

# Monadic Parsing in F#

Frank Thomsen

frank@sector0.dk

## Abstract

This article is a short tutorial on how to write extensible recursive descent parsers with no mutable state using monads in F#. The parsers that are build using the techniques described here are able to accept ambiguous grammars, and have arbitrary lenght lookahead. These  $LL(*)$  (Parr 2007) parsers are not limited to any finite number of lookaheads. Although this potentially reduces performance in comparison to machine generated bottom-up parsers, the techniques described in this article will make simpler and more elegant parsers. Also the parsers eliminate the need for lexical analysis (tokenization) which is done on the fly.

## 1 Parser Type

We start by defining the signature of our parser functions. The type signature is:

```
type Parser<'r> = Parser of (char list -> ('r*char list) list)
```

That is, a parser is a function that takes a list of characters and produces a list of tuples. The tuple consists of the result and the - usually updated - character list that is the remainder of the input to be parsed. Putting these result tuples in a list enables us to accept ambiguous grammars.

A parser function also needs to be applied so we define a partial function for that:

```
let parse (Parser p) = p
```

## 2 Monadic Bind Combinators

The monadic bind combinator `>>=` will run a parser and apply those results (remember that a parser returns a list of results) to the next parser. The combinator takes a parser and a function that, given a result, returns a new parser. The result of applying the second parser to the list of results is a list of lists of results, and these lists will be concatenated before returning:

```
let (>>=) p f = Parser(fun cs ->
    List.concat [for (r,cs') in parse p cs -> parse (f r) cs'])
```

A parser is typically of the general format:

```
parser1 >>= fun r1 ->
parser2 >>= fun r2 ->
...
parsern >>= fun rn ->
mreturn (s r1 r2 ... rn)
```

where  $s$  is a *semantic* function that is applied to the results of the parsers.

We might not be interested in the result of a parser but only in the fact that it succeeded. So instead of writing

```
parser1 >>= fun _ ->
parser2 >>= fun _ ->
...
```

we simply want to discard the result right away and write:

```
parser1 >>
parser2 >>
...
```

The combinator `>>` can be expressed in terms of `>>=`:

```
let (>>) p q = p >>= fun _ -> q
```

### 3 Fundamental Parsers

In order to construct more complex parsers and combinators, both general and specific to a grammar, we define a few basic parsers. The first simply injects a value into a result without consuming any input characters:

```
let mreturn r = Parser(fun cs -> [(r,cs)])
```

The next is a parser that returns an empty result, often denoted  $\lambda$ ,  $\Lambda$  or  $\epsilon$ :

```
let lambda = Parser(fun _ -> [])
```

Next is the `item` parser. This parser consumes a single character unconditionally:

```
let item = Parser(fun cs ->
  match cs with [] -> [] | c::cs' -> [(c,cs')])
```

Using these fundamental parser we can define two other fundamental parser. The first is the conditional parser:

```
let sat cond =
  item >>= fun c -> if cond c then mreturn c else lambda
```

which takes the first character and applies a conditional function to it. If the condition evaluates to `true` the character is returned, otherwise an empty result is returned.

The `char` parser consumes a character if and only if the character matches:

```
let char c = sat ((=)c)
```

and the `digit` parser consumes a character if and only if the character is in the range `[0..9]`:

```
let digit = sat (fun c ->
  (List.tryFind ((=)c) ['0'..'9']).IsSome)
```

The `alpha` parser acts like `digit`:

```
let alpha = sat (fun c ->
  (List.tryFind ((=)c)(List.append
    ['a'..'z'] ['A'..'Z'])).IsSome)
```

## 4 Choice Combinators

Often we need to be able to make a choice between two or more different parsers. The choice combinator does just that:

```
let (<|>) p q = Parser(fun cs ->
  match parse p cs with
  | [] -> parse q cs
  | rs -> rs)
```

The choice combinator takes two arguments `p` and `q`, both of which are parsers. It first applies the first parser. If it succeeds then that result is returned- If the result is an empty list it means it failed, and `q` is parsed. Note that `q` might also fail.

There is another and equally important choice combinator: the ambiguous choice combinator. It appends the result of each parser to each other and returns that as a result. If one or the other parser fails with `[]` the result is that from the other parser.

```
let (++) p q = Parser(fun cs ->
  List.append (parse p cs) (parse q cs))
```

## 5 Recursive Combinators

The Kleene\* and Kleene+ (0-or-many and 1-or-many repetitions of a parser, respectively) combinators are expressed in terms of each other:

```

let rec many0 p = many1 p <|> mreturn []
and many1 p = p >>= fun r -> many0 p >>= fun rs -> mreturn (r::rs)

```

Sometimes it is necessary to check that not only a single character but an entire string can be parsed, for example when parsing keywords:

```

let rec symbol cs =
  match cs with
  | [] -> mreturn []
  | c::cs' -> char c >> symbol cs' >> mreturn cs

```

## 6 Ambiguous Choice and LL(\*)

The ++ combinator is important since it lets us parse ambiguous grammars. Suppose we have the simple grammar:

$$\begin{aligned}
 \textit{expr} &\rightarrow \textit{number};' \\
 \textit{number} &\rightarrow \textit{integer} \mid \textit{decimal} \\
 \textit{integer} &\rightarrow \textit{digit}^+ \\
 \textit{decimal} &\rightarrow \textit{digit}^+ '.' \textit{digit}^*
 \end{aligned}$$

Our first attempt at writing a parser looks like this:

```

let rec expr =
  number >>= fun n ->
  char ';' >>
  mreturn n
and number = integer <|> dec
and integer =
  many1 digit >>= fun digits ->
  mreturn <apply semantic function on digits>
and dec =
  many1 digit >>= fun digits ->
  char '.' >>
  many0 digit >>= fun decimals ->
  mreturn <apply semantic function on nums and decimals>

```

At first glance it looks fine. We expect to parse a number followed by a semicolon and nothing more in this example. But what happens if we try to parse the string “123.4;”? The `number` parser returns an integer. The rest of `expr` fails since it expects a semicolon but finds a period.

If we substitute the use of <|> with ++ the result is different:

```

...
and number = integer ++ dec
...

```

The `++` combinator applies both parsers and return `[(123,['.'],'4';';'); (123.4,[';'])]` as a result. The `>>=` combinator takes this list of results and applies the next parser (here `char '.'`...) on both. Since the next character after 123 is not `.` this is discarded and only 123.4 is usable.

This simple example demonstrates the ability to write parsers with arbitrary length lookahead without having to explicitly implement a lookahead parser.

## 7 Other Useful Parsers and Functions

A few string helper functions might come in handy since a string and a character list are not the same thing in F#. We therefore use the unary functions

```
// convert a string to a list of characters
let (~&) (str:string) = str.ToCharArray() |> List.ofArray
// convert a list of characters to a string
let (~%) (chars:char list) = new String(Array.ofList chars)
```

There are other combinators that are quite useful:

- Parse a series of parser *p* separated by a parser *sep* and return the list of results from parsing *p*. For example, parsing the string *aba* or *abababa*:

```
let sepBy p sep =
  p >>= fun r ->
    many0 (sep >> p) >>= fun rs ->
      mreturn (r::rs)
```

- Parse *p* followed by another parser *e*:

```
let endBy p e =
  p >>= fun r ->
    e >>
      mreturn r
```

- Parse a new line and an end-of-line. Note that “`\r\n`” is a newline on win32 systems.

```
let newline = symbol &"\r\n"
let endofline = many0 (char ' ') >> newline >> many0 (char ' ')
```

- Parse any number ( $\geq 1$ ) of spaces, here defined as either `' '` or `'\t'`:

```
let space = many1 (char ' ' <|> char '\t')
```

- Either parse the parser *p* or nothing. In grammar terms it is expressed as *p* /  $\Lambda$  and is the equivalent of 0 or 1 successful applications of *p*.

```
let orLambda p = (p >>= fun v -> mreturn [v]) <|> mreturn []
```

Of course, it is possible to construct any number of parsers and combinators given the ones already mentioned here. It all depends on the particular needs when implementing a parser for any given grammar, and this tutorial is by no means exhaustive.

## 8 Error Messages

Parsers that either return a successful result or nothing at all are not particularly useful; we need to be able to combine error messages and send them back up the chain so they can be displayed to the programmer.

We would like to be able to write an error message like “*Error at line x, column y. Unexpected end-of-file. Expected '.', ';' or '\n'.*”. To do this we extend our parser definition thus:

```
type Parser<'r> = Parser of (char list ->
  (('r*char list) list*string list*string*int))
```

That is, a parser is a function that takes a list of characters and returns a tuple with 4 values. Their meanings are:

- (**'r\*char list**) **list** : the result of the parsing. An empty list if the parser fails.
- **string list** : a list of strings of what the parser *expected* when it failed.
- **string** : a description of the *unexpected*, for example “*end-of-file*” or “*;*”.
- **int** : a pointer to where the error is. Here we use the number of characters *not* parsed yet. This number can easily be converted to line- and column number in the original input.

The monadic bind combinator must still run a parser and apply the results to the next parser. Results will still be concatenated. Each result list is followed by a list of strings describing what the parser expected, and these must likewise be concatenated. The bind combinator looks like this:

```
let (>>=) p f = Parser(fun cs ->
  match parse p cs with
  | ([],exs,unex,pos) -> ([],exs,unex,pos)
  | (rs,_,_,_) ->
    List.map (fun (r,cs') -> parse (f r) cs') rs
    |> List.fold (fun (rs,exs,_,_) (rs',exs',unex,pos) ->
      (List.append rs' rs,List.append exs' exs,unex,pos))
      ([],[],"",len cs))
```

The >> combinator is the same as before.

The choice operators also needs to handle the new parser type:

```
let (++) p q = Parser(fun cs ->
  let (prs,pexs,punex,ppos) = parse p cs
  let (qrs,qexs,qunex,qpos) = parse q cs
  ( List.append prs qrs,
    List.append pexs qexs,
    (if ppos<qpos then punex else qunex),
    min ppos qpos))

let (<|>) p q = Parser(fun cs ->
  match parse p cs with
  | ([],exs,_,_) ->
    let (rs,exs',unex,pos) = parse q cs
    (rs,List.append exs exs',unex,pos)
  | other -> other)
```

And the fundamental parsers mreturn and lambda likewise:

```
let mreturn r = Parser(fun cs -> ([r,cs]),[],"",len cs))

let lambda = Parser(fun cs -> ([],[],"",len cs))
```

where len is a partial function that calls List.length. In addition we need an err parser that takes a string that describes something unexpected:

```
let err unex = Parser(fun cs -> ([],[],unex,len cs))
```

It is not always desirable to return a long list of what was expected (think: the entire alphabet when a digit was expected) so a *replace-expected-errors* operator >>@ is needed:

```
let (>>@) p exp = Parser(fun cs ->
  match parse p cs with
  | ([],_,unex,pos) -> ([],[exp],unex,pos)
  | other -> other)
```

The item parser is changed so it returns an error message if it is applied to an empty input:

```
let item = Parser(fun cs ->
  match cs with
  | [] -> ([],[],"end-of-file",0)
  | c::cs' -> ([c,cs'],[],"",0))
```

The sat parser is enhanced to take a string. This string will describe what was expected if the condition is not met and is injected into the result using the >>@ operator:

```

let sat cond exp =
  (item >>= fun c -> if cond c then mreturn c else err %[c])
  >>@ exp

```

The `char` parser uses `sat` and now looks like this:

```

let char c = sat ((=)c) %[c]

```

This way the `char` parser will return the character in the list of expected's if the input does not satisfy the condition.

The parsers `digit` and `alpha` uses `sat` so they too must supply an error message in case of failure:

```

let digit = sat (fun c ->
  (List.tryFind ((=)c) ['0'..'9']).IsSome) "a digit"

let alpha = sat (fun c ->
  (List.tryFind ((=)c) (List.append
    ['a'..'z'] ['A'..'Z'])).IsSome) "a letter"

```

The `symbol` parser which takes a list of characters and recursively applies the `char` parser to the input is enhanced using the `>>@` parser in case of failure:

```

let rec symbol cs =
  (match cs with
  | [] -> mreturn []
  | c::cs' -> char c >> symbol cs' >> mreturn cs)
  >>@ %cs

```

The `endBy` parser must take the last error message from `p` and concatenate it with the error message from `e` using the `++` combinator:

```

let endBy p e =
  p >>= fun r ->
    (p ++ e) >>
    mreturn r

```

To test that we have reached the end of the input we have the parser `endoffile`:

```

let endoffile = Parser(fun cs ->
  match cs with
  | [] -> ([([],[])], [], "", 0)
  | _ -> ([], ["end-of-file"], %[List.head cs], len cs))

```

## 9 An example

Let us use what we have written to implement a parser that accepts languages of the following simple grammar (Scott 2006):

```
program    → stmt_list
stmt_list  → stmt stmt_list |  $\Lambda$ 
stmt       → identifier " := " expr | "read" identifier | "write" expr
expr       → term term_tail
term_tail  → add_op term term_tail |  $\Lambda$ 
term       → factor factor_tail
factor_tail → mul_op factor factor_tail |  $\Lambda$ 
factor     → '(' expr ')' | identifier | number
add_op     → '+' | '-'
mul_op     → '*' | '/'
```

First order of business is to specify what the parser will be returning. We want the parser to return a list of some type of expression which we can later evaluate. Evaluation can then either be executing the program in an interpreter, or some sort of compilation with generation of machine code, byte code, IL-code or something else. In this example we evaluate by interpreting, and the parser will return a list of expressions which is specified in the following discriminated union:

```
type Expr =
  | Number of decimal
  | Identifier of string
  | Assignment of string*Expr
  | Read of string
  | Write of Expr
  | AddOp of Expr*Expr
  | SubOp of Expr*Expr
  | MulOp of Expr*Expr
  | DivOp of Expr*Expr
```

The values of this discriminated union becomes the semantic functions in our production rules. The production rules can be implemented using the combinators like this:

```
let rec program = endBy (sepBy stmt endofline) endoffile
and stmt = read <|> write <|> assignment
and read =
  ( symbol &"read" >>
    many1 (char ' ') >>
    many1 alpha >>= fun identifier ->
      Read (%identifier) |> mreturn) >>@ "Read <var-name>"
and write =
```

```

( symbol &"write" >>
  many1 (char ' ') >>
  expr >>= fun e ->
    Write e |> mreturn) >>@ "Write <expression>"
and assignment =
( many1 alpha >>= fun identifier ->
  symbol &":=" >>
  expr >>= fun e ->
    Assignment(%identifier,e) |> mreturn)
  >>@ "<var-name>:=<expression>"
and expr =
  term >>= fun t ->
    (term_tail >>= fun (op,b) -> mreturn (op t b)) <|> mreturn t
and term_tail =
  addop >>= fun op ->
  expr >>= fun b ->
  mreturn (op,b)
and term =
  factor >>= fun f ->
  (factor_tail >>= fun (op,b) -> mreturn (op f b)) <|> mreturn f
and factor_tail =
  mulop >>= fun op ->
  term >>= fun b ->
  mreturn (op,b)
and factor =
( char '(' >>= fun _ ->
  expr >>= fun e ->
  char ')' >>
  mreturn e)
<|> number
<|> (many1 alpha >>= fun identifier ->
  Identifier(%identifier) |> mreturn)
and addop =
( char '+' >> mreturn (fun a b -> AddOp(a,b))) <|>
( char '-' >> mreturn (fun a b -> SubOp(a,b)))
and mulop =
( char '*' >> mreturn (fun a b -> MulOp(a,b))) <|>
( char '/' >> mreturn (fun a b -> DivOp(a,b)))
and number = integer ++ decimal
and integer =
  many1 digit >>= fun digits ->
  Decimal.Parse(%digits+",0") |> Number |> mreturn
and decimal =
  many1 digit >>= fun nums ->
  char '.' >>
  many1 digit >>= fun decimals ->

```

```

    Decimal.Parse(%nums + "," + %decimals)
    |> Number
    |> mreturn

```

This parser is capable of parsing programs like this one:

```

let prog =
  "read a
  read b
  sum:=a+b
  prod:=a*b
  c:=b
  write sum
  write prod
  write c
  write ((43.2*a)+(2*b))/32.45"

```

Finally, evaluating the program can be done by calling `parse program &prog` and passing the result to the `evaluate` function in the very simple evaluation sample code below:

```

let vars = new Dictionary<string,decimal>()

let rec eval stmt =
  match stmt with
  | Number d -> d
  | Identifier id ->
    if vars.ContainsKey(id) then vars.[id]
    else new Exception(sprintf "Unknown variable '%s'" id) |> raise
  | Assignment (id,exp) ->
    let v = eval exp
    vars.Add(id,v)
    v
  | Read id ->
    Console.Write (sprintf "%s > " id)
    let v = Console.ReadLine() |> Decimal.Parse
    vars.Add(id,v)
    v
  | Write exp ->
    let v = eval exp
    Console.WriteLine v
    v
  | AddOp (e1,e2) -> (eval e1) + (eval e2)
  | SubOp (e1,e2) -> (eval e1) - (eval e2)
  | MulOp (e1,e2) -> (eval e1) * (eval e2)
  | DivOp (e1,e2) -> (eval e1) / (eval e2)

```

```
let rec evaluate stmts =  
  match stmts with  
  | [] -> ()  
  | stmt::stmts' ->  
    eval stmt |> ignore  
    evaluate stmts'
```

As I have shown it is quite easy to define readable, elegant parsers using monads. Furthermore, it is easy to use the existing combinators and parsers to write new ones.

## References

- Parr, T. (2007) *The Definite ANTLR Reference - Building Domain-Specific Languages*.  
Scott, M.L. (2006) *Programming Language Pragmatics*, 2<sup>nd</sup> edition.