CS61B Lecture #3: Containers

- Readers Available: from Vick Copy (corner of Euclid and Hearst) not Copy Central! Also online.

- Please read Chapter 2 of Assorted Materials on Java from the reader.

- Room change: Discussion 114 (3-4 Thurs.) is now in 289 Cory (used to be 3111 Etch.)

- Midterm is tentatively scheduled for the evening of 9 March (Thursday).

- Project 1 will be due the preceding week (1 March).

Values and Containers

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

- **Simple containers** contain values:

  \[
  x: 3 \quad L: \quad p: \quad \frac{\text{}}{\text{}}
  \]

  Examples: variables, fields, individual array elements, parameters.

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

  \[
  \text{Class Object } \quad \text{Array Object } \quad \text{Empty Object}
  \]

  \[
  h \quad t \quad \frac{\text{}}{\text{}}
  \]

  \[
  0 \quad 1 \quad 2 \quad \frac{\text{}}{\text{}}
  \]

  \[
  42 \quad 17 \quad 9 \quad \frac{\text{}}{\text{}}
  \]

  \[
  \text{Alternative Notation}
  \]

  \[
  h: 3 \quad \frac{\text{}}{\text{}}
  \]

  \[
  0 \quad 42 \quad \frac{\text{}}{\text{}}
  \]

  \[
  1 \quad 17 \quad \frac{\text{}}{\text{}}
  \]

  \[
  2 \quad 9 \quad \frac{\text{}}{\text{}}
  \]
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

0 1

0 9 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).

```
  p: simple container (local variable)   named simple containers (fields)
      ————> 3 ————> 7
        within structured containers

      h   t

      structured containers (anonymous)
```

- 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;
}
```
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 non-destructive, because it leaves the input objects unchanged, as shown on the left. A destructive method may modify the input objects, so that the original data is no longer available, as shown on the right:

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

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

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

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

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

```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
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 list operations]
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: Mon Jan 23 14:30:41 2006
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: Mon Jan 23 14:30:41 2006
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;
}
```
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 → 45
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 45 58
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;
}
```