Names, Scopes, Bindings

Textbook Chapter 3

Binding

- Association of name with value
  - Language design time
  - Language implementation time
  - Program writing time
  - Compile time
  - Link time
  - Load time
  - Run time
  - Operators
  - Size of int in C++
  - Constants
  - +, non-virtual fun.
  - Function impl.
  - Init. of static vars.
  - Virtual functions

- Static vs. dynamic
  - E.g., compile time vs. run time

Object Lifetimes

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

Static vs. Dynamic Scope

```scheme
(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
```

Static vs. Dynamic Scope in C

```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
```
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

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

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
Construction of Closures

\( \text{(define } x \ 1) \)
\( \text{(define (bar) } x) \)

Global scope:

\[
\begin{array}{c}
  x & 1 \\
  \text{bar} & \text{closure} \\
\end{array}
\rightarrow \text{lambda } () \ x
\]

Closure contains environment in which function is defined

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

Construction of Function Scopes

\( \text{(define } x \ 1) \)
\( \text{(define (bar) } x) \)
\( \text{(define (foo f) } \text{(define } x \ 2) \ f) \)
\( \text{(foo bar)} \)

Global scope:

\[
\begin{array}{c}
  x & 1 \\
  \text{bar} & \text{closure} \\
  \text{foo} & \text{scope: } \text{f scope: } f \\
\end{array}
\rightarrow \text{lambda } () \ x
\]

foo scope: f scope: f

\[
\begin{array}{c}
  f & 2 \\
\end{array}
\]
Functions as Return Values

\[\text{(define (add } x)\]
\[\text{ (lambda (y)}\]
\[\text{ (+ } x \text{ y))}\]
\[\text{(define add1 (add 1))}\]
\[\text{(define add5 (add 5))}\]
\[\text{(define i (add1 10))}\]
\[\text{(define j (add5 10))}\]

Add Example in C Syntax

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

Environments for Add Example

---

CSC 4101: Programming Languages