

Chapter 15

Compiling Control Structures

while loop

```
while (x)
{
    print(x);
    x = x - 1;
}
```

Assembly code for while loop

```
@L0:                                ; first label
    p        x                      ; code generated by expr()
    jz        @L1                    ; jump on false to exit loop

    p        x                      ; body of loop
    dout
    pc        x
    p        x
    pwc       1
    sub
    stav

    ja        @L0                    ; jump back to exit test
@L1:                                ; second label
```

Translation grammar for while statement

```
void whileStatement(): {String label1, label2;}  
{  
    "while"  
    {label1=cg.getLabel();}  
    {cg.emitLabel(label1);}  
    "("  
    expr()  
    ")"  
    {label2=cg.getLabel();}  
    {cg.emitInstruction("jz", label2);}  
    statement()  
    {cg.emitInstruction("ja", label1);}  
    {cg.emitLabel(label2);}  
}
```

if statement

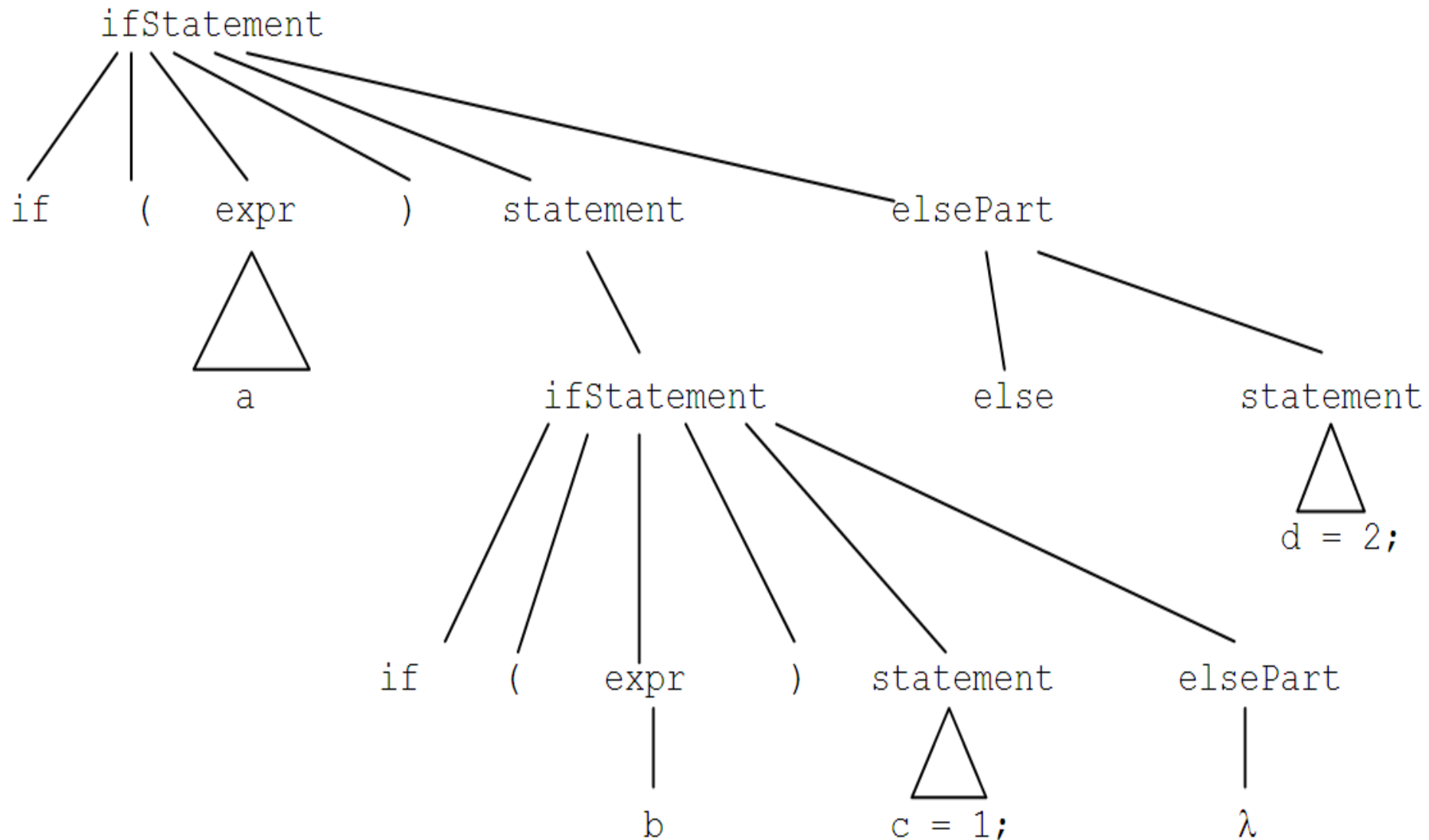
```
ifStatement → "if" "(" expression ")" statement elsePart  
elsePart   → "else" statement  
elsePart   →  $\lambda$ 
```

Second, this grammar, which is the standard grammar for the `if` statement, is ambiguous. We can demonstrate this by showing two parse trees for the nested `if` statement

```
if (a)  
  if (b)  
    c = 1;  
  else  
    d = 2;
```

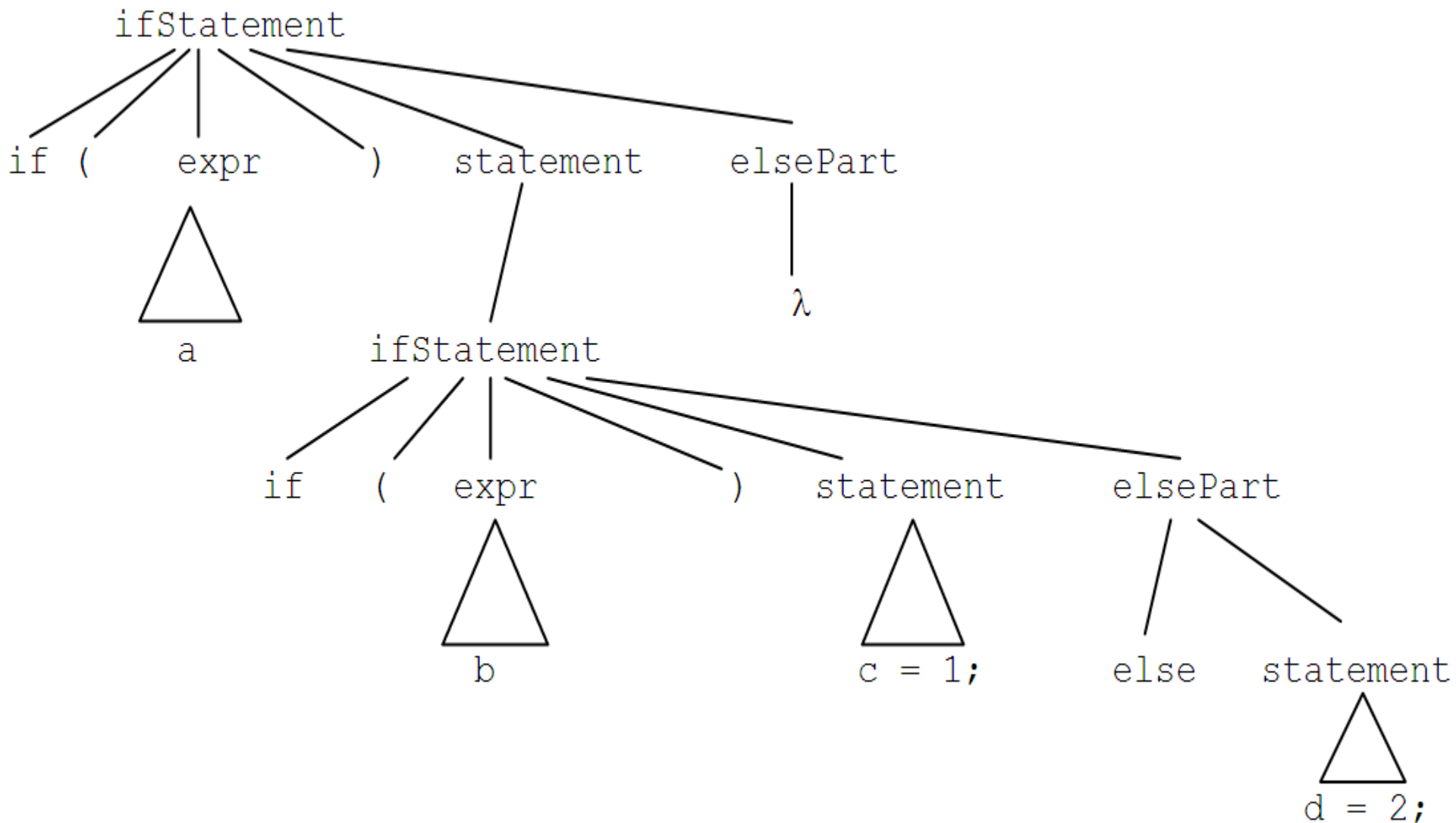
Parse tree 1

a)



Parse tree 2

b)

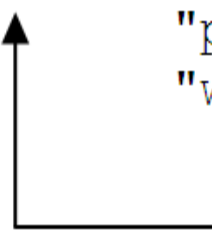


Disambiguate if statement

`elsePart` \rightarrow "else" statement
`elsePart` $\rightarrow \lambda$

Selection Set

```
{"else"}  
{"else", <ID>, "println",  
  "print", ";", "{", "}",  
  "while", "if", "do" }
```

 delete this else

Translation grammar if statement

```
void ifStatement(): {String label1;}
{
    "if"
    "("
    expr()
    ")"
    {label1=cg.getLabel();}
    {cg.emitInstruction("jz", label1);}
    statement()
    elsePart(label1)
}
```

```
void elsePart(String label1): {String label2;}
{
    "else"
    {label2=cg.getLabel();}
    {cg.emitInstruction("ja", label2);}
    {cg.emitLabel(label1);}
    statement()
    {cg.emitLabel(label2);}
    |
    {cg.emitLabel(label1);}           // if no else part
}
```

Range checking

```
1 void factor(): {Token t;}
2 {
3     t=<UNSIGNED>
4     {cg.emitInstruction("pwc", t.image);}
5     |
6     "+"
7     t = <UNSIGNED>
8     {cg.emitInstruction("pwc", t.image);}
9     |
10    "-"
11    t = <UNSIGNED>
12    {cg.emitInstruction("pwc", "-" + t.image);}
13    |
14    t=<ID>
15    {symTab.enter(t.image);}
16    {cg.emitInstruction("p", t.image);}
17    |
18    "("
19    expr()
20    ")"
21 }
```

Inserting checks

To specify range checking in our translation grammar, we simply insert actions to perform the required checks. We insert

```
{  
    if (t.image.length() > 5 ||  
        Integer.parseInt(t.image) > 32767)  
        throw genEx("integer (-32768 to 32767)");  
}
```

after lines 3 and 7 in Fig. 15.8, and we insert

```
{  
    if (t.image.length() > 5 ||  
        Integer.parseInt(t.image) > 32768)  
        throw genEx("integer (-32768 to 32767)");  
}
```

after line 11.

Handling backslash-quote

```
println("hello\\\\"bye");
```

If a quote in a string constant is preceded by an even number (including zero) of backslashes, the quote is the terminating quote. If, however, it is preceded by an odd number of backslashes, then it is not the terminating quote.

Backslash-quote in JavaCC

```
15 MORE:
16 {
17     "\"": IN_STRING      // matches initial quote in string
18 }
```

Using MORE blocks

```
55 <IN_STRING>
56 MORE:
57 {
58     "\\\""           // matches backslash, quote
59     |
60     "\\\"           // matches backslash, backslash
61     |
62     <~["\"", "\n", "\r"]> // everything else except ", \n, \r
63 }
64 //-----
65 <IN_STRING>
66 TOKEN:
67 {
68     <STRING: "\"">    // matches terminating quote
69 }
```

Universal block

```
TOKEN:  // active when token manager in the DEFAULT state
{
```

```
    // no catch-all expression in this block
```

```
}
```

```
//=====
```

```
<DOG>          // active when token manager is in DOG state
```

```
TOKEN:
```

```
{
```

```
    // no catch-all expression in this block
```

```
}
```

```
//=====
```

```
<CAT>          // active when token manager is in CAT state
```

```
TOKEN:
```

```
{
```

```
    // no catch-all expression in this block
```

```
}
```

```
//=====
```

```
<*>          // <*> marks a universal block
```

```
TOKEN:
```

```
{
```

```
    <ERROR: ~[]>
```

```
}
```

Handling strings that span lines

```
println("hello\
bye");
```

```
pc      @L0
sout
^@L0: "hellobye"
```


Handling strings that span lines JavaCC

```
15 MORE:
16 {
17     "\"": IN_STRING        // matches initial quote in string
18 }
```

Using matchedToken

```
53 <IN_STRING>
54 MORE:
55 {
56     "\\\""           // matches backslash quote
57     |
58     "\\\"\\\"       // matches backslash backslash
59     |
60     "\\\"r\\n\"      // matches backslash \r \n
61     |
62     "\\\"n\"         // matches backslash \n
63     |
64     "\\\"r\"         // matches backslash \r
65     |
66     <~["\"", "\n", "\r"]> // all except ", \n, and \r
67 }
68 //-----
69 <IN_STRING>
70 TOKEN:
71 {
72     <STRING: "\""> // matches terminating quote
73     {
74         matchedToken.image =
75             matchedToken.image.replace("\\\"r\\n", "");
76         matchedToken.image =
77             matchedToken.image.replace("\\\"r", "");
78         matchedToken.image =
79             matchedToken.image.replace("\\\"n", "");
80     } : DEFAULT
81 }
```

Another approach

```
53 <IN_STRING>
54 MORE:
55 {
56     "\\n"      // remove backslashed newline from image
57     {image.setLength(image.length() - 2);}
58 |
59     "\\r"      // remove backslashed ret from image
60     {image.setLength(image.length() - 2);}
61 |
62     "\\r\n"    // remove backslashed ret/newline from image
63     {image.setLength(image.length() - 3);}
64 |
65     "\\\""      // matches backslash, quote
66 |
67     "\\\"\\\"  // matches backslash, backslash
68 |
69     <~["\"", "\n", "\r"]> // all except ", \n, \r
70 }
71 //-----
72 <IN_STRING>
73 TOKEN:
74 {
75     <STRING: "\""> // matches terminating quote
76     // set image field in token to StringBuffer image
77     {matchedToken.image = image.toString();} : DEFAULT
78 }
```

Error recovery

```
1 private void statement()
2 {
3     try
4     {
5         switch(currentToken.kind)
6         {
7             case ID:
8                 assignmentStatement();
9                 break;
10            case PRINTLN:
11                printlnStatement();
12                break;
13            default:
14                throw genEx("statement");
15        }
16    }
17    catch(RuntimeException e)
18    {
19        System.err.println(e.getMessage());
20        cg.emitString("; " + e.getMessage());
21
22        // advance past next semicolon
23        while (currentToken.kind != SEMICOLON &&
24               currentToken.kind != EOF)
25            advance();
26        if (currentToken.kind != EOF)
27            advance();
28    }
29 }
```

statementList() for error recovery

a) Original code in S1 for statementList()

```
1 private void statementList()
2 {
3     switch(currentToken.kind)
4     {
5         case ID:
6         case PRINTLN:
7             statement();
8             statementList();
9             break;
10        case EOF:
11            ;
12            break;
13        default:
14            throw genEx("Expecting statement or <EOF>");
15    }
16 }
```

b) New statementList() method for error recovery

```
1 private void statementList()
2 {
3     if (currentToken.kind == EOF)
4         return;
5     statement();
6     statementList();
7 }
```

New compoundStatement() method

- 1) Leave `statementList()` as it is in Fig. 15.13b to keep error recovery fully in effect.
- 2) Create a new non-terminal `compoundList()` similar to `statementList()` in Fig. 15.13a.

```
private void compoundList()  
{  
    if (currentToken.kind == RIGHTBRACE)  
        return;  
    statement();  
    compoundList();  
}
```

- 3) Replace the call of `statementList()` in `compoundStatement()` with a call of `compoundList()`.

Error recovery JavaCC

```
8 void statementList(): {}
9 {
10     {if (getToken(1).kind==EOF) return;}
11     statement()
12     statementList()
13 }
14 //-----
15 void statement(): {}
16 {
17     try
18     {
19         assignmentStatement()
20     |
21         printlnStatement()
22     }
23     catch(ParseException e)
24     {
26         System.err.println(e.getMessage());
27         cg.emitString(e.getMessage());
28
29         while (getToken(1).kind != SEMICOLON &&
30             getToken(1).kind != EOF)
31             getNextToken();           // like advance() in S1
32         if (getToken(1).kind != EOF)
33             getNextToken();           // like advance() in S1
34     }
35 }
```