You want to invent a new language, and you want to do this in F#, don't you? And, of course, you want to base its parser on Monadic Parser Combinators. You've always wanted. So, let's make it happen!

## Index

## Introduction

The compiler of your beautiful new language — with which you will bring the use of `goto` and `null` back to life — will have several components:

- A Lexer, for splitting the source code into tokens.
- A Parser, to convert the token sequence into a syntax tree, and to check the grammatical structure.
- An Intermediate Code Generator.
- A Linker, and so on.

Here we are focusing on the very first component: a piece of code able to analyze the source, to check its syntax against the esoteric formal grammar you defined, and to generate something very well structured for the next components to crunch.

It would be a (probably very complex) function with a signature like:

```
type SourceCode = string

type AbstractSyntaxTree =
    | Goto of Label
    | VariableDefinition of ...
    | ...

val parser : SourceCode -> AbstractSyntaxTree
```

If you think about it, that's not qualitatively different from deserializing a JSON string:

```
type JSON = string

val jsonDeserializer : JSON  -> MyObject
```

Of course, it's likely that a programming language grammar is more complex than the JSON grammar. But the two concepts are alike, and so are the signatures.

Tree-sitter too does something similar. It parses a string like:

```
"let x = 42"
```

and it emits a tree like:

```
(program
  (variable_declaration
    (lexical_declaration
      (identifier)
      (assignment_expression
        (number)))))
```

We can imagine the Tree-sitter grammar for F# as a function with this signature:

```
val treeSitter : SourceCode -> TreeSitterSExpression
```

I guess you see the pattern.
A parser is a function that takes loosely-structured data (usually — but not necessarily — text), and tries to build a more structured data out of it, accordingly to the rules of a formal grammar.

## Mr.James, It's Parsers all The Way Down

We say that the input data is loosely-structured because, in fact, it is not granted to adhere to the rules of the chosen grammar. Indeed, if it violates them, then we expect the parser to fail and to emit an error, to help the user identify the syntax errors.

There are multiple approaches to parsing, including the renowned Regular Expressions.
Monadic Parser Combinators are a particularly fascinating one: they are an example of Recursive Descent Parsers. This means that no matter how complex the parser for a grammar is, it is defined based on smaller, simpler parsers, and those in turn are defined based on even smaller and simpler ones, and so on recursively, down to the trivial parsers.
You can see the same from the opposite perspective: starting from the trivial parsers, by *combining* them together and then by combining their results, recursively, the parser for any arbitrary grammar can be built.

Now, if writing the trivial parsers is, well, trivial, the only challenge that's left is to learn how to *combine* parsers. That is, how Parsers Combinators work.

That's the goal of these pages.

## How we will proceed

There are many similar series online, some specific to F# — such as The "Understanding Parser Combinators" series by Scott Wlaschin — many others based on Haskell, like the excellent Parser Combinators: a Walkthrough, Or: Write you a Parsec for Great Good by Antoine Leblanc.
This post tries to stand out in a few different ways:

- If other attempts to this topic left you scratching your head, this series should make things a lot easier.
  I've done my best to keep the learning curve as smooth as possible. Having to pick between being brief and assuming you knew a lot, or taking a longer path I went with the latter. I think it's nicer to know why stuff works rather than being hit with jargon-heavy explanations.
- Many tutorials begin with writing a simple parser — conventionally, the single-character parser. This does not. Instead, we will focus on combinators first, postponing the implementation of concrete parsers. When I was first introduced to parsers, I was just confused: what on earth does it mean to parse a single character returning a character? What's the point? Where is this leading

me?
I hope I can help you skip past that initial disorientation entirely.

- Parser Combinators are the the *leit-motiv* and serve as the central theme of this book.
  Nevertheless, we'll often stray from the main path and let our imagination roam, exploring a variety of other subjects along the way. You can consider these pages an invitation to discover Functors, Applicatives, and Monads.
- We will write code with Test-Driven Development.
  Isn't it ironic that we developers often lament the absence of tests in our daily job projects and yet, when it comes to writing posts, tutorials and books, we never address testing at all?

Fine, enough with the introduction. Ready? Treat yourself to a sorbet, then let's get started.

### Notes
I am not a native English speaker: if you spot any typo or weird sentence, feel free to send me a pull request.

This blog is crafted by people, not AI. Illustrations are original work by Nanou.

Next: 5 Shades Of Composability

## References
- Recursive Descent Parser
- Tree-sitter
- Scott Wlaschin - The "Understanding Parser Combinators" series
- Antoine Leblanc - Parser Combinators: a Walkthrough, Or: Write you a Parsec for Great Good
- [Turtles all the way down][turtles]

## Comments
GitHub Discussions

{% include fp-newsletter.html %} You have surely noted that people into functional programming have an obsession with the notion of composition. For example, you could have stumbled into sentences such as:

- Exceptions don't compose well.
- Locks are bad because they don't compose.
- Monads compose nicely.

and the like.

This installment will try to help you develop an intuition about what composition is, why you want your parsers to be *composable* and what Applicative Functors and Monads have to do with all of this.

### What's The Fuss About Composition?
You want your esoteric language to support a new groundbreaking serialization format, surely destined to eradicate JSON and YAML. It will let developers express a record as:

```
inst Person
   - Id <- *b19b8e87-3d39-4994-8568-0157a978b89a*
   - Name <- <<Richard>>
   - Birthday <- date{16/03/1953}
```

That's what I call a gorgeous syntax!
To deserialize such a string into the F# record:

```
type Person =
    { Id: Guid
      Name: string
      Birthday: DateOnly }
```

you need a parsing function of type:

```
val parsePerson: string -> Person
```

Let's reflect how to implement it. Again: we are more interested in the journey of *decomposing* the problem into smaller problems and then of *combining* them to generate a parser, rather than in writing this specific parser by hand.
`parsePerson` must address parsing the syntax specific to a record structure, such as that initial `inst` keyword and the series of `- fieldName <- value` strings.
As for the field values, though, `parsePerson` could smartly delegate parsing of GUIDs, strings and dates to some specialized sub-functions. Each would have a specific signature:

```
val parseGuid: string -> Guid
val parseString: string -> string
val parseDateOnly: string -> DateOnly
```

`parseGuid` knows how to get a GUID value from a string surrounded by `*`, `parseString` would extract strings from texts surronded by << and >>, and so on. `parsePerson` will not need to worry about those details, so the skeleton of its implementation can be something like:

```
open System
open Xunit
open Swensen.Unquote

let inline __<'T> : 'T = failwith "Not implemented yet"

type Person =
    { Id: Guid
      Name: string
      Birthday: DateOnly }


let parseGuid: string -> Guid = __
let parseString: string -> string = __
let parseDateOnly: string -> DateOnly = __

let parsePerson: string -> Person =
    fun input ->
        let parseRecordStructure: string -> string * string * string = __

        let guidPart, namePart, birthdayPart = parseRecordStructure input

        { Id = parseGuid guidPart
          Name = parseString namePart
          Birthday = parseDateOnly birthdayPart }


[<Fact(Skip = "incomplete example")>]
let ``it parses a Person`` () =

    let input =
        """inst Person
```

```
   - Id <- *b19b8e87-3d39-4994-8568-0157a978b89a*
   - Name <- <<Richard>>
   - Birthday <- date{16/03/1953}
"""

    let expected =
        { Id = Guid.Parse("b19b8e87-3d39-4994-8568-0157a978b89a")
          Name = "Richard"
          Birthday = DateOnly(1953, 03, 16) }

    test <@ parsePerson input = expected @>
```

In other words, `parsePerson` is a *composition* of:

- some logic specific to the syntax of a record.
- and some lower level parsers.

Is this what functional programmers mean with *composition*? Well, kind of. It's less black and white than this.

## 5 Shades Of Composability

First of all, there is no clear consensus about what "to compose well" means. Search for "*monads are composable*" and "*monads don't compose*": you will find plenty of articles supporting either the claims.

I like to think that the line separating *composable* and *non-composable* is blurry. Given 2 instances of `X`, whatever `X` is, you can either have that:

1. They just **cannot** be combined together.
2. They **can** be combined together, but the result is **not an `X`** anymore.
3. They **can** be combined and they even **form another `X`**; but the result might **behave differently** from expected.
4. They **can** be combined together to **form another `X`**, **100% preserving** all the expected properties. But combining them is **hard** and not scalable.
5. They **can** be combined together to form **another `X`**, **100% preserving** all the expected properties. And combining them is **easy** (and *elegant*, for some definition of *elegant*).

If you will, you can see these levels as follows:

| Level | They can be combined | forming another `X` | preserving their properties | It is easy |
|---|---|---|---|---|
| 1 | No | - | - | - |
| 2 | Yes | No | - | - |
| 3 | Yes | Yes | No | - |
| 4 | Yes | Yes | Yes | No |
| 5 | Yes | Yes | Yes | Yes |

Of course, for your esoteric language and your serialization format, you aim to write parsers proudly fitting the last level.

To clarify each level, let me give you some examples.

### Case 1: things that do not compose

Surprisingly, the building blocks of most programming languages just don't compose.

Take expressions and statements, for example. Expressions can be composed via operators (like in `a * b` and `list1 ++ list2`); statements can be composed sequencing them, like in:

```
use writer = new StreamWriter(filename)
writer.WriteLine("Hello, world!")
```

possibly in combination of control flow structures such as `if`, `for` and `while`.
However, this creates asymmetry. Control structures like `if` can use expressions:

```
if(condition) { ...  }
```

`if`, a statement, gets `condition`, an expression.
The opposite is not true. Expressions can't use control structures. This:

```
int myList = for(int i=0; i<10; i++) { ... };
```

does not even compile.

Similarly, you can pass the expression `sqrt(42)` as an argument to a function. You cannot pass a `for` statement as an argument to a function. This just doesn't make sense, right?
So, in a sense, "expressions and statements don't compose".

By the way, that's one of the appealing traits of some functional languages: they treat control structures as first-class objects, unifying the 2 worlds. They offer greater composability by allowing control logic to be manipulated just like any other value. For example, this is valid F# code:

```
let squares = [for x in 1..10 do yield x*x]
```

**Case 2: composing `Xs` results in something other than `X`.**
Or, more concisely: some things are not closed under composition.
The canonical example is with integer numbers: they compose via division, but they result in float numbers.

Objects are another notable case. You can compose `Wheel` and `Engine`, but you want the result to be `Car`, not something that is both a `Wheel` and an `Engine`.

**Case 3: Things that compose in surprising ways**
The canonical example is again with numbers. In many languages' floating-point arithmetic: `0.1 + 0.2` computes to `0.30000000000000004`, not exactly to `0.3`. You can say that float numbers compose via the sum operation, but not so nicely.

Possibly, another more interesting example is with multi-threading functions using locks. They *do compose*, but in a surprising and unsafe way. Imagine that you have the guarantee that every process requesting locks eventually releases them. Given that you can count on this property for every process in isolation, does the composition of 2 processes hold the same guarantee?
Unfortunately, no. Consider 2 functions acquiring 2 locks `x` and `y`, in opposite order:

```
open System.Threading
open System.Threading.Tasks
open Xunit
open Swensen.Unquote

let x = obj ()
let y = obj ()

let threadA =
    async {
        return
```

```
            lock x (fun () ->
                Thread.Sleep(1000)

                lock y (fun () -> 21))
    }

let threadB =
    async {
        return
            lock y (fun () ->
                Thread.Sleep(1000)

                lock x (fun () -> 21))
    }

let combined () =
    task {
        let taskA = Async.StartAsTask threadA
        let taskB = Async.StartAsTask threadB

        let! a = taskA
        let! b = taskB

        return a + b
    }

[<Fact>]
let ``threadA only`` () =
    task {
        let! b = threadA
        test <@ b = 21 @>
    }

[<Fact>]
let ``threadB only`` () =
    task {
        let! a = threadB
        test <@ a = 21 @>
    }


[<Fact(Skip = "Never terminates because of a deadlock")>]
let ``thread A and B combined cause a deadlock`` () =
    task {
        let! ab = combined ()
        test <@ ab = 42 @>
    }
```

Although when run separately each async function is guaranteed to successfully return, their combination might generate a deadlock. So, function using locks do compose into other functions using locks, but *not nicely*: you cannot guarantee all the invariants still hold.

**What about our manual parser?**

Getting back to our fictional Parser:

```
let parsePerson: string -> Person =
    fun input ->
        let parseRecordStructure: string -> string * string * string = __

        let guidPart, namePart, birthdayPart = parseRecordStructure input

        { Id = parseGuid guidPart
          Name = parseString namePart
          Birthday = parseDateOnly birthdayPart }
```

in which slot does it — and other similarly written parsers — fall?

I hope that the next installment will manage to convince you that it's a case for the 4th level: indeed, imperative parsers like this *do compose*, and mostly without unexpected suprises. But the code you need to write would not scale. It will easily explode from convoluted to crazy unmaintanable.

Did I already tell you that by moving to Applicative and Monadic Parser Combinators you will reach the 5th level, the complete zen illumination and probably a couple of other super-powers?

OK, let's have a break here. You deserve a hot infusion and some relax. When ready, jump to the next chapter.

Previous - Intro ~ Next - That's a Combinator!

## References

## Comments

GitHub Discussions

{% include fp-newsletter.html %} So, let's challenge the composability of imperative parsers. Suppose that other than `parsePerson`:

```
parsePerson: string -> Person
```

we also wish to write a Parser for `RockTrio` objects:

```
parseRockTrio: string -> RockTrio
```

where `RockTrio` is:

```
type RockTrio =
    { Name: string
      BassPlayer: Person
      GuitarPlayer: Person
      Drummer: Person}
```

Beside some specific syntax that might exist for the serializazion of rock trios— and which we don't care at the moment — the point is that we can think of writing `parseRockTrio` leveraging `parseString` and `parsePerson`.

What if we also have `SoloArtist` to be parsed as:

```
type SoloArtist =
    { NickName: string
      Artist: Person }
```

No problem: we can again delegate most of the work to `parseString` and `parsePerson`. This is what I call reuse!

### Your first Parser Combinator

Wait a sec: what if the input string happens to contain *either* a `RockTrio` *or* a `SoloArtist`?

We will need to try both the parsers and to keep the value of the one that happens to succeed. Oh! So there must exist a notion of *success* and *failure*! This means that somehow a parser needs to signal when it failed. Uhm… Maybe we can let parsers raise exceptions in case of failure. Let's define:

```
exception ParseException of string
```

OK, fine: provided that `RockTrio` and `SoloArtist` are both cases of the same union type:

```
type RockBand =
    | RockTrio of RockTrio
    | SoloArtist of SoloArtist
```

our `parseBand` parser could be:

```
open System
open AutoFixture
open Swensen.Unquote
open global.Xunit
open ParserCombinators.Chapter02.ParsingAPerson

type RockTrio =
    { Name: string
      BassPlayer: Person
      GuitarPlayer: Person
      Drummer: Person }

type SoloArtist = { NickName: string; Artist: Person }

type RockBand =
    | RockTrio of RockTrio
    | SoloArtist of SoloArtist

exception ParseException of string

let parseBand parseRockTrio parseSoloArtist input : RockBand =
    try
        parseRockTrio input
    with :? ParseException ->
        parseSoloArtist input

let fixture = Fixture()
fixture.Customize<DateOnly>(
    _.FromFactory(fun (dt: DateTime) -> DateOnly.FromDateTime(dt)))

let rockTrio = fixture.Create<RockTrio>()
let soloArtist = fixture.Create<SoloArtist>()

[<Fact>]
let ``parses RockTrio`` () =
    let successfullyParseRockTrio input = RockTrio rockTrio
    let wontBeUsed input = SoloArtist soloArtist

    let parser = parseBand successfullyParseRockTrio wontBeUsed

    test <@ parser "some input" = RockTrio rockTrio @>

[<Fact>]
```

```
let ``parses SoloArtist if parsing RockTrio fails`` () =
    let justFail input = raise (ParseException "Failing to parse a Rock Trio")
    let successfullyParseSoloArtist input = SoloArtist soloArtist

    let parser = parseBand justFail successfullyParseSoloArtist

    test <@ parser "some input" = SoloArtist soloArtist @>
```

I'm using AutoFixture (with a little trick for handling `DateOnly`) because I am too lazy for defining every test instances.

The implementation:

```
let parseBand parseRockTrio parseSoloArtist input : RockBand =
    try
        parseRockTrio input
    with :? ParseException ->
        parseSoloArtist input
```

is straightforward. Also, if you entirely abandon yourself to the F# type inference, you realize that it is super generic too: indeed, it works with any couple of parsers. We could generalize it as:

```
let (<|>) first second =
    fun input ->
        try
            first input
        with :? ParseException ->
            second input

type Cases =
    | First
    | Second

[<Fact>]
let ``uses first parser if successful`` () =
    let successfullyParseFirst input = First
    let wontBeUsed input = Second

    let parser = successfullyParseFirst <|> wontBeUsed

    test <@ parser "whatever input" = First @>

[<Fact>]
let ``falls back to second parser if first parser fails`` () =
    let justFail input = raise (ParseException "I was meant to fail")
    let successfullyParseSecond input = Second

    let parser = justFail <|> successfullyParseSecond

    test <@ parser "whatever input" = Second @>
```

Let's read the signature again:

```
val (<|>) : (string -> 'a) -> (string -> 'a) -> (string -> 'a)
```

This is a function that, given 2 generic Parsers `(string -> 'a)`, returns a new Parser `(string -> 'a)`. Think about it: so far, we have always created parsers by writing their code, directly. At most we have reused some pre-existing parsers. But here something new is happening: this is a higher-

order function that *combines* parsers, *generating* a brand new one, seemingly out of thin air.
Here's how it is used:

```
let parseRockTrioOrSoloArtist = parseRockTrio <|> parseSoloArtist
```

Look ma, we got a new parser without writing its code!
Kudos! You just have invented a Parser Combinator! It's not a monadic one yet but, I mean, wow! Congrats!

By the way: remember the levels 4 and 5? This is the case of "Given 2 instances of `X` they can be combined together to form another `X`, 100% preserving all the expected properties." It's up to you to judge if the code for generating `parseRockTrioOrSoloArtist` is easy and elegant enough to deserve the Level 5 reward.

## Who Can Stop Us Now?

So, we have built our first Parser Combinator `<|>` which generates a new Parser from 2 possibly failing ones. This could be the first building block of a grammar of Parser Combinators, with which to build the parser of any arbitrarily complex language. Using a bit of fantasy, you could conceive other Parser Combinators such as:

| Name | Signature | Generates a parser that… |
|---|---|---|
| `many` | `(string -> 'a) -> (string -> 'a list)` | parses zero or more occurrences of something, collecting the results in a list. |
| `many1` | `(string -> 'a) -> (string -> 'a list)` | same as above, but expects at least 1 occurrence. |
| `skipMany` | `(string -> 'a) -> (string -> ())` | parses zero or more occurrences of something, discarding results. |
| `skipMany1` | `(string -> 'a) -> (string -> ())` | same as above, but expects at least 1 occurrence. |
| `between` | `(string -> 'open) -> (string -> 'close) -> (string -> 'a) -> (string -> 'a)` | parses something between opening and closing elements. |
| … | | |

It turns out that if you manage to design a set of very expressful and fine tuned building blocks, you don't need to write the code of many parsers: indeed, you will be able to generate any imaginable parser only combinining the most trivial parsers that could be conceived, that are:

| Name | Signature | Generates a parser that… |
|---|---|---|
| `eof` | `(string -> ())` | succeeds only at the end of file. |
| `any` | `(string -> char)` | succeeds no matter what the input contains. |

Don't despair. We will get to this.

But first, I wish you to realize that we cannot proceed before solving a structural problem: our parsing logic is too much coupled with the effectful logic.

I imagine you may not even see this problem — what the heck is the *effectful logic*, to begin with? How can this be a show stopper? And it's fine: the parsing logic is still very simple and usually problems tend to bite only when the complexity reaches higher levels. It is also true, though, that when problems start biting, it is often too late to fix them. So, better investigate.

The good news is, this problem isn't per se a barrier but an invitation: in the next chapter we will intentionally increase the complexity of our parsers, so to see the problem arise. Then we will code-bend it into an improvement, finally getting to Applicative Functors and Monads. Bear with me.

Have a slice of Black Forest Cake, you deserve it and you need energy for the next chapter.

Previous - 5 Shades Of Composability ~ Next - I Told You Not Mess With The Signature!!

## Comments

GitHub Discussions

{% include fp-newsletter.html %}

### Arialdo, You Are A Liar

When I wrote:

```
let parsePerson: string -> Person =
    fun input ->
        let parseRecordStructure: string -> string * string * string = __

        let guidPart, namePart, birthdayPart = parseRecordStructure input

        { Id = parseGuid guidPart
          Name = parseString namePart
          Birthday = parseDateOnly birthdayPart }
```

I have been very reticent. It's not hard to believe that `parseGuid` somehow parses a GUID from the string:

```
*b19b8e87-3d39-4994-8568-0157a978b89a*
```

contained in `guidPart`, but how `parseRecordStructure` managed to extract that string from the whole:

```
inst Person
   - Id <- *b19b8e87-3d39-4994-8568-0157a978b89a*
   - Name <- <<Richard>>
   - Birthday <- date{16/03/1953}
```

is still a mystery. I just wrote:

```
let guidPart, namePart, birthdayPart = parseRecordStructure input
```

Voilà! The 3 strings, in one shot! If only it were so easy… Reality is a bit more complicated than this: the syntax of a record is not just a simple list of elements; it's a mix of field values and other syntactic elements. We can imagine that `parseRecordStructure` needs to identify the initial string:

```
inst Person
   - Id <-
```

Only then it can delegate to `parseGuid`; right after this, it needs to check that the input continues with:

```

   - Name <-
```

and so on. I just glissed on this complexity. What a cheater.
Not to mention that the input string might contain extra spaces and newlines such as in:

```
inst
           Person

           - Id          <-   *b19b8e87-3d39-4994-8568-0157a978b89a*
           - Name        <-   <<Richard>>
           - Birthday    <-   date{16/03/1953}
```

OK, things are getting really sophisticated, now. We need to break this problem down into smaller ones.

## Passing the Baton

An idea that would immensely help is: each parser could return 3 pieces of information:

1. The parsed value (this is the main goal of a parser).
2. If it either succeeded or failed (we covered this with Exceptions)
3. How much of the input string it consumed — so, basically, where it stopped.

The last new information is the key. The next parser can start parsing where the previous one finished, so the input string can be consumed, sequentially, from the first to the last character.

So, rather than:

```
val parser : string -> 'a
```

a parser would rather have the signature:

```
val parser : string -> ('a * string)
```

Returning a tuple with the (polymorphic) parsed value *plus* the unconsumed input, a parser can easily hand the work over to the next parser. You might recognize this as the signature of the State Monad (go read State Monad for The Rest of Us if you are curious). The basic usage pattern, then, could be:

- Invoke a parser.
- Keep the parsed value in a variable.
- Keep processing: invoke the next parser, feeding it with the unconsumed input, so that it can continue from the right position.
- Repeat until you are done with all the syntactic elements.
- Finally, compose all the parsed values into the desired object.
- Return this object *plus* the unconsumed input: after all, this parser itself may be part of a larger parser.

With this pattern in mind, `parsePerson` turns into something like:

```
let parseRecord input = __
let parseGuid input = __
let parseUpToName input = __
let parseString input = __
let parseUpToBirthday input = __
let parseBirthday input = __
let parseTillTheEnd input = __

let parsePerson: string -> (Person * string) =
    fun input ->

        let _, rest = parseRecord input
        let id, rest = parseGuid rest
        let _, rest  = parseUpToName rest
        let name, rest = parseString rest
        let _, rest = parseUpToBirthday rest
        let birthday, rest = parseBirthday rest
        let _, rest
        t parseTillTheEnd rest

        { Id = id
          Name = name
          Birthday = birthday },
        rest
```

No, wait: we also have to consider error handling:

```
let parsePerson: string -> Person * string =
    fun input ->
        try
            let _, rest = parseRecord input
            let id, rest = parseGuid rest
            let _, rest = parseUpToName rest
            let name, rest = parseString rest
            let _, rest = parseUpToBirthday rest
            let birthday, rest = parseBirthday rest
            let _, rest = parseTillTheEnd rest

            { Id = id
              Name = name
              Birthday = birthday },
            rest
        with ParseException e ->
            raise (ParseException $"Failed to parse Person because of {e}")
```

You can imagine that in the first invocation, `parseRecord` consumes the string:

```
inst Person
   - Id <-
```

It can ignore the output: it just needs either to get to the point where `parseGuid` can proceed, or to fail if the string is not found.
Similarly `parseUpToName` would consume:

```
   - Name <-
```

and so on.
OK, that's not too complicated. But I bet you agree: it's a bit repetitive. There is nothing capturing the syntax structure, like something modeling the notion of "each item is prefixed with a field name and separated by its value by a `<-`". Instead, it's all mechanical and not very elegant.

Also, passing those `rest` values around is deadly tedious. I'm personally too lazy to even copy paste that monotonous code. As it often happens, developers' laziness is the catalyst of abstraction: this code immediately ignites our wish to factor the duplication away into a separate, generic function to parse based on a list of parsers, and to return a list of parsed value (being in a list, necesserily of the same type):

```
open Xunit
open Swensen.Unquote

let sequence (parsers: (string -> 'v * string) list) =
    fun (input: string) ->
```

```
        let rec parseRec parsers (rest: string) acc =
            match parsers with
            | [] -> (List.rev acc, rest)
            | currentParser :: nextParsers ->
                let parsedValue, newRest = currentParser rest
                parseRec nextParsers newRest (parsedValue :: acc)

        parseRec parsers input []


type Something = Something of int

let mockParser (i: int) (input: string) = (Something i, input[1..])

[<Fact>]
let ``applies all the parsers consuming 1 character for parser`` () =

    let fiveParsers = [ 1..5 ] |> Seq.map mockParser |> Seq.toList

    let parser = fiveParsers |> sequence

    let parsedValues =
        [ Something 1; Something 2; Something 3; Something 4; Something 5 ]

    test <@ parser "12345abc" = (parsedValues, "abc") @>
```

Woah! That's way harder than the previous one. Besides that, isn't it another Parser Combinator? Does it come in handy for our `parsePerson`? Not really, because it requires that all the parsed elements are members of the same type `'a`. If we really wanted to use this combinator in `parsePerson`, we would need to make `Guid`, `string` and `DateOnly` instances of the same type, for example by wrapping them in a single union type:

```
type MyTypes =
   | GuidCase of Guid
   | StringCase of string
   | DateOnlyCase of DateOnly
```

While this it surely overkill for a serialization language, it is indeed the typical approach for programming language parsers. Let's keep this in mind. Whatever, probably this is not a very useful building block, after all. We have to live with this series of:

```
let value1, rest = parse1 input
let value2, rest = parse2 rest
let value3, rest = parse3 rest
let value4, rest = parse4 rest
....
let valueN, rest = parseN rest
```

for a bit more.

Please, note that this mechanism of passing `rest` around — which is now polluting `parsePerson` — has nothing to do with parsing a `Person`: it is the consequence of having changed the parser signature; if you will, it was caused by a *structural* or a *non-functional* change. Therefore, it is a problem doomed to affect all our parsers, from now on. Damn!
This is what the previous chapter referred to as the *effectful logic*. The *effect* is the need of passing `rest` around, from a call to the next one. As long as we won't be able to factor it away somewhere else (yes: in a Monad), it will spoil the elegance of all our parsers.

## Please, Gimme A Type

Speaking about elegance, I don't know about you, but these verbose signatures:

```
val sequence<'a> (string -> 'a * string) list -> string -> 'a list * string
```

are really starting to get on my nerves. We should do something to make them simpler. Type aliases for the win! Just defining:

```
type Parser<'a> = string -> 'a * string
```

turns `<|>` and `sequence`'s signatures to:

```
val (<|>) : 'a Parser -> 'a Parser -> 'a Parser

val sequence : 'a Parser list -> 'a list Parser
```

Ah! Much, much better!

Don't you feel now inspired to pour a bit more complication into our parsers? We saw before how a change to the parser signature was reflected into a more convoluted code structure in the parser implementation. Let's keep exploring this path to see where it leads us.

## Friends Don't Let Friends Use Exceptions

You read what we coded so far and you torment yourself thinking "*Exception sucks. I am a functional programmer, great Scott! I am supposed to use an `Either` or a `Result` instead!*"

OK, I'm sold: let's use a `Result`, then.

There are 2 possibilities. Either we return the unconsumed input only in case of a successful parsing:

```
type ParseError = string
type Input = string
type Rest = string

type Parser<'a> = Input -> Result<'a * Rest, ParseError>
```

or we return it in any case:

```
type Parser<'a> = Input -> Result<'a, ParseError> * Rest
```

Note the position of `Rest`: in one case it is part of the successful case of `Result`, in the other it is external to `Result`. Both approaches are viable and both will throw a wreck on the code we have written so far, making it apparent that we coupled the error handling concern (the *effectful logic*) with the parsing logic.
Let's use the first signature.

## From Exceptions To Functional Error Handling

Adapting `<|>` and its tests is a piece of cake:

```
let (<|>) (first: 'a Parser) (second: 'a Parser) : 'a Parser =
    fun input ->
        let result = first input

        match result with
        | Ok _ as ok -> ok
        | Error r -> second input

type Cases =
    | First
    | Second
```

```
[<Fact>]
let ``uses first parser if successful`` () =
    let successfullyParseFirst input = Ok(First, "rest")
    let wontBeUsed input = Ok(Second, "rest")

    let parser = successfullyParseFirst <|> wontBeUsed

    test <@ parser "whatever input" = Ok(First, "rest") @>

[<Fact>]
let ``falls back to second parser if first parser fails`` () =
    let justFail input = Error "I was meant to fail"
    let successfullyParseSecond input = Ok(Second, "rest")

    let parser = justFail <|> successfullyParseSecond

    test <@ parser "whatever input" = Ok(Second, "rest") @>
```

Voilà, no more exceptions!
Unfortunately, the same cannot be said for `parsePerson`:

```
let parsePerson: Person Parser =
    fun input ->

        match parseRecord input with
        | Ok(_, rest) ->
            match parseGuid rest with
            | Ok(id, rest) ->
                match parseUpToName rest with
                | Ok(_, rest) ->
                    match parseString rest with
                    | Ok(name, rest) ->
                        match parseUpToBirthday rest with
                        | Ok(_, rest) ->
                            match parseBirthday rest with
                            | Ok(birthday, rest) ->
                                match parseTillTheEnd rest with
                                | Ok(_, rest) ->
                                    Ok(
                                        { Id = id
                                          Name = name
                                          Birthday = birthday },
                                        rest
                                    )
                                | Error err -> Error err
                            | Error err -> Error err
                        | Error err -> Error err
                    | Error err -> Error err
                | Error err -> Error err
            | Error err -> Error err
        | Error err -> Error err
```

Holy cow! This is absolutely horrific. There is more error control code than domain logic! But this was somehow expected: changing the signature of `Parser` implies some kind *structural logic* to be executed when parsers — *all the parsers* — are executed. In our case we pushed ourselves to the limit combining 2 structural changes: passing `rest` around and matching error cases. And we got a Pyramid of Doom

The good news: the attempts to factor this mess out will lead us to invent Applicative Functors and Monads. Let's reflect how we should proceed.

A quick espresso? Good idea, it's the perfect moment for a break! See you at the 5th chapter.

Previous - That's a Combinator! ~ Next - A Different Kind of Coupling

## References

- Pyramid of Doom

## Comments

GitHub Discussions

{% include fp-newsletter.html %}

### A Tale Of 2 Coupling Types

The `parsePerson` function delegates the parsing of GUIDs, strings and dates to external functions. We think that this decouples it from of the parsing logic of the specific fields. While this is indeed the case, the code we just obtained clearly shows that some problems remain. By now, you should have guessed why; there are in fact 2 types of coupling:

- A function can be coupled with *the logic* of other components.
  This cannot be our case: `parsePerson` does not even know how GUIDs are represented; this logic is completely delegated to `parseGuid`.
- A function could be coupled with *the mechanics* of glueing things together, what we called the *effectful logic*.
  This means that even if it is *functionally isolated*, the code is *structurally* influenced by the *glueing mechanism*. This must be our case.

Now, if you more into OOP than into functional programming, it is likely that you never heard of the second form of coupling. After all, in OOP "*glueing things together*" is rarely a big deal. In OOP there are a few techniques for gluing things together — such as sending messages to objects in a sequence, or passing values around or applying values to functions — and all of them are natively implemented by the programming language. And all boil down to the notion of *function application*.

### Dumb and Smart Function Applications

The native function application works just fine as long as it operates within the simple case of things with compatible signatures:

```
f : 'a -> 'b
g : 'b -> 'c
```

Languages natively know how to glue `f` with `g` because the output of `f` can be passed, just as it is, to `g`.

```
let glued x = g (f x)
```

This leads to 2 traits in the OOP's function application:

- We rarely have to worry about it.
  Even more: we intentionally design our methods so that their signature makes the compiler happy.

When things have incompatible signatures, we write wrappers and adapters to work around the incompatibility.

- We often assume function application is dumb.
  It just passes a value to the next function, doing nothing else meanwhile, and we are happy with that. There are few exceptions: indeed, design patterns like Decorator and Aspect Oriented Programming are a way to add some logic to method calls.

The farest we go with making things intentionally incompatible and with adding new functionalities to function application, in OOP, is with Async calls:

```
f : 'a -> Task<'b>
g : 'b -> Task<'c>
```

Those functions just don't combine natively. We dare to go this direction only because it is an easy problem to solve: the language reserves a special treatment to this case, providing us with the dedicated keywords `async` and `await`. For example, in C#:

```
async Task<C> GluedAsync(A x) =>
    await g(await f(x));
```

In a sense, exceptions are also an example of this. If your language did not implement exceptions, you would need to handle errors like Go does:

- Checking every and each call for returned errors.
- Propagating the error upstream.
- Passing the call stack too.

etc.

Your domain code would be horribly polluted by this error handling stuff. A way out of this could be to extend the native function application so that, other than just passing a value from a function to the next one, it would *also* tackle the error handling responsibility. Exceptions are so convenient to use because the native function application does all of this, under the hood.

## Breaking The Rules

Both exceptions and the `async`/`await` mechanism are ad-hoc, built-in solutions. We cannot expect that the native F# function application provided a special treatment for parser functions returning `Results` of tuples. This is too specific to our peculiar use case.

In fact, in FP it's often the case that we intentionally design the function signatures ignoring the native gluing mechanism. We take the freedom to design functions that don't fit together because function application is easy to extend. And because this gives us the chance to put some custom logic in the gluing mechanism.

As an FP programmer you don't settle with the dumb native function application. You want fancier ones: you want them to deal with async calls, with exceptions. Or to log each call; or, again, to deal with errors via a `Result` instance instead of exceptions, as in our case. Or — why not? — to do some combinatorial calculation. I stress that in "You want fancier ones" I intentionally used a plural: in fact, really, you want a family of function applications, one for each of your specific use case.

FP techniques provide a way more generic solution than special keywords like `async` and `await`. If you read Monads for The Rest of Us, the notion of Applicative Functors and Monads as an extension of function application should not be new to you.

Here's the takeaway: if in OOP the signature incompatibility is *a problem* to be avoided or to be solved by the means of wrappers and adapters, in FP the same incompatibility is *a design tool* to be leveraged.

So, let's see how to fix the pyramid of doom we wrote in `parsePerson` by distilling a new function application. And let's see how this leads us to re-invent — yet another time — Monads.

Take a break, bite an apple, then jump to the next installment.

Previous - I Told You Not To Mess With The Signature! ~ Next - Mapping the Journey

## Comments

GitHub Discussions

{% include fp-newsletter.html %} I sometimes feel uneasy when books keep providing too many details without giving a hint of where things are headed. This chapter takes a more relaxed approach, setting aside implementation details to offer you an early overview of what to expect. I hope it helps you getting oriented as you read the next chapters.

The gist of the previous few pages is: we should factor structural dependencies away from our code. This will allow us to create and combine parsers while focusing on the essence of parsing, without getting bogged down by the uninsteresting mechanical details of passing `rest` and of pattern-matching errors. Once we understand how to abstract these concerns, we can set our sights on developing the following tools:

1. Some easy-to-use functions for combining generic parsers.
2. Some operators for building more fluent parsing-combining expressions.
3. A special syntax for manipulating parsers in imperative style — of course, while still being purely functional.
4. Lifting functions, to project ordinary functions into the Parser world.
5. Parser-powered function application

Let me show you some examples for each of those 4 categories.

### Functions For Combining Generic Parsers

Your language needs a switch statement. You get a burst of inspiration and come up with this:

```
SWITCH
/condition/ ~~ <block>
/condition/ ~~ <block>
/condition/ ~~ <block>
...
```

The beauty of this syntax! I'm such a fan of your language! You wish to parse this as an instance of `Switch`:

```
type SwitchBranch = { Condition : Condition; Block : Block }
type Switch = { Branches : SwitchBranch list }
```

Imagine that you already managed to have a parser for `predicate` and one for `block`. Presto! Here is `switchParser`:

```
let branch =
    tuple3
        (condition |> between (pchar '/') (pchar '/'))
        (pstring " ~~ ")
        (block |> between (pchar '<') (pchar '>'))
    |> map (fun (cond, _, block) -> { Condition = cond; Block = block })

let branches =
    branch |> sepBy (pchar '\n')

let switchParser =
    tuple3
        (pstring "SWITCH")
        (pchar '\n')
        branches
    |> map (fun (_, _, branches) -> { Branches = branches })
```

Read it from the bottom:

- the `switchParser` parses 3 elements in a sequence (`tuple3`):
  - the initial string `SWITCH`
  - a newline
  - and then all the branches (the `branches` parser).
- In turn, the branches section is:
  - a repetition of branch elements (`branch`)
  - separated by newlines `sepBy (pchar '\n')`
- And, finally, what's the syntax of a branch? It's 3 elements:
  - a condition, surronded by `/`
  - a lovely ~~
  - and a block, between < and >

I hope you get the idea: you can use combinators like `between` and `sepBy` to *describe* your syntax and to build parsers without being distracted by the unconsumed input and the error handling. You can see this as an internal Domain Specific Language that tries to be more descriptive than imperative.

Also, note that the arguments we feed `between` and `sepBy` with are parsers themselves. The outputs of `between` and `sepBy` are also parsers, which are then fed into `branch`, producing yet another parser. This parser is subsequently passed to `branches`, which multiplies it and generates a new parser. Finally, all of this culminates in `switchParser`, the outermost parser. Satoshi Kon would be surely delighted by this recursive dreamscape, where each parser unfolds into another parser, like a never ending spiral of dreams nested within dreams.

## Operators For Building Parsing-Combining Expressions

Sometimes code is more expressive when infix operators are used. The syntax of F# is often regarded as a notable example, because it allows you to write expressions in a way that closely resembles natural language. Instead of a series of nested function calls like:

```
let res =
    saveAudit "user_flow" (
        sendWelcomeIfNew "welcome_template" (
            updateLastLogin true (
                fetchProfile "basic" (
                    getUser 42))))
```

one may prefer a cascade of calls connected with by the pipe operator `|>`:

```
let res =
    getUser 42
    |> fetchProfile "basic"
    |> updateLastLogin true
    |> sendWelcomeIfNew "welcome_template"
    |> saveAudit "user_flow"
```

Since F# supports custom operators (C#, why, why don't you?) it is only logic that you will want some convenient infix operators for manipulating parsers.

Here's an example. You want a combinator to transform a *parser of something* into a *parser of something surrounded by tags*. It would take 3 parameters:

- A parser able to detect an opening tag.
- A parser for the closing tag.
- The parser you want to enrich.

Here's an implementation:

```
let between before after parser =
    before >>. parser .>> after
```

Besides the internal implementation of those `>>.` and `.>>` — which we will see in the next pages — you can think to them as pipe operators similar to the familiar `|>`: they connect the left parser with the right parser. See the `.` on one side of them? It indicates which parser you want to obtain the result from; the other parser will be executed, but then its result will be ignored.

So, an expression like `a >>. b` can be read as:

- give me a parser that
- expects whatever the parser `a` expects
- then continues parsing whatever `b` is good at parsing
- and, finally, returns only the thing parsed by `b`, dropping the result of `a`.

We will build several other little operators, like `|>>`, `>>=` and `<|>`. You'll have plenty of time to grasp them. For now, just get the idea: you will end up enriching F# with a bunch of new little grammatical constructs and syntactic elements, to make your parsing language more expressive.

## Special Syntax For Writing Imperative Code

Sometimes infix operators are beautiful. Sometimes the dense syntax they produce is too much for our brain to crunch, and we prefer a more familiar, imperative style. Wouldn't be amazing if F# let you write imperative-like code, while making sure it's still functional? Enter do-notation, or computation expressions. Here is how the `between` combinator we defined before can be written with this style:

```
let between parser openTag closeTag =
    parse {
        let! _ = openTag
        let content = parser
        let! _ = closeTag

        return content
    }
```

- See the `parse {` in the second line? It makes it clear you are building a parser.
- Each line runs a parser and stores the resulting parsed value in variable, for future use.
- Notice how they use a special parser-powered `let!` keyword.
- It is apparent which values are being ignored and which one is returned.

Despite the syntax seems a series of statements, it is in fact a combination of high-order functions. F#'s syntactic sugar magic lets you ignore this fact and just focus on the task at hand. We will see in

a few pages how this works under the hood. For the time being, I invite you to see this as a way to easily express parsing activities that you wish to execute in a specific sequence.
Let me show you a second example. Say you want to parse a tuple:

```
(42, 99)
```

as an instance of:

```
(int * int)
```

So, it's a `(` followed by a number, then a comma, then some spaces, etc. The corresponding needed parser is pretty much a literal translation of this description:

```
let tuple : (int * int) Parser =
    parser {
        let! _ = str "("

        let! a = number
        let! _ = comma
        let! _ = many space
        let! b = number

        let! _ = str ")"

        return (a, b)
    }
```

Isn't this very conveniently linear? It looks like just assigning parsed values to variables. In fact, what you see on the right side of a `let!` is not a parsed value, but a parser. The special `let!` runs the parser on the right side, saves its result in the variable on the left side (possibly, ignoring it) and then continues parsing the rest, doing all the magic about passing `rest` and pattern matching the `Result`.

Of course, you can add any complexity there, like recursive calls or nested computation expressions. More on this in the upcoming pages.

## Lifting Functions

Manipulating parsers is so fun and rewarding. But often, you would prefer to solve the problem at hand in terms of values, rather than in terms of the parsers that emit those values: it's just one level of indirection less.
If I may borrow a metaphor, it's like there are 2 separate realms: the poor's man world of ordinary functions, manipulating simple values; and the elevated World Of Parsers, up there beyond the clouds, a realm full of funny operators, Functors and Monads. It would be awesome to work down here on the ground, as we are already used to do, then to pop the result into a magic elevator, hit the button for the Parser World floor, and take it all in up there, for free. This is what the lifting functions and operators are about. Let me show you.

Consider the case of parsing an arithmetic expression:

```
42+79
```

In the AST of your language, this can be represented as an instance of `Expression`:

```
type Operation = Sum | Sub | Mul | Div

type Expression = Expression of int * int * Operation
```

Building an instance of `Expression` is trivially a matter of defining:

```
let buildExpression (a: int) (op: Operation) (b: int) =
    Expression (a, b, op)
```

and of invoking it:

```
let expression = buildExpression 42 79 Sum
```

Now, let's push `buildExpression` into the elevator. It will lift it into the world of parsers, so that it becomes a `buildExpressionOnSteroids`:

```
let buildExpressionOnSteroids = lift3 buildExpression
```

That's it. While `buildExpression` signature was:

```
val buildExpression : int -> Operation -> int -> Expression
```

by the application of `lift3` the signature turned into:

```
val buildExpressionOnSteoids : int Parser -> Operation Parser-> int
Parser -> Expression Parser
```

It became a parser combinator manipulating parsers to produce another parser! Unbelievable! Think about it: from a humble factory building *something* and knowing absolutely nothing about parsing, you managed to create a function that *parses that something*. Diabolic.

## Parser-Powered Function Application

Look this other example. As the Benevolent Dictator For Life of your language, you proclaim that the syntax:

```
7 times date{16/03/1953}
```

builds a list of 7 dates (all the same), boxed inside a `MultiDate` object. Sounds like a very useful construct, doesn't it?

```
type MultiDate = MultiDate of (DateOnly list)

let multiDate : MultiDate Parser = __

[<Fact>]
let ``parses a MultiDate`` () =
  let input = "7 times date{16/03/1953}"

  let date = DateOnly(1953, 03, 16)
  test <@ run multiDate input =
            Success (MultiDate [date; date; date; date; date; date; date], "") @>
```

To build `multiDate`, you can start by splitting the input `7 times date{16/03/1953}` into its syntactical components:

1. `7`: the number of dates you wish.
2. : a space
3. `times`: one of your language's keywords.
4. : a space
5. `DateOnly(1953, 03, 16)`: the date.

With those 5 values, building a `MultiDate` instance is a breeze:

```
let makeMultiDate (n: int) (_space: char) (command: string) (_space2: char) (date:
DateOnly) : Foo =
    let dates = [ for i in 0 .. n - 1 -> date ]
    MultiDate dates
```

The problem is: you don't have *values*; instead, you have *parsers of values*:

```
let nP:        int Parser      = intParser
let spaceP:    char Parser     = charParser ' '
let commandP:  string Parser   = str "times"
let dateP:     DateOnly Parser = parseDateOnly
```

Can you feed `makeMultiDate` with parsers instead of with actual values?

```
let multiDate : MultiDate =
    makeMultiDate     nP     spaceP     commandP     spaceP     dateP
```

Of course you can't! This won't even compile! That's not how function application works.
What if instead of the native F# function application you use a specialized *parser-aware function application*?

```
let multiDate: MultiDate Parser =
//  makeMultiDate     nP     spaceP     commandP     spaceP     dateP
    makeMultiDate <!> nP <*> spaceP <*> commandP <*> spaceP <*> dateP
```

What the heck? It works!!! This funny syntax gives you back is *a parser* for `MultiDate`. How can it be? There must be some black magic involved!

### Did It Pique Your Curiousity?

If all of this sounds confusing, that's perfectly fine: I just hope it also sounds a bit exciting.
What you saw above involves a fair bit of syntactic sugar, and a good amount of behind-the-scenes magic. As with any magic trick, true understanding comes from peeking behind the curtain and rebuilding it from scratch. That's exactly what we are doing in the next chapter.

Enough with reading code: take a moment for a Yerba mate, warm up the keyboard and finally hit some keys!

Previous - A Different Kind of Coupling ~ Next - Parser-Powered Function Application!

## References

Satoshi Kon - Paprika (2006)

## Comments

GitHub Discussions

{% include fp-newsletter.html %} Until this point, we have worked with these types:

```
type ParseError = string
type Input = string
type Rest = string

type Parser<'a> = Input -> Result<'a, Rest, ParseError>
```

So defined, `Parser` is a type alias. It's a good idea to improve the encapsulation wrapping the function on the right side in a Single-case Discriminated Union:

```
type Parser<'a> = Parser of (Input -> Result<'a * Rest, ParseError>)
```

Note the `Parser of` case constructor. We can also grant `Result<Rest * 'a, ParseError>` the dignity of its own type, independent from `Result`:

```
type ParseResult<'a> =
    | Success of ('a * Rest)
    | Failure of ParseError
```

```
type Parser<'a> = Parser of (Input -> 'a ParseResult)
```

So defined, the details about passing `rest` and handling errors are not directly visible from the outside. Good for information hiding. A drawback of having the function wrapped inside `Parser`, though, is that you can't direcly apply it to a string input anymore. A helper function will come in handy:

```
let run (p: 'a Parser) (input: string)=
    match p with
    | Parser f -> f input


type SomeType =
    | One
    | Two
    | Three

[<Fact>]
let ``runs a parser`` () =
    let mockParser = Parser(fun _ -> Success(One, "rest"))

    test <@ run mockParser "any input" = Success(One, "rest") @>
```

Actually, you can simplify `run` as:

```
let run (Parser f) input =
    f input
```

If you think to `Parser` as a container, you can imagine `run` as the function that *opens* it and reveals its content. This is a common theme when working with monads: even if monads are not boxes — nor burritos — sometimes the idea of *opening a monad*, operating with its content, and then *wrapping* the result back in another Monad comes in handy.
Fine. The `Parser` type and the `run` function are an excellent starting point.

## A Grammar For A Parsing Language

Let's give a look to the ending point, too. Over the course of the next chapters, we will develop several functions and operators. They will form the syntactic elements of a grammar with which you'll be able to build whatever parser.

Let me list them below so you know right away where we're headed.

Name Signature Purpose .>>.andThen ‘a Parser -> ‘b Parser -> (‘a * ‘b) Parser Parse ‘a, then ‘b, and finally return both in a tuple. >>=bind ‘a Parser -> (‘a -> ‘b Parser) -> ‘b Parser Parse ‘a and pass it to a continuation. <!><<|map (‘a -> ‘b) -> ‘a Parser -> ‘b Parser Transform a Parser of ‘a into a Parser of ‘b. |>>pipe ‘a Parser -> (‘a -> ‘b) -> ‘b Parser Like F# pipe operator |>, but operating with Parsers. <*>ap (‘a -> ‘b) Parser -> ‘a Parser -> ‘b Parser Partial application of a Parser argument to a multi-parameters function. <|> ‘a Parser -> ‘a Parser -> ‘a Parser Try applying a Parser. It if fails, try another one. .>> ‘a Parser -> ‘b Parser -> ‘a Parser Apply 2 parsers, returning the result of the first one only. >>. ‘a Parser -> ‘b Parser -> ‘b Parser Apply 2 parsers, returning the result of the second one only. many ‘a Parser -> ‘a list Parser Repeatedly apply a parser until it fails, returning a list of parsed values. many1 ‘a Parser -> ‘a list Parser Same as above, but expects at least 1 occurrence. skipMany ‘a Parser -> () Parser Parse zero or more occurrences of something, discarding the result. skipMany1 ‘a Parser -> () Parser Same as above, but expects at least 1 occurrence. between ‘o Parser -> ‘c Parser -> ‘a Parser -> ‘a Parser Parse something between opening and closing elements. sepBy

‘a Parser -> ‘b Parser -> ‘a list Parser Parse a list of ‘a elements separate by b. returnp ‘a -> ‘a Parser Lift a plain value into the Parser world. liftmap (‘a -> ‘b) -> ‘a Parser -> ‘b Parser Elevate a 1-parameter function into the Parsers world. lift2 (‘a -> ‘b -> ‘c) -> ‘a Parser -> ‘b Parser -> ‘c Parser Elevate a 2-parameter function into the Parsers world. lift3 (‘a -> ‘b -> ‘c -> ‘d) -> ‘a Parser -> ‘b Parser -> ‘c Parser -> ‘d Parser Elevate a 3-parameter function into the Parsers world. pipe2 ‘a Parser -> ‘b Parser -> (‘a -> ‘b -> ‘c) -> ‘c Parser Apply 2 Parser arguments to a 2-parameter function expecting values. pipe3 ‘a Parser -> ‘b Parser -> ‘c Parser -> (‘a -> ‘b -> ‘c -> ‘d) -> ‘d Parser Apply 3 Parser arguments to a 3-parameter function expecting values. Don’t feel overwhelmed. They are way easier to write than they appear at first.
As a starter, we will build `|>>` and `<<|`.

## From The F# Native Function Application...

Our goal is to develop an alternative to the F# native function application that, under the hood, takes care of passing `rest` around and of handling errors. This bears the question: what’s the native F# function application, to begin with?

```
[<Fact>]
let ``function application`` () =
    let twice x = x * 2

    test <@ twice 42 = 84 @>
```

Do you see that whitespace between `twice` and `42`?

```
twice 42
     ^
```

With a bit of fantasy, you can imagine that this is an actual operator: it applies `twice` to the value `42`. If it were a real operator, its signature would be:

```
val ( ) : ('a -> 'b) -> 'a -> 'b
```

This is fictional code, though: a whitespace cannot be used as an operator name. But wait a moment! F# *does provide* an actual operator with that exact signature! It’s `<|`. If you were to write it manually, you would have:

```
let (<|) (f: 'a -> 'b) (a: 'a) : 'b = f a

[<Fact>]
let ``function application, via an operator`` () =
    let twice x = x * 2

    test <@ twice <| 42 = 84 @>
```

Its real implementation (FSharp.Core/prim-types.fs#L4546) is:

```
let inline (<|) func arg1 = func arg1
```

Almost identical! Oh, that feeling when a reimplementation matches the original! So gratifying...
`<|` is the same of the famous pipe operator `|>`, only with flipped parameters. Let’s reinvent `|>` too:

```
let (|>) (a: 'a) (f: 'a -> 'b) : 'b = f a

[<Fact>]
let ``function application with pipe`` () =
    let twice x = x * 2

    test <@ 42 |> twice = 84 @>
```

`|>` is actually defined like this in the F# code (FSharp.Core/prim-types.fs#L4540):

```
let inline (|>) arg func = func arg
```

## ...To A Parser-Powered Function Application

If we want to define a parser-powered function application, don't we just need to change the signature of `<|` to accept an `'a Parser` instead of just an `'a`? Good question. Let's see. We can try to define 2 enhanced operators:

| F# native | Parser-powered |
|---|---|
| `<\|` | `<<\|` |
| `\|>` | `\|>>` |

with these signatures:

| Version | Operator | Signature |
|---|---|---|
| F# native | `<\|` | `('a -> 'b) -> 'a -> 'b` |
| Parser-powered | `<<\|` | `('a -> 'b) -> 'a Parser -> 'b` |

and:

| Version | Operator | Signature |
|---|---|---|
| F# native | `\|>` | `'a        -> ('a -> 'b) -> 'b` |
| Parser-powered | `\|>>` | `'a Parser -> ('a -> 'b) -> 'b` |

```
let (<<|) (f: 'a -> 'b) (aP: 'a Parser) : 'b = __

let (|>>) (aP: 'a Parser) (f: 'a -> 'b) : 'b = __

// A never-failing mock parser always returning 42
let p42: int Parser =
    Parser (fun _ -> Success(42, "rest"))

[<Fact>]
let ``parser-powered function application`` () =
    let twice x = x * 2

    test <@ twice <<| p42 = 84 @>
    test <@ p42 |>> twice = 84 @>
```

But this is a complete nonsense! Think about it: an `int Parser` is not an `int` value; it's *a promise* of an `int`. Put your ear up to it and you'll hear it whispering:

> True, I am not an actual value myself. But I promise I can provide you with one.
> Just feed me with an input string: I will do my best to parse it and, if I don't fail, I will eventually give you back that much-desired `int` value.

To keep the metaphor going, if `twice <<| p42` could speak, it would tell you:

> I see what you want to do! You want to multiply some integer value by 2.
> You know that we don't have that integer value just yet; you also know we will eventually obtain one as soon as you feed with an input string.

> Let’s make a deal: pass me the calculation you want to run. I will parse that input string for you, then I will apply that function. I promise.
>
> I mean: you gave me a Parser, I pay you back with another Parser. It’s only fair.

This means that the signature of our enhanced operators should rather be:

| Version | Operator | Signature |
|---|---|---|
| F# native | `<\|` | `('a -> 'b) -> 'a -> 'b` |
| Parser-powered | `<<\|` | `('a -> 'b) -> 'a Parser -> 'b Parser` |

and:

| Version | Operator | Signature |
|---|---|---|
| F# native | `\|>` | `'a -> ('a -> 'b) -> 'b` |
| Parser-powered | `\|>>` | `'a Parser -> ('a -> 'b) -> 'b Parser` |

Since both `<<|` and `|>>` now return a `'b Parser`, the assertions:

```
test <@ twice <<| p42 = 84 @>
test <@ p42 |>> twice = 84 @>
```

will not compile. We cannot compare the returned `'b Parser` with `84`. Is a sense, you got back a future, successfully parsed `84` wrapped in a box. You need `run` to unwrap it:

```
test <@ run (twice <<| p42) "some input" = Success ("rest", 84) @>
test <@ run (p42 |>> twice) "some input" = Success ("rest", 84) @>
```

Putting all together:

```
let (<<|) (f: 'a -> 'b) (aP: 'a Parser) : 'b Parser = __

let (|>>) (aP: 'a Parser) (f: 'a -> 'b) : 'b Parser = __

let p42: int Parser =
    Parser (fun _ -> Success(42, "rest"))

[<Fact>]
let ``parser-powered function application`` () =
    let twice x = x * 2

    test <@ run (twice <<| p42) "some input" = Success(84, "rest") @>
    test <@ run (p42 |>> twice) "some input" = Success(84, "rest") @>
```

It compiles.
Now that we are armed with a failing test, we are ready to implement `|>>` and `<<|`. Not only can you be driven by tests: types can drive you too. Listen to the signature:

```
let (<<|) (f: 'a -> 'b) (aP: 'a Parser) : 'b Parser = ...
```

It tells us to return an instance of `Parser`. Fine, let’s obey. We need to invoke the `Parser` case constructor:

```
let (<<|) (f: 'a -> 'b) (aP: 'a Parser) : 'b Parser =
    Parser ...
```

What does the case constructor want as an argument? A function from an input `string` to something. Fine, type system, I trust you:

```
let (<<|) (f: 'a -> 'b) (aP: 'a Parser) : 'b Parser =
    Parser (fun input ->
        ...)
```

Now:

- If you want to return a `'b Parser` you have to obtain a `'b` value.
- The only way to get one is through `f`, which is an `'a -> 'b`.
- In order to invoke `f` you need an `'a` value.
- Damn. You don't have it. You have a parser of `'a`, instead, so something that can give you an `'a` value, if only you run it with a `string` input.
- But look! You do have a `string` input, because you live inside a lambda!
- So, just *run* `aP` with `input` and you will get an `a`   `ParseResult`. If it is successful, you will find the `'a` value there, together with the unconsumed input.
- (If it fails, you are safe to let `<<|` fail too).
- With the obtained `'a` value, you can finally invoke `f`, and get a `'b` value.
- You can wrap the `'b` value and the unconsumed value in a tuple, and finally return it, successfully.

Let's translate all of this to code:

```
let (<<|) (f: 'a -> 'b) (aP: 'a Parser) : 'b Parser =
    Parser (fun input ->
        let ar : 'a ParseResult = run aP input
        match ar with
        | Success (a, rest) -> Success (f a, rest)
        | Failure s -> Failure s )
```

Relying on type inference, you can simplify this to:

```
let (<<|) f aP =
    Parser (fun input ->
        match run aP input with
        | Success (rest, a) -> Success (rest, f a)
        | Failure s -> Failure s )

let (<<|) f aP =
    Parser (fun input ->
        match run aP input with
        | Success (a, rest) -> Success (f a, rest)
        | Failure s -> Failure s )
```

Not bad! And not too difficult, after all. This function is interesting because it captures the *effectful logic*:

- It passes the unconsumed input forward.
- It deals with failures.

It does not contain any logic other than this. It really keeps concerns separate:

- `f` encapsulates the actual domain logic.
- `<<|` encapsulates the *glueing* logic.

As we said in chapter 5, it's a form of structural decoupling, obtained with a tool other than the classic dependency injection.

Defining `|>>` is trivial: just swap the parameters:

```
let (|>>) aP f = f <<| aP
```

Green tests. Hurray!
In the next pages, we'll keep using `|>>`: similarly to `|>`, its form strongly evokes the idea of sending an argument through a pipe to the next function. As we'll see shortly, however, when it comes to `<<|`, we will prefer a different interpretation, one that involves mapping a function from the lower world to the higher realm of parsers. In this case, we'll prefer the name `map` or the symbol `<!>`. More on this later. First, let's see a couple of examples how you can use these brand new operators.

## What An Epic Time To Live!

Let's cover this use case: you want to parse an input in a specific way, but you can't get to the desired result all at once. Instead, first, you parse the input into a value that is not exactly what you want. Then, in each of the next steps, you apply a function to transform that value to something else, possibly of a different type, until you get the final type you had in mind. Of course, since we are talking about parser combinators, you'd like to combine all these steps into a single, composed parser.

For example, it is trivial to write a DateTime parser for the boring format `yyyy-MM-dd hh:mm:ss` if only you cheat and you use the native `TryParse` method:

```
let dateTimeP: DateTime Parser =
    Parser (fun input ->
        match DateTime.TryParse(input[..18]) with
        | true, dateTime -> Success (dateTime, input[19..])
        | false, _ -> Failure "Expected a date")

[<Fact>]
let ``date test`` () =
    test <@ run dateTimeP "2025-01-01 18:11:12, the rest" = Success (DateTime(2025,
01, 01, 18, 11, 12), ", the rest") @>
```

But when it's time to use `DateTime` as an internal representation of dates in your esoteric programming language, you reject the idea: it's way too conventional. You once read about the Epoch time which represents time by the number of seconds since the midnight of 1 January 1970. Why 1970, you wonder? Because Unix was created in the early 1970s.

What a lame excuse! You rather prefer a format that celebrates yourself. What about using:

```
type EpicTime = EpicTime of double
```

measuring time as the number of second since where *you* were born? After all, that was the most epic moment in the universe history.

(Let's say, it was on 18 February 2005, at 18:24:02, OK?)

Here's how to convert a depressing `DateTime` to a gorgeous `EpicTime`:

```
type EpicTime = EpicTime of double

let toEpicTime (date: DateTime) =
    let aRemarkableDate = DateTime(2005, 02, 28, 18, 24, 02)
    let secondsFromTheEpicDate = date.Subtract(aRemarkableDate).TotalSeconds
    EpicTime secondsFromTheEpicDate

[<Fact>]
let ``Epic Time is so much better than Epoch Time`` () =
    test <@ DateTime(2025, 01, 01, 18, 11, 12) |> toEpicTime = EpicTime 626140030.0
@>
```

As a fan of your language, I see how `EpicTime 626140030.0` is a way more convenient and intuitive representation than `01 January 2025 18:11:12`. Only, now we need an `EpicTime Parser`. Presto! You can combine the `DateTime Parser` with `toEpicTime`, using `|>>`:

```
let epicTimeP : EpicTime Parser =
    dateTimeP |>> toEpicTime

[<Fact>]
let ``epicTime test`` () =
    test <@ run epicTimeP "2025-01-01 18:11:12, the rest" = Success (EpicTime
626140030.0, ", the rest") @>
```

Look how concise:

```
let epicTimeP = dateTimeP |>> toEpicTime
```

It does not even mention a single function parameter: it a pure combination of parsers and functions. This style is called Point Free Style or Tacit Programming, and is typical when playing with functional combinators. It's fundamentally the consequence of manipulating things in an elevated, more abstract context. We will get back to this later.

## Mapping strings to types

Let's see an other simple example based on the very same idea: applying a function to the value eventually parsed by a parser or, anticipating the lingo you will encounter, *mapping* functions to parsers.
Say you want to parse keywords of your programming language. You can proceed by steps. First, you start with a parsers that just checks if the input contains specific keywords. In other words, a parser reading an input string and succeeding if it contains a specific string. This trivial parser would return the specific string, if it was found in the input; otherwise, it would fail. Later, you can refine it to return specific custom types, by the means of `|>>`.

To check the presence of a specific string, if you want to be generic, you can define a parser factory, something that gets the string you want to match and generates a parser for it:

```
let str (s: string) =
    Parser (fun input ->
        if input.StartsWith(s) then Success (s, input[s.Length..])
        else Failure $"Expected {s}" )

[<Fact>]
let ``test str`` () =
    test <@ run (str "foo") "foo-then something else" = Success ("foo", "-then
something else") @>
    test <@ run (str "foo") "notfoo--then something else" = Failure "Expected foo" @>
```

Keep `str` in mind: we are going to use it over and over. You can use it as a building block to parse more refined parsers. For example, here's how to define booleans in your language. With a stroke of genius you take the decision to have three-state booleans:

```
type Boolish =
    | SoTrue
    | SoFalse
    | Occasionally
```

Which keywords should your language use? `"true"` and `"false"` are so overrated and boring... What about using German instead?

| Boolish value | Keyword |
|---|---|
| `SoTrue` | `richtig` |
| `SoFalse` | `falsch` |
| `Occasionally` | `gelegentlich` |

Sounds promising! Building a parser for the keyword `falsch` is straighforward if you use `str`:

```
let falschStr: string Parser = str "falsch"

[<Fact>]
let ``parsing the string 'falsch'`` () =

    test <@ run falschStr "falsch as a 3 dollar bill" = Success ("falsch", " as a 3
dollar bill") @>
    test <@ run falschStr "if(2+2=5 -> gelegentlich) then foo()" = Failure "Expected
falsch" @>
```

We are not done yet. `falschParser` is a `String Parser`, not a `Boolish Parser`. `|>>` to the resque!

```
let boolishFalscheP = falschStr |>> (fun _ -> SoFalse)

[<Fact>]
let ``parsing the string 'falsch as an instance of Boolish`` () =
    test <@ run boolishFalscheP "falsch as a 3 dollar bill" = Success (SoFalse, " as
a 3 dollar bill") @>
```

## F-word

Do you realize know what you have just accomplished? You have invented Functors! In order to understand this better, we need to examine the `<<|` signature and to reinterpret it through a new lens. Here's the implementation:

```
let (<<|) (f: 'a -> 'b) (ap: 'a Parser) =
    Parser (fun input ->
        match run ap input with
        | Success (a, rest) -> Success (f a, rest)
        | Failure s -> Failure s )
```

and here the signature:

```
val (<<|) : ('a -> 'b) -> 'a Parser -> 'b Parser
```

Given that all the functions in F# are curried, there are 2 ways to read it:

| Input | Output | Interpretation |
|---|---|---|
| `('a -> 'b) -> 'a Parser` | `'b Parser` | Apply a function to the result of a parser |
| `('a -> 'b)` | `'a Parser -> 'b Parser` | Lift a function to the Parser world |

You can easily apply the 1st interpretation to the `EpicTime` case. You have a function from `DateTime` (the sad cat) to `EpicTime` (the happy, punk cat):

```
val toEpicTime : DateTime -> EpicTime
```

and you wanted to apply it to *the value inside the* `DateTime Parser` box (the glass can):

```
val dateTimeP: DateTime Parser
```

You can do this with:

```
let epicTimeP : EpicTime Parser =
    toEpicTime <<| dateTimeP
```

The types in the game are:

| Element | Signature |
|---|---|
| `toEpicTime` | `DateTime -> EpicTime` |
| `dateTimeP` | `DateTime Parser` |
| `<<\|` | `(DateTime -> EpicTime) -> DateTime Parser -> EpicTime Parser` |

You can imagine how `<<|`:

- Gets the glass can with the sad `DateTime` cat.
- Opens the can freeing the cat.
- Applies `toEpicTime` to the sad `DateTime` cat turning it into a happy, punk `EpicTime` cat.
- Then, secures the cat back in the glass can.

Generally: `<<|` lets you apply a function to "the content" of a box. This box is called Functor and `<<|` is also called `map`. You will often hear expressions such as "A Functor is something you can map over".

The second — more fascinating and powerful — interpretation arises from the second signature interpretation. Or as soon as you partially apply `<<|`:

```
let toEpicTimeP = (<<|) toEpicTime
```

Wait, having an operator used in prefix fashion is a bit weird. Let's define an alias before proceeding. We can call it either `map` or `lift`, and the reason will be immediately clear:

```
let map = (<<|)

let toEpicTimeP = map toEpicTime
```

Read `map`'s signature using the 2nd interpretation, as:

```
val map: ('a -> 'b) -> ('a Parser -> 'b Parser)
```

`map` is that combinator that given a function `f : 'a -> 'b` operating on ordinary values *lifts* it to work on parsers, as an `'a Parser -> 'b Parser` function. It maps things from the lower world to the elevated, Parser-powered universe.

Applied to our case:

- You have `toEpicTime: DateTime -> EpicTime`.
- You lift `toEpicTime` with `map`:

```
let epicTime = map toEpicTime
```

- It is transformed to `toEpicTimeP: Parser DateTime -> Parser   EpicTime`.

When you can feed it with a `DateTime Parser`:

```
let epicTimeParser = toEpicTimeP dateTimeP
```

you will give you back an `EpicTime Parser`:

```
let toEpicTimeP = map toEpicTime
```

```
[<Fact>]
```

```
let ``applying a lift toEpicTime`` () =
    test <@ run (toEpicTimeP dateTimeP) "2025-01-01 18:11:12, the rest" = Success
(EpicTime 626140030.0, ", the rest") @>
```

Wow… That was a mouthful, wasn't it? Not only are Functors incredibly powerful, but they are also pervasive in Functional Programming, forming a fundamental building block. During your journey, you will surely encounter them in many other contexts.

We've covered many details about transforming a single parser and adapting it to our needs, but we haven't yet looked at sequencing two parsers in a row.

It's time to take a break, enjoy a salmiakki, and then jump to chapter 8 to explore this new topic.

Previous - Mapping the Journey ~ Next - Here Comes The Tuple

## References

- F# source code: `<|` operator
- F# source code: `|>` operator
- Tacit Programming
- Epoch time

## Comments

GitHub Discussions

{% include fp-newsletter.html %} Very well. We have a powerful tool at our disposal for applying functions to the content of a parser. The biggest limitation is that it works with a single parser only. In this installment we will distill one of the simplest approaches for combining 2 parsers together.

### Variable Assignment

Let's develop another syntax costruct of your stunning language. Torn between using either `=` or `:=` for variable assignment, you resolve the dilemma by not using any symbol whatsoever. A simple:

```
42foo
```

will assign the `int` value `42` to the variable `foo`. Terrific.
Internally, you want to store the parsed value in an instance of type `IntVariable`:

```
type VariableName = VariableName of string

type Assignment =
  { variableName: VariableName
    value: int }
```

If you had a parser returning an `(int * VariableName)` tuple, building an `Assignment` parser would be a breeze, with the help of your old friend `map`:

```
// Test parser, magically returning (42, VariableName "foo")
let tupleP: (int * VariableName) Parser =
    Parser(fun _ -> Success((42, VariableName "foo"), "rest"))

let assignmentP: Assignment Parser =
    tupleP
    |>> (fun (value, name) -> { variableName = name; value = value })

[<Fact>]
let ``from tuple to Assignment`` () =
    let expected: Assignment =
        { variableName = VariableName "foo"
          value = 42 }

    test <@ run assignmentP "42foo" = Success(expected, "rest") @>
```

How to build a real `tupleP`, though, is a whole different story. Even if you already had both an `int` parser and a `VariableName` parser, how to compose them to produce an `(int * VariableName)` parser is not immediately evident. The reason why this exercise is worth to be done is because its generalization leads to the discovery of the next important building block in our functional programming journey: the Applicative Functor. Let's see.

## Let There Be Tuples

We can start designing this combinator from the desired signature. We want it to be as generic as possible, therefore we assume that we start from an `'a Parser` and a `'b Parser`. Since there is a notion of parsing elements in a sequence, we will call it `andThen`:

```
let andThen<'a, 'b>: 'a Parser -> 'b Parser -> ('a * 'b) Parser = __
```

By the way, we use a tuple as the return value because in the most general case `'a` and `'b` are different types: putting non homogeneus value in a list or an array is just not possible (F# is not Ruby). We could also use a class or a record, but tuples are simpler. Let's go with them.

The conventional operator symbol for `andThen` is `.>>.`:

```
let (.>>.) = andThen
```

Let's have a test for guiding the implementation:

```
type VariableName = VariableName of string

type Assignment =
    { variableName: VariableName
      value: int }

[<Fact>]
let ``combine 2 parsers generating a parser of tuples`` () =
    let intP : int Parser =
        Parser (fun input -> Success (42, input[2..]))

    let variableNameP : VariableName Parser = str "foo" |>> VariableName

    let tupleP = intP .>>. variableNameP

    test <@ run tupleP "42foo the rest" =
        Success ((42, VariableName "foo"), " the rest") @>
```

Of course, in the test we don't care how `intP` and `variableNameP` work, so it's fine to give them a dummy, hardcoded implementation. As for the implementation of `andThen`, as usual we can let types drive us. We know we have to return a `Parser`. So, let's build one:

```
let andThen<'a, 'b>: 'a Parser -> 'b Parser -> ('a * 'b) Parser =
    Parser ...
```

The Case Constructor wants a function from `input: string`. Let's go:

```
let andThen (aP: 'a Parser) (bP: 'b Parser): ('a * 'b) Parser =
    Parser (fun input ->
        ...)
```

OK. In case of success, we have to return a tuple `(valueA, valueB)`, together with the unconsumed input. How to obtain `valueA`? We have an `'a Parser`, we have an input. That's easy, with `run`:

```
let andThen<'a, 'b> (aP: 'a Parser) (bP: 'b Parser): ('a * 'b) Parser =
    Parser (fun input ->
        let resultA = run aP input
        ...
```

It's fair to assume that if `aP` fails, the whole `andThen` must also fail:

```
let andThen<'a, 'b> (aP: 'a Parser) (bP: 'b Parser): ('a * 'b) Parser =
    Parser (fun input ->
        let resultA = run aP input
        match resultA with
        | Failure f -> Failure f
        ...
```

If `aP` succeeds, it returns the parsed value `valueA` (the first part of the tuple you want to return) plus the unconsumed input `restA`, :

```
let andThen<'a, 'b> (aP: 'a Parser) (bP: 'b Parser): ('a * 'b) Parser =
    Parser (fun input ->
        let resultA = run aP input
        match resultA with
        | Failure f -> Failure f
        | Success (valueA, restA) ->
            ...
```

We are almost done. With `restA` it's easy to also run the second parser `bP`:

```
let andThen<'a, 'b> (aP: 'a Parser) (bP: 'b Parser): ('a * 'b) Parser =
    Parser (fun input ->
        let resultA = run aP input
        match resultA with
        | Failure f -> Failure f
        | Success (valueA, restA) ->
            let resultB = run bP restA
            ...
```

Same story here: should `bP` fail, we let `andThen` fail; otherwise, we successfully return the tuple:

```
let andThen<'a, 'b> (aP: 'a Parser) (bP: 'b Parser): ('a * 'b) Parser =
    Parser (fun input ->
        let resultA = run aP input
        match resultA with
        | Failure f -> Failure f
        | Success (valueA, restA) ->
            let resultB = run bP restA
            match resultB with
            | Failure f -> Failure f
            | Success (valueB, restB) -> Success ((valueA, valueB), restB))
```

You can make the whole expression slightly shorter like this:

```
let andThen<'a, 'b> (aP: 'a Parser) (bP: 'b Parser) : ('a * 'b) Parser =
    Parser(fun input ->
        match run aP input with
        | Failure f -> Failure f
        | Success(valueA, restA) ->
            match run bP restA with
            | Failure f -> Failure f
            | Success(valueB, restB) -> Success((valueA, valueB), restB))
```

You are done! Keep `.>>.` in your tool belt, it will come in easy very often.
Armed with `andThen` / `.>>.` and `|>>`, you can finally build the `Assignment` parser:

```
let intP: int Parser = Parser(fun input -> Success(42, input[2..]))

let variableNameP: VariableName Parser = str "foo" |>> VariableName

let assignmentP =
    intP .>>. variableNameP
    |>> (fun (i,v) -> { variableName = v; value = i })


[<Fact>]
let ``combine 2 parsers generating a parser of tuples`` () =

    let expected = {variableName = VariableName "foo"; value = 42}
    test <@ run assignmentP "42foo the rest" = Success(expected, " the rest") @>
```

Nice! You did it!

## Umpf

Can I say something? This syntax:

```
let assignmentP =
    intP .>>. variableNameP
    |>> (fun (i,v) -> { variableName = v; value = i })
```

just sucks. I swear that there are occasions where `.>>.` shines. I also swear that you will eventually get used to such succint, operator dense expressions. However, I am sure that you are happy to know that in the next chapters you will learn how to write `andThen` / `.>>.` using a completely different syntax:

```
let andThen aP bP =
    parser {
        let! a = aP
        let! b = bP
        return (a, b) }
```

Isn't it just easier to interpret? Funny enough, you will also learn to write it in a more concise way like this:

```
let andThen = lift2 (fun a b -> (a, b))
```

which will lead you to understand the mindblowingly short Haskell version:

```
let andThen = liftA2 (,)
```

But be patient, we will get there. I guess you can reward yourself with a slice of castagnaccio and then move to Chapter 9, where we will play with the idea of ignoring parsers. Buon appetito!

Previous - Parser-Powered Function Application ~ Next - Things You Don't Care About

## Comments

GitHub Discussions

{% include fp-newsletter.html %} There is another elementary way to sequence 2 parsers: to only return the parsed value of one and just ignoring the value of the other.

WAT? Why one would possibly wish to do that? What's the point of parsing something only to discard its value? Well, think about it: I said "ignoring the parsed value of the other", not "ignoring the other" altogether. I mean, you will still run the parser you are going to ignore the result of. It will still fail if it does not like the input you are giving it. Indeed, it is perfectly legit that, while processing an input string, you wish something specific to be present, and not to be interested in getting back its value.

You have already strumbled upon this case. Remember when in Chapter 2 we imagined a possible (exquisite) syntax for a record:

```
inst Person
   - Id <- *b19b8e87-3d39-4994-8568-0157a978b89a*
   - Name <- <<Richard>>
   - Birthday <- date{16/03/1953}
```

to be parsed into:

```
type Person =
    { Id: Guid
      Name: string
      Birthday: DateOnly }
```

Focus on the `Id` field. You defined that its GUID value must be surronded by `*`. You want to make sure that exactly one leading `*` and one trailing `*` are found, and you want your parser to raise an error if that's not the case. Yet, in order to build a `Guid`, you are only interested in capturing the value `b19b8e87-3d39-4994-8568-0157a978b89a` between the stars. The `*` values themselves don't contribute to the resulting value.

To build the parser for a GUID you needs to give it this recipe:

- Make sure there is a `*`. Raise an error if you don't find it. If you find one, don't even bother store the result, I don't need it.
- Capture an `xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx` string. Do remember its value, I do need it!
- Make sure there is another `*`. Again, either raise an error or just go ahead without storing its value.

You remember how in Chapter 6 we imagined to build:

```
let surroundedBy before after parser = before >>. parser .>> after
```

and how I told you to read `a >>. b` as:

- give me a parser that
- expects whatever the parser `a` expects
- then continues parsing whatever `b` is good at parsing
- and, finally, returns only the thing parsed by `b`, dropping the result of `a`.

That's exactly the same use case. I guess it's time to build `>>.` and `.>>`. It won't be hard.

## Keep The Right!

Let's start with `>>.`. Notice the `.` on the right side? It's the hint that this operator runs both the parsers, but only keeps the result of the right one. Here's how you could test it:

```
type Prefix = Prefix
type Content = Content of int
```

```
let (>>.) leftP rightP = failwith "Not yet implemented"

[<Fact>]
let ``keep right only`` () =
    let prefixP = str "the prefix/"
    let contentP = str "the content"

    let prefixedP = prefixP >>. contentP

    test <@ run prefixedP "the prefix/the content/the rest" =
        Success("the content", "/the rest") @>
```

If you think about it, the implementation must be very similar to the one of `.>>.`. Only, rather than returning a tuple with both the values, you can return the right value only. If you Copy/paste `.>>.`, you will not struggle to modify it as:

```
let (>>.) leftP rightP =
    Parser(fun input ->
        let resultL = run leftP input

        match resultL with
        | Failure f -> Failure f
        | Success(_, restL) ->
            let resultR = run rightP restL

            match resultR with
            | Failure f -> Failure f
            | Success(valueR, restR) -> Success(valueR, restR))
```

Test green! Bravo!

## Composing Parsers-based Functions

Is there a better alternative to this implementation? I argue: every time that some code is heavily based on copypasta, you can bet your bottom dollar that there is shorter and better alternative: most of the times you will win. Let's think about it:

- if `>>.` is like `.>>.`, but returning the second element of the tuple only,
- and if `snd` is the function returning the second element of a tuple,
- then `>>.` can be thought as the composition of `.>>.` and `snd`.

Interesting. What does it mean to compose 2 functions each returning a parser? Quick review how to compose ordinary functions. If you have:

```
val f : 'a -> 'b
val g : 'b -> 'c
```

then:

```
val fComposedG : 'a -> 'c

let fComposedG = fun a -> g(f(a))
```

You could conceive an operator for this:

```
// ('a -> 'b) -> ('b -> 'c) -> ('a -> 'c)
let (>>) f g = fun a -> g(f(a))
```

```
[<Fact>]
let ``function composition`` () =
    // string -> (string * int)
    let mkTuple (s: string) = (s, s.Length)

    // (string * int) -> int
    let snd (_, b) = b

    let composed = mkTuple >> snd

    test <@ "abcd" |> composed = 4 @>
```

Indeed, this operator is natively provided by F# (FSharp.Core/prim-types.fs#L4546):

```
let inline (>>) func1 func2 x = func2 (func1 x)
```

Let's give it a try, on our Parser-returnign functions:

```
let (>>.) leftP rightP =
    leftP .>>. rightP
    >> snd
```

Uhm, no… This does not even compile. Of course it does not! A parser is not a plain function; it's a function wrapped in a type. You need a different operator, with this signature:

```
val combineP : 'a Parser -> ('a -> 'b) -> 'b Parser
```

Well well well, look at that! This is our friend `|>>`, the dual of Functor's `map` which we developed in Chapter 7! Let's see if it works:

```
let (>>.) leftP rightP =
    leftP .>>. rightP
    |>> snd
```

Indeed, this compiles, and the test is green.
Wait a minute! Does it mean that using `fst` instead of `snd` we will obtain `.>>` as well? Let's see:

```
let (>>.) leftP rightP =
    leftP .>>. rightP
    |>> fst


[<Fact>]
let ``keep left only`` () =
    let contentP = str "the content"
    let suffix = str "/the suffix"

    let prefixedP = contentP .>> suffix

    test <@ run prefixedP "the content/the suffix/the rest" =
        Success("the content", "/the rest") @>
```

Yes! Green! And, by the way: it's again more test code than implementation. A very good sign!

## Feeling Surronded

I guess you see the pattern here:

- You started writing very low-level building blocks such as `|>>` and `<<|`.
- Those combinators gave you the chance to encapsulate the structural traits (passing unconsumed input and handling errors) once for all.

- Now you are building other higher level building blocks just combining the existing ones, without repeating yourself, ending up with very concise code.

Let's keep flying in this direction, building on top of `.>>` and `>>.`: let's invent a combinator for ignoring the elements surronding something you want to parse. You saw it already in Chapter 6. Its signature is:

```
val between<'o, 'c, 'v> : 'o Parser -> 'c Parser -> 'v Parser
```

where:

- `'o Parser` detects the opening element.
- `'c Parser` detects the closing element.
- `'v Parser` parses the actual value you are interested in.

This could be used, for example, for parsing a date inside an XML tag:

```
let between opening closing content = failwith "Not yet implemented"

[<Fact>]
let ``date in tags`` () =

    let o = str "<birthday>"
    let c = str "</birthday>"
    let dateOnlyP = Parser (fun input ->
        Success(DateOnly.Parse(input[..9]), input[10..]))

    let contentInTagsP = dateOnlyP |> between o c

    test <@ run contentInTagsP "<birthday>2025-12-11</birthday>the rest" =
                Success (DateOnly(2025,12,11), "the rest")@>
```

The implementation is straighforward:

```
let between opening closing content =
    opening >>. content .>> closing
```

That's it. Green test.
Note the parameters of `between`: they are not simply the strings you want to act as boundaries; instead, they are themselves parsers. You understand what this means: they can be arbitrarily complex. If you managed to develop a parser for a whole code block and a parser for comments, you can easily define a parser that detects a comment between 2 arbitarily complex code blocks:

```
let commentBetweenBlocksP = commentP |> between codeBlockP codeBlockP
```

It is that easy.
As an outrageously useless example, here's how to parse a string surrounded by dates:

```
[<Fact>]
let ``greeting between dates`` () =

    let helloP = str "Hello!"
    let dateOnlyP = Parser (fun input ->
        Success(DateOnly.Parse(input[..9]), input[10..]))

    let contentInTagsP = helloP |> between dateOnlyP dateOnlyP

    test <@ run contentInTagsP "2025-12-11Hello!2025-12-11 the rest" = Success
("Hello!", " the rest")@>
```

I hope this quirky example doesn't give you any wild ideas for funny language syntax constructs. Instead, please: take a break, enjoy a kiwi, and carry on with Chapter 10. I can only recommend not to eat much and stay light: we are going to apply functions ad nauseam.

Previous - Here Comes The Tuple ~ Next - Applying Functions, Ad Nauseam

## References

- F# source code: >> operator

## Comments

GitHub Discussions

{% include fp-newsletter.html %} There is something magic about the native F# function application: once you apply a function to an argument, if the result is another function, you can just run function application again. And if you get yet another function, you can do the same, ad nauseam.
See this example:

```
[<Fact>]
let ``function application with 2 parameters`` () =

    let fa: int  -> (bool -> string) = fun a -> fun b -> $"{a}, {b}"
    let fb:          bool -> string  = fa 42
    let c:                   string  = fb true

    test <@ c = "42, True" @>
```

- `fa` is a function that takes `a`, an `int`, and that returns another function, `bool -> string`.
- If you use the native function application to apply `f` to 42, you get `fb: bool -> string` back.
- Now, thare's nothing special in that `bool -> string` returned value: it is just another function. So, you can keep using the F# native function application to pass it the next argument, a `true` value, which finally gets you back a `string`.

It's easy to see how this does not stop with 2-parameter functions. Here's a test for a 3-parameter function:

```
[<Fact>]
let ``function application with 3 parameters`` () =

    let fa: int -> (bool -> (string -> string)) =
        fun a -> fun b -> fun c -> $"{a}, {b}, {c}"

    let fb: bool -> (string -> string) = fa 42
    let fc: string -> string = fb true
    let d: string = fc "foobar"

    test <@ d = "42, True, foobar" @>
```

Now:

```
let f:  int  -> (bool -> (string -> string)) =
    fun a -> fun b -> fun c -> $"{a}, {b}, {c}"
```

is a very verbose way to define a function returning a function, in turn, returning a function. F# lets you:

- remove the parenthesis from the signature, since function application associates to the right:

```
let f:  int  -> bool -> string -> string =
    fun a -> fun b -> fun c -> $"{a}, {b}, {c}"
```

- skip that `fun a -> fun b -> fun c` boiler plate and just write `f a b c`. This can be expressed saying that in F# all the functions are automatically curried:

```
let f a b c = $"{a}, {b}, {c}"
```

- mentally see this as a function of 3 parameters.
- apply it to multiple parameters in a single shot, with:

```
let d = f 42 true "foobar"
```

Ah, much better! But, note: it's just syntactic sugar. This is still a function returning a function — in turn, returning a function.

## A Crocked Function Application

What about the Parser-Powered Function Application `<<|` / `map` that we have so proudly distilled in Chapter 7? Can we also apply it *ad nauseam*?

Let's see. We start from a generic 3-parameter function `'a -> 'b -> 'c -> 'd`. In this context, we are not concerned how it is implemented, we can just focus on its signature:

```
[<Fact>]
let ``Parser-powered function application with 3 parameters`` () =

    let f (a: int) (b: 'b) (c: 'c) = __
    ...
```

Then, we apply it to an argument `'a Parser`, of course using `map`. Let me use the symbol `<!>` instead of `<<|` or `map`; after all, we mentioned that they are all synonyms:

```
let (<!>) = map

[<Fact>]
let ``Parser-powered function application with 3 parameters`` () =

    let a: 'a Parser = __

    let f (a: int) (b: 'b) (c: 'c) = __

    let fa: ('b -> 'c -> 'd) Parser =
        f <!> a

    ...
```

`<!>` allows passing an `'a Parser` to a function expecting an `'a`. But, darnit! Look at the resulting `fa`'s signature! The result is not just another function with 1 parameter less, like it happened before . It's not even a function anymore: it's a function wrapped inside a Parser. If you think what `<!>`'s purpose is, this makes sense. If you apply `<!>` to an `'a -> 'b` function, you get this:

Focus on the returned value, in this case `'b`: it ends up being wrapped in a `Parser`.
Now, if you think to a 2-parameter function `'a -> 'b -> 'c` as a 1-parameter function `'a -> ('b -> 'c)` — so as a function which just happens to return another function — then applying `<!>` gets you this:

This means that we cannot apply `<!>` again, ad nauseam…

Does it mean that we need a different operator? Yes, that's exactly the point! It could be demonstrated that for such cases Functor's `map` is of little help: there is no possible way to perform the next function application only using `map`'s capabilities. It's time to invent a more powerful version of Functors: enter Applicative Functors.

## Beyond Functors

You already guessed the next steps: we will implement a new operator, dedicating it yet another symbol, and letting its signature lead the way. Then, hopefully, we will manage to use the new operator to express, in a smarter and more concise way, some of the things we have distilled so far. Ideally, we could discover that the new operator is so powerful to incorporate `map` itself.
Without further ado, let's distill <*>. With a burst of creativity, we will call it "apply" or `ap`.

Let's recover from where we left:

```
[<Fact>]
let ``Parser-powered function application with 3 parameters`` () =

    let aP: 'a Parser = __
    let bP: 'b Parser = __
    let cP: 'c Parser = __

    let f (a: int) (b: 'b) (c: 'c) = __

    let fa: ('b -> 'c -> 'd) Parser = f <!> aP
    ...
```

We want to apply `fa`, a 2-parameter function inside a Parser, to the next argument, a `'b Parser`. In order to proceed, let me use a little syntax maneuver, so that the result will resemble the native F# function application: hopefully, this will let us see what's going on in a more streamlined way. With the native F# function application, when you have a multiparameter function:

```
let f: 'a -> 'b -> 'c -> 'd = __
```

you can apply it to arguments just by separating them with white spaces:

```
let d = f a b c
```

With a bit of imagination, you can think to those white spaces as an native F# pseudo-operator, as we did with `map`. We got to:

```
let dP = f <!> aP ...
```

The idea is to keep running function application using an improved, Parser-powered <*> operator:

```
let dP = f <!> aP <*> bP <*> cP
```

You see the equivalence?

```
let d:  'd         = f     a      b      c
let dP: 'd Parser  = f <!> aP <*> bP <*> cP
```

Basically, we are writing an enhanced version of whitespace.

If you don't like the fact that <*> is used for all the arguments but the first one, you might prefer this alternative syntax:

```
let dP: 'd Parser  = pure' f <*> aP <*> bP <*> cP
```

It's not hard to verify that it's completely equivalent. Anyway, implementing <*> is not hard at all. You just have to be driven by the type signature. Let's start with the simplest case of 1-parameter functions. `ap` / <*> is that operator that given a 1-parameter function wrapped in a Parser:

```
val ap : ('a -> 'b) Parser -> ...

let (<*>) = ap
```

lets us apply the wrapped `'a -> 'b` to an `'a Parser` argument:

```
val ap : ('a -> 'b) Parser -> 'a Parser -> ...
```

What will this give us back? Let's think about it. Naturally, we cannot get back a `'b` value: a Parser is a promise of a value, so if we give parsers it's fair to be paid back with other parsers. It's legitimate to assume we get back a `'b Parser`:

```
val ap : ('a -> 'b) Parser -> 'a Parser -> 'b Parser
```

Let's implement it:

```
let ap fP aP = __

let (<*>) = ap

[<Fact>]
let ``ap with a 1-parameter function`` () =
    let aP = Parser (fun input -> Success(42, input))

    let fP = Parser (fun input -> Success (f, input))

    test <@ run (fP <*> aP) "some input" = Success (84, "some input")@>
```

Not how, to keep things simple, we are using 2 trivial parsers, which do not even consume the input: `aP` just returns a Parser-wrapped 42, `fP` a Parser-wrapped `fun i -> i * 2`. For cases like these, we can have a convenience function `pure'`, which takes whatever value you give it in and wraps it into a doing-nothing Parser:

```
let pure' a = Parser (fun input -> Success (a, input))

// ('a -> 'b) Parser -> 'a Parser -> 'b Parser
let ap fP aP = __

[<Fact>]
let ``ap with a 1-parameter function`` () =
    let aP = pure' 42

    let fP = pure' (fun a -> a * 2)

    test <@ run (fP <*> aP) "some input" = Success (84, "some input")@>
```

You know how to proceed. The signature of `ap` tells you to return a `'b Parser`. Just build one:

```
let ap fP aP = Parser (fun input ->
    ...)
```

Now, have a function `f` inside a Parser (`fP`) and the argument `a` also inside a Parser (`aP`). You also have an `input`. Using the box analogy, it seems that solving this riddle is a matter of:

- opening both the boxes;
- extracting the contained `f` and `a`;

- applying `f` to `a`;
- possibly giving up in case of failure;
- and passing the unconsumed input around;
- then, successfully return the result.

Conventionally, the first box to open is the one containing the function:

```
let ap fP aP = Parser (fun input ->
    match run fP input with
    | Failure e ->  Failure e
    | Success (f, rf) ->
        ...
```

Of course, if we get an error, we give up. If we are successful, we get the inner function `f` the unconsumed input `rf` ("rest of `f`"). Fine, we have all the ingredients to open the `a` box:

```
let ap fP aP = Parser (fun input ->
    match run fP input with
    | Failure e ->  Failure e
    | Success (f, rf) ->
        match run aP rf with
        | Failure s -> Failure s
        | Success (a, ra) -> ...
```

Cool. We have `f`, we have `a` and the unconsumed input `ra`. Time to finally apply `f` to `a`, and to wrap the result in a `Success`:

```
let ap fP aP = Parser (fun input ->
    match run fP input with
    | Failure e ->  Failure e
    | Success (f, rf) ->
        match run aP rf with
        | Failure s -> Failure s
        | Success (a, ra) -> Success (f a, ra))
```

Green tests.

## Dealing with Details, Again?

Wait a second: why did we have to pattern match and to pass unconsumed input around? Didn't we say that we could always build on top of the previous building blocks?
As I anticipated, it turns our that the Applicative Functor's `<*>` operator cannot be built in terms of humble Functor's `map`. This could even be demonstrated mathematically. Even further: we can easily show that, since Applicative Functors are more powerful than Functors, we can rewrite `map` in terms of `pure'` and `ap`:

```
let map f aP = pure' f <*> aP
```

This `map` implementation may look obscure, but it is in fact very logic. If you compare the signatures of `map` and `ap`:

```
val map:  ('a -> 'b)        -> 'a Parser -> 'b Parser
val ap:   ('a -> 'b) Parser -> 'a Parser -> 'b Parser
```

you see that the only difference is that in `map` the `f` parameter is not wrapped inside a Parser. So, if you lift `f` inside a parser with `pure' f`, you would get exactly the `ap` signature; that is, you can proceed applying `ap` to the `'a Parser` argument.

Try running all the past tests you wrote so far. Woah! What a beautiful display of green! It seems that with `pure'` and <*> you really discovered something deep.

## Apply, Apply, Apply!

We opened this chapter claiming that there is something magic about the native F# function application, because you can apply a function returning a function returning a function — ad nauseam — thanks to the unsuspectedly powerful whitespace pseudo-operator.

Does your brand new <*> have the same super-power? Let's see. We want to have the equivalent of this test:

```
[<Fact>]
let ``function application with 3 parameters, inlined`` () =

    let f a b c = $"{a}, {b}, {c}"

    let a = 42
    let b = true
    let c = "foobar"

    let d = f a b c

    test <@ d = "42, True, foobar" @>
```

but using Parser arguments:

```
[<Fact>]
let ``ap with a 3-parameter function`` () =
    let f a b c = $"{a}, {b}, {c}"

    let aP = pure' 42
    let bP = pure' true
    let cP = pure' "foobar"

    let dP = f <!> aP <*> bP <*> cP

    test <@ run dP "some input" = Success ("some input", "42, True, foobar") @>
```

Amazing! It works!

## Applicative Functors

I did not introduce `pure'` as a coincidence: in fact, the combination of `pure'` and <*> is what conventionally defines Applicative Functors. There are 2 possible interpretation of Applicative Functors. The first is about what we have just experimented: seeing them as an extension of function application. It should not come as a surprise that we have made every effort to ensure that using Applicative Functors resembled just applying functions:

```
let d:  'd         = f     a      b      c
let dP: 'd Parser  = f <!> aP <*> bP <*> cP
```

I have always been fascinated by the way the Idris programming language took this interpretation to the extreme, with Conor McBride's Idiom Brackets:

```
d = [| f aP bP cP |]
```

Like this, it really seems an ordinary function application! But enough for now. Time to get our hands dirty building some real parsers with <*> and `pure'`, and to play with the second interpretation of Applicative Functors, that has to do with the notion of lifting functions. Treat yourself with a rösti and get prepared to Chapter 11.

Previous - Things You Don't Care About ~ Next - Lifting Functions

## References

- Idris
- Idris - Idiom Brackets

## Comments

GitHub Discussions

{% include fp-newsletter.html %}

## Comments

GitHub Discussions

{% include fp-newsletter.html %}

### Fancy dates

Do you remember when in Chapter 6 we fantasized about the syntax:

```
7 times date{16/03/1953}
```

to build a list of `7` dates boxed inside a `MultiDate` object? Such a stunning feature deserves a parser, and Applicative Functors can make it happen. Let's design it top-down, fully applying the idea that complex parsers can be thought as a combination of even simpler parsers, down to the smallest parsers you can create. Of course, we start from a test:

```
type MultiDate = MultiDate of DateOnly list

let multiDateP: MultiDate Parser = __

[<Fact>]
let ``parses a MultiDate`` () =
    let input = "7 times date{16/03/1953} the rest"

    let date = DateOnly(1953, 03, 16)
    test <@ run multiDateP input =
        Success(MultiDate [ date; date; date; date; date; date; date ], " the rest")
@>
```

Applicative Parsers let you think in terms of values, rather than in terms of parsers, and then feed your functions with parsers using <!> and <*>. So, the idea is to split the input `7 times date{16/03/1953}` into its syntactical components:

- `7`: the number.
- `times`: one of your language's commands.
- `date{16/03/1953}` to be parsed as `DateOnly(1953, 03, 16)`.

and to define a builder function in terms of those ordinary parsed values, postponing the problem of defining their parsers:

```
let makeMultiDate (n: int) (_command: string) (date: DateOnly) : MultiDate =
    let dates = List.replicate n date
    MultiDate dates
```

Note that `_command` is ignored. In fact, we want the parser to make sure the required string is found, but not to include it in the returned value.
Armed with the factory `makeMultiDate`, we can apply it to parsers using `<!>` and `<*>`:

```
let intP: int Parser = __
let times: string Parser = __
let fancyDate: DateOnly Parser = __

let multiDateP =
  makeMultiDate <!> intP <*> times <*> fancyDate
```

Since we are proceeding top-down, we can proceed building the underlying parsers. `times` is trivial, it's a string parser for the hardcoded value `" times "`:

```
let times = str " times "
```

Let's go with `fancyDate` next. To parse `date{16/03/1953}` as a `DateOnly` we need to make this test pass:

```
let fancyDate: DateOnly Parser = __


[<Fact>]
let ``parses the fancy date syntax`` () =
  let input = "date{16/03/1953} the rest"

test <@ run fancyDate input = Success(DateOnly(1953, 03, 16), " the rest") @>
```

You have all the necessary building blocks, if you observe that `date{16/03/1953}` can be thought as:

- the date `16/03/1953`
- surrounded by `{` and `}`
- and prefixed by `date`, whose value can be ignored.

You can use `between` for `{` and `}`, and `>>.`:

```
let openBrace = str "{"
let closeBrace = str "}"

let date: DateOnly Parser =  __

let fancyDate =
    str "date" >>. (date |> between openBrace closeBrace)

[<Fact>]
let ``parses the fancy date syntax`` () =
  let input = "date{16/03/1953} the rest"

  test <@ run fancyDate input = Success(DateOnly(1953, 03, 16), " the rest") @>
```

Here we assume we are using the already implemented `>>.` and `between`. But if you think about it, they are both very simple to define, using an Applicative Functor:

```
let (>>.) left right =
    let takeRight _ right = right
    takeRight <!> left <*> right

let between openTagP closedTagP contentP =
    let buildBetween _ content _ = content
    buildBetween <!> openTagP <*> contentP <*> closedTagP
```

In both cases, it's a matter of writing an ordinary factory function, and then of applying it to parsers, using `<!>` and `<*>`. It's alway the same trick, over and over.
Going deeper, it's `date`'s turn. `date` is the parser for the inner `16/03/1953` syntax. It can be defined — guess how? — using an Applicative Functor, feeding its factory method `makeDateOnly`:

```
let digitsP (nDigits: int) : int Parser = __

let slash = str "/"
let day = digitsP 2
let month = digitsP 2
let year = digitsP 4

let makeDateOnly day _slash1 month _slash2 year =
  DateOnly(year, month, day)

let dateP =
    makeDateOnly <!> day <*> slash <*> month <*> slash <*> year


[<Fact>]
let ``parses the date part`` () =
  let input = "16/03/1953 the rest"

  test <@ run dateP input = Success(DateOnly(1953, 03, 16), " the rest") @>
```

Dropping down another level (we are almost done!) we have to define `digitsP`. This function takes the expected number of digits and returns an `int Parser` to parse a number with exactly that many digits. At this stage, we haven't yet developed the ideal building blocks to make this parser elegant. Let me cheat using `Int32.Parse`:

```
let digitsP nDigits = Parser (fun input ->
    try
        Success(Int32.Parse(input[..nDigits-1]), input[nDigits..])
    with
    | _ -> Failure "Could not parse the int")

[<Fact>]
let ``parses numbers with a specific number of digits`` () =

    test <@ run (digitsP 1) "9" = Success (9, "")@>
    test <@ run (digitsP 2) "42" = Success (42, "")@>
    test <@ run (digitsP 2) "42 the rest" = Success (42, " the rest")@>
    test <@ run (digitsP 4) "1942 the rest" = Success (1942, " the rest")@>
    test <@ run (digitsP 4) "19429 the rest" = Success (1942, "9 the rest")@>

    test <@ run (digitsP 4) "19 the rest" = Failure "Could not parse the int"@>
    test <@ run (digitsP 4) "foo bar baz" = Failure "Could not parse the int"@>
```

We will improve this parser in the next chapters. Incidentally, this is a key feature of Parser Combinators: because of their recursive design, improvements of any low level building block will positively propagate to all the parsers built on top of it. So, don't stress too much over the result, for now; we'll soon make it better.

We are left with one last building block to write: `intP`. It is supposed to parse the `7` in `7 times date{16/03/1953}`. We can just use `digitsP 1`, can't we?

```
let intP: int Parser = digitsP 1
```

Sure. But this is a dirty trick. How do you know that the number is a 1-digit one? What if the input string had `42` or `42000` instead of `7`? Parsing dates was an easy task, because the day's and the month's parts were always `2` digits numbers, year's part always a `4` digits one. Here we are facing a new challenge: how to parse an integer with an unknown number of digits? We don't know yet how to deal with unknown things. This require a new tool in our toolbelt, something giving our parsers the ability to *try* a parsing, and to eventually recover from a failure. We will cover exactly this in the next chapter. Instead, let's close this one with a last twist: let's invent the notion of *lifting functions*, a possible second interpretation of Applicative Functors.

## Lifting functions

We learnt that a function taking values can be applied to parsers of those values by the means of replacing the white-space pseudo-operator with `<!>` and `<*>` like this:

```
let f:  'value         = f     a      b      c
let fP: 'value Parser  = f <!> aP <*> bP <*> cP
```

In a sense, the combination of `<!>` and `<*>` elevates a function from the world of ordinary values to the parser world. This lifting happens as we provide one argument after the other. It would be nice to have an operator to perform that lifting beforehand, even before we have an argument to feed the function with. In other words, it would be amazing if we could convert:

```
f:   a -> b -> c -> d
```

into:

```
fP: 'a Parser -> 'b Parser -> 'c Parser -> 'd Parser
```

in one shot. That's the work for `lift3`:

But implementing it is a piece of cake! We don't even need a test for this, type checking will suffice:

```
let lift3 f =
    fun a b c -> f <!> a <*> b <*> c
```

or, more concisely:

```
let lift3 f a b c = f <!> a <*> b <*> c
```

`lift3` comes in handy to simplify some expressions. For example, instead of:

```
let multiDateP =
    makeMultiDate <!> intP <*> times <*> fancyDate
```

you can just write:

```
let multiDateP =
    (lift3 makeMultiDate) intP times fancyDate
```

It's like writing parser-powered code while removing all the boilerplate code from sight, so to get back the original linear, pure code. Sweet!
As the name suggests, `lift3` works for 3-parameter functions. For 2-parameter functions `lift2` is similarly defined as:

```
let lift2 f a b = f <!> a <*> b
```

Removing 1 parameter more, it's easy to define `lift`, for lifting 1-parameter functions:

```
let lift f a = f <!> a
```

But look! η-reducing this expression — that is, removing `a` and `f` from both sides — it's easy to see that `lift` is in fact our old friend `map`:

```
let lift = map
```

Given the diagram, it all makes sense.

That'll do for now! Let's take a break: you deserve a pistacchio kulfi to refresh your mind.
See you later in Chapter 12.

Previous - Applying Functions, Ad Nauseam ~ Next - Things You Are Not Sure About

## References

Eta Reduction

## Comments

GitHub Discussions

{% include fp-newsletter.html %}

## Comments

GitHub Discussions

{% include fp-newsletter.html %} There are only 2 important missing features in the Parser Combinator library you are building:

- Context Sensitivity.
- Backtracking.

*Context Sensitivity* is the ability of a parser to reference the result of other parsers. Basically, it's equivalent to equipping parsers with a form of memory. This capability will allow you to parse more complex languages and will dramatically change the way you write parsers. This requires introducing Monads. We will get there in the very next chapter.

*Backtracking* is the ability to recover from errors and to explore alternative parsing paths. This is necessary for the last use case we encountered: parsing an integer of an unknown number of digits.

Let's start from the latter.

### Alternative

The basic operator for implementing backtracking is `<|>`: its purpose is to try the parser on its left, first; if that parser fails, instead of propagating the error, it *backtracks* trying the parser on the right, using the original input. If both parsers fail, only then a parsing error is returned. We implemented it already in Chapter 3, but before introducing a `Parser` type wrapper. The implementation is straightforward:

```
let orElse tryFirst fallback  =
    Parser (fun input ->
        match run tryFirst input with
        | Success _ as first -> first
        | Failure _ -> run fallback input)

let (<|>) = orElse
```

```
type SomeResult = One | Two

[<Fact>]
let ``applies the first parser if successful`` () =
    let firstParser = Parser (fun _ -> Success (One,  "the rest"))
    let fallback = Parser (fun _ -> Success (Two, "the rest"))

    let trying = firstParser <|> fallback

    test <@ run trying "some input" = Success (One, "the rest") @>

[<Fact>]
let ``if the first parser fails, applies the fallback parser`` () =
    let alwaysFails = Parser (fun _ -> Failure "failed!")
    let fallback = Parser (fun _ -> Success (Two, "the rest"))

    let trying = alwaysFails <|> fallback

    test <@ run trying "some input" = Success (Two, "the rest") @>
```

## Choice

Naturally, you can apply `<|>` multiple times:

```
type WhateverResult = Whatever

[<Fact>]
let ``sequence of <|>`` () =
    let p1 = Parser (fun _ -> Failure "failed!")
    let p2 = Parser (fun _ -> Failure "failed!")
    let p3 = Parser (fun _ -> Failure "failed!")
    let p4 = Parser (fun _ -> Failure "failed!")
    let p5 = Parser (fun _ -> Failure "failed!")
    let fallback = Parser (fun _ -> Success (Whatever, "the rest"))

    let trying = p1 <|> p2 <|> p3 <|> p4 <|> p5 <|> fallback

    test <@ run trying "some input" = Success (Whatever, "the rest") @>
```

This invites us to conceive a combinator that tries all the parsers we feed it with:

```
let choice (parsers: 'a Parser list) : 'a Parser = __


[<Fact>]
let ``applies the first successful parser`` () =

    let failing = Parser (fun _ -> Failure "failed!")

    let p1 = failing
    let p2 = failing
    let p3 = failing
    let p4 = failing
    let p5 = failing
    let succeeding = Parser (fun _ -> Success ("the rest", Whatever))
    let p6 = failing
    let p7 = failing

    let firstSucceeding = choice [p1; p2; p3; p4; succeeding; p5; p6; p7;]

    test <@ run firstSucceeding "some input" = Success ("the rest", Whatever) @>
```

A possible recursive implementation could be:

```
let rec choice<'a> (parsers: 'a Parser list) : 'a Parser =
    match parsers with
    | [] ->
        Parser (fun _ -> Failure "No parsers succeeded")
    | [p] ->
        p
    | p :: ps ->
        p <|> choice ps
```

Amazingly, a shorter working version is:

```
let choice parsers =
    List.reduce (<|>) parsers
```

It would be nice to write it even more concisely, in Point Free Style as:

```
let choice =
    List.reduce (<|>)
```

but F# type inference would scream at us.

Technically speaking, this super-short version is based on the fact that a `Parser`, together with the binary operation `<|>` *forms a semigroup*. In simple words, this means that we managed to have an operation to reduce 2 different items into 1, and this is a very well known pattern in functional programming. Indeed, `List.reduce` is based on that pattern. Its documentation states:

```
val reduce: reduction: ('T -> 'T -> 'T) -> list: 'T list -> 'T

'T is Parser<'a>

[...]
reduction - The function to reduce two list elements to a single element.
```

This is encouraging: whenever you happen to develop a custom operator and then you discover that the standard F# library natively supports it, that's the sign that you hit the nail on the head.

**Month names**

Let's see how to apply `choice` to a concrete case. Say that you want to parse a date in the format `12 Oct 2025`. For the month part, you would like to parse:

| Input | Parse result |
|---|---|
| `"Jan"` | 1 |
| `"Feb"` | 2 |
| `"Mar"` | 3 |
| `"Apr"` | 4 |
| `"May"` | 5 |
| `"Jun"` | 6 |
| `"Jul"` | 7 |
| `"Aug"` | 8 |
| `"Sep"` | 9 |
| `"Oct"` | 10 |
| `"Nov"` | 11 |
| `"Dec"` | 12 |

Here is how we could create 12 parsers in one shot:

```
let months: string list =
    [ "Jan"
      "Feb"
      "Mar"
      "Apr"
      "May"
      "Jun"
      "Jul"
      "Aug"
      "Sep"
      "Oct"
      "Nov"
      "Dec" ]

let monthParsers: int Parser list =
    months
    |> List.mapi (fun idx month -> ((fun _ -> idx + 1) <!> (str month)))
```

Then, we can use `choice` to coalesce them in a single parser:

```
let monthParser = choice monthParsers

[<Fact>]
let ``parses a month`` () =
    test <@ run monthParser "Oct 2025" = Success (10, " 2025") @>
    test <@ run monthParser "Apr 2009" = Success (4, " 2009") @>
    test <@ run monthParser "not a month" = Failure "Expected Dec" @>
```

Note that when this parser fails, it emits the error produced by the last parser in the collection. This is less than ideal. There are techniques to improve that, but let's not get sidetracked. We have other interesting combinators to invent, first.

## Simplify Until It Can't Be Simpler

Have you noticed how, the more combinators we add to our toolbelt, the smaller is the need of developing manually-written parsers? Basically, the only parser we wrote without combinators is `str`. We distilled almost all the other ones by the application of `<!>`, `<*>`, `<|>` and their friends. Even further: it's not hard to see how `str` itself can be decomposed into smaller parsers. What about, for example, combining:

- A collection of parsers, each for a specific char (like the parser for `a`, the parser for `5` and the like).
- `choice`, the combinator we just wrote, to generate a parser for any of the provided parsers. Or even better, `anyOf`, to generate a parser for any of the provided characters. So, a parser for a digit would be simple `anyOf ['0'..'9']`.
- `many`, to keep parsing, until the input string cannot be parsed anymore. This way, `many (anyOf ['0'..'9'])` would parse any sequence of digits (including the empty one).

You see how these few tools are enough for parsing strings, digits, sequences of digits (i.e., numbers), etc. And if you take it to the extreme, you also see how, amazingly, we can decompose the parser for a specific character even more: the classic approach is to have the parser `any` for *any* character, combined with `satisfy`, a combinator that imposes some restrictions to parsers (in this case: making sure the character parsed by `any` is the desired one).
In a sense, we are scraping the bottom of the barrel, with parser combinators: the hand-made parsers we stricly need are reducing to the really trivial ones. In other words, we are seeing with our eyes how, at its core, parser design thrives on simplicity: it's an art of seeing patterns in the complexity, and of abstracting each pattern in a specific combinator. Applyuing this aproach over and over will lead us to end up with an interesting asymmetry: a very reach grammar of combinators, a next-to-empty collection of actual parsers.

Following this path, let's build `anyOf` and `many`.

### anyOf

`anyOf` is just a helper function. It takes a list of characters and it builds a parser for any of them. Under the hoods, it uses `charP`, a parser for a single character:

```
let charP (c: char) = Parser (fun input ->
    if input.StartsWith(c)
    then Success (c, input[1..])
    else Failure "Expected '{c}'" )

let anyOf chars =
    chars |> List.map charP |> choice


let digit = anyOf ['0'..'9']

[<Fact>]
let ``parses any digit`` () =
    test <@ run digit "42 the rest" = Success ('4', "2 the rest") @>
    test <@ run digit "92 the rest" = Success ('9', "2 the rest") @>
```

### many

Remember that we started investigating this very topic in the attempt of parsing an unsigned integer of an unknown number of digits. We are very close to this goal. Be ready to see a disappointing implementation, though: in fact, we are approaching the limit of what is possible with Functors and Applicative Functors, and soon we will need Monads.

The idea is:

- To build a combinator `many` that keeps trying a specific parser over and over, until it fails. The parser generated by `many` would return the list of all the successfully parsed values.

- To feed `many` with the `digit` parser we just built.
- The result will be a Parser for a list of digits. We know how to convert a list of digits to a number, so it's a matter of applying this logic by the means `map`.

Let's go.

```
let many<'a> (parser: 'a Parser): 'a list Parser = __

let toInteger (digits: char list) : int = __

let intP: int Parser = __

[<Fact>]
let ``parse numbers of any number of digits`` () =
    test <@ run intP "1 the rest" = Success (1, " the rest") @>
    test <@ run intP "42 the rest" = Success (42, " the rest") @>
    test <@ run intP "2025+7999" = Success (2025, "+7999") @>
```

Implementing `intP` it is a walk in the park:

```
let intP = many digit |>> toInteger
```

Read it as:

- `intP` is that parsers that expectes an arbitrary number of digits.
- Since the result of `many digit` is a `char list`, we need to convert it to an `int` with `toInteger`.
- But we don't have a `char list`: we have a `Parser` of `char list`. So, we need to lift `toInteger` into the parser world, using the parser-powered pipe operator `|>>`.

Implementing `toInteger` is ordinary F#, nothing to do with parsers:

```
let toInteger (digits: char list) : int =
    digits
    |> List.map string
    |> String.concat ""
    |> int

[<Fact>]
let ``from list of chars to integer`` () =
    test <@ ['4';'2'] |> toInteger = 42 @>
    test <@ ['1';'9';'9'] |> toInteger = 199 @>
    test <@ ['2';'0';'2';'5'] |> toInteger = 2025 @>
```

The last missing piece is, finally, `many`:

```
let many<'a> (parser: 'a Parser): 'a list Parser = __


[<Fact>]
let ``applies a parser many times`` () =

    let manyWell = many (str "well!")

    test <@ run manyWell "well!well!well! the rest" =
    Success(["well!";"well!";"well!"], " the rest") @>
```

Implementing `many` is actually quite challenging. We have to build a list of results, so recursion comes naturally to mind. The hard part is that we are not really building a list, but a parser emitting a list. What can help is to think that `many`, by itself, can never fail: it keeps applying a parser, and when this fails, it just stops cycling. Even if the input is empty, or if the parser immediately fails, `many` would happily succeed, returning an empty list.

Here's a possible implementation:

```
let many<'a> (parser: 'a Parser): 'a list Parser = Parser (fun input ->
    let rec zeroOrMore input =
        match run parser input with
        | Failure _ -> ([], input)
        | Success (result, rest) ->
            match (zeroOrMore rest) with
            | [], rest -> (result :: [], rest)
            | others, rest -> (result :: others, rest)

    Success(zeroOrMore input))
```

Not so easy, right? As you see, it makes use of a `zeroOrMore` inner function which does not operate in the parser world. It just executes the parser with `run`, recursing during the list building. As soon as `parser` fails, it stops.

Now, this is what I call a disappointing implementation. We have gone through 11 chapters, developing building blocks after building blocks, only to be back to square one, building `many` by the means of pattern matching and passing `rest` around. That's depressing.

Wait a minute! Can't we lift the function for recursively bulding a list to the Parser world using `<!>` and `<*>`? I mean, if building a list can be done cons-ing values with:

```
let cons head tail = head :: tail
```

can't we just lift it with:

```
let rec many parser =
    cons <!> parser <*> (many parser)
```

To be precise: this version is not quite correct, as it requires at least 1 application of parser (in the parser jargon: this is `many1`). `many` should succeeds also in case 0 applications. Easy peasy, `<|>` to the resque:

```
let rec many parser =
    (cons <!> parser <*> (many parser)) <|> (pure' [])
```

Here we go! If the first part (the lifted `cons`) fails, we just a lifted empty list.

It perfectly type checks. This is promising! "If it compiles it works", they told you. Until it does not. Try yourself: if you run the test, it enters an infinite loop. It compiled only because ⊥, or bottom, the ideal type representing never-returning functions, is a member of all the types. What a scam…

In simpler words, the problem here is that function application is eager: F# evaluates all the arguments before passing them to a function. If we had a lazy language, like Haskell, this implementation could possibly work, but that's not the case with F#.

## Many, For The Rest Of Us

What about this implementation?

```
let rec many parser =
    parse {
        let! x = parser
        let! xs = many parser
        return x :: xs
    } <|> (pure' [])
```

This is a syntax we never encountered before. I don't expect you to immediately understand it, if you never encountered do-notation. But maybe you can grasp some of it:

- It's a parser, because of that initial `parse {`.
- It returns the list `x :: xs`.
- The head `x` is somehow related to running the parser (see that `let! x = parser`).
- The tail `xs` is related to a recursive call to `many parser`.
- The last `<|> pure' []` accounts for what we saw before: `many` shall not fail if the parser `parser` cannot be applied even once.

I bet that you agree: besides the funny new syntactic elements, this version is way more linear than the original:

```
let many<'a> (parser: 'a Parser) : 'a list Parser =
    Parser(fun input ->
        let rec zeroOrMore input =
            match run parser input with
            | Failure _ -> ([], input)
            | Success(result, rest) ->
                match (zeroOrMore rest) with
                | [], rest -> (result :: [], rest)
                | others, rest -> (result :: others, rest)
         Success(zeroOrMore input))
```

Ladies and gentlemen, enter monads and monadic computation expressions. We've delayed this out long enough. It's time to open that door. The next chapter should provide the rational why and in which cases we need monadic parsers. Then, we will invent them.

Take a long break. Enjoy a Swiss cheese fondue (and take your time to digest it). We will see in Chapter 13

Previous - Lifting Functions ~ Next - Things You Want To Remember

## References

- nCatLab - Semigroup
- reduce - FSharp.Core/list.fsi#L1717
- Wikipedia - cons
- Bottom

## Comments

GitHub Discussions

{% include fp-newsletter.html %} Let's write the parser for an XML node. This is a task dressed up as a walk in the park, but it is in fact hiding an insidious maze inside. Allow me to show you why.

### Opening and closing tags

We assume that a node is any text surrounded by an opening tag — such as `<pun>` — and its corresponding closing tag — in this case `</pun>`. The parser should work with arbitrary tag names, so any of the following strings should be successfully parsed:

- `<pun>I started out with nothing, and I still have most of it</pun>`

- `<gardenPathSentence>Time flies like an arrow; fruit flies like   bananas</gardenPathSentence>1`
- `<well>well</well>`

1 Parsing a Garden Path Sentence is really a topic on its own.

For the sake of simplicity, we won't support nested nodes nor attributes. Let's say that parsing nodes should give us back instances of this record:

```
type Node =
    { tag: string
      content: string }
```

How hard can it be? The first recipe that comes into mind is:

- We parse the tag name between < and > using `between`.
- Then we parse the content, combining `many` and `anyOf`.
- Then, we parse again the tag name, this time betweeen `</` and `>`. Of course, we will use `between` again.
- Finally, we combine all that we parsed to build an instance of `Node`, either using the Applicative Functor's `<*>` or lifting the `Node` costructor with `lift3`.

Instead of `many`, which succeeds also in case of empty collections, we are going to use `many1`, a flavour of `many` requiring at least 1 successful parsing:

```
let many1<'a> (parser: 'a Parser) : 'a list Parser =
    let build f s = f :: s
    build <!> parser <*> (many parser)


[<Fact>]
let ``applies a parser at least 1 time`` () =

    let manyWell = many1 (str "well!")

    test <@ run manyWell "well!well!well! the rest" =
            Success([ "well!"; "well!"; "well!" ], " the rest") @>

    test <@ run manyWell "the rest" =
            Failure("Expected well!") @>
```

It really seems that we have all the ingredients we need. Let's write the node parser down to code:

```
type Node =
    { tag: string
      content: string }

let alphaChars = [ 'a' .. 'z' ] @ [ 'A' .. 'Z' ]
let punctuationMarks = [' '; ';'; ','; '.']

let tagNameP = (many1 (anyOf alphaChars)) |>> (fun s -> String.Join("", s))

let openingTagP = tagNameP |> between (str "<")  (str ">")
let closingTagP = tagNameP |> between (str "</") (str ">")

let contentP = many (anyOf (alphaChars @ punctuationMarks)) |>> String.Concat

let buildNode openingTag content _closingTag =
    { tag = openingTag
      content = content }

let nodeP = buildNode <!> openingTagP <*> contentP <*> closingTagP

[<Fact>]
let ``parses an XML node`` () =
  let s = "<pun>Broken pencils are pointless</pun>the rest"

  let expected =
      { tag = "pun"
        content = "Broken pencils are pointless"}

  test <@ run nodeP s = Success (expected, "the rest") @>
```

Not too difficult, after all. What was the big deal?
The big deal is: this implementation is wrong. Did you spot the bug?

## semordnilap tags

If you did not, let me make it more apparent. Indulge me while I introduce a little silly change in the XML grammar, in line with the craziness of your beautiful programming language: let's ask the user to type the closing tag name backward, as a semordnilap. This will have the delightful effect of producing tag couples like `<stressed>...</desserts>`, `<repaid>...</diaper>`, `<evilStar>...</RatsLive>`. Amusing!

Now: parser combinators are composable, so simply improving the `closingTag` parser should allow the entire XML node parser to benefit from the change. After all, that's exactly their selling point, right? Reversing a string is dead easy:

```
let reverse (s: string) = new string(s.ToCharArray() |> Array.rev)
```

Therefore, creating a parser for closing tags should be a matter of lifting this `reverse` function to the parser world. Maybe we could try mapping `reverse`, with `<!>`:

```
let PemaNgat = reverse <!> tagNameP

let openingTagP = tagNameP |> between (str "<") (str ">")
let closingTagP = PemaNgat |> between (str "</") (str ">")
```

Does this work? I don't know, pal, how can I tell? Didn't we just forget to work with TDD? Where are tests? Let's put it right at once:

```
[<Theory>]
[<InlineData("foo")>]
[<InlineData("barBaz")>]
[<InlineData("evil")>]
[<InlineData("live")>]
let ``possible tag names`` (s: string) =
    test <@ run tagNameP s = Success(s, "")@>

[<Theory>]
[<InlineData("oof")>]
[<InlineData("zaBrab")>]
[<InlineData("live")>]
[<InlineData("evil")>]
let ``possible closing tag names`` (s: string) =
    test <@ run PemaNgat s = Success(reverse s, "")@>
```

```
// The same implementation as before
let nodeP = buildNode <!> openingTagP <*> contentP <*> closingTagP

[<Fact>]
let ``parses an XML tag node with semordnilap tags`` () =
  let s = "<hello>ciao ciao</olleh>"

  let expected =
      { tag = "hello"
        content = "ciao ciao"}

  test <@ run nodeP s = Success (expected, "") @>
```

Yes, it seems to work.
Did you notice that I included `evil` and `live` in both the tests for the opening tag and for the closing tag? And that in both cases the tests are green? Well, that's not surprising: `evil` is a legit *closing* tag name, because it's the reverse of `live`. And `live` too is a legit *closing* tag name, because it's the reverse of `evil`. Also, both are legit *opening* tag names. In short, both are valid for both cases. Indeed, the test for the closing tag requires that a string is the reverse of something. On second thought, it's a very loose constraint: any string is the reverse of, ehm, its reverse.

Does this mean that this test would pass no matter the string? Let's find it out with a random string:

```
[<Fact>]
let ``a random string can be both an opening and a closing tag name`` () =
    let random = Random()

    let randomString =
        [| for _ in 1 .. 10 -> alphaChars.[random.Next(alphaChars.Length)] |]
        |> String

    test <@ run PemaNgat randomString = Success(reverse randomString, "")@>
```

Wait a minute! Does it mean that our XML node parser would accept any closing tag, even if its name does not match with the opening tag? Let's see:

```
[<Fact>]
let ``accepts a wrong closing tag`` () =

  let s = "<hello>ciao ciao</picture>"

  let expected =
      { tag = "hello"
        content = "ciao ciao"}

  test <@ run nodeP s = Success (expected, "") @>```
```

Uh oh: geen test! Not a good news, indeed… (Note to self: next time, not only shall I write tests before the implementation, but I should also not forget the red phase of the red-green-refactor TDD mantra. Also, I should test both the happy and the failure case).

Doubt: is this a bug related to reversing the string, because of the semordnilap-based syntax? Let's try using unmatched tags with to the conventional tag name rule:

```
let tagNameP = many1 (anyOf ['a'..'Z'])
```

```
let openingTagP = tagNameP |> between (str "<") (str ">")
let closingTagP = tagNameP |> between (str "</") (str ">")

[<Fact>]
let ``XML node test`` () =
  let s = "<pun>Broken pencils are pointless</picture>"

  let expected =
      { tag = "pun"
        content = "Broken pencils are pointless"}

  test <@ run nodeP s = Success (expected, "") @>
```

Oh, no! It's still green! So, this bug is really inherent.

## Lack of context

If you think about it, in the definition of `openingTagP` and `closingTagP`:

```
let tagNameP = many1 (anyOf ['a'..'Z'])

let openingTagP = tagNameP |> between (str "<") (str ">")
let closingTagP = tagNameP |> between (str "</") (str ">")
```

there is no indication at all that the tag name of the closing tag must match the string parsed by the opening tag.
"How so?" I can hear you cry: "They are using the very same `tagNameP`! They must match the same tag name! It's literally written there!"
Not quite. `openingTagP` and `closingTagP` share the same tag name *parser*, not the same tag name *value*. Remember? A parser is a function eventually returning a parsed value. It's not that value. It's like a promise of a value. Run the very same `tagNameP` on 2 different, valid inputs and you will get 2 different parsed values.

Indeed: `tagNameP`, as it is defined, would succeed with *any* sequence of letters. `PemaNgat` would also succeed with *any* sequence of letters. Possibly, and most likely, with different and unrelated ones. The word "unrelated" is the key here: there is really no connection between the two parsers.

What we would rather do, instead, is to build `closingTagP` as the parser expecting *exactly* the *value* parsed by `openingTagP`. Something like:

```
let tagNameP = many1 (anyOf ['a'..'Z'])

let openingTagP = tagNameP |> between (str "<") (str ">")
let closingTag (openingTagName: string) =
    (str openingTagName) |> between (str "</") (str ">")
```

You see the tragedy? The value of `openingTagName` is not known until we physicall run the `openingTagP` parser. Until this page, we have encountered several parsers depending on other parsers. But this is in fact the first time we have a parser depending on *the result* of another parser. Watching this from another perspective: it's the first time that the elements of our grammar requires parsers having a notion of their surrounding context.

## Context-sensitiveness

Do you remember when I stated "We assume that a node is whatever is surrounded by an opening tag — such as `<joke>` — and its *corresponding* closing tag"? The aspect we just missed to take into account is related to that *corresponding* concept. It's only intuitive that this must have to do with some kind of relationship between the elements of a grammar and, consequently, some kind of *binding* between its parsers.

Indeed, grammars with elements depending on each other, like in the case of our matching opening and closing tags, are called Context-sensitive Grammars. A parser for this family of grammars requires a new tool that — it could be demonstrated — cannot be built as a composition of the applicative parsers we have distilled so far. We need a brand new mechanism.
This new tool is indeed pretty simple: we just need an operator similar to the Applicative Functor's <*>, only a bit smarter; a function able to pass the value successfully parsed by a parser to the next parser. So, something that could *bind* two parsers in a row. Not surprisingly, we will call this operator `bind` — or >>=, because we functional programmers can't get enough of symbols — and the resulting notion *monad*.

Implementing it will be super easy, just a matter of following the type signature, but the consequences will be revolutionary.

Curious? Grab a liquorice and jump to Chapter 14: we are going to write it.

Previous - Things You Are Not Sure About ~ Next - Mind the Context

## References

- Garden Path Sentence
- semordnilaP
- Context-sensitive Grammar

## Comments

GitHub Discussions

{% include fp-newsletter.html %} Boiling the problem down, we can say that, in Context-sensitive grammars, the key challenge is that the parsing of later elements depends on values obtained earlier. For example, suppose you have an `int Parser` to read a number. Then, based on that number, you dynamically create a new parser tailored to it, essentially a parser factory `int -> Foo Parser`, that takes the parsed integer and returns a parser for a structure depending on that integer.

If this sounds like a contrived example, here are other cases where this context-sensitiveness may result more intuitively justified:

- Making sure that parentheses are properly balanced in nested expressions such as `(f a, g (h (b, c)))`: each opening bracket must have a corresponding closing one. While you proceed parsing, somehow you have to carry along a counter.
- Checking that variables are declared before use.
- Indentation-based block structures in F#: the number of leading spaces of each line depends on the indentation level established earlier. Again, you have to carry along some context.
- Parsing command line options, where the presence of one option modifies the availability of subsequent options. For this case you might count on a parser factory that, depending on the previous option, generates one or another parser for the next option.
- Besides programming languages, as a linguistic example, checking that in subject+verb+object structures the agreement in gender and number is ensured. As a trivial example, you want that "I am" and "You are" succeed, and that "I are" and "You am" fail.

## It All Begins With A Signature

Keeping types generic, this means that first we find an `'a Parser`, then eventually an `'a -> 'b Parser` follows. Our goal is to write the parser combinator `bind` (or >>=) that, given those 2 elements, generates a `'b Parser`:

```
val bind : 'a Parser -> ('a -> 'b Parser) -> 'b Parser
```

We are also going to use our old acquaintance `pure'`, but we will adhere to the convention to call it `return'`.

```
let return' = pure'
```

First things first: we need a unit test. Let's continue working with the XML tag example:

```
let bind m f = __

let (>>=) = bind
let return' = pure'


type Node =
    { tag: string
      content: string }

let alphaChars = [ 'a' .. 'z' ] @ [ 'A' .. 'Z' ]
let punctuationMarks = [' '; ';'; ','; '.']

let tagNameP = many1 (anyOf alphaChars) |>> String.Concat

let openingTagP = tagNameP |> between (str "<")  (str ">")
let makeClosingTagP tagName = (str tagName) |> between (str "</") (str ">")

let contentP = many (anyOf (alphaChars @ punctuationMarks)) |>> String.Concat


let nodeP = __

[<Fact>]
let ``closingTag works in a context-sensitive grammar`` () =
  let s = "<pun>Broken pencils are pointless</pun>rest"

  let expected =
      { tag = "pun"
        content = "Broken pencils are pointless" }

  test <@ run nodeP s = Success (expected, "rest") @>

[<Fact>]
let ``not matching closing tags raise a failure`` () =
  let s = "<pun>Broken pencils are pointless</xml>rest"

  test <@ run nodeP s = Failure "Expected pun" @>
```

Even before implementing >>=, it is worth to analyze its use. The disrupting element is the closing tag parser, since it depends on the `tagName` value parsed by the previous parser. The combination of the 2 parsers is obtained by the application of >>=. Given its signature, you can use it like this:

```
let openThenCloseP =
    openingTagP >>= (fun tagName ->
            let closingTagP = makeClosingTagP tagName
            ...)
```

- First parse the opening tag (`openingTagP`).
- Then, pass forward the value it parses (`>>= (fun tagName -> ...`) as the argument to a continuation.
- The continuation can use that value to invoke `makeClosingTagP` to generate a tailored `closingTagP` parser
- …

We are not required to immediately use the `tagName` value: in fact, between the opening and the closing tags, we want to take the chance to parse the content. It's a matter of using a chain of `>>=` applications:

```
let nodeP =
    openingTagP >>= (fun tagName ->
        contentP >>= (fun content ->
            (makeClosingTagP tagName) >>= (fun _tagName ->
                return' { tag = tagName; content = content })))
```

If you squint your eyes you could read the funny `>>=` syntax as:

```
let openCloseP =
         openingTagP    >>=    (fun tagName -> ...)
// apply openingTagP   then    pass tagName to a lambda continuation
```

so you can read the whole sequence as:

- In order to parse an XML node
- first parse the opening tag (`openingTagP`).
- Then, pass forward the value it parses (`>>= (fun tagName -> ...`)
- to a continuation. This, in turn will parse the content (`contentP`)
- eventually passing forward the parsed value (`>>= (fun content ->   ...`)
- to the next part. This will use `tagName` to build the parser for the closing tag (`closingTagP tagName`)
- Finally, handing over (`>>= fun _tagName ->`) to the last part (not interested in the last parsed value)
- whose purpose is to just return an instance of the tag record (wrapped in a Parser, with `return'`).

If you find this code convoluted because of the value passing boiler plate, you are absolutely right: it sucks. Hang in there for a few more minutes: soon we will introduce a technique to dramatically streamline the code.

Fine. Let's finally implement this infamous `bind` combinator.

## Follow the type signature

```
// 'a Parser -> ('a -> 'b Parser) -> 'b Parser
let bind m f = ...
```

Going with the flow and following the type signature, we know we have to return a `'b Parser`:

```
let bind m f = Parser (fun s ->
    ...)
```

We have the input string `s` and `m`, the `'a Parser`. If we run this parser with the input string, we will get back a parsing result, possibly containing a parsed value `a: 'a`:

```
let bind m f = Parser (fun s ->
    let resultA = run m s
    ...)
```

We are not sure that the parsing succeeded. We'd better pattern match. Of course, in case of failure, we can let `binda just fail.

```
let bind m f = Parser (fun s ->
    let resultA = run m s
    match resultA with
    | Failure f -> Failure f
    | Success(a, rest) ->
        ...)
```

In case of success, we get the `'a` value and the unconsumed input: exactly what we needed to get the `'b Parser`:

```
let bind m f = Parser (fun s ->
    let resultA = run m s
    match resultA with
    | Failure f -> Failure f
    | Success(a, rest) ->
        let bParser = f a
        ...)
```

We are done! We got the `'b Parser` we wanted. We cannot just return it, because our code is surrounded by `Parser (fun s -> ...)` and we would end up with a parser inside a parser. Idea: we can `run` the `b Parser` with the `rest` input to get its parsed value:

```
let bind m f = Parser (fun s ->
    let resultA = run m s
    match resultA with
    | Failure f -> Failure f
    | Success(a, rest) ->
        let bParser = f a
        run bParser rest)
```

Test it. Green! You just made `Parser` a Monad.

### Is That All, Folks?

You might not be impressed by this result (surprisingly, `Parser` did not turn into a burrito). In fact, it's an explosive one. This little unsuspected `bind` function, together with `return'`, is so powerful that it could replace everything you did in the last 13 chapters. It's such a game changer that F# provides native support for its use, which will bring a dramatic shift to both the syntax and style of your code, for the better.

This has been a tough chapter and you deserve some rest. If you never enjoyed a Tamil Kootu, that's the perfect chance to give it a try. Chapter 15, here we come!

Previous - Things You Want To Remember ~ Next - One Combinator to Rule Them All

## Comments

GitHub Discussions

{% include fp-newsletter.html %} We will dedicate the next couple of chapters to giving ourselves a pat on the back to celebrate this little `bind` function as our most outstanding invention since chapter

1.
Here’s the plan:

- We will see how Monads entail and incorporate Functors and Applicative Functors, and how `bind` can be used to reimplement `map` and <*>. Basically, we will find out that `bind` was the missing Swiss Army Knife to implement everything and the kitchen sink.
- Then, we will see how F#‘s Computation Expressions can be used to simplify the use of `bind` and, consequently, the syntax of parsers. We will discover an interesting imperative style that, despite apparences, is perfectly pure functional.
- Finally, we will revisit the first goal using this new syntactic style. Hopefully, at last!, the whole Monadic Parser Combinators topic will make click.

## Universal Laws

There is something deeply rewarding in Functional Programming: every now and then, when you stumble upon some discovery, you find that it is so profoundly true and powerful that you can use it to express a wide range of seemingly unrelated ideas, often more concisely and elegantly than before.

Graham Hutton wrote a famous paper on one of those cases: A tutorial on the universality and expressiveness of fold. Indeed, `fold` (or `Aggregate` in LINQ lingo) is so powerful that if you stripped `Select`, `Where`, `Sum`, `First`, `Zip`, `CountBy` and other functions away from LINQ, only saving `Aggregate`, believe it or not, you could reimplement all of them entirely in terms of `Aggregate`. I cannot recommend to challenge you with this enlighting and fun exercise more.

`bind` plays in the same league. Indeed, I truly hope that, in the sections ahead, I will be able to show you what a tremendous power you uncovered with >>=.

## Look Ma, Functors!

Do you remember when in Chapter 7 we defined `map`?

```
let map (f: 'a -> 'b) (ap: 'a Parser) : 'b Parser =
    Parser (fun input ->
        let ar : 'a ParseResult = run ap input
        match ar with
        | Success (a, rest) -> Success (f a, rest)
        | Failure s -> Failure s )
```

Initially, we named it `<<|`, because we wanted to see it as the on-steroid sibling of the reverse-pipe operator `<|`. Then we started interpreting it as a way to map a function to the *content* of a parser, and we gave it the alias `<!>`. Finally, in Chapter 10 we discovered Applicative Functors, and we found out that `map` could be expressed in terms of <*> and `pure'`:

```
let map f a =
    pure' f <*> a
```

We can do the same with Monads. Challenge yourself before reading the solution: how to write `map` only using >>= and `return'`? It’s not an easy exercise but it is worth trying, and very rewarding.

Here is how I would do this. I would start by analyzing the signatures:

```
map : ('a -> 'b) -> 'a Parser -> 'b Parser
bind : 'a Parser -> ('a -> 'b Parser) -> 'b Parser
return : 'a -> 'a Parser
```

We want to build `map`, so we want to complete this implementation:

```
let map (f: 'a -> 'b) (aP: 'a Parser) =
    ...
```

Both `map` and `bind` return a `'b Parser`, so the only challenge is with the input parameters.
As input, we have `f` and `aP`. Can we just pass them as they are to `return'` and `bind`?
Well, `aP` has already the right type for `bind`, as it matches the 1st parameter. The second parameter, though, should be `'a -> 'b Parser`, while we have `'a -> 'b`. But we know that `return'` can help lifting a `'b` to `'b Parser`:

```
let map (f: 'a -> 'b) (aP: `a Parser) =
    let f' = fun a ->
        let b:  'b        = f a
        let bP: 'b Parser = return' b
        b
```

Good, that's it! We just have to invoke `bind` now:

```
let map (f: 'a -> 'b) (aP: `a Parser) =
    let f' = fun a ->
        let b = f a
        let bP = return' b
        b

    bind aP f'
```

We can make it way shorter inlining the variables:

```
let map (f: 'a -> 'b) (aP: `a Parser) =
    let f' = fun a -> return' f a

    bind aP f'
```

and then observing that:

```
    let f' = fun a -> return' f a
```

can be written in Point-Free style with the >> operator:

```
let map (f: 'a -> 'b) (aP: `a Parser) =
    let f' = f >> return'

    bind aP f'
```

It helps me to read >> as "*and then*", so that the expression:

```
f >> return'
```

reads as:

```
apply f, and then return'
```

which is exactly what the meaning of the original:

```
fun a -> return' f a
```

This gets us to:

```
let map (f: 'a -> 'b) (aP: `a Parser) =
    bind aP (f >> return')
```

or, using the infix alias >>=:

```
let map f aP =
    aP >>= (f >> return')
```

Wow! How concise! The compiler is happy with the signature and every, every single test is still green. This is heavy! It's actually quite something! I bet that the result appears cryptic and magic, at first. I swear that, after playing enough with FP, you will find it understandable. And I promise that, when we will finally introduce the *do notation* by the means of F# Computation Expressions, everything will get very intuitive.

## Look Ma, Applicative Functors Too!

Writing `map` in terms of >>= was cool. But we already wrote it in terms of <*>, so shall we be so impressed?

What if we killed <*>'s implementation and redefined it in terms >>= and `return'`? That would be similar to the case of `Aggregate` and LINQ: >>= would really be all we ever needed, the one-size-fits-all tool, the mythical silver-bullet operator.

In Chapter 10 we wrote:

```
let ap fP aP = Parser (fun input ->
    match run fP input with
    | Failure e ->  Failure e
    | Success (f, rf) ->
        match run aP rf with
        | Failure s -> Failure s
        | Success (a, ra) -> Success (f a, ra))

let (<*>) = ap
```

How can we write this in terms of >>=? OK, this is tought. I have no idea where to start from. Shall we try analyzing the signatures, like we did with `map`?

```
ap :  ('a -> 'b) Parser -> 'a Parser -> 'b Parser
bind : 'a Parser -> ('a -> 'b Parser) -> 'b Parser
return : 'a -> 'a Parser
```

Honestly, I don't see any easy combination. I can't help but feeling lost. It's just beyond what my brain can process. What can help my poor limited understanding is the following mental translation. Whenever I see the >>= operator in an expression like:

```
foo >>= (fun bar -> baz)
```

I interpret it like:

```
someParser >>= (fun theValueItParsed -> whatIWantToDoWithThatValue)
```

This matches 1:1 the signature:

```
bind : 'a Parser -> ('a -> 'b Parser) -> 'b Parser
```

The rule of thumb I keep in mind is:

- Whenever I find a Parser
- I can apply >>=.
- What follows is a function that simply receives the parsed value.
- So I can just operate on that value, ignoring that I am in the context of parsers.
- The only caveat I have to remember: at the end, I have to return a Parser, not a bare value.

Basically, I often use this metaphor: >>= is a lens that lets me look *inside* the parser box, so I can completely forget about parsers and deal directly with values:

```
parser >>= (fun parsedValue -> ...)
```

Fine. Going back to rewriting `ap`:

```
// ('a -> 'b) Parser -> 'a Parser -> 'b Parser
let ap (fP: ('a -> 'b) Parser) (aP: 'a Parser) =
    ...
```

`fP` is a function, and `aP` is the value to feed it with. Unfortunately, they are both inside a parser. No problem: we'll use the >>= lens to extract their values. We will have to apply >>= twice, one time to look inside `fP`, the other time for `aP`. Let's start with accessing `f` inside `fP`:

```
let ap (fP: ('a -> 'b) Parser) (aP: 'a Parser) =
    fP >>= (fun f ->
        ...)
```

Let's do the same with `aP`:

```
let ap fP (aP: 'a Parser) =
    fP >>= (fun f ->
        aP >>= (fun a ->
            ...))
```

Good. We have `f` and its argument `a`. That's easy! Applying `f` to `a` will get us back `b`:

```
let ap fP (aP: 'a Parser) =
    fP >>= (fun f ->
        aP >>= (fun a ->
            let b = f a
            ...))
```

Can we just return `b`? No, both the >>= signature and the signature of `aP` itself claim we should return a `'b Parser`, not a `'b`. Easy! `return'` to the resque:

```
let ap fP (aP: 'a Parser) =
    fP >>= (fun f ->
        aP >>= (fun a ->
            let b = f a
            return' b))
```

Done! Let's make it shorter, now, by inlining the temporary variable:

```
let ap fP (aP: 'a Parser) =
    fP >>= (fun f ->
        aP >>= (fun a ->
            return' f a))
```

and then, again applying >>:

```
let ap fP (aP: 'a Parser) =
    fP >>= (fun f ->
        aP >>= (f >> return'))
```

The compiler is happy, tests are green, so this expression must be correct.

## Not Really My Vibe

If you are one of those horrible developers who are proud when the code is super-concise, magic and almost impenetrable to your colleagues, you can stop here and praise yourself. In theory you could

even keep rewriting `many`, `many1`, `>>.`, `.>>`, `between`, `sepBy` and all the other parser combinators we have invented in the past chapters using >>= only. It is technically possible. Just know that every time you do that, a fairy loses its wings.
To me, in most of the cases, it makes little sense. I personally find this result too cryptic and not particularly expressive.

So, here's my alternative plan: I would rather get an energizing Tiramisù; then I will quickly proceed with Chapter 16, in which I intend to transform this horrible syntax into something more digestible for the rest of us. Then, I promise, there will be a very convincing reason to rewrite *some* of the past combinators with Monads. In the meanwhile, buon appetito.

Previous - Mind the Context ~ Next - A Programmable Semicolon

## References

- Graham Hutton - A tutorial on the universality and expressiveness of fold

## Comments

GitHub Discussions

{% include fp-newsletter.html %} Monads are beautiful, and so is F#. No wonders that the latter natively supports the former. There a little trick to extend the F# syntax to support monadic parser combinators. Indeed, since F# natively knows how to deal with Monads, via Computation Expressions, it's a matter telling it which Monad implementation to use.
The implementations of `bind` and `return'` will suffice:

```
type ParseBuilder() =
    member this.Bind(m, f) = m >>= f
    member this.Return(v) = return' v

let parse = ParseBuilder()
```

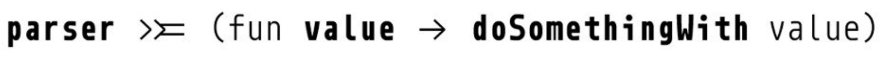
Here we go! From now on, whenever F# finds a piece of code inside a `parser { }` code block, it knows it can rely on the `Parser`'s `bind` and `return'` implementation. Even better, we won't even need to explicitly mention `bind` and `return'` in that block: F# will let us use some very sweet syntactic sugar. For whatever expression like:

```
parser >>= (fun value -> f value)
```

we can just write:

```
let! value = parser
f value
```

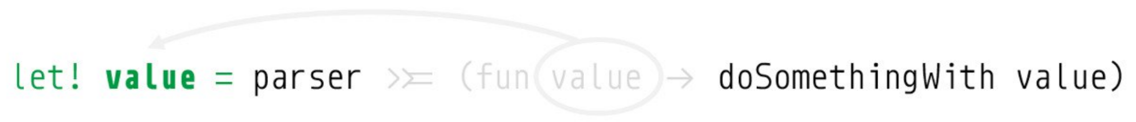
It helps me to imagine these movements. If you have an expression such as:



```
parser >>= (fun value -> doSomethingWith value)
```

{:width="75%"}

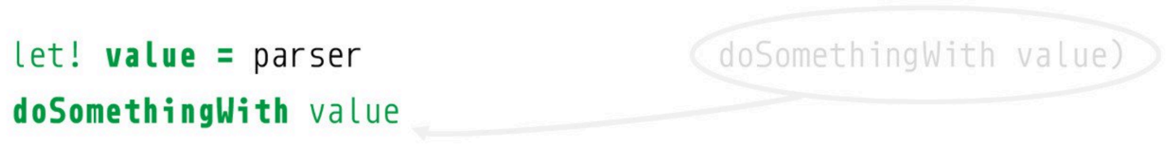
first, you move `value` to the left:



```
let! value = parser >>= (fun value -> doSomethingWith value)
```

{:width="75%"}

That is, instead of using `value` as a lambda parameter, you promote it to be a simple variable. Just use `let!` instead of `let`. Then, you move the body of the lambda `doSomethingWith value` to the next line:



```
let! value = parser                 doSomethingWith value)
doSomethingWith value
```

{:width="75%"}

And that's it. You can just remove all the boilerplate syntax elements, such as the >>= operator, `fun`, etc.

```
let! value = parser
doSomethingWith value
```

{:width="75%"}

So, keep in mind this transformation:

```
parser >>= (fun value -> doSomethingWith value)
```

```
let! value = parser
doSomethingWith value
```

Only, remember to wrap the latter in a `parse { }` block.
I find the first step amusing, because it reminds me that variables are syntactic sugar for lambda expressions.

This style was first introduced Haskell 1.3, under the name of *do notation* (see Changes from Haskell 1.2 to Haskell 1.3 - Monad Syntax). That's often what I call it too, although the correct name for F# is Computation Expression.

## `map` In Do Notation

Here's `map`, in its initil implementation, from Chapter 7:

```
let map (f: 'a -> 'b) (aP: 'a Parser) : 'b Parser =
    Parser (fun input ->
        let ar : 'a ParseResult = run aP input
        match ar with
        | Success (rest, a) -> Success (f a, rest)
        | Failure s -> Failure s )
```

Here is it how we defined it, in Chapter 10, in terms of <*> and `pure'`:

```
let map = (<<|)
let map (f: 'a -> 'b) (aP: 'a Parser) : 'b Parser =
    pure' f <*> aP
```

And, finally, the version of Chapter 15, implemented with `bind` and `return'`:

```
let map (f: 'a -> 'b) (aP: `a Parser) =
    bind aP (f >> return')
```

If we wanted to use the `parser` computation expression, we could mechanically apply the syntactic sugar movements listed before. Otherwise, we could always think to `let!` as a convenient and almost magic way to put our hands on the parsed value. Either way will lead us to:

```
let map f aP = parse {
    let! a = aP
    let b = f a
    return b
}
```

Read it as:

- mapping `f` over the parser `aP`
- generates another parser (`parse { ... }`)
- working like this:
- it takes the value `a` parsed by `aP` (`let! a = aP`)
- and it applies `f` to it (`let b = f a`)
- and this is the value the new parser would return (`return      b`). Notice that we are directly using the native `return`, not our custom `return'`.

Does it work? Let us update the original test we had for `map`:

```
[<Fact>]
let ``parser-powered function application`` () =
    let twice x = x * 2

    let p42: int Parser =
        Parser (fun _ -> Success(42, "rest"))

    let expr = parse {
        let! v = p42
        let twiceTheValue = twice v
        return twiceTheValue
    }

    test <@ run expr "some input" = Success(84, "rest") @>
```

Green.
It is worth to reflect on the difference between `let!` and `let` in:

- `let! a = aP`
- `let b = f a`

In `let! a = aP`, you basically run the parser `aP` and you get back the parsed value.
Instead, the `let` in `let b = f a` is the conventional `let` you have always used.

Of course, you can make the whole implemenetation a bit shorter:

```
let map f aP = parse {
    let! a = aP
    return' f a
}
```

As far as I know, when using this style there is no way to obtain a Point Free style.

## `ap` In Do Notation

Do you remember how we defined the Applicative Functor's `ap` in Chapter 10?

```
// ap : ('a -> 'b) Parser -> 'a Parser -> 'b Parser
let ap fP aP = Parser (fun input ->
    match run fP input with
    | Failure e ->  Failure e
    | Success (r, rf) ->
        match run aP rf with
        | Failure s -> Failure s
        | Success (a, ra) -> Success (f a, ra))
```

The 2 parameters of `ap` are both wrapped in a Parser. Indeed, the implementation revolves around running both parsers to get to the contained value, then applying the function to the value. We can really translate this literally, using the `parser` computation expression:

```
let ap fP aP =
    parse {
        let! f = fP
        let! a = aP

        return f a
    }
```

Isn't it sweet?

## Do Notation Everywhere

Computation Expressions are particularly effective at capturing the meaning of a parser and at making the intent clear. In Chapter 9 we defined `between` using `>>.` and `.>>`:

```
let between opening closing content =
    opening >>. content .>> closing
```

If you wanted to implement the same in do notation, you could ask yourself: "What is `between` supposed to do?"

- First, it should parse the opening tag, ignoring the result.
- Then, it should parse the content.
- Then, the closing tag, ignoring the result.
- And, finally, it should return the content.

Well, here is a literal translation:

```
let between openingP closingP contentP =
    parse {
        let! _ = openingP
        let! content = contentP
        let! _ = closingP

        return content
    }
```

Straighforward and readable, isn't it?

Here's a rewarding exercise to do: to go through all the parser combinators we have written until this point and to reimplement them in do notation style. Not only will you find this very easy — almost a matter of translating the requirements word by word — but likely you will find the resulting expression more eloquent and expressive.

Here are some examples.

## Old Wine In New Bottles

**`.>>.`**

Apply 2 parsers, returning both results in a tuple.
This is a literal translation:

```
let (.>>.) aP bP =
    parse {
        let! a = aP
        let! b = bP

        return (a, b)
    }
```

**`.>>`**

Apply 2 parsers, returning the result of the first one only.
Here the trick is to ignore the result of the second parser.

```
let (.>>) firstP secondP =
    parse {
        let! first = firstP
        let! _ = secondP

        return first
    }
```

**`>>.`**

Apply 2 parsers, returning the result of the second one only.
That's trivial! This time we just need to ignore the first result:

```
// 'a Parser -> 'b Parser -> 'b Parser

let (>>.) firstP secondP =
    parse {
        let! _ = firstP
        let! second = secondP

        return second
    }
```

**`many`**

Repeatedly apply a parser until it fails, returning a list of parsed values.
An idea could be to implement it as recursive function. We parse a first element, then we rely on recursion to parse the rest of the elements. Returning the result is a matter of building a list:

```
let rec many1 p =
    parse {
        let! x = p
        let! xs = many1 p

        return x :: xs
    }
```

Notice that this is the implementation of `many1`, requiring at least 1 element. `many` is easily obtained combining `many1` with the empty sequence case, by the use of `<|>`:

```
let rec many p =
    many1 p
    <|> (pure' [])
```

Compare this with what we obtained in Chapter 11:

```
let many<'a> (parser: 'a Parser): 'a list Parser = Parser (fun input ->
    let rec zeroOrMore input =
        match run parser input with
        | Failure _ -> (input, [])
        | Success (result, rest) ->
            match (zeroOrMore rest) with
            | [], rest -> (result :: [], rest)
            | others, rest -> (result :: others, rest)

    Success(zeroOrMore input))
```

and:

```
let rec many parser =
    (cons <!> parser <*> (many parser)) <|> (pure' [])
```

With Computation Expression we obtained an astoundingly easier formulation, don't you think?

**`skipMany`**

Parse zero or more occurrences of something, discarding the result. That's easy! We just need to parse many elements, only to ignore them:

```
let skipMany p =
    parse {
        let! _ = many p
        return ()
    }
```

**`sepBy`**

Parse a list of elements separated by a separator.
Finally, a more challenging one! Here's a possible implementation. A list of elements separated by a separator is an element followed by many groups "separator + element". We could capture the idea of "separator + element" with a parser on its own, to be used with `many`.

```
let rec sepBy separator parser =
    let sepThenP =
        parse {
            do! separator
            let! element = parser
            return element
        }

    parse {
        let! first = parser
        let! rest = many sepThenP
        return first :: rest
    }
```

**`lift3`**

Elevate a 3-parameter function into the Parsers world.
It's the combinator with this signature:

```
val lift3 : ('a -> 'b -> 'c -> 'd) -> 'a Parser -> 'b Parser -> 'c Parser -> 'd
Parser
```

This is easy to implement if you interpret the signature as:

Get a 3-parameter function and the 3 arguments, each wrapped in a parser. Parse each argumement then apply the function to the parsed values.

```
let lift3 f aP bP cP =
    parse {
        let! a = aP
        let! b = bP
        let! c = cP

        return f a b c
    }
```

### A Programmable Semicolon

`<|>`

## References

- Computation Expression
- Variables are syntactic sugar for lambda expressions
- Changes from Haskell 1.2 to Haskell 1.3 - Monad Syntax

## Comments

GitHub Discussions

{% include fp-newsletter.html %}