General:

- `newtype Parser a = P (String -> [(a,String)])`
- Predicate: a function that takes one argument and returns a boolean
  - if `pred x == True` then `x` satisfies predicate `pred`

Parsing.hs:

- `sat :: (Char -> Bool) -> Parser Char`
  - returns a character if that character satisfies the predicate
- `digit, letter, alphanum :: Parser Char`
  - parses a digit, letter, or alpha-numeric letter respectively
- `char :: Char -> Parser Char`
  - `char 'a'` parses exactly the character `'a'`
- similar to above: `digit letter alphanum lower upper string`
- `many :: Parser a -> Parser [a]`
  - parses 0 or more instances of `a` and collects them into a list
- `many1 :: Parser a -> Parser [a]`
  - same as `many`, but
- `(+++)` choice:
  - parse first argument if possible, else parse second argument
  - first successfully parsed argument is returned

```
(+++) :: Parser a -> Parser a -> Parser a
p +++ q =  P (\inp -> case parse p inp of
                 []        -> parse q inp
                 [(v,out)] -> [(v,out)])
```

- `((>>=))` sequential composition
  - `a >>= b` unboxes monad `a` into an output `a0` and then unboxes monad `b` with input `a0`

```
type Parser a = String -> [(a, String)]
-- implementation for in-class mostly-complete
-- parser 'monads'
(>>=) :: Parser a -> (a -> Parser b) -> Parser b
(>>=) p1 p2 = \inp -> case parse p1 inp of
      []  ->  []
      [(v, out)] -> parse (p2 v) out
```

  - usage:

```
doubleDigit :: Parser [Char]
doubleDigit =
  digit >>= \a ->
  digit >>= \b ->
  return [a,b]
-- is equivalent to
doubleDigit' :: Parser [Char]
doubleDigit' = do
  a <- digit
  b <- digit
  return [a,b]
```

  - `(>>)` is the same except that it discards the result of the first monad (thus it has signature `(>>) :: Parser a -> Parser b -> Parser b`)

Parsing Examples:

- bind and lambda method of parsing:
  - parse a number:
- parse arithmetic expressions using do syntax:

```
expr :: Parser Int
expr  = do t <- term
           do {char '+'
              ;e <- expr
              ;return (t + e)
              }
            +++ return t
term :: Parser Int
term  = do f <- factor
           do char '*'
              t <- term
              return (f * t)
            +++ return f
factor :: Parser Int
factor  = do d <- digit
             return (digitToInt d)
           +++ do char '('
                  e <- expr
                  char ')'
                  return e
eval   :: String -> Int
eval xs = fst (head (parse expr xs))
```

Trees:

- represent either a leaf node or some kind of internal node
- arithmetic tree declaration:

```
data Expr = Val Int
          | Neg Expr
          | Add Expr Expr
          | Mul Expr Expr
```

- how to fold over a tree:

```
-- exprFold  valF        negF      addF
exprFold :: (Int->b) -> (b->b) -> (b->b->b) ->
-- mulF        input   output
  (b->b->b) -> Expr -> b
exprFold valF _ _ _ (Val i) = valF i
exprFold valF negF addF mulF (Neg e)
  = negF  (exprFold valF negF addF mulF e)
exprFold valF negF addF mulF (Add s1 s2)
  = addF  (exprFold valF negF addF mulF s1)
          (exprFold valF negF addF mulF s2)
exprFold valF negF addF mulF (Mul s1 s2)
  = mulF  (exprFold valF negF addF mulF s1)
          (exprFold valF negF addF mulF s2)
```