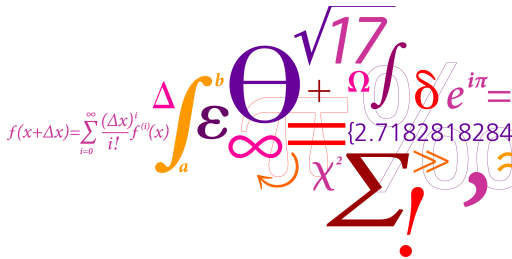


02257 Applied Functional Programming

Parsing using combinators

Michael R. Hansen



DTU Compute

Department of Applied Mathematics and Computer Science

Parsing: translation from strings to structured values

Wide-spectrum applications

- programming languages implementation
 - domain specific languages
 - processing of various kinds of data formats
 -
-
- Theory and tools are well-developed in the context of PL
 - Parser combinators provide a light-weight alternative for simpler settings

A simple example

Consider the type:

```
type E = V of string | C of int | Sub of E*E
```

Example of concrete syntax:

```
x2 - (2 -x1)
```

should be parsed as: `Sub (V "x2", Sub (C 2, V "x1"))`

A grammar for expressions

```
E -> V | C | ( E ) | E - E
```

where where terminal symbols are

- *V* - identifier
- *C* - integer
- the symbols '(', ')', ' - '

Construction of a recursive descent parser

How?

Some Background

Some early work

- R. Frost and J. Launchbury, Constructing natural language interpreters in a lazy functional language. The computer journal 32(2): 108-121, 1989
- G. Hutton, Higher-order functions for parsing. Journal of functional programming 2(3): 323-343, 1992

Parsec (and FParsec) supports parsing with unbounded look-ahead; but is designed so that it works best for LL(1) grammars:

- First 'L': Read input from left to right
- Second 'L': Make leftmost derivation
- Lookahead '1': Make parsing choice on the basis of just the next input symbol

An LL(1) grammar is

- not ambiguous
- not left recursive

Many libraries exist

We use FParsec

- The Parsec library: <https://wiki.haskell.org/Parsec> is ported to many languages
- <https://www.quanttec.com/fparsec/>
- <https://fsharpforfunandprofit.com/posts/understanding-parser-combinators/>
- Chapter 12: Computation expressions from Functional programming using F#, Hansen and Rischel. DTU Library
- Grammars and parsing with Haskell Using Parser Combinators, P. Sestoft and K. Friis Larsen, Notes 2013. Uploaded to DTU Learn.

On a Parser type

A parser $p : \text{Parser} \langle 'a \rangle$ denotes

a computation of a value of type $'a$ from a string.

Typical ingredients of a type $\text{Parser} \langle 'a \rangle$

```
string -> ('a * string) list
```

Further parts of a parser state may include

- error handling
- positions in the string
-

The type in FParsec has the form $\text{Parser} \langle 'a, 'u \rangle$, where $'u$ is a user state. Can be `unit` in the project.

Explicit type annotations like $\text{Parser} \langle E, \text{unit} \rangle$ are occasionally needed to avoid **Value Restriction** errors.

A simple example II: An LL(1) grammar

Our example grammar:

$$\begin{aligned} E \rightarrow & V \mid C \mid (E) \\ & \mid E - E \end{aligned}$$

can be systematically rewritten into an LL(1) grammar

$$\begin{aligned} T &\rightarrow V \mid C \mid (E) \\ E &\rightarrow T \text{ Eopt} \\ \text{Eopt} &\rightarrow - T \text{ Eopt} \mid \text{epsilon} \end{aligned}$$

using the techniques presented in notes by Sestoft and Friis Larsen

- We can now construct a recursive descent parser

```
spaces: Parser<unit,'u>
```

```
pstring: string -> Parser<string,'u>
```

```
pint32: Parser<int32,'u>
```

```
many1Satisfy2L: (char -> bool) -> (char -> bool)  
                -> string -> Parser<string,'a>
```

```
(.>>): Parser<'a,'b> -> Parser<'c,'b> -> Parser<'a,'b>
```

```
preturn: 'a -> Parser<'a,'b>
```

```
run: Parser<'a,unit> -> string -> ParserResult<'a,unit>
```

where `ParserResult` is a disjoint union with

- a success part and
- a failure part

A simple lexical analysis

There is a partitioning of scanning and parsing when using scanner and parser generators.

It is meaningful to maintain this distinction when using combinators

- to handle white space consistently and systematically, for example

Blanks after a token are skipped:

```
// token: Parser<'a','b> -> Parser<'a','b>  
let token p = p .>> spaces;;
```

```
//symbol: string -> Parser<string,'a>  
let symbol s = token (pstring s);;
```

```
let pinteger:Parser<int,unit> = token pint32;;
```

```
let ident:Parser<string,unit> =  
  let charOrDigit c = isLetter c || isDigit c  
  token(many1Satisfy2L isLetter charOrDigit "identifier");;
```

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

```
(>>=) : Parser<'a,'b> ->  
      ('a -> Parser<'c,'b>) -> Parser<'c,'b>
```

```
between: Parser<'a,'b> -> Parser<'c,'b> ->  
        Parser<'d,'b> -> Parser<'d,'b>
```

where

- choice $p_1 <|> p_2$ first tries p_1 . If this fails without changing the parser state, then p_2 is tried.

- $>>=$ is also known as bind.

monads

Bind is used to explain computation expressions

```
let pV = parse { let! x = ident  
                return V x };;
```

```
let pV = ident >>= fun x -> preturn (V x);;
```

A forward-reference technique is used to handle recursion in object declarations

```
let (pE, pERef) = createParserForwardedToRef<E, unit>()
```

```
let pT = pV <|> pC
      <|> parse { let! _ = symbol "("
                  let! e = pE
                  let! _ = symbol ")"
                  return e }
```

```
let rec pEopt e = parse { let! _ = symbol "-"
                          let! e' = pT
                          return! pEopt (Sub(e, e')) }
      <|> preturn e;;
```

```
pERef.Value <- parse { let! e = pT
                      return! pEopt e};;
```

- The construction follows the grammar closely
- See uploaded notes by Sestoft and Friis Larsen

Monadic parser combinators:

- powerful light-weight setting for writing recursive decent parsers
- a genuine functional approach based on combinators