Parsing Monad

Build a Library for constructing Parser.

jacc
↓ parser gen
Happy

Haskell code

lex
↓
Alex

} shift reduce parsing

LR(1)
LALR(1)

LL - Parsing

Recursive descent parsing

(-- Ocaml - menhir LR)

newtype Parser a = Parser (String → a)

Parser Combinator libraries

Parsec, megaParsec, altoParsec

Parser should handle these

(1) "Rest of the input"

(2) "Parse Error".

String → Maybe (a, string)

newtype Parser a = Parser (string → Maybe (a, string))

satisfy :: (Char → Bool) → Parser Char

digit = satisfy isDigit

char : Char → Parser Char

char x = satisfy (\c → c == x)

char x = satisfy ((==) x)

satisfy pr = Parser fn

where

fn :: String → Maybe (char, string)

fn (x : xs) = if pr x then Just (x, xs)

else Nothing

fn (nil) = Nothing

(<|>) : Parser a → Parser a → Parser a

p1 <|> p2 =

type Result a = Maybe (a, String)

newtype Parser a = Parser (String → Result a)

runParser :: Parser a → String → Result a

runParser (Parser fn) = fn

p1 <|> p2 = Parser fn

where fn input = case runParser p1 input of

Nothing → runParser p2 input

x → x

many :: Parser a → Parser [a]

many p = Parser fn

where fn input =

// this can be

if the input doesnt match p' then return empty else keep matching the remaining

this can be done differently.

newtype Parser a = Parser (String → Result a)

```
instance Functor Parser where

fmap :: (a -> b) -> Parser a -> Parser b

fmap f pa = Parser fn

  where fn :: String -> Result b

        fn input = ~~fn input = runParser~~

        fn input = fmap f (runParser pa input)
```

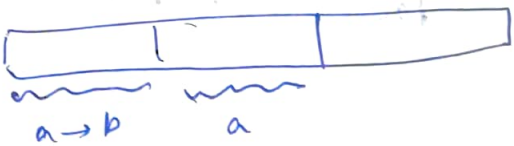
```
data Result a = Ok a String
              | Err

Instance Functor Result where
  fmap f (Ok a s) = Ok (fa) s
  fmap f Err      = Err
```

HW

Define the applicative instance of

```
<*> Parser (a -> b) -> Parser a
                    -> Parser b
```



Try out megaParsec or Parsec library to write small parsers

Parser as Monads

```
data Result a = Ok a String
              | Err
```

```
newtype Parser a = Parser (String -> Result a)
```

```
runParser :: Parser a -> (String -> Result a)
runParser (Parser fn) = fn
```

```
instance Functor Result where
  -- fmap :: (a->b) -> Result a -> Result b
  fmap f (Ok x str) = Ok (f x) str
  fmap f Err        = Err
```

```
instance Functor Parser
  -- fmap :: (a->b) -> Parser a -> Parser b
  fmap f pa = Parser (pfn :: String -> Result b)
    where pfn str = fmap f (runParser pa str)
```

```
instance Applicative Parser where
  -- pure :: a -> Parser a
  pure a = Parser (pfn :: String -> Result a)
    where pfn str = Ok a str

(<*>) :: Parser (a -> b) -> Parser (a) -> Parser (b)

pf <*> pa = Parser ( pb :: String -> Result b)
  where
    pb str = case runParser pf str of
               Ok f rest ->
               Err

                          -> Err

pf <*> pa = Parser (pb :: String -> Result b)
where
  pb str = case runParser pf str of
             Ok f rest -> case runParser pa rest of
             Err       -> Err

                          Ok a restp -> Ok (f a) rest
                          Err        -> Err

             Err -> Err
```

```
pf <*> pa = Parser (pb . String -> Result b)
```

where

```
pb str = case runParser pf str of
           Ok f rest -> fmap f (runParser pa rest)
           Err -> Err
```

```
digit = satisfy isDigit
alpha = satisfy isAlpha

many :: Parser a -> Parser [a]
many1 :: Parser a -> Parser [a]
```

```
data Person String Int

name = many1 alpha
age = read <$> (many1 digit)
```

```
Person <$> name <*> age
```

Define many interms of Applicative interface.

```
many1 p = p :) <$> p <*> many p
```

```
many p = many1 p <|> pure []
```

5-4-24

```
data Result a = Ok a String
              | Err

newtype Parser a = Parser (String → Result a)

instance Monad Parser where
  (>>=) :: Parser a → (a → Parser b) → Parser b

pb = do x ← pa
```

```
data S = RuleA Char S Char
       | RuleB Char S Char
       | RuleE

sp = RuleA <$> char 'a' <*> sp
                        <*> char 'a'

<|> RuleB
<|> pure RuleE
```

```
do x ← getInt Line
   repeat x getLine
-- reads an integer and read
   that many lines.

repeat :: Monad m ⇒ Int → m a → m [a]
repeat x action = if x <= 0 then pure []
                  else
                  (:) <$> action <*> repeat (x-1)
                                     action.
```

```
char :: Char → Parser Char
satisfy :: (Char → Bool) →
                  Parser Char
```

```
repeat :: Int → IO a → IO [a]
repeat x action = if x <= 0 then pure []
                  else y ← action
                       ys ←
```

```
(>>=) pa pbf = Parser fn
  where
    fn str = case runParser pa str where
               Err -> Err
               OK x rest -> runParser (pbf x) rest
```

```
integer :: Parser Int
integer = many digit
digit = satisfy isDigit
many :: Parser a -> Parser [a]
integer = read <$> (many digit)
```

Read through megaparsing.
or lazy parsing.

11-3-24

```
type String = [Char]
```

1) Text
2) Byte String — Lazy bytestring / strict byte string

State computation

```
newtype State s a = State {runState :: s -> (a,s)}
```

equivalent to

```
newtype State s a = State (s -> (a -> s))
runState (State fn) = fn
```

(a → s): value emitted + new state

s: initial state

instance Functor (State s) where

-- fmap :: (a→b) → State s a → State s b

fmap f sa = State fn

where

fn s₀ = case runstate sa s₀ of

(a, s₁) → (fa, s₁)

-- Do applicative by yourself

(>>=) :: State s a → (a → State s b) → State s b

do x ← sa
   fn x

(>>=) sa fn = State g

where

g s₀ = case runstate sa s₀ of

(a, s₁) → runstate (fn a) s₁

~~get :: State s s → s~~

get :: State s s

get = State fn

where

fn s₀ = (s₀, s₀)

put :: s → State s ()

put s = State fn

where

fn s₀ =

-- try writing the put function.