Applicative and Monadic Parsing Combinators
A practical overview

Aurick Qiao

Department of Computer Science
University of Waterloo
a2qiao@uwaterloo.ca

October 23, 2013
Introduction

Idea: Manipulate parsers as first-class citizens

- Define functions that operate on parsers and produce parsers
- Allows production of new parsers from existing ones
Introduction

Advantages:
- Easy to build complex parsers
- Parser in code resembles the grammar

Disadvantages:
- Usually less efficient than other methods
- Issues in dealing with left-recursion
In general, parser combinator libraries provide:

- A fixed set of simple parsers
- Functions that combine parsers to form more complex parsers

A parser for a more complex language is created by combining the provided simple parsers using the provided functions.
Introduction

Two ways of implementing parser combinators:

- **Applicative**
  - Simpler to use and read
  - Implementation by S. Doaitse Swierstra

- **Monadic**
  - More powerful (parses context-sensitive grammars)
  - Implementation of Parsec

Main difference is in the way concatenation is handled (more later)
Applicative Parser Combinators

Reference:
"Combinator Parsing: A Short Tutorial"
Applicative Parser Combinators

Type of a parser:

type Parser s t = [s] -> [(t,[s])]  

Partially parses a list of tokens of type s and produces a list of successful parse results. A partial result is a tuple of a ”witness” and a list of unprocessed tokens. A witness is the final value computed by the parser.
pSym, pReturn, and pFail

pSym takes a symbol as argument and produces a parser that successfully parses only that symbol, returning the symbol as a witness if successful.

\[
p\text{Sym} :: \ Eq\ s \Rightarrow s \to \text{Parser } s\ s
\]
\[
p\text{Sym}\ a\ [] = []
\]
\[
p\text{Sym}\ a\ (s:ss) = \begin{array}{l}
if\ s == a\ then\ [(s,ss)] \\
else\ []
\end{array}
\]
pSym, pReturn, and pFail

pReturn always succeeds without consuming any tokens and produces its argument as the witness.

\[
p\text{Return} :: t \rightarrow \text{Parser } s \ t
\]
\[
p\text{Return } a \ ss = [(a, ss)]
\]
pSym, pReturn, and pFail

pFail always fails.

\[
pFail :: \text{Parser } s \ t
\]
\[
pFail ss = []
\]
represents the concatenation parser. It uses its two arguments one after the other to parse the list of tokens.

\[(<<\star\rangle) :: \text{Parser} \ s \ (b \to a) \to \text{Parser} \ s \ b \to \text{Parser} \ s \ a\]

\[(p1 \star p2) \ ss = [(v1 \ v2, \ ss2) | (v1, \ ss1) \leftarrow p1 \ ss, (v2, \ ss2) \leftarrow p2 \ ss1]\]

To combine the two witnesses, the first witness is applied to the second. This seems arbitrary, but works very elegantly in conjunction with the <$>$ operator.
\(<*\rangle\) and \(<|>\)

\(<|>\) represents the alternation parser. It parses the list of tokens using both of its arguments and returns all parse results of both.

\((<|>)\) :: Parser s a -> Parser s a -> Parser s a

\((p1 <|> p2) ss = p1 ss ++ p2 ss\)
Precedence Levels

The following precedence levels for <*, <$>, and <|> ensure the least number of parentheses:

    infixl 5 <*
    infixr 3 <|>
The definitions of <|> and <*> strongly suggest using them in the following pattern:

\[
\begin{align*}
  f1 &:: a \to b \to c \to d \\
  p1 &:: \text{Parser } s \ a \\
  p2 &:: \text{Parser } s \ b \\
  p3 &:: \text{Parser } s \ c \\
  f2 &:: e \to f \to d \\
  p4 &:: \text{Parser } s \ e \\
  p5 &:: \text{Parser } s \ f \\
  p\text{Return } f1 &\bowtie p1 \bowtie p2 \bowtie p3 \\
  <|> &p\text{Return } f2 \bowtie p4 \bowtie p5
\end{align*}
\]
Canonical Structure

\[ \text{pReturn } f_1 \leftrightarrow p_1 \leftrightarrow p_2 \leftrightarrow p_3 \]
\[ \text{<|> pReturn } f_2 \leftrightarrow p_4 \leftrightarrow p_5 \]

This will apply \( f_1 \) to the witnesses of \( p_1 \), \( p_2 \), and \( p_3 \) and produce a Parser \( s \) \( d \) that is just a \( \text{pReturn} \) that produces the result of \( f_1 \) as the witness. Then, it will do the same thing with \( f_2 \), \( p_4 \), and \( p_5 \).
The canonical structure resembles the structure of function application. `<$>` and `'opt'` are essentially syntactic sugar to make them look more similar.

\[
\text{infix 7 `<$>`}
\]
\[
(<$>) :: (b \rightarrow a) \rightarrow (\text{Parser } s \text{ b}) \rightarrow \text{Parser } s \text{ a}
\]
\[
f <$> p = \text{pReturn } f <\ast> p
\]

\[
\text{infix 3 `'opt'`}
\]
\[
\text{opt} :: \text{Parser } s \text{ a} \rightarrow a \rightarrow \text{Parser } s \text{ a}
\]
\[
p 'opt' v = p <\mid> \text{pReturn } v
\]
Now it is possible to write parsers of the form

\[
p = f \ <$> \ p \ <*> p \ <*> \ldots \ <*> p \quad \text{‘opt’} \ f \ <$> \ p \ <*> p \ <*> \ldots \ <*> p \\
\quad \ldots \\
\quad \text{‘opt’} \ f \ <$> \ p \ <*> p \ <*> \ldots \ <*> p 
\]

This closely resembles the form of function applications, hence ”applicative” parser combinators.
Example

Consider the grammar $S \rightarrow '( ' S ' ) ' S | \epsilon$. Then the function `parens` defined below will parse a string in the language and produce the maximum nesting depth.

```haskell
parens :: Parser Char Int
parens = (\_ b _ d \rightarrow max (1 + b) d <$>
    pSym '(' <*> parens <*> pSym ')' <*> parens
  'opt' 0
```

The structure of the parser is very similar to the structure of the grammar it parses.
Problems - Left Recursion

Since combinators recurse on non-terminals of the grammar, they cannot handle left-recursion.

\[ S \rightarrow S \ '+' \ integer \ | \ integer \]

The direct translation of this grammar will recurse indefinitely until a stack overflow.
A potential solution is to re-write the grammar so that it is right-recursive:

$$S \rightarrow integer + S | integer$$

Works well for addition, but this grammar is right-associative.

$$S \rightarrow integer - S | integer$$

Translating this grammar directly will produce the wrong result.
Problems - Left Recursion

One way to solve this is to "flatten" the grammar:

\[ S \rightarrow \text{integer} - \text{integer} - \ldots - \text{integer} \]

Define a new function:

\[
\text{pMany} :: \text{Parser s a} \rightarrow \text{Parser s [a]}
\]

\[
\text{pMany } p = (:) \langle\$\rangle p \langle*\rangle \text{pMany } p \text{ ‘opt‘ [}]
\]

Use pMany to get a list of operations and operands, then use \text{foldl} to evaluate the expression. This computation is defined as the parser pChainL.
Flattening the grammar works, but is specific to each grammar. Each grammar needs to manually be flattened before a parser can be written for it.
Problems - Error Reporting

When a list of symbols fails to parse, only [] is returned, this is unacceptable. Ideally, it should return information about:

- Position of the failure
- Symbols involved in the failure
- Parse result of the preceding couple of symbols
Problems - Space

There is a significant space issue arising from the $<$|$>$ operator. $p_1$ $<$|$>$ $p_2$ must keep its entire input in memory until it decides for sure which choice to make. Only after can it discard the unneeded input.
Reference:
"Parsec: Direct Style Monadic Parser Combinators For The Real World"
Monadic Parser Combinators

Monadic parser combinators are the same as applicative parser combinators, with the exception that `(<*>)` is replaced with `>>=`

\[
\begin{align*}
(<*>) & : : \text{Parser} \ (a \rightarrow b) \rightarrow \text{Parser} \ a \rightarrow \text{Parser} \ b \\
(>>=) & : : \text{Parser} \ a \rightarrow (a \rightarrow \text{Parser} \ b) \rightarrow \text{Parser} \ b
\end{align*}
\]

The result of the first parser is used to construct a new parser at runtime. This means it is possible to parse context-sensitive grammars using monadic parser combinators.
The Parser Monad

Wrap the Parser type as a monad.

instance Monad (Parser s) where
    return = pReturn
    (p >>= f) ss = [(v2, ss2) |
                    (v1, ss1) <- p ss,
                    (v2, ss2) <- (f v1) ss1]
Example

Consider the grammar $S \rightarrow '(' S ')' S | \epsilon$. This is the same parens function defined using $\gg\gg$ instead of $\ll\ll$.

```
parens = ( pSym '(' >>= \_ ->
    parens2 >>= \b ->
    pSym ')' >>= \_ ->
    parens2 >>= \d ->
    pReturn (max (1 + b) d) )
<|>
    pReturn 0
```

Parentheses are needed since each anonymous function will include `everything` after it.
Example - Do Notation

The same code using Haskell’s do notation.

parens = do pSym '('
    b <- parens
    pSym ')
    d <- parens
    return (max (1 + b) d)
<|>
do return 0
Context Sensitive Grammars

Since monadic combinators generate parsers during run-time, it can parse context-sensitive grammars. For example, the language of all square words.

\[
\text{pAny :: Parser } s \ s \\
p\text{Any } [] = [] \\
p\text{Any } (s:ss) = [(s, ss)]
\]

\[
\text{pMany :: Parser } s \ [s] \\
p\text{Many} = \text{do return } [] \\
\ |
\text{do } c \leftarrow \text{pAny} \\
\quad s \leftarrow \text{pMany} \\
\quad \text{return } (c : s)
\]
Context Sensitive Grammars

\[
p\text{Same} :: \text{Eq } s \Rightarrow [s] \rightarrow \text{Parser } s [s]
p\text{Same } [] = \text{do } \text{return } []
p\text{Same } (s:ss) = \text{do } s' \leftarrow p\text{Sym } s
\quad ss' \leftarrow p\text{Same } ss
\quad \text{return } (s':ss')
\]

\[
p\text{Square} :: \text{Eq } s \Rightarrow \text{Parser } s [s]
p\text{Square} = \text{do } ss \leftarrow p\text{Many}
\quad ss' \leftarrow p\text{Same } ss
\quad \text{return } (ss++ss')
\]
Parsec

Parsec is a practical monadic combinator parser library written in Haskell. It has many advantages over naïve implementations. However, it is only designed to work on LL(1) grammars.
There are several nice properties of LL(1) grammars:

- They are unambiguous, so every sequence will have at most one parse result
- A parser only needs one symbol of look-ahead and thus does not require backtracking
LL(1) Grammars

The properties of LL(1) parsers combined with laziness in Haskell allow Parsec to:

- solve the space issues present in naïve parser combinator implementations
- display helpful error messages, reporting the position the error occurred at, the cause of the error, as well as all productions that would have been legal
Consider

\((p \triangleleft q)\)

On an LL(1) grammar, the parser can always decide which branch to take by looking at the current input symbol. If \(p\) consumes any input, it is not necessary to try \(q\). In either case, the current symbol does not need to be kept in memory.
There needs to be a way to tell if a parser has consumed input.

type Parser a = String -> Consumed a

data Consumed a = Consumed (Reply a) | Empty (Reply a)

data Reply a = Ok a String | Error
Basic Combinators

The `return` parser succeeds, returning its one argument, and does not consume any input.

```
return :: a -> Parser a

return x input = Empty (Ok x input)
```
The `satisfy` parser checks the current symbol against its argument and only consumes a symbol if it succeeds.

`satisfy :: (Char -> Bool) -> Parser Char`

`satisfy test [] = Empty Error`
`satisfy test (c:cs) = if test c then
    Consumed (Ok c cs)
  else
    Empty Error`
The idea behind \((p >>= q)\) as follows:

- If \(p\) succeeds without consuming input, then the result is determined by the second parser.
- If \(p\) consumes input, then the result is \textit{Consumed}. Because of lazy evaluation, this is returned right away before parsing with \(q\).
(>>=) :: Parser a -> (a -> Parser b) -> Parser b
(p >>= f) input =
    case (p input) of
      Empty reply1
        -> case (reply1) of
            (Ok x rest) -> ((f x) rest)
            Error      -> Empty Error
      Consumed reply1
        -> Consumed
            (case (reply1) of
                (Ok x rest)
                  -> case ((f x) rest) of
                      Consumed reply2 -> reply2
                      Empty reply2    -> reply2
                      error            -> error
            )
      )
The idea behind \((p <|> q)\) as follows:

- If \(p\) succeeds without consuming input, then it is a valid parse result. \(q\) is still checked in order to implement the "longest match" rule.
- If \(p\) consumes input, then since the grammar is unambiguous, \(p\) is the correct branch. Return the result from \(p\).
( <|> ) :: Parser a -> Parser a -> Parser a
(p <|> q) input =
case (p input) of
    Empty Error -> (q input)
    Empty ok    -> case (q input) of
                   Empty _  -> Empty ok
                   consumed -> consumed
    consumed -> consumed
Because of lazy evaluation, <|> obtains a result after at most one symbol has been consumed. It then knows which side to recurse to and can let go of the original input. This solves the space issues of other combinator implementations.
Since many useful grammars are not LL(1), Parsec implements a combinator that allows further look-ahead.

```haskell
try :: Parser a -> Parser a
try p input = case (p input) of
    Consumed Error -> Empty Error
    other -> other
```

try changes the Consumed Error result to Empty Error so that \( p \mid q \) combinator will always try \( q \). Putting try around every parser will allow it to be LL(\( \infty \)).
try can be used to perform lexing at the same time as parsing.

```haskell
expr = do { string "let"; whiteSpace; letExpr }
        <|> identifier
```

On input "letter", since `expr` is LL(1), will fail to recognize it as an identifier.

```
> run expr "letter"
parse error at (line 1,column 4):
unexpected "t"
expecting white space
```
Now using try:

```haskell
expr = do{ try (string "let");
          whiteSpace; letExpr }
<|> identifier
```

On input "letter", will now recognize it as an identifier.
Error Reporting

When an error occurs, Parsec reports its position, the unexpected symbol, and a list of expected productions. First, parsers must be able to know the position of the input.

type Parser a = State -> Consumed a

data State = State String Pos
Error Position

There is no reason to limit the position to being the index in the input list. For example, it is more useful to know the line number (number of newline characters before the index). The position is computed by:

```haskell
nextPos :: Pos -> Char -> Pos
```

nextPos computes the next position based on the previous position and the Char being consumed.
Error Message

Parsers must also be able to return error messages.

```haskell
data Message = Message Pos String [String]
```

Message contains the position of the error, the unexpected input, and a list of valid productions.

```haskell
data Reply a = Ok a State Message
            | Error Message
```

Note the Ok reply also carries an error message.
Labels

In order to report valid productions, there needs to be a way to label them.  `<?>` labels a production with a name.

```haskell
(<?) :: Parser a -> String -> Parser a

(p <? exp) state =
  case (p state) of
    Empty (Error msg)
      -> Empty (Error (expect msg exp))
    Empty (Ok x st msg)
      -> Empy (Ok x st (expect msg exp))
    other -> other

expect (Msg pos inp _) exp = Msg pos inp [exp]
```
Basic Parsers With Error

return always succeeds and does not consume any input. It attaches an empty message to the reply.

\[
\text{return} :: \ a \rightarrow \text{Parser} \ a \\
\text{return} \ x \ \text{state} = \\
\quad \text{Empty} \ (\text{Ok} \ x \ \text{state} \ (\text{Message} \ \text{pos} \ [] \ []))
\]
Basic Parsers With Error

satisfy needs to update the input position on success and attach an error message with the current position and symbol on failure.
Basic Parsers With Error

satisfy :: (Char -> Bool) -> Parser Char
satisfy test (State input pos) =
  case input of
    (c:cs) | test c
      -> let newPos = nextPos pos c
           newState = State cs newPos
           in seq newPos
             (Consumed
              (Ok c newState
               (Msg pos [] [])))
    (c:cs) -> Empty (Error
                      (Msg pos [c] []))
    []     -> Empty (Error
                     (Msg pos "end of input" []))
Basic idea behind \( p \lor q \):  
- When both \( p \) and \( q \) result in Empty, need to merge error messages in order to compute valid productions  
- Note that error messages should be merged even if they result in Ok  
- If either results in Consumed, return that result
The following shows why the Ok reply also needs to carry an error message:

test = do{ (digit <|> return '0')
  ; letter
}

The first symbol can be either a digit or a letter. On an illegal input, (digit <|> return '0') will have an Ok reply but digit still needs to be reported as a valid production.
((<|>) :: Parser a -> Parser a -> Parser a
(p <|> q) state =
  case (p state) of
    Empty (Error msg1)
    -> case (q state) of
      Empty (Error msg2)
      -> mergeError msg1 msg2
      Empty (Ok x inp msg2)
      -> mergeOk x inp msg1 msg2
    consumed
    -> consumed
    Empty (Ok _ inp msg1)
    -> case (q state) of
      Empty (Error msg2)
      -> mergeOk x inp msg1 msg2
      Empty (Ok _ _ msg2)
      -> mergeOk x inp msg1 msg2
    consumed
    -> consumed
  consumed -> consumed
Merging Error Messages

mergeOk x inp msg1 msg2
    = Empty (Ok x inp (merge inp1 inp2))

mergeError msg1 msg2
    = Empty (Error (merge msg1 msg2))

merge (Msg pos inp exp1) (Msg _ _ exp2)
    = Msg pos inp (exp1 ++ exp2)
Basic idea behind \( p \gg= q \):

- Note that on an `Empty` response, the error message returned is empty, so it does not matter if it is merged or not.
- For the above reason, it is correct to merge when `either` \( p \) and \( q \) result in `Empty`.
(>>=) :: Parser a -> (a -> Parser b) -> Parser b
(p >>= f) state =
  case (p state) of
    Empty reply1
      -> case reply1 of
        (Ok x inp msg1)
          -> case ((f x) rest) of
            Empty (Ok _ _ msg2) -> mergeOk x inp msg1 msg2
            Empty (Error msg2) -> mergeError msg1 msg2
            consumed            -> consumed
        error       -> error
    Consumed reply1
      -> Consumed
      (case reply1 of
        (Ok x inp msg1)
          -> case ((f x) inp) of
            Consumed reply2 -> reply2
            Empty reply2   -> reply2
            error           -> error
      )