پیوست الف

بخش جلویی کامپایلر

بخش جلویی کامل کامپایلر که در این پیوست آمده است، مبتنی بر کامپایلر ساده ای است که به طور غیر رسمی در بخش های ۲-۵ تا ۲-۸ توصیف شد. تفاوت عمده آن با فصل ۲ این است که این بخش جلویی که پرش را برای عبارات بولی مانند آنچه که در بخش ۶-۶ آمده است ، تولید می کند. با نحو زبان مبداء شروع می کنیم که توسط گرامری توصیف شد که باید برای تجزیه بالا به پایین تعدیل شود..

کد جاوا برای مترجم شامل پنج پکیج است : **main** ، **lexer** ، **symbol** ، **parser** ، و **inter** . پکیج **inter** شامل کلاس هایی برای ساختارهای زبان با نحو انتزاعی است. چون کد مربوط به تجزیه کننده با بقیه پکیج ها تعامل دارد، در آخر بحث می شود. هر پکیج به صورت یک دایرکتوری و هر کلاس در یک فایل ذخیره شده است .

وقتی برنامه مبداء وارد تجزیه کننده می شود، شامل رشته ای از نشانه ها است ، لذا شیءگرایی ، کار زیادی با کد تجزیه کننده ندارد. با خروج از تجزیه کننده برنامه مبداء شامل درخت نحوی است که ساختارها یا گره ها به صورت اشیایی پیاده سازی می شوند. این اشیا با موارد زیر سروکار دارند: ساخت گره درخت نحوی ، کنترل انواع ، تولید کد میانی سه آدرسی (پکیج **inter** را ببینید).

الف – ۱ . برنامه مبداء

هر برنامه در این زبان شامل بلوکی با اعلان های اختیاری و دستورات است . نشانه **basic** انواع پایه را نشان می دهد.

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

اگر با انتساب ها به جای عملگرهایی در عبارات ، به صورت دستورات رفتار شود، ترجمه ساده می شود:

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

قوانین تولید مربوط به عبارات، شرکت پذیری در تقدم عملگرها را اداره می کند. آنها برای هر سطح تقدم از یک غیر پایانه و برای عبارات پرانتزدار، شناسه ها ، مرجع های آرایه و ثوابت از غیر پایانه یا فاکتور استفاده می کنند:

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

الف – ۲. Main

اجرا در متد **main** در کلاس **main** شروع می شود. متد **main** یک تحلیلگر لغوی در یک تجزیه کننده را ایجاد می کند و سپس متد **program** را در تجزیه کننده فراخوانی می نماید :

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

الف – ۳ . تحلیلگر لغوی

پکیج **lexer** بسط کد تحلیلگر لغوی در بخش ۵-۶-۲ است. کلاس **Tag** ثوابتی را برای نشانه ها تعریف می کند:

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

سه ثابت **INDEY** ، **MINUS** و **TEMP** نشانه های لغوی نیستند. در درخت های نحوی استفاده می شوند.

کلاس های **Token** و **Num** همانند بخش ۵-۶-۲ هستند که متد **toString** به آن اضافه شده است :

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

کلاس **Word** لغت های مربوط به کلمه های رزروی ، شناسه ها و نشانه های مرکب مثل **&&** را اداره می کند. علاوه براین برای مدیریت شکل نوشتاری عملگرها در کد میانی مثل منهای یکانی به کار می رود. به عنوان مثال، متن مبداء ۲ – دارای شکل میانی **minus 2** است .

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

کلاس **Real** برای اعداد ممیز شناور است :

```
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) }
```

متد اصلی در کلاس **Lexer** ، یعنی تابع **Scan** ، اعداد، شناسه ها و کلمه های رزرو شده را مانند آنچه که در بخش ۵-۶-۲ بحث شد، تشخیص می دهد.

خطوط ۱۳-۹ در کلاس **Lexer** کلمه های کلیدی انتخابی را رزرو می کند. خطو.ط ۱۶-۱۴ لغت هایی را برای اشیای **Word.True** و **Word.False** در کلاس **Word** تعریف شدند. اشیای مربوط به انواع پایه **int** ، **char** ، **bool** ، و **float** در کلاس **Type** تعریف شدند که زیرکلاس **Word** است. کلاس **Type** از پکیج **Symbol** است.

```
 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)    }
```

تابع **readch()** (خط ۱۸) برای خواندن کاراکتر بعدی در متغیر **peek** به کار می رود. نام **readch** دوباره استفاده شد تا به تشخیص نشانه های مرکب کمک کند. به عنوان مثال ، وقتی ورودی > دیده شد، فراخوانی **readch('=')** کاراکتر بعدی را در **peek** می خواند و بررسی می کند آیا = است یا خیر.

```
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)    }
```

تابع **scan** با حذف فضای سفید شروع می شود (خطوط ۳۰ – ۲۶). نشانه های مرکب مثل => (خطوط ۴۴-۳۱) و اعدادی مثل ۳۶۵ و ۳،۱۴ (خطوط ۵۸-۴۵) را قبل از جمع آوری کلمات (خطوط ۷۰-۵۹) تشخیص می دهد.

```
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)      }
```

سرانجام ، هر کاراکتر باقیمانده به عنوان نشانه برگردانده می شود (خطوط ۷۲-۷۱).

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

الف – ۴ . جدول های نماد و انواع

پکیج **symbol** جدول های نماد و انواع را پیاده سازی می کند.

کلاس **Env** همانند شکل ۳۷-۲ است . در حالی که کلاس **Lexer** رشته ها را به کلمه ها نگاشت می کند، کلاس **Env** نشانه های کلمه را به اشیای کلاس **Id** نگاشت می کند، که در پکیج **inter** به همراه کلاس هایی برای عبارات و دستورات تعریف شده است:

```
 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) }
```

کلاس **Type** را زیرکلاس **Word** تعریف می کنیم ، زیرا نام های نوع پایه مثل **int** کلمه های رزروی اند، که باید توسط تحلیلگر لغوی از لغت ها به اشیای مناسبی نگاشت شوند. اشیای مربوط به انواع پایه عبارتند از **Type.Int** ، **Type.Char Type.Float** ، و **Type.Bool** (خطوط ۱۰ – ۷). تمام این آرایه ها فیلد ارثی **tag** هستند، لذا تجزیه کننده با آنها یکسان رفتار می کند.

```
 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 );
```

توابع **numeric** (خطوط ۱۴-۱۱) و **max** (خطوط ۲۰-۱۵) برای تبدیل انواع مفیدند:

```
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) }
```

تبدیل ها بین انواع عددی **Type.Char** ، **Type.Int** ، و **Type.Float** مجاز است. وقتی عملگر محاسباتی به دو نوع عددی اعمال می شود، نتیجه بزرگتر از بین آنهاست.

آرایه ها تنها نوع ساختاری در زبان مبداء است، فراخوانی **super** در خط ۷ ، فیلد **width** را مقدار می دهد که برای محاسبه آدرس ضروری است. علاوه براین ، **lexeme** و **tok** را برابر با مقدار پیش فرض قرار می دهدکه استفاده نشدند.

```
 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) }
```

الف – ۵. کد میانی برای عبارات

پکیج **inter** شامل سلسله مراتب کلاس **Node** است. **Node** دو زیر کلاس دارد: **Expr** برای گره های عبارت و **Stmt** برای گره های دستور. این بخش ، **Expr** و زیر کلاس های آن را بحث می کند. بعضی از متدهای موجود در **Expr** با کد بولی و پرش سروکار دارند که به همراه سایر بخش های زیرکلاس **Expr** در بخش الف – ۶ بحث می شوند.

گره ها در درخت نحوی به عنوان اشیایی از کلاس **Node** پیاده سازی می شوند. گزارش خطا ، فیلد **lexline** (خط ۴ در فایل **Node.java** شماره خط مبداء ساختار را در این گره ذخیره می کند. خطوط ۱۰-۷ برای انتشار کد سه آدرسی به کار می روند:

```
 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) }
```

ساختارهای عبارت توسط زیر کلاس **Expr** اداره می شود. کلاس **Expr** دارای فیلدهای **op** و **type** (خطوط ۵-۴ فایل **Expr.java**) است که به ترتیب ، عملگر و نوع را در گره نشان می دهند.

```
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; }
```

متد **gen** (خط ۷) "جمله ای " را برمی گرداند که می تواند در سمت راست دستور سه آدرسی به کار می رود. با توجه به عبارت $E = E_1 + E_2$ ، متد **gen** جمله $x_1 + x_2$ را برمی گرداند که x_1, x_2 به ترتیب آدرس هایی برای هر مقدار E_1 و E_2 هستند، مقدار برگشتی **this** در صورتی مناسب است که این شیء ، آدرس باشد به زیر کلاس های **Expr** دوباره **gen** را پیاده سازی می کنند.

متد **reduce** (خط ۸) عبارت را محاسبه می کند یا به یگ آدرس "کاهش" می دهد. یعنی ، یک ثابت ، شناسه ، یا نام موقت را برمی گرداند. با توجه به عبارت **E** ، متد **reduce** مقدار موقت **t** را

برمی گرداند که شامل مقدار **E** است. **this** در صورتی یک مقدار برگشتی مناسب است که این شیء یک آدرس باشد.

بحث درباره متدهای **jumping** و **emitjumps** (خطوط ۱۸-۹) را به بخش الف – ۶ موکول می کنیم . آن ها کد پرش را برای عبارات بولی تولید می کنند.

```
 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) }
```

کلاس **Id** پیاده سازی های پیش فرض **gn** و **reduce** را در کلاس **Expr** به ارث می برد، زیرا شناسه ، یک آدرس است.

```
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) }
```

گره مربوط به شناسه کلاس **Id** ، یک برگ است. فراخوانی **super (id , p)** (خط ۵ در فایل **Id.java**) ، **id** و **p** را به ترتیب در فیلدهای **op** و **type** ذخیره می کند. فیلد **offset** (خط ۴) آدرس نسبی این شناسه را نگه می دارد.

کلاس **Op** پیاده سازی **reduce** را فراهم می کند (خطوط ۱۰-۵ در فایل **Op.java**) که توسط زیر کلاس **Arith** برای عملگرهای محاسباتی ، **Unary** برای عملگرهای یکانی ، **Access** برای دستیابی به آرایه ، به ارث برده می شود. در هر مورد ، **reduce** ، متد **gen** را برای تولید یک جمله فراخوانی می کند، دستوری را مسترد می کند تا این جمله را به نام موقت جدید نسبت دهد، و مقدار موقت را برمی گرداند.

```
 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) }
```

کلاس **Arith** عملگرهای دودویی مثل + و * را پیاده سازی می کند. سازنده **Arith** با فراخوانی **super (tok, null)** (خط ۶) شروع می شود که **tok** نشانه ای است که عملگر را نشان می دهد و **null** جانگهداری برای نوع است . نوع در خط ۷ با استفاده از **type.max** تعیین می شود که بررسی می کند آیا دو عملوند می توانند به نوع عددی مشترکی تبدیل شوند یا خیر . کد مربوط به **type.max** در بخش الف – ۴ آمده است. اگر بتوانند تبدیل شوند، **type** برابر با نوع نتیجه می شود ، در غیر این صورت ، خطای نوع گزارش داده می شود (خط ۸) . این کامپایلر ساده ، انواع را بررسی می کند، اما تبدل نوع را انجام نمی دهد.

```
 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) }
```

متد **gen** سمت راست دستور سه آدرسی را به وسیله کاهش زیر عبارت به آدرس ها و اعمال عملگر به آدرس ها می سازد (خط ۱۱ در فایل **Arith.java**) . به عنوان مثال ، فرض کنید **gen** در ریشه برای **a + b * c** فراخوانی شود. فراخوانی **reduce** مقدار **a** را به عنوان آدرس زیرعبارت **a** و متغیر موقت

t را به عنوان آدرس **b * c** برمی گرداند. در این مدت ، **reduce** دستور **t = b * c** را منتشر می کند. متد **gen** گره **Arith** جدید را برمی گرداند، به طوری که عملگر * و آدرس های **a** و **t** عملوندهای آن هستند.

توجه به این نکته ارزشمند است که نام های موقت دارای نوع هستند. بنابراین سازنده **Temp** با یک پارامتر نوع فراخوانی می شود (خط ۶ در فایل **Temp.java**).

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

کلاس **Unary** همتای یک پارامتری کلاس **Arith** است :

```
1) package inter;                    // File Unary.java
2) import lexer.*; import symbols.*;
3) public class Unary extends Op {
4)    public Expr expr;
5)    public Unary(Token tok, Expr x) {    // handles minus, for ! see 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) }
```

الف – ۶. کد پرش برای عبارات بولی

کد پرش برای عبارات بولی **B** توسط متد **jumping** تولید می شود که دو برچسب **t** و **f** را به عنوان پارامتر می پذیرد که به ترتیب خروجی های درست و نادرست **B** نامیده می شوند، اگر ارزش **B** برابر با **true** باشد، این کد شامل پرش به **t** وگرنه پرش به **f** است. طبق قرارداد، برچسب چند به معنای این است که کنترل از **B** خارج می شود و به دستور بعد از **B** می رود.

با کلاس **Constant** شروع می کنیم . سازنده **Constant** در خط ۴ نشانه **tok** و نوع **p** را به عنوان پارامتر می پذیرد . برگی را در درخت نحوی می سازد که دارای برچسب **tok** و نوع **p** است . برای سهولت ، سازنده **Constant** دوباره تعریف شد (خط ۵) تا شیء ثابت را از یک مقدار صحیح بسازد.

```
 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) }
```

متد **jumping** (خطوط ۱۲-۹ در فایل **Constant.java**) دو پارامتر دارد که برچسب های **t** و **f** هستند. اگر این ثابت شیء ایستای **True** باشد (تعریف شده در خط ۷) و **t** برچسب خاص چند نباشد، آنگاه پرش به **t** تولید می شود. در غیر این صورت ، اگر شیء **False** باشد (خط ۸) و **f** غیر صفر باشد، آنگاه پرش به **f** تولید می شود.

کلاس **Logical** عملگرهای مشترکی را برای کلاس های **Or** ، **And** و **Not** فراهم می کند. فیلدهای **x** و **y** (خط ۴) متناظر با عملوندهای عملگر منطقی اند. (گرچه کلاس **Not** عملگر یکانی را پیاده سازی می کند، برای سهولت ، زیرکلاس **Logical** است).سازنده **Logic (tok, a, b)** (خطوط ۱۰-۵) گره نحو را با عملگر **tok** و عملوندهای **a** و **b** می سازد. برای این کار، از تابع **check** استفاده می کند تا تضمین کند **a** و **b** بولی اند. متد **gen** در انتهای این بخش بحث می شود.

```
 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) }
```

در کلاس **Or** ، متد **jumping** (خطوط ۱۰-۵) کد پرش را برای عبارت بولی $B = B_1 \| B_2$ تولید می کند. فعلاً ، فرض کنید نه خروج درست **t** و نه خروج نادرست **f** مربوط به **B** ، برچسب خاص صفر نیستند. چون اگر B_1 درست باشد **B** نیز درست است ، خروج درست B_1 باید **t** باشد و خروج نادرست متناظر با اولین دستور B_2 است. خروج های درست و نادرست B_2 همانند **B** است .

```
 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) }
```

در حالت کلی، یعنی خروج درست از **B** می تواند برچسب خاص صفر باشد. متغیر **label** (خط ۶ در فایل **Or.java**) تضمین می کند که خروج درست B_1 به درستی برابر با انتهای کد مربوط به **B** قرار می گیرد. اگر **t** صفر باشد، آنگاه **label** برابر با برچسب جدیدی می شود که پس از تولید کد مربوط به B_1 و B_2 منتشر می شود.

کد مربوط به کلاس **And** شبیه **Or** است .

```
 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) }
```

کلاس **Not** با ساطر عملگرهای بولی اشتراکات کافی دارد که می توان آن را به عنوان زیرکلاس **Logical** در نظر گرفت ، گرچه **Not** عملگر یکانی را پیاده سازی می کند. کلاس پایه دو عملوند را انتظار دارد، لذا **b** دوبار در فراخوانی **super** در خط ۴ ظاهر می شود. فقط **y** (خط ۴ فایل **Logical.java**) در متدهای خطوط ۶-۵ استفاده می شود. در خط ۵ ، متد **jumping** فراخوانی **y.jumping** را انجام می دهد که خروج های درست و نادرست رزرو شده اند.

```
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) }
```

کلاس **Rel** عملگرهای < ، <= ، == ، =! ، =< ، و > را پیاده سازی می کند. تابع **check** (خطوط ۹-۵) بررسی می کند که دو عملوند همنوع هستند و آرایه نیستند. برای سهولت ، تبدیل نوع مجاز نیست.

```
 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) }
```

متد **jumping** (خطوط ۱۵-۱۰ در فایل **Rel.java**) با تولید کد برای زیرعبارات **x** و **y** (خطوط ۱۲-۱۱) شروع می شود. سپس متد **jumping** را فراخوانی می کند که در خطوط ۱۸-۱۰ فایل **Expr.java** در بخش الف-۵ تعریف شد. اگر **t** و **f** برچسب صفر نباشند، آنگاه **emitjumps** کد زیر را اجرا می کند:

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

اگر **t** یا **f** برچسب خالی صفر باشند (از فایل **Expr.java**) حداکثر یک دستور تولید می شود:

```
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
```

برای کاربرد دیگر **emitjump** ، کد مربوط به کلاس **Access** را در نظر بگیرید . زبان مبداء اجازه می دهد مقادیر بولی به شناسه ها و عناصر آرایه نسبت داده شوند، لذا عبارت بولی می تواند دستیابی به آرایه باشد. کلاس **Access** دارای متد **gen** برای تولید کد "عادی " و متد **jumping** برای کد پرش است . متد **jumping** (خط ۱۱) متد **emitjump** را پس از کاهش این دستیابی آرایه به یک مقدار موقت فراخوانی می کند. سازنده (خطوط ۹-۶) با آرایه تخت **a** ، اندیس **i** ، و نوع **p** مربوط به یک عنصر در آرایه تخت فراخوانی می شود. کنترل نوع در اثنای محاسبه آدرس آرایه صورت می گیرد.

```
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) }
```

کد پرش نیز می تواند برای برگرداندن مقدار بولی به کار رود. کلاس **Logical** دارای متد (خطوط ۲۴-۱۵) است که مقدار موقت **temp** را برمی گرداند، که مقدارش توسط جریان کنترل از طریق کد پرش مربوط به این عبارت تعیین می گردد. در خروج درست این عبارت بولی ، **temp** برابر با **true** می شود، در خروج نادرست ، **temp** برابر با **false** می شود. متغیر موقت در خط ۱۷ اعلان می شود، به طوری که خروج درست برابر با دستور بعدی و خروج نادرست برابر با برچسب جدید **f** می شود. دستور بعدی **true** را در **temp** قرار می دهد (خط ۱۹) که پس از آن یک پرش به برچسب جدید **a** وجود دارد (خط ۲۰) . کد موجود در خط ۲۱ برچسب **f** و دستوری را منتشر می کندکه **false** را به **temp** نسبت می دهد. این قطعه که با برچسب **a** خاتمه می یابد که در خط ۲۲ تولید شد. سرانجام ، **gen** مقدار **temp** را برمی گرداند (خط ۲۳).

الف – ۷ . کد میانی برای دستورات

هر ساختار دستور توسط زیرکلاس **Stmt** پیاده سازی می شود. فیلدهای مربوط به قطعات یک ساختار در زیر کلاس مربوط وجود دارد. به عنوان مثال ، **While** دارای فیلدهایی برای تست عبارت و یک زیردستور است.

خطوط ۴-۳ در کد زیر برای کلاس **Stmt** با ساختار درخت نحوی سروکار دارد. سازنده **Stmt()** کاری انجام نمی دهد، زیرا کار در زیر کلاس انجام می شود. شیء ایستای **Stmt.Null** (خط ۴) دنباله خالی از دستورات را نشان می دهد.

```
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) }
```

خطوط ۷-۵ با تولید کد سه آدرسی سروکار دارد. متد **gen** با دو برچسب **b** و **a** فراخوانی می شود، که **b** ابتدای کد مربوط به این دستور و **a** ابتدای دستور بعد از کد مربوط به این دستور را مشخص می کند. متد **gen** خط (۵) یگ جانگهدار برای متدهای **gen** در زیرکلاس ها است. زیر کلاس های **While** و **Do** برچسب خود را در فیلد **after** (خط ۶) ذخیره می کنند، لذا می توانند توسط هر دستور **break** برای پرش به ساختار در برگیرنده به کار گرفته شود. شیء **Stmt.Enclosing** در اثنای ترجمه برای مشخص کردن ساختار دربرگیرنده به کار می رود.

سازنده کلاس **If** ، گره ای را برای دستور **if (E) S** می سازد. فیلدهای **expr** و **stmt** به ترتیب گره هایی را برای **E** و **S** نگه می دارند. توجه کنید که **expr** فیلدی در کلاس **Expr** است. به طور مشابه ، **stmt** نام فیلدی در کلاس **Stmt** است .

```
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) }
```

کد مربوط به شیء **If** شامل کد پرش برای **expr** است که پس از آن کد مربوط به **stmt** قرار دارد. همان طور که در بخش الف – ۶ گفته شد، فراخوانی **expr.jumping (0, f)** در خط ۱۱ مشخص می کند که اگر **expr** به **true** ارزیابی شود، کنترل باید به پایین کد مربوط به **expr** منتقل شود وگرنه باید به برچسب **a** منتقل شود.

پیاده سازی کلاس **Else** که شرط هایش بخش **else** را اداره می کند، شبیه کلاس **If** است :

```
 1) package inter;                    // File 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 for stmt1
11)       int label2 = newlabel();   // label2 for stmt2
12)       expr.jumping(0,label2);    // fall through to stmt1 on true
13)       emitlabel(label1); stmt1.gen(label1, a); emit("goto L" + a);
14)       emitlabel(label2); stmt2.gen(label2, a);
15)    }
16) }
```

ساخت شیء **While** بین سازنده **()While** که گره ای با فرزندان تهی را ایجاد می کند (خط ۵)، و تابع مقداردهی اولیه **init(n, s)** که فرزند **expr** را برابر با **x** و فرزند **stmt** را برابر با **s** قرار می دهد (خطوط ۹-۶) ، شکسته می شود. تابع **gen (b, a)** برای تولید کد سه آدرسی (خط ۱۶-۱۰) هسته تابع متناظر **gen** در کلاس **If** است . تفاوت آن ها این است که برچسب **a** در فیلد **after** (خط ۱۱) ذخیره می شود و در انتهای کد مربوط به **stmt** پرش به **b** (خط ۱۵) برای تکرار بعدی حلقه **While** وجود دارد.

```
 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) }
```

کلاس **Do** خیلی شبیه به کلاس **While** است .

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

کلاس **Set** ، انتساب ها را شناسه ای در سمت چپ و یک عبارت در سمت راست پیاده سازی می کند. اغلب که در کلاس **Set** مربوط به ساخت گره و کنترل انواع است (خطوط (۱۳-۵) . تابع **gen** دستور سه آدرسی را منتشر می کند (خطوط ۱۶-۱۴).

```
 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) }
```

کلاس **SetElem** انتساب ها به عنصر آرایه را پیاده سازی می کند:

```
 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();
18)       emit(array.toString() + " [ " + s1 + " ] = " + s2);
19)    }
20) }
```

کلاس **Set** دنباله ای از دستورات را پیاده سازی می کند. تست های مربوط به دستورات تهی در خطوط ۷-۶ برای اجتناب از برچسب ها است. توجه کنید که برای دستور تهی **stmt.null** کدی تولید نمی شود، زیرا متد **gen** در کلاس **stmt** کاری انجام نمی دهد.

```
 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) }
```

دستور **break** کنترل را به خارج از حلقه یا دستور **switch** دربرگیرنده انتقال می دهد. کلاس **Break** از فیلد **stmt** برای ذخیره ساختار دستور دربرگیرنده استفاده می کند (تجزیه کننده تضمین می کند

که **Stmt.Enclosing** گره درخت نحوی مربوط به ساختار در برگیرنده را نشان می دهد). کد مربوط به شیء **Break** پرش به چپ **stmt.after** است که دستور بلافاصله بعد از کد مربوط به **stmt** را مشخص می کند.

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

الف – ۸ . تجزیه کننده

تجزیه کننده ، رشته ای از نشانه ها را می خواند و با فراخوانی توابع سازنده مناسبی از بخش های الف – ۵ تا الف – ۷ ، درخت نحوی را می سازد. جدول نماد فعلی همانند الگوی ترجمه در شکل ۳۸-۲ در بخش ۷-۲ است.

پکیج **parser** شامل یک کلاس به نام **Parser** است :

```
 1) package parser;                   // File Parser.java
 2) 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 tagen
 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)    }
```

همانند مترجم عبارت ساده ی موجود در بخش ۵-۲ ، کلاس **Parser** دارای رویه ای برای هر غیر پایانه است . این رویه ها مبتنی بر گرامری است که با حذف بازگشتی چپ از گرامر زبان مبداء در بخش الف – ۱ ایجاد می شود.

تجزیه ، با فراخوانی به رویه **program** شروع می شود، که **()block** را فراخوانی می کند (خط ۱۶) تا رشته ورودی را تجزیه کند و درخت نحوی را بسازد. خطوط ۱۸-۱۷ کد میانی را تولید می کند.

```
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)   }
```

اداره کردن جدول نماد صریحاً در رویه **block** آمده است. متغیر **top** (که در خط ۵ اعلان شد)، جدول نماد بالایی را نگه می دارد، متغیر **saveEnv** (خط ۲۱) پیوندی به جدول نماد قبلی است.

```
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)   }
```

اعلان ها، وارده های جدول نماد را برای شناسه ها ایجاد می کند (خط ۳۶). گرچه در این جا نشان داده شد، اعلان ها می توانند موجب شوند که دستورات حافظه را برای شناسه ها در زمان اجرا رزرو نمایند.

```
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
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)   }
```

رویه **stmt** دارای دستور **switch** با **case** هایی است که متناظر با قوانین تولید غیر پایانه **Stmt** هستند. هر **case** ، گره ای را برای یک ساختار می سازد. برای این کار از توابع سازنده بحث شده در بخش الف – ۷ استفاده می شود گره های مربوط به دستورات **while** و **do** وقتی ساخته می شوند که تجزیه کننده کلمه کلیدی بازکننده را ببیند، گره ها قبل از تجزیه دستور ساخته می شوند تا اجازه دهد هر دستور **break** بتواند به حلقه دربرگیرنده برگردد. حلقه های تودرتو با استفاده از متغطر **Stmt.Enclosing** در کلاس **Stmt** و **savedStmt** (اعلان شده در خط ۵۲) ، برای نگهداری حلقه دربرگیرنده فعلی استفاده می کند.

```
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;
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)   }
```

برای سهولت ، کد مربوط به انتساب ها در رویه کمکی **assign** قرار می گیرد.

```
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)  }
```

تجزیه عبارات محاسباتی و بولی شبیه است . در هر دو مورد، یک گره درخت نحوی مناسب ایجاد می شود. تولید کد برای این ها متفاوت است . در بخش های الف – ۵ و الف – ۶ بحث شد.

```
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)    }
```

بقیه کد در تجزیه کننده با فاکتورها در عبارات سروکار دارد. رویه کمکی **offset** کدی را برای محاسبه آدرس تولید می کند که در بخش ۳-۴-۶ بحث شد.

```
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;  // inherit id
186)       Type type = a.type;
187)       match('['); i = bool(); match(']');     // first index, 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 == '[' ) {      // multi-dimensional 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) }
```

الف – ۹. ایجاد بخش جلویی کامپایلر

کد مربوط به پکیج ها در پنج دایرکتوری **main** ، **lexer** ، **symbole** ، **parser** ، و **inter** ذخیره می شوند. فرمان های ایجاد کامپایلر، از سیستمی به سیستم دیگر فرق می کند. در پیاده سازی های یونیکس به صورت زیر عمل می شود:

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

فرمان **javac** فایل های **class.** را برای هر کلاسی ایجاد می کند. مترجم می تواند با تایپ **Main** ، **java** **main** و سپس نام برنامه مبداء برای ترجمه عمل می کند. مثلاً ، محتویات فایل **text** به صورت زیر است :

```
1) {                    // File 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) }
```

با این ورودی ، بخش جلویی کامپایلر، خروجی زیر را تولید می کند:

```
 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:
```

امتحان کنید.

پیوست ب

یافتن راه حل های مستقل خطی

الگوریتم ب – ۱ : یافتن مجموعه ماکزیممی از راه حل های مستقل خطی بـر $A\vec{x} \geq \vec{0}$ ، و بیـان آن هـا برحسب سطرهایی از ماتریس **B** .

ورودی : ماتریس **A** به ابعاد $n \times m$.

خروجی : ماتریس **B** از راه حل های مستقل خطی برای $A\vec{x} \geq \vec{0}$.

روش انجام کار : شبه کد الگوریتم در ادامه آمده است. توجه کنید که **X[y]** نشان دهنده سـطر **y** مـاتریس **X** ، **X[x : y]** نشان دهنده سطرهای **y** تا **z** ماتریس **X** ، و **X[y : z][u : v]** نشان دهنده چهارگوش ماتریس **X** در سطرهای **y** تا **z** و ستون های **u** تا **v** .

$M = A^T$;
$r_0 = 1$;
$c_0 = 1$;
$B = I_{n \times n}$; /* an n-by-n identity matrix */

while (**true**) {

/* **1.** Make $M[r_0 : r' - 1][c_0 : c' - 1]$ into a diagonal matrix with positive diagonal entries and $M[r' : n][c_0 : m] = 0$. $M[r' : n]$ are solutions. */
$r' = r_0$;
$c' = c'_0$;
while (there exists $M[r][c] \neq 0$ such that $r - r'$ and $c - c'$ are both ≥ 0) {
Move pivot $M[r][c]$ to $M[r'][c']$ by row and column interchange;
Interchange row r with row r' in B;
if ($M[r'][c'] < 0$) {
$M[r'] = -1 * M[r']$;
$B[r'] = -1 * B[r']$;
}
for ($row = r_0$ to n) {
if ($row \neq r'$ and $M[row][c'] \neq 0$ {
$u = -(M[row][c']/M[r'][c'])$;
$M[row] = M[row] + u * M[r']$;
$B[row] = B[row] + u * B[r']$;
}
}
$r' = r' + 1$;
$c' = c' + 1$;
}

```
/* 2. Find a solution besides M[r' : n]. It must be a
            nonnegative combination of M[r_0 : r' - 1][c_0 : m] */
Find k_{r_0}, ..., k_{r'-1} ≥ 0 such that
            k_{r_0} M[r_0][c' : m] + ··· + k_{r'-1} M[r' - 1][c' : m] ≥ 0;
if ( there exists a nontrivial solution, say k_r > 0 ) {
      M[r] = k_{r_0} M[r_0] + ··· + k_{r'-1} M[r' - 1];
      NoMoreSoln = false;
}     else /* M[r' : n] are the only solutions */
      NoMoreSoln = true;
```

```
/* 3. Make M[r_0 : r_n - 1][c_0 : m] ≥ 0 */
if ( NoMoreSoln ) { /* Move solutions M[r' : n] to M[r_0 : r_n - 1] */
     for ( r = r' to n )
          Interchange rows r and r_0 + r - r' in M and B;
     r_n = r_0 + n - r' + 1;
else { /* Use row addition to find more solutions */
     r_n = n + 1;
     for ( col = c' to m )
          if ( there exists M[row][col] < 0 such that row ≥ r_0 )
               if ( there exists M[r][col] > 0 such that 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;
                              Interchange M[row] with M[r_n];
                              Interchange B[row] with B[r_n];
                         }
}
```

```
/* 4. Make 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 {
            Pick an r such that M[r][col] > 0 and 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. If necessary, repeat with rows M[r_n : n] */
    if ( (NoMoreSoln or r_n > n or r_n == r_0) {
        Remove rows r_n to n from B;
        return B;
    }
    else {
        c_n = m + 1;
        for ( col = m to 1 step -1 )
            if ( there is no M[r][col] > 0 such that r < r_n {
                c_n = c_n - 1;
                Interchange column col with c_n in M;
            }
        r_0 = r_n;
        c_0 = c_n;
    }

}
```