

Advanced Programming 2022

Parsing and Parser Combinators, Continued

Andrzej Filinski
`andrzej@di.ku.dk`

Department of Computer Science
University of Copenhagen

September 22, 2022

- ▶ General motivation for learning about parsing
 - ▶ a commonly needed CS skill, not only for PL implementors
- ▶ Introduced basic notions of CFGs
 - ▶ terminals, nonterminals, productions/rules
- ▶ Introduced basics of parser combinators
 - ▶ Parser monad, simple parsers, parser-combining forms (sequences, alternatives, iteration)

Today: Some more advanced topics

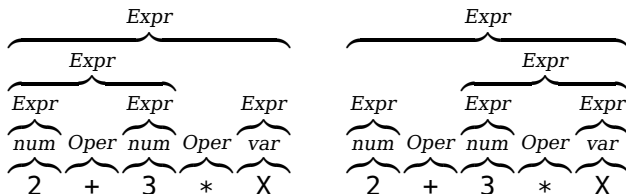
- ▶ Grammars with operator precedences and associativities
 - ▶ Disambiguating
 - ▶ Eliminating left recursion
- ▶ Lexing issues
 - ▶ Esp. whitespace (where allowed, where required)
- ▶ Parsing paradigms
 - ▶ Shallow vs. deep vs. no backtracking
 - ▶ ReadP and Parsec combinator libraries

Parsing expressions with operators

- ▶ Had definition of expressions:

$$\text{Expr} ::= \text{var} \mid \text{num} \mid \text{Expr Oper Expr} \mid '(' \text{Expr} ')'$$
$$\text{Oper} ::= '+' \mid '*'$$

- ▶ Prime example of *ambiguous* grammar: string of nonterminals “2+3*X” can be derived in two ways from *Expr*:



- ▶ Presumably only the latter grouping was intended, but grammar gives no indication of this.
 - ▶ And once input is parsed into AST, any choice of grouping is hard (or impossible) to undo.

Disambiguation-augmented grammars

- ▶ Common in practice to supplement ambiguous formal CFG with semi-formal *disambiguation* rules.
- ▶ Another classic example: “dangling else” problem

$$\begin{aligned} Stmt &::= \dots \\ &\quad | \text{ 'if' '(' Expr ')' Stmt 'else' Stmt} \\ &\quad | \text{ 'if' '(' Expr ')' Stmt} \end{aligned}$$

- ▶ How to parse “if (t1) if (t2) X=1; else X=2;”?
 - ▶ Most languages: else belongs to *innermost* unfinished if.
- ▶ Can leave to parser implementation.
 - ▶ If it has a well specified disambiguation strategy, e.g. Yacc, or our Parser monad.
- ▶ Or (often preferable): rewrite the grammar so *only* intended parse is possible.
 - ▶ Exercise! (See, e.g., Wikipedia for hint if needed.)

Operator parsing 1: precedence

- ▶ Want to express that `'*` groups tighter than `'+`.
- ▶ In general, whole hierarchy of operators
 - ▶ E.g., `'<=` groups even looser than `'+`, while `'^` (exponentiation) groups even tighter than `'*`.
- ▶ Conventional to assign to each operator a *precedence*: small(ish) number that indicates grouping strength of operator compared to others.
 - ▶ Only relative ordering matters, not magnitude of difference.
 - ▶ E.g., in Haskell, `(+)` has precedence 6, `(*)` has precedence 7.
- ▶ *Stratify* grammar according to operator precedences, from low to high:

$$\text{Expr} ::= \text{Term} \mid \text{Expr} \text{'+' Expr}$$
$$\text{Term} ::= \text{Factor} \mid \text{Term} \text{'*'} \text{Term}$$
$$\text{Factor} ::= \text{num} \mid \text{var} \mid \text{'(' Expr ')')}$$

- ▶ Now only one possible way to parse `"2+3*X"`.

Operator parsing 2: associativity

- ▶ Precedence-stratified grammar,

$$\text{Expr} ::= \text{Term} \mid \text{Expr} \text{ '+' } \text{Expr}$$

is still ambiguous: two ways to split and parse “2+3+X”.

- ▶ For addition, does not matter much which one we choose
 - ▶ Except for potential overflow and/or loss of precision.
- ▶ But if we also allow ‘-’ between terms, parsing “2-3+X” by grouping as “2- (3+X)” would be wrong.
 - ▶ As would parsing “2-3-X” as “2- (3-X)”, so not a matter of relative precedence of ‘+’ and ‘-’.
- ▶ Rather, among operators of *same* precedence, should have a well defined grouping direction (*associativity*).
- ▶ Most operators (‘+’, ‘-’) associate to the left, but some to right:
 - ▶ E.g., Haskell: ‘^’ (exponentiation), ‘:’ (cons), ‘->’ (function space [as a type constructor!])
 - ▶ ‘++’ (list/string append), though *semantically* associative (like ‘+’), is conventionally *parsed* as associating to the right. (Why?)

Disambiguating associativity

- ▶ Consider ambiguous grammar:

$$\text{Expr} ::= \text{Term} \mid \text{Expr AddOp Expr} \quad \text{AddOp} ::= '+' \mid '-'$$

- ▶ To express that *AddOps* are left-associative, take instead:

$$\text{Expr} ::= \text{Term} \mid \text{Expr AddOp Term}$$

- ▶ I.e. in a valid parse, the *right* argument of an *AddOp* cannot itself contain another *AddOp* (unless parenthesized).
- ▶ Now only one way to parse “2-3+X”.
- ▶ Symmetrically, for right-associative operators, can take:

$$\text{LExp} ::= \text{Expr} \mid \text{Expr} ':' \text{LExp}$$

- ▶ So only way to parse “2+3:4:l” as a *LExp* (where ‘+’ has higher precedence than ‘:’) is like “(2+3):(4:l)”.
- ▶ And for operators that shouldn’t associate at all:

$$\text{CExp} ::= \text{Expr} \mid \text{Expr RelOp Expr} \quad \text{RelOp} ::= '==' \mid '<' \mid \dots$$

- ▶ Then “2==3==X” is a syntax error.
- ▶ Whereas “(2==3)==X” or “2==(3==X)” would be (syntactically) OK.

Left recursion

- Consider *unambiguous* grammar:

$$\begin{aligned} \text{Exp} &::= \text{Term} \mid \text{Exp AddOp Term} & \text{AddOp} &::= '+' \mid '-' \\ \text{Term} &::= \text{Num} \mid '(' \text{Exp} ')' \end{aligned}$$

- Coded (naively/literally) with parser combinators:

```
pExp :: Parser Expr
```

```
pExp = do e1 <- pExp; ao <- pAddOp; e2 <- pTerm
      return $ ao e1 e2
      <|> pTerm -- try "always"-succeeding case last
```

```
pTerm :: Parser Expr
```

```
pTerm = do n <- pNum; return $ Num n
      <|> do symbol "("; e <- pExp; symbol ")"; return e
```

```
pAddOp :: Parser (Expr -> Expr -> Expr)
```

```
pAddOp = do symbol "+"; return Add
      <|> do symbol "-"; return Sub
```

- Can't parse input "2+2" with pExp! Infinite recursion.
- Left recursion: parser can directly or indirectly call itself, *without consuming any input* in between.

Eliminating left recursion

- ▶ Some parser generators can handle (indeed, prefer!), left-recursive grammars.
 - ▶ But for top-down parsers (incl. Parsec, ReadP): deadly.
- ▶ Note that for right-associative (or non-associative) operators, grammar is *not* left-recursive:

$$Exp ::= Term \mid Term \text{ '}' Exp \qquad Term ::= Num \mid \text{ '(' } Exp \text{ ')'}$$

- ▶ And *right* recursion in grammar is OK (for left-to-right parser).
- ▶ Unfortunately, can't just change associativity of + and – from standard mathematical practice to simplify parsing...
- ▶ Better solution: rewrite grammar to (in EBNF):

$$Exp ::= Term \{ AddOp Term \}$$

- ▶ “An expression is a term, followed by zero or more additions and/or subtractions of further terms.”

Eliminating left recursion, cont'd

- ▶ EBNF: $Exp ::= Term \{ AddOp Term \}$
- ▶ BNF: $Exp ::= Term Exp' \quad Exp' ::= \epsilon \mid AddOp Term Exp'$
- ▶ Parser-combinator code:

```
pExp :: Parser Expr
pExp = do e <- pTerm; pExp' e

pExp' :: Expr -> Parser Expr -- argument is "accumulator"
pExp' e1 = do ao <- pAddOp; e2 <- pTerm; pExp' (ao e1 e2)
          <|> return e1 -- try trivial alternative last
```
- ▶ Can extract above pattern into *utility* combinator chainl1:

```
chainl1 :: Parser a -> Parser (a -> a -> a) -> Parser a
p `chainl1` po = do a <- p; p' a where
  p' a1 = do o <- po; a2 <- p; p' (a1 `o` a2)
          <|> return a1
```



```
pExp = pTerm `chainl1` pAddOp
```

Left factoring

- ▶ Consider a grammar with a *right*-associative operator

$$LExp ::= Exp \mid Exp \text{ ':' } LExp$$

- ▶ No left-recursion (assuming *Exp* expressed properly).
- ▶ But can't tell up front which of the two alternatives to use.
 - ▶ Necessitates backtracking parser, and sometimes wastes work.
- ▶ Since both alternatives start with *Exp*, can parse *Exp* unconditionally first, and *then* choose:

$$LExp ::= Exp \ LExp' \quad LExp' ::= \epsilon \mid \text{' ':' } LExp$$

```
p `chainr1` po = do a <- p; p' a where
  p' a1 = do o <- po; a2 <- p `chainr1` po; return (a1 `o` a2)
  <|> return a1
pLExp = pExp `chainr1` (do symbol ":"; return Cons)
```

- ▶ Other opportunities for left-factoring abound, e.g., in:

$$S ::= \dots \mid \text{'if' } E \text{ 'then' } S \text{ 'fi'} \mid \text{'if' } E \text{ 'then' } S \text{ 'else' } S \text{ 'fi'}$$

Whitespace

- ▶ Most grammars allow arbitrary whitespace (including comments) between tokens:

$$\textit{Whitespace} ::= \epsilon \mid (\text{ ' ' } \mid \text{ '\t' } \mid \text{ '\n' } \mid \textit{Comment}) \textit{Whitespace}$$

- ▶ Do **not** want to insert *Whitespace* between all pairs of adjacent symbols in productions.
 - ▶ Nor explicit calls to whitespace-skipping throughout the parser!
- ▶ Need a systematic approach: make all *token parsers* (alone) responsible for skipping adjacent whitespace.
 - ▶ Clearly enough to skip *before* each token, and at *very end* of input; or vice versa.
- ▶ For many parser libraries, usually much preferable to skip ws *after* each token (and at *very beginning*)
 - ▶ Invariant: each parser will immediately see first *real* char of input.
 - ▶ Avoids re-skipping whitespace at start of every alternative.
 - ▶ Much like left-factoring the grammar.

Skipping whitespace in parsers

- Easy to add whitespace-skipping to token parsers:

```
whitespace :: Parser ()
```

```
whitespace = -- (for ReadP: use munch or skipSpaces)  
do many (satisfy isSpace); return ()
```

*-- use only with *actual* lexeme/token parsers!*

```
lexeme :: Parser a -> Parser a
```

```
lexeme p = do a <- p; whitespace; return a
```

```
symbol :: String -> Parser ()
```

```
symbol s = lexeme $ do string s; return ()
```

```
pNum :: Parser Int
```

```
pNum = lexeme $ do ds <- many1 (satisfy isDigit)  
return $ read ds
```

...

Token separation

- ▶ Sometimes whitespace is *required* to separate adjacent tokens.
- ▶ Consider, e.g., grammar:

$$\text{Expr} ::= \dots \mid \text{'let' Var '=' Expr 'in' Expr}$$

- ▶ How to define keyword `:: String -> Parser ()`, for

```
pExpr = do keyword "let"; v <- pPvar; symbol "="  
          e1 <- pExpr; keyword "in"; e2 <- pExpr  
          return $ Let v e1 e2
```
- ▶ Naive approach: keyword = symbol
 - ▶ Would accept "letx=5inx+1": probably undesirable.
- ▶ On the other hand, "let x=5in(x+1)" is OK, so can't simply require *at least one* whitespace char after keywords, either.
- ▶ Workable solution: read entire word first, then check at end:

```
keyword s = lexeme $ do s' <- many1 (satisfy isAlphaNum)  
                        if s' == s then return ()  
                        else pfail $ "expected " ++ s
```

Delimiting keywords, continued

- ▶ Previous solution is slightly wasteful
 - ▶ Will only detect mismatch after reading entire word, even if differs from expected keyword on first char.
 - ▶ Repeats work when used in alternatives.

- ▶ Alternative approach: *negated parsers*

```
newtype Parser a = P {runP :: String -> Either ParseError (a, String)}  
notFollowedByAny :: (Char -> Bool) -> Parser ()  
-- notFollowedBy p will succeed (without consuming anything)  
-- iff input *does not* start with a character satisfying p.  
notFollowedBy p =  
    P (\s -> case runP (satisfy p) s of  
        Right _ -> Left "not allowed here"  
        Left _ -> Right ((), s))
```

```
keyword s = lexeme $ do string s; notFollowedByAny isAlphaNum  
eof = notFollowedByAny (const True) -- succeeds iff at end
```


Keywords, concluded

- ▶ Final twist: keywords are often reserved
- ▶ So cannot use for variable names:

```
reserved :: [String]
```

```
reserved = ["if", "for", "while", "do", ...]
```

```
type Var = String
```

```
pVar :: parser Var
```

```
pVar = lexeme $
```

```
  do c <- satisfy isAlpha
```

```
    cs <- many (satisfy isAlphaNum)
```

```
    let i = c:cs
```

```
    if i `notElem` reserved then return i
```

```
      else pfail "variable can't be a reserved word"
```

Lookahead and backtracking

- ▶ For alternatives, our Parser tries all productions in turn, until one succeeds.
- ▶ In $A ::= \alpha_1 \mid \alpha_2$, a parsing failure anywhere within α_1 will cause α_2 to be tried.
- ▶ But if parsing all of α_1 succeeds, α_2 is discarded
- ▶ Sometimes known as *shallow backtracking*.
 - ▶ Allows unlimited lookahead when picking alternative.
 - ▶ Ordering of alternatives is significant.
- ▶ Nice balance between convenience and efficiency, but not only possible design choice.

Shallow backtracking is not always enough

- ▶ **Ex:** “An A is zero or more ‘x’s and/or ‘y’s, followed by a ‘z’.”

$$A ::= B \text{ 'z' } \quad B ::= \text{'x'} B \mid \text{'y'} B \mid \epsilon$$

- ▶ Greedy approach for parsing B will work fine.
- ▶ **Ex:** “An A is zero or more ‘x’s and/or ‘y’s, followed by an ‘x’.”

$$A ::= B \text{ 'x' } \quad B ::= \text{'x'} B \mid \text{'y'} B \mid \epsilon$$

- ▶ Greedy parsing of B may eat too many characters, causing rest of A to fail!
- ▶ Need to rewrite grammar to not require lookahead, e.g.:

$$A ::= \text{'x'} C \mid \text{'y'} A \quad C ::= \text{'x'} C \mid \text{'y'} A \mid \epsilon$$

- ▶ String can end after an ‘x’, but not after a ‘y’.

Alternative: deep backtracking parser

- Consider grammar:

$$A ::= \alpha B \gamma \mid \delta \quad B ::= \beta_1 \mid \beta_2$$

- If parsing γ fails, could try *different* way of parsing B , instead of jumping straight to δ (as in shallow backtracking).
 - E.g, β_2 instead of β_1
 - Note that β_1 and β_2 might consume different amounts of input, e.g., if $\beta_1 = b$ and $\beta_2 = \epsilon$.
 - “No choice is final until *entire* input successfully parsed”.
- **Idea:** make the parser return *all possible* ways of parsing a nonterminal at beginning of string.
- Only minimal changes required in Parser monad:

```
-- was: P { runP :: String -> Either ParseError (a, String) }
newtype Parser a = P { runP :: String -> [(a, String)] }
instance Monad Parser where -- code identical to before:
  return a = P (\s -> return (a,s)) -- builds on [] monad!
  m >=> f = P (\s -> do (a,s') <- runP m s; runP (f a) s')
  pfail e = P (\s -> []) -- ignores the message
```

List-based parsing, continued

- Also, a few simple changes in core parser combinators:

```
newtype Parser a = P { runP :: String -> [(a, String)] }
```

```
getc :: Parser Char
```

```
getc = P (\s -> case s of "" -> []; (c:s') -> return (c,s'))
```

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

```
p1 <|> p2 = P (\s -> runP p1 s ++ runP p2 s)
```

```
notFollowedByAny p =
```

```
  P (\s -> case runP (satisfy p) s of  
    [] -> return ((), s)  
    _ -> [])
```

```
parseString :: Parser a -> String -> Either ParseError a
```

```
parseString p s =
```

```
  case runP (do whitespace; a <- p; eof; return a) s of  
    [] -> Left "cannot parse"  
    [(a,_)] -> Right a -- the _ must be "", since 'eof' ok  
    _ -> error "oops, my grammar is ambiguous!"
```

Pros and cons of deep backtracking

- ▶ Some gain in convenience (can handle more grammars directly).
- ▶ Potentially excessive backtracking.

- ▶ Easy to induce quadratic, or even exponential, behavior.

- ▶ Worse: may split tokens in unexpected places

- ▶ E.g., for `pNum :: Parser Int`, defined exactly as before:
`runP pNum "123!" ≈ [(123,"!"), (12,"3!"), (1,"23!")]`

- ▶ Sometimes need to explicitly force longest parse:

```
munch :: (Char -> Bool) -> Parser [Char]
munch p = do as <- many (satisfy p); notFollowedByAny p
           return as
```

- ▶ But can be implemented much more efficiently:

```
p1 <<|> p2 = -- like <|>, but only tries p2 if p1 fails
P (\s -> case runP p1 s of [] -> runP p2 s; l -> l)
```

```
munch p = do c <- satisfy p; cs <- munch p; return (c:cs)
           <<|> return [] -- note: "many" uses <|> here
```

ReadP parser library

- ▶ Behaves like list-based parser on previous slides (as further described in “Monadic Parsing” article), but internally implemented more efficiently.
- ▶ Uses `+++` for symmetric choice instead of `<|>`, and a few other naming differences.
 - ▶ May want to use compatibility def: `(<|>) = (+++)` to avoid hardcoding dependence on ReadP in all rules.
- ▶ See Hutton & Meijer article for principles, and Hoogse ReadP for full API.
 - ▶ Note: use `readP_to_S` instead of `runP` to invoke top-level parser.
- ▶ Welcome to use for AP assignments, but beware of pitfalls from previous slides...
 - ▶ Also, absolutely no feedback on errors!
 - ▶ Hint: use (approximate) *bisection* to track down location of parsing errors when debugging grammar and/or input strings.

Other extreme: non-backtracking parsers

- ▶ Also possible to parse without backtracking/lookahead.
 - ▶ Potentially more efficient
 - ▶ (Mainly because uses more *programmer* effort to transform the grammar into suitably restricted form first.)
 - ▶ In particular, *requires* manual left-factorization.
- ▶ In $A ::= \alpha_1 \mid \alpha_2$, commit to α_1 branch as soon as it successfully parses the *first* input token.
 - ▶ Only tries α_2 if α_1 failed *straight away*.
 - ▶ Actually OK for many practical grammars (LL(1) class).
- ▶ By default, lookahead is only one *character*.
 - ▶ Not enough to distinguish, e.g., throw and try at start of a sentence.
 - ▶ Or between *any* keyword and, e.g., a variable in an assignment, or procedure name in a call.

Limited backtracking: try

- ▶ To see more of the input before committing, need extra combinator:

```
try :: Parser p -> Parser p
```

- ▶ try p tries to run p, but if it fails *anywhere*, pretends that it failed already on first input character.
- ▶ Typical usage:

```
pStmt = do symbol "{"; ...      -- no "try" needed here
          <|>
          do try (keyword "if"); ... -- don't commit after "i"
          <|>
          do v <- pVar; symbol "="; ... -- nothing else to try
```
- ▶ try can actually span over any parsers, not just single tokens.
 - ▶ Extreme case: try p₁ <|> p₂ simulates unbounded (shallow) backtracking.
 - ▶ But negates advantages of backtracking-less parser.
 - ▶ Principle: only “try”-protect as much of each alternative as needed to determine that none of the following will work.

Parsec parser library

- ▶ Efficient, non-backtracking (by default) parser.
- ▶ See Leijen & Meijer article for principles, and Hoogse Parsec for full API.
- ▶ Perhaps main advantage: gives pretty good error messages out-of-the-box.
 - ▶ Location of error (line & column)
 - ▶ List of tokens (or higher-level symbols) valid at this point.
 - ▶ Can improve error messages further by extra hints.
 - ▶ Don't waste time on that for *AP*!
- ▶ For post-*AP* work in Haskell, may want to consider Megaparsec, or other recent parsing libraries.
 - ▶ Original Parsec starting to show its age.
 - ▶ But beware of excessive hype/FUD, spotty documentation from potential replacements.

What next

- ▶ Assignment 3 (parsing Boa) is out, due on Friday, 30/9
 - ▶ Warmup part: write a much smaller parser using ReadP (and optionally also Parsec)
- ▶ Labs Thursday: as usual, see Absalon for rooms.
- ▶ Assignment 2 due tomorrow
 - ▶ Do run your `code.zip` past OnlineTA before submitting it!
 - ▶ May find both fatal packaging errors and subtle corner cases.
- ▶ Next week(s): Erlang!
 - ▶ But you haven't seen the last of Haskell yet...