CSE 505

Lecture #5

September 12, 2012

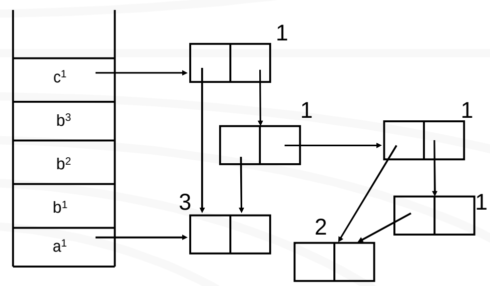
For a discussion of shallow and deep copy in several languages, please visit the Wikipedia page entitled

'Object Copy'.

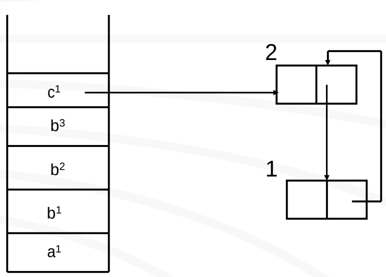
Heap Storage Management

- Reference Counting
  - each heap cell maintains an integer count
  - initialize to 1
  - increment and decrement upon pointer assignment/re-assignment
  - reclaim cell when count reaches 0
  - cannot reclaim circular structures
  - slows down program due to reference mgmt

Ripple effect when count becomes 0



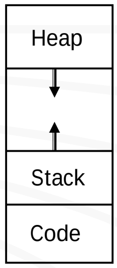
Reference Counting Cannot Reclaim Circular Structures



Heap Storage Management

- Mark-scan Garbage Collection
  - No reference counting
  - When memory runs low, traverse heap objects from stack and mark all reachable objects
  - Sequentially scan memory and reclaim un-marked memory cells
  - Can reclaim circular structures
  - Time taken is inversely proportional to amount of garbage

Stack – Heap Spaces



Unix limit command

```
nickelback % limit

cputime       unlimited
filesize      unlimited
datasize      unlimited
stacksize     10240 kbytes
coredumpsize 0 kbytes
memoryuse     unlimited
vmemoryuse    unlimited
descriptors  1024
memorylocked 32 kbytes
maxproc       200
```


How much recursion with 10MB stack?

```
#include <stdio.h>

void recurse() {
    int data[262144];          // 1 MB
    recurse();
}

int main(){
    recurse();
}
```

Can change setting: % unlimit stacksize

Testing heap allocation

```
#include <stdio.h>
#include <stdlib.h>

typedef struct list {      // will take about 1MB of storage
      int data[262144];
      struct list *next;
      } LIST;

int main(){
   while (1) {
   LIST* node = (LIST*) malloc(sizeof(LIST));
   if (node == (LIST*) NULL) {
      printf("Ran out of heap space\n");
      exit(1);
   }
   printf("%d\n", sizeof(LIST));
   }
}
```

Higher-order constructs can simulate advanced control structures.

Iterators
Another Use of Procedure Parameters

Motivation: Modular Iteration over Structured Data

Array traversal:
    for (i = 1; i <= n; i++) { ... print a[i] ... }

Similarly, we would like to do List traversal:
    foreach x in elements(l) do { ... print x ... }

And also Tree traversal:
    foreach x in elements(t) do { ... s := s + x ... }

Different Tree Traversals

```
foreach x in elements_preorder(t) do {
        ... print x ...
}
```

```
foreach x in elements_postorder(t) do {
        ... print x ...
}
```

```
foreach x in elements_inorder(t) do {
        ... print x ...
}
```


Two Types of Iterators

- Internal Iterators
  - based upon the 'foreach-do' contructs
  - used in Smalltalk, CLU, ... Python
- External Iterators
  - while (not done(iter)) {
    next(iter) ... advance(iter) ...
    }
  - used in Java, ...

Iterators: Definition and Use

Defining an Iterator:

(i) **iterator** result_type name(formals_pars) { iterator_body }
(ii) **yield** value;
(iii) **return;**

Using an Iterator:

(iv) **foreach** control_variable **in** iterator_call **do** foreach_body;
(v) **exit**;

Defining an Iterator

```
iterator int mystery1() {
  int f1 =1;
  int f2 = 1;
  yield f1;
  while(true) {
        yield f2;
        (f1, f2) := (f2, f1+f2);
                        // Python's parallel assignment
  }
}
```

foreach x in mystery1() do { ... print x ... }

Defining a List Iterator

```
class List { int val;
             List next; }

iterator int elements(List l) {
        while(l != nil) {
                yield l.val;
                l := l.next;
        }
}
```

call-by-value parameter passing.

Using a List Iterator

```
mystery() {
        List l1, l2;
        ... initialize l1 to '(10 20 30 40) ...

        l2 := nil;
        foreach x in elements(l1) do {
                l2 := cons(x, l2);
        }
}
```

Translating Iterators using Procedure Parameters

main:

```
List l1, l2;   ...
l2 := nil;
foreach x in elements(l1) do {
        l2 := cons(x, l2);
        }
print l2;
```

```
iterator int elements(List l) {
        while(l != nil) {
                yield l.val;
                l := l.next
        }
}
```

Goal: We need to replace iterator by a void procedure, and need to find replacements for foreach and yield.

Translating Iterators using Procedure Parameters

```
thk(int x) {
    l2 := cons(x, l2);
}
```

```
List l1, l2;   ...
l2 := nil;
[redacted] elements(l1) [redacted] {
[redacted]
        }
print l2;
```

thk

```
[redacted] elements(List l) {
        while(l != nil) {
                [redacted]
                l := l.next
        }
}
```

thk

Translated Program

main:

```
List l1, l2;
void thk(int x) {
        l2 := cons(x, l2);
}
l2 := nil;
elements(l1, thk);
print l2;
```

```
void elements(List l, void thk(int)) {
        while (l != nil) do {
                thk(l.val);
                l := l.next;
        }
```

l2 is known by static scope

Tree Iterator

```
class Tree { val: int,   left: Tree,   right: Tree}

iterator int elements(Tree t) {
if (null(t)) return;
  else { foreach i:int in elements(t.left) { yield i; }
          yield t.val;
          foreach i:int in elements(t.right) { yield i;}
       }
}
```

Recursive Iterator!

In Python Syntax

```
def elements(t):
 if t:
        for x in elements(t.left):
             yield x
        yield t.val
        for x in elements(t.right):
             yield x
```

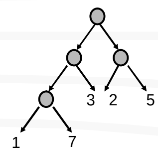

Tree Iterator – translation

```
class Tree { val: int,   left: Tree,   right: Tree}

void elements(Tree t,  void fb(int)) {
   void fb1(int i) { fb(i);}
   void fb2(int i) { fb(i);}
   if (null(t)) return;
   else { elements(t.left, fb1);
          fb(t.val);
          elements(t.right, fb2);
        }
}
```

Tree Iterator – optimized translation

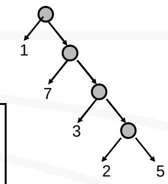
```
class Tree { val: int,   left: Tree,   right: Tree}

void elements(Tree t,  void fb(int)) {
     if (null(t)) return;
    else { elements(t.left,  fb);
             fb(t.val);
             elements(t.right,  fb);
         }
}
```


Limitations of Iterators – The "Same Fringe" Problem



Tree1

Tree1 and Tree2 have the same "fringe," [1,7,3,2,5]



Tree2

Iterators vs Coroutines

Internal Iterators

```
foreach x in elements(t) do {
        ....
}
```

External Iterators (Coroutines)

```
e := create elements(t);
while (not done(e)) {
        ... x := resume(e) ...
};
terminate(e);
```


Coroutines

Coroutine Body

(i) **coroutine**
name(*formals_pars*) { *coroutine_body* }
(ii) **suspend**; or **suspend** *value*;

(iii) **c := create** *name*(*actual_pars*);
(iv) **done c;**
(v) **resume c**; or **v := resume c**;
(vi) **terminate(c);**

Caller

Iterators vs Coroutines

Definition

- Iterator must return a value; a coroutine need not.
- Both iterators and coroutines are similar in having parameters and bodies. The yield of an iterator is essentially same as that of a suspend of a coroutine, except that a suspend need not return a value.

Use

- An iterator must be invoked in a foreach statement, which is responsible for creating an instance of the iterator and resuming it after every iteration of the foreach loop.
- Coroutine creation, resumption, and test for completion are de-coupled. Hence coroutines offer more freedom.
- Coroutine helps define an external iterator.

Continuations

- The concept of "continuation" was introduced in the Scheme programming language to capture the current execution point
- We may return to this concept after examining higher-order functions.

Parameter Passing for Structured Types

- The approach for arrays, records, etc. is similar to that of simple types such as int, real, etc.
- Parameters of interest: value, result, value-result, reference.
- Object-binding schemes of interest: quasi-dynamic and fully-dynamic. (Static variables are usually not passed as parameters.)

Quasi-Dynamic Variables

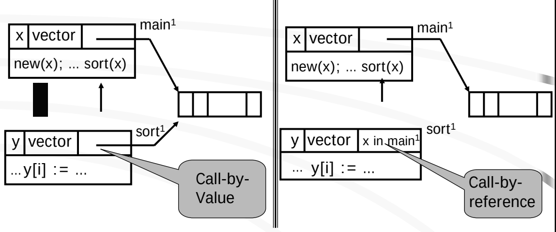
- type vector = int[100]; ...
  type rational = class {int numr, int denr}; ...
- vector x; rational r;
  ... sort(x); ... normalize(r); ...

Quasi-dynamic Storage Allocation

- sort(var vector a) { ... }
- normalize(inout rational r) { ... }
- Value, result, value-result involve copying contents.
- Call-by-reference preferred for large arrays.

Fully-Dynamic Variables

- type vector = int[100] ↑; ... vector x;
- sort(vector y) { ... } sort(var vector y) { ... }

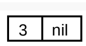


Brief Excursion into Lisp

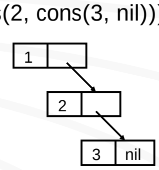
- Lisp is expression-oriented (or functional) language with good support for list processing.
- Lisp has higher-order functions, i.e., function parameters.
- Common Lisp uses static scoping.
- Common Lisp has a rich collection of primitives, and advanced features: objects, packages, and meta-level constructs.

Lisp uses cons for building lists

e.g. cons(3, nil)



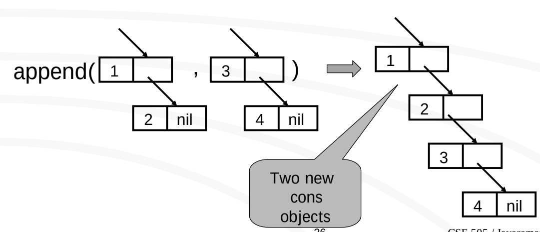
e.g. cons(1, cons(2, cons(3, nil)))



List Concatenation

append(cons(1, cons(2, nil)) , cons(3,cons(4,nil)))

⇒ cons(1,cons(2,cons(3,cons(4,nil))))



Definition of append

```
List append(List l1, l2) {
        if (l1 = nil)  return l2;
        else  return cons(l1.val,
                          append(l1.next, l2)
                         );
}
```

class List { int val; List next; }

Lisp's "Cambridge Prefix" syntax

f(x, y, z) ⇒ (f x y z)

Lisp syntax

cons(3, nil) ⇒ (cons 3 nil)

cons(1, cons(2, cons(3, nil))) ⇒

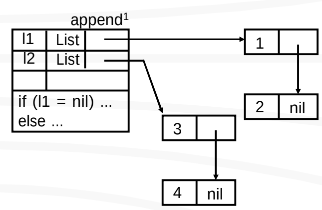
(cons 1 (cons 2 (cons 3 nil)))

Defining append in Lisp

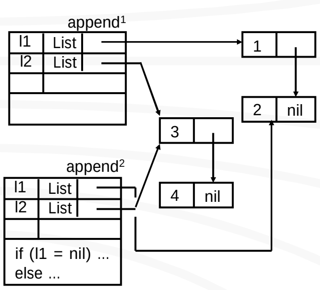
```
List append(List l1, l2) {
        if (l1 = nil)  return l2;
        else  return cons(l1.val,
                          append(l1.next, l2));
}
```

```
(defun append (l1 l2)
       (if  (null l1)
            l2
            (cons (first  l1)
                  (append  (rest l1)  l2))))
```

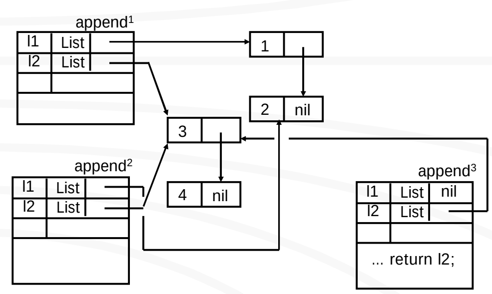

Execution of append



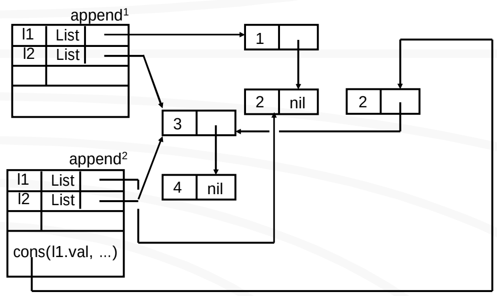
Execution of append (cont'd)



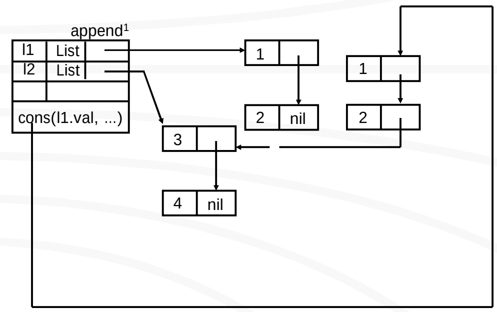
Execution of append (cont'd)



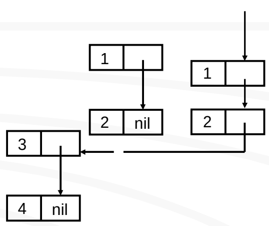
Execution of append (cont'd)



Execution of append (cont'd)



End of execution of append



Observations on Lisp

- Weakly typed language. This is legal:

  (append ‘(1 2) ‘(3 4))
  (append ‘(apple orange grape) ‘(pear))
  (append ‘(1 2.5) ‘(3 banana))

- Literal constants for lists:
  `(1 2)
  ‘(apple orange grape)
  ...

Observations on Lisp (cont'd)

Lisp is an expression-oriented language:

(append ‘(apple orange) ‘(grape peach))

(cons ‘apple (append ‘(orange) ‘(grape peach)))

(cons ‘apple (cons ‘orange (append nil ‘(grape peach))))

(cons ‘apple (cons ‘orange ‘(grape peach)))

‘(apple orange grape peach)

Higher Order Functions in Lisp

```
(defun map (F L)
       (if (null L)
           nil
           (cons (funcall F (first L))
                 (map F (rest L)))
       ))
```

> (map #'sqrt '(1 4 9 16 25))
(1.0 2.0 3.0 4.0 5.0)

> (map (lambda (x) (+ x 1)) '(1 2 3 4 5))
(2 3 4 5 6)

Higher Order Functions in Lisp

```
(defun map (F L)
        (if (null L)
            nil
            (cons (funcall F (first L))
                     (map F (rest L)))
        ))
```

```
> (map  #'sqrt  '(1 4 9 16 25))
        (1.0 2.0 3.0 4.0 5.0)

> (map  (lambda (x) (+ x 1))   '(1 2 3 4 5))
       (2 3 4 5 6)
```


Functions as results (Python)

```
def linear(a, b):
        def f(x):
                return a*x + b
        return f
```

The function linear returns another function (f), which when applied to argument (for x) will compute the result a*x + b.
Using linear:

```
g = linear(3, 4);
print g(10);
```


Functions as results (ML)

```
fun map(f) =
        let g(l) = if null(l) then []
                        else cons(f(first(l)), g(rest(l)))
        in  g
        end;
fun square(x) = x*x;
fun cube(x) = x*x*x;
...
val h = map(cube);
...
... h([1,2,3,4,5]) ...
```


Functions as results (Javascript)

```
<SCRIPT language="JavaScript">
var map = function(f){
            return function(a) {
                        var b = new Array();
                        for( var i=0; i <= a.length; i++ )
                                { b[i] = f(a[i]); }
                         return b;
            }};

var nums = [1,2,3,4,5];
var square = map( function(x){return x*x;} );
var answer = square(nums);

for( var i=0;  i <= answer.length;  i++ )
        { document.write(answer[i] + ' ');  }

</SCRIPT>
```

Functions as results (Javascript)

```
<SCRIPT language="JavaScript">

var double = function(x) { return x*2; }
var halve   = function(x) { return x/2; }
var triple  = function(x) { return 3*x; }

var comp = function(f){
                return function(g){
                        return function(x){ return f(g(x)); }
        }}

var f1 = comp(triple);

var f2 = f1(halve);
var f3 = f1(double);

document.write('comp(triple)(halve)(10) = ' + f2(10) + '<br><br>');
</SCRIPT>
```


Lambda Calculus

Expressions are made of variables, abstractions, and applications:

Expr ::= Var |
λ Var . Expr |
(Expr Expr)

Alonzo Church (1903-95)

Church visits Buffalo

- Alonzo Church visited Buffalo in May 1990 to receive an Honorary Doctorate
- One-day celebration in honor of his visit
- He gave a lecture on “A Theory of the Meaning of Names”
- Many of his PhD students – now famous scientists in their own right – also attended.

Examples of lambda terms

- λ x . x
- λ f . λ x . x
- λ f . λ x . (f (f x))
- λ f . λ g . λ x . (f (g x))
- ...

Resemble “anonymous functions”

Relation to Functions

λ-Calculus: λf . λx . (f (f x))

Lisp: (lambda (f) (lambda (x) (f (f x))))

Javascript:

```
function f {return
            function (x) { f(f(x)); }
       }
```

Computation = β-Reduction

($\underline{(\lambda f . \lambda x . (f\ (f\ x))\quad \lambda x . x)}$ a)

⇒ $\underline{(\lambda x . (\lambda x . x\ (\lambda x . x\ \ x))\ a)}$

⇒ (λ x . x $\underline{(\lambda x . x\ a)}$)

⇒(λ x . x a)

⇒ a

Another β-Reduction

($\underline{(\lambda f . \lambda x . (f\ (f\ x))\quad \lambda x . x)}$ a)

⇒ (λ x . (λ x . x $\underline{(\lambda x . x\ \ x) }$) a)

⇒ (λ x . $\underline{(\lambda x . x\ x)\ }$ a)

⇒(λ x . x a)

⇒ a

Yet Another β-Reduction

($\underline{(\lambda f . \lambda x . (f\ (f\ x))\quad \lambda x . x)}$ a)

⇒ (λ x . $\underline{(\lambda x . x\ (\lambda x . x\ \ x))\ }$ a)

⇒ $\underline{(\lambda x .\ (\lambda x . x\ x)\ a)}$

⇒(λ x . x a)

⇒ a

Confluence Property

"If a lambda term T reduces to two terms T1 and T2, then T1 and T2 can be reduced to a common term U."

$$\begin{tikzcd}
 & T \arrow[dl, "*"'] \arrow[dr, "*"] & \\
T_1 \arrow[dr, "*"'] & & T_2 \arrow[dl, "*"] \\
 & U &
\end{tikzcd}$$

Normal Form

If a term T reduces to a term U, and U cannot be reduced any further (by β-reduction), then U is said to be in normal form.

Normal Form: "The normal form of a term is unique (up to renaming of bound variables), if the normal form exists."

Renaming bound variables:

$\lambda\, x\,.\,x = \lambda\, y\,.\,y$

$\lambda\, f\,.\,\lambda\, x\,.\,(f\ \ x) = \lambda\, g\,.\,\lambda\, y\,.\,(g\ \ y)$

Nontermination is Posssible!

$(\ \underline{\lambda\, x\,.\,(x\ \ x)\quad \lambda\, x\,.\,(x\ \ x)\ )}$

$\Rightarrow$

$(\ \underline{\lambda\, x\,.\,(x\ \ x)\quad \lambda\, x\,.\,(x\ \ x)\ )}$

$\Rightarrow$

$(\ \underline{\lambda\, x\,.\,(x\ \ x)\quad \lambda\, x\,.\,(x\ \ x)\ )}$

$\Rightarrow$

...