Distinctive Data Types

Disjoint union of types: Example: *Arithmetic Expressions*

Syntax:

$$ae ::= int \mid var \mid ae * ae \mid ae + ae$$

Type Expression:

$$EXpr = int \uplus String \uplus Expr \times Expr \uplus Expr + Expr$$

Java Encloding:

```
public enum Expr_Case {
        Const, Varm, Plus, Times}
class Expr {
 int select;
 int val;
 String var;
 Expr arg1;
 Expr arg2;}
```

Dynamic Data Types in Caml

```
type list = {hd:int; tl:list}
```

The above does not work!
(*no null values* $\Rightarrow$ *only solutions are infinite*!)
The Solution:
The disjoint union of a singleton - empty cartesian product - and of the cartesian product $int \times list$ is defined below:

```
type list = nil | Cons of int * list
```

Dynamic Data Types in C

In C, the value of such an expression of a record type is the record itself
$\Rightarrow$ there is no value null

```
struct List {
 int hd;
 struct List tl;};
```

is empty!
What is implicit in Java must be explicitly stated in C:

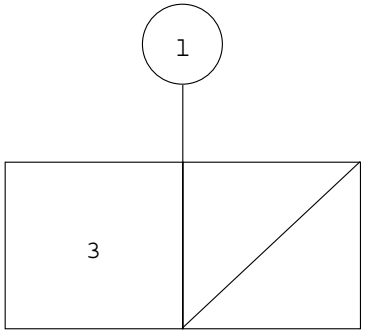
```
struct List {
 int hd;
 struct List* tl;};
```

Specific syntax for a reference (pointer)
a special reference called NULL that can never be associated in memory

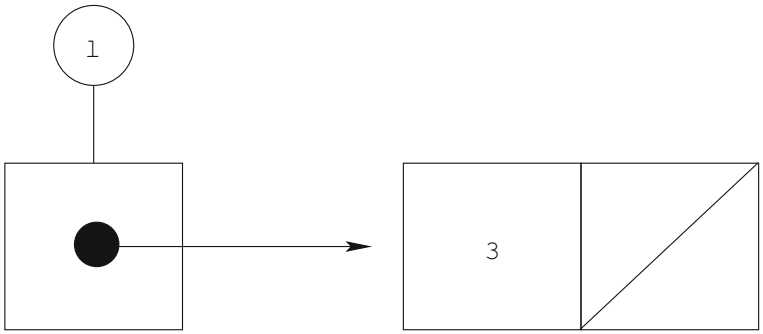
Dynamic Data Types in C - Cont.

```
Singleton List:
 struct List l = {3,NULL};
```

the state constructed in C is:

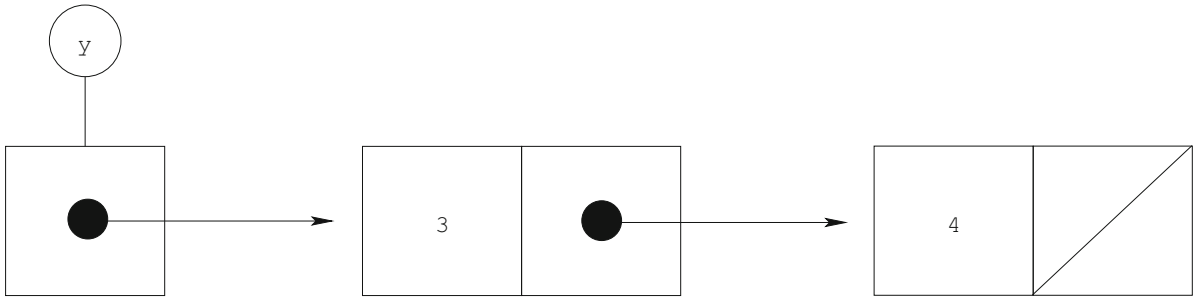


Remember in Java:

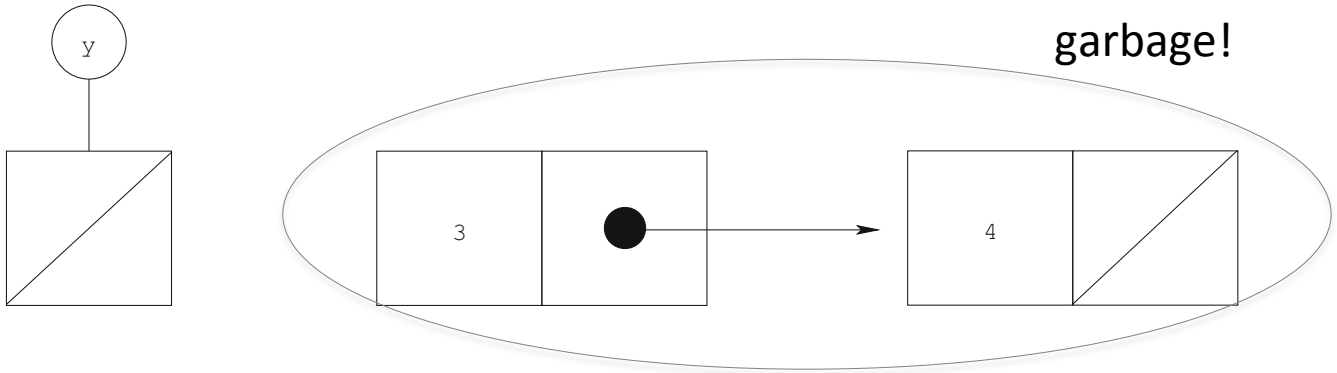


Garbage Collection

```
y = new List(3, new List(4,null));
```




```
y = null;
```



The presence or the absence of these two cells in memory has no observable effect.

Garbage Collectors take care of this!

Programming with Lists

1) Membership: $x \in l$?

```
static boolean mem (String x, List l) {
 if (l == null) return false;
 if (equal(x,l.hd)) return true;
 return mem(x,l.tl);}
```

Alt:

```
static boolean mem (String x, List l) {
 while (l != null) {
        if (equal(x,l.hd)) return true;
        l = l.tl;}
return false; }
```

Programming with Lists - Cont.

2) Concatenation: $x_1, \cdots, x_n \quad , \quad y_1, \cdots, y_n$

Suppose List is a list of characters.

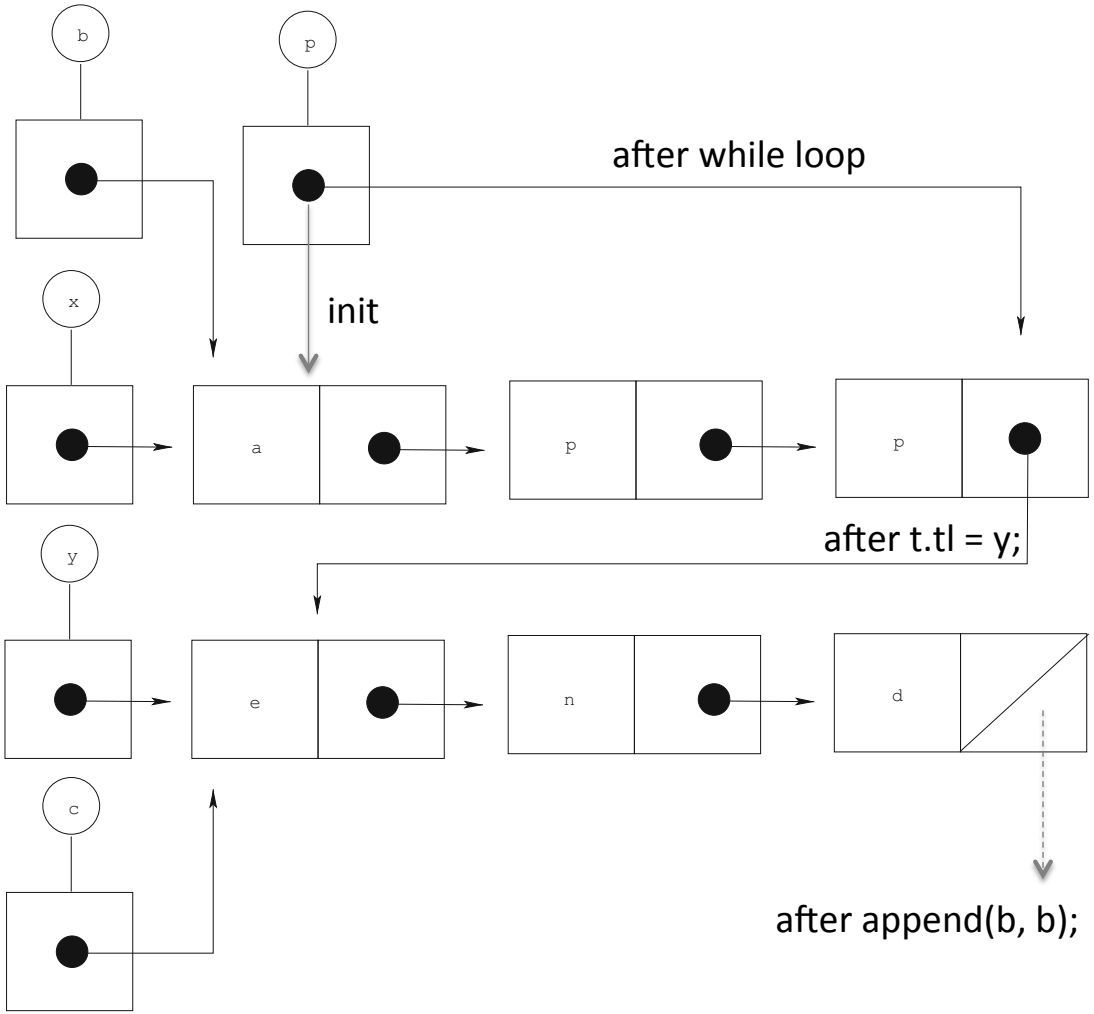
Modify "imperative"

```
static List append (List x, List y) {
 if (x == null) return y;
 if (x == null) return y;    //not needed, but nice!
 List t = x;                 //temporary
 while (t.tl != null) t = t.tl;  //walk to end
 t.tl = y;
 return x;}                  //why x?
```

Programming with Lists - Cont.

Modify - Cont.: Example

```
List b = new List( a ,new List( p ,new List( p , null )));
List c = new List( e ,new List( n ,new List( d , null )));
```



Now, Consider
append(b, b)

Benefit: no allocation
~: modify arguments

Programming with Lists - Cont.

2) Concatenation: Cont.

Copy

```
static List copy (List x) {
 if (x == null) return x;
List p = x;
  List q = new List(x.hd,null);
  List r = q;
  while (p.tl != null) {
   q.tl = new List(p.tl.hd,null); q = q.tl;p = p.tl;}
  return r;}

static  List copyAppend(List x,List y) {
    append(copy(x),y); }
```

Not efficient: traverses x in copy(x)

Programming with Lists - Cont.

Copy - Cont. How to do better?

1. inline code of copy into *copyAppend* to get access to q [book does this]
2. get a hold of q!

```
class Pair {
    public final List left;
    public final List right;
    public Pair(List x, List y) {
        this.left=x;
        this.right=y; }
}
```

Change some lines of *copy* as follows:
1st: *static Pair copy'(List x)* last: *return Pair(r, q);* Now:

```
copyAppend {
    Pair p = copy(x);
    p.right.tl=y;
    return p.left; }
```

Programming with Lists - Cont.

2) Concatenation: Cont.

Using Recursion

```
static List append(List x, List y) {
if(x == null) return y;
 return new List(x.hd, append(x.tl,y));}
```

Programming with Lists - Cont.

3) List Inversion (extra arguments): $x_1, \cdots, x_n \rightsquigarrow x_n, \cdots, x_1$

```
static List reverse (final List x) {
 if (x == null) return null;
 return add(reverse(x.tl),x.hd);}

static List add (final List x, final int y) {
 if (x == null) return new List(y,null);
 return new List(x.hd,add(x.tl,y));}
```

The complexity: $O(n^2)$!!! The linear time method is as follows:

```
static List revappend (final List x, final List y) {
 if (x == null) return y;
 return revappend(x.tl,new List(x.hd,y));}

static List reverse (final List x) {
 return revappend(x,null);}
```

Lists and Arrays

Lists allow for simpler programs, but *arrays allow* for more efficient ones.