PART I

Introduction to Functional Programming

- **CHAPTER 1:** A Look at Functional Programming History
- **CHAPTER 2:** Putting Functional Programming into a Modern Context

1

A Look at Functional Programming History

WHAT'S IN THIS CHAPTER?

- An explanation functional programming
- A look at some functional languages
- The relationship to object oriented programming

Functional programming has been around for a very long time. Many regard the advent of the language LISP, in 1958, as the starting point of functional programming. On the other hand, LISP was based on existing concepts, perhaps most importantly those defined by Alonzo Church in his lambda calculus during the 1930s and 1940s. That sounds highly mathematical, and it was — the ideas of mathematics were easy to model in LISP, which made it the obvious language of choice in the academic sector. LISP introduced many other concepts that are still important to programming languages today.

WHAT IS FUNCTIONAL PROGRAMMING?

In spite of the close coupling to LISP in its early days, functional programming is generally regarded a paradigm of programming that can be applied in many languages — even those that were not originally intended to be used with that paradigm. Like the name implies, it focuses on the application of functions. Functional programmers use functions as building blocks to create new functions — that's not to say that there are no other language elements available to them, but the function is the main construct that architecture is built from.

Referential transparency is an important idea in the realm of functional programming. A function that is referentially transparent returns values that depend only on the input parameters that are passed. This is in contrast to the basic ideas of imperative programming,

where program state often influences return values of functions. Both functional and imperative programming use the term *function*, but the mathematical meaning of the referentially transparent function is the one used in functional programming. Such functions are also referred to as pure functions, and are described as having no side effects.

It's often impossible to define whether a given programming language is a functional language or not. On the other hand, it is possible to find out the extent to which a language supports approaches commonly used in the functional programming paradigm — recursion, for example. Most programming languages generally support recursion in the sense that programmers can call into a particular function, procedure, or method from its own code. But if the compilers and/or runtime environments associated with the language use stack-based tracking of return addresses on jumps like many imperative languages do, and there are no optimizations generally available to help prevent stack overflow issues, then recursion may be severely restricted in its applications. In imperative languages, there are often specialized syntax structures to implement loops, and more advanced support for recursion is ignored by the language or compiler designers.

Higher order functions are also important in functional programming. Higher order functions are those that take other functions as parameters or return other functions as their results. Many programming languages have some support for this capability. Even C has a syntax to define a type of a function or, in C terms, to refer to the function through a function pointer. Obviously this enables C programmers to pass around such function pointers or to return them from other functions. Many C libraries contain functions, such as those for searching and sorting, that are implemented as higher order functions, taking the essential data-specific comparison functions as parameters. Then again, C doesn't have any support for anonymous functions — that is, functions created on-the-fly, in-line, like lambda expressions, or for related concepts such as closures.

Other examples of language capabilities that help define functional programming are explored in the following chapters in this book.

For some programmers, functional programming is a natural way of telling the computer what it should do, by describing the properties of a given problem in a concise language. You might have heard the saying that functional programming is more about telling computers what the problem is they should be solving, and not so much about specifying the precise steps of the solution. This saying is a result of the high level of abstraction that functional programming provides. Referential transparency means that the only responsibility of the programmer is the specification of functions to describe and solve a given set of problems. On the basis of that specification, the computer can then decide on the best evaluation order, potential parallelization opportunities, or even whether a certain function needs to be evaluated at all.

For some other programmers, functional programming is not the starting point. They come from a procedural, imperative, or perhaps object oriented background. There's much anecdotal evidence of such programmers analyzing their day-to-day problems, both the ones they are meant to solve by writing programs, and the ones they encounter while writing those programs, and gravitating toward solutions from the functional realm by themselves. The ideas of functional programming often provide very natural solutions, and the fact that you can arrive there from different directions reinforces that point.

FUNCTIONAL LANGUAGES

Functional programming is not language specific. However, certain languages have been around in that space for a long time, influencing the evolution of functional programming approaches just as much as they were themselves influenced by those approaches to begin with. The largest parts of this book contain examples only in C#, but it can be useful to have at least an impression of the languages that have been used traditionally for functional programming, or which have evolved since the early days with functional programming as a primary focus.

Here are two simple functions written in LISP:

```
(defun calcLine (ch col line maxp)
  (let
    ((tch (if (= col (- maxp line)) (cons ch nil) (cons 46 nil))))
    (if (= col maxp) tch (append (append tch (calcLine ch (+ col 1) line maxp)) tch))
    )
  )

(defun calcLines (line maxp)
  (let*
    ((ch (+ line (char-int #\A)))
      (l (append (calcLine ch 0 line maxp) (cons 10 nil)))
      )
    (if (= line maxp) l (append (append l (calcLines (+ line 1) maxp)) l))
    )
  )
```

The dialect used here is Common Lisp, one of the main dialects of LISP. It is not important to understand precisely what this code snippet does. A much more interesting aspect of the LISP family of dialects is the structure and the syntactic simplicity exhibited. Arguably, LISP's Scheme dialects enforce this notion further than Common Lisp, Scheme being an extremely simple language with very strong extensibility features. But the general ideas become clear immediately: a minimum of syntax, few keywords and operators, and obvious blocks. Many of the elements you may regard as keywords or other built-in structures — such as `defun` or `append` — are actually macros, functions, or procedures. They may indeed come out of the box with your LISP system of choice, but they are not compiler magic. You can write your own or replace the existing implementations. Many programmers do not agree that the exclusive use of standard round parentheses makes code more readable, but it is nevertheless easy to admire the elegance of such a basic system.

The following code snippet shows an implementation of the same two functions, the same algorithm, in the much newer language Haskell:

```
calcLine :: Int -> Int -> Int -> Int -> String
calcLine ch col line maxp =
  let tch = if maxp - line == col then [chr ch] else "." in
  if col == maxp
    then tch
```

```
    else tch ++ (calcLine ch (col+1) line maxp) ++ tch

calcLines :: Int -> Int -> String
calcLines line maxp =
  let ch = (ord 'A') + line in
  let l = (calcLine ch 0 line maxp) ++ "\n" in
  if line == maxp
    then l
    else l ++ (calcLines (line+1) maxp) ++ l
```

There is a very different style to the structure of the Haskell code. Different types of brackets are used to create a list comprehension. The `if...then...else` construct is a built-in, and the `++` operator does the job of appending lists. The type signatures of the functions are a common practice in Haskell, although they are not strictly required. One very important distinction can't readily be seen: Haskell is a strongly typed language, whereas LISP is dynamically typed. Because Haskell has extremely strong type inference, it is usually unnecessary to tell the compiler about types explicitly; they are known at compile time. There are many other invisible differences between Haskell and LISP, but that's not the focus of this book.

Finally, here's an example in the language Erlang, chosen for certain Erlang specific elements:

```
add(A, B) ->
    Calc = whereis(calcservice),
    Calc ! {self(), add, A, B},
    receive
        {Calc, Result} -> Result
    end.

mult(A, B) ->
    Calc = whereis(calcservice),
    Calc ! {self(), mult, A, B},
    receive
        {Calc, Result} -> Result
    end.


loop() ->
    receive
        {Sender, add, A, B} ->
            Result = A + B,
            io:format("adding: ~p~n", [Result]),
            Sender ! {self(), Result},
            loop();
        {Sender, mult, A, B} ->
            Result = A * B,
            io:format("multiplying: ~p~n", [Result]),
            Sender ! {self(), Result},
            loop();
        Other ->
            io:format("I don't know how to do ~p~n", [Other]),
            loop()
    end.
```

This is a very simple learning sample of Erlang code. However, it uses constructs pointing at the Actor model based parallelization support provided by the language and its runtime system. Erlang is not a very strict functional language — mixing in the types of side effects provided by `io:format` wouldn't be possible this way in Haskell. But in many industrial applications, Erlang has an important role today for its stability and the particular feature set it provides.

As you can see, functional languages, like imperative ones, can take many different shapes. From the very simplistic approach of LISP to the advanced syntax of Haskell or the specific feature set of Erlang, with many steps in between, there's a great spectrum of languages available to programmers who want to choose a language for its functional origins. All three language families are available today, with strong runtime systems, even for .NET in the case of the LISP dialect Clojure. Some of the ideas shown by those languages will be discussed further in the upcoming chapters.

THE RELATIONSHIP TO OBJECT ORIENTED PROGRAMMING

It is a common assumption that the ideas of functional programming are incompatible with those of other schools of programming. In reality, most languages available today are hybrid in the sense that they don't focus exclusively on one programming technique. There's no reason why they should, either, because different techniques can often complement one another.

Object oriented programming brings a number of interesting aspects to the table. One of them is a strong focus on encapsulation, combining data and behavior into classes and objects, and defining interfaces for their interaction. These ideas help object oriented languages promote modularization and a certain kind of reuse on the basis of the modules programmers create. An aspect that's responsible for the wide adoption object oriented programming languages have seen in mainstream programming is the way they allow modeling of real-world scenarios in computer programs. Many business application scenarios are focused on data storage, and the data in question is often related to physical items, which have properties and are often defined and distinguished by the way they interact with other items in their environments. As a result, object oriented mechanisms are not just widely applicable, but they are also easy to grasp.

When looking at a complicated industrial machine, for example, many programmers immediately come up with a way of modeling it in code as a collection of the wheels and cogs and other parts. Perhaps they consider viewing it as an abstract system that takes some raw materials and creates an end product. For certain applications, however, it may be interesting to deal with what the machine does on a rather abstract level. There may be measurements to read and analyze, and if the machine is complex enough, mathematical considerations might be behind the decisions for the parts to combine and the paths to take in the manufacturing process. This example can be abstractly extended toward any non-physical apparatus capable of generating output from input.

In reality, both the physical and the abstract viewpoints are important. Programming doesn't have a golden bullet, and programmers need to understand the different techniques at their disposal and make the decision for and against them on the basis of any problem with which they are confronted. Most programs have parts where data modeling is important, and they also have parts where algorithms are important. And of course they have many parts where there's no clear distinction,

where both data modeling and algorithms and a wide variety of other aspects are important. That's why so many modern programming languages are hybrid. This is not a new idea either — the first object oriented programming language standardized by ANSI was Common Lisp.

SUMMARY

Today's .NET platform provides one of the best possible constellations for hybrid software development. Originally a strong, modern and newly developed object oriented platform, .NET has taken major steps for years now in the functional direction. Microsoft F# is a fully supported hybrid language on the .NET platform, the development of which has influenced platform decisions since 2002. At the other end of the spectrum, albeit not all too far away, there's C#, a newly developed language strongly based in object orientation, that has been equally influenced by functional ideas almost from its invention. At the core of any program written in either language there's the .NET Framework itself, arguably the strongest set of underlying libraries that has ever been available for application development.