More List Idioms

End of the List

Tuesday, October 30, 2007

CS111 Computer Programming

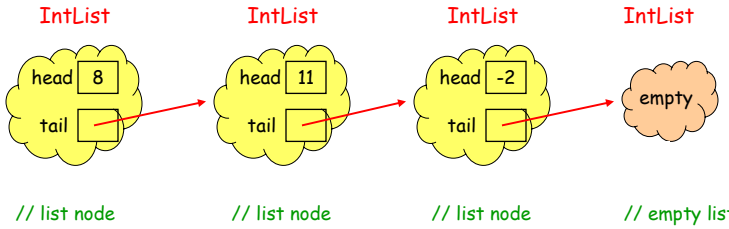
Department of Computer Science
Wellesley College

IntLists represent lists of integers

IntLists are defined recursively:

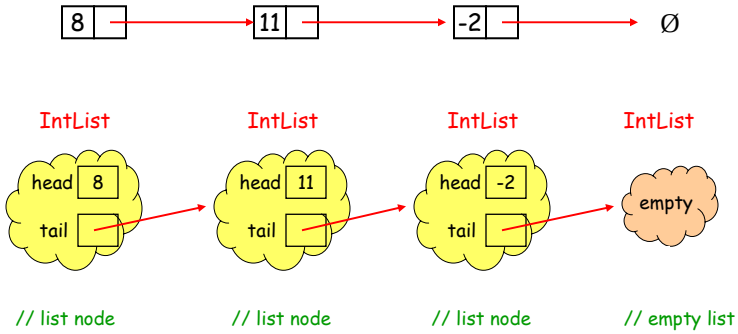
An IntList is either

- the empty list, or
- a (non-empty) list node with two parts:
  1. a head which is an integer, and
  2. a tail which is another IntList.



Box-and-pointer notation

We represent list nodes and empty lists using a shorthand known as box-and-pointer notation.



Five core IntList methods

`public static int head (IntList L)`
Returns the integer that is the head component of the integer list node **L**.
Signals an exception if **L** is empty.

`public static IntList tail (IntList L)`
Returns the integer list that is the tail component of the integer list node **L**.
Signals an exception if **L** is empty.

`public static boolean isEmpty (IntList L)`
Returns **true** if **L** is an empty integer list and **false** if **L** is an integer list node.

`public static IntList empty()`
Returns an empty integer list.

`public static IntList prepend (int n, IntList L)`
Returns a new integer list node whose head is **n** and whose tail is **L**.

Review: IntList Summation

```
public class IntListOps {

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

}
```

Testing sumList in DrJava:

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


Testing via the main method

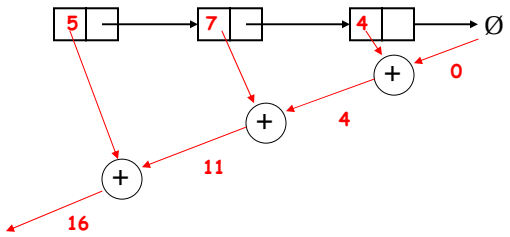
```
public class IntListOps {

  public static int sumList (IntList L) {…}

  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]
sumList([5,7,4]) = 16
```

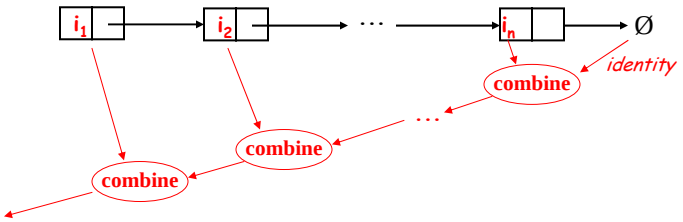
sumList Illustrates the List Accumulation Idiom

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



The General List Accumulation Idiom

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

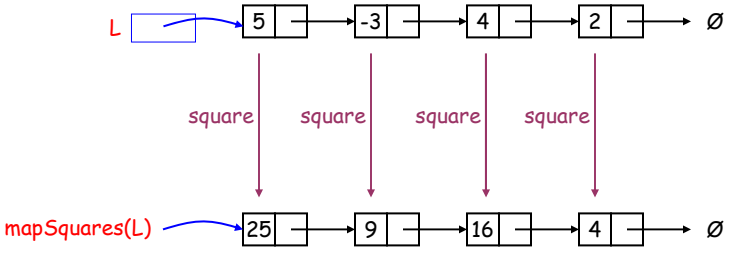


Instantiating the List Accumulation Idiom

accum	combine(elt,ans)	*identity*
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 (returns boolean)	(elt > 0) && ans	true
isSomePositive (returns boolean)	(elt > 0) || ans	false
copyList (returns IntList)	prepend(elt,ans)	empty()
append	// see later	// see later

Squaring numbers

- Method mapSquares(IntList L) returns a new list, such that each element in the list is the "square" of the corresponding element in "L".
- For example,



mapSquares(IntList L);

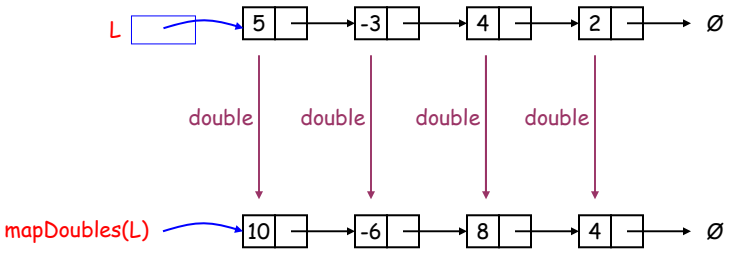
```
// Returns a new list whose elements are the sqaures of
// the corresponding elements in the given list
public static IntList mapSquares(IntList L) {

   if (                   )  {  // base case
       return
   } else {                     // recursive case
       return

   }
}
```


mapSquares(IntList L);

```
// Returns a new list whose elements are the sqaures of
// the corresponding elements in the given list
public static IntList mapSquares(IntList L) {

   if ( InList.isEmpty(L) )  {  // base case
       return empty();
   } else {                     // recursive case
       return IntList.prepend(IntList.head(L)*IntList.head(L),
                              mapSquares(IntList.tail(L)));
   }
}
```

Doubling numbers

- Method mapDoubles(IntList L) returns a new list, such that each element in the list is "twice" the corresponding element in "L".
- For example,

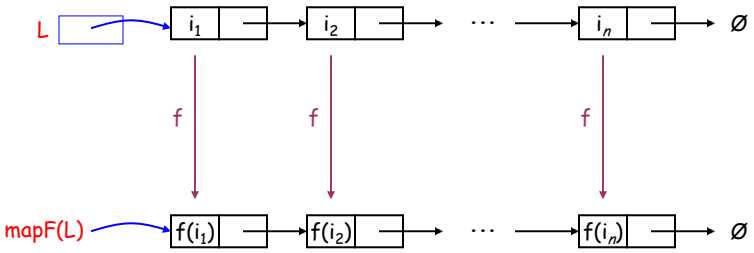


mapDoubles(IntList L);

```
// Returns a new list whose elements are twice
// the corresponding elements in the given list
public static IntList mapDoubles(IntList L) {

   if ( InList.isEmpty(L) )  {  // base case
       return empty();
   } else {                     // recursive case
       return IntList.prepend(2 * IntList.head(L),
                              mapDoubles(IntList.tail(L)));
   }
}
```

The General Mapping Idiom

- Method mapF(IntList L) returns a new list, such that each element in the list is the result of applying function *f* to the corresponding element in "L".
- For example,

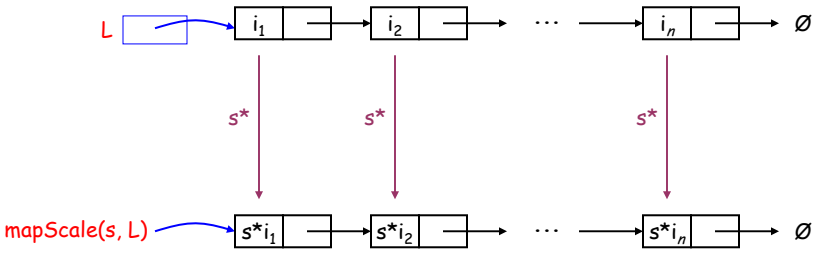


mapF(IntList L):

```
// Returns a new list whose elements are the result of applying
// function f to the corresponding elements in the given list
public static IntList mapF(IntList L) {

   if ( InList.isEmpty(L) )  {  // base case
       return empty();
   } else {                     // recursive case
       return IntList.prepend( f( IntList.head(L) ),
                               mapF(IntList.tail(L)));
   }
}
```

Mapping sometimes needs extra arguments

- Method mapScale(int s, IntList L) returns a new list, such that each element in the list is the result of multiplying *s* by the corresponding element in "L".
- For example,

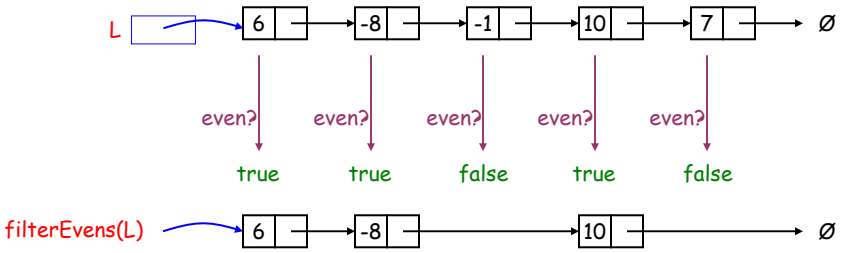


mapScale(int s, IntList L);

```
// Returns a new list whose elements are the result of applying
// function f to the corresponding elements in the given list
public static IntList mapScale(int s, IntList L) {

   if ( InList.isEmpty(L) )  {  // base case
       return empty();
   } else {                     // recursive case
       return IntList.prepend( s * IntList.head(L),
                               mapScale(s, IntList.tail(L)));
   }
}
```

Filtering even numbers

- Method filterEvens(IntList L) returns a new list containing elements of "L" that are even (i.e., evenly divisible by 2).
- For example,



filterEvens(IntList L);

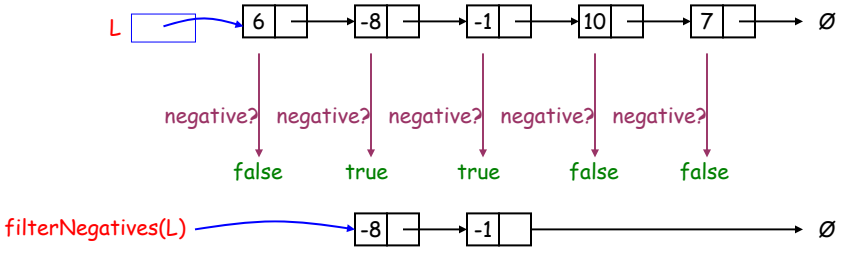
```
// Returns sublist containing elements of L that are even
public static IntList filterEvens(IntList L) {

   if (                        )  {              // base case
       return
   } else if (                                ) {  // recursive case
       return

   } else {
       return
  }
}
```

filterEvens(IntList L);

```
// Returns sublist containing elements of L that are even
public static IntList filterEvens(IntList L) {

   if (   IntList.isEmpty(L)   )  {              // base case
       return IntList.empty();
   } else if (  (IntList.head(L) % 2) == 0   ) {  // recursive case
       return IntList.prepend(IntList.head(L),
                              filterEvens(IntList.tail(L)));
   } else {
       return filterEvens(IntList.tail(L));
  }
}
```


Filtering negative numbers

- Method filterNegatives(IntList L) returns a new list containing elements of "L" that are less than zero.
- For example,

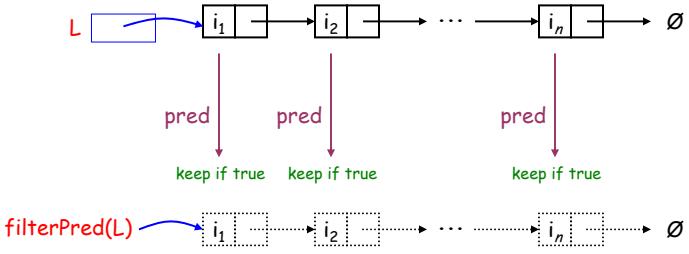


filterNegatives(IntList L);

```
// Returns sublist containing elements of L that are negative
public static IntList filterNegatives(IntList L) {

   if (   IntList.isEmpty(L)   )  {              // base case
       return IntList.empty();
   } else if (  IntList.head(L) < 0  ) {  // recursive case
       return IntList.prepend(IntList.head(L),
                              filterNegatives(IntList.tail(L)));
   } else {
       return filterNegatives(IntList.tail(L));
  }
}
```


The General Filtering Idiom

- Method filterPred(IntList L) returns a new list containing elements of "L" that satisify predicate *pred*.
- For example,



filterPred(IntList L);

```
// Returns a new list containing elements of L that
// satisify predicate pred
public static IntList filterPred(IntList L) {

   if (   IntList.isEmpty(L)   )  {          // base case
       return IntList.empty();
   } else if (  pred( IntList.head(L) ) ) {  // recursive case
       return IntList.preprend(IntList.head(L),
                               filterPred(IntList.tail(L)));
   } else {
       return filterPred(IntList.tail(L));
  }
}
```

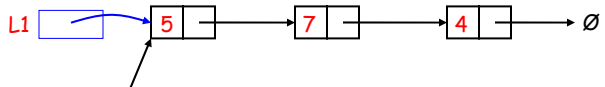

Filters sometimes need extra arguments

```
// Returns a new list containing elements of L that
// are divisors of n
public static IntList filterDivisors(int n, IntList L) {
   if (   IntList.isEmpty(L)   )  {          // base case
       return IntList.empty();
   } else if ( (n % IntList.head(L)) == 0 ) {  // recursive case
       return IntList.preprend(IntList.head(L),
                               filterDivisors(n, IntList.tail(L)));
   } else {
       return filterDivisors(n, IntList.tail(L));
  }
}
```

```
➢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]"))
[]
```

List Glue: prepend

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

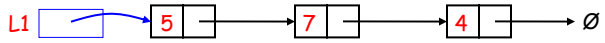


- IntList.prepend(6, L1)
[6,5,7,4]

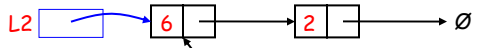

```
// Creates new node whose head is 6
// Shares nodes with list L1
```


List Glue: append

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



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

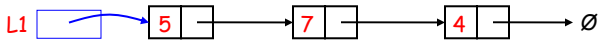


- IntListOps.append(L1, L2)
[5,7,4,6,2]


```
// Creates new nodes corresponding
//          to those from first list
// Shares nodes with second list
```

List Glue: postpend

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



- IntListOps.postpend(L1, 6)

[5,7,4,6]


```
// Creates new nodes corresponding
//                 to those from L1
// Creates new node whose head is 6
```


append(IntList L1, IntList L2)

```
// Returns a list containing the elements of the first list
// followed by the elements of the second list. Creates new nodes
// corresponding to those from the first list, but nodes of the
// second list are shared.
public static IntList append(IntList L1, IntList L2) {









}
```

JEM illustrating invocation of append()

Object Land

Execution Land

```
IntListOps.append(IntList.fromString("[5,7,4]"),
                  IntList.fromString("[6,2]"))
```


JEM illustrating invocation of append()

Object Land

Execution Land

```
IntListOps.append( IL1 ,
                  IntList.fromString("[6,2]"))
```

JEM illustrating invocation of append()

Object Land

Execution Land

```
IntListOps.append(IL1,
                  IL5)
```


JEM illustrating invocation of append()

Object Land

Execution Land

```
IntListOps.append(IL1,
                  IL5)
```

IntListOps.append(IL1, IL5)

L1 IL1 L2 IL5

```
if (IntList.isEmpty(L1)) {
  return L2;
} else {
  return prepend(IntList.head(L1),
         append(IntList.tail(L1), L2)); }
```

JEM illustrating invocation of append()

Object Land

Execution Land

```
IntListOps.append( IL1,
                   IL5 )
```

```
IntListOps.append( IL1, IL5 )
```

L1 IL1 L2 IL5

```
if (false) {
  return L2;
} else {
  return prepend(IntList.head(L1),
                 append(IntList.tail(L1), L2)); }
```


JEM illustrating invocation of append()

Object Land

Execution Land

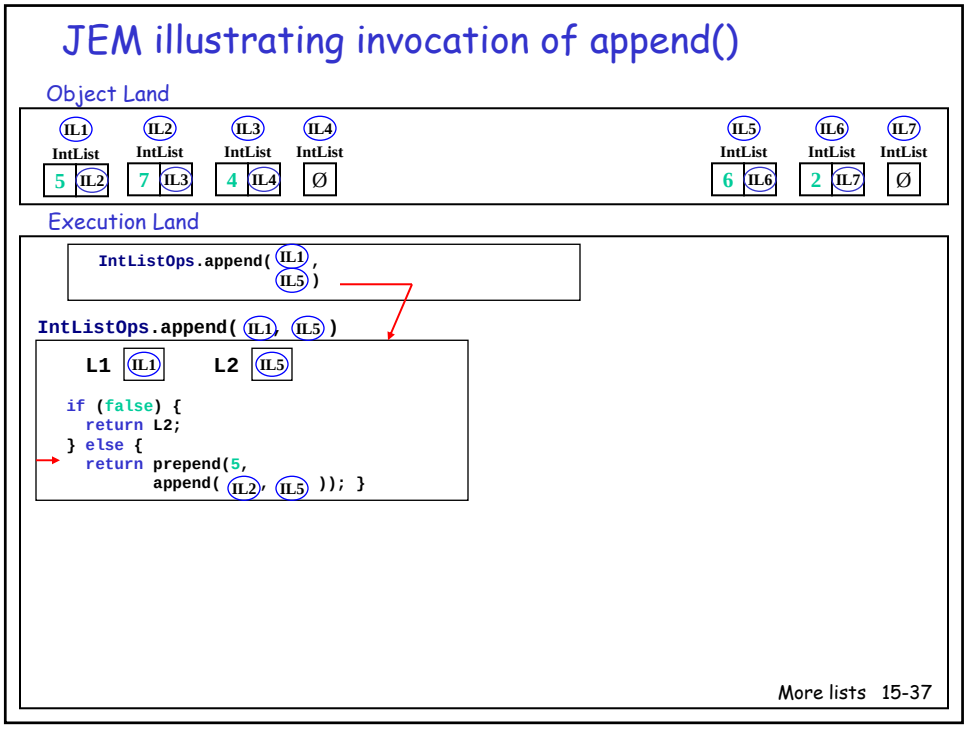
```
IntListOps.append( IL1,
                   IL5 )
```

```
IntListOps.append( IL1, IL5 )
```

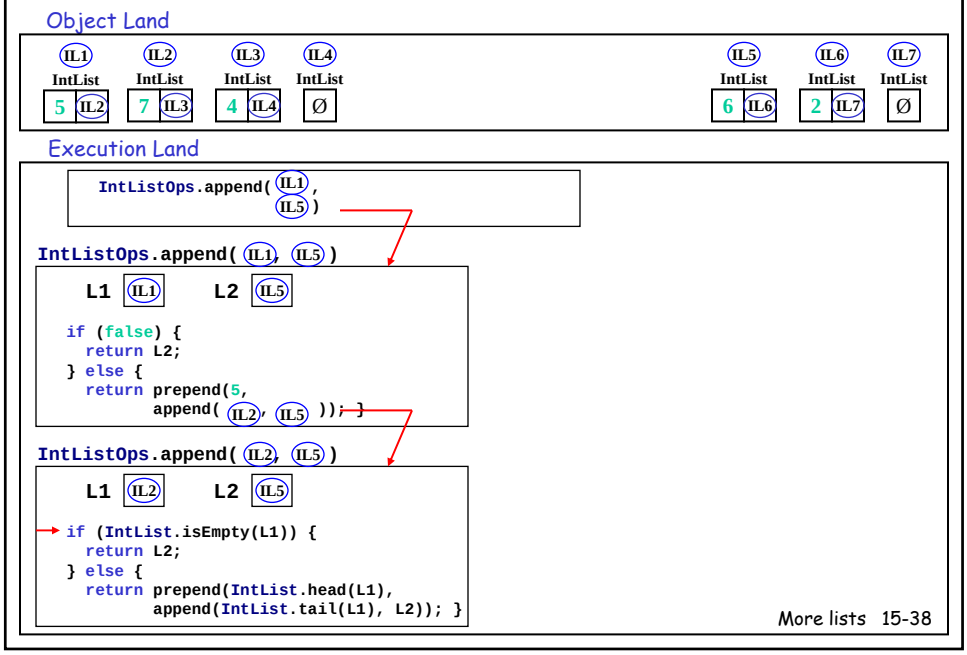
L1 IL1 L2 IL5

```
if (false) {
  return L2;
} else {
  return prepend(5,
                 append(IntList.tail(L1), L2)); }
```

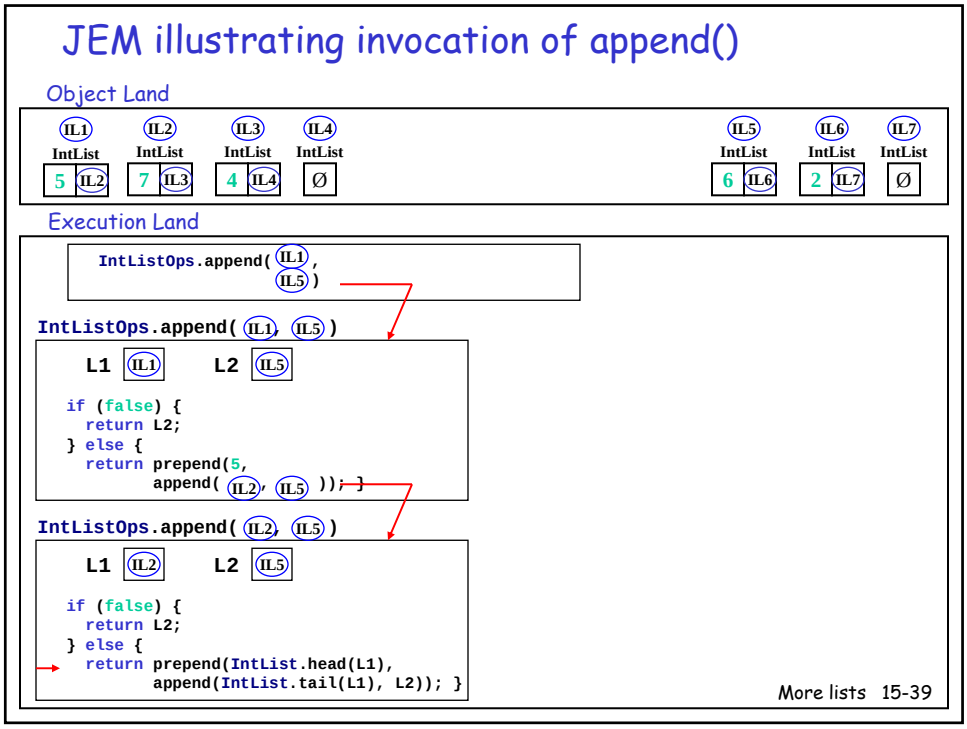
JEM illustrating invocation of append()



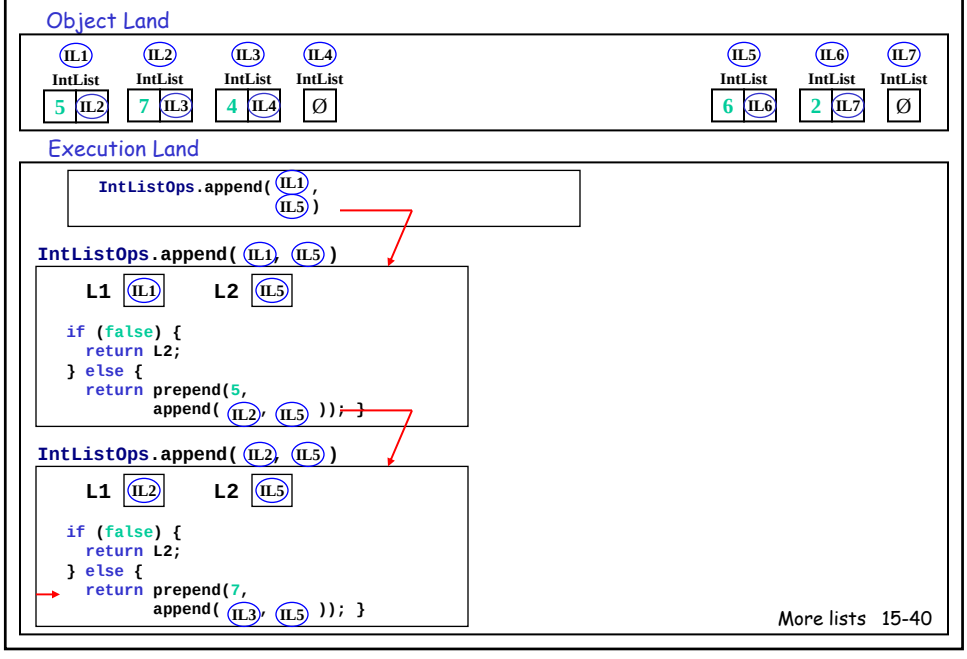
JEM illustrating invocation of append()



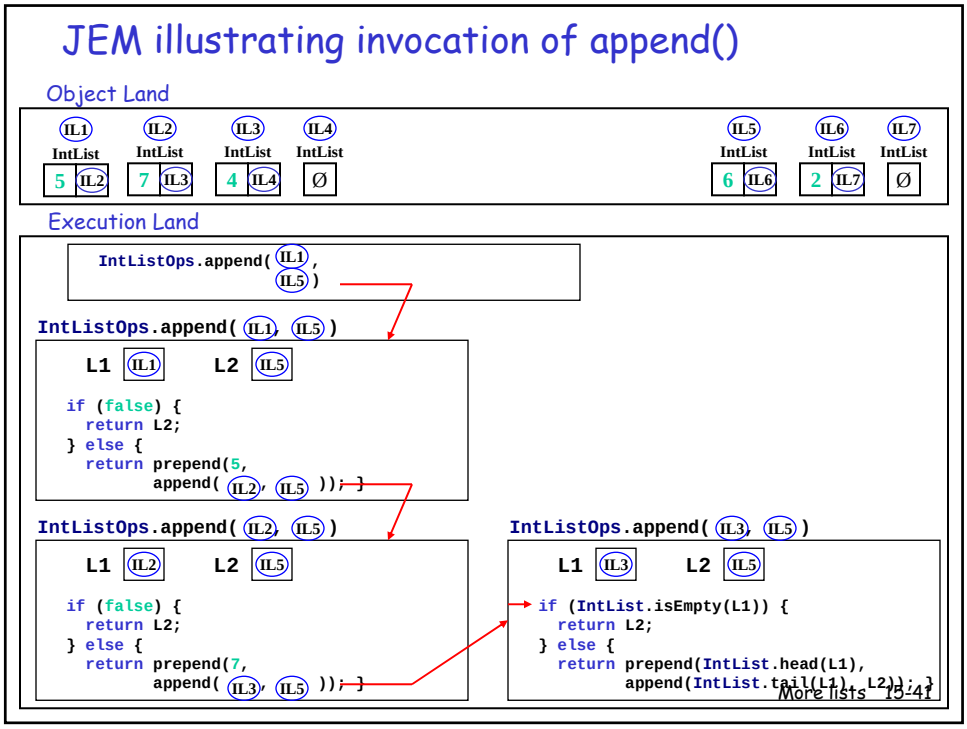
JEM illustrating invocation of append()



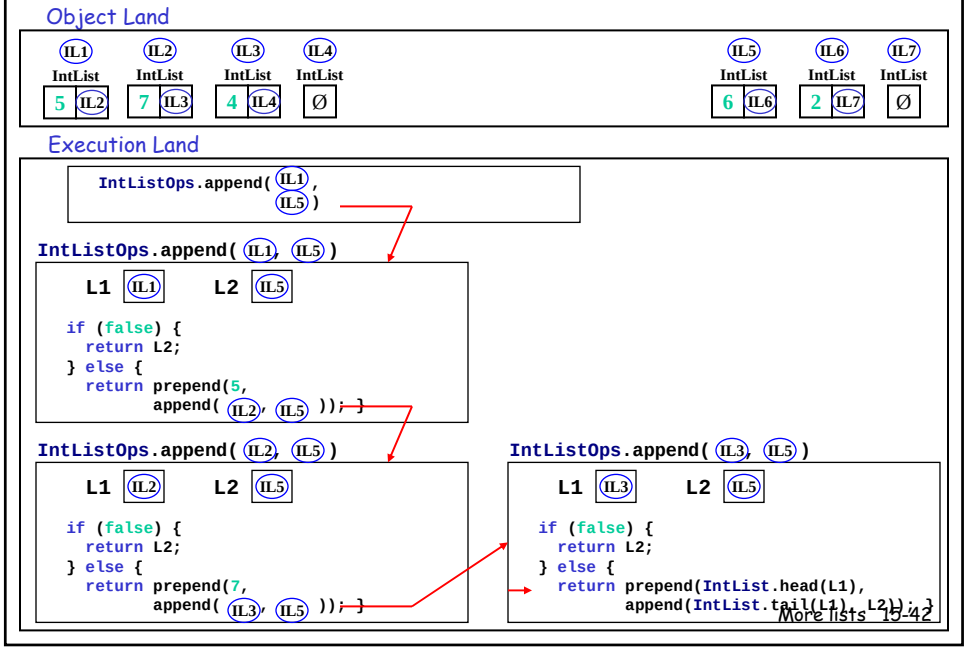
JEM illustrating invocation of append()



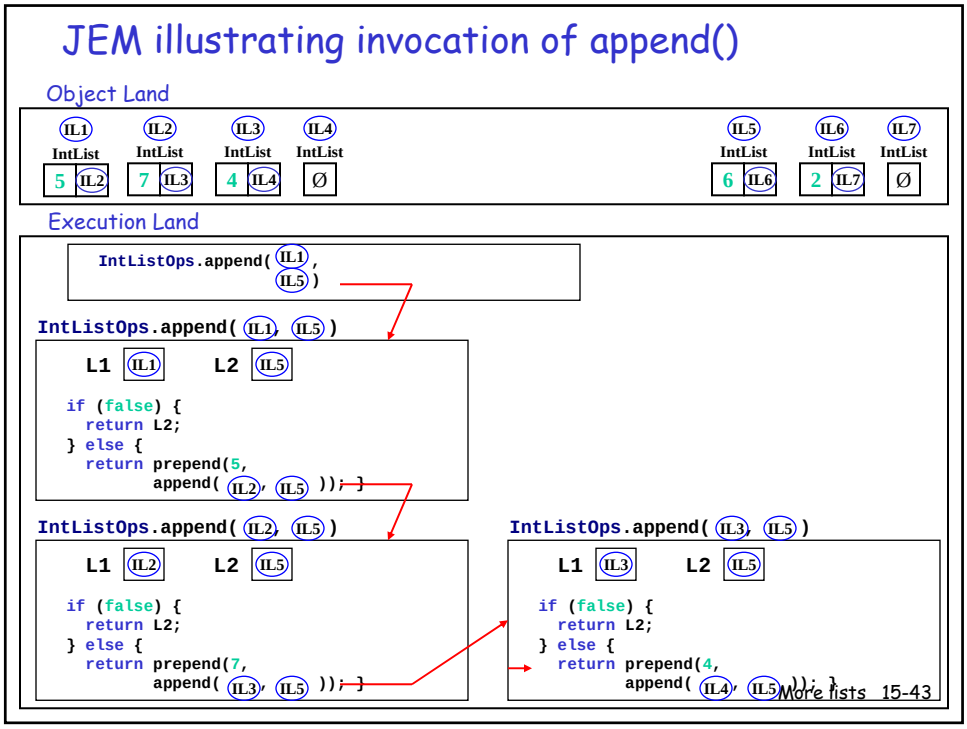
JEM illustrating invocation of append()



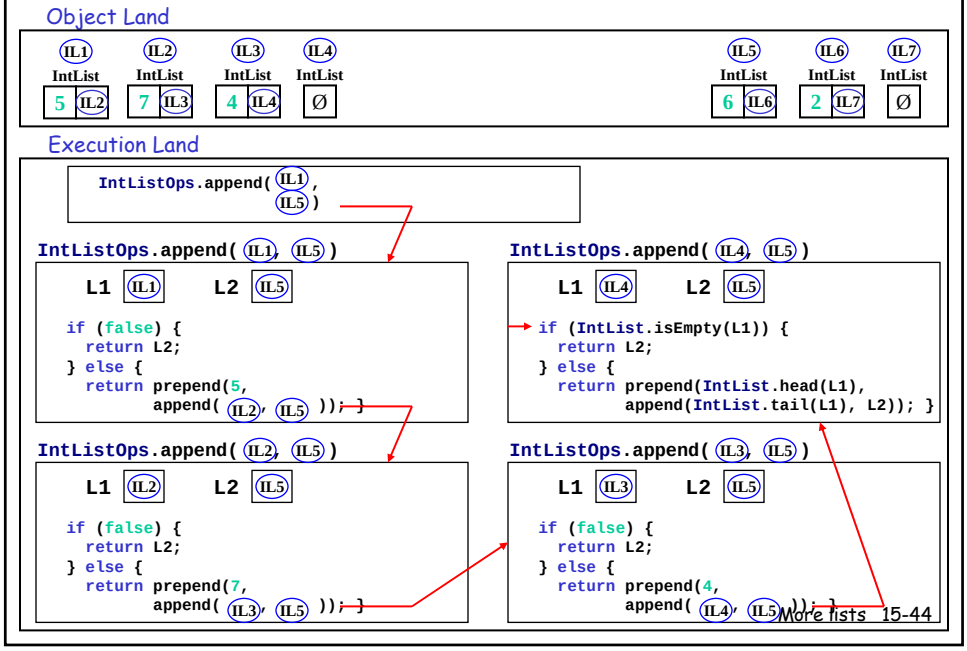
JEM illustrating invocation of append()



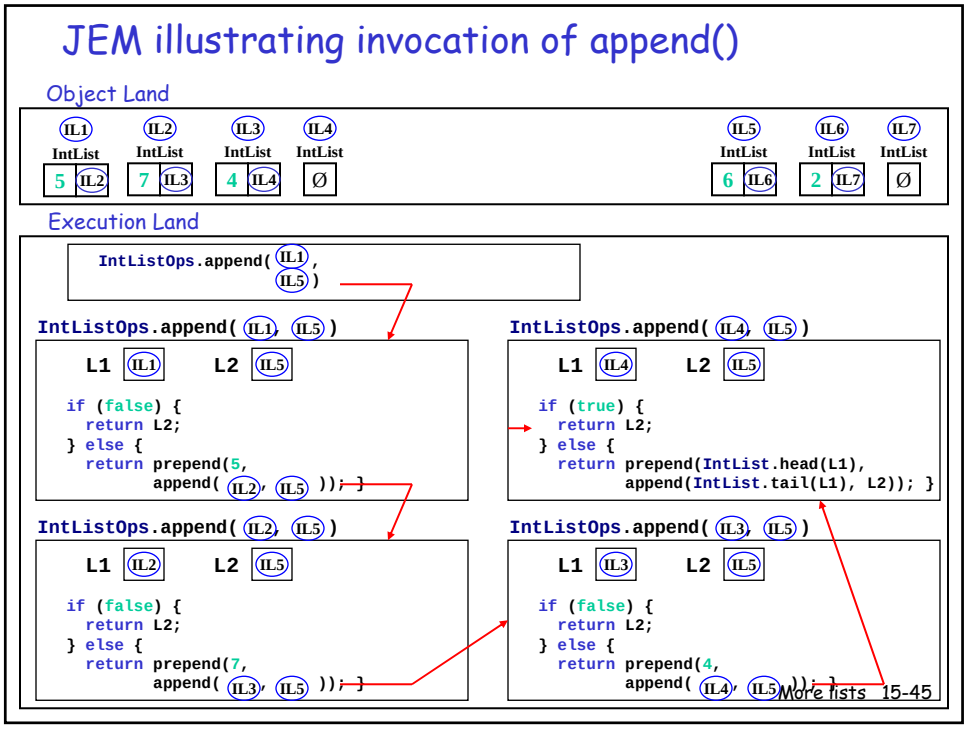
JEM illustrating invocation of append()



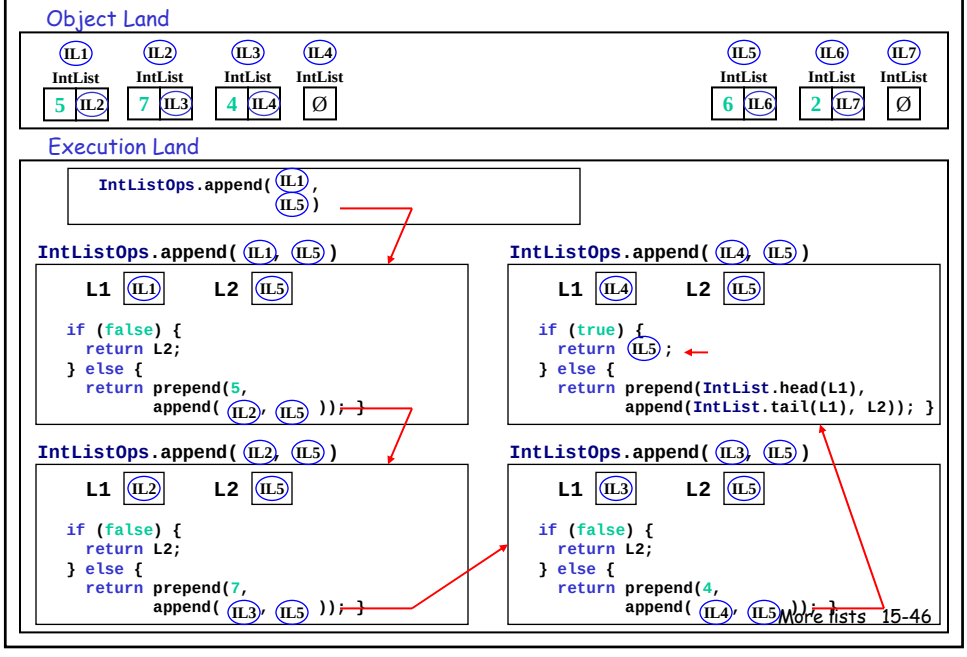
JEM illustrating invocation of append()

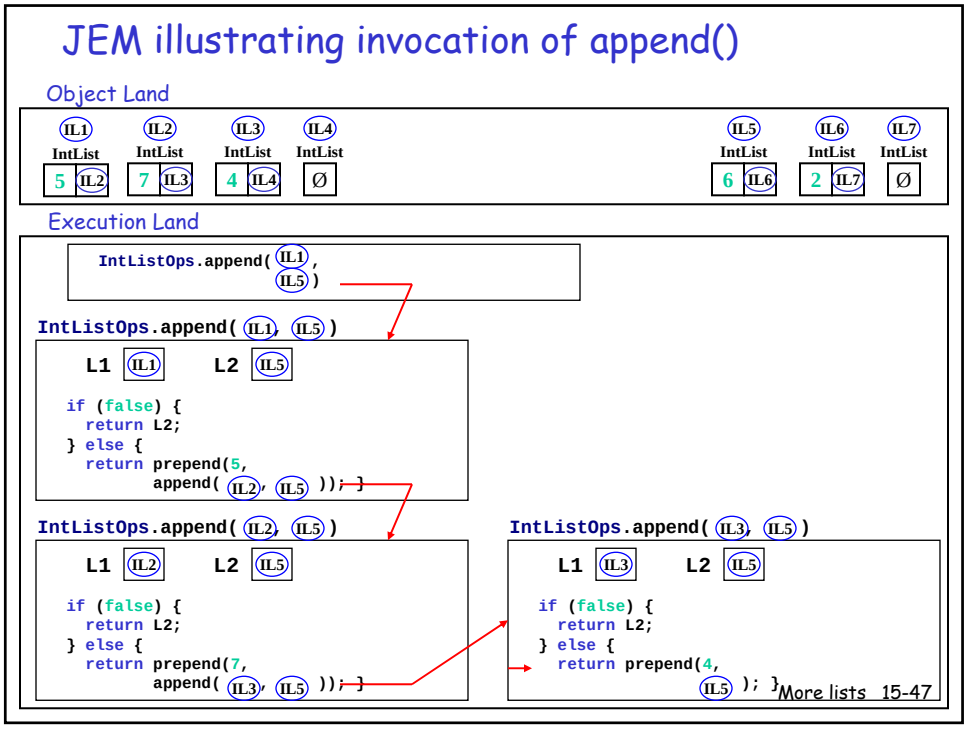


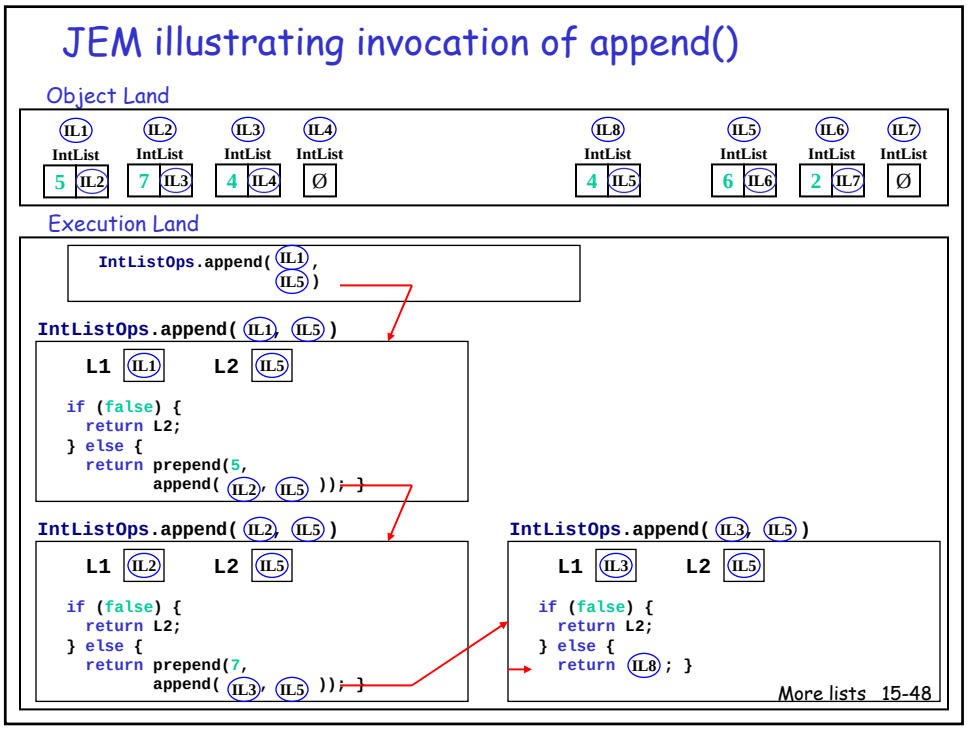
JEM illustrating invocation of append()



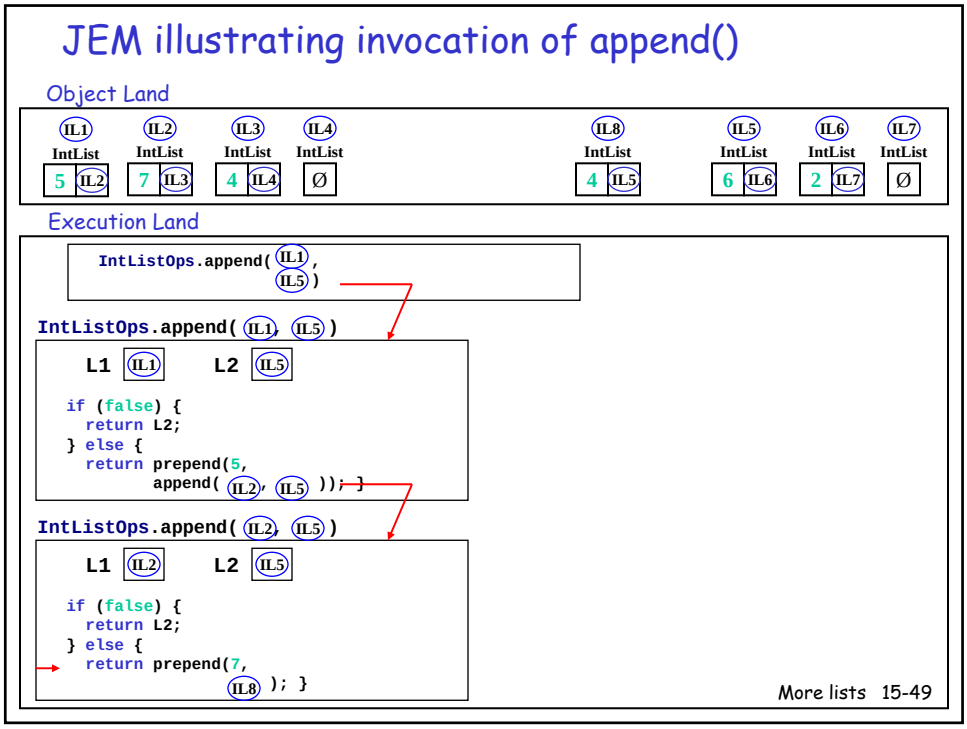
JEM illustrating invocation of append()



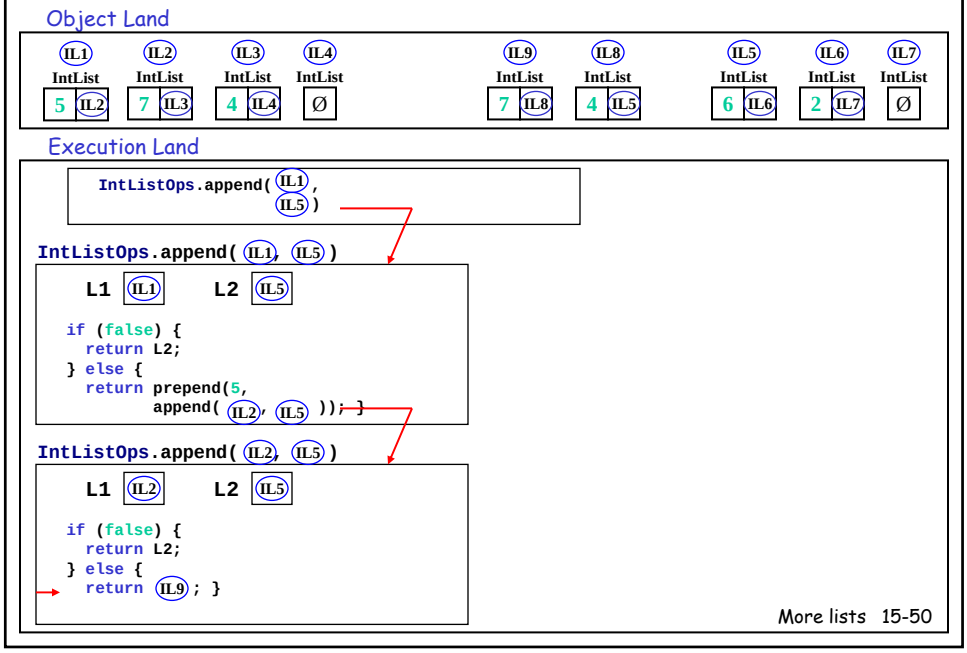

JEM illustrating invocation of append()
Object Land
IL1 IntList 5 IL2
IL2 IntList 7 IL3
IL3 IntList 4 IL4
IL4 IntList Ø
IL5 IntList 6 IL6
IL6 IntList 2 IL7
IL7 IntList Ø
Execution Land
IntListOps.append(IL1, IL5)
IntListOps.append(IL1, IL5)
L1 IL1 L2 IL5
if (false) {
return L2;
} else {
return prepend(5,
append(IL2, IL5)); }
IntListOps.append(IL2, IL5)
L1 IL2 L2 IL5
if (false) {
return L2;
} else {
return prepend(7,
append(IL3, IL5)); }
IntListOps.append(IL3, IL5)
L1 IL3 L2 IL5
if (false) {
return L2;
} else {
return prepend(4,
IL5); }
More lists 15-47


JEM illustrating invocation of append()
Object Land
IL1 IntList 5 IL2
IL2 IntList 7 IL3
IL3 IntList 4 IL4
IL4 IntList Ø
IL8 IntList 4 IL5
IL5 IntList 6 IL6
IL6 IntList 2 IL7
IL7 IntList Ø
Execution Land
IntListOps.append(IL1, IL5)
IntListOps.append(IL1, IL5)
L1 IL1 L2 IL5
if (false) {
return L2;
} else {
return prepend(5,
append(IL2, IL5)); }
IntListOps.append(IL2, IL5)
L1 IL2 L2 IL5
if (false) {
return L2;
} else {
return prepend(7,
append(IL3, IL5)); }
IntListOps.append(IL3, IL5)
L1 IL3 L2 IL5
if (false) {
return L2;
} else {
return IL8; }
More lists 15-48

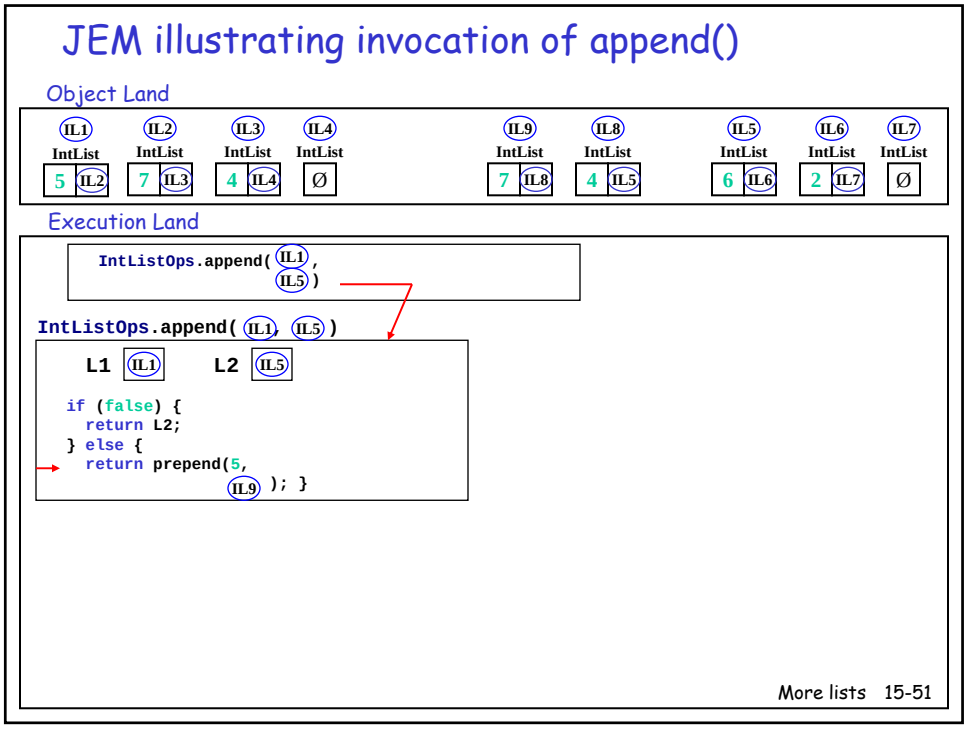
JEM illustrating invocation of append()



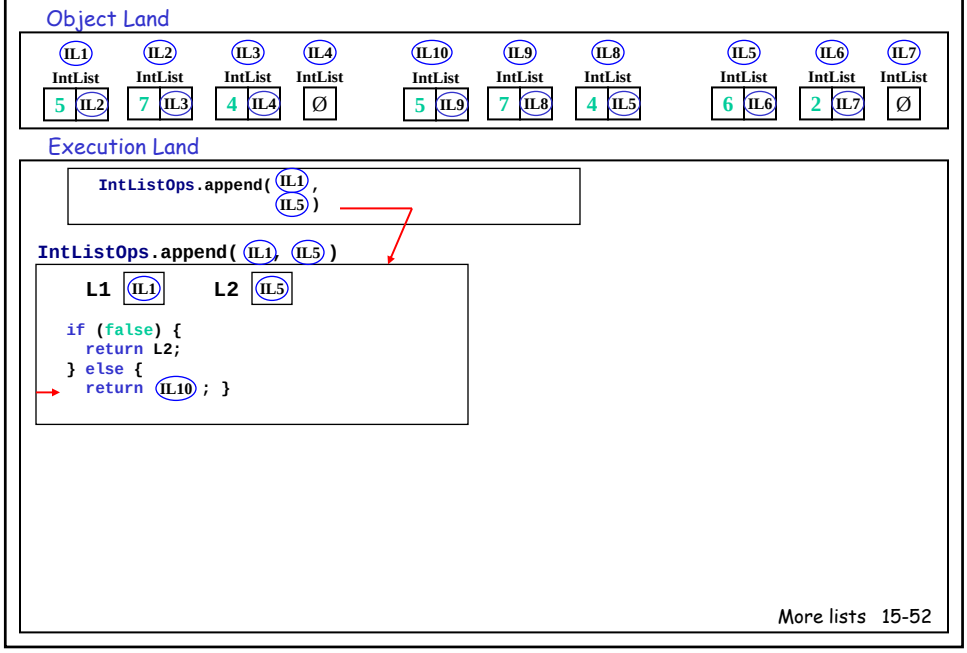
JEM illustrating invocation of append()



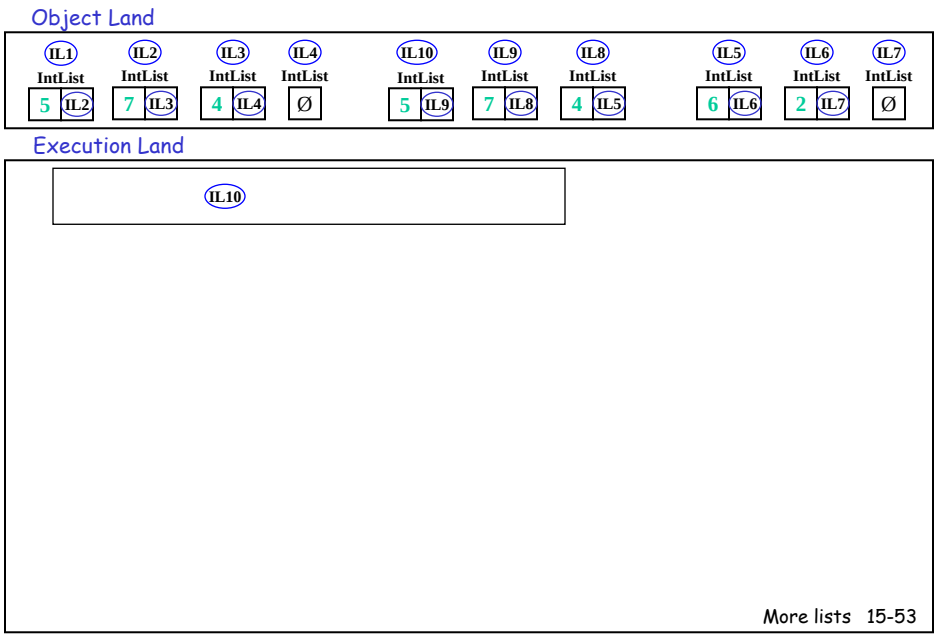
JEM illustrating invocation of append()



JEM illustrating invocation of append()



JEM illustrating invocation of append()



postpend(IntList L, int n)

```
// Returns a list containing the elements of the list followed by n.
// Creates new nodes corresponding to those from the list, and
// creates new node whose head is n.
public static IntList postpend (IntList L, int n) {









}
```

reverse(IntList L)

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

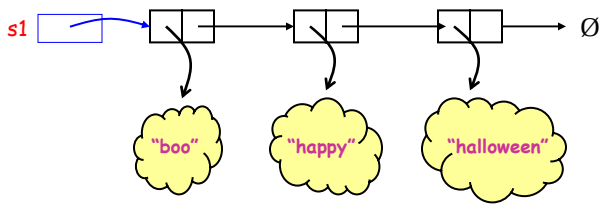
  [4,7,5,6]
- IntListOps.reverse(IntList.fromString("[5,7,4]"))

  [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 L) {
    if (IntList.isEmpty(L)) {

    } else {

    }
}
```


Lists are not just for integers



Five core StringList methods

`public static String head (StringList L)`
Returns the string that is the head component of the string list node **L**.
Signals an exception if **L** is empty.

`public static StringList tail (StringList L)`
Returns the string list that is the tail component of the string list node **L**.
Signals an exception if **L** is empty.

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

`public static StringList empty()`
Returns an empty string list.

`public static StringList prepend (int s, StringList L)`
Returns a new string list node whose head is **s** and whose tail is **L**.

Everything we did before, we can do now

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