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`
- function composition:

```
-- the . operator composes functions:
(f . g) x == f (g x)
```

useful functions:

```
--
nubBy :: (a -> a -> Bool) -> [a] -> [a]
nubBy pred xs = -- unique elements only from xs as
                -- determined by pred
nub :: Eq a => [a] -> [a]
nub xs = nubBy (==) a -- unique elements from xs
--
words :: String -> [String]
words xs =  -- list of whitespace-separated
            -- words from xs
--
-- concatenate container of lists
concat :: Foldable t => t [a] -> [a]
-- or for list-of-lists specifically:
concat :: [[a]] -> [a]
concat xs = foldl (++) [] xs
--
-- like concat, but use a function to get the inner lists
concatMap :: (a -> [b]) -> [a] -> [b]
concatMap f xs =  foldr ((++) . f) [] xs
```

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)
```

  - basically, just collect values into some type `b` and use supplied functions at each node to fold into single value
  - useful for evaluating simple things like:

```
-- evaluate an expression
evalExpr'    = exprFold id (\x -> 0 - x) (+) (*)
id --  integers map to integers
(\x -> 0 - x) -- negation
-- everything else is just simple numeric operators
--
-- count leaves in a tree
countLeaves' = exprFold (\_ -> 1) id (+) (+)
(\_ -> 1) -- leaf integer node is one node
id -- negation node has only one child, pass on count
(+) (+) -- nodes with two children: add number
        -- of leaf grandchildren
```

HW2: Water Gates:

```
waterGate :: Int -> Int
waterGate n =
 length -- number of True's
 $ filter id -- filter just True's
 $ waterGate' n initial -- initial call to helper
 where
  -- start with all gates closed
  initial = replicate n False
  --
  -- flip states
  waterGate' 1 state = map not state
    -- base case: flip every state
  waterGate' n state = flip n $ waterGate' (n-1) state
    -- otherwise, first get the state for (n-1) and then flip ev
  --
  -- flip every nth gate
  flip :: Int -> [Bool] -> [Bool]
  flip 1 xs = map not xs -- flip every gate
  -- flip only gates which index are multiples of n
  flip nth xs = [ if (i `mod` nth == 0) then not x else x
                -- zip each state with it's index
                | (x,i) <- (zip xs [1..])  ]
```

HW2: Goldbach's Other Conjecture:

```
-- check if a number is prime
primeTest :: Integer -> Bool
```

```haskell
primeTest 1 = False
primeTest t =  and [ (gcd t i) == 1 | i <- [2..t-1]]
-- all numbers less than n that are double a square
twiceSquares :: Integer -> [Integer]
twiceSquares n = takeWhile (<n) [ 2 *x^2 | x <- [1..]]
-- list of odd numbers
oddList = map (\x -> 2*x + 1) [0..]
-- all odd numbers that are composite (not prime)
allOddComp = [ o | o <- (drop 1 oddList)
                 , not (primeTest o)     ]
-- if a number satisfies conditions for conjecture
-- method: for enough square nubmers, check if n-(that number)
-- is prime
satsConds n = or [ primeTest k |
                  k <- map (\x->(n-x)) (twiceSquares n) ]
-- find the first number
goldbachNum = head [ x | x <- allOddComp, not (satsConds x) ]
```

HW4: Sets:

```haskell
type Set a = [a]
a = mkSet [1,2,3,4,5]
b = mkSet [1,2,3]
addToSet :: Eq a => Set a -> a -> Set a
addToSet s a | a `elem` s = s
             | otherwise = a : s
mkSet :: Eq a => [a] -> Set a
mkSet lst = foldl addToSet [] lst
isInSet :: Eq a => Set a -> a -> Bool
isInSet [] _ = False
isInSet [a] b = a == b
isInSet (x:xs) b | x == b = True
                 | otherwise = isInSet xs b
subset :: Eq a => Set a -> Set a -> Bool
subset sub super = and [ isInSet super x | x <- sub  ]
setEqual :: Eq a => Set a -> Set a -> Bool
setEqual a b = subset a b && subset b a
-- instance (Eq a) => Eq (Set a) where
--    a == b = subset a b && subset b a
setProd :: Set a -> Set a -> [(a,a)]
setProd a b = [ (ai,bj) | ai <- a
                        , bj <- b
                        ]
```