Appendix A

A Complete Front End

The complete compiler front end in this appendix is based on the informally described simple compiler of Sections 2.5-2.8. The main difference from Chapter 2 is that the front end generates jumping code for boolean expressions, as in Section 6.6. We begin with the syntax of the source language, described by a grammar that needs to be adapted for top-down parsing.

The Java code for the translator consists of five packages: `main`, `lexer`, `symbols`, `parser`, and `inter`. Package `inter` contains classes for the language constructs in the abstract syntax. Since the code for the parser interacts with the rest of the packages, it will be discussed last. Each package is stored as a separate directory with a file per class.

Going into the parser, the source program consists of a stream of tokens, so object-orientation has little to do with the code for the parser. Coming out of the parser, the source program consists of a syntax tree, with constructs or nodes implemented as objects. These objects deal with all of the following: construct a syntax-tree node, check types, and generate three-address intermediate code (see package `inter`).

A.1 The Source Language

A program in the language consists of a block with optional declarations and statements. Token **basic** represents basic types.

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

Treating assignments as statements, rather than as operators within expressions, simplifies translation.

Object-Oriented Versus Phase-Oriented

With an object-oriented approach, all the code for a construct is collected in the class for the construct. Alternatively, with a phase-oriented approach, the code is grouped by phase, so a type checking procedure would have a case for each construct, and a code generation procedure would have a case for each construct, and so on.

The tradeoff is that an object-oriented approach makes it easier to change or add a construct, such as "for" statements, and a phase-oriented approach makes it easier to change or add a phase, such as type checking. With objects, a new construct can be added by writing a self-contained class, but a change to a phase, such as inserting code for coercions, requires changes across all the affected classes. With phases, a new construct can result in changes across the procedures for the phases.

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

The productions for expressions handle associativity and precedence of operators. They use a nonterminal for each level of precedence and a nonterminal, *factor*, for parenthesized expressions, identifiers, array references, and constants.

$$\begin{array}{rcl}
\mathit{bool} & \rightarrow & \mathit{bool}\ \texttt{||}\ \mathit{join}\ \mid\ \mathit{join} \\
\mathit{join} & \rightarrow & \mathit{join}\ \texttt{\&\&}\ \mathit{equality}\ \mid\ \mathit{equality} \\
\mathit{equality} & \rightarrow & \mathit{equality}\ \texttt{==}\ \mathit{rel}\ \mid\ \mathit{equality}\ \texttt{!=}\ \mathit{rel}\ \mid\ \mathit{rel} \\
\mathit{rel} & \rightarrow & \mathit{expr}\ \texttt{<}\ \mathit{expr}\ \mid\ \mathit{expr}\ \texttt{<=}\ \mathit{expr}\ \mid\ \mathit{expr}\ \texttt{>=}\ \mathit{expr}\ \mid \\
 & & \quad \mathit{expr}\ \texttt{>}\ \mathit{expr}\ \mid\ \mathit{expr} \\
\mathit{expr} & \rightarrow & \mathit{expr}\ \texttt{+}\ \mathit{term}\ \mid\ \mathit{expr}\ \texttt{-}\ \mathit{term}\ \mid\ \mathit{term} \\
\mathit{term} & \rightarrow & \mathit{term}\ \texttt{*}\ \mathit{unary}\ \mid\ \mathit{term}\ \texttt{/}\ \mathit{unary}\ \mid\ \mathit{unary} \\
\mathit{unary} & \rightarrow & \texttt{!}\ \mathit{unary}\ \mid\ \texttt{-}\ \mathit{unary}\ \mid\ \mathit{factor} \\
\mathit{factor} & \rightarrow & \texttt{(}\ \mathit{bool}\ \texttt{)}\ \mid\ \mathit{loc}\ \mid\ \textbf{num}\ \mid\ \textbf{real}\ \mid\ \textbf{true}\ \mid\ \textbf{false}
\end{array}$$

A.2 Main

Execution begins in method `main` in class `Main`. Method `main` creates a lexical analyzer and a parser and then calls method `program` in the parser:

```
package main;                  // File Main.java
import java.io.*; import lexer.*; import parser.*;
```

```
public class Main {
   public static void main(String[] args) throws IOException {
      Lexer lex = new Lexer();
      Parser parse = new Parser(lex);
      parse.program();
      System.out.write('\n');
   }
}
```

A.3 Lexical Analyzer

Package `lexer` is an extension of the code for the lexical analyzer in Section 2.6.5. Class `Tag` defines constants for tokens:

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

Three of the constants, `INDEX`, `MINUS`, and `TEMP`, are not lexical tokens; they will be used in syntax trees.

Classes `Token` and `Num` are as in Section 2.6.5, with method `toString` added:

```
package lexer;                // File Token.java
public class Token {
   public final int tag;
   public Token(int t) { tag = t; }
   public String toString() {return "" + (char)tag;}
}
```

```
package lexer;                // File Num.java
public class Num extends Token {
   public final int value;
   public Num(int v) { super(Tag.NUM); value = v; }
   public String toString() { return "" + value; }
}
```

Class `Word` manages lexemes for reserved words, identifiers, and composite tokens like `&&`. It is also useful for managing the written form of operators in the intermediate code like unary minus; for example, the source text `-2` has the intermediate form `minus 2`.

```
package lexer;                // File Word.java
public class Word extends Token {
   public String lexeme = "";
   public Word(String s, int tag) { super(tag); lexeme = s; }
   public String toString() { return lexeme; }
   public static final Word
      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) }
```

Class Real is for floating point numbers:

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

The main method in class Lexer, function scan, recognizes numbers, identifiers, and reserved words, as discussed in Section 2.6.5.

Lines 9-13 in class Lexer reserve selected keywords. Lines 14-16 reserve lexemes for objects defined elsewhere. Objects Word.True and Word.False are defined in class Word. Objects for the basic types int, char, bool, and float are defined in class Type, a subclass of Word. Class Type is from package symbols.

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

Function readch() (line 18) is used to read the next input character into variable peek. The name readch is reused or overloaded (lines 19-24) to help recognize composite tokens. For example, once input < is seen, the call readch('=') reads the next character into peek and checks whether it is =.

```
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)    }
```

Function `scan` begins by skipping white space (lines 26-30). It recognizes composite tokens like `<=` (lines 31-44) and numbers like `365` and `3.14` (lines 45-58), before collecting words (lines 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)       }
```

Finally, any remaining characters are returned as tokens (lines 71-72).

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

A.4 Symbol Tables and Types

Package `symbols` implements symbol tables and types.

Class `Env` is essentially unchanged from Fig. 2.37. Whereas class `Lexer` maps strings to words, class `Env` maps word tokens to objects of class `Id`, which is defined in package `inter` along with the classes for expressions and statements.

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

We define class `Type` to be a subclass of `Word` since basic type names like `int` are simply reserved words, to be mapped from lexemes to appropriate objects by the lexical analyzer. The objects for the basic types are `Type.Int`, `Type.Float`, `Type.Char`, and `Type.Bool` (lines 7-10). All of them have inherited field `tag` set to `Tag.BASIC`, so the parser treats them all alike.

```
 1) package symbols;              // File Type.java
 2) import lexer.*;
 3) public class Type extends Word {
 4)    public int width = 0;          // width is used for storage allocation
 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 );
```

Functions `numeric` (lines 11-14) and `max` (lines 15-20) are useful for type conversions.

```
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) }
```

Conversions are allowed between the “numeric” types `Type.Char`, `Type.Int`, and `Type.Float`. When an arithmetic operator is applied to two numeric types, the result has the “max” of the two types.

Arrays are the only constructed type in the source language. The call to `super` on line 7 sets field `width`, which is essential for address calculations. It also sets `lexeme` and `tok` to default values that are not used.

```
 1) package symbols;                // File Array.java
 2) import lexer.*;
 3) public class Array extends Type {
 4)    public Type of;                 // array *of* type
 5)    public int size = 1;            // number of elements
 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 Intermediate Code for Expressions

Package `inter` contains the `Node` class hierarchy. `Node` has two subclasses: `Expr` for expression nodes and `Stmt` for statement nodes. This section introduces `Expr` and its subclasses. Some of the methods in `Expr` deal with booleans and jumping code; they will be discussed in Section A.6, along with the remaining subclasses of `Expr`.

Nodes in the syntax tree are implemented as objects of class `Node`. For error reporting, field `lexline` (line 4, file *Node.java*) saves the source-line number of the construct at this node. Lines 7-10 are used to emit three-address code.

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

Expression constructs are implemented by subclasses of `Expr`. Class `Expr` has fields `op` and `type` (lines 4-5, file *Expr.java*), representing the operator and type, respectively, at a node.

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

Method `gen` (line 7) returns a "term" that can fit the right side of a three-address instruction. Given expression $E = E_1 + E_2$, method `gen` returns a term x_1+x_2, where x_1 and x_2 are addresses for the values of E_1 and E_2, respectively. The return value `this` is appropriate if this object is an address; subclasses of `Expr` typically reimplement `gen`.

Method `reduce` (line 8) computes or "reduces" an expression down to a single address; that is, it returns a constant, an identifier, or a temporary name. Given expression E, method `reduce` returns a temporary t holding the value of E. Again, `this` is an appropriate return value if this object is an address.

We defer discussion of methods `jumping` and `emitjumps` (lines 9-18) until Section A.6; they generate jumping code for boolean expressions.

```
 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 ; // nothing since both t and f fall through
18)    }
19)    public String toString() { return op.toString(); }
20) }
```

Class `Id` inherits the default implementations of `gen` and `reduce` in class `Expr`, since an identifier is an address.

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

The node for an identifier of class `Id` is a leaf. The call `super(id,p)` (line 5, file *Id.java*) saves `id` and `p` in inherited fields `op` and `type`, respectively. Field `offset` (line 4) holds the relative address of this identifier.

Class `Op` provides an implementation of `reduce` (lines 5-10, file *Op.java*) that is inherited by subclasses `Arith` for arithmetic operators, `Unary` for unary operators, and `Access` for array accesses. In each case, `reduce` calls `gen` to generate a term, emits an instruction to assign the term to a new temporary name, and returns the temporary.

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

Class `Arith` implements binary operators like `+` and `*`. Constructor `Arith` begins by calling `super(tok,null)` (line 6), where `tok` is a token representing the operator and `null` is a placeholder for the type. The type is determined on line 7 by using `Type.max`, which checks whether the two operands can be coerced to a common numeric type; the code for `Type.max` is in Section A.4. If they can be coerced, `type` is set to the result type; otherwise, a type error is reported (line 8). This simple compiler checks types, but it does not insert type conversions.

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

Method `gen` constructs the right side of a three-address instruction by reducing the subexpressions to addresses and applying the operator to the addresses (line 11, file *Arith.java*). For example, suppose `gen` is called at the root for `a+b*c`. The calls to `reduce` return `a` as the address for subexpression `a` and a temporary `t` as the address for `b*c`. Meanwhile, `reduce` emits the instruction `t=b*c`. Method `gen` returns a new `Arith` node, with operator `*` and addresses `a` and `t` as operands.[1]

It is worth noting that temporary names are typed, along with all other expressions. The constructor `Temp` is therefore called with a type as a parameter (line 6, file *Temp.java*).[2]

```
 1) package inter;                    // File Temp.java
 2) import lexer.*; import symbols.*;
 3) public class Temp extends Expr {
```

[1]For error reporting, field `lexline` in class `Node` records the current lexical line number when a node is constructed. We leave it to the reader to track line numbers when new nodes are constructed during intermediate code generation.

[2]An alternative approach might be for the constructor to take an expression node as a parameter, so it can copy the type and lexical position of the expression node.

```
   static int count = 0;
   int number = 0;
   public Temp(Type p) { super(Word.temp, p); number = ++count; }
   public String toString() { return "t" + number; }
}
```

Class `Unary` is the one-operand counterpart of class `Arith`:

```
package inter;                 // File Unary.java
import lexer.*; import symbols.*;
public class Unary extends Op {
   public Expr expr;
   public Unary(Token tok, Expr x) {    // handles minus, for ! see Not
      super(tok, null);  expr = x;
      type = Type.max(Type.Int, expr.type);
      if (type == null ) error("type error");
   }
   public Expr gen() { return new Unary(op, expr.reduce()); }
   public String toString() { return op.toString()+" "+expr.toString(); }
}
```

A.6 Jumping Code for Boolean Expressions

Jumping code for a boolean expression B is generated by method `jumping`, which takes two labels `t` and `f` as parameters, called the true and false exits of B, respectively. The code contains a jump to `t` if B evaluates to true, and a jump to `f` if B evaluates to false. By convention, the special label `0` means that control falls through B to the next instruction after the code for B.

We begin with class `Constant`. The constructor `Constant` on line 4 takes a token `tok` and a type `p` as parameters. It constructs a leaf in the syntax tree with label `tok` and type `p`. For convenience, the constructor `Constant` is overloaded (line 5) to create a constant object from an integer.

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

Method `jumping` (lines 9-12, file *Constant.java*) takes two parameters, labels `t` and `f`. If this constant is the static object `True` (defined on line 7) and `t` is not the special label `0`, then a jump to `t` is generated. Otherwise, if this is the object `False` (defined on line 8) and `f` is nonzero, then a jump to `f` is generated.

Class Logical provides some common functionality for classes Or, And, and Not. expr1 and expr2 (line 4) correspond to the operands of a logical operator. (Although class Not implements a unary operator, for convenience, it is a subclass of Logical.) The constructor Logical(tok,a,b) (lines 5-10) builds a syntax node with operator tok and operands a and b. In doing so it uses function check to ensure that both a and b are booleans. Method gen will be discussed at the end of this section.

```
 1) package inter;                // File 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);                     // null type to start
 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) }
```

In class Or, method jumping (lines 5-10) generates jumping code for a boolean expression $B = B_1 \,||\, B_2$. For the moment, suppose that neither the true exit t nor the false exit f of B is the special label 0. Since B is true if B_1 is true, the true exit of B_1 must be t and the false exit corresponds to the first instruction of B_2. The true and false exits of B_2 are the same as those of B.

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

In the general case, `t`, the true exit of B, can be the special label `0`. Variable `label` (line 6, file *Or.java*) ensures that the true exit of B_1 is set properly to the end of the code for B. If `t` is `0`, then `label` is set to a new label that is emitted after code generation for both B_1 and B_2.

The code for class `And` is similar to the code for `Or`.

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

Class `Not` has enough in common with the other boolean operators that we make it a subclass of `Logical`, even though `Not` implements a unary operator. The superclass expects two operands, so `x2` appears twice in the call to `super` on line 4. Only `expr2` (declared on line 4, file *Logical.java*) is used in the methods on lines 5-6. On line 5, method `jumping` simply calls `expr2.jumping` with the true and false exits reversed.

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

Class `Rel` implements the operators `<`, `<=`, `==`, `!=`, `>=`, and `>`. Function `check` (lines 5-9) checks that the two operands have the same type and that they are not arrays. For simplicity, coercions are not permitted.

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

Method `jumping` (lines 10-15, file *Rel.java*) begins by generating code for the subexpressions `expr1` and `expr2` (lines 11-12). It then calls method `emitjumps` defined on lines 10-18, file *Expr.java*, in Section A.5. If neither `t` nor `f` is the special label `0`, then `emitjumps` executes the following

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

At most one instruction is generated if either `t` or `f` is the special label `0` (again, from file *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 ; // nothing since both t and f fall through
```

For another use of `emitjumps`, consider the code for class `Access`. The source language allows boolean values to be assigned to identifiers and array elements, so a boolean expression can be an array access. Class `Access` has method `gen` for generating "normal" code and method `jumping` for jumping code. Method `jumping` (line 11) calls `emitjumps` after reducing this array access to a temporary. The constructor (lines 6-9) is called with a flattened array `a`, an index `i`, and the type `p` of an element in the flattened array. Type checking is done during array address calculation.

```
 1) package inter;                 // File 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 is element type after
 7)       super(new Word("[]", Tag.INDEX), p);  // flattening the array
 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) }
```

Jumping code can also be used to return a boolean value. Class `Logical`, earlier in this section, has a method `gen` (lines 15-24) that returns a temporary `temp`, whose value is determined by the flow of control through the jumping code for this expression. At the true exit of this boolean expression, `temp` is assigned `true`; at the false exit, `temp` is assigned `false`. The temporary is declared on line 17. Jumping code for this expression is generated on line 18 with the true exit being the next instruction and the false exit being a new label `f`. The next instruction assigns `true` to `temp` (line 19), followed by a jump to a new label `a` (line 20). The code on line 21 emits label `f` and an instruction that assigns `false` to `temp`. The code fragment ends with label `a`, generated on line 22. Finally, `gen` returns `temp` (line 23).

A.7 Intermediate Code for Statements

Each statement construct is implemented by a subclass of `Stmt`. The fields for the components of a construct are in the relevant subclass; for example, class `While` has fields for a test expression and a substatement, as we shall see.

Lines 3-4 in the following code for class `Stmt` deal with syntax-tree construction. The constructor `Stmt()` does nothing, since the work is done in the subclasses. The static object `Stmt.Null` (line 4) represents an empty sequence of statements.

```
 1) package inter;                    // File 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) {} // called with labels begin and after
 6)    int after = 0;                   // saves label after
 7)    public static Stmt Enclosing = Stmt.Null;  // used for break stmts
 8) }
```

Lines 5-7 deal with the generation of three-address code. The method `gen` is called with two labels `b` and `a`, where `b` marks the beginning of the code for this statement and `a` marks the first instruction after the code for this statement. Method `gen` (line 5) is a placeholder for the `gen` methods in the subclasses. The subclasses `While` and `Do` save their label `a` in the field `after` (line 6) so it can be used by any enclosed break statement to jump out of its enclosing construct. The object `Stmt.Enclosing` is used during parsing to keep track of the enclosing construct. (For a source language with continue statements, we can use the same approach to keep track of the enclosing construct for a continue statement.)

The constructor for class `If` builds a node for a statement **if** (E) S. Fields `expr` and `stmt` hold the nodes for E and S, respectively. Note that `expr` in lower-case letters names a field of class `Expr`; similarly, `stmt` names a field of class `Stmt`.

```
 1) package inter;                    // File 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(); // label for the code for stmt
11)       expr.jumping(0, a);     // fall through on true, goto a on false
12)       emitlabel(label); stmt.gen(label, a);
13)    }
14) }
```

The code for an `If` object consists of jumping code for `expr` followed by the code for `stmt`. As discussed in Section A.6, the call `expr.jumping(0,a)` on line

11 specifies that control must fall through the code for `expr` if `expr` evaluates to true, and must flow to label `a` otherwise.

The implementation of class `Else`, which handles conditionals with else parts, is analogous to that of class `If`:

```
package inter;                // File Else.java
import symbols.*;
public class Else extends Stmt {
   Expr expr; Stmt stmt1, stmt2;
   public Else(Expr x, Stmt s1, Stmt s2) {
      expr = x; stmt1 = s1; stmt2 = s2;
      if( expr.type != Type.Bool ) expr.error("boolean required in if");
   }
   public void gen(int b, int a) {
      int label1 = newlabel();   // label1 for stmt1
      int label2 = newlabel();   // label2 for stmt2
      expr.jumping(0,label2);    // fall through to stmt1 on true
      emitlabel(label1); stmt1.gen(label1, a); emit("goto L" + a);
      emitlabel(label2); stmt2.gen(label2, a);
   }
}
```

The construction of a `While` object is split between the constructor `While()`, which creates a node with null children (line 5), and an initialization function `init(x,s)`, which sets child `expr` to `x` and child `stmt` to `s` (lines 6–9). Function `gen(b,a)` for generating three-address code (lines 10–16) is in the spirit of the corresponding function `gen()` in class `If`. The difference is that label `a` is saved in field `after` (line 11) and that the code for `stmt` is followed by a jump to `b` (line 15) for the next iteration of the while loop.

```
 1) package inter;                // File 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;               // save label a
12)       expr.jumping(0, a);
13)       int label = newlabel();   // label for stmt
14)       emitlabel(label); stmt.gen(label, b);
15)       emit("goto L" + b);
16)    }
17) }
```

Class `Do` is very similar to class `While`.

```
package inter;                // File Do.java
import symbols.*;
public class Do extends Stmt {
   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();   // label for expr
13)       stmt.gen(b,label);
14)       emitlabel(label);
15)       expr.jumping(b,0);
16)    }
17) }
```

Class Set implements assignments with an identifier on the left side and an expression on the right. Most of the code in class Set is for constructing a node and checking types (lines 5-13). Function gen emits a three-address instruction (lines 14-16).

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

Class SetElem implements assignments to an array element:

```
 1) package inter;                // File 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();
```

```
      emit(array.toString() + " [ " + s1 + " ] = " + s2);
   }
}
```

Class Seq implements a sequence of statements. The tests for null statements on lines 6-7 are for avoiding labels. Note that no code is generated for the null statement, Stmt.Null, since method gen in class Stmt does nothing.

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

A break statement sends control out of an enclosing loop or switch statement. Class Break uses field stmt to save the enclosing statement construct (the parser ensures that Stmt.Enclosing denotes the syntax-tree node for the enclosing construct). The code for a Break object is a jump to the label stmt.after, which marks the instruction immediately after the code for stmt.

```
package inter;                // File Break.java
public class Break extends Stmt {
   Stmt stmt;
   public Break() {
      if( Stmt.Enclosing == Stmt.Null ) error("unenclosed break");
      stmt = Stmt.Enclosing;
   }
   public void gen(int b, int a) {
      emit( "goto L" + stmt.after);
   }
}
```

A.8 Parser

The parser reads a stream of tokens and builds a syntax tree by calling the appropriate constructor functions from Sections A.5-A.7. The current symbol table is maintained as in the translation scheme in Fig. 2.38 in Section 2.7.

Package parser contains one class, Parser:

```
package parser;                // File Parser.java
import java.io.*; import lexer.*; import symbols.*; import inter.*;
```

```
 3) public class Parser {
 4)    private Lexer lex;    // lexical analyzer for this parser
 5)    private Token look;   // lookahead token
 6)    Env top = null;       // current or top symbol table
 7)    int used = 0;         // storage used for declarations
 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)    }
```

Like the simple expression translator in Section 2.5, class `Parser` has a procedure for each nonterminal. The procedures are based on a grammar formed by removing left recursion from the source-language grammar in Section A.1.

Parsing begins with a call to procedure `program`, which calls `block()` (line 16) to parse the input stream and build the syntax tree. Lines 17-18 generate intermediate code.

```
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)    }
```

Symbol-table handling is shown explicitly in procedure `block`.[3] Variable `top` (declared on line 5) holds the top symbol table; variable `savedEnv` (line 21) is a link to the previous symbol table.

```
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)    }
```

Declarations result in symbol-table entries for identifiers (see line 30). Although not shown here, declarations can also result in instructions to reserve storage for the identifiers at run time.

```
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;            // expect look.tag == Tag.BASIC
```

[3]An attractive alternative is to add methods `push` and `pop` to class `Env`, with the current table accessible through a static variable `Env.top`.

```
36)      match(Tag.BASIC);
37)      if( look.tag != '[' ) return p; // T -> basic
38)      else return dims(p);            // return array type
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)   }
```

Procedure `stmt` has a switch statement with cases corresponding to the productions for nonterminal *Stmt*. Each case builds a node for a construct, using the constructor functions discussed in Section A.7. The nodes for while and do statements are constructed when the parser sees the opening keyword. The nodes are constructed before the statement is parsed to allow any enclosed break statement to point back to its enclosing loop. Nested loops are handled by using variable `Stmt.Enclosing` in class `Stmt` and `savedStmt` (declared on line 52) to maintain the current enclosing loop.

```
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;         // save enclosing loop for breaks
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;  // reset 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;  // reset Stmt.Enclosing
80)         return donode;
```

```
      case Tag.BREAK:
         match(Tag.BREAK); match(';');
         return new Break();
      case '{':
         return block();
      default:
         return assign();
      }
   }
```

For convenience, the code for assignments appears in an auxiliary procedure, `assign`.

```
   Stmt assign() throws IOException {
      Stmt stmt;  Token t = look;
      match(Tag.ID);
      Id id = top.get(t);
      if( id == null ) error(t.toString() + " undeclared");
      if( look.tag == '=' ) {       // S -> id = E ;
         move();  stmt = new Set(id, bool());
      }
      else {                        // S -> L = E ;
         Access x = offset(id);
         match('=');  stmt = new SetElem(x, bool());
      }
      match(';');
      return stmt;
   }
```

The parsing of arithmetic and boolean expressions is similar. In each case, an appropriate syntax-tree node is created. Code generation for the two is different, as discussed in Sections A.5-A.6.

```
   Expr bool() throws IOException {
      Expr x = join();
      while( look.tag == Tag.OR ) {
         Token tok = look;  move();  x = new Or(tok, x, join());
      }
      return x;
   }
   Expr join() throws IOException {
      Expr x = equality();
      while( look.tag == Tag.AND ) {
         Token tok = look;  move();  x = new And(tok, x, equality());
      }
      return x;
   }
   Expr equality() throws IOException {
      Expr x = rel();
      while( look.tag == Tag.EQ || look.tag == Tag.NE ) {
         Token tok = look;  move();  x = new Rel(tok, x, rel());
      }
      return x;
   }
   Expr rel() throws IOException {
      Expr x = expr();
      switch( look.tag ) {
      case '<': case Tag.LE: case Tag.GE: case '>':
         Token tok = look;  move();  return new Rel(tok, x, expr());
      default:
         return x;
      }
   }
   Expr expr() throws IOException {
      Expr x = term();
      while( look.tag == '+' || look.tag == '-' ) {
         Token tok = look;  move();  x = new Arith(tok, x, term());
      }
      return x;
   }
   Expr term() throws IOException {
      Expr x = unary();
      while(look.tag == '*' || look.tag == '/' ) {
         Token tok = look;  move();   x = new Arith(tok, x, unary());
      }
      return x;
   }
   Expr unary() throws IOException {
      if( look.tag == '-' ) {
         move();  return new Unary(Word.minus, unary());
      }
      else if( look.tag == '!' ) {
         Token tok = look;  move();  return new Not(tok, unary());
      }
      else return factor();
   }
```

The rest of the code in the parser deals with "factors" in expressions. The auxiliary procedure `offset` generates code for array address calculations, as discussed in Section 6.4.3.

```
   Expr factor() throws IOException {
      Expr x = null;
      switch( look.tag ) {
      case '(':
         move(); x = bool(); match(')');
         return x;
      case Tag.NUM:
         x = new Constant(look, Type.Int);    move(); return x;
      case Tag.REAL:
         x = new Constant(look, Type.Float);  move(); return x;
      case Tag.TRUE:
         x = Constant.True;                   move(); return x;
      case Tag.FALSE:
         x = Constant.False;                  move(); return x;
      default:
         error("syntax error");
         return x;
      case Tag.ID:
         String s = look.toString();
         Id id = top.get(look);
         if( id == null ) error(look.toString() + " undeclared");
         move();
         if( look.tag != '[' ) return id;
         else return offset(id);
      }
   }
   Access offset(Id a) throws IOException {   // I -> [E] | [E] I
      Expr i; Expr w; Expr t1, t2; Expr loc;  // inherit id
      Type type = a.type;
      match('['); i = bool(); match(']');     // first index, I -> [ E ]
      type = ((Array)type).of;
      w = new Constant(type.width);
      t1 = new Arith(new Token('*'), i, w);
      loc = t1;
      while( look.tag == '[' ) {      // multi-dimensional I -> [ E ] I
         match('['); i = bool(); match(']');
         type = ((Array)type).of;
         w = new Constant(type.width);
         t1 = new Arith(new Token('*'), i, w);
         t2 = new Arith(new Token('+'), loc, t1);
         loc = t2;
      }
      return new Access(a, loc, type);
   }
}
```

A.9 Creating the Front End

The code for the packages appears in five directories: main, lexer, symbol, parser, and inter. The commands for creating the compiler vary from system to system. The following are from a UNIX implementation:

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

The javac command creates .class files for each class. The translator can then be exercised by typing java main.Main followed by the source program to be translated; e.g., the contents of file test

```
{                 // File test
   int i; int j; float v; float x; float[100] a;
   while( true ) {
      do i = i+1; while( a[i] < v);
      do j = j-1; while( a[j] > v);
      if( i >= j ) break;
      x = a[i]; a[i] = a[j]; a[j] = x;
   }
}
```

On this input, the front end produces

```
L1:L3:  i = i + 1
L5:     t1 = i * 8
        t2 = a [ t1 ]
        if t2 < v goto L3
L4:     j = j - 1
L7:     t3 = j * 8
        t4 = a [ t3 ]
        if t4 > v goto L4
L6:     iffalse i >= j goto L8
L9:    goto L2
L8:    t5 = i * 8
       x = a [ t5 ]
L10:   t6 = i * 8
       t7 = j * 8
       t8 = a [ t7 ]
       a [ t6 ] = t8
L11:   t9 = j * 8
       a [ t9 ] = x
       goto L1
L2:
```

Try it.