

Staged Selective парсер-комбинаторы

Staged Selective Parser Combinators

Косарев Дмитрий

По статье «Staged Selective Parser Combinators»
с конференции IFCP 2020

1 марта 2021



Build date: March 1, 2021

- 1 Парсер-комбинаторы (кратко)
- 2 Заслуги работы (библиотеки Parsley)
- 3 Monads и Selectives
- 4 Превращение монадического парсера в аппликативный
 - chainr1
 - chainl1
- 5 Законы
- 6 Реализация
 - Staging
- 7 Заключение

Парсер-комбинаторы. Примитивные операции

```
satisfy :: (Char -> Bool) -> Parser Char
```

```
try :: Parser a -> Parser a
```

```
lookAhead :: Parser a -> Parser a
```

```
notFollowedBy :: Parser a -> Parser ()
```

Парсер-комбинаторы. Примитивные операции

```
satisfy :: (Char -> Bool) -> Parser Char
```

```
item :: Parser Char
```

```
item = satisfy (const True)
```

```
char :: Char -> Parser Char
```

```
char c = satisfy (==c)
```

```
try :: Parser a -> Parser a
```

```
lookAhead :: Parser a -> Parser a
```

```
notFollowedBy :: Parser a -> Parser ()
```

```
eof :: Parser ()
```

```
eof = notFollowedBy item
```

```

class Functor (f :: * -> *) where
    fmap :: (a -> b) -> f a -> f b

-- application abstracted
class Functor f => Applicative f where
    -- / Lift a value.
    pure :: a -> f a
    -- / Sequential application.
    (<*>) :: f (a -> b) -> f a -> f b

-- Nondeterminism (or try-catch) abstracted
class Applicative f => Alternative f where
    -- / The identity of '<|>'
    empty :: f a
    -- / An associative binary operation
    (<|>) :: f a -> f a -> f a

```

Монад пока нет,
это нарочно

Идея парсер-комбинаторов

Использовать обычные функции, чтобы строить большие парсеры

```
char :: Char -> Parser Char
```

```
sequence :: Applicative f => [f a] -> f [a]
```

```
sequence = foldr (<:>) (pure [])
```

```
traverse :: Applicative f => (a -> f b) -> [a] -> f [b]
```

```
traverse f = sequence . map f
```

```
string :: String -> Parser String
```

```
string = traverse char
```






```
string :: String -> Parser String
```

```
oneOf = foldr (<|>) empty . map char
```










Оглавление

- 1 Парсер-комбинаторы (кратко)
- 2 Заслуги работы (библиотеки Parsley)
- 3 Monads и Selectives
- 4 Превращение монадического парсера в аппликативный
 - chainr1
 - chainl1
- 5 Законы
- 6 Реализация
 - Staging
- 7 Заключение










Parsec vs. Parsley

	Performance	Error messages	Grammar	Debugging	Analysis
Parsec					

Parsec vs. Parsley

	Performance	Error messages	Grammar	Debugging	Analysis
Parsec					
Parsley		404			

Parsec vs. Parsley

	Performance	Error messages	Grammar	Debugging	Analysis
Parsec					
Parsley		404			

Как этого удалось добиться? *Метапрограммирование!* (Typed Template Haskell)

Оглавление

- 1 Парсер-комбинаторы (кратко)
- 2 Заслуги работы (библиотеки Parsley)
- 3 Monads и Selectives**
- 4 Превращение монадического парсера в аппликативный
 - chainr1
 - chainl1
- 5 Законы
- 6 Реализация
 - Staging
- 7 Заключение

Monad

```
class Applicative m => Monad (m :: * -> *) where
  -- | Inject a value into the monadic type.
  return      :: a -> m a
  return      = pure

  -- | Sequentially compose two actions,
  -- passing any value produced
  -- by the first as an argument to the second.
  (>>=)      :: forall a b. m a -> (a -> m b) -> m b
```

```
satisfy :: (Char -> Bool) -> Parser Char
satisfy p = item >>= (\c ->
    if p c then pure c
    else      empty)
```

```
ident :: Parser String
ident = (satisfy isAlpha <:>
    many (satisfy isAlphaNum) ) >>= (\c ->
    if isKeyword c then empty
    else      pure c)
```

```
(>?>) :: Monad m => m a -> (a -> Bool) -> m a
m >?> f = m >>= \x -> if f x then pure x else empty
```

```
satisfy :: (Char -> Bool) -> Parser Char
satisfy p = item >?> p
```

```
ident :: Parser String
ident = (satisfy isAlpha <:>
        many (satisfy isAlphaNum) )
      >?> (not . isKeyword)
```

Монады классные, что же с ними не так и зачем нужны Selective
функторы?

Selective

```
-- case expression abstracted
class Applicative f => Selective f where
  -- In the original paper
  select :: f (Either a b) -> f (a -> b) -> f b

  -- alternative definition, used by Parsley
  branch :: f (Either a b) -> f (a -> c) -> f (b -> c)
         -> f c
```


Реализация `>?>`

С помощью `Monad`

`(>?>) :: Monad m => m a → (a → Bool) → m a`

`m >?> f = m >>= \ x → if f x then pure x else empty`

Реализация >?>

С помощью **Monad**

```
(>?>) :: Monad m => m a → (a → Bool) → m a  
m >?> f = m >>= \ x → if f x then pure x else empty
```

С помощью **Selective**

```
(>?>) :: Selective m => m a → (a → Bool) → m a  
fx >?> f = select ( (\ x →  
    if f x then Right ()  
    else Left () ) <$> fx ) empty
```

Чего не умеют **Selective**?

- Использовать несколько предыдущих результатов
 $mf \gg= \backslash f \rightarrow mg \gg= \backslash g \rightarrow mh \gg= \backslash h \rightarrow \dots$
- Монадический `join`:
 $\text{join} :: m (m a) \rightarrow m a$

Замечание

Если **Selective** не умеют `join` или **Applicative** не умеют КЗ-грамматики, то это не значит, что работающий парсер нельзя написать.

Оглавление

- 1 Парсер-комбинаторы (кратко)
- 2 Заслуги работы (библиотеки Parsley)
- 3 Monads и Selectives
- 4 Превращение монадического парсера в аппликативный**
 - chainr1
 - chainl1
- 5 Законы
- 6 Реализация
 - Staging
- 7 Заключение

Оглавление

- 1 Парсер-комбинаторы (кратко)
- 2 Заслуги работы (библиотеки Parsley)
- 3 Monads и Selectives
- 4 Превращение монадического парсера в аппликативный**
 - **chainr1**
 - chainl1
- 5 Законы
- 6 Реализация
 - Staging
- 7 Заключение

```
chainr1 :: Parser a -> Parser (a -> a -> a) -> Parser a
chainr1 p op = scan
  where
    scan      = do { x <- p; rest x }
    rest x    = do { f <- op
                    ; y <- scan
                    ; return (f x y)
                    }
              <|> return x
```

```
chainr1 p op = p >>= rest  
  where
```

```
    rest x = (op >>= \f ->  
              chainr1 p op >>= \y ->  
                pure (f x y))  
    <|> pure x
```

```

chainr1 :: Parser a -> Parser (a -> a -> a) -> Parser a
chainr1 p op = p <*> rest
  where
    rest :: Parser (a -> a)
    -- N.B. arguments must go away
    rest x = (op >>= \f ->
              chainr1 p op >>= \y ->
                pure (f x y))
              <|> pure x

-- (<*>) :: Applicative f => f a -> f (a -> b) -> f b

```



```
chainr1 :: Parser a -> Parser (a -> a -> a) -> Parser a
chainr1 p op = p <*> rest
  where
```

```
    rest      = (op >>= \f ->
                  chainr1 p op >>= \y ->
                  pure (flip f y))
              <|> pure id
```

-- Уже компилируется, но надо избавиться от двух >>=
-- и это очень просто

```
chainr1 :: Parser a -> Parser (a -> a -> a) -> Parser a
chainr1 p op = p <*> rest
  where
```

```
    rest = (flip <$> op <*> chainr1 p op)
          <|> pure id
```

-- C chainr1 всё

-- C chainl1 будет посложнее

Оглавление

- 1 Парсер-комбинаторы (кратко)
- 2 Заслуги работы (библиотеки Parsley)
- 3 Monads и Selectives
- 4 **Превращение монадического парсера в аппликативный**
 - chainr1
 - **chainl1**
- 5 Законы
- 6 Реализация
 - Staging
- 7 Заключение

```
chainl1 :: Parser a -> Parser (a -> a -> a) -> Parser a
chainl1 p op = do { x <- p; rest x }
  where
    rest x    = do { f <- op
                     ; y <- p
                     ; rest (f x y)
                     }
               <|> return x
```

```
chainl1 :: Parser a -> Parser (a -> a -> a) -> Parser a
chainl1 p op = p >>= rest
  where
```

```
    rest x = op >>= \f ->
              p  >>= \y ->
                rest (f x y)
              <|> return x
```

```

chainl1 :: Parser a -> Parser (a -> a -> a) -> Parser a
chainl1 p op = p <*> rest
  where
    -- N.B. Аргумент нужно убирать
    rest x = op >>= \f ->
      p >>= \y ->
        rest (f x y)
      <|> return id

-- рекурсивный вызов мешает
-- Интуиция: реализуем foldl через foldr

```

```
chainl1 :: Parser a -> Parser (a -> a -> a) -> Parser a
chainl1 p op = p <*> rest
  where
```

```
rest    = (op >>= \f ->
           p  >>= \y ->
           (\ g x -> g (f x y)) <$> rest)
        <|> pure id
```

-- Теперь превратим $(\backslash g x \rightarrow g (f x y))$ в парсер,
-- чтобы вытолкнуть *rest* наружу

```

chainl1 :: Parser a -> Parser (a -> a -> a) -> Parser a
chainl1 p op = p <*> rest
  where

    rest    = (op >>= \f ->
                p  >>= \y ->
                pure (\ g x -> g (f x y)) ) <*> rest
              <|> pure id

-- (<*>) :: Applicative f => f (a -> b) -> f a -> f b

```



```
chainl1 :: Parser a -> Parser (a -> a -> a) -> Parser a
chainl1 p op = p <*> rest
  where
```

```
rest    = (op >>= \f ->
           p  >>= \y ->
             -- pure (\ g x -> g (f x y))
             -- pure (\ g x -> g (flip f y x))
             -- pure (\ g -> g . flip f y))
           pure (flip (.) (flip f y))
           ) <*> rest
  <|> pure id
```

```
chainl1 :: Parser a -> Parser (a -> a -> a) -> Parser a
chainl1 p op = p <*> rest
  where

    rest    = (op >>= \f ->
                p  >>= \y ->
                flip (.) <$> pure (flip f y))
              <*> rest
    <|> pure id
```

```
chainl1 :: Parser a -> Parser (a -> a -> a) -> Parser a
chainl1 p op = p <*> rest
  where
```

```
rest    = flip (.) <$>
          (op >>= \f ->
            p  >>= \y ->
              pure (flip f y))
          ) <*> rest
  <|> pure id
```

```
chainl1 :: Parser a -> Parser (a -> a -> a) -> Parser a
chainl1 p op = p <*> rest
  where
    rest :: Parser (a -> a)
    rest =
      flip (.) <$> (flip <$> op <*> chainl1 p op) <*> rest
      <|> pure id

-- Конец
```

Оглавление

- 1 Парсер-комбинаторы (кратко)
- 2 Заслуги работы (библиотеки Parsley)
- 3 Monads и Selectives
- 4 Превращение монадического парсера в аппликативный
 - chainr1
 - chainl1
- 5 Законы**
- 6 Реализация
 - Staging
- 7 Заключение

Законы Applicative

```
-- application abstracted
class Functor f => Applicative f where
  -- / Lift a value.
  pure  :: a -> f a
  -- / Sequential application.
  (<*>) :: f (a -> b) -> f a -> f b
```

```
pure id <*> p = p -- (1)
pure f <*> pure x = pure (f x) -- (2)
u <*> pure x = pure (λf -> f x) <*> u -- (3)
u <*> (v <*> w) = pure (.) <*> u <*> v <*> w -- (4)
```

Законы Selective

```
-- case expression abstracted
class Applicative f => Selective f where
  branch :: f (Either a b) -> f (a -> c) -> f (b -> c) -> f c

branch (pure (Left  x)) p q = p <*> pure x           -- (9)
branch (pure (Right y)) p q = q <*> pure y           -- (10)
branch b (pure f) (pure g)  = pure (either f g) <*> b -- (11)
    branch (x *> y) p q = x *> branch y p q          -- (12)
        branch b p empty = branch (pure swap <*> b) empty p -- (13)

branch (branch b empty (pure f)) empty k =
  branch (pure g <*> b) empty k  -- (14)
where
  g = either (const (Left ())) (either (const (Left ())) Right. f)
```

Законы Alternative

```
-- Nondeterminism (or try-catch) abstracted
class Applicative f => Alternative f where
  -- / The identity of '<|>'
  empty :: f a
  -- / An associative binary operation
  (<|>) :: f a -> f a -> f a
```

$(p \text{ <|> } q) \text{ <|> } r = p \text{ <|> } (q \text{ <|> } r) \quad \text{-- (5)}$

$\text{empty} \text{ <|> } p = p \text{ <|> } \text{empty} = p \quad \text{-- (6)}$

$\text{empty} \text{ <*> } p = \text{empty} \quad \text{-- (7)}$

$\text{pure } x \text{ <|> } p = \text{pure } x \quad \text{-- (8)}$

Законы парсеров

```
try (satisfy f) = satisfy f -- (15)
```

```
try (negLook p) = negLook p -- (16)
```

```
look empty      = empty      -- (17)
```

```
look (pure x)    = pure x     -- (18)
```

```
negLook empty    = pure ()    -- (19)
```

```
negLook (pure x) = empty      -- (20)
```

```
look (look p)      = look p      -- (21)
```

```
look p <|> look q   = look (try p <|> q) -- (22)
```

```
negLook (negLook p) = look p      -- (23)
```

```
look (negLook p)    = negLook (look p) = negLook p -- (24)
```

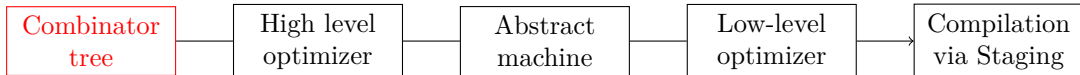
```
-- de Morgan law where *> is conjunction
```

```
negLook (try p <|> q) = negLook p *> negLook q -- (25)
```

```
negLook p <|> negLook q = negLook (look p *> look q) -- (26)
```

Оглавление

- 1 Парсер-комбинаторы (кратко)
- 2 Заслуги работы (библиотеки Parsley)
- 3 Monads и Selectives
- 4 Превращение монадического парсера в аппликативный
 - chainr1
 - chainl1
- 5 Законы
- 6 Реализация
 - Staging
- 7 Заключение



```
newtype Fix (syn :: (* -> *) -> (* -> *)) (a :: *) where  
  In :: syn (Fix syn) a -> Fix syn a
```

```
newtype Parser a = Parser (Fix ParserF a)  
data ParserF (k :: * -> *) (a :: *) where  
  Pure :: a -> ParserF k a  
  Satisfy :: (Char -> Bool) -> ParserF k Char  
  Try :: k a -> ParserF k a  
  Look :: k a -> ParserF k a  
  NegLook :: k () -> ParserF k ()  
  Empty :: ParserF k a  
  Branch :: k (Either x y) -> k (x -> a) -> k (y -> a) -> ParserF k a  
  (:<*>:) :: k (a -> b) -> k a -> ParserF k b  
  (:*>:) :: k a -> k b -> ParserF k b  
  (:<*:): :: k a -> k b -> ParserF k a  
  (:<|>:) :: k a -> k a -> ParserF k a
```

High level optimizer. Пример 1

```
string :: String -> Parser String
string = traverse char

string "ab"
-- unrolling

pure (:) <*> char 'a' <*> (pure (:) <*> char 'b' <*> pure [])
-- Applicative fusion optimizations ....

satisfy (== 'a') *> satisfy (== 'b') *> pure "ab"
```

Законы аппликативов

```
pure id <*> p = p -- (1)
pure f <*> pure x = pure (f x) -- (2)
u <*> pure x = pure (λf -> f x) <*> u -- (3)
u <*> (v <*> w) = pure (.) <*> u <*> v <*> w -- (4)
```

High level optimizer. Пример 2

$$(f \text{ <\$> } p) \text{ <*> } (g \text{ <\$> } q)$$

$$(\backslash x \ y \text{ -> } (f \ x) \ (g \ y)) \text{ <\$> } p \text{ <*> } q$$

Сэкономили одну операцию

High level optimizer. Пример 3

```
ident :: String → Maybe String  
ident = some (oneOf ['a' .. 'z']) `filteredBy` (not . isKeyword)
```

Превращается в более эффективный

```
ident input =  
  let loop (c : cs) dxs finish | isAlpha c = loop cs (dxs . (c:)) finish  
      loop cs dxs finish = finish (dxs []) cs  
  in case input of  
    c : cs | isAlpha c → loop cs id (λ xs _ → if isKeyword (c:xs)  
                                                then Nothing  
                                                else Just (c:xs))  
    _ → Nothing
```

High level optimizer. Борьба с рекурсией – вставка let

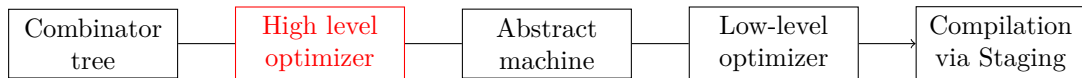
Исходный вариант:

```
many :: Parser a -> Parser a
many p = (p <:> many p) <|> (pure p)
```

Оптимизированный вариант:

```
many :: Parser a -> Parser [a]
many p =
  let go = (p <:> go) <|> (pure p)
  go
```

High level optimizer. Больше анализов



- Consumption («поглощения») + cut («отсечения»)
 - Consumption – сколько символов «поедает» парсер
 - Cut нужны для более адекватных сообщениях об ошибках
 - Вставка правильных "отсечений" может быть нетривиальна
- Termination
 - Анализ точный для КС-грамматик и полностью аппликативных парсеров
 - В остальных случаях анализ неточный и ложные срабатывания игнорируются

Абстрактная машина

- CPS
- Нужен стек, так как нет монад, но хочется КЗ

```
data M (k :: [*] -> * -> * -> *) (xs :: [*]) (r :: *) (a :: *) where
  Halt :: M k [a] Void a
  Push :: x -> k (x : xs) r a -> M k xs r a
  Pop  :: k xs r a -> M k (x : xs) r a
  ...
```

- `k` – дерево из комбинаторов (обычно `Fix M`);
- `xs` – type-level список, хранит типы значений на стеке перед исполнением инструкции;
- `r` – это тип возвращаемого машиной значения (полезно для рекурсии);
- `a` – тип финального результата парсинга, соответствует типу исходного парсера на комбинаторах.

Компиляция (1/6). Операции со стеком

```
compile :: Fix ParserF a -> Fix M [] Void a  
compile = cata compAlg halt
```

```
type CodeGen a x = forall xs r . Fix M (x : xs) r a -> Fix M xs r a
```

```
compAlg :: ParserF (CodeGen a) x -> Fix M (x : xs) r a -> Fix M xs r a  
compAlg (Pure x) = push x  
compAlg (p :*>: q) = p . pop . q  
compAlg (p :<*: q) = p . q . pop
```

Компиляция (2/6). Аппликативы

```
data M (k :: [*] -> * -> * -> *) (xs :: [*]) (r :: *) (a :: *) where
    ...
    Lift2 :: (x -> y -> z) -> k (z : xs) r a -> M k (y : x : xs) r a
    Swap  :: k (x : y : xs) r a -> M k (y : x : xs) r a

app = lift2 id
compAlg :: ParserF (CodeGen a) x -> Fix M (x : xs) r a -> Fix M xs r a
compAlg (pf :<*>: px) = pf . px . app
```

Компиляция (3/6). Selectives

```
data M (k :: [*] -> * -> * -> *) (xs :: [*]) (r :: *) (a :: *) where
    ...
    Case :: k (x : xs) r a -> k (y : xs) r a -> M k (Either x y : xs) r a

compAlg :: ParserF (CodeGen a) x -> Fix M (x : xs) r a -> Fix M xs r a
compAlg (Branch b l r) = λk ->
    b (case (l (swap (app k))) (r (swap (app k))))
```

- Запускаем `b`
- В продолжении проверяем с помощью `case` на `Left/Right`
- В зависимости от результата запускаем `l` или `r` вместе с `k`

Компиляция (4/6). Alternatives

Функции `catch`, `handle`, `commit` для работы со стеком, где лежат *handlers*...

```
data M (k :: [*] -> * -> * -> *) (xs :: [*]) (r :: *) (a :: *) where
```

```
...
```

```
Fail :: M k xs r a
```

```
Catch :: k xs r a -> k (String : xs) r a -> M k xs r a
```

```
Commit :: k xs r a -> M k xs r a
```

```
handle :: (Fix M xs r a -> Fix M (x : xs) r a) -> Fix M (String : xs) r a  
        -> Fix M (x : xs) r a -> Fix M xs r a
```

```
handle p h k = catch (p (commit k)) h
```

```
compAlg :: ParserF (CodeGen a) x -> Fix M (x : xs) r a -> Fix M xs r a
```

```
compAlg (p :<|>: q) = λk -> handle p (parsecHandle (q k)) k
```

```
compAlg Empty = const fail
```

```
...
```

Компиляция (5/6). Примитивные инструкции

```
data M k (xs :: [*]) r a where
  ...
  Sat :: (Char -> Bool) -> k (Char : xs) r a -> M k xs r a
  Tell :: k (String : xs) r a -> M k xs r a
  Seek :: k xs r a -> M (String : xs) r a
  Ret :: M k [r] r a
  Call :: MuVar x -> k (x : xs) r a -> M k xs r a

compAlg (Satisfy p) = sat p
compAlg (Try p) = handle p (seek fail)
compAlg (Look p) = tell . p . swap . seek
compAlg (Let _  $\mu$ ) = call  $\mu$ 
```

- `tell` кладет вход на стек
- `seek` возвращает в исходное состояние

Компиляция (6/6). Negative lookahead

```
negLook p = try (look p *> empty) <|> pure ()
```

Компиляция (6/6). Negative lookahead

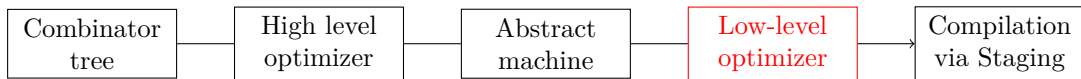
```
negLook p = try (look p *> empty) <|> pure ()
```

А вот так правильно

```
negLookM p = join (try (look p *> pure empty) <|> pure (pure ()))
```

Реализация в машине

```
compAlg (NegLook p) = λk ->  
  handle (tell . p . pop . seek) (seek (push () k)) fail
```

- Join points (φ -узлы)

`compAlg` (p :<|>: q) = $\lambda k \rightarrow$ handle p (parsecHandle (q k)) k

Решается с помощью специальной инструкции машины **Join**, которая вставляется после **Case** и **Catch**

- Хвостовая рекурсия

- Достигается введением инструкций **Jump** и **Call**

- Deep inspection (via histomorphism)

- Поиск различных шаблонов и их замена на более компактные операции

Consumption анализ, попытка 2

Оценка снизу сколько символов "поедает" парсер:

```
inputConsumed :: Fix M -> Int
inputConsumed = cata alg where
  alg :: M Int -> Int
  alg Halt = 0
  alg (Push _ k) = k
  alg (Sat _ k) = k + 1
  alg (Catch p q) = min p q
  alg (Call _ _) = 0
  alg (MkJoin  $\varphi$  b k) = b + k
  alg (Join  $\varphi$ ) = 0
  ...
```

-- нессимистично

N.B. Типы упрощены для краткости

Оглавление

- 1 Парсер-комбинаторы (кратко)
- 2 Заслуги работы (библиотеки Parsley)
- 3 Monads и Selectives
- 4 Превращение монадического парсера в аппликативный
 - chainr1
 - chainl1
- 5 Законы
- 6 Реализация
 - Staging
- 7 Заключение

Staging

Обычная функция возведения в степень

```
power :: Nat -> (Int -> Int)
power 0 = λx -> 1
power n = λx -> x * power (n - 1) x
```

Staged функция, где степень известна *статически*, а основание – динамически

```
power_ :: Nat -> Code (Int -> Int)
power_ 0 = [| λx -> 1 |]
power_ n = [| λx -> x * $(power_ (n-1)) |]

power5 = $(power_ 5)
        = $([| λx -> x * x * x * x * x * 1 |])
        = λx -> x * x * x * x * x * 1
```

Quoting vs. splicing

Если $x :: a$ то
 $[|x|] :: \text{Code } a$

и если $qx :: \text{Code } a$, то
 $\$(qx) :: a$.

Staging для парсер-комбинаторов (пример)

```
--  
  
nonzero :: Parser Char  
nonzero :: oneOf ['1'..'9']  
digit :: Parser Char  
digit = char '0' <|> nonzero  
natural :: Parser Int  
natural =  
    read <$> (nonzero <:> many digit)  
  
ident :: Parser String  
ident = satisfy (    isAlpha)  
    <:> many (satisfy (    alphaNum))
```

Staging для парсер-комбинаторов (пример)

```
--  
{-# OPTIONS_GHC -fplugin=LiftPlugin #-}  
nonzero :: Parser Char  
nonzero :: oneOf ['1'..'9']  
digit :: Parser Char  
digit = char '0' <|> nonzero  
natural :: Parser Int  
natural =  
    code read <$> (nonzero <:> many digit)  
  
ident :: Parser String  
ident = satisfy (code isAlpha)  
    <:> many (satisfy (code alphaNum))
```

Компиляция в абстрактную машину

```
type Eval xs r a =  $\Gamma$  xs r a -> Maybe a
eval :: Fix M [ ] Void a -> (String -> Maybe a)
eval m =  $\lambda$  input -> cata alg m ( $\Gamma$  input HNil [] (error "Empty call stack"))
  where
    alg :: M Eval xs r a -> Eval xs r a
    alg Halt = evalHalt
    alg (Push x k) = evalPush x k
    alg ...

data  $\Gamma$  xs r a =  $\Gamma$ 
  { input :: String, ops :: HList xs
    , hs :: [String -> Maybe a], retCont :: r -> String -> Maybe a }

data HList (xs :: [*]) where
  HNil :: HList []
  HCons :: x -> HList xs -> HList (x : xs)
```

Реализация машины (1/5)

```
type Eval' xs r a = Code (Γ xs r a -> Maybe a)
eval' :: Fix M xs Void a -> Code (String -> Maybe a)
data Γ' xs r a = Γ'
  { input :: Code String , ops :: Code (HList xs)
  , hs :: Code [String -> Maybe a] , retCont :: Code (r -> String -> Maybe a)

type Eval'' xs r a = Γ' xs r a -> Code (Maybe a)
data Γ'' xs r a = Γ''
  { input :: Code String, ops :: QList xs
  , hs :: [Code (String -> Maybe a) ]
  , retCont :: Code (r -> String -> Maybe a) }
data QList (xs :: [*]) where
  QNil :: QList [ ]
  QCons :: Code x -> QList xs -> QList (x : xs)
```


Реализация машины (2/5)

```
eval :: Fix M [ ] Void a -> (String -> Maybe a)
eval m = λ input ->
  cata alg m (Γ input HNil [] (error "Empty call stack"))
  where alg :: M Eval xs r a -> Eval xs r a
        alg Halt = evalHalt
        alg (Push x k) = evalPush x k
        alg ...

evalHalt :: (Γ [a] Void a -> Maybe a)
evalHalt = λγ -> let HCons x = ops γ in Just x

eval''' m = [| λ input ->
  $(cata alg''' m (Γ''' [| input |] QNil [] [| noret |])) |]
  where ...
  evalHalt''' γ = let QCons qx = ops γ in [| Just $(qx) |]
```

Реализация машины (3/5)

```
evalLift2 :: (x -> y -> z) -> Eval (z : xs) r a
          -> (Γ (y : x : xs) r a -> Maybe a)
evalLift2 f k = λγ -> let HCons y (HCons x xs) = ops γ in
                      k (γ { ops = HCons (f x y) xs })

evalLift2''' qf k γ =
  let QCons qy (QCons qx xs) = ops γ in
  k (γ { ops = QCons [| ($(qf) $(qx) $(qy)) |] xs })
```

Реализация машины (4/5)

```
evalFail :: (Γ xs r a -> Maybe a)
evalFail = λγ -> case hs γ of
    h : [] -> h (input γ)
    _      -> Nothing
```

В случае ошибки пытаемся восстановиться первым попавшимся способом

```
evalFail''' γ = case hs γ of
    qh : [] -> [| $(qh) $(input γ) |]
    _      -> [| Nothing |]
```

Реализация машины (5/5)

```
evalSat :: (Char -> Bool) -> Eval (Char : xs) r a -> ( $\Gamma$  xs r a -> Maybe a)
evalSat f k =  $\lambda\gamma \rightarrow$  case input  $\gamma$  of
  c : cs | f c -> k ( $\gamma$  { input = cs, ops = HCons c (ops  $\gamma$ ) })
  _       -> evalFail  $\gamma$ 
```

Функция evalSat – *полностью* динамическая, в отличие от предыдущих

```
evalSat''' qf k  $\gamma$  = [| case $(input  $\gamma$ ) of
  c : cs | $(qf) c -> $(k ( $\gamma$  { input = [|cs|], ops = QCons [|c|] (ops  $\gamma$ ) })))
  _       -> $(evalEmpt  $\gamma$ ) |]
```

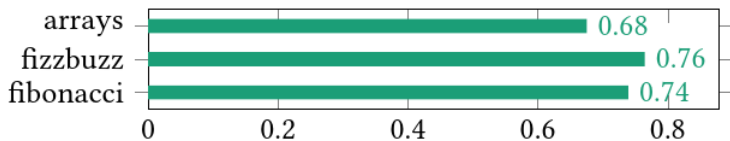


Fig. 6. Performance of bison parsing Nandlang, time relative to Parsley (as ByteString)

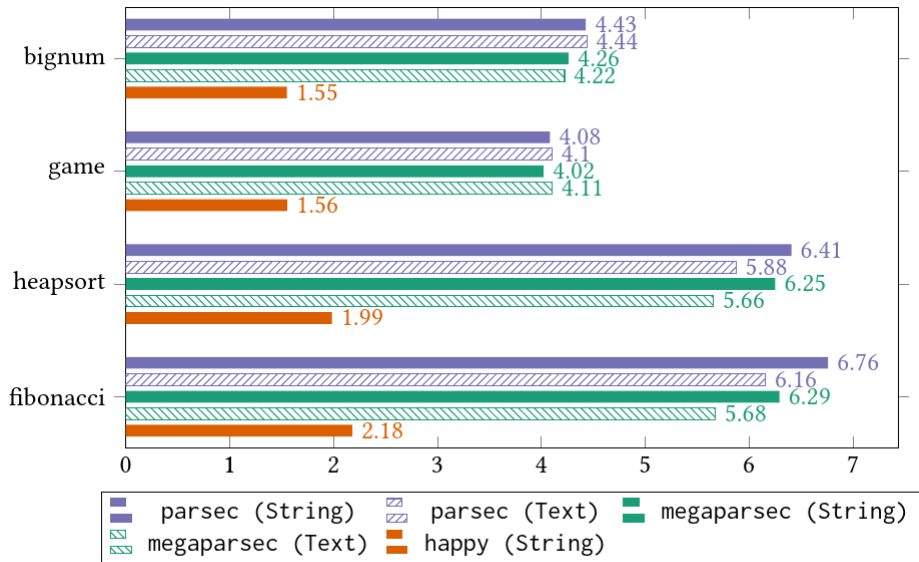


Fig. 5. Performance of libraries parsing JavaScript, time relative to Parsley

Оглавление

- 1 Парсер-комбинаторы (кратко)
- 2 Заслуги работы (библиотеки Parsley)
- 3 Monads и Selectives
- 4 Превращение монадического парсера в аппликативный
 - chainr1
 - chainl1
- 5 Законы
- 6 Реализация
 - Staging
- 7 Заключение

Достижения:

- Оптимизированная библиотека персер-комбинаторов с отличной производительностью

Задачи на будущее:

- Обработка ошибок
- Обход недостатков из-за отсутствия монад:
 - Доступ к предыдущим результатам парсинга (например, через "регистры" [2])



Staged Selective Parser Combinators

Jamie Willis & Nicolas Wu & Matthew Pickering

<https://doi.org/10.1145/3409002>



Garnishing Parsec With Parsley: A Staged Selective Parser Combinator Library

Jamie Willis

<https://www.youtube.com/watch?v=tJcyY9L2z84>



Selective Applicative Functors

Andrey Mokhov & Georgy Lukyanov & Simon Marlow & Jeremie Dimino

<https://doi.org/10.1145/3341694>



Библиотека FastParse для Scala

Documentation



Try vs. lookahead

<https://stackoverflow.com/questions/20020350>