CS61B Lecture #4: Values and Containers

• Today: Chapter 4 from A Java Reference.
• Peruse: Chapters 2, 6, 7.
• Today. Simple classes. Scheme-like lists. Destructive vs. non-destructive operations. Models of memory.
Values and Containers

- **Values** are numbers, booleans, and pointers. Values never change.
  
  \[
  3 \quad \text{'}a' \quad \text{true} \quad \frac{1}{\downarrow} \quad \frac{\backslash}{\rightarrow}
  \]

- **Simple containers** contain values:
  
  \[
  x: 3 \quad L: \quad p: \quad \rightarrow
  \]
  
  Examples: variables, fields, individual array elements, parameters.

- **Structured containers** contain (0 or more) other containers:

<table>
<thead>
<tr>
<th>Class Object</th>
<th>Array Object</th>
<th>Empty Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>h t</td>
<td>0 1 2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>42 17 9</td>
<td></td>
</tr>
</tbody>
</table>

  **Alternative Notation**
  
  \[
  h: 3 \quad t: \quad h: \quad 0 42 \quad 1 17 \quad 2 9
  \]
Pointers

- **Pointers** (or **references**) are values that **reference** (point to) containers.

- One particular pointer, called **null**, points to nothing.

- In Java, structured containers contain only simple containers, but pointers allow us to build arbitrarily big or complex structures anyway.

![Diagram of pointers and containers]

```
0 1
3 9
0
17
```
Containers in Java

- Containers may be named or anonymous.
- In Java, all simple containers are named, all structured containers are anonymous, and pointers point only to structured containers. (Therefore, structured containers contain only simple containers).

```
named simple containers (fields)
within structured containers
```

```
p: simple container (local variable)
  3  
  7  
  h t
```

- In Java, assignment copies values into simple containers.
- Exactly like Scheme!
Defining New Types of Object

- Class declarations introduce new types of objects.
- Example: list of integers:

```java
public class IntList {
    // Constructor function
    // (used to initialize new object)
    /** List cell containing (HEAD, TAIL). */
    public IntList (int head, IntList tail) {
        this.head = head; this.tail = tail;
    }

    // Names of simple containers (fields)
    public int head;
    public IntList tail;
}
```
Primitive Operations

IntList Q, L;

L = new IntList(3, null);
Q = L;

Q = new IntList(42, null);
L.tail = Q;

L.tail.head += 1;
// Now Q.head == 43
// and L.tail.head == 43
Destructive vs. Non-destructive

**Problem:** Given a (pointer to a) list of integers, \( L \), and an integer increment \( n \), return a list created by incrementing all elements of the list by \( n \).

```c
/** List of all items in P incremented by n. */
static IntList incrList (IntList P, int n) {
    if (P == null)
        return null;
    else return new IntList (P.head+n, incrList(P.tail, n));
}
```

We say \texttt{incrList} is \textit{non-destructive}, because it leaves the input objects unchanged, as shown on the left. A \textit{destructive} method may modify the input objects, so that the original data is no longer available, as shown on the right:

After \( Q = \text{incrList}(L, 2) \):

\[
\begin{array}{c}
L: & \rightarrow 3 & \rightarrow 43 \\
Q: & \rightarrow 5 & \rightarrow 45 \\
\end{array}
\]

After \( Q = \text{dincrList}(L, 2) \) (destructive):

\[
\begin{array}{c}
L: & \rightarrow 5 & \rightarrow 45 \\
Q: & \rightarrow 5 & \rightarrow 45 \\
\end{array}
\]
An Iterative Version

An iterative incrList is tricky, because it is not tail recursive. Easier to build things first-to-last, unlike recursive version:

static IntList incrList (IntList P, int n) {
    if (P == null)
        return null;
    IntList result, last;
    result = last
        = new IntList (P.head+n, null);
    while (P.tail != null) {
        P = P.tail;
        last.tail
            = new IntList (P.head+n, null);
        last = last.tail;
    }
    return result;
}
An Iterative Version

An iterative incrList is tricky, because it is not tail recursive. Easier to build things first-to-last, unlike recursive version:

```java
static IntList incrList (IntList P, int n) {
    if (P == null)
        return null;
    IntList result, last;
    result = last
        = new IntList (P.head+n, null);
    while (P.tail != null) {
        P = P.tail;
        last.tail
            = new IntList (P.head+n, null);
        last = last.tail;
    }
    return result;
}
```

P: 3 43 56

last: 3

result: 5
An Iterative Version

An iterative `incrList` is tricky, because it is not tail recursive. Easier to build things first-to-last, unlike recursive version:

```java
static IntList incrList (IntList P, int n) {
    if (P == null)
        return null;
    IntList result, last;
    result = last
        = new IntList (P.head+n, null);
    while (P.tail != null) {
        P = P.tail;
        last.tail
            = new IntList (P.head+n, null);
        last = last.tail;
    }
    return result;
}
```

```
P: 3 → 43 → 56

last:

result: 5
```
An Iterative Version

An iterative incrList is tricky, because it is not tail recursive. Easier to build things first-to-last, unlike recursive version:

```java
static IntList incrList (IntList P, int n) {
    if (P == null)
        return null;
    IntList result, last;
    result = last
        = new IntList (P.head+n, null);
    while (P.tail != null) {
        P = P.tail;
        last.tail
            = new IntList (P.head+n, null);
        last = last.tail;
    }
    return result;
}
```

Last modified: Fri Sep 5 12:59:28 2008
An Iterative Version

An iterative `incrList` is tricky, because it is not tail recursive. Easier to build things first-to-last, unlike recursive version:

```java
static IntList incrList (IntList P, int n) {
    if (P == null)
        return null;
    IntList result, last;
    result = last
        = new IntList (P.head+n, null);
    while (P.tail != null) {
        P = P.tail;
        last.tail
            = new IntList (P.head+n, null);
        last = last.tail;
    }
    return result;
}
```

![Diagram of `incrList` function with example input and output.]
An Iterative Version

An iterative incrList is tricky, because it is not tail recursive.
Easier to build things first-to-last, unlike recursive version:

```java
static IntList incrList (IntList P, int n) {
    if (P == null)
        return null;
    IntList result, last;
    result = last
        = new IntList (P.head+n, null);
    while (P.tail != null) {
        P = P.tail;
        last.tail
            = new IntList (P.head+n, null);
        last = last.tail;
    }
    return result;
}
```

Last modified: Fri Sep 5 12:59:28 2008
An Iterative Version

An iterative incrList is tricky, because it is not tail recursive. Easier to build things first-to-last, unlike recursive version:

```java
static IntList incrList (IntList P, int n) {
    if (P == null)
        return null;
    IntList result, last;
    result = last
        = new IntList (P.head+n, null);
    while (P.tail != null) {
        P = P.tail;
        last.tail
            = new IntList (P.head+n, null);
        last = last.tail;
    }
    return result;
}
```

Last modified: Fri Sep 5 12:59:28 2008
An Iterative Version

An iterative incrList is tricky, because it is not tail recursive. Easier to build things first-to-last, unlike recursive version:

```java
static IntList incrList (IntList P, int n) {
    if (P == null)
        return null;
    IntList result, last;
    result = last
        = new IntList (P.head+n, null);
    while (P.tail != null) {
        P = P.tail;
        last.tail
            = new IntList (P.head+n, null);
        last = last.tail;
    }
    return result;
}
```

Last modified: Fri Sep 5 12:59:28 2008