附录 A　一个完整的编译器前端

这个附录给出了一个完整的编译器前端，它是基于 2.5 节至 2.8 节中非正式描述的简单编译器编写的。和第 2 章的主要不同之处在于，这个前端像 6.6 节中描述的那样为布尔表达式生成跳转代码。我们首先给出源语言的语法。描述这个语法所用的文法需要进行调整，以适应自顶向下的语法分析技术。

这个翻译器的 Java 代码由五个包组成：`main`、`lexer`、`symbol`、`parser` 和 `inter`。包 `inter` 中包含的类处理用抽象语法表示的语言结构。因为语法分析器的代码和其他各个包交互，所以它将在最后描述。每个包存放在一个独立的目录中，每个类都有一个单独的文件。

作为语法分析器的输入时，源程序就是一个由词法单元组成的流，因此面向对象特性和语法分析器的代码之间没有什么关系。当由语法分析器输出时，源程序就是一棵抽象语法树，树中的结构或结点被实现为对象。这些对象负责处理下列工作：构造一个抽象语法树结点、类型检查、生成三地址中间代码（见包 `inter`）。

A.1　源语言

这个语言的一个程序由一个块组成，该块中包含可选的声明和语句。语法符号 `basic` 表示基本类型。

$$
\begin{array}{rcl}
\mathit{program} & \rightarrow & \mathit{block} \\
\mathit{block} & \rightarrow & \{\ \mathit{decls}\ \mathit{stmts}\ \} \\
\mathit{decls} & \rightarrow & \mathit{decls}\ \mathit{decl} \mid \epsilon \\
\mathit{decl} & \rightarrow & \mathit{type}\ \mathbf{id}\ ; \\
\mathit{type} & \rightarrow & \mathit{type}\ [\ \mathbf{num}\ ] \mid \mathbf{basic} \\
\mathit{stmts} & \rightarrow & \mathit{stmts}\ \mathit{stmt} \mid \epsilon
\end{array}
$$

把赋值当作一个语句（而不是表达式中的运算符）可以简化翻译工作。

面向对象与面向步骤

在一个面向对象方法中，一个构造的所有代码都集中在这个与构造对应的类中。但是在面向步骤的方法中，这个方法中的代码是按照步骤进行组织的，因此一个类型检查过程中对每个构造都有一个 case 分支，且一个代码生成过程对每个构造也都有一个 case 分支，等等。

对这两者进行衡量，可知使用面向对象方法会使得改变或增加一个构造（比如 for 语句）变得较容易；而使用面向步骤的方法会使得改变或增加一个步骤（比如类型检查）变得比较容易。使用对象来实现时，增加一个新的构造可以通过写一个自包含的类来实现；但是如果要改变一个步骤，比如插入自动类型转换的代码，就需要改变所有受影响的类。使用面向步骤的方式时，增加一个新构造可能会引起各个步骤中的多个过程的改变。

$$
\begin{array}{rcl}
\mathit{stmt} & \rightarrow & \mathit{loc} = \mathit{bool}\ ; \\
 & \mid & \mathbf{if}\ (\ \mathit{bool}\ )\ \mathit{stmt} \\
 & \mid & \mathbf{if}\ (\ \mathit{bool}\ )\ \mathit{stmt}\ \mathbf{else}\ \mathit{stmt} \\
 & \mid & \mathbf{while}\ (\ \mathit{bool}\ )\ \mathit{stmt} \\
 & \mid & \mathbf{do}\ \mathit{stmt}\ \mathbf{while}\ (\ \mathit{bool}\ )\ ; \\
 & \mid & \mathbf{break}\ ; \\
 & \mid & \mathit{block} \\
\mathit{loc} & \rightarrow & \mathit{loc}\ [\ \mathit{bool}\ ] \mid \mathbf{id}
\end{array}
$$

表达式的产生式处理了运算符的结合性和优先级。它们对每个优先级级别都使用了一个非终结符号，而非终结符号 *factor* 用来表示括号中的表达式、标识符、数组引用和常量。

```
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

程序的执行从类 Main 的方法 main 开始。方法 main 创建了一个词法分析器和一个语法分析器，然后调用语法分析器中的方法 program。

```
1) package main;                    // 文件 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 词法分析器

包 lexer 是 2.6.5 节中的词法分析器的代码的扩展。类 Tag 定义了各个词法单元对应的常量：

```
1) package lexer;                   // 文件 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) }
```

其中的三个常量 INDEX、MINUS 和 TEMP 不是词法单元，它们将在抽象语法树中使用。

类 Token 和 Num 和 2.6.5 节的相同，但是增加了方法 toString：

```
1) package lexer;                   // 文件 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;                   //文件 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) }
```

类 Word 用于管理保留字、标识符和像 && 这样的复合词法单元的词素。它也可以用来管理在中间代码中运算符的书写形式；比如单目减号。例如，源文本中的 -2 的中间形式是 minus 2。

```
1) package lexer;                    // 文件 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) }
```

类 Real 用于处理浮点数：

```
1) package lexer;                    // 文件 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) }
```

如我们在 2.6.5 节中讨论的，类 Lexer 的主方法，即函数 scan，识别数字、标识符和保留字。

类 Lexer 中的第 9 ~ 13 行保留了选定的关键字。第 14 ~ 16 行保留了在其他地方定义的对象的词素。对象 Word.True 和 Word.False 在类 Word 中定义。对应于基本类型 int、char、bool 和 float 的对象在类 Type 中定义。类 Type 是 Word 的一个子类。类 Type 来自包 symbols。

```
1) package lexer;                    // 文件 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)   }
```

函数 readch( )(第 18 行)用于把下一个输入字符读到变量 peek 中。名字 readch 被复用或重载，(第 19 ~ 24 行)，以便帮助识别复合的词法单元。比如，一看到输入字符 <，调用 readch(" = ")就会把下一个字符读入peek，并检查它是否为 =。

```
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)   }
```

函数 scan 一开始首先略过所有的空白字符(第 26 ~ 30 行)。它首先试图识别像 < = 这样的复合词法单元(第 31 ~ 34 行)和像 365 及 3.14 这样的数字(第 45 ~ 58 行)。如果不成功，它就试图读入一个字符串(第 59 ~ 70 行)。

```
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)      }
```

最后，peek 中的任意字符都被作为词法单元返回(第 71 ~ 72 行)。

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

A.4 符号表和类型

包 symbols 实现了符号表和类型。

类 Env 实质上和图 2-37 中的代码一样。类 Lexer 把字符串映射为字，类 Env 把字符串词法单元映射为类 Id 的对象。类 Id 和其他的对应于表达式和语句的类一起都在包 inter 中定义。

```
1) package symbols;                    // 文件 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) }
```

我们把类 Type 定义为类 Word 的子类，因为像 int 这样的基本类型名字就是保留字，将被词法分析器从词素映射为适当的对象。对应于基本类型的对象是 Type.Int、Type.Float、Type.Char 和 Type.Bool(第 7 ~ 10 行)。这些对象从超类中继承了字段 tag，相应的值被设置为 Tag.BASIC，因此语法分析器以同样的方式处理它们。

```
1) package symbols;                    // 文件 Type.java
2) import lexer.*;
3) public class Type extends Word {
4)    public int width = 0;            //width用于存储分配
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 );
```

函数 numeric(第 11 ~ 14 行)和 max(第 15 ~ 20 行)可用于类型转换。

```
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) }
```

在两个“数字”类型之间允许进行类型转换，“数字”类型包括 Type.Char、Type.Int 和 Type.Float。当一个算术运算符应用于两个数字类型时，结果类型是这两个类型的“max”值。

数组是这个源语言中唯一的构造类型。在第 7 行中调用 super 设置字段 width 的值。这个值在计算地址时是必不可少的。它同时也把 lexeme 和 tok 设置为默认值，这些值没有被使用。

```
1) package symbols;                    // 文件 Array.java
2) import lexer.*;
3) public class Array extends Type {
4)    public Type of;                  // 数组的元素类型
5)    public int size = 1;             // 元素个数
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 表达式的中间代码

包 inter 包含了 Node 的类层次结构。Node 有两个子类:对应于表达式结点的 Expr 和对应于语句结点的 Stmt。本节介绍 Expr 和它的子类。Expr 的某些方法处理布尔表达式和跳转代码,这些方法和 Expr 的其他子类将在 A.6 节中讨论。

抽象语法树中的结点被实现为类 Node 的对象。为了报告错误,字段 lexline(文件 Node.java 的第 4 行)保存了本结点对应的构造在源程序中的行号。第 7 ~ 10 行用来生成三地址代码。

```
1) package inter;                    // 文件 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) }
```

表达式构造被实现为 Expr 的子类。类 Expr 包含字段 op 和 type(文件 Expr.java 的第 4 ~ 5 行),分别表示了一个结点上的运算符和类型。

```
1) package inter;                    // 文件 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; }
```

方法 gen(第 7 行)返回了一个“项”,该项可以成为一个三地址指令的右部。给定一个表达式 $E = E_1 + E_2$,方法 gen 返回一个项 $x_1 + x_2$,其中 x_1 和 x_2 分别是存放 E_1 和 E_2 值的地址。如果这个对象是一个地址,就可以返回 this 值。Expr 的子类通常会重新实现 gen。

方法 reduce(第 8 行)把一个表达式计算(或者说“归约”)成为一个单一的地址。也就是说,它返回一个常量、一个标识符,或者一个临时名字。给定一个表达式 E,方法 reduce 返回一个存放 E 的值的临时变量 t。如果这个对象是一个地址,那么 this 仍然是正确的返回值。

我们把对方法 jumping 和 emitjumps(第 9 ~ 18 行)的讨论推迟到 A.6 节中进行,它们为布尔表达式生成跳转代码。

```
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 ; // 不生成指令,因为 t 和 f 都直接穿越
18)   }
19)   public String toString() { return op.toString(); }
20) }
```

因为一个标识符就是一个地址,类 Id 从类 Expr 中继承了 gen 和 reduce 的默认实现。

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

对应于一个标识符的类 Id 的结点是一个叶子结点。函数调用 super(id,p)(文件 *Id.java* 的第 5 行)把 id 和 p 分别保存在继承得到的字段 op 和 type 中。字段 offset(第 4 行)保存了这个标识符的相对地址。

类 Op 提供了 reduce 的一个实现(文件 Op.java 的第 5 ~ 10 行)。这个类的子类包括:表示算术运算符的子类 Arith,表示单目运算符的子类 Unary 和表示数组访问的子类 Access。这些子类都继承了这个实现。在每种情况下,reduce 调用 gen 来生成一个项,生成一个指令把这个项赋值给一个新的临时名字,并返回这个临时名字。

```
1) package inter;                    // 文件 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) }
```

类 Arith 实现了双目运算符,比如 + 和 * 。构造函数 Arith 首先调用 super(tok,null)(第 6 行),其中 tok 是一个表示该运算符的词法单元,null 是类型的占位符。相应的类型在第 7 行使用函数 Type.max 来确定,这个函数检查两个运算分量是否可以被类型强制为一个常见的数字类型;Type.max 的代码在 A.4 节中给出。如果它们能够进行自动类型转换,type 就被设置为结果类型;否则就报告一个类型错误(第 8 行)。这个简单编译器检查类型,但是它并不插入类型转换代码。

```
1) package inter;                    // 文件 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) }
```

方法 gen 把表达式的子表达式归约为地址,并将表达式的运算符作用于这些地址(文件 *Arith.java* 的第 11 行),从而构造出了一个三地址指令的右部。比如,假设 gen 在 a + b * c 的根部被调用。其中对 reduce 的调用返回 a 作为子表达式 a 的地址,并返回 t 作为 b * c 的地址。同时,reduce 还生成指令 t = b * c。方法 gen 返回了一个新的 Arith 结点,其中的运算符是 * ,而运算分量是地址 a 和 t。[⊖]

⊖ 为了报告错误,在构造一个结点时,类 Node 中的字段 lexline 记录了当前的文本行号。我们把在中间代码生成过程中构造新的结点时跟踪行号的任务留给读者。

值得注意的是，和所有其他表达式一样，临时名字也有类型。因此，构造函数 Temp 被调用时有一个类型参数(文件 *Temp.java* 的第 6 行)。[⊖]

```
1) package inter;                    // 文件 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) }
```

类 Unary 和类 Arith 对应，但是处理的是单目运算符：

```
1) package inter;                    // 文件 Unary.java
2) import lexer.*; import symbols.*;
3) public class Unary extends Op {
4)    public Expr expr;
5)    public Unary(Token tok, Expr x) {    // 处理单目减法，对！的处理见 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 布尔表达式的跳转代码

布尔表达式 *B* 的跳转代码由方法 jumping 生成。这个方法的参数是两个标号 t 和 f，它们分别称为表达式 *B* 的 *true* 出口和 *false* 出口。如果 *B* 的值为真，代码中就包含一个目标为 t 的跳转指令；如果 *B* 的值为假，就有一个目标为 f 的指令。按照惯例，特殊标号 0 表示控制流从 *B* 穿越，到达 *B* 的代码之后的下一个指令。

我们从类 Constant 开始。第 4 行上的构造函数 Constant 的参数是一个词法单元 tok 和一个类型 p。它在抽象语法树中构造出一个标号为 tok、类型为 p 的叶子结点。为方便起见，构造函数 Constant 被重载(第 5 行)，重载后的构造函数可以根据一个整数创建一个常量对象。

```
1) package inter;                    // 文件 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);
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) }
```

方法 jumping(文件 *Constant.java* 的第 9 ~ 12 行)有两个参数：标号为 t 和 f。如果这个常量是静态对象 True(在第 7 行中定义)，t 不是特殊标号 0，那么就会生成一个目标为 t 的跳转指令。否则，如果这是对象 False(在第 8 行中定义)且 f 非零，那么就会生成一个目标为 f 的跳转指令。

⊖ 另一种可行的方法是让这个构造函数以一个表达式结点作为参数，这样它就可以复制这个表达式结点的类型和文本位置。

类 Logical 为类 Or、And 和 Not 提供了一些常见功能。字段 expr1 和 expr2(第 4 行)对应于一个逻辑运算符的运算分量(虽然类 Not 实现了一个单目运算符，为方便起见，我们还是把它当作 Logical 的子类)。构造函数 Logical(tok,a,b)(第 5 ~ 10 行)构造出了一个语法树的结点，其运算符为 tok，而运算分量为 a 和 b。在完成这些工作时，它调用函数 check 来保证 a 和 b 都是布尔类型。方法 gen 将会在本节的最后讨论。

```
 1) package inter;                    // 文件 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);                        // 开始时类型设置为空
 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) }
```

在类 Or 中，方法 jumping(第 5 ~ 10 行)生成了一个布尔表达式 $B = B_1 \parallel B_2$ 的跳转代码。当前假设 B 的 true 出口 t 和 false 出口 f 都不是特殊标号 0。因为如果 B_1 为真，B 必然为真，所以 B_1 的 true 出口必然是 t，而它的 false 出口对应于 B_2 的第一条指令。B_2 的 true 和 false 出口和 B 的相应出口相同。

```
 1) package inter;                    // 文件 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) }
```

在一般情况下，B 的 true 出口 t 可能是特殊标号 0。变量 label(文件 Or.java 的第 6 行)保证了 B_1 的 true 出口被正确地设置为 B 的代码的结尾处。如果 t 为 0，那么 label 被设置为一个新的标号，并在 B_1 和 B_2 的代码被生成后再生成这个新标号。

类 And 的代码和 Or 的代码类似。

```
 1) package inter;                    // 文件 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) }
```

虽然类 Not 实现的是一个单目运算符，这个类和其他布尔运算符之间仍然具有相当多的共同之处，因此我们把它作为 Logical 的一个子类。它的超类具有两个运算分量，因此在第 4 行对 super 的调用中 x2 出现了两次。在第 5 ~ 6 行的方法中，只有 expr2(文件 Logical.java 的第 4 行上声明)被用到。在第 5 行，方法 jumping 仅仅把 true 出口和 false 出口对调，调用 expr2.jumping。

```
1) package inter;                    // 文件 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) }
```

类 Rel 实现了运算符 <、<=、==、!=、>= 和 >。函数 check(第 5 ~ 9 行)检查两个运算分量是否具有相同的类型，但它们不是数组类型。为简单起见，这里不允许类型强制转换。

```
1) package inter;                    // 文件 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) }
```

方法 jumping(文件 *Rel.java* 的第 10 ~ 15 行)首先为子表达式 expr1 和 expr2 生成代码(第 11 ~ 12 行)。然后它调用方法 emitjumps，这个方法在 A.5 节的文件 *Expr.java* 中的第 10 ~ 18 行中定义。如果 t 和 f 都不是特殊标号 0，那么 emitjumps 执行下列代码：

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

如果 t 或 f 是特殊标号 0，那么最多只会生成一个指令(同样是来自文件 *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 ; // 不生成指令，因为t和f都直接穿越
```

在生成类 Access 的代码时演示了方法 emitjumps 的另一种用法。源语言允许把布尔值赋给标识符和数组元素，因此一个布尔表达式可能是一个数组访问。类 Access 有一个方法 gen，用来生成“正常”代码，另一个方法 jumping 用来生成跳转代码。方法 jumping(第 11 行)在把这个数组访问归约为一个临时变量后调用 emitjumps。这个类的构造函数(第 6 ~ 9 行)被调用

时的参数为一个平坦化的数组 a、一个下标 i 和该数组的元素类型 p。在生成数组地址计算代码的过程中完成了类型检查。

```
1) package inter;                    // 文件 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是将数组平坦化后的元素类型
7)       super(new Word("[]", Tag.INDEX), p);
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) }
```

跳转代码还可以被用来返回一个布尔值。本节中较早描述的类 Logical 有一个方法 gen(第 15 ~ 24 行)。这个方法返回一个临时变量 temp。这个变量的值由这个表达式的跳转代码中的控制流决定。在这个布尔表达式的 true 出口，temp 被赋予 true 值；在 false 出口，temp 被赋予 false 值。这个临时变量在第 17 行声明。这个表达式的跳转代码在第 18 行生成，其中的 true 出口是下一条指令，而 false 出口是一个新标号 f。下一条指令把 true 值赋给 temp(第 19 行)，后面紧跟目标为新标号 a 的跳转指令(第 20 行)。第 21 行上的代码生成标号 f 和一个把 false 赋给 temp 的指令。这个代码片段的结尾是标号 a，该标号在第 22 行生成。最后，gen 返回 temp(第 23 行)。

A.7 语句的中间代码

每个语句构造被实现为 Stmt 的一个子类。一个构造的组成部分对应的字段是相应子类的对象。例如，如我们将看到的，类 While 有一个对应于测试表达式的字段和一个子语句字段。

下面的类 Stmt 的代码中的第 3 ~ 4 行处理抽象语法树的构造。构造函数 Stmt( )不做任何事情，因为相关处理工作是在子类中完成的。静态对象 Stmt.Null(第 4 行)表示一个空的语句序列。

```
1) package inter;                    // 文件 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) {} // 调用时的参数是语句开始处的标号和语句的下一条指令的标号
6)    int after = 0;                   // 保存语句的下一条指令的标号
7)    public static Stmt Enclosing = Stmt.Null;  // 用于break 语句
8) }
```

第 5 ~ 7 行处理三地址代码的生成。方法 gen 被调用时两个参数分别是标号 a 和 b，其中 b 标记这个语句的代码的开始处，而 a 标记这个语句的代码之后的第一条指令。方法 gen(第 5 行)是子类中的 gen 方法的占位符。子类 While 和 Do 把它们的标号 a 存放在字段 after(第 6 行)中。当任何内层的 break 语句要跳出这个外层构造时就可以使用这些标号。对象 Stmt.Enclosing 在语法分析时被用于跟踪外层构造。(对于包含 continue 语句的源语言，我们可以使用同样的方法来跟踪一个 continue 语句的外层构造。)

类 If 的构造函数为语句 **if** (E) S 构造一个结点。字段 expr 和 stmt 分别保存了 E 和 S 对应的结点。请注意，小写字母组成的 expr 是一个类 Expr 的字段的名字。类似地，stmt 是类为

Stmt 的字段的名字。

```
1) package inter;                    // 文件 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(); // stmt的代码的标号
11)      expr.jumping(0, a);     //为真时控制流穿越，为假时转向a
12)      emitlabel(label); stmt.gen(label, a);
13)   }
14) }
```

一个 If 对象的代码包含了 expr 的跳转代码，然后是 stmt 的代码。如 A.6 节中所讨论的，第 11 行的调用 expr.jumping(0,a)指明如果 expr 的值为真，控制流必须穿越 expr 的代码；否则控制流必须转向标号 a。

类 Else 处理条件语句的 else 部分。它的实现和类 If 的实现类似：

```
1) package inter;                    // 文件 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用于语句stmt1
11)      int label2 = newlabel();    // label2用于语句stmt2
12)      expr.jumping(0,label2);     // 为真时控制流穿越到stmt1
13)      emitlabel(label1); stmt1.gen(label1, a); emit("goto L" + a);
14)      emitlabel(label2); stmt2.gen(label2, a);
15)   }
16) }
```

一个 While 对象的构造过程分为两个部分：构造函数 While( )创建了一个子结点为空的结点(第 5 行)；初始化函数 int(x,s)把子结点 expr 设置成为 x，把子结点 stmt 设置成为 s(第 6 ~ 9 行)。函数 gen(b,a)用于生成三地址代码(第 10 ~ 16 行)。它和类 If 中的相应函数gen( )在本质上有着相通之处。不同之处在于标号 a 被保存在字段 after 中(第 11 行)，且 stmt 的代码之后紧跟着一个目标为 b 的跳转指令(第 15 行)。这个指令使得 while 循环进入下一次迭代。

```
1) package inter;                    // 文件 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;                  // 保存标号a
12)      expr.jumping(0, a);
13)      int label = newlabel();     // 用于stmt的标号
14)      emitlabel(label); stmt.gen(label, b);
15)      emit("goto L" + b);
16)   }
17) }
```

类 Do 和类 While 非常相似。

```
1) package inter;                    // 文件 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();    // 用于 expr 的标号
13)      stmt.gen(b,label);
14)      emitlabel(label);
15)      expr.jumping(b,0);
16)   }
17) }
```

类 Set 实现了左部为标识符且右部为一个表达式的赋值语句。在类 Set 中的大部分代码的目的是构造一个结点并进行类型检查(第 5 ~ 13 行)。函数 gen 生成一个三地址指令(第 14 ~ 16 行)。

```
1) package inter;                    // 文件 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) }
```

类 SetElem 实现了对数组元素的赋值。

```
1) package inter;                    // 文件 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) }
```

类 Seq 实现了一个语句序列。在第 6 ~ 7 行上对空语句的测试是为了避免使用标号。请注意，空语句 Stmt.Null 不会产生任何代码，因为类 Stmt 中的方法 gen 不做任何处理。

```
1) package inter;                    // 文件 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) }
```

一个 break 语句把控制流转出它的外围循环或外围 switch 语句。类 Break 使用字段 stmt 来保存它的外围语句构造(语法分析器保证 Stmt.Enclosing 表示了其外围构造对应的语法树结点)。一个 Break 对象的代码是一个目标为标号 stmt.after 的跳转指令。这个标号标记了紧跟在 stmt 的代码之后的指令。

```
1) package inter;                    // 文件 Break.java
2) public class Break extends Stmt {
3)    Stmt stmt;
4)    public Break() {
5)       if( Stmt.Enclosing == Stmt. 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 语法分析器

语法分析器读入一个由词法单元组成的流，并调用适当的在 A.5 ~ A.7 节中讨论的构造函数，构建出一棵抽象语法树。当前符号表按照 2.7 节中图 2-38 的翻译方案进行处理。

包 parser 包含一个类 Parser：

```
1) package parser;                   // 文件 Parser.java
2) import java.io.*; import lexer.*; import symbols.*; import inter.*;
3) public class Parser {
4)    private Lexer lex;     // 这个语法分析器的词法分析器
5)    private Token look;    // 向前看词法单元
6)    Env top = null;        // 当前或顶层的符号表
7)    int used = 0;          // 用于变量声明的存储位置
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)   }
```

和 2.5 节中的简单表达式的翻译器类似，类 Parser 对每个非终结符号有一个过程。消除 A.1 节中源语言文法中的左递归后可以得到一个新的文法。这些过程就是基于这个新文法创建的。

语法分析过程首先调用了过程 program，这个过程又调用了 block( )（第 16 行）来对输入流进行语法分析，并构建出抽象语法树。第 17 ~ 18 行生成了中间代码。

```
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)   }
```

对符号表的处理明确显示在过程 block 中[⊖]。变量 top（在第 5 行中声明）存放了最顶层的符号表，变量 savedEnv（第 21 行）是一个指向前面的符号表的连接。

```
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)   }
```

程序中的声明会被处理为符号表中有关标识符的条目（见第 30 行）。虽然这里没有显示，声明还可能生成在运行时刻为标识符保留存储空间的指令。

```
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;              // 期望 look.tag == Tag.BASIC
36)      match(Tag.BASIC);
37)      if( look.tag != '[' ) return p; // T -> basic
38)      else return dims(p);              // 返回数组类型
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)   }
```

过程 stmt 有一个 switch 语句。这个语句的各个 case 分支对应于非终结符号 Stmt 的各个产生式。每个 case 分支都使用 A.7 节中讨论的构造函数来建立某个构造对应的结点。当语法分析器碰到 while 语句和 do 语句的开始关键字的时候，就会创建这些语句的结点。这些结点在相应语句进行完语法分析之前就构造出来，这可以使得任何内层的 break 语句回指到它的外层循环语句。当出现嵌套的循环时，我们通过使用类 Stmt 中的变量 Stmt.Enclosing 和 savedStmt（在第 52 行声明）来保存当前的外层循环的。

```
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;          // 用于为break语句保存外层的循环语句
```

⊖ 另一种很具有吸引力的方法是向类 Env 中添加方法 push 和 pop，而当前的符号表可以通过一个静态变量 Env.top 来访问。

```
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;  // 重置 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;  // 重置 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)    }
```

为方便起见，赋值语句的代码出现在一个辅助过程 assign 中。

```
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)   }
```

对算术运算和布尔表达式的语法分析很相似。在每种情况下都会创建一个正确的抽象语法树结点。如 A.5 节和 A.6 节所讨论的，这两者的代码生成方法有所不同。

```
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;
```

```
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)    }
```

在语法分析器中的其余代码处理表达式“因子”。辅助过程 `offset` 按照 6.4.3 节中讨论的方法为数组地址计算生成代码。

```
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;  // 继承 id
186)       Type type = a.type;
187)       match('['); i = bool(); match(']');     // 第一个下标，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 == '[' ) {      // 多维下标，I->[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 创建前端

这个编译器的各个包的代码存放在五个目录中：main、lexer、symbols、parser 和 inter。创建编译器的命令行根据系统的不同而不同。下面是编译器的 UNIX 实现：

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

上面的 javac 命令为每个类创建了 .class 文件。要练习使用我们的翻译器，只需要输入 java main.Main，后面跟上将要被翻译的源程序，比如文件 test 中的内容：

```
1) {                  // 文件 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) }
```

对于这个输入，这个前端输出：

```
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:
```

尝试一下。

附录 B　寻找线性独立解

算法 B.1　找出 $A\vec{x} \geqslant \vec{0}$ 的最大的线性独立解集合，并将它们表示为矩阵 B 的各行。

输入： 一个 $M \times N$ 的矩阵 A。

输出： 由 $A\vec{x} \geqslant \vec{0}$ 的各个线性独立解组成的矩阵 B。

方法： 算法以伪代码的方式在下面给出。请注意，$X[y]$ 表示矩阵 X 的第 y 行，$X[y:z]$ 表示矩阵 X 的第 $y \sim z$ 行，而 $X[y:z][u:v]$ 表示矩阵 X 中的第 $y \sim z$ 行及第 $u \sim v$ 列的方块。 □

```
M = A^T;
r_0 = 1;
c_0 = 1;
B = I_{n×n}; /* 一个 n×n 的单元矩阵 */

while ( true ) {

    /* 1. 使 M[r_0 : r' - 1][c_0 : c' - 1] 为一个对角线元素为正的对角矩
          阵，并且满足 M[r' : n][c_0 : m] = 0。M[r' : n] 为解。*/
    r' = r_0;
    c' = c'_0;
    while ( 存在 M[r][c] ≠ 0 ) 使得
            r - r' 和 c - c' 都 ≥ 0 ) {
        通过行列互换，把中心点 M[r][c] 移动到 M[r'][c']
        把 B 中的第 r 行和第 r' 行互换;
        if ( M[r'][c'] < 0 ) {
            M[r'] = -1 * M[r'];
            B[r'] = -1 * B[r'];
        }
        for ( row = r_0 to n ) {
            if ( row ≠ r' and M[row][c'] ≠ 0 ){
                u = -(M[row][c']/M[r'][c']);
                M[row] = M[row] + u * M[r'];
                B[row] = B[row] + u * B[r'];
            }
        }
        r' = r' + 1;
        c' = c' + 1;
    }

    /* 2. 找出 M[r' : n] 之外的一个解，这个解一定是
          M[r_0 : r' - 1][c_0 : m] 的一个非负组合*/
    找出 k_{r_0}, ..., k_{r'-1} ≥ 0 使得
            k_{r_0} M[r_0][c' : m] + ··· + k_{r'-1} M[r' - 1][c' : m] ≥ 0;
    if ( 如果存在一个非平凡解，比如 k_r > 0 ) {
        M[r] = k_{r_0} M[r_0] + ··· + k_{r'-1} M[r' - 1];
        NoMoreSoln = false;
    } else /* M[r' : n] 就是全部解 */
        NoMoreSoln = true;

    /* 3. 使得 M[r_0 : r_n - 1][c_0 : m] ≥ 0 */
    if ( NoMoreSoln ) { /* 把解 M[r' : n] 移动到 M[r_0 : r_n - 1] */
        for ( r = r' to n )
            交换 M 和 B 中的行 r 和 r_0 + r - r';
        r_n = r_0 + n - r' + 1;}
    else { /* 使用行相加的方法来找出更多的解 */
```

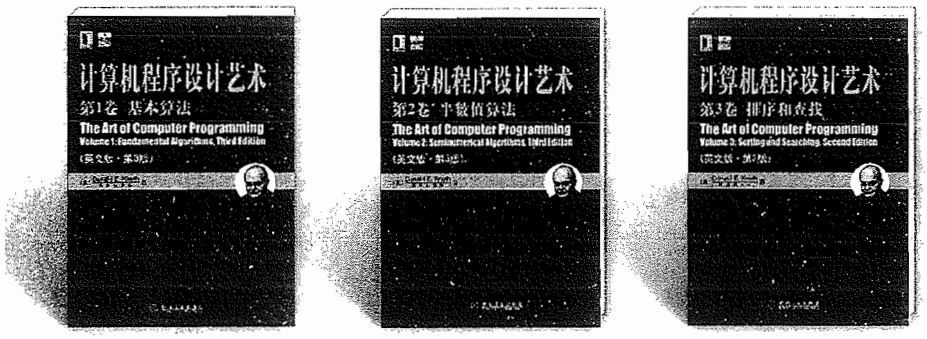
```
        r_n = n + 1;
        for ( col = c' to m )
            if ( 存在 M[row][col] < 0 使得 row ⩾ r_0 )
                if ( 存在 M[r][col] > 0 使得 r ⩾ r_0 )
                    { for ( row = r_0 to r_n − 1 )
                        if ( M[row][col] < 0 ) {
                            u = ⌈(−M[row][col]/M[r][col])⌉;
                            M[row] = M[row] + u * M[r];
                            B[row] = B[row] + u * B[r];
                        }
                    }
                else
                    for ( row = r_n − 1 to r_0 step -1 )
                        if ( M[row][col] < 0 ){
                            r_n = r_n − 1;
                            把M[row]和 M[r_n] 对换;
                            把B[row]和B[r_n] 对换;
                        }
    }

    /* 4. 使得 M[r_0 : r_n − 1][1 : c_0 − 1] ⩾ 0 */
    for ( row = r_0 to r_n − 1 )
        for ( col = 1 to c_0 − 1 )
            if ( M[row][col] < 0 ){
                选取一个 r 使得 M[r][col] > 0 且 r < r_0;
                u = ⌈(−M[row][col]/M[r][col])⌉;
                M[row] = M[row] + u * M[r];
                B[row] = B[row] + u * B[r];
            }
    /* 5. 如果有必要，对M[r_n : n]中的各行重复处理 */
    if (NoMoreSoln or r_n > n or r_n == r_0) {
        从 B 中删除从 r_n 到 n各行 ;
        return B;
    }
    else {
        c_n = m + 1;
        for ( col = m to 1 step -1 )
            if ( 不存在 M[r][col] > 0 使得 r < r_n ) {
                c_n = c_n − 1;
                交换 M 中的第 col 列和第 c_n 列;
            }
        r_0 = r_n;
        c_0 = c_n;
    }
}
```

计算机科学巨匠
Donald E.Knuth(高德纳)
经典巨著

《计算机程序设计艺术 第1卷 基本算法(英文影印版.第3版)》
ISBN：978-7-111-22709-0
定价：95.00

《计算机程序设计艺术 第2卷 半数值算法(英文影印版.第3版)》
ISBN：978-7-111-22718-2
定价：109.00

《计算机程序设计艺术 第3卷 排序和查找(英文影印版.第2版)》
ISBN：978-7-111-22717-5
定价：109.00



“它本来是作为参考书撰写的，但有人发现每一卷都可以饶有兴致地从头读到尾。一位中国的程序员甚至把他的阅读经历比做读诗。如果你认为你确实是一个好的程序员，读一读Knuth的《计算机程序设计艺术》吧，要是你真把它读通了，你就可以给我递简历了。”

—— Bill Gates

《计算机程序设计艺术：第1卷 第1册（双语版）MMIX：新千年的RISC计算机》
ISBN：7-111-18031-3
定价：45.00

《计算机程序设计艺术：第4卷 第2册 生成所有元组和排列（双语版）》
ISBN：7-111-17773-8
定价：45.00

《计算机程序设计艺术 第4卷 第3册 生成所有组合和分划（双语版）》
ISBN：7-111-17774-6
定价：45.00

《计算机程序设计艺术(第4卷 第4册.双语版)》
ISBN：978-7-111-20825-9
定价：42.00