



The University of North Carolina at Chapel Hill

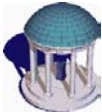
---

COMP 144 Programming Language Concepts  
Spring 2003

## ML Runtime, Deep/Shallow Access, Referential Transparency

Stotts

1



## Terms

---

Deep vs. Shallow access

A-lists vs. Central Environment Tables

Deep vs. Shallow binding (of referencing environments)

FunArg problem

Closures

2



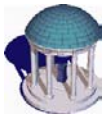
## Beyond ML

---

*These are issues that are not specific to ML  
but occur in other functional language  
implementations as well as in non-functional*

*They both pertain to implementing dynamic  
scoping*

3



## Scope in ML is Lexical

---

Top level environment has all pre-defined bindings


Every `val x = 5;` binding adds another row to the symbol table when compiling/interpreting

Each row hides earlier bindings of the same name, does not destroy them

Local bindings can be made in functions definitions


Locals are removed from the symbol stack when the function definition is complete

4



## Name/Value Bindings

---

  
 ↑  
 New bindings

<b>x</b>	<b>var</b>	<b>[1,2,3]</b>	<b>int list</b>
<b>sq</b>	<b>fn</b>	<b>\z: z*26.3</b>	<b>real -&gt; real</b>
<b>k2</b>	<b>var</b>	<b>26.3</b>	<b>real</b>
<b>x</b>	<b>var</b>	<b>5</b>	<b>bool</b>
<b>r1</b>	<b>var</b>	<b>(7,3.14)</b>	<b>int * real</b>


```

val r1 = (7,3.14);
val x = false;
val k2 = 26.3;
fun sq z:real = z*k2;
val x = [1,2,3];
    
```

*name*
*cat*
*value*
*type*

*Start -- Global level*

6



## Binding Stack

---

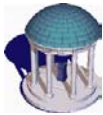
When a name is referenced, you search the stack from the top looking for the first occurrence of the name

This corresponds to the most recently created binding... in ML this will either be the global binding, or a Let binding

Evaluating expressions that give values to be bound use the current bindings in the stack

- see the defn of "sq", uses the binding to "k2" and puts the value "26.3" in the body of the function

6

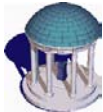


## Binding Stack

### When a function is defined:

- Name is looked up in the stack to get the definition
- Local ("let") bindings are added to the binding stack
- Local bindings are popped off the stack at end of definition

7




## ML Example: static scope


```
val x = 5;
fun f1 z = z * x;
fun f2 z =
  let val x = 2
  in (f1 z) * x
  end;

f1 4; (* evals to 20 *)
f2 4; (* evals to 40 with static scope *)
      (* dyn scope would cause eval to 16 *)
```

8



## Simple binding stack



New bindings

x	var	[1,2,3]	int list
sq	fn	λz: z*26.3	real -> real
k2	var	26.3	real
x	var	5	bool
r1	var	(7,3.14)	int * real


May have to search top to bottom (deep) to find a binding

$O(n)$  complexity

This style table of name / value pairs (comes from old Lisp)

name
cat
value
type

9



## Hash table of stacks

Can implement ML scope with symbol table from before  
 hash table with lists at each hash entry  
 each list managed as a stack

1

curLev

$O(1)$  search time for a binding

	sq	fn	λz: z*26.3	real -> real
	x	var	[1,2,3]	int list
	x	var	5	int
	r1	var	(7,3.14)	int * real
	k2	var	26.3	real

10



## Dynamic Scope

---

Deep vs. Shallow access

A-lists vs. Central Environment Tables

**A-list** very similar to the single stack of bindings,  $O(n)$  search time for each access at run-time

**CET** very similar to hash table of mini-stacks,  $O(1)$  access search but maintenance at each function call/return

11



## Deep and Shallow Access

---

- **Deep Access, Shallow Access** are *both* concepts related to dynamic scoping
- Confusingly often called “*deep binding*”, “*shallow binding*” elsewhere
- Terms come from search strategy and symbol table organization... **single stack of bindings** (deep) vs. **hash table of little stacks** (shallow)

12

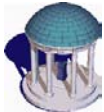


## Binding of Referencing Environments

---

- Scope rules are used to determine the referencing environment
  - Static and dynamic scoping
- Some languages allow *references to subroutines*
  - Are the scope rules applied when the reference is created or when the subroutine is called?
- In *shallow (late) binding*, the referencing environment is bound when the subroutine is called
- In *deep (early) binding*, the referencing environment is bound when the reference is created

13

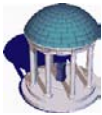


## Deep and Shallow Binding

---

- **Deep Binding, Shallow Binding** are *both* concepts related to giving a function/subroutine a referencing environment in which to run
- This is important when a subprogram is passed in or out as a parameter to another (a ‘funarg’)
- When a funarg passed in is run, does it use the environment it has when run? Or the one it has when defined?
- When a funarg is passed out and run, the environment it was created in is *gone* (runtime stack)

14



## Example

- Pascal uses static scoping
- Prints 1 if shallow binding is used
- Prints 2 if deep binding is used

```
program BindingExample (input, output);  
procedure A (I : integer; procedure P);  
  
    procedure B;  
    begin  
        writeln (I);  
    end;  
  
begin (* A *)  
    if I > 1 then  
        P  
    else  
        A (2, B);  
    end;  
  
procedure C; begin end;  
  
begin (* main *)  
    A (1, C);  
end.
```

15



## Closures

- Deep binding is implemented using *closures*
- A closure is the combination of a reference to a subroutine and an explicit representation of its referencing environment
  - Typically implemented with
    - » A pointer to the subroutines code
    - » If the scoping is dynamic, we need a way to temporarily unroll all the changes since the reference was created
- *Scott 3.4 reading*

16



## FunArgs

---

- **funarg problem**
- The failure of traditional stack-based implementations of procedure calls in the presence of "first-class" functions (functions that can be passed as procedure parameters and returned as procedure results).
- **Upwards funarg problem:** the problem of returning a function as a procedure result; requires allocating activation records on the heap and returning a closure containing a pointer to code and a pointer to the enclosing activation record.
- **Downwards funarg problem:** the problem of passing a function as a procedure parameter; requires a tree structure for activation records.

17



## Referential Transparency

---

- Functional programming languages try to enforce **referential transparency**
- A binding is immutable...
- Any time you see a name, you may substitute in the value bound to that name and NOT alter the semantics of the expression.
- *"no side effects"*

18



## Referential Transparency

- *“equals can be substituted for equals”*:
  - if two expressions are defined to have equal values, then one can be substituted for the other in any larger expression without affecting the result of the computation. For example, in
    - »  $s = \text{sqrt}(2); z = f(s, s);$  we can eliminate "s" and write
    - »  $z = f(\text{sqrt}(2), \text{sqrt}(2));$

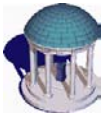
19



## Referential Transparency

- *A function is called referentially transparent if, given the same parameter(s), it always returns the same result.*
- In mathematics all functions are referentially transparent,
- In programming this is not always the case, with use of imperative features in languages.
- The subroutine/function called could affect some global variable that will cause a second invocation to return a different value

20

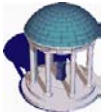


## Referential Transparency

### Another example

Take a "function" that takes no parameters and returns input from the keyboard. A call to this function may be `GetInput()`. The return value of `GetInput()` depends on what the user feels like typing in, so multiple calls to `GetInput()` with identical parameters (the empty list) may return different results.

21



## Referential Transparency

### *Why is referential transparency important?*

Because it allows the programmer to reason about program behavior, which can help in proving correctness, finding bugs that couldn't be found by testing, simplifying the algorithm, assist in modifying the code without breaking it, or even finding ways of optimizing it.

```
s = sqrt(9);  
x = s*s + 17*k / (s-1) ;  
// can replace x with  
// sqrt(9)*sqrt(9) + 17*k / (sqrt(9)-1)  
// 9 + 17*k / 2
```

22