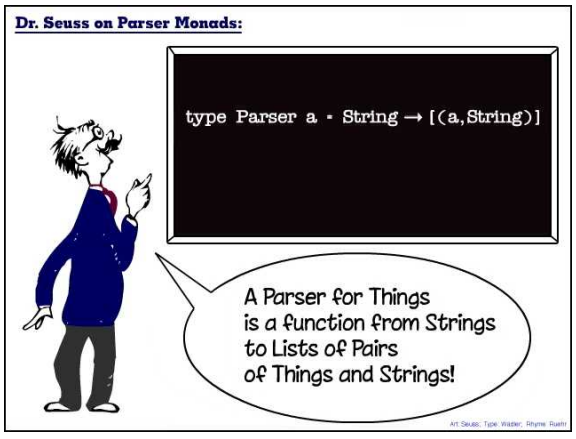
```
module Lab2 where

module Test where

import Lab2

import Control.Applicative
import Data.Char
```

Laboratorul 2

În acest laborator veți deveni experți în abordarea Haskell asupra noțiunii de analiză lexicală / analiză sintactică (lexing / parsing).

Pentru acest lucru, veți începe de la cel mai mic parser posibil (acela care știe să recunoască un caracter cu o properietate dată) și veți construi pe baza lui *combinatori de parsare* din ce în ce mai complecși.

Ce este un parser?



Pentru un tip dat `a`, un parser încearcă să obțină un obiect de tipul `a` dintr-un șir de intrare. De aici, este clar că tipul de intrare al unui parser va fi un șir de caractere (`String`).

Pentru a permite o abordare compozițională, i.e., construirea de parsere pentru expresii complexe folosind parsere pentru expresii mai simple, vrem ca un parser sa poata analiza doar începutul unui șir, deci el va calcula perechi de tipul `(a, String)`, unde prima componentă a perechii corespunde obiectului obținut în urma analizei, iar a doua corespunde șirului de caractere rămas după obținerea lui `a`.

Deoarece parsarea poate fi de multe ori ambguă, tipul rezultat e o listă de astfel de perechi reprezentând obiectele rezultate și șirurile rămase de procesat.

Un parser în Haskell

Pentru a putea defini diverse instanțe de clase de tipuri pentru parser, va trebui să începem prin a îl încapsula într-un tip înregistrare.

```
newtype Parser a = Parser { apply :: String -> [(a, String)] }
```

Dacă avem un parser, îl putem aplica unui șir de caractere folosind `apply` (vedeți și mai jos)

```
-- >>> apply anychar "abc"
-- [('a',"bc")]
```

Cel mai mic parser

Cel mai mic parser (dar foarte util, după cum veți vedea), este un parser pentru un singur caracter care trebuie să satisfacă un predicat dat:

```
satisfy :: (Char -> Bool) -> Parser Char
satisfy p = Parser go
  where
    go [] = []   -- imposibil de parsat șirul vid
    go (c:input)
      | p c = [(c, input)]   -- dacă predicatul ține, întoarce c și
restul șirului de intrare
      | otherwise = []       -- în caz contrar, imposibil de parsat
```

De exemplu, putem parsa un caracter alfanumeric din șirul de intrare:

```
-- | satisfy
-- >>> apply (satisfy isAlphaNum) "abc"
-- [('a',"bc")]
-- >>> apply (satisfy isAlphaNum) "2bc"
-- [('2',"bc")]
-- >>> apply (satisfy isAlphaNum) "@bc"
-- []
```

Exercițiu

Folosind `satisfy` (și funcții din `Data.Char`), definiți parsere pentru următoarele tipuri de caractere:

```
--- | Acceptă orice caracter
anychar :: Parser Char
anychar = undefined

-- | anychar
-- >>> apply anychar "&ab"
-- [('&',"ab")]

--- | acceptă doar caracterul dat ca argument
char :: Char -> Parser Char
char = undefined

-- | char
-- >>> apply (char 'i') "ionel"
-- [('i',"onel")]
-- >>> apply (char 'i') "Ionel"
-- []

--- | acceptă o cifră
digit :: Parser Char
digit = undefined

-- | digit
-- >>> apply digit "9"
-- [('9',"")]
-- >>> apply digit "Ionel"
-- []
-- >>> apply digit "99"
-- [('9',"9")]

--- | acceptă un spațiu (sau tab, sau sfârșit de linie -- vedeți funcția
din Data.Char )
space :: Parser Char
space = undefined

-- | space
-- >>> apply space " "
-- [(' ',"")]
```

```
-- >>> apply space "a "
-- []
-- >>> apply space "\na"
-- [('\n',"a")]
```

Parser de sfârșit

Un alt parser de bază, a cărui utilitate o vom vedea puțin mai încolo, este parser-ul pentru șirul vid, care va putea fi folosit pentru a cere explicit ca șirul de intrare să fi fost procesat complet:

```
--- | succes doar dacă am șirul de intrare este vid
endOfInput :: Parser ()
endOfInput  = Parser go
  where
    go "" = [((), "")]
    go _ = []
```

Spre monada `Parser`

Un parser este o funcție cu efecte laterale: primește un șir la intrare, și pe lângă identificarea unui obiect din șir produce și șirul rămas; mai mult obține listă de astfel de perechi.

De aceea, pentru a putea compune mai ușor parsere spre a obține un nou parser din mai multe componente, fără a ne preocupa prea mult de non-determinism și de trimiterea mai departe a șirului de intrare, este foarte util să facem `Parser` instanță a clasei `Monad`.

- Functor ne spune cum transformăm tipul rezultat al unei computații fără a afecta computația în sine.

```
instance Functor Parser where
    fmap f pa = Parser (\input -> [(f a, rest) | (a, rest) <- apply
pa input])
```

De exemplu, putem folosi instanța de functor pentru a aplica o funcție unui parser pentru a obține un parser pentru tipul rezultat (`<$>` este operatorul infix asociat lui `fmap`):

```
-- | fmap
-- >>> apply (digitToInt <$> digit) "7ab"
```

```
-- [(7,"ab")]
```

- Applicative ne spune cum putem combina o computație pentru o funcție cu o computație pentru tipul de intrare al funcției pentru a produce o computație pentru tipul de ieșire al funcției, propagând toate efectele laterale

```
instance Applicative Parser where
    pure a = Parser (\input -> [(a, input)])
    pf <*> pa = Parser (\input -> [(f a, resta) | (f, restf) <- apply
pf input, (a, resta) <- apply pa restf])
```

Un exemplu posibil este de a folosi instanța de Applicative pentru a parsa un număr de două cifre:

```
parseCifra = digitToInt <$> digit
douaCifre c1 c2 = 10 * c1 + c2

-- | parseCifra
-- >>> apply parseCifra "7ab"
-- [(7,"ab")]
-- >>> douaCifre 2 3
-- 23
-- >>> apply (pure douaCifre <*> parseCifra <*> parseCifra) "23c"
-- [(23,"c")]
```

Observație: Din legile satisfăcute de Applicative putem folosi `f <$> p` în loc de `pure f <*> p`

Applicative pune la dispoziție și doi operatori derivați, `*>` și `<*`, care permit ignorarea valorii unei computații:

```
-- | Sequence actions, discarding the value of the first argument.
(*>) :: Applicative f => f a -> f b -> f b
pa *> pb = pure (\a b -> b) <*> pa <*> pb

-- | Sequence actions, discarding the value of the second argument.
(<*) :: Applicative f => f a -> f b -> f a
pa *> pb = pure (\a b -> b) <*> pa <*> pb
```

De exemplu, putem folosi acești operatori pentru a parsa o cifră între paranteze:

```
-- | parse-brackets
-- >>> apply (char '(' *> digit  <* char ')') "(1)23"
-- [('1',"23")]
```

De asemeni, putem folosi `<*` împreună cu `endOfInput` pentru a garanta că tot șirul de intrare a fost consumat:

```
-- | endOfInput
-- >>> apply (digit <* endOfInput) "123"
-- []
-- >>> apply (digit <* endOfInput) "1"
-- [('1',"")]
```

Exercițiu

În general, odată ce avem un parser pentru intregul șir, ne interesează o singură parsare, care să consume tot șirul de intrare.

Scrieți o funcție `parse` care, date fiind un parser și un șir de intrare, verifică dacă printre soluțiile întoarse de parser există exact una care consumă tot șirul. Dacă da, atunci întoarce obiectul parsat ; dacă nu, va semnala o eroare. Pentru a putea semnala o problemă, va folosi tipul `Either String a` ca rezultat.

Sugestie: Folosiți `endOfInput` ca mai sus și `apply`.

```
parse :: Parser a -> String -> Either String a
parse = undefined
```

```
-- | parse
-- >>> parse digit "1"
-- Right '1'
-- >>> parse digit "12"
-- Left "Sirul de intrare nu a fost complet consumat sau parsare ambigua"
```

Parser ca monadă

Clasa Monad ne permite să secvențiem acțiuni pentru care alegerea unei acțiuni vitoare depinde de rezultatul acțiunii precedente:

```
instance Monad Parser where
    pa >>= k = Parser (\input -> [(b, restb) | (a, resta) <- apply pa
input, (b, restb) <- apply (k a) resta])
```

Instanța de monadă a lui Parser ne permite, de exemplu, să folosim notația do pentru a secvența mai multe parsere. În particular, pentru a parsa o cifră între paranteze, am putea folosi următoarea definitie:

```
cifraIntreParanteze :: Parser Int
```

```
cifraIntreParanteze
  = do
    char '('
    d <- digit
    char ')'
    return (digitToInt d)
```

Exercițiu

Scrieți un parser pentru a extrage o cifră prefixată de un semn `+` sau `-` și obține valoarea întreagă a cifrei (luând în calcul și semnul).

Folosiți instanța de monadă pentru a defini parserul

```
cifraSemn :: Parser Int
cifraSemn = undefined

-- | cifraSemn
-- >>> apply cifraSemn "-123"
-- [(-1,"23")]
-- >>> apply cifraSemn "+23"
-- [(2,"3")]
-- >>> apply cifraSemn "23"
-- []

-- esueaza deoarece nu are semn
```

Puteți scrie parserul de mai sus folosind doar operațiile din `Applicative`?

Exercițiu

Scrieți un parser pentru un șir de caractere dat ca argument

```
string :: String -> Parser String
string = undefined
```

Sugestie: Puteți folosi recursie

```
-- | string
-- >>> apply (string "Hi") "Hike"
-- [("Hi","ke")]
-- >>> apply (string "May") "March"
-- []
```

```
-- >>> apply (string "Hi") "HiHi"
-- [("Hi","Hi")]
-- >>> apply (string "March") "March"
-- [("March","")]
```

Parsarea mai multor alternative

Regulile de analiză lexicală și sintactică se pot scrie adeseori folosind variante. De exemplu, definiția unui număr natural ca secvență de cifre poate fi scrisă (gramatical) astfel:

```
<Natural> ::= <Digit> <Natural>
            | <Digit>
```

Pentru a putea defini un parser prin combinarea mai multor alternative, vom face `Parser` instanță a clasei `Alternative`.

```
instance Alternative Parser where
    empty = Parser (const [])
    p <|> p' = Parser (\input -> apply p input ++ apply p' input)
```

- `<|>` combină două parsere, executându-le pe amândouă pe același input și considerând rezultatele de la amândouă.
- `empty` reprezintă parserul vid (folosit pe post de element neutru al lui `<|>`)

Exercițiu (mai greu)

Implementați un parser pentru numere naturale pe baza gramaticii de mai sus

```
naiveNatural :: Parser Int
naiveNatural = undefined
```

`some` and `many`

`Alternative` pune la dispoziție și două funcții pentru aplicarea unui parser în mod repetat, `some` și `many`.

```
-- | One or more.
some :: Alternative f => f a -> f [a]
some v = pure (:) <*> v <*> many v

-- | Zero or more.
```

```
many :: Alternative f => f a -> f [a]
many v = some_v <|> pure []
```

Exerciții (ușoare)

Folosiți `many`, `some` si parserele definite mai sus pentru a defini (monadic sau applicativ) următoarele parsere:

```
-- | Elimină zero sau mai multe apariții ale lui `space`
whiteSpace :: Parser ()
whiteSpace = undefined

-- | whiteSpace
-- >>> apply whiteSpace " \t\nksdw"
-- [((),"ksdw"),((),"\nksdw"),((),"\t\nksdw"),((),"  \t\nksdw")]
-- >>> apply whiteSpace "ionel"
-- [((),"ionel")]

-- | parses a natural number (one or more digits)
nat :: Parser Int
nat = undefined

-- | nat
-- >>> apply nat "12ab"
-- [(12,"ab"),(1,"2ab")]
-- >>> apply nat "ionel"
-- []

-- | aplică un parser, și elimină spațiile de după
lexeme :: Parser a -> Parser a
lexeme = undefined

-- | lexeme
-- >>> apply (lexeme (string "Hello")) "Hello  World!"
-- [("Hello","World!"),("Hello"," World!"),("Hello","  World!")]

-- | parses a natural number and skips the space after it
natural :: Parser Int
natural = undefined

-- | Parses the string and skips whiteSpace after it
symbol :: String -> Parser String
symbol = undefined
```

```
-- | symbol
-- >>> apply (symbol "if") "if (x) ..."
-- [("if","(x) ..."),("if"," (x) ...")]

-- | Parses the string, skips whiteSpace, returns unit
reserved :: String -> Parser ()
reserved = undefined

-- | reserved
-- >>> apply (reserved "if") "if (x) ..."
-- [((),"(x) ..."),((),"  (x) ...")]

-- | parsează virgulă, eliminând spațiile de după
comma :: Parser ()
comma = undefined
```

Exerciții cu paranteze

Folosind stilul monadic sau aplicativ, și combinatorul `symbol` de mai sus implementați următoarele parsere:

```
-- | parsează argumentul intre paranteze rotunde
--   elimină spațiile de după paranteze
parens :: Parser a -> Parser a
parens = undefined

-- | parsează argumentul intre paranteze pătrate
--   elimină spațiile de după paranteze
brackets :: Parser a -> Parser a
brackets = undefined
```

Exercitiu: alte parsere

```
-- | una sau mai multe instanțe, separate de virgulă,
--   cu eliminarea spațiilor de după fiecare virgulă
--   intoarce lista obiectelor parsate
commaSep1 :: Parser a -> Parser [a]
commaSep1 = undefined
```

Sugestie: parsați un element, apoi folosiți `many` pentru combinația dintre `comma` și parserul de element

```
-- | commaSep1
-- >>> apply (commaSep1 natural) "3  ,  4  , a, 5"
-- [([3,4],", a, 5"),([3,4]," , a, 5"),([3,4],"  , a, 5"),([3],",  4
 , a, 5"),([3]," ,  4  , a, 5"),([3],"  ,  4  , a, 5")]

-- | zero sau mai multe instanțe, separate de virgulă,
--   cu eliminarea spațiilor de după fiecare virgulă
--   intoarce lista obiectelor parsate
commaSep :: Parser a -> Parser [a]
commaSep = undefined
```

Sugestie: Folosiți `commaSep1` de mai sus.

```
-- | date fiind parsere pentru prima literă si pentru felul literelor
următoare
--   scrieți un parser pentru un identificator
ident :: Parser Char -> Parser Char -> Parser String
ident identStart identLetter = undefined

-- | ca mai sus, dar elimină spatiile de după
identifier :: Parser Char -> Parser Char -> Parser String
identifier start letter = lexeme (ident start letter)

-- | identifier
-- >>> apply (identifier (satisfy isAlpha) (satisfy isAlphaNum)) "ij1
+ 3"
-- [("ij1","+ 3"),("ij1"," + 3"),("ij","1 + 3"),("i","j1 + 3")]
```