Names, Scopes, Bindings

Textbook Chapter 3

1

Binding

- **Association of name with value**
  - **Language design time** — **operators**
  - **Language implementation time** — **size of int in C++**
  - **Program writing time** — **constants**
  - **Compile time** — **+, non-virtual fun.**
  - **Link time** — **function impl.**
  - **Load time** — **init. of static vars.**
  - **Run time** — **virtual functions**
- **Static vs. dynamic**
  - **E.g., compile time vs. run time**

2

Object Lifetimes

- **Creating of object** — **new C()**
- **Creation of binding** — **p = new C();**
- **Use of binding** — **p.foo();**
- **De-/reactivation of binding** — **call fun/return**
- **Destruction of binding** — **p = null;**
- **Destruction of object** — **garbage collect.**

3

Storage Allocation

- **Static**
  - **allocated at link time** **static fields**
- **Stack**
  - **allocated on function call** **local vars**
  - **deallocated on return**
- **Heap**
  - **allocated dynamically** **using new**
  - **manual deallocation or GC**

4

Static vs. Dynamic Scope

```
(define x 1)
(define (bar) x)
(define (foo f)
    (define x 2)
    (f))
(define y (foo bar))
```

- **Static scoping:** **y = 1**
- **Dynamic scoping:** **y = 2**

5

Static vs. Dynamic Scope in C

```
int x = 1
int bar() { return x; }
int foo(int(*f)()) {
    int x = 2;
    return f();
}
int y = foo(bar);
```

- **Static scoping:** **y = 1**
- **Dynamic scoping:** **y = 2**

6

Implementation of C

- Local variables allocated on stack
- Uses pointer to code for representing function value
- Static scoping is easy: non-local variables are global

7

Implementation of Pascal

- Local variables allocated on stack
- Uses pointer to code for representing function value
- Static scoping with nested functions: pass pointer to enclosing scope as additional implicit argument

8

Implementation of Scheme

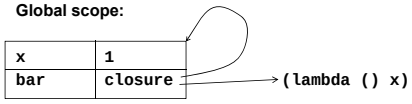
- Functional language
  - Nested functions
  - Static scoping
  - Functions as return values
- Local variables allocated on heap w/ GC
- Use Closures as function values
  - Pointer to code + Pointer to environment
- Remember environment in which function is defined

9

Construction of Closures

```
(define x 1)
(define (bar) x)
```

Global scope:



Closure contains environment in which function is defined

10

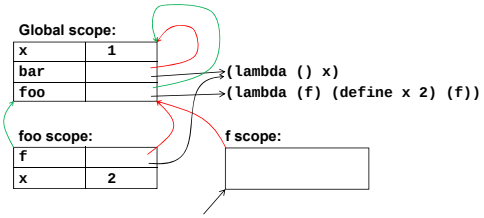
Function Call

- Take environment out of closure
- Create function scope as nested scope within closure environment
- Define parameters in function scope
- Evaluate function body in fun scope

11

Construction of Function Scopes

```
(define x 1)
(define (bar) x)
(define (foo f) (define x 2) (f))
(foo bar)
```



12

Functions as Return Values

```
(define (add x)
    (lambda (y)
        (+ x y)))

(define add1 (add 1))
(define add5 (add 5))
(define i (add1 10))
(define j (add5 10))
```

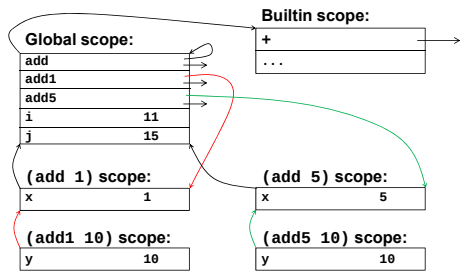
13

Add Example in C Syntax

```
(int(*)(int)) add(int x) {
    int foo(int y) { return x+y; }
    return foo;
}
int (*add1)(int) = add(1);
int (*add5)(int) = add(5);
int i = add1(10);
int j = add5(10);
```

14

Environments for Add Example



15