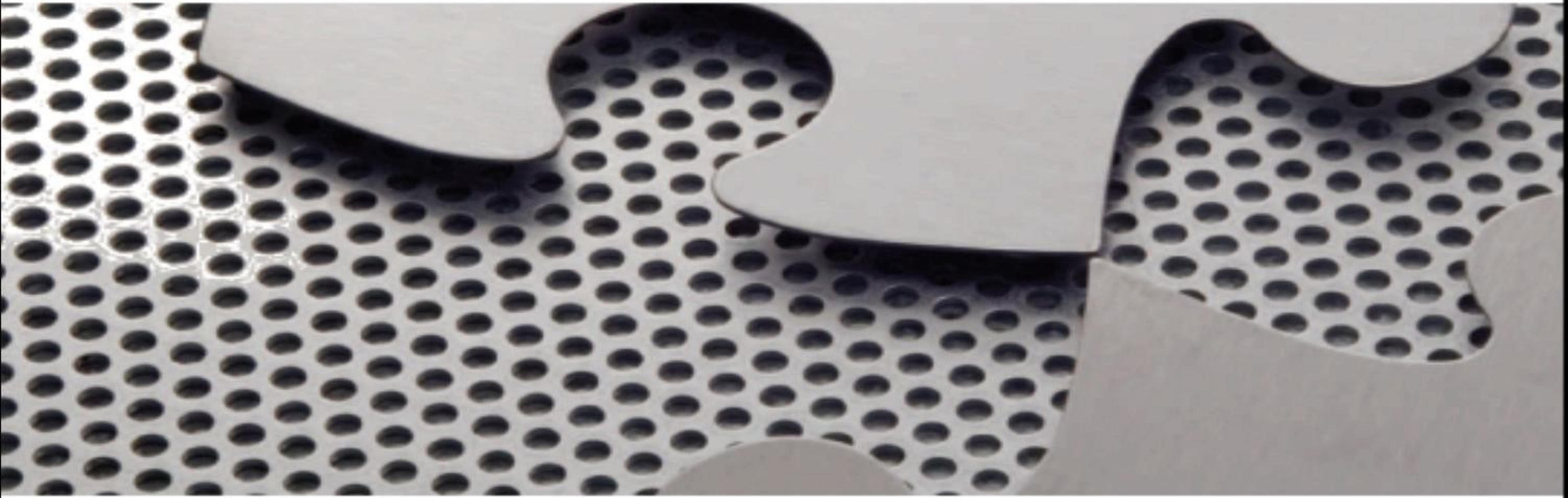
Programming Languages
Third Edition



Chapter 9

Control I – Expressions and Statements

Objectives

- Understand expressions
- Understand conditional statements and guards
- Understand loops and variation on WHILE
- Be familiar with the GOTO controversy and loop exits
- Understand exception handling
- Compute the values of static expressions in TinyAda

Introduction

- This chapter discusses basic and structured abstraction of control through the use of expressions and statements
- **Expression**: returns a value and produces no side effect
- **Statement**: executed for its side effects and returns no value
- In functional languages (also called **expression languages**), virtually all language constructs are expressions

Introduction (cont’d.)

- C could be called an **expression-oriented language**
  - Expressions make up a much larger portion of the language than statements
- If no side effects, expressions are closest in appearance to mathematics
  - Have semantics similar to those of mathematical expressions
- Semantics of expressions with side effects have a significant control component

Introduction (cont’d.)

- Explicit control structures first appeared as GOTOs
- Algol60 brought **structured control**
  - Control statements transfer control to and from statements that are **single-entry**, **single exit**, such as **blocks**
- Some languages do away with GOTOs altogether, but there is still debate on the utility of GOTOs within the context of structured programming

Expressions

- Basic expressions consist of literals and identifiers
- Complex expressions are built up recursively from basic expressions by the application of operators and functions
  - May involve grouping symbols such as parentheses
- Example: in the expression 3 + 4 * 5
  - + **operator** is applied to its two **operands**, 3 and the subexpression 4 * 5
- **Unary operator**: takes one operand
- **Binary operator**: takes two operands

Expressions (cont'd.)

- Operators can be written in infix, postfix, or prefix notation
  - Postfix and prefix forms do not require parentheses to express the order in which operators are applied
- Operators are predefined, written in infix form (if binary), with special associativity and precedence rules
- Functions are user-defined, with the operands viewed as **arguments** or **actual parameters**
- This distinction is arbitrary, since operators and functions are equivalent concepts

Expressions (cont'd.)

- Distinction is significant, since built-in operators were implemented as highly optimized **inline code**
  - Functions required the building of **activations**
- Modern translators often inline even user-defined functions
- Lisp requires expressions to be **fully parenthesized** because it can take variable numbers of arguments as operands
- **Applicative order evaluation** (or **strict evaluation**) rule: all operands are evaluated first, then operators are applied to them

Expressions (cont'd.)

- Example: applicative order evaluation
  - The + and – nodes are evaluated to 7 and -1
  - Then the * is applied to get -7

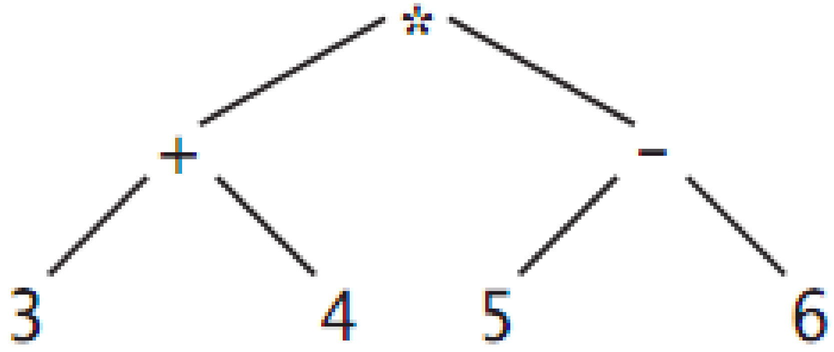
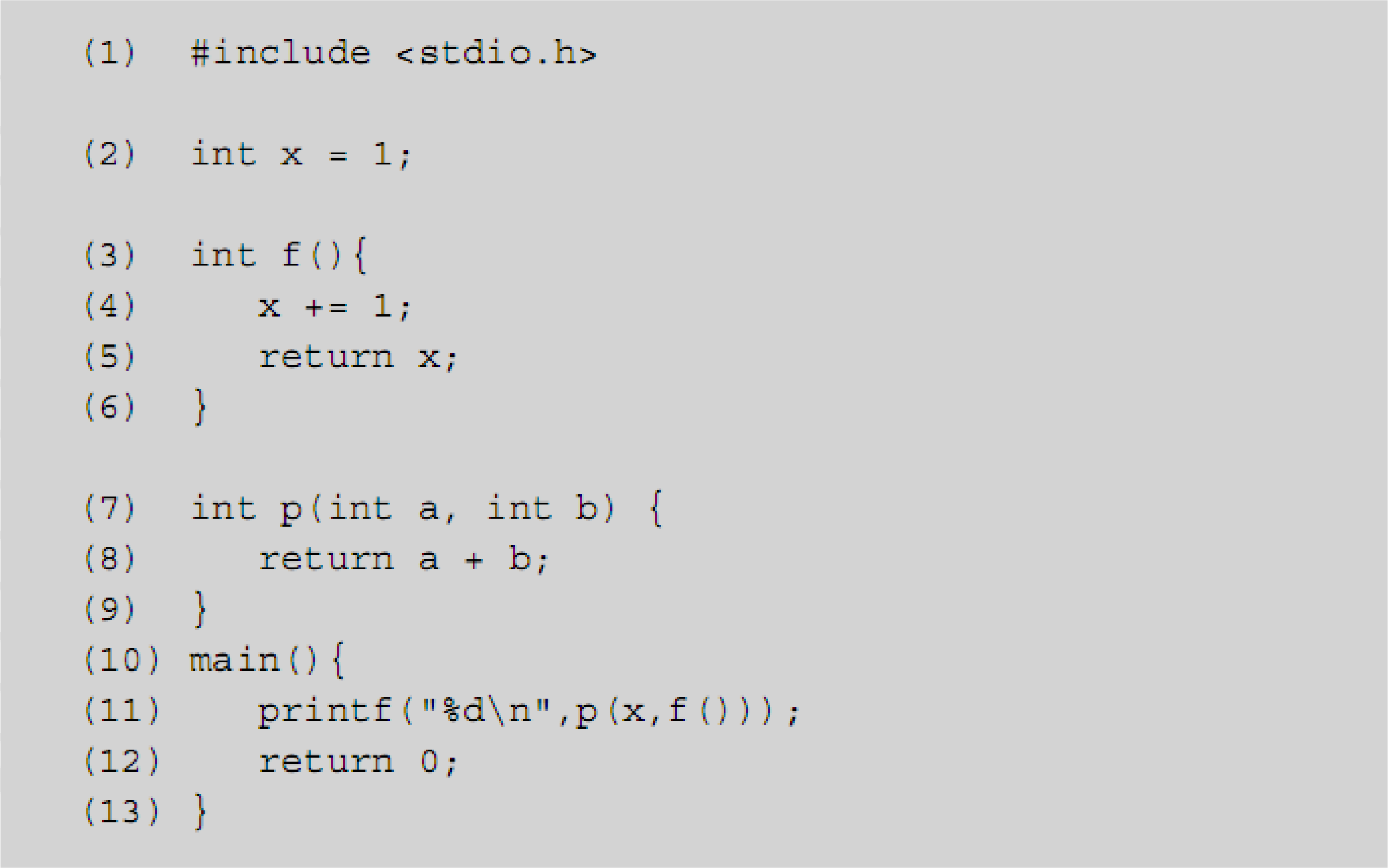


Figure 9.2 Syntax tree for the expression (3 + 4) * (5 – 6)

Expressions (cont'd.)

- Natural order to process (3+4) and (5-6) is left to right, but many languages do not specify an order
  - Machines may have different requirements for the structure of calls to procedures and functions
  - Translators may attempt to rearrange for efficiency
- If there are no side effects, order of evaluation of subexpressions will make no difference
  - If there are side effects, there may be differences

Expressions (cont'd.)



```
#include <stdio.h>

int x = 1;

int f(){
   x += 1;
   return x;
}

int p(int a, int b) {
   return a + b;
}
main(){
   printf("%d\n",p(x,f()));
   return 0;
}
```

Figure 9.3 C Program showing evaluation order matters in the presence of side effects

Expressions (cont'd.)

- Sometimes expressions are explicitly constructed with side effects in mind
- In C, assignment is an expression
  - Example: In C code:  `x = (y = z)`
    - y=z both assigns and returns a value, which is assigned to x
- **Sequence operator**: allows several expressions to be combined into a single expression and evaluated sequentially
  - Example: In C code:  `x = (y+=1,x+=y,x+1)`

Expressions (cont'd.)

- **Short-circuit evaluation**: Boolean expressions are evaluated left to right up to the point where the truth value of the entire expression becomes known, and then evaluation stops
  - Example: in Ada:  `x or true`
    - Is always true, regardless of the value of x
- Order of evaluation is important in short-circuit evaluation
- **If expressions** and **case expressions** also may not be completely evaluated

Expressions (cont'd.)

- If (or if-then-else) operator: is a **ternary operator** with three operands
- **Mix-form:** distributes parts of the syntax of the operator throughout the expression
  - Example: in ML code: 

```
if e1 then e2 else e3
```

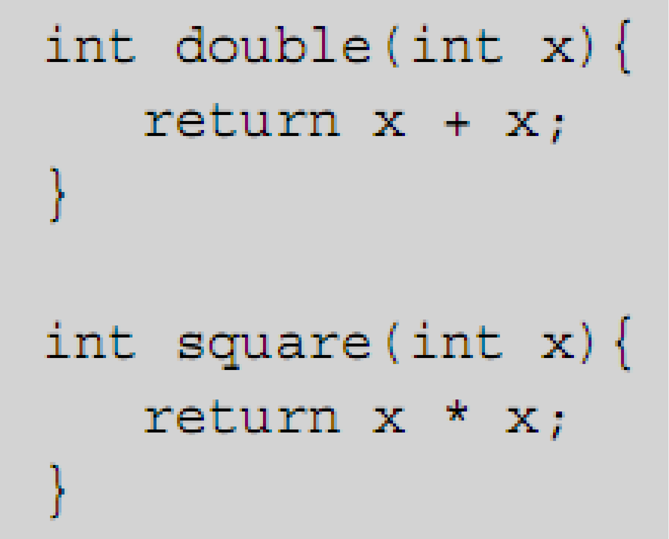
- If-expressions never have all of their subexpressions evaluated
- **Case expression**: similar to a series of nested-if expressions
- **Delayed evaluation** (or **nonstrict evaluation**): when operators delay evaluating their operands

Expressions (cont'd.)

- **Substitution rule** (or **referential transparency**): any two expressions that have the same value in the same scope may be substituted for each other
  - Their values always remain equal regardless of the evaluation context
  - Note that this prohibits variables in the expressions
- **Normal order evaluation**: each operation begins its evaluation before its operands are evaluated, and each operand is evaluated only if it is needed for the calculation of the operation

Expressions (cont'd.)

- Example: in C code:
  - Consider the expression `square(double(2))`
  - Square is replaced by `double(2)*double(2)`
  - Without evaluating `double(2)`
  - Then it is replaced by `2 + 2`
- Normal order evaluation implements a kind of code inlining

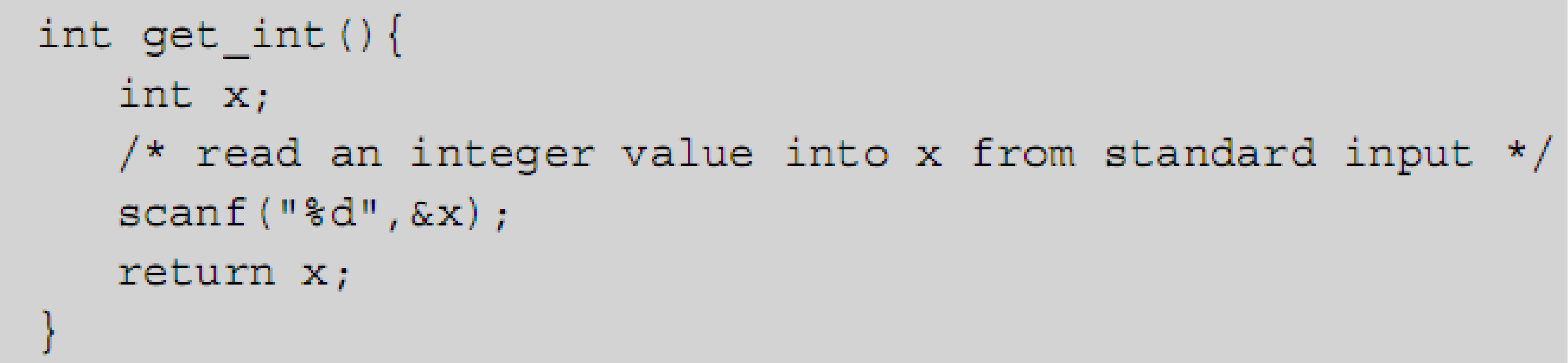


```
int double(int x){
    return x + x;
}

int square(int x){
    return x * x;
}
```

Expressions (cont'd.)

- With no side effects, normal order evaluation does not change the semantics of a program
- Example with side effects in C code:



```
int get_int(){
    int x;
    /* read an integer value into x from standard input */
    scanf("%d",&x);
    return x;
}
```

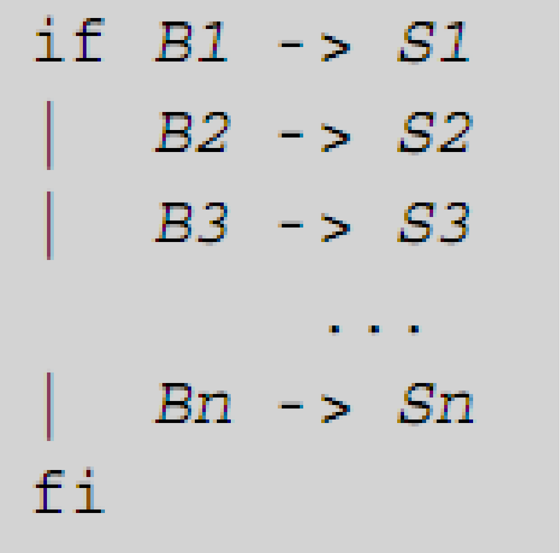
  - Expression `square(get_int())` would be expanded into `get_int()*get_int()`
    - Would read in two integer values instead of one

Expressions (cont'd.)

- Normal order evaluation:
  - Appears as **lazy evaluation** in the functional language Haskell
  - Appears as **pass by name** parameter passing technique for functions in Algol60

Conditional Statements and Guards

- **if-statement**: typical form of structured control
  - Execution of a group of statements occurs only under certain conditions
- **Guarded if** statement:
  - All `Bi`’s are Boolean expressions called **guards**
  - All `Si`’s are statement sequences
  - If one `Bi` evaluates to true, the corresponding `Si` is executed
  - If more than one `Bi` is true, only one `Si` is executed



```
if B1 -> S1
|  B2 -> S2
|  B3 -> S3
       ...
|  Bn -> Sn
fi
```

Conditional Statements and Guards (cont'd.)

- It does not say that the first true `Bi` is chosen
  - This introduces nondeterminism into programming
- It leaves unspecified whether all guards are evaluated
  - A useful feature for concurrent programming
- Usual implementation is to sequentially evaluate all `Bi`'s until a true one is found, then execute the corresponding `Si`
- If-statements and case-statements are the major ways that the guarded if are implemented

If-Statements

- Basic form of the if-statement in EBNF in C code:

if-statement → `if` (*expression*) *statement* [`else` *statement*]

  - A statement can be either a single statement or a sequence of statements surrounded by braces
- This if statement is problematic, as there are two different parse trees possible:

```
if (e1) if (e2) S1 else S2
```

  - Called the **dangling-else** problem

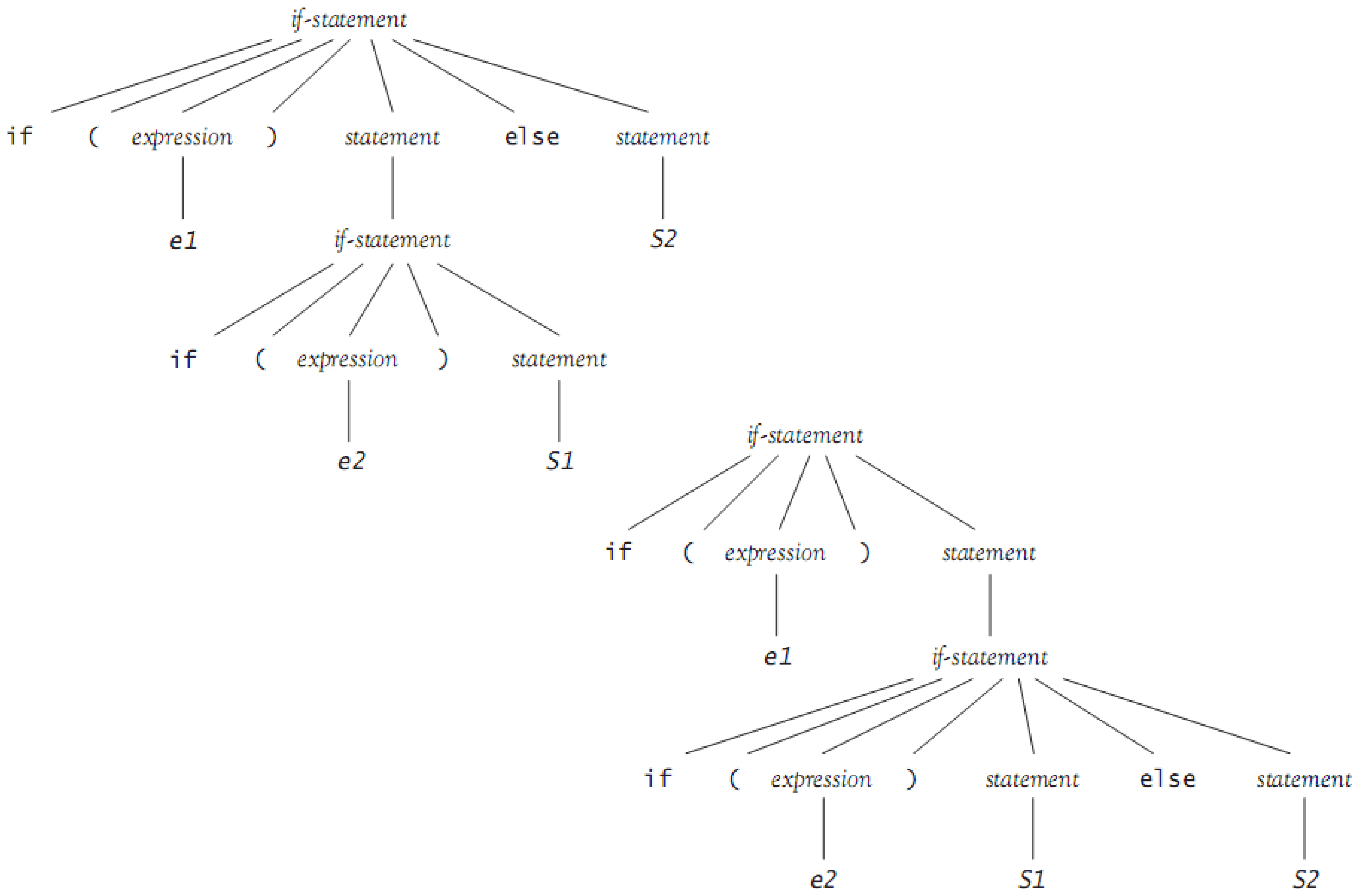
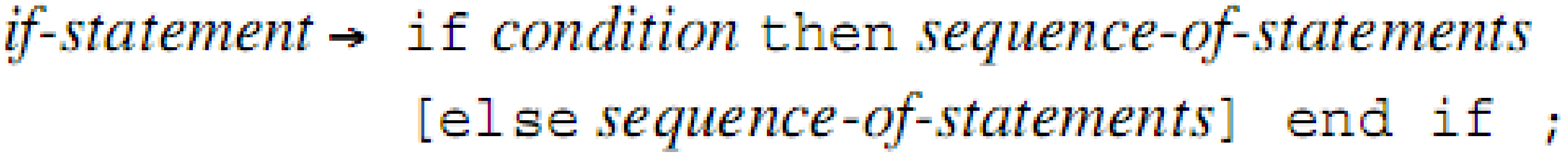


Figure 9.4 Two parse trees for the statement `if` (e1) `if` (e2) S1 `else` S2

If-Statements (cont'd.)

- C and Pascal enforce a **disambiguating rule**:
  - `else` is associated with the closest `if` that does not already have an `else` part
  - Called the **most closely nested** rule for if-statements
- A better way to solve the dangling-else problem is to use a bracketing keyword, as in the Ada rule:

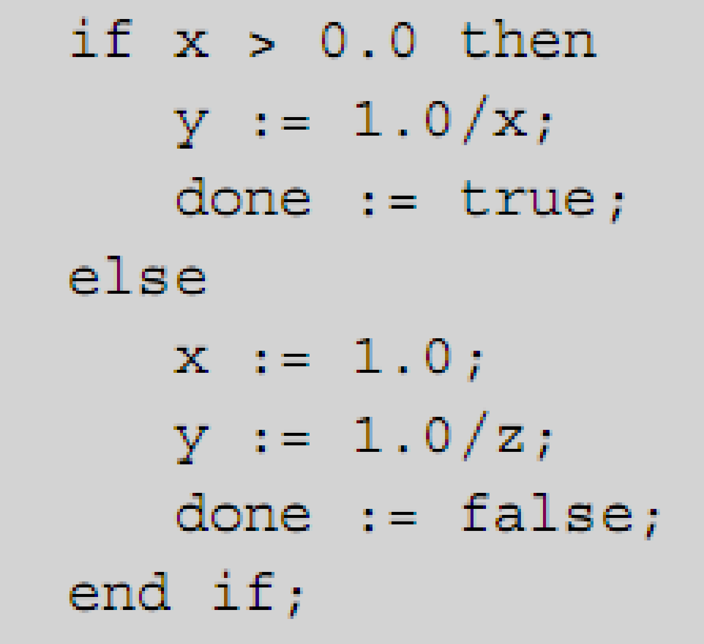


if-statement → `if` *condition* `then` *sequence-of-statements*
[`else` *sequence-of-statements*] `end if ;`

ambiguity

If-Statements (cont'd.)

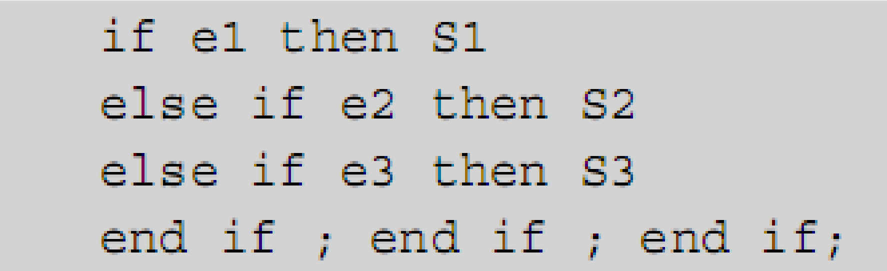
- This also removes the necessity of using brackets to open a new sequence of statements:



```
if x > 0.0 then
   y := 1.0/x;
   done := true;
else
   x := 1.0;
   y := 1.0/z;
   done := false;
end if;
```

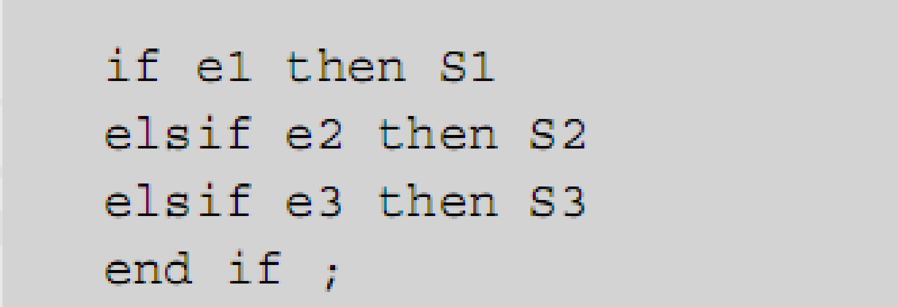
If-Statements (cont’d.)

- `elsif` in Ada eliminates multiple `end ifs` when there are many alternatives:



```
if e1 then S1
else if e2 then S2
else if e3 then S3
end if ; end if ; end if;
```

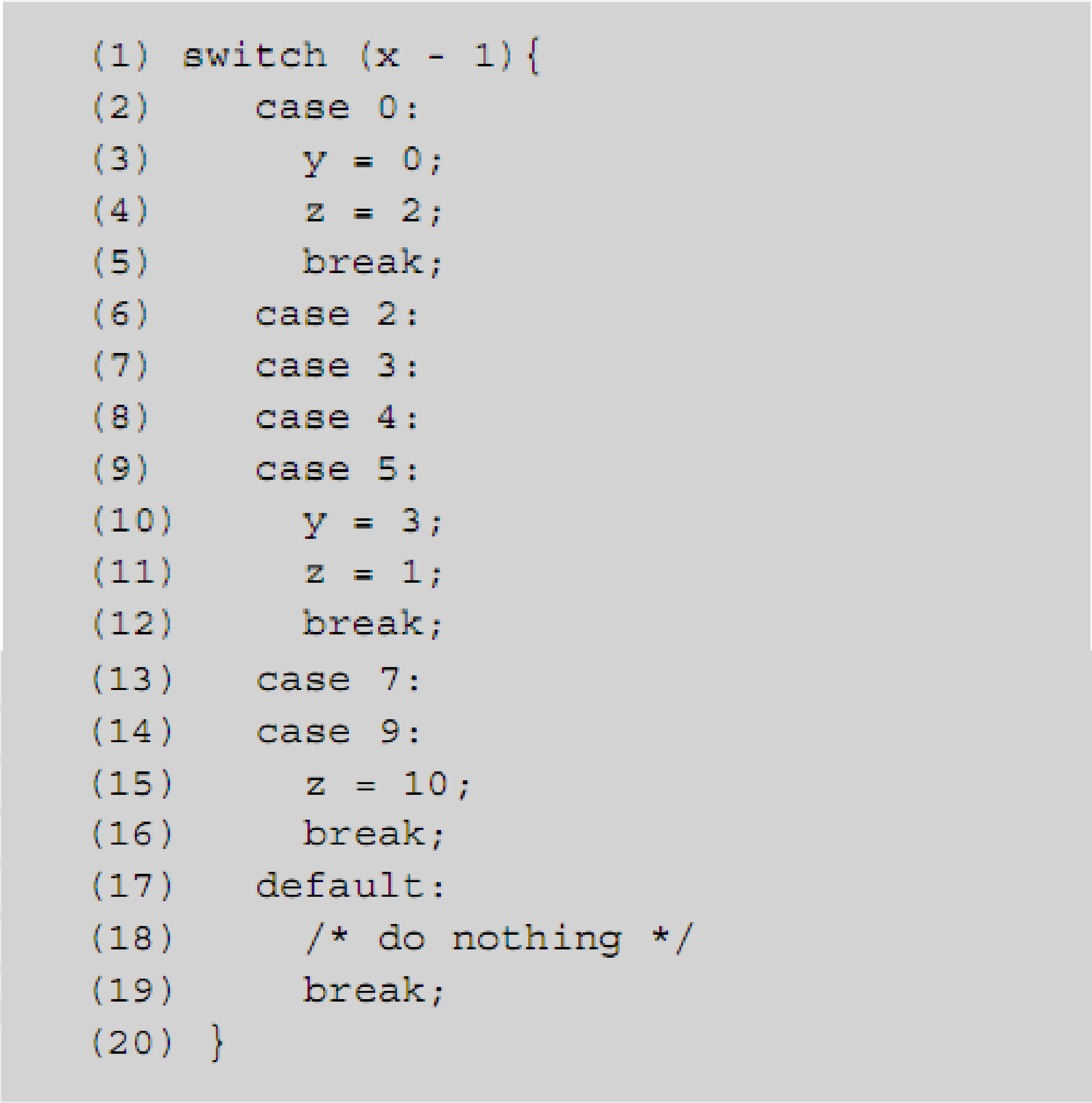
  - Becomes:



```
if e1 then S1
elsif e2 then S2
elsif e3 then S3
end if ;
```

Case- and Switch-Statements

- **Case-** or **switch-statement**: a guarded if where the guards are ordinal values selected by an ordinal expression
- Semantics in C:
  - Evaluate the controlling expression
  - Transfer control to the `case` statement where the value is listed
  - No two listed cases may have the same value
  - Case values may be literals or compile-time constant expressions
  - If no value matches, transfer to the `default` case



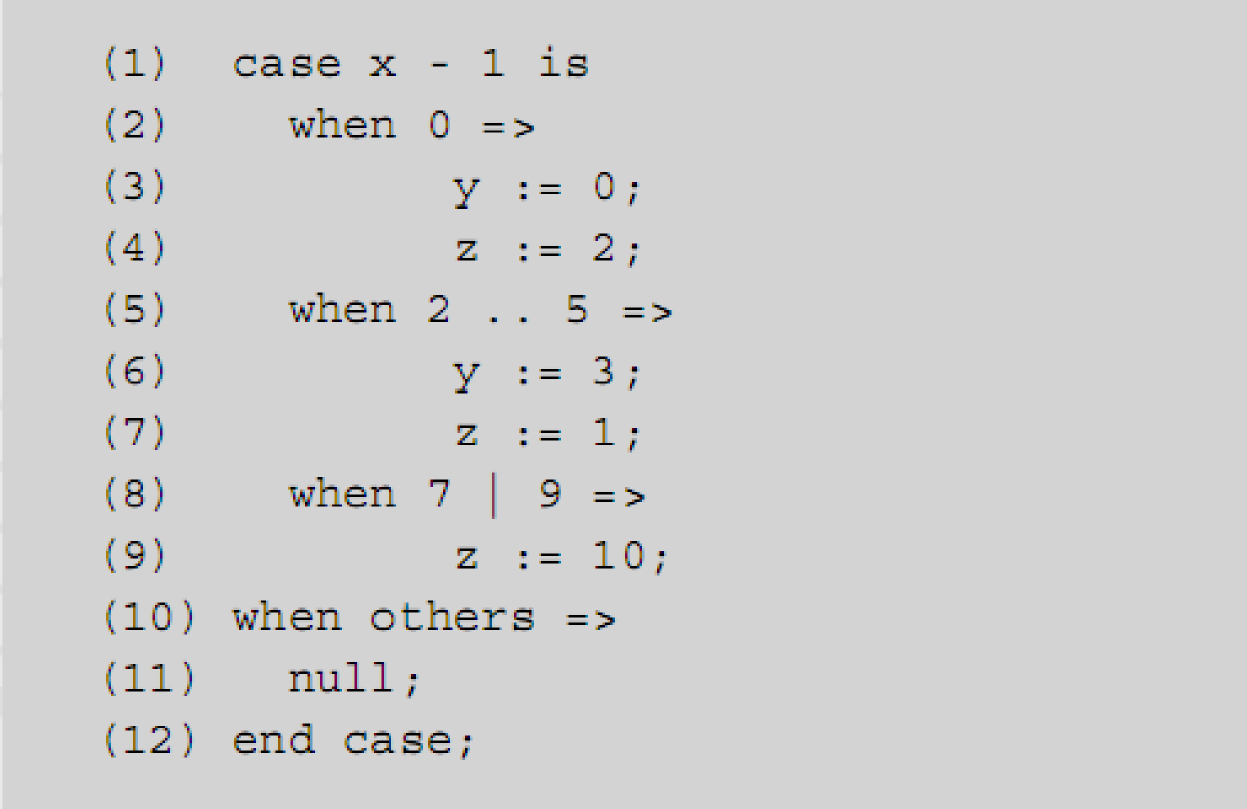
```
switch (x - 1){
   case 0:
     y = 0;
     z = 2;
     break;
   case 2:
   case 3:
   case 4:
   case 5:
     y = 3;
     z = 1;
     break;
   case 7:
   case 9:
     z = 10;
     break;
   default:
     /* do nothing */
     break;
}
```

Figure 9.5 An example of the use of the switch statement in C

Case- and Switch-Statements (cont'd.)

- If there is no `default` case, control falls through to the next statement after the `switch`
- Some novel features:
  - Case labels are treated syntactically as ordinary labels
  - Without a break statement, execution falls through to the next case
- Ada allows case values to be grouped and requires that they be exhaustive
  - Compile-time error if a legal value is not listed

Case- and Switch-Statements (cont’d.)

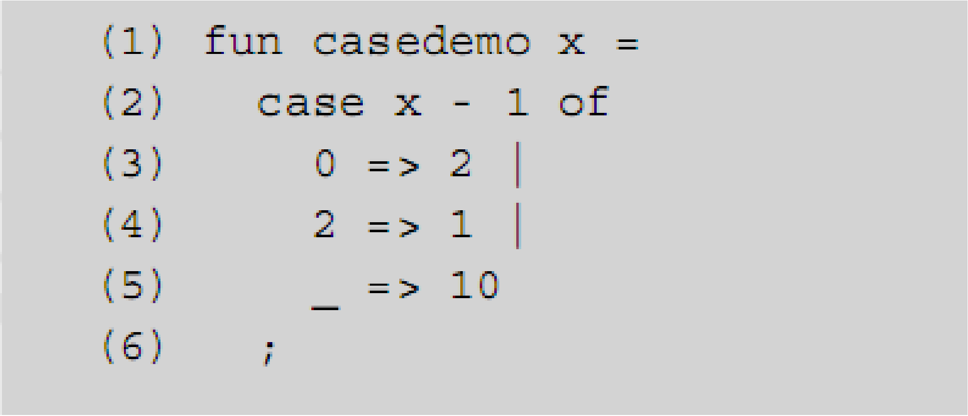


```
case x - 1 is
  when 0 =>
        y := 0;
        z := 2;
  when 2 .. 5 =>
        y := 3;
        z := 1;
  when 7 | 9 =>
        z := 10;
when others =>
  null;
end case;
```

Figure 9.6 An example of the use of the case statement in Ada, corresponding to the C example of Figure 9.5

Case- and Switch-Statements (cont'd.)

- ML's case construct is an expression that returns a value, rather than a statement
  - Cases are separated by vertical bars
  - Case expressions are patterns to be matched
  - **Wildcard pattern** is the underscore

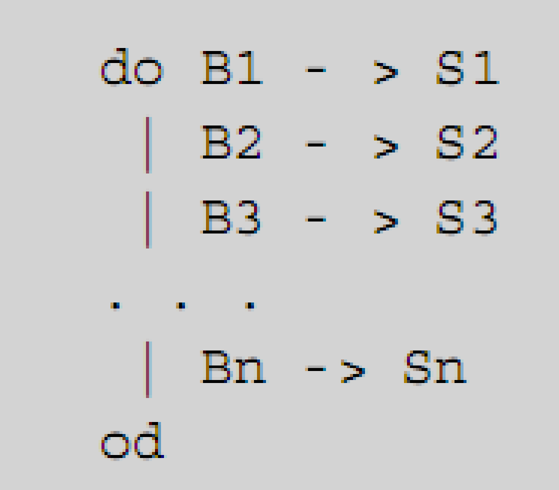


```
fun casedemo x =
  case x - 1 of
    0 => 2 |
    2 => 1 |
    _ => 10
  ;
```

Figure 9.7 An example of a case expression in ML

Loops and Variations on WHILE

- **Guarded do**: a general form for a loop construct
  - Statement is repeated until all `Bi`’s are false
  - At each step, one of the true `Bi`’s is selected nondeterministically, and the corresponding `Si` is executed



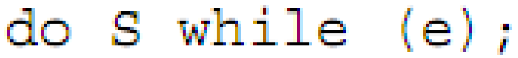
```
do B1 - > S1
 | B2 - > S2
 | B3 - > S3
. . .
 | Bn -> Sn
od
```

Loops and Variations on WHILE (cont'd.)

- Basic loop construct: a guarded do with only one guard
  - Eliminates nondeterminism
- In C: `while (e) S`
- In Ada: `while e loop S1 ... Sn end loop;`
- The test expression (e) is evaluated first
  - Must be Boolean in Ada and Java, but not C or C++
  - If true (or non-zero), then S is executed and the process repeats

Loops and Variations on WHILE (cont'd.)

- Some languages have an alternative form that ensures the loop is executed at least once
  - In C and Java: the `do` (or `do-while`) statement



```
do S while (e);
```

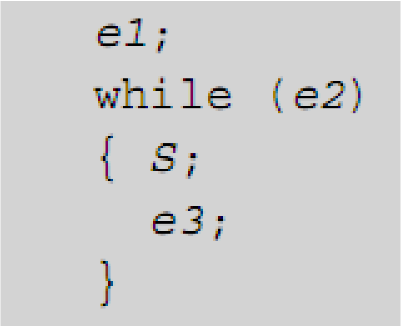
- Termination of the `do` or `while` loop is explicitly specified only at the beginning or end of the loop
- C and Java provide a `break` statement to exit completely from inside a loop
  - `continue` statement skips the remainder of the body of the loop but resumes with the next iteration

Loops and Variations on WHILE (cont'd.)

- **For-loop** in C/C++ and Java:

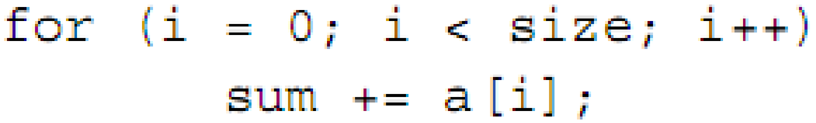
```
for ( e1; e2; e3 ) S;
```

  - Is completely equivalent in C to:
    - e1 is the **initializer**
    - e2 is the **test**
    - e3 is the **update**



```
e1;
while (e2)
{ S;
  e3;
}
```

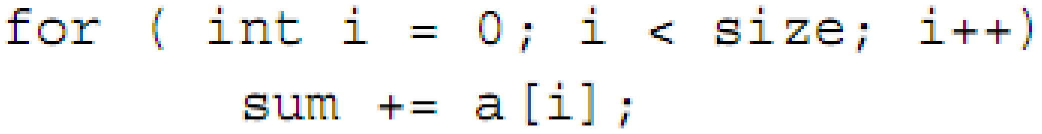
- For-loop is typically used to run through a set of values from first to last



```
for (i = 0; i < size; i++)
        sum += a[i];
```

Loops and Variations on WHILE (cont'd.)

- C++ and Java allow a for-loop initializer (**index**) to be declared in the loop:



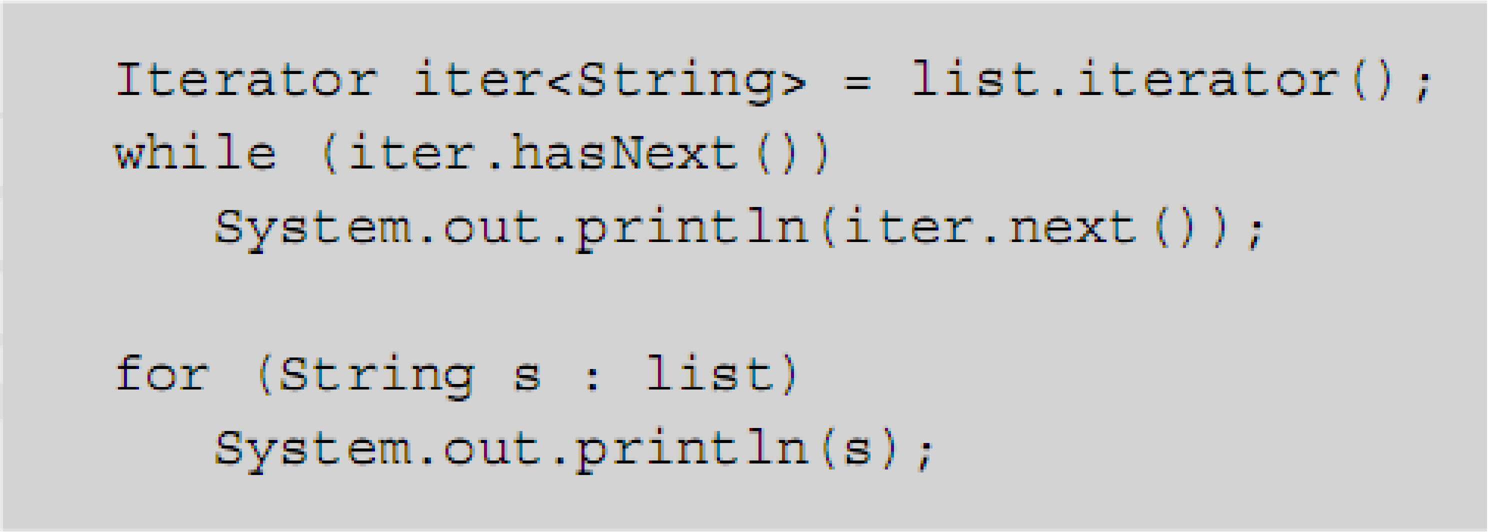
```
for ( int i = 0; i < size; i++)
        sum += a[i];
```

- Many languages restrict the format of the for-loop
- Most restrictions involve the control variable `i`:
  - Value of `i` cannot be changed in the body of the loop
  - Value of `i` is undefined after loop termination
  - `i` must be of restricted type and may not be a parameter to a procedure or record field

Loops and Variations on WHILE (cont'd.)

- Other questions about loop behavior include:
  - Are bounds evaluated only once? If so, bounds may not change after execution begins
  - Is the loop executed at all if the lower bound is greater than the upper bound?
  - Is the control variable value undefined if an `exit` or `break` statement is used?
  - What translator checks are performed on loop structures?
- Object-oriented languages use an iterator object for looping over elements of a collection

Loops and Variations on WHILE (cont'd.)



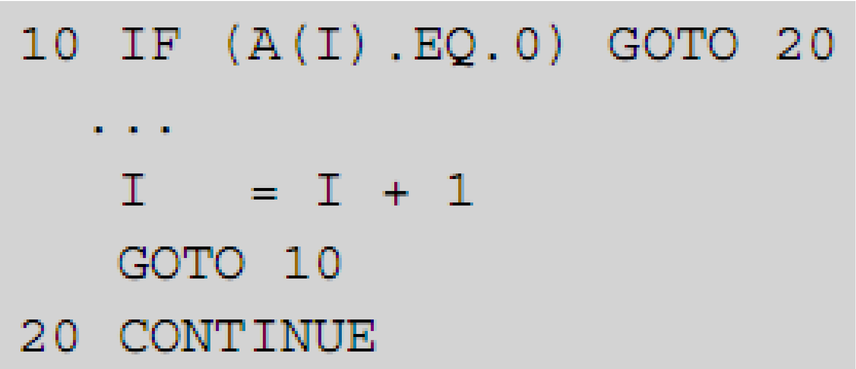
```
Iterator iter<String> = list.iterator();
while (iter.hasNext())
   System.out.println(iter.next());

for (String s : list)
   System.out.println(s);
```

Figure 9.8 Two ways to use a Java iterator to process a list

The GOTO Controversy and Loop Exits

- `Gotos` were used heavily in early programming languages such as Fortran77 and BASIC
- Example in Fortran77:



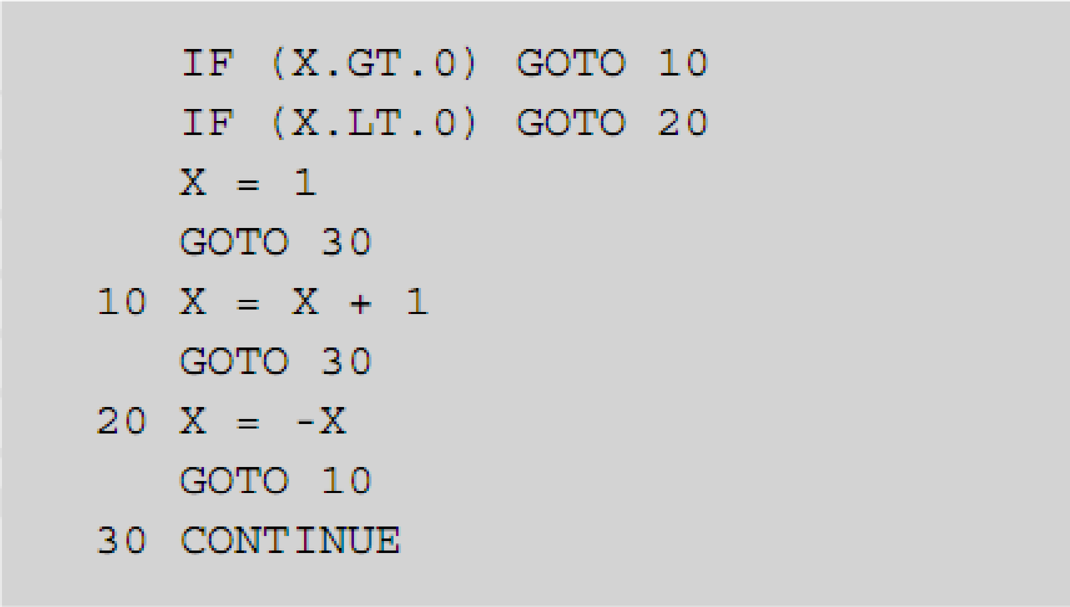
```
10 IF (A(I).EQ.0) GOTO 20
  ...
   I   = I + 1
   GOTO 10
20 CONTINUE
```

  - Is equivalent to this C code:

```
while (a[i] != 0)  i++;
```

The GOTO Controversy and Loop Exits (cont'd.)

- In the 1960s with structured control use increasing, debate began about the usefulness of `gotos`
  - Can lead to **spaghetti code**



```
   IF (X.GT.0) GOTO 10
   IF (X.LT.0) GOTO 20
   X = 1
   GOTO 30
10 X = X + 1
   GOTO 30
20 X = -X
   GOTO 10
30 CONTINUE
```

Figure 9.9 An example of spaghetti code in Fortran77

The GOTO Controversy and Loop Exits (cont'd.)

- In 1966, Bohm and Jacopini produced theoretical result that `goto`s were completely unnecessary
- In 1968, Dijkstra published "GOTO Statement Considered Harmful"
  - Proposed that its use be severely controlled or abolished
- Many considered `goto`s to be justified in certain cases
- In 1987, Rubin published ""Goto considered harmful" considered harmful"

The GOTO Controversy and Loop Exits (cont'd.)

- Still some debate on the propriety of unstructured exits from loops
  - Some argue there should only be one exit in a loop
  - Others argue that may require more complicated code for certain situations
- Example: searching an array for a given element
  - Method returns the index of the target element if it is in the array, or -1 otherwise
- Example: sentinel-based loop for processing a series of input values
  - Called the **loop and a half problem**

The GOTO Controversy and Loop Exits (cont'd.)

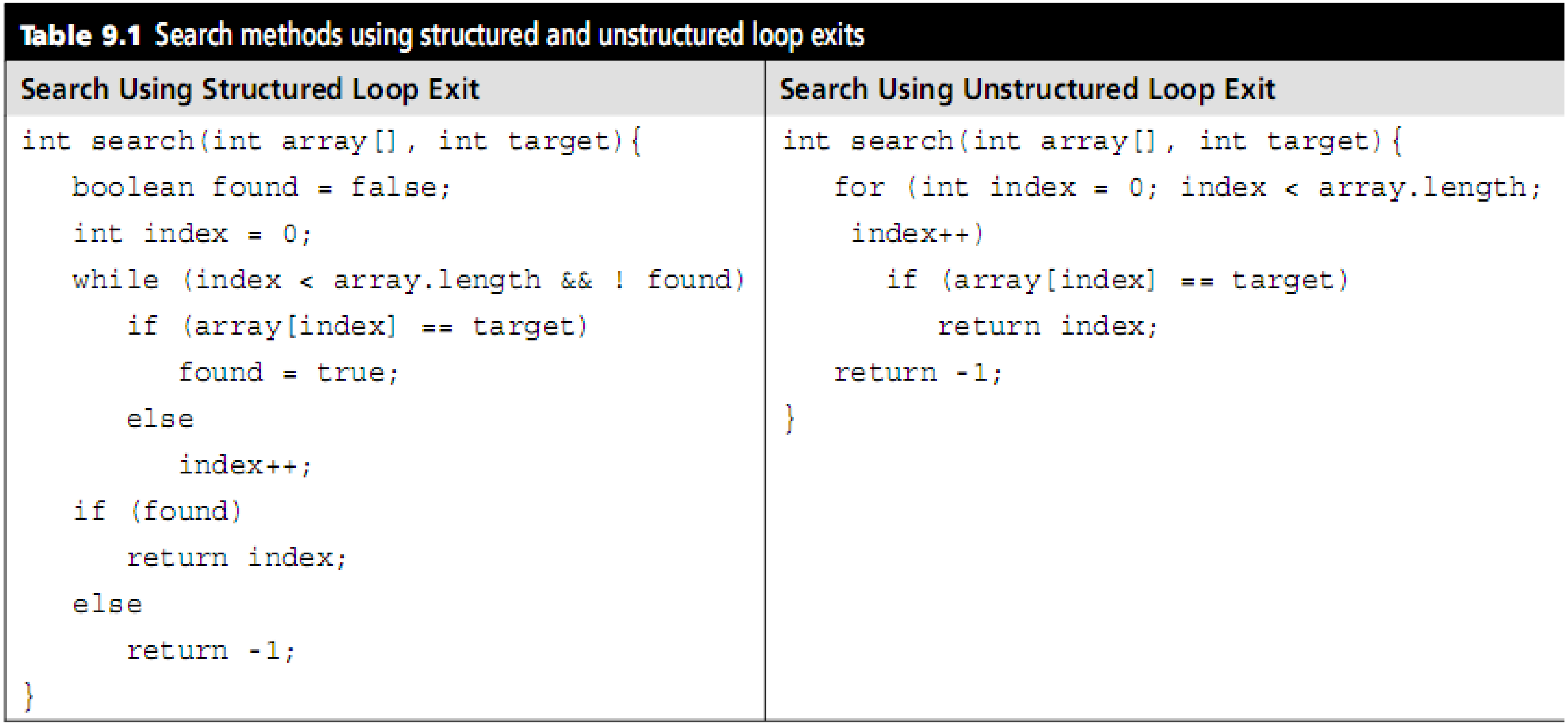


Table 9.1 Search methods using structured and unstructured loop exits

Search Using Structured Loop Exit

```
int search(int array[], int target){
   boolean found = false;
   int index = 0;
   while (index < array.length && ! found)
      if (array[index] == target)
         found = true;
      else
         index++;
   if (found)
      return index;
   else
      return -1;
}
```

Search Using Unstructured Loop Exit

```
int search(int array[], int target){
   for (int index = 0; index < array.length;
    index++)
      if (array[index] == target)
         return index;
   return -1;
}
```

The GOTO Controversy and Loop Exits (cont'd.)

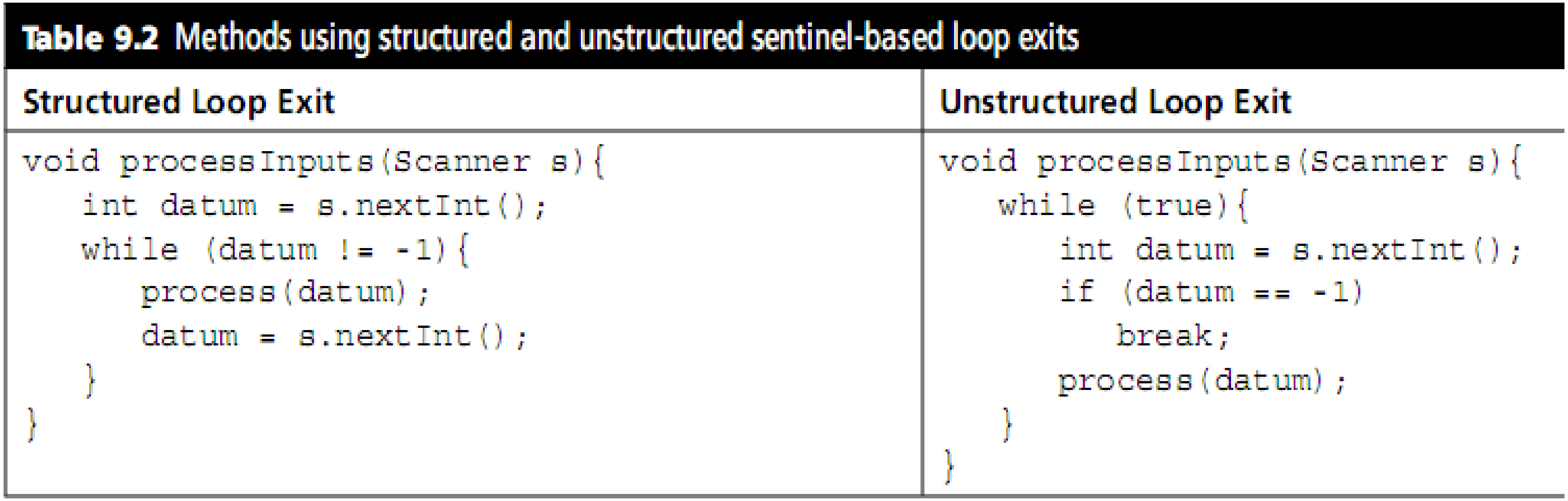


Table 9.2 Methods using structured and unstructured sentinel-based loop exits

Structured Loop Exit

```
void processInputs(Scanner s){
   int datum = s.nextInt();
   while (datum != -1){
      process(datum);
      datum = s.nextInt();
   }
}
```

Unstructured Loop Exit

```
void processInputs(Scanner s){
   while (true){
      int datum = s.nextInt();
      if (datum == -1)
         break;
      process(datum);
   }
}
```

Exception Handling

- **Explicit control mechanisms**: at the point where transfer of control takes place, there is a syntactic indication of the transfer
- **Implicit transfer of control**: the transfer is set up at a different point than where the actual transfer takes place
- **Exception handling**: control of error conditions or other unusual events during execution
  - Involves the declaration of both exceptions and exception handlers

Exception Handling (cont'd.)

- When an exception occurs, it is said to be **raised** or **thrown**
- Examples of exceptions:
  - Runtime exceptions: out-of-range array subscripts or division by zero
  - Interpreted code: syntax or type errors
- **Exception handler**: procedure or code sequence designed to be executed when a particular exception is raised
- An exception handler is said to **handle** or **catch** an exception

Exception Handling (cont'd.)

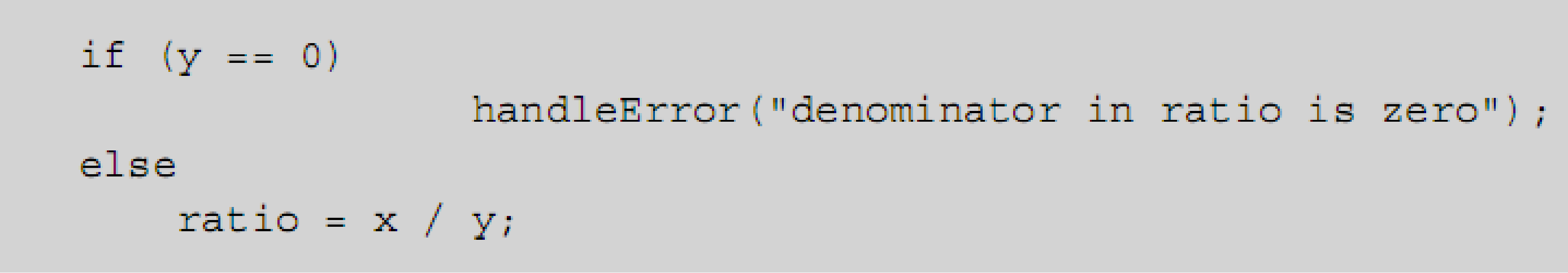
- Virtually all major current languages have built-in exception-handling mechanisms
  - Those languages without this sometimes have libraries available that provide it
- Exception handling attempts to imitate the features of a hardware interrupt or error trap
  - If the underlying machine or operating system is left to handle the error, the program will usually abort or crash
- Programs that crash fail the test of **robustness**

Exception Handling (cont'd.)

- Cannot expect a program to be able to handle every possible error that can occur
  - Too many possible failures, including hardware
- **Asynchronous exceptions**: when the underlying operating system detects a problem and needs to terminate a program
  - Not in response to program code being executed
- **Synchronous exceptions**: exceptions that occur in direct response to actions by the program

Exception Handling (cont'd.)

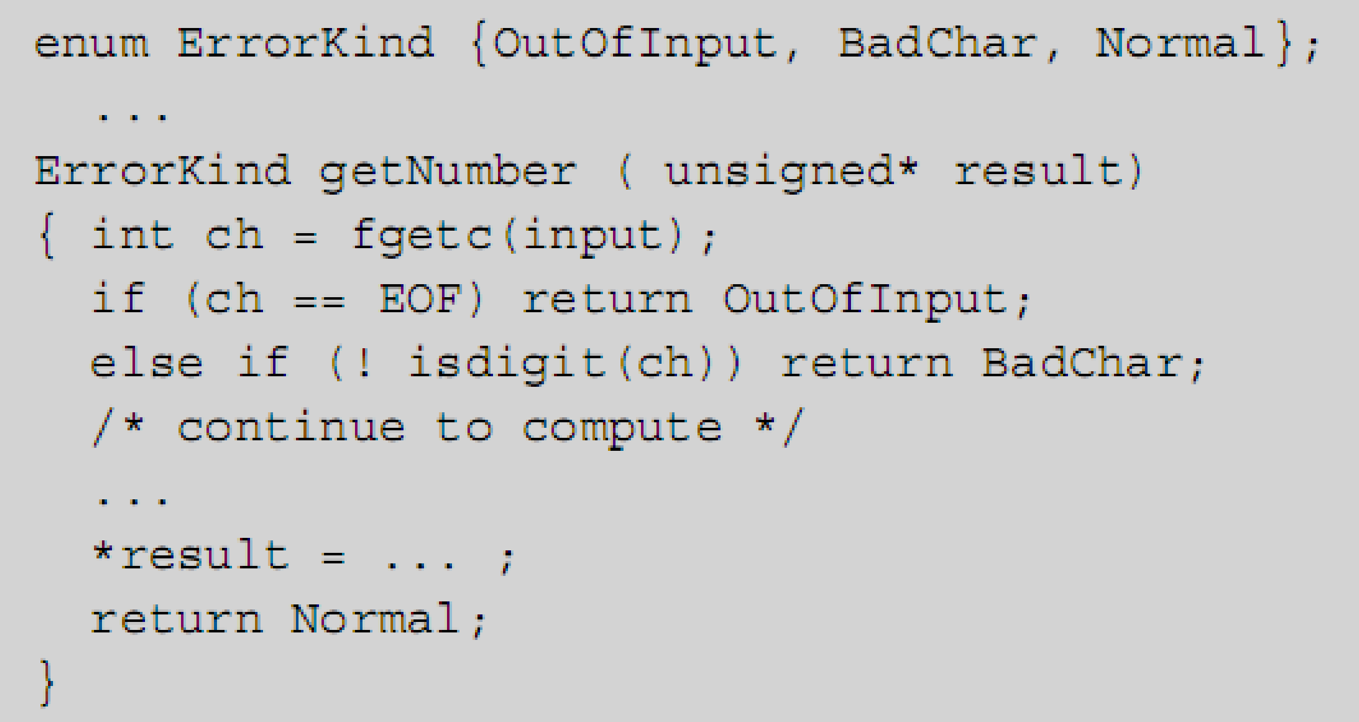
- User-defined exceptions can only be synchronous
- Predefined or library exceptions may include some asynchronous exceptions
- Exception handling assumes that it is possible to test for exceptions in the language
- Can handle the error at the location where it occurs:



```
if (y == 0)
                handleError("denominator in ratio is zero");
else
    ratio = x / y;
```

Exception Handling (cont'd.)

- Can pass an error condition back to a caller of a procedure



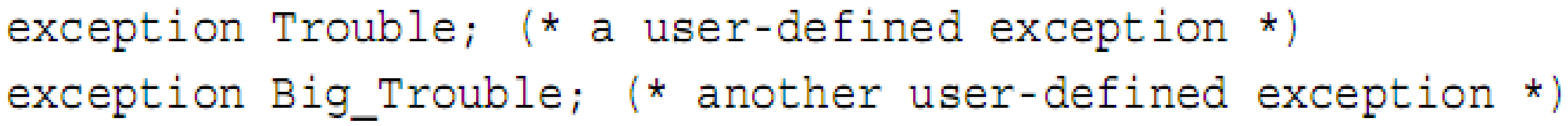
```
enum ErrorKind {OutOfInput, BadChar, Normal};
  ...
ErrorKind getNumber ( unsigned* result)
{ int ch = fgetc(input);
  if (ch == EOF) return OutOfInput;
  else if (! isdigit(ch)) return BadChar;
  /* continue to compute */
  ...
  *result = ... ;
  return Normal;
}
```

Exception Handling (cont'd.)

- Can also create a separate exception-handling procedure to call
- To make error handling easier, we would like to declare exceptions in advance of their occurrence and specify the actions to be taken
- To do so, must consider issues related to:
  - Exceptions
  - Exception handlers
  - Control

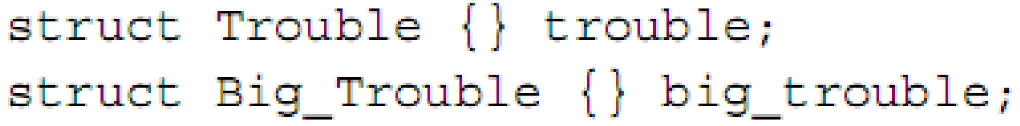
Exceptions

- Exception is typically represented by a data object, either predefined or user-defined
  - In a functional language, it will be a value
  - In a structured or object-oriented language, it will be a variable or an object of some structured type
- Example: in ML or Ada:



```
exception Trouble; (* a user-defined exception *)
exception Big_Trouble; (* another user-defined exception *)
```

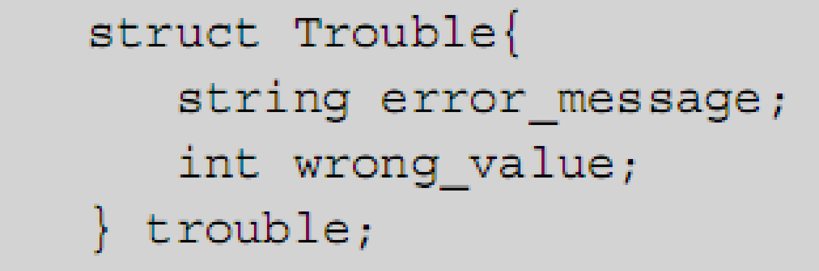
- Example: in C++:



```
struct Trouble {} trouble;
struct Big_Trouble {} big_trouble;
```

Exceptions (cont'd.)

- Typically want to include additional information with an exception, such as error message or summary of data involved



```
struct Trouble{
    string error_message;
    int wrong_value;
} trouble;
```

- Exception declarations typically observe the same scope rules as other declarations
  - May be desirable to declare user-defined exceptions globally to ensure they are reachable

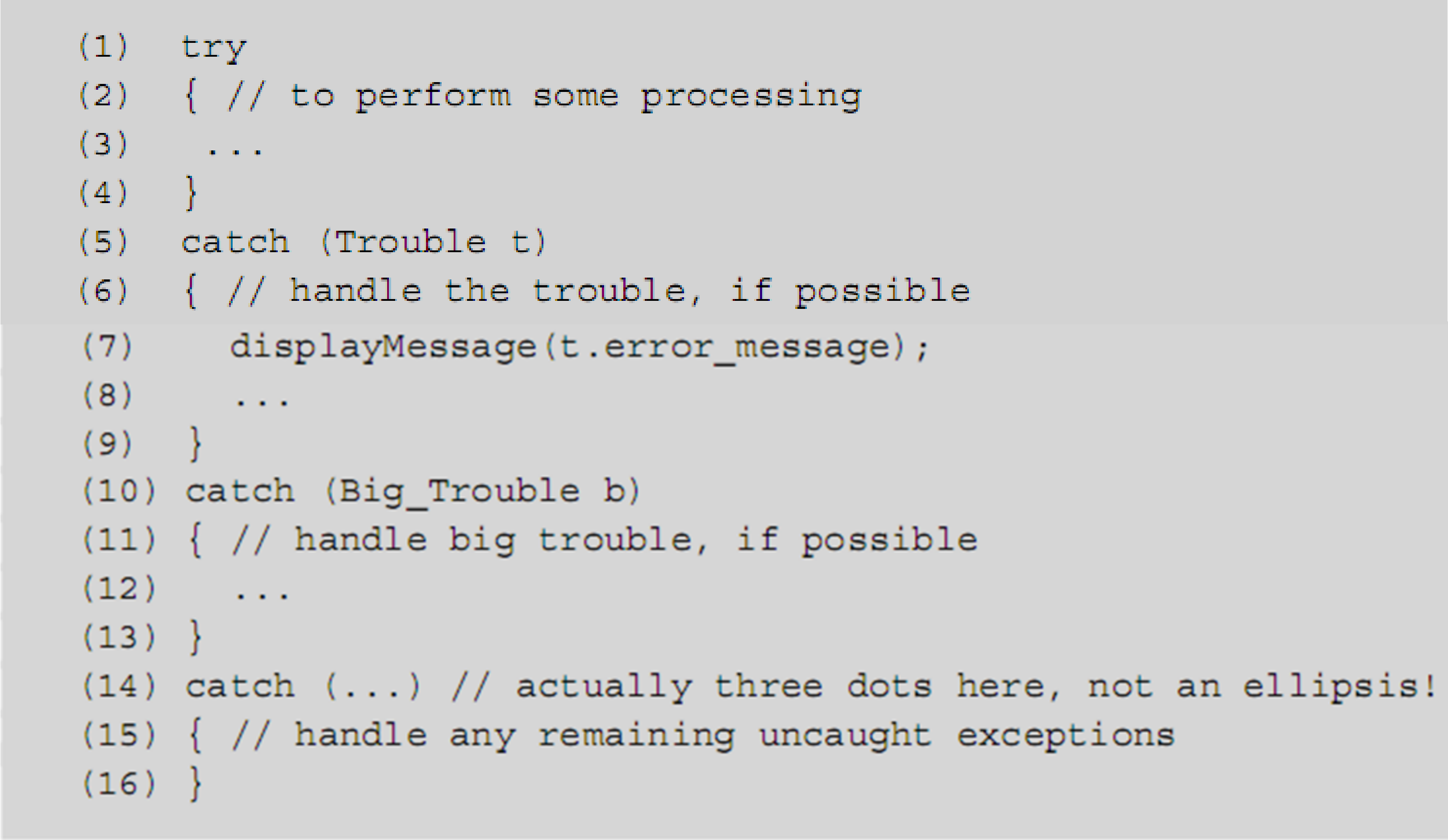
Exceptions (cont’d.)

- Most languages provide some predefined exception values or types, either directly or in standard library modules

Exception Handlers

- In C++, exception handlers are associated with **try-catch** blocks
  - Any number of `catch` blocks can be included
  - Each `catch` block takes the exception type as a parameter and includes a compound statement of actions to be taken
  - Last `catch` block with parameter of … is to catch any exceptions not handled in the prior `catch` blocks

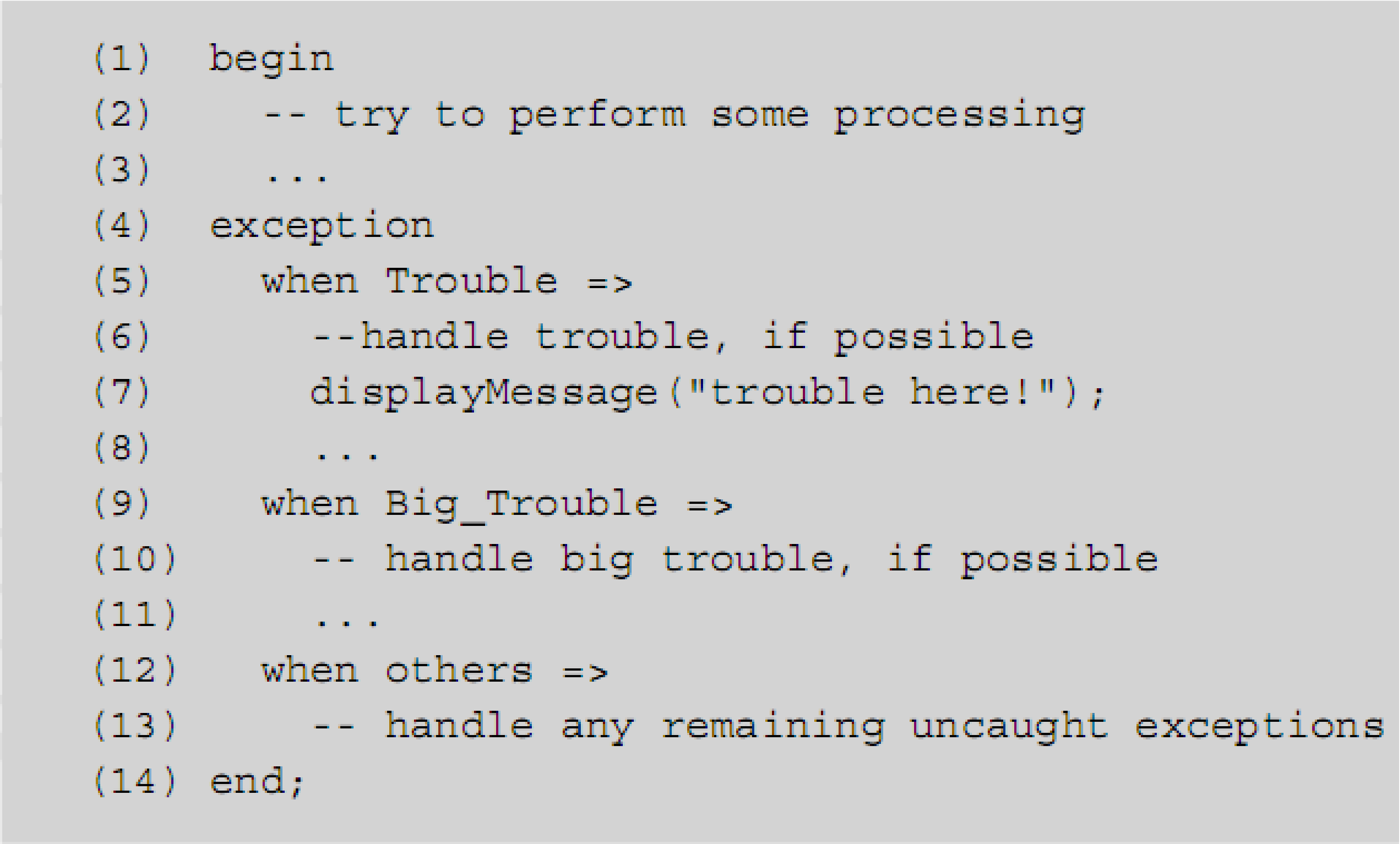
Exception Handlers (cont'd.)



```
try
{ // to perform some processing
   ...
}
catch (Trouble t)
{ // handle the trouble, if possible
    displayMessage(t.error_message);
    ...
}
catch (Big_Trouble b)
{ // handle big trouble, if possible
    ...
}
catch (...) // actually three dots here, not an ellipsis!
{ // handle any remaining uncaught exceptions
}
```

Figure 9.10 A C++ try-catch block

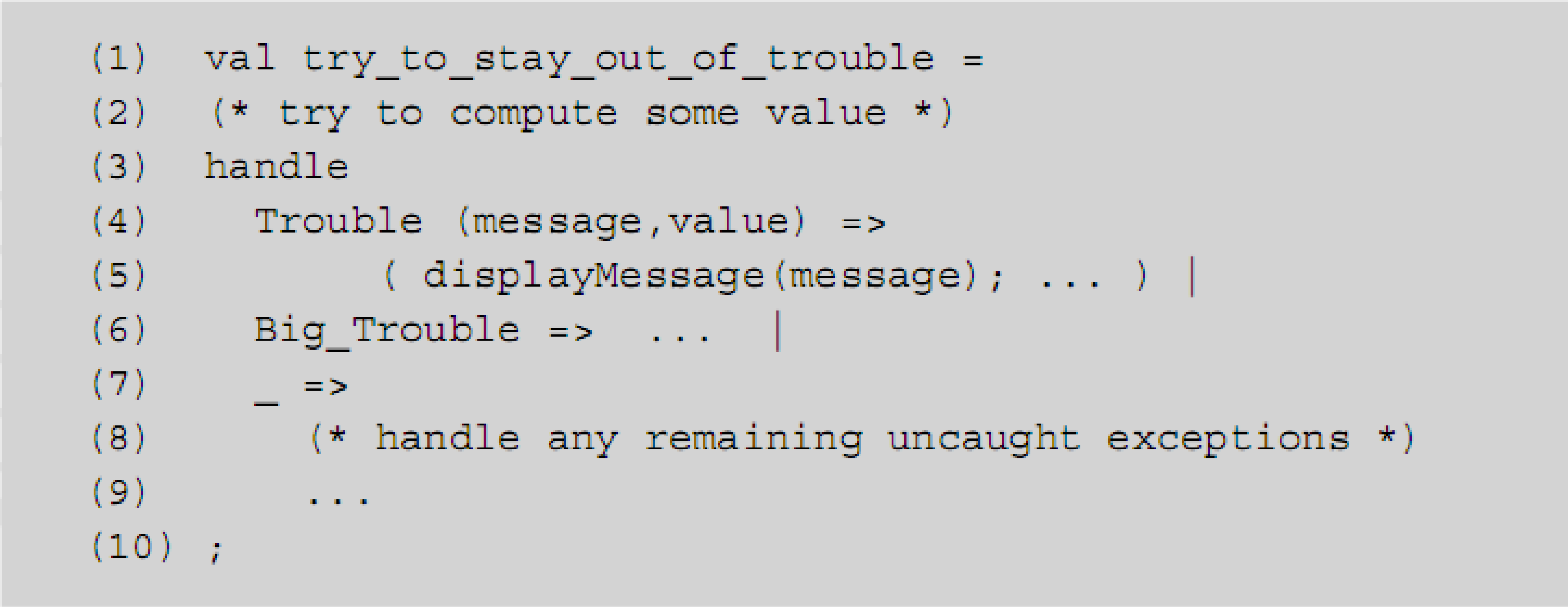
Exception Handlers (cont’d.)



```
begin
  -- try to perform some processing
  ...
exception
  when Trouble =>
    --handle trouble, if possible
    displayMessage("trouble here!");
    ...
  when Big_Trouble =>
    -- handle big trouble, if possible
    ...
  when others =>
    -- handle any remaining uncaught exceptions
end;
```

Figure 9.11 An Ada try-catch block corresponding to Figure 9.9

Exception Handlers (cont'd.)



```
val try_to_stay_out_of_trouble =
(* try to compute some value *)
handle
  Trouble (message,value) =>
       ( displayMessage(message); ... ) |
  Big_Trouble =>  ...   |
  _ =>
    (* handle any remaining uncaught exceptions *)
    ...
;
```

Figure 9.12 An example of ML exception handling corresponding to Figures 9.10 and 9.11

Exception Handlers (cont'd.)

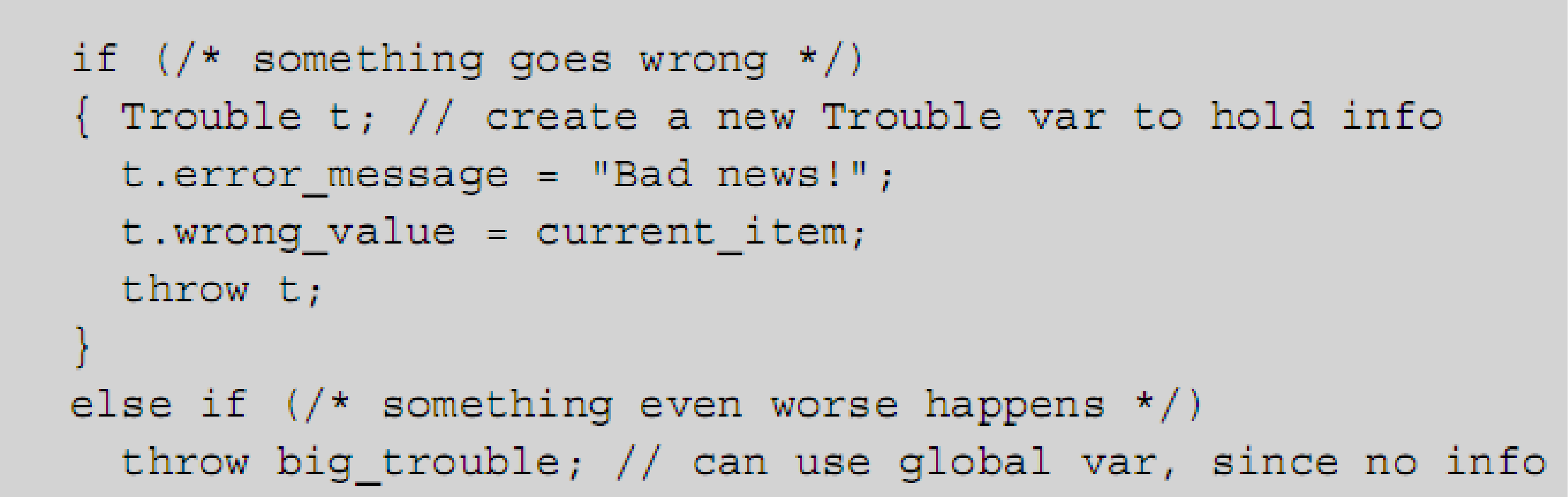
- Predefined handlers typically print a minimal error message, indicating type of exception and possibly some additional information, then terminate the program
- In Ada and ML, there is no way to change the behavior of default handlers
  - Can disable it in Ada
- In C++, can replace the default handler with a user-defined handler using the `<exceptions>` standard library module

Control

- Predefined or built-in exceptions are either automatically raised by the runtime system or can be manually raised by the program
- User-defined exceptions can only be raised by the program
- In C++, an exception can be raised with the `throw` reserved word
- Ada and ML both use the reserved word `raise`

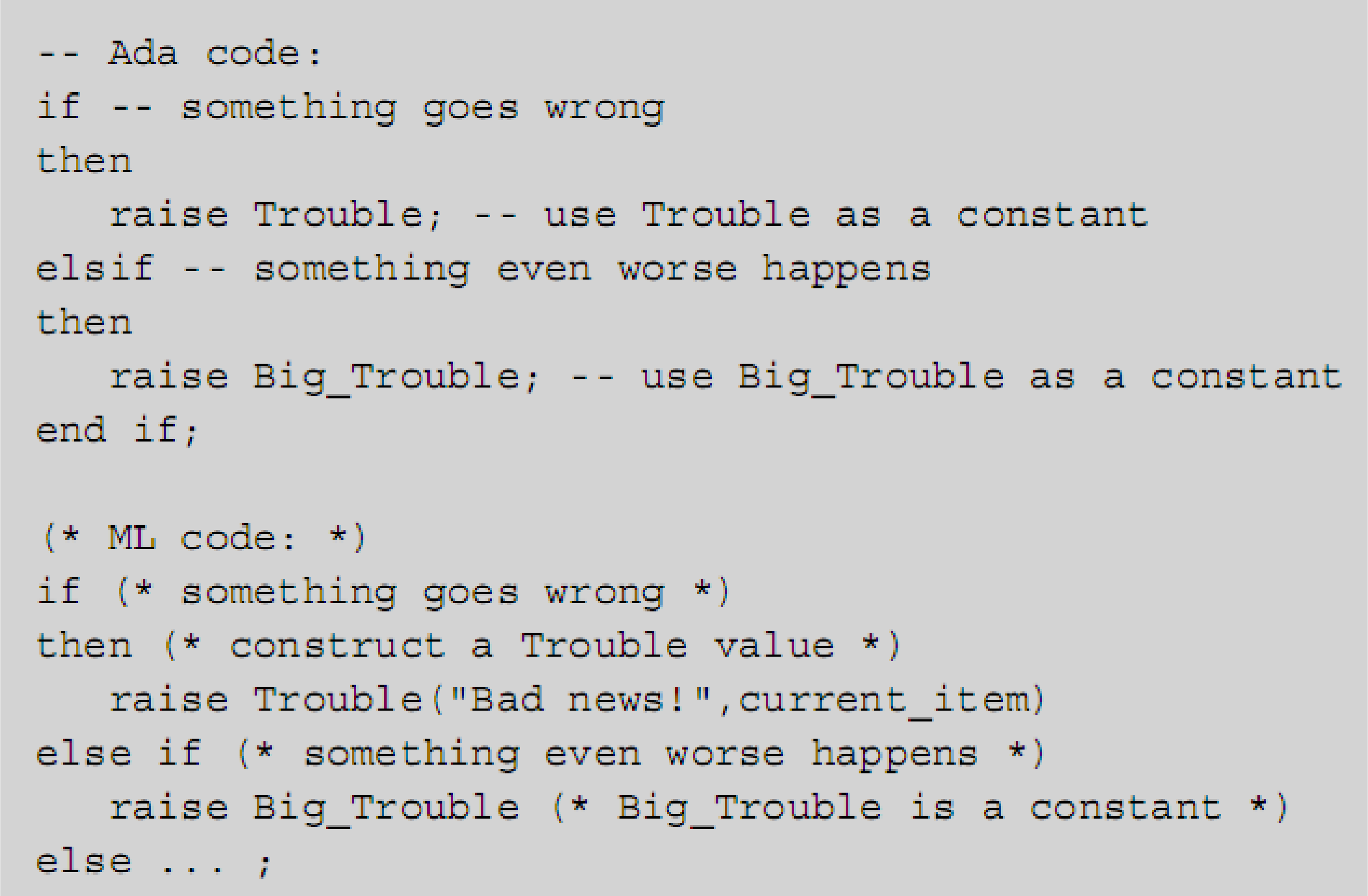
Control (cont’d.)

- Example: In C++ code



```
if (/* something goes wrong */)
{ Trouble t; // create a new Trouble var to hold info
  t.error_message = "Bad news!";
  t.wrong_value = current_item;
  throw t;
}
else if (/* something even worse happens */)
  throw big_trouble; // can use global var, since no info
```

Control (cont'd.)



```
-- Ada code:
if -- something goes wrong
then
   raise Trouble; -- use Trouble as a constant
elsif -- something even worse happens
then
   raise Big_Trouble; -- use Big_Trouble as a constant
end if;

(* ML code: *)
if (* something goes wrong *)
then (* construct a Trouble value *)
   raise Trouble("Bad news!",current_item)
else if (* something even worse happens *)
   raise Big_Trouble (* Big_Trouble is a constant *)
else ... ;
```

Control (cont'd.)

- When an exception is raised, the current computation is abandoned, and the runtime system begins to search for a handler
- In Ada and C++, the current block is searched first, then the enclosing block, and so on
  - This is called **propagating the exception**
- If the outermost block of a function or procedure is reached without finding a handler, the call is exited and the exception is raised in the caller
- Process continues until a handler is found or the main program is exited, calling the default handler

Control (cont’d.)

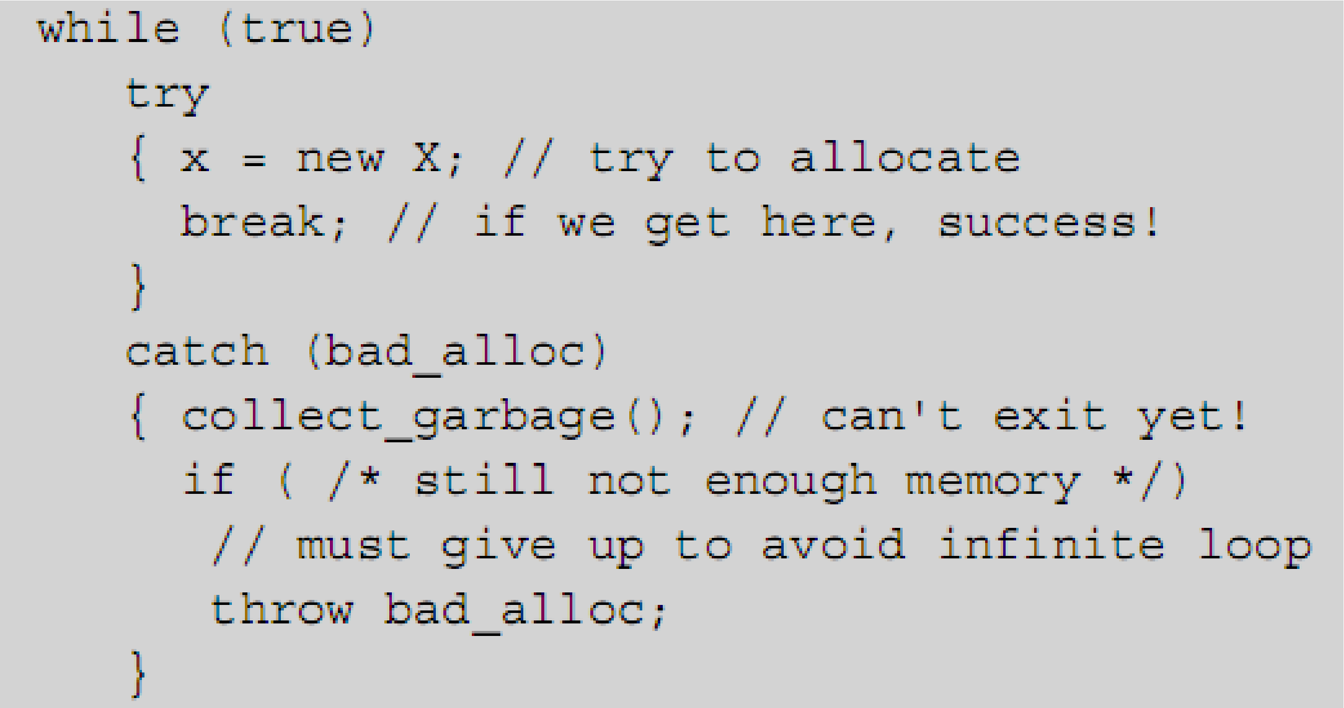
- **Call unwinding** (or **stack unwinding**): process of exiting back through function calls to the caller during the search for a handler
- Once a handler is found and executed, where should execution continue?
  - **Resumption model**: continue at the point at which the exception was first raised, and redo that same statement or expression
  - **Termination model**: continue with the code immediately following the block or expression in which the handler that was executed is found

Control (cont’d.)

- Most modern languages use the termination model
  - Generally easier to implement and fits better into structured programming techniques
  - Can simulate the resumption model when needed
- Avoid overusing exceptions to implement ordinary control situations because:
  - Exception handling often carries substantial runtime overhead
  - Exceptions represent a not very structured control alternative
- Use simple tests instead

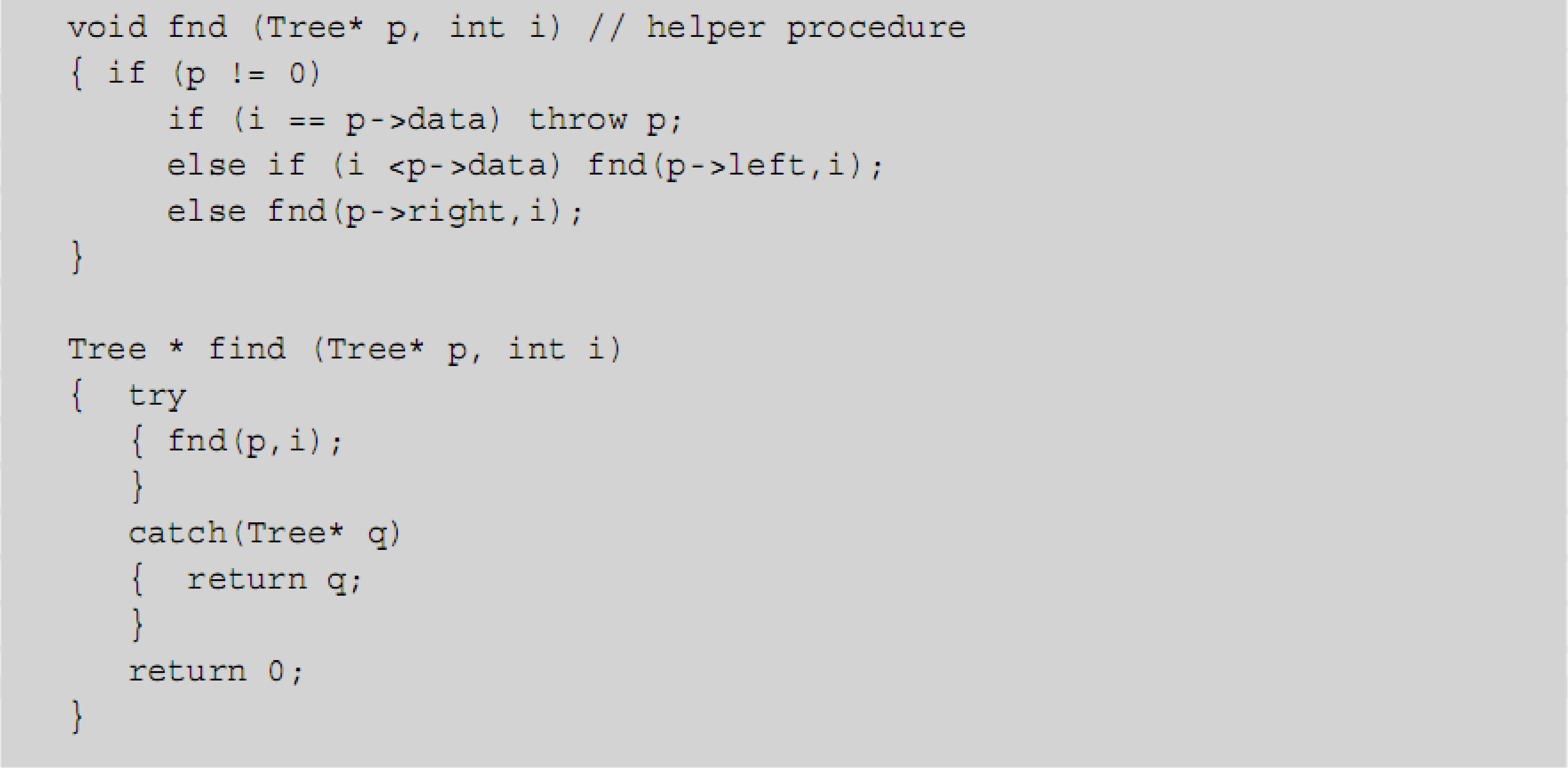
Control (cont’d.)

- Example: In C++, simulating the resumption model when a call to new failed due to insufficient memory



```
while (true)
   try
   { x = new X; // try to allocate
     break; // if we get here, success!
   }
   catch (bad_alloc)
   { collect_garbage(); // can't exit yet!
     if ( /* still not enough memory */)
      // must give up to avoid infinite loop
      throw bad_alloc;
   }
```

Control (cont'd.)



```
void fnd (Tree* p, int i) // helper procedure
{ if (p != 0)
     if (i == p->data) throw p;
     else if (i <p->data) fnd(p->left,i);
     else fnd(p->right,i);
}

Tree * find (Tree* p, int i)
{  try
   { fnd(p,i);
   }
   catch(Tree* q)
   {  return q;
   }
   return 0;
}
```

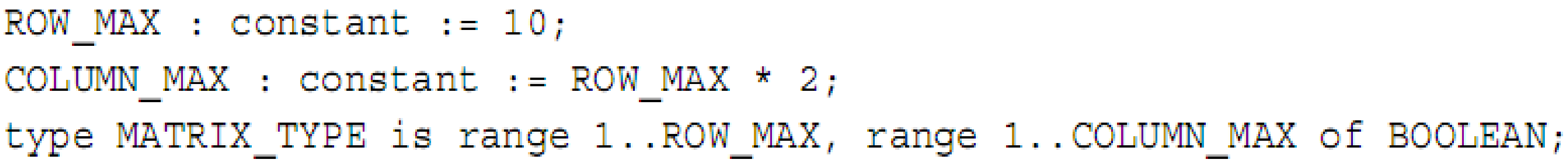
Figure 9.13 A binary tree `find` function in C++ using exceptions (adapted from Stroustrup [1997], pp. 374–375)

Case Study: Computing the Values of Static Expressions in TinyAda

- Pascal requires its symbolic constants to be defined as literals
- Ada allows static expressions as constants
- **Static expression**: any expression not including a variable or a function call, whose value can be determined at compile time

The Syntax and Semantics of Static Expressions

- Static expressions can appear in two types of TinyAda declarations:
  - A number declaration, which defines a symbolic constant
  - A range type definition, which defines a new subrange type
- Example:



```
ROW_MAX : constant := 10;
COLUMN_MAX : constant := ROW_MAX * 2;
type MATRIX_TYPE is range 1..ROW_MAX, range 1..COLUMN_MAX of BOOLEAN;
```

The Syntax and Semantics of Static Expressions (cont'd.)

- Syntactically, static expressions look just like other expressions
- Semantically, they are also similar
- To ensure the results can be computed at compile time, static expressions cannot include variables or parameter references

Entering the Values of Symbolic Constants

- Each symbolic constant has a `value` attribute in its symbol entry record
- Cannot reuse the parsing method expression presented in an earlier chapter because:
  - All expression methods return a type descriptor, which is still needed for type checking and to set type attributes of constant identifiers and subrange types
  - These methods permit variables and parameter names
  - Not all expressions are static

Looking Up the Values of Static Expressions

- New method `staticPrimary` will give the value of simplest form of a static expression
  - Will be an integer or character literal in the token stream or the value of a constant identifier
- Similar in structure to its nonstatic counterpart, except:
  - New method returns a symbol entry
  - New method looks up an identifier rather than calling the method `name`

Computing the Values of Static Expressions

- If operators are encountered, we must deal with two or more symbol entries, each of which is the result of parsing an operand expression