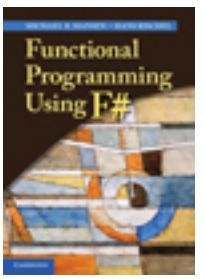

Functional Programming Using F#

Michael R. Hansen, Hans Rischel

Book DOI: http://dx.doi.org/10.1017/CBO9781139093996

Online ISBN: 9781139093996

Hardback ISBN: 9781107019027

Paperback ISBN: 9781107684065

Chapter

12 - Computation expressions pp. 279-310

Chapter DOI: http://dx.doi.org/10.1017/CBO9781139093996.013

Cambridge University Press

12

Computation expressions

A *computation expression* of F# provides the means to express a specific kind of computations in a way where low-level details are hidden and only visible through the use of special syntactic constructs like `yield`, `let!`, `return`, etc. These constructs are not part of the normal F# syntax and are only allowed inside computation expressions. Each kind of computation expression is defined by a *computation builder object* that contains the meaning of the special constructs.

A computation expression *ce* belonging to the builder object *comp* appears in the F# program in a construct of the form:

comp { *ce* }

This construct is an expression that evaluates to a value called a *computation*.

We have already seen examples of computation expressions in the form of sequence expressions with builder object `seq` (cf. Table 11.2) and query expressions with builder object `query` (cf. Section 11.8). The sequence computation expressions allow us to define computations on sequences without having to bother about the laziness and other implementation details, and the query expressions allow you to make database queries. In Chapter 13 we introduce asynchronous computation expressions with builder object `async` where you can define asynchronous computations without having to bother about low-level details in the current state of the system. The `seq`, `query` and `async` computation expressions are parts of F#.

This chapter describes the internals of computation expressions and how you may define your own builder objects. This allow you to define new kinds of computations with special state, flow of control and data management. The implementation of these special features is in the builder object only, and the low-level implementation details are therefore invisible at the computation-expression level.

The chapter explains and illustrates the machinery of computation expressions using a number of examples of builder objects. The builder object `mySeq` explains the use and meaning of the `yield` and `for` constructs using sequence expressions, while the `maybe` builder object explains the use and meaning of the `let!` and `return` constructs by using them to handle error cases in the evaluation of expressions represented by trees. This is followed by a description of some fundamental properties of the `For`, `Yield`, `Bind` and `Return` builder object methods that correspond to laws in the theory of monads – the mathematical foundation of computation expressions. The chapter ends with a presentation of monadic parsers. They provide a general method to construct programs capturing data from strings with a complex syntactic structure.

12.1 The agenda when defining your own computations

Defining a new kind of computation expressions comprises three parts:

- Defining the computation type *comp*<'a>.
- Defining the computation builder class *CompBuilderClass*.
- Declaring the computation builder object *comp*.

This may look as follows:

```
type comp<'a> = ...
...
type CompBuilderClass() =
  t.Bind(c: comp<'a>, f: 'a->comp<'b>): comp<'b> = ...
  t.Return(x: 'a): comp<'a> = ...
  ...
let comp = CompBuilderClass()
```

The members *comp*`.Bind` and *comp*`.Return` provide the meaning of `let!` and `return` and those constructs are therefore allowed in the computation expression *ce* in connection with expressions of the form *comp*`{`*ce*`}`. An overview of the technical setting is given in Section 12.4, where Table 12.2 presents the main syntactical constructs and their translations to composition of members from the associated builder object, while Table 12.3 gives the usual types of the members.

The same name (meta symbol *comp*) is usually used for the type of computations as well as for the builder object (an exception is the computation type `Async<'a>` with builder object `async`). This double use of the name causes no conflict as the type and the builder object are used in different contexts

By a *computation* we shall understand a value of type *comp*<'a> for some type 'a. The pragmatics behind the development of new kinds of computations is that they are like recipes in a cook book. A computation encapsulates pieces of programs but these pieces are only executed if the computation is *started*. In the same way, a recipe is usually an operational description of how to cook a dish; but the actual cooking is postponed until the recipe is started (used) by a cook.

Evaluating a computation expression, for example:

```
let c = comp { ... }
```

is like editing a recipe. The special syntactic constructs in Table 12.2 build combined computations in the same way as the *pour* "operator" builds a combined recipe in: "cook the vegetables, make a béchamel sauce and *pour* the sauce on the vegetables."

Each kind of computation expression has it own means for starting computations, for instance:

`seq<'a>`**:** Operations starting computations are, for example, indexing *sq*`.[`*n*`]` that gives a value of type 'a and the function `Seq.toList` that gives a value of type 'a `list`.

`Async<'a>`**:** An example is the function `Async.Start` starting the computation where the execution in the normal case will cause some side-effects and eventually terminate with a result of type 'a.

12.2 Introducing computation expressions using sequence expressions

In the introduction of sequence expressions in Section 11.6 it was shown that such expressions are more convenient to use in certain cases than the functions of the `Seq`-library. In this section we shall introduce the basic notions of computation expressions by showing how user-defined sequence expressions can be implemented.

Consider the sequence expression:

```
seq {for i in seq [1 .. 3] do
        for ch in seq ['a' .. 'd'] do
           yield (i,ch) };;
```

that expresses the sequence of pairs: (1,'a'), (1,'b'), . . ., (3,'d'). This sequence expression can be considered a *recipe* to get this sequence since the *evaluation* of the declaration:

```
let pairRecipe = seq {for i in seq [1 .. 3] do
                        for ch in seq ['a' .. 'd'] do
                           yield (i,ch) };;
val pairRecipe : seq<int * char>
```

will not cause any pair to be computed. Instead `pairRecipe` is bound to a *computation* (or a recipe) that can be *started* (or cooked) at a later stage, for example, when the last element is requested:

```
Seq.nth 11 pairRecipe;;
val it : int * char = (3, 'd')
```

We will now show how the outer `for`-construct above

```
for i in seq {1 .. 3} do ce(i)
```

can be expressed using functions from the `Seq` library, and the obtained insight will be used in the next section to implement our own computation expressions for sequences.

We denote the computation `seq {` $ce(\texttt{i})$ `}` corresponding to the body $ce(\texttt{i})$ by `f i`:

```
let f i = seq {for ch in seq ['a' .. 'd'] do
                  yield (i,ch) };;
val f : 'a -> seq<'a * char>
```

and we get:

```
f 1;;
val it : seq<int*char> = seq [(1,'a');(1,'b');(1,'c');(1,'d')]
f 2;;
val it : seq<int*char> = seq [(2,'a');(2,'b');(2,'c');(2,'d')]
f 3;;
val it : seq<int*char> = seq [(3,'a');(3,'b');(3,'c');(3,'d')]
```

The sequence (1,'a'), (1,'b'), . . ., (3,'d') of pairs denoted by `pairRecipe` is hence obtained by appending the three sequences `f 1`, `f 2` and `f 3`, and we get:

```
seq [(1,'a'); ... ;(3,'d')] = Seq.collect f (seq [1 .. 3])
```

where we have used the definition of `Seq.collect` in Table 11.1 on Page 254.

The meaning of the actual `for` construct can hence be expressed using `Seq.Collect`:

```
seq { for i in seq [1 .. 3] do ce(i) }
= Seq.collect f (seq [1 .. 3])
```

where:

```
f = fun i -> seq { ce(i) }
```

12.3 The basic functions: `For` and `Yield`

A new kind of computation expressions can be declared in F# through the declaration of a *builder* class, that implements a suitable selection of functions (cf. Section 12.1). We illustrate this concepts by defining a builder class for sequences. This builder class contains member functions that can perform operations on values of a parameterized type, that here is named `mySeq<'a>`:

```
type mySeq<'a> = seq<'a>;;
```

To be able to make a computation expression corresponding to the `pairRecipe` example, the builder class must provide implementations for the two functions:

```
For:     mySeq<'a> * ('a-> mySeq<'b>) -> mySeq<'b>
Yield:   'a -> mySeq<'a>
```

where `For` defines the meaning of the `for` construct and `Yield` defines the meaning of the `yield` construct, in the sense of the translations shown in Table 12.1.

Construct: C	**Translation**: $T(C)$
`for` x `in` e `do` ce	`For(`e`, fun` x `->` $T(ce)$`)`
`yield` e	`Yield(`e`)`

Table 12.1 *Translations for* `for` *and* `yield`

It was shown above that the `for` construct can be expressed using `Seq.collect`. Furthermore, the function `Yield`, with the type `'a -> mySeq<'a>` "lifts" an element to a sequence and, therefore, `Yield` a returns the singleton sequence just containing a. Hence, we arrive at the definitions:

$$\texttt{For}(sq, f) = \texttt{Seq.collect}\ f\ sq \tag{12.1}$$

$$\texttt{Yield}(a) = \texttt{Seq.singleton}\ a \tag{12.2}$$

The machinery is illustrated in Figure 12.1 on the `pairRecipe` example, where an ellipse represents a sequence and a dashed box represent a sequence obtained by concatenating the contained sequences. The figure illustrates the meaning of the `for` construct, where f is applied to each element of $i \in sq$. Each application $f(i)$ contribute with a part of the resulting sequence where the result is obtained by concatenation of the sequences:

$$f(1), f(2), f(3)$$

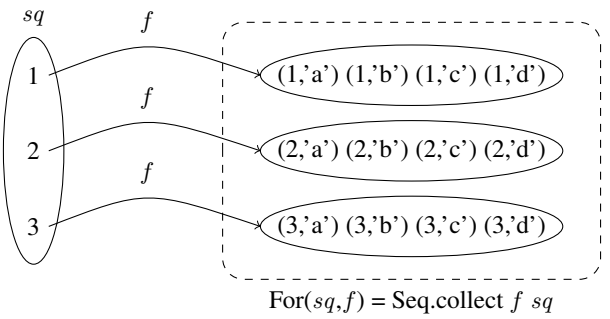

Figure 12.1 An example illustrating the definition of `For(`sq`,`f`)`

The declaration of a builder class is based directly on (12.1) and (12.2):

```
type MySeqClass() =
    member bld.Yield a: mySeq<'a> = Seq.singleton a
    member bld.For(sq:mySeq<'a>, f:'a -> mySeq<'b>):mySeq<'b>
              =  Seq.collect f sq;;
```

and the builder object is obtained as follows:

```
let mySeq = MySeqClass();;
```

We can now make our own computation expressions (limited to the `for` and `yield` constructs), for example, to declare a function that makes the *Cartesian* product of two sequences sqx and sqy by constructing a sequence containing all pairs (x, y) where x is a member of sqx and y is a member of sqy:

```
let cartesian sqx sqy = mySeq {for x in sqx do
                                  for y in sqy do
                                     yield (x,y) };;
val cartesian : mySeq<'a> -> mySeq<'b> -> mySeq<'a * 'b>
```

A declaration based on recursive functions or on the functions in the `Seq` library would not have a comparable simplicity. Try, for example, Exercise 11.10.

Using the translation in Table 12.1, this declaration is, behind the scene, translated to:

```
let cartesian sqx sqy =
   mySeq.For(sqx,
             fun x -> mySeq.For(sqy,
                                fun y -> mySeq.Yield (x,y)));;
```

We shall not go into further details here concerning how `mySeq` can be extended to capture more facilities of sequence expressions. Sequence expressions are handled in F# by a direct translation to composition of functions from the `Seq`-library and not by the use of a builder class. Further information can be found in the on-line documentation of F#.

Construct: C	**Translation**: $T(C)$
`do!` *e*	*comp*`.Bind(`*e*`, fun () ->` $T(ce)$`)`
ce	
`if` *e* `then` *ce*	`if` *e* `then` $T(ce)$ `else` *comp*`.Zero()`
`for` *pat* `in` *e* `do` *ce*	*comp*`.For(`*e*`, fun` *pat* `->` $T(ce)$`)`
`let!` *pat* `=` *e*	*comp*`.Bind(`*e*`, fun` *pat* `->` $T(ce)$`)`
ce	
`return` *e*	*comp*`.Return(`*e*`)`
`return!` *e*	*comp*`.ReturnFrom(`*e*`)`
`try` *ce* `finally` *e*	*comp*`.TryFinally` `(`*comp*`.Delay(fun () ->` $T(ce)$`), fun () ->` *e*`)`
`use` *pat* `=` *e*	*comp*`.Using(`*e*`, fun` *pat* `->` $T(ce)$`)`
ce	
`while` *e* `do` *ce*	*comp*`.While` `(fun () ->` *e*`,` *comp*`.Delay(fun () ->` $T(ce)$`))`
`yield` *e*	*comp*`.Yield(`*e*`)`
`yield!` *e*	*comp*`.YieldFrom(`*e*`)`
ce_1 `;` ce_2	*comp*`.Combine(`$T(ce_1)$`,` *comp*`.Delay(fun () ->` $T(ce_2)$`))`

Table 12.2 *Translation of selected syntactic forms inside comp*`{...}`

12.4 The technical setting when defining your own computations

Defining a builder class *CompBuilderClass* for a type *comp*`<'a>` of computations is like defining a (rather simple) programming language. The syntactical constructs, like `for` and `yield`, that are possible in expressions *comp* `{` *ce* `}` are given in advance; but the meaning of the wanted constructs must be provided in the class *CompBuilderClass* of the builder object *comp*. The meaning of an expression *ce* is given by a translation $T(ce)$, as described in Table 12.1, to a composition of the member functions of the builder object *comp*.

A selection of possible syntactical constructs for computation expressions are enumerated in Table 12.2. Meta symbols are used in this table as follows:

- ce, ce_1 and ce_2 denote computation expressions occurring inside the brackets in an expression *comp* `{ ... }`.
- e denotes an expression. That expression may have the form $comp'$ `{` ce' `}` and may be subject to further translations using the builder object $comp'$.
- pat denotes a pattern.

Usual types of the member functions are given in Table 12.3. The builder class *CompBuilderClass* must contain declarations of member functions such that the translation gives a correctly typed expression; but there are rather few restrictions beside that. One restriction is that the builder class *CompBuilderClass* must contain either the members `For` and `Yield` or the members `Bind` and `Return` (or all four member functions). These operations are the fundamental ones and we shall study their properties in Section 12.9.

We shall not do any attempt to cover all these constructs, but subsequent sections will describe `Bind`, `Return`, `ReturnFrom`, `Zero` and `Delay`.

The meaning of an expression *comp* `{` *ce* `}` is defined by a translation as follows:

$$comp\ \{\ ce\ \} = \begin{cases} comp\texttt{.Delay(fun () ->}\ T(ce)\texttt{)} & \text{if } \texttt{Delay} \text{ is defined} \\ T(ce) & \text{otherwise} \end{cases}$$

Member	Type(s)	Used in
`Bind`	*comp*<'a> * ('a -> *comp*<'b>) -> *comp*<'b>	defining `let!` and `do!`
`Combine`	*comp*<'a> * *comp*<'a> -> *comp*<'a> *comp*<unit> * *comp*<'a> -> *comp*<'a>	sequencing in computation expressions
`Delay`	(unit -> *comp*<'a>) -> *comp*<'a>	controlling side-effects
`For`	seq<'b> * ('b -> *comp*<'a>) -> *comp*<'a> seq<'b> * ('b -> *comp*<'a>) -> seq<'a>	defining `for...do`
`Return`	'a -> *comp*<'a>	defining `return`
`ReturnFrom`	*comp*<'a> -> *comp*<'a>	defining `return!`
`TryFinally`	*comp*<'a> * (unit -> unit) -> *comp*<'a>	defining `try...finally`
`Using`	'b * ('b -> *comp*<'a>) -> *comp*<'a> when 'a :> IDisposable	defining `use`-bindings
`While`	(unit -> bool) * *comp*<'a> -> *comp*<'a>	defining `while...do`
`Yield`	<'a> -> *comp*<'a>	defining `yield`
`YieldFrom`	*comp*<'a> -> *comp*<'a>	defining `yield!`
`Zero`	unit -> *comp*<'a>	empty `else`-branches

Table 12.3 *Usual types for members of a builder object comp*

Hence, the member function `Delay` provides an possibility to impose a delay from the very start of a computation expression. We shall have a closer look at this in Section 12.8.

12.5 Example: Expression evaluation with error handling

Consider the following type for expressions that are generated from integer constants and variables, using operators for addition and integer division:

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

An expression evaluates to a value in a given environment env : `Map<string,int>` that associates values with variables. Errors may, however, occur due to a division by zero or in the case that the environment does not give any value to a variable.

To avoid evaluations terminating by an exception we declare a function of type:

```
I: expr -> Map<string,int> -> option<int>
```

where

$$\texttt{I}\ e\ env = \begin{cases} \texttt{None} & \text{in case of errors, and} \\ \texttt{Some}\ v & \text{otherwise, where } v \text{ is the result of evaluating } e \text{ in } env. \end{cases}$$

The function can be declared as follows:

```
let I e env =
    let rec eval = function
        | Num i      -> Some i
        | Var x      -> Map.tryFind x env
        | Add(e1,e2) -> match (eval e1, eval e2) with
                        | (Some v1, Some v2) -> Some(v1+v2)
                        | _                  -> None
        | Div(e1,e2) -> match (eval e1, eval e2) with
                        | (_ , Some 0)       -> None
                        | (Some v1, Some v2) -> Some(v1/v2)
                        | _                  -> None
    eval e;;
```

Unfortunately, about half of this declaration addresses the manipulation of `option` values in a rather un-elegant manner. The declaration of I is dominated by the error-handling and the actual applications of the operators are almost put aside in a corner. We want to construct a `maybe<'a>` computation expression with an associated builder class `MaybeClass` that takes care of the Some/None distinguishing cases of the error handling.

12.6 The basic functions: `Bind`, `Return`, `ReturnFrom` and `Zero`

We first present a simple version of the type `maybe<'a>` to introduce the main concepts behind the members of the builder class. This first version will not allow delaying and activation of computations, but these concepts are included in subsequent versions of the builder class. A value of the type `maybe<'a>` is like a container possibly containing a value and the type is declared as follows:

```
type maybe<'a> = option<'a>;;
```

where the value `None` denotes an error situation – the container is empty – and `Some` v denotes that the container contains the value v.

It is not necessary to introduce `maybe<'a>` as a synonym for `option<'a>`; but in doing so we are following the convention introduced in Section 12.1 that the type and the builder object should have the same name.

We shall use expressions `maybe{`*ce*`}` where the constructs `let!`, `return`, `return!` and `if... then...` (no else clause) are used in the computational expression *ce*. To do so the builder class `MaybeClass` must contain declarations for the members:

```
Bind      :   maybe<'a> * ('a-> maybe<'b>) -> maybe<'b>
Return    :   'a -> maybe<'a>
ReturnFrom:   maybe<'a> -> maybe<'a>
Zero      :   unit -> maybe<'a>
```

where `Bind` defines the meaning of the `let!` construct, `Return` the meaning of the `return` construct, `ReturnFrom` the meaning of the `return!` construct, and `Zero` defines the meaning of the `if-then` construct in the sense of the translations shown in Table 12.4.

Construct: C	**Translation**: $T(C)$
`let!` x = e ce	`Bind`(e, `fun` x `->` $T(ce)$)
`return` e	`Return`(e)
`return!` e	`ReturnFrom`(e)
`if` e `then` ce	`if` e `then` $T(ce)$ `else Zero()`

Table 12.4 *Translations for* `let!`, `return`, `return!` *and* `if-then`

The operational readings of these constructs are as follows:

- The construct `let!` x = m `in` ce reads:
  - Bind x to the value in the container m (if this value exists) and
  - use this binding in ce.

  This construct is analogous to the `for` construct in the previous section.
- The construct `return` e reads: put e in a container.
  This construct is analogous to the `yield` construct in the previous section.
- The construct `return!` e reads: here is the container e.
  Note that e must be a container.
- The value `Zero()` is used in the case of an empty `else` clause.

With this intuition and the fact that `let!` x = m `in` ce is translated to `Bind`$(T(m), f)$ where $f =$ `fun` x `->` $T(ce)$, we arrive at the declaration:

```
type MaybeClass() =
    member bld.Bind(m:maybe<'a>, f:'a->maybe<'b>):maybe<'b> =
        match m with | None   -> None
                     | Some a -> f a
    member bld.Return a:maybe<'a> = Some a
    member bld.ReturnFrom m:maybe<'a> = m
    member bld.Zero():maybe<'a> = None;;
let maybe = MaybeClass();;
```

By use of expressions `maybe{`ce`}` the handling of error situations is managed by the `Bind` function and not in the handling of dyadic operators and divisions:

```
let I e env =
    let rec eval = function
        | Num i      -> maybe {return i}
        | Var x      -> maybe {return! Map.tryFind x env}
        | Add(e1,e2) -> maybe {let! v1 = eval e1
                               let! v2 = eval e2
                               return v1+v2}
        | Div(e1,e2) -> maybe {let! v2 = eval e2
                               if v2<>0 then
                                  let! v1 = eval e1
                                  return v1/v2}
    eval e;;
val I : expr -> Map<string,int> -> maybe<int>
```

Observe that the tags `None` and `Some` are absent from this program and that the declarations focus just on the computation of the value of an expression.

For example:

```
let e1 = Add(Div(Num 1, Num 0), Num 2);;
let e2 = Add(Add(Var "x", Var "y"), Num 2);;

let env = Map.ofList [("x",1);("y",2)];;

let v1 = I e1 env;;
val v1 : maybe<int> = None

let v2 = I e2 env;;
val v2 : maybe<int> = Some 5
```

The examples show that the `maybe` computation expressions eagerly evaluate `e1` and `e2` to values of type `maybe<int>`. Hence the computation expressions for this simple version of the class `MaybeClass` are not real recipes, they actually correspond to cooked dishes. This deficiency is repaired in the next section.

12.7 Controlling the computations: Delay and Start

The evaluation of an expression e can be *delayed* by "packing" it into a closure:

```
fun () -> e
```

as we already have seen in Section 11.3. For example:

```
let c1 = fun () -> 1+2;;
val c1 : unit -> int
```

The addition operation is *started* by a function application:

```
c1();;
val it : int = 3
```

We use another definition of the type `maybe<'a>`:

```
type maybe<'a> = unit -> option<'a>;;
```

in order to be able to control the delay and activation of `maybe` computation expressions. A value of this type is a recipe in the shape of a closure, and the delaying of computations is "hard coded" into the type. Such recipes must be started explicitly when the value of the computation is asked for. The following functions for delay and start will be used:

```
let delay v = fun () -> v;;
val delay : 'a -> unit -> 'a

let start m = m();;
val start : (unit -> 'a) -> 'a
```

The builder class `MaybeClass` is revised to get another meaning of the `let!` construct. The new definition of `Bind` captures that the construct

```
let! x = m in ce
```

matches the following operational reading:

1. Start the computation m.
2. Bind x to the value a of this computation if it terminates properly with a in the container.
3. Use this binding in the recipe ce.

Notice that the `let!` construct translates to `Bind`(m, f) where f is `fun x -> T(ce)`.

These considerations lead to the first revised version of the builder class:

```
type MaybeClass() =
    member bld.Bind(m:maybe<'a>, f:'a->maybe<'b>):maybe<'b> =
              match start m with
                  | None   -> delay None
                  | Some a -> f a
    member bld.Return a:maybe<'a> = delay(Some a)
    member bld.ReturnFrom v:maybe<'a> = delay v
    member bld.Zero():maybe<'a> = delay None;;

let maybe = MaybeClass();;
```

The declaration for `Bind` matches the operational reading of the `let!` construct. Delays of the option-values are needed to lift these values to the type `maybe`.

Notice that the declaration of `I` *e env* can be based on this revised `maybe` builder without any change. Doing so gives controlled computations in the sense of recipes:

```
let v1 = I e1 env;;
val v1 : maybe<int>

let v2 = I e2 env;;
val v2 : maybe<int>
```

The recipes `v1` and `v2` must be started to get values computed:

```
start v1;;
val it : int option = None

start v2;;
val it : int option = Some 5
```

Since an expression like `maybe { let! x = m ... }` translates to `Bind(m,...)` the computation `m` will actually be started (check the declaration of `Bind`) and the values `v1` and `v2` contain in this sense partly cooked ingredients. This can be observed if side effects are introduced into, for example, in the clause where addition is treated:

```
...
| Add(e1,e2) ->
    maybe {let! v1 = eval e1
           let! v2 = eval e2
           return (printfn "v1: %i  v2: %i" v1 v2 ; v1+v2)}
...
```

The result of executing the following declarations with this version of `maybe`:

```
let v2 = I e2 env;;
v1: 1  v2: 2
v1: 3  v2: 2
val v2 : maybe<int>

start v2;;
val it : int option = Some 5
```

shows that the computation is started and active until the outermost `return` or `return!` statement is reached.

12.8 The basic function: `Delay`

The builder class `CompBuilderClass` of a type `comp<'a>` with builder object `comp` may contain a member:

```
Delay: (unit -> comp<'a>) -> comp<'a>
```

The translation of an expression `comp{`*ce*`}` will then use this delay function in the translation of a computation expression *ce*:

`comp {` *ce* `}` translates to `comp.Delay(fun () ->` *T*`(`*ce*`))`

This gives a possibility to enforce a delay from the very start of a computation expression: We add the following declaration of `Delay` to the `MaybeClass` declaration in the previous section:

```
type MaybeClass() =
    ... As above from Bind to Zero ...
    member bld.Delay f:maybe<'a> = fun () -> start (f());;
```

The effect of this can be observed using the above "side-effect example," where the printing of the two lines with values to be added move from the declaration of `v2` to its activation:

```
let v2 = I e2 env;;
val v2 : maybe<'a>

start v2;;
v1: 1  v2: 2
v1: 3  v2: 2
val it : int option = Some 5
```

Hence, with the introduction of the `Delay` member in the class declaration, the expressions of the form `maybe{`*ce*`}` will denote genuine recipes. These recipes are expressed in an operational manner by describing *how* to cook the dish; but the actual cooking is delayed until the recipe is started.

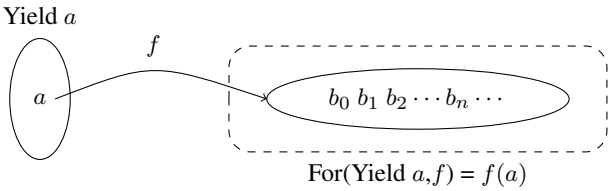

Figure 12.2 An illustration of the law: For(Yield a,f) = $f(a)$

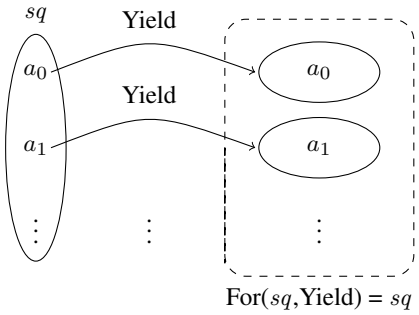

Figure 12.3 An illustration of the law: For(sq, Yield) = sq

12.9 The fundamental properties of `For` and `Yield`, `Bind` and `Return`

When declaring builder classes like `MySeqClass` and `MaybeClass` the only restriction imposed on `For` and `Yield` and `Bind` and `Return` is that they should have the correct types. But there are some laws that meaningful implementations should obey. These laws originate from the theory of *monads* for functional programming, a theory that provides the mathematical foundation for computation expressions.

The intuition behind these laws will be presented using the builder class for `mySeq<'a>` as example, where values of this type are considered as *containers* for values of type `'a` and computation expressions are recipes for filling containers. But the laws are not biased towards the `mySeq` computation expression builder.

The laws for `For` *and* `Yield`

The first law expresses that Yield is a kind of left unit for For:

$$\text{For}(\text{Yield}\ a, f) = f(a) \tag{12.3}$$

Hence binding f to the element of the container yielded by a is the same as applying f to a. This is illustrated in Figure 12.2.

The second law expresses that Yield is a right unit for For:

$$\text{For}(sq, \text{Yield}) = sq \tag{12.4}$$

Hence yielding the elements of a container sq equals that container. This is illustrated in Figure 12.3.

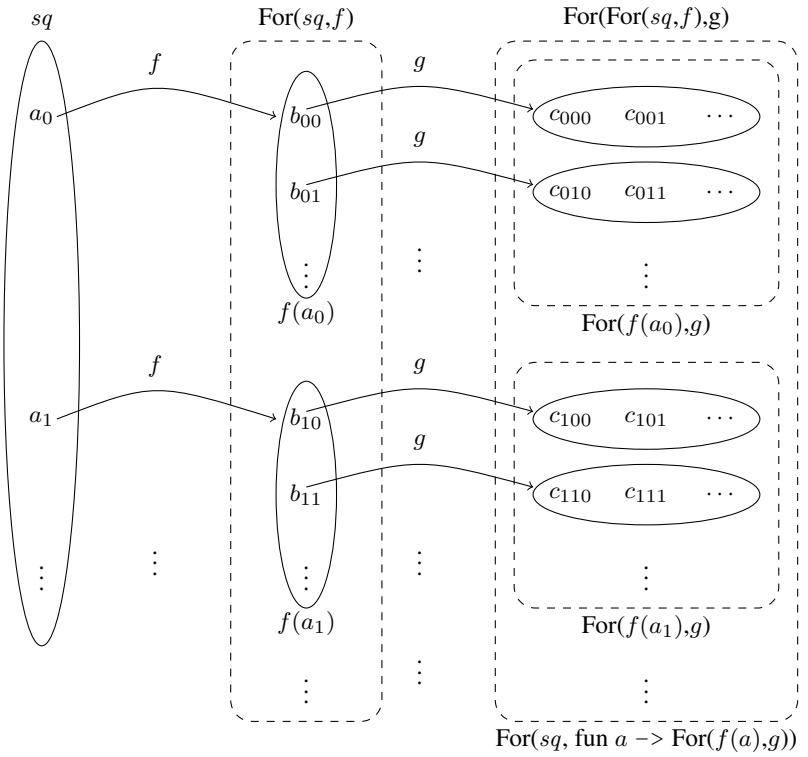

Figure 12.4 The law: For(For(sq,f),g) = For(sq, fun a -> For($f(a)$,g))

The last law expresses a kind of associativity of `For`:

$$\text{For}(\text{For}(sq, f), g) = \text{For}(sq, \text{fun } a \text{ -> } \text{For}(f(a), g)) \tag{12.5}$$

This law is explained in terms of the `for` construct. Observe first that the computation expression `for` a `in` sq `do` $f(a)$ translates as follows:

$$T(\texttt{for } a \texttt{ in } sq \texttt{ do } f(a)) \;=\; \text{For}(sq, \texttt{fun } a \texttt{ -> } f(a)) \;=\; \text{For}(sq, f)$$

Using this technique we arrive at the following alternative formulation of the law:

```
for b in (for a in sq do f(a)) do
    g(b)
```

is equivalent to

```
for a in sq do
    for b in f(a) do
        g(b)
```

The law expresses two ways of filling a container. The left-hand-side way is by filling it using $g(b)$ where b is in the container obtained from `for` a `in` sq `do` $f(a)$). The right-hand-side way is by filling it using $g(b_{ij})$ where b_{ij} is in the container obtained from $f(a_i)$, where a_i is in the container sq. This is illustrated in Figure 12.4.

The laws for `Bind` *and* `Return`

We shall now have a closer look at the resemblance between `Bind` and `Return` and `For` and `Yield`, respectively. Actually the types for `For` and `Yield` may be considered special cases of the type of `Bind` and `Return`, and it is possible to define computation expressions for sequences that make use of `let!` and `return` rather than `for` and `yield`.

To illustrate this, consider the following (not recommendable) declarations:

```
type myStrangeSeq<'a> = seq<'a>;;

type MyStrangeSeqClass() =
    member bld.Return a: myStrangeSeq<'a> =
        Seq.singleton a
    member bld.Bind(sqs, f):myStrangeSeq<'b> =
        Seq.collect f sqs;;

let myStrangeSeq = MyStrangeSeqClass();;
```

The `pairRecipe` example can now be given in the following way:

```
let pairRecipe = myStrangeSeq {let! i = seq {1..3}
                               let! ch = seq {'a'..'d'}
                               return (i,ch)};;
val pairRecipe : myStrangeSeq<int * char>

Seq.toList pairRecipe;;
val it : (int * char) list =
  [(1, 'a'); (1, 'b'); (1, 'c'); (1, 'd');
   (2, 'a'); (2, 'b'); (2, 'c'); (2, 'd');
   (3, 'a'); (3, 'b'); (3, 'c'); (3, 'd')]
```

Therefore, the fundamental laws `Bind` and `Return` are the same as those for `For` and `Yield`:

$$\text{Bind}(\text{Return}\ a, f) = f(a) \tag{12.6}$$

$$\text{Bind}(m, \text{Return}) = m \tag{12.7}$$

$$\text{Bind}(\text{Bind}(m, f), g) = \text{Bind}(m, \text{fun}\ a \text{ -> } \text{Bind}(f(a), g)) \tag{12.8}$$

It is left as an exercise to justify that these properties hold for the `maybe` example.

12.10 Monadic parsers

This section is a continuation of Section 10.2 where regular expressions were used in capturing data from text lines. Functions using match on regular expressions with capturing groups give an efficient capture of data with a simple structure. The capability of such functions is, however, limited and we had, for example, to use ad hoc tricks in Section 10.2 to capture data with a nested list structure. Data with a *recursive structure* appearing, for example, in representing *expressions* to be input by the program cannot be handled in this way.

Techniques to construct *parsers* to solve such a problem are well-known in compiler technology (as described, for example, in [2]). In this section we use a technique known as monadic parsing that has been developed by the "Haskell community" (cf. [15, 14, 7]). It gives a simple construction of parsers that is well-suited for small-scaled parsing – and it gives an interesting example of computation expressions. Large-scaled parsing used for instance in compilers is made using a parser generator (cf. [12, 13]). The following presentation of monadic parsers in F# is based on the Haskell implementation described in [7].

The first step in constructing a parser is to make a *grammar* describing the structure of the input data. This comprises making regular expressions to describe the *tokens*, i.e. the smallest pieces of information: names, numbers, operators and delimiters to be captured by *token parsers* constructed from the regular expressions. The combination of the token parsers to get a parser is then based directly on the grammar. We define a number of *parser combinators* to be used in this construction. The definition of the parser will usually consist of a set of mutually recursive declarations.

We use two examples to illustrate the technique: the simple example of person data from Section 10.2 and the more complicated example of algebraic expressions. In the first example we may, for example, get an input line of the form:

```
John 35 2 Sophie 27 Richard 17 89 3
```

and the captured value should then be:

```
[("John", [35;2]); ("Sophie", [27]); ("Richard", [17;89;3])]
```

of type

```
(string * (int list)) list
```

In the second example we have algebraic expressions like the string:

```
-a1 + 2 * (a2 - 3)
```

We want to capture this string as the value:

```
Add (Neg (Var "a1"), Mul (Num 2, Sub (Var "a2", Num 3)))
```

of type

```
type Expr = | Num of int | Var of string
            | Neg of Expr | Add of Expr * Expr
            | Sub of Expr * Expr | Mul of Expr * Expr;;
```

The captured value corresponds to the expression tree in Figure 12.5. This example has a number of interesting features beside the recursion: two levels of operator precedence (multiplication and addition operators), a precedence level with two operators (+ and −), and use of the same operator symbol (−) with two different meanings as prefix and infix operator.

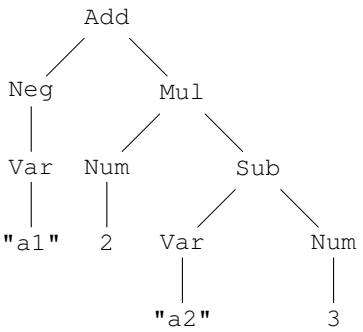

Figure 12.5 Expression tree for expression `-a1 + 2 * (a2 - 3)`

Grammars

In the first example we have two tokens `name` and `number` with regular expressions:

```
open System.Text.RegularExpressions;;
let nameReg   = Regex @"\G\s*([a-zA-Z][a-zA-Z0-9]*)";;
let numberReg = Regex @"\G\s*([0-9]+)";;
```

where the tokens `name` and `number` denote the set of strings matching the corresponding regular expressions `nameReg` and `numberReg`.

We shall capture structured data through rules described by *context-free grammars*. For the person data, the rules should capture strings with the wanted syntax:

- A personData string consists of a series of zero or more person strings.
- A person string consists of a `name` followed by a series of zero or more `numbers`.

The corresponding context-free grammar looks as follows:

$$\begin{array}{lcl} \text{personData} & ::= & \text{personList} \\ \text{personList} & ::= & \Lambda \mid \text{person personList} \\ \text{person} & ::= & \texttt{name numberList} \\ \text{numberList} & ::= & \Lambda \mid \texttt{number numberList} \end{array}$$

This grammar has the following components:

- Four *non-terminal symbols*: personData, personList, person and numberList.
- Two *tokens*: `name` and `number`.
- Four *rules*. Each rule has a non-terminal at the left-hand side and a *definition* at the right-hand side. A definition may have one or more *choices* separated by | and each choice consists of a possibly empty sequence of tokens and non-terminal symbols. The Greek letter Λ is used to denote the empty sequence in grammars.

In order to be able to distinguish tokens from non-terminal symbols in a grammar, we write a token in teletype font (like in `name` and `number`) while a non-terminal symbol is written in roman font (like in person and numberList).

Each non-terminal symbol denotes a *syntax class* that is the set of all strings that can be generated from that non-terminal symbol. A string is generated from a non-terminal symbol by a *derivation* that repeatedly replaces a non-terminal symbol by a choice in its definition or a token by a matching string. The derivation terminates when there are no more non-terminals or tokens to be substituted.

For example, the string `Peter 5 John` belongs to the syntax class personData due to the derivation:

$$
\begin{array}{ll}
 & \text{personData} \\
\Rightarrow & \text{personList} \\
\Rightarrow & \text{person personList} \\
\Rightarrow & \texttt{name}\ \text{numberList personList} \\
\Rightarrow & \texttt{"Peter"}\ \text{numberList personList} \\
\Rightarrow & \texttt{"Peter"}\ \texttt{number}\ \text{numberList personList} \\
\Rightarrow & \texttt{"Peter 5"}\ \text{numberList personList} \\
\Rightarrow & \texttt{"Peter 5"}\ \Lambda\ \text{personList} \\
\Rightarrow & \texttt{"Peter 5"}\ \text{person personList} \\
\Rightarrow & \texttt{"Peter 5"}\ \texttt{name}\ \text{numberList personList} \\
\Rightarrow & \texttt{"Peter 5 John"}\ \text{numberList personList} \\
\Rightarrow & \texttt{"Peter 5 John"}\ \Lambda\ \text{personList} \\
\Rightarrow & \texttt{"Peter 5 John"}\ \Lambda \\
\Rightarrow & \texttt{"Peter 5 John"}
\end{array}
$$

While a derivation generates a string from a non-terminal symbol, we are interested in the parsing of a string, that is, the creation of a derivation on the basis of a given string. The technique for monadic parsing to be presented will make this derivation on the basis of recursive definitions following the structure of the grammar, and for this approach to be well-defined the grammar must satisfy that there is no derivation of the form:

$$N \Rightarrow \cdots \Rightarrow Nw$$

where N is a non-terminal symbol and w is a sequence of tokens, non-terminal symbols and strings. In particular, there should be no *left-recursive rule* of the form:

$$N \quad ::= \quad N\ w$$

in the grammar. See Exercises 12.3 and 12.4.

We shall use grammars written in EBNF notation (extended Backus Naur form) allowing, for example, use of the repetition operator $\star$ that was introduced in connection with regular expressions (cf. Section10.2). Using the EBNF notation the above grammar gets a more compact form:

$$
\begin{array}{lll}
\text{personData} & ::= & \text{person}\star \\
\text{person} & ::= & \texttt{name number}\star
\end{array}
$$

In the second example with expressions we have tokens `num`, `var`, `addOp`, `mulOp`, `sign`, `leftPar`, `rightPar`, and `eos` where `eos` denotes end of string. The corresponding regular expressions are as follows:

```
let numReg        = Regex @"\G\s*((?:\053|-|)\s*[0-9]+)";;
let varReg        = Regex @"\G\s*([a-zA-z][a-zA-Z0-9]*)";;
let plusMinReg    = Regex @"\G\s*(\053|\055)";;
let addOpReg      = plusMinReg;;
let signReg       = plusMinReg;;
let mulOpReg      = Regex @"\G\s*(\052)";;
let leftParReg    = Regex @"\G\s*(\050)";;
let rightParReg   = Regex @"\G\s*(\051)";;
let eosReg        = Regex @"\G(\s*)$";;
```

The cautious reader will observe that the `addOp` and `sign` tokens have the same regular expressions. We will comment on this in the subsection on token parsers.

We choose syntax classes expr, term and factor to express the composite forms in expressions and we get at the following grammar for expressions, where the rule for factor has a number of different choices:

$$\begin{array}{lll} \text{expr} & ::= & \text{term}\ (\texttt{addOp}\ \text{term})\star \\ \text{term} & ::= & \text{factor}\ (\texttt{mulOp}\ \text{factor})\star \\ \text{factor} & ::= & \texttt{num} \mid \texttt{var} \mid \texttt{sign}\ \text{factor} \mid \texttt{leftPar}\ \text{expr}\ \texttt{rightPar} \end{array}$$

Building a grammar of this kind requires careful considerations of the following syntactic issues:

- Precedence levels of operators.
- Left or right association of operators.

The example with expressions has two levels of precedence of operators:

1. Multiplication operator $\star$.
2. Addition operators $+$ and $-$.

while all operators associate to the left. The precedence rules of operators are captured in the grammar:

- A factor can be used as a term.
- A term cannot be used as a factor – it has to be viewed as an expression and then enclosed in parentheses in order to get a factor.

The associations of the operators are capture in the structure of the constructs expressing repetitions, and the rule for term would, for example, have been:

$$\text{term} \quad ::= \quad (\text{factor}\ \texttt{mulOp})\star\ \text{factor}$$

if the multiplication operator should associate to the right.

The original grammars (in the Algol 60 report) used *left recursion* in the grammar of expressions, like in the following:

$$\begin{array}{lll} \text{expr} & ::= & \text{term} \mid \text{expr}\ \texttt{addOp}\ \text{term} \\ \text{term} & ::= & \text{factor} \mid \text{term}\ \texttt{mulOp}\ \text{factor} \\ \text{factor} & ::= & \texttt{num} \mid \texttt{var} \mid \texttt{sign}\ \text{factor} \mid \texttt{leftPar}\ \text{expr}\ \texttt{rightPar} \end{array}$$

The left recursion appears in the rule where expr can be expanded to expr `addOp` term and similar for term. This grammar has the (theoretical) advantage that the steps in the derivation of an expression correspond to the steps in building the expression tree. Such grammars can, however, *not* be used in our method because the corresponding parser will enter an infinite loop. See Exercise 12.4.

Parsers

Each character in the input string is identified by its *position* that is an non-negative integer. A *parser* with *result type* `'a` scans the string searching for matches starting a specified *start position pos*. A *match* is a pair (a, pos') of a *captured value* a of type `'a` and the *end position* pos' where a succeeding parsing may take over. The parser is hence "consuming" the characters from position *pos* up to (but not including) the end position pos' in producing the result a. The collection of all possible matches starting at a specified position can hence be represented by a list:

$$[(a_0, pos_0);\ (a_1, pos_1);\ \ldots;\ (a_{n-1}, pos_{n-1})]$$

This corresponds to the following type of a parser with result type `'a`:

```
type parser<'a> = string -> int -> ('a * int) list;;
```

Note that we allow several possible matches. This is not a complication – it is actually a key feature in monadic parsers. An empty list indicates that the parser has failed to find any matches. Suppose, for example, that we have a parser `expr` for algebraic expressions. Parsing the input string `"-a1 + 2 * (a2 - 3)"` from position `0`, using the expression `expr "-a1 + 2 * (a2 - 3)" 0`, should then give a list with three matches:

```
[(Neg (Var "a1"), 3);
 (Add (Neg (Var "a1"),Num 2), 7);
 (Add (Neg (Var "a1"),Mul (Num 2,Sub (Var "a2",Num 3))),18)]
```

Position 3 is just after `"-a1"`, position 7 is just after `"2"` while position 18 is at the end of the string.

Token parsers

Tokens are parsed using *token parsers*. We consider two kinds of token parsers:

1. A token parser with captured data (normally to be converted).
2. A token parser without relevant captured data.

A token parser of the first kind is made using the `token` function. The regular expression `reg` must contain a capturing group. The function `conv` converts the captured data:

```
open TextProcessing;;

let token (reg: Regex) (conv: string -> 'a) : parser<'a> =
    fun str pos ->
        let ma = reg.Match(str,pos)
```

```
        match ma.Success with
            | false -> []
            | _     ->
                let pos2 = pos + ma.Length
                [( conv(captureSingle ma 1), pos2)];;
val token : (Regex -> (string->'a) -> parser<'a>) = <fun:...>
```

Token parsers without captured data are made using the `emptyToken` function. The regular expression need not contain any capturing group and there are no conversion function. The parser captures the dummy value `()` of type `unit` and its function is solely to recognize and "consume" the data matching the regular expression:

```
let emptyToken (reg: Regex) : parser<unit> =
    fun str pos ->
        let ma = reg.Match(str,pos)
        match ma.Success with
        | false -> []
        | _     -> let pos2 = pos + ma.Length
                   [( (), pos2)];;
val emptyToken : (Regex -> parser<unit>) = <fun:clo...>
```

Note that the function `captureSingle` from the `TextProcessing` library (cf. Table 10.4 and Appendix B) is used in the above declarations.

Token parsers in the examples

We declare token parsers `name` and `number` in the first example using the corresponding regular expressions:

```
let name   = token nameReg id;;
let number = token numberReg int;;
```

The conversion function is the pre-defined identity function `id` for the `name` token parser because the captured string should be used literally "as is". The `number` token parser uses the conversion function `int` to convert the captured string of digits to an integer. The token parsers `num`, `var`, `sign`, `addOp`, `mulOp`, `leftPar` and `rightPar` in the second example should give values that can be used directly in building the expression tree (like the tree shown in Figure 12.5). The token parser `addOp` should hence capture a "value" that can be used to join two sub-trees, for example:

```
fun x y -> Add(x,y)
```

of type:

```
Expr -> Expr -> Expr
```

when parsing the character '+'. The `addOp` token parser will hence be of type:

```
parser<Expr->Expr->Expr>
```

These token parsers use the following conversion functions:

```
let numFct (str: string) = Num (int str);;
let varFct = Var;;
let addOpFct = function
    | "+" -> fun x y -> Add(x,y)
    |  _  -> fun x y -> Sub(x,y);;
let mulOpFct _ = fun x y -> Mul(x,y);;
let signFct = function
    | "+" ->  id
    |  _  ->  fun x -> Neg x;;
```

and their declarations are as follows:

```
let num      = token numReg numFct;;
let var      = token varReg varFct;;
let addOp    = token addOpReg addOpFct;;
let mulOp    = token mulOpReg mulOpFct;;
let sign     = token signReg signFct;;
let leftPar  = emptyToken leftParReg;;
let rightPar = emptyToken rightParReg;;
let eos      = emptyToken eosReg;;
```

The cautious reader will observe that the token parsers `addOp` and `sign` parse the *same* strings – with *different* captures. This works in monadic parsing because the parsing is strictly *top-down*: The `expr` parser (to be constructed later) acts according to the context and calls the `addOp` token parser when an addition operator may occur – and the `sign` token parser when a sign change operator may occur.

Computation expressions for building parsers

The computation expressions aim at simplifying the construction of a parser by hiding all the technicalities concerning the character positions in the input string. The key point in defining the `parser` computation expressions is to define the `Bind` member in the builder class that provides the meaning to the computation expression:

```
let! a = p
ce(a)
```

where `p:parser<'a>` is a parser giving parses of type `'a` and the computation expression $ce($`a`$)$ has type `parser<'b>`. The construct is translated to `Bind(p,`f`)`, where f is `fun a ->` $T(ce($`a`$))$ using the translation T described in Section 12.4.

The operational reading of this construct is:

1. Start the parser `p`,
2. bind `a` to a result a of the parser `p`, and
3. use this binding in the computation expression $ce($`a`$)$.

The parser `p` is activated by a function application `p` str pos, where str is the input string and pos is a position. This resembles the activation of a `maybe` value on Page 288. This activation of `p` gives a list of pairs $[(a_0, pos_0), \ldots, (a_n, pos_n)]$, where a_i is a captured value

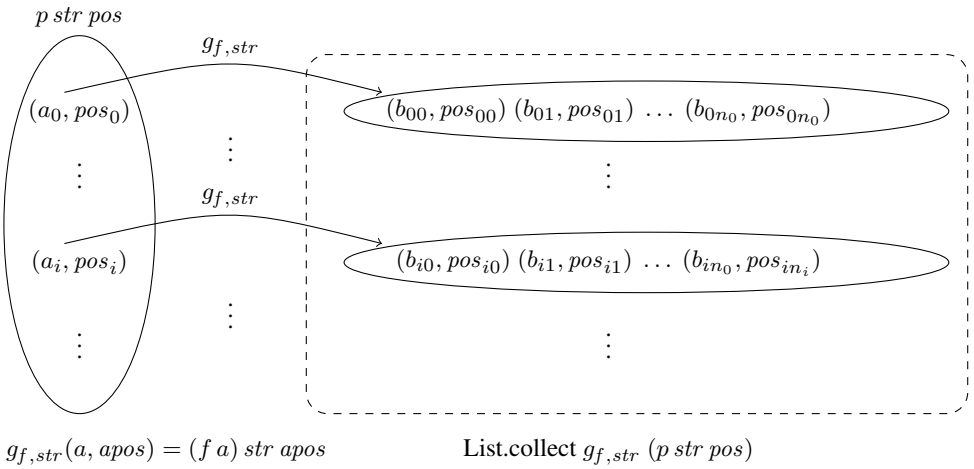

Figure 12.6 Illustrating `Bind(p,f) = fun str pos -> collect` $g_{f,str}\ (p\ str\ pos)$

for the part of the input string that starts at position pos and ends at $pos_i - 1$. Application of f to a_i yields a parser that must be activated to the input string str and the start position pos_i for that parser. This resembles the definition of the `for` construct for sequences, see Figure 12.1. This complete process is illustrated in Figure 12.6. Notice that the `Bind` function takes care of all the data management concerning the positions.

These ideas lead to the following computation expression class and builder object:

```
type ParserClass() =
  member t.Bind(p: parser<'a>, f: 'a->parser<'b>):parser<'b> =
    fun str pos ->
      List.collect (fun (a,apos) -> f a str apos) (p str pos)
  member bld.Zero() = (fun _ _ -> []): parser<'a>
  member bld.Return a = (fun str pos -> [(a,pos)]): parser<'a>
  member bld.ReturnFrom (p: parser<'a>) = p;;

let parser = ParserClass();;
```

The zero result `bld.Zero()` is the parser without any matches.

The "constant" parser (corresponding to the right-hand side of `bld.Return`):

```
parser { return a }
```

"captures" the value `a` without consuming any characters, that is, it gives the value `a` at the end position of the previously used parser.

When a string cannot be parsed the final result is the empty list; but no informative error report is handled by the builder class. In Exercise 12.8 you are asked make new builder class for parsers that takes care of a simple error handling.

Sequencing of parsers. Parsers for fixed forms

Assume that we have n parsers

$$p_1 : \texttt{parser<'}a_1\texttt{>}$$
$$p_2 : \texttt{parser<'}a_2\texttt{>}$$
$$\ldots$$
$$p_n : \texttt{parser<'}a_n\texttt{>}$$

and a function:

$$F\texttt{: '}a_1 \texttt{ * '}a_2 \texttt{ * ... * '}a_n \texttt{ -> '}b$$

for some types $'a_1, 'a_2, \ldots, 'a_n, 'b$.

Any match of the *sequenced parser* starting from position *pos* is then obtained by getting a sequence of contiguous matches (starting from position *pos*):

$$(a_1, pos_1), (a_2, pos_2), \ldots, (a_n, pos_n)$$

of the parser $p_1, p_2, \ldots, p_n$ and applying the function F to the captured values to get a match of the sequenced parser (starting from position *pos*):

$$(\ F(a_1, a_2, \ldots, a_n),\ pos_n\ )$$

Using computation expression the sequenced parser can be written:

```
parser { let! a1 = p1
         let! a2 = p2
         ...
         let! an = pn
         return F(a1, a2, ..., an) }
```

The `return` expression is inserted at a place where activation of the parsers $p_1, p_2, \ldots, p_n$ have already consumed all relevant characters and where it only remains to return the value $F(a_1, a_2, \ldots, a_n)$ without consuming any further characters.

Sequencing of parsers is used when building parsers for *fixed forms* (containing no part that is repeated an unspecified number of times). The simplest examples are parsers built using the `pairOf` combinator:

```
let pairOf p1 p2 = parser {let! x1 = p1
                           let! x2 = p2
                           return (x1,x2)};;
val pairOf : (parser<'a> -> parser<'b> -> parser<'a*'b>) = ...
```

We may, for instance, combine the `name` and `number` token parsers using `pairOf`:

```
let nameNumber = pairOf name number;;
val nameNumber : parser<string * int>

nameNumber " abc 473 " 0;;
val it : ((string * int) * int) list = [(("abc", 473), 8)]
```

In building a parser in the first example we will use the `pairOf` parser combinator to combine parsers of a name and of a list of numbers.

One may define a `tripleOf` combinator in a similar way, but it is not of much use as most grammars require specially built parsers for their sequencing constructs. In the expression example we have, for instance, the form of an *expression enclosed in parentheses*. A simplified version of this is a form with a variable enclosed in parentheses like:

```
( abc)
```

A parser for this form can be obtained by sequencing the token parsers `leftPar`, `var` and `rightPar`:

```
let varInPars = parser {let! _ = leftPar
                        let! x = var
                        let! _ = rightPar
                        return x };;
val varInPars : parser<expr>

varInPars "( abc) " 0;;
val it : (expr * int) list = [(Var "abc", 6)]
```

The recursive declaration of the `expr` parser given later in this section comprises a sequencing of the parsers `leftPar`, `expr` and `rightPar`.

The choice combinator

The choice combinator `<|>` defines the choice parser:

```
p1 <|> p2
```

for two parsers of same type:

```
p1; parser<'a>
p2: parser<'a>
```

The set of matches of `p1 <|> p2` is simply the *union* of the set of matches of `p1` and the set of matches of `p2`.

This is captured in the following declaration where the lists of matches of `p2` is appended to the list of matches of `p1`:

```
let (<|>) (p1: parser<'a>) (p2: parser<'a>) =
    (fun str pos -> (p1 str pos) @ (p2 str pos)): parser<'a>;;
val ( <|> ) : parser<'a> -> parser<'a> -> parser<'a>
```

We may for instance make a parser capturing a variable or a number:

```
let numOrVar = num <|> var;;
val numOrVar : parser<expr>

numOrVar "ab 35" 0;;
val it : (expr * int) list = [(Var "ab", 2)]
numOrVar "ab 35" 2;;
val it : (expr * int) list = [(Num 35, 5)]
```

Repetitive constructs. Combinators `listOf`*,* `infixL` *and* `infixR`

The `listOf` combinator is used when parsing lists where the parser `p` captures a single list element:

```
let rec listOf p = parser {return []}
                   <|> parser {let! x  = p
                               let! xs = listOf p
                               return x::xs};;
val listOf : parser<'a> -> parser<'a list>
```

The parser `listOf p` will either return an empty list:

```
parser {return []}
```

or parse one list element:

```
let! x  = p
```

and put this element in front of the remaining list and return the result:

```
let! xs = listOf p
return x::xs
```

The parser `listOf number` will for instance parse lists of numbers:

```
(listOf number) " 3 5 7 " 0;;
val it : (int list * int) list =
  [([], 0); ([3], 2); ([3; 5], 4); ([3; 5; 7], 6)]
```

The `infixL` combinator is used when building a parser for a syntactic form where an arbitrary number of operands are intermixed with infix operators that are on the same precedence level and associates to the left.

As an example we consider strings like

```
a - b + 5
```

where numbers or variables are intermixed with addition operators (+ or −). A parser for this form can be obtained using the below defined `infixL` operator:

```
let psL = numOrVar |> infixL addOp numOrVar;;
val psL : parser<expr>

psL "a - b + 5" 0;;
val it : (expr * int) list =
  [(Var "a", 1); (Sub (Var "a",Var "b"), 5);
   (Add (Sub (Var "a",Var "b"),Num 5), 9)]
```

The three matches in the result correspond to the strings:

String	Captured value
”a”	`Var "a"`
”a - b”	`Sub(Var "a", Var "b")`
”a - b + 5”	`Add(Sub(Var "a", Var "b"), Num 5)`

Note that the last expression tree:

```
Add(Sub(Var "a", Var "b"), Num 5)
```

of the full string `"a - b + 5"` reflects the *left association* of the operators: *first* we subtract `b` from `a` and *afterwards* we add `5` to the result.

The `infixL` combinator is defined in a slightly more general setting with two *operand parsers* `p` and `q`:

```
p : parser<'a>
q : parser<'b>
```

and an *operator parser* `op`:

```
op: parser<'a -> 'b -> 'a>
```

corresponding to strings with one operand *astr* matching `p`, *n* operators $op_1, \ldots, op_n$ matching `op`, and *n* operands $bstr_1, \ldots, bstr_n$ matching `q`:

$$astr \quad op_1 \quad bstr_1 \quad op_2 \quad bstr_2 \quad \ldots \quad bstr_{n-1} \quad op_n \quad bstr_n$$

The parser `p |> infixL op q` should have $n + 1$ matches on this string corresponding to a sequence of matches of `p`, `op` and `q`:

String	Capture	Captures of `p |> infixL op q`
astr	`p` captures a	a
op_1	`op` captures f_1	
$bstr_1$	`q` captures b_1	$a_1 = f_1\ a\ b_1$
op_2	`op` captures f_2	
$bstr_2$	`q` captures b_2	$a_2 = f_2\ a_1\ b_2$
		$\ldots$
op_n	`op` captures f_n	
$bstr_n$	`q` captures b_n	$a_n = f_n\ a_{n-1}\ b_n$

The recursive pattern in these captures leads to the declaration:

```
let rec infixL op q =
  fun p ->
    p <|>
    parser { let! a  = p
             let! f1 = op
             let! b1 = q
             let  a1 = f1 a b1
             let  p1 = parser { return a1 }
             return! p1 |> infixL op q } ;;
val infixL : parser<('a -> 'b -> 'a)> -> parser<'b>
             -> parser<'a> -> parser<'a>
```

The `infixR` combinator is declared as follows:

```
let rec infixR op q = fun p ->
    q <|>
    parser { let! a = p
             let! f = op
             let! b = p |> infixR op q
             return f a b } ;;
val infixR : parser<('a -> 'b -> 'b)> -> parser<'b>
             -> parser<'a> -> parser<'b>
```

It is similar to `infixL` but it builds a tree corresponding to the evaluation of operators associating to the right. Using `infixR` in the above example of addition operators would hence give other expression trees:

```
let psR = numOrVar |> infixR addOp numOrVar ;;
val psR : parser<expr>

psR  "a - b + 5"  0;;
val it : (expr * int) list =
  [(Var "a", 1); (Sub (Var "a",Var "b"), 5);
   (Sub (Var "a",Add (Var "b",Num 5)), 9)]
```

where `Sub(Var "a",Add(Var "b",Num 5))` would define an evaluation of

```
a - b + 5
```

such that `5` is first added to `b` and the result then afterwards subtracted from `a`.

Making parsers

The parsers in the examples are based directly on the token parsers and the EBNF grammar:

- A parser is defined for each syntax class.
- Each operator in a syntactic rule in the grammar is translated to a suitable parser combinator.

One should, however, pay attention to the word "suitable": The parser combinators should not only correspond to the syntax but they must also give the right conversion of the textual form to captured value. You will frequently have to write your own parsers for fixed sequence constructs (like `leftPar expr rightPar` in the second example) but is it a good idea to try to design the syntax and the structure (that is, type) of the captured value such that repetitive constructs can be handled using the above parser combinators.

Making the parser in the first example is almost straightforward:

```
let person = pairOf name (listOf number);;
val person : parser<string * int list>

let personData = listOf person;;
val personData : parser<(string * int list) list>
```

```
personData "John 35 2 Sophie 27 Richard 17 89 3" 0;;
val it : ((string * int list) list * int) list =
  [([], 0);
   ([("John", [])], 4);
   ([("John", [35])], 7);
   ...
   ([("John", [35; 2]); ("Sophie", [27]);
                ("Richard", [17; 89; 3])], 35)]
```

The example with expressions is more complex but the required parser combinators have been introduced above, so the grammar can be translated directly into a monadic parser:

```
let rec expr    = term   |> infixL addOp term
    and term    = factor |> infixL mulOp factor
    and factor  = num <|> var
                  <|> parser {let! f = sign
                              let! x = factor
                              return (f x)}
                  <|> parser {let! _ = leftPar
                              let! x = expr
                              let! _ = rightPar
                              return x};;
val expr : parser<Expr>
val term : parser<Expr>
val factor : parser<Expr>
```

The F# compiler issues a warning telling that the above system of mutually recursive declarations may contain cyclic definitions – but the declaration is, nevertheless, accepted by the compiler.

Applying the parser `expr` to the sample string gives the wanted result:

```
expr "-a1 + 2 * (a2 - 3)" 0;;
val it : (Expr * int) list =
  [(Neg (Var "a1"), 3); (Add (Neg (Var "a1"),Num 2), 7);
   (Add(Neg(Var "a1"),Mul(Num 2,Sub(Var "a2",Num 3))), 18)]
```

Parsing the full string

The `personData` and `expr` parsers deliver a match for each matching sub-string. Parsers matching only the full string are made using the `eos` token parser that matches end-of-string (possibly preceded by blank characters):

```
let personDataString = parser {let! dt = personData
                               let! _  = eos
                               return dt };;
val personDataString : parser<(string * int list) list>
personDataString "John 35 2 Sophie 27 Richard 17 89 3" 0;;
val it : ((string * int list) list * int) list =
  [([("John", [35; 2]); ("Sophie", [27]);
     ("Richard", [17; 89; 3])], 35)]
```

```
let exprString = parser { let! ex = expr
                          let! _  = eos
                          return ex };;
val exprString : parser<Expr>

exprString "-a1 + 2 * (a2 - 3)" 0;;
val it : (Expr * int) list =
  [(Add(Neg(Var "a1"),Mul(Num 2,Sub(Var "a2",Num 3))), 18)]
```

Reporting errors

A simple error reporting can be obtained by letting the token parsers update a global variable `maxPos`. The declarations of `token` and `emptyToken` are then preceded by

```
let mutable maxPos = 0
let updateMaxPos pos = if pos > maxPos then maxPos <- pos;;
```

and an extra line is added to the `token` function

```
let token (reg: Regex) (conv: string -> 'a) : parser<'a> =
  fun str pos ->
    let ma = reg.Match(str,pos)

    match ma.Success with
    | false -> []
    | _     ->
       let pos2 = pos + ma.Length
       updateMaxPos pos2
       [( conv(captureSingle ma 1), pos2)];;
```

and similarly for `emptyToken`.

Using this set-up we introduce the type `ParseResult<'a>`

```
type ParseResult<'a> = ParseOk of 'a | ParseError of int;;
```

in order to report an error when an input string cannot be parsed. In the case of such an error, the global variable `maxPos` identifies the position where the error was detected and this position is reported:

```
let parseString (p: parser<'a>) (s: string) =
    maxPos <- 0
    match p s 0 with
    | (a,_)::_ -> ParseOk a
    |   _      -> ParseError maxPos;;
val parseString : parser<'a> -> string -> ParseResult<'a>)

parseString exprString "a - b + c";;
val it : ParseResult<Expr> =
         ParseOk (Add (Sub (Var "a",Var "b"),Var "c"))
```

```
parseString exprString "a - b * (1 + c" ;;
val it : ParseResult<Expr> = ParseError 14
```

where the error in the last case was found at position `14` in the string.

In Exercise 12.8 you are asked to hide the error handling in the builder class for parsers.

Summary

This chapter has introduced the notion of computation expressions of F#. Computation expressions offer a possibility for using special syntactic constructs like `let!`, `return`, etc. with a user-defined meaning through the declaration of so-called builder classes. This concept is based on the theory of monads for functional programming introduced in connection with the Haskell programming language.

The chapter uses sequence expressions (introduced in Chapter 11) and error handling in connection with expression evaluation as examples to show how you may define your own computation expressions. The last section shows how parsers can be constructed in a convenient manner using computation expressions.

Asynchronous computations that will be introduced in Section 13.4 is an important example of computation expressions.

Exercises

12.1 Consider the following "alternative" to the declaration for `bld.Delay` on Page 290:

```
type MaybeClass() =
    ...
    member bld.Delay f:maybe<'a> = delay(start (f()));;
```

This new declaration would not give the desired effect. Explain why.

12.2 Consider the expression evaluation on Page 287. Make a new class declaration for computation expressions that takes care of the evaluation in the environment env and simplify the declaration of the function `I` accordingly. Hint: Consider computations as functions having the type: `Map<string,'a> -> option<'a>`.

12.3 The following grammar for non-empty lists of numbers uses left recursion:

$$\text{numberList} \quad ::= \quad \texttt{number} \mid \text{numberList}\ \texttt{number}$$

A parser strictly following the structure of this grammar:

```
let rec numberLst = parser {let! n = number
                            return [n] }
                    <|>
                    parser {let! ns = numberLst
                            let! n = number
                            return ns @ [n]};;
```

has a problem. Analyze the parser and explain what the problem is.

12.4 Explain the problem with the grammar for expressions on Page 297 that uses left recursion.

12.5 Consider the formulas of propositional logic introduced in Exercise 6.7. In the string representation of such formulas conjunction $\wedge$ can be written either as `&` or as `and`, disjunction either as `|` or as `or` and negation either as `!` or as `not`. For example, the formula

$$\neg(P \wedge \neg(Q \vee R))$$

has several string representations. Two of them are:

```
"neg(P and neg(Q | R))"    and    "!(P & neg(Q or R))"
```

Write a parser for such formulas that takes care of:

- conjunction and disjunction associates to the left,
- conjunction has higher precedence that disjunction, and
- negation has highest precedence.

12.6 Consider the dating bureau in Exercise 4.23. Make a string representation of the file of the dating bureau and a parser that can convert such strings into the representation of the file used in your solution to Exercise 4.23.

12.7 Declare a parser combinator:

```
pFold : ('a -> 'd -> 'a) ->
              parser<'d> -> parser<'a>  -> parser<'a>
```

such that `pFold` $f\ t\ p$ captures the values

$a,\ a_1 = f\ a\ d_1,\ a_2 = f\ a_1\ d_2,\ \ldots,\ a_k = f\ a_{k-1}\ d_k$

if p first captures the value a and repeated use of t afterwards captures the values $d_1, d_2, \ldots, d_k$. Use this parse combinator to make an alternative declaration of `infixL` of the form:

```
let infixL op q p = pFold (fun ...) (pairOf op q) p ;;
```

12.8 The report of errors can be hidden in builder class for parsers and in this exercise you shall make a new version of parsers based on the type declarations:

```
type ParseResult<'a> = ParseOk of 'a | ParseError of int
type parser<'a> = string -> int -> ParseResult<('a*int) list>
```

The builder class should not make use of any mutable variable like `maxPos`, see Page 308, and a value `ParseError` n should occur when an error is discovered at position n.

- Make a new version of the builder class using the above type declarations.
- Revise the two functions `token` and `emptyToken` for generating token parsers accordingly.
- Revise the declaration of the choice operator `(<|>)` and test your builder class using the examples for parsing person data (see Page 306) and expressions (see Page 307).