Advanced Programming 2017

Introduction to Parsing and Parser Combinators

Andrzej Filinski
`andrzej@di.ku.dk`

Department of Computer Science
University of Copenhagen

September 19, 2017

Background

- In practical applications, often have to read in structured textual data for further processing.
- Could be actual program (script) code, but also serialized database dumps, marked-up text, web page templates, configuration files, build rules, ...
- If expressed in a standard format (XML, JSON, YAML, ...), can probably find a ready-made reader.
  - Though maybe not *exactly* for your language/platform
  - Or for the *almost*-standard format you actually need to read.
    - Proprietary/experimental extensions
    - Standards version skew
    - Not-strictly compliant data
- But otherwise, you're on your own.
  - As a computer scientist, you'll need to be able to cope.

Introduction to parsing

- *Parsing* is (for our purposes) the process of constructing
  - … a *structured* representation of data in memory …
  - … from its *linearized* representation as a text string.
- The structured representation is usually a simple data structure, the *abstract syntax tree (AST)*.
  - Haskell's (and other functional languages') algebraic data types (`data`) are an excellent match for this.
  - Have already seen for Assignment 1.
- The description of the linearized format is usually given by a *grammar*.
  - Describes *concrete syntax* of language being parsed
    - And (where non-obvious) how the concrete syntax corresponds to the abstract.
  - Must be suitable for machine processing
    - Sometimes a few tweaks are necessary

Parsing principles

- Most languages (both human and machine-oriented) have natural two-level structure:
- The *lexical* level deals with rules for the structure of the atomic *tokens* of the language:
  - Exact rules for format of numerals, identifiers, keywords, operator symbols, ...
    - What goes into *leaves* (or at least single nodes) of the AST
  - Often have finitary, or simple iterative, description:
    - "A keyword is one of the following: `if`, `then`, `else`, `for`, `do`, ..."
    - "A valid identifier is a letter followed by zero or more letters, digits, or underscores"
  - Also, specifies general lexical properties not associated with individual tokens, e.g.:
    - case sensitivity of identifiers/keywords
    - whitespace rules (i.e., where allowed/required)
    - comment format(s)

Parsing principles, continued

- The *syntactic* level deals with rules for how tokens may be composed into higher-level units: expressions, statements, functions, modules, ...
  - Typically reflects the structure of the AST
    - But with a little more "noise": parentheses, separator symbols for lists ("`,`" vs. "`;`" vs. "` `"), ...
  - Almost always involves non-trivial recursion:
    - An *expression* is a number, or a variable, or two *expressions* separated by an infix operator, or ...
    - A *statement* is an assignment of an expression to a variable, or a sequence of *statements* enclosed in braces, or ...
- In practice, boundary between lexical and syntactic levels often a bit fuzzy (more later).
  - "Scannerless parsing": unified view.

Formal grammars

- A grammar says how individual characters or tokens (called *terminal symbols* or just *terminals*) are arranged into higher-level constructs (*nonterminals*).
- Grammar contains a collection of *productions*, specifying how each nonterminal may be built out of sequences of *zero or more* terminals and/or nonterminals.
- One of the nonterminals is the designated *start* symbol.
  - "Entry point" of the grammar for a single parsing run.
  - Cf. `main` function in a C/Java/Haskell program.
- **Meta-**notation: when talking about grammars in general, will use:
  - uppercase letters (A, B, C, ...) for nonterminals,
  - lowercase letters (a, b, c, ...) for terminals
  - (lowercase) Greek letters (α, β, γ, ...) for (possibly empty) sequences of terminals and nonterminals.

Context-free grammars

- A CFG (aka "BNF grammar": for "Backus–Naur Form" or "Backus Normal Form") has the following general shape ("ragged matrix"):

$$\begin{array}{rcl} A_1 & ::= & \alpha_{11} \mid \cdots \mid \alpha_{1n_1} \\ & \vdots & \\ A_m & ::= & \alpha_{m1} \mid \cdots \mid \alpha_{mn_m} \end{array}$$

- Informally: (mutually) recursive definitions of all the nonterminals.
- Important subclass of CFGs: *regular grammars*; nonterminals are only allowed as *last* element of an α_{ij} ("tail recursion").
- Also definable: *context-sensitive*, or even more general grammars, may also have additional symbols on *LHS*s of productions.
  - Rarely used in CS practice, but more commonly in linguistics.

Writing specific grammars

- Many common concrete conventions for writing actual grammars; will generally use the following for *AP*:
  - nonterminals written as capitalized identifiers in *italics*
  - terminals written as character sequences between single quotes and in `typewriter` font
  - symbols in sequences are separated by spaces.
  - empty sequence of symbols written as ϵ (instead of nothing)
- E.g.. syntactic grammar for small imperative language:

$$
\begin{array}{rcl}
\mathit{Stmt} & ::= & \mathit{Var}\ \text{‘}\texttt{=}\text{’}\ \mathit{Expr}\ \text{‘}\texttt{;}\text{’} \\
 & \mid & \text{‘}\texttt{\{}\text{’}\ \mathit{Stmts}\ \text{‘}\texttt{\}}\text{’} \\
\mathit{Stmts} & ::= & \epsilon \\
 & \mid & \mathit{Stmt}\ \mathit{Stmts}
\end{array}
\qquad
\begin{array}{rcl}
\mathit{Expr} & ::= & \mathit{Number} \\
 & \mid & \mathit{Var} \\
 & \mid & \mathit{Expr}\ \mathit{Oper}\ \mathit{Expr} \\
 & \mid & \text{‘}\texttt{(}\text{’}\ \mathit{Expr}\ \text{‘}\texttt{)}\text{’} \\
\mathit{Oper} & ::= & \text{‘}\texttt{+}\text{’} \mid \text{‘}\texttt{*}\text{’}
\end{array}
$$

(Assumes *Number*, *Var* defined elsewhere)

Lexical grammars

- Lexical structure of tokens can also be described by grammars.
- Example: lexical grammar for variable names:

$$
\begin{array}{rcl}
\mathit{Var} & ::= & \mathit{VHChar}\ \mathit{VarRest} \\
\mathit{VarRest} & ::= & \epsilon \mid \mathit{VRChar}\ \mathit{VarRest} \\
\mathit{VHChar} & ::= & \text{‘}\mathtt{A}\text{’} \mid \cdots \mid \text{‘}\mathtt{Z}\text{’} \\
\mathit{VRChar} & ::= & \mathit{VHChar} \mid \text{‘}\mathtt{0}\text{’} \mid \cdots \mid \text{‘}\mathtt{9}\text{’} \mid \text{‘}\_\text{’}
\end{array}
$$

- "…" is *informal* range notation. In completely formal grammar, would have to write out all alternatives.
- Not technically a *regular grammar* (not "tail recursive")
  - But could be turned into one, by unfolding definitions of character classes and refactoring.
- In practice, lexical grammar often specified (partially) textually.
  - Character classes may be huge (e.g., letters in Unicode)
  - Avoids clutter in syntactic grammar to express that arbitrary whitespace or comments allowed between proper tokens.

Extended BNF (EBNF) notation.

- Often seen in real-world specifications: regexp-like extensions for writing CFGs a bit more concisely.
  - Easily “desugared” into plain BNF.
- Internal grouping:

$$A ::= \alpha \; (\; \beta_1 \mid \cdots \mid \beta_n) \; \gamma \quad \rightsquigarrow \quad \begin{array}{l} A ::= \alpha \; B \; \gamma \\ B ::= \beta_1 \mid \cdots \mid \beta_n \end{array}$$

- Optional elements:

$$A ::= \alpha \; [\; \beta \;] \; \gamma \quad \rightsquigarrow \quad \begin{array}{l} A ::= \alpha \; B \; \gamma \\ B ::= \beta \mid \epsilon \end{array}$$

- Iterated elements

$$A ::= \alpha \; \{ \; \beta \; \} \; \gamma \quad \rightsquigarrow \quad \begin{array}{l} A ::= \alpha \; B \; \gamma \\ B ::= \beta \; B \mid \epsilon \end{array}$$

- Notation can also be nested or used multiple times in single production, but with diminishing readability gains.

Derivations and parsing

- Given grammar (regular or context free), can say that a nonterminal A *derives* string of terminals α, written $A \Rightarrow \alpha$.
- Begin with start symbol, repeatedly replace some nonterminal with *one* of its production RHSs, until until only terminals left.
- Example (with selected nonterminal underlined in each step)
  $\underline{\mathit{Expr}} \Rightarrow \underline{\mathit{Expr}}\ \texttt{+}\ \mathit{Expr} \Rightarrow \underline{\mathit{Num}}\ \texttt{+}\ \mathit{Expr} \Rightarrow \texttt{4 +}\ \underline{\mathit{Expr}} \Rightarrow \texttt{4 +}\ \underline{\mathit{Var}} \Rightarrow \texttt{4 + X}$
- For parsing, considering the opposite problem: given sequence of terminals, can it be derived from the start symbol?
  - And by which exact productions? (To get *parse tree*)
- Not obvious that this is even effectively decidable.
  - Worst case "only" $O(n^3)$, where n is length of input [Earley]
  - For "well behaved" grammars, can do significantly better, often close to $O(n)$.
  - In practice, most grammars are indeed well behaved.
    - If hard to parse for a computer, not great for humans either
    - … even if brain *may* have "built-in" parsing acceleration [Chomsky]

Parsing in practice

- Traditional compiler wisdom: lexing done with regular grammars (regexps), parsing with CFGs.
- In practice: not quite.
- Real languages (especially legacy ones) often have mildly context-sensitive (or worse) lexing or parsing rules that defy easy categorization. Some random examples:
    - Fortran 66 “Hollerith constants”: `CALL WRITE(12HHELLO WORLD!)`
    - Programmer-specified operator precedence and associativity: Haskell’s `infixr 5 ++`. (In SML, can even be lexically scoped!)
    - C’s “typedef problem”: is `(a)*b` a cast of a pointer dereference, or a multiplication? Depends on prior declaration of `a`.
    - Non-reserved keywords (Fortran, PL/I), e.g., `FORI` may be variable name, or start of a `FOR`-loop, depending on context.
    - Indentation sensitivity (Python, Haskell).

Parsing tools

- Parsing according to a CFG may look scary.
- General parser-generator tools exist: Lex/Flex, Yacc/Bison, ...
  - (Quasi-)ports to various languages, including Haskell (Alex, Happy)
  - Translate a grammar specification into (often table-based) parsing program.
- Often give best absolute performance:
  - Can use fancier parsing algorithms that require heavy preprocessing
  - Avoid “interpretation overhead” at parsing time.
- But for majority of applications, lexing/parsing is far from the most time-consuming part.
  - So performance matters less than programmer productivity.
- Also, need various hacks to handle complications from previous slide.

Parser combinators

- For many purposes, preferable to hand-code a parser from (almost) scratch, using some variant of *recursive-descent* parsing.
- Can "escape" to full host language whenever needed during parsing.
- Particularly convenient in Haskell:
  - Small library of *parser combinators* make parser specification almost as concise and readable as grammar itself
  - Lazy evaluation and infix operators help keep notation extra compact.
  - Monad-based approach makes it easy to incorporate backtracking (if/when desired), so lookahead is not a problem.
  - Additional features and optimizations can also be pushed into the combinators by tweaking the parsing monad.

A simple parsing task

- Recall (part of) grammar of simple imperative language:

$$
\begin{array}{rcl}
\mathit{Stmt} & ::= & \mathit{Var}\ \text{`}\texttt{=}\text{'}\ \mathit{Expr}\ \text{`}\texttt{;}\text{'} \\
 & | & \text{`}\{\text{'}\ \mathit{Stmts}\ \text{`}\}\text{'} \\
\mathit{Stmts} & ::= & \epsilon \\
 & | & \mathit{Stmt}\ \mathit{Stmts}
\end{array}
\qquad
\begin{array}{rcl}
\mathit{Expr} & ::= & \mathit{Number} \\
 & | & \mathit{Var} \\
 & | & \mathit{Expr}\ \mathit{Oper}\ \mathit{Expr} \\
 & | & \text{`}(\text{'}\ \mathit{Expr}\ \text{`})\text{'} \\
\mathit{Oper} & ::= & \text{`}+\text{'} \mid \text{`}*\text{'}
\end{array}
$$

- Want to parse statements and expressions into following AST:

```haskell
type Var = String
data Stmt = Assign Var Expr | Seq [Stmt]
data Expr = Num Int | Ref Var | Add Expr Expr | Mul Expr Expr
            -- note: no separate case for parenthesized Expr
```

- How would we like such a parser to look?

Parser code using combinators

```haskell
-- Imported from general parser library:
newtype Parser a = ...
instance Monad Parser where ...  -- enables do-notation
symbol :: String -> Parser ()
(<|>) :: Parser a -> Parser a -> Parser a
many :: Parser a -> Parser [a]
parseString :: Parser a -> String -> Either ParseError a

pStmt :: Parser Stmt
pStmt = do v <- pVar; symbol "="; e <- pExpr
           return $ Assign v e
        <|>
        do symbol "{"; ss <- many pStmt; symbol "}"
           return $ Seq ss

pVar :: Parser Var   -- TBD
pExpr :: Parser Expr -- TBD
```

Implementing the parser library

- Can often use existing library as “black box”.
- But very useful to be aware of underlying principles
  - Understand limitations on grammars handled.
    - E.g., left-recursion, lookahead, ...
    - Cf. LALR(1) restriction for Yacc.
  - Understand performance characteristics
    - E.g., exponential slowdown if grammar specified inappropriately
  - Can add extensions where needed
    - Utility combinators (using only exported API of library)
    - Genuine extensions (using library internals)
- In reading materials, can see two complete parser libraries (ReadP, Parsec).
- In lectures, will show bits and pieces of simplified version, combining aspects of both.

First step: what is a `Parser`?

- Need to support *sequencing*, where consecutive parser calls consume input string incrementally.
- First try: a pure state monad

```
newtype Parser a = P (runP :: String -> (a, String))
```

- E.g., `pNum :: Parser Int` will read the number at start of input string and return its value and the rest of the string (for following parsers)
- But what if parsing fails, i.e., if string does *not* start with a digit?
- Could call `error` and abort entire program.
  - Not very nice on library users.
  - Problematic for parsing nonterminals with multiple productions, because can't recover from failure and try another parse.

Parsing with failures

- Better: explicitly account for possibility of failure:

```
newtype Parser a = P (runP :: String -> Maybe (a, String))
```

- Even more general: include error information upon failure:

```
type ParseError = ...
newtype Parser a =
  P {runP :: String -> Either ParseError (a, String)}
```

- (Maybe version corresponds to taking `type ParseErrror = ()`)
- Make into a `Monad`:

```
type ParseError = String
instance Monad Parser where
  return a = P (\s -> return (a,s)) -- builds on Either monad
  m >>= f = P (\s -> do (a,s') <- runP m s; runP (f a) s')
  fail e = P (\s -> Left e)
-- also: instance Functor Parser, instance Applicative Parser
```

Some examples of simple parsers

```haskell
-- Read single char from input stream
getc :: Parser Char
getc = P (\s -> case s of
                  (c : s') -> return (c, s')
                  "" -> Left "unexpected end of input")

-- read char of specific class
-- for use with, e.g., Data.Char.isDigit :: Char -> Bool
satisfy :: (Char -> Bool) -> Parser Char
satisfy p = do c <- getc
               if p c then return c
               else fail $ "unexpected '" ++ [c] ++ "'"

-- skip expected string
string :: String -> Parser ()
string s = mapM_ (\c -> satisfy (== c)) s
 -- uses MapM_ :: Monad m => (a -> m b) -> [a] -> m ()
```

Alternatives

- How to try alternative productions for a terminal?
- Choosing between two RHSs, preferring first:

```
(<|>) :: Parser a -> Parser a -> Parser a
p1 <|> p2 =
  P (\s -> case runP p1 s of
             Right (a,s') -> Right (a,s')
             Left e1 -> case runP p2 s of
                          Right (a,s') -> Right (a,s')
                          Left e2 -> Left $ e1 ++ " or " e2)
```

- If `p1` succeeds, will *not* try `p2`.
  - In most practical grammars, at most one alternative can succeed, anyway
  - Often, infeasible alternatives fail already on first symbol.
  - **Caution:** if one alternative can be empty, will always succeed, so should try it last (unless doing *deep* backtracking, next time).
- If both `p1` and `p2` fail, try to combine error msgs, rather than only return msg from `p2` (which may be confusing).

Iteration

- Often want to parse any number of consecutive occurrences of grammar element (cf. “$\{\cdots\}$” in EBNF):

```
many :: Parser a -> parser [a]
many p = do a <- p; as <- many p; return (a:as)
          <|> return [] -- ordering matters! (for our Parser)

pNum :: Parser Int
pnum = do d <- satisfy isDigit; ds <- many (satisfy isDigit)
           return $ read (d:ds)  -- using instance Read Int
-- Ex: runP pNum "123!" == Right (123, "!")
```

- Could also define `many1` (aka `some`): *at least one* occurrence.
- **Danger:** if `p` can succeed without consuming anything, `many p` will run forever!
  - In particular, `many (many p)` will *not* work.
  - Can often fix by rephrasing the grammar.
    - Or use fancier `many p`, that exits iteration if `p` succeeded without consuming any input.

Complete parses

- In general, all parsers consume input from start of string and leave remainder.
- To parse *complete* input, should check that nothing left after start symbol has been parsed.
- Simple check when eventually getting out of `Parser` monad:

```
parseString :: String -> Either ParseError Stmt
parseString s =
  case runP pStmt s of
    Right (a, "") -> Right a
    Right (_, _) -> Left "Garbage left at end of input"
    Left e -> Left $ "Parsing failed: " ++  e
```

What next

- Thursday's lecture: more advanced features and issues:
    - Dealing with left recursion
    - Operator precedences and associativites
    - Lexing issues (esp. whitespace)
    - More on error reporting
    - Deep vs. shallow backtracking
    - ...
- Assignment 1 due tomorrow evening.
    - Remember testing!
    - Not just running examples from handout and/or OnlineTA.
- Assignment 2 (out later today) will be a parser for SubScript.
- Lab session today 11–12.
    - Mostly intended for TA help with Assignment 1, and/or Assignment 0 resubmission.
    - Some grammar exercises at end of Sestoft & Larsen parser notes.
        - But you'll need to read the notes first!