```
{-------------------------------------------------------------------------

                A LIBRARY OF MONADIC PARSER COMBINATORS

                              29th July 1996
                           Revised, October 1996
                         Revised again, November 1998

                 Graham Hutton               Erik Meijer
            University of Nottingham    University of Utrecht

This Haskell 98 script defines a library of parser combinators, and is taken
from sections 1-6 of our article "Monadic Parser Combinators".  Some changes
to the library have been made in the move from Gofer to Haskell:

   * Do notation is used in place of monad comprehension notation;

   * The parser datatype is defined using "newtype", to avoid the overhead
     of tagging and untagging parsers with the P constructor.

     Comentata de Dan Popa, Univ. Bacau, 16 feb. 2006
     Cu intrebari incluse !
     Atentie: In aceasta versiune a bibliotecii functia de aplicare e papply.
-------------------------------------------------------------------------}

module ParseLib
   (Parser, item, papply, (+++), sat, many, many1, sepby, sepby1, chainl,
    chainl1, chainr, chainr1, ops, bracket, char, digit, lower, upper,
    letter, alphanum, string, ident, nat, int, spaces, comment, junk,
    parse, token, natural, integer, symbol, identifier, module Monad) where

import Char
import Monad

infixr 5 +++

--- The parser monad ----------------------------------------------------

newtype Parser a   = P (String -> [(a,String)])


{- map:
   Transfera functii dintre valori in functii dintre parsere ce intorc valorile.
   Daca un parser intoarce o anumita valoare, il putem astfel afecta, trecand
   valoarea printr-o functie.Lipseste asa ceva pentru functii binare, ternare.
 -}

instance Functor Parser where
   -- map         :: (a -> b) -> (Parser a -> Parser b)
   fmap f (P p)    = P (\inp -> [(f v, out) | (v,out) <- p inp])
```

```haskell
{- return:
  Returnul monadei parserelor. Cu ajutorul sau se scrie interpretorul/parserul
  rezultat care da o anumita valoare la finalul evaluarii.
  E urmat de operatorul bind.
-}
instance Monad Parser where
   -- return      :: a -> Parser a
   return v        = P (\inp -> [(v,inp)])

   -- >>=         :: Parser a -> (a -> Parser b) -> Parser b
   (P p) >>= f     = P (\inp -> concat [papply (f v) out | (v,out) <- p inp])


{- mzero:
  Parserul (parserle) zero, elemente neutre la adunarea monadica.
  Rateaza parsarea si da lista vida ca raspuns.
  Adunarea parserelor creeaza un parser care parseaza 'nedetermminist'
  in doua moduri si concateneaza listele cu rezultate.
-}
instance MonadPlus Parser where
   -- mzero            :: Parser a
   mzero               = P (\inp -> [])

   -- mplus            :: Parser a -> Parser a -> Parser a
   (P p) `mplus` (P q)  = P (\inp -> (p inp ++ q inp))

--- Other primitive parser combinators ------------------------------------

{- Parserul pentru un caracter oarecare, ce-l accepta. -}
item              :: Parser Char
item               = P (\inp -> case inp of
                                  []     -> []
                                  (x:xs) -> [(x,xs)])

{- Forteaza un parser sa dea o lista de perechi ! -}
force             :: Parser a -> Parser a
force (P p)        = P (\inp -> let x = p inp in
                                (fst (head x), snd (head x)) : tail x)

{- Forteaza un parser sa dea doar prima parsare gasita ! -}
first             :: Parser a -> Parser a
first (P p)        = P (\inp -> case p inp of
                                  []     -> []
                                  (x:xs) -> [x])

{- Aplicarea functiei dintr-un parser (P p) pe inputul inp -}
papply            :: Parser a -> String -> [(a,String)]
papply (P p) inp   = p inp

--- Derived combinators ---------------------------------------------------

{-
```

```
  Da doar prima parsare posibila din cele facute de doua parsere.
  Deoarece esecul este codat ca lista vida si e coada oricarei liste,
  vom obtine o singura parsare posibila, una a parserului care
  reuseste, in cazul cand doar unul din parsere reuseste!
-}
(+++)             :: Parser a -> Parser a -> Parser a
p +++ q            = first (p `mplus` q)

{- Parseaza caracterul care satisface conditia.     -}
{- Pe cand un combinator similar pentru stringuri ? -}
sat               :: (Char -> Bool) -> Parser Char
sat p              = do {x <- item; if p x then return x else mzero}


{- Parseaza succesiuni de texte acceptate de un alt parser si da ca
   rezultat al interpretarii lista valorilor.
   Succesiuni de 0 sau mai multe texte.                        -}
many              :: Parser a -> Parser [a]
many p             = force (many1 p +++ return [])

{- Parseaza succesiuni de texte acceptate de un alt parser si da ca
   rezultat al interpretarii lista valorilor.
   Succesiuni de 1 sau mai multe texte.                        -}
many1             :: Parser a -> Parser [a]
many1 p            = do {x <- p; xs <- many p; return (x:xs)}

{- Parseaza succesiuni de texte parsate de doua parsere,
   (unul pentru text,altul pentru separator)
   returneaza lista valorilor acelor texte.
   Textul se repeta inclusiv de zero ori.                 -}
sepby             :: Parser a -> Parser b -> Parser [a]
p `sepby` sep      = (p `sepby1` sep) +++ return []

{- Parseaza succesiuni de texte parsate de doua parsere,
   (unul pentru text,altul pentru separator)
   returneaza lista valorilor acelor texte.
   Textul se repeta de 1 sau mai multe ori.                 -}
sepby1            :: Parser a -> Parser b -> Parser [a]
p `sepby1` sep     = do {x <- p; xs <- many (do {sep; p}); return (x:xs)}

{-  Pentru constructia de termi si factori pe ideea ca se mai poate
    adauga mereu un operator si un element. Operatorul trebuie sa fie
    asociativ la stanga si binar si returnat ca valoare de un parser
    care-l recunoaste. Admite O sau mai multi termi/factori.        -}
chainl            :: Parser a -> Parser (a -> a -> a) -> a -> Parser a
chainl p op v      = (p `chainl1` op) +++ return v

{-  Pentru constructia de termi si factori pe ideea ca se mai poate
    adauga mereu un operator si un element. Operatorul trebuie sa fie
    asociativ la stanga si binar si returnat ca valoare de un parser
    care-l recunoaste. Admite 1 sau mai multi termi/factori.        -}
chainl1           :: Parser a -> Parser (a -> a -> a) -> Parser a
```

```
p `chainl1` op     = do {x <- p; rest x}
                     where
                        rest x = do {f <- op; y <- p; rest (f x y)}
                                 +++ return x

{-  Similar dar pentru operatii asociative la dreapta !? -}
chainr            :: Parser a -> Parser (a -> a -> a) -> a -> Parser a
chainr p op v      = (p `chainr1` op) +++ return v

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

{-  Dintr-o lista de Parseri-operatori, calculati ca valori ale interpretarii
    obtine un parser care poate recunoaste toti acei operatori deodata.
    Util pentru parserele addop, mullop si similari proveniti din operatori. -}
ops               :: [(Parser a, b)] -> Parser b
ops xs             = foldr1 (+++) [do {p; return op} | (p,op) <- xs]


{- Construieste parserul unei secvente parantezate din parserele care
   recunosc paranteza initiala, interiorul, paranteza finala.
   Bineinteles, rezultatul e dat de parserul interiorului. -}
bracket           :: Parser a -> Parser b -> Parser c -> Parser b
bracket open p close = do {open; x <- p; close; return x}

--- Useful parsers --------------------------------------------------------

{- Parserul care recunoaste exact un anumit caracter
   si-l returneazaa ca valoare a interpretarii.     -}
char              :: Char -> Parser Char
char x             = sat (\y -> x == y)

{- Parserul care recunoaste exact un  caracter-cifra
   si-l returneazaa ca valoare a interpretarii.
   Pe cand unul care returneaza chiar valoarea cifrei ?    -}
digit             :: Parser Char
digit              = sat isDigit

{- Parserul care recunoaste exact un caracter-minuscula.
   si-l returneazaa ca valoare a interpretarii.     -}
lower             :: Parser Char
lower              = sat isLower

{- Parserul care recunoaste exact un caracter-majuscula
   si-l returneazaa ca valoare a interpretarii.     -}
upper             :: Parser Char
upper              = sat isUpper

{- Parserul care recunoaste exact un caracter-alfabetic
```

```
   si-l returneazaa ca valoare a interpretarii.     -}
letter            :: Parser Char
letter             = sat isAlpha

{- Parserul care recunoaste exact un caracter-alfanumeric
   si-l returneazaa ca valoare a interpretarii.          -}
alphanum          :: Parser Char
alphanum           = sat isAlphaNum

{- Parserul care recunoaste exact un string si-l returneaza. -}
string            :: String -> Parser String
string ""          = return ""
string (x:xs)      = do {char x; string xs; return (x:xs)}

{- Parserul care recunoaste identificatori fara spatii vecine. -}
ident             :: Parser String
ident              = do {x <- lower; xs <- many alphanum; return (x:xs)}

{- Parserul care recunoaste numere naturale fara semn. -}
nat               :: Parser Int
nat                = do {x <- digit; return (digitToInt x)} `chainl1` return op
                     where
                        m `op` n = 10*m + n

{- Parserul care recunoaste numere naturale cu minus sau fara semn. -}
int               :: Parser Int
int                = do {char '-'; n <- nat; return (-n)} +++ nat

--- Lexical combinators -------------------------------------------------------

{- Parserul pentru un sir de spatii. -}
spaces            :: Parser ()
spaces             = do {many1 (sat isSpace); return ()}


{- Parserul pentru comentarii Haskell. Alte parsere pentru
   comentarii care cred ca se pot face din combinatorul pentru paranteze. -}
comment           :: Parser ()
comment            = do {string "--"; many (sat (\x -> x /= '\n')); return ()}

{- Parserul pentru text nesemnificativ, comentarii sau spatii. -}
junk              :: Parser ()
junk               = do {many (spaces +++ comment); return ()}

{- Combinatorul care elimina textul inutil din fata unui element de parsat.
   Atentie: In aceasta versiune a bibliotecii functia de aplicare e papply.-}
parse             :: Parser a -> Parser a
parse p            = do {junk; p}

{- Combinatorul care elimina textul inutil DE DUPA unui element de parsat.
   Fara sa piarda valoarea elementului !
   Folosind acest combinator token pentru a incapsula un parser avem garantia
```

```haskell
   ca urmatorul va prelucra exact urmatorul atom lexical semnificativ,
   practic urmatorul atom lexical.                                    -}
token             :: Parser a -> Parser a
token p            = do {v <- p; junk; return v}

--- Token parsers ---------------------------------------------------------

{- Parserul care recunoaste numere naturale fara semn urmate de text inutil. -}
natural           :: Parser Int
natural            = token nat

{-
Parserul care recunoaste numere naturale cu/fara semn urmate de text inutil.
-}
integer           :: Parser Int
integer            = token int

{- Parserul care recunoaste stringuri urmate de text inutil. -}
symbol            :: String -> Parser String
symbol xs          = token (string xs)


{- Combinatorul care dintr-o lista de stringuri genereaza parserul care
   accepta toate stringurile identificatori care NU SUNT pe lista !
   Iar daca nu sunt identificatori returneaza fail ! -}
identifier        :: [String] -> Parser String
identifier ks      = token (do {x <- ident; if not (elem x ks) then return x
                                                              else mzero})

-------------------------------------------------------------------------
```