

**CS322 W'05 Lecture Notes**  
**Lecture 9.1**

## Procedures as Parameters

It can be handy to pass **procedures** as parameters to other, **higher-order** procedures. This feature is supported by many languages, including Pascal, Ada, ML, and C/C++ (but not directly by Java).

Examples:

- Parameterized algorithms (e.g. in C):

```
typedef int (* leqfn) (int, int);

void isort(int n, int a[], leqfn leq) {
    int i, j, t;
    for (i = n-1; i >= 0; i--) {
        t = a[i];
        for (j = i;
             j < n-1 && leq(a[j+1], t);
             j++)
            a[j] = a[j+1];
        a[j] = t;
    }
}

int up(int p, int q) { return p <= q; }
int down(int p, int q) { return p >= q; }

int a[] = {2, 1, 3};
isort(3, a, up); /* a = {1, 2, 3} */
isort(3, a, down); /* a = {3, 2, 1} */
```

## Procedures as Parameters (cont.)

- Call-backs from surrounding system:

```
typedef void (* click_handler)(int);

void handler(int button) {
    switch(button) {
        case 1: cut();
        case 2: copy();
        case 3: paste();
    }
}

registerClickHandler(handler);
```

Standard implementation: Just pass a pointer to the first instruction of the procedure.

## Procedures as Parameters (cont.)

- Parameterized data structure traversals (e.g. in ML)

```
fun map (g: int -> int, u: int list)
      : int list =
  let fun f (v : int list) : int list =
      case v of
        nil => nil
      | (h::t) => (g h)::(f t)
  in f u
end
```

```
fun add3 x = x + 3
fun sub1 x = x - 1
val w = map (add3, [1,2,3])
      (* yields [4,5,6] *)
val z = map (sub1, [1,2,3])
      (* yields [0,1,2] *)
```

ML also supports **anonymous function** values, i.e., functions that can be defined without being named. Could do above example as:

```
val w = map (fn x => x + 3, [1,2,3])
val z = map (fn x => x - 1, [1,2,3])
```

## Using Local (Nested) Procedures

- Sometimes want to pass **local** functions as parameters.

```
fun meandeltas (u : int list) : int list =  
  let val mean : int = compute_mean u  
      fun compute_delta (x:int) = x - mean  
  in map (compute_delta,u)  
  end
```

- Lexical scope rules apply, so function body can use data associated with outer function.
- Here `compute_delta` uses the value of `mean`, which is local to `meandeltas`.
- Cannot express this in C/C++/Java, which have no nested functions.
- Possible implementation: pass **pair** of (code-pointer, access-link) as “**value**” of procedure.
- Depends on fact that access link is still valid when procedure is called!

## More Nested Procedures

Another example:

```
fun deltas(n:int, u:int list) =  
  let fun compute_delta (x:int) = x - n  
      in map (compute_delta,u)  
      end
```

```
...deltas(3,[1,7,5]) (* yields [~2,4,2] *) ..
```

- Here `compute_delta` depends on the value of variable `n`, which is a local parameter of `deltas`.

What if we want to compute `deltas` on a several different lists with a fixed `n`?

- Can be handy to treat **procedure values** just like other values, e.g., to **return** them as function results or **store** them into variables.

**“First-class” Procedures Example**

```

fun deltas' (n:int) : int list -> int list =
  let fun deltas (u : int list) : int list =
        let fun compute_delta (x:int) : int = x - n
            in map (compute_delta,u)
        end
  in deltas
end

```

```

val g : int list -> int list = deltas' 3
...
val x : int list = g [1,7,5] (* yields [~2,4,2] *)
val y : int list = g [2,4,6] (* yields [~1,1,3] *)

val z : int list = deltas' 3 [2,4,6]
                    (* yields [~1,1,3] *)

```

ML also provides syntactic sugar to make such “**Curried**” functions easier to write. Above program is equivalent to:

```

fun deltas' (n:int) (u:int list) : int list =
  let fun compute_delta (x:int) = x - n
      in map (compute_delta,u)
  end

```

## Using Curried functions

- When defining “multi-argument” functions in ML, have a choice using a tuple argument and Currying.
- Can apply Curried version `deltas'` to either one or two arguments.
- Function application associates to the **left**, so

```
deltas' 3 [2,4,6] =
  (deltas' 3) [2,4,6]
```

- Function type arrows associate to the **right**, so the type of `deltas'` is

```
int -> int list -> int list =
  int -> (int list -> int list)
```

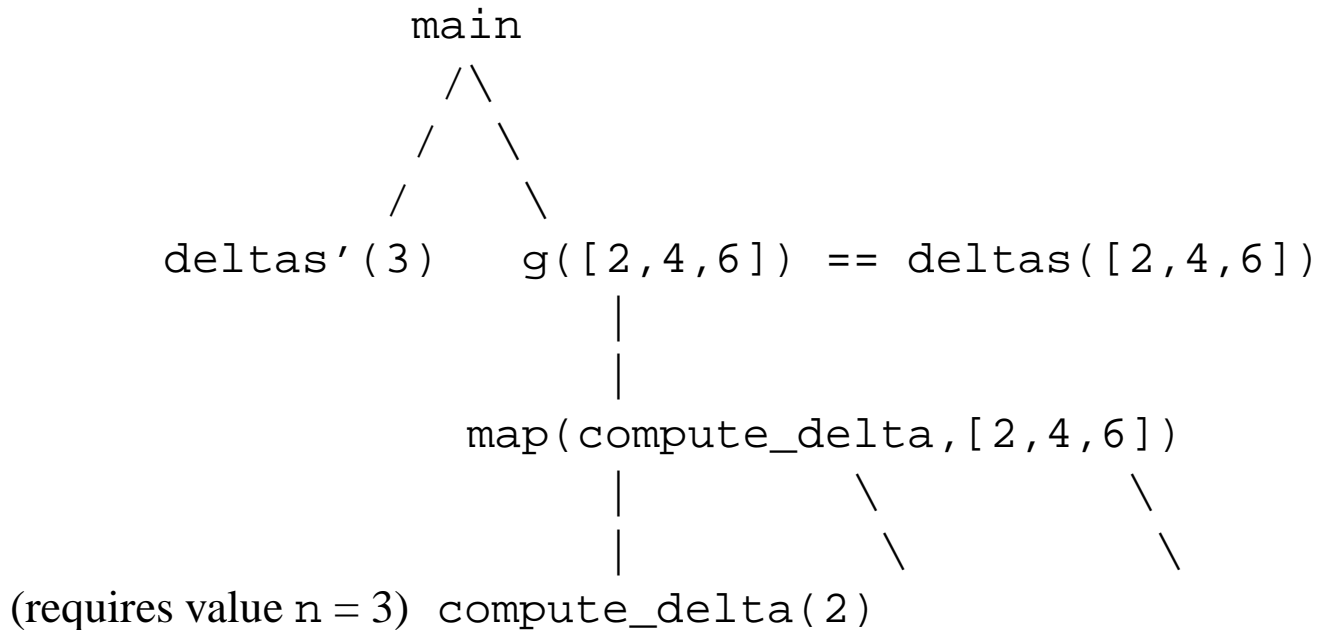
- Currying most often useful when passing partially applied functions to **other** higher-order functions, e.g.:

```
map (deltas' 3, [[1,7,5],[2,4,6]])
  (* yields [[~2,4,2],[~1,1,3]] *)
```

(Here we assume `map` works on lists of any type, in this case lists of lists of integers)

## Problems with first-class procedures

Consider activation tree for **deltas'** example:



Activation of `deltas'` is no longer live when `compute_delta` is called!

If `n` is stored in activation record for `deltas'` and activation-record is stack-allocated, it will be gone at the point where `compute_delta` needs it!

To avoid this problem:

- Pascal prohibits “upward funargs;” procedure values can only be passed downward, and can’t be stored.
- Some other languages only permit “top-level” procedures to be manipulated as procedure values (in C, this means **all** procedures!).

## Heap Storage for Procedure Values

- Languages supporting first-class nested procedures (e.g., Lisp, Scheme, ML, Haskell, etc.) solve problem by using heap to store variables like  $n$ .
- Simple solution: Just put all activation records in the heap to begin with! (Garbage collection is a must!)
- More refined solution: Represent procedure values by a heap-allocated “closure” record, containing the procedure’s code pointer and values of the non-local (“free”) variables referenced by the procedure.
- Involves taking **copies** of the values of non-local variables, so only works when values are **immutable**.
- Can always introduce extra level of indirection to achieve this.

## Functional Programming

What does **functional** mean? Two main senses:

- Functions are supported as “**first-class**” values.
- Programs consist of functions with **no side effects** (also say programs are **pure** or **applicative**).

Claim: functional programs are:

- clearer;
- easier to get right;
- easier to test;
- easier to transform;
- easier to parallelize;
- easier to prove things about.

Important examples:

- Lisp, Scheme (“strict”, dynamically typed, impure)
- Standard ML, CAML (“strict”, statically typed, impure)
- Haskell (“lazy”, statically typed, pure)

## **ML Example: Dictionaries with Sorted Lists**

```
datatype dict = Node of int * string * dict * dict
              | Leaf
```

```
fun insert (d:dict) (k:int,v:string) : dict =
  case d of
    Node (k',v',left,right) =>
      if k < k' then
        Node (k',v',insert left (k,v),right)
      else if k > k' then
        Node (k',v',left, insert right (k,v))
      else (* k = k' *)
        Node (k,v',left,right)
  | Leaf => Node (k,v,Leaf,Leaf)
```

```
fun member (d:dict) (k:int) : bool =
  case d of
    Node (k',v',left,right) =>
      (k < k' andalso member left k) orelse
      (k > k' andalso member right k) orelse
      k = k'
  | Leaf => false
```

```
- val a = insert Leaf (1,"a");
val a = Node (1,"a",Leaf,Leaf) : dict
- val b = insert a (7,"b");
val b = Node (1,"a",Leaf,
              Node (7,"b",Leaf,Leaf)) : dict
- val c = insert b (4,"c");
val c = Node (1,"a",Leaf,
              Node (7,"b",Node (4,"c",Leaf,Leaf)
                          Leaf)) : dict
- val q = member b 4;
val q = false : bool
```

## Features

- Identifiers denote values, not changeable variables.
- Datatype definition introduces new tree type, constructed using `Leaf` and `Node`, and tested and deconstructed using **pattern matching**.
- `insert` is a **function** that returns a **new** tree **without** changing its argument!
- Control structure is **recursion**, not iteration.

## Why bother?

Functions can't have side-effects. Therefore, they can't have dangerous, hidden side-effects!

Consider this C fragment:

```
insert(1, "q", t);  
x = f(t); (* source not here... *)  
s = member(1, t);
```

Will `s` be `true`? It depends whether `f` modifies `t`!

Compare this functional code:

```
val t' = insert (1, "q") t  
val x = f t'  
val s = member 1 t'
```

Here `s` must be `true`, because `f` **can't** modify its argument (or anything else)! Also compare:

```
val t' = insert (1, "q") t  
val (x, t'') = f t'  
val s = member 1 t''
```

## It's testable.

- True functions can always be **tested separately**.

```
fun appendx k t =  
  let val v = find k t  
      val t' = delete k t  
  in insert (k,v ^ "x") t'  
  end
```

If `appendx` is wrong, then its definition is wrong, or `find` or `delete` or `insert` must be wrong.

The problem **can't** be due to a hidden interaction between `find`, `delete`, and/or `insert`.

Any coupling between functions must be made explicit in their arguments or return values.

This helps discourage coupling!

(In principle, debugging by divide and conquer can even be automated. In practice, conventional trouble-shooting is much easier.)