The list game
The 5 basic rules of the game
Recursive list functions
Higher order functions
Recursion

- Recall that recursion is a problem solving technique whereby a problem is expressed in terms of one or more smaller versions of itself.
- Continuing this process produces one or more simple problems that do not require recursion for their solution.
- Such simple problems are called **base cases**.
- Reaching a base case is necessary in order to stop the recursion.
List notation

- We can use the following notation to define a list of \( n \) elements (nodes):

\[
[e_1, e_2, \ldots, e_n]
\]

- Here \( e_k \) represents both the list element \( k \) (node \( k \)) and the data stored in it.

- The empty list can be denoted by

\[
[ ]
\]
Recursively defined list

We can define a non-empty list \([e_1, e_2, \ldots, e_n]\) by breaking it into two parts:

- head: the data part \(e_1\) of the first node
- tail: the list \([e_2, \ldots, e_n]\) of the remaining nodes

Since the tail is a smaller list (one less node) this is a recursive definition of a list.

The base case is the empty list \([\ ]\).

For the empty list neither the head nor the tail is defined.
The 5 rules for lists (1)

- **empty()** returns the empty list denoted by

  \[
  \begin{array}{c}
  [ ] \leftarrow \text{empty}()
  \end{array}
  \]

- **cons(head, tail)** returns a new list constructed from the given head and tail, denoted by

  \[
  \begin{array}{c}
  \text{list} \leftarrow \text{cons}(\text{head}, \text{tail})
  \end{array}
  \]

- All lists must be constructed using these two rules
The 5 rules for lists (2)

- head(list) returns the head (data part of first node) of a list, denoted by
  \[ \text{head} \leftarrow \text{head}(\text{list}) \]

- tail(list) returns the tail (another list) of the given list denoted by
  \[ \text{tail} \leftarrow \text{tail}(\text{list}) \]

- These two rules are used to deconstruct a list
The 5 rules for lists (3)

- The final rule lets us test for an empty list
- \( \text{isEmpty}(\text{list}) \) returns true if list is empty and false if list is not empty, and is denoted by

\[
\text{boolean} \leftarrow \text{isEmpty}(\text{list})
\]
Constructing lists (1)

- A one element list can be constructed as follows

\[ [e_1] \leftarrow \text{cons}(e_1, \text{empty}( )) \]

- A two element list can be constructed as follows

\[ [e_1, e_2] \leftarrow \text{cons}(e_1, \text{cons}(e_2, \text{empty}( ))) \]
Constructing lists (2)

In general any \( n+1 \) element list can be constructed in terms of an \( n \) element list as follows

\[
[e, e_1, e_2, \ldots, e_n] \leftarrow \text{cons}(e, [e_1, e_2, \ldots, e_n])
\]
Playing the list game

- We want to write a library of list processing functions using only the 5 rules of the game.
- This means that lists can only be constructed using the *empty* and *cons* functions.
- Since lists are defined recursively then any list function must also be recursive.
- Our first two basic library functions are the *length* function and the *append* function.
The length function

The length of a list can be defined recursively:

- 0 for the empty list
- 1 + length of tail for a non-empty list

Pseudo-code algorithm is

```
FUNCTION length(list) RETURNS Integer
    IF isEmpty(list) THEN
        RETURN 0
    ELSE
        RETURN 1 + length(tail(list))
    END IF
END
```
Inductive proof

- Length of empty list is 0 (base case)
- Assume the length of an $n$ element list is $n$ and show that length of an $n + 1$ element list is $n + 1$

\[
\text{length}([e, e_1, e_2, \ldots, e_n])
\]
\[
\rightarrow 1 + \text{length}(\text{tail}([e, e_1, e_2, \ldots, e_n]))
\]
\[
\rightarrow 1 + \text{length}([e_1, e_2, \ldots, e_n])
\]
\[
\rightarrow 1 + n
\]
Append two lists (1)

- We are given the two lists

\[
list1 \leftarrow [e_1, e_2, \ldots, e_m]
\]

\[
list2 \leftarrow [f_1, f_2, \ldots, f_n]
\]

- The \textit{append} function should produce the list

\[
[e_1, e_2, \ldots, e_m, f_1, f_2, \ldots, f_n]
\]

- Append is often referred to as concatenation
Append function (1)

There are three cases to consider

- list1 is empty: then the result is list2 (base case)
- list2 is empty: then the result is list1 (base case)
- Neither list1 nor list2 is empty (recursive case)
FUNCTION append(list1, list2) RETURNS a list
IF isEmpty(list1) THEN
    RETURN list2
ELSE IF isEmpty(list2) THEN
    RETURN list1
ELSE
    RETURN ?
END IF
END FUNCTION

think of a smaller version of append
Append function (3)

- We need a smaller version of the append operation for \textit{list1} and \textit{list2}.
- The smaller version is to append the tail of \textit{list1} and \textit{list2}.
  - \texttt{append(tail(list1), list2)}
- This is almost the answer. It is missing only the head of \textit{list1} so we can just use \texttt{cons} to put the head of \textit{list1} at the front of this list.
  - \texttt{cons(head(list1), append(tail(list1), list2))}
FUNCTION `append(list1, list2)` RETURNS a list

IF `isEmpty(list1)` THEN
    RETURN `list2`
ELSE IF `isEmpty(list2)` THEN
    RETURN `list1`
ELSE
    RETURN `cons(head(list1), append(tail(list1), list2))`
END IF
END FUNCTION
Trace of the append function

\[ \text{list1} \leftarrow [e_1, e_2] \quad \text{list2} \leftarrow [f_1, f_2] \]

\[ \text{append(list1, list2)} \]

\[ \rightarrow \text{cons(head(list1), append(tail(list1), list2))} \]

\[ \rightarrow \text{cons(e_1, append([e_2], list2))} \]

\[ \rightarrow \text{cons(e_1, cons(head([e_2]), append(tail([e_2]), list2)))} \]

\[ \rightarrow \text{cons(e_1, cons(e_2, append([], list2)))} \]

\[ \rightarrow \text{cons(e_1, cons(e_2, list2))} \]

\[ \rightarrow \text{cons(e_1, [e_2, f_1, f_2])} \quad \rightarrow \quad [e_1, e_2, f_1, f_2] \]
Insertion sort (1)

We want to write a recursive version of the insertion sort algorithm:

```
FUNCTION insertionSort(list) RETURNS a list
    ...
END FUNCTION
```

A smaller version of `insertionSort` is to apply it to the tail of the list.
A smaller version of $\text{insertionSort}(\text{list})$ is to apply it to the tail of the list: $\text{insertionSort}(\text{tail}(\text{list}))$

If we can solve this problem then the tail of the list is sorted.

This reduces the original problem to finding out where the head $e'$ of the list should go:

```plaintext
FUNCTION \text{insertInSortedList}(e', \text{list}) \text{ RETURNS a list}
... 
END FUNCTION
```
Insertion sort (3)

We can now express the `insertionSort` function in terms of the `insertInSortedList` function as follows:

```
FUNCTION insertionSort(list) RETURNS a list
  IF isEmpty(list) THEN
    RETURN list
  ELSE
    RETURN insertInSortedList(head(list), insertionSort(tail(list)));
  END IF
END FUNCTION
```
Insertion sort (4)

There are three cases for placing an element in the correct position in a sorted list:

1. If the list is empty then the result is the one element list:
   - `cons(e, empty())`

2. If `e` is smaller than the head the result is:
   - `cons(e, list)`

3. Otherwise `e` must go in the tail (recursive case):
   - `cons(head(list), insertInSortedList(e, tail(list)))`
FUNCTION insertInSortedList(e, list) RETURNS a list
    IF isEmpty(list) THEN
        RETURN cons(e, empty())
    ELSE IF e < head(list) THEN
        RETURN cons(e, list)
    ELSE
        RETURN cons(head(list), insertInSortedList(e, tail(list)))
    END FUNCTION
trace of insertInSortedList (1)

We suppose that we have some list elements (nodes) 
$e_1, e_2, \ldots, e_n$ whose data order is 

$e_1 < e_2 < \ldots < e_n$

We show that 

\[
\text{insertInSortedList}(e_3, [e_1, e_2, e_4, e_5]) 
\rightarrow [e_1, e_2, e_3, e_4, e_5]
\]

To save space let insert be shorthand for insertInSortedList
trace of insertInSortedList (2)

\[
\text{insert}(e_3, [e_1, e_2, e_4, e_5])
\rightarrow \text{cons}(\text{head}([e_1, e_2, e_4, e_5]), \text{insert}(e_3, \text{tail}([e_1, e_2, e_4, e_5])))
\rightarrow \text{cons}(e_1, \text{insert}(e_3, [e_2, e_4, e_5]))
\rightarrow \text{cons}(e_1, \text{cons}(\text{head}([e_2, e_4, e_5]), \text{insert}(e_3, \text{tail}([e_2, e_4, e_5])))
\rightarrow \text{cons}(e_1, \text{cons}(e_2, \text{insert}(e_3, [e_4, e_5])))
\rightarrow \text{cons}(e_1, \text{cons}(e_2, \text{cons}(e_3, [e_4, e_5])))
\rightarrow \text{cons}(e_1, \text{cons}(e_2, [e_3, e_4, e_5]))
\rightarrow \text{cons}(e_1, [e_2, e_3, e_4, e_5])
\rightarrow [e_1, e_2, e_3, e_4, e_5]
\]
trace of insertionSort

\[
\begin{align*}
\text{sort}([e_3, e_2, e_1]) & \\
\rightarrow \text{insert}(\text{head}([e_3, e_2, e_1]), \text{sort}(\text{tail}([e_3, e_2, e_1]))) & \\
\rightarrow \text{insert}(e_3, \text{sort}([e_2, e_1])) & \\
\rightarrow \text{insert}(e_3, \text{insert}(\text{head}([e_2, e_1]), \text{sort}(\text{tail}([e_2, e_1]))) & \\
\rightarrow \text{insert}(e_3, \text{insert}(e_2, \text{sort}([e_1]))) & \\
\rightarrow \text{insert}(e_3, \text{insert}(e_2, \text{insert}(\text{head}([e_1]), \text{sort}(\text{tail}([e_1]))) & \\
\rightarrow \text{insert}(e_3, \text{insert}(e_2, \text{insert}(e_1', \text{empty}()))) & \\
\rightarrow \text{insert}(e_3, \text{insert}(e_2, \text{cons}(e_1, \text{empty}()))) & \\
\rightarrow \text{insert}(e_3, \text{insert}(e_2, [e_1])) & \\
\rightarrow \text{insert}(e_3, [e_1, e_2]) & \\
\rightarrow [e_1, e_2, e_3]
\end{align*}
\]
List game rules in Java

- If we use instance methods then the list operations have the syntax
  - `list.head()`, `tail.cons(head)`, `list.tail()`, `list.isEmpty()`

- To make the operations more like functions (functional programming) we use static methods so the operations have the syntax
  - `head(list)`, `tail(list)`, `isEmpty(list)`, `cons(head,tail)`
  - In Java `null` is used for the `empty` function (create an empty list)
This class represents a non-empty list as a head and a tail and defines the static methods for the list operations. The empty list is represented by null.

```java
package listGameStatic;
public class List<E> {
    private E head;
    private List<E> next;

    private List(E head, List<E> next) {
        this.head = head;
        this.next = next;
    }
}
```

constructor is private so that cons is the only way to construct a non-empty list
**List class (2)**

Constructing a list with cons. *This is the only place where the private constructor is used.*

```java
public static <E> List<E> cons(E head, List<E> tail)
{
    return new List<E>(head, tail);
}
```
Breaking a list into its head and tail

```java
public static <E> E head(List<E> list)
{  if (list == null)
    throw new IllegalArgumentException("...");
   return list.head;
}

public static <E> List<E> tail(List<E> list)
{  if (list == null)
    throw new IllegalArgumentException("...");
   return list.next;
}
```
List class (4)

testing a list to determine if its is empty.

```java
public static <E> boolean isEmpty(List<E> list) {
    return list == null;
}
```
The list functions can be used to write a recursive toString method for displaying a list in the form List[a,b,c,…]

```java
public String toString()
{
    return "List[" + toString(this) + "]";
}

private String toString(List<E> list)
{
    if (isEmpty(list))
        return "";
    else if (isEmpty(tail(list)))
        return head(list) + "";
    else
        return head(list) + "," +
               toString(tail(list));
}
```

No comma after last element
The Book class is a good example to use to illustrate the list game. Its specification is

```java
public class Book implements Comparable<Book>
{
    public Book(String title, String author,
                 double price, int inStock) {...}

    public String getAuthor() {...}
    public String getTitle() {...}
    public double getPrice() {...}
    public int getInStock() {...}
    public String toString() {...}
    public int compareTo(Book b) {...}
    public boolean equals(Object obj) {...}
}
```
If \(b_1, b_2, b_3\) are Book objects then the list \([b_1, b_2, b_3]\) of these three books can be constructed using

\[
\text{List<Book> list = List.cons(b1, List.cons(b2, List.cons(b3, null)))};
\]

The list \([1, 2, 3]\) of three integers can be constructed using

\[
\text{List<Integer> list = List.cons(1, List.cons(2, List.cons(3, null)))};
\]
Using the new static import we can leave out the class name in front of each of the static list operations. This can be done using

```java
import static listGameStatic.List.*;
```

or explicitly (more readable) as

```java
import static listGameStatic.List.cons;
import static listGameStatic.List.head;
import static listGameStatic.List.tail;
import static listGameStatic.List.isEmpty;
```

Then a list of books can be defined more compactly as

```java
List<Book> list =
    cons(b1, cons(b2, cons(b3, null)));
```
We can now write list functions and place them in a class called ListLibrary as static methods. Also use a static import statement.

```java
class ListLibrary {
    // place static methods here for each recursive list function
}
```
Length function

Length of a non-empty list is one more than the length of its tail. Put this function in the ListLibrary class.

```java
public static <E> int length(List<E> list) {
    if (isEmpty(list))
        return 0; // base case
    else
        return 1 + length(tail(list));
}
```
To append two non-empty lists construct a list whose head is the head of the first list and whose tail is what you get by appending the tail of the first list and the second list.

Put this function in the ListLibrary class

```java
public static <E> List<E> append(
    List<E> list1, List<E> list2)
{
    if (isEmpty(list1))
        return list2;
    if (isEmpty(list2))
        return list1;
    return cons(head(list1),
                 append(tail(list1), list2));
}
```
To reverse a list we need to reverse the tail and then append the head (as a one element list) to the reversed tail. Put this function in the ListLibrary class.

```java
def public static <E> List<E> reverse(List<E> list) {
    if (isEmpty(list))
        return null;
    else
        return append(reverse(tail(list)),
                        cons(head(list), null));
}
```
Here the generic type is `<E extends Comparable<E>>` since we assume an ordering given by the `compareTo` method.

**NOTE:** `extends` also means implements in the type parameter

Put this function in the `ListLibrary` class.

```java
public static <E extends Comparable<E>> List<E> insertInSortedList(E data, List<E> list) {
    if (isEmpty(list))
        return cons(data, null);
    else if (data.compareTo(head(list)) < 0)
        return cons(data, list);
    else // recursive case
        return cons(head(list),
                    insertInSortedList(data, tail(list)));
}
```
Here the generic type is `<E extends Comparable<E>>` since we assume an ordering given by the `compareTo` method. Put this function in the `ListLibrary` class.

```java
public static <E extends Comparable<E>>
    List<E> insertionSort(List<E> list)
{
    if (isEmpty(list))
        return list;
    return insertInInSortedList(head(list),
        insertionSort(tail(list)));
}
```
Testing the list functions

Make a class called ListGameTester to test the methods. It can have the following structure.

```java
package listGameStatic;
import static listGameStatic.List.*;
import static listGameStatic.ListLibrary.*;
public class ListGameTester
{
    public void doTest01(String title)
    {
        //...
    }
    // ... other doTest methods
    public static void main(String[] args)
    {
        ListGameTester tester = new ListGameTester();
        tester.doTest01("Test the cons function");
        //...
    }
}
```
Repeat function

Repeat a list \( n \) times.
Example: If list is \([1,2,3]\) then the result of \( \text{repeat}(list,3) \) is \([1,2,3,1,2,3,1,2,3]\)

public static <E> List<E> repeat(List<E> list, int n) {
    if (n <= 0)
        return null;
    else
        return append(repeat(list,n-1), list);
}
find function

Find the element in a list with given data. Return true if the element was found else return false.

```java
public static <E>
boolean find(E data, List<E> list)
{
    if (isEmpty(list))
        return false;
    else
        return data.equals(head(list))
            || find(data, tail(list));
}
```
first function

**Extract the first n elements of a list and return as a new list**

Example: If the list is [1,2,3,4] and n = 2 the result is [1,2].

If list is empty or n <= 0 return an empty list (null)

```java
public static <E> List<E> first(int n, List<E> list)
{
    // do it yourself
}
```
last function

Extract the last $n$ elements of a list and return as a new list
Example: If the list is $[1,2,3,4]$ and $n = 2$ the result is $[3,4]$. If list is empty or $n \leq 0$ return an empty list (null)

```java
public static <E> List<E> last(int n, List<E> list) {
    // do it yourself
}
```
merge function

merge two sorted lists and return the result
Example: If list1=[1,2,4,8] and list2 = [1,3,5,9] the result is [1,1,2,3,4,5,8,9]

```java
public static <E extends Comparable<E>> List<E> merge(List<E> list1, List<E> list2)
{
    // do it yourself
}
```
mergeSort function

merge, length, first and last can be used to do mergeSort

```java
public static <E extends Comparable<E>> List<E> mergeSort(List<E> list)
{
    int len = length(list);
    if (len < 2)
        return list; // already sorted
    else
    {
        int m = len / 2;
        int n = len - m;
        return merge(mergeSort(first(m, list)),
                     mergeSort(last(n,list)));
    }
}
```
equal function

return true if two lists are equal else return false. The base case is that two empty lists are equal.

public static <E> boolean equal(List<E> list1, List<E> list2)
{
    // do it yourself using equals method
    // defined in class E
}
isPalindrome function

return true if a list reads the same forwards or backwards. For example the list [1,2,3,4,3,2,1] is a palindrome

```java
public static <E>
    boolean isPalindrome(List<E> list)
{
    // do it yourself
}
```
return true if a list reads the same in either direction
For example the list [1,2,3,4,3,2,1] is a palindrome

```java
public static <E extends Comparable<E>>
    E findMin(List<E> list)
{
    // do it yourself
}
```

The following auxiliary function that computes the minimum of two elements of type E is useful

```java
public static <E extends Comparable<E>>
    E min2(E data1, E data2)
{
    return (data1.compareTo(data2) < 0) ? data1 : data2;
}
```


Remove first element of a list whose data part is given. Return the resulting list

```java
public static <E> List<E> remove(E data, List <E> list)
{
    // do it yourself
}
```
Apply selection sort to the given list and return the sorted list. Hint: Use findMin to find the smallest element in list. The recursive case is to apply selectionSort to the list obtained by removing this smallest element and putting the smallest element on the front of this list.

```java
public static <E extends Comparable<E>> List<E> selectionSort(List<E> list)
{
    // do it yourself
}
```
makeList function (1)

A private helper method to recursively make array elements data[a], data[a+1], ..., data[b] into a list and return the list

```java
private static <E>
    List<E> makeList(E[] data, int a, int b)
{
    if (a == b)
        return cons(data[a], null);
    else
        return cons(data[a], makeList(data, a+1, b));
}
```
Now use the helper method and the variable number of arguments feature provided by Java 5 to write a method that makes a list from the arguments to makeList

```java
public static <E> List<E> makeList(E ... data)
{
    return makeList(data, 0, data.length - 1);
}
```

This method can also be called using an array argument of type E[] data
The following statement makes a list of integers [2,4,6,8,10]. It is equivalent to:

```
List<Integer> list = cons(2, cons(4, cons(6, cons(8, null))));
```

```
List<Integer> list1 = makeList(2,4,6,8,10);
```

The following version creates the same list using an array:

```
Integer[] intArray = new Integer[]{2,4,6,8,10};
...
List<Integer> list2 = makeList(intArray);
```
Delete books in stock (1)

- Recall that the Book class has data fields for title, author, price and number in stock.
- There are get methods for each field.
- In particular there is a getInStock() method that returns the number of books in stock.
- Write a recursive function deleteNotInStock that takes a list of books and produces a list that has the books not in stock removed.
- Base case is the empty list (result is empty list).
Here is an example for a list of 6 books. Three books have an inStock value of 0

Book b1 = new Book(...,0); // only inStock shown
Book b2 = new Book(...,5);
Book b3 = new Book(...,0);
Book b4 = new Book(...,10);
Book b5 = new Book(...,0);
Book b6 = new Book(...,5);

List<Book> list = cons(b1,cons(b2, cons(b3, cons(b4, cons(b5, cons(b6, null))))));

// continued next slide
Delete books in stock (3)

The resulting list is [b2,b4,b6]

List<Book> listInStock =
    deleteNotInStock(list);
System.out.println("list = " + list);
System.out.println("In stock = " +
    listInStock);
Complete the following method. The resulting list should be a list of books with those not in stock removed.

```java
public static List<Book> deleteNotInStock(List<Book> list) {
    if (isEmpty(list))
        return list;

    // do the rest yourself
    // hint: there are two recursive cases
```
COSC 2006: Data Structures I

The list game

Lists of ordered pairs
Lists of ordered pairs

- We have written several recursive functions that use the 5 rules of the list game.
- Each of these functions worked with a single type of list.
- We can also write functions that convert lists of one type to lists of another type.
The distribute function

This function takes a list and produces a related list of ordered pairs.

For example consider the list \([e_1, e_2, \ldots, e_n]\) of elements taken from some set and consider another set \(\{a, b, c, \ldots\}\).

Then the function \(\text{dist}\) is defined as follows:

\[
\text{dist}(a, [e_1, e_2, \ldots, e_n]) = [(a, e_1), (a, e_2), \ldots, (a, e_n)]
\]

We say that \(\text{dist}\) distributes an element of the set \(\{a, b, c, \ldots\}\) over a list of the form \([e_1, e_2, \ldots, e_m]\) to produce a list of ordered pairs.
Recursive definition

- If the list is empty then we define
  \[ \text{dist}(a, [ ]) = [ ] \]

- If the list is not empty then we define

\[
\text{dist}(a, \text{list}) \\
\rightarrow \text{cons}( (a, \text{head(\text{list}) } ), \text{dist}( a, \text{tail(\text{list}) } ))
\]

ordered pair
trace of the dist function

\[ \text{dist}(a, [e_1, e_2, e_3]) \]

\[ \rightarrow \text{cons}((a, \text{head}([e_1, e_2, e_3])), \text{dist}(a, \text{tail}([e_1, e_2, e_3]))) \]

\[ \rightarrow \text{cons}((a, e_1), \text{dist}(a, [e_2, e_3])) \]

\[ \rightarrow \text{cons}((a, e_1), \text{cons}((a, e_2), \text{dist}(a, \text{tail}([e_2, e_3]))) \]

\[ \rightarrow \text{cons}((a, e_1), \text{cons}((a, e_2), \text{dist}(a, [e_3]))) \]

\[ \rightarrow \text{cons}((a, e_1), \text{cons}((a, e_2), \text{cons}((a, e_3), \text{dist}(a, [])))) \]

\[ \rightarrow \text{cons}((a, e_1), \text{cons}((a, e_2), \text{cons}((a, e_3), []))) \]

\[ \rightarrow \text{cons}((a, e_1), \text{cons}((a, e_2), [(a, e_3)])) \]

\[ \rightarrow \text{cons}((a, e_1), [(a, e_2), (a, e_3)]) \]

\[ \rightarrow [(a, e_1), (a, e_2), (a, e_3)] \]
First we need a generic class to represent ordered pairs

```java
package listGameStatic;
public class Pair<X,Y> {
    private X first; private Y second;

    public Pair(X first, Y second) {
        this.first = first;
        this.second = second;
    }
    public X first() { return first; }
    public Y second() { return second; }
    public String toString() {
        return "(" + first + "," + second + ")";
    }
}
```
Now we can use Pair to obtain the following definition of dist:

```java
public static <E,F> List<Pair<E,F>> dist (E data, List<F> list)
{
    if (isEmpty(list))
    {
        return null;
    }
    return cons( new Pair<E,F>(data,head(list)),
                 dist(data, tail(list)));
}
```
Test the dist function by making a list of integers and letting the set \{a,b,c,...\} be the set of characters represented by the Character class. Then distribute the letter 'a' over the list

```java
public void doTest()
{
    List<Integer> list = makeList(1,2,3,4,5);
    List<Pair<Character,Integer>> d =
        dist('a',list);
    System.out.println(d);
}
```

The result is the list of ordered pairs
\[
\begin{array}{l}
    (a,1), (a, 2), (a, 3), (a, 4), (a, 5) \\
\end{array}
\]
The zip function

- The zip function takes two lists and zips them up as a list of ordered pairs.

\[
zip([e_1, e_2, \ldots, e_m], [f_1, f_2, \ldots, f_n])
\rightarrow [(e_1, f_1), (e_2, f_2), \ldots, (e_k, f_k)] \quad k = \min(m, n)
\]

- We stop zipping when the shortest list is finished.
Recursive definition of zip

- If either list is empty then we define
  \[ \text{zip} \ (\text{list}_1, \text{list}_2) = [] \]

- Otherwise we define
  \[
  \text{zip}(\text{list}_1, \text{list}_2) \\
  \rightarrow \text{cons}( \ (\text{head}(\text{list}_1), \text{head}(\text{list}_2)) , \\
  \text{zip}(\text{tail}(\text{list}_1), \text{tail}(\text{list}_2)))
  \]
Now we can use Pair to obtain the following definition of zip:

```java
public static <E,F> List<Pair<E,F>> zip (List<E> list1, List<F> list2)
{
    if (isEmpty(list1) || isEmpty(list2))
        return null;
    return
    cons( new Pair<E,F>(head(list1),head(list2)),
            zip(tail(list1), tail(list2)))
}
Test the zip function

As an example use a list of characters and a list of integers

List<Character> list1
    = makeList('a','b','c','d');
List<Integer> list2
    = makeList(1,2,3,4,5);

List<Pair<Character,Integer>> z =
    zip(list1, list2);
System.out.println(z); 

The output is 
[ (a,1), (b,2), (c,3), (d,4) ]
The list game

Higher order functions
Higher order list functions

- We can also write higher order list functions.
- They apply functions to the element of a list.
- Examples:
  - **map**: apply a function to the elements of a list.
  - **filter**: remove elements from a list that satisfy some boolean valued condition.
  - **reduce**: reduce a list to a scalar value by applying a binary function (standard example is adding elements of a list to get the sum).
The map function (1)

Consider lists of type $X$ and lists of type $Y$

Let $lists(X)$ represent all lists of type $X$

Let $lists(Y)$ represent all lists of type $Y$

Then map maps lists of type $X$ to lists of type $Y$

$\map: lists(X) \rightarrow lists(Y)$

This is done by applying a function $f: X \rightarrow Y$ to the elements of the list of type $X$ to get a list of type $Y$:

$\map(f, [a,b,c,...]) = [f(a), f(b), f(c), ...]$
The map function (2)

- *map* is called a higher level function because it takes a function as one of its arguments.

**Example:**

- \( X = \{\text{integers in range 0 to 127}\}, \ Y = \{\text{characters}\} \)

- Let \( f = \text{asciiToChar} \) convert ascii codes to their corresponding characters: \( \text{asciiToChar}: X \to Y \)

- \( \text{map}(f, [65, 66, 67, 68]) \)
  \[ = [f(65), f(66), f(67), f(68)] \]
  \[ = ['A', 'B', 'C', 'D'] \]
Interfaces can be used to provide function arguments in Java.

The following interface applies to all functions that take a single double argument and return a double argument.

```java
public interface DoubleFunction
{
    public double eval (double x);
}
```
Java function arguments (2)

- Here is an example of a class that implements the interface.

```java
public class Square implements DoubleFunction
{
    public double eval (double x)
    {
        return x * x;
    }
}
```

- An object of this class has an eval method that represents the squaring function
Suppose we want to write a method that takes a DoubleFunction as argument and produces a simple table of values of the function:

```java
class TableRunner {
    public static void table(DoubleFunction f, double a, double b, double stepSize) {
        for (double x = a; x <= b; x += stepSize) {
            System.out.println(x + " " + f.eval(x));
        }
    }
}
```
It is now easy to call this table function using any class that implements the DoubleFunction interface. For example using the Square class we could call the table method as follows.

```java
table(new Square(), 0.0, 1.0, 0.1);
```

If we had a similar class called Cube we could call it using

```java
table(new Cube(), 0.0, 1.0, 0.1);
```
The map function (3)

We can now return to the map function. We can use generic types to define our interface for functions from type X to type Y:

```java
package higherOrderFunctions;

public interface Function<X,Y> {
    public Y eval (X x);
}
```

Here eval takes an argument of type X and returns a value of type Y.
The map function (4)

We can now write the map function: construct list obtained by applying \( f \) to the head and the list you get by applying map to the tail.

```java
package higherOrderFunctions;
// put this in class ListLibrary2
public static <X,Y> List<Y> map(Function<X,Y> f, List<X> list)
{
    if (isEmpty(list))
        return null;
    else
        return cons(f.eval(head(list)),
                    map(f, tail(list)));
}
```
trace of the map function

\[ \text{map}(f, [a, b, c]) \]

\[ \rightarrow \text{cons}(f(a), \text{map}(f, \text{tail}([a, b, c]))) \]

\[ \rightarrow \text{cons}(f(a), \text{map}(f, [b, c])) \]

\[ \rightarrow \text{cons}(f(a), \text{cons}(f(b), \text{map}(f, [c]))) \]

\[ \rightarrow \text{cons}(f(a), \text{cons}(f(b), \text{cons}(f(c), \text{map}(f, [])))) \]

\[ \rightarrow \text{cons}(f(a), \text{cons}(f(b), \text{cons}(f(c), []))) \]

\[ \rightarrow \text{cons}(f(a), \text{cons}(f(b), [f(c)])) \]

\[ \rightarrow \text{cons}(f(a), [f(b), f(c)]) \]

\[ \rightarrow [f(a), f(b), f(c)] \]
Ascii code example (1)

First we want to convert an integer list [65,66,67,68,69,70] of ascii codes to the corresponding list of characters.

This gives the list [A,B,C,D,E,F] of characters having these ascii codes.

Then we want to zip these two lists to get the list of pairs
[ (65,A), (66,B), (67,C), (68,D), (69,E), (70,F) ]
Ascii code example (2)

First define the following function that implements the `Function<Integer, Character>` interface. For each integer it returns the character with that ascii code.

```java
package higherOrderFunctions;

public class AsciiToChar implements Function<Integer, Character>
{
    public Character eval(Integer i)
    {
        return (char) i.intValue();
    }
}
```

Autoboxing will make a `Character` object from this.
Now use map and zip to obtain the list of ordered pairs with first element given by the ascii code and second element given by the character

```java
List<Integer> codes
    = makeList(65, 66, 67, 68, 69, 70);
List<Character> chars
    = map(new AsciiToChar(), codes);
List<Pair<Integer,Character>> pairs
    = zip(codes,chars);

Result is the list
[ (65,A), (66,B), (67,C), (68,D), (69,E), (70,F) ]
```
The filter function (1)

- The filter function takes a boolean valued filter function and a list as arguments.
- A new list is created containing only those elements of the given list for which the filter function is true.
- A filter function implements the interface

\[ \text{Function}\langle X, \text{Boolean}\rangle \]
The filter function (2)

- Suppose we have a list of integers [1, 2, 3, 4, 5, 6]
- We want to produce the list [2, 4, 6] of only the even integers. Then we can use the class

```java
package higherOrderFunctions;
public class Even implements Function<Integer, Boolean> {
    public Boolean eval(Integer x) {
        return x % 2 == 0;
    }
}
```

Automatic unboxing and boxing occurs here.
The filter function (3)

The filter function has the following form for lists of type X and a Boolean valued filter function f.

```java
package higherOrderFunctions;
// put this in class ListLibrary2
public static <X> List<X> filter(Function<X, Boolean> f, List<X> list) {
    if (isEmpty(list))
        return list;
    else if (f.eval(head(list)))
        return cons(head(list),
                   filter(f, tail(list)));
    else
        return filter(f, tail(list));
}
```
We want to delete all books from a list of books that have an inStock value of 0. First we write the following filter function which returns a true if the book is in stock.

```java
package higherOrderFunctions;
import listGameStatic.Book;

public class InStock implements Function<Book, Boolean> {

    public Boolean eval(Book b) {
        return (b.getInStock() != 0);
    }
}
```
We can now use the `InStock` function in the `filter` function to produce the list of books:

```java
List<Book> list = makeList(b1, b2, b3, b4, b5, b6);
List<Book> listInStock = filter(new InStock(), list);
System.out.println("list = " + list);
System.out.println("In stock = " + listInStock);
```
The reduce function

- Sometimes we want to reduce a list to a value

Example: add the elements of a list
  - \([1,2,3,5,4,8] \rightarrow 23\)

Example: multiply the elements of a list
  - \([1,2,3,2,4] \rightarrow 48\)

Here we need a function with two arguments to apply to the list since we need to reduce the tail of a list to a value and then apply function to the head and this value
  - \(f : X \times X \rightarrow Y\)
Trace of the reduce function

\[ \text{reduce}(f, [a, b, c], s) \]

\[ \rightarrow f(\text{head}([a, b, c]), \text{reduce}(f, \text{tail}([a, b, c]), s)) \]

\[ \rightarrow f(a, \text{reduce}(f, [b, c], s)) \]

\[ \rightarrow f(a, f(b, \text{reduce}(f, [c], s))) \]

\[ \rightarrow f(a, f(b, f(c, \text{reduce}(f, [], s)))) \]

\[ \rightarrow f(a, f(b, f(c, s))) \]
Substitute $\text{sum}(x, y) = x + y$ and $s = 0$

$f(a, f(b, f(c, s)))$

$\rightarrow \text{sum}(a, \text{sum}(b, \text{sum}(c, 0)))$

$\rightarrow \text{sum}(a, \text{sum}(b, c + 0))$

$\rightarrow \text{sum}(a, b + c + 0)$

$\rightarrow a + b + c + 0$
Substitute the product function

Substitute \( \text{product}(x, y) = x \times y \) and \( s = 1 \)

\[
f(a, f(b, f(c, s)))
\]

\[
\rightarrow \text{product}(a, \text{product}(b, \text{product}(c, 1)))
\]

\[
\rightarrow \text{product}(a, \text{product}(b, c \times 1))
\]

\[
\rightarrow \text{product}(a, b \times c \times 1)
\]

\[
\rightarrow a \times b \times c \times 1
\]
Here we need the following interface to define a binary function on a set $X$.

```java
package higherOrderFunctions;

public interface BinaryFunction<X,Y> {
    public Y eval (X x, X y);
}
```
Now we can define the reduce function. As arguments it requires a binary function, a list, and an initial value for the reduction (e.g., for a sum the initial value would be 0, 1 for a product)

```java
package higherOrderFunctions;
// put this in class ListLibrary2
public static <X> X reduce(BinaryFunction<X,X> f, List<X> list, X b) {
    if (isEmpty(list))
        return b;
    return f.eval( head(list), reduce(f, tail(list), b));
}
```
If we want to sum the elements of an Integer list then we need the following function

```java
package higherOrderFunctions;

public class Sum implements BinaryFunction<Integer, Integer> {
    public Integer eval(Integer x, Integer y) {
        return x + y;
    }
}
```
If we want to multiply the elements of an Integer list then we need the following function

```java
package higherOrderFunctions;

public class Product implements BinaryFunction<Integer, Integer> {
    public Integer eval(Integer x, Integer y) {
        return x * y;
    }
}
```
Now we can use the following statements to take the sum or product of the elements of a list of integers.

```java
List<Integer> list = makeList(1,2,3,4,5,6);
int sum = reduce(new Sum(), list, 0);
int product = reduce(new Product(), list, 1);
System.out.println(sum);  // result is 21
System.out.println(product); // result is 720
```
The reduce2 function (1)

It is also possible to write a 2 argument version of reduce which uses the head of a one element list as the initial value.

```java
package higherOrderFunctions;
// put this in class ListLibrary2
public static <X> X reduce2(BinaryFunction<X,X> f, List<X> list)
{
    if (isEmpty(list))
        return null; // undefined on empty list
    else if (isEmpty(tail(list)))
        return head(list);
    else
        return f.eval(head(list),
                        reduce2(f, tail(list)));
}
```
The reduce2 function (2)

Now we can use the following statements to take the sum or product of the elements of a list of integers.

```java
List<Integer> list = makeList(1,2,3,4,5,6);
int sum = reduce2(new Sum(), list);
int product = reduce2(new Product(), list);
System.out.println(sum);  // result is 21
System.out.println(product); // result is 720
```
Use box like this for comment

Put code here