8

Control in Sequential Languages

After looking briefly at the history of jumps and structured control, we will study exceptions and continuations. Exceptions are a form of jump that exits a block or function call, returning to some previously established point for handling the exception. Continuations are a more general form of "return" based on calling a function that is passed into a block for this purpose. The chapter concludes with a discussion of *force* and *delay,* complimentary techniques for delaying computation by placing it inside a function and forcing delayed computation with a function call.

8.1 STRUCTURED CONTROL

8.1.1 Spaghetti Code

In Fortran or assembly code, it is easy to write programs with incomprehensible control structure. Here is a short code fragment that illustrates a few of the possibilities. The fragment includes a Fortran `CONTINUE` statement, which is an instruction that does nothing but is used for the purpose of placing a label between two instructions. If you scan the code from top to bottom, you might get the idea that the instructions between labels 10 and 20 act together to perform some meaningful task. However, then as you scan downward, you can see that it is possible later to jump to instruction 11, which is in the middle of this set of instructions:

```
10 IF (X .GT. 0.000001) GO TO 20
   X = -X
11 Y = X*X - SIN(Y)/(X+1)
   IF (X .LT. 0.000001) GO TO 50
20 IF (X*Y .LT. 0.00001) GO TO 30
   X = X-Y-Y
30 X = X+Y
   ...
50 CONTINUE
```


https://doi.org/10.1017/CBO9780511804175.009 Published online by Cambridge University Press

GUY STEELE

An energetic man with sincere conviction and a sense of humor, Guy Lewis Steele, Jr., is a Distinguished Engineer at Sun Microsystems, Inc. He received his A.B. in applied mathematics from Harvard College (1975) and his S.M. and Ph.D. in computer science and artificial intelligence from MIT (1977, 1980). He is known for his seminal work on Scheme, including the revolutionary continuation-based Rabbit compiler described in his S.M. thesis, and subsequent years of work on Common Lisp, parallel computing, and Java. In addition to reference books on Common Lisp, Fortran, C, and Java, Guy was the original lead author of *The Hacker's Dictionary*, now revised as *The New Hacker's Dictionary*.

Guy Steele's interests also include chess and music. A Life Member of the United States Chess Federation, he has sung in the bass section of the MIT Choral Society, the Masterworks Chorale, and in choruses with the Pittsburgh Symphony Orchestra and the Boston Concert Opera. Guy composed The Telnet Song, published in the April 1984 ACM, which pokes lighthearted fun at the exponential explosion of ctrl-Q keystrokes needed to exit cascaded telnet sessions.

In a masterful 1998 plenary address at the Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA) conference, Guy illustrated the difficulty of growing a language by starting with words of one syllable and defining every longer word in his talk from one-syllable words and previously defined words of two or more syllables. You can find the text of this speech on the web. Guy Steele was awarded the ACM Grace Murray Hopper Award in 1988.

```
X = A
Y = B-A + C*C
GO TO 11
...
```

https://doi.org/10.1017/CBO9780511804175.009 Published online by Cambridge University Press

Although no short sequence can begin to approximate the Byzantine control flow of many archaic Fortran programs, this example may give you some feel for the kind of confusing control flow that was used in many programs in the early days of computing.

8.1.2 Structured Control

In the 1960s, programmers began to understand that unstructured jumps could make it difficult to understand a program. This was partly a realization about programming style and partly a realization about programming languages: If incomprehensible control flow is bad programming style, then programming languages should provide mechanisms that make it easy to organize the control structure of programs. This led to the development of some constructs that structure jumps:

```
if ... then ... else ... end
while ... do ... end
for ... { ... }
case ...
```

These are now adopted in virtually all modern languages.

In modern programming style, we group code in logical blocks, avoid explicit jumps except for function returns, and cannot jump *into* the middle of a block or function body.

The restriction on jumps into blocks illustrates the value of leaving a construct out of a programming language. If a label is placed in the middle of a function body and a program executes a jump to this label, what should happen? Should an activation record be created for the function call? If not, then local variables will not be meaningful. If so, then how will function parameters stored in the activation record be set? Without executing a call to the function, there are no parameter values to use in the call. Because these questions have no good, convincing, clear answers, it is better to design the compiler to reject programs that might jump into the middle of a function body.

Although most introductory programming books and courses today emphasize the importance of clean control structure and discourage the use of `go to` (if the language even allows it), it took many years of discussion and debate to reach this modern point of view. One reason for the change in perspective over the years is the decreasing importance of instruction-level efficiency. In 1960 and even 1970, there were many applications in which it was useful to save the cost of a test, even if it meant complicating the control structure of the program. Therefore, programmers considered it important to be able to jump out of the middle of a loop, avoiding another test at the top of the loop.

In the 1980s and 1990s, as computer speed increased, the number of applications in which a small change in efficiency would truly matter decreased significantly, to the point at which, in the 1990s, Java was introduced without any `go to` statement. Those interested in history may enjoy reading E.W. Dijkstra's March 1968 letter to the editor of *Communications of the ACM*, "Go To Considered Harmful," later

https://doi.org/10.1017/CBO9780511804175.009 Published online by Cambridge University Press

posted on the Association for Computing Machinery web site as the October 1995 "Classic of the Month." The letter begins with these sentences:

> For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of **go to** statements in the programs they produce. More recently I discovered why the use of the **go to** statement has such disastrous effects, and I became convinced that the **go to** statement should be abolished from all "higher level" programming languages. . . .

Simple control structures such as if-then-else have now been in common use since the rise of Pascal in the late 1970s. Exceptions and continuations, discussed in the remainder of this chapter, are more recent innovations in programming languages.

8.2 EXCEPTIONS

8.2.1 Purpose of an Exception Mechanism

Exceptions provide a structured form of jump that may be used to exit a construct such as a block or function invocation. The name *exception* suggests that exceptions are to be used in exceptional circumstances. However, programming languages cannot enforce any sort of intention like this. Exceptions are a basic mechanism that can be used to achieve the following effects:

- jump out of a block or function invocation
- pass data as part of the jump
- return to a program point that was set up to continue the computation.

In addition to jumping from one program point to another, there is also some memory management associated with exceptions. Specifically, unnecessary activation records may be deallocated as the result of the jump. We subsequently see how this works.

Exception mechanisms may be found in many programming languages, including Ada, C++, Clu, Java, Mesa, ML, and PL/1. Every exception mechanism includes two constructs:

- a statement or expression form for *raising* an exception, which aborts part of the current computation and causes a jump (transfer of control),
- a *handler* mechanism, which allows certain statements, expressions, or function calls to be equipped with code to respond to exceptions raised during their execution.

Another term for raising an exception is *throwing* an exception; another term for handling an exception is *catching* an exception.

There are several reasons why exceptions have become an accepted language construct. Many languages do not otherwise have a clean mechanism for jumping out of a function call, for example, aborting the call. Rather than using `go to`s, which can be used in unstructured ways, many programmers prefer exceptions, which can be used to jump only out of some part of a program, not into some part of the program that has not been entered yet. Exceptions also allow a programmer to pass data as part of the jump, which is useful if the program tries to recover from some kind of error condition. Exceptions provide a useful dynamic way of determining to where

https://doi.org/10.1017/CBO9780511804175.009 Published online by Cambridge University Press

a jump goes. If more than one handler is declared, the correct handler is determined according to dynamic scoping rules. This effect would not be easy to achieve with other forms of control jumps.

The importance of dynamic scoping is illustrated in the following example.

Example 8.1

This example involves computing the inverse of a matrix. For the purpose of this example, a matrix is an array of real numbers. We compute matrix multiplication, used in linear algebra, by multiplying the entries of two matrices together in a certain way. If A is a matrix and I is the identity matrix (with ones along the diagonal and zeros elsewhere), then the inverse A^{-1} of A is a matrix such that the product $AA^{-1} = I$. A square matrix has an inverse if and only if something called the *determinant* of A is not zero. It is approximately as much computational work to find the determinant of a matrix as it is to invert a matrix.

Suppose we write a function for inverting matrices that have real-number entries. Because only matrices with nonzero determinants can be inverted, we cannot correctly compute the inverse of a matrix if the determinant turns out to be zero. One way of handling this difficulty might be to check the determinant before calling the invert function. This is not a good idea, however, as computing the determinant of a matrix is approximately as much work as inverting it. Because we can easily check the determinant as we try to convert the inverse, it makes more sense to try to invert the matrix and then raise an exception in the event that we detect a zero determinant. This leads to code structured as follows:

```
exception Determinant; (* declare Determinant exception *)
fun invert(aMatrix) =
  ...
    if ...
    then raise Determinant (* if matrix determinant is zero *)
         else ...
    end;
invert(myMatrix) handle Determinant ...;
```

In this example, the function invert raises an exception if the determinant of aMatrix turns out to be zero. If the function raises this exception as a result of the call invert(myMatrix), then because of the associated handler declaration, the call to invert is aborted and control is transferred to the code immediately following handle Determinant.

In this example, the exception mechanism is used to handle a condition that makes it impossible to continue the computation. Specifically, if the determinant is zero, the matrix has no inverse, and it is impossible for the invert function to return the inverse of the matrix.

If you think about it, this example illustrates the need for some kind of dynamic scoping mechanism. More specifically, suppose that there are several calls to invert in a program that does matrix calculations. Each of these calls could cause the Determinant

https://doi.org/10.1017/CBO9780511804175.009 Published online by Cambridge University Press

exception to be raised. When a call raises this exception, where would we like the exception to be handled?

The main idea behind dynamic scoping of exception handlers is that the code in which the invert function is called is the best place to decide what to do if the determinant is zero. If exceptions were statically scoped, then raising an exception would jump to the exception handler that is declared lexically before the definition of the invert function. This handler would be in the program text before the invert function. If invert is in a program library, then the handler would have to be in the library. However, the person who wrote the library has no idea what to do in the cases in which the exception occurs. For this reason, exceptions are dynamically scoped: When an exception is raised, control jumps to the handler that was most recently established in the dynamic call history of the program. We will see how this works in more detail in after looking at some specific exception mechanisms.

8.2.2 ML Exceptions

The ML mechanism has three parts:

- Exceptions *declarations*, which declare the name of an exception and specify the type of data passed when this exception is raised,
- A *raise* expression, which raises an exception and passes data to the handler,
- *Handler* declarations, which establish handlers.

An ML exception declaration has the form

```
exception ⟨name⟩ of ⟨type⟩
```

where ⟨name⟩ is the name of the exception and ⟨type⟩ is the type of data that are passed to an exception handler when this exception is raised. The last part of this declaration, of ⟨type⟩, is optional. For example, the exception Ovflw below does not pass data, but the exception Signal requires an integer value that is passed to the receiving handler:

```
exception Ovflw;
exception Signal of int;
```

An ML raise expression has the form

```
raise ⟨name⟩   ⟨arguments⟩
```

where ⟨name⟩ is a previously declared exception name and ⟨arguments⟩ have a type that matches the exception declaration. For example, the exceptions previously declared can be raised with the following expressions:

https://doi.org/10.1017/CBO9780511804175.009 Published online by Cambridge University Press

```
raise Ovflw;
raise Signal (x+4);
```

An ML handler is part of an expression form that allows a handler to be specified. An expression with handler has the form

```
⟨exp1⟩ handle ⟨pattern⟩ => ⟨exp2⟩;
```

We evaluate this expression by evaluating ⟨exp1⟩. If the evaluation of ⟨exp1⟩ terminates normally, then this is the value of the larger expression. However, if ⟨exp1⟩ raises an exception that matches ⟨pattern⟩, then any values passed in raising the exception are bound according to ⟨pattern⟩ and ⟨exp2⟩ is evaluated. In this case, the value of ⟨exp2⟩ becomes the value of the entire expression. If ⟨exp2⟩ raises an exception or ⟨exp2⟩ raises an exception that does not match ⟨pattern⟩, then ⟨exp1⟩ handle ⟨pattern⟩ => ⟨exp2⟩ has an uncaught exception that can be caught by the handler established by an enclosing expression or function call. A more general form involving multiple patterns is described below.

Examples

Here is a simple example that uses the "overflow" exception previously mentioned:

```
exception Ovflw;          (* Declare exception name *)
fun f(x) = if x <= Min    (* Function with exception *)
           then raise Ovflw
           else 1/x;
(* – Expression that handles Ovflw in two ways – *)
( f(x) handle Ovflw => 0 ) / (f(x) handle Ovflw => 1)
```

Note that the final expression has two different handlers for the Ovflw exception. In the numerator, the handler returns value 0, making the fraction zero if overflow occurs. In the denominator, it would cause division by zero if the handler returns zero; the handler therefore returns 1 if the Ovflw exception is raised. This example shows how the choice of handler for an exception raised inside a function depends on how the function is called.

Here is another example, illustrating the way that data may be passed and used:

```
exception Signal of int;
fun f(x) = if x=0 then raise Signal(0)
      else if x=1 then raise Signal(1)
      else if x 10 then raise Signal(x-8)
      else (x-2) mod 4;
```

https://doi.org/10.1017/CBO9780511804175.009 Published online by Cambridge University Press

```
f(10) handle Signal(0) => 0
      |  Signal(1) => 1
      |  Signal(x) => x+8;
```

The handler in this expression uses pattern matching, which follows the form established for ML function declarations. More specifically, the meaning of an expression of the form

```
<exp> handle <pattern1> => <exp1>
           |  <pattern2> = <exp2>
                 ...
           |  <patternn> => <expn>
```

is determined as follows:

1. The expression to the left of the handle keyword is evaluated.
2. If this expression terminates normally, its value is the value of the entire expression with handler declaration; the handler is never invoked. (If evaluation of this expression does not terminate, then evaluation of the enclosing expression cannot terminate either.) If the expression raises an exception that matches <pattern$_i$> (and there is no matching handler declared within <exp>), then the handler is invoked.
3. If the handler is invoked, pattern matching works just as an ordinary ML function call. The value passed by exception is matched against <pattern$_1$>, <pattern$_2$>, . . ., <pattern$_n$> in order until a match is found. If the value matches <pattern$_i$>, this causes any variables in <pattern$_i$> to be bound to values. The corresponding expression <exp$_i$> is evaluated with the bindings created by pattern matching.

8.2.3 C++ Exceptions

C++ exceptions are similar in spirit to exceptions in ML and other languages. The C++ syntax involves try blocks, a throw statement, and a catch block to handle exceptions that have been thrown within the associated try block. C++ exceptions are slightly less elegant than ML exceptions because C++ exceptions are not a separate kind of entity recognized by the type system.

The try block surrounds statements in which exceptions may be thrown. Here is a code fragment showing the form of a try block:

```
try {
     // statements that may throw exceptions
}
```

A throw statement may be executed within a try block, either directly by a statement

https://doi.org/10.1017/CBO9780511804175.009 Published online by Cambridge University Press

in the try block or from a function called directly or indirectly from the block. A throw statement contains an expression and passes the value of the expression. Here is an example in which a character value is thrown:

```
throw "This generates a char * exception";
```

A catch block may immediately follow a try block and receive any thrown exceptions. Here is an example a catch-block receiving char * exceptions:

```
catch (char *message) {
        // statements that process the thrown char * exception
}
```

C++ uses types to distinguish between different kinds of exceptions. The throw statement may be used to throw different types of values:

```
throw "Hello world";           // throws a char *
throw 18;                         // throws an int
throw new String("hello"); // throws a String *
```

In a block that may throw more than one type of exceptions, multiple catch blocks may be used:

```
try {
     // code may throw char pointers and other pointers
}
catch (char *message) {
     // code processing the char pointers thrown as exceptions
}
catch (void *whatever) {
     // code processing all other pointers thrown as exceptions
}
```

Although it is possible to throw objects, some care is required because local objects are deallocated when an exception is thrown. More discussion of objects and storage allocation in C++ appears in Chapter 12.

Here is a more complete C++ example, combining try, throw, and catch:

```
void f(char * c) {
   . . . if (c == 0) throw exception("Empty string argument");
}
main() {
```

https://doi.org/10.1017/CBO9780511804175.009 Published online by Cambridge University Press

```
try {
 . . . f(x); . . .
}
catch (exception) {
 exit(1);
}
}
```

As in ML and other languages, throwing a C++ exception causes a control transfer to the most recently established handler that is appropriate for the exception. Whereas ML uses pattern matching to determine whether a handler is appropriate for an exception, C++ uses type matching.

As is generally the case for C/C++, the type-matching issues are a little more complicated that we would like. To be specific, a handler of the form

```
catch(T t)
catch(const T t)
catch(T& t)
catch(const T& t)
```

can catch exception objects of type E if

- `T` and `E` are the same type, or
- `T` is an accessible base class of `E` at the throw point, or
- `T` and `E` are pointer types and there exists a standard pointer conversion from `E` to `T` at the throw point.

The rules for standard pointer conversion are also a little complicated, but we do not discuss them here because they are not critical for understanding the basic idea of exceptions.

One significant difference between ML and C++ that is important when programming with exceptions is that ML is garbage collected and C++ is not. In both languages, raising or throwing an exception will cause all run-time stack activation records between the point of the throw and the point of the catch to be deallocated. However, storage that is reachable from activation records may no longer be reachable after the exception, as the activation records with pointers no longer exist. This problem is discussed at the end of Subsection 8.2.4. There are also some details regarding the way exceptions work in constructors and destructors of objects that will be of interest to C++ programmers.

8.2.4 More about Exceptions

This section contains some additional examples of exceptions, with ML used as an illustrative syntax, and we discuss the interaction between exceptions, storage management, and static type checking. If you have used exceptions in a language different from ML, you might think about how the exception mechanism you have used

https://doi.org/10.1017/CBO9780511804175.009 Published online by Cambridge University Press

is different and whether the difference is a consequence of some basic difference between the underlying programming languages.

Exceptions for Error Conditions

Exceptions arose as a mechanism for handling errors that occur when a program is running. One common form of run-time error occurs when an operation is not defined on some particular arguments. For example, division by zero raises an exception in many languages that have exception mechanisms. Here is a simple ML example, involving the left-subtree function that is not meaningful for trees that have only one node:

```
datatype 'a tree = Leaf of 'a | Node of 'a tree * 'a tree;
exception No_Subtree;
fun lsub ( Leaf x ) = raise No_Subtree
|   lsub (Node(x,y)) = x;
```

In this example, a function lsub(t) returns the left subtree of t if the tree has two subtrees (left and right). However, if there is no left subtree, then the No_Subtree exception is raised.

Exceptions for Efficiency

Sometimes it is useful to terminate a computation when the answer is evident. Exceptions can be useful for this purpose, as an exception terminates a computation. Consider the following code for computing the product of the integers stored at the leaves of a tree. This is written for the tree data type previously defined:

```
fun prod (Leaf x) = x : int
|   prod (Node(x,y)) = prod(x) * prod(y);
```

This correctly computes the product of all the integers stored at the leaves of a tree, but is inefficient in the case that some of the leaves are zero. If we are frequently computing the product for large trees that have zero at one or more leaves, then we might want to optimize this function as follows:

```
fun prod(aTree) =
    let exception Zero
        fun p(Leaf x) = if x = 0 then raise Zero else x
        |   p(Node(x,y)) = p(x) * p(y)
    in
        p(aTree) handle Zero => 0
    end;
```

In this function, a test is performed at each leaf to see if the value is zero. If it is, then an

https://doi.org/10.1017/CBO9780511804175.009 Published online by Cambridge University Press

exception is raised and no other leaves of the tree are examined. This function is less efficient than the preceding one for trees without a zero, but is more efficient if zero is found. Even if the last leaf is zero, raising an exception avoids all the multiplications that would otherwise be performed.

Static and Dynamic Scope

We look at two ML code fragments that use identifier X in analogous ways. In the first example, X is an expression variable and hence is accessed according to static scoping rules. In the second, X is the name of an exception, and so its handlers are used according to dynamic scoping rules. The point of these two code fragments is to see the differences between the two scoping rules and to clearly illustrate that exception handlers are determined by dynamic scope.

The following code illustrates static scoping:

```
val x = 6;
let fun f(y) = x
and g(h) = let val x = 2 in h(1) end
in
      let val x = 4 in g(f) end
end;
```

Under static scoping, the value returned by the call to g(f) is the value of x in the scope where f is declared, which is 6.

If we rewrite this code making X an exception, we will see what value we get under dynamic scoping. One thing to remember when looking at the following code is that ML exceptions use a postfix notation, so although the code is structurally quite similar, it looks somewhat different at first glance:

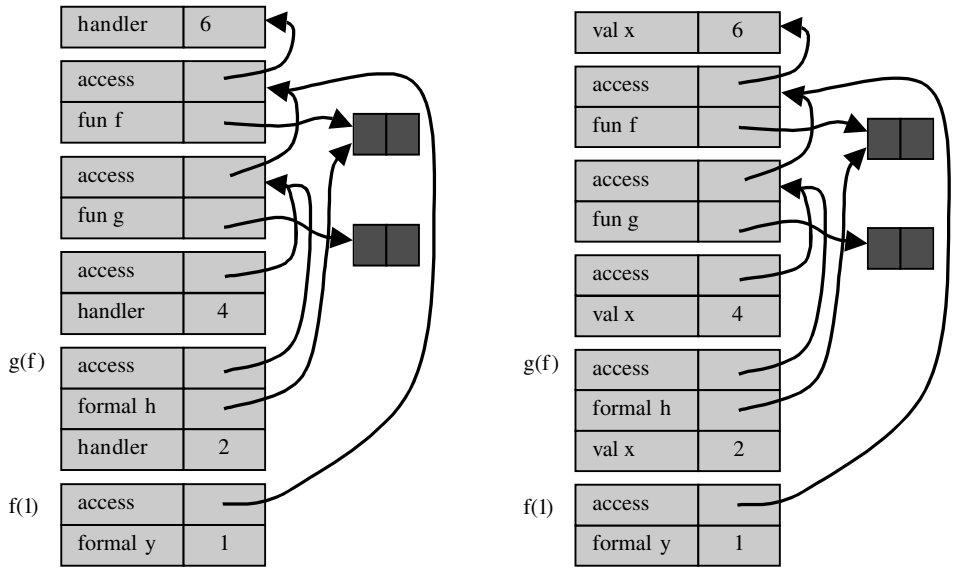
```
exception X;
(let fun f(y) = raise X
and g(h) = h(1) handle X => 2
in
     g(f) handle X => 4
end ) handle X => 6;
```

Here, the value of the g(f) expression is 2. The handler X that is used is the latest one at the time the function raising the exception is called.

The following illustration shows the run-time stack for each code fragment, following the style explained in Chapter 7, with the exception code on the left and the corresponding code with values in place of exceptions on the right. For brevity, only the handler (or identifier) values and the access links are shown in each activation record, except that the parameter value is also shown in the activation record for each function call. If you begin at the bottom of the stack and search up the stack for the most recent handler, which is the rule for dynamic scope, this is the one

https://doi.org/10.1017/CBO9780511804175.009 Published online by Cambridge University Press

with value 2. On the other hand, static scoping according to access links leads to value 6.



This illustration also clearly shows which activation records will be popped off the run-time stack when an exception is raised. Specifically, when the exception `X` is raised in the code associated with the stack on the left, control transfers to the handler identified in the top activation record. At this point, all of the activation records below the activation record with handler `X` and value `6` are removed from the run-time stack because they are not needed to continue the computation.

Typing and Exceptions

In ML, the expression

```
e1 handle A => e2;
```

returns either the value of `e1` or the value of `e2`. This suggests that the types of `e1` and `e2` should be the same. To see why in more detail, look at this expression, which contains the preceding one:

```
1 + (e1 handle A => e2);
```

Here, the value of `e1 handle A => e2` is added to `1`, so both `e1` and `e2` must be integer expressions. More generally, the type system can assign only one type to the expression `(e1 handle A => e2)`. Because `(e1 handle A => e2)` can return either the value of `e1` or the value of `e2`, both `e1` and `e2` must have the same type.

https://doi.org/10.1017/CBO9780511804175.009 Published online by Cambridge University Press

The situation for raise is very different, as raise <exception> does not have a value. To see how this works, consider the expression

```
1 + raise No_Value
```

where No_Value is a previously declared exception. In this expression, the addition will never be performed because raising the exception jumps to the nearest handler, which is outside the expression shown. In ML, the type of raise <exception> is a type variable 'a, which allows the type-inference algorithm to give raise <exception> any type.

Exceptions and Resource Allocation

Raising an exception may cause a program to jump out of any number of in-line blocks and function invocations. In each of these blocks, the program may have allocated data on the stack or heap. In ML, C++, and other contemporary languages, all data allocated on the run-time stack will be reclaimed after an exception is raised. This occurs when the handler is located and intervening activation records are popped off the run-time stack.

Data allocated on the heap are treated differently in different languages. In ML, data on the heap that is no longer reachable after the exception will become garbage. Because ML is garbage collected, the garbage collector will reclaim this unreachable data. The situation is illustrated by the following code:

```
exception X;
(let
   val y = ref [1,2,3]
in
   ... raise X
end) handle X => ... ;
```

The local declaration let val x= ... in ... end causes a list [1,2,3] to be built in the heap. When raise X raises an exception inside the scope of this declaration, control is transferred to the handler outside this scope. The reference y, stored in the activation record associated with the local declaration, is popped off the stack. The list [1,2,3] remains in the program heap and is later collected whenever the garbage collector happens to run.

In C++, storage that is reachable from activation records may no longer be reachable after an exception is thrown. Because C++ is not garbage collected, it is up to the programmer to make sure that any data stored on the heap are explicitly deallocated. However, it may be impossible to explicitly deallocate memory that was previously allocated by the program if the only pointers to the data were those on the run-time stack. In particular, in the C++ version of the preceding program, with a linked list created in the heap, the only pointer to the list is the pointer y on the run-time stack. After the exception is thrown, the pointer y is gone, and there is no way to reach

https://doi.org/10.1017/CBO9780511804175.009 Published online by Cambridge University Press

the data. There are two general solutions: Either make sure there is some way of reaching all heap data from the handler or accept the phenomenon and live with the memory leak. The C++ implementation provides some assistance for this problem by invoking the destructor of each object on the run-time stack as the containing activation record is popped off the stack. This solves the list problem if we build the list by placing one list object on the stack and by including code in the list destructor that follows the list pointer and invokes the destructor of any reachable list nodes.

The general problem with managing unreachable memory also occurs with other resources. For example, if a file is opened between the point of the handler and the point where an exception is thrown, there may be no way to close the open file. The same problem may occur with synchronization locks on concurrently accessible data areas. There is no systematic language solution that seems effective for handling these situations cleanly. Resource management is simply a complication that must be handled with care when one is programming with exceptions.

8.3 CONTINUATIONS

Continuations are a programming technique, based on higher-order functions, that may be used directly by a programmer or may be used in program transformations in an optimizing compiler. Programming with continuations is also related to the systems programming concepts of *upcall* or *callback* functions. As mentioned in Section 7.4, a callback is a function that is passed to another function so that the second function may call the first at a later time.

The concept of continuation originated in denotational semantics in the treatments of jumps (`goto`) and various forms of loop exit and in systems programming in the notion of upcall discussed in Section 7.4. Continuations have found application in continuation-passing-style (CPS) compilers, beginning with the groundbreaking Rabbit compiler for Scheme, developed by Guy Steele and Gerald Sussman in the mid-1970s. Since that time, continuations have been essential to many of the competitive optimizing compilers for functional languages.

8.3.1 A Function Representing "The Rest of the Program"

The basic idea of a continuation can be illustrated with simple arithmetic expressions. For example, consider the function

```
fun f(x,y) = 2.0*x + 3.0*y + 1.0/x + 2.0/y;
```

Assume that this body of the function is evaluated from left to right, so that if we stop evaluation just before the first division, we will have computed `2.0*x + 3.0*y` and, when the quotient `1.0/x` is completed, the evaluation will proceed with an addition, another division, and a final addition. The *continuation* of the subexpression `1.0/x` is the rest of the computation to be performed *after* this quotient is computed. We

https://doi.org/10.1017/CBO9780511804175.009 Published online by Cambridge University Press

can write the computation completed before this division and the continuation to be invoked afterward explicitly as follows:

```
fun f(x,y) = let val befor = 2.0*x + 3.0*y
                 fun continu(quot) = befor + quot + 2.0/y
             in
                 continu(1.0/x)
             end;
```

The evaluation order and result for the second definition of f will be the same as the first; we have just given names to the part to be computed before and the part to be computed after the division. The function called continu is the continuation of 1.0/x; it is exactly what the computer will do after dividing 1.0 by x. The continuation of the subexpression 1.0/x is a function, as the value of f(x,y) depends on the value of the subexpression 1.0/x.

Let us suppose that, if x is zero, it makes sense to return befor/5.2 as the value of the function, and that otherwise we know that y should be nonzero and we can proceed to compute the function normally. Rather than change the function in a special-purpose way, we can illustrate the general idea by defining a division function that is passed the continuation, applying the continuation only in the case in which the divisor is nonzero:

```
fun divide(numerator,denominator,continuation,alternate_value) =
   if denominator > 0.0001
      then continuation(numerator/denominator)
      else alternate_value;

fun f(x,y) = let val befor = 2.0*x + 3.0*y
                 fun continu(quot) = befor + quot + 2.0/y
             in
                 divide(1.0, x, continu, befor/5.2)
             end;
```

This version of f now uses an error-avoiding version of division that can exit in either of two ways, the normal exit applying the continuation to the result of division and an error exit returning some alternative value passed as a parameter. This is not the most general version of division, however, because, in general, we might have one computation we might wish to perform when division is possible and another to perform when division is not. We can represent the computation on error as a function also, leading to the following revision. Here we have assumed, for simplicity, that computation after error is a function that need not be passed any of the other arguments:

https://doi.org/10.1017/CBO9780511804175.009 Published online by Cambridge University Press

```
fun divide(numerator,denominator,normal_cont,error_cont) =
  if denominator > 0.0001
      then normal_cont(numerator/denominator)
      else error_cont()

fun f(x,y) = let val befor = 2.0*x + 3.0*y
          fun continu(quot) = befor + quot + 2.0/y
          fun error_continu() = befor/5.2
          in
            divide(1.0, x, continu, error_continu)
          end
```

For this specific computation, it is a relatively minor point that the division befor/5.2 is now done only if error_continu is invoked. However, in general, it is far more useful to have normal continuation and error continuation both presented as functions, as error handling may require some computation after the error has been identified.

The preceding example can be handled more simply with exceptions. Ignoring the precomputation of (2.0*x + 3.0*y), which a compiler could identify as a common subexpression anyway, we can write the function as follows:

```
exception Div;
fun f(x,y) = (2.0*x + 3.0*y +
          1.0/(if x > 0.001 then x else raise Div) + 2.0/y
          ) handle Div => (2.0*x + 3.0*y)/5.2
```

In general, continuations are more flexible than exceptions, but also may require more programming effort to get exactly the control you want.

8.3.2 Continuation-Passing Form and Tail Recursion

There is a program form called *continuation-passing form* in which each function or operation is passed a continuation. This allows each function or operation to terminate by calling a continuation. As a consequence, no function needs to return to the point from which it was called. This property of continuation-passing form may remind you of tail calls, discussed in Subsection 7.3.4, as a tail call need not return to the calling function. We will investigate the correspondence after looking at an example.

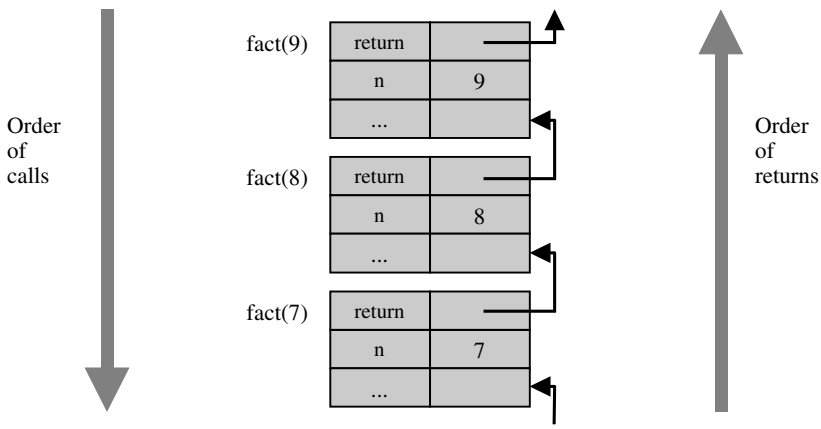
There are systematic rules for transforming an expression or program to CPS. The main idea is that each function or operation should take a continuation representing the remaining computation after this function completes. If we begin with a program that does not contain exceptions or other jumps, then each operation will be expected to terminate normally. Because each function or operation will therefore terminate by calling the function passed to it as a continuation, each function or operation will terminate by executing a tail call to another function.

https://doi.org/10.1017/CBO9780511804175.009 Published online by Cambridge University Press

We can transform the standard factorial function

```
fun fact(n) = if (n=0) then 1 else n*fact(n-1);
```

to continuation-passing form by first examining the continuation of each call. Consider the computation of fact(9), for example. This computation begins with an activation record for fact(9), then a recursive call to fact(8). The activation record for fact(8) points to the activation record for fact(9), as the multiplication associated with fact(9) must be performed after the call for fact(8) returns. The chain of activation records is shown in the following illustration:



Because the invocation fact(9) multiplies the result of fact(8) by 9, the continuation of fact(8) is λx. 9*x.

Similarly, after fact(7) returns, the invocation of fact(8) will multiply the result by 8 and then return to fact(9), which will in turn multiply by 9. Written as a function in lambda notation, the continuation of fact(7) is λy.(λx. 9*x) (8*y).

By similar reasoning, the continuation of fact(6) is λz.(λy. (λx. 9*x) (8*y)) (7*z).

Generalizing from these examples, the continuation of fact(n-1) is the composition of (λx. n*x) with the continuation of fact(n).

Using this insight, we can write a general CPS function to compute factorial. The function

```
f(n, k) = if n=0 then k(1) else f(n-1, λx. k (n*x) )
```

takes a number n and a continuation k as arguments. If n=0, then the function passes 1 to the continuation. If n>0, then the function makes a recursive call. Because the recursive call is to a CPS function, the continuation passed to the call must be the continuation of factorial of n-1. Given factorial x of n-1, the continuation multiplies by n and then passes the result to the continuation k of the factorial of n. Here is a sample calculation, with initial continuation the identity function, illustrating how the continuation-passing function works:

https://doi.org/10.1017/CBO9780511804175.009 Published online by Cambridge University Press

```
f(3, λx.x) = f(2, λy.(( λx.x) (3*y)))
           = f(1, λx.(( λy.3*y)(2*x)))
           = λx.(( λy.3*y)(2*x)) 1
           = 6
```

Intuitively, the continuation-passing form of factorial performs a sequence of recursive calls, accumulating a continuation. When the recursion terminates by reaching n=0, the continuation is applied to 1 and that produces the return value n!. When the continuation is applied, the continuation consists of a sequence of multiplications that compute factorial. This is exactly the same sequence of multiplications as would be done in the ordinary recursive factorial. However, in the case of ordinary recursive factorial, each multiplication would be done by a separate invocation of the factorial function.

Continuations and Tail Recursion. The following ML continuation-passing factorial function appears to use a tail recursive function f to do the calculation: fun factk(n) = let fun:

```
    f(n, k) = if n=0 then k(1) else f(n-1, fn x => k (n*x) )
 in
    f(n, fn x => x)
end
```

The inner function f is tail recursive in the sense that the recursive call to f is the entire value of the function in the case n>0. Therefore, there is no need for the recursive call to return to the calling invocation. However, this does *not* mean that the continuation-passing function factk runs in constant space. The reason has to do with closures and scoping of variables. Specifically, the continuation

```
fn x => k (n*x)
```

contains the identifier x that is a parameter of the function. The continuation is therefore represented as a closure, with a pointer to the activation record of the calling invocation of f. Because the activation record is needed as part of the closure, it is not possible to use tail recursion optimization to eliminate the need for a new activation record.

Factorial can, however, be written in a tail recursive form:

```
fun fact1(n) = let fun
                   f(n,k) = if n=0 then k else f(n-1, n*k)
               in
                   f(n,1)
               end;
```

https://doi.org/10.1017/CBO9780511804175.009 Published online by Cambridge University Press

As discussed in Subsection 7.3.4, the tail recursive form is more efficient, as the compiler can generate code that does not allocate a new activation record for each call. Instead, the same activation record can be used for all invocations of the function f, with each call effectively assigning new values to local variables n and k.

We can derive the tail recursive form from the continuation-passing form by using a little insight. The main idea is that each continuation in the continuation-passing form will be a function that multiplies its argument by some number. We can therefore achieve the same effect by passing the number instead of the function. Although there is no standard, systematic way of transforming an arbitrary function into a tail recursive function that can be executed by use of constant space, there is a systematic transformation to continuation-passing form and, in some instances, a clever individual can produce a tail recursive function from the continuation-passing form.

8.3.3 Continuations in Compilation

Continuations are commonly used in compilers for languages with higher-order functions. The main steps in a continuation-based compiler follow the design pioneered by Steele and Sussman for Scheme (MIT AI MEMO 474, MIT, 1978):

1. lexical analysis, parsing, type checking
2. translation to lambda calculus form
3. conversion to CPS
4. optimization of CPS
5. closure conversion – eliminate free variables
6. elimination of nested scopes
7. register spilling – no expression with more than n free vars
8. generation of target-assembly language program
9. Assembly to produce target-machine program

The core step that makes continuations useful is step 4, optimizing the continuation-passing form of a program. The merit of continuations is that continuations make control flow explicit – each part of the computation explicitly calls a function to do the next part of the computation.

Furthermore, because a continuation-passing operation terminates with a call instead of a return, it is possible to compile calls directly into jumps. Arranging the target code in the correct order can eliminate many of these jumps. Although additional discussion of continuation-based compilation is beyond the scope of this book, there are several compiler books that address this subject in detail and many additional articles and web sources.

8.4 FUNCTIONS AND EVALUATION ORDER

Exceptions and continuations are forms of jumps that are used in high-level programming languages. A final technique for manipulating the order of execution in programs is to use function definitions and calls. More specifically, if a calculation can be put off until later, it may be placed inside a function and passed to code that will eventually decide when to do the calculation. This is most useful if the calculation

https://doi.org/10.1017/CBO9780511804175.009 Published online by Cambridge University Press

might not be needed at all. In our brief survey of this simple technique, we look at controlling the order of evaluation for efficiency.

Delay and force are programming forms that can be used together to optimize program performance. Delay and Force are explicit program constructs in Scheme, but the main idea can be used in any language with functions and static scope. To see how Delay and Force are used, we start with an example. Consider a function declaration and call of the following form:

```
fun f(x,y) =
    ... x ... y ...
...
f(e1, e2)
```

Suppose that

- the value of y is needed only if the value of x has some property, and
- the evaluation of e2 is expensive.

In these circumstances, it is a good idea to delay the evaluation of e2 until we determine (inside the body of f) that the value of e2 is actually needed. If and when we make this determination, we may then force the evaluation of e2. In other words, we would like to be able to write something like the following, in which Delay(e) causes the evaluation of e to be delayed until we call Force(Delay(e)):

```
fun f(x,y) =
    ... x ... Force(y) ...
...
f(e1, Delay(e2))
```

If Force(y) occurs only where the value of y is needed and y is needed at most only once in the body of f, then it should be clear that this form is more efficient than the preceding one. Specifically, if many calls do not require the value of y, then each of these calls will run much more quickly, as we avoid evaluating e2 and we are assuming that evaluation of e2 is expensive. In cases in which the value of e2 is needed, we do the same amount of work as before, plus the presumably small overhead we introduced by adding Delay and Force to the computation.

We discuss two remaining questions: How can we express Delay and Force in a conventional programming language and what should happen if a delayed value is needed more than once?

Let us begin with the first problem, expressing Delay and Force in a conventional language, assuming for simplicity that there will only be one reference to the delayed value. Delay *cannot* be an ordinary function. To see this, suppose briefly that Delay(e) is implemented as a function Delay applied to the expression e. In most languages, the semantics of function calls requires that we evaluate the arguments to a function before invoking the function. Then in evaluating Delay(e) we will evaluate e first.

https://doi.org/10.1017/CBO9780511804175.009 Published online by Cambridge University Press

However, this defeats the purpose of Delay, which was to avoid evaluating e unless its value was actually needed.

Because Delay cannot be a normal function, we are left with two options:

- make Delay a built-in operation of the programming language where we wish to delay evaluation, or
- implement the conceptual construct Delay(e) as some program expression or sequence of commands that is not just a function applied to e.

The first option works fine, but only if we are prepared to modify the implementation of a programming language. Scheme, for example, inherits the concept of special form, a function that does not evaluate its arguments, from Lisp. When special forms are used, a Delay operation that does not evaluate its argument can be defined.

In other languages, we can think of Delay(e) as an abbreviation that gets expanded in some way before the program is compiled. An implementation of Delay and Force that works in ML is

```
Delay(e) == λ().e , which is actually written fn() => e
Force(e) == e()
```

where == means "macro expand to this form before compiling the program."

Here Delay(e) makes e into a parameterless function. The notation λ ().e indicates a function that takes no parameters and that returns e when called. This delays evaluation because the body of a functions is not evaluated until the function is called.

Force evaluates expressions that have previously been delayed. Because delayed expressions are zero-argument functions, Force calls a zero-argument function to cause the function body to be evaluated.

Example 8.2

Here is an example in which the Takeuchi function, tak, is used. The function tak runs for a *very* long time, without using so much stack space that the run-time stack overflows. (Try it!) Because of this characteristic of tak, it is often used as benchmark for testing the speed of function calls in a compiler or interpreter. In this example, the function f has two arguments; the second argument is used only if the first is odd. The purpose of Force and Delay in this example is to make f(fib(9), time_consuming(9)) run more quickly; fib is the Fibonacci function and time_consuming uses tak:

```
fun time_consuming(n) =
    let fun tak(x,y,z) = if x <= y then y
                else tak(tak(x-1,y,z), tak(y-1,z,x), tak(z-1,x,y))
    in
       tak(3*n, 2*n, n)
    end;
fun fib(n) = if n=0 orelse n = 1 then 1 else fib(n-1) + fib(n-2);
fun odd(n) = (n mod 2 ) = 1;
fun f(x,y) = if odd(x) then 1 else fib(y);
```

https://doi.org/10.1017/CBO9780511804175.009 Published online by Cambridge University Press

If we evaluate the following expression directly, it will run for a very long time:

```
f(fib(9), time_consuming(9));
```

Instead we rewrite the function f to expect a Delay-ed value as the second argument, and Force evaluation only if needed:

```
fun lazy_f(x,y) = if odd(x) then 1 else fib( y() );
```

Here is the corresponding call, with calculation of time_consuming(9) delayed:

```
lazy_f( fib(9), fn () => (time_consuming(9)));
```

Because fib(9) is odd, this expression terminates much more quickly than the expression without Delay.

The versions of Delay and Force described in Example 8.2 rely on static scoping and save time only if the delayed function argument is used at most once. Static scoping is necessary to preserve the semantics of the program. Without static scoping, placing an expression inside a function and passing it to another function might change the values of identifiers that appear in the expression. The reason why Delay and Force do not help if the expression is used twice or more is that each occurrence would involve evaluating the delayed expression. This will take extra time and may give the wrong result if the expression has side effects.

It is a relatively simple programming exercise to write versions of Delay and Force that work when the delayed value is needed more than once. The main idea is to store a flag that indicates whether the expression has been evaluated once or not. If not, and the value is needed, then the expression is evaluated and stored so that it can be retrieved without further evaluation when it is needed again. This trick is a form of evaluation that is referred to as *call-by-need* in the literature.

Here is a simple ML version of the code needed to delay a value that may be needed more than once. A delayed value will be a reference cell containing an "unevaluated delay":

```
Delay ( e ) == ref(UN(fn () => e ))
```

where the constructor UN and the type of "delays" are defined by

```
datatype 'a delay = EV of 'a | UN of unit -> 'a;
```

Intuitively, a delayed value is an assignable cell that will contain an unevaluated value until it is evaluated. After that, the assignable cell will contain an evaluated value. The type delay is a union of the two possibilities. A "delay" is either an evaluated

https://doi.org/10.1017/CBO9780511804175.009 Published online by Cambridge University Press

value, tagged with constructor EV, or an unevaluated delay, tagged with constructor UN. The tagged unevaluated delay is a function of no arguments that can be called to get an evaluated value.

The corresponding force function,

```
fun force(d) = let val v = ev(!d) in (d := EV(v); v) end;
```

uses assignment and a subsidiary function ev that evaluates a delay:

```
fun ev(EV(x)) = x
|   ev(UN(f)) = f();
```

In words, if a delay is already evaluated, then ev has nothing to do. Otherwise, ev calls the function of no arguments to get an evaluated delay. To give a concrete example, here is the code for a delayed evaluation of time_consuming(9), followed by a call to force to evaluate it:

```
val d = ref(UN(fn () => time_consuming(9)));
force(d);
```

This call to force evaluates the delayed expression and has a side effect so that, on subsequent calls to force, no further evaluation is needed.

8.5 CHAPTER SUMMARY

We looked at several ways of controlling the order of execution and evaluation in sequential (nonconcurrent) programs. Here are the main topics of the chapter:

- structured programming without go to,
- exceptions,
- continuations,
- Delay and Force.

Control and Go to. Because structured programming is commonly accepted and taught, we did not look at the entire historical controversy surrounding go to statements. The main conclusion in the "Go to considered harmful" debate is that, in the end, program clarity is often more important than absolute efficiency. This has always been true to some degree, but as computer speed has increased, the relation between programmer time and program execution time has shifted. In modern software development, it is not worth several days of programmer time to reduce the execution time of a large application by one or two instruction cycles. Programmer time is expensive, and clever programming can lead to costly mistakes and increased debugging time. An instruction or two takes so little time that for most applications noone will notice the difference.

https://doi.org/10.1017/CBO9780511804175.009 Published online by Cambridge University Press

Exceptions. Exceptions are a structured form of jump that may be used to exit a block or function call and pass a return value in the process. Every exception mechanism includes a statement or expression form for *raising* an exception and a mechanism for defining *handlers* that respond to exceptions. When an exception is raised and several handlers have been established, control is transferred to the handler associated with the most recent activation record on the run-time stack. In other words, handlers are selected according to dynamic scope, not the static scope rules used for most other declarations in modern general-purpose programming languages.

Continuations. Continuation is a programming technique based on higher-order functions that may be used directly in programming or in program transformations in an optimizing compiler. Some mostly functional languages, such as Scheme and ML, provide direct support for capturing and invoking continuations.

Intuitively, the continuation of a statement or expression is a function representing the computation remaining to be performed after this statement or expression is evaluated. There is a general, systematic method for transforming any program into continuation-passing form. A function in continuation-passing form does not return, but calls a continuation (passed as an argument to the function) in order to continue after the function is complete. Continuation-passing form is related to tail recursion (see Subsection 7.3.4). Formally, a continuation-passing function appears to be tail recursive, as all calls are tail calls. However, continuation-passing functions may pass functions are arguments.

This makes it impossible to perform tail recursion elimination – the function created and passed out of a function call may need the activation record of the function in order to maintain the value of statically scoped global variables. However, a clever programmer can sometimes use the continuation-passing version of a function to devise a tail recursive function that will be compiled as efficiently as an iterative loop.

Continuation-passing form is used in a number of contemporary compilers for languages with higher-order functions.

Delay and Force. `Delay` and `Force` may be used to delay a computation until it is needed. When the delayed computation is needed, `Force` is used. `Delay` and `Force` may be implemented in conventional programming languages by use of functions: The delayed computation is placed inside a function and `Force` is implemented by calling this function. This technique is simplest to apply if functions can be declared anywhere in a program. If a delayed value may be needed more than once, this value may be stored in a location that will be used in every subsequent call to `Force`.

EXERCISES

8.1 Exceptions

Consider the following functions, written in ML:

```
exception Excpt of int;
fun twice(f,x) = f(f(x)) handle Excpt(x) => x;
fun pred(x) = if x = 0 then raise Excpt(x) else x-1;
fun dumb(x) = raise Excpt(x);
fun smart(x) = 1 + pred(x) handle Excpt(x) => 1;
```

https://doi.org/10.1017/CBO9780511804175.009 Published online by Cambridge University Press

What is the result of evaluating each of the following expressions?

(a) twice(pred,1);

(b) twice(dumb,1);

(c) twice(smart,0);

In each case, be sure to describe which exception gets raised and where.

8.2 Exceptions

ML has functions hd and tl to return the head (or first element) and tail (or remaining elements) of a list. These both raise an exception Empty if the list is empty. Suppose that we redefine these functions without changing their behavior on nonempty lists, so that hd raises exception Hd and tl raises exception Tl if applied to the empty list nil:

```
- hd(nil);
uncaught exception Hd
- tl(nil);
uncaught exception Tl
```

Consider the function

```
fun g(l) = hd(l)::tl(l) handle Hd => nil;
```

that behaves like the identity function on lists. The result of evaluating g(nil) is nil. Explain why. What makes the function g return properly without handling the exception Tl?

8.3 Exceptions

The following two versions of the closest function take an integer x and an integer tree t and return the integer leaf value from t that is closest in absolute value to x. The first is a straightforward recursive function, the second uses an exception:

```
datatype 'a tree = Leaf of 'a | Nd of ('a tree) * ('a tree);

fun closest(x, Leaf(y))  = y:int
|   closest(x, Nd(y, z)) = let val lf = closest(x, y) and rt = closest(x, z) in
                           if abs(x-lf) < abs(x-rt) then lf else rt end;

fun closest(x, t) =
    let
        exception Found
        fun cls (x, Leaf(y)) = if x=y then raise Found else y:int
        |   cls (x, Nd(y, z)) = let val lf = cls(x, y) and rt = cls(x, z) in
                                if abs(x-lf) < abs(x-rt) then lf else rt end
in
        cls(x, t) handle Found => x
end;
```

(a) Explain why both give the same answer.

(b) Explain why the second version may be more efficient.

https://doi.org/10.1017/CBO9780511804175.009 Published online by Cambridge University Press

8.4 Exceptions and Recursion

Here is an ML function that uses an exception called Odd.

```
fun f(0) = 1
|   f(1) = raise Odd
|   f(3) = f(3-2)
|   f(n) = (f(n-2) handle Odd => ~n)
```

The expression ~n is ML for $-n$, the negative of the integer n.

When f(11) is executed, the following steps will be performed:

```
call f(11)
call f(9)
call f(7)
   ...
```

Write the remaining steps that will be executed. Include only the following kinds of steps:

- function call (with argument)
- function return (with return value)
- raise an exception
- pop activation record of function off stack without returning control to the function
- handle an exception

Assume that if f calls g and g raises an exception that f does not handle, then the activation record of f is popped off the stack without returning control to the function f.

8.5 Tail Recursion and Exception Handling

Can we use tail recursion elimination to optimize the following program?

```
exception OddNum;
let fun f(0,count) = count
      | f(1, count) = raise OddNum
      | f(x, count) = f(x-2, count+1) handle OddNum => -1
```

Why or why not? Explain. This is a tricky situation – try to explain succinctly what the issues are and how they might be resolved.

8.6 Evaluation Order and Exceptions

Suppose we add an exception mechanism similar to the one used in ML to pure Lisp. Pure Lisp has the property that if every evaluation order for expression e terminates, then e has the same value under every evaluation order. Does pure Lisp with exceptions still have this property? [*Hint:* See if you can find an expression containing a function call $f(e_1, e_2)$ so that evaluating e_1 before e_2 gives you a different answer than evaluating the expression with e_2 before e_1.]

8.7 Control Flow and Memory Management

An *exception* aborts part of a computation and transfers control to a handler that was established at some earlier point in the computation. A *memory leak* occurs when memory allocated by a program is no longer reachable, and the memory will not be

https://doi.org/10.1017/CBO9780511804175.009 Published online by Cambridge University Press

deallocated. (The term "memory leak" is used only in connection with languages that are not garbage collected, such as C.) Explain why exceptions can lead to memory leaks in a language that is not garbage collected.

8.8 Tail Recursion and Continuations

(a) Explain why a tail recursive function, as in

```
fun fact(n) =
    let fun f(n, a) = if n=0 then a
                      else f(n-1, a*n)
    in f(n, 1) end;
```

can be compiled so that the amount of space required for computing fact(n) is independent of n.

(b) The function f used in the following definition of factorial is "formally" tail recursive: The only recursive call to f is a call that need not return:

```
fun fact(n) =
    let fun f(n,g) = if n=0 then g(1)
                     else f(n-1, fn x=>g(x)*n)
    in f(n, fn x => x) end;
```

How much space is required for computing fact(n), measured as a function of argument n? Explain how this space is allocated during recursive calls to f and when the space may be freed.

8.9 Continuations

In addition to continuations that represent the "normal" continued execution of a program, we can use continuations in place of exceptions. For example, consider the following function f that raises an exception when the argument x is too small:

```
exception Too_Small;
fun f(x) = if x<0 then raise Too_Small else x/2;
(1 + f(y)) handle Too_Small => 0;
```

If we use continuations, then f could be written as a function with two extra arguments, one for normal exit and the other for "exceptional exit," to be called if the argument is too small:

```
fun f(x, k_normal, k_exn) = if x<0 then k_exn() else k_normal(x/2);
f(y, (fn z => 1+z), (fn () => 0));
```

(a) Explain why the final expressions in each program fragment will have the same value for any value of y.

(b) Why would tail call optimization be helpful when we use the second style of programming instead of exceptions?

https://doi.org/10.1017/CBO9780511804175.009 Published online by Cambridge University Press