MOD08: Functional Programming

Parser Combinators & ParSec

Marco Gerards

08-05-2019

Organisation of the course

Block	Topic
1	- Introduction to Functional Programming
	- Higher order functions
2	- Types and type classes
	- Parsing (application)
3	- Parser combinators and ParSec (application)
	- Advanced type classes
4	- Code generation (application)
5-7	- Project

Organisation of the course

Block	Topic
1	- Introduction to Functional Programming
	- Higher order functions
2	- Types and type classes
	- Parsing (application)
3	- Parser combinators and ParSec (application)
	- Advanced type classes
4	- Code generation (application)
5-7	- Project

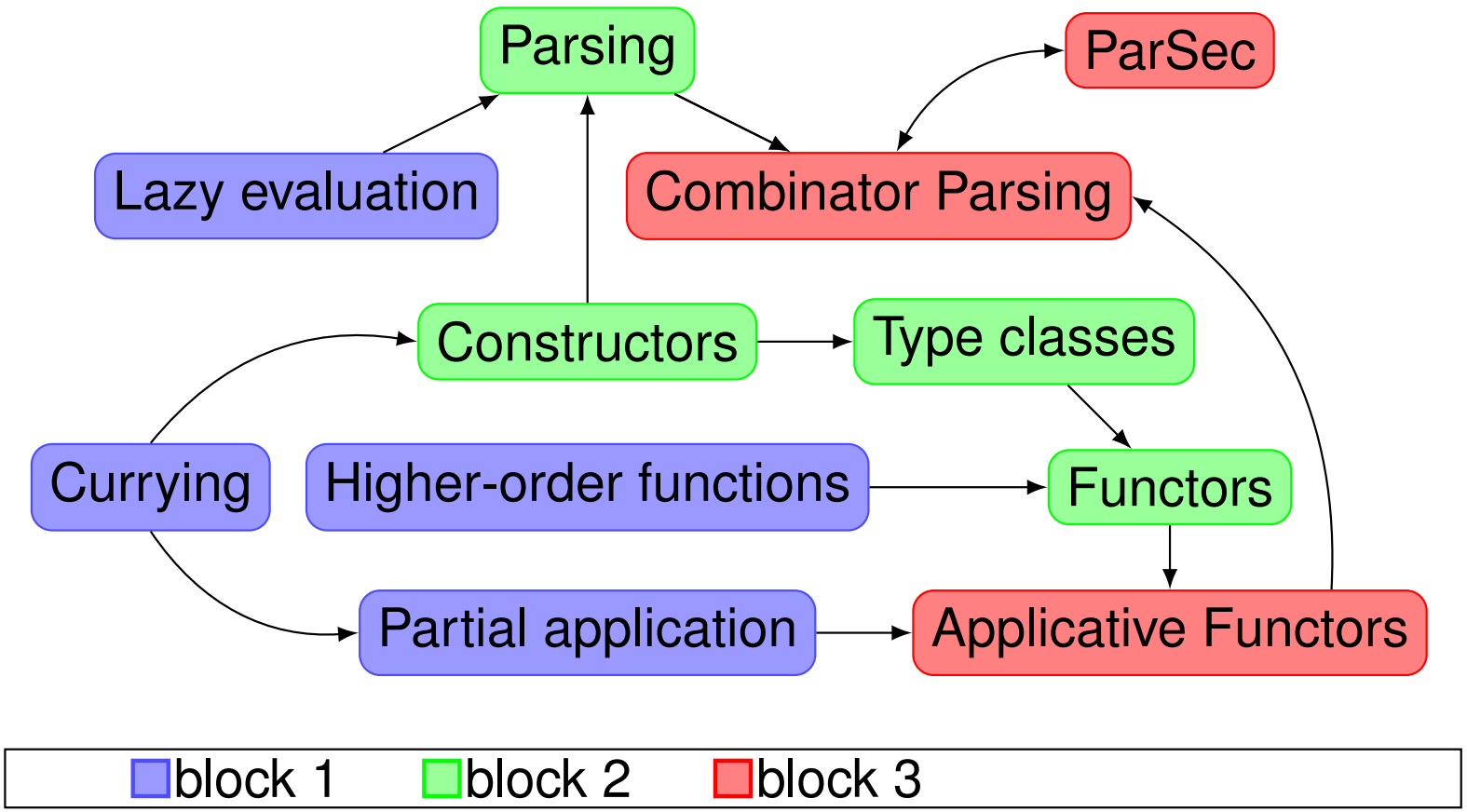
This lecture: learning goals

"Solve non-trivial programming problems in Functional Programming"

Application: Code generation / EDSLs (used in last lab session)

Application: Parser Combinators + ParSec (used in next lab)

Connection of some of the topics between blocks



Domain Specific Languages (DSLs)

- General purpose languages

  C, Java, Haskell, ...

Domain Specific Languages (DSLs)

- General purpose languages

  C, Java, Haskell, ...

- Domain Specific Languages

Domain Specific Languages (DSLs)

- General purpose languages
  C, Java, Haskell, ...
- Domain Specific Languages
  - Type setting / web
    HTML, LaTeX, SVG

Domain Specific Languages (DSLs)

- General purpose languages
  C, Java, Haskell, ...
- Domain Specific Languages
  - Type setting / web
    HTML, LaTeX, SVG
  - Hardware description
    VHDL, Verilog, Lava, Clash

Domain Specific Languages (DSLs)

- General purpose languages
  C, Java, Haskell, ...
- Domain Specific Languages
  - Type setting / web
    HTML, LaTeX, SVG
  - Hardware description
    VHDL, Verilog, Lava, Clash
  - Compiler construction
    Antlr, YACC, ParSec

Embedded Domain Specific Languages (EDSLs)

- Embedded DSL: uses features of the host language

Embedded Domain Specific Languages (EDSLs)

- Embedded DSL: uses features of the host language
- Shallow embedding
  - Language described directly in the host language
  - Directly “compiled”/used/applied
  - Examples: QuickCheck, ParSec

Embedded Domain Specific Languages (EDSLs)

- Embedded DSL: uses features of the host language
- Shallow embedding
  - Language described directly in the host language
  - Directly “compiled”/used/applied
  - Examples: QuickCheck, ParSec
- Deep Embedding
  - A data structure represents the EDSL

Embedded Domain Specific Languages (EDSLs)

- Embedded DSL: uses features of the host language
- Shallow embedding
  - Language described directly in the host language
  - Directly “compiled”/used/applied
  - Examples: QuickCheck, ParSec
- Deep Embedding
  - A data structure represents the EDSL
  - Example, an expression language:

```
data Expr = Plus   Expr Expr
          | Min    Expr Expr
          | Times  Expr Expr
          | C      Int
```

Embedded Domain Specific Languages (EDSLs)

- Embedded DSL: uses features of the host language
- Shallow embedding
  - Language described directly in the host language
  - Directly “compiled”/used/applied
  - Examples: QuickCheck, ParSec
- Deep Embedding
  - A data structure represents the EDSL
  - Example, an expression language:

```
data Expr = Plus  Expr Expr
          | Min   Expr Expr
          | Times Expr Expr
          | C     Int
```

  - Easy to: evaluate / transform / translate

```
calc :: Expr -> Int
calc (Plus  x y) = calc x + calc y
calc (Min   x y) = calc x - calc y
calc (Times x y) = calc x * calc y
calc (C     x)   = x
```

EDSL Example: small stack processor (1/3)

```
data Op = OpAdd | OpMul | OpSub
        deriving Show

data Instr = InsPush Int
           | InsCalc Op
           deriving Show
```

EDSL Example: small stack processor (1/3)

```
data Op = OpAdd | OpMul | OpSub
        deriving Show

data Instr = InsPush Int
           | InsCalc Op
           deriving Show

-- (3+1) * 4
prog = [InsPush 4,
        InsPush 1,
        InsPush 3,
        InsCalc OpAdd,
        InsCalc OpMul ]
```

EDSL Example: small stack processor (2/3)

```
-- (3+1) * 4
prog = [InsPush 4,
        InsPush 1,
        InsPush 3,
        InsCalc OpAdd,
        InsCalc OpMul ]
```

EDSL Example: small stack processor (2/3)

```
-- (3+1) * 4
prog = [InsPush 4,
        InsPush 1,
        InsPush 3,
        InsCalc OpAdd,
        InsCalc OpMul ]

g OpAdd = (+)
g OpMul = (*)
g OpSub = (-)

f stack (InsPush n)  = n : stack
f stack (InsCalc op) = v : drop 2 stack
  where v = g op (stack!!1) (stack!!0)

execute prog = scanl f [] prog
```

EDSL Example: small stack processor (3/3)

```
codegen :: Expr -> [Instr]
codegen (C      x)   = [InsPush x]
codegen (Plus   x y) = codegen x ++ codegen y
                       ++ [InsCalc OpAdd]
...
codegen (Times x y) = codegen x ++ codegen y
                      ++ [InsCalc OpMul]
```

EDSL Example: small stack processor (3/3)

```
codegen :: Expr -> [Instr]
codegen (C      x)   = [InsPush x]
codegen (Plus   x y) = codegen x ++ codegen y
                       ++ [InsCalc OpAdd]

...
codegen (Times x y) = codegen x ++ codegen y
                      ++ [InsCalc OpMul]
```

```
*Main> let y = ((C 3) `Plus` (C 2)) `Times` (C 4)
```

EDSL Example: small stack processor (3/3)

```
codegen :: Expr -> [Instr]
codegen (C      x)   = [InsPush x]
codegen (Plus   x y) = codegen x ++ codegen y
                       ++ [InsCalc OpAdd]

...
codegen (Times x y) = codegen x ++ codegen y
                      ++ [InsCalc OpMul]
```

```
*Main> let y = ((C 3) `Plus` (C 2)) `Times` (C 4)
*Main> calc y
```

EDSL Example: small stack processor (3/3)

```
codegen :: Expr -> [Instr]
codegen (C      x)   = [InsPush x]
codegen (Plus   x y) = codegen x ++ codegen y
                       ++ [InsCalc OpAdd]

...
codegen (Times x y) = codegen x ++ codegen y
                      ++ [InsCalc OpMul]
```

```
*Main> let y = ((C 3) `Plus` (C 2)) `Times` (C 4)
*Main> calc y
20
```

EDSL Example: small stack processor (3/3)

```
codegen :: Expr -> [Instr]
codegen (C      x)   = [InsPush x]
codegen (Plus   x y) = codegen x ++ codegen y
                       ++ [InsCalc OpAdd]

...
codegen (Times x y) = codegen x ++ codegen y
                       ++ [InsCalc OpMul]
```

```
*Main> let y = ((C 3) `Plus` (C 2)) `Times` (C 4)
*Main> calc y
20
*Main> let code = codegen y
```

EDSL Example: small stack processor (3/3)

```
codegen :: Expr -> [Instr]
codegen (C      x)   = [InsPush x]
codegen (Plus   x y) = codegen x ++ codegen y
                       ++ [InsCalc OpAdd]

...
codegen (Times x y) = codegen x ++ codegen y
                      ++ [InsCalc OpMul]
```

```
*Main> let y = ((C 3) `Plus` (C 2)) `Times` (C 4)
*Main> calc y
20
*Main> let code = codegen y
*Main> code
```

EDSL Example: small stack processor (3/3)

```
codegen :: Expr -> [Instr]
codegen (C      x)   = [InsPush x]
codegen (Plus   x y) = codegen x ++ codegen y
                       ++ [InsCalc OpAdd]

...
codegen (Times x y) = codegen x ++ codegen y
                      ++ [InsCalc OpMul]
```

```
*Main> let y = ((C 3) `Plus` (C 2)) `Times` (C 4)
*Main> calc y
20
*Main> let code = codegen y
*Main> code
[InsPush 3,InsPush 2,InsCalc OpAdd,InsPush
4,InsCalc OpMul]
```

EDSL Example: small stack processor (3/3)

```
codegen :: Expr -> [Instr]
codegen (C     x)   = [InsPush x]
codegen (Plus  x y) = codegen x ++ codegen y
                      ++ [InsCalc OpAdd]

...
codegen (Times x y) = codegen x ++ codegen y
                      ++ [InsCalc OpMul]
```

```
*Main> let y = ((C 3) `Plus` (C 2)) `Times` (C 4)
*Main> calc y
20
*Main> let code = codegen y
*Main> code
[InsPush 3,InsPush 2,InsCalc OpAdd,InsPush
4,InsCalc OpMul]
*Main> execute code
```

EDSL Example: small stack processor (3/3)

```
codegen :: Expr -> [Instr]
codegen (C      x)   = [InsPush x]
codegen (Plus   x y) = codegen x ++ codegen y
                       ++ [InsCalc OpAdd]

...
codegen (Times x y) = codegen x ++ codegen y
                      ++ [InsCalc OpMul]
```

```
*Main> let y = ((C 3) `Plus` (C 2)) `Times` (C 4)
*Main> calc y
20
*Main> let code = codegen y
*Main> code
[InsPush 3,InsPush 2,InsCalc OpAdd,InsPush
4,InsCalc OpMul]
*Main> execute code
[[],[3],[2,3],[5],[4,5],[20]]
```

Parsing in Haskell

```
parseC :: String -> (ParseTree, String)
```

Parsing in Haskell

```
parseC :: String -> (ParseTree, String)

parseT :: [Token] -> (ParseTree, [Token])
```

This lecture: what problem will we solve?

- Downside of our recursive descent parsers; consider:

```
parseR :: [Char] -> (ParseTree, [Char])
parseR ('b':xs) = (Node R [Leaf 'b', t1, t2], zs)
 where (t1, ys) = parseQ xs -- 'b' Q
       (t2, zs) = parseQ ys -- 'b' Q Q
```

This lecture: what problem will we solve?

- Downside of our recursive descent parsers; consider:

```
parseR :: [Char] -> (ParseTree, [Char])
parseR ('b':xs) = (Node R [Leaf 'b', t1, t2], zs)
 where (t1, ys) = parseQ xs -- 'b' Q
       (t2, zs) = parseQ ys -- 'b' Q Q
```

- How would you parse the following?

```
<S> ::= <Q> <Q> <Q> <Q> <Q> <Q>
<Q> ::= 'a' 'b' | 'c'
```

This lecture: what problem will we solve?

- Downside of our recursive descent parsers; consider:

```
parseR :: [Char] -> (ParseTree, [Char])
parseR ('b':xs) = (Node R [Leaf 'b', t1, t2], zs)
 where (t1, ys) = parseQ xs -- 'b' Q
       (t2, zs) = parseQ ys -- 'b' Q Q
```

- How would you parse the following?

```
<S> ::= <Q> <Q> <Q> <Q> <Q> <Q>
<Q> ::= 'a' 'b' | 'c'
```

- Labor intensive and error prone

This lecture: what problem will we solve?

- Downside of our recursive descent parsers; consider:

```
parseR :: [Char] -> (ParseTree, [Char])
parseR ('b':xs) = (Node R [Leaf 'b', t1, t2], zs)
 where (t1, ys) = parseQ xs -- 'b' Q
       (t2, zs) = parseQ ys -- 'b' Q Q
```

- How would you parse the following?

```
<S> ::= <Q> <Q> <Q> <Q> <Q> <Q>
<Q> ::= 'a' 'b' | 'c'
```

- Labor intensive and error prone
- Parser combinators: EDSL (shallow embedding) to avoid this

EDSL Example: grammar and EDSL

```
<S> ::= <Q> | <R>
<Q> ::= 'a' 'b'
<R> ::= 'b' <Q> <Q>
      | 'c' <Q> 'c'
```

EDSL Example: grammar and EDSL

```
<S> ::= <Q> | <R>
<Q> ::= 'a' 'b'
<R> ::= 'b' <Q> <Q>
      | 'c' <Q> 'c'
```

```
data S = S1 Q | S2 R
  deriving Show

data Q = Q Char Char
  deriving Show

data R = R1 Char Q Q | R2 Char Q Char
  deriving Show
```

Parser combinators: sneak preview of ParSec

```
<S> ::= <Q> | <R>
<Q> ::= 'a' 'b'
<R> ::= 'b' <Q> <Q>
      | 'c' <Q> 'c'
```

Parser combinators: sneak preview of ParSec

```
<S> ::= <Q> | <R>
<Q> ::= 'a' 'b'
<R> ::= 'b' <Q> <Q>
      | 'c' <Q> 'c'
```

```
parseS :: Parser S
parseS = S1 <$> parseQ <|> S2 <$> parseR

parseQ :: Parser S
parseQ = Q <$> char 'a' <*> char 'b'

parseR :: Parser R
parseR =     R1 <$> char 'b' <*> parseQ <*> parseQ
         <|> R2 <$> char 'c' <*> parseQ <*> char 'c'
```

Parser combinators: elementary parsers

- Typical parser type (several variants exist!)

```
data Parser r = P {
  runParser :: String -> [(r, String)]
}
```

Parser combinators: elementary parsers

- Typical parser type (several variants exist!)

```
data Parser r = P {
  runParser :: String -> [(r, String)]
}
```

- Failure ⇒ `[]`

Parser combinators: elementary parsers

- Typical parser type (several variants exist!)

```
data Parser r = P {
  runParser :: String -> [(r, String)]
}
```

- Failure ⇒ `[]`
- Parser for a single character

```
char :: Char -> Parser Char
char c  = P p
 where p (x:xs) | c == x    = [(x, xs)]
                | otherwise = []
```

Parser combinators: elementary parsers

- Typical parser type (several variants exist!)

```
data Parser r = P {
  runParser :: String -> [(r, String)]
}
```

- Failure ⇒ []
- Parser for a single character

```
char :: Char -> Parser Char
char c  = P p
 where p (x:xs) | c == x    = [(x, xs)]
                | otherwise = []
```

- Example:

```
*Main> runParser (char 'a') "abc"
[('a',"bc")]
*Main> runParser (char 'a') "cba"
[]
```

Parsers are Functors (1/2)

- `Parser` is of kind `* -> *`

```
data Parser r = P {
  runParser :: String -> [(r, String)]
}
```

Parsers are Functors (1/2)

- `Parser` is of kind `* -> *`

```
data Parser r = P {
  runParser :: String -> [(r, String)]
}
```

- Motivation
  - `char 'x':: Parser Char` the parse result is a `Char`
  - Can we transform to type `Parser a` for any `a`?

Parsers are Functors (1/2)

- `Parser` is of kind `* -> *`

```
data Parser r = P {
  runParser :: String -> [(r, String)]
}
```

- Motivation
  - `char 'x':: Parser Char` the parse result is a `Char`
  - Can we transform to type `Parser a` for any `a`?
- Functor instance:

```
instance Functor Parser where
  fmap f p
    = P (\x -> [ (f r, s) | (r,s) <- runParser p x ])
```

Parsers are Functors (2/2)

Parsers are Functors (2/2)

```
instance Functor Parser where
  fmap f p = P (\x -> [(f r, s) | (r,s) <- runParser p x])
```

```
*Main> :t char 'a'
```

Parsers are Functors (2/2)

```
instance Functor Parser where
  fmap f p = P (\x -> [(f r, s) | (r,s) <- runParser p x])
```

```
*Main> :t char 'a'
char 'a' ::  Parser Char
```

Parsers are Functors (2/2)

```
instance Functor Parser where
  fmap f p = P (\x -> [(f r, s) | (r,s) <- runParser p x])
```

```
*Main> :t char 'a'
char 'a' ::  Parser Char
*Main> :t ord
```

Parsers are Functors (2/2)

```
instance Functor Parser where
  fmap f p = P (\x -> [(f r, s) | (r,s) <- runParser p x])
```

```
*Main> :t char 'a'
char 'a' ::  Parser Char
*Main> :t ord
ord ::  Char -> Int
```

Parsers are Functors (2/2)

```
instance Functor Parser where
  fmap f p = P (\x -> [(f r, s) | (r,s) <- runParser p x])
```

```
*Main> :t char 'a'
char 'a' ::  Parser Char
*Main> :t ord
ord ::  Char -> Int
*Main> :t ord <$> char 'a'
```

Parsers are Functors (2/2)

```
instance Functor Parser where
  fmap f p = P (\x -> [(f r, s) | (r,s) <- runParser p x])
```

```
*Main> :t char 'a'
char 'a' ::  Parser Char
*Main> :t ord
ord ::  Char -> Int
*Main> :t ord <$> char 'a'
ord <$> char 'a' ::  Parser Int
```

Parsers are Functors (2/2)

```
instance Functor Parser where
  fmap f p = P (\x -> [(f r, s) | (r,s) <- runParser p x])
```

```
*Main> :t char 'a'
char 'a' ::  Parser Char
*Main> :t ord
ord ::  Char -> Int
*Main> :t ord <$> char 'a'
ord <$> char 'a' ::  Parser Int
*Main> runParser (char 'a') "abc"
```

Parsers are Functors (2/2)

```
instance Functor Parser where
  fmap f p = P (\x -> [(f r, s) | (r,s) <- runParser p x])
```

```
*Main> :t char 'a'
char 'a' ::  Parser Char
*Main> :t ord
ord ::  Char -> Int
*Main> :t ord <$> char 'a'
ord <$> char 'a' ::  Parser Int
*Main> runParser (char 'a') "abc"
[('a',"bc")]
```

Parsers are Functors (2/2)

```
instance Functor Parser where
  fmap f p = P (\x -> [(f r, s) | (r,s) <- runParser p x])
```

```
*Main> :t char 'a'
char 'a' ::  Parser Char
*Main> :t ord
ord ::  Char -> Int
*Main> :t ord <$> char 'a'
ord <$> char 'a' ::  Parser Int
*Main> runParser (char 'a') "abc"
[('a',"bc")]
*Main> runParser (ord <$> char 'a') "abc"
```

Parsers are Functors (2/2)

```
instance Functor Parser where
  fmap f p = P (\x -> [(f r, s) | (r,s) <- runParser p x])
```

```
*Main> :t char 'a'
char 'a' ::  Parser Char
*Main> :t ord
ord ::  Char -> Int
*Main> :t ord <$> char 'a'
ord <$> char 'a' ::  Parser Int
*Main> runParser (char 'a') "abc"
[('a',"bc")]
*Main> runParser (ord <$> char 'a') "abc"
[(97,"bc")]
```

Parsers are Functors (2/2)

```
instance Functor Parser where
  fmap f p = P (\x -> [(f r, s) | (r,s) <- runParser p x])
```

```
*Main> :t char 'a'
char 'a' ::  Parser Char
*Main> :t ord
ord ::  Char -> Int
*Main> :t ord <$> char 'a'
ord <$> char 'a' ::  Parser Int
*Main> runParser (char 'a') "abc"
[('a',"bc")]
*Main> runParser (ord <$> char 'a') "abc"
[(97,"bc")]
*Main> runParser (Leaf <$> char 'a') "abc"
```

Parsers are Functors (2/2)

```
instance Functor Parser where
  fmap f p = P (\x -> [(f r, s) | (r,s) <- runParser p x])
```

```
*Main> :t char 'a'
char 'a' ::  Parser Char
*Main> :t ord
ord ::  Char -> Int
*Main> :t ord <$> char 'a'
ord <$> char 'a' ::  Parser Int
*Main> runParser (char 'a') "abc"
[('a',"bc")]
*Main> runParser (ord <$> char 'a') "abc"
[(97,"bc")]
*Main> runParser (Leaf <$> char 'a') "abc"
[(Leaf 'a',"bc")]]
```

Parsers are Functors (2/2)

```
instance Functor Parser where
  fmap f p = P (\x -> [(f r, s) | (r,s) <- runParser p x])
```

```
*Main> :t char 'a'
char 'a' ::  Parser Char
*Main> :t ord
ord ::  Char -> Int
*Main> :t ord <$> char 'a'
ord <$> char 'a' ::  Parser Int
*Main> runParser (char 'a') "abc"
[('a',"bc")]
*Main> runParser (ord <$> char 'a') "abc"
[(97,"bc")]
*Main> runParser (Leaf <$> char 'a') "abc"
[(Leaf 'a',"bc")]]
```

Question 1

```haskell
data Parser r = P {
  runParser :: String -> [(r, String)]
}
```

Consider the following function:

```haskell
f :: Parser a -> Parser a -> ???
f p1 p2 = P (\x -> runParser p1 x ++ runParser p2 x)
```

What is the type of `f`? (i.e., determine ???)

A This code is incorrect

B `f :: Parser a -> Parser a -> [Parser a]`

C `f :: Parser a -> Parser a -> Parser [a]`

D `f :: Parser a -> Parser a -> Parser a`

Question 2

```haskell
data Parser r = P {
  runParser :: String -> [(r, String)]
}
char :: Char -> Parser Char
```

Consider the following function:

```haskell
f :: Parser a -> Parser a -> ???
f p1 p2 = P (\x -> runParser p1 x ++ runParser p2 x)
```

What is the result of the following expression?

```haskell
runParser (f (char 'a') (char 'b')) "abc"
```

A `[("ab", "c")]`
B `[('a', "bc")]`
C `[('b', "bc")]`
D `[('c', "")]`

Parser combinator: alternative

- Alternative combinator

```
(<|>) :: Parser a -> Parser a -> Parser a
p1 <|> p2 = P (\x -> runParser p1 x ++ runParser p2 x)
```

Parser combinator: alternative

- Alternative combinator

```
(<|>) :: Parser a -> Parser a -> Parser a
p1 <|> p2 = P (\x -> runParser p1 x ++ runParser p2 x)
```

- Different implementation:

```
(<|>) :: Parser a -> Parser a -> Parser a
(P p1) <|> (P p2) = P p
  where p x = p1 x ++ p2 x
```

Parser combinator: alternative

- Alternative combinator

```
(<|>) :: Parser a -> Parser a -> Parser a
p1 <|> p2 = P (\x -> runParser p1 x ++ runParser p2 x)
```

- Different implementation:

```
(<|>) :: Parser a -> Parser a -> Parser a
(P p1) <|> (P p2) = P p
  where p x = p1 x ++ p2 x
```

- Interpretation:
  - `p <|> q` uses both `p` and `q`
  - Considers *all* alternative ways to parse the input

```
*Main> runParser (char 'a' <|> char 'a') "aa"
```

Parser combinator: alternative

- Alternative combinator

```
(<|>) :: Parser a -> Parser a -> Parser a
p1 <|> p2 = P (\x -> runParser p1 x ++ runParser p2 x)
```

- Different implementation:

```
(<|>) :: Parser a -> Parser a -> Parser a
(P p1) <|> (P p2) = P p
  where p x = p1 x ++ p2 x
```

- Interpretation:
  - `p <|> q` uses both `p` and `q`
  - Considers *all* alternative ways to parse the input

```
*Main> runParser (char 'a' <|> char 'a') "aa"
[('a',"a"),('a',"a")]
```

Application of the alternative combinator <|>

```
<S> ::= <Q> | <R>
<Q> ::= 'a' 'b'
<R> ::= 'b' <Q> <Q>
      | 'c' <Q> 'c'
```

Application of the alternative combinator <|>

```
<S> ::= <Q> | <R>
<Q> ::= 'a' 'b'
<R> ::= 'b' <Q> <Q>
      | 'c' <Q> 'c'
```

- Assume we have:

```
data S = S1 Q | S2 R
           deriving Show
parseQ :: Parser Q
parseR :: Parser R
```

Application of the alternative combinator <|>

```
<S> ::= <Q> | <R>
<Q> ::= 'a' 'b'
<R> ::= 'b' <Q> <Q>
      | 'c' <Q> 'c'
```

- Assume we have:

```
data S = S1 Q | S2 R
           deriving Show
parseQ :: Parser Q
parseR :: Parser R
```

- Make a value of type S:
  - `S1 <$> parseQ :: Parser S`
  - `S2 <$> parseR :: Parser S`

Application of the alternative combinator <|>

```
<S> ::= <Q> | <R>
<Q> ::= 'a' 'b'
<R> ::= 'b' <Q> <Q>
      | 'c' <Q> 'c'
```

- Assume we have:

```
data S = S1 Q | S2 R
           deriving Show
parseQ :: Parser Q
parseR :: Parser R
```

- Make a value of type S:
  - `S1 <$> parseQ :: Parser S`
  - `S2 <$> parseR :: Parser S`
- Create a parser for S:

```
(S1 <$> parseQ) <|> (S2 <$> parseR) :: Parser S
```

Question 3

```haskell
data Parser r = P {
  runParser :: String -> [(r, String)]
}
```

Consider a function `f` of type:

```haskell
f :: Parser (a -> b) -> Parser a -> Parser b
```

What is the type of the following expression?

```haskell
f ((,) <$> (char 'a')) (char 'b')
```

A `Parser Char`

B `Parser [Char]`

C `Parser (Char, Char)`

D `Parser (Char -> Char)`

Answer 3

```haskell
data Parser r = P {
  runParser :: String -> [(r, String)]
}
```

Consider a function `f` of type:

```haskell
f :: Parser (a -> b) -> Parser a -> Parser b
```

What is the type of the following expression?

```haskell
f ((,) <$> (char 'a')) (char 'b')
```

Answer 3

```haskell
data Parser r = P {
  runParser :: String -> [(r, String)]
}
```

Consider a function `f` of type:

```haskell
f :: Parser (a -> b) -> Parser a -> Parser b
```

What is the type of the following expression?

```haskell
f ((,) <$> (char 'a')) (char 'b')
```

- `(,) :: a -> b -> (a,b)`

Answer 3

```haskell
data Parser r = P {
  runParser :: String -> [(r, String)]
}
```

Consider a function `f` of type:

```haskell
f :: Parser (a -> b) -> Parser a -> Parser b
```

What is the type of the following expression?

```haskell
f ((,) <$> (char 'a')) (char 'b')
```

- `(,) :: a -> b -> (a,b)`
- `(char 'a') :: Parser Char`

Answer 3

```haskell
data Parser r = P {
  runParser :: String -> [(r, String)]
}
```

Consider a function `f` of type:

```haskell
f :: Parser (a -> b) -> Parser a -> Parser b
```

What is the type of the following expression?

```haskell
f ((,) <$> (char 'a')) (char 'b')
```

- `(,) :: a -> b -> (a,b)`
- `(char 'a') :: Parser Char`
- `((,) <$> (char 'a')) :: Parser (b -> (Char, b))`

Answer 3

```haskell
data Parser r = P {
  runParser :: String -> [(r, String)]
}
```

Consider a function `f` of type:

```haskell
f :: Parser (a -> b) -> Parser a -> Parser b
```

What is the type of the following expression?

```haskell
f ((,) <$> (char 'a')) (char 'b')
```

- `(,) :: a -> b -> (a,b)`
- `(char 'a') :: Parser Char`
- `((,) <$> (char 'a')) :: Parser (b -> (Char, b))`

```haskell
f ((,) <$> (char 'a')) (char 'b') :: Parser (Char,Char)
```

Parser combinator: sequence

- Sequence combinator

```
(<*>) :: Parser (a -> b) -> Parser a -> Parser b
p1 <*> p2 = P (\s -> [ (r1 r2, s2)
                     | (r1, s1) <- runParser p1 s,
                       (r2, s2) <- runParser p2 s1 ])
```

Parser combinator: sequence

- Sequence combinator

```
(<*>) :: Parser (a -> b) -> Parser a -> Parser b
p1 <*> p2 = P (\s -> [ (r1 r2, s2)
                     | (r1, s1) <- runParser p1 s,
                       (r2, s2) <- runParser p2 s1 ])
```

- Alternative implementation:

```
(<*>) :: Parser (a -> b) -> Parser a -> Parser b
(P p1) <*> (P p2) = P p
  where p s = [ (r1 r2, s2) | (r1, s1) <- p1 s,
                              (r2, s2) <- p2 s1 ]
```

Sequence combinator and currying

```
(<*>) :: Parser (a -> b) -> Parser a -> Parser b
p1 <*> p2 = P (\s -> [ (r1 r2, s2)
                     | (r1, s1) <- runParser p1 s,
                       (r2, s2) <- runParser p2 s1 ])
```

- `p, q :: Parser Char`

Sequence combinator and currying

```
(<*>) :: Parser (a -> b) -> Parser a -> Parser b
p1 <*> p2 = P (\s -> [ (r1 r2, s2)
                     | (r1, s1) <- runParser p1 s,
                       (r2, s2) <- runParser p2 s1 ])
```

- `p, q :: Parser Char`
- `(,) <$> p :: ???`

Sequence combinator and currying

```
(<*>) :: Parser (a -> b) -> Parser a -> Parser b
p1 <*> p2 = P (\s -> [ (r1 r2, s2)
                     | (r1, s1) <- runParser p1 s,
                       (r2, s2) <- runParser p2 s1 ])
```

- `p, q :: Parser Char`
- `(,) <$> p :: ???`
  `(,) <$> p :: Parser (b -> (Char, b))`

Sequence combinator and currying

```
(<*>) :: Parser (a -> b) -> Parser a -> Parser b
p1 <*> p2 = P (\s -> [ (r1 r2, s2)
                     | (r1, s1) <- runParser p1 s,
                       (r2, s2) <- runParser p2 s1 ])
```

- `p, q :: Parser Char`
- `(,) <$> p :: ???`
  `(,) <$> p :: Parser (b -> (Char, b))`
- `((,) <$> p) <*> q :: ???`

Sequence combinator and currying

```
(<*>) :: Parser (a -> b) -> Parser a -> Parser b
p1 <*> p2 = P (\s -> [ (r1 r2, s2)
                     | (r1, s1) <- runParser p1 s,
                       (r2, s2) <- runParser p2 s1 ])
```

- `p, q :: Parser Char`
- `(,) <$> p :: ???`
  `(,) <$> p :: Parser (b -> (Char, b))`
- `((,) <$> p) <*> q :: ???`
  `((,) <$> p) <*> q :: Parser (Char, Char)`

Sequence combinator and currying

```
(<*>) :: Parser (a -> b) -> Parser a -> Parser b
p1 <*> p2 = P (\s -> [ (r1 r2, s2)
                     | (r1, s1) <- runParser p1 s,
                       (r2, s2) <- runParser p2 s1 ])
```

- `p, q :: Parser Char`
- `(,) <$> p :: ???`
  `(,) <$> p :: Parser (b -> (Char, b))`
- `((,) <$> p) <*> q :: ???`
  `((,) <$> p) <*> q :: Parser (Char, Char)`
- General structure: use currying

```
parseA :: Parser a
...
f :: (a -> b -> c -> d -> e)
...
g :: Parser e
g =(((f <$> parseA) <*> parseB) <*> parseC) <*> parseD
```

Sequence combinator: Examples

```
parser1 = ((,) <$> (char 'a')) <*> (char 'b')
```

Sequence combinator: Examples

```
parser1 = ((,) <$> (char 'a')) <*> (char 'b')
```

```
*Main> runParser parser1 "ab"
```

Sequence combinator: Examples

```
parser1 = ((,) <$> (char 'a')) <*> (char 'b')
```

```
*Main> runParser parser1 "ab"
[(('a','b'),"")]
```

Sequence combinator: Examples

```
parser1 = ((,) <$> (char 'a')) <*> (char 'b')
```

```
*Main> runParser parser1 "ab"
[(('a','b'),"")]
*Main> runParser parser1 "abc"
```

Sequence combinator: Examples

```
parser1 = ((,) <$> (char 'a')) <*> (char 'b')
```

```
*Main> runParser parser1 "ab"
[(('a','b'),"")]
*Main> runParser parser1 "abc"
[(('a','b'),"c")]
```

Sequence combinator: Examples

```
parser1 = ((,) <$> (char 'a')) <*> (char 'b')
```

```
*Main> runParser parser1 "ab"
[(('a','b'),"")]
*Main> runParser parser1 "abc"
[(('a','b'),"c")]
*Main> runParser parser1 "aa"
```

Sequence combinator: Examples

```
parser1 = ((,) <$> (char 'a')) <*> (char 'b')
```

```
*Main> runParser parser1 "ab"
[(('a','b'),"")]
*Main> runParser parser1 "abc"
[(('a','b'),"c")]
*Main> runParser parser1 "aa"
[]
```

Sequence combinator: Examples

```
parser1 = ((,) <$> (char 'a')) <*> (char 'b')
```

```
*Main> runParser parser1 "ab"
[(('a','b'),"")]
*Main> runParser parser1 "abc"
[(('a','b'),"c")]
*Main> runParser parser1 "aa"
[]
```

```
parser2 = ((+) <$> (ord<$>char '1')) <*> (ord<$>char '2')
```

Sequence combinator: Examples

```
parser1 = ((,) <$> (char 'a')) <*> (char 'b')
```

```
*Main> runParser parser1 "ab"
[(('a','b'),"")]
*Main> runParser parser1 "abc"
[(('a','b'),"c")]
*Main> runParser parser1 "aa"
[]
```

```
parser2 = ((+) <$> (ord<$>char '1')) <*> (ord<$>char '2')
```

```
*Main> runParser parser2 "11"
```

Sequence combinator: Examples

```
parser1 = ((,) <$> (char 'a')) <*> (char 'b')
```

```
*Main> runParser parser1 "ab"
[(('a','b'),"")]
*Main> runParser parser1 "abc"
[(('a','b'),"c")]
*Main> runParser parser1 "aa"
[]
```

```
parser2 = ((+) <$> (ord<$>char '1')) <*> (ord<$>char '2')
```

```
*Main> runParser parser2 "11"
[]
```

Sequence combinator: Examples

```
parser1 = ((,) <$> (char 'a')) <*> (char 'b')
```

```
*Main> runParser parser1 "ab"
[(('a','b'),"")]
*Main> runParser parser1 "abc"
[(('a','b'),"c")]
*Main> runParser parser1 "aa"
[]
```

```
parser2 = ((+) <$> (ord<$>char '1')) <*> (ord<$>char '2')
```

```
*Main> runParser parser2 "11"
[]
*Main> runParser parser2 "12"
```

Sequence combinator: Examples

```
parser1 = ((,) <$> (char 'a')) <*> (char 'b')
```

```
*Main> runParser parser1 "ab"
[(('a','b'),"")]
*Main> runParser parser1 "abc"
[(('a','b'),"c")]
*Main> runParser parser1 "aa"
[]
```

```
parser2 = ((+) <$> (ord<$>char '1')) <*> (ord<$>char '2')
```

```
*Main> runParser parser2 "11"
[]
*Main> runParser parser2 "12"
[(99,"")]
```

ParSec

ParSec

```
module Lecture5 where

import Text.ParserCombinators.Parsec

-- for emptyDef:
import Text.ParserCombinators.Parsec.Language

import qualified Text.Parsec.Token as Token
```

Tokenizing (1/3)

Define a record that describes the tokens in your language:

```
languageDef =
    emptyDef { Token.commentStart    = "/*"
             , Token.commentEnd      = "*/"
             , Token.commentLine     = "//"
             , Token.identStart      = letter
             , Token.identLetter     = alphaNum
             , Token.reservedNames   = [ "for"
                                       , "while"
                                       , "if"
                                       ]
             , Token.reservedOpNames = [ "-", "+", "*",
                                       ]
             }
```

Tokenizing (2/3)

Create tokenizer functions for these tokens:

```
-- Create lexer (=tokenizer) for your language
lexer = Token.makeTokenParser languageDef

-- Create functions for all types of tokens
identifier = Token.identifier lexer
integer = Token.integer lexer
parens = Token.parens lexer
symbol = Token.symbol lexer
reserved = Token.reserved lexer
-- ... etc
-- Read the Text.Parsec.Token documentation!
```

Tokenizing (3/3)

- Parser type: `ParsecT s u m a`
  - Parsec types:

    ```
    type Parsec s u = ParsecT s u Identity
    ```

  - Use provided type synonym `Parser` instead!
  - `char 'X' :: Parser Char`
  - Do not use the character parser for the exercises
    - It does not remove surrounding whitespace!

Tokenizing (3/3)

- Parser type: `ParsecT s u m a`
  - Parsec types:

    ```
    type Parsec s u = ParsecT s u Identity
    ```

  - Use provided type synonym `Parser` instead!
  - `char 'X' :: Parser Char`
  - Do not use the character parser for the exercises
    - It does not remove surrounding whitespace!

Tokenizing (3/3)

- Parser type: `ParsecT s u m a`
  - Parsec types:

    ```
    type Parsec s u = ParsecT s u Identity
    ```

  - Use provided type synonym `Parser` instead!
  - `char 'X' :: Parser Char`
  - Do not use the character parser for the exercises
    - It does not remove surrounding whitespace!
- Parse an identifier

  ```
  identifier :: Parser String
  ```

Tokenizing (3/3)

- Parser type: `ParsecT s u m a`
  - Parsec types:

    ```
    type Parsec s u = ParsecT s u Identity
    ```

  - Use provided type synonym `Parser` instead!
  - `char 'X' :: Parser Char`
  - Do not use the character parser for the exercises
    - It does not remove surrounding whitespace!
- Parse an identifier

  ```
  identifier :: Parser String
  ```

- Parse a reserved symbol (e.g., `if`))

  ```
  reserved :: String -> Parser ()
  ```

Tokenizing (3/3)

- Parser type: `ParsecT s u m a`
  - Parsec types:

    ```
    type Parsec s u = ParsecT s u Identity
    ```
  - Use provided type synonym `Parser` instead!
  - `char 'X' :: Parser Char`
  - Do not use the character parser for the exercises
    - It does not remove surrounding whitespace!
- Parse an identifier

  ```
  identifier :: Parser String
  ```
- Parse a reserved symbol (e.g., `if`))

  ```
  reserved :: String -> Parser ()
  ```
- Parse some symbol (e.g., +):

  ```
  symbol :: String -> Parser String
  ```

ParSec example: <identifier> = <identifier>

ParSec example: <identifier> = <identifier>

- ParSec code for <identifier> = <identifier>

```
assignment :: Parser (String, String, String)
assignment = (,,) <$> identifier <*>
                      symbol "=" <*> identifier
```

ParSec example: <identifier> = <identifier>

- ParSec code for <identifier> = <identifier>

```
assignment :: Parser (String, String, String)
assignment = (,,) <$> identifier <*>
                      symbol "=" <*> identifier
```

- In GHCi:

```
*ParsecEx> parse assignment "" "abc = def"
Right ("abc","=","def")
*ParsecEx> parse assignment "" "abc == def"
Left (line 1, column 6):
unexpected "="
```

ParSec example: <identifier> = <identifier>

- ParSec code for <identifier> = <identifier>

```
assignment :: Parser (String, String, String)
assignment = (,,) <$> identifier <*>
                      symbol "=" <*> identifier
```

- In GHCi:

```
*ParsecEx> parse assignment "" "abc = def"
Right ("abc","=","def")
*ParsecEx> parse assignment "" "abc == def"
Left (line 1, column 6):
unexpected "="
```

- Downside: "=" is stored, but perhaps not needed

ParSec example: <identifier> = <identifier>

- ParSec code for <identifier> = <identifier>

```
assignment :: Parser (String, String, String)
assignment = (,,) <$> identifier <*>
                      symbol "=" <*> identifier
```

- In GHCi:

```
*ParsecEx> parse assignment "" "abc = def"
Right ("abc","=","def")
*ParsecEx> parse assignment "" "abc == def"
Left (line 1, column 6):
unexpected "="
```

- Downside: "=" is stored, but perhaps not needed
- Arguments of `parse`:
  - Your parser: `Parse a`
  - Source name for error messages (e.g., file name): `String`
  - Input string to parse: `String`

Ignoring parsing results

Ignoring parsing results

- Combinator *> ignores the *left* parsing result

```
assignment' :: Parser (String, String)
assignment' = (,) <$> identifier <*>
                      (symbol "=" *> identifier)
```

Ignoring parsing results

- Combinator *> ignores the *left* parsing result

```
assignment' :: Parser (String, String)
assignment' = (,) <$> identifier <*>
                      (symbol "=" *> identifier)
```

- Combinator <* ignores the *right* parsing result

```
assignment'' :: Parser (String, String)
assignment'' = (,) <$> (identifier <* symbol "=")
                       <*> identifier
```

<|> in ParSec

- Previous definition of <|> uses both parsers

<|> in ParSec

- Previous definition of <|> uses both parsers
- Parsec: use the second parser *only* when the first parser consumes no input

<|> in ParSec

- Previous definition of <|> uses both parsers
- Parsec: use the second parser *only* when the first parser consumes no input
- Does *not* accept “aa”:

```
a = char 'a'
b = char 'b'
parseX =  (,) <$> a <*> b
      <|> (,) <$> a <*> a
```

<|> in ParSec

- Previous definition of <|> uses both parsers
- Parsec: use the second parser *only* when the first parser consumes no input
- Does *not* accept “aa”:

```
a = char 'a'
b = char 'b'
parseX =  (,) <$> a <*> b
      <|> (,) <$> a <*> a
```

- `try` combinator
  - `try :: Parser a -> Parser a`
  - Failure: `try` pretends that no input was consumed
  - *Does* accept “aa”:

```
parseY =  try ((,) <$> a <*> b)
      <|> (,) <$> a <*> a
```

More combinators

- Parse something within parentheses:
  - `<while> = "while" "(" integer ")"`
  - ParSec:

```
while :: Parser Integer
while = reserved "while" *> (parens integer)
```

More combinators

- Parse something within parentheses:
  - `<while> = "while" "(" integer ")"`
  - ParSec:

```
while :: Parser Integer
while = reserved "while" *> (parens integer)
```

- Similar: `braces`, `angles`, `brackets`, `stringLiteral`, etc.

More combinators

- Parse something within parentheses:
  - `<while> = "while" "(" integer ")"`
  - ParSec:

```
while :: Parser Integer
while = reserved "while" *> (parens integer)
```

- Similar: `braces`, `angles`, `brackets`, `stringLiteral`, etc.
- Read the documentation for `Text.ParserCombinators.Parsec`

EBNF constructions

- Zero or more integers: `(integer)*`
  - `many integer :: Parser [Integer]`
  - `parse (many integer) "" "1 23" == Right [1,23]`
  - `parse (many integer) "" "" == Right []`

EBNF constructions

- Zero or more integers: `(integer)*`
  - `many integer :: Parser [Integer]`
  - `parse (many integer) "" "1 23" == Right [1,23]`
  - `parse (many integer) "" "" == Right []`
- One or more integers: `(integer)+`
  - `many1 integer :: Parser [Integer]`
  - `parse (many1 integer) "" "" == Left ...`

EBNF constructions

- Zero or more integers: `(integer)*`
  - `many integer :: Parser [Integer]`
  - `parse (many integer) "" "1 23" == Right [1,23]`
  - `parse (many integer) "" "" == Right []`
- One or more integers: `(integer)+`
  - `many1 integer :: Parser [Integer]`
  - `parse (many1 integer) "" "" == Left ...`
- Zero or one `Integer`: `(integer)?`
  - `option 42 integer :: Integer -> Parser [Integer]`
  - `parse (option 42 integer) "" "" == Right 42`
  - `parse (option 42 integer) "" "10" == Right 10`

EBNF constructions

- Zero or more integers: `(integer)*`
  - `many integer :: Parser [Integer]`
  - `parse (many integer) "" "1 23" == Right [1,23]`
  - `parse (many integer) "" "" == Right []`
- One or more integers: `(integer)+`
  - `many1 integer :: Parser [Integer]`
  - `parse (many1 integer) "" "" == Left ...`
- Zero or one `Integer`: `(integer)?`
  - `option 42 integer :: Integer -> Parser [Integer]`
  - `parse (option 42 integer) "" "" == Right 42`
  - `parse (option 42 integer) "" "10" == Right 10`
- Zero or one `Integer`: `(integer)?`
  - `optionMaybe integer :: Integer -> Parser [Integer]`
  - `parse (optionMaybe integer) "" "" == Right Nothing`
  - `parse (optionMaybe integer) "" "10" == Right 10`

Operator associativity

- Right associative expressions: $(1 * (2 * (\ldots)))$

```
mult :: Parser (Integer -> Integer -> Integer)
mult = (\_ -> (*)) <$> symbol "*"

rightassoc :: Parser Integer
rightassoc = integer `chainr1` mult
```

Operator associativity

- Right associative expressions: $(1 * (2 * (\ldots)))$

```
mult :: Parser (Integer -> Integer -> Integer)
mult = (\_ -> (*)) <$> symbol "*"

rightassoc :: Parser Integer
rightassoc = integer `chainr1` mult
```

- Expression trees:

```
data Tree = Leaf Integer | Node Tree Tree
                  deriving Show

mult' :: Parser (Tree -> Tree -> Tree)
mult' = (\_ -> Node) <$> symbol "*"

rightassoc' :: Parser Tree
rightassoc' = (Leaf <$> integer) `chainr1` mult'
```

Error reporting

- Parser for a function name:

```
f :: Parser String
f = identifier
```

Error reporting

- Parser for a function name:

```
f :: Parser String
f = identifier
```

```
*ParsecEx> parse f "" "*"
Left (line 1, column 1):
unexpected "*"
expecting identifier
```

Error reporting

- Parser for a function name:

```
f :: Parser String
f = identifier
```

```
*ParsecEx> parse f "" "*"
Left (line 1, column 1):
unexpected "*"
expecting identifier
```

- Custom error handling

```
f' :: Parser String
f' = identifier <?> "function name"
```

Error reporting

- Parser for a function name:

```
f :: Parser String
f = identifier
```

```
*ParsecEx> parse f "" "*"
Left (line 1, column 1):
unexpected "*"
expecting identifier
```

- Custom error handling

```
f' :: Parser String
f' = identifier <?> "function name"
```

```
*ParsecEx> parse f' "" "*"
Left (line 1, column 1):
unexpected "*"
expecting function name
```

Error reporting

- Parser for a function name:

```
f :: Parser String
f = identifier
```

```
*ParsecEx> parse f "" "*"
Left (line 1, column 1):
unexpected "*"
expecting identifier
```

- Custom error handling

```
f' :: Parser String
f' = identifier <?> "function name"
```

```
*ParsecEx> parse f' "" "*"
Left (line 1, column 1):
unexpected "*"
expecting function name
```

- `p <?> str` ⇒ `str` is used when `p` fails + consumes no input

Parsec: simple parser

```
<S> ::= <Q> | <R>
<Q> ::= 'a' 'b'
<R> ::= 'b' <Q> <Q>
      | 'c' <Q> 'c'
```

```
parseS :: Parser S
parseS = S1 <$> parseQ <|> S2 <$> parseR

parseQ :: Parser Q
parseQ = Q <$> char 'a' <*> char 'b'

parseR :: Parser R
parseR =     R1 <$> char 'b' <*> parseQ <*> parseQ
         <|> R2 <$> char 'c' <*> parseQ <*> char 'c'
```

Parsec: simple parser

```
<S> ::= <Q> | <R>
<Q> ::= 'a' 'b'
<R> ::= 'b' <Q> <Q>
      | 'c' <Q> 'c'
```

```
parseS :: Parser S
parseS = S1 <$> parseQ <|> S2 <$> parseR

parseQ :: Parser Q
parseQ = Q <$> char 'a' <*> char 'b'

parseR :: Parser R
parseR =     R1 <$> char 'b' <*> parseQ <*> parseQ
         <|> R2 <$> char 'c' <*> parseQ <*> char 'c'
```

Do not use the `char` parser for the assignments!

Self study

- This lecture: “Combinator parsing with ParSec”
  - Examples in `pp-student-block3.zip`
  - Use Hoogle for documentation
  - Books (not advised, they present a slightly different (difficult) approach):
    - Programming in Haskell (Hutton): Chapter 13
    - ParSec: Real World Haskell: Chapter 16

Self study

- This lecture: “Combinator parsing with ParSec”
  - Examples in `pp-student-block3.zip`
  - Use Hoogle for documentation
  - Books (not advised, they present a slightly different (difficult) approach):
    - Programming in Haskell (Hutton): Chapter 13
    - ParSec: Real World Haskell: Chapter 16

- Next lecture: “Advanced type classes”
  - Topics: `Monoid`, `Applicative` and `Foldable`
  - Books:
    - Learn you a Haskell (Lipovaca): Chapter 8 (Functor), Chapter 11 (Applicative, Monoid, Foldable)
    - Programming in Haskell (Hutton): Chapter 12.1 (Functor), Chapter 12.2 (Applicative), Chapter 14.1 (Monoid), Chapter 14.2 (Foldable)