CS558 Programming Languages
Winter 2006
Lecture 4d
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).
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} */
EXAMPLE: CALL-BACKS FROM SURROUNDING SYSTEM

System provides this interface:

```c
typedef void (* click_handler)(int);
void registerClickHandler(click_handler h);
```

Application defines and registers call-back function:

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

registerClickHandler(handler);
```
**Example: Parameterized data structure traversals in ML**

```ml
fun map (g:'a -> 'b, u: 'a list) : 'b list = 
    case u of 
    nil => nil 
    | (h::t) => (g h)::(map (g,t))

fun inc x = x + 1
fun paren s = "(" ^ s ^ ")"
val w = map (inc,
[1,2,3])  yields [2,3,4]
val z = map (paren, ["a","bc",""])  yields ["(a)","(bc)","]"

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

```ml
val w = map (fn x => x + 1, [1,2,3])
val z = map (fn s => "(" ^ s ^ ")", ["a","bc",""])
```

In fact, the following declarations are identical (except that the second one isn’t recursive):

```ml
fun foo x = e
val foo = fn x => e
```
NESTED PROCEDURE DECLARATIONS

In Pascal, Ada, ML, etc., we can nest procedure declarations inside other procedure declarations. (Cannot do this in C, C++, Java!)

```ml
fun map (g:'a -> 'b, u: 'a list) : 'b list =
  let fun f (v : 'a list) : 'b list =
    case v of
      nil => nil
    | (h::t) => (g h)::(f t)
  in f u
end
```

- Parameters and local variables of outer procedures are visible within inner procedures (using lexical scoping rules).
- Purpose: localize scope of nested procedures, and avoid the need to pass auxiliary parameters defined in outer scopes.
- Semantics of a function definition now depend on values of function’s free variables.
- Key implementation question: what is the lifetime of the function?
Suppose nested procedure $p$ uses non-local variables? How are they found?

**Semantically**, it suffices to know the static environment surrounding the declaration of $p$ was encountered.

An interpreter can simply attach the current variable environment to its description of $p$ when it encounters $p$’s declaration and records it in the function environment.

- When the interpreter applies $p$, it evaluates its body in an initial environment taken from the recorded description, which is then extended with $p$’s parameters and locals.
- When the interpreter looks up a variable while executing $p$, it looks first among $p$’s locals and parameters, and then in the lexically-enclosing environment.
Here are appropriate dynamic semantic rules.

\[ \langle \text{fn } x \Rightarrow e, E, S \rangle \Downarrow \langle [x, e, E], S \rangle \] (Fn)

\[ \langle e_1, E, S \rangle \Downarrow \langle [x, e', E'], S'' \rangle \quad \langle e_2, E, S' \rangle \Downarrow \langle v', S''' \rangle \]

\[ \langle e', E' + \{ x \mapsto v' \}, S'' \rangle \Downarrow \langle v, S'''' \rangle \]

\[ \langle (\mathbf{@} e_1 e_2), E, S \rangle \Downarrow \langle v, S'''' \rangle \] (Appl)
**USING NESTED PROCEDURES**

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

  ```
  fun scale_list(s:int,u:int list):int list = 
    let fun scale(x:int) = s*x 
    in map (scale,u) 
    end
  ```

  ```
  scale_list(3,[1,7,5])  (yields [3,21,15])
  ```

- Lexical scope rules apply, so function body can use data associated with outer function.

- Here `scale` uses the value of `s`, which is a parameter of `scale_list`.

What if we wanted to compute `scale_list` on a several different lists with a fixed `s`?

- Want to create a function that represents `scale_list` specialized to a particular value of `s`.

- Solution: Write a function that *returns* another function!
fun scale_list' (s:int) : int list -> int list =  
  let fun sc (u : int list) : int list =  
    let fun scale (s:int) : int = s*n  
    in map (scale,u)  
  in sc  
end

val g : int list -> int list = scale_list’ 3  
...
val x : int list = g [1,7,5]  \[(yields \[3,21,15]\)]  
val y : int list = g [2,4,6]  \[(yields \[6,12,18]\)]

val z : int list = scale_list’ 3 [2,4,6]  \[(yields \[6,12,18]\)]
ML also provides syntactic sugar to make such “Curried” functions easier to write. Above program is equivalent to:

```ml
fun scale_list’ (s:int) (u:int list) : int list =
  let fun scale (s:int) = s*x
     in map (scale,u)
  end
```

- When defining “multi-argument” functions in ML, have a choice using a tuple argument and Currying.
- Can apply Curried version `scale_list’` to either one or two arguments.
- Function application associates to the **left**, so
  
  \[
  \text{scale_list’ } 3 \ [2,4,6] = \text{(scale_list’ } 3) \ [2,4,6]\]

- Function type arrows associate to the **right**, so `scale_list’` has type

  \[
  \text{int -> int list -> int list } = \text{int -> (int list -> int list)}
  \]
Curried Functions (2)

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

  \[
  \text{map (scale\_list' 3, } [\[1,7,5\],\[2,4,6\]]) \\
  (\text{yields } [\[3,21,15\],\[6,12,18\]])
  \]

- Note: unlike the \text{map} we’ve used here, the “built-in” definition of \text{map} in the SML standard library is itself defined as a Curried function, with type (’a \to ’b) \to ’a list \to ’b list.
To **compile** nested procedures, need to fix a way for generated code to access (non-global) free variables.

- If we’re using conventional activation records, the free variables for a procedure \( p \) live in the activation record of some **statically enclosing** procedure \( q \).

**Assuming** \( p \)’s lifetime is contained within \( q \)’s lifetime, then can access \( q \)’s variables via a pointer to its activation record.

- Usually done by maintaining (at runtime) a **chain** of **static links** from each activation to the lexically enclosing procedure’s activation.

- To access a free variable, the generated code de-references one or more links in the chain and then uses a known offset relative to link target. This has (modest) runtime cost.

- To pass \( p \) as a parameter to another function, we package its code address together with its own static link.

- **But** what happens if the **lifetime** of \( p \) outlives that of \( q \)? (When can this happen?)
Problems with first-class procedures

Consider activation tree for `scale_list'` example:

```
main
  /
 /\  
|  
/   \  
/     \  
scale_list'(3) g([2,4,6]) == sc([2,4,6])
  |     |
  |     |
  | map(scale,[2,4,6])
  |     |
  |     |
  |     (requires value s = 3) scale(2)
```

Activation of `scale_list'` is no longer live when `scale` is called!

If `s` is stored in activation record for `scale_list'` and activation-record
is stack-allocated, it will be gone at the point where `scale` 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!).
- Languages supporting first-class nested procedures (e.g., Lisp, Scheme, ML, Haskell, etc.) solve problem by using the heap to store variables like $s$.
- 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 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.)
The ability to manipulate functions as **first-class** values is one of the hallmarks of a **functional language**.

Functional languages encourage sophisticated **abstraction** mechanisms. (Already saw use of `map`.)

Consider the following problems:

Sum a list of integers

```haskell
fun sum l =
    case l of
        nil => 0
    | h::t => h + (sum t)
```

Multiply a list of integers:

```haskell
fun prod l =
    case l of
        nil => 1
    | h::t => h * (prod t)
```
Copy a list (of anything):

```haskell
fun copy l = 
  case l of 
    nil => nil 
  | h::t => h::(copy t)
```

Query: How does \textit{copy} differ from the identity function \texttt{fn x => x}?

Calculate the length of a list (of anything):

```haskell
fun len l = 
  case l of 
    nil => 0 
  | h::t => 1 + (len t)
```
We can abstract over the common inductive pattern displayed by these examples:

```haskell
fun foldr f n l =
  case l of
    nil => n
  | h::t => f(h,foldr f n t)

fun sum l = foldr (fn (x,y) => x+y) 0 l
fun prod l = foldr (op *) 1 l
fun copy l = foldr (op ::) nil l
fun len l = foldr (fn (_,y) => 1+y) 0 l
```

Function `foldr` computes a value working from the tail of the list to the head (from right to left). Argument `n` is the value to return for the nil list. Argument `f` is the function to apply to each element and the previously computed result.
Can view \texttt{foldr} \( f \ n \ l \) as replacing each \texttt{::} constructor in \( l \) with \( f \) and the \texttt{nil} constructor with \( n \). For example:

\[
l = x_1 :: (x_2 :: (\ldots :: (x_n :: \texttt{nil})))
\]

\[
\text{foldr} \ (\text{op } *) \ 1 \ l =
\]

\[
x_1 * (x_2 * (\ldots (x_n * 1)))
\]