# End of the list

## More List Idioms

Friday, November 3, 2006

**CS111 Computer Programming**

Department of Computer Science
Wellesley College

## Review: IntList Summation

```
public class IntListOps {
   public static int sumList (IntList list) {
      if (IntList.isEmpty(list))  { // base case
         return 0;
       } else { // recursive case
         return IntList.head(list) + sumList(IntList.tail(list));
      }
   }
}
```

### Testing sumList in DrJava:

```
➢IntListOps.sumList(IntList.fromString("[5,7,4]"))
16
```

## Abbreviating IntList: Approach 1

```
public class IntListOps {
   public static int sumList (IntList list) {
      if (isEmpty(list))  {
         return 0;  // base case
       } else { // recursive case
         return head(list) + sumList(tail(list));
      }
   }
   // Define local versions of IntList methods
   public static int head (IntList L) {return IntList.head(L);}
   // Similarly for isEmpty, tail, empty, prepend, fromList, …
}
```

Testing sumList in DrJava:

```
➢IntListOps.sumList(IntList.fromString("[5,7,4]"))
16
```



## Abbreviating IntList: Approach 2

```
public class IntListOps {
   public static IntList IL;
   public static int sumList (IntList list) {
      if (IL.isEmpty(list))  {
         return 0;  // base case
       } else { // recursive case
         return IL.head(list) + sumList(IL.tail(list));
       }
   }
}
```

Testing sumList in DrJava:

```
➢IntList IL;
➢IntListOps ILO;
➢ILO.sumList(IL.fromString("[5,7,4]"))
16
```

## Testing via the main method

```
public class IntListOps {
  public static int sumList (IntList list) {…}

  public static void main (String[] args) {
    if (args.length == 2) {
      IntList L = IntList.fromString(args[1]);
      if (args[0].equals("sumList"))
        System.out.println("sumList(" + L + ") = " + sumList(L);
      else if …
    } else …
  }
}
```

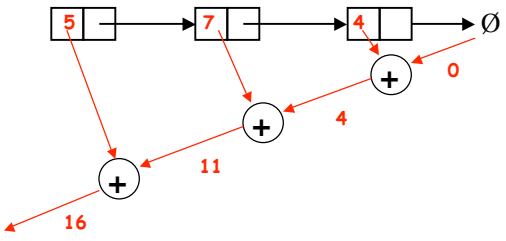
Testing sumList in DrJava:

```
➢java IntListOps sumList [5,7,4]
16
```



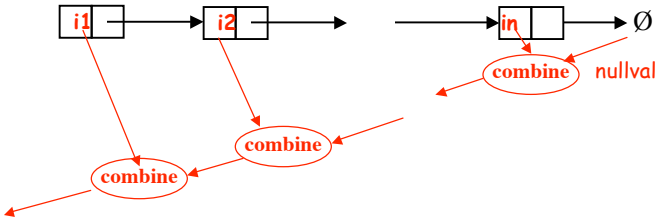
## sumList Illustrates the List Accumulation Idiom

```
// Returns the sum of the elements of an integer list
public static int sumList (IntList list) {
   if (isEmpty(list))  {
      return 0;  // base case
    } else {
      head(list) + sumList(tail(list)) // recursive case
    }
}
```



## The General List Accumulation Idiom

```
// Need to instantiate the red parts
public static int accum (IntList list) {
   if (isEmpty(list))  {
      return nullval;  // base case
    } else {
      combine(head(list),accum(tail(list)) // recursive case
    }
}
```

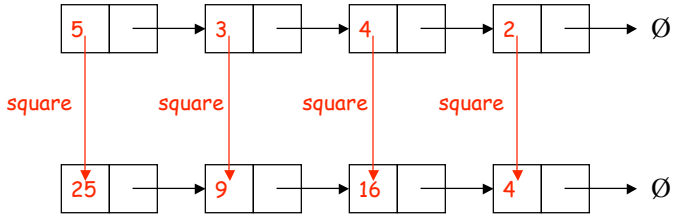





## Instantiating the List Accumulation Idiom

| accum | combine(elt,ans) | nullval |
|---|---|---|
| sumList | elt + ans | 0 |
| prodList | elt * ans | 1 |
| length | 1 + ans | 0 |
| minList | Math.min(elt,ans) | Integer.MAX_VALUE |
| maxList | Math.max(elt,ans) | Integer.MIN_VALUE |
| areAllPositive | (elt > 0) && ans | true |
| isSomePositive | (elt > 0) \|\| ans | false |
| copyList | prepend(elt,ans) | empty() |
| append | see later | see later |

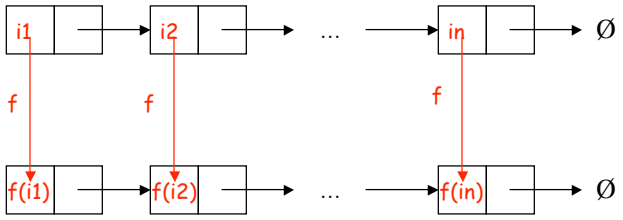
## Review: mapSquares Illustrates the List Mapping Idiom




```
// Returns a new list whose elements are the squares
// of the corresponding elements in the given list
public static IntList mapSquares (IntList list) {
   if (isEmpty(list))  { // base case
      return empty();  // could use list instead
    } else { // recursive case
      return prepend(head(list) * head(list), mapSquares(tail(list));
    }
}
```

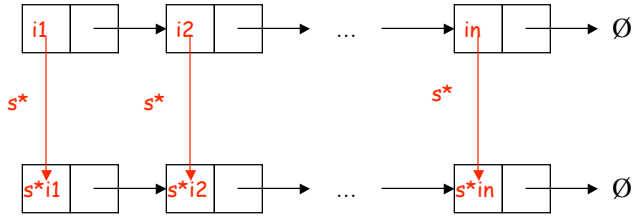


## The General Mapping Idiom




```
public static IntList mapF (IntList list) {
   if (isEmpty(list))  { // base case
      return empty();  // could use list instead
    } else { // recursive case
      return prepend(f(head(list)), mapF(tail(list));
    }
}
```

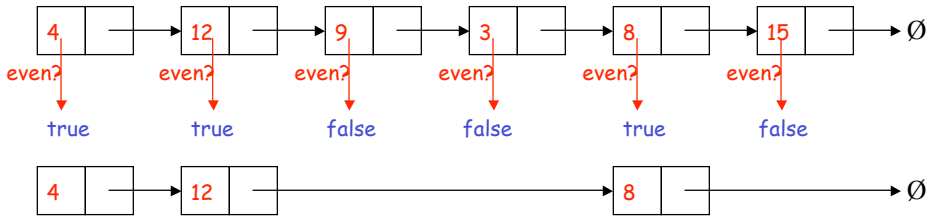
# Mapping Sometimes Needs Extra Arguments




```
public static IntList mapScale (int s, IntList list) {
   if (isEmpty(list))  { // base case
      return empty();  // could use list instead
    } else { // recursive case
      return prepend(s * head(list)), mapScale (s, tail(list));
    }
}
```

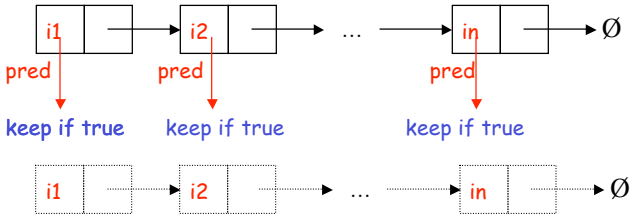


# filterEvens Illustrates the List Filtering Idiom




```
// Returns a new list whose elts are the even elts of the given list
public static IntList filterEvens (IntList list) {




}
```

## The General List Filtering Idiom




```
// Returns a new list whose elts satisfy predicate pred from given list
public static IntList filterPred (IntList list) {
   if (isEmpty(list)) {
      return empty();
   } else if (pred(head(list))) {
      return prepend(head(list), filterPred(tail(list)));
   }  else {
      return filterPred(tail(list));
   }
}
```



## Filters sometimes need extra arguments

```
// Returns a new list whose elts are divisors of n from given list
public static IntList filterDivisors(int n, IntList list) {
   if (isEmpty(list)) {
      return empty();
   } else if ((n%(head(list))) == 0) {
      return prepend(head(list), filterDivisors(n, tail(list)));
   }  else {
      return filterDivisors(n, tail(list));
   }
}
```

- IntListOps.filterDivisors(36, IntList.fromString("[2,3,4,5,6,7,8,9]"))

[2,3,4,6,9]

- IntListOps.filterDivisors(37, IntList.fromString("[2,3,4,5,6,7,8,9]"))

[]

## fromTo illustrates the List Generation Idiom

```
// Returns a list of the elements between from and to
public static IntList fromTo(int lo, int hi) {




}
```

- IntListOps.fromTo(3,7)

[3,4,5,6,7]

- IntListOps.fromTo(-4,12)

[-4,-3,-2,-1,0,1,2,3,4,5,6,7,8,9,10,11,12]



## Lists Enhance Modularity

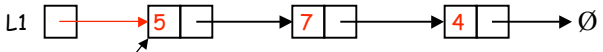
Can implement many methods in terms of standard methods that produce and consume lists. For example:

```
public static int factorial (int n) {
   return prodList(fromTo(1,n));
}

public static sumOfSquaredEvens (int lo, int hi) {
   return sumList(mapSquare(filterEven(fromTo(lo, hi))));
}
```

Note that these methods are not recursive! The recursion is hidden in the standard list-processing methods.

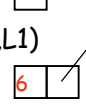
## List Glue

- IntList L1 = IntList.fromString("[5,7,4]");



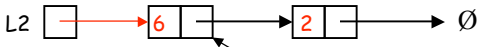

- IntList.prepend(6,L1)

[6,5,7,4]



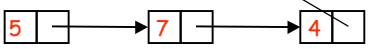

(shares nodes in list)

- IntList L2 = IntList.fromList("[6,2]");




- IntListOps.append(L1, L2)

[5,7,4,6,2]




(copies nodes in first list, but shares those in second)

- IntListOps.postpend(L1, 6)

[5,7,4,6]


(copies nodes in list)



## append(IntList list1, IntList list2)

```
// Returns a list containing the elements of the first list
// followed by the elements of the second list. The nodes of the
// first list are copied, but the nodes of the second list are shared.
public static IntList append(IntList list1, IntList list2) {




}
```

## Invocation Tree for append(L1,L2)



## postpend(IntList list, int n)

```
// Returns a list containing the elements of the list followed by n.
// The nodes of the list are copied, and a new node is made for n.
public static IntList postpend (IntList list, int n) {




}
```

## reverse (IntList list)

- IntListOps.reverse(IntList.fromString("[6,5,7,4]))

[4,7,5,6]

- IntListOps.reverse(IntList.fromString("[5,7,4]"))

[4,7,5]

- IntListOps.reverse(IntList.empty())

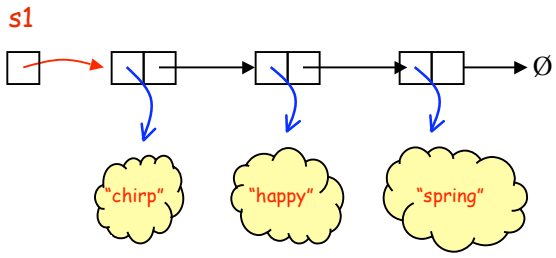
[]

```
// Returns a new list containing the elements of the given list
// in reverse order.
public static IntList reverse (IntList list) {
   if (isEmpty(list)) {

   } else {

   }
}
```



## Invocation Tree for reverse(L1)

## Lists are not just for ints






## Five core StringList methods

```
public static String head (StringList L)
```
Returns the string that is the head component of the list of strings node **L**.

```
public static StringList tail (StringList L)
```
Returns the list of strings that is the tail component of the list of strings node **L**.

```
public static boolean isEmpty (StringList L)
```
Returns **true** if **L** is an empty list of strings and **false** if **L** is a list of strings node.

```
public static StringList empty()
```
Returns an empty list of strings.

```
public static StringList prepend (String x, StringList L)
```
Returns a new list of strings node whose head is **x** and whose tail is **L**.

## Everything we did before, we can do now

```
// count the number of elements in a StringList
public static int length (StringList list) {
    if (isEmpty(list)) { // Really StringList.isEmpty
        return 0;
    } else {
        return 1 + length(tail(list)); // Really StringList.tail
    }
}
```

or

```
public static int length (StringList list) {
    return isEmpty(list)? 0 : 1 + length(tail(list));
}
```



## We can even make lists of lists!

- Sometimes we need keep more than one list at a time. Multiple lists might be organized as "lists of lists."
- List of all subsets of the set [1,2,3]:
  [ [], [1], [2], [3], [1,2], [2,3], [1,3], [1,2,3]]
- List of tails of the list [7, 1, 4]:
  [ [7, 1, 4], [1, 4], [4], [] ]

We'll say more about these in a later lecture.