CHAPTER 9

Working with Sequences and Structured Data

In the previous chapter, you learned about some techniques used to manipulate *unstructured data*, including how to parse and format structured data as unstructured text. In this chapter, you focus on F# programming techniques related to *structured data*.

Structured data, and its processing, is a broad topic that lies at the heart of much applied F# programming. In this chapter, you will learn about a number of applied topics in structured programming, including:

- *building, transforming* and *querying* in-memory data structures—in particular, using sequence-based programming
- *working with abstract syntax representations,* including some efficiency-related representation techniques
- *building new data structures,* including implementing *equality* and *comparison* operations for these types
- *defining structured views* of existing data structures using *active patterns*
- using *recursion* and *tail-calls*, especially over recursively structured types.

Closely related to programming with structured data is the use of *recursion* in F# to describe some algorithms. In order to use recursion effectively over structured data, you will also learn about *tail calls* and how they affect the use of the "stack" as a computational resource.

Getting Started with Sequences

Many programming tasks require the iteration, aggregation, and transformation of data streamed from various sources. One important and general way to code these tasks is in terms of values of the type `System.Collections.Generic.IEnumerable<type>`, which is typically abbreviated to `seq<type>` in F# code. A `seq<type>` is a value that can be *iterated*, producing results of type `type` on demand. Sequences are used to wrap collections, computations, and data streams and are frequently used to represent the results of database queries. The following sections present some simple examples of working with `seq<type>` values.

D. Syme et al., *Expert F# 3.0*

Using Range Expressions

You can generate simple sequences using *range expressions*. For integer ranges, these take the form of `seq {n .. m}` for the integer expressions `n` and `m`:

```
> seq {0 .. 2};;

val it : seq<int> = seq [0; 1; 2;]
```

You can also specify range expressions using other numeric types, such as `double` and `single`:

```
> seq {-100.0 .. 100.0};;

val it : seq<float> = seq [-100.0; -99.0; -98.0; -97.0; ...]
```

By default, F# Interactive shows the value of a sequence only to a limited depth; `seq<'T>` values are *lazy* in the sense that they compute and return the successive elements on demand. This means you can create sequences representing very large ranges, and the elements of the sequence are computed only if they're required by a subsequent computation. In the next example, you don't actually create a concrete data structure containing one trillion elements, but rather, you create a sequence value that has the *potential* to yield this number of elements on demand. The default printing performed by F# Interactive forces this computation up to depth 4:

```
> seq {1I .. 1000000000000I};;

val it : seq<Numerics.BigInteger> =  seq [1 ; 2; 3; ...]
```

The default increment for range expressions is always 1. A different increment can be used via range expressions of the form `seq {n .. skip .. m}`:

```
> seq {1 .. 2 .. 5};;

val it : seq<int> = seq [1; 3; 5]

> seq {1 .. -2 .. -5};;

val it : seq<int> = seq [1; -1; -3; -5]
```

If the skip causes the final element to be overshot, then the final element isn't included in the result:

```
> seq {0 .. 2 .. 5};;

val it : seq<int> = seq [0; 2; 4]
```

The `(..)` and `(.. ..)` operators are overloaded operators in the same sense as `(+)` and `(-)`, which means their behavior can be altered for user-defined types. Chapter 6 discusses this in more detail.

Iterating a Sequence

You can iterate sequences using the `for ... in ... do` construct, as well as the `Seq.iter` aggregate operator discussed in the next section. Here is a simple example of the first:

```
> let range = seq {0 .. 2 .. 6};;

val range : seq<int>

> for i in range do printfn "i = %d" i;;
i = 0
i = 2
i = 4
i = 6
```

This construct forces the iteration of the entire `seq`. Use it with care when you're working with sequences that may yield a large number of elements.

Transforming Sequences with Aggregate Operators

Any value of type `seq<type>` can be iterated and transformed using functions in the `Microsoft.FSharp.Collections.Seq` module. For example:

```
> let range = seq {0 .. 10};;

val range : seq<int>

> range |> Seq.map (fun i -> (i,i*i));;

val it : seq<int * int> = seq [(0, 0); (1, 1); (2, 4); (3, 9); ...]
```

Table 9-1 shows some important functions in this library module. The following operators necessarily evaluate all the elements of the input `seq` immediately:

- `Seq.iter`: This iterates all elements, applying a function to each.
- `Seq.toList`: This iterates all elements, building a new list.
- `Seq.toArray`: This iterates all elements, building a new array.

Most other operators in the `Seq` module return one or more `seq<type>` values and force the computation of elements in any input `seq<type>` values only on demand.

Table 9-1. *Some Important Functions and Aggregate Operators from the* `Seq` *Module*

Operator	Type
`Seq.append`	`seq<'T> -> seq<'T> -> seq<'T>`
`Seq.concat`	`seq<#seq<'T>> -> seq<'T>`
`Seq.choose`	`('T -> 'U option) -> seq<'T> -> seq<'U>`

Operator	Type
Seq.delay	(unit -> seq<'T>) -> seq<'T>
Seq.empty	seq<'T>
Seq.iter	('T -> unit) -> seq<'T> -> unit
Seq.filter	('T -> bool) -> seq<'T> -> seq<'T>
Seq.map	('T -> 'U) -> seq<'T> -> seq<'U>
Seq.singleton	'T -> seq<'T>
Seq.truncate	int -> seq<'T> -> seq<'T>
Seq.toList	seq<'T> -> 'T list
Seq.ofList	'T list -> seq<'T>
Seq.toArray	seq<'T> -> 'T []
Seq.ofArray	'T [] -> seq<'T>

Which Types Can Be Used as Sequences?

Table 9-1 includes many uses of types such as `seq<'T>`. When a type appears as the type of an argument, the function accepts any value that's compatible with this type. Chapter 5 explained the notions of subtyping and compatibility in more detail; the concept should be familiar to OO programmers because it's the same as that used by languages such as C#, which itself is close to that used by Java and C++. In practice, you can easily discover which types are compatible with which others by using F# Interactive and tools such as Visual Studio: when you hover over a type name, the compatible types are shown. You can also refer to the online documentation for the F# libraries and the .NET Framework, which you can easily obtain using the major search engines.

Here are some of the types compatible with `seq<'T>`:

- *Array types*: For example, `int[]` is compatible with `seq<int>`.
- *F# list types:* For example, `int list` is compatible with `seq<int>`.
- *All other F# and .NET collection types:* For example, `System.Collections.Generic.SortedList<string>` is compatible with `seq<string>`.

Using Lazy Sequences from External Sources

Sequences are frequently used to represent the process of streaming data from an external source, such as from a database query or from a computer's file system. For example, the following recursive function constructs a `seq<string>` that represents the process of recursively reading the names of all the files under a given path. The return types of `Directory.GetFiles` and `Directory.GetDirectories` are `string[]`; and, as noted earlier, this type is always compatible with `seq<string>`:

```
open System.IO
let rec allFiles dir =
    Seq.append
        (dir |> Directory.GetFiles)
        (dir |> Directory.GetDirectories |> Seq.map allFiles |> Seq.concat)
```

This gives the following type:

```
val allFiles : dir:string -> seq<string>
```

Here is an example of the function being used on one of our machines:

```
> allFiles @"c:\WINDOWS\system32";;

val it : seq<string> =
  seq
    ["c:\WINDOWS\system32\12520437.cpx"; "c:\WINDOWS\system32\12520850.cpx";
     "c:\WINDOWS\system32\aaclient.dll";
     "c:\WINDOWS\system32\accessibilitycpl.dll"; ...]
```

The `allFiles` function is interesting partly because it shows many aspects of F# working seamlessly together:

- *Functions as values:* The function `allFiles` is recursive and is used as a first-class function value within its own definition.
- *Pipelining:* The pipelining operator `|>` provides a natural way to present the transformations applied to each subdirectory name.
- *Type inference:* Type inference computes all types in the obvious way, without any type annotations.
- *.NET interoperability:* The `System.IO.Directory` operations provide intuitive primitives, which can then be incorporated in powerful ways using succinct F# programs.
- *Laziness where needed:* The function `Seq.map` applies the argument function lazily (on demand), which means subdirectories aren't read until required.

One subtlety with programming with on-demand or lazy values such as sequences is that side effects such as reading and writing from an external store shouldn't in general happen until the lazy sequence value is consumed. For example, the previous `allFiles` function reads the top-level directory `C:\` as soon as `allFiles` is applied to its argument. This may not be appropriate if the contents of `C:\` are changing. You can delay the computation of the sequence by using the library function `Seq.delay` or by using a sequence expression, covered in the next section, in which delays are inserted automatically by the F# compiler.

Using Sequence Expressions

Aggregate operators are a powerful way of working with `seq<type>` values. However, F# also provides a convenient and compact syntax called *sequence expressions* for specifying sequence values that can be built using operations such as `choose`, `map`, `filter`, and `concat`. You can also use sequence expressions to specify the shapes of lists and arrays. It's valuable to learn how to use sequence expressions:

- They're a compact way of specifying interesting data and generative processes.
- They're used to specify database queries when using data-access layers such as Microsoft's Language Integrated Queries (LINQ). See Chapter 15 for examples of using sequence expressions this way.

- They're one particular use of *computation expressions*, a more general concept that has several uses in F# programming. Chapter 9 discusses computation expressions, and we show how to use them for asynchronous and parallel programming in Chapter 13.

The simplest form of a sequence expression is `seq { for value in expr .. expr -> expr }`. Here, `->` should be read "yield." This is a shorthand way of writing `Seq.map` over a range expression. For example, you can generate an enumeration of numbers and their squares as follows:

```
> let squares = seq { for i in 0 .. 10 -> (i, i * i) };;

val squares : seq<int * int>
```

The more complete form of this construct is `seq { for pattern in sequence -> expression }`. The pattern allows you to decompose the values yielded by the input enumerable. For example, you can consume the elements of `squares` using the pattern `(i,iSquared)`:

```
> seq { for (i, iSquared) in squares -> (i, iSquared, i * iSquared) };;

val it : seq<int * int * int> =
  seq [(0, 0, 0); (1, 1, 1); (2, 4, 8); (3, 9, 27); ...]
```

The input `sequence` can be a `seq<type>` or any type supporting a `GetEnumerator` method.

Enriching Sequence Expressions with Additional Logic

A sequence expression often begins with `for ... in ...`, but you can use additional constructs. For example:

- *A secondary iteration:* `for pattern in seq do seq-expr`
- *A filter:* `if expression then seq-expr`
- *A conditional:* `if expression then seq-expr else seq-expr`
- *A* `let` *binding:* `let pattern = expression in seq-expr`
- *Yielding a value:* `yield expression`

Secondary iterations generate additional sequences, all of which are collected and concatenated together. Filters let you skip elements that don't satisfy a given predicate. To see both of these in action, the following computes a checkerboard set of coordinates for a rectangular grid:

```
let checkerboardCoordinates n =
   seq { for row in 1 .. n do
            for col in 1 .. n do
                let sum = row + col
                if sum % 2 = 0 then
                    yield (row, col)}
```

```
> checkerboardCoordinates 3;;

val it : seq<int * int> = seq [(1, 1); (1, 3); (2, 2); (3, 1); ...]
```

Using let clauses in sequence expressions allows you to compute intermediary results. For example, the following code gets the creation time and last-access time for each file in a directory:

```
let fileInfo dir =
    seq { for file in Directory.GetFiles dir do
            let creationTime = File.GetCreationTime file
            let lastAccessTime = File.GetLastAccessTime file
            yield (file, creationTime, lastAccessTime)}
```

In the previous examples, each step of the iteration produces zero or one result. The final yield of a sequence expression can also be another sequence, signified through the use of the `yield!` keyword. The following sample shows how to redefine the `allFiles` function from the previous section using a sequence expression. Note that multiple generators can be included in one sequence expression; the results are implicitly collated together using `Seq.append`:

```
let rec allFiles dir =
    seq { for file in Directory.GetFiles dir do
            yield file
          for subdir in Directory.GetDirectories dir do
            yield! allFiles subdir}
```

Generating Lists and Arrays Using Sequence Expressions

You can also use range and sequence expressions to build list and array values. The syntax is identical, except the surrounding braces are replaced by the usual `[ ]` for lists and `[| |]` for arrays (Chapter 4 discusses arrays in more detail):

```
> [1 .. 4];;

val it: int list = [1; 2; 3; 4]

> [for i in 0 .. 3 -> (i, i * i)];;

val it : (int * int) list = [(0,0); (1,1); (2,4); (3,9)]

> [|for i in 0 .. 3 -> (i, i * i)|];;

val it : (int * int) [] = [|(0, 0); (1, 1); (2, 4); (3, 9)|]
```

■ **Caution** F# lists and arrays are finite data structures built immediately rather than on demand, so you must take care that the length of the sequence is suitable. For example, `[1I .. 1000000000I ]` attempts to build a list that is 1 billion elements long.

More on Working with Sequences

In this section, we look at more techniques for working with sequences of data. This extends the initial techniques you learned in the previous section. For example, consider the `map` and `filter` operations. You can use these operators in a straightforward manner to query and transform in-memory data. For instance, given a table representing some people in your contacts list, you can select those names that start with the letter *A* as:

```
/// A table of people in our startup
let people =
    [("Amber", 27, "Design")
     ("Wendy", 35, "Events")
     ("Antonio", 40, "Sales")
     ("Petra", 31, "Design")
     ("Carlos", 34, "Marketing")]

/// Extract information from the table of people
let namesOfPeopleStartingWithA =
    people
      |> Seq.map (fun (name, age, dept) -> name)
      |> Seq.filter (fun name -> name.StartsWith "A")
      |> Seq.toList
```

At the end of the set of sequence operations, we use `Seq.toList` to evaluate the sequence once and convert the results into a concrete list. Similarly, the following finds all those in the design department:

```
/// Extract the names of designers from the table of people
let namesOfDesigners =
    people
      |> Seq.filter  (fun (_, _, dept) -> dept = "Design")
      |> Seq.map (fun (name, _, _) -> name)
      |> Seq.toList
```

The output from evaluating these declarations is:

```
val people : (string * int * string) list =
  [("Amber", 27, "Design"); ("Wendy", 35, "Events"); ("Antonio", 40, "Sales");
   ("Petra", 31, "Design"); ("Carlos", 34, "Marketing")]
val namesOfPeopleStartingWithA : string list = ["Amber"; "Antonio"]
val namesOfDesigners : string list = ["Amber"; "Petra"]
```

The map and filter operations on sequences, arrays, lists, and other structured data types are examples of *queries*. In these cases, the queries are expressed by using pipelined combinators that accept the data structure as input. In database terminology, the "map" and "filter" operations are, respectively, called "select" and "where.". Sequence-based functional programming gives you the tools to apply in-memory query logic on all types that are compatible with the F# sequence type, such as F# lists, arrays, sequences, and anything else that implements the `IEnumerable<'a>/seq<'a>` interface.

In the rest of this section, you learn about additional operations over sequences. Some further statistical and numeric operations, such as summing and averaging over sequences, are described in Chapter 10.

Using Other Sequence Operators: Truncate and Sort

The Seq module contains many other useful functions in addition to the map and filter operators, some of which were described in Chapter 3. For instance, a useful query-like function is Seq.truncate, which takes the first *n* elements of a sequence and discards the rest. Likewise, you can sort a sequence by using Seq.sort. In the following example, you extract the first 3,000 even numbers from an unbounded stream of random numbers, and for each you return a pair of the number and its square. The sort operation uses the default comparison semantics for the type of elements in the sequence.

```
/// A random-number generator
let rand = System.Random()

/// An infinite sequence of numbers
let randomNumbers = seq { while true do yield rand.Next(100000) }

/// The first 10 random numbers, sorted
let firstTenRandomNumbers =
    randomNumbers
      |> Seq.truncate 10
      |> Seq.sort                          // sort ascending
      |> Seq.toList

/// The first 3000 even random numbers and sort them
let firstThreeThousandEvenNumbersWithSquares =
    randomNumbers
      |> Seq.filter (fun i -> i % 2 = 0)  // "where"
      |> Seq.truncate 3000
      |> Seq.sort                          // sort ascending
      |> Seq.map (fun i -> i, i*i)         // "select"
      |> Seq.toList
```

The output for evaluating this expression is:

```
// random - results will vary!

val randomNumbers : seq<int>
val firstTenRandomNumbers : int list =
  [9444; 14443; 15015; 20448; 31038; 46145; 69447; 85050; 85509; 92181]
val firstThreeThousandEvenNumbersWithSquares : (int * int) list =
  [(56, 3136); (62, 3844); (66, 4356); (68, 4624); (70, 4900); (86, 7396);
   (144, 20736); (238, 56644); (248, 61504); (250, 62500); ... ]
```

The operations Seq.sortBy and Seq.sortWith take custom key-extraction and key-comparison functions, respectively. For example, the next code sample takes the first 10 numbers from our random-number sequence and sorts them by the *last* digit only (so 17510 appears before 16351, because the last digit 0 is lower than 1):

```
/// The first 10 random numbers, sorted by last digit
let firstTenRandomNumbersSortedByLastDigit =
    randomNumbers
      |> Seq.truncate 10
```

```
    |> Seq.sortBy (fun x -> x % 10)
    |> Seq.toList
```

The output is:

```
val firstTenRandomNumbersSortedByLastDigit : int list =
  [51220; 56640; 88543; 97424; 90744; 11784; 23316; 1368; 71878; 89719]
```

Selecting Multiple Elements From Sequences

Often, you will find yourself writing sequence transformations that select *multiple* elements or *zero-or-one* elements for each input element in the sequence. This is in contrast to the operation `Seq.map`, which always selects *one* new element. The types of `Seq.collect` and `Seq.choose` are:

```
module Seq =
    val choose : chooser : ('T -> 'U option) -> source : seq<'T> -> seq<'U>
    val collect : mapping : ('T -> #seq<'U>) -> source : seq<'T> -> seq<'U>
    val map : mapping : ('T -> 'U) -> source : seq<'T> -> seq<'U>
```

For example, given a list of numbers 1 to 10, the following sample shows how to select the "triangle" of numbers 1, 1, 2, 1, 2, 3, 1, 2,3, 4, etc. At the end of the set of sequence operations, we use `Seq.toList` to evaluate the sequence once and convert the results into a concrete list.

```
// Take the first 10 numbers and build a triangle 1, 1, 2, 1, 2, 3, 1, 2, 3, 4, …
let triangleNumbers =
    [ 1 .. 10 ]
      |> Seq.collect (fun i -> [ 1 .. i ] )
      |> Seq.toList
```

The output is:

```
val triangleNumbers : int list =
  [1; 1; 2; 1; 2; 3; 1; 2; 3; 4; 1; 2; 3; 4; 5; 1; 2; 3; 4; 5; 6; 1; 2; 3; 4;
   5; 6; 7; 1; 2; 3; 4; 5; 6; 7; 8; 1; 2; 3; 4; 5; 6; 7; 8; 9; 1; 2; 3; 4; 5;
   6; 7; 8; 9; 10]
```

Likewise, you can use `Seq.choose` for the special case in which returning only zero-or-one elements, returning an option value `None` to indicate no new elements are produced and `Some` *value* to indicate the production of one element. The `Seq.choose` operation is logically the same as combining a `filter` and `map` operation into a single step. In the example below, you create an 8x8 "game board" full of random numbers, in which each element of the data structure includes the index position and a "game value" at that position. (Note that you will reuse this data structure below to explore several other sequence operations.)

```
let gameBoard =
    [ for i in 0 .. 7 do
          for j in 0 .. 7 do
              yield (i,j,rand.Next(10)) ]
```

We then select the positions containing an even number:

```
let evenPositions =
   gameBoard
     |> Seq.choose (fun (i,j,v) -> if v % 2 = 0 then Some (i,j) else None)
     |> Seq.toList
```

The output is:

```
// random - results will vary!
val gameBoard : (int * int * int) list =  [(0, 0, 9); (0, 1, 2); (0, 2, 8); (0, 3, 2); ...]
val evenPositions : (int * int) list =  [(0, 1); (0, 2); (0, 3); ... ]
```

Finding Elements and Indexes in Sequences

One of the most common operations in sequence programming is searching a sequence for an element that satisfies a specific criteria. The operations you use to do this are `Seq.find`, `Seq.findIndex` and `Seq.pick`. These both will raise an exception if an element or index is not found, and they also have corresponding nonfailing partners `Seq.tryFind`, `Seq.tryFindIndex` and `Seq.tryPick`. The types of these operations are:

```
module Seq =
    val find : predicate : ('T -> bool) -> source : seq<'T> -> 'T
    val findIndex : predicate : ('T -> bool) -> source : seq<'T> -> int
    val pick : chooser : ('T -> bool) -> source : seq<'T> -> 'T
    val tryFind : predicate : ('T -> bool) -> source : seq<'T> -> 'T option
    val tryFindIndex : predicate : ('T -> bool) -> source : seq<'T> -> int option
    val tryPick : chooser : ('T -> 'U option) -> source : seq<'T> -> 'U option
```

For example, given the `board` you defined above, you can search the game board for the first location with a zero game value by using `Seq.tryFind`. This will return `None` if the search fails and `Some` *value* if it succeeds. Likewise, you can combine a selection and projection into a single operation using `Seq.tryPick`.

```
let firstElementScoringZero =
   gameBoard |> Seq.tryFind (fun (i, j, v) -> v = 0)

let firstPositionScoringZero =
   gameBoard |> Seq.tryPick (fun (i, j, v) -> if v = 0 then Some(i, j) else None)
```

The output is:

```
// random - results will vary!

val firstElementScoringZero : (int * int * int) option = Some (1, 5, 0)
val firstPositionScoringZero : (int * int) option = Some (1, 5)
```

Grouping and Indexing Sequences

Search operations such as `Seq.find` search the input sequence linearly, from left to right. Frequently, it will be critical to performance of your code to *index* your operations over your data structures. There are numerous ways to do this, usually by building an indexed data structure, such as a Dictionary (see Chapter 4) or an indexed functional map (see Chapter 3).

In the process of building an indexed collection, you will often start with sequences of data and then *group* the data into buckets. Grouping is also useful, for many other reasons, when categorizing data for structural and statistical purposes. In the following example, you group the elements of the `gameBoard` defined previously into buckets according to the game value at each position:

```
let positionsGroupedByGameValue =
   gameBoard
       |> Seq.groupBy (fun (i, j, v) -> v)
       |> Seq.sortBy (fun (k, v) -> k)
       |> Seq.toList
```

The output is:

```
val positionsGroupedByGameValue : (int * seq<int * int * int>) list =
  [(0, <seq>); (1, <seq>); (2, <seq>); (3, <seq>); (4, <seq>); (5, <seq>);
   (6, <seq>); (7, <seq>); (8, <seq>); (9, <seq>)]
```

Note that each element in the grouping is a key and a sequence of values for that key. It is very common to immediately post-process the results of grouping into a dictionary or map. The built-in functions `dict` and `Map.ofSeq` are useful for this purpose, as they build dictionaries and immutable tree-maps, respectively. For example:

```
let positionsIndexedByGameValue =
   gameBoard
       |> Seq.groupBy (fun (i, j, v) -> v)
       |> Seq.sortBy (fun (k, v) -> k)
       |> Seq.map (fun (k, v) -> (k, Seq.toList v))
       |> dict
let worstPositions = positionsIndexedByGameValue.[0]
let bestPositions = positionsIndexedByGameValue.[9]
```

The output is:

```
// random - results will vary!

val positionsIndexedByGameValue :
  System.Collections.Generic.IDictionary<int,(int * int * int) list>
val worstPositions : (int * int * int) list =
  [|(0, 6, 0); (2, 1, 0); (2, 3, 0); (6, 0, 0); (7, 0, 0)|]
val bestPositions : (int * int * int) list =
  [|(0, 3, 9); (0, 7, 9); (5, 3, 9); (6, 4, 9); (6, 7, 9)|]
```

Note that for highly-optimized grouping and indexing, it can be useful to rewrite your core indexed tables into direct uses of highly optimized data structures, such as `System.Collections.Generic.Dictionary`, as described in Chapter 4.

Folding Sequences

Some of the most general operators supported by most F# data structures are `fold` and `foldBack`. These apply a function to each element of a collection and accumulate a result. For `fold` and `foldBack`, the function is applied in left-to-right or right-to-left order, respectively. If you use the name `fold`, the ordering typically is left to right. Both functions also take an initial value for the accumulator. For example:

```
> List.fold (fun acc x -> acc + x) 0 [4; 5; 6];;

val it : int = 15

> Seq.fold (fun acc x -> acc + x) 0.0 [4.0; 5.0; 6.0];;

val it : float = 15.0

> List.foldBack (fun x acc -> min x acc) [4; 5; 6; 3; 5] System.Int32.MaxValue;;

val it : int = 3
```

The following are equivalent, but no explicit anonymous function values are used:

```
> List.fold (+) 0 [4; 5; 6];;

val it : int = 15

> Seq.fold (+) 0.0 [4.0; 5.0; 6.0];;

val it : float = 15.0

> List.foldBack min [4; 5; 6; 3; 5] System.Int32.MaxValue;;

val it : int = 3
```

If used carefully, the various `foldBack` operators are pleasantly compositional, because they let you apply a selection function as part of the accumulating function:

```
> List.foldBack (fst >> min) [(3, "three"); (5, "five")] System.Int32.MaxValue;;

val it : int = 3
```

The F# library also includes more direct accumulation functions, such as `Seq.sum` and `Seq.sumBy`. These use a fixed accumulation function (addition) with a fixed initial value (zero), and they are described in Chapter 10.

■ **Caution:** Folding operators are very powerful and can help you avoid many explicit uses of recursion or loops in your code. They're sometimes overused in functional programming, however, and they can be hard for novice users to read and understand. Take the time to document uses of these operators, or consider using them to build simpler operators that apply a particular accumulation function.

Cleaning Up in Sequence Expressions

It's common to implement sequence computations that access external resources such as databases but that return their results on demand. This raises a difficulty: how do you manage the lifetime of the resources for the underlying operating-system connections? One elegant solution is via `use` bindings in sequence expressions:

- When a `use` binding occurs in a sequence expression, the resource is initialized each time a client enumerates the sequence.
- The connection is closed when the client disposes of the enumerator.

For example, consider the following function that creates a sequence expression that reads the first two lines of a file on demand:

```
open System.IO

let firstTwoLines file =
    seq {use s = File.OpenText(file)
         yield s.ReadLine()
         yield s.ReadLine()}
```

Let's now create a file and a sequence that reads the first two lines of the file on demand:

```
> File.WriteAllLines("test1.txt", [|"Es kommt ein Schiff";
                                    "A ship is coming"|]);;
> let twolines() = firstTwoLines "test1.txt";;

val twolines : unit -> seq<string>
```

At this point, the file hasn't yet been opened, and no lines have been read from the file. If you now iterate the sequence expression, the file is opened, the first two lines are read, and the results are consumed from the sequence and printed. Most important, the file has now also been closed, because the `Seq.iter` aggregate operator is careful to dispose of the underlying enumerator it uses for the sequence, which in turn disposes of the file handle generated by `File.OpenText`:

```
> twolines() |> Seq.iter (printfn "line = '%s'")

line = 'Es kommt ein Schiff'
line = A ship is coming'
```

Expressing Some Operations Using Sequence Expressions

Sequences written using `map`, `filter`, `choose`, and `collect` can often be rewritten to use a sequence expression, as described in Chapter 3. For example, the `triangleNumbers` and `evenPositions` examples above can also be written as:

```
let triangleNumbers =
    [for i in 1 .. 10 do
         for j in 1 .. i do
             yield (i, j)]

let evenPositions =
    [for (i, j, v) in gameBoard do
         if v % 2 = 0 then
             yield (i, j)]
```

The output in each case is the same. In many cases, rewriting to use generative sequence expressions results in considerably clearer code. There are pros and cons to using sequence-expression syntax for some parts of queries:

- Sequence expressions are very good for the subset of queries expressed using iteration (`for`), filtering (`if/then`), and mapping (`yield`). They're particularly good for queries containing multiple nested `for` statements.
- Other query constructs, such as ordering, truncating, grouping, and aggregating, must be expressed directly using aggregate operators, such as `Seq.sortBy` and `Seq.groupBy`, or by using the more general notion of "query expressions" (see Chapter 14).
- Some queries depend on the index position of an item within a stream. These are best expressed directly using aggregate operators such as `Seq.mapi`.
- Many queries are part of a longer series of transformations chained by `|>` operators. Often, the type of the data being transformed at each step varies substantially through the chain of operators. These queries are best expressed using operator chains.

■ **Note:** F# also has a more general "query-expression" syntax that includes support for specifying grouping, aggregation, and joining operations. This is mostly used for querying external data sources and is discussed in Chapter 14.

Structure Beyond Sequences: Working with Trees

In Chapter 5, you learned about some simple techniques to represent record and tree-structured data in F# using *record types* and *union types*. In Chapter 8, you learned how to move from an unstructured, textual representation such as XML to a structured, tree-like structures representation of the input data. In the following sections, you will learn techniques for working with structured data, in particular tree structures associated with the representation of the abstract syntax of languages. In particular, you will look at some important recurring techniques in designing and working with *abstract syntax representations*.

Example: Abstract Syntax Representations

Let's look at the design of the abstract syntax type `Scene` from Listing 9-1. The type `Scene` uses fewer kinds of nodes than the concrete XML representation: the concrete XML has node kinds `Circle`, `Square`, `Composite`, and `Ellipse`, whereas `Scene` has just three (`Rect`, `Ellipse`, and `Composite`), with two derived constructors, `Circle` and `Square`, defined as static members of the `Scene`:

```
static member Circle(center:PointF,radius) =
    Ellipse(RectangleF(center.X-radius,center.Y-radius,
                       radius*2.0f,radius*2.0f))

/// A derived constructor
static member Square(left,top,side) =
    Rect(RectangleF(left,top,side,side))
```

This is a common step when abstracting from a concrete syntax; details are dropped and unified to make the abstract representation simpler and more general. Extra functions are then added that compute specific instances of the abstract representation. This approach has pros and cons:

- Transformational and analytical manipulations are almost always easier to program if you have fewer constructs in your abstract syntax representation.
- You must be careful not to eliminate truly valuable information from an abstract representation. For some applications, it may really matter if the user specified a `Square` or a `Rectangle` in the original input; for example, an editor for this data may provide different options for editing these objects.

The AST uses the types `PointF` and `RectangleF` from the `System.Drawing` namespace. This simplification is a design decision that should be assessed: `PointF` and `RectangleF` use 32-bit, low-precision, floating-point numbers, which may not be appropriate if you're eventually rendering on high-precision display devices. You should be wary of deciding on abstract representations on the basis of convenience alone, although of course this is useful during prototyping.

The lesson here is that you should look carefully at your abstract syntax representations, trimming out unnecessary nodes and unifying nodes where possible, but only as long as doing so helps you achieve your ultimate goals.

Common operations on abstract syntax trees include traversals that collect information and transformations that generate new trees from old. For example, the abstract representation from Listing 9-1 has the property that, for nearly all purposes, the `Composite` nodes are irrelevant (this wouldn't be the case if you added an extra construct, such as an `Intersect` node). This means you can flatten to a sequence of `Ellipse` and `Rectangle` nodes:

```
let rec flatten scene =
    seq { match scene with
          | Composite scenes -> for x in scenes do yield! flatten x
          | Ellipse _ | Rect _ -> yield scene }
```

Here, `flatten` is defined using sequence expressions, introduced in Chapter 3. Its type is:

```
val flatten : scene:Scene -> seq<Scene>
```

Let's look at this more closely. Recall from Chapter 3 that sequences are on-demand (lazy) computations. Using functions that recursively generate `seq<'T>` objects can lead to inefficiencies in your

code if your abstract syntax trees are deep. It's often better to traverse the entire tree in an *eager* way (eager traversals run to completion immediately). For example, it's typically faster to use an accumulating parameter to collect a list of results. Here's an example:

```
let rec flattenAux scene acc =
    match scene with
    | Composite(scenes) -> List.foldBack flattenAux scenes acc
    | Ellipse _
    | Rect _ -> scene :: acc

let flatten2 scene = flattenAux scene [] |> Seq.ofList
```

The following does an eager traversal using a local mutable instance of a `ResizeArray` as the accumulator and then returns the result as a sequence. This example uses a local function and ensures that the mutable state is locally encapsulated:

```
let flatten3 scene =
    let acc = new ResizeArray<_>()
    let rec flattenAux s =
        match s with
        | Composite(scenes) -> scenes |> List.iter flattenAux
        | Ellipse _ | Rect _ -> acc.Add s
    flattenAux scene;
    Seq.readonly acc
```

The types of these are:

```
val flatten2 : scene:Scene -> seq<Scene>
val flatten3 : scene:Scene -> seq<Scene>
```

There is no hard and fast rule about which of these is best. For prototyping, the second option—doing an efficient, eager traversal with an accumulating parameter—is often the most effective. Even if you implement an accumulation using an eager traversal, however, returning the result as an on-demand sequence still can give you added flexibility later in the design process.

Transforming Abstract Syntax Representations

In the previous section, you saw examples of accumulating traversals of a syntax representation. It's common to traverse abstract syntax in other ways:

- *Leaf rewriting (mapping)*: Translating some leaf nodes of the representation but leaving the overall shape of the tree unchanged
- *Bottom-up rewriting*: Traversing a tree but making local transformations on the way up
- *Top-down rewriting*: Traversing a tree, but before traversing each subtree, attempting to locally rewrite the tree according to some particular set of rules
- *Accumulating and rewriting transformations*: For example, transforming the tree left to right but accumulating a parameter along the way

For example, this mapping transformation rewrites all leaf ellipses to rectangles:

```
let rec rectanglesOnly scene =
    match scene with
    | Composite scenes -> Composite (scenes |> List.map rectanglesOnly)
    | Ellipse rect | Rect rect -> Rect rect
```

Often, whole classes of transformations are abstracted into aggregate transformation operations, taking functions as parameters. For example, here is a function that applies one function to each leaf rectangle:

```
let rec mapRects f scene =
    match scene with
    | Composite scenes -> Composite (scenes |> List.map (mapRects f))
    | Ellipse rect -> Ellipse (f rect)
    | Rect rect -> Rect (f rect)
```

The types of these functions are:

```
val rectanglesOnly : scene:Scene -> Scene
val mapRects : f:(RectangleF -> RectangleF) -> scene:Scene -> Scene
```

Here is a use of the `mapRects` function to adjust the aspect ratio of all the `RectangleF` values in the scene (`RectangleF` values support an `Inflate` method):

```
let adjustAspectRatio scene =
    scene |> mapRects (fun r -> RectangleF.Inflate(r, 1.1f, 1.0f / 1.1f))
```

Using On-Demand Computation with Abstract Syntax Trees

Sometimes it's feasible to delay loading or processing some portions of an abstract syntax tree. For example, imagine if the XML for the small geometric language from the previous section included a construct such as the following, in which the `File` nodes represent entire subtrees defined in external files:

```
<Composite>
     <File file='spots.xml'/>
     <File file='dots.xml'/>
</Composite>
```

It may be useful to delay loading these files. One general way to do this is to add a `Delay` node to the `Scene` type:

```
type Scene =
    | Ellipse   of RectangleF
    | Rect      of RectangleF
    | Composite of Scene list
    | Delay     of Lazy<Scene>
```

You can then extend the `extractScene` function of Listing 9-1 with the following case to handle this node:

```
let rec extractScene (node : XmlNode) =
    let attribs = node.Attributes
    let childNodes = node.ChildNodes
    match node.Name with
    | "Circle" ->
        ...
    | "File" ->
        let file = attribs.GetNamedItem("file").Value
        let scene = lazy (let d = XmlDocument()
                          d.Load(file)
                          extractScene(d :> XmlNode))
        Scene.Delay scene
```

Code that analyzes trees (for example, via pattern matching) must typically be adjusted to force the computation of delayed values. One way to handle this is to first call a function to eliminate immediately delayed values:

```
let rec getScene scene =
    match scene with
    | Delay d -> getScene (d.Force())
    | _ -> scene
```

Here is the function `flatten2` from the "Processing Abstract Syntax Representations" section, but redefined to first eliminate delayed nodes:

```
let rec flattenAux scene acc =
    match getScene(scene) with
    | Composite scenes -> List.foldBack flattenAux scenes acc
    | Ellipse _ | Rect _ -> scene :: acc
    | Delay _ -> failwith "this lazy value should have been eliminated by getScene"

let flatten2 scene = flattenAux scene []
```

It's generally advisable to have a single representation of laziness within a single syntax tree design. For example, the following abstract syntax design uses laziness in too many ways:

```
type SceneVeryLazy =
    | Ellipse of Lazy<RectangleF>
    | Rect of Lazy<RectangleF>
    | Composite of seq<SceneVeryLazy>
    | LoadFile of string
```

The shapes of ellipses and rectangles are lazy computations; each `Composite` node carries a `seq<SceneVeryLazy>` value to compute subnodes on demand, and a `LoadFile` node is used for delayed file loading. This is a bit of a mess, because a single `Delay` node would, in practice, cover all these cases.

■ **Note:** The `Lazy<'T>`type is defined in `System` and represents delayed computations. You access a lazy value via the `Value` property. F# includes the special keyword `lazy` for constructing values of this type. Chapter 8 also covered lazy computations.

Caching Properties in Abstract Syntax Trees

For high-performance applications of abstract syntax trees, it can occasionally be useful to cache computations of some derived attributes within the syntax tree itself. For example, let's say you want to compute bounding boxes for the geometric language described in Listing 9-1. It's potentially valuable to cache this computation at `Composite` nodes. You can use a type such as the following to hold a cache:

```
type SceneWithCachedBoundingBox =
    | Ellipse of RectangleF
    | Rect of RectangleF
    | CompositeRepr of SceneWithCachedBoundingBox list * RectangleF option ref
```

This is useful for prototyping, although you should be careful to encapsulate the code that is responsible for maintaining this information. Listing 9-1 shows the full code for doing this.

Listing 9-1. *Adding the cached computation of a local attribute to an abstract syntax tree*

```
type SceneWithCachedBoundingBox =
    | Ellipse of RectangleF
    | Rect of RectangleF
    | CompositeRepr of SceneWithCachedBoundingBox list * RectangleF option ref

    member x.BoundingBox =
        match x with
        | Ellipse rect | Rect rect -> rect
        | CompositeRepr (scenes, cache) ->
            match !cache with
            | Some v -> v
            | None ->
                let bbox =
                    scenes
                    |> List.map (fun s -> s.BoundingBox)
                    |> List.reduce (fun r1 r2 -> RectangleF.Union(r1, r2))
                cache := Some bbox
                bbox

    /// Create a Composite node with an initially empty cache
    static member Composite(scenes)  = CompositeRepr(scenes, ref None)
```

Other attributes that are sometimes cached include the hash values of tree-structured terms and the computation of all the identifiers in a subexpression. The use of caches makes it more awkward to pattern-match on terms. This issue can be largely solved by using active patterns, covered later in this chapter.

Memoizing Construction of Syntax Tree Nodes

In some cases, abstract syntax tree nodes can end up consuming significant portions of the application's memory budget. In this situation, it can be worth memoizing some or all of the nodes constructed in the tree. You can even go as far as memoizing *all* equivalent nodes, ensuring that equivalence between nodes can be implemented by pointer equality, a technique often called *hash-consing*. Listing 9-2 shows an abstract representation of propositional logic terms that ensures that any two nodes that are syntactically

identical are shared via a memoizing table. Propositional logic terms are terms constructed using *P* AND *Q*, *P* OR *Q*, NOT *P*, and variables *a*, *b*, and so on. A noncached version of the expressions is:

```
type Prop =
    | And of Prop * Prop
    | Or of Prop * Prop
    | Not of Prop
    | Var of string
    | True
```

Listing 9-2. *Memoizing the construction of abstract syntax tree nodes*

```
type Prop =
    | Prop of int

and PropRepr =
    | AndRepr of Prop * Prop
    | OrRepr of Prop * Prop
    | NotRepr of Prop
    | VarRepr of string
    | TrueRepr

open System.Collections.Generic

module PropOps =

    let internal uniqStamp = ref 0
    type internal PropTable() =
        let fwdTable = new Dictionary<PropRepr, Prop>(HashIdentity.Structural)
        let bwdTable = new Dictionary<int, PropRepr>(HashIdentity.Structural)
        member t.ToUnique repr =
            if fwdTable.ContainsKey repr then fwdTable.[repr]
            else let stamp = incr uniqStamp; !uniqStamp
                 let prop = Prop stamp
                 fwdTable.Add (repr, prop)
                 bwdTable.Add (stamp, repr)
                 prop
        member t.FromUnique (Prop stamp) =
            bwdTable.[stamp]

    let internal table = PropTable ()

    // Public construction functions
    let And (p1, p2) = table.ToUnique (AndRepr (p1, p2))
    let Not p = table.ToUnique (NotRepr p)
    let Or (p1, p2)  = table.ToUnique (OrRepr (p1, p2))
    let Var p = table.ToUnique (VarRepr p)
    let True = table.ToUnique TrueRepr
    let False = Not True
```

```
    // Deconstruction function
    let getRepr p = table.FromUnique p
```

You can construct terms using the operations in `PropOps` much as you would construct terms using the nonmemoized representation:

```
> open PropOps;;
> True;;

val it : Prop = Prop 1

> And (Var "x",Var "y");;

val it : Prop = Prop 5

> getRepr it;;

val it : PropRepr = AndRepr(Prop 3, Prop 4)

> And(Var "x",Var "y");;

val it : Prop = Prop 5
```

In this example, when you create two syntax trees using the same specification, `And (Var "x",Var "y")`, you get back the same `Prop` object with the same stamp `5`. You can also use memoization techniques to implement interesting algorithms; in Chapter 12, you see an important representation of propositional logic called a *binary decision diagram* (BDD) based on a memoization table similar to the previous example.

The use of unique integer stamps and a lookaside table in the previous representation also has some drawbacks; it's harder to pattern match on abstract syntax representations, and you may need to reclaim and recycle stamps and remove entries from the lookaside table if a large number of terms is created or if the overall set of stamps must remain compact. You can solve the first problem by using active patterns, covered next in this chapter. If necessary, you can solve the second problem by scoping stamps in an object that encloses the `uniqStamp` state, the lookaside table, and the construction functions. Alternatively, you can explicitly reclaim the stamps by using the `IDisposable` idiom described in Chapter 8, although this approach can be intrusive to your application.

Active Patterns: Views for Structured Data

Pattern matching is a key technique provided in F# for decomposing abstract syntax trees and other abstract representations of languages. So far in this book, all the examples of pattern matching have been directly over the core representations of data structures: for example, directly matching on the structure of lists, options, records, and discriminated unions. But pattern matching in F# is also *extensible*—that is, you can define new ways of matching over existing types. You do this through a mechanism called *active patterns*.

This book covers only the basics of active patterns. They can be indispensable, because they can let you continue to use pattern matching with your types even after you hide their representations. Active patterns also let you use pattern matching with .NET object types. The following section covers active patterns and how they work.

Converting the Same Data to Many Views

In high-school math courses, you were probably taught that you can view complex numbers in two ways: as rectangular coordinates $x + yi$ or as polar coordinates of a phase r and magnitude φ. In most computer systems, complex numbers are stored in the first format, although often the second format is more useful.

Wouldn't it be nice if you could look at complex numbers through either lens? You could do this by explicitly converting from one form to another when needed, but it would be better to have your programming language look after the transformations needed to do this for you. Active patterns let you do exactly that. First, here is a standard definition of complex numbers:

```
[<Struct>]
type Complex(r : float, i : float) =
    static member Polar(mag, phase) = Complex(mag * cos phase, mag * sin phase)
    member x.Magnitude = sqrt(r * r + i * i)
    member x.Phase = atan2 i r
    member x.RealPart = r
    member x.ImaginaryPart = i
```

Here is a pattern that lets you view complex numbers as rectangular coordinates:

```
let (|Rect|) (x : Complex) = (x.RealPart, x.ImaginaryPart)
```

Here is an active pattern to help you view complex numbers in polar coordinates:

```
let (|Polar|) (x : Complex) = (x.Magnitude, x.Phase)
```

The key thing to note is that these definitions let you use `Rect` and `Polar` as tags in pattern matching. For example, you can now write the following to define addition and multiplication over complex numbers:

```
let addViaRect a b =
    match a, b with
    | Rect (ar, ai), Rect (br, bi) -> Complex (ar + br, ai + bi)

let mulViaRect a b =
    match a, b with
    | Rect (ar, ai), Rect (br, bi) -> Complex (ar * br - ai * bi, ai * br + bi * ar)
```

As it happens, multiplication on complex numbers is easier to express using polar coordinates, implemented as:

```
let mulViaPolar a b =
    match a, b with
    | Polar (m, p), Polar (n, q) -> Complex.Polar (m * n, p + q)
```

Here is an example of using the (`|Rect|`) and (`|Polar|`) active patterns directly on some complex numbers via the pattern tags `Rect` and `Polar`. You first make the complex number 3+4*i*:

```
> fsi.AddPrinter (fun (c : Complex) -> sprintf "%gr + %gi" c.RealPart c.ImaginaryPart)
```

```
> let c = Complex (3.0, 4.0);;

val c : Complex = 3r + 4i

> c;;

val it : Complex = 3r + 4i

>  match c with
   | Rect (x, y) -> printfn "x = %g, y = %g" x y;;

x = 3, y = 4

>  match c with
   | Polar (x, y) -> printfn "x = %g, y = %g" x y;;

x = 5, y = 0.927295

> addViaRect c c;;

val it : Complex = 6r + 8i

> mulViaRect c c;;

val it : Complex = -7r + 24i

> mulViaPolar c c;;

val it : Complex = -7r + 24i
```

As you may expect, you get the same results if you multiply via rectangular or polar coordinates. The execution paths are quite different, however. Let's look closely at the definition of `mulViaRect`:

```
let mulViaRect a b =
    match a, b with
    | Rect (ar, ai), Rect (br, bi) ->
        Complex (ar * br - ai * bi, ai * br + bi * ar)
```

When F# needs to match the values `a` and `b` against the patterns `Rect (ar, ai)` and `Rect (br, bi)`, it doesn't look at the contents of `a` and `b` directly. Instead, it runs a function as part of pattern matching (which is why they're called *active* patterns). In this case, the function executed is (`|Rect|`), which produces a pair as its result. The elements of the pair are then bound to the variables `ar` and `ai`. Likewise, in the definition of `mulViaPolar`, the matching is performed partly by running the function (`|Polar|`).

The functions (`|Rect|`) and (`|Polar|`) are allowed to do anything, as long as each ultimately produces a pair of results. Here are the types of (`|Rect|`) and (`|Polar|`):

```
val ( |Rect| ) : x:Complex -> float * float
val ( |Polar| ) : x:Complex -> float * float
```

These types are identical, but they implement completely different views of the same data.

The definitions of `addViaRect` and `mulViaPolar` can also be written using pattern matching in argument position:

```
let add2 (Rect (ar, ai)) (Rect (br, bi)) = Complex (ar + br, ai + bi)
let mul2 (Polar (r1, th1)) (Polar (r2, th2)) = Complex (r1 * r2, th1 + th2)
```

Matching on .NET Object Types

One useful thing about active patterns is that they let you use pattern matching with existing.NET object types. For example, the .NET object type `System.Type` is a runtime representation of types in .NET and F#. Here are the members found on this type:

```
type System.Type with
    member IsGenericType : bool
    member GetGenericTypeDefinition : unit -> Type
    member GetGenericArguments : unit -> Type[]
    member HasElementType : bool
    member GetElementType : unit -> Type
    member IsByRef : bool
    member IsPointer : bool
    member IsGenericParameter : bool
    member GenericParameterPosition : int
```

This type looks very much like one you'd like to pattern match against. There are clearly three or four distinct cases here, and pattern matching helps you isolate them. You can define an active pattern to achieve this, as shown in Listing 9-3.

Listing 9-3. *Defining an active pattern for matching on* `System.Type` *values*

```
let (|Named|Array|Ptr|Param|) (typ : System.Type) =
    if typ.IsGenericType
    then Named(typ.GetGenericTypeDefinition(), typ.GetGenericArguments())
    elif typ.IsGenericParameter then Param(typ.GenericParameterPosition)
    elif not typ.HasElementType then Named(typ, [||])
    elif typ.IsArray then Array(typ.GetElementType(), typ.GetArrayRank())
    elif typ.IsByRef then Ptr(true, typ.GetElementType())
    elif typ.IsPointer then Ptr(false, typ.GetElementType())
    else failwith "MSDN says this can't happen"
```

This then lets you use pattern matching against a value of this type:

```
open System

let rec formatType typ =
    match typ with
    | Named (con, [||]) -> sprintf "%s" con.Name
    | Named (con, args) -> sprintf "%s<%s>" con.Name (formatTypes args)
    | Array (arg, rank) -> sprintf "Array(%d,%s)" rank (formatType arg)
    | Ptr(true, arg) -> sprintf "%s&" (formatType arg)
```

```
    | Ptr(false, arg) -> sprintf "%s*" (formatType arg)
    | Param(pos) -> sprintf "!%d" pos

and formatTypes typs =
    String.Join(",", Array.map formatType typs)
```

or collect the free generic type variables:

```
let rec freeVarsAcc typ acc =
    match typ with
    | Array (arg, rank) -> freeVarsAcc arg acc
    | Ptr (_, arg) -> freeVarsAcc arg acc
    | Param _ -> (typ :: acc)
    | Named (con, args) -> Array.foldBack freeVarsAcc args acc

let freeVars typ = freeVarsAcc typ []
```

Defining Partial and Parameterized Active Patterns

Active patterns can also be *partial*. You can recognize a partial pattern by a name such as `(|MulThree|_|)` and by the fact that it returns a value of type `'T option` for some `'T`. For example:

```
let (|MulThree|_|) inp = if inp % 3 = 0 then Some(inp / 3) else None
let (|MulSeven|_|) inp = if inp % 7 = 0 then Some(inp / 7) else None
```

Finally, active patterns can also be *parameterized*. You can recognize a parameterized active pattern by the fact that it takes several arguments. For example:

```
let (|MulN|_|) n inp = if inp % n = 0 then Some(inp / n) else None
```

The F# quotation API `Microsoft.FSharp.Quotations` uses both parameterized and partial patterns extensively.

Hiding Abstract Syntax Implementations with Active Patterns

Earlier in this chapter, you saw the following type that defines an optimized representation of propositional logic terms using a unique stamp for each syntactically unique term:

```
type Prop = Prop of int
and internal PropRepr =
    | AndRepr of Prop * Prop
    | OrRepr of Prop * Prop
    | NotRepr of Prop
    | VarRepr of string
    | TrueRepr
```

What happens, however, if you want to pattern match against values of type `Prop`? Even if you exposed the representation, all you would get is an integer, which you would have to look up in an internal table. You can define an active pattern for restoring matching on that data structure, as shown in Listing 9-4.

Listing 9-4. *Extending Listing 9-2 with an active pattern for the optimized representation*

```
module PropOps =
    ...
    let (|And|Or|Not|Var|True|) prop =
        match table.FromUnique prop with
        | AndRepr (x, y) -> And (x, y)
        | OrRepr (x, y) -> Or (x, y)
        | NotRepr x -> Not x
        | VarRepr v -> Var v
        | TrueRepr -> True
```

This code defines an active pattern in the auxiliary module `PropOps` that lets you pattern match against `Prop` values, despite the fact that they're using optimized unique-integer references under the hood. For example, you can define a pretty-printer for `Prop` terms as follows, even though they're using optimized representations:

```
open PropOps

let rec showProp precedence prop =
    let parenIfPrec lim s = if precedence  < lim then "(" + s + ")" else s
    match prop with
    | Or (p1, p2) -> parenIfPrec 4 (showProp 4 p1 + " || " + showProp 4 p2)
    | And (p1, p2) -> parenIfPrec 3 (showProp 3 p1 + " && " + showProp 3 p2)
    | Not p -> parenIfPrec 2 ("not " + showProp 1 p)
    | Var v -> v
    | True -> "T"
```

Likewise, you can define functions to place the representation in various normal forms. For example, the following function computes *negation normal form* (NNF), where all instances of NOT nodes have been pushed to the leaves of the representation:

```
let rec nnf sign prop =
    match prop with
    | And (p1, p2) ->
        if sign then And (nnf sign p1, nnf sign p2)
        else Or (nnf sign p1, nnf sign p2)
    | Or (p1, p2) ->
        if sign then Or (nnf sign p1, nnf sign p2)
        else And (nnf sign p1, nnf sign p2)
    | Not p ->
        nnf (not sign) p
    | Var _ | True ->
        if sign then prop else Not prop

let NNF prop = nnf true prop
```

The following demonstrates that two terms have equivalent NNF normal forms:

```
> let t1 = Not(And(Not(Var("x")), Not(Var("y"))));;

val t1 : Prop = Prop 8

> fsi.AddPrinter(showProp 5);;
> t1;;

val it : Prop = not (not x && not y)

> let t2 = Or(Not(Not(Var("x"))),Var("y"));;

val t2 : Prop = not (not x) || y

> (t1 = t2);;

val it : bool = false

> NNF t1;;

val it : Prop = x || y

> NNF t2;;

val it : Prop = x || y

> NNF t1 = NNF t2;;

val it : bool = true
```

Equality, Hashing, and Comparison for New Structured Data Types

Equality, Hashing, and Comparison

Many efficient algorithms over structured data are built on primitives that efficiently compare and hash representations of information. In Chapter 5, you saw a number of predefined generic operations, including generic comparison, equality, and hashing, accessed via functions such as:

```
val compare : 'T -> 'T -> int when 'T : comparison
val (=) : 'T -> 'T -> bool when 'T : equality
val (<) : 'T -> 'T -> bool when 'T : comparison
val (<=) : 'T -> 'T -> bool when 'T : comparison
val (>) : 'T -> 'T -> bool when 'T : comparison
val (>=) : 'T -> 'T -> bool when 'T : comparison
val min : 'T -> 'T -> 'T when 'T : comparison
```

```
val max : 'T -> 'T -> 'T when 'T : comparison
val hash : 'T -> int when 'T : equality
```

First, note that these are *generic* operations—they can be used on objects of many different types. This can be seen by the use of `'T` in the signatures of these operations. The operations take one or two parameters of the same type. For example, you can apply the `=` operator to two `Form` objects, or two `System.DateTime` objects, or two `System.Type` objects, and something reasonable happens. Some other important derived generic types, such as the immutable (persistent) `Set<_>` and `Map<_,_>` types in the F# library, also use generic comparison on their key type:

```
type Set<'T when 'T : comparison> = ...
type Map<'Key, 'Value when 'Key : comparison> = ...
```

These operations and types are all *constrained*, in this case by the `equality` and/or `comparison` constraints. The purpose of constraints on type parameters is to make sure the operations are used only on a particular set of types. For example, consider equality and ordered comparison on a `System.Windows.Forms.Form` object. Equality is permitted, because the default for nearly all .NET object types is reference equality:

```
let form1 = new System.Windows.Forms.Form()
let form2 = new System.Windows.Forms.Form()
form1 = form1    // true
form1 = form2    // false
```

Ordered comparison isn't permitted, however:

```
let form1 = new System.Windows.Forms.Form()
let form2 = new System.Windows.Forms.Form()
form1 <= form2
error FS0001: The type 'Windows.Forms.Form' does not support the 'comparison' constraint. For
example, it does not support the 'System.IComparable' interface
```

That's good! There is no natural ordering for form objects, or at least no ordering is provided by the .NET libraries.

Equality and comparison can work over the structure of types. For example, you can use the equality operators on a tuple only if the constituent parts of the tuple also support equality. This means that using equality on a tuple of forms is permitted:

```
> let form1 = new System.Windows.Forms.Form();;
> let form2 = new System.Windows.Forms.Form();;
> (form1, form2) = (form1, form2);;

val it : bool =  true

> (form1, form2) = (form2, form1);;

val it : bool = false
```

But using ordered comparison of a tuple isn't:

```
> (form1, "Data for Form1") <= (form2, " Data for Form2");;

error FS0001: The type 'Windows.Forms.Form' does not support the 'comparison' constraint. For
example, it does not support the 'System.IComparable' interface
```

Again, that's good—this ordering would be a bug in your code. Now, let's take a closer look at when `equality` and `comparison` constraints are satisfied in F#.

- The `equality` constraint is satisfied if the type definition doesn't have the `NoEquality` attribute, and any dependencies also satisfy the `equality` constraint.
- The `comparison` constraint is satisfied if the type definition doesn't have the `NoComparison` attribute, and the type definition implements `System.IComparable`, and any dependencies also satisfy the `comparison` constraint.

An `equality` constraint is relatively weak, because nearly all CLI types satisfy it. A `comparison` constraint is a stronger constraint, because it usually implies that a type must implement `System.IComparable`.

Asserting Equality, Hashing, and Comparison Using Attributes

These attributes control the comparison and equality semantics of type definitions:

- `StructuralEquality` and `StructuralComparison`: Indicate that a structural type must support equality and comparison
- `NoComparison` and `NoEquality`: Indicate that a type doesn't support equality or comparison
- `CustomEquality` and `CustomComparison`: Indicate that a structural type supports custom equality and comparison

Let's look at examples of these. Sometimes you may want to assert that a structural type must support structural equality, and you want an error at the definition of the type if it doesn't. Do this by adding the `StructuralEquality` or `StructuralComparison` attribute to the type:

```
[<StructuralEquality; StructuralComparison>]
type MiniIntegerContainer = MiniIntegerContainer of int
```

This adds extra checking. In the following example, the code gives an error at compile time—the type can't logically support automatic structural comparison, because one of the element types doesn't support ordered comparison:

```
[<StructuralEquality; StructuralComparison>]
type MyData = MyData of int * string * string * System.Windows.Forms.Form


   error FS1177: The struct, record or union type 'MyData' has the
   'StructuralComparison' attribute but the component type
   'System.Windows.Forms.Form' does not satisfy the 'comparison' constraint
```

Fully Customizing Equality, Hashing, and Comparison on a Type

Many types in the .NET libraries come with custom equality, hashing, and comparison implementations. For example, `System.DateTime` has custom implementations of these.

F# also allows you to define custom equality, hashing, and comparison for new type definitions. For example, values of a type may carry a unique integer tag that can be used for this purpose. In such cases, we recommend that you take full control of your destiny and define custom comparison and equality operations on your type. For example, Listing 9-5 shows how to customize equality, hashing (using the predefined `hash` function), and comparison based on a unique `stamp` integer value. The type definition includes an implementation of `System.IComparable` and overrides of `Object.Equals` and `Object.GetHashCode`.

Listing 9-5. *Customizing equality, hashing, and comparison for a record type definition*

```
/// A type abbreviation indicating we're using integers for unique stamps
/// on objects
type stamp = int

/// A structural type containing a function that can't be compared for equality
[<CustomEquality; CustomComparison>]
type MyThing =
    {Stamp : stamp;
     Behaviour : (int -> int)}

    override x.Equals(yobj) =
        match yobj with
        | :? MyThing as y -> (x.Stamp = y.Stamp)
        | _ -> false

    override x.GetHashCode() = hash x.Stamp
    interface System.IComparable with
      member x.CompareTo yobj =
          match yobj with
          | :? MyThing as y -> compare x.Stamp y.Stamp
          | _ -> invalidArg "yobj" "cannot compare values of different types"
```

The `System.IComparable` interface is defined in the .NET libraries:

```
namespace System

    type IComparable =
          abstract CompareTo : obj -> int
```

Recursive calls to compare subexpressions are processed using the functions:

```
val hash : 'T -> int when 'T : equality
val (=) : 'T -> 'T -> bool when 'T : equality
val compare : 'T -> 'T -> int when 'T : comparison
```

Listing 9-6 shows the same for a union type, this time using some helper functions.

Listing 9-6. *Customizing generic hashing and comparison on a union type*

```
let inline equalsOn f x (yobj : obj) =
    match yobj with
    | :? 'T as y -> (f x = f y)
    | _ -> false

let inline hashOn f x =  hash (f x)

let inline compareOn f x (yobj : obj) =
    match yobj with
    | :? 'T as y -> compare (f x) (f y)
    | _ -> invalidArg "yobj" "cannot compare values of different types"

type stamp = int

[<CustomEquality; CustomComparison>]
type MyUnionType =
    | MyUnionType of stamp * (int -> int)

    static member Stamp (MyUnionType (s, _)) = s

    override x.Equals y = equalsOn MyUnionType.Stamp x y
    override x.GetHashCode() = hashOn MyUnionType.Stamp x
    interface System.IComparable with
        member x.CompareTo y = compareOn MyUnionType.Stamp x y
```

Listing 9-6 also shows how to implement the `System.Object` method `GetHashCode`. This follows the same pattern as generic equality.

Finally, you can declare that a structural type should use reference equality:

```
[<ReferenceEquality>]
type MyFormWrapper = MyFormWrapper of System.Windows.Forms.Form * (int -> int)
```

There is no such thing as reference comparison (the object pointers used by .NET move around, so the ordering would change). You can implement that by using a unique tag and custom comparison.

Suppressing Equality, Hashing, and Comparison on a Type

You can suppress equality on an F# defined type by using the `NoEquality` attribute on the definition of the type. This means the type isn't considered to satisfy the equality constraint. Likewise, you can suppress comparison on an F# defined type by using the `NoComparison` attribute on the definition of the type:

```
[<NoEquality; NoComparison>]
type MyProjections =
    | MyProjections of (int * string) * (string -> int)
```

Adding these attributes to your library types makes client code safer, because it's less likely to inadvertently rely on equality and comparison over types for which these operations make no sense.

Customizing Generic Collection Types

Programmers love defining new generic collection types. This is done less often in .NET and F# programming than in other languages, because the F# and .NET built-in collections are so good, but it's still important.

Equality and comparison play a role here. For example, it's common to have collections in which some of the values can be indexed using hashing, compared for equality when searching, or compared using an ordering. For example, seeing a constraint on this signature on a library type would come as no surprise:

```
type Graph<'Node when 'Node : equality>() = ...
```

The presence of the constraint is somewhat reassuring, because the requirement on node types is made clearer. Sometimes it's also desirable to be able to compare entire containers: for example, to compare one set with another, one map with another, or one graph with another. Consider the simplest generic collection type of all, which holds only one element. You can define it easily in F#:

```
type MiniContainer<'T> = MiniContainer of 'T
```

In this case, this is a structural type definition, and F# infers that there is an equality and comparison dependency on `'T`. All done! You can use `MiniContainer<_>` with values of any type, and you can do equality and comparison on `MiniContainer` values only if the element type also supports equality and comparison. Perfect.

If `MiniContainer` is a class type or has customized comparison and equality logic, however, then you need to be more explicit about dependencies. You can declare dependencies by using the `EqualityConditionalOn` and `ComparisonConditionalOn` attributes on the type parameter. You should also use the operators `Unchecked.equals`, `Unchecked.hash`, and `Unchecked.compare` to process elements recursively. With these attributes, `MiniContainer<A>` satisfies the `equality` and `comparison` constraints if `A` satisfies these constraints. Here's a full example:

```
type MiniContainer<[<EqualityConditionalOn; ComparisonConditionalOn >]'T>(x : 'T) =
    member x.Value = x
    override x.Equals(yobj) =
        match yobj with
        | :? MiniContainer<'T> as y -> Unchecked.equals x.Value y.Value
        | _ -> false

    override x.GetHashCode() = Unchecked.hash x.Value

    interface System.IComparable with
      member x.CompareTo yobj =
          match yobj with
          | :? MiniContainer<'T> as y -> Unchecked.compare x.Value y.Value
          | _ -> invalidArg "yobj" "cannot compare values of different types"
```

■ **Note:** Be careful about using generic equality, hashing, and comparison on mutable data. Changing the value of a field may change the value of the hash or the results of the operation. It's normally better to use the operations on immutable data or data with custom implementations.

Tail Calls and Recursive Programming

In the previous section, you saw how to process tree-structured using recursive functions. In this section, you learn about important topics associated with programming with recursive functions: stack usage and tail-calls.

When F# programs execute, two resources are managed automatically, *stack*- and *heap*-allocated memory. Stack space is needed every time you call an F# function and is reclaimed when the function returns or when it performs a *tail call*. It's perhaps surprising that stack space is more limited than space in the garbage-collected heap. For example, on a 32-bit Windows machine, the default settings are such that each thread of a program can use up to 1MB of stack space. Because stack is allocated every time a function call is made, a very deep series of nested function calls causes a `StackOverflowException` to be raised. For example, on a 32-bit Windows machine, the following program causes a stack overflow when `n` reaches about 79000:

```
let rec deepRecursion n =
    if n = 1000000 then () else
    if n % 100 = 0 then
        printfn "--> deepRecursion, n = %d" n
    deepRecursion (n+1)
    printfn "<-- deepRecursion, n = %d" n
```

You can see this in F# Interactive:

```
> deepRecursion 0;;

--> deepRecursion, n = 0
...
--> deepRecursion, n = 79100
--> deepRecursion, n = 79200
--> deepRecursion, n = 79300
Process is terminated due to StackOverflowException
Session termination detected. Press Enter to restart.
```

Stack overflows are extreme exceptions, because it's often difficult to recover correctly from them. For this reason, it's important to ensure that the amount of stack used by your program doesn't grow in an unbounded fashion as your program proceeds, especially as you process large inputs. Furthermore, deep stacks can hurt in other ways; for example, the .NET garbage collector traverses the entire stack on every garbage collection. This can be expensive if your stacks are very deep.

Because recursive functions are common in F# functional programming, this may seem to be a major problem. There is, however, one important case in which a function call recycles stack space eagerly: a *tail call*. A tail call is any call that is the last piece of work done by a function. For example, Listing 9-7 shows the same program with the last line deleted.

Listing 9-7. *A simple tail-recursive function*

```
let rec tailCallRecursion n : unit =
    if n = 1000000 then () else
    if n % 100 = 0 then
        printfn "--> tailCallRecursion, n = %d" n
    tailCallRecursion (n+1)
```

The code now runs to completion without a problem:

```
> tailCallRecursion 0;;

...
--> tailCallRecursion, n = 999600
--> tailCallRecursion, n = 999700
--> tailCallRecursion, n = 999800
--> tailCallRecursion, n = 999900
```

When a tail call is made, the .NET Common Language Runtime can drop the current stack frame before executing the target function, rather than waiting for the call to complete. Sometimes this optimization is performed by the F# compiler. If the `n = 1000000` check were removed in the previous program, the program would run indefinitely. (Note that `n` would cycle around to the negative numbers, because arithmetic is unchecked for overflow unless you open the module `Microsoft.FSharp.Core.Operators.Checked`.)

Functions such as `tailCallRecursion` are known as *tail-recursive* functions. When you write recursive functions, you should check either that they're tail recursive or that they won't be used with inputs that cause them to recurse to an excessive depth. The following sections give some examples of techniques you can use to make your functions tail recursive.

Tail Recursion and List Processing

Tail recursion is particularly important when you're processing F# lists, because lists can be long and recursion is the natural way to implement many list-processing functions. For example, here is a function to find the last element of a list (this must traverse the entire list, because F# lists are pointers to the head of the list):

```
let rec last l =
    match l with
    | [] -> invalidArg "l" "the input list should not be empty"
    | [h] -> h
    | h::t -> last t
```

This function is tail recursive, because no work happens after the recursive call `last t`. Many list functions are written most naturally in non-tail-recursive ways, however. Although it can be a little annoying to write these functions using tail recursion, it's often better to use tail recursion than to leave the potential for stack overflow lying around your code. For example, the following function creates a list of length `n` in which every entry in the list is the value `x`:

```
let rec replicateNotTailRecursiveA n x =
    if n <= 0 then []
    else x :: replicateNotTailRecursiveA (n - 1) x
```

The problem with this function is that work is done after the recursive call. This becomes obvious when you write the function in this fashion:

```
let rec replicateNotTailRecursiveB n x =
    if n <= 0 then []
    else
        let recursiveResult = replicateNotTailRecursiveB (n - 1) x
        x :: recursiveResult
```

Clearly, a value is being constructed by the expression `x :: recursiveResult` after the recursive call `replicateNotTailRecursiveB (n-1) x`. This means that the function isn't tail recursive. The solution is to write the function using an *accumulating parameter*. This is often done by using an auxiliary function that accepts the accumulating parameter:

```
let rec replicateAux n x acc =
    if n <= 0 then acc
    else replicateAux (n - 1) x (x :: acc)

let replicate n x = replicateAux n x []
```

Here, the recursive call to `replicateAux` is tail recursive. Sometimes the auxiliary functions are written as inner recursive functions:

```
let replicate n x =
    let rec loop i acc =
        if i >= n then acc
        else loop (i + 1) (x :: acc)
    loop 0 []
```

The F# compiler optimizes inner recursive functions such as these to produce an efficient pair of functions that pass extra arguments as necessary.

When you're processing lists, accumulating parameters often accumulate a list in reverse order. This means a call to `List.rev` may be required at the end of the recursion. For example, consider this implementation of `List.map`, which isn't tail recursive:

```
let rec mapNotTailRecursive f inputList =
    match inputList with
    | [] -> []
    | h :: t -> (f h) :: mapNotTailRecursive f t
```

Here is an implementation that neglects to reverse the accumulating parameter:

```
let rec mapIncorrectAcc f inputList acc =
    match inputList with
    | [] -> acc            // whoops! Forgot to reverse the accumulator here!
    | h :: t -> mapIncorrectAcc f t (f h :: acc)

let mapIncorrect f inputList = mapIncorrectAcc f inputList []
```

```
> mapIncorrect (fun x -> x * x) [1; 2; 3; 4];;

val it : int list = [16; 9; 4; 1]
```

Here is a correct implementation:

```
let rec mapAcc f inputList acc =
    match inputList with
    | [] -> List.rev acc
```

```
    | h::t -> mapAcc f t (f h :: acc)

let map f inputList = mapAcc f inputList []
```

```
> map (fun x -> x * x) [1; 2; 3; 4];;

val it : int list = [1; 4; 9; 16]
```

Tail Recursion and Object-Oriented Programming

You often need to implement object members with a tail-recursive implementation. For example, consider this list-like data structure:

```
type Chain =
    | ChainNode of int * string * Chain
    | ChainEnd of string

    member chain.LengthNotTailRecursive =
        match chain with
        | ChainNode(_, _, subChain) -> 1 + subChain.LengthNotTailRecursive
        | ChainEnd _ -> 0
```

The implementation of `LengthNotTailRecursive` is *not* tail recursive, because the addition `1 +` applies to the result of the recursive property invocation. One obvious tail-recursive implementation uses a local recursive function with an accumulating parameter, as shown in Listing 9-8.

Listing 9-8. *Making an object member tail recursive*

```
type Chain =
    | ChainNode of int * string * Chain
    | ChainEnd of string

    // The implementation of this property is tail recursive.
    member chain.Length =
        let rec loop c acc =
            match c with
            | ChainNode(_, _, subChain) -> loop subChain (acc + 1)
            | ChainEnd _ -> acc
        loop chain 0
```

■ **Note:** The list-processing functions in the F# library module `Microsoft.FSharp.Collections.List` are tail recursive, except where noted in the documentation. Some of them have implementations that are specially optimized to take advantage of the implementation of the `list` data structure.

Tail Recursion and Processing Unbalanced Trees

This section considers tail-recursion problems that are much less common in practice but for which it's important to know the techniques to apply if required. The techniques also illustrate some important aspects of functional programming—in particular, an advanced technique called *continuation passing*.

Tree-structured data are generally more difficult to process in a tail-recursive way than list-structured data. For example, consider this tree structure:

```
type Tree =
    | Node of string * Tree * Tree
    | Tip of string

let rec sizeNotTailRecursive tree =
    match tree with
    | Tip _ -> 1
    | Node(_, treeLeft, treeRight) ->
        sizeNotTailRecursive treeLeft + sizeNotTailRecursive treeRight
```

The implementation of this function isn't tail recursive. Luckily, this is rarely a problem, especially if you can assume that the trees are *balanced*. A tree is balanced when the depth of each subtree is roughly the same. In that case, a tree of depth 1,000 will have about 21,000 entries. Even for a balanced tree of this size, the recursive calls to compute the overall size of the tree won't recurse to a depth greater than 1,000—not deep enough to cause stack overflow except when the routine is being called by some other function already consuming inordinate amounts of stack. Many data structures based on trees are balanced by design; for example, the `Set` and `Map` data structures implemented in the F# library are based on balanced binary trees.

Some trees can be unbalanced, however. For example, you can explicitly make a highly unbalanced tree:

```
let rec mkBigUnbalancedTree n tree =
    if n = 0 then tree
    else Node("node", Tip("tip"), mkBigUnbalancedTree (n - 1) tree)

let tree1 = Tip("tip")
let tree2 = mkBigUnbalancedTree 15000 tree1
let tree3 = mkBigUnbalancedTree 15000 tree2
let tree4 = mkBigUnbalancedTree 15000 tree3
let tree5 = mkBigUnbalancedTree 15000 tree4
let tree6 = mkBigUnbalancedTree 15000 tree5
```

Calling `sizeNotTailRecursive(tree6)` now risks a stack overflow (note that this may depend on the amount of memory available on a system; change the size of the tree to force the stack overflow). You can solve this in part by trying to predict whether the tree will be unbalanced to the left or the right and by using an accumulating parameter:

```
let rec sizeAcc acc tree =
    match tree with
    | Tip _ -> 1 + acc
    | Node(_, treeLeft, treeRight) ->
        let acc = sizeAcc acc treeLeft
        sizeAcc acc treeRight
```

```
let size tree = sizeAcc 0 tree
```

This algorithm works for `tree6`, because it's biased toward accepting trees that are skewed to the right. The recursive call that processes the right branch is a tail call, which the call that processes the left branch isn't. This may be OK if you have prior knowledge of the shape of your trees. This algorithm still risks a stack overflow, however, and you may have to change techniques. One way to do this is to use a much more general and important technique known as *continuation passing*.

Using Continuations to Avoid Stack Overflows

A continuation is a function that receives the result of an expression after it's been computed. Listing 9-9 shows an example implementation of the previous algorithm that handles trees of arbitrary size.

Listing 9-9. *Making a function tail recursive via an explicit continuation*

```
let rec sizeCont tree cont =
    match tree with
    | Tip _ -> cont 1
    | Node(_, treeLeft, treeRight) ->
        sizeCont treeLeft (fun leftSize ->
          sizeCont treeRight (fun rightSize ->
            cont (leftSize + rightSize)))

let size tree = sizeCont tree (fun x -> x)
```

What's going on here? Let's look at the type of `sizeCont` and `size`:

```
val sizeCont : tree:Tree -> cont:(int -> 'a) -> 'a
val size : tree:Tree -> int
```

The type of `sizeCont tree cont` can be read as "compute the size of the tree and call `cont` with that size." If you look at the type of `sizeCont`, you can see that it will call the second parameter of type `int -> 'T` at some point—how else could the function produce the final result of type `'T`? When you look at the implementation of `sizeCont`, you can see that it does call `cont` on both branches of the `match`.

Now, if you look at recursive calls in `sizeCont`, you can see that they're both tail calls:

```
sizeCont treeLeft (fun leftSize ->
  sizeCont treeRight (fun rightSize ->
    cont (leftSize + rightSize)))
```

That is, the first call to `sizeCont` is a tail call with a new continuation, as is the second. The first continuation is called with the size of the left tree, and the second is called with the size of the right tree. Finally, you add the results and call the original continuation `cont`. Calling `size` on an unbalanced tree such as `tree6` now succeeds:

```
> size tree6;;

val it : int = 50001
```

How did you turn a tree walk into a tail-recursive algorithm? The answer lies in the fact that continuations are function objects, which are allocated on the garbage-collected heap. Effectively, you've generated a work list represented by objects, rather than keeping a work list via a stack of function invocations.

As it happens, using a continuation for both the right and left trees is overkill, and you can use an accumulating parameter for one side. This leads to a more efficient implementation, because each continuation-function object is likely to involve one allocation (short-lived allocations such as continuation objects are very cheap but not as cheap as not allocating at all!). For example, Listing 9-10 shows a more efficient implementation.

Listing 9-10. *Combining an accumulator with an explicit continuation*

```
let rec sizeContAcc acc tree cont =
    match tree with
    | Tip _ -> cont (1 + acc)
    | Node (_, treeLeft, treeRight) ->
        sizeContAcc acc treeLeft (fun accLeftSize ->
        sizeContAcc accLeftSize treeRight cont)

let size tree = sizeContAcc 0 tree (fun x -> x)
```

The behavior of this version of the algorithm is:

1. You start with an accumulator `acc` of 0.
2. You traverse the left spine of the tree until a `Tip` is found, building up a continuation for each `Node` along the way.
3. When a `Tip` is encountered, the continuation from the previous `Node` is called with `accLeftSize` increased by 1. The continuation makes a recursive call to `sizeContAcc` for its right tree, passing the continuation for the second-to-last node along the way.
4. When all is done, all the left and right trees have been explored, and the final result is delivered to the (`fun x -> x`) continuation.

As you can see from this example, continuation passing is a powerful control construct, although it's used only occasionally in F# programming.

Another Example: Processing Syntax Trees

One real-world example where trees may become unbalanced is syntax trees for parsed languages when the inputs are very large and machine generated. In this case, some language constructs may be repeated very large numbers of times in an unbalanced way. For example, consider this data structure:

```
type Expr =
    | Add of Expr * Expr
    | Bind of string * Expr * Expr
    | Var of string
    | Num of int
```

This data structure would be suitable for representing arithmetic expressions of the forms *var*, *expr* + *expr*, and *bind var* = *expr in expr*. This chapter and Chapter 11 are dedicated to techniques for representing

and processing languages of this kind. As with all tree structures, most traversal algorithms over this type of abstract syntax trees aren't naturally tail recursive. For example, here is a simple evaluator:

```
type Env = Map<string, int>

let rec eval (env : Env) expr =
    match expr with
    | Add (e1, e2) -> eval env e1 + eval env e2
    | Bind (var, rhs, body) -> eval (env.Add(var, eval env rhs)) body
    | Var var -> env.[var]
    | Num n -> n
```

The recursive call `eval env rhs` isn't tail recursive. For the vast majority of applications, you never need to worry about making this algorithm tail recursive. Stack overflow may be a problem, however, if bindings are nested to great depth, such as in *bind v1 = (bind v2 = . . . (bind v1000000 = 1. . .)) in v1+v1*. If the syntax trees come from human-written programs, you can safely assume this won't be the case. If you need to make the implementation tail recursive, however, you can use continuations, as shown in Listing 9-11.

Listing 9-11. *A tail-recursive expression evaluator using continuations*

```
let rec evalCont (env : Env) expr cont =
    match expr with
    | Add (e1, e2) ->
        evalCont env e1 (fun v1 ->
        evalCont env e2 (fun v2 ->
        cont (v1 + v2)))
    | Bind (var, rhs, body) ->
        evalCont env rhs (fun v1 ->
        evalCont (env.Add(var, v1)) body cont)
    | Num n ->
        cont n
    | Var var ->
        cont (env.[var])

let eval env expr = evalCont env expr (fun x -> x)
```

■ **Note:** Programming with continuations can be tricky, and you should use them only when necessary, or use the F# `async` type as a way of managing continuation-based code. Where possible, abstract the kind of transformation you're doing on your tree structure (for example, a map, fold, or bottom-up reduction) so you can concentrate on getting the traversal right. In the previous examples, the continuations all effectively play the role of a *work list*. You can also reprogram your algorithms to use work lists explicitly and to use accumulating parameters for special cases. Sometimes this is necessary to gain maximum efficiency, because an array or a queue can be an optimal representation of a work list. When you make a work list explicit, the implementation of an algorithm becomes more verbose, but in some cases, debugging can become simpler.

Summary

This chapter covered some of the techniques you're likely to use in your day-to-day F# programming when working with *sequences* and *structured data*. Nearly all the remaining chapters use some of the techniques described in this chapter, and Chapter 12 goes deeper into symbolic programming based on structured-data programming techniques.

In the next chapter, you'll learn about programming techniques for another fundamental kind of data: programming with numeric data using structural and statistical methods.