

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 library functions:

```
-- Data.List
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
--
-- get the longest prefix of xs for which pred is true
-- and also return the rest of the list
span :: (a -> Bool) -> [a] -> ([a], [a])
span pred xs = (takeWhile pred xs, dropWhile pred xs)
--
-- repeat a = infinite list of a
repeat :: a -> [a]
repeat x = map (\_ -> x) [1..]
repeat x = [ x | _ <- [1..] ]
-- replicate n a = list of length n repeating a
replicate :: Int -> a -> [a]
replicate n x = map (\_ -> x) [1..n]
replicate n x = [ x | _ <- [1..n] ]
--
-- folds (works on any foldable, not just lists)
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z [a,b,c] = a `f` (b `f` (c `f` z))
foldr f z [a,b,c] = f a $ f b $ f c z
-- combines into z from right to left
-- can potentially work on an empty list if one of the
-- folds does not evaluate it's second argument
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f z [a,b,c] = ((z `f` a) `f` b) `f` c
foldl f z [a,b,c] = f (f (f z a) b) c
-- evaluates from right to left
-- will not work on infinite list because it must start
-- at the end of the list
--
-- these are the same as above, except they take the
-- first two elements for the first application of f
foldr1 :: (a -> a -> a) -> [a] -> a
foldl1 :: (a -> a -> a) -> [a] -> a
```

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'
- **item :: Parser Char**
 - * parses any character
- 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 every nth state
    --
    -- 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
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:

```
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
  ]
```

Prev Exam: Run Length Encoding:

```
import Parsing
import Data.Char

q4 = do
  d <- sat isUpper
  e <- char (toLower d)
  f <- many item
  return [d,e]

ones = (map (\_ -> 1) [1..])

myRLE [] = []
myRLE ls = myhelper (zip ones ls)

myhelper [(n,c)] = [(n,c)]
myhelper ((n,c):(m,d):rest)
  | (d == c) = myhelper ((n+m),c):rest
  | otherwise = (n,c):myhelper ((m,d):rest)
```

Rock Paper Scissors:

```
data RPS = Rock | Paper | Scissors
  deriving (Eq, Show)

rps :: RPS -> RPS -> Int
rps a b | a == b = 0
rps Rock Scissors = 1
rps Paper Rock = 1
rps Scissors Paper = 1
rps _ _ = 2

rps2 :: RPS -> RPS -> Int
rps2 a b =
  if a == b then 0 else case (a,b) of
    (Rock, Scissors) -> 1
    (Paper, Rock) -> 1
    (Scissors, Paper) -> 1
    _ -> 2
```

99 problems:

```
-- 9. pack consecutive duplicates into sublists
pack (x:xs) = let (first,rest) = span (==x) xs
  in (x:first) : pack rest

pack [] = []
-- example:
pack [1,2,3,2,2,3] == [[1,1],[2],[3],[2,2],[3]]
```

Java:

- **Class Invariant:** A logical condition that ensures that an object of a class is in a well-defined state.
 - * public methods assume that invariant holds before it's called, and makes sure to preserve the invariant property
- garbage collection deals only with memory. You must manage other resources manually, such as concurrent locks, OS file handles, etc...
- anonymous classes are a thing
- inner class: class declared inside another class implicitly holds a reference to it's outer class.
 - This means instances of the inner class can use non-static fields/methods of the outer class.
 - This is especially handy for callbacks e.g. on android
- abstract class vs interface:**
 - Interface: all fields are **public static final**, all methods are **public**

- **abstract class** can extend exactly one parent class and implement any number of interfaces
- interface can extend (not implement) any number of interfaces
- abstract class can have constructor that initializes values and whatnot, but interface cannot
 - * you can't instantiate an abstract class directly, you can only call it's constructor from inside the constructor of a child class. Could still be useful though.
- interfaces and abstract classes can never be instantiated
 - * references of interface type refer to an instance of a class that implements that interface
 - * references of abstract class type refer to an instance of a subclass of that type

Inheritance and Virtual Methods:

- Java classes can inherit from one class only
 - * inheritance: **class ChildClass extends** ParentClass
 - * child class gets access to public fields/methods (of course) and protected fields/methods
 - * child class does not get access to private fields/methods
 - * TODO abstract classes/methods
- interfaces: **class MultiPurpose implements** Interface1, IFace2...
 - * basically an end around lack of multiple inheritance
 - * interfaces cannot be instantiated, but you can have a reference of interface type. In that case, the object it points to is a real concrete class, but all that you know about it is that it implements the specified interface
- virtual dispatch:
 - * all public non-static class methods are virtual. This means that if a subclass overrides a parent class method, the decision of which method implementation to use is made at runtime, depending on which type of object the reference actually refers to.
 - * all interface methods are by definition virtual
 - * private methods are not virtual, static methods are not virtual
- notable interfaces:
 - * **Iterable<T>**: contains **Iterator<T>** **iterator()** method, to iterate over container
 - * **Iterator<E>**: encapsulates an iteration over a container. Unlike C++ iterators, this iterator knows when it's reached the end, instead of relying on comparison to a one-past-end iterator
 - **boolean hasNext()**: ask if we're at the end
 - **E next()**: retrieve next element, and advance iterator
 - * **Runnable**: single **void run()** method. This is the interface that threads use

Generics:

- type bound: **class SortedList<T extends Comparable & Serializable>**
 - * you use **extends** for constraints that are classes or interfaces
- wildcards: **static void printAll(List<?> lst)** use ? for when you want to accept an object of any type

```
import java.lang.*;
class GenericWildcards {
    // T is the binding of the generic parameter
    private static class GenericBox<T> {
        // it is optional, but we need it if we want to do
        // things with that type
        public T t;
        public GenericBox(T t) { this.t = t; }
    }
    private static class NumberBox<T extends Number> {
        public T t;
        public NumberBox(T t) { this.t = t; }
    }
    public static void printBox(GenericBox<?> b) {
        // here we use the ? wildcard with no type binding
        // because we don't need to do things with that type
        // specifically
        System.out.println(b.t);
    }
    // method generic goes before return type
```

```
public static <T> void printWithParameter(GenericBox<T>
    b) {
    System.out.println(b.t);
}
public static void main(String[] args) {
    // this is using raw types, generally considered bad
    GenericBox rawBox = new GenericBox("asdf1"); //
    // compiler warnings
    // this cast is ok because raw types hold
    java.lang.Object
    Object o1 = rawBox.t;
    // this causes no warnings for same reason as
    // assignment above doesn't
    printBox(rawBox);

    // this is just using an unknown type, java says it's
    // fine
    GenericBox<?> unknownBox = new GenericBox<>("asdf2");
    // this is also OK because <?> explicitly makes the
    // generic parameter as java.lang.Object
    Object o2 = unknownBox.t;
    printBox(unknownBox);

    GenericBox<String> stringBox = new
    GenericBox<>("asdf3");
    // the type parameter above allows Java to infer that
    // this cast is safe
    String s = stringBox.t;
    System.out.println(s);
    // must specify type between class access and method
    // name (works the same for instance methods too)
    GenericWild-
    cards.<String>printWithParameter(stringBox);

    // correct stuff works like expected
    NumberBox<Integer> nb1 = new NumberBox<>(5);
    System.out.println(nb1.t + 1);
    //
    // this will fail to even allow NumberBox<String>
    // because that type doesn't work
    // NumberBox<String> sb1 = new NumberBox<>("asdf");
    //
    // this will fail because the inferred type of
    // NumberBox<>("asdf") is NumberBox<String>, which isn't
    // allowed
    // NumberBox<?> sb1 = new NumberBox<>("asdf");
}
```

Threading:

```
import java.util.concurrent.locks.ReentrantLock;
import java.util.concurrent.locks.Condition;
```

ReentrantLock:

- **ReentrantLock**: basically a mutex
- **ReentrantLock.lock()**: acquire the lock (blocking)
 - * does **not** throw **InterruptedException**
- **ReentrantLock.unlock()**: release the lock
 - * does **not** throw **InterruptedException**
 - * you should always wrap your locking code in a **try{} block** (including the call to **lock()** itself) and put the call to **unlock()** in a **finally{} block**.
 - This way, **unlock()** gets called no matter any exception

Condition:

- created from a lock, allows one thread to send a message to another thread
 - * create from lock instance using **lock.newCondition()**
- **await()**: release this lock and wait for the condition to be signaled.
 - When the signal happens, **await()** will automatically re-acquire the lock before returning (this means you will still have to unlock manually)
 - * you can only **await()** when you are holding the lock, and when it returns, you still have the lock, so it acts like you never unlocked it
 - * **does** throw **InterruptedException**
- **signal()**: wake up a single thread that is waiting on the condition
 - * must be holding lock to signal it's condition

- * must manually release lock before other thread will return from `await()` (because the other thread must also acquire the lock)
- * does **not** throw `InterruptedException`
- `signalAll()`: similar to `signal()` except that every thread is woken up
- * still only one thread will be able to use the lock-protected resource at a time, because locks
- * does **not** throw `InterruptedException`

Threads:

- `static void Thread.sleep(long ms)`: sleep for ms
- * throws `InterruptedException` if the thread was interrupted before time elapsed
- make new thread with `new Thread(Runnable r)`
- * start that thread with `thread.start()`

```
import java.lang.*;
import java.util.concurrent.locks.ReentrantLock;
import java.util.concurrent.locks.Condition;
public class Main2 {
    public static class Counter {
        public int count = 0;
        public ReentrantLock lock;
        public Condition updated;
        public Counter() {
            this.lock = new ReentrantLock();
            this.updated = lock.newCondition();
        }
    }

    public static class CounterThread implements Runnable {
        private Counter counter;
        public CounterThread(Counter c) {counter = c;}
        @Override
        public void run() {
            while (true) {
                try {
                    counter.lock.lock();
                    counter.count += 1;
                    System.out.println(counter.count);
                    counter.updated.signalAll();
                }
                // lock() does not throw InterruptedException
                // catch (InterruptedException e) {}
                finally {counter.lock.unlock();}

                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {}
            }
        }
    }

    public static class IntervalPrinter implements Runnable {
        private Counter counter;
        private int mod;
        private String message;
        public IntervalPrinter(Counter c, int mod, String msg) {
            counter = c;
            this.mod = mod;
            message = msg;
        }
        @Override
        public void run() {
            while (true) {
                int val = 0;
                try {
                    counter.lock.lock();
                    counter.updated.await();
                    val = counter.count;
                }
                catch (InterruptedException e) {}
                finally {counter.lock.unlock();}

                if (val % mod == 0) {
                    System.out.println(message);
                }
            }
        }
    }
}
```

```
    }
}

public static void main(String []args) {
    Counter c = new Counter();
    new Thread(new IntervalPrinter(c,3,"fizz")).start();
    new Thread(new IntervalPrinter(c,5,"buzz")).start();
    new Thread(new CounterThread(c)).start();
}
}
```

Reflection:

java.lang.Class<T>:

- allows you to reflect on class T
- `toString()` returns class declaration (more or less)
- `getSimpleName()` returns just the name part of it
 - * `Main.class.getSimpleName()` → `"Main"`
- to get:
 - * `Class<?> c = SomeClassName.class;`
 - * `Class<?> c = someObjectInstance.getClass();`
 - * `Class<?> c = Class.forName("SomeClassName");`
 - throws `ClassNotFoundException`
- `Method[] getMethods()`
 - * all public member methods, including those inherited from super-classes and implemented in interfaces
- `Method[] getDeclaredMethods()`
 - * excludes inherited methods, includes any that are declared in class regardless of public, private, static, etc...
- `Constructor<T> getConstructor(Class<?>... pt)`
 - * get a constructor for T that matches parameter types `Class<?>... pt`
 - * throws `NoSuchMethodException` if there is no constructor matching those parameter types
- `T newInstance()`
 - * create a new instance of T using the default constructor
- TODO use `Constructor` class to create class using non-default constructor
- TODO fields

java.lang.reflect.Method:

- `String toString()` → method prototype as string
 - * includes modifiers, method name, parameters, etc...
- `String getName()` → name of method as string
- `int getModifiers()` → `int` representing modifiers
 - * use `java.lang.reflect.Modifier` static methods to check:
 - `Modifier.isStatic(m.getModifiers())`
- `Class<?>[] getParameterTypes()` → types of parameters of method
 - * if no parameters, returns empty array
 - * does not include implicit `this` parameter for instance methods
- `Type[] getGenericParameterTypes()`: same, but returns a `Type` instance that accurately represents the generic info from the actual source
- `Class<?> getReturnType()`: get the return type
 - * if it's void, it returns a void type
- `Object invoke(Object obj, Object... args)`
 - * invoke a method on an object. Subject to virtual method lookup
 - * if the method is static, `obj` may be null
 - * if the method returns a primitive type, it is wrapped; if void, returns null
 - * throws `IllegalAccessException` if you can't run that method because it's private or something
 - * if target method throws, it throws `InvocationTargetException` wrapping whatever was thrown