CSE 505

Lecture #6

September 19, 2012

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.)

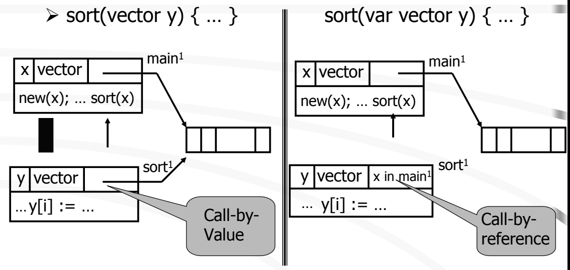
Quasi-Dynamic Variables

- 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) { … }

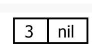


Brief Excursion into Lisp

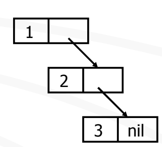
- Lisp is expression-oriented (or functional) language with good support for list processing.
- Lisp has higher-order functions, i.e., function parameters.
- Common Lisp uses static scoping.
- Common Lisp has a rich collection of primitives, and advanced features: objects, packages, and meta-level constructs.

Lisp uses cons for building lists

e.g. cons(3, nil)



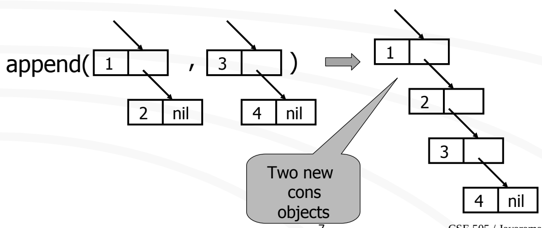
e.g. cons(1, cons(2, cons(3, nil)))



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))) ⇒

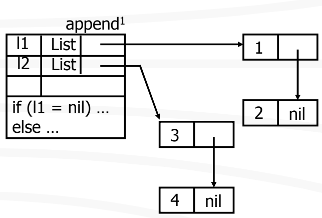
(cons 1 (cons 2 (cons 3 nil)))

Defining append in Lisp

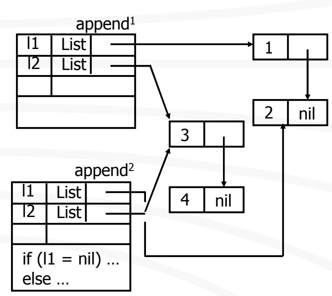
```
List append(List l1, l2) {
        if (l1 = nil) return l2;
        else return cons(l1.val,
                         append(l1.next, l2));
}
```

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

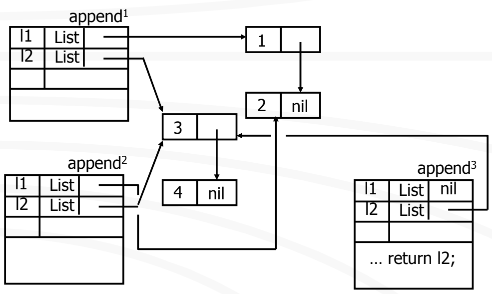

Execution of append



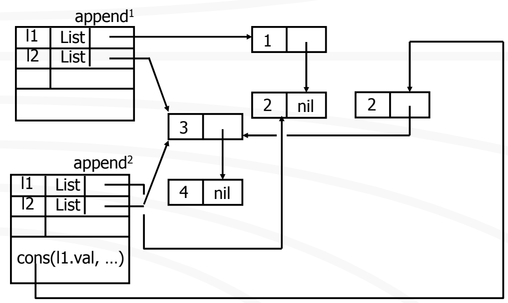
Execution of append (cont'd)



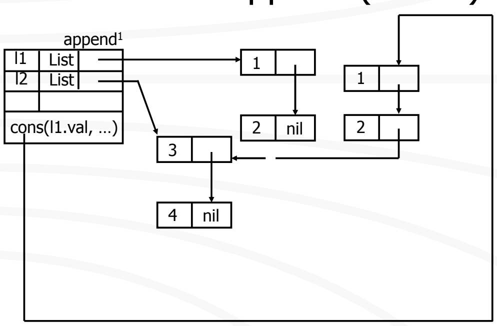
Execution of append (cont'd)



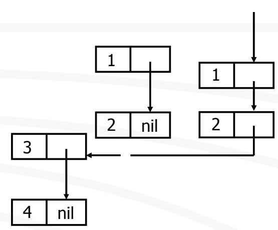
Execution of append (cont'd)



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)
```


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

The quintessential higher-order language, with functions as input and output. In fact, data are also encoded as functions!

World's smallest PL ☺:

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

Inventor: Alonzo Church

- Church visited Buffalo in May 1990.
- Received an Honorary Doctorate
- One-day celebration in honor of his visit, attended by several famous logicians.
- Church spoke on a "Theory of the Meaning of Names"

Alonzo Church (1903-95)

A Theory of the Meaning of Names

by

ALONZO CHURCH

For the name relation, i.e., the relation between a name and what it is a name of, the standard words in ordinary English are the verb to denote and the noun denotation, going back at least to John Stuart Mill[1] and perhaps earlier.[2] We shall follow this in the present paper, allowing the verb to designate and the noun designation as occasional alternative terminology, but certainly not to refer to and reference, which are

Examples of lambda terms

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

Sometimes called "anonymous functions"

Don't over-do parentheses

- In ordinary arithmetic, x = (x) = ((x)) = …
- This is incorrect in lambda calculus.
- In lambda calculus, (T1 T2) is the application of function T1 to argument T2.
- Thus, (x) is not syntactically correct.

Informal Meaning

- λx. x
  - ➔ identity function
- λx. λy. x
  - ➔ a function of two parameters that returns the first parameter
- λf. λg. λx. (f (g x))
  - ➔ the composition of two functions, f ∘ g

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)); }
           }
```

Bound and Free Occurrences

$\lambda x.(\underline{x}\ \ \underline{y})$

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

$\lambda x.\ (\lambda y.\ (\lambda x.(\underline{z}\ \ \underline{y})\ \ \underline{x})\ \ \underline{x})$

Definition of free (T)

free(V) = {V}, where V is variable

free(λV.T) = free(T) − {V}

free((T1 T2)) = free(T1) U free(T2)

Another def'n of free variables

(from Lecture Notes)

V occurs_free_in W iff V = W

V occurs_free_in λW.T iff V ≠ W and V occurs_free_in T

V occurs_free_in (T_1 T_2) iff V occurs_free_in T_1 or V occurs_free_in T_2

Substitution

(will be used for parameter passing)

Substitution of all free occurrences of a variable V by term T1 in term T2:

T2 [V ← T1]

e.g. $\lambda x.\ (\underline{f}\ \ (\underline{f}\ \ \underline{x}))\ [\ f \leftarrow \lambda y.y\ ]$

$= \lambda x.\ (\lambda y.y\ \ (\lambda y.y\ \ \underline{x}))$

Note: $\lambda x.\ (\underline{f}\ \ (\underline{f}\ \ \underline{x}))\ [\ x \leftarrow y\ ]$

$\neq \lambda x.\ (\underline{f}\ \ (\underline{f}\ \ \underline{y}))$ -- since x is bound

Substitution (cont'd)

$\lambda x.\ (\underline{f}\ \ (\underline{f}\ \ \underline{x}))\ [\ f \leftarrow \lambda y.(y\ x)\ ]$

$\neq \lambda x.\ (\lambda y.(y\ x)\ \ (\lambda y.(y\ x)\ \ \underline{x}))$

This is called the "variable capture" problem.

Correct way to do the substitution:

$\lambda x'.\ (\lambda y.(y\ x)\ \ (\lambda y.(y\ x)\ \ \underline{x'}))$

Renaming Bound Variables

Renaming the binder variables is always permissible – similar to renaming the formal parameters of a function. Thus:

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

$\lambda x.\ (\underline{f}\ \ (\underline{f}\ \ \underline{x})) = \lambda x'.\ (\underline{f}\ \ (\underline{f}\ \ \underline{x'}))$

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

Reduction Rules

Three famous reduction rules: α, β, η

α-reduction is renaming of binder variables – it doesn't really "reduce" the term.

β-reduction resembles call-by-name, and is based on the substitution rule:

$$(\lambda V.T1 \quad T2) \rightarrow_{\beta} T1\ [\ V \leftarrow T2\ ]$$

η-reduction is not so common:

$$\lambda V.(T \quad V) \rightarrow_{\eta} T \quad \text{if } V \notin \text{free}(T)$$

Examples of β-reduction

Ex 1: $(\lambda x.x \ \ a) \Rightarrow x\ [\ x \leftarrow a\ ] = a$

Ex 2: $(\ (\lambda f.\ \lambda x.\ (f\ \ (f\ \ x))\quad \lambda x.\ x)\quad a)$

Ex 3: $(\lambda f.\ \lambda x.\ (\underline{f}\ \ (\underline{f}\ \ \underline{x}))\quad \lambda y.(y\ x)\ )$

Computation = β-Reduction

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

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

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

$\Rightarrow (\lambda x.x\ \ a)$

$\Rightarrow a$

Another β-Reduction

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

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

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

$\Rightarrow (\lambda x.\ x\ \ a)$

$\Rightarrow a$

Yet Another β-Reduction

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

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

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

$\Rightarrow (\lambda x.\ x\ \ a)$

$\Rightarrow a$

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."

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

Unique Normal Form

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

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

Proof by Contradiction

Suppose T has two normal forms N_1 and N2:

$$T \rightarrow^* N_1 \text{ and } T \rightarrow^* N_2$$

By Confluence Property,

$$N_1 \rightarrow^* U \text{ and } N_2 \rightarrow^* U$$

But N1 and N2 are irreducible, hence must be the same except for alpha-reductions, i.e., variable renaming.

Nontermination is Posssible!

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

$\Rightarrow$

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

$\Rightarrow$

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

$\Rightarrow$

…

Leftmost Reductions

- How should we reduce a term in order that the normal form can be derived, if it exists?
- Answer: Choose the leftmost "redex" at every step.

- Let Ω = (λx . (x x) λx. (x x))
- Then, (λx.a Ω) $\rightarrow$ a, by leftmost reduction
- A nonterminating reduction sequence is:
  (λx.a Ω) $\rightarrow$ (λx.a Ω) $\rightarrow$ …

Different Reduction Orders

- Leftmost Innermost
- Parallel Innermost
- Rightmost Innermost
- Parallel Outermost
- Leftmost Outermost = Leftmost

Church-Rosser Property

Defn: T1 <==> T2 if T1 ==> T2 or T2 ==> T1, where ==> uses one of the three reduction rules.

Defn: T1 <==>* T2 uses <==> 0 or more times.

Church-Rosser Property: If T1 <==>* T2 then there is a term U s.t. T1 ==>* U and T2 ==>* U.

Diagram:

$$T1 \Longleftrightarrow U_1 \Longleftrightarrow \ldots\ldots \Longleftrightarrow U_n \Longleftrightarrow T2$$

$$T1 \xrightarrow{*} U \xleftarrow{*} T2$$

Relation between <==>* and ==>*

Note: T1 <==>* T2 does NOT imply T1 ==>* T2 or T2 ==>* T1.

In reality, given T1 <==>* T2, the situation is:

$$
\begin{tikzcd}[column sep=small]
T1 \arrow[dr, "*"'] & \Longleftrightarrow^{*} & P_1 \arrow[dl, "*"] \arrow[dr, "*"'] & \Longleftrightarrow^{*} & P_2 \arrow[dl, "*"] & \ldots & P_n \arrow[dr, "*"'] & \Longleftrightarrow^{*} & T2 \arrow[dl, "*"] \\
 & V_1 & & V_2 & & & & V_n &
\end{tikzcd}
$$

Confluence implies Church-Rosser

Proof (informal):

$$
\begin{tikzcd}[column sep=small]
T1 \arrow[dr, "*"'] & \Longleftrightarrow^{*} & P_1 \arrow[dl, "*"] \arrow[dr, "*"'] & \Longleftrightarrow^{*} & P_2 \arrow[dl, "*"] & \ldots & P_n \arrow[dr, "*"'] & \Longleftrightarrow^{*} & T2 \arrow[dl, "*"] \\
 & V_1 \arrow[dr] & & V_2 & & & & V_n \arrow[dl] & \\
 & & \ldots \arrow[drr] & & & & \ldots \arrow[dll] & & \\
 & & & & U & & & &
\end{tikzcd}
$$

Church-Rosser implies Confluence

Proof (easy): Given:

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

Therefore: T_1 <==>* T_2

Therefore:

$$
\begin{tikzcd}
{} \arrow[dr, "*"] & & {} \arrow[dl, "*"'] \\
 & U &
\end{tikzcd}
$$

by Church-Rosser