Functional and Logic Programming

Lecture 11 — Parsing

Pedro Vasconcelos
DCC/FCUP

2025

Parsing

- Convert a text into some data structure (e.g. a syntax tree)
- Essential component of many programs:
  - interpreters and compilers
  - processors for markup languages (e.g. XML, HTML, Markdown)
  - processors for domain-specific languages

Overview

- There are many techniques for parsing
- We will see how to build purely functional parsers
- Based on *combinators*, i.e. small functions that we can glue together
- We will not be concerned with:
  - efficiency
  - generality
  - producing good error messages
- But there there are parser combinator libraries that deal with these issues (e.g. the *Parsec* library in Haskell)

A type for parsers

First attempt: a function from an input string to a tree.

```
type Parser = String → Tree
```

A type for parsers

In order to combine results of parsers, we also need to return the non-consumed part of the input string.

```
type Parser = String → (Tree, String)
```

A type for parsers

We also need to consider the possiblity of a parser failing. We do this by returning a list.

```
type Parser = String → [(Tree, String)]
```

The result of parsing will be:

- a list `[(t,s)]` with a single pair if the parser succeeded; `t` is the resulting tree and `s` is the remaining input;
- the empty list `[]` if the parser failed.

A type for parsers

Finally, we generalize by allowing results of any type (not just a fixed type `Tree`).

```
type Parser a = String -> [(a, String)]
```

We can read this like a rhyme (in the style of Dr Seuss):

> *"A parser for things*
> *Is a function from strings*
> *To lists of pairs*
> *Of things and strings"*

Encapsulation

To allow overloading some operators and hide the implementation, let us declare a `newtype` instead of a type synonym.

```
newtype Parser a = P (String → [(a, String)])
```

- `P` is the constructor for parsers
- It will only be visible inside our library
- The clients of the library will build parsers using only the exported combinators

Executing a parser

To execute a parser, we simply have to extract the function and apply it to the input.

```
parse :: Parser a → String → [(a,String)]
parse (P f) txt = f txt
```

Basic combinators

A parser that always fails.

```
failure :: Parser a
failure = P (\_ → [])
```

A parser that succeeds without consuming any input.

```
return :: a → Parser a
return x = P (\s → [(x,s)])
```

A parser that gives the next character (or fails).

```
getc :: Parser Char
getc = P (\s → case s of
                 [] → []
                 (x:xs) → [(x, xs)])
```

Sequencing

An operator (pronounced "*bind*") that executes one parser and feeds the result into another.

```
(>>=) :: Parser a → (a → Parser b) → Parser b
p >>= f = P (\s → case parse p s of
                    [] → []
                    (v,s') → parse (f v) s')
```

NB: `return` and `>>=` are overloaded and thus defined as instances of the typeclass `Monad` (see the supplied library code).

Sequencing (cont.)

The bind operator `>>=` together with `return` allows combining several parsers $p_1, p_2, \ldots, p_n$ in sequence (where `f` is some function to combine the results).

```
p1 >>= \v1 ->
p2 >>= \v2 ->
⋮
pn >>= \vn ->
return (f v1 v2 ... vn)
```

Sequencing (cont.)

Alternatively, we can use do-notation and the compiler will translate it into the previous expression.

```
do v1 <- p1
   v2 <- p2
   ⋮
   vn <- pn
   return (f v1 v2 ... vn)
```

Sequencing (cont.)

We can use braces `{ }` and semicolons instead of indentation to make the structure explict:

```
do { v1 <- p1;
     v2 <- p2;
     ⋮
     vn <- pn;
     return (f v1 v2 ... vn)
   }
```

This looks like an imperative program, but is only “syntactic sugar” for the sequencing operator `>>=`.

Many languages allow operator overloading, but in Haskell you can also overload the meaning of the semicolon!

Alternative

Try the first parser; if it succeeds, return the result; otherwise try the second parser.

```
(<|>) :: Parser a → Parser a → Parser a
p <|> q = P (\s → case parse p s of
                    [] → parse q s
                    [(v,s')] → [(v,s')])
```

Derived combinators

Accept a character if it satisfies a predicate i.e. a boolean function.

```
satisfy :: (Char → Bool) → Parser Char
satisfy f = do x ← getc
               if f x then return x else failure
```

Special case: accept a given character.

```
char :: Char → Parser Char
char x = satisfy (==x)
```

Combinators for repetition

Repeat a parser zero ou more times.

```
many :: Parser a → Parser [a]
many p = many1 p <|> return []
```

Repeat a parser one or more times.

```
many1 p :: Parser a → Parser [a]
many1 p = do v ← p
             vs ← many p
             return (v:vs)
```

In both cases, we return the list of results.

Examples

Let us run some parsers in GHCi:

```
>>> parse (satisfy isDigit) "123"
[('1',"23")]

>>> parse (satisfy isDigit) "abc"
[]

>>> parse (many (satisfy isDigit)) "123abc"
[("123", "abc")]
```

More example parsers

```
-- Discard a sequence of zero or more spaces.
spaces :: Parser ()
spaces = do many (satisfy isSpace)
            return ()

-- Parse a natural number
natural :: Parser Integer
natural = do xs ← many1 (satisfy isDigit)
             return (read xs)
```

A calculator

- Let us write a basic arithmetic calculator
- Arithmetic expressions are made up of integers, `+` and `*` and parenthesis
- We will need a parser to produce a syntax tree of the expression:

```
data Expr = Num Integer
          | Add Expr Expr
          | Mult Expr Expr
```

- We can then evaluate the syntax tree using a simple recursive function:

```
eval :: Expr → Integer
eval (Num n) = n
eval (Add e1 e2) = eval e1 + eval e2
eval (Mult e1 e2) = eval e1 * eval e2
```

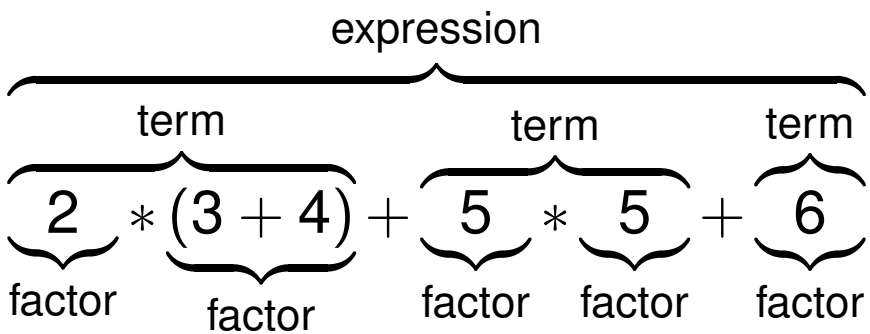
Priorities

To handle operator priorities we will separate the language into three categories:

expressions sequences of terms separated by $+$

terms sequence of factors separated by $*$

factors integers numbers or expressions between parenthesis $($ and $)$



Grammar

We can formalize this language using a context-free grammar.

$$
\begin{array}{rcl}
\text{expr} & ::= & \text{expr } \texttt{'+'} \text{ term} \;\mid\; \text{term} \\
\text{term} & ::= & \text{term } \texttt{'*'} \text{ factor} \;\mid\; \text{factor} \\
\text{factor} & ::= & \text{natural} \;\mid\; \texttt{'('} \text{ expr } \texttt{')'}
\end{array}
$$

Grammar (cont.)

- The previous grammar isn't suitable for implementing our parser because it is left recursive
- We will modify the grammar to avoid this issue
- You'll learn how to do this in general in the Compilers course

Implementation

Expressions start with a term followed by an arbitrary number of additions of terms.

$$\begin{aligned} \text{expr} &::= \text{term exprCont} \\ \text{exprCont} &::= \texttt{'+'}\ \text{term exprCont} \mid \varepsilon \end{aligned}$$

```
expr :: Parser Expr
expr = do t ← term
          exprCont t

exprCont :: Expr → Parser Expr
exprCont acc = do char '+'
                  t ← term
                  exprCont (Add acc t)
               <|> return acc
```

Implementation (cont.)

Terms start with a factor followed by an arbitrary number of multiplication of factors.

$$\begin{aligned} \text{term} &::= \text{factor termCont} \\ \text{termCont} &::= \texttt{'*'}\ \text{factor termCont} \mid \varepsilon \end{aligned}$$

```
term :: Parser Expr
term = do f ← factor
          termCont f

termCont :: Expr → Parser Expr
termCont acc =  do char '*'
                   f ← factor
                   termCont (Mul acc f)
                <|> return acc
```

Implementation (cont.)

Finally, factors are either numbers or expressions between parenthesis.

$$\text{factor} \quad ::= \quad \text{natural} \quad | \quad \texttt{'('}\ \text{expr}\ \texttt{')'}$$

```
factor :: Parser Expr
factor = do n ← natural
            return (Num n)
         <|>
         do char '('
            e ← expr
            char ')'
            return e
```

Some parsing examples

```
>>> parse expr "1+2+3"
[(Add (Add (Num 1) (Num 2)) (Num 3), "")]

>>> parse expr "1+2*3"
[(Add (Num 1) (Mul (Num 2) (Num 3)),"")]

>>> parse expr "(1+2)*3"
[(Mul (Add (Num 1) (Num 2)) (Num 3),"")]

>>> parse expr "1++2*3"
[(Num 1, "++2*3")]

>>> parse expr "*2*3"
[]
```

Putting all together

Let us now write the complete program for the calculator:

- Read the complete input
- Split it into lines
- For each line:
  1. Attempt to parse it
  2. If it succeeds, evaluate the expression and print the result
  3. Otherwise print an error message

Putting all together (cont.)

```
main :: IO ()
main = do txt ← getContents
          calculator (lines txt)

-- | read-eval-print loop
calculator :: [String] → IO ()
calculator []  = return ()
calculator (l:ls) = do putStrLn (evaluate l)
                       calculator ls

-- | calculate a single expression
evaluate :: String → String
evaluate txt
  = case parse expr txt of
      [ (tree, "") ] →  show (eval tree)
      _ → "parse error; try again"
```