APÊNDICE

A

UM *FRONT-END* COMPLETO

O *front-end* completo do compilador neste apêndice é baseado no compilador simples descrito informalmente nas seções 2.5 a 2.8. A principal diferença do Capítulo 2 é que o *front-end* gera código de desvio para expressões boolianas, como na Seção 6.6. Começamos com a sintaxe da linguagem fonte, descrita por uma gramática que precisa ser adaptada para a análise descendente.

O código Java para o tradutor consiste em cinco pacotes: `main`, `lexer`, `symbol`, `parser` e `inter`. O pacote `inter` contém classes para as construções da linguagem na sintaxe abstrata. Como o código para o analisador sintático interage com o restante dos pacotes, ele será discutido mais tarde. Cada pacote é armazenado como um diretório separado com um arquivo por classe.

Entrando no analisador sintático, o programa fonte consiste em um fluxo de tokens, de modo que a orientação por objeto tem pouco a fazer em relação ao código para o analisador sintático. Saindo do analisador sintático, o programa fonte consiste em uma árvore de sintaxe, com construções ou nós implementados como objetos. Esses objetos tratam de tudo o que segue: construir um nó da árvore sintática, verificar tipos e gerar código intermediário com três endereços (ver o pacote `inter`).

Orientado por objeto versus orientado por fase

Com uma abordagem orientada por objeto, todo o código para uma construção é coletado na classe para a construção. Alternativamente, com uma abordagem orientada por fase, o código é agrupado por fase, de modo que um procedimento de verificação de tipo teria um *case* para cada construção, e um procedimento de geração de código teria um *case* para cada construção, e assim por diante.

A dificuldade na escolha é que uma abordagem orientada por objeto facilita a mudança ou a inclusão de uma construção, como comandos 'for', e uma abordagem orientada por fase facilita a mudança ou a inclusão de uma fase, como a verificação de tipo. Com objetos, uma construção nova pode ser acrescentada pela escrita de uma classe autocontida, mas uma mudança em uma fase, como a inserção de código para coerções, exige mudanças em todas as classes afetadas. Com fases, uma construção nova pode gerar mudanças entre os procedimentos para as fases.

A.1 A LINGUAGEM FONTE

Um programa na linguagem consiste em um bloco com declarações e comandos opcionais. O token **basic** representa os tipos básicos.

program	→	*block*
block	→	{ *decls stmts* }
decls	→	*decls decl* | ϵ
decl	→	*type* **id** ;
type	→	*type* [**num**] | **basic**
stmts	→	*stmts stmt* | ϵ

Tratar atribuições como comandos, em vez de operadores dentro de expressões, simplifica a tradução.

stmt → *loc* = *bool* ;
| **if** (*bool*) *stmt*
| **if** (*bool*) *stmt* **else** *stmt*
| **while** (*bool*) *stmt*
| **do** *stmt* **while** (*bool*) ;
| **break** ;
| *block*

loc → *loc* [*bool*] | **id**

As produções para as expressões tratam da associatividade e precedência de operadores. Elas usam um não-terminal para cada nível de precedência e um não-terminal, *factor*, para as expressões entre parênteses, identificadores, referências de arranjos e constantes.

bool → *bool* || *join* | *join*
join → *join* && *equality* | *equality*
equality → *equality* == *rel* | *equality* != *rel* | *rel*
rel → *expr* < *expr* | *expr* <= *expr* | *expr* >= *expr* |
expr > *expr* | *expr*
expr → *expr* + *term* | *expr* − *term* | *term*
term → *term* * *unary* | *term* / *unary* | *unary*
unary → ! *unary* | − *unary* | *factor*
factor → (*bool*) | *loc* | **num** | **real** | **true** | **false**

A.2 MAIN

A execução começa no método `main` da classe `Main`. O método `main` cria um analisador léxico e um analisador sintático e, depois, chama o método `program` no analisador sintático:

```
 1) package main;          // Arquivo Main.java
 2) import java.io.*; import lexer.*; import parser.*;
 3) public class Main {
 4)     public static void main(String[] args) throws IOException {
 5)         Lexer lex = new Lexer();
 6)         Parser parse = new Parser(lex);
 7)         parse.program();
 8)         System.out.write('\n');
 9)     }
10) }
```

A.3 ANALISADOR LÉXICO

O pacote `lexer` é uma extensão do código para o analisador léxico da Seção 2.6.5. A classe `Tag` define constantes para tokens:

```
1) package lexer;          // Arquivo Tag.java
2) public class Tag {
3)   public final static int
4)       AND    = 256,   BASIC  = 257,   BREAK  = 258,   DO    = 259,   ELSE  = 260,
5)       EQ     = 261,   FALSE  = 262,   GE     = 263,   ID    = 264,   IF    = 265,
6)       INDEX  = 266,   LE     = 267,   MINUS  = 268,   NE    = 269,   NUM   = 270,
7)       OR     = 271,   REAL   = 272,   TEMP   = 273,   TRUE  = 274,   WHILE = 275;
8) }
```

Três das constantes, `INDEX`, `MINUS` e `TEMP`, não são tokens léxicos; elas serão usadas nas árvores sintáticas.

As classes `Token` e `Num` são como na Seção 2.6.5, com o método `toString` acrescentado:

```
1) package lexer;          // Arquivo Token.java
2) public class Token {
3)    public final int tag;
4)    public Token(int t) { tag = t; }
5)    public String toString() {return "" + (char)tag;}
6) }
```

```
1) package lexer;          // Arquivo Num.java
2) public class Num extends Token {
3)    public final int value;
4)    public Num(int v) { super(Tag.NUM); value = v; }
5)    public String toString() { return "" + value; }
6) }
```

A classe `Word` gerencia lexemas para palavras reservadas, identificadores e tokens compostos como `&&`. Ela também é útil para gerenciar a forma escrita dos operadores no código intermediário, como o menos unário; por exemplo, o texto fonte `-2` tem a forma intermediária `minus 2`.

```
1) package lexer;          // Arquivo Word.java
2) public class Word extends Token {
3)    public String lexeme = "";
4)    public Word(String s, int tag)  {super(tag); lexeme = s; }
5)    public String toString() { return lexeme; }
6)    public static final Word
7)       and = new Word( "&&", Tag.AND ),  or = new Word( "||", Tag.OR ),
8)       eq  = new Word( "==", Tag.EQ  ),  ne = new Word( "!=", Tag.NE ),
9)       le  = new Word( "<=", Tag.LE  ),  ge = new Word( ">=", Tag.GE ),
10)       minus  = new Word( "minus", Tag.MINUS ),
11)       True   = new Word( "true",  Tag.TRUE  ),
12)       False  = new Word( "false", Tag.FALSE ),
13)       temp   = new Word( "t",     Tag.TEMP  );
14) }
```

A classe `Real` é para números de ponto flutuante:

```
1) package lexer;          // Arquivo Real.java
2) public class Real extends Token {
3)    public final float value;
4)    public Real(float v) { super(Tag.REAL); value = v; }
5)    public String toString(){ return "" + value; }
6) }
```

O método principal na classe Lexer, função `scan`, reconhece números, identificadores e palavras reservadas, conforme discutimos na Seção 2.6.5.

As linhas 9-13 na classe `Lexer` reservam palavras-chave selecionadas. As linhas 14-16 reservam lexemas para objetos definidos em outras partes. Os objetos `Word.True` e `Word.False` são definidos na classe `Word`. Os objetos para os tipos básicos `int`, `char`, `bool` e `float` são definidos na classe `Type`, uma subclasse de `Word`. A classe `Type` é faz parte do pacote `symbols`.

```
1) package lexer;          // Arquivo Lexer.java
2) import java.io.*; import java.util.*; import symbols.*;
3) public class Lexer {
4)    public static int line = 1;
5)    char peek = ' ';
6)    Hashtable words = new Hashtable();
7)    void reserve(Word w) { words.put(w.lexeme, w); }
8)    public Lexer() {
9)       reserve( new Word("if",    Tag.IF)    );
10)      reserve( new Word("else",  Tag.ELSE)  );
11)      reserve( new Word("while", Tag.WHILE) );
12)      reserve( new Word("do",    Tag.DO)    );
13)      reserve( new Word("break", Tag.BREAK) );
```

```
14)        reserve( Word.True );  reserve( Word.False );
15)        reserve( Type.Int  );  reserve( Type.Char  );
16)        reserve( Type.Bool );  reserve( Type.Float );
17)    }
```

A função `readch()` (linha 18) é usada para ler o próximo caractere de entrada na variável `peek`. O nome `readch` é reutilizado ou sobrecarregado (linhas 19-24) para auxiliar a reconhecer tokens compostos. Por exemplo, quando a entrada < é vista, a chamada `readch('=')` lê o próximo caractere em `peek` e verifica se é =.

```
18)    void readch() throws IOException { peek = (char)System.in.read(); }
19)    boolean readch(char c) throws IOException {
20)        readch();
21)        if( peek != c ) return false;
22)        peek ='';
23)        return true;
24)    }
```

A função `scan` começa ignorando espaços em branco (linhas 26-30). Ela reconhece tokens compostos como <= (linhas 31-44) e números como `365` e `3.14` (linhas 45-58), antes de reconhecer palavras (linhas 59-70).

```
25)    public Token scan() throws IOException {
26)        for( ; ; readch() ) {
27)            if( peek == '' || peek == '\t' ) continue;
28)            else if( peek == '\n' ) line = line + 1;
29)            else break;
30)        }
31)        switch( peek ) {
32)        case '&':
33)            if( readch('&') ) return Word.and;  else return new Token('&');
34)        case'|':
35)            if( readch('|') ) return Word.or;   else return new Token('|');
36)        case'=':
37)            if( readch('=') ) return Word.eq;   else return new Token('=');
38)        case'!':
39)            if( readch('=') ) return Word.ne;   else return new Token('!');
40)        case'<':
41)            if( readch('=') ) return Word.le;   else return new Token('<');
42)        case'>':
43)            if( readch('=') ) return Word.ge;   else return new Token('>');
44)        }
45)        if( Character.isDigit(peek) ) {
46)            int v = 0;
47)            do {
48)                v = 10*v + Character.digit(peek, 10); readch();
49)            } while( Character.isDigit(peek) );
50)            if( peek != '.' ) return new Num(v);
51)            float x = v; float d = 10;
52)            for(;;) {
53)                readch();
54)                if( ! Character.isDigit(peek) ) break;
55)                x = x + Character.digit(peek, 10) / d; d = d*10;
56)            }
57)            return new Real(x);
58)        }
59)        if( Character.isLetter(peek) ) {
60)            StringBuffer b = new StringBuffer();
61)            do {
62)                b.append(peek); readch();
63)            } while( Character.isLetterOrDigit(peek) );
64)            String s = b.toString();
65)            Word w = (Word)words.get(s);
66)            if( w != null ) return w;
67)            w = new Word(s, Tag.ID);
```

```
68)            words.put(s, w);
69)            return w;
70)        }
```

Finalmente, quaisquer caracteres restantes são retornados como tokens (linhas 71-72).

```
71)        Token tok = new Token(peek); peek = '';
72)        return tok;
73)    }
74) }
```

A.4 TABELAS DE SÍMBOLOS E TIPOS

O pacote `symbols` implementa as tabelas de símbolos e tipos.

A classe `Env` é basicamente inalterada da Figura 2.37. Enquanto a classe `Lexer` mapeia cadeias em palavras, a classe `Env` mapeia tokens de palavra a objetos da classe `Id`, que é definida no pacote `inter` com as classes para as expressões e comandos.

```
1) package symbols;          // Arquivo Env.java
2) import java.util.*; import lexer.*; import inter.*;
3) public class Env {
4)    private Hashtable table;
5)    protected Env prev;
6)    public Env(Env n) { table = new Hashtable(); prev = n; }
7)    public void put(Token w, Id i) { table.put(w, i); }
8)    public Id get(Token w) {
9)       for( Env e = this; e != null; e = e.prev ) {
10)         Id found = (Id)(e.table.get(w));
11)         if( found != null ) return found;
12)      }
13)      return null;
14)   }
15) }
```

Definimos a classe `Type` como sendo uma subclasse de `Word`, porque os nomes de tipo básicos como `int` são simplesmente palavras reservadas, a serem mapeadas de lexemas para objetos apropriados pelo analisador léxico. Os objetos para os tipos básicos são `Type.Int`, `Type.Float`, `Type.Char` e `Type.Bool` (linhas 7-10). Todos eles têm o campo herdado `tag` definido como `Tag.BASIC`, de modo que o analisador sintático os trata da mesma forma.

```
1) package symbols;          // Arquivo Type.java
2) import lexer.*;
3) public class Type extends Word {
4)    public int width = 0;            // width é usado para alocação de memória
5)    public Type(String s, int tag, int w) { super(s, tag); width = w; }
6)    public static final Type
7)       Int   = new Type( "int",   Tag.BASIC, 4 ),
8)       Float = new Type( "float", Tag.BASIC, 8 ),
9)       Char  = new Type( "char",  Tag.BASIC, 1 ),
10)      Bool  = new Type( "bool",  Tag.BASIC, 1 );
```

As funções `numeric` (linhas 11-14) e `max` (linhas 15-20) são úteis para as conversões de tipo.

```
11)   public static boolean numeric(Type p) {
12)      if (p == Type.Char || p == Type.Int || p == Type.Float) return true;
13)      else return false;
14)   }
15)   public static Type max(Type p1, Type p2 ) {
16)      if ( ! numeric(p1) || ! numeric(p2) ) return null;
17)      else if ( p1 == Type.Float || p2 == Type.Float ) return Type.Float;
18)      else if ( p1 == Type.Int   || p2 == Type.Int   ) return Type.Int;
19)      else return Type.Char;
20)   }
21) }
```

As conversões são permitidas entre os tipos 'numéricos' `Type.Char`, `Type.Int` e `Type.Float`. Quando um operador aritmético é aplicado a dois tipos numéricos, o resultado é o 'max' dos dois tipos.

Os arranjos são o único tipo construído na linguagem fonte. A chamada para `super` na linha 7 define o campo `width`, que é essencial para os cálculos de endereço. Ela também define `lexeme` e `tok` para valores default que não são usados.

```
 1) package symbols;          // Arquivo Array.java
 2) import lexer.*;
 3) public class Array extends Type {
 4)     public Type of;                    // arranjo *of* type
 5)     public int size = 1;               // número de elementos
 6)     public Array(int sz, Type p) {
 7)         super("[]", Tag.INDEX, sz*p.width); size = sz;  of = p;
 8)     }
 9)     public String toString() { return "[" + size + "] " + of.toString(); }
10) }
```

A.5 CÓDIGO INTERMEDIÁRIO PARA EXPRESSÕES

O pacote `inter` contém a hierarquia de classe `Node`. `Node` possui duas subclasses: `Expr` para nós de expressão e `Stmt` para nós de comando. Esta seção apresenta `Expr` e suas subclasses. Alguns dos métodos em `Expr` tratam boolianos e código de desvio; eles serão discutidos na Seção A.6, com as subclasses restantes de `Expr`.

Os nós na árvore sintática são implementados como objetos da classe `Node`. Para o relato de erros, o campo `lexline` (linha 4, arquivo *Node.java*) guarda o número da linha fonte da construção nesse nó. As linhas 7-10 são usadas para emitir código de três endereços.

```
 1) package inter;          // Arquivo Node.java
 2) import lexer.*;
 3) public class Node {
 4)     int lexline = 0;
 5)     Node() { lexline = Lexer.line; }
 6)     void error(String s) { throw new Error("near line "+lexline+": "+s); }
 7)     static int labels = 0;
 8)     public int newlabel() { return ++labels; }
 9)     public void emitlabel(int i) { System.out.print("L" + i + ":"); }
10)     public void emit(String s) { System.out.println("\t" + s); }
11) }
```

Construções de expressão são implementadas pelas subclasses de `Expr`. A classe `Expr` possui campos `op` e `type` (linhas 4-5, arquivo *Expr.java*), representando o operador e tipo, respectivamente, em um nó.

```
1) package inter;          // Arquivo Expr.java
2) import lexer.*; import symbols.*;
3) public class Expr extends Node {
4)     public Token op;
5)     public Type type;
6)     Expr(Token tok, Type p) { op = tok; type = p; }
```

O método `gen` (linha 7) retorna um 'termo' que pode caber no lado direito de um comando de três endereços. Dada a expressão $E = E_1 + E_2$, o método `gen` retorna um termo $x_1 + x_2$, onde x_1 e x_2 são endereços para os valores de E_1 e E_2, respectivamente. O valor de retorno `this` é apropriado se esse objeto for um endereço; as subclasses de `Expr` tipicamente reimplementam `gen`.

O método `reduce` (linha 8) calcula ou 'reduz' uma expressão a um único endereço, ou seja, retorna uma constante, um identificador ou um nome temporário. Dada a expressão E, o método `reduce` retorna um temporário t contendo o valor de E. Novamente, `this` é um valor de retorno apropriado se esse objeto for um endereço.

Adiamos a discussão dos métodos `jumping` e `emitjumps` (linhas 9-18) até a Seção A.6; eles geram código de desvio para expressões boolianas.

```
 7)    public Expr gen() { return this; }
 8)    public Expr reduce() { return this; }
 9)    public void jumping(int t, int f) { emitjumps(toString(), t, f); }
10)    public void emitjumps(String test, int t, int f) {
11)       if( t != 0 && f != 0 ) {
12)          emit("if " + test + " goto L" + t);
13)          emit("goto L" + f);
14)       }
15)       else if( t != 0 ) emit("if " + test + " goto L" + t);
16)       else if( f != 0 ) emit("iffalse " + test + " goto L" + f);
17)       else ; // nada, porque ambos t e f fall through
18)    }
19)    public String toString() { return op.toString(); }
20) }
```

A classe `Id` herda as implementações default de `gen` e `reduce` na classe `Expr`, porque um identificador é um endereço.

```
1) package inter;          // Arquivo Id.java
2) import lexer.*; import symbols.*;
3) public class Id extends Expr {
4)    public int offset;      // endereço relativo
5)    public Id(Word id, Type p, int b) { super(id, p); offset = b; }
6) }
```

O nó para um identificador da classe `Id` é uma folha. A chamada `super(id,p)` (linha 5, arquivo *Id.java*) guarda `id` e `p` nos campos herdados `op` e `type`, respectivamente. O campo `offset` (linha 4) contém o endereço relativo desse identificador.

A classe `Op` oferece uma implementação de `reduce` (linhas 5-10, arquivo *Op.java*) que é herdada pelas subclasses `Arith` para operadores aritméticos, `Unary` para operadores unários, e `Access` para acessos a arranjo. Em cada caso, `reduce` chama `gen` para gerar um termo, emite uma instrução para atribuir o termo a um novo nome temporário, e retorna o temporário.

```
1) package inter;          // Arquivo Op.java
2)import lexer.*; import symbols.*;
3) public class Op extends Expr {
4)    public Op(Token tok, Type p)  { super(tok, p); }
5)    public Expr reduce() {
6)       Expr x = gen();
7)       Temp t = new Temp(type);
8)       emit( t.toString() + " = " + x.toString() );
9)       return t;
10)   }
11) }
```

A classe `Arith` implementa operadores binários como `+` e `*`. O construtor `Arith` começa chamando `super(tok,null)` (linha 6), onde `tok` é um token representando o operador e `null` é um marcador de lugar para o tipo. O tipo é determinado na linha 7 usando `Type.max`, que verifica se os dois operandos podem ser convertidos para um tipo numérico comum; o código para `Type.max` está na Seção A.4. Se eles puderem ser convertidos, `type` é definido como o tipo do resultado; caso contrário, um erro de tipo é informado (linha 8). Esse compilador simples verifica tipos, mas não insere conversões de tipo.

```
1) package inter;          // Arquivo Arith.java
2) import lexer.*; import symbols.*;
3) public class Arith extends Op {
4)    public Expr expr1, expr2;
5)    public Arith(Token tok, Expr x1, Expr x2)  {
6)       super(tok, null); expr1 = x1; expr2 = x2;
7)       type = Type.max(expr1.type, expr2.type);
8)       if (type == null ) error("type error");
9)    }
10)   public Expr gen() {
11)      return new Arith(op, expr1.reduce(), expr2.reduce());
12)   }
13)   public String toString() {
```

```
14)         return expr1.toString()+" "+op.toString()+" "+expr2.toString();
15)     }
16) }
```

O método `gen` constrói o lado direito de uma instrução de três endereços, reduzindo as subexpressões para endereços e aplicando o operador aos endereços (linha 11, arquivo *Arith.java*). Por exemplo, suponha que `gen` seja chamado na raiz para `a+b*c`. As chamadas a `reduce` retornam `a` como o endereço para a subexpressão `a` e um temporário `t` como endereço para b*c. Enquanto isso, `reduce` emite a instrução `t=b*c`. O método `gen` retorna um novo nó `Arith`, com o operador `*` e endereços `a` e `t` como operandos.[1]

Vale a pena observar que os nomes temporários têm tipo como todas as outras expressões. O construtor `Temp`, portanto, é chamado com um tipo como parâmetro (linha 6, arquivo *Temp.java*).[2]

```
1) package inter;          // Arquivo Temp.java
2) import lexer.*; import symbols.*;
3) public class Temp extends Expr {
4)     static int count = 0;
5)     int number = 0;
6)     public Temp(Type p) { super(Word.temp, p); number = ++count; }
7)     public String toString() { return "t" + number; }
8) }
```

A classe `Unary` é o correspondente de um operando da classe `Arith`:

```
 1) package inter;          // Arquivo Unary.java
 2) import lexer.*; import symbols.*;
 3) public class Unary extends Op {
 4)     public Expr expr;
 5)     public Unary(Token tok, Expr x) {     // trata operador menos, para !, ver Not
 6)         super(tok, null);  expr = x;
 7)         type = Type.max(Type.Int, expr.type);
 8)         if (type == null ) error("type error");
 9)     }
10)     public Expr gen() { return new Unary(op, expr.reduce()); }
11)     public String toString() { return op.toString()+" "+expr.toString(); }
12) }
```

A.6 CÓDIGO DE DESVIO PARA EXPRESSÕES BOOLIANAS

O código de desvio para uma expressão booliana *B* é gerado pelo método `jumping`, que recebe dois rótulos `t` e `f` como parâmetros, chamados respectivamente de saídas verdadeira e falsa de *B*. O código contém um desvio para `t` se *B* for avaliado como verdadeiro, e um desvio para f se *B* for avaliado como falso. Por convenção, o rótulo especial `0` significa que o controle passa por *B* em direção a próxima instrução após o código de *B*.

Começamos com a classe `Constant`. O construtor `Constant` na linha 4 recebe um token `tok` e um tipo `p` como parâmetros. Ele constrói uma folha na árvore de sintaxe com o rótulo `tok` e tipo `p`. Por conveniência, o construtor `Constant` é sobrecarregado (linha 5) para criar um objeto constante a partir de um inteiro.

```
1) package inter;          // Arquivo Constant.java
2) import lexer.*; import symbols.*;
3) public class Constant extends Expr {
4)     public Constant(Token tok, Type p) { super(tok, p); }
5)     public Constant(int i) { super(new Num(i), Type.Int); }
6)     public static final Constant
7)         True  = new Constant(Word.True,  Type.Bool),
8)         False = new Constant(Word.False, Type.Bool);
```

1 Para o relato de erro, o campo `lexline` na classe Node registra o número corrente da linha léxica quando um nó é construído. Deixamos para o leitor a tarefa de acompanhar os números de linha quando novos nós forem construídos durante a geração de código intermediário.

2 Uma técnica alternativa poderia ser para o construtor receber um nó de expressão como parâmetro, de modo que possa copiar o tipo e a posição léxica do nó de expressão.

```
 9)    public void jumping(int t, int f) {
10)       if ( this == True && t != 0 ) emit("goto L" + t);
11)       else if ( this == False && f != 0) emit("goto L" + f);
12)    }
13) }
```

O método `jumping` (linhas 9-12, arquivo *Constant.java*) utiliza dois parâmetros, rotulados com `t` e `f`. Se essa constante for o objeto estático `True` (definido na linha 7) e `t` não for o rótulo especial `0`, então um desvio para `t` é gerado. Caso contrário, se esse for o objeto `False` (definido na linha 8) e `f` for diferente de zero, então um desvio para `f` é gerado.

A classe `Logical` oferece alguma funcionalidade comum para as classes `Or`, `And` e `Not`. Os campos `x` e `y` (linha 4) correspondem aos operandos de um operador lógico. (Embora a classe `Not` implemente um operador unário, por conveniência, ela é uma subclasse de `Logical`.) O construtor `Logical(tok,a,b)` (linhas 5-10) constrói um nó de sintaxe com operador `tok` e operandos `a` e b. Ao fazer isso, ele usa a função `check` para garantir que tanto `a` quanto `b` sejam boolianos. O método `gen` será discutido no fim desta seção.

```
1) package inter;          // Arquivo Logical.java
2) import lexer.*; import symbols.*;
3) public class Logical extends Expr {
4)    public Expr expr1, expr2;
5)    Logical(Token tok, Expr x1, Expr x2) {
6)       super(tok, null);                        // tipo nulo para começar
7)       expr1 = x1; expr2 = x2;
8)       type = check(expr1.type, expr2.type);
9)       if (type == null ) error("type error");
10)   }
11)   public Type check(Type p1, Type p2) {
12)      if ( p1 == Type.Bool && p2 == Type.Bool ) return Type.Bool;
13)      else return null;
14)   }
15)   public Expr gen() {
16)      int f = newlabel(); int a = newlabel();
17)      Temp temp = new Temp(type);
18)      this.jumping(0,f);
19)      emit(temp.toString() + " = true");
20)      emit("goto L" + a);
21)      emitlabel(f); emit(temp.toString() + " = false");
22)      emitlabel(a);
23)      return temp;
24)   }
25)   public String toString() {
26)      return expr1.toString()+" "+op.toString()+" "+expr2.toString();
27)   }
28) }
```

Na classe `Or`, o método `jumping` (linhas 5-10) gera código de desvio para uma expressão booliana $B = B_1 \,||\, B_2$. Por um momento, suponha que nem a saída verdadeira `t` nem a saída falsa `f` de B seja o rótulo especial `0`. Como B é verdadeiro se B_1 é verdadeiro, a verdadeira saída de B_1 deve ser `t`, e a saída falsa corresponde à primeira instrução de B_2. As saídas verdadeira e falsa de B_2 são as mesmas de B.

```
1) package inter;          // Arquivo Or.java
2) import lexer.*; import symbols.*;
3) public class Or extends Logical {
4)    public Or(Token tok, Expr x1, Expr x2) { super(tok, x1, x2); }
5)    public void jumping(int t, int f) {
6)       int label = t != 0 ? t : newlabel();
7)       expr1.jumping(label, 0);
8)       expr2.jumping(t,f);
9)       if( t == 0 ) emitlabel(label);
10)   }
11) }
```

No caso geral, `t`, a verdadeira saída de *B* pode ser o rótulo especial `0`. A variável `label` (linha 6, arquivo *Or.java*) garante que a saída verdadeira de B_1 seja definida corretamente com o fim do código para *B*. Se `t` for `0`, então `label` é definido como um novo rótulo que é emitido após a geração de código para B_1 e B_2.

O código para a classe `And` é semelhante ao código para `Or`.

```
1) package inter;          // Arquivo And.java
2) import lexer.*; import symbols.*;
3) public class And extends Logical {
4)     public And(Token tok, Expr x1, Expr x2) { super(tok, x1, x2); }
5)     public void jumping(int t, int f) {
6)         int label = f != 0 ? f : newlabel();
7)         expr1.jumping(0, label);
8)         expr2.jumping(t,f);
9)         if( f == 0 ) emitlabel(label);
10)    }
11) }
```

A classe `Not` tem tanto em comum com os outros operadores boolianos que a fazemos uma subclasse de `Logical`, embora `Not` implemente um operador unário. A superclasse espera dois operandos, de modo que `b` aparece duas vezes na chamada a `super` na linha 4. Somente `y` (declarado na linha 4, arquivo *Logical.java*) é usado nos métodos das linhas 5-6. Na linha 5, o método `jumping` simplesmente chama `y.jumping` com as saídas verdadeira e falsa invertidas.

```
1) package inter;          // Arquivo Not.java
2) import lexer.*; import symbols.*;
3) public class Not extends Logical {
4)     public Not(Token tok, Expr x2) { super(tok, x2, x2); }
5)     public void jumping(int t, int f) { expr2.jumping(f, t); }
6)     public String toString() { return op.toString()+" "+expr2.toString(); }
7) }
```

A classe `Rel` implementa os operadores <, <=, ==, !=, >= e >. A função `check` (linhas 5-9) verifica se os dois operandos têm o mesmo tipo e se não são arranjos. Para simplificar, as coerções não são permitidas.

```
1) package inter;          // Arquivo Rel.java
2) import lexer.*; import symbols.*;
3) public class Rel extends Logical {
4)     public Rel(Token tok, Expr x1, Expr x2) { super(tok, x1, x2); }
5)     public Type check(Type p1, Type p2) {
6)         if ( p1 instanceof Array || p2 instanceof Array ) return null;
7)         else if( p1 == p2 ) return Type.Bool;
8)         else return null;
9)     }
10)    public void jumping(int t, int f) {
11)        Expr a = expr1.reduce();
12)        Expr b = expr2.reduce();
13)        String test = a.toString() + " " + op.toString() + " " + b.toString();
14)        emitjumps(test, t, f);
15)    }
16) }
```

O método `jumping` (linhas 10-15, arquivo *Rel.java*) começa gerando código para as subexpressões `x` e `y` (linhas 11-22). Depois, ele chama o método `emitjumps` definido nas linhas 10-18, arquivo *Expr.java*, na Seção A.5. Se nem `t` nem `f` for o rótulo especial `0`, então `emitjumps` executa o seguinte

```
12)          emit("if " + test + " goto L" + t);        // Arquivo Expr.java
13)          emit("goto L" + f);
```

No máximo uma instrução é gerada se `t` ou `f` é o rótulo especial `0` (novamente, do arquivo *Expr..java*):

```
15)      else if( t != 0 ) emit("if " + test + " goto L" + t);
16)      else if( f != 0 ) emit("iffalse " + test + " goto L" + f);
17)      else ; // nada, porque ambos t e f fall through
```

Para ver outro uso de `emitjumps`, considere o código para a classe `Access`. A linguagem fonte permite que valores booleanos sejam atribuídos a identificadores e a elementos de arranjo, de modo que uma expressão booliana possa ser um acesso a arranjo. A classe `Access` possui o método `gen` para gerar código 'normal' e o método `jumping` para o código de desvio. O método `jumping` (linha 11) chama `emitjumps` depois de reduzir esse acesso de arranjo a um temporário. O construtor (linhas 6-9) é chamado com um arranjo achatado `a`, um índice `i`, e o tipo `p` de um elemento no arranjo achatado. A verificação de tipo é feita durante o cálculo do endereço de arranjo.

```
1) package inter;          // Arquivo Access..java
2) import lexer.*; import symbols.*;
3) public class Access extends Op {
4)     public Id array;
5)     public Expr index;
6)     public Access(Id a, Expr i, Type p) {     // p é o tipo de elemento após
7)         super(new Word("[]", Tag.INDEX), p);  // achatar o arranjo
8)         array = a; index = i;
9)     }
10)    public Expr gen() { return new Access(array, index.reduce(), type); }
11)    public void jumping(int t,int f) { emitjumps(reduce().toString(),t,f); }
12)    public String toString() {
13)        return array.toString() + " [ " + index.toString() + " ]";
14)    }
15) }
```

O código de desvio também pode ser usado para retornar um valor booliano. A classe `Logical`, anteriormente nesta seção, tem um método `gen` (linhas 15-24) que retorna um `temp` temporário, cujo valor é determinado pelo fluxo de controle pelo código de desvio para essa expressão. Na saída verdadeira dessa expressão booliana, `temp` é atribuído o valor `true`; na saída falsa, `temp` recebe o valor `false`. O temporário é declarado na linha 17. O código de desvio para essa expressão é gerado na linha 18 com a saída verdadeira sendo a próxima instrução e a saída falsa sendo um novo rótulo `f`. A próxima instrução atribui `true` a `temp` (linha 19), seguido por um desvio para um novo rótulo a (linha 20). O código na linha 21 emite o rótulo `f` e uma instrução que atribui `false` a `temp`. O fragmento de código termina com o rótulo `a`, gerado na linha 22. Finalmente, gen retorna `temp` (linha 23).

A.7 CÓDIGO INTERMEDIÁRIO PARA COMANDOS

Cada construção de comando é implementada por uma subclasse de `Stmt`. Os campos para os componentes de uma construção estão na subclasse relevante; por exemplo, a classe `While` possui campos para uma expressão de teste e um subcomando, conforme veremos.

As linhas 3-4 no código a seguir para a classe `Stmt` tratam da construção da árvore de sintaxe. O construtor `Stmt()` não faz nada, porque o trabalho é feito nas subclasses. O objeto estático `Stmt.null` (linha 4) representa uma seqüência vazia de comandos.

```
1) package inter;          // Arquivo Stmt.java
2) public class Stmt extends Node {
3)     public Stmt() { }
4)     public static Stmt Null = new Stmt();
5)     public void gen(int b, int a) {} // chamado com rótulos begin e after
6)     int after = 0;                   // guarda rótulo after
7)     public static Stmt Enclosing = Stmt.null;  // usado para comandos break
8) }
```

As linhas 5-7 tratam a geração do código de três endereços. O método `gen` é chamado com dois rótulos `b` e `a`, onde `b` marca o início do código para esse comando e `a` marca a primeira instrução após o código para esse comando. O método gen (linha 5) é um marcador de lugar para os métodos `gen` nas subclasses. As subclasses `While` e `Do` guardam seu rótulo `a` no campo `after` (linha 6), de modo que possa ser usado por qualquer comando break interno para desviar para fora de sua construção envolvente. O objeto `Stmt.Enclosing` é usado durante a análise sintática para acompanhar a construção envolvente. (Para uma linguagem fonte com comandos `continue`, podemos usar a mesma abordagem para acompanhar a construção envolvente de um comando continue.)

O construtor para a classe `If` constrói um nó para um comando **if** (*E*) *S*. Os campos `expr` e `stmt` contêm os nós para *E* e *S*, respectivamente. Observe que `expr` em letras minúsculas nomeia um campo de classe `Expr`; de modo semelhante, `stmt` nomeia um campo de classe `Stmt`.

```
1) package inter;          // Arquivo If.java
2) import symbols.*;
3) public class If extends Stmt {
4)     Expr expr; Stmt stmt;
5)     public If(Expr x, Stmt s) {
6)         expr = x;  stmt = s;
7)         if( expr.type != Type.Bool ) expr.error("boolean required in if");
8)     }
9)     public void gen(int b, int a) {
10)        int label = newlabel(); // rótulo do código para stmt
11)        expr.jumping(0, a);      // segue se for true, vai para a se for false
12)        emitlabel(label); stmt.gen(label, a);
13)    }
14) }
```

O código para um objeto `If` consiste em código de desvio para `expr` seguido pelo código para `stmt`. Conforme discutimos na Seção A.6, a chamada `expr.jumping(0,f)` na linha 11 especifica que o controle deve seguir o código de `expr` se `expr` for avaliado como true, e deve fluir para o rótulo `a` em caso contrário.

A implementação da classe `Else`, que trata de condicionais com partes `else`, é semelhante a da classe If:

```
1) package inter;          // Arquivo Else.java
2) import symbols.*;
3) public class Else extends Stmt {
4)     Expr expr; Stmt stmt1, stmt2;
5)     public Else(Expr x, Stmt s1, Stmt s2) {
6)         expr = x; stmt1 = s1; stmt2 = s2;
7)         if( expr.type != Type.Bool ) expr.error("boolean required in if");
8)     }
9)     public void gen(int b, int a) {
10)        int label1 = newlabel();    // label1 para stmt1
11)        int label2 = newlabel();    // label2 para stmt2
12)        expr.jumping(0,label2);     // segue para stmt1 se expr for true
13)        emitlabel(label1); stmt1.gen(label1, a); emit("goto L" + a);
14)        emitlabel(label2); stmt2.gen(label2, a);
15)    }
16) }
```

A construção de um objeto `While` é dividida entre o construtor `While()`, que cria um nó com filhos nulos (linha 5) e uma função de inicialização `init(x,s)`, que define o filho `expr` como `x` e o filho `stmt` como `s` (linhas 6-9). A função `gen(b,a)` para gerar o código de três endereços (linhas 10-16) está no mesmo espírito da função correspondente gen() na classe `If`. A diferença é que o rótulo `a` é guardado no campo `after` (linha 11) e que o código para `stmt` é seguido por um desvio para `b` (linha 15) para a próxima iteração do *loop* `while`.

```
1) package inter;          // Arquivo While.java
2) import symbols.*;
3) public class While extends Stmt {
4)     Expr expr; Stmt stmt;
5)     public While() { expr = null; stmt = null; }
6)     public void init(Expr x, Stmt s) {
7)         expr = x;  stmt = s;
8)         if( expr.type != Type.Bool ) expr.error("boolean required in while");
9)     }
10)    public void gen(int b, int a) {
11)        after = a;                  // guarda rótulo a
12)        expr.jumping(0, a);
13)        int label = newlabel();    // rótulo para comando
14)        emitlabel(label); stmt.gen(label, b);
15)        emit("goto L" + b);
16)    }
17) }
```

A classe `Do` é muito semelhante à classe `While`.

```
1) package inter;          // Arquivo Do.java
2) import symbols.*;
3) public class Do extends Stmt {
4)     Expr expr; Stmt stmt;
5)     public Do() { expr = null; stmt = null; }
6)     public void init(Stmt s, Expr x) {
7)         expr = x; stmt = s;
8)         if( expr.type != Type.Bool ) expr.error("boolean required in do");
9)     }
10)    public void gen(int b, int a) {
11)        after = a;
12)        int label = newlabel();    // rótulo para expr
13)        stmt.gen(b,label);
14)        emitlabel(label);
15)        expr.jumping(b,0);
16)    }
17) }
```

A classe `Set` implementa atribuições com um identificador no lado esquerdo e uma expressão à direita. A maior parte do código na classe `Set` é para construir um nó e verificar tipos (linhas 5-13). A função `gen` emite uma instrução de três endereços (linhas 14-16).

```
1) package inter;          // Arquivo Set.java
2) import lexer.*; import symbols.*;
3) public class Set extends Stmt {
4)     public Id id; public Expr expr;
5)     public Set(Id i, Expr x) {
6)         id = i; expr = x;
7)         if ( check(id.type, expr.type) == null ) error(“type error”);
8)     }
9)     public Type check(Type p1, Type p2) {
10)        if ( Type.numeric(p1) && Type.numeric(p2) ) return p2;
11)        else if ( p1 == Type.Bool && p2 == Type.Bool ) return p2;
12)        else return null;
13)    }
14)    public void gen(int b, int a) {
15)        emit( id.toString() + " = " + expr.gen().toString() );
16)    }
17) }
```

A classe `SetElem` implementa atribuições a um elemento do arranjo:

```
1) package inter;          // Arquivo SetElem.java
2) import lexer.*; import symbols.*;
3) public class SetElem extends Stmt {
4)     public Id array; public Expr index; public Expr expr;
5)     public SetElem(Access x, Expr y) {
6)         array = x.array; index = x.index; expr = y;
7)         if ( check(x.type, expr.type) == null ) error("type error");
8)     }
9)     public Type check(Type p1, Type p2) {
10)        if ( p1 instanceof Array || p2 instanceof Array ) return null;
11)        else if ( p1 == p2 ) return p2;
12)        else if ( Type.numeric(p1) && Type.numeric(p2) ) return p2;
13)        else return null;
14)    }
15)    public void gen(int b, int a) {
16)        String s1 = index.reduce().toString();
17)        String s2 = expr.reduce().toString();
18)        emit(array.toString() + " [ " + s1 + " ] = " + s2);
19)    }
20) }
```

A classe `Seq` implementa uma seqüência de comandos. Os testes para comandos nulos nas linhas 6-7 são para evitar rótulos. Observe que nenhum código é gerado para o comando nulo, `Stmt.null`, pois o método `gen` na classe `Stmt` não faz nada.

```
1) package inter;          // Arquivo Seq.java
2) public class Seq extends Stmt {
3)     Stmt stmt1; Stmt stmt2;
4)     public Seq(Stmt s1, Stmt s2) { stmt1 = s1; stmt2 = s2; }
5)     public void gen(int b, int a) {
6)         if ( stmt1 == Stmt.null ) stmt2.gen(b, a);
7)         else if ( stmt2 == Stmt.null ) stmt1.gen(b, a);
8)         else {
9)             int label = newlabel();
10)            stmt1.gen(b,label);
11)            emitlabel(label);
12)            stmt2.gen(label,a);
13)        }
14)    }
15) }
```

Um comando `break` passa o controle para fora de um comando `loop` ou `switch` envolvente. A classe `Break` usa o campo `stmt` para guardar a construção do comando envolvente (o analisador sintático garante que `Stmt.Enclosing` denota o nó da árvore de sintaxe para a construção envolvente). O código para um objeto `Break` é um desvio para o rótulo `stmt.after`, que marca a instrução imediatamente após o código para stmt.

```
1) package inter;          // Arquivo Break.java
2) public class Break extends Stmt {
3)     Stmt stmt;
4)     public Break() {
5)         if( Stmt.Enclosing == null ) error("unenclosed break");
6)         stmt = Stmt.Enclosing;
7)     }
8)     public void gen(int b, int a) {
9)         emit( "goto L" + stmt.after);
10)    }
11) }
```

A.8 ANALISADOR SINTÁTICO

O analisador sintático lê um fluxo de tokens e constrói uma árvore de sintaxe chamando as funções construtoras apropriadas das seções A.5-A.7. A tabela de símbolos corrente é mantida como no esquema de tradução da Figura 2.38, da Seção 2.7.

O pacote `parser` contém uma classe, `Parser`:

```
1) package parser;          // Arquivo Parser.java
2) import java.io.*; import lexer.*; import symbols.*; import inter.*;
3) public class Parser {
4)     private Lexer lex;    // analisador léxico para este analisador sintático
5)     private Token look;   // lookahead tagen
6)     Env top = null;       // tabela de símbolos corrente ou do topo
7)     int used = 0;         // memória usada para declarações
8)     public Parser(Lexer l) throws IOException { lex = l; move(); }
9)     void move() throws IOException { look = lex.scan(); }
10)    void error(String s) { throw new Error("near line "+lex.line+": "+s); }
11)     void match(int t) throws IOException {
12)        if( look.tag == t ) move();
13)        else error("syntax error");
14)    }
```

Assim como o tradutor de expressão simples da Seção 2.5, a classe `Parser` possui um procedimento para cada não-terminal. Os procedimentos são baseados em uma gramática formada pela remoção da recursão à esquerda da gramática da linguagem fonte da Seção A.1.

A análise começa com uma chamada ao procedimento `program`, que chama `block()` (linha 16) para analisar sintaticamente o fluxo de entrada e construir a árvore de sintaxe. As linhas 17-18 geram código intermediário.

```
15)   public void program() throws IOException {  // program -> block
16)      Stmt s = block();
17)      int begin = s.newlabel();  int after = s.newlabel();
18)      s.emitlabel(begin);  s.gen(begin, after);  s.emitlabel(after);
19)   }
```

O tratamento da tabela de símbolos é mostrado explicitamente no procedimento `block`.[3] A variável `top` (declarada na linha 5) contém a tabela de símbolos do topo; a variável `savedEnv` (linha 21) é um elo para a tabela de símbolos anterior.

```
20)   Stmt block() throws IOException {  // block -> { decls stmts }
21)      match('{');  Env savedEnv = top;  top = new Env(top);
22)      decls(); Stmt s = stmts();
23)      match('}');  top = savedEnv;
24)      return s;
25)   }
```

As declarações resultam em entradas da tabela de símbolos para identificadores (ver linha 36). Embora não aparecendo aqui, as declarações também podem resultar em instruções para reservar áreas de memória para os identificadores em tempo de execução.

```
26)   void decls() throws IOException {
27)      while( look.tag == Tag.BASIC ) {   // D -> type ID ;
28)         Type p = type(); Token tok = look; match(Tag.ID); match(';');
29)         Id id = new Id((Word)tok, p, used);
30)         top.put( tok, id );
31)         used = used + p.width;
32)      }
33)   }
34)   Type type() throws IOException {
35)      Type p = (Type)look;             // espera look.tag == Tag.BASIC
36)      match(Tag.BASIC);
37)      if( look.tag !='[ ' ) return p;  // T -> basic
38)      else return dims(p);             // retorna tipo do arranjo
39)   }
40)   Type dims(Type p) throws IOException {
41)      match('[');  Token tok = look;  match(Tag.NUM);  match(']');
42)      if( look.tag == '[' )
43)      p = dims(p);
44)      return new Array(((Num)tok).value, p);
45)   }
```

O procedimento `stmt` possui um comando switch com *cases* correspondendo às produções para o não-terminal *Stmt*. Cada *case* constrói um nó para uma construção, usando as funções construtoras discutidas na Seção A.7. Os nós para os comandos while e do são construídos quando o analisador sintático vê a palavra-chave de abertura. Os nós são construídos antes que o comando seja analisado sintaticamente, para permitir que qualquer comando break interno aponte de volta para o seu *loop* envolvente. Os *loops* aninhados são tratados usando a variável `Stmt.Enclosing` da classe `Stmt` e `savedStmt` (declarado na linha 52) para manter o *loop* envolvente corrente.

```
46)   Stmt stmts() throws IOException {
47)      if ( look.tag =='}' ) return Stmt.Null;
48)      else return new Seq(stmt(), stmts());
49)   }
50)   Stmt stmt() throws IOException {
51)      Expr x;  Stmt s, s1, s2;
52)      Stmt savedStmt;          // guarda o loop
53)      switch( look.tag ) {
54)      case ';':
55)         move();
56)         return Stmt.Null;
57)      case Tag.IF:
58)         match(Tag.IF); match('('); x = bool(); match(')');
59)         s1 = stmt();
```

```
60)           if( look.tag != Tag.ELSE ) return new If(x, s1);
61)           match(Tag.ELSE);
62)           s2 = stmt();
63)           return new Else(x, s1, s2);
64)        case Tag.WHILE:
65)           While whilenode = new While();
66)           savedStmt = Stmt.Enclosing; Stmt.Enclosing = whilenode;
67)           match(Tag.WHILE); match('('); x = bool(); match(')');
68)           s1 = stmt();
69)           whilenode.init(x, s1);
70)           Stmt.Enclosing = savedStmt;  // reinicia Stmt.Enclosing
71)           return whilenode;
72)        case Tag.DO:
73)           Do donode = new Do();
74)           savedStmt = Stmt.Enclosing; Stmt.Enclosing = donode;
75)           match(Tag.DO);
76)           s1 = stmt();
77)           match(Tag.WHILE); match('('); x = bool(); match(')'); match(';');
78)           donode.init(s1, x);
79)           Stmt.Enclosing = savedStmt;  // reinicia Stmt.Enclosing
80)           return donode;
81)        case Tag.BREAK:
82)           match(Tag.BREAK); match(';');
83)           return new Break();
84)        case '{':
85)           return block();
86)        default:
87)           return assign();
88)        }
89)     }
```

Por conveniência, o código para atribuições aparece em um procedimento auxiliar, `assign`.

```
90)     Stmt assign() throws IOException {
91)        Stmt stmt;  Token t = look;
92)        match(Tag.ID);
93)        Id id = top.get(t);
94)        if( id == null ) error(t.toString() + " undeclared");
95)        if( look.tag =='=' ) {        // S -> id = E ;
96)           move();  stmt = new Set(id, bool());
97)        }
98)        else {                        // S -> L = E ;
99)           Access x = offset(id);
100)          match('=');  stmt = new SetElem(x, bool());
101)       }
102)       match(';');
103)       return stmt;
104)    }
```

A análise sintática das expressões aritméticas e boolianas é semelhante. Em cada caso, um nó apropriado da árvore de sintaxe é criado. A geração de código para os dois é diferente, conforme discutimos nas seções A.5-A.6.

```
105)    Expr bool() throws IOException {
106)       Expr x = join();
107)       while( look.tag == Tag.OR ) {
108)          Token tok = look;  move();  x = new Or(tok, x, join());
109)       }
110)       return x;
```

[3] Uma alternativa atraente é acrescentar métodos `push` e `pop` à classe `Env`, com a tabela corrente acessível por meio de uma variável estática `Env.top`.

```
111)    }
112)    Expr join() throws IOException {
113)       Expr x = equality();
114)       while( look.tag == Tag.AND ) {
115)          Token tok = look;  move();  x = new And(tok, x, equality());
116)       }
117)       return x;
118)    }
119)    Expr equality() throws IOException {
120)       Expr x = rel();
121)       while( look.tag == Tag.EQ || look.tag == Tag.NE ) {
122)          Token tok = look;  move();  x = new Rel(tok, x, rel());
123)       }
124)       return x;
125)    }
126)    Expr rel() throws IOException {
127)       Expr x = expr();
128)       switch( look.tag ) {
129)       case'<': case Tag.LE: case Tag.GE: case'>':
130)          Token tok = look;  move();  return new Rel(tok, x, expr());
131)       default:
132)          return x;
133)       }
134)    }
135)    Expr expr() throws IOException {
136)       Expr x = term();
137)       while( look.tag == '+' || look.tag == '-' ) {
138)          Token tok = look;  move();  x = new Arith(tok, x, term());
139)       }
140)       return x;
141)    }
142)    Expr term() throws IOException {
143)       Expr x = unary();
144)       while(look.tag == '*' || look.tag == '/' ) {
145)          Token tok = look;  move();    x = new Arith(tok, x, unary());
146)       }
147)       return x;
148)    }
149)    Expr unary() throws IOException {
150)       if( look.tag == '-' ) {
151)          move();  return new Unary(Word.minus, unary());
152)       }
153)       else if( look.tag == '!' ) {
154)          Token tok = look;  move();  return new Not(tok, unary());
155)       }
156)       else return factor();
157)    }
```

O restante do código no analisador sintático trata 'fatores' nas expressões. O procedimento auxiliar `offset` gera código para cálculos de endereço de arranjo, conforme discutimos na Seção 6.4.3.

```
158)    Expr factor() throws IOException {
159)       Expr x = null;
160)       switch( look.tag ) {
161)       case'(':
162)          move(); x = bool(); match(')');
163)          return x;
164)       case Tag.NUM:
165)          x = new Constant(look, Type.Int);     move(); return x;
166)       case Tag.REAL:
167)          x = new Constant(look, Type.Float);   move(); return x;
168)       case Tag.TRUE:
169)          x = Constant..True;                    move(); return x;
```

```
170)        case Tag.FALSE:
171)           x = Constant.False;                    move(); return x;
172)        default:
173)           error("syntax error");
174)           return x;
175)        case Tag.ID:
176)           String s = look.toString();
177)           Id id = top.get(look);
178)           if( id == null ) error(look.toString() + " undeclared");
179)           move();
180)           if( look.tag != '[' ) return id;
181)           else return offset(id);
182)        }
183)     }
184)     Access offset(Id a) throws IOException {    // I -> [E] | [E] I
185)        Expr i; Expr w; Expr t1, t2; Expr loc;  // herda id
186)        Type type = a.type;
187)        match('['); i = bool(); match(']');      // primeiro índice, I -> [ E ]
188)        type = ((Array)type).of;
189)        w = new Constant(type.width);
190)        t1 = new Arith(new Token('*'), i, w);
191)        loc = t1;
192)        while( look.tag == '[' ) {       // I multidimensional -> [ E ] I
193)           match('['); i = bool(); match(']');
194)           type = ((Array)type).of;
195)           w = new Constant(type.width);
196)           t1 = new Arith(new Token('*'), i, w);
197)           t2 = new Arith(new Token('+'), loc, t1);
198)           loc = t2;
199)        }
200)        return new Access(a, loc, type);
201)     }
202) }
```

A.9 CRIANDO O *FRONT-END*

O código para os pacotes aparece em cinco diretórios: `main`, `lexer`, `symbol`, `parser` e `inter`. Os comandos para criar o compilador variam de um sistema para outro. Os comandos seguintes são de uma implementação do UNIX:

```
javac lexer/*.java
javac symbols/*.java
javac inter/*.java
javac parser/*.java
javac main/*.java
```

O comando `javac` cria arquivos `.class` para cada classe. O tradutor pode então ser exercitado digitando-se `java main.Main` seguido pelo programa fonte a ser traduzido; por exemplo, o conteúdo do arquivo `test`.

```
1)  {                  // Arquivo test
2)     int i; int j; float v; float x; float[100] a;
3)     while( true ) {
4)        do i = i+1; while( a[i] < v);
5)        do j = j-1; while( a[j] > v);
6)        if( i >= j ) break;
7)        x = a[i]; a[i] = a[j]; a[j] = x;
8)     }
9)  }
```

Nessa entrada, o *front-end* produz

```
 1) L1:L3:   i = i + 1
 2) L5:      t1 = i * 8
 3)          t2 = a [ t1 ]
 4)          if t2 < v goto L3
 5) L4:      j = j - 1
 6) L7:      t3 = j * 8
 7)          t4 = a [ t3 ]
 8)          if t4 > v goto L4
 9) L6:      iffalse i >= j goto L8
10) L9:     goto L2
11) L8:     t5 = i * 8
12)         x = a [ t5 ]
13) L10:    t6 = i * 8
14)         t7 = j * 8
15)         t8 = a [ t7 ]
16)         a [ t6 ] = t8
17) L11:    t9 = j * 8
18)         a [ t9 ] = x
19)         goto L1
20) L2:
```

Experimente.