Monad practice: Parsing combinators

Parsing and validation of input is a hard task that may represents over 95% of programming errors worldwide.

Let's make a short work of that problem.

Parsing preliminaries

What does a normal left-to-right recursive parser look like?

- successively translates input string into the structure (lazily!)
- remembers the “rest of the string” that still needs to be translated

Parsing combinator idea:

```
newtype Parser r = Parser (State InputToProcess r)
```


Vorsicht
Funktor
2 m Abstand halten

What do we want to get?

```
parenthesized innerParser = do char '('
                               res ← innerParser
                               char ')'
                               return res

numberOrWordOrBoth = number ⟨|⟩ parenthesized number ⟨|⟩ word ⟨|⟩ (number

parseNum :: (Num a, Read a) ⇒ Parser a
parseNum = (read ⟨$⟩ many1 parseDigit)
           ⟨|⟩ do char '-'
                  negate • read ⟨$⟩ many1 parseDigit
```

Errors and non-determinism should be handled automatically.

Alternative

Alternative contains an 'empty' element a an operation for gluing computations (similarly to how Monoid can glue values).

Semantics: If the computation can finish with multiple outcomes, Alternative generalizes specification of the other outcomes. Of some of the outcomes fail, the program typically uses the next one.

```
class Applicative f ⇒ Alternative f where
  empty :: f a
  (⟨|⟩) :: f a → f a → f a
```

Useful instances: Maybe, [ ], Validation, Except, ..., Parser *r*

Simple parser implementation

```
newtype Parser a = Parser (State String (Maybe a))

instance Functor Parser where
  fmap f (Parser s) = Parser $ fmap (fmap f) s

instance Applicative Parser where
  pure a = Parser . pure . pure $ a
  Parser l <*> Parser r = Parser $ do
    a <- l
    case a of
      Nothing -> pure Nothing
      Just f -> fmap (fmap f) r
```

Simple parser implementation

```
instance Monad Parser where
  return = pure
  Parser l >>= p =
    Parser $ do a <- l
                case a of
                  Nothing -> return Nothing
                  Just a -> let Parser r = p a in r
```

Simple parser implementation

```
instance Alternative Parser where
  empty = Parser $ pure empty
  Parser l <|> Parser r =
    Parser $ do backup <- get
                l' <- l
                case l' of
                  Nothing -> put backup >> r
                  a -> return a
```

Helpers

```
parseFail = empty
parseGet = Parser $ Just ⟨$⟩ get
parseSet s = Parser $ Just ⟨$⟩ put s

pEof = do s ← parseGet
          case s of
            "" → return ()
            _ → parseFail

doParse (Parser p) = runState p
doParseEof p = fst . doParse (p >>= λr → pEof >> return r)
```

More helpers

```
pAny = do s ← parseGet
          case s of
            (c : cs) → parseSet cs ≫ return c
            _ → parseFail

pCharCond f = pAny ≫= λc → if f c then return c
                                  else parseFail

pOneOf cs = pCharCond (∈ cs)
pAnyExcept cs = pCharCond $ ¬ • (∈ cs)
pChar c = pCharCond (≡ c)

pStr s = mapM pChar s
```

What can we do now?

```
pDigitTuple = do
  pChar '('
  x ← pCharCond isDigit
  pChar ','
  y ← pCharCond isDigit
  pChar ')'
  return [x,y]
```

```
doParse pDigitTuple "(1,2"      ⇝ State "(1,2" Nothing
doParse pDigitTuple "(1,2)xx"   ⇝ State "xx" (Just "12")
doParseEof pDigitTuple "(1,2)xx" ⇝ State "(1,2)xx" Nothing
```

```
pMany1 p = do x ← p
              xs ← pMany p
              return (x : xs)

pMany p = pMany1 p ⟨|⟩ pure [ ]

pBracketed l r p = do l
                      res ← p
                      r
                      return res

pDelim l r = pBracketed (pStr l) (pStr r)
pBrackets = pDelim "[" "]"
pBraces = pDelim "{" "}"

pQuoted q = pDelim q q • pMany $ pAnyExcept q
```

What can we do now?

pBracedDigits = *pBraces* (*pMany1* (*pCharCond isDigit*))
pBracedNumber :: Parser Int
pBracedNumber = *read* ⟨$⟩ *pBracedDigits*

doParse pBracedNumber `"{123]"` ⇝ State `"{123]"` Nothing
doParse pBracedNumber `"{123}"` ⇝ State `""` (Just 123)
doParse (*pMany pBracedNumber*) `"{123}{2345}"` ⇝ State `""` (Just [123, 2345

Sequences with delimiters

```
infixr 4 <:>
a <:> b = (:) <$> a <*> b

pSep1 sep p = p <:> (sep >> pSep1 sep p)
                <|>
                (:[]) <$> p     -- also: fmap pure p

pSep sep p = pSep1 sep p <|> pure []

pCommaDelimited = pSep (pChar ',')
```

JSON parser in 20 lines of code or so

```
data JSON = JSInt Int
          | JSStr String
          | JSList [JSON]
          | JSObj [(String, JSON)]

pJSON = pJSInt <|> pJSStr <|> pJSList <|> pJSObj

pJSInt = JSInt . (read :: String -> Int)
           <$> pMany1 (pOneOf ['0'..'9'])
pJSStr = JSStr <$> (pQuoted "\"" <|> pQuoted '\'')

pJSList = JSList <$> pBrackets (pCommaDelimited pJSON)

pJSObj = JSObj <$> pBraces (pCommaDelimited objItem)
  where objItem = do JSStr key <- pJSStr
                     pChar ':'
                     ((,) key) <$> pJSON
```

How to ignore spaces?

```
import Data.Char

spaces = pMany $ pCharCond isSpace
lexeme = (spaces≫)

lexJSInt = lexeme pJSInt
```

Parsec library

Package `parsec` is the “original” implementation that contains all this ready for use.

```
import Text.Parsec.String (Parser)
import Text.Parsec.Char
import Text.Parsec (try)
import Text.Parsec.Combinator (between, eof, many)
```

try is used for backtracking:

- in some cases it is better not to backtrack (e.g., if you want to recover from errors)
- remembering all backtrack points is hardly efficient

```
try (char 'a') <|> char 'b'
```

Available libraries

Megaparsec — newer variant of Parsec:

- better error message (unexpected vs. expected, accumulated errors)
- faster processing
- extra heap of utilities
- parsing from infinite `Streams`

Attoparsec — miniature variant

- not that much features
- extreme speed due to `ByteString` specialization
  (comparable to manually optimized parsers in C)

Packages and libraries

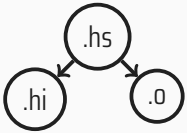
Separated compilation

Compiling programs as a whole is quite demanding:

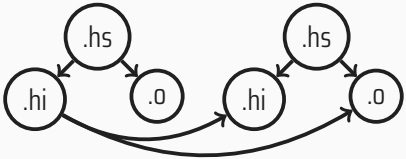
- takes time
- requires memory
- optimization suffers from complexity explosions

Typical solution: compile all files separately and link them later

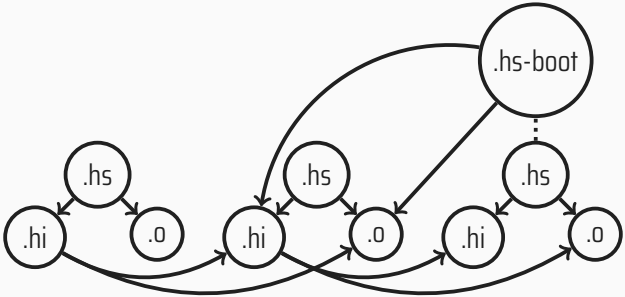
Separated compiles



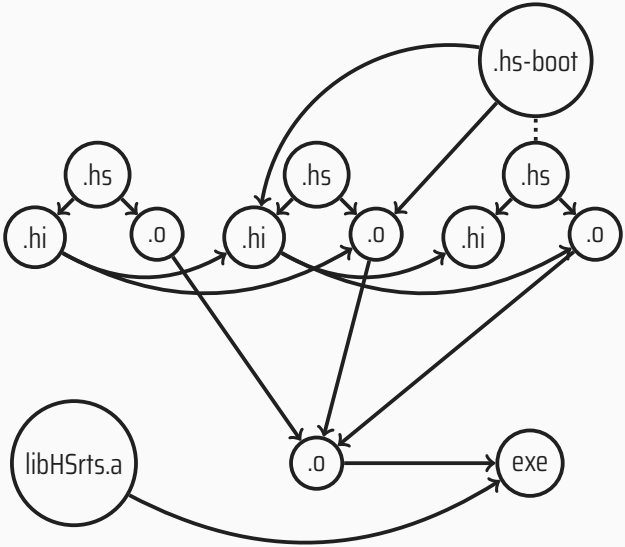
Separated compiles



Separated compiles



Separated compiles



How to import stuff?

Import from a module:

```
import Data.List
import qualified Data.List as L
import Data.List (nub)
import Data.List hiding (nub)
```

The module:

```
module Data.List where

nub = ...
```

(module name must match the module file path, in this case as `Data/List.hs`)

Packages and paths

GHC recognizes the concept of packages; these can be easily added to the list of packages available for compilation.

`ghc-pkg` tool:

- `list`, `find-module module`, `describe pkg`
- `expose pkg`, `hide pkg`
- `register filename`, `unregister pkg`, `update filename`
  `filename` is a path to the configuration file of the package.

It is possible to have several package databases (e.g., system and local user ones).

cabal

`cabal` is a system for simple creating, building, distributing and installing of packages and their dependencies.

Idea:

- Each relatively independent piece of code becomes a package!
  - gets a name
  - dependencies get described
- We use `cabal` to make the package!
  - builds
  - we get a reasonable installation and registration for `ghc`
- The package gets wrapped nicely and sent to others!
  - Hackage the haskell package database
- We can download, build and install packages from other people!
  - using `cabal`, from Hackage

boom.cabal

```
name:                boom
version:             0.1.0.0
synopsis:            explosion
license:             GPL-3
license-file:        LICENSE
author:              The Guy
maintainer:          guy@village.hr
category:            Web
build-type:          Simple
extra-source-files:  ChangeLog.md
cabal-version:       >=1.10

executable explode
  main-is:             Main.hs
```

boom.cabal

```
executable explode
  main-is:             Main.hs
  build-depends:
    base >=4.9,
    aeson,
    scotty,
    SHA,
    network,
    utf8-string

  hs-source-dirs:      src
```

Get the database of the available packages:

```
cabal update
```

Install something and register it for the currently available `ghc`:

```
cabal install acme-schoenfinkel
```

Get the database of the available packages:

```
cabal update
```

Install something and register it for the currently available `ghc`:

```
cabal install acme-schoenfinkel
cabal info acme-schoenfinkel
cabal list
```

What if we want to make a new package?

```
cd boom
cabal init
```

...you fill in an interactive form and get `boom.cabal`

Dependency installation:

```
cabal install --only-dependencies
```

Building:

```
cabal build
```

Running:

```
cabal run
```

Installation to the current environment:

```
cabal install
```

“Virtual environments” are automatic in any project directory that contains a `.cabal` file.

Nice HTML/TeX documentation:

```
cabal haddock
```

Typeset nice colorful code for publication:

```
cabal hscolour
```

Publishing the package:

`cabal check` …checks for several common issues

`cabal sdist` …creates a `.tar.gz` for manual distribution

`cabal upload` …uploads to Hackage

Publishing quality packages

```
cabal install hlint

cabal install hindent

cabal check
```

stack

`stack` is a dependency freezer for Haskell.

stack

`stack` is a dependency freezer for Haskell.

stack

Main benefits of `stack`:

- it has its own hackage (Stackage)
  - packages and dependncies are (somehow) tested for compatibility
  - it (sometimes) helps
- in case of various scary setups it can save a lot of time

Rules:

- Cabal package: 0–1 libraries, 0–N programs
  - described in `package.cabal`
- Stack project: 0–N cabal packages
  - the project and the frozen environment is described in `stack.yaml`
  - package metadata (i.e., the contents of generated packages) in `package.yaml`
  - `.cabal` files are generated automatically and cannot be changed

stack

Packages from others:

- `cd package` …the directory contains `stack.yaml`
- `stack setup`
- `stack build` …typically downloads the correct GHC etc.
- `stack exec someprogram`

Own packages:

- `stack new packagename new-template`
- `stack test` …runs tests
- `stack repl` …runs `ghci` in the frozen environment

stack

General advice: if you want to use `stack` for your next project, you're likely doing something wrong.

Why not to use `stack`:

- `cabal` has evolved into a strictly better tool, and the residual use-cases are industry abominations
- `stack` wastes resources
- `stack` splits the ecosystem
- the result is 'portable' but hardly maintainable (a bit like dockerized tools)
- `stack` installation is weird
- try `nix` instead


DO NOT STACK
S-653, ULINE, 800-295-5510

Why would you probably need to use `stack`?

- the program has 9701734347576 dependencies and uncontrollably breaks upon any change therein
- the program is a critical part of infrastructure administered by people who don't know UNIX
- the programmer does not know UNIX
- management decided
- the target operating system is massively underwhelming
- you time-warped into the old epochs before `cabal` version 1.2 got released