Advanced Programming 2017

Parsing and Parser Combinators, Continued

Andrzej Filinski
andrzej@di.ku.dk

Department of Computer Science
University of Copenhagen

September 21, 2017

Last time

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

Today: Some more advanced topics

- Grammars with operator precedences and associativities
  - Eliminating left recursion
- Lexing issues
  - Esp. whitespace (where allowed, where required)
- Parsing paradigms
  - Shallow vs deep vs no backtracking
  - `ReadP` and `Parsec` combinator libraries
- A bit about Assignment 2 skeleton

Parsing expressions with operators

- Had definition of expressions:

$$
\begin{aligned}
\mathit{Expr} &::= \mathit{Var} \mid \mathit{Num} \mid \mathit{Expr}\ \mathit{Oper}\ \mathit{Expr} \mid \text{‘}\texttt{(}\text{’}\ \mathit{Expr}\ \text{‘}\texttt{)}\text{’} \\
\mathit{Oper} &::= \text{‘}\texttt{+}\text{’} \mid \text{‘}{*}\text{’}
\end{aligned}
$$

- Prime example of *ambiguous* grammar: string of nonterminals "2+3∗X" can be derived in multiple ways from *Expr*:

$$
\overbrace{\overbrace{\overbrace{\overbrace{\texttt{2}}^{\mathit{Num}}}^{\mathit{Expr}}\ \overbrace{\texttt{+}}^{\mathit{Oper}}\ \overbrace{\overbrace{\texttt{3}}^{\mathit{Num}}}^{\mathit{Expr}}}^{\mathit{Expr}}\ \overbrace{*}^{\mathit{Oper}}\ \overbrace{\overbrace{\texttt{X}}^{\mathit{Var}}}^{\mathit{Expr}}}^{\mathit{Expr}}
\qquad
\overbrace{\overbrace{\overbrace{\texttt{2}}^{\mathit{Num}}}^{\mathit{Expr}}\ \overbrace{\texttt{+}}^{\mathit{Oper}}\ \overbrace{\overbrace{\overbrace{\texttt{3}}^{\mathit{Num}}}^{\mathit{Expr}}\ \overbrace{*}^{\mathit{Oper}}\ \overbrace{\overbrace{\texttt{X}}^{\mathit{Var}}}^{\mathit{Expr}}}^{\mathit{Expr}}}^{\mathit{Expr}}
$$

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

Disambiguation-augmented grammars

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

$$\begin{array}{rcl} \mathit{Stmt} & ::= & \cdots \\ & | & \text{‘}\texttt{if}\text{’ ‘}\texttt{(}\text{’ } \mathit{Expr} \text{ ‘}\texttt{)}\text{’ } \mathit{Stmt} \text{ ‘}\texttt{else}\text{’ } \mathit{Stmt} \\ & | & \text{‘}\texttt{if}\text{’ ‘}\texttt{(}\text{’} \mathit{Expr} \text{ ‘}\texttt{)}\text{’ } \mathit{Stmt} \end{array}$$

  - How to parse "`if (t1) if (t2) X=1; else X=2;`"?
  - Most languages: `else` belongs to *innermost* enclosing `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 even possible.
  - Exercise! (See, e.g., Wikipedia for hint if needed.)

Operator parsing 1: precedence

- Want to express that ‘$*$’ binds tighter than ‘+’
- In general, whole hierarchy of operators (e.g, ‘<=’ binds even looser than ‘+’, while ‘^’ (exponentiation) binds tighter than ‘$*$’)
- Conventional to assign to each operator a *precedence*: small(ish) number, that indicates binding strength of operator compared to others.
  - Only relative ordering matters, not magnitude of distance
  - E.g., in Haskell, `(+)` has precedence 6, `(*)` has precedence 7.
- *Stratify* grammar according to operator precedences

$$
\begin{aligned}
\mathit{Expr} &::= \mathit{Term} \mid \mathit{Expr}\ \text{‘+’}\ \mathit{Expr} \\
\mathit{Term} &::= \mathit{Factor} \mid \mathit{Term}\ \text{‘}{*}\text{’}\ \mathit{Term} \\
\mathit{Factor} &::= \mathit{Num} \mid \mathit{Var} \mid \text{‘(’}\ \mathit{Expr}\ \text{‘)’}
\end{aligned}
$$

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

Operator parsing 2: associativity

- Precedence-stratified grammar,

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

  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 it 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, is also *parsed* as associating to the right (why?)

Disambiguating associativity

- Consider ambiguous grammar:
  $$Expr ::= Term \mid Expr\ AddOp\ Expr \qquad AddOp ::= \text{‘}\texttt{+}\text{’} \mid \text{‘}\texttt{-}\text{’}$$
- To express that *AddOp*’s are left-associative, take instead:
  $$Expr ::= Term \mid Expr\ Addop\ Term$$
  - I.e. in a valid parse, the RHS of an *AddOp* cannot contain another *AddOp* (unless parenthesized).
  - Now only one way to parse “`2-3+X`”.
- Symmetrically, for right-associative operators, can take:
  $$LExpr ::= Expr \mid Expr\ \text{‘}\texttt{:}\text{’}\ LExpr$$
  - So only way to parse “`2+3:4:l`” as a *LExpr* (where ‘`+`’ has higher precedence than ‘`:`’) is like “`(2+3):(4:l)`”.
- And for operators that shouldn’t associate at all:
  $$CExpr ::= Expr \mid Expr\ RelOp\ Expr \qquad Relop ::= \text{‘}\texttt{==}\text{’} \mid \text{‘}\texttt{<}\text{’} \mid \cdots$$
  - Then “`2==3==X`” is a syntax error.
  - Whereas “`(2==3)==X`” or “`2==(3==X)`” would be (syntactically) OK.

Left recursion

- Consider simple, unambiguous grammar:
  $Exp ::= Exp\ AddOp\ Term \quad Term ::= Num \mid \text{‘('}\ Exp\ \text{‘)'} \quad AddOp ::= \text{‘+'} \mid \text{‘-}$
- What if we code it with parser combinators:

```
pExp :: Parser Expr
pExp = do e1 <- pExp; ao <- pAddOp; e2 <- pTerm;
          return $ ao e1 e2
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 even parse input "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 recursive-descent parsers (incl. `Parsec`, `ReadP`): deadly.
- Note that for right-associative (or non-associative) operators, grammar is *not* left-recursive:

$$\mathit{Exp} ::= \mathit{Term} \mid \mathit{Term}\ `\texttt{:}'\ \mathit{Exp} \qquad \mathit{Term} ::= \mathit{Num} \mid `\texttt{(}'\ \mathit{Exp}\ `\texttt{)}'$$

  - And right-recursion in grammar is fine 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):

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

  - Expression is a term, followed by zero or more additions or subtractions of terms.

Eliminating left recursion, cont'd

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

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

pExp' :: Expr -> Parser Expr
pExp' e1 = do ao <- pAddOp; e2 <- pTerm; pExp' (ao e1 e2)
          <|> return e1
```

- Can extract above pattern into *utility* combinator `chainl1`:

```haskell
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 a

pExp = pTerm `chainl1` pAddOp
```

Left factoring

- Consider a grammar with a right-associative operator

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

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

$$LExp ::= Exp\ LExp' \qquad LExp' ::= \epsilon \mid \text{`}\texttt{:}\text{' } LExp$$

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

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

$$S ::= \ldots \mid \text{`}\texttt{if}\text{' } E \text{ `}\texttt{then}\text{' } S \text{ `}\texttt{fi}\text{'} \mid \text{`}\texttt{if}\text{' } E \text{ `}\texttt{then}\text{' } S \text{ `}\texttt{else}\text{' } S \text{ `}\texttt{fi}\text{'}$$

Whitespace

- Most grammars allow arbitrary whitespace between tokens:

$$\mathit{Whitespace} ::= \epsilon \mid (\text{‘ ’} \mid \text{‘}\texttt{\textbackslash t}\text{’} \mid \text{‘}\texttt{\textbackslash n}\text{’})\ \mathit{Whitespace}$$

- 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 *token* parsers responsible for skipping adjacent whitespace.
  - Clearly enough to skip *before* each token, and at *very end*; or vice versa.
- In fact, much preferable to skip *after* tokens (and at *very beginning*)
  - Invariant: each terminal parser will 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 to whitespace-skipping parser builder:

```
whitespace :: Parser ()
whitespace = -- better: use skipMany/munch combinator
  do many (satisfy isSpace); return ()

token :: Parser a -> Parser a
token p = do a <- p; whitespace; return p

symbol :: String -> Parser ()
symbol s = token $ string s

pNum :: Parser Int
pNum = token $ do ds <- many1 (satisfy isDigit)
                  return $ read ds
...
```

Token separation

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

$$\mathit{Expr} ::= \cdots \mid \text{‘}\texttt{let}\text{’}\ \mathit{Var}\ \text{‘}\texttt{=}\text{’}\ \mathit{Expr}\ \text{‘}\texttt{in}\text{’}\ \mathit{Expr}$$

- How to define `keyword :: String -> Parser ()`?
- Naive approach: `keyword = string`
  - Would accept “`letx=5inx+1`”: probably undesirable.
- On the other hand, “`let x=5in(x+1)`” is OK, so can’t just require at least one whitespace char after keywords, either.
- Workable solution: read entire *word* first, then check at end:

```
keyword s = do s' <- many1 (satisfy isAlphaNum)
               if s' == s then return ()
               else fail $ "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*

```
notFollowedBy :: Parser a -> Parser () -- slightly odd name
-- notFollowedBy p will succeed (without consuming anything)
-- iff input string does not start with a p.
notFollowedBy p =
  P (\s -> case runP p s of
             Right _ -> Left "illegal here"
             Left _ -> Right ((), s))

keyword s = token $ do string s
                       notFollowedBy (satisfy isAlphaNum)
eof = notFollowdBy getc  -- succeeds iff at end of input
```

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 =
  do c <- satisfy isAlpha
     cs <- many (satisfy isAlphaNum)
     let i = c:cs
     if i `notElem` reserved then return i
     else fail "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 succeds, 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

- Example: "An A is zero or more 'x's and 'y's, followed by a 'z'."

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

  - Greedy approach for parsing B will work fine.
- Example: "An A is one or more 'x's and 'y's, ending in an 'x'."

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

  - 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.:

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

  - String can end after an 'x', but not after a 'y'.

Alternative: deep backtracking parser

- Consider grammar:

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

- If parsing γ fails, could try different way of parsing B instead of jumping straight to 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 sucessfully parsed".
- Idea: make the parser returns *all possible* ways of parsing a nonterminal at beginning of string.
- Only minimal changes required in `Parser` monad:

```
Parser a = P { runP :: String -> [(a, String)] }
instance Monad Parser where
  return a = P (\s -> return (a,s)) -- builds on [] monad!
  m >>= f = P (\s -> do (a,s') <- runP m s; runP (f a) s')
  fail e = P (\s -> []) -- ignore message
```

List-based parsing, continued

- Also a few changes in basic parser combinators:

```
-- 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)

notFollowedBy p =
  P (\s -> case runP p s of [] -> return (); _ -> [])

parseString :: Parser a -> String -> Either ParseError a
parseString p s =
    case run (do a <- p; eof; return p) of
    [] -> Left "cannot parse"
    [(a,_)] -> Right a  -- the _ will be "" due to eof parser
    _ -> error "looks like 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 expontential, 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)
             notfollowedby (satisfy 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 (or as described in Hutton article), but internally implemented more efficiently.
- Uses `+++` for symmetric choice instead of `<|>`, and a few other naming differences.
- Hoogle for `ReadP` to see full API.
- 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.
  - Potentially more efficient
  - (Mainly because uses more programmer effort to transform the grammar into suitably restricted form first).
    - In particular, manual left-factorization.
- In $A ::= \alpha_1 \mid \alpha_2$, commit to α_1 branch as soon as it parses the first token of the input.
  - 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, pretend that it failed already on first input character.
- Typical usage:

  ```
  pStmt = do symbol "{"; ... -- no "try" needed here
          <|>
          do try (keyword "if"); ...
          <|>
          do v <- pVar; symbol "="; ... -- nor here
  ```

- `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 parser.
- See Leijen article for principles, and Hoogle for `Parsec` to see 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*!

Something completely different

- In SubScript assignments, parser (and interpreter, for that matter) module have very short export lists.
  - Avoids leaking internal implementation details.
  - Avoids clashing with client's own functions/types on bulk import.
    - Never export more than listed in documentation!
- But then, how to thoroughly unit-test all the internal functions?
  - Don't put testing code into implementation!
- Solution: split into two modules:
  - *Implementation module* exports everything.
  - *Interface module* imports from the implementation and re-exports only the required names.
- Link against (import from) appropriate module:
  - Actual clients (and black-box testing code) link against interface module.
  - White-box testing code links against implementation module.
- Already set up this way in skeleton code for Assignment 2.

What next

- Assignment 2 is out, due on Wednesday
- Labs today 12:45–15:00
  - Room A110 discontinued due to disuse; go to one of the others.
- Next week: Prolog and logic programming
  - Will see deep backtracking again!