtable*

é atualizado para que as declarações que serão analisadas nos nós “abaixo” do nó corrente sejam aplicadas na parte da tabela de símbolos correspondente a essa classe. O método *VarCheckClassBodyNode* é invocado para tratar do corpo da classe e, no retorno, o valor de *Curtable* é restaurado.

10.2 Análise da declaração de variáveis e métodos

Nesta fase da análise semântica são também analisados as declarações das variáveis da classe, seus métodos e construtores. O método que analisa o corpo da classe simplesmente invoca os métodos que tratam de cada uma das listas: de classes aninhadas, de variáveis, de construtores e de métodos, nesta ordem. O método que trata das classes aninhadas é o mesmo que trata as declarações de classes, ou seja, *VarCheckClassDeclListNode*.

Programa 10.4

```

1  public void VarCheckClassBodyNode(ClassBodyNode x)
2  {
3      if (x == null) return;
4
5      VarCheckClassDeclListNode(x.clist);
6      VarCheckVarDeclListNode(x.vlist);
7      VarCheckConstructDeclListNode(x.ctlist);
8
9      // se não existe constructor(), insere um falso
10     if ( Curtable.methodFindInclass("constructor", null) == null)
11     {
12         Curtable.add(new EntryMethod("constructor",
13                                     Curtable.levelup, true));
14     }
15     VarCheckMethodDeclListNode(x.mlist);
16 }
```

Os métodos que tratam das declarações de variáveis são mostrados no programa 10.5. O primeiro trata da lista de declarações e o segundo insere cada variável na tabela de símbolos. Inicialmente, o método tenta encontrar na tabela de símbolos a classe utilizada na declaração. Por exemplo, ao analisar a declaração

MyClass x, y;

é necessário verificar se a classe *MyClass* foi declarada.

Note que mesmo se a variável estiver sendo declarada como um tipo básico – *int* ou *string* –, essa busca utilizando *Symtable.classFindUp* irá funcionar. Para isso, introduzimos na tabela de símbolos duas entradas fictícias que representam os tipos básicos e que, por estarem no nível mais alto da tabela, são acessíveis de qualquer ponto do programa.

Caso a classe da declaração não seja encontrada, um erro ocorre. Caso contrário, para cada variável na declaração é inserida uma entrada na tabela de símbolos corrente indicando, além do seu tipo, sua dimensão.

Nenhum controle é feito sobre a existência de declarações repetidas, ou seja, se já há uma variável com o mesmo nome. Isto ocorre em virtude das características semânticas que queremos dar à linguagem. Em X^{++} , não se deve permitir que uma variável sobreponha ou redeclare uma outra variável em uma superclasse. Assim, neste ponto da análise semântica seria possível verificar se a variável sendo definida já foi declarada na mesma classe, mas não seria possível verificar se isso ocorre em uma superclasse, lembrando que a superclasse pode aparecer depois, dentro do código-fonte, e que suas declarações de variável ainda não foram analisadas. Por isso, esse aspecto será verificado na próxima fase da análise semântica.

Programa 10.5

```

1  public void VarCheckVarDeclListNode(ListNode x)
2  {
3      if (x == null) return;
4      try {
5          VarCheckVarDeclNode( (VarDeclNode) x.node);
6      }
7      catch (SemanticException e)
8      {
9          System.out.println(e.getMessage());
10         foundSemanticError++;
11     }
12     VarCheckVarDeclListNode(x.next);
13 }
14
15 public void VarCheckVarDeclNode(VarDeclNode x)
16                     throws SemanticException
17 {
18     EntryTable c;
19     ListNode p;
20
21     if (x == null) return;
22
23     // acha entrada do tipo da variável
24     c = Curtable.classFindUp(x.position.image);
25     // se não achou, ERRO
26     if (c == null)
27         throw new SemanticException(x.position, "Class "+
28                         x.position.image + " not found");
29
30     // para cada variável da declaração, cria uma entrada na tabela
31     for (p = x.vars; p != null; p = p.next)
32     {
33         VarNode q = (VarNode) p.node;
34         Curtable.add(new EntryVar(q.position.image, c, q.dim));
35     }
36 }
```

Em seguida é feita a análise dos construtores. Para cada declaração de construtor, inicialmente são analisados os parâmetros declarados. Cada parâmetro é analisado e verifica-se se o tipo correspondente existe, como foi feito na declaração de variáveis. Caso o tipo não exista, um erro ocorre. Na linha 45 do programa 10.6 é montada uma lista (um objeto do tipo *EntryRec* descrito no capítulo 8) com os tipos e dimensões de cada parâmetro. Na verdade, essa lista é montada de forma invertida pois cada parâmetro é inserido no início da lista.

Fora do *while*, essa lista é invertida e, então, utilizada para procurar o construtor na tabela de símbolos. O método *Symtable.methodFindInClass* é utilizado com este intuito. Esse método é mostrado no programa 10.7 e utiliza na pesquisa o nome de um método e uma lista de tipos, por meio de um objeto *EntryRec*. Além disso, ele limita a sua busca à classe corrente. Isso porque o construtor é um método sem nome, declarado por meio da palavra-chave *constructor* no programa-fonte, mas tratado pelo analisador como um método qualquer. A diferença é que ele é inserido na tabela de símbolos utilizando o nome “*constructor*”, já que não é permitido um método com esse nome. Porém, em uma superclasse, tal construtor, com os mesmos parâmetros, pode ter sido declarado, o que não invalida a declaração corrente. Em resumo, um erro só ocorre se na classe corrente um construtor com os mesmos parâmetros foi declarado. Caso contrário, o construtor é inserido na tabela corrente.

Na linha 58 do programa 10.6, vemos que o construtor é inserido como um método normal. Na chamada ao construtor de *EntryMethod*, o primeiro argumento é o nome dado ao método, ou seja, “*constructor*”. O segundo é o tipo de retorno do método e, neste caso, usamos a entrada na tabela de símbolos da classe atual, que está em *CurtTable.levelup*. O terceiro argumento é a dimensão do tipo de retorno. E, finalmente, o último é a lista de tipos dos parâmetros formais do método.

Programa 10.6

```

1  public void VarCheckConstructDeclListNode(ListNode x)
2  {
3      if (x == null) return;
4
5      try {
6          VarCheckConstructDeclNode( (ConstructDeclNode) x.node);
7      }
8      catch (SemanticException e)
9      {
10         System.out.println(e.getMessage());
11         foundSemanticError++;
12     }
13     VarCheckConstructDeclListNode(x.next);
14 }
15
16
17 public void VarCheckConstructDeclNode(ConstructDeclNode x)
18         throws SemanticException
19 {
20     EntryMethod c;
21     EntryRec r = null;
22     EntryTable e;
23     ListNode p;
```

```

24 VarDeclNode q;
25 VarNode u;
26 int n;
27
28     if (x == null) return;
29     p = x.body.param;
30     n = 0;
31     while (p != null) // para cada parâmetro do construtor
32     {
33         q = (VarDeclNode) p.node; // q = no com a declaração do parâmetro
34         u = (VarNode) q.vars.node; // u = no com o nome e dimensão
35         n++;
36         // acha a entrada do tipo na tabela
37         e = Curtable.classFindUp(q.position.image);
38
39         // se não achou: ERRO
40         if (e == null)
41             throw new SemanticException(q.position, "Class " +
42                             q.position.image + " not found");
43
44         // constrói a lista com os parâmetros
45         r = new EntryRec(e, u.dim, n, r);
46         p = p.next;
47     }
48
49     if (r != null)
50         r.inverte(); // inverte a lista
51
52     // procura construtor com essa assinatura dentro da mesma classe
53     c = Curtable.methodFindInclass("constructor", r);
54
55     if (c == null)
56     {           // se não achou, insere
57         c = new EntryMethod("constructor", Curtable.levelup, 0, r);
58         Curtable.add(c);
59     }
60     else // construtor já definido na mesma classe: ERRO
61         throw new SemanticException(x.position, "Constructor " +
62                         Curtable.levelup.name +
63                         "(" + (r == null? "" : r.toString()) + ")" +
64                         " already declared");
65 }
```

Um outro detalhe deve ser notado no programa 10.4 mostrado anteriormente. Em X++, toda classe deve ter um construtor sem parâmetros que será chamado caso o corpo de um construtor de uma subclasse não utilize explicitamente o comando *super* (ver capítulo 12 para mais detalhes). Assim, se a classe não possuir a declaração de tal construtor, uma entrada falsa é inserida na tabela de símbolos (Programa 10.4, linha 12). O último argumento da chamada ao construtor *EntryMethod* indica justamente que se trata de um construtor falso.

Programa 10.7

```

1  public EntryMethod methodFindInClass(String x, EntryRec r)
2  {
3      EntryTable p = top;
4      EntryClass q;
5
6      // para cada entrada da tabela
7      while (p != null)
8      {
9          // verifica se tipo é EntryMethod e compara o nome
10         if (p instanceof EntryMethod && p.name.equals(x) )
11         {
12             EntryMethod t = (EntryMethod) p;
13             // compara os parâmetros
14             if (t.param == null)
15             {
16                 if (r == null) return t;
17             }
18             else
19             {
20                 if (t.param.equals(r))
21                     return t;
22             }
23         }
24         p = p.next; // próxima entrada
25     }
26     return null; // não achou
27 }
```

*

No programa 10.8 são mostrados os métodos que tratam das declarações de métodos da classe. O procedimento é muito semelhante para a declaração de construtores. As diferenças iniciam-se na linha 50, onde o nome do método é procurado na tabela, e não a palavra “constructor”. Em seguida, caso o método não exista na classe corrente, é adicionada uma entrada na tabela de símbolos descrevendo o método, ou seja, seu nome, tipo de retorno, dimensão do retorno e tipos dos parâmetros formais. Toda essa informação é retirada do nó corrente da árvore sintática.

Programa 10.8

```

1  public void VarCheckMethodDeclListNode(ListNode x)
2  {
3      if (x == null) return;
4      try {
5          VarCheckMethodDeclNode( (MethodDeclNode) x.node);
6      }
7      catch (SemanticException e)
8      {
9          System.out.println(e.getMessage());
10         foundSemanticError++;
11     }
12     VarCheckMethodDeclListNode(x.next);
13 }
```

14

```

1 public void VarCheckMethodDeclNode(MethodDeclNode x)
2             throws SemanticException
3 {
4     EntryMethod c;
5     EntryRec r = null;
6     EntryTable e;
7     ListNode p;
8     VarDeclNode q;
9     VarNode u;
10    int n;
11
12    if (x == null) return;
13    p = x.body.param;
14    n = 0;
15    while (p != null)           // para cada parâmetro do método
16    {
17        n++;
18        q = (VarDeclNode) p.node; // q = nó da declaração do parâmetro
19
20        u = (VarNode) q.vars.node; // u = nó com o nome e dimensão
21        // acha a entrada na tabela do tipo
22        e = Curtable.classFindUp(q.position.image);
23        // se não achou, ERRO
24        if (e == null)
25            throw new SemanticException(q.position, "Class " +
26                                         q.position.image + " not found");
27
28        // constrói lista de tipos dos parâmetros
29        r = new EntryRec(e, u.dim, n, r);
30        p = p.next;
31    }
32    if (r != null)
33        r = r.inverte(); // inverte a lista
34
35    // procura na tabela o tipo de retorno do método
36    e = Curtable.classFindUp(x.position.image);
37    if (e == null)
38        throw new SemanticException(x.position, "Class " +
39                                     x.position.image + " not found");
40
41    // procura método na tabela, dentro da mesma classe
42    c = Curtable.methodFindInclass(x.name.image, r);
43    if (c == null)
44    {
45        // se não achou, insere
46        c = new EntryMethod(x.name.image, e, x.dim, r);
47        Curtable.add(c);
48    }
49    else // método já definido na mesma classe, ERRO
50        throw new SemanticException(x.position, "Method " +
51                                     x.name.image + "(" + (r == null? "" : r.toString()) +
52                                     + ")" + " already declared");
53
54 }

```

Com isso termina a segunda fase da análise semântica. Ao seu final, temos na tabela de símbolos as classes declaradas e, para cada uma delas, a lista de variáveis, métodos e construtores. Também neste caso, os métodos responsáveis pela análise são bastante simples e poucos tipos de erros semânticos podem ocorrer. Além disso, somente parte da árvore sintática precisa ser analisada. Os nós que estão dentro (ou abaixo) da declaração de métodos e construtores não são analisados, pois nenhuma informação pertinente a essa fase da análise semântica existe aí. O trabalho realmente “pesado” da análise semântica será feito na terceira e última fase, que será vista no capítulo 11.

10.3 O programa principal

Da mesma maneira que no capítulo 9, é preciso alterar o método principal do nosso compilador para que este inclua a chamada a esta segunda fase da análise semântica. Isto é feito como mostrado no programa 10.9.

Programa 10.9

```

1 // verifica se pode operar sobre a árvore sintática
2 if ( parser.token_source.foundLexError()
3     + parser.contParseError == 0)
4 {
5     if (print_tree) // exibir a árvore
6     {
7         PrintTree prt = new PrintTree();
8         prt.printRoot(root); // chama método para imprimir árvore
9     }
10 VarCheck tc = new VarCheck();
11 try {
12     tc.VarCheckRoot(root);
13     System.out.println("0 Semantic Errors found");
14 }
15 catch (SemanticException e)
16 {
17     System.out.println(e.getMessage());
18 }
19 }
```

Na linha 10 é criado o objeto do tipo *VarCheck* e, na linha 12, é chamado o método que trata do nó raiz da árvore sintática. Note que esse método trata de executar a primeira fase da análise semântica e verificar que nenhum erro tenha sido encontrado, antes de prosseguir com a sua parte da análise. Lembre, também, que a classe *VarCheck* estende a classe *ClassCheck* e, por isso, apenas um objeto do tipo *VarCheck* é criado.

10.4 Arquivos-fonte do compilador

No diretório *cap10* foi incluída a versão nova do arquivo *.jj* e, no diretório *cap10/sema-analysis*, o arquivo *VarCheck.java*, descrito neste capítulo.

Como exemplo, no diretório *ssamples* temos o arquivo *bintree-erro-semantico-fase2.x*, em que tanto um construtor da classe *data* como um método da classe *bintree* são declarados duas vezes. Nesse segundo caso, embora os dois métodos *insert* tenham sido declarados com tipos de retorno diferentes, seus nomes e tipos de parâmetros são idênticos, o que causa uma “colisão” entre eles e, por isso, um erro.

Assim, ao executarmos o nosso compilador com esse arquivo temos:

```
cap10$ java parser.langX ..//ssamples/bintree-erro-semantico-fase2.x
X++ Compiler - Version 1.0 - 2004
Reading from file ..//ssamples/bintree-erro-semantico-fase2.x . .
0 Lexical Errors found
0 Syntactic Errors found
Line 25 column 1: Constructor data(int, int, int) already declared
Line 81 column 1: Method insert(data) already declared
2 Semantic Errors found (phase 2)
```

nesses níveis. Isto é, só é possível obter um resultado útil se os tipos das variáveis forem consistentes entre si.

• **Stringificação:** uma maneira de transformar palavras em tipos primitivos.

Capítulo 11

Análise Semântica – Parte Final

Finalmente chegamos à última etapa da análise semântica. Esta é a mais complexa das três fases. Nela serão tratados os diversos aspectos deixados pendentes nas fases anteriores, como, por exemplo, a verificação da definição circular de classes e, principalmente, a checagem de tipos. Na seção 11.1 serão comentadas as principais características e destacada a importância da checagem de tipos num compilador. Em seguida, será apresentada a sua implementação para nosso compilador X^{++} em três seções: a primeira descreve a análise semântica para os nós mais altos como declaração de classes e suas variáveis; a segunda descreve a análise para os comandos da linguagem; e a terceira, para os diversos tipos de expressões.

11.1 Checagem de tipos

Dependendo do tipo de operação que queremos realizar, devemos utilizar comandos específicos, que, em geral, possuem regras bem definidas de formação. Por exemplo, ao utilizarmos em X^{++} o comando de seleção *if*, devemos ter algo como

```
if ( expr )
    comando 1;
else
    comando 2;
```

onde *expr* deve ser uma expressão cujo resultado seja do tipo *int* (lembrando que X^{++} não possui tipo booleano). Obviamente, essa verificação não pode ser feita durante a análise sintática.

Então, como é feita essa checagem? Vamos iniciar com um exemplo simples.

```
if ( b )
    return 0;
```

Esse trecho de programa gera a árvore sintática mostrada na figura 11.1. Para analisar se o comando *if* é legal, isto é, se sua expressão de controle produz um resultado inteiro, seria bastante simples. Devemos

- procurar a variável *b* na tabela de símbolos;
- pegar o tipo com que a variável foi declarada;
- lançar uma exceção caso o tipo não seja *int*.

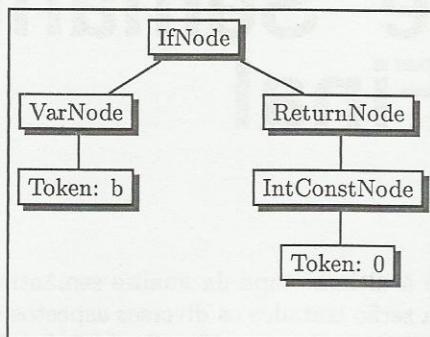


Figura 11.1 – Exemplo de checagem de tipos para comando *if*.

Devemos lembrar, porém, que a expressão de controle nem sempre é uma variável simplesmente. Se olharmos na definição do nó *IfNode*, veremos que o filho correspondente à expressão de controle é do tipo *ExpreNode*, que é definida conforme mostra o programa 11.1. *ExpreNode* é uma classe abstrata e representa uma expressão qualquer, como, por exemplo, uma simples variável, uma constante, uma soma de duas subexpressões, uma operação de alocação de um objeto.

Programa 11.1

```

1 package syntacticTree;
2
3 import Token;
4
5 abstract public class ExpreNode extends GeneralNode {
6
7     public ExpreNode(Token t)
8     {
9         super(t);
10    }
11 }
  
```

Para sermos mais precisos, eis a lista das classes que estendem *ExpreNode* e o que cada uma delas representa.

- *IntConstNode*: uma constante do tipo *int*, como em

```
a = 10;
```

- *StringConstNode*: uma constante do tipo *string*, como em

```
a = "I am a string"
```

- *NullConstNode*: uso da constante *null*, como em

```
a = null
```

- *VarNode*: uso de uma variável como a seguir (aqui temos dois *VarNodes*)

```
a = b
```

- *AddNode*: uma operação de adição ou subtração entre duas subexpressões, como

```
a = b - c
```

- *MultNode*: uma operação de multiplicação ou divisão ou resto entre duas subexpressões, como em

```
a = b % c
```

- *UnaryNode*: uma operação de negação unária, como

```
a = - 123
```

- *RelationalNode*: uma operação de comparação ou subtração entre duas subexpressões, como

```
a - b >= c
```

- *NewObjectNode*: operação de alocação de um novo objeto, como em

```
a = new MyClass(1021)
```

- *NewArrayNode*: alocação de um array, como em

```
a = new int[69]
```

- *CallNode*: chamada de um método, como em

```
v.myMethod(a, b, c)
```

- *DotNode*: acesso a uma variável de instância de algum objeto, como em

```
a = myObject.myField
```

- *IndexNode*: acesso a um elemento de um array como em

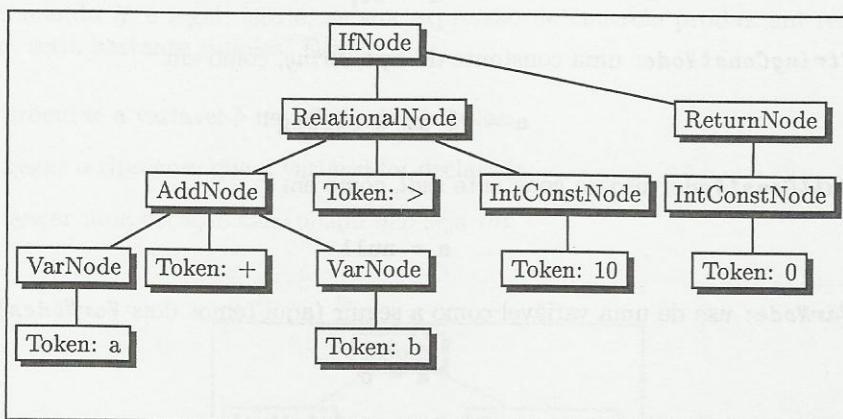


Figura 11.2 – Segundo exemplo de checagem de tipos para comando *if*.

$a = b[10]$

Vamos supor, então, um segundo exemplo de comando *if*, cuja árvore sintática é apresentada na figura 11.2.

```
if ( a + b > 10 )
    return 0;
```

Para esse exemplo, a situação se complica um pouco mais. Antes de saber qual o tipo da expressão de controle, é necessário calcular o tipo de cada subexpressão. Por exemplo, é preciso saber se `a + b` e `10` são expressões do tipo inteiro para poderem ser comparadas por meio do operador `>`. Por sua vez é preciso saber se as variáveis `a` e `b` são do tipo inteiro para poderem ser somadas e produzir um resultado inteiro.

Como podemos perceber, os tipos das expressões devem ser calculados “de baixo para cima”, a partir das subexpressões mais simples, até que se chegue à expressão completa. De maneira ilustrativa, vejamos no programa 11.2 o que devem fazer os métodos ao analisarem alguns dos nós da árvore sintática do comando *if* da figura 11.2.

Programa 11.2

```

1  void TypeCheck(IfNode x)
2  {
3      t = TypeCheck(x.expr);
4      se t é "int"
5          OK;
6      senão
7          erro: expressão deve ser inteira;
8  }
9
10 "tipo" TypeCheck(ExpreNode x)
11 {

```

```

12     descobre a classe "real" de x e chama o método adequado;
13     retorna o valor retornado na chamada;
14 }
15
16 "tipo" TypeCheck(RelationalNode x)
17 {
18     t1 = TypeCheck(x.expr1);
19     t2 = TypeCheck(x.expr2);
20     se t1 e t2 são "int"
21         retorna como resultado "int";
22     se x.position == "==" ou x.position == "!="
23         se t1 é null e t2 é "string" ou vice-versa
24             retorna como resultado "int"
25         se t1 é null e t2 é um objeto ou vice-versa
26             retorna como resultado "int"
27         se t1 e t2 são null
28             retorna como resultado "int"
29         se t1 e t2 são objetos de tipos comparáveis
30             retorna como resultado "int"
31     senão
32         erro: operandos inválidos em operação relacional
33 }
34
35 "tipo" TypeCheck(VarNode x)
36 {
37     procura x.position na tabela de símbolos;
38     se achou
39         retorna tipo da variável na tabela;
40     senão
41         erro: variável não definida;
42 }
```

Note que assim como na análise do comando *if*, cada tipo de *ExpreNode* possui suas própria regras de verificação. Por exemplo, ao analisar um nó do tipo *RelationalNode* são necessárias diversas regras para validar a expressão:

- se os dois operandos forem inteiros, a expressão está OK e o seu tipo é *int*;
- se os dois operandos forem *string*, a expressão é válida somente se o operador for de igualdade ou desigualdade;
- idem para dois objetos de tipos comparáveis (tipos iguais ou um é subclasse do outro);
- strings e objetos podem ser comparados quanto à igualdade e à desigualdade com um outro operando que seja *null*.

Além de verificar a validade das expressões, a checagem de tipos é também útil para guiar a geração de código, como veremos no capítulo 12. Vamos supor, por exemplo, que temos a seguinte expressão:

```
MyClass x;
a = x.myMethod( b + c);
```

Se as variáveis *b* e *c* foram do tipo *int*, a geração de código deve:

- criar as instruções para somar esses valores e produzir um resultado inteiro;
- chamar o método *MyClass.myMethod(int)*.

No caso, porém, em que *a* é um string e *b*, um inteiro, o código gerado deve ser diferente. O código deverá ser gerado de forma que as seguintes operações sejam realizadas:

- transformar o valor inteiro de *b* em um string;
- concatenar o valor de *a* com esse valor transformado, produzindo como resultado um string;
- chamar o método *MyClass.myMethod(string)*.

Ou seja, além de gerar instruções para tratar de maneira diversa os operandos de tipos diferentes, a análise dos tipos das expressões serviu como guia para decidir qual método da classe *MyClass* deve ser invocado: aquele que recebe como argumento um string ou um número inteiro.

Nas próximas seções mostraremos com detalhes cada um dos métodos que tratam dos nós da árvore sintática, implementados na classe *TypeCheck*. Estas serão seções longas, pois diferentemente das fases anteriores, toda a árvore sintática tem que ser visitada, e não apenas os níveis mais altos. Como consequência, praticamente todos os tipos de nós têm tratamentos particulares a serem feitos por métodos específicos, definidos nesta classe.

11.2 Análise das declarações

Iniciamos, desta vez, com o programa principal, que deve ser alterado da mesma forma que no capítulo 10. A mudança feita no programa 11.3, linha 12, é concernente à utilização de um objeto do tipo *TypeCheck* para disparar a análise semântica.

Programa 11.3

```

1 // verifica se pode operar sobre a árvore sintática
2 if ( parser.token_source.foundLexError()
3     + parser contParseError == 0)
4 {
5     if (print_tree) // exibir a árvore
6     {
7         PrintTree prt = new PrintTree();
8         prt.printRoot(root); // chama método para imprimir árvore
9     }
}
```

```

10     TypeCheck tc = new TypeCheck();
11     try {
12         tc.TypeCheckRoot(root);
13         System.out.println("0 Semantic Errors found");
14     }
15     catch (SemanticException e)
16     {
17         System.out.println(e.getMessage());
18     }

```

Na classe *TypeCheck*, temos o método *TypeCheckRoot*, que é responsável por disparar a análise semântica. Esse método primeiro chama o método *VarCheckRoot* visto no capítulo 10 e que é responsável por executar as duas fases anteriores da análise semântica. Por ser a classe *TypeCheck* uma subclasse de *VarCheck*, toda a informação coletada nas fases anteriores (em particular, a tabela de símbolos) pode ser utilizada aqui. O programa 11.4 mostra a declaração da classe e do método *TypeCheckRoot*.

Programa 11.4

```

1  public class TypeCheck extends VarCheck
2  {
3      int nesting; // controla o nível de aninhamento em comandos repetitivos
4      protected int Nlocals; // conta número de variáveis locais num método
5      type Returntype; // tipo de retorno de um método
6      protected final EntrySimple STRING_TYPE, // variáveis que apontam os
7                      INT_TYPE, // tipos básicos na tabela de símbolos
8                      NULL_TYPE;
9      protected EntryMethod CurMethod; // método sendo analisado
10     boolean cansuper; // indica se chamada super é permitida
11     protected boolean store; // indica se está armazenando valor numa var
12
13     public TypeCheck()
14     {
15         super();
16         nesting = 0;
17         Nlocals = 0;
18         STRING_TYPE = (EntrySimple) Maintable.classFindUp("string");
19         INT_TYPE = (EntrySimple) Maintable.classFindUp("int");
20         NULL_TYPE = new EntrySimple("$NULL$");
21         Maintable.add(NULL_TYPE);
22     }
23
24     public void TypeCheckRoot(ListNode x) throws SemanticException
25     {
26         VarCheckRoot(x); // faz análise das variáveis e métodos
27         TypeCheckClassDeclListNode(x); // faz análise do corpo dos métodos
28         if (foundSemanticError != 0) // se houve erro, lança excesso
29             throw new SemanticException(foundSemanticError +
30                                 " Semantic Errors found (phase 3)");
31     }

```

Vemos que diversas variáveis são declaradas na classe. Os comentários dão uma noção da sua utilização, porém durante a explicação dos métodos da classe, iremos detalhá-las com mais cuidado. De importante temos somente a inclusão – a exemplo do que foi feito com os tipos básicos da linguagem – de um tipo fictício para representar o tipo de dado assumido pela constante `null`. Uma entrada com o nome “`$NULL$`” é inserida no nível mais alto da tabela de símbolos, de modo a ser acessível em qualquer escopo do programa sendo analisado. Além disso, fazemos uma busca aos tipos básicos da linguagem e atribuímos às variáveis `STRING_TYPE` e `INT_TYPE` de modo que todas as vezes que precisarmos utilizar essas entradas, não necessitaremos pesquisar a tabela de símbolos.

Como sempre, iniciamos pelos nós mais altos da árvore, ou seja, com os métodos que tratam da lista de classes e das declarações (Programa 11.5). Não há nada de novo no primeiro método. No segundo, a única novidade é a verificação de uma possível declaração circular de classes. Na linha 25 é feita uma chamada ao método `circularSuperclass` passando como argumentos a entrada na tabela de símbolos e a entrada da sua superclasse declarada. O que o método faz é ir “subindo” com chamadas recursivas na hierarquia das classes até que ou o segundo argumento tenha o valor `null` ou o segundo argumento seja igual ao primeiro. No primeiro caso, não há erro de circularidade. No segundo, encontrou-se uma cadeia hierárquica em que a classe original aparece como ancestral dela mesma e, portanto, um erro foi detectado.

Programa 11.5

```

1  public void TypeCheckClassDeclListNode(ListNode x)
2  {
3      if(x == null) return;
4      try {
5          TypeCheckClassDeclNode( (ClassDeclNode) x.node);
6      }
7      catch (SemanticException e)
8      {
9          // se um erro ocorreu na análise da classe,
10         // dá a mensagem, mas faz a análise para a próxima classe
11         System.out.println(e.getMessage());
12         foundSemanticError++;
13     }
14     TypeCheckClassDeclListNode(x.next);
15 }
16
17 public void TypeCheckClassDeclNode(ClassDeclNode x)
18             throws SemanticException
19 {
20     Symtable temphold = Curtable; // salva tabela corrente
21     EntryClass nc;
22
23     if (x == null) return;
24     nc = (EntryClass) Curtable.classFindUp(x.name.image);
25     if ( circularSuperclass(nc, nc.parent) )
26     {
27         // se existe declaração circular, ERRO
28         nc.parent = null;
29         throw new SemanticException(x.position, "Circular inheritance");
30     }

```

```

30     Curtable = nc.nested; // tabela corrente = tabela da classe
31     TypeCheckClassBodyNode(x.body);
32     Curtable = temphold; // recupera tabela corrente
33 }
34
35 // verifica se existe referência circular de superclasses
36 private boolean circularSuperclass(EntryClass orig, EntryClass e)
37 {
38     if (e == null) return false;
39     if (orig == e)
40         return true;
41     return circularSuperclass(orig, e.parent);
42 }

```

O método que analisa o corpo da classe é apresentado no programa 11.6. Também aqui nada de novo ocorre, apenas a chamada à análise da lista de classes aninhadas, das variáveis, dos construtores e dos métodos. Neste ponto estamos já familiarizados com os métodos que simplesmente percorrem essas listas e, por isso, não serão apresentados.

Programa 11.6

```

1 public void TypeCheckClassBodyNode(ClassBodyNode x)
2 {
3     if (x == null) return;
4
5     TypeCheckClassDeclListNode(x.clist);
6     TypeCheckVarDeclListNode(x.vlist);
7     TypeCheckConstructDeclListNode(x.ctlist);
8     TypeCheckMethodDeclListNode(x.mlist);
9 }

```

Vamos, agora, abordar a declaração das variáveis de instância da classe. O método percorre cada uma das variáveis que estão sendo declaradas pelo nó da árvore sintática e, para cada uma, chama o método *Syntable.varFind*, que procura a variável na tabela de símbolos. Essa procura, conforme mostra o programa 11.7, é feita – diversamente da pesquisa de classes que utiliza a estrutura de aninhamento das classes – por meio da estrutura hierárquica da classe corrente. Ou seja, a variável é procurada na classe corrente, depois na sua superclasse, na supersuperclasse, e assim por diante. O segundo argumento, na linha 12, indica que desejamos a segunda ocorrência da variável na hierarquia. Se tal ocorrência existir, um erro foi detectado. Há também uma versão de *Syntable.varFind()* que utiliza um único argumento e retorna sempre a primeira ocorrência da variável na hierarquia de classes.

Programa 11.7

```

1 public void TypeCheckVarDeclNode(VarDeclNode x)
2             throws SemanticException
3 {
4     ListNode p;
5     EntryVar l;

```

```

6
7     if (x == null) return;
8     for (p = x.vars; p != null; p = p.next)
9     {
10         VarNode q = (VarNode) p.node;
11         // tenta pegar 2a. ocorrência da variável na tabela
12         l = Curtable.varFind(q.position.image, 2);
13
14         // se conseguiu, a variável foi definida duas vezes, ERRO
15         if (l != null)
16             throw new SemanticException(q.position, "Variable "+
17                 q.position.image + " already declared");
18     }
19 }
```

*

Note que as características de pesquisa de variáveis descritas determinam um importante aspecto semântico da linguagem *X⁺⁺*: a existência de qualquer forma de “aninhamento de objetos” dentro de classes aninhadas (diferente de Java). Isso significa que se uma classe *B* é definida no interior de uma classe *A*, duas instâncias dessas duas classes são objetos completamente distintos e o objeto do tipo *B* não prescinde da existência de um objeto do tipo *A* para ser criado. Portanto, dentro da classe *B*, as variáveis declaradas em *A* não são acessíveis (e *varFind* está de acordo com essa política). Em *X⁺⁺*, porém, somente a classe *A* (ou classes internas a esta) podem utilizar a classe *B*.

Programa 11.8

```

1  /* Esse método procura a n-ésima ocorrência do símbolo x na
2  tabela e também na(s) tabela(s) da(s) superclasse(s), apontada
3  por levelup.parent. Procura por uma entrada do tipo EntryVar */
4
5  public EntryVar varFind(String x, int n)
6  {
7      EntryTable p = top;
8      EntryClass q;
9
10     while (p != null)
11     {
12         if (p instanceof EntryVar && p.name.equals(x))
13             if ( --n == 0)
14                 return (EntryVar) p;
15         p = p.next;
16     }
17     q = levelup;
18     if (q.parent == null) return null;
19     return q.parent.nested.varFind(x, n);
20 }
21
22 public EntryVar varFind(String x)
23 {
24     return varFind(x, 1);
25 }
```

*

Em seguida, veremos o método que trata a declaração de um construtor (Programa 11.9). A primeira parte do método é simples e semelhante à fase anterior da análise semântica. A lista com os tipos de parâmetros é montada para que se possa procurar, usando o método *Symtable.methodFind*, a entrada na tabela de símbolos correspondente ao construtor. Esse método é bastante semelhante ao *varFind*, visto anteriormente, e será mostrado mais adiante, ainda neste capítulo pois determina características importantes da linguagem. Ele percorre a hierarquia de classes à procura do método com o nome e os tipos de parâmetros utilizados.

Note que não é realizada nenhuma verificação se o método foi encontrado ou não. Isso faz sentido, visto que temos certeza que o método existe – e está na classe corrente –, pois foi certamente inserido na tabela, na fase anterior. É bem verdade que poderíamos evitar essa pesquisa associando ao nó *ConstructDeclNode* a entrada na tabela de símbolos, no momento em que ela foi inserida, na fase anterior. Porém, didaticamente, consideramos melhor não fazê-lo e manter o método inalterado.

Programa 11.9

```

1  public void TypeCheckConstructDeclNode(ConstructDeclNode  x)
2          throws SemanticException
3  {
4      EntryMethod t;
5      EntryRec r = null;
6      EntryTable e;
7      EntryClass thisclass;
8      EntryVar thisvar;
9      ListNode p;
10     VarDeclNode q;
11     VarNode u;
12     int n;
13
14     if (x == null) return;
15     p = x.body.param;
16     n = 0;
17
18     // monta a lista com os tipos dos parâmetros
19     while (p != null)
20     {
21         q = (VarDeclNode) p.node; // q = nó com a declaração do parâmetro
22         u = (VarNode) q.vars.node; // u = nó com o nome e dimensão
23         n++;
24
25         // acha a entrada do tipo na tabela
26         e = Curtable.classFindUp(q.position.image);
27
28         // constrói a lista com os tipos dos parâmetros
29         r = new EntryRec(e, u.dim, n, r);
30         p = p.next;
31     }
32
33     if (r != null)
34         r = r.inverte(); // inverte a lista
35

```

```

36   // acha a entrada do construtor na tabela
37   t = Curtable.methodFind("constructor", r);
38   CurMethod = t; // guarda método corrente
39
40   // inicia um novo escopo na tabela corrente
41   Curtable.beginScope();
42
43   // pega a entrada da classe corrente na tabela
44   thisclass = (EntryClass) Curtable.levelup;
45
46   thisvar = new EntryVar("this", thisclass, 0);
47   Curtable.add(thisvar); // inclui variável local "this" com número 0
48   Returntype = null; // tipo de retorno do método = nenhum
49   nesting = 0; // nível de aninhamento de comandos for
50   Nlocals = 1; // inicializa número de variáveis locais
51   TypeCheckMethodBodyNode(x.body);
52   t.totallocals = Nlocals; // número de variáveis locais do método
53   Curtable.endScope(); // retira variáveis locais da tabela
54 }
```

A partir da linha 38, o método começa a preparar o analisador semântico para trabalhar no corpo do método, onde os comandos e expressões serão analisados. Inicialmente, guarda-se na variável *CurMethod* a entrada do método sendo analisado. Como não existem métodos aninhados em X^{++} , não precisamos preocupar-nos em salvar o valor anterior e depois recuperá-lo. Em seguida, é chamado o método *Syntable.beginScope* na tabela de símbolos corrente. O seu objetivo é marcar o estado em que se encontra a tabela corrente de modo que este possa ser recuperado quando terminar a análise do corpo do método. Os símbolos inseridos na tabela dentro do corpo do método como variáveis locais e parâmetros formais são retirados da tabela de símbolos de forma a não interferir na análise dos outros métodos. Lembre-se de que existe uma tabela por classe, e não uma por método, e assim dois métodos têm que compartilhar a mesma tabela sem que um interfira no outro. O método que retorna a tabela ao estado “salvo” por *beginScope* é *endScope*, chamado na linha 53 após a análise do corpo do método. Esses dois métodos foram apresentados no capítulo 8.

Voltando ao programa 11.9, vemos na linha 44 se atribuir à variável *thisclass* a referência à entrada da classe corrente na tabela de símbolos. Em seguida, insere-se na tabela corrente uma entrada de uma variável com o nome *this*. Assim, dentro do construtor, pode-se fazer referência a tal variável para referenciar o objeto corrente, como se faz em Java. Um outro modo de conseguir tal característica seria declarar o símbolo “*this*” como um token reservado (como, aliás, acontece em Java). A vantagem da nossa abordagem é que os métodos de análise semântica e geração de código não precisam tratar a variável de um modo especial (por exemplo, criando-se um *ThisNode* na árvore sintática). Além disso, o nome pode ser utilizado para outros fins, como nome de método ou variável. Isto nem sempre é bom. Imagine uma variável de instância com o nome “*this*”. A decisão de qual abordagem utilizar depende das características semânticas que se desejam obter da linguagem-alvo.

Antes de chamar a análise para o corpo do construtor, ainda algumas variáveis são preparadas:

- **Returntype**: guarda o tipo e a dimensão de retorno do método; como se trata de um construtor, que não retorna valor, a variável é inicializada com `null`;
- **nesting**: variável que controla o nível de aninhamento em que um determinado comando aparece dentro do construtor;
- **Nlocals**: conta o número de variáveis locais definidas no construtor. Inicialmente é 1, pois a variável `this` já foi inserida.

Ao retornar da análise do corpo do construtor, o número de variáveis locais utilizadas no método é guardado na entrada do método, na tabela de símbolos. Essa informação é utilizada na geração de código. O método termina com a limpeza da tabela. A análise da declaração de um método é bem semelhante à de um construtor e, por isso, não precisa ser apresentada aqui. As únicas diferenças são que, ao procurar o método na tabela de símbolos, utiliza-se o nome do método que está no nó da árvore sintática e que a variável `Returntype` é inicializada com um objeto do tipo `Type`, que indica o tipo e a dimensão do retorno esperado pelo método que está sendo declarado. Essas duas diferenças e a classe `Type` são mostradas no programa 11.10.

Programa 11.10

```

1  public void TypeCheckMethodDeclNode(MethodDeclNode x)
2          throws SemanticException
3  {
4      . . .
5
6      // acha a entrada do método na tabela
7      t = CurTable.methodFind(x.name.image, r);
8      CurMethod = t; // guarda método corrente
9
10     // Returntype = tipo de retorno do método
11     Returntype = new type(t.type, t.dim);
12
13     . . .
14 }
15
16 ////////////////////////////////////////////////////////////////// Class type //////////////////////////////////////////////////////////////////
17 package semanalysis;
18
19 import symtable.*;
20
21 public class type {
22     public EntryTable ty;    // entrada na tabela do tipo
23     public int dim;         // dimensão
24
25     public type(EntryTable t, int d)
26     {
27         ty = t;
28         dim = d;

```

```

29 }
30
31 public String dscJava()
32 {
33     return EntryTable.strDim(dim) + ty.dscJava();
34 }
35
36 }
37


---



```

Abordaremos, agora, o corpo do método ou construtor. Ambos são tratados pelo mesmo método, *TypeCheckMethodBodyNode*. Inicialmente, deve-se tratar da lista de parâmetros formais. Estes devem ser inseridos na tabela de símbolos como variáveis locais normais. Isto é feito, como mostrado na linha 5 do programa 11.11, chamando o método *TypeCheckLocalVarDeclListNode* e passando como argumento a lista de parâmetros que está no nó corrente, ou seja, *x.param*.

```

1   public void TypeCheckMethodBodyNode(MethodBodyNode x) Programa 11.11
2   {
3       if (x == null) return;
4       // trata parâmetro como variável local
5       TypeCheckLocalVarDeclListNode(x.param);
6
7       cansuper = false;
8       if (Curtable.levelup.parent != null)
9       { // existe uma superclasse para a classe corrente?
10
11           // acha primeiro comando do método
12           StatementNode p = x.stat;
13           while ( p instanceof BlockNode )
14               p = (StatementNode) ((BlockNode) p).stats.node;
15
16           // verifica se é chamada super
17           cansuper = p instanceof SuperNode;
18       }
19       try {
20           TypeCheckStatementNode(x.stat);
21       }
22       catch (SemanticException e)
23       {
24           System.out.println(e.getMessage());
25           foundSemanticError++;
26       }
27   }
28
29   public void TypeCheckLocalVarDeclListNode(ListNode x)
30   {
31       if (x == null) return;
32       try {
33           TypeCheckLocalVarDeclNode( (VarDeclNode) x.node);
34       }

```

```

35     catch (SemanticException e)
36     {
37         System.out.println(e.getMessage());
38         foundSemanticError++;
39     }
40     TypeCheckLocalVarDeclListNode(x.next);
41 }
42

```

11.3 Análise dos comandos

A partir deste ponto serão tratados os comandos do método. Na realidade, um *MethodBodyNode* tem como corpo um único comando do tipo *StatementNode*. Essa é também uma classe abstrata que representa qualquer um dos tipos de comandos, como um *IfNode*, para um comando *if*, ou *AtribNode*, para um comando simples de atribuição. Neste ponto em que estamos, teremos, em geral, o filho *x.stat* como um *BlockNode*, que serve para agrupar diversos comandos entre um { e um }.

A primeira verificação feita pelo método *TypeCheckMethodBodyNode*, ou melhor, iniciada aqui, é com relação à utilização do comando *super(...)* para invocar o construtor de uma superclasse. As regras para utilizá-lo são simples: a classe corrente deve ter uma superclasse declarada; ele só pode ser utilizado dentro de um construtor; e ele tem que ser o primeiro comando dentro do corpo do método. Outras regras poderiam ser adicionadas, como em Java, que não permite a utilização de variáveis de instância como argumentos da chamada, mas para o caso da nossa linguagem *X⁺⁺*, apenas essas três regras serão utilizadas.

A verificação real dessas regras é feita dentro do método que trata da análise semântica de um nó *SuperNode*, que representa a chamada em si. Esse método será visto mais adiante, neste capítulo, mas vale adiantar que a idéia é verificar naquele ponto se: a variável *Returntype* foi inicializada com o valor *null*, o que caracterizaria o método corrente como sendo um construtor; e verificar a variável *cansuper* que indica se o comando *super(...)* é ou não permitido no ponto onde foi inserido. Como mostrado nas linhas 8 a 18, a variável recebe o valor verdadeiro se a classe corrente possui uma superclasse e se o primeiro nó de comando, diferente de um *BlockNode*, é um *SuperNode*. Note que um programa do tipo

```

constructor()
{
    super();
}

```

é sintaticamente válido e também não deve ser considerado como uma violação das regras descritas. Por isso, o método que analisa o corpo do método pode ter que procurar pelo *super(...)*, por diversos níveis de aninhamento de blocos até encontrar o nó

correspondente ao primeiro comando “real”. Após essa operação, a análise semântica prossegue com a análise dos comandos.

Assim como é feito para a classe *ExprNode*, a classe *StatementNode* requer um tratamento especial para descobrir qual o tipo real do nó a ser analisado. O primeiro tipo de comando que pode aparecer é um bloco, ou “comando composto”, que serve para agrupar diversos comandos e é representado na árvore sintática por um *BlockNode*. A sua análise é bastante simples, como mostrado no programa 11.12. Antes de analisar a lista de comandos, é chamado o método *beginScope* para marcar o início de um escopo novo e, depois da análise, é realizada a chamada ao método *endScope*. O resto é uma chamada a *TypeCheckStatementListNode*, que, por sua vez, invoca *TypeCheckStatementNode* para cada nó (sempre do tipo *StatementNode*) dentro da lista.

Programa 11.12

```

1  public void TypeCheckBlockNode(BlockNode x)
2  {
3      Curtable.beginScope(); // início de um escopo
4      TypeCheckStatementListNode(x.stats);
5      Curtable.endScope(); // final do escopo, libera vars. locais
6  }
7
8  public void TypeCheckStatementListNode(ListNode x)
9  {
10
11     if (x == null) return;
12
13     try {
14         TypeCheckStatementNode( (StatementNode) x.node );
15     }
16     catch (SemanticException e)
17     {
18         System.out.println(e.getMessage());
19         foundSemanticError++;
20     }
21     TypeCheckStatementListNode(x.next);
22 }
```

O próximo tipo de comando que veremos é uma declaração de variáveis locais, representada por um nó *VarDeclNode*, lembrando que uma declaração de variável pode aparecer em qualquer ponto do método. Esse nó é o mesmo utilizado na declaração de variáveis de instância por representar a mesma construção sintática, embora semanticamente essas duas situações requeram ações bem diferentes. Poder-se-ia criar dois tipos de nós diferentes, um para declarações de variáveis de instância e outro para variáveis locais. O único inconveniente de usar um único tipo de nó é que a classe desse nó é declarada como subclasse de *StatementNode*, o que seria um pouco estranho no caso de declaração de variáveis de instância.

Na declaração de variáveis locais, em primeiro lugar se deve, como mostrado no início do programa 11.13, verificar se o tipo usado na declaração existe. Utiliza-se, para tal, o mesmo método *Symtable.classFindUp*. Obviamente um erro ocorre se

o tipo não é encontrado. Em seguida, dentro do comando *for* da linha 12, cada uma das variáveis é inserida na tabela de símbolos. Algumas verificações são feitas, visando a evitar conflitos nessas declarações.

Programa 11.13

```

1  public void TypeCheckLocalVarDeclNode(VarDeclNode x)
2      throws SemanticException
3  {
4      ListNode p;
5      VarNode q;
6      EntryVar l, u;
7      EntryTable c;
8
9      if (x == null) return;
10
11     // procura tipo da declaração na tabela de símbolos
12     c = Curtable.classFindUp(x.position.image);
13     // se não achou, ERRO
14     if (c == null)
15         throw new SemanticException(x.position, "Class "+
16             x.position.image + " not found.");
17     for (p = x.vars; p != null; p = p.next)
18     {
19         q = (VarNode) p.node;
20         l = Curtable.varFind(q.position.image);
21
22         // se variável ja existe é preciso saber que tipo de variável é
23         if (l != null)
24         {
25             // verifica se é local, definida no escopo corrente
26             if (l.scope == Curtable.scptr) // se for, ERRO
27                 throw new SemanticException(q.position, "Variable "+
28                     p.position.image + " already declared");
29
30             // c.c. verifica se é uma variável de classe
31             if (l.localcount < 0) // se for, dá uma advertência
32                 System.out.println("Line " + q.position.beginLine +
33                     " Column " + q.position.beginColumn +
34                     " Warning: Variable " + q.position.image +
35                     " hides a class variable");
36             else // senão, é uma variável local em outro escopo
37                 System.out.println("Line " + q.position.beginLine +
38                     " Column " + q.position.beginColumn +
39                     " Warning: Variable " + q.position.image +
40                     " hides a parameter or a local variable");
41         }
42         // insere a variável local na tabela corrente
43         Curtable.add(
44             new EntryVar(q.position.image, c, q.dim, Nlocals++))
45     }
46 }
```

A primeira verificação é em relação a outra variável local definida no mesmo escopo. Além do corpo do método, cada bloco de comandos define também um escopo para variáveis. Assim, uma variável local deve ser acessível a partir do ponto onde foi declarada até o final do bloco no qual foi definida. Uma variável local, quando inserida na tabela de símbolos, ocupa o “topo” da tabela, que, como visto anteriormente, funciona na forma de uma pilha. O método *Syntable.varFind* inicia a busca sempre do topo dessa pilha, priorizando, assim, a procura por variáveis locais.

Uma chamada a *Syntable.varFind*, na linha 20, procura na tabela de símbolos uma variável com o mesmo nome que seja acessível neste ponto do programa. Se tal variável for encontrada, então certamente ocorreu um erro se ela foi declarada no escopo corrente. Para verificar se isto aconteceu, compara-se o campo *scope* da variável que já estava na tabela com o campo *scptr* da tabela corrente que indica qual é o escopo atual. Se forem iguais, um erro ocorreu. Se não forem iguais, a variável encontrada pode ser uma variável local que está em outro escopo, um parâmetro formal ou uma variável de instância declarada nesta classe ou em uma de suas superclasses, o que não consideraremos um erro em X^{++} . Note que o comportamento de *Syntable.varFind* de pesquisar um símbolo a partir do topo da tabela corrente é adequado, pois caso uma entrada para a variável seja encontrada, será sempre a instância correspondente à declaração mais recentemente aí inserida, ou seja, sempre nos escopos mais próximos ao escopo corrente.

A próxima verificação a ser feita é sobre uma variável de instância com o mesmo nome da variável sendo definida. Embora não seja um erro em X^{++} , podemos advertir o programador sobre tal fato. Para tal checagem, verificamos o valor do campo *localcount* na entrada encontrada na tabela de símbolos. A cada variável local inserida na tabela, atribui-se um número seqüencial, começando com 0 para a variável *this*, 1 para o primeiro parâmetro formal, e assim sucessivamente. Se olharmos o método que insere as variáveis definidas na classe, veremos que o valor para esse campo, no caso das variáveis da classe, é sempre -1. Por isso, podemos diferenciar uma variável local de uma de instância.

E, finalmente, se a variável encontrada não está no mesmo escopo e não é de instância, só resta a possibilidade de que seja uma variável local em um escopo diverso. Da mesma forma, uma advertência é emitida.

Após tudo isso, se nenhum erro ocorreu, resta inserir a variável na tabela de símbolos. A chamada a *Syntable.add*, na linha 44, faz isso. A entrada criada para a variável é do tipo *EntryVar* e os argumentos utilizados na chamada do construtor são o nome da variável, o seu tipo, a sua dimensão e o seu número seqüencial. O número a ser atribuído à variável é controlado pela variável *Nlocals*, que é inicializada antes de se iniciar a análise do corpo de um método e incrementada a cada inserção de variável na tabela. No nosso compilador, variáveis locais definidas em escopos disjuntos não compartilham o mesmo número.

Veremos, agora, um comando simples, mas que introduz uma novidade, que é o uso do cálculo do tipo de expressões para a checagem de tipos. No comando *print expr*; temos *expr* como uma expressão qualquer, de acordo com a gramática da linguagem. Mas o que queremos é permitir que o programa “imprima” apenas valores que sejam strings. Assim, no método *TypeCheckPrintNode* mostrado no programa 11.14 faz-se uma chamada a *TypeCheckExpreNode* que retorna um objeto do tipo *type* e que

descreve o tipo e a dimensão do resultado da expressão. Se esse resultado não for do tipo *string* e de dimensão 0 (não é permitido imprimir um array), um erro é sinalizado. O modo como o tipo da expressão é calculado depende do tipo da expressão e será discutido na seção 11.4

Programa 11.14

```

1  public void TypeCheckPrintNode(PrintNode x) throws SemanticException
2  {
3      type t;
4
5      if (x == null) return;
6
7      // t = tipo e dimensão do resultado da expressão
8      t = TypeCheckExpreNode(x.expr);
9
10     // tipo tem que ser string e dimensão tem que ser 0
11     if (t.ty != STRING_TYPE || t.dim != 0)
12         throw new SemanticException(x.position,
13                                     "string expression required");
14 }
```

O próximo comando a ser visto é um *read*, que deveria ser tão simples quanto um *print*, mas algumas verificações adicionais serão necessárias. Vejamos por quê. Primeiro, porque a expressão usada num comando de entrada de dados não pode ser qualquer. Deve ser uma referência a uma “posição de memória”, como uma variável ou posição de um vetor. Assim, os comandos

```

read a;
read b[10+c];
read c[x].d.e[k];
```

são válidos, ao contrário dos seguintes, que são proibidos:

```

read a + b;
read c(d, e);
```

O analisador sintático nos auxilia ao distinguir os comandos proibidos. Para tal, o comando de leitura é definido na gramática X^{++} como *read lvalue*, o que elimina alguns tipos de expressões como o do penúltimo exemplo que acabamos de mostrar. Não elimina, porém, o último exemplo, uma vez que na construção da subárvore correspondente a *lvalue* podemos ter como raiz os seguintes tipos de nós: um *DotNode*, um *IndexNode*, um *VarNode* ou um *CallNode*. Este quarto corresponde exatamente ao último exemplo e deve ser evitado. Por isso, o método que analisa o *ReadNode* deve verificar qual é o tipo do seu nó filho e acusar um erro caso seja um *CallNode*.

Deve-se verificar, ainda, se a leitura não está sendo feita numa variável ilegal. Por fortuna, não temos em X^{++} variáveis “só de leitura” como as variáveis *final* em Java. Porém, existe uma leitura que é ilegal na variável *this*. Isso não significa que essa variável não possa ser usada na expressão do *read*. Por exemplo, os seguintes comandos são perfeitamente válidos:

```
read this.x;
read this.y[this.k];
```

Mas é ilegal o comando

```
read this;
```

Além disso, somente dados inteiros ou strings podem ser lidos e isso também tem que ser verificado.

No programa 11.15, vemos inicialmente a verificação do tipo de nó que compõe o comando. Se não for um dos três tipos permitidos, então um erro ocorrerá. Em seguida, caso seja um nó do tipo *VarNode*, verifica-se se a variável sendo lida é a variável local número 0. Se for, trata-se de *this* e um erro ocorrerá. Note que procuramos a variável na tabela de símbolos, e se esta não for encontrada (*v == null*), nada será feito pois esse erro será tratado no método que analisa aquele nó *VarNode*. Poderíamos também verificar simplesmente se o nome da variável é *this*, em vez de procurá-la na tabela. Por fim é verificado se o tipo da expressão resultante é inteiro ou string e se a dimensão é 0, pois não se pode ler um array.

Programa 11.15

```

1  public void TypeCheckReadNode(ReadNode x) throws SemanticException
2  {
3      type t;
4
5      if (x == null) return;
6
7      // verifica se o nó filho tem um tipo válido
8      if ( ! (x.expr instanceof DotNode ||
9              x.expr instanceof IndexNode ||
10             x.expr instanceof VarNode) )
11          throw new SemanticException(x.position,
12                             "Invalid expression in read statement");
13      // verifica se é uma atribuição para "this"
14      if ( x.expr instanceof VarNode )
15      {
16          EntryVar v = Curtable.varFind(x.expr.position.image);
17          if ( v != null && v.localcount == 0) // é a variável local 0?
18          {
19              throw new SemanticException(
20                  x.position, "Reading into variable " +
21                  " \"this\" is not legal");
22          }
23      }
24      // verifica se o tipo é string ou int
25      t = TypeCheckExpreNode(x.expr);
26      if (t.ty != STRING_TYPE && t.ty != INT_TYPE )
27          throw new SemanticException(
28              x.position, "Invalid type. Must be int or string");
29
30      // verifica se não é array
```

```

31     if (t.dim != 0 )
32         throw new SemanticException(x.position, "Cannot read array");
33 }

```

O comando `return` é bastante simples e muito parecido com um `print`. A única novidade é que o tipo da expressão não é conhecido *a priori* e deve coincidir com o tipo de retorno declarado para o método. Assim, vemos no programa 11.16 que o tipo da expressão é comparado com o conteúdo da variável `Returntype`. Se o resultado da expressão é `null`, isso indica que não há uma expressão no comando e o tipo em `Returntype` deve ser também `null`, indicando tratar-se de um construtor.

Se a análise da expressão retorna um tipo, então verifica-se se o método corrente é um construtor e, caso seja, um erro é sinalizado. Caso nem a expressão nem o tipo de retorno sejam nulos, verifica-se se o tipo e a dimensão são iguais. Note que aqui há um pequeno inconveniente. Suponhamos que a expressão resulte em um objeto de um tipo `classA`, que o tipo de retorno do método seja `classB` e que `classA` seja um descendente direto ou indireto de `classB`. Nesse caso, um erro seria indicado e o programador deveria fazer a conversão para o tipo correto. O leitor mais atento deve estar se perguntando como isso poderia ser feito já que não há um operador de coerção em `X++`. Veremos, em seguida, o comando de atribuição e como a coerção é feita.

Programa 11.16

```

1 public void TypeCheckReturnNode(ReturnNode x) throws SemanticException
2 {
3     type t;
4
5     if (x == null) return;
6     // t = tipo e dimensão do resultado da expressão
7     t = TypeCheckExpreNode(x.expr);
8
9     // verifica se é igual ao tipo do método corrente
10    if (t == null)
11    {   // t == null não tem expressão no return
12        if (Returntype == null)
13            return;
14        else // se Returntype != null e é um método, então ERRO
15            throw new SemanticException(x.position,
16                                         "Return expression required");
17    }
18    else
19    {
20        if (Returntype == null) // retorno num construtor, ERRO
21            throw new SemanticException(x.position,
22                                         "Constructor cannot return a value");
23    }
24
25    // compara tipo e dimensão
26    if (t.ty != Returntype.ty || t.dim != Returntype.dim )
27        throw new SemanticException(x.position, "Invalid return type");
28 }

```

O comando de atribuição é semelhante ao comando `read` visto anteriormente, pois um valor deve ser armazenado numa posição de memória. No caso da atribuição, o tipo da expressão pode ser qualquer um, porém as demais regras do comando `read` ainda valem. Assim, o início do programa 11.17 até a linha 25 já é conhecido da análise do `read`. Depois, deve-se verificar se o tipo do lado esquerdo da atribuição é compatível com o valor da expressão do lado direito.

Programa 11.17

```

1  public void TypeCheckAtribNode(AtribNode x) throws SemanticException
2  {
3      type t1, t2;
4      EntryVar v;
5
6
7      if (x == null) return;
8
9      // verifica se o nó filho tem um tipo válido
10     if ( ! (x.expr1 instanceof DotNode ||
11             x.expr1 instanceof IndexNode ||
12             x.expr1 instanceof VarNode) )
13         throw new SemanticException(x.position,
14                                     "Invalid left side of assignment");
15
16     // verifica se é uma atribuição para "this"
17     if ( x.expr instanceof VarNode )
18     {
19         EntryVar v = Curtable.varFind(x.expr.position.image);
20         if ( v != null && v.localcount == 0 ) // é a variável local 0?
21         {
22             throw new SemanticException(x.position,
23                             "Assigning to variable \"this\" is not legal");
24         }
25     }
26
27     t1 = TypeCheckExpreNode(x.expr1);
28     t2 = TypeCheckExpreNode(x.expr2);
29
30     // verifica tipos das expressões
31     // verifica dimensões
32     if ( t1.dim != t2.dim )
33         throw new SemanticException(x.position,
34                                     "Invalid dimensions in assignment");
35
36     // verifica se lado esquerdo é uma classe e direito é null, OK
37     if (t1.ty instanceof EntryClass && t2.ty == NULL_TYPE )
38         return;
39
40     // verifica se t2 é subclasse de t1
41     if ( ! ( isSubClass(t2.ty,t1.ty) || isSubClass(t1.ty,t2.ty) ) )
42         throw new SemanticException(x.position,
43                                     "Incompatible types for assignment ");
44 }
```

Note que dissemos compatível e não iguais. No programa 11.17, atribui-se a t_1 o tipo da expressão do lado esquerdo e a t_2 , o do lado direito. As verificações efetuadas são as seguintes:

- linha 32: verifica-se se as dimensões são iguais. Por exemplo, a primeira atribuição a seguir é perfeitamente legal. A segunda, no entanto, não o é de acordo com esta regra.

```
int[][] x, y;
x = new int[10][10];
y = x[0];
```

- linha 37: se t_1 é um tipo qualquer que representa uma classe (sua entrada na tabela é uma *EntryClass*), e não um tipo simples (sua entrada seria uma *EntrySimple*), e o lado direito resulta na constante null, então a atribuição é legal. O valor de t_2 é comparado com a entrada que inserimos na tabela para representar um tipo especial para o valor null, no início desta fase da análise semântica. Um exemplo válido e um não válido seriam, respectivamente:

```
MyClass x;
x = null;
int k;
k = null;
```

- linha 41: verifica-se se o tipo em t_1 é subclasse do tipo em t_2 ou vice-versa. Os dois casos são aceitos como legais.

Nesse último caso, vale uma explicação sobre a semântica que queremos dar a este comando. Primeiro, se o tipo do lado direito for uma subclasse do tipo do lado esquerdo, então a atribuição é feita naturalmente, sem problemas. No caso inverso, antes de fazer a atribuição, o sistema de execução faz por conta própria a coerção dos tipos. Obviamente, se a coerção não puder ser feita, um erro de execução ocorre. Vejamos um exemplo:

```
class A extends C
{
    int methA()
    {
        C var1;
        A var2;
        B var3;
        var1 = new A();
        var2 = var1;
        var3 = var1;
    }
}

class B extends C
{
```

```

    ...
}

class C
{
    ...
}

```

Na primeira atribuição, var1 é de uma superclasse da expressão do lado direito, assim a atribuição é feita sem problemas. Na segunda, a expressão esquerda é do tipo A, que é uma subclasse da classe de var1 e, assim, a atribuição é feita mediante da coerção da expressão direita para o tipo encontrado no lado esquerdo. Como o tipo real de var1 é o mesmo, nenhum problema ocorre. Já na terceira atribuição, temos um erro, pois tentamos fazer a coerção de um objeto A para a classe B, o que não é possível. Esse erro não é detectado pelo analisador semântico e resultará num erro de execução.

E para tipos simples, qual é a regra? Bem, o método *isSubclass* mostrado no programa 11.18 trata também de maneira correta este caso. Quando um dos argumentos passados para ele é um tipo simples, ele retorna verdadeiro somente se o outro é exatamente o mesmo tipo. Assim, atribuições que envolvem tipos simples têm que ter em ambos os lados exatamente o mesmo tipo de expressão.

Programa 11.18

```

1  protected boolean isSubClass(EntryTable t1, EntryTable t2)
2  {
3      // verifica se são o mesmo tipo (vale para tipos simples)
4      if ( t1 == t2 )
5          return true;
6
7      // verifica se são classes
8      if ( ! ( t1 instanceof EntryClass && t2 instanceof EntryClass ) )
9          return false;
10
11     // procura t2 nas superclasses de t1
12     for ( EntryClass p = ((EntryClass)t1).parent; p != null; p = p.parent)
13         if ( p == t2 ) return true;
14     return false;
15 }

```

O comando *super(...)* é usado no construtor de uma determinada classe para fazer-se uma chamada ao construtor da sua superclasse. As verificações que devem ser feitas para validar o comando são:

- verificar se o comando está sendo usado dentro de um construtor;
- verificar se ele é o primeiro comando do construtor;
- verificar se a classe em que o construtor está possui uma superclasse;

- verificar se a superclasse possui um construtor que case com a chamada sendo analisada (número e tipo de parâmetros).

Vejamos, então, como essas verificações são implementadas no analisador semântico (Programa 11.19). A primeira condição é verificada por meio do valor da variável *Returntype*, que, como visto, é *null* quando um construtor está sendo analisado. Em seguida, utilizamos a variável *cansuper*, à qual foi atribuído o valor *true* no início da análise do método, se o primeiro comando é uma chamada *super*. Então, se essa variável é verdadeira, estamos analisando o primeiro comando do método, portanto o comando é legal. Note que logo após essa verificação, o valor da variável é mudado de modo que qualquer outro comando *super* que apareça neste construtor não seja aceito.

Programa 11.19

```

1  public void TypeCheckSuperNode(SuperNode x) throws SemanticException
2  {
3      type t;
4
5      if (x == null) return;
6      if ( Returntype != null)
7          throw new SemanticException(x.position,
8              "super is only allowed in constructors");
9
10     if ( ! cansuper )
11         throw new SemanticException(x.position,
12             "super must be first statement in the constructor");
13
14     cansuper = false; // não permite outro super
15
16     // p aponta para a entrada da superclasse da classe corrente
17     EntryClass p = Curtable.levelup.parent;
18     if (p == null)
19         throw new SemanticException(x.position,
20             "No superclass for this class");
21
22     // t.ty possui um EntryRec com os tipos dos parâmetros
23     t = TypeCheckExpreListNode(x.args);
24     // procura o construtor na tabela da superclasse
25     EntryMethod m =
26         p.nested.methodFindInclass("constructor", (EntryRec) t.ty);
27
28     // se não achou, ERRO
29     if ( m == null )
30         throw new SemanticException(x.position, "Constructor " + p.name +
31             "(" + (t.ty == null ? "" :( (EntryRec)t.ty).toStr()) +
32             ") not found");
33     // indica que existe chamada a super no método
34     CurMethod.hassuper = true;
35 }
```

Em seguida, toma-se a entrada na tabela de símbolos da superclasse da classe corrente. Isso é feito na linha 17. Se esse valor é nulo, então a classe corrente não possui uma superclasse e, portanto, o comando `super` é ilegal.

Na linha 23 é feita a análise da lista de argumentos da chamada. Como resultado, obtém-se um objeto do tipo `EntryRec`, que, como vimos, é uma lista de tipos, que descreve os tipos de cada expressão utilizada na chamada. Com base neste `EntryRec` é feita uma busca na superclasse (somente na superclasse) por um construtor que case com os tipos calculados dos argumentos. Caso esse construtor não seja encontrado, um erro foi localizado. A mensagem para esse erro é algo do tipo

```
line 10 column 3: Constructor MyClass(int[][], string) not found.
```

O método `EntryRec.toStr()` utilizado na linha 31 do programa 11.19 devolve um string que representa a lista de tipos representada pelo objeto `EntryRec`. Esse método é bastante simples e dispensa explicações.

Um analisador semântico mais completo poderia efetuar outros tipos de verificação em relação ao comando `super`. Em particular, poderia restringir o uso das variáveis de instância da classe corrente na chamada, pois estas ainda não foram inicializadas. Porém não o faremos no nosso compilador, mesmo porque em outros comandos da linguagem não se verifica se uma variável foi inicializada antes de ser usada, pois isso exigiria que nosso analisador semântico realizasse análise de fluxo de controle e de dados, o que aumentaria sensivelmente sua complexidade. Cada variável de instância recebe um valor-padrão, de acordo com o seu tipo:

- int: valor-padrão é 0;
- string: valor-padrão é `null`;
- objetos e arrays: valor-padrão é `null`.

Já as variáveis locais devem ser explicitamente inicializadas pelo programador. Caso isso não seja feito e tente-se usar o valor da variável não inicializada, um erro de execução ocorre.

Vejamos, agora, a análise de um comando `for`. O método que trata de um `ForNode` é mostrado no programa 11.20. Não há nenhuma novidade, apenas as chamadas para cada um dos filhos, verificando se o tipo da expressão de controle é do tipo correto, ou seja, um `int`. Cada uma dessas chamadas é feita de forma independente, dentro de um `try`, de modo que um erro em uma delas não interrompa a análise semântica do comando. Caso um erro apareça, a mensagem correspondente é emitida, mas a análise continua para as demais partes do comando. Antes da última chamada ao método `TypeCheckStatementNode`, a variável `nesting` é incrementada indicando que o nível de aninhamento do comando dentro do `for` é superior ao do próprio `for` em uma unidade. Essa informação é usada no tratamento do comando `break` que veremos a seguir. O tratamento de um `IfNode` segue o mesmo estilo do tratamento do `for`, por isso não precisa ser apresentado aqui.

Programa 11.20

```
1 public void TypeCheckForNode(ForNode x)
2 {
3     type t;
4
5     if (x == null) return;
6     // analisa inicialização
7     try
8     {
9         TypeCheckStatementNode(x.init);
10    }
11    catch (SemanticException e)
12    {
13        System.out.println(e.getMessage());
14        foundSemanticError++;
15    }
16    // analisa expressão de controle
17    try
18    {
19        t = TypeCheckExprNode(x.expr);
20        if (t.ty != INT_TYPE || t.dim != 0)
21            throw new SemanticException(x.expr.position,
22                                         "Integer expression expected");
23    }
24    catch (SemanticException e)
25    {
26        System.out.println(e.getMessage());
27        foundSemanticError++;
28    }
29    // analisa expressão de incremento
30    try
31    {
32        TypeCheckStatementNode(x.incr);
33    }
34    catch (SemanticException e)
35    {
36        System.out.println(e.getMessage());
37        foundSemanticError++;
38    }
39    // analisa comando a ser repetido
40    try
41    {
42        nesting++; // incrementa o aninhamento
43        TypeCheckStatementNode(x.stat);
44    }
45    catch (SemanticException e)
46    {
47        System.out.println(e.getMessage());
48        foundSemanticError++;
49    }
50    nesting--; // decrementa o aninhamento
51 }
```

O comando `break` em X^{++} é utilizado para interromper a execução de um `for` e, por isso, não faz sentido ser utilizado fora de tal comando. Essa é a única verificação feita para esse comando, como mostra o programa 11.21.

```

1  public void TypeCheckBreakNode(BreakNode x) throws SemanticException
2  {
3      if (x == null) return;
4      // verifica se está dentro de um for. Se não, ERRO
5      if (nesting <= 0 )
6          throw new SemanticException(x.position,
7                               "break not in a for statement");
8 }
```

E, com isso, terminamos a parte de análise dos comandos. Vejamos na seção 11.4 como efetuar a análise de cada tipo de expressão de X^{++} .

11.4 Análise de expressões

Como dissemos anteriormente, uma expressão em X^{++} pode ser uma simples constante ou variável, ou uma expressão composta de diversas expressões mais simples. O tipo de uma expressão é calculado com base nos tipos das suas subexpressões.

Para as constantes, obter o seu tipo é trivial, como mostra o programa 11.22. O próprio tipo do nó da árvore sintática determina qual o tipo da constante. Somente no caso de uma constante inteira é necessária uma verificação extra que não é feita na análise sintática. Uma constante inteira é definida no analisador sintático apenas como uma seqüência de dígitos, porém se for demasiado grande, o número não será válido. Para verificar se é uma constante válida, utilizamos uma chamada ao método `Integer.parseInt` da API Java. Se tal chamada não lançar uma exceção, a constante é válida. Note que utilizando esse método para validar a constante, estamos assumindo que o intervalo válido para uma constante inteira em X^{++} é o mesmo utilizado em Java. A constante `null` tem sempre o tipo `NULL_TYPE` inserido no nível mais alto da tabela de símbolos.

```

1  public type TypeCheckIntConstNode(IntConstNode x)
2      throws SemanticException
3  {
4      int k;
5
6      if (x == null) return null;
7
8      // tenta transformar imagem em número inteiro
9      try
10     {
11         k = Integer.parseInt(x.position.image);
12     }
13     catch(NumberFormatException e)
```

```

14     { // se deu erro, formato é inválido
15         // (possivelmente fora dos limites)
16         throw new SemanticException(x.position, "Invalid int constant");
17     }
18     return new type(INT_TYPE, 0);
19 }

20

21
22 // ----- Constante string -----
23
24 public type TypeCheckStringConstNode(StringConstNode x)
25 {
26     if (x == null) return null;
27     return new type( STRING_TYPE, 0);
28 }

29
30 // ----- Constante null -----
31
32 public type TypeCheckNullConstNode(NullConstNode x)
33 {
34     if (x == null) return null;
35     return new type( NULL_TYPE, 0);
36 }

```

Para uma variável também não é difícil calcular seu tipo. Basta consultar a tabela de símbolos. Isto é feito no programa 11.23, que utiliza o conhecido método *Symtable.varFind* para procurar a variável cujo nome está no nó *VarNode* sendo analisado. Se a variável é encontrada, o seu tipo é extraído da tabela de símbolos.

Programa 11.23

```

1 public type TypeCheckVarNode(VarNode x) throws SemanticException
2 {
3     EntryVar p;
4
5     if (x == null) return null;
6
7     // procura variável na tabela
8     p = Curtable.varFind(x.position.image);
9
10    // se não achou, ERRO
11    if (p == null)
12        throw new SemanticException(x.position, "Variable " +
13                                  x.position.image + " not found");
14    return new type(p.type, p.dim);
15 }

```

Esses foram os nós que representam os tipos mais simples de expressões em *X⁺⁺*. Vejamos, agora, o tratamento de um nó um pouco mais sofisticado, o *CallNode*, que representa uma chamada de método. O método que trata desse tipo de nó é mostrado no programa 11.24. Ele, inicialmente, calcula o tipo da expressão que está

no seu primeiro filho e que representa a expressão à esquerda do “.” na chamada do método. Uma vez calculado o tipo são feitas duas verificações quanto à sua validade:

- se a dimensão é maior que zero. Caso seja, um erro foi encontrado, pois um array não pode ter um método associado a ele;
- se o tipo calculado corresponde a uma classe. Caso não seja, um erro foi encontrado, pois tipos simples não podem ter métodos.

Em seguida são calculados os tipos para cada um dos argumentos, por meio da chamada a *TypeCheckExpreListNode* na linha 23. Essa chamada retorna um objeto do tipo *type*, cuja variável *ty* é um *EntryNode* com os tipos calculados para os argumentos da chamada. Utilizando o nome do método, que é um dos filhos do nó corrente, e a lista de tipos dos argumentos, realiza-se uma busca na tabela de símbolos para achar o método adequado. É claro que a busca deve ser feita na classe adequada, ou seja, na classe que foi calculada no início e que representa a expressão à esquerda do ponto na chamada do método. Por exemplo, na seguinte expressão

```
x.y.z[16].myMethod(10, -18);
```

suponhamos que a expressão *x.y.z[16]* leve a um objeto do tipo *ClassA*. Então, devemos procurar na tabela de símbolos por um método chamado *myMethod*, com dois parâmetros inteiros, e essa busca deve ser feita na classe *ClassA* (e suas classes ancestrais). Se tal método não for encontrado, então um erro ocorreu, como mostrado na linha 31. Caso contrário, a entrada do método na tabela de símbolos diz-nos qual é o tipo retornado pela chamada. Esse tipo é, então, retornado pelo método que trata o *CallNode* (linha 36).

Programa 11.24

```

1  public type TypeCheckCallNode(CallNode x) throws SemanticException
2  {
3      EntryClass c;
4      EntryMethod m;
5      type t1, t2;
6
7      if (x == null) return null;
8
9      // calcula tipo do primeiro filho
10     t1 = TypeCheckExpreNode(x.expr);
11
12     // se for array, ERRO
13     if (t1.dim > 0)
14         throw new SemanticException(x.position,
15             "Arrays do not have methods");
16
17     // se não for uma classe, ERRO
18     if (!(t1.ty instanceof EntryClass))
19         throw new SemanticException(x.position, "Type " + t1.ty.name +
20             " does not have methods");
21

```

```

22 // pega tipos dos argumentos
23 t2 = TypeCheckExpreListNode(x.args);
24
25
26 // procura o método desejado na classe t1.ty
27 c = (EntryClass) t1.ty;
28 m = c.nested.methodFind(x.meth.image, (EntryRec) t2.ty);
29
30 // se não achou, ERRO
31 if (m == null)
32     throw new SemanticException(x.position, "Method " + x.meth.image
33             + "(" + (t2.ty == null ? "" : ((EntryRec) t2.ty).toString())
34             + ") not found in class " + c.name);
35
36 return new type(m.type, m.dim);
37 }

```

Um comentário importante sobre *Symtable.methodFind*, que acabamos de utilizar, deve ser feito. Esse método é chamado sempre que precisamos utilizar, no programa *X⁺⁺*, um método ou construtor. Por exemplo, na expressão *a = b.myMethod(c, d)*, supondo que *c* e *d* sejam variáveis dos tipos *ClassC* e *ClassD*, respectivamente, devemos “calcular” o tipo de *b* (suponhamos que seja *ClassB*) e, então, procurar na tabela de símbolos o método *myMethod(ClassC, ClassD)*, a partir da classe *ClassB* e em todos os seus ancestrais. Olhando *Symtable.methodFind* no programa 11.25, vemos que ele não difere muito de outros métodos de pesquisa na tabela de símbolos como *varFind*, por exemplo. Note, porém, que quando uma entrada do tipo *EntryMethod*, com o mesmo nome do método procurado, é encontrada na tabela de símbolos faz-se, então, uma comparação dos tipos dos argumentos (armazenados em *r*) e os tipos dos parâmetros formais armazenados na entrada encontrada. Essa comparação é feita por meio da chamada a *EntryRec.equals*, que compara se os tipos e dimensões de cada elemento da lista de tipos casam exatamente.

Isto significa que para que um método utilizado numa chamada seja encontrado, os tipos utilizados como argumentos devem ser exatamente iguais aos tipos declarados. Chamadas utilizando, por exemplo, objetos de uma subclasse da classe utilizada na declaração do método resultariam em erro. Num compilador real, tal característica seria totalmente indesejada. O compilador Java, por exemplo, possui algumas regras para determinar o “casamento” entre tipos dos argumentos e dos parâmetros formais, até mesmo para o caso em que mais de um método combina com a chamada¹. No nosso caso, aceitaremos a restrição de trabalhar apenas com casamentos exatos.

Programa 11.25

```

1 public EntryMethod methodFind(String x, EntryRec r)
2 {
3     EntryTable p = top;
4     EntryClass q;
5
6     while (p != null)

```

¹Para mais detalhes sobre as regras utilizadas em Java ver *J. Gosling et al., “The Java language specification, 2nd ed., pp. 347-350, Addison-Wesley, 2000*

```

7      {
8          if ( p instanceof EntryMethod && p.name.equals(x) )
9          {
10             EntryMethod t = (EntryMethod) p;
11             if (t.param == null)
12             {
13                 if (r == null)
14                     return t;
15             }
16             else
17             {
18                 if (t.param.equals(r))
19                     return t;
20             }
21         }
22         p = p.next;
23     }
24     q = levelup;
25     if ( q.parent == null )
26         return null;
27     return q.parent.nested.methodFind(x, r);
28 }

```

A análise da utilização de uma variável de instância representada por meio de um nó *DotNode* é bastante semelhante à chamada de método que acabamos de ver. Por isso, não será necessário apresentar aqui o método que efetua tal análise. A única diferença é que, após calcular o tipo da expressão à esquerda do ponto e verificar se é um tipo válido (ou seja, uma classe), faz-se a procura por uma variável de instância nessa classe. O tipo retornado pelo método é obtido da entrada da variável na tabela de símbolos. Obviamente se a variável não for encontrada na classe desejada nem em alguma classe ancestral, um erro ocorreu.

Na análise de uma chamada de método, utilizamos o método *TypeCheckExpreListNode* que analisa e calcula os tipos de uma lista de expressões, usadas como argumentos da chamada. Este método é mostrado no programa 11.26. O que ele faz é chamar o método *TypeCheckExpreNode* que trata da primeira expressão da lista, retornando o seu tipo que é armazenado na variável *t* (linha 12). Em seguida, o próprio método é chamado recursivamente para tratar do restante da lista (linha 22). Depois é criado um objeto do tipo *EntryRec* que contém o tipo e a dimensão da primeira expressão e que aponta para o outro *EntryRec* devolvido na chamada recursiva da linha 22. Esse objeto é, então, encapsulado num objeto do tipo *type* e devolvido. Quando o valor utilizado na chamada de *TypeCheckExpreListNode* é *null*, então se chegou ao fim da lista de expressões. Neste caso é retornado um objeto *type* com tipo *null* e dimensão 0. Isto ocorre até mesmo quando a chamada de método é feita sem nenhum argumento. No programa 11.24, o valor devolvido *null* é utilizado para pesquisar a tabela de símbolos: caso um método sem parâmetros exista, será encontrado nesta pesquisa.

Programa 11.26

```

1 public type TypeCheckExpreListNode(ListNode x)
2 {
3     type t, t1;
4     EntryRec r;
5     int n;
6
7     if (x == null) return new type (null,0);
8
9     try
10    {
11         // pega tipo do primeiro nó da lista
12         t = TypeCheckExpreNode( (ExpreNode) x.node);
13     }
14     catch (SemanticException e)
15    {
16         System.out.println(e.getMessage());
17         foundSemanticError++;
18         t = new type(NULL_TYPE, 0);
19     }
20
21     // pega tipo do restante da lista. t1.ty contém um EntryRec
22     t1 = TypeCheckExpreListNode(x.next);
23
24     // n = tamanho da lista em t1
25     n = (t1.ty == null) ? 0 : ( (EntryRec) t1.ty).cont;
26
27     // cria novo EntryRec com t.ty como 1º. elemento
28     r = new EntryRec(t.ty, t.dim, n+1, (EntryRec) t1.ty);
29
30     // cria type com r como variável ty
31     t = new type(r, 0);
32     return t;
33 }
```

*

A análise da operação de indexação de um array é apresentada no programa 11.27 e apresenta algumas semelhanças com a análise da chamada de métodos. Inicialmente, calcula-se a expressão do primeiro filho do *IndexNode* sendo analisado. O resultado dessa expressão deve ser um array, do qual se deseja utilizar um elemento. Por isso, o tipo retornado na chamada da linha 9 deve ter dimensão maior que zero. Além disso, deve-se verificar o tipo da expressão usada como índice, que deve ser do tipo inteiro. Se essas duas condições são corretamente verificadas, então o tipo da expressão a ser retornado é facilmente calculado. É o mesmo tipo da expressão calculada inicialmente com a dimensão decrementada de uma unidade.

Programa 11.27

```

1 public type TypeCheckIndexNode(IndexNode x) throws SemanticException
2 {
3     EntryClass c;
4     type t1, t2;
5 
```



```

16 // t.ty recebe a lista de tipos dos argumentos
17 t = TypeCheckExpreListNode(x.args);
18
19 // procura um construtor com essa assinatura
20 Symtable s = ((EntryClass)c).nested;
21 p = s.methodFindInclass("constructor", (EntryRec) t.ty);
22 // se não achou, ERRO
23 if (p == null)
24     throw new SemanticException(x.position, "Constructor " +
25         x.name.image + "(" + (t.ty == null ? "" :
26             (EntryRec) t.ty).toString() + ") not found");
27
28 // retorna c como tipo, dimensão = 0, local = -1 (não local)
29 t = new type(c, 0);
30 return t;
31 }

```

A alocação de arrays é semelhante. Inicialmente é feita a busca na tabela de símbolos da classe ou tipo básico que compõe cada elemento do array. Em seguida, para cada expressão que está na lista de expressões no filho *dims* do nó corrente, verifica-se se o seu tipo é um inteiro, uma vez que essas expressões devem representar o número de elementos em cada uma das dimensões do array. Se todas as expressões forem do tipo correto, então o método retornará como tipo da expressão de alocação, o tipo recuperado da tabela de símbolos com o número de dimensões igual ao número de expressões no filho *dim* do nó corrente. O método é mostrado no programa 11.29

Programa 11.29

```

1 public type TypeCheckNewArrayNode(NewArrayNode x)
2                     throws SemanticException
3 {
4     type t;
5     EntryTable c;
6     ListNode p;
7     ExpreNode q;
8     int k;
9
10    if (x == null) return null;
11    // procura o tipo da qual se deseja criar um array
12    c = Curtable.classFindUp(x.name.image);
13    // se não achou, ERRO
14    if (c == null)
15        throw new SemanticException(x.position, "Type " + x.name.image +
16            " not found");
17
18    // para cada expressão das dimensões, verifica se tipo é int
19    for (k = 0, p = x.dims; p != null; p = p.next)
20    {
21        t = TypeCheckExpreNode( (ExpreNode) p.node);
22        if (t.ty != INT_TYPE || t.dim != 0)
23            throw new SemanticException(p.position,
24                "Invalid expression for an array dimension");

```

```

25         k++;
26     }
27     return new type(c, k);
28 }

```

Vejamos, agora, como é feita a análise de uma operação unária, ou seja, de um nó do tipo *UnaryNode*. Tal nó representa ou uma operação de negação aritmética, ou uma negação lógica. Em ambos os casos, a expressão sobre a qual o operador unário é aplicado deve ser do tipo inteira. Assim, esta é a única verificação realizada. Se o tipo da expressão é inteira e sem dimensão, tudo está correto. Caso contrário, um erro é indicado. No programa 11.30 vemos o método que trata deste tipo de nó.

```

1 public type TypeCheckUnaryNode(UnaryNode x) Programa 11.30
2             throws SemanticException
3 {
4     type t;
5
6     if (x == null) return null;
7
8     t = TypeCheckExpreNode(x.expr);
9
10    // se dimensão > 0, ERRO
11    if (t.dim > 0)
12        throw new SemanticException(x.position, "Can not use unary " +
13                                     x.position.image + " for arrays");
14
15    // só int é aceito
16    if (t.ty != INT_TYPE)
17        throw new SemanticException(x.position,
18                                     "Incompatible type for unary " + x.position.image);
19
20    return new type(INT_TYPE, 0);
21 }

```

O mesmo tipo de verificação deve ser feito para operações binárias como somas, multiplicações ou comparações de valores. Iniciaremos com as somas e subtrações, representadas num nó *AddNode*. Este nó possui como filhos as duas subexpressões e o operador a ser aplicado a elas. Segundo a construção feita pelo analisador sintático, este operador pode ser de soma ou de subtração. Cabe ao método mostrado no programa 11.31 verificar se os tipos das expressões e o operador combinam e determinar o tipo de retorno. A verificação feita é bastante simples. São calculados os tipos das subexpressões e é recuperado o operador que pode ser *PLUS* ou *MINUS* (constantes definidas no analisador sintático). Em seguida, verifica-se se alguma das subexpressões possui dimensão maior que zero, o que é um erro. A partir da linha 21 são contados quantos dos operandos são do tipo inteiro e quantos são do tipo string. A variável *i* é usada para contar os inteiros e a *j*, para contar os do tipo string. Assim, o nó é válido se

- $i == 2, j == 0$ – temos a soma ou subtração de dois inteiros, e o resultado é um inteiro;
- $i == 1, j == 1$ – temos um string somado a ou subtraído de um inteiro (ou vice-versa). A operação só é válida se o operador for de soma, representando a concatenação do string com a representação textual do número (como em Java). O resultado é um string;
- $i == 0, j == 2$ – temos dois strings. Só é válida a concatenação (soma) entre eles. O resultado é um string.

O nosso compilador não aceita outras operações de soma/subtração. Por exemplo, não aceita, como em Java, a soma de um objeto de qualquer tipo com um string, o que representa a concatenação do string com o resultado da chamada ao método `toString()` sobre aquele objeto. Em X^{++} , o programador deve explicitamente extrair a representação do objeto na forma de um string, para, então, poder utilizá-lo em uma concatenação.

Programa 11.31

```

1  public type TypeCheckAddNode(AddNode x)
2      throws SemanticException
3  {
4      type t1, t2;
5      int op; // operação
6      int i, j;
7
8      if (x == null) return null;
9
10     op = x.position.kind;
11     t1 = TypeCheckExpreNode(x.expr1);
12     t2 = TypeCheckExpreNode(x.expr2);
13
14     // se dimensão > 0, ERRO
15     if (t1.dim > 0 || t2.dim > 0)
16         throw new SemanticException(x.position, "Can not use " +
17                                     x.position.image + " for arrays");
18
19     i = j = 0;
20
21     if (t1.ty == INT_TYPE)
22         i++;
23     else
24     if (t1.ty == STRING_TYPE)
25         j++;
26
27     if (t2.ty == INT_TYPE)
28         i++;
29     else
30     if (t2.ty == STRING_TYPE)
31         j++;
32
33     // dois operadores inteiro, OK

```

```

34     if (i == 2)
35         return new type(INT_TYPE, 0);
36
37     // um inteiro e um string. Só pode somar
38     if ( op == langXConstants.PLUS && i+j == 2)
39         return new type(STRING_TYPE, 0);
40
41     throw new SemanticException(x.position, "Invalid types for " +
42           x.position.image);
43 }

```

Não abordaremos o método que trata de multiplicação ou divisão, pois é parecido com o método do programa 11.31, porém mais simples, uma vez que somente são aceitas expressões do tipo inteiro. Passaremos, então, às expressões relacionais, tratadas pelo método do programa 11.32. Neste método são feitas diversas verificações que tentaremos explicar a seguir:

- linha 14: duas subexpressões inteiras podem ser comparadas de qualquer forma (qualquer operador). O tipo retornado será sempre inteiro, uma vez que X^{++} não possui o tipo booleano;
- linha 18: as subexpressões devem ter as mesmas dimensões. Isso porque objetos e arrays só podem ser comparados quanto à sua igualdade (operador ==) ou desigualdade (operador !=) e, por isso, objetos com dimensões diferentes são obviamente diferentes e, portanto, não devem ser comparados;
- linha 23: arrays só podem ser comparados quanto à sua igualdade e à desigualdade, e outros operadores geram um erro;
- linha 30: dois objetos podem ser comparados se forem da mesma classe ou um for subclasse do outro. O método *isSubClass* retorna verdadeiro também quando os dois tipos são iguais, ou seja, se tivermos dois strings sendo comparados, eles serão verificados aqui. Para eles somente comparação de igualdade ou desigualdade pode ser feita;
- linha 35: este último caso verifica se a comparação é entre um objeto e o tipo \$NULL\$ que representa a constante null. Nesse caso também somente igualdade e desigualdade podem ser comparadas.

Programa 11.32

```

1  public type TypeCheckRelationalNode(RelationalNode x)
2          throws SemanticException
3  {
4      type t1, t2;
5      int op; // operação
6
7      if (x == null) return null;
8
9      op = x.position.kind;
10     t1 = TypeCheckExpreNode(x.expr1);

```

```

11     t2 = TypeCheckExpreNode(x.expr2);
12
13     // se ambos são int, retorna OK
14     if ( t1.ty == INT_TYPE && t2.ty == INT_TYPE )
15         return new type(INT_TYPE, 0);
16
17     // se a dimensão é diferente, ERRO
18     if ( t1.dim != t2.dim )
19         throw new SemanticException(x.position,
20             "Can not compare objects with different dimensions");
21
22     // se dimensão > 0, só pode comparar igualdade
23     if ( t1.dim > 0 && op != langXConstants.EQ &&
24         op != langXConstants.NEQ )
25         throw new SemanticException(x.position, "Can not use " +
26             x.position.image + " for arrays");
27
28     // se dois são objetos do mesmo tipo, pode comparar igualdade
29     // isso inclui dois strings
30     if ( ( isSubClass(t2.ty,t1.ty) || isSubClass(t1.ty,t2.ty) ) &&
31         (op == langXConstants.NEQ || op == langXConstants.EQ) )
32         return new type(INT_TYPE, 0);
33
34     // se um é objeto e outro, null, pode comparar igualdade
35     if ( ( ( t1.ty instanceof EntryClass && t2.ty == NULL_TYPE) ||
36             ( t2.ty instanceof EntryClass && t1.ty == NULL_TYPE) ) &&
37             (op == langXConstants.NEQ || op == langXConstants.EQ) )
38         return new type(INT_TYPE, 0);
39
40     throw new SemanticException(x.position, "Invalid types for " +
41             x.position.image);
42 }

```

11.5 Arquivos-fonte do compilador

No diretório *cap11/semanaliysis* foram adicionados os arquivos *TypeCheck.java* e *type.java*, que contêm, respectivamente, a classe *TypeCheck* descrita neste capítulo e a classe *type* utilizada para representar um tipo e uma dimensão, geralmente para armazenar o resultado de uma expressão.

No diretório *ssamples*, incluímos o arquivo *bintree-erro-semantico-fase3.x* que possui alguns erros semânticos, detectáveis nesta fase da análise. Conforme mostra a execução a seguir, os erros introduzidos foram:

- o comando `return;` sem nenhum argumento foi utilizado dentro de um método que deveria retornar um inteiro;
- usou-se a comparação `left != 0`, e a variável `left` é um objeto do tipo `bintree`, portanto não poderia ser comparada com um número inteiro;

- no comando print foi feita a concatenação de um objeto com um string, o que é proibido;
- foi utilizada a seguinte expressão `t = new bintree(w, 1);`, e a classe bintree não possui um construtor com dois argumentos.

As mensagens dadas pelo compilador são as seguintes:

```
cap11$ java parser.langX ..ssamples/bintree-erro-semantico-fase3.x
X++ Compiler - Version 1.0 - 2004
Reading from file ..ssamples/bintree-erro-semantico-fase3.x . .
0 Lexical Errors found
0 Syntactic Errors found
Line 32 column 22: Return expression required
Line 59 column 16: Invalid types for !=
Line 83 column 14: Invalid types for +
Line 97 column 8: Constructor bintree(data, int) not found
4 Semantic Errors found (phase 3)
```

Capítulo 12

Geração de Código

Finalmente chegamos à fase de geração de código. Iremos utilizar aqui uma abordagem bastante simples, mas que dará ao leitor uma boa noção de como a geração de código é feita, além da satisfação de poder ver seus programas *X⁺⁺*, compilados com seu próprio compilador, realmente executando.

Utilizaremos como máquina-alvo a Máquina Virtual Java, ou JVM (Java Virtual Machine), como a chamaremos aqui. A utilização da JVM permite que deixemos de lado diversos aspectos complicados da geração de código, como alocação de registradores, alocação de memória para variáveis etc. A JVM possui tratamento “nativo” para os tipos de dados utilizados em *X⁺⁺* (inteiro e string), o que também facilita muito o trabalho da geração de código. Além disso, aspectos de tempo de execução como linkedição dinâmica entre as classes geradas, são de responsabilidade total da JVM.

Iniciaremos o capítulo, então, com uma pequena introdução à JVM, que visa a explicar melhor o seu funcionamento e as suas instruções. Recomenda-se, porém, que o leitor consulte as referências específicas sobre o assunto para obter maiores detalhes. Em seguida, será apresentada também uma introdução ao Jasmin, um programa “montador” que permite gerar código para a JVM a partir de um arquivo texto, contendo a descrição de uma classe e instruções JVM.

E, por fim, apresentaremos a implementação da geração de código e do runtime *X⁺⁺*.

12.1 A Máquina Virtual Java

A especificação completa da JVM pode ser encontrada no livro

The Java Virtual Machine specification, de Tim Lindholm e Frank Yellin
(2^aed., Addison Wesley, 1999),

também disponível on-line em <http://java.sun.com/docs/books/vmspec/index.html>.

Essa especificação apresenta, de maneira completa, como deve ser a implementação de uma JVM, qualquer que seja a plataforma. Ela descreve, entre outras características:

- os tipos de dados tratados pela JVM;
- o conjunto de instruções e como é o comportamento da JVM ao executar cada uma delas;
- o formato de um arquivo executável (um arquivo .class) da JVM;
- como funciona o processo de carregamento, inicialização e linkagem de classes na JVM.

Cada arquivo executável da JVM (também chamado *class file*) descreve uma classe. Neste arquivo existem algumas descrições sobre a classe em si, como: qual é sua superclasse e de qual arquivo-fonte ela foi gerada, a descrição das variáveis de instância, variáveis de classe, métodos e construtores dessa classe etc. As variáveis são descritas em função de:

- seu tipo, que pode ser um tipo básico, suportado pela JVM, como *integer*, *long*, *float*, *double*, *boolean*, ou uma classe que pode ser a própria classe definida no arquivo ou em algum outro arquivo de classe;
- suas propriedades, como *public*, *protected*, *private*, *final*, *static* etc.

Cada método e construtor, além do tipo e da propriedade, necessitam de outras informações para serem completamente especificados. Entre elas, temos:

- os tipos de cada parâmetro esperado pelo método ou construtor;
- o número de variáveis locais que o método utiliza;
- a altura máxima que a pilha de execução (ver adiante) atinge;
- as instruções, que descrevem o comportamento do método;
- uma tabela descrevendo como são tratadas as exceções dentro deste método;
- uma tabela que relaciona cada instrução do método com uma linha no programa-fonte;
- uma tabela que relaciona cada variável local utilizada pelo método com o nome de uma variável local no programa-fonte.

Nem todos esses dados são essenciais na descrição dos métodos ou construtores. Por exemplo, a tabela que relaciona as variáveis locais com nomes no programa-fonte é útil para depuração, mas sua ausência não impede a correta execução de um método.

Quando um programa é executado na JVM, um método é “acionado” mediante sua chamada em um outro método ou mediante sua chamada pela própria JVM, caso ele seja o método principal do programa, ou seja, o método *static public void main*

(`String[]`). Todo método possui uma instrução inicial, ou seja, cada método possui apenas um ponto de entrada. As instruções são executadas na seqüência que aparecem no método, salvo obviamente quando são executadas instruções de desvio, de chamadas de métodos ou que lançam exceções.

Quando recebe o controle de execução, o método possui duas áreas reservadas para si: uma área onde são armazenadas suas variáveis locais e uma área utilizada para executar operações. A primeira é um vetor que as instruções endereçam diretamente utilizando, para isso, o índice de cada elemento do vetor. É nesse vetor que são também armazenados os argumentos passados para o método. No caso de métodos de classe (estáticos), os argumentos ocupam as primeiras posições desse vetor. No caso de um método de instância (não estático), a posição zero é ocupada pela referência ao objeto-alvo da chamada e os argumentos são armazenados a partir da posição um. Um valor armazenado no vetor de variáveis locais pode tomar um ou dois espaços do vetor, dependendo do seu tipo. Um elemento do tipo inteiro ou uma referência a um objeto toma um único espaço. Tipos como long ou double tomam dois espaços, mas são sempre endereçados pelas instruções utilizando o número do primeiro elemento que ocupam no vetor. Neste caso, uma instrução que acessasse diretamente o segundo elemento é ilegal. A figura 12.1 mostra a configuração inicial do vetor de variáveis locais de um método de classe e um de instância. As posições inicialmente não ocupadas do vetor são utilizadas para armazenar as variáveis locais e valores temporários do método.

```
static void myMethod(int x, long k) void myMethod(int x, long k)
{
    ...
}
```

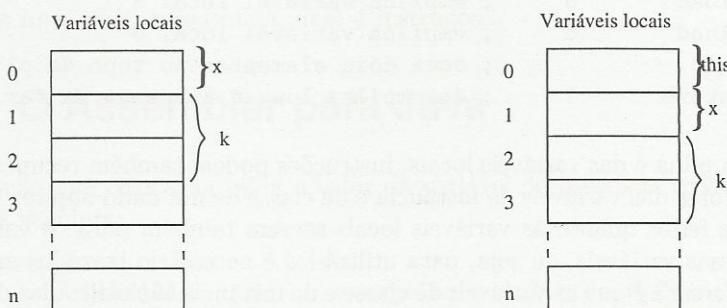


Figura 12.1 – Exemplo de alocação de argumentos no vetor de variáveis locais.

O espaço reservado para a realização de operações funciona na forma de uma pilha. A maioria das instruções utiliza os valores armazenados no topo como operandos e armazena seus resultados também no topo desta pilha. Tomemos como exemplo a instrução `iadd` que toma dois valores inteiros como operandos e produz como resultado um valor inteiro que é a soma dos dois operandos. Quando essa instrução é executada, a pilha deve ter, pelo menos, dois elementos e os dois elementos que estiverem nas posições no topo da pilha devem ser inteiros. Como resultado da instrução, os dois elementos que estavam no topo são retirados, a pilha decresce de duas unidades e o

valor correspondente à soma daqueles valores é, então, empilhado. No final, a altura da pilha fica uma unidade inferior àquela que tinha antes da execução do *iadd*. A figura 12.2 mostra esta situação.

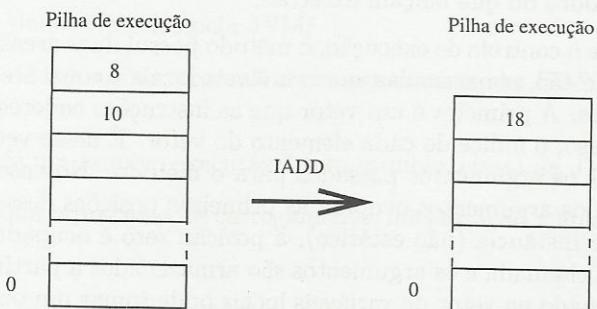


Figura 12.2 – Pilha na execução da instrução *iadd*.

Assim, muitas das instruções da JVM atuam sobre a pilha, onde são executadas as operações. Outras instruções servem para armazenar ou recuperar os valores que estão nas variáveis locais. Assim, por exemplo, para somar dois valores longos que estão armazenados nas variáveis locais números 3 e 6 e colocar o resultado na variável número 1 deve-se primeiro empilhar os valores armazenados no vetor de variáveis locais para, então, fazer a soma e depois transportar o resultado de volta para o vetor de variáveis locais. As instruções correspondentes para isso seriam:

```

lload      3      ; empilha variável local 3
lload      6      ; empilha variável local 6
ladd          ; soma dois elementos no topo da pilha
lstore     1      ; desempilha long e armazena na var. local 1

```

Além da pilha e das variáveis locais, instruções podem também recuperar e armazenar os valores das variáveis de instância e de classe de um dado objeto. Os mesmos comentários feitos quanto às variáveis locais servem também para os valores armazenados nessas variáveis, ou seja, para utilizá-los é necessário trazê-los antes para a pilha. A diferença é que as variáveis de classe e de instância são utilizadas diretamente por meio de seu nome.

Podemos classificar as 227 instruções correntemente definidas para a JVM nas seguintes categorias:

- manipulação da pilha como *push*, *pop* e *dup*: 9 instruções;
- empilhamento de constantes como *ldc*, *iconst_0*, *acconst_null*: 20 instruções;
- empilhamento de variáveis locais como *iload*, *aload*: 25 instruções;
- armazenamento de valores nas variáveis locais como *istore*, *astore*: 25 instruções;
- criação de objetos ou arrays como *new*, *multianewarray*: 4 instruções;

- empilhamento de valores de arrays como *iaload*, *aaload*: 8 instruções;
- armazenamento de valores em arrays como *iastore*, *aastore*: 8 instruções;
- manipulação de variáveis como *putfield*, *getfield*: 4 instruções;
- verificação de tipos como *checkcast*, *instanceof*: 2 instruções;
- operações aritméticas como *iadd*, *ineg*: 24 instruções;
- operações lógicas como *ishl*, *ixor*: 12 instruções;
- conversões como *i2l*, *l2i*: 15 instruções;
- transferência de controle como *ifeq*, *goto*, *jsr*: 21 instruções;
- comparações como *lcmp*, *dcmpl*: 5 instruções;
- desvio tipo *switch*: 2 instruções;
- retorno de método como *return*, *ireturn*: 6 instruções;
- exceções, *atthrow*: 1 instrução;
- controle de monitores *monitoreenter*, *monitorexit*: 2 instruções;
- chamadas de métodos como *invokevirtual*, *invokestatic*: 4 instruções;
- reservados, *breakpoint*, *impdep1*, *impdep2*: 3 instruções;
- operações “quick” como *ldc_quick*, *new_quick*: 23 instruções;
- outras *iinc*, *wide*, *arraylength*, *nop*: 4 instruções.

12.2 O Assembler para Java

Uma instrução que coloca na pilha o valor da variável de instância `myVar` da classe `MyClass` é a seguinte:

```
getfield 18 ; empilha o valor de MyClass.myVar
```

Mas, como dissemos antes, uma variável de instância é endereçada pelo seu nome. Então, o que representa o número 18 na instrução *getfield*? Bem, por questão de eficiência, as instruções são representadas por meio de seu código (1 byte) e algumas delas têm seus argumentos, geralmente, de tamanho fixo. Assim, em vez de armazenar “`MyClass.myVar`” como parte da instrução, cada classe possui uma tabela de constantes. Dados como a referência à variável `MyClass.myVar` são armazenados nesta tabela e referenciados nas instruções. Assim, o número 18 indica a entrada nessa tabela que deve ser usada na execução da instrução.

Este é apenas um dos muitos detalhes que fazem com que a construção de um arquivo de classe seja, no mínimo, trabalhoso. Para facilitar este trabalho, existem alguns programas e bibliotecas que oferecem uma interface mais agradável para tal

tarefa. É o caso do Jasmin (Java Assembler Interface). Por meio dele, podemos transformar um arquivo texto, que descreve a classe (suas variáveis, métodos etc.), em um arquivo de classe. Ou seja, trata-se de um verdadeiro montador para a JVM. Por exemplo, a instrução citada a pouco seria descrita da seguinte forma:

```
; empilha o valor de MyClass.myVar
getfield  MyClass/myVar Ljava/util/Vector;
```

A primeira parte após a instrução indica o nome da variável e a segunda, o tipo que se espera dessa variável. No Apêndice C são dados alguns detalhes sobre o Jasmin, até mesmo as referências de onde encontrá-lo e como obtê-lo na Internet. Nesta seção apresentaremos apenas algumas características que utilizaremos mais adiante.

Para descrever os componentes da classe, o Jasmin define um conjunto de diretivas. Uma das que usaremos na geração de código para *X⁺⁺* é a diretiva .source, que especifica qual o nome do arquivo-fonte que originou o arquivo de classe. Utilizaremos, ainda, as diretivas .class e .super. A primeira diz qual é o nome da classe que está descrita no arquivo e a segunda, o nome da sua superclasse. Um arquivo típico do Jasmin inicia-se com as seguintes diretivas:

```
.source      bintree.x      ; compilado de um arquivo .x
.class public bintree       ; classe public chamada bintree
.super java/lang/Object     ; não possui uma superclasse declarada
```

Nesse exemplo, a classe foi compilada de um arquivo *X⁺⁺*, é pública, chamada *bintree* e não possui uma superclasse explicitamente declarada, por isso é herdeira de *java.lang.Object*, uma vez que em Java (e na JVM) toda classe deriva de *java.lang.Object*, direta ou indiretamente. Nos diversos exemplos anteriores utilizamos o ";" para marcar o início de um comentário. Essa é a sintaxe usada pelo Jasmin. Tudo que segue o ";" até o final da linha é considerado comentário.

Para definir uma variável não local, utiliza-se a diretiva .field. Por exemplo, as variáveis que em Java seriam definidas como

```
public int myField;
static private Vector myVector;
```

em Jasmin seriam definidas como

```
.field public myField I
.field private static myVector Ljava.util.Vector;
```

Os métodos são definidos por meio de duas diretivas, na forma:

```
.method <tipo de acesso> <assinatura>
    <instruções>
.end method
```

Entre as duas diretivas são colocadas as instruções que formam o método. Na diretiva `.method` devem ser especificadas as propriedades do método, como `static` ou `public`, e a sua assinatura dizendo qual é o nome, quais são os parâmetros esperados e qual é o seu tipo de retorno. Por exemplo, poderíamos ter

```
.method static public main([Ljava/lang/String;)V
    ; instruções
.end method
```

No Jasmin e na JVM, os construtores não são tratados de maneira distinta dos métodos de instância. O tipo de retorno para eles é sempre declarado `void`, indicado por `V`, como neste último exemplo. Seu nome, por padrão, é `<init>`. Assim, um construtor sem parâmetros que chama o construtor da sua superclasse `java.lang.Object` e retorna é definido como

```
.method public <init>()V ; construtor
    aload_0
    invokespecial java/lang/Object/<init>() ; chama super()
    return
.end method
```

Duas outras diretivas que precisaremos utilizar são

```
.limit locals <número inteiro>
.limit stack <número inteiro>
```

Elas são colocadas dentro da definição de cada método e estabelecem, respectivamente, o número de variáveis locais que o método utiliza e o tamanho máximo que a pilha atinge durante a execução do método. Por exemplo:

```
.method public <init>()V ; construtor
.limit locals 1
.limit stack 1
    aload_0
    invokespecial java/lang/Object/<init>() ; chama super()
    return
.end method
```

Esses são os principais elementos de Jasmin, além das instruções propriamente ditas, que utilizaremos neste capítulo. Veremos, em seguida, a implementação do gerador de código.

12.3 Implementação

A implementação do gerador de código é mais uma vez baseada na visita aos nós da árvore sintática. Assim, a declaração e o início da classe são semelhantes ao que

vimos nos capítulos anteriores e são mostrados no programa 12.1. Embora faça parte de um pacote diferente – *codegen* –, a classe *CodGen* é uma subclasse do analisador semântico, uma vez que as informações computadas durante a análise semântica, em particular a tabela de símbolos, são utilizadas na geração de código. Inicialmente são declaradas diversas variáveis, cujos usos serão explicados durante este capítulo. Em seguida, como já estamos habituados, é feita a chamada à análise semântica passando como argumento a raiz da árvore sintática.

A diferença no método *CodeGenRoot* é que, além da raiz da árvore sintática, ele recebe como parâmetro o nome do arquivo-fonte, exatamente como este foi fornecido na linha de comando para o compilador.¹ Esse nome é separado em duas partes, o caminho no sistema de arquivos e o nome propriamente dito. Essas duas partes serão utilizadas na criação do arquivo objeto. Caso a análise não esteja sendo feita a partir de um arquivo, e, sim, da entrada-padrão, então o nome recebido como argumento contém o string vazio "".

Programa 12.1

```

1  public class CodeGen extends TypeCheck
2  {
3      // extensão do arquivo a ser gerado
4      static final String Jasextension = ".jas";
5
6      String SourceFile;           // nome do arquivo-fonte
7      String SourcePath;          // diretório do arquivo fonte
8      String SourceAbs;           // nome completo do fonte
9      PrintWriter fp = null;     // arquivo onde o código é escrito
10     String breaklabel = null;   // armazena rótulo de desvio de um break
11     int currlabel = 0;          // número do próximo rótulo a ser gerado
12     boolean hasStart = false;   // indica se classe tem um método start
13     boolean store = false;      // indica se variável está sendo assinalada
14     int stackSize, stackHigh;   // variáveis que controlam o uso da pilha
15
16     public CodeGen()
17     {
18         super();      // chama construtor de TypeCheck
19     }
20
21     public void CodeGenRoot(ListNode x, String filename)
22             throws SemanticException, GenCodeException
23     {
24         TypeCheckRoot(x);        // faz análise semântica
25         System.out.println("0 Semantic error found");
26
27         int i;
28         i = filename.lastIndexOf(File.separator); // pega o nome do fonte
29         SourceFile = filename.substring(i+1);
30         if (i < 0)
31             SourcePath = "";    // pega o caminho
32         else
33             SourcePath = filename.substring(0,i);

```

¹O método *main* foi omitido por ser semelhante ao visto nos capítulos anteriores.

```

34     SourceAbs = filename; // guarda também o caminho completo
35     CodeGenClassDeclListNode(x); // chama geração para a raiz
36 }

```

Como este também é um capítulo longo omitiremos os métodos que são repetições de outros vistos anteriormente, e que simplesmente invocam a geração de código para os filhos. Passamos, assim, ao método *CodeGenClassDeclNode* que trata de um nó de declaração de uma classe. Esse método é mostrado no programa 12.2.

Devemos lembrar inicialmente que um nó *ClassDeclNode* pode representar uma declaração de classe aninhada e, por isso, é necessário que algumas variáveis sejam salvas antes de iniciar o processamento das informações no nó. Por exemplo, uma das operações feitas neste nó é a criação de um arquivo onde será gravado o programa objeto no formato do Jasmin. A referência ao objeto *PrintWriter* que representa esse arquivo é armazenada na variável *fp*. Quando uma classe aninhada vai ser tratada, essa variável já está “ocupada” com um objeto que é o arquivo objeto da sua classe mais externa. Por isso, o valor de *fp* tem que ser salvo em *fpOld*, para que um outro arquivo possa ser criado e sua referência armazenada em *fp*. Daquele ponto em diante (na subárvore sob o nó da classe aninhada), todo o código será gravado neste novo arquivo. Ao terminar a análise dessa classe é necessário restaurar o valor de *fp* para que o restante do código da classe externa possa ser produzido (no arquivo correto).

Após salvar as variáveis necessárias, é recuperada a entrada na tabela de símbolos correspondente à classe sendo declarada. Dali é recuperada a entrada da sua superclasse. Essas duas entradas são armazenadas em *nc* e *ns*, respectivamente. A partir dessas duas entradas são calculados os nomes “completos” dessas duas classes. Esses nomes, seguindo o padrão adotado na JVM, são formados da seguinte maneira:

- se a classe não é uma classe aninhada, seu nome completo coincide com seu nome declarado. Por exemplo, *MyClassA*;
- se a classe é aninhada seu nome completo é o nome da classe onde ela foi declarada mais um “\$”, mais o seu nome declarado. Por exemplo, *MyClassA\$MyClassB*.

Se a classe sendo declarada não possui uma superclasse (explicitamente declarada), assume-se que sua superclasse é *java.lang.Object*. Ao arquivo criado para a geração de código é dado o nome que é a concatenação do nome do arquivo-fonte com o nome completo da classe e mais o sufixo *.jas*. Por exemplo, a classe *MyClassA\$MyClassB* declarada no arquivo fonte *myfile.x* irá produzir um arquivo objeto chamado *myfile.x.MyClassA\$MyClassB.jas*.

Programa 12.2

```

1 public void CodeGenClassDeclNode(ClassDeclNode x)
2         throws GenCodeException
3 {
4     Symtable temphold = Curtable; // salva tabela corrente
5     boolean tempstart = hasStart; // salva variável hasStart
6     EntryClass c = null;
7     EntryClass nc, ns=null;

```

```

8  String Filename = null, sname;
9  FileOutputStream os;
10 PrintWriter fpOld = fp; // salva arquivo sendo gerado
11
12  if (x == null)
13      return;
14
15  // acha a classe na tabela
16  nc = (EntryClass) Curtable.classFindUp(x.name.image);
17  ns = nc.parent; // pega superclasse
18
19  // pega nome da superclasse
20  if (ns == null)
21      sname = "java/lang/Object"; // nenhuma superclasse
22  else
23      sname = ns.completeName();
24
25  Filename = SourceAbs + "." + nc.completeName() + Jasextension;
26  try
27  {
28      // cria arquivo para gerar código da classe
29      os = new FileOutputStream(Filename);
30  }
31  catch (FileNotFoundException e)
32  {
33      throw new GenCodeException(
34          "Cannot create output file: " + Filename);
35  }
36
37  fp = new PrintWriter(os);
38  System.out.println("Generating " + Filename);
39
40  // escreve o cabeçalho do arquivo: .source .class e .super
41  putCode("-----");
42  putCode(" Code generated by X++ compiler");
43  putCode(" Version 0.1 - 2002");
44  putCode("-----");
45  putCode(".source " + SourceFile); // escreve .source
46  putCode(".class public " + nc.completeName()); // .classe
47  putCode(".super " + sname); // e .super
48
49  Curtable = nc.nested; // tabela corrente = tabela da classe
50  hasStart = false;
51  CodeGenClassBodyNode(x.body);
52  if (hasStart) // verifica se classe tem método start
53      createMain();
54  // fechar arquivo para classe
55  fp.close();
56  fp = fpOld; // recupera arquivo anterior
57  Curtable = tempHold; // recupera tabela corrente
58  hasStart = tempStart; // recupera flag
59 }

```

Uma vez calculados os nomes da classe e da sua superclasse e criado o arquivo objeto, é gravado o cabeçalho no formato Jasmin, como descrito no início deste capítulo. Além de um comentário indicando que o arquivo foi gerado pelo compilador *X++*, são inseridas as diretivas *.class*, *.super* e *.source*. Utilizamos o método *putCode*, que veremos mais adiante, para inserir código no arquivo objeto. O que ele faz é basicamente transcrever o string passado como argumento em uma nova linha no arquivo.

Na linha 51 é feita a chamada da geração de código para o corpo da classe. Em seguida é feita uma verificação sobre o valor da variável *hasStart*. Essa variável indica se no corpo da classe foi declarado o método *int start()*. Caso tenha sido declarado, este será o ponto de entrada para a classe, o método pelo qual a classe pode ser executada. Existe, porém, um problema: como indicar para a JVM que irá executar a classe que este é o ponto de entrada, visto que a JVM sempre inicia a execução por meio do método *static void main(String[])*? O que faremos é criar o método *main* que efetua a chamada ao nosso ponto de entrada *start*. É o que realiza o método *createMain*, chamado na linha 53 do programa 12.2 e mostrado no programa 12.3.

Programa 12.3

```

1  private void createMain()
2  {
3      EntryClass v = (EntryClass) Curtable.levelup;
4
5      putCode();
6      putCode(";Entry point for the JVM");
7      putCode(".method static public main([Ljava/lang/String;)V");
8      putCode(".limit locals 1");
9      putCode(".limit stack 1");
10     putCode("invokestatic langXrt/Runtime/initialize()I");
11     putCode("ifne           end");
12     putCode("invokestatic " + v.completeName() + "/start()I");
13     putCode("pop");
14     putLabel("end:");
15     putCode("invokestatic langXrt/Runtime/finilizy()V");
16     putCode("return");
17     putCode(".end method");
18 }
```

*

O método *main* gerado tem o seguinte formato:

```

;Entry point for the JVM
.method static public main([Ljava/lang/String;)V
.limit locals 1
.limit stack 1
invokestatic langXrt/Runtime/initialize()I
ifne           end
invokestatic bintree/start()I
pop
end:   invokestatic langXrt/Runtime/finilizy()V
```

```

return
.end method

```

Antes de iniciar a execução da classe por meio da chamada a `start`, o método chama `langXrt.Runtime.initialize()` que retorna um inteiro. Esse método será discutido na seção 12.4 e o que ele faz é inicializar o “runtime” da nossa linguagem. O valor por ele retornado indica se a inicialização foi bem-sucedida (valor igual a zero) ou se um erro ocorreu. Em caso de erro, a chamada a `start` não é feita.

O leitor mais atento deve ter percebido que a abordagem escolhida causará um problema se o programador decidiu declarar um método `main(String[])` no seu programa *X⁺⁺*. Permanece aí um exercício (ver Apêndice A) a ser resolvido: até onde isso é um problema e quais seriam as possíveis soluções.

Agora que já temos um arquivo onde gerar o código da classe vamos realizar essa operação para cada um dos elementos dentro da classe. O método que trata do corpo da classe é mostrado no programa 12.4. Para as classes aninhadas, como já vimos, o processo se repete e um novo arquivo será criado. Antes da geração de código para os construtores, o método procura na tabela de símbolos um construtor sem parâmetros para a classe corrente. Ele certamente existe, pois, como vimos no capítulo 10, caso ele não seja declarado pelo programador, o próprio analisador semântico cuidará de inseri-lo na tabela de símbolos. Neste caso, o campo `fake` da entrada é verdadeiro e indica que uma implementação-padrão deve ser gerada para este construtor, o que é efetuado pelo método `GeraConstructorDefault` mostrado no programa 12.5.

Programa 12.4

```

1 public void CodeGenClassBodyNode(ClassBodyNode x)
2         throws GenCodeException
3 {
4     EntryMethod l;
5
6     if (x == null) return;
7
8     CodeGenClassDeclListNode(x.clist);
9     CodeGenVarDeclListNode(x.vlist);
10    l = Curtable.methodFindInclass("constructor", null);
11    if (l.fake)
12        GeraConstructorDefault(); // se construtor é falso, gera default
13
14    CodeGenConstructDeclListNode(x.ctlist);
15    CodeGenMethodDeclListNode(x.mlist);
16 }

```

Programa 12.5

```

1 private void GeraConstructorDefault()
2 {
3     String sup;
4     EntryClass cc;
5
6     cc = (EntryClass) Curtable.levelup; // pega classe corrente

```

```

7     if (cc.parent != null) // acha nome da superclasse
8         sup = cc.parent.completeName();
9     else
10        sup = "java/lang/Object";
11
12    // gera o código para o construtor default
13    putCode();
14    putCode(";Default constructor. Calls super()");
15    putCode(".method public <init>()V");
16    putCode(".limit locals 1");
17    putCode(".limit stack 1");
18    putCode("aload_0");
19    putCode("invokespecial " + sup + "<init>()V");
20    putCode("return");
21    putCode(".end method");
22 }

```

O tratamento de uma declaração de variável de instância é bastante simples, como mostra o programa 12.6. Para cada nó *VarDeclNode* no corpo da classe é recuperada a entrada correspondente na tabela de símbolos e depois é gerada uma diretiva .field. O método *dscJava* é declarado abstrato na classe *EntryTable* e implementado em cada uma das suas subclasses. Ele devolve um string que representa o tipo da entrada (no caso, o tipo da variável) num formato-padrão definido pela JVM. Por exemplo, para os tipos de dados em *X⁺⁺*, temos as seguintes representações:

Tipo	Representação
int	I
string	Ljava/lang/String;
MyClass	LMyClass;
int[]	[I
MyClassA\$MyClassB [] []	[[LMyClassA\$MyClassB;

Programa 12.6

```

1  public void CodeGenVarDeclNode(VarDeclNode x)
2  {
3      ListNode p;
4      EntryVar l;
5      EntryTable c = null;
6
7      if (x == null) return;
8
9      for (p = x.vars; p != null; p = p.next)
10     {
11         VarNode q = (VarNode) p.node;
12
13         // pega variável na tabela
14         l = Curtable.varFind(q.position.image);
15
16         // gera diretiva .field
17         putCode(".field public " + l.name + " " + l.dscJava());
18     }

```

O tratamento de uma declaração de construtor, feito pelo método no programa 12.7, é um pouco longo. Assim, analisaremos por partes. Inicialmente é montada a lista de tipos dos parâmetros que é, então, usada para localizar a entrada do construtor na tabela de símbolos. Aqui vale notar mais uma vez que a busca poderia ser evitada se associássemos a entrada da tabela ao nó da árvore sintática no momento em que o símbolo é inserido na tabela de símbolos. Essa mesma observação vale para muitos outros pontos na geração de código. Assim, não vamos repeti-la toda vez, ficando sob incumbência do leitor identificá-los.

Na linha 34 e subsequentes é calculada a representação dos tipos dos parâmetros na forma de um string, de acordo com padrão da JVM descrito anteriormente. Utilizando essa informação é gerada a diretiva `.method` no arquivo de saída. O nome utilizado, de acordo com o padrão do Jasmin e JVM, é `<init>` e o tipo de retorno é `void`, representado pelo V. A seguir, alguns exemplos:

Construtor	Cabeçalho
<code>constructor()</code>	<code>.method public <init>()V</code>
<code>constructor(int x)</code>	<code>.method public <init>(I)V</code>
<code>constructor(string x)</code>	<code>.method public <init>(Ljava/lang/String;)V</code>
<code>constructor(int x[] [])</code>	<code>.method public <init>([[I)V</code>

Programa 12.7

```

1  public void CodeGenConstructDeclNode(ConstructDeclNode  x)
2  {
3      EntryMethod t;
4      EntryRec r = null;
5      EntryTable e;
6      EntryClass thisclass;
7      EntryVar thisvar;
8      ListNode p;
9      VarDeclNode q;
10     VarNode u;
11     int n;
12     String sup = "";
13
14     if (x == null) return;
15     p = x.body.param;
16     n = 0;
17
18     // monta a lista com os tipos dos parâmetros
19     while (p != null)
20     {
21         // q = nó com a declaração do parâmetro
22         q = (VarDeclNode) p.node;
23         u = (VarNode) q.vars.node; // u = nó com o nome e dimensão
24         n++;
25
26         // acha a entrada do tipo na tabela
27         e = Curtable.classFindUp(q.position.image);
28
29         // constrói a lista com os tipos dos parâmetros

```

```

30         r = new EntryRec(e, u.dim, n, r);
31         p = p.next;
32     }
33
34     String parlist = "";
35     if (r != null)
36     {
37         r = r.inverte(); // inverte a lista
38         // monta descritor para cada um dos parâmetros
39         parlist = r.dscJava();
40     }
41
42     // acha a entrada do construtor na tabela
43     t = Curtable.methodFind("constructor", r);
44     CurMethod = t; // guarda método corrente
45
46     // gera código para o cabeçalho do construtor
47     putCode();
48     putCode(".method public <init>(" + parlist + ")V");
49     // número máximo de locais utilizadas
50     putCode(".limit locals " + t.totallocals);
51     // pega a entrada da classe corrente na tabela
52     thisclass = (EntryClass) Curtable.levelup;
53     // pega o nome da superclasse
54     if (thisclass.parent != null)
55         sup = ((EntryClass)thisclass.parent).completeName();
56     else
57         sup = "java/lang/Object";
58
59     // verifica se tem chamada ao superconstrutor
60     if (! CurMethod.hassuper)
61     { // se não, chama super()
62         putCode();
63         putCode("aload_0", 1);
64         putCode("invokespecial " + sup + "<init>()V", -1);
65     }
66
67     // inicia um novo escopo na tabela corrente
68     Curtable.beginScope();
69
70     thisvar = new EntryVar("this", thisclass, 0);
71     Curtable.add(thisvar); // inclui variável local "this" com número 0
72     Nlocals = 1;          // inicializa número de variáveis locais
73
74     // inicializa número de espaços ocupados na pilha
75     stackSize = stackHigh = 0;
76     CodeGenMethodBodyNode(x.body);
77     Curtable.endScope(); // retira variáveis locais da tabela
78     putCode("return");
79     putCode(".limit stack " + (t.totalstack = stackSize));
80     putCode(".end method"); // final do método
81 }

```

Em seguida, na linha 50, é gerada a diretiva `.limit locals` utilizando a informação sobre o número de variáveis locais que o método usa e que foi previamente (análise semântica) calculado e guardado na tabela de símbolos. A partir daí, inicia-se a geração das instruções do construtor. Antes de iniciar a geração para o corpo do construtor, verifica-se se este se inicia com uma chamada ao construtor da superclasse. Essa informação foi armazenada na entrada do método, na tabela de símbolos. Caso não exista tal chamada é gerado código chamando o construtor da superclasse sem nenhum argumento (linha 64).

Em seguida são feitas algumas inicializações antes de se chamar o método para gerar código para o corpo do construtor. Em particular, são inicializadas as variáveis `stackSize` e `stackHigh` que controlam o número de espaços utilizados na pilha. Esta informação é utilizada para gerar a diretiva `.limit stack`, após o retorno de `CodeGenMethodBodyNode`. Vejamos, então, como esse valor é calculado.

A cada instrução que é gerada, utilizando-se o método `putCode`, pode-se informar qual é a variação (positiva ou negativa) no tamanho da pilha. Por exemplo, a instrução `iadd` é gerada sempre se chamando o método `putCode("iadd", -1)`, indicando que a pilha decresce de uma unidade com a execução desta instrução. Assim, dentro do método `putCode` exibido no programa 12.8, pode-se controlar o tamanho que a pilha deve ter após a execução da instrução e, portanto, também o tamanho máximo que a pilha alcança durante a execução do método. A variável `stackHigh` guarda o tamanho da pilha “corrente”, após a execução da instrução, e `stackSize`, o tamanho máximo alcançado. Para que esta abordagem funcione há algumas restrições que a geração de código deve seguir. Elas serão explicadas na seção 12.3.3.

Programa 12.8

```

1  private void putCode(String s, int inc)
2  {
3      stackHigh += inc;
4      if (stackHigh > stackSize )
5          stackSize = stackHigh;
6      fp.println("\t\t" + s);
7  }
8
9  private void putCode(String s)
10 {
11     putCode(s, 0);
12 }
13
14 private void putCode()
15 {
16     fp.println();
17 }
18
19 private void putLabel(String s)
20 {
21     fp.println(s);
22 }
```

A geração de código para a declaração de métodos segue o mesmo esquema da declaração de construtores, por isso será omitida aqui.

12.3.1 Geração de código para os comandos

Passemos, então, à geração de código para os comandos. A declaração de variáveis locais não gera código, mas precisamos, como já feito na análise semântica, cuidar de incluí-las na tabela de símbolos, pois quando um comando fizer referência a alguma delas precisaremos de algumas informações para a geração de código. Assim, o método que trata da declaração de variáveis é similar ao visto no capítulo 11. Algo similar acontece para o tratamento de comando composto, *CodeGenBlockNode(BlockNode x)*.

Iniciamos com um comando bastante simples: o comando print. Na geração é utilizada uma chamada ao método *print* sobre a variável *System.out* do runtime Java, que também está disponível no runtime *X⁺⁺*. Esta é uma das vantagens em se utilizar a JVM como alvo da geração de código. O método mostrado no programa 12.9 primeiro gera código que empilha o objeto *System.out*, depois chama o método que gera o código para a expressão do print, deixando o resultado no topo da pilha, e, então, gera o código para a chamada do método *PrintStream.print*, que manda para a saída-padrão o string que está no topo da pilha. Note na chamada de *putCode* os valores e incremento e decremento do tamanho da pilha.

Programa 12.9

```

1  public void CodeGenPrintNode(PrintNode x)
2  {
3      type t;
4
5      if (x == null) return;
6
7      putCode();
8      putCode(";begins print ");
9      // coloca System.out na pilha
10     putCode("getstatic java/lang/System/out Ljava/io/PrintStream;", 1);
11
12     // coloca resultado da expressão na pilha
13     store = false;
14     t = CodeGenExpreNode(x.expr);
15     // chama PrintStream.print(String)
16     putCode("invokevirtual " +
17             "java/io/PrintStream/print(Ljava/lang/String;)V", -2);
18 }
```

O método que trata de um comando read também é bastante simples, porém possui algumas particularidades que merecem ser comentadas. Como não adotamos a abordagem de armazenar no nó da árvore sintática o tipo da expressão, temos que chamar novamente o método que faz a checagem de tipo. Não para verificar se existe algum problema, pois isso foi feito na análise semântica, mas porque precisamos saber qual o tipo da expressão para gerar o código de chamada do método correto para fazer

a leitura ou de um inteiro, ou de um string. Ambos os métodos encontram-se na classe *langXrt.Runtime* que implementa o runtime *X⁺⁺* (Seção 12.4). A chamada a qualquer um deles deixa um novo valor na pilha, que é o valor lido na entrada-padrão.

O armazenamento do valor não pode ser gerado neste método. Para tal, seria necessário saber qual o tipo de armazenamento que deve ser feito: se a uma variável local, a uma variável de instância ou a um elemento de um array. Para isso, seria necessário analisar a subárvore abaixo deste nó. Em vez disso, é marcada a variável *store*, indicando-se ao método que trata do nó abaixo, que esta é uma operação de atribuição. Os métodos que tratam dos três tipos de nós que podem aparecer como filhos do nó corrente devem, então, verificar esse flag e gerar código de acordo.

Programa 12.10

```

1  public void CodeGenReadNode(ReadNode x)
2  {
3      type t = null;
4
5      if (x == null) return;
6      try
7      {
8          // tem que verificar o tipo da expressão
9          t = TypeCheckExpreNode(x.expr);
10     }
11     catch (Exception e) {}
12
13     if (t.ty == INT_TYPE) // chama readInt, resultado na pilha
14         putCode("invokestatic langXrt/Runtime/readInt()I", 1);
15     else // lê string chama readString, resultado na pilha
16         putCode("invokestatic " +
17                 "langXrt/Runtime/readString()Ljava/lang/String;", 1);
18
19     store = true; // indica que é operação de armazenagem
20     CodeGenExpreNode(x.expr);
21 }
```

Por exemplo, vejamos no programa 12.11 como funciona o método que trata de um *IndexNode*, visto que ele precisa calcular o valor da expressão que indica qual é o array a ser utilizado e da expressão índice. Note que antes de calcular essas expressões, a variável *store* recebe o valor false, pois abaixo deste nó, por exemplo, no cálculo do índice, a operação desejada não é de armazenagem. Se tivermos uma expressão como

```
read a[b[i]];
```

então a chamada a esse método para a primeira indexação (em relação ao array a) deve receber a variável *store* com o valor verdadeiro. Já para a indexação de b, o seu valor deve ser falso, pois se deseja apenas usar o valor de b[i] e não redefini-lo.

Além disso, se for uma operação de armazenagem, há o inconveniente de que o valor a ser armazenado já foi colocado no topo da pilha e, por isso, gera-se a instrução de

swap, que inverte os dois valores no topo. Antes de gerar as instruções para recuperar ou armazenar o valor no array, devemos ter na pilha a referência ao array, o índice do elemento desejado e, por último, o valor a ser armazenado (se for este o caso). Note que os métodos que geram código para as expressões continuam a retornar o tipo da expressão, pois essa informação pode ser utilizada no método que faz a chamada, como acontece no programa 12.11.

Programa 12.11

```

1  public type CodeGenIndexNode(IndexNode x)
2  {
3      EntryClass c;
4      type t1, t2;
5      boolean b = store;
6
7      if (x == null) return null;
8
9      // calcula primeiro filho, o array
10     store = false;
11     t1 = CodeGenExpreNode(x.expr1);
12
13     // se for para armazenar, troca topo
14     if (b)
15         putCode("swap");
16
17     // pega índice
18     t2 = CodeGenExpreNode(x.expr2);
19
20     if (b) // realiza a operação, armazenar ou carregar na pilha
21     {
22         putCode("swap"); // troca o topo
23         if (t1.ty == INT_TYPE && t1.dim == 1) // array de inteiros
24             putCode("iastore", -3);
25         else
26             putCode("aastore", -3);
27     }
28     else
29     {
30         if (t1.ty == INT_TYPE && t1.dim == 1) // array de inteiros
31             putCode("iaload", -1);
32         else
33             putCode("aaload", -1);
34     }
35     store = b;
36     return new type(t1.ty, t1.dim - 1);
37 }
```

Voltando aos comandos, vejamos o de atribuição. Em alguns aspectos assemelha-se ao *read*, no sentido de que instruções para o armazenamento de um valor deverão ser geradas em algum ponto. Mas difere de *read* em relação à leitura de um valor, pois somente inteiros ou strings serão aceitos. No caso da atribuição, objetos e arrays também podem ser utilizados. Por isso, no programa 12.12, na linha 19, é gerada a

instrução que converte o tipo do valor a ser atribuído, sem que o programador o tenha feito explicitamente. É possível, porém, que a coerção não seja válida. Nesse caso nenhuma atitude é tomada e um erro (exceção) ocorre no programa, na execução da instrução *checkcast*.

```

1  public void CodeGenAtribNode(AtribNode x)           Programa 12.12
2  {
3      type t1 = null, t2;
4
5      if (x == null) return;
6
7      // pega tipo da expressão à esquerda da atribuição
8      try {
9          t1 = TypeCheckExpreNode(x.expr1);
10     }
11     catch (Exception e) {}
12
13     // calcula expressão à direita
14     store = false;
15     t2 = CodeGenExpreNode(x.expr2);
16
17     // faz coerção de tipos
18     if (t1.ty != t2.ty && isSubClass(t1.ty,t2.ty))
19         putCode("checkcast " + t1.dscJava());
20
21     // armazena resultado
22     store = true;
23     CodeGenExpreNode(x.expr1);
24 }
```

O comando *super* deve ser traduzido numa chamada de método. Para isso, inicialmente é empilhada a referência ao objeto que é o alvo da chamada. É a variável local zero, ou seja, a variável *this*. Depois é gerado o código para cada uma das expressões que serão os argumentos da chamada (linha 19 do Programa 12.13). Em seguida é gerada a instrução com a chamada para o método *<init>* da superclasse da classe corrente. A variável *arglist* contém a descrição, na forma de um string, dos tipos dos argumentos utilizados. A variável *removeStack* contém o número de argumentos. Somando-se a estes o objeto *this*, tem-se o número de elementos que serão retirados da pilha na execução da instrução (considerando-se já o fato de que o método não retorna nenhum valor).

```

1  public void CodeGenSuperNode(SuperNode x)           Programa 12.13
2  {
3      type t;
4      String sup;
5
6      if (x == null) return;
7
8      // p aponta para a entrada da superclasse da classe corrente
```

```

9   EntryClass p = Curtable.levelup.parent;
10
11  sup = p.completeName();
12
13  putCode();
14  putCode(";begins super");
15  putCode("aload_0", 1);
16
17  // coloca os parâmetros na pilha
18  store = false;
19  t = CodeGenExpreListNode(x.args);
20  String arglist = (t.ty == null) ? "" : t.dscJava();
21  int removeStack = (t.ty == null) ? 0 : ((EntryRec) t.ty).cont;
22
23  putCode("invokespecial " +
24      sup + "/<init>(" + arglist + ")V", -(removeStack+1));
25 }

```

No programa 12.14, vemos o método que gera código para um comando for. A lógica utilizada para esse comando deve ser a seguinte:

- gerar código para o comando de inicialização (linha 12);
- colocar no código o rótulo *forloopN*:;, onde *N* é um número seqüencial devolvido pelo método *newLabel()* (linha 15);
- fazer a geração para a expressão de controle. O código gerado deixa um valor inteiro no topo da pilha (linha 18);
- gerar código que teste se o valor que ficou na pilha é zero e, se for, desvie a execução para o final do *for* (rótulo *rofN*), pois o valor zero indica que a expressão é falsa (linha 21);
- gerar código para o comando dentro do for e, em seguida, para o comando de incremento (linha 24);
- gerar uma instrução *goto* para o início do laço do for, ou seja, para o rótulo *forloopN* (linha 29);
- gerar o rótulo *rofN*: que é o final do *for* (linha 31).

Programa 12.14

```

1  public void CodeGenForNode(ForNode x)
2  {
3      String lab = newLabel();
4      String oldbreaklabel = breaklabel; // salva label do break
5
6
7      if (x == null) return;
8
9      putCode();

```

```

10    putCode(";begins for " + lab);
11    // gera código para inicialização
12    CodeGenStatementNode(x.init);
13
14    // gera label para laço do comando
15    putLabel("forloop" + lab + ":");

16    // gera código para condição e deixa resultado na pilha
17    store = false;
18    CodeGenExpreNode(x.expr);

19
20    // testa o resultado
21    putCode("ifeq rof" + lab, -1);
22    breaklabel = "rof" + lab; // novo label para break: fim do for
23    // gera código para o corpo do for
24    CodeGenStatementNode(x.stat);
25    // gera código para incremento
26    CodeGenStatementNode(x.incr);

27
28    // gera retorno ao laço
29    putCode("goto forloop" + lab);
30    // gera label do fim do comando
31    putLabel(breaklabel + ":");

32
33    breaklabel = oldbreaklabel; // restaura label do break
34 }

```

Um ponto a se notar é a utilização da variável *breakLabel* para armazenar o rótulo que representa o final do comando. Esse rótulo é utilizado na geração do comando *break*, como mostra o programa 12.15. A cada comando *for* aninhado esse rótulo deve ser ajustado para o final do comando, tomindo-se o cuidado, porém, de salvar seu valor anterior, de modo que ao final da geração de código para o *for* mais interno, o valor do rótulo do comando externo seja recuperado.

Programa 12.15

```

1 public void CodeGenBreakNode(BreakNode x)
2 {
3     if (x == null) return;
4
5     putCode();
6     putCode(";begins break ");
7     // gera desvio para o label do break
8     putCode("goto " + breaklabel);
9 }

```

Não apresentaremos os métodos que geram código para os comandos *return* e *if* por serem semelhantes a métodos já apresentados para outros comandos. E nem para o comando vazio, por ser trivial. Na seção 12.3.2 serão apresentados os métodos que geram código para as expressões em X^{++} .

12.3.2 Geração de código para as expressões

Um dos métodos que tratam da geração de código para expressões foi visto na seção 12.3.1. É o método que trata de uma expressão de indexação de um array. Continuemos vendo a geração de código para as expressões mais básicas, ou seja, variáveis e constantes. A geração para constante inteira ou string é basicamente a mesma e consiste em colocar na pilha o valor da constante. Isto é mostrado nos métodos do programa 12.16. Somente uma instrução, do tipo *ldc*, cuida desta operação.

Programa 12.16

```

1  public type CodeGenIntConstNode(IntConstNode x)
2  {
3      if (x == null) return null;
4
5      // carrega a constante na pilha
6      putCode("ldc " + x.position.image, 1);
7      return new type(INT_TYPE, 0);
8  }
9
10 public type CodeGenStringConstNode(StringConstNode x)
11 {
12     if (x == null) return null;
13
14     // carrega a constante na pilha
15     putCode("ldc " + x.position.image, 1);
16     return new type( STRING_TYPE, 0);
17 }
```

Para a variável é necessário encontrar na tabela de símbolos a entrada correspondente e verificar se esta se trata de uma variável local ou de classe. Caso seja variável local, (linha 13 do Programa 12.17) indicada pelo valor de *localcount* maior que zero na entrada da variável na tabela de símbolos, gera-se a instrução que simplesmente pega o valor da variável e joga-a na pilha (no caso de não ser uma atribuição) ou da pilha para a variável (caso seja uma atribuição). Aqui também é utilizada a variável *store* que indica ao gerador de código caso se deva gerar código para armazenar ou recuperar um valor. No início do método, faz-se a escolha de qual instrução utilizar, com base no valor da variável *store*. Ainda, é preciso decidir, para o caso das variáveis locais, se o valor a ser manipulado é um inteiro ou um string ou array (tratados como objetos na JVM). Assim, a comparação na linha 14 serve para decidir-se se o prefixo da instrução será um “i” para inteiros ou um “a” para strings ou arrays.

No caso de variáveis de classe, deve-se utilizar instruções específicas para recuperação de seus valores. Como se trata de um nome de variável simplesmente, está se referindo a uma variável do objeto corrente, ou seja, do objeto *this* ou, mais precisamente, ao objeto cuja referência está na variável local zero. Por isso, na linha 25 é gerado código que empilha esse valor e, em seguida, o código de *getfield* ou *putfield* de acordo com o valor de *store*. A própria instrução indica qual o nome da variável a fazer parte da operação. Uma observação a ser feita é que no caso de operação de armazenagem, o valor a ser colocado na variável está já no topo da pilha e, por isso, após a instrução que empilha o valor de *this*, é gerada uma instrução *swap* que in-

verte os dois valores no topo da pilha, deixando-os na posição correta para a próxima instrução, o *putfield*.

Programa 12.17

```

1  public type CodeGenVarNode(VarNode x)
2  {
3      EntryVar p;
4      String s = store ? "store" : "load";
5      String s2 = store ? "putfield" : "getfield";
6
7      if (x == null) return null;
8
9      // procura variável na tabela
10     p = Curtable.varFind(x.position.image);
11
12     if (p.localcount >= 0)
13     {   // é variável local
14         if (p.type == INT_TYPE && p.dim == 0) // escolhe iload ou aload
15             putCode("i" + s + " " + p.localcount, store ? -1: 1);
16         else
17             putCode("a" + s + " " + p.localcount, store ? -1: 1);
18     }
19     else
20     {   // variável da classe
21         // pega a classe a que ela pertence
22         EntryClass v = p.mytable.levelup;
23
24         // empilha "this"
25         putCode("aload_0", 1);
26         // se for para armazenar, troca topo da pilha
27         if (store)
28             putCode("swap");
29
30         putCode(s2 + " " + v.completeName() + "/" + p.name + " "
31                 + p.dscJava(), store ? -2: 0);
32     }
33
34     return new type(p.type, p.dim);
35 }
```

Passemos à criação de um objeto, utilizando a operação *new*. O método que gera código para esse tipo de nó é mostrado no programa 12.18. A primeira operação a ser efetuada por esse método é encontrar na tabela de símbolos a entrada correspondente à classe da qual se deseja criar o objeto. Depois disso, é emitida a instrução *new* com o nome completo da classe. Em seguida é gerada a instrução *dup*, que copia a referência ao objeto criada pela instrução anterior. Depois é gerado código para os argumentos do construtor e, finalmente, a chamada ao construtor que combine com os tipos dos argumentos utilizados. No final, o código gerado faz com que o objeto criado e inicializado pela chamada ao construtor esteja no topo da pilha.

Programa 12.18

```

1  public type CodeGenNewObjectNode(NewObjectNode x)
2  {
3      type t;
4      EntryMethod p;
5      EntryClass c;
6
7      if (x == null) return null;
8
9      // procura a classe da qual se deseja criar um objeto
10     c = (EntryClass) Curtable.classFindUp(x.name.image);
11
12     // cria o objeto
13     putCode("new " + ((EntryClass) c).completeName(), 1);
14     // duplica para chamar construtor
15     putCode("dup", 1);
16
17     // gera código para os argumentos do construtor
18     t = CodeGenExpreListNode(x.args);
19
20     String arglist = (t.ty == null) ? "" : t.dscJava();
21     int removeStack = (t.ty == null) ? 0: ((EntryRec) t.ty).cont;
22
23     // gera chamada do construtor
24     putCode("invokespecial " + c.completeName() +
25             "/<init>(" + arglist +")V", -(removeStack+1));
26
27     // objeto criado está na pilha
28     t = new type(c, 0);
29     return t;
30 }
```

Vejamos, agora, um dos tipos de expressão binária de X^{++} , que é a expressão relacional. Como vimos no capítulo 11, ela pode ser a comparação entre inteiros ou entre strings, objetos ou arrays, mas neste último caso somente a comparação de igualdade ou desigualdade (operadores `==` ou `!=`) é aceita. Inicialmente é gerado código para as duas subexpressões, de modo que o resultado da primeira (à esquerda) fique na penúltima posição da pilha e o resultado da subexpressão à direita do operador, no topo. Resta, portanto, gerar código para que a comparação seja feita e o resultado seja o valor zero no topo da pilha, se a comparação é falsa, ou 1, caso seja verdadeira. Caso os operandos sejam inteiros, com a chamada a `selectBinInstruct` na linha 17 do programa 12.19, calcula-se qual a instrução da JVM adequada para a comparação desejada. Por exemplo, `if_cmplt` para “`<`” ou `if_cmpge` para “`">=`”. Caso seja a comparação entre strings, objetos ou arrays, uma das duas instruções é escolhida: `if_acmpneq` ou `if_acmpneq`.

Essas instruções efetuam a comparação desejada entre os dois operadores no topo da pilha e executam um desvio caso tal comparação resulte verdadeira. Assim, a instrução a ser gerada logo após a comparação é um `bipush 0` que coloca o valor 0 (indicando resultado falso) no topo da pilha. E, logo depois, um desvio para um ponto

onde a expressão já foi todo calculada. Posteriormente é gerado o código para quando a expressão é verdadeira, que empilha o valor 1 e pronto.

Vamos tomar como exemplo a expressão $a < b$. O código gerado seria algo como

```

    iload  3      ; cálculo da 1a. subexpressão
    iload  2      ; cálculo da 2a. subexpressão
    if_cmplt   relexpr8
    bipush  0
    goto    pxeler8
relexpr8:
    bipush  1
pxeler8:

```

Programa 12.19

```

1  public type CodeGenRelationalNode(RelationalNode x)
2  {
3      type t1, t2;
4      int op; // operação
5      String lab = newLabel();
6
7      if (x == null) return null;
8
9      op = x.position.kind;
10
11     // gera código para as duas subexpressões
12     t1 = CodeGenExpreNode(x.expr1);
13     t2 = CodeGenExpreNode(x.expr2);
14
15     if ( t1.dim == 0 && t1.ty == INT_TYPE )
16     {   // se for inteiro, pega instrução correspondente
17         String s = selectBinInstruct(op);
18         putCode(s + " relexpr" + lab, -2);
19     }
20     else
21     {   // se for array, objeto ou string, usa a mesma instrução
22         if ( op == langXConstants.EQ) // só == ou != são permitidos
23             putCode("if_acmpeq relexpr" + lab, -2);
24         else
25             putCode("if_acmpne relexpr" + lab, -2);
26     }
27     putCode("bipush 0", 1);
28     putCode("goto pxeler" + lab);
29     putLabel("relexpr" + lab + ":" );
30     putCode("bipush 1"); // não incrementa stack, pois seria contar
31                           // duas vezes, já que o bipush 0 incrementou
32     putLabel("pxeler" + lab + ":" );
33
34     return new type(INT_TYPE, 0);
35 }
```

Por último, veremos como gerar código para uma chamada de método. O programa 12.20 mostra o método que trata desse tipo de expressão. Inicialmente, gera-se código para a primeira expressão do nó que indica qual o objeto sobre o qual a chamada será realizada. O código gerado deixa o objeto no topo da pilha.

Em seguida, gera-se código para as expressões dos argumentos, que deixa cada um deles da “esquerda para a direita” em posições sucessivas da pilha. Depois disso, acha-se a entrada correspondente ao método na tabela de símbolos e calcula-se o string que representa os tipos dos argumentos e, finalmente, gera-se a instrução *invokevirtual* que faz a chamada ao método desejado.

Programa 12.20

```

1 public type CodeGenCallNode(CallNode x)
2 {
3     EntryClass c;
4     EntryMethod m;
5     type t1, t2;
6
7     if (x == null) return null;
8
9     // pega objeto correspondente ao primeiro filho
10    t1 = CodeGenExpreNode(x.expr);
11
12    // gera código para os argumentos
13    t2 = CodeGenExpreListNode(x.args);
14
15    // procura o método desejado na classe t1.ty
16    c = (EntryClass) t1.ty;
17    m = c.nested.methodFind(x.meth.image, (EntryRec) t2.ty);
18
19    String arglist = (t2.ty == null) ? "" : t2.dscJava();
20    int removeStack = (t2.ty == null) ? 0: ((EntryRec) t2.ty).cont;
21
22    // gera chamada
23    putCode("invokevirtual " + c.completeName() + "/" + m.name + "(" +
24          arglist + ")" + m.dscJava(), -removeStack);
25
26    return new type(m.type, m.dim);
27 }
```

Essas são as expressões para as quais precisamos ver como gerar código. As demais, que não apresentaremos aqui, são as de alocação de arrays, binárias de soma/subtração e multiplicação/divisão, expressão unária, acesso a campo de um objeto e lista de expressões. Elas são bastante semelhantes a outros métodos que vimos anteriormente e, portanto, o próprio leitor será capaz de compreendê-las sem muito esforço.

12.3.3 Algumas restrições

Existem dois problemas na geração de código que apresentamos: a geração da instrução *return* no final de um método ou construtor e a contagem do tamanho da pilha utilizada pelo método.

As instruções de retorno de um método são geradas pelo nosso compilador cada vez que um nó *ReturnNode* é encontrado na árvore sintática. Além disso, sempre no final de cada método ou construtor, o compilador gera por conta própria um comando de retorno para evitar que a execução “caia fora” do método.² Quando o método retorna algum valor, o compilador tem que, além de gerar a instrução de retorno, escolher um valor para retornar. No caso, escolheram-se zero para métodos inteiros e null para strings, objetos e arrays.

Este certamente não é o melhor comportamento. O que o compilador deve fazer, caso falte um comando de retorno no final do método, é avisar ao programador do fato, emitindo um erro semântico. Além disso, é possível que o comando de retorno gerado pelo compilador seja inútil. Por exemplo, ao compilar o seguinte método

```
int MyMethod(int k)
{
    if ( k > 0 )
        return 1;
    else
        return -1;
}
```

teríamos o código gerado com um *ireturn* no final gerado de forma inútil, como vemos a seguir. Embora isso não comprometa a geração de código, é uma solução, no mínimo, pouco elegante (porém, fácil de implementar).

```
iload 1
ldc 0
if_icmpgt relexpr0
bipush 0
goto pxeler0
relexpr0:
bipush 1
pxeler0:
ifeq else1
ldc 1
ireturn
goto fil1
else1: ldc 1
ineg
ireturn
fil1: ldc 0
ireturn      ; código gerado pelo compilador.
```

²Tal erro seria apontado pela JVM ao tentarmos executar o programa.

O segundo problema relaciona-se com a contagem do número de espaços que precisamos reservar na pilha para que o método funcione. Como descrevemos, essa contagem é feita passando-se como argumento para o método *putCode* o número de posições que cada instrução adiciona à pilha (valor negativo, caso retire mais elementos do que empilhe). Vejamos, porém, o seguinte trecho de código:

```

iload 3
dup
ifeq L1:
ireturn

L1:
iload 1
...

```

Se nosso compilador gerasse código dessa maneira, teríamos um problema, visto que as instruções são geradas na mesma ordem em que aparecem no código, o que não representa necessariamente todos os fluxos de execução possíveis. Nesse caso, suponhamos que, ao emitir a instrução *ifeq L1* o contador da altura atual da pilha tenha o valor 2. Após esta instrução, terá valor 1 e após o *ireturn* terá valor 0. Ao gerar a próxima instrução, portanto, o valor do contador estará errado. Deveria ser 1, pois a execução chegou a esse ponto com a execução do *ifeq*, após o qual a pilha tem altura 1.

Um meio de contornar este problema foi impor a restrição de que toda instrução de desvio é gerada pelo compilador, de tal modo que após sua execução a pilha estará necessariamente vazia. Assim, “blocos” contíguos de código partem sempre de uma pilha sem nenhum elemento. A única exceção ocorre na geração do código para uma expressão relacional (Programa 12.19). Ali, a instrução *goto* gerada na linha 28 deixa um elemento na pilha. Em compensação, a instrução *bipush 1* na linha 30 é gerada sem contabilizar nenhum incremento na pilha, de forma que ao final do cálculo da expressão sabe-se que a pilha cresceu em exatamente uma unidade.

A maneira “correta” de solucionar esses dois problemas seria mediante da implementação de um passo a mais da análise semântica que realizasse o que se chama de análise de fluxo de controle e de fluxo de dados. Para tanto, o método é representado na forma de um grafo direcionado, em que cada vértice representa um bloco indivisível de comandos e cada aresta, as possíveis transferências de controle na sua execução. Associando-se informação sobre os pontos do grafo onde uma variável recebe um valor e pontos onde essa variável é usada, pode-se ainda localizar outros tipos de problemas, como o uso de variáveis não inicializadas. Um compilador comercial certamente deveria implementar a análise do fluxo de controle e dados.

12.4 O runtime X^{++}

O ambiente de execução para a linguagem X^{++} aproveita-se muito do ambiente da JVM. Isto significa que não temos que nos preocupar com aspectos como carregamento, verificação, inicialização da classe etc. A implementação do runtime X^{++} limita-se a um par de métodos que são chamados dentro do método *main* de uma

classe, antes e depois da execução do método *start*, que, como vimos, é o ponto de entrada de um programa *X⁺⁺*. Há, também, um par de métodos para efetuar a leitura de valores inteiros e strings e é só utilizado na implementação do comando *read*.

O runtime é implementado em Java, na classe *langXrt.Runtime*. Ao executar-se um programa *X⁺⁺*, deve-se incluir tal classe no *classpath* fornecido à JVM para que o runtime seja acessível. No programa 12.21 são mostrados os dois primeiros métodos, que fazem a inicialização e o fechamento da execução de um programa *X⁺⁺*.

Programa 12.21

```

1  public class Runtime {
2      static BufferedReader in;
3
4      static public int initialize()
5      {
6          System.out.println("Language X runtime system. Version 0.1");
7          in = new BufferedReader(new InputStreamReader(System.in));
8          if (in == null)
9          {
10              System.err.println("Error initializing X language runtime system");
11              return -1;
12          }
13          return 0;
14      }
15
16
17      static public void finilizy()
18      {
19          try
20          {
21              in.close();
22          }
23          catch (IOException e) {}
24      }

```

Na realidade, a inicialização resume-se em criar um objeto do tipo *java.io.BufferedReader* que será utilizado para ler valores da entrada-padrão nos comandos *read*. Caso não seja possível criar tal objeto, o runtime emite uma mensagem de erro e retorna o valor -1. O método *main* da classe, que chamou o método *initialize*, verifica esse valor e, caso haja um erro, termina a execução sem chamar o método *start*. O método *finilizy* simplesmente fecha o objeto *BufferedReader* criado na inicialização.

Os outros dois métodos na classe são mostrados no programa 12.22. Como citado anteriormente, eles são utilizados na geração de código do comando *read*. Um deles lê do *BufferedReader* e retorna um inteiro e o outro lê e retorna um string. Caso algum erro na leitura ocorra, o runtime emite uma mensagem de erro, mas mesmo assim retorna normalmente, com um valor-padrão. No caso de leitura de um número inteiro, caso haja erro no formato do número, então nenhuma mensagem é emitida,

apenas o valor-padrão é retornado. O valor retornado no caso de um número inteiro é 0, e no caso de string, é *null*.

Programa 12.22

```

1  static public int readInt()
2  {
3      String s = null;
4      int k;
5
6      try
7      {
8          s = in.readLine();
9          k = Integer.parseInt(s);
10     }
11     catch (IOException e)
12     {
13         System.err.println("Error reading from standard input");
14         System.err.println("Reason: " + e.getMessage());
15         return 0;
16     }
17     catch (NumberFormatException f)
18     {
19         return 0;
20     }
21     return k;
22 }
23
24
25 static public String readString()
26 {
27     String s = null;
28
29     try
30     {
31         s = in.readLine();
32     }
33     catch (IOException e)
34     {
35         System.err.println("Error reading from standard input");
36         System.err.println("Reason: " + e.getMessage());
37     }
38     return s;

```

12.5 Arquivos-fonte do compilador

As novidades neste capítulo são a inclusão dos arquivos do gerador de código e do runtime. No diretório *cap12/codgen* foram incluídos os arquivos *CodeGen.java* e *GenCodeException*. O primeiro contém os métodos descritos neste capítulo e o segundo, a definição da exceção lançada caso algum problema ocorra na geração de código. No

diretório *cap12/langXrt* foi colocado o arquivo *Runtime.java* com a implementação do runtime *X++*.

Assim, para gerar o compilador e o runtime, são necessários os seguintes passos:

```
cap12/parser$ javacc langX++.jj
Java Compiler Compiler Version 2.1 (Parser Generator)
Copyright (c) 1996-2001 Sun Microsystems, Inc.
Copyright (c) 1997-2001 WebGain, Inc.
(type "javacc" with no arguments for help)
Reading from file langX++.jj . .
File "TokenMgrError.java" does not exist. Will create one.
File "ParseException.java" does not exist. Will create one.
File "Token.java" does not exist. Will create one.
File "SimpleCharStream.java" does not exist. Will create one.
Parser generated successfully.
cap12$ javac parser/langX.java
cap12$ javac langXrt/Runtime.java
```

Na seção 12.6 veremos como compilar, gerar os arquivos de classe e executar uma “aplicação” em *X++*.

12.6 Exemplos

Utilizaremos como exemplo o programa *bintree*, que vimos no capítulo 3. Para compilar o programa e gerar código, utilizaremos o comando de sempre:

```
cap12$ java parser.langX ../ssamples/bintree.x
X++ Compiler - Version 1.0 - 2004
Reading from file ../ssamples/bintree.x . .
0 Lexical Errors found
0 Syntactic Errors found
0 Semantic error found
Generating ../ssamples/bintree.x.bintree.jas
Generating ../ssamples/bintree.x.bintree$data.jas
Code generated
```

Com esse comando foram gerados os arquivos *../ssamples/bintree.x.bintree.jas* e *../ssamples/bintree.x.bintree\$data.jas* contendo o código Jasmin para as classes *bintree* e *data*, respectivamente. Os programas 12.23 e 12.24, no final deste capítulo mostram pequenos trechos desses arquivos para que o leitor possa comparar o código gerado com o programa-fonte.

Em seguida criamos, com o auxílio do Jasmin, os arquivos “executáveis” propriamente ditos. Devemos utilizar os seguintes comandos:

```
cap12$ cd ../ssamples
ssamples$ jasmin bintree.x.bintree.jas
```

```
Generated: bintree.class  
ssamples$ jasmin bintree.x.bintree$data.jas  
Generated: bintree$data.class
```

E, finalmente, estamos prontos para executar nosso primeiro programa em X^{++} . O resultado é mostrado a seguir. Cada seqüência de três números corresponde ao dia, mês e ano que serão lidos pelo programa e a data inserida numa árvore binária. Por exemplo, a primeira data corresponde a 12/10/200. Após onze datas digitadas (datas repetidas são rejeitadas), o programa imprimirá a árvore e terminará.

```
ssamples$ java -classpath .:/cap12 bintree  
Language X runtime system. Version 0.1
```

```
12
```

```
10
```

```
2000
```

```
5
```

```
5
```

```
2001
```

```
3
```

```
4
```

```
1999
```

```
25
```

```
7
```

```
2000
```

```
25
```

```
7
```

```
2000
```

```
Elemento ja existe
```

```
3
```

```
9
```

```
1999
```

```
28
```

```
2
```

```
1999
```

```
13
```

```
10
```

```
2000
```

```
24
```

```
8
```

```
2002
```

```
25
```

```
6
```

```
2000
```

```
13
```

```
12
```

```
2000
```

```
28/2/1999
```

```
3/4/1999
```

3/9/1999
25/6/2000
25/7/2000
12/10/2000
13/10/2000
13/12/2000
5/5/2001
24/8/2002

Programa 12.23

```
1      ;-----  
2      ; Code generated by X++ compiler  
3      ; Version 0.1 - 2002  
4      ;-----  
5      .source bintree.x  
6      .class public bintree  
7      .super java/lang/Object  
8      .field public key Lbintree$data;  
9      .field public left Lbintree;  
10     .field public right Lbintree;  
11  
12     ;Default constructor. Calls super()  
13     .method public <init>()V  
14     .limit locals 1  
15     .limit stack 1  
16     aload_0  
17     invokespecial java/lang/Object/<init>()V  
18     return  
19     .end method  
20  
21     .method public <init>(Lbintree$data;)V  
22     .limit locals 2  
23  
24     aload_0  
25     invokespecial java/lang/Object/<init>()V  
26     aload_1  
27     aload_0  
28     swap  
29     putfield bintree/key Lbintree$data;  
30     aconst_null  
31     aload_0  
32     swap  
33     putfield bintree/left Lbintree;  
34     aconst_null  
35     aload_0  
36     swap  
37     putfield bintree/right Lbintree;  
38     return  
39     .limit stack 2  
40     .end method  
41     ...
```

Programa 12.24

```
1      ;-----  
2      ; Code generated by X++ compiler  
3      ; Version 0.1 - 2002  
4      ;-----  
5      .source bintree.x  
6      .class public bintree$data  
7      .super java/lang/Object  
8      .field public dia I  
9      .field public mes I  
10     .field public ano I  
11  
12     .method public <init>()V  
13     .limit locals 1  
14  
15     aload_0  
16     invokespecial java/lang/Object/<init>()V  
17     ldc 1900  
18     aload_0  
19     swap  
20     putfield bintree$data/dia I  
21     ldc 1  
22     aload_0  
23     swap  
24     putfield bintree$data/mes I  
25     ldc 1  
26     aload_0  
27     swap  
28     putfield bintree$data/ano I  
29     return  
30     .limit stack 2  
31     .end method  
32
```

seu exemplo, embora não seja esse o caso, é comum a maioria das linguagens de programação terem uma estrutura de laço de tipo while.

Portanto se adota essa definição de laço de tipo while para o nosso compilador.

Teórico que não é possível implementar os laços de tipo while e for.

Portanto os exercícios são divididos em dois tipos: os que envolvem a implementação de laços de tipo while e os que envolvem a implementação de laços de tipo for.

Apêndice A

Exercícios

Portanto os exercícios são divididos em dois tipos: os que envolvem a implementação de laços de tipo while e os que envolvem a implementação de laços de tipo for.

Neste apêndice serão propostos alguns exercícios. Preferimos relacioná-los aqui, no final do livro a de colocá-los em cada capítulo, como tradicionalmente se faz, pois a maioria deles refere-se a propostas de alterações na implementação do compilador X++ e, por isso, tais exercícios não estão relacionados a um único capítulo. Ao contrário, podem envolver alterações desde o analisador léxico até a geração de código.

A.1 Escreva alguns programas utilizando X++. Isso pode ser feito a partir dos capítulos iniciais, onde a linguagem é definida e permitirá ao leitor habituar-se com a linguagem. À medida que for progredindo no texto e construindo o compilador, o leitor provavelmente terá que corrigir os programas que escreveu e poderá “sentir” melhor a linguagem. Alguns exemplos clássicos podem ser utilizados como:

- cálculo do factorial, iterativo e recursivo;
- o problema das oito damas no tabuleiro de xadrez;
- o problema das torres de Hanói;
- implementação de classes para pilhas, filas e listas;
- solução de equações do segundo grau.

A.2 Modifique o compilador de modo que na declaração de variáveis sejam aceitas expressões de inicialização como

```
int MyMethod()
{
    int k = 20, j = d, l = k + j;
    MyClass x = null; y = new MySubclass(10);
}
```

A.3 Como vimos no capítulo 12, pode haver um problema de conflito quando o programador X++ declara no seu programa um método com o nome main, uma vez que

um método com o mesmo nome pode ser gerado pelo compilador. Verifique o que acontece, isto é, qual é comportamento

- *do Jasmin em relação a isso, ou seja, ele gera código para ambos os métodos, ou detecta o problema e não gera código?*
- *da JVM, caso o código seja gerado?*

Uma vez identificada a amplitude do problema, proponha e implemente uma forma de solucioná-lo.

A.4 Conforme mencionado na capítulo 11, o compilador ao procurar por um método na tabela de símbolos somente considera casamentos exatos entre os tipos dos argumentos e dos parâmetros formais. Defina uma política mais flexível para essa busca, usando como base a política definida para Java e implemente esta política no compilador X++.

A.5 Um dos atributos que podem ser anexados ao código Java é uma tabela que relaciona cada instrução com uma linha do programa-fonte. Tal tabela pode ser utilizada, por exemplo, por um depurador ou pela própria JVM, para indicar o ponto onde uma exceção foi lançada. Estude como gerar esta informação no Jasmin e inclua-a na geração de código.

A.6 Implemente a operação de coerção de tipo explícita, isto é, feita pelo programador.

A.7 Modifique o compilador de modo que variáveis locais definidas em escopos paralelos possam utilizar o mesmo espaço no array de variáveis locais.

A.8 Defina “this” como uma palavra reservada da linguagem, e não como uma variável que é inserida na tabela de símbolos pelo analisador semântico.

A.9 Faça com que o compilador aceite a operação de soma de um objeto com um string. A semântica dessa operação deve ser a mesma adotada em Java.

Apêndice B

JavaCC

O programa JavaCC é um gerador de analisador sintático que gera código Java. Ele foi inicialmente desenvolvido pela Sun que, depois de algum tempo passou seu desenvolvimento para a *Metamata Inc.*, que posteriormente foi incorporada pela *Webgain Inc.*. Sua versão atual é a 2.1 e encontra-se num repositório da própria Sun.

Para os interessados em conhecer melhor o JavaCC, sua página na Internet é <https://javacc.dev.java.net/>. Existem, também, uma lista de discussão e um newsgroup sobre o produto. De acordo com a própria empresa, dezenas de milhares de downloads foram feitos do JavaCC e milhares de usuários utilizam-no em aplicações "sérias". Na página, pode-se encontrar ainda um link para uma série de gramáticas prontas para utilização pelo JavaCC. São gramáticas, por exemplo, de Java, C, C++, SQL HTML, e muitas outras.

O JavaCC define uma linguagem própria para descrição, num único arquivo, do analisador léxico e do analisador sintático. Iniciando com o analisador léxico, essa linguagem permite que cada token seja definido na forma de uma expressão regular que pode ser tão simples como

```
TOKEN :  
{  
    < CLASS: "class" >  
}
```

ou mais complexa como

```
TOKEN : {  
    < string_constant: // constante string como "abcd bcda"  
        "\\""( ~["\\\"", "\\n", "\\r"])* "\\" >  
}
```

Em ambos os casos, os tokens definidos entre < > serão utilizados como constantes inteiras, acessíveis dentro do analisador sintático. Além de tokens, podem ser definidos no analisador léxico quais caracteres ou expressões devem ser ignorados (*Skip*) e

também os tokens especiais que são tokens que não são passados para o analisador sintático, mas armazenados e recuperados a partir de um token normal.

Em adição às expressões regulares, é possível definir “estados” para especificar os tokens, skips ou tokens especiais, mais ou menos como num autômato finito. No exemplo a seguir, a seqüência “/*” será ignorada pelo analisador léxico e fará este mudar para o estado *multilinecomment*. Todas as definições feitas anteriormente no estado-padrão não seriam reconhecidas com o analisador léxico neste novo estado. Somente o segundo skip a seguir poderia ser usado. Ele determina que caso a seqüência “*/” apareça na entrada, então o analisador léxico volta ao estado-padrão. E que para qualquer coisa diferente de “*/”, a entrada é apenas ignorada.

```
SKIP :
{
    "/*" : multilinecomment
}

<multilinecomment> SKIP:
{
    "/*" : DEFAULT
    | <~[]>
}
```

Um recurso extra oferecido pelo JavaCC é permitir que código Java seja associado ao reconhecimento de um determinado token. Quando o analisador léxico reconhece na entrada aquele token, além das ações normais tomadas, o analisador léxico executa o código associado a ele.

O programador deve também definir o nome da classe que irá abrigar o analisador sintático. Isso é feito por meio dos comandos PARSER_BEGIN(*xxx*) e PARSER_END(*xxx*), sendo *xxx* o nome dado ao analisador sintático. Tudo que aparece entre esses dois comandos será inserido como código nesta classe. Assim, podem ser inseridos *imports*, variáveis, métodos etc. Em particular, o método *main* que recebe os argumentos da linha de comando, instancia o analisador sintático e inicia a análise sintática pode ser incluído neste ponto.

O resto das declarações são as produções da gramática que se deseja implementar. O JavaCC utiliza a técnica descendente recursiva de análise. Nessa abordagem cada não-terminal da gramática é implementado por meio de um método que, quando chamado, procura reconhecer na entrada a estrutura do não-terminal. Não existe *backtracking*, ou seja, caso um método seja chamado para reconhecer as produções de certo não-terminal e não consiga fazê-lo, não existe possibilidade de retornar a um ponto anterior na entrada e tentar uma outra opção, ou seja, uma outra produção que possa levar ao reconhecimento da entrada.

Quando num não-terminal existem diversas possíveis produções a serem seguidas, o analisador sintático gerado pelo JavaCC toma sempre a primeira a casar com a entrada. O comportamento-padrão é considerar apenas um símbolo da entrada para decidir se o casamento ocorre ou não. O programador pode, porém, alterar esse comportamento determinando globalmente um número diferente de símbolos ou as-

sociando a um possível ponto de decisão o número de símbolos da entrada que devem ser casados para que aquele caminho seja seguido. Isto é feito por meio do comando LOOKAHEAD.

Seguindo a filosofia da análise descendente recursiva, as declarações dos não-terminais são parecidas com a declaração de um método. A diferença é que o corpo do não-terminal possui as produções descritas mediante seqüências de tokens e estruturas de repetição, escolhas ou opcionais. Um exemplo simples retirado dos exemplos do próprio JavaCC é mostrado a seguir. Nele, além da produção propriamente dita, associou-se código Java, que é executado quando ocorre o casamento da entrada, com uma parte da produção. No caso, o não-terminal *Input* é utilizado como símbolo inicial para reconhecer uma linguagem formada por pares de { e }. O não-terminal *MatchedBraces* consome um {, chame-se recursivamente, caso o próximo símbolo da entrada seja um outro { e, depois, consome o } que casa com o {. Além disso, o método criado para reconhecer o não-terminal retorna um valor inteiro que é incrementado a cada retorno da chamada. No final, tem-se na chamada mais externa o número total de pares reconhecidos.

```
void Input() :
{ int count; }
{
    count=MatchedBraces() <EOF>
    { System.out.println("The levels of nesting is " + count); }
}

int MatchedBraces() :
{ int nested_count=0; }
{
    <LBRACE> [ nested_count=MatchedBraces() ] <RBRACE>
    { return ++nested_count; }
}
```

O código associado às produções podem ser bem mais complexos. Pode-se armazenar o valor dos tokens reconhecidos e pode-se ter até mesmo blocos try/catch como em Java. No exemplo a seguir, as produções do não-terminal *methoddecl* estão dentro de um try, de modo que se algum erro sintático ocorrer (uma exceção *ParseException* é lançada), o bloco do catch é executado e, com isso, pode-se implementar alguma estratégia de recuperação de erros. No comando *return*, dentro do *try*, as informações colhidas no método são utilizadas para criar um nó da árvore sintática.

```
MethodDeclNode methoddecl(RecoverySet g) throws ParseException :
{
    Token t1 = null,
        t2 = null;
    int k = 0;
    MethodBodyNode m = null;
}
```

```

{
    try {
        ( t1 = <INT> | t1 = <STRING> | t1 = <IDENT> )
        (<LBRACKET> <RBRACKET> { k++; } )*
        t2 = <IDENT> m = methodbody(g)
        { return new MethodDeclNode(t1, k, t2, m); }
    }
    catch (ParseException e)
    {
        consumeUntil(g, e, "methoddecl");
        return new MethodDeclNode(t1, k, t2, m);
    }
}

```

O JavaCC é flexível também ao fornecer diversos argumentos para customizar o analisador sintático. Esses argumentos podem ser passados para o JavaCC ao analisar a gramática ou no início do próprio arquivo da gramática, numa sessão OPTIONS. Vejamos alguns exemplos:

- LOOKAHEAD = n: estabelece o valor global de lookahead como *n*;
- STATIC = true/false: indica se o analisador sintático gerado terá seus membros estáticos ou não. Com uma classe não estática, podem-se criar diversas instâncias do analisador sintático;
- DEBUG_PARSER = true/false: indica se deve ser gerado código que emite mensagens sobre o funcionamento do analisador sintático, como métodos chamados e não-terminais reconhecidos;
- DEBUG_TOKEN_MANAGER = true/false: idem para o analisador léxico;
- UNICODE_INPUT = true/false: indica se o analisador léxico deve aceitar código Unicode;
- IGNORE_CASE = true/false: indica se deve ignorar as diferenças entre letras maiúsculas e minúsculas.

Existem também programas que, trabalhando com o JavaCC, auxiliam na geração da árvore sintática e das classes que fazem as visitas aos nós dessa árvore. O próprio JavaCC possui o JJTree. Ele é um pré-processador para o arquivo da gramática (sem código associado às produções) que insere o código necessário para a criação da árvore sintática. O resultado é um arquivo que deve, então, ser processado pelo JavaCC. Por meio de certas anotações no arquivo original, processado pelo JJTree, o programador pode, de certo modo, customizar a criação da árvore sintática.

Outro programa parecido é o JTB (Javacc Tree Builder). Ele também pré-processa a gramática gerando um arquivo que deve ser tratado pelo JavaCC e que contém o código para criação da árvore sintática. Além disso, gera as classes correspondentes aos nós da árvore sintática e uma série de classes que implementa o padrão *Visitor* para se percorrer a árvore.

Resumindo, o JavaCC é uma ferramenta bastante poderosa, flexível e, com um pouco de prática, fácil de utilizar. Além disso, seu extenso uso pela comunidade tem disponibilizado diversas facilidades extras, como ferramentas e gramáticas, que o tornam ainda mais atrativo.

Apêndice C

Jasmin

O Jasmin (Java Assembler Interface) é um programa que permite a criação de um arquivo de classe Java a partir de um arquivo texto, no estilo de um montador para arquiteturas tradicionais. Ele acompanha o livro Java Virtual Machine de Jon Meyer e Troy Downing, publicado pela O' Reilly (<http://www.oreilly.com/catalog/javavm>). O site do projeto Jasmin, de onde foi retirada a maior parte das informações deste capítulo é www.cat.nyu.edu/meyer/jasmin.

O Jasmin possui uma sintaxe simples que reflete diretamente a maioria dos atributos de um arquivo de classe. Mesmo assim é uma ferramenta flexível e que exime o programador da tarefa de formatar e gerar do arquivo de classe. Ele também não efetua muitas checagens sobre o código montado, sendo possível gerarem-se classes que não serão aceitas pela JVM. Por exemplo, ele não efetua consistências externas como verificar se outras classes, referenciadas na classe sendo gerada, realmente existem ou estão disponíveis. Mesmo consistências internas como verificar se uma instrução é sempre alcançada com a pilha na mesma altura – característica requerida pela JVM – não são efetuadas. Por outro lado, conhecendo-se um pouco o formato do arquivo de classe e as instruções de JVM, é possível iniciar-se imediatamente na programação com o Jasmin e experimentar as suas mais diversas características, como métodos, variáveis, subrotinas, exceções etc.

Ao fazer o download do Jasmin, o leitor obtém um arquivo compactado com tudo dentro, executáveis, fonte, exemplos etc. Basta descompactar tal arquivo e incluir o seu subdiretório *bin* no caminho de busca de programas do seu sistema operacional e (quase) pronto. Quase porque o programa é escrito em Java e depende de uma JVM para executá-lo.

Para fazer funcionar o Jasmin, uma vez inserido no caminho de pesquisa de programas, basta executar

```
> jasmin myfile.jas
```

O comando *jasmin* é, na verdade, um arquivo de script que invoca a JVM com os devidos argumentos para executar a classe *jasmim.Main* e processar o arquivo

desejado. Esse script vem em dois “sabores”. Um para ser executado no DOS/Window e outro para ser executado pelo *CShell* no linux ou SunOS (ou qualquer outro sistema que suporte o ambiente Java e o *CShell*).

Em termos de argumentos é também bastante simples e o usuário pode apenas fornecer qual o diretório em que o arquivo de classe deve ser gerado. Por exemplo:

```
> jasmin -d ../classes myfile.j
```

vai colocar o arquivo de saída dentro do diretório *../classes*, respeitando o diretório que representa o pacote da classe gerada. Por exemplo, se a classe a ser gerada é *mypackage.MyClass*, então o arquivo de saída *MyClass.class* será gravado no diretório *../classes/mypackage*.

Os “comandos” num fonte Jasmin são divididos em três categorias:

- diretivas;
- instruções;
- rótulos.

As diretivas fornecem informações sobre a classe, nem sempre relacionadas com o código propriamente dito. A declaração de uma diretiva inicia-se sempre com o nome da diretiva seguido de zero ou mais argumentos. As diretivas aceitas pelo Jasmin são:

.catch	.class	.end	.field
.implements	.interface	.limit	.line
.method	.source	.super	.throws
.var			

Vejamos alguns exemplos:

```
.class mypackage/MyClass
.super java/lang/Object
```

Essas duas diretivas são utilizadas uma vez em cada arquivo e especificam, respectivamente, o nome do arquivo/classe a ser gerado e qual a sua superclasse.

```
.method public myMethod(Ljava/util/Vector;)V
.end method
```

Estas duas diretivas são utilizadas em pares, para marcar o início e o final da declaração de um método. A primeira define, além do nome do método, seus atributos e assinatura. Tudo o que vier entre essas diretivas diz respeito ao método, como, por exemplo, as diretivas

```
.limit stack 10
.limit locals 3
```

que identificam o número máximo de posições da pilha de execução e o número de variáveis locais que o método utiliza. Em particular, dentro de um método, aparece o segundo tipo de comando que são as instruções. Elas espelham as instruções definidas para a JVM. A seqüência em que as instruções aparecem no arquivo-fonte Jasmin reflete a seqüência que as instruções serão geradas no array de bytes do arquivo de classe. Os nomes aceitos pelo Jasmin são aqueles utilizados na definição da JVM, assim como os parâmetros que cada instrução pode aceitar. Exemplos de instruções:

```
ldc      "Hello World"
iinc    1 -1
bipush  10
```

E o último tipo de comando aceito são os rótulos. Eles são também utilizados sempre dentro de métodos e servem para identificar pontos específicos (deslocamentos) dentro do array de bytes que compõe cada método. Por exemplo:

```
ldc      "Hello World"
Label1:
iinc    1 -1
bipush  10
```

Neste caso, o rótulo Label1: identifica o deslocamento dentro do array de bytes do método onde se inicia o código da instrução *iinc*. Os rótulos são úteis quando utilizados, por exemplo, como alvo de comandos de desvio como

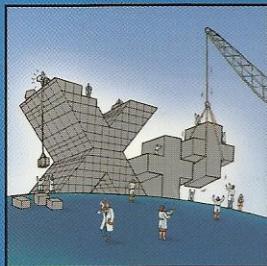
```
ifeq    Label1
ldc     "Alo Mundo"
goto   Label2
Label1:
ldc     "Hello World"
Label2:
areturn
```

Diversas outras características podem ser definidas para a classe, como:

- associação de um número de linha do código-fonte com um bloco de instruções por meio da diretiva *.line*;
- definição de um nome para uma variável local num determinado trecho do código, por meio da diretiva *.var*;
- definição de uma determinada instrução como um *exception handler* para determinado bloco de código por meio da diretiva *catch*.

Assim, o Jasmin é uma ferramenta bastante interessante e recomenda-se a leitura mais cuidadosa da documentação no site do projeto. Existem ainda algumas outras ferramentas que podem interagir com o Jasmin. É o caso, por exemplo, do *Class Vista*

que é um disassembler de arquivos de classe. Ele pode produzir arquivos texto, no formato definido pelo Jasmin, a partir de um arquivo de classe. O *ClassVista* pode ser encontrado em diversos sites de download grátis na Internet, como, por exemplo. <http://www.portaljava.com>.



Como Construir um Compilador

Utilizando Ferramentas Java

A maioria dos livros sobre compiladores aborda o assunto sob um ponto de vista genérico, com evidente embasamento teórico, fornecendo uma visão geral das técnicas e ferramentas utilizadas para a construção de um compilador.

Este livro tem um enfoque diferenciado, mostrando em detalhes cada uma das etapas do desenvolvimento de um compilador, utilizando ferramentas Java e técnicas específicas. Pode ser utilizado como texto de apoio em cursos de construção de compiladores, conferindo-lhes um caráter mais prático.

Também atende aos profissionais de desenvolvimento de software que necessitam aperfeiçoar ou atualizar seus conhecimentos, uma vez que as informações aqui contidas podem ser utilizadas no desenvolvimento de diversos tipos de aplicações, não se limitando apenas à área acadêmica nem à construção de compiladores propriamente dita.

Cada capítulo do livro apresenta uma etapa do processo de compilação, discutindo as técnicas e ferramentas para a implementação de um compilador para uma linguagem orientada a objetos denominada X++, utilizada como estudo de caso. Não somente o compilador é criado com ferramental baseado na linguagem Java, mas também tem como plataforma-alvo a Máquina Virtual Java (JVM).

Tópicos abordados no livro:

- Introdução à compilação.
- Descrição da linguagem X++, utilizada como estudo de caso. Apresenta conceitos básicos sobre linguagens e como defini-las.
- Análise léxica. Mostra como se utiliza a ferramenta JavaCC para criar um analisador léxico para a linguagem X++.
- Análise sintática. Mostra como construir um analisador sintático para X++ utilizando JavaCC.
- Árvore sintática. Mostra como construir a árvore sintática usando o JavaCC e como utilizá-la nas etapas seguintes.
- Tabela de símbolos e análise semântica. Mostra como implementar uma tabela de símbolos utilizada na checagem de tipos e validação semântica.
- Geração de código. Mostra como utilizar a ferramenta Jasmin para gerar o código-objeto (bytecode) Java.

TODO O CÓDIGO-FONTE UTILIZADO NO LIVRO ESTÁ DISPONÍVEL PARA DOWNLOAD NO SITE DA NOVATEC.

■ SOBRE O AUTOR

Márcio Eduardo Delamaro é graduado em Ciência da Computação, com mestrado e doutorado em Engenharia de Software. Durante o doutorado, trabalhou na Purdue University, nos Estados Unidos. Fez pós-doutorado no Politecnico di Milano e na Università degli Studi di Milano, na Itália. Foi professor na Universidade Federal de Mato Grosso do Sul (UFMS) e Universidade Estadual de Maringá (UEM). Atualmente leciona disciplinas relacionadas a Linguagens Formais, Paradigmas de Linguagens de Programação e Engenharia de Software nos cursos do Centro Universitário Eurípides de Marília (Univem). O autor pode ser contatado através do e-mail delamaro@novateceditora.com.br.

ISBN 85-7522-055-1