Java

Java, an Imperative Language

- Imperative languages often **permit** the use of functional programming.
- e.g. sometimes just say “no” to side-effects.
- Better yet, use functions and side-effects articulately and to **best advantages of both**.

Java Jive§

(review)

§ http://members.tripod.com/swingstyle/mlist/msts/msts07.html

James Gosling, Inventor of Java

James Gosling received a BSc in Computer Science from the University of Calgary, Canada in 1977. He received a PhD in Computer Science from Carnegie-Mellon University in 1983. He is currently a Distinguished Engineer at Sun Microsystems. He has built satellite data acquisition systems, a multiprocessor version of Unix, several compilers, mail systems and window managers. He has also built a WYSIWYG text editor, a constraint based drawing editor and a text editor called `Emacs' for Unix systems. More recently he has been the lead engineer for the Java/HotJava system.

http://java.sun.com/people/jag/

Java vs. rex

- The analog to *function* in rex is *method* in Java. Functions are applied as aFunction(x, y, z), while methods are applied like x.aMethod(y, z). principal argument (an object)

- Argument and return **types** must be declared in Java, not in rex.

- Both allow recursion.

- A library ("package") Polya (http://www.cs.hmc.edu/~keller/polya) provides much of rex functionality in Java (and a similar package exists for C++).

The empty Java program

```
class empty
{

public static void main(String arg[ ])
  {
  }

}
```

The empty Java program

The one and only class of this program

```
class empty
{
```

Makes this method accessible from the outside.

```
public static void main(String arg[ ])
  {
  }

}
```

The main method for this class (called at start-up).

External arguments for this method.

Result type of this method (none).

Says that this method depends only on the class, not any one object.

The "hello, world" program in Java

```
class hello
{
public static void main(String arg[])
  {
  System.out.println("hello, world");
  }
}
```

The “hello, world” program in Java

The empty program + one line.

```
class hello
{
public static void main(String arg[])
  {
  System.out.println("hello, world");
  }
}
```

The print-with-end-of-line method for object System.out.

The standard output stream object, pre-defined in the System class.

The “System” class.

Running Java on turing

- Current version is 1.3.1
- To compile:

UNIX convention for compiler, e.g. javac, cc

```
javac hello.java
```

- To execute:

No "c" here.

No ".class" here.

```
java hello
```

One-time setup for using Java on turing

In your .cshrc file, include these lines:

```
setenv JAVA_HOME /usr/local/jdk1.31

setenv CLASSPATH $JAVA_HOME/lib:/cs/cs60/java/:.
```

Note: additions don't take effect until next login *or* until you execute:

```
source ~/.cshrc
```

Running Java on turing

```
turing 101> ls hello.*
hello.java
```

Check what's there.

```
turing 102> javac hello.java
```

Compile it.

```
turing 103> ls hello.*
hello.class   hello.java
```

Check what's there now.

```
turing 104> java hello
hello, world
```

Run it.

Be astounded by results.

Useful Hacky Shortcuts

Since java is a prefix of javac, this tends to confound using command completion (e.g. **`!j`** in the Cshell).

Consider putting in your .cshrc the following command definitions:

```
alias jc 'javac \!$.java'                  #compile java
alias je 'java'                             #execute
alias jx 'javac \!$.java ; java \!$'        #both
```

Example usage:

```
jc foo       # same as javac foo.java
je foo       # same as java foo
jx foo       # same as javac foo.java; java foo
```

Then use **`!jc`** or **`!je`** to re-do previous commands of same type.

Java Objects

- Java data items are either:
  - primitive, such as
    - int, long, float, double, char
  - Objects, such as
    - String, Long, Double
    - Arrays are sort of like objects too.

Purposes of Objects

- **Aggregate** various data objects together
- Allow mutation of the **state** of data objects
- Control use and access of data according to specific **disciplines**
- and other good stuff

Immutable Objects

- *An* Object is **immutable** if it state never changes once it is created.
- **Functional programming** deals with immutable objects *almost* exclusively
  - (exception: delayed evaluation)
- The aggregating and disciplined access properties of Objects are still very useful.

Introducing Polya

- Permits rex-like computation within Java
- Name derives from 2 things:
  - George Polya, famous mathematician who wrote about problem solving ("How to Solve It")
    - Polya package *solves the problem* of rapidly prototyping list-handling programs
  - *Poly*list (polymorphic list) data structures, as in rex

Trivial Program using Polya

```
import polya.*;

class helloPolya
{
public static void main(String arg[])
  {
  System.out.println(Polylist.list("Hello", "Polya"));
  }
}
```

A list constructor.

Output of Trivial Program

```
(Hello Polya)
```

Parens that Polya puts around lists.

Polylist.nil

```
import polya.*;
```

`Polylist.nil` represents the empty list.

Using Polya to Implement Unicalc Functionality (a4)

- *As in a3*, we want to represent Quantities.
- In Java, instead of using a list of 3 things, we will use a little more structure:

```
public class Quantity
{
double Factor;
Polylist Num;
Polylist Denom;
        ... more stuff to come ...
}
```

Object Creation

- Objects are created using **constructors.**
- For a given Class of Objects, there can be multiple types of constructors, each providing different types of parameters to define the creation of an object.

Constructors for Quantities

- Constructors always take the same name as their Class.
- Therefore, all constructors for class `Quantity` will be called (you guessed it)

  `Quantity`
- Constructors will differ depending on types.
- One constructor can call another.

Basic Quantity Constructor

- To define a quantity, we need to give values to all three internal variables:

```
public class Quantity
{
            ... stuff you already saw ...
Quantity(double Factor, Polylist Num, Polylist Denom)
  {
  this.Factor = Factor;
  this.Num    = Num;
  this.Denom  = Denom;
  }
            ... more stuff to come ...
}
```

Variables in red represent values in *this* Quantity.
Variables in green represent values local to this constructor.
The latter go away when the constructor is left.

Convenience Quantity Constructor where the denominator is empty

```
public class Quantity
{
                  ... stuff you already saw ...
Quantity(double Factor, Polylist Num)
  {
  this(Factor, Num, Polylist.nil);
  }
                  ... more stuff to come ...
 }
```

Variables in green represent values local to this constructor.
`this(…)` means "call the constructor of this class with the indicated arguments.

Convenience Quantity Constructor

where both numerator and denominator are empty

```
public class Quantity
{
                 ... stuff you already saw ...
Quantity(double Factor)
  {
  this(Factor, Polylist.nil, Polylist.nil);
  }
                 ... more stuff to come ...
}
```

Variables in green represent values local to this constructor.
`this(…)` means "call the constructor of this class with the indicated arguments.

Convenience Quantity Constructor

where the numerator is a single unit and denominator is empty

```
public class Quantity
{
                ... stuff you already saw ...
Quantity(double Factor, String NumUnit)
  {
  this(Factor, Polylist.list(NumUnit), Polylist.nil);
  }
                ... more stuff to come ...
}
```

Variables in green represent values local to this constructor.
this(…) means “call the constructor of this class with the indicated arguments.

Convenience Methods

- In the skeletal solution to a4, we avoid having to prefix with

  `Polylist.`

  by defining local versions of functions:

```
/**
 * local cons for convenience
 */

static Polylist cons(Object F, Polylist R) { return Polylist.cons(F, R); }


/**
 * local list of 1 argument, for convenience
 */

static Polylist list(Object X1) { return Polylist.list(X1); }
```

Getters

- Attributes of objects should never be accessed within an object simply by referring to them:

```
Quantity x = new Quantity(…);
…
System.out.println(x.Num);
```

BAD

- **except** possibly for debugging purposes.
- Instead, use a **getter** method:

```
int getNum()
{
return Num;
}
...
System.out.println(x.getNum());
```

GOOD

Reasons?

Static?

- What is *Static* all about?
- In Java, a method may or may not depend on the state of a specific Object:
  - methods that do not depend on this state should be annotated as
    `static`

Static can only call Static

- A static method can only depend on
  - variables local to the method
  - variables delcared as static
  - other static methods
- A static method, therefore, **cannot** depend on:
  - variables not declared as static
  - other methods not declared as static
- The compiler will tell you, but maybe in a cryptic way.

Example

```
class myBad
{
int x;

myBad(int x)
  {
  this.x = x;
  }

int getX()
  {
  return x;
  }

static int test()
  {
  getX() > 0;
  }
}
```

Illegal:
static depends on non-static

R and S expressions

- Output from Polya is set up for "S expressions"
  - `(Hello Polya)`
- Input can be in the form of S expressions or "R expressions" (from rex)

R and S expressions

- R expression from rex:

  ```
  [1, 2, [buckle, [my, shoe]]]
  ```

- Corresponding S expression:

  ```
  (1 2 (buckle (my shoe)))
  ```

- They are the same except that S expressions use round parentheses instead of square ones and S expressions omit commas.

S Expressions

- Widely-used
  - Devised by John McCarthy at MIT for the Lisp programming language
  - Used in Scheme language
  - A nearly-universal scheme for structuring data in readable form.

R->S Expression Transformer

The following Java program reads input as R expressions, then prints the result as an S expression (the default):

```
import polya.*;

class RtoS
  {
  static String promptString = "Rexp: ";         // default prompt

  public static void main(String arg[])
    {
    Tokenizer in = new Tokenizer(System.in);     // R exp tokenizer, defines parsing of R and S exps

    Object ob;                                   // the object to be read

    while( true )
      {
      System.out.print(promptString);            // prompt for input

      ob = in.nextRexp();                        // read object as R expression (one step)

      if( ob == Tokenizer.eof ) break;           // break on end-of-file (leave while loop)

      System.out.println(ob);                    // show the object
      }
    System.out.println();                        // new line when program ends
    }
  }
```

R->S Expression Output

Execution of the previous program:

```
Rexp: [This, is, an, R, expression]
(This is an R expression)

Rexp: [This, is, 1, 2]
(This is 1 2)

Rexp: [[Floats, 1.23], [Ints, 45], [Strings], [Lists, [of, Lists]]]
((Floats 1.23) (Ints 45) (Strings) (Lists (of Lists)))
```

R Expression Analyzer

```
import polya.*;

class analyzeRexp
  {
  static String promptString = "analyze Rexp: "; // default prompt

  public static void main(String arg[])
    {
    Tokenizer in = new Tokenizer(System.in);     // R exp tokenizer

    Object ob;                                   // the object to be analyzed

    while( true )
      {
      System.out.print(promptString);            // prompt for input

      ob = in.nextRexp();                        // read object as R expression

      if( ob == Tokenizer.eof ) break;           // break on end-of-file

      System.out.println(Polylist.analysis(ob)); // show analysis
      }
    System.out.println();                        // new line when program ends
    }
  }
```

R Expression Analyzer Execution

```
analyze Rexp: [This, is, 1, 2]

A Polylist consisting of 4 elements:
  This (class java.lang.String)
  is (class java.lang.String)
  1 (class java.lang.Long)
  2 (class java.lang.Long)

analyze Rexp: [[Floats, 1.23], [Ints, 45], [Strings], [Lists, [of, Lists]]]

A Polylist consisting of 4 elements:
  A Polylist consisting of 2 elements:
    Floats (class java.lang.String)
    1.23 (class java.lang.Double)
  A Polylist consisting of 2 elements:
    Ints (class java.lang.String)
    45 (class java.lang.Long)
  A Polylist consisting of 1 element:
    Strings (class java.lang.String)
  A Polylist consisting of 2 elements:
    Lists (class java.lang.String)
    A Polylist consisting of 2 elements:
      of (class java.lang.String)
      Lists (class java.lang.String)
```

Polylist Operations

```
import polya.*;

class listOps
  {
  public static void main(String arg[])
    {
    Polylist L = Polylist.list(new Long(1), new Long(2), new Long(3));
    Polylist M = Polylist.list("apple", "banana", "grape", "kiwi");

    System.out.println("L                       = " + L);
    System.out.println("L.first()               = " + L.first());
    System.out.println("L.rest()                = " + L.rest());
    System.out.println("L.reverse()             = " + L.reverse());
    System.out.println(”M                       = " + M);
    System.out.println("L.append(M)             = " + L.append(M));
    System.out.println("L.cons(new Long(0))     = " + L.cons(new Long(0)));
    System.out.println("M.nth(2)                = " + M.nth(2));
    System.out.println("Polylist.list(L, M)     = " + Polylist.list(L, M));
    }
  }
```

Polya List Operation Execution

```
reference: Polylist L = Polylist.list(new Long(1), new Long(2), new Long(3));
           Polylist M = Polylist.list("apple", "banana", "grape", "kiwi");
```

```
L                       = (1 2 3)

L.first()               = 1

L.rest()                = (2 3)

L.reverse()             = (3 2 1)

M                       = (apple banana grape kiwi)

L.append(M)             = (1 2 3 apple banana grape kiwi)

L.cons(new Long(0))     = (0 1 2 3)

M.nth(2)                = grape

Polylist.list(L, M)     = ((1 2 3) (apple banana grape kiwi))
```

Wrappers

- new Long(0)??
- Items in a Polylist must be Objects. Primitives (ints, longs, floats, doubles, chars …) are not Objects in Java.
- The constructor **`Long( )`** makes an Object for any long by creating a "wrapper" which is an object.
- Other wrappers: **`Integer()`**, **`Float()`**, **`Double()`**, **`Boolean()`**, Snoop, Ice-T, …

Strings

- In contrast to `long`, `int`, `float`, etc. `Strings` are already objects.
- Consequently, `Strings` do not need extra wrappers.
- `Polylists` are also Objects.

Getters for Wrappers

- These can be applied to any Object derived from class **`Number`**, which includes **`Long`**, **`Integer`**, …:

  `longValue()`, **`intValue()`**, …

- Use the on-line javadoc pages on the web to find info:

  http://java.sun.com/j2se/1.3/docs/api/

Conversion to String

- Class `String` includes the following static methods (not constructors):

```
valueof(double d)
valueof(long x)
```

    ...

- Each returns a `String`.

Cheap Conversion to String

- "Adding" a number to a string will convert the number to a string, then **concatenate**:

```
String s = "" + 31415;
```

Conversion **from** String

- Use the appropriate static method in the class **to which** you wish to convert, e.g.
  - **`Long.parseLong(String nm)`**
  - **`Double.parseDouble(String nm)`**

- (Don't use **`getLong`**, which has a different meaning entirely.)

Polya I/O

- Output: When Polya prints a **`Polylist`**, it checks each item in the list; if it is a **`Polylist`**, it adds the parentheses to show the list grouping.

- Input: When a **`Tokenizer`** does **`nextSexp()`**, it reads a single well-formed S expression.

S expression reading in Polya

- An integer (no decimal point or exponent) becomes a **`Long`**.
- A floating numeral (decimal point or exponent) becomes a **`Double`**.
- Something with (...) becomes a **`Polylist`**.
- Anything else becomes a **`String`**.

Type Discrimination

- The type of an Object can be discriminated using the `instanceof` operator:

```
Object ob = in.nextSexp();

if( ob instanceof Long ) …

if( ob instanceof Polylist ) ...
```

Equality Checking

- To check whether two Objects are *equal*, **DO NOT USE ==**. This only checks whether the **references** to those objects are identical. The Objects could be equal, but be different Objects. This applies for **`Strings`**, for example.

- **DO** USE **`equals`**:

```
if( ob1.equals(ob2) )
```

Essential Polylist Operations

- **list(Ob_0, Ob_1, …., Ob_N)** creates a Polylist from objects (up N = 15 or so).
- **L.first()** returns the first element.
- **L.rest()** returns the rest of the elements as a **Polylist**.
- **L.isEmpty()** returns true if the list is empty.
- **L.nonEmpty()** returns true if non-empty.
- **L.cons(Ob_0)** creates a new list with **Ob_0** first.
- **Polylist.nil** is the empty **Polylist**.

Other Polylist Operations

- `L.append(M)` creates a new list with elements of `M` following those of `L`.
- `L.reverse()` creates a new list reversing `L`.
- `L.nth(long N)` returns the `N`th element of `L` `(N = 0, 1, 2, …)` [use sparingly! Why??]
- `L.second()`, `L.third()`, `L.fourth()`, `L.fifth()`, `L.sixth()` … do the obvious
- `L.toString()` converts the whole list to a single `String`

More Polylist Operations

- `Polylist.explode(S)` explodes string S into a Polylist of its characters.
- `L.array()` returns a Java array of the Objects in list `L`.
- `L.prefix(N)` returns the length N prefix of L.
- `range(M, N)` returns a list `(M M+1 …. N)`.
- `range(M, N, S)` returns a list `(M M+S M+2S …. N)`.

A Recursive List Pattern (without using map)

- ad-hoc map-like operations, build list **outside-in**, using recursion:

```
static Polylist scale(long factor, Polylist L)
  {
  if( L.isEmpty() )
    return Polylist.nil;

  long first = ((Long)L.first()).longValue();

  Long result = new Long(factor*first);

  return cons(result, scale(factor, L.rest()));
  }
```

unwrap

wrap

recurse

An Iterative List Pattern

- build list **inside-out**, using ordinary iteration and an accumulator

```
static Polylist scaleAndReverse(long factor, Polylist L)
  {
  Polylist result = Polylist.nil;

  for( ; L.nonEmpty() ; L = L.rest() )
    {
    long first = ((Long)L.first()).longValue();

    result = cons(new Long(factor*first), result);
    }

  return result;
  }
```

unwrap

wrap

An Iterative Reduce Pattern

- collapse list into a value using ordinary iteration

```
static long sum(Polylist L)
  {
  long result = 0;

  for( ; L.nonEmpty() ; L = L.rest() )
    {
    long first = ((Long)L.first()).longValue();

    result += first;
    }

  return result;
  }
```

unwrap

An Recursive Merge Pattern

- merge two lists of Longs in increasing order

```
static Polylist merge(Polylist L, Polylist M)
  {
  if( L.isEmpty() )
    return M;

  if( M.isEmpty() )
    return L;

  long firstL = ((Long)L.first()).longValue();

  long firstM = ((Long)M.first()).longValue();

  if( firstL <= firstM )
    return merge(L.rest(), M).cons(L.first());
  else
    return merge(L, M.rest()).cons(M.first());
  }
```

unwrap

Try this

- determine whether an Object occurs in a Polylist

```
static boolean member(Object Ob, Polylist L)
  {








  }
```

If you used recursion, try it with iteration, and vice-versa

- determine whether an Object occurs in a Polylist

```
static boolean member(Object Ob, Polylist L)
  {









  }
```

Higher-Order Functions in Polya (Advanced)

- Java does not have the notion of Higher-Order methods.
- However, it is possible to achieve the effect in a slightly round-about way.
- To create a function, say of one argument, create a Class that implements the **Function1** interface.
- Such a class needs to define the public method apply: Object -> Object

Example: An argument to map

```
class Scaler implements Function1
{
long factor;

Scaler(long factor)
  {
  this.factor = factor;
  }

public Object apply(Object Arg)
  {
  long arg = ((Long)Arg).longValue();

  return new Long(factor*arg);
  }
}
```

type expected as argument to Polylist.map

constructor

function specifying result as a function of list element

ugly code to cast arg to Long, then get wrapped value

Example: map usage

```
L.map(new Scaler(100));
```

scale factor

Function1 being mapped

List over which mapping occurs

How Lists are Implemented in Polya and rex

- Lists are an **abstraction**

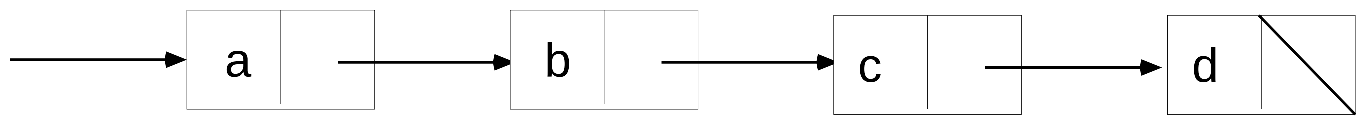
- The actual implementation manipulates objects behind the scenes, using references (really pointers).

- References are numeric values that uniquely determine the object being referenced.

The “Open List” model is used in Polya and rex

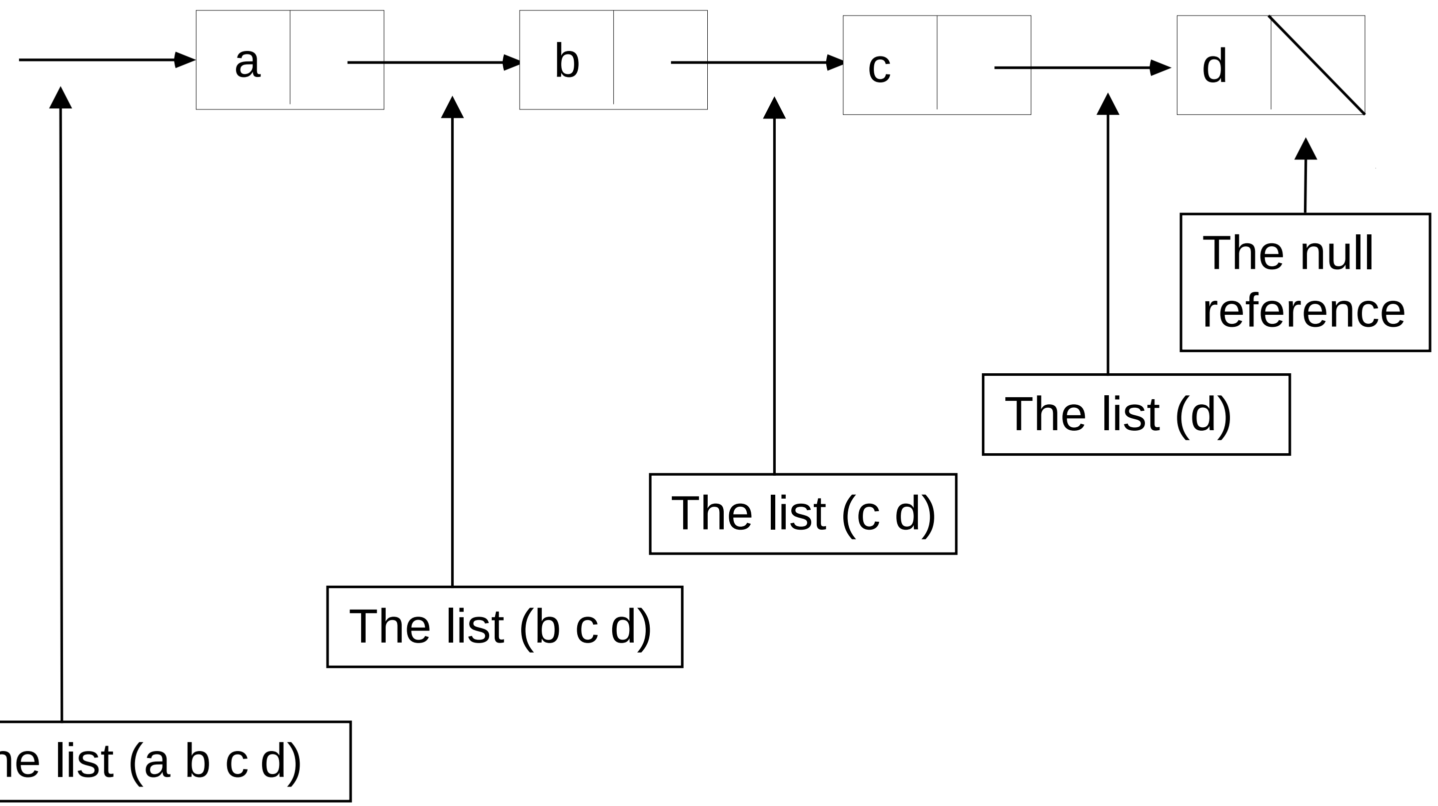
- Two list models are described in the text:
  - Open lists:
    - Elements and sublists can be shared
    - Mutation of lists is discouraged
    - Mathematically elegant
  - Closed lists:
    - Sharing generally not done
    - Mutation of lists is ok, because they are encapsulated
    - Mathematically less attractive
  - Closed lists built by wrapping open lists

An Open List



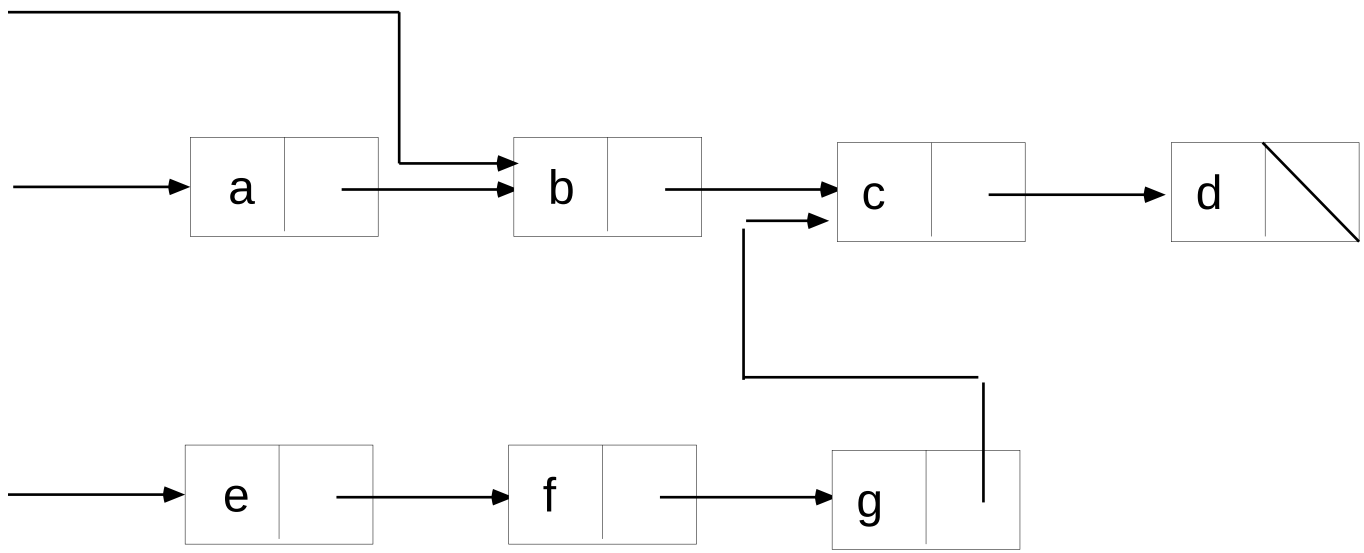
- Each list element begins a list in its own right.
- A list is **identified with** a reference to its first element.
- The empty list is identified with the null reference (or pointer).

Open Lists Identified with References



Sharing in Open Lists

Display the list identified with each reference.

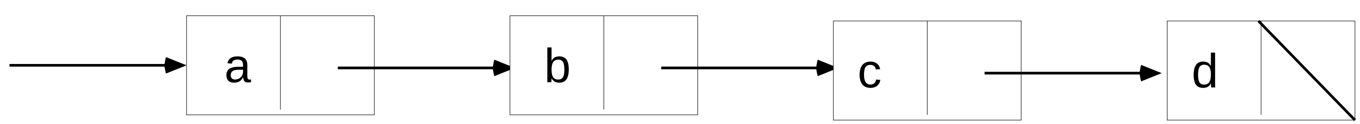


Why is list mutation discouraged?

Passing an Open List as an Argument to a Function

- To pass an open list as an argument, we simply pass a reference to its first element.

- The list is **not** literally copied.

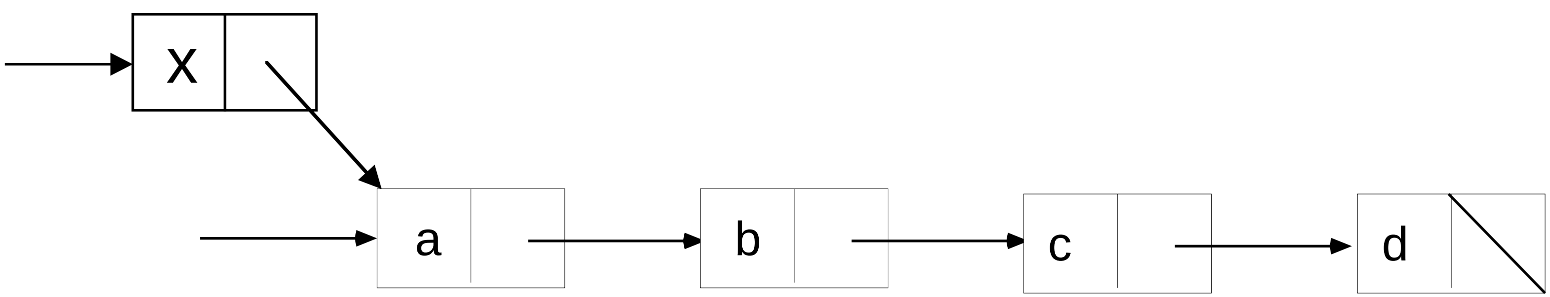


Open List Consing

- To cons an element to an open list, we simply put the element in a new cell and hook the cell to the original list:

caution: mixed notation

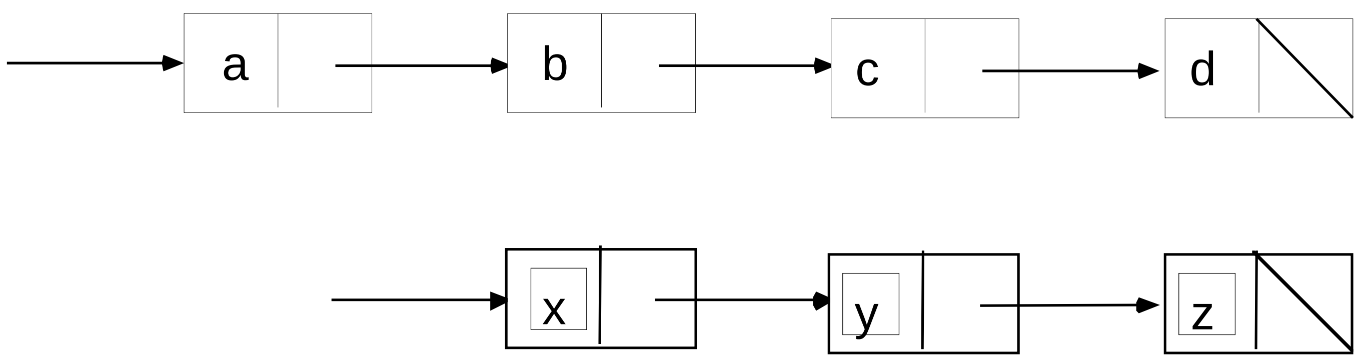
- consing x to the front yields



Appending Open Lists

- What happens when we append one open list to another, as in

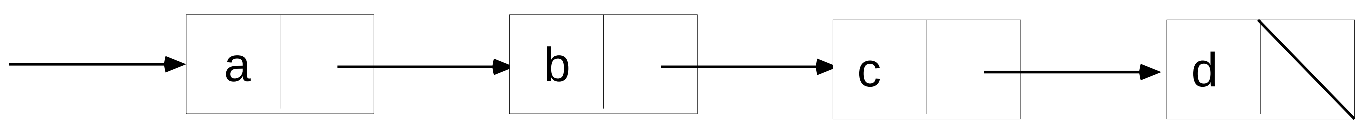
```
L.append(M)?
```



Reversing an Open List

- What happens when we reverse an open list?

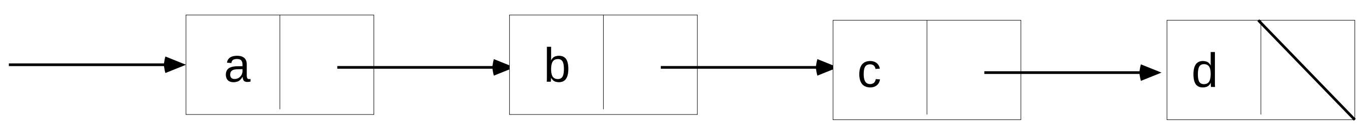
```
L.reverse()
```



Mapping an Open List

- What happens when we map an open list?

```
L.map(fun)
```



What about Trees?

- We saw early on that trees can be implemented with lists, provided that lists can have lists as elements.
- Since Polya allows any Object to be elements of a Polylist, it allows Polylists to be such elements.
- Therefore, Polya can implement trees.

Next Topics

- **Closed** Lists
- Other closed structures
  - stacks
  - queues
  - deques
  - priority queues