Lecture 9: Interpreter runtime structures

David Hovemeyer

September 30, 2020

601.428/628 Compilers and Interpreters

Today

- Scopes, environments, function calls
- Implementing functions and function calls
- Closures
- Memory management, garbage collection

Scopes, environments, and function calls

Scope, lifetime

- Scope: in what region(s) of the program is a particular variable visible?
- Lifetime: *when* in the execution of the program does a variable exist?
- These are related but distinct concepts

Example program

```
var a, b, c;
a = 1;
b = 2;
c = 3;

function add1(a) {
  var b;
  b = 1;
  c = 4;
  a + b;
}

var d;
d = add1(c);

println(a);
println(b);
println(c);
d;
```

Example program

```
var a, b, c;
a = 1;
b = 2;
c = 3;

function add1(a) {
  var b;
  b = 1;
  c = 4;
  a + b;
}

var d;
d = add1(c);

println(a);
println(b);
println(c);
d;
```

scope of global variable a

Example program

```
var a, b, c;
a = 1;
b = 2;
c = 3;

function add1(a) {
  var b;
  b = 1;
  c = 4;
  a + b;
}

var d;
d = add1(c);

println(a);
println(b);
println(c);
d;
```

scope of global variable a

global variable a not visible here: <u>shadowed</u> by parameter a

Example program

```
var a, b, c;
a = 1;
b = 2;
c = 3;

function add1(a) {
  var b;
  b = 1;
  c = 4;
  a + b;
}

var d;
d = add1(c);

println(a);
println(b);
println(c);
d;
```

Scope of global variable b

Example program

```
var a, b, c;
a = 1;
b = 2;
c = 3;

function add1(a) {
  var b;
  b = 1;
  c = 4;
  a + b;
}

var d;
d = add1(c);

println(a);
println(b);
println(c);
d;
```

Scope of global variable b

global variable b not visible here: shadowed by local variable

Example program

```
var a, b, c;
a = 1;
b = 2;
c = 3;

function add1(a) {
   var b;
   b = 1;
   c = 4;
   a + b;
}

var d;
d = add1(c);

println(a);
println(b);
println(c);
d;
```

global variable c's scope is the entire program
- not shadowed by any identically-named parameters or local variables

Example program

```
var a, b, c;
a = 1;
b = 2;
c = 3;

function add1(a) {
  var b;
  b = 1;
  c = 4;
  a + b;
}

var d;
d = add1(c);

println(a);
println(b);
println(c);
d;
```

Assignment to global variable!

global variable c's scope is the entire program

- not shadowed by any identically-named parameters or local variables

Example program

```
var a, b, c;
a = 1;
b = 2;
c = 3;

function add1(a) {
  var b;
  b = 1;
  c = 4;
  a + b;
}

var d;
d = add1(c);

println(a);
println(b);
println(c);
d;
```

scope of parameter a:
body of function

Example program

```
var a, b, c;
a = 1;
b = 2;
c = 3;

function add1(a) {
  var b
  b = 1;
  c = 4;
  a + b;
}

var d;
d = add1(c);

println(a);
println(b);
println(c);
d;
```

scope of local variable b:
from point of definition to
end of function body

Example program

```
var a, b, c;
a = 1;
b = 2;
c = 3;

function add1(a) {
  var b;
  b = 1;
  c = 4;
  a + b;
}

var d;
d = add1(c);

println(a);
println(b);
println(c);
d;
```

scope of global variable d:
point of definition to end of module

Variable lifetime

- Global variables exist for the duration of the execution of the program
- Parameters and local variables exist for the duration of a function call
  - Call stack: each call pushes an *activation record*
  - A calls B, B calls C, C calls D, etc. — arbitrarily many calls can be in progress at any point
    - In practice, the call stack is usually limited in size
  - Recursion: A calls itself
    - Caller and callee always have distinct activation records

Environment

- We'll use the term *environment* for a data structure containing a collection of variables that have a common lifetime
- Global environment: has definitions of global variables
  - Global variables are visible throughout the program unless shadowed by a variable in an "inner" scope
- Function call environment: created dynamically (on the call stack) to represent parameters and local variables of a called function

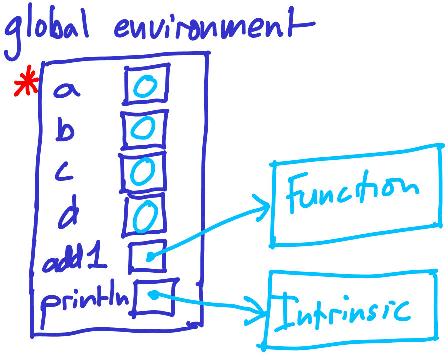
Nesting of environments

- Nesting: an "inner" environment can reference variables in an "outer" environment
  - But not vice versa!
- In our interpreter, there are only two levels of nesting, the global environment and function call environments
- In a *block-structured* language, every "block" defines a new environment
  - C and C++ are block-structured languages

Example program

```
var a, b, c;
a = 1;
b = 2;
c = 3;

function add1(a) {
  var b;
  b = 1;
  c = 4;
  a + b;
}

var d;
d = add1(c);

println(a);
println(b);
println(c);
d;
```

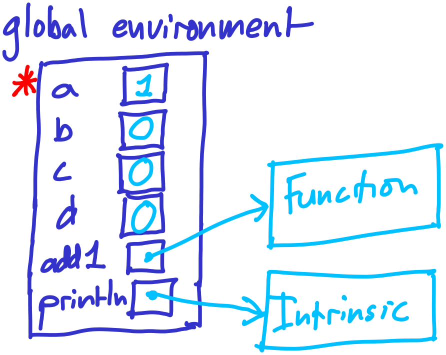
* denotes current environment



Example program

```
var a, b, c;
a = 1;
b = 2;
c = 3;

function add1(a) {
  var b;
  b = 1;
  c = 4;
  a + b;
}

var d;
d = add1(c);

println(a);
println(b);
println(c);
d;
```

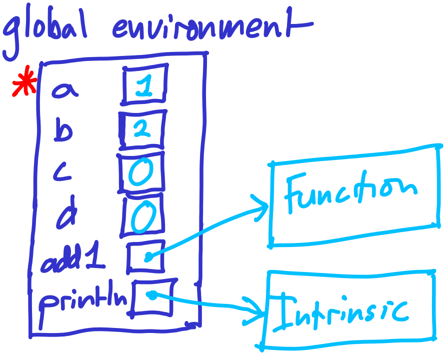
* denotes current environment



Example program

```
var a, b, c;
a = 1;
b = 2;
c = 3;

function add1(a) {
  var b;
  b = 1;
  c = 4;
  a + b;
}

var d;
d = add1(c);

println(a);
println(b);
println(c);
d;
```

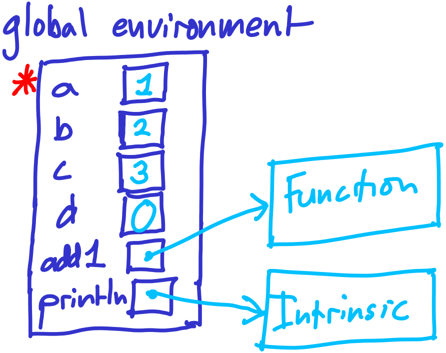
* denotes current environment



Example program

```
var a, b, c;
a = 1;
b = 2;
c = 3;

function add1(a) {
   var b;
   b = 1;
   c = 4;
   a + b;
}

var d;
d = add1(c);

println(a);
println(b);
println(c);
d;
```

* denotes current environment



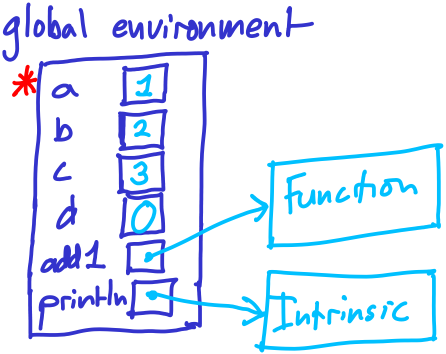
Example program

* denotes current environment

```
var a, b, c;
a = 1;
b = 2;
c = 3;

function add1(a) {
  var b;
  b = 1;
  c = 4;
  a + b;
}

var d;
d = add1(c);
```

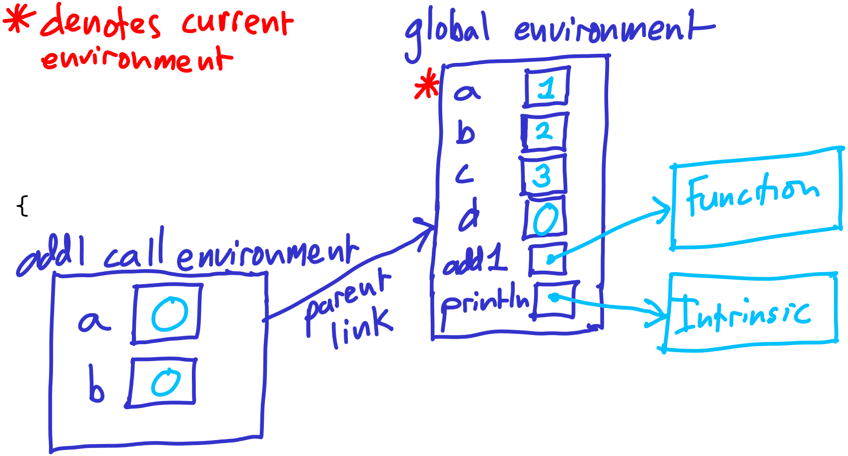
function call: create environment

```
println(a);
println(b);
println(c);
d;
```



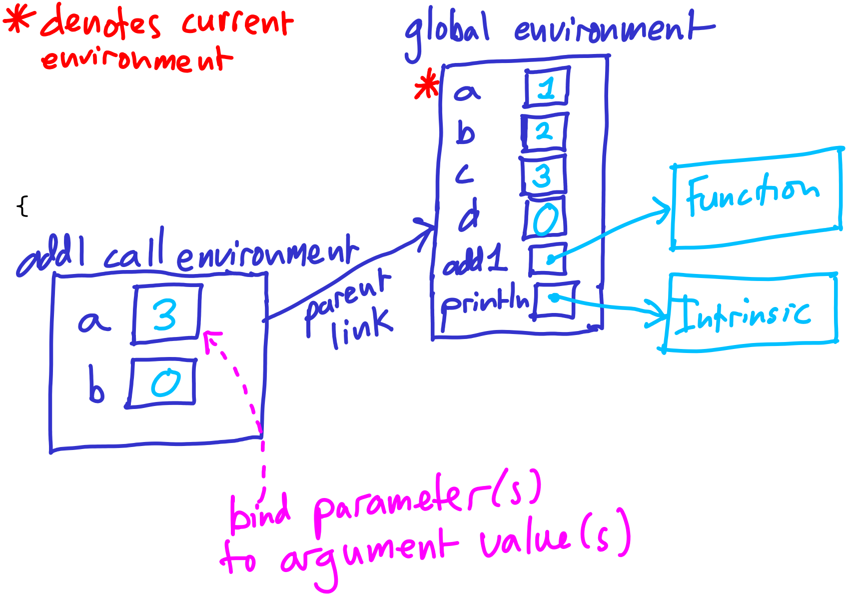
Example program

```
var a, b, c;
a = 1;
b = 2;
c = 3;

function add1(a) {
   var b;
   b = 1;
   c = 4;
   a + b;
}

var d;
d = add1(c);

println(a);
println(b);
println(c);
d;
```



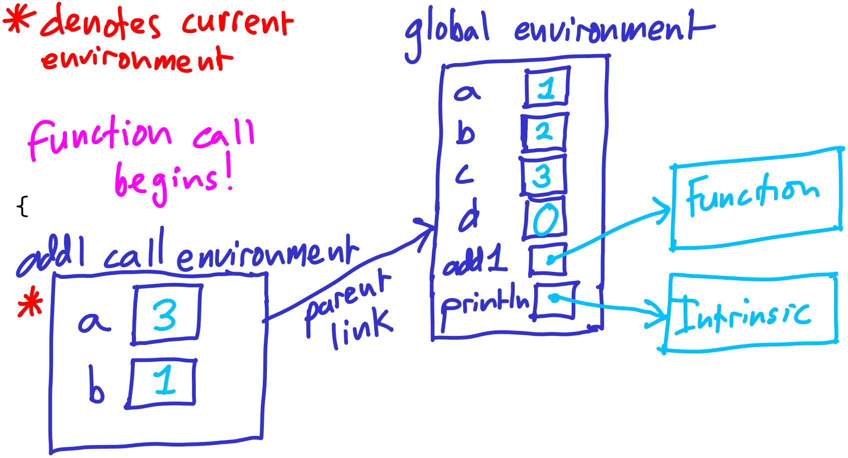
Example program

```
var a, b, c;
a = 1;
b = 2;
c = 3;

function add1(a) {
   var b;
   b = 1;
   c = 4;
   a + b;
}

var d;
d = add1(c);

println(a);
println(b);
println(c);
d;
```



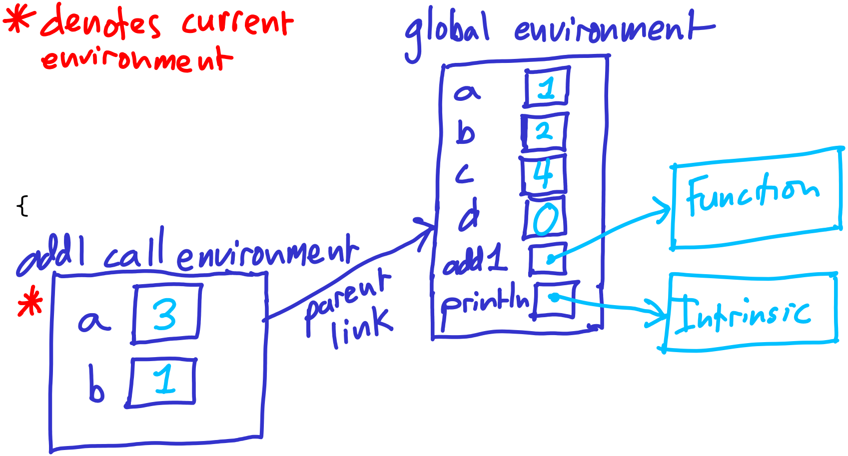
Example program

```
var a, b, c;
a = 1;
b = 2;
c = 3;

function add1(a) {
  var b;
  b = 1;
  c = 4;
  a + b;
}

var d;
d = add1(c);

println(a);
println(b);
println(c);
d;
```



Example program

```
var a, b, c;
a = 1;
b = 2;
c = 3;

function add1(a) {
   var b;
   b = 1;
   c = 4;
   a + b;
}

var d;
d = add1(c);

println(a);
println(b);
println(c);
d;
```

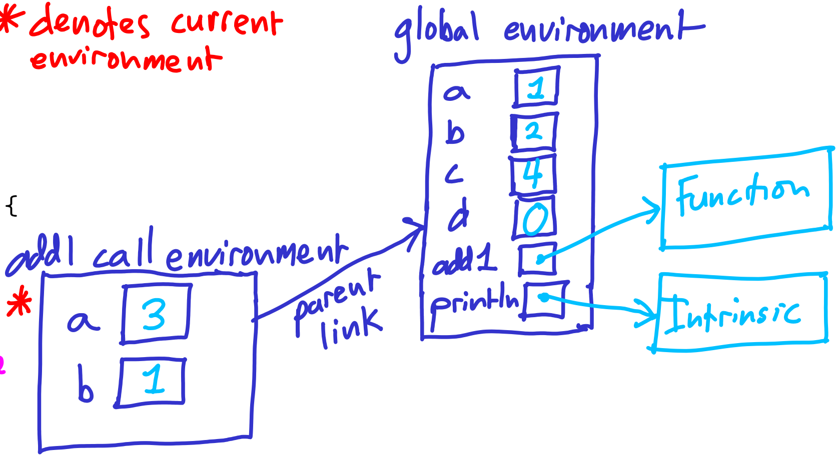


Example program

```
var a, b, c;
a = 1;
b = 2;
c = 3;

function add1(a) {
  var b;
  b = 1;
  c = 4;
  a + b;
}

var d;
d = add1(c);

println(a);
println(b);
println(c);
d;
```

result value is 5

* denotes current environment

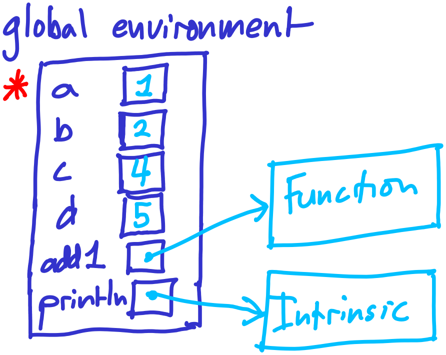


Example program

```
var a, b, c;
a = 1;
b = 2;
c = 3;

function add1(a) {
   var b;
   b = 1;
   c = 4;
   a + b;
}

var d;
d = add1(c);

println(a);
println(b);
println(c);
d;
```

* denotes current environment

value of function call expression is 5

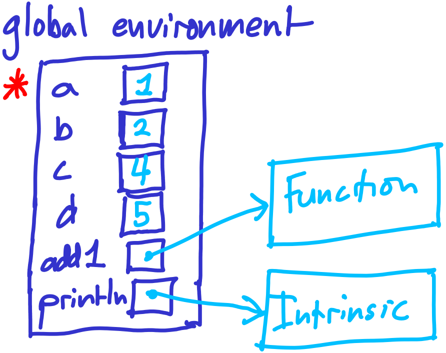


Example program

```
var a, b, c;
a = 1;
b = 2;
c = 3;

function add1(a) {
  var b;
  b = 1;
  c = 4;
  a + b;
}

var d;
d = add1(c);

println(a);
println(b);
println(c);
d;
```

* denotes current environment

prints $\boxed{1}$

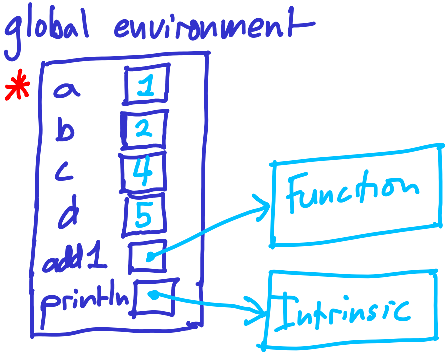


Example program

```
var a, b, c;
a = 1;
b = 2;
c = 3;

function add1(a) {
  var b;
  b = 1;
  c = 4;
  a + b;
}

var d;
d = add1(c);

println(a);
println(b);
println(c);
d;
```

* denotes current environment

prints $\boxed{2}$

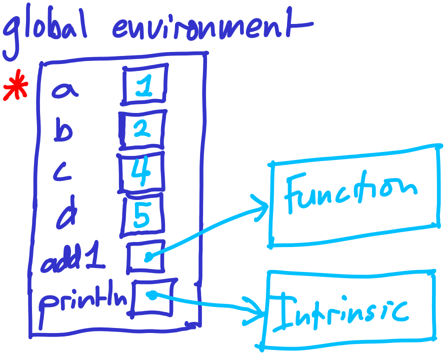


Example program

```
var a, b, c;
a = 1;
b = 2;
c = 3;

function add1(a) {
  var b;
  b = 1;
  c = 4;
  a + b;
}

var d;
d = add1(c);

println(a);
println(b);
println(c);
d;
```

prints $\boxed{4}$

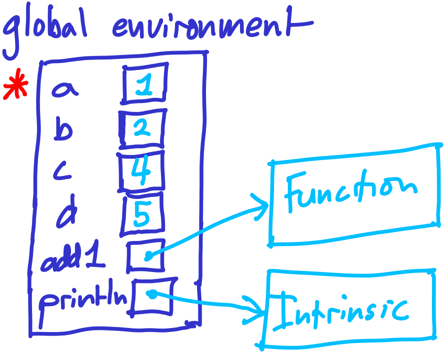
* denotes current environment



Example program

```
var a, b, c;
a = 1;
b = 2;
c = 3;

function add1(a) {
   var b;
   b = 1;
   c = 4;
   a + b;
}

var d;
d = add1(c);

println(a);
println(b);
println(c);
d;
```

* denotes current environment



Program result: 5

Lexical addresses

In a language where every variable's scope is known statically, we can use *lexical addresses* to associate variable references with their definitions

Each variable has an integer position

Lexical address is pair (*depth*, *position*)

- *depth*: 0 if referenced variable is in current environment, 1 if in parent, 2 if in grandparent, etc.

Determining lexical addresses

Analyze source:

- Keep track of current (static) environment, initially the global environment
- Enter a nested scope $\rightarrow$ enter a nested environment (with previous environment as its parent)
- Leave a nested scope $\rightarrow$ return to parent environment
- Keep track of names defined (variables, functions)
- As long as definitions precede uses, we can associate each reference to a name with an entry in a static environment
- Static (pre-execution) environments are also called *symbol tables*
  - Much more about these when we move on to compilers!

Example program

```
var a, b, c;
a = 1;
b = 2;
c = 3;

function add1(a) {
  var b;
  b = 1;
  c = 4;
  a + b;
}

var d;
d = add1(c);

println(a);
println(b);
println(c);
d;
```

Initially: global env.
w/ intrinsics defined

global env.

name	position
println	0

Example program

```
var a, b, c;
a = 1;
b = 2;
c = 3;

function add1(a) {
  var b;
  b = 1;
  c = 4;
  a + b;
}

var d;
d = add1(c);

println(a);
println(b);
println(c);
d;
```

definitions of a, b, c

global env.

name	position
println	0
a	1
b	2
c	3

Example program

```
var a, b, c;
a = 1; (0,1)
b = 2; (0,2)
c = 3; (0,3)

function add1(a) {
  var b;
  b = 1;
  c = 4;
  a + b;
}

var d;
d = add1(c);

println(a);
println(b);
println(c);
d;
```

assign lexical addresses

global env.

name	position
println	0
a	1
b	2
c	3

Example program

```
var a, b, c;
a = 1; (0,1)
b = 2; (0,2)
c = 3; (0,3)

function add1(a) {
  var b;
  b = 1;
  c = 4;
  a + b;
}

var d;
d = add1(c);

println(a);
println(b);
println(c);
d;
```

create nested env. for function, define add1 in global env

global env.

name	position
println	0
a	1
b	2
c	3
add1	4

parent link

add1 env.

name	position

Example program

```
var a, b, c;
a = 1; (0,1)
b = 2; (0,2)
c = 3; (0,3)

function add1(a) {
  var b;
  b = 1;
  c = 4;
  a + b;
}

var d;
d = add1(c);

println(a);
println(b);
println(c);
d;
```

define param a,
local var b

global env.

name	position
println	0
a	1
b	2
c	3
add1	4

parent link

add1 env.

name	position
a	0
b	1

Example program

```
var a, b, c;
a = 1; (0,1)
b = 2; (0,2)
c = 3; (0,3)

function add1(a) {
  var b;
  b = 1; (0,1)
  c = 4; (1,3)
  a + b;
}
(0,0) (0,1)

var d;
d = add1(c);

println(a);
println(b);
println(c);
d;
```

assign lexical addrs for var refs

global env.

name	position
println	0
a	1
b	2
c	3
add1	4

parent link

add1 env.

name	position
a	0
b	1

Example program

```
var a, b, c;
a = 1; (0,1)
b = 2; (0,2)
c = 3; (0,3)

function add1(a) {
  var b;
  b = 1; (0,1)
  c = 4; (1,3)
  a + b;
}
(0,0) (0,1)

var d;
d = add1(c);

println(a);
println(b);
println(c);
d;
```

assign lexical addrs for var refs

reference to global variable, depth is 1 (parent env.)

parent link

global env.

name	position
println	0
a	1
b	2
c	3
add1	4

add1 env.

name	position
a	0
b	1

Example program

```
var a, b, c;
a = 1; (0,1)
b = 2; (0,2)
c = 3; (0,3)

function add1(a) {
  var b;
  b = 1; (0,1)
  c = 4; (1,3)
  a + b;
}
(0,0) (0,1)

var d;
d = add1(c);

println(a);
println(b);
println(c);
d;
```

define global var d

global env.

name	position
println	0
a	1
b	2
c	3
add1	4
d	5

Example program

```
var a, b, c;
a = 1; (0,1)
b = 2; (0,2)
c = 3; (0,3)

function add1(a) {
   var b;
   b = 1; (0,1)
   c = 4; (1,3)
   a + b;
}
(0,0) (0,1)

var d;
d = add1(c);
(0,5) (0,4) (0,3)
(0,0) println(a); (0,1)
(0,0) println(b); (0,2)
(0,0) println(c); (0,3)
d; (0,5)
```

assign remaining lexical addresses

global env.

name	position
println	0
a	1
b	2
c	3
add1	4
d	5

Function calls

Executing a function call

- As we've seen, executing a function call means:
  - Creating a new environment for it (with global environment as parent)
  - Evaluating argument expressions
  - Binding function parameters to argument values in the new function call environment
  - Evaluating the body of the function in the new function call environment
  - Result of evaluating body is result of function
    - Becomes value of function call expression at call site

Environment representation

How to represent an environment? One possibility:

```
struct Environment {
  std::map<std::string, struct Value> vars;
  struct Environment *parent;
};
```

Assumes that variables will be looked up by name

Another possibility

If lexical addresses are computed for variable/function references, then the *position* of the variable in the environment is the key to accessing it

- Names are only needed when assigning lexical addresses prior to execution

So:

```
struct Environment {
  // In both of these vectors, index=position
  std::vector<std::string> names; // not needed at runtime!
  std::vector<struct Value> vals; // only needed at runtime!
  struct Environment *parent;
}
```

Assumes that variables will be looked up (at runtime) by position

Function representation

Key information in function representation:

- Identifying information about all parameters and local variables
  - Enough information to create and initialize a function call environment at runtime
  - Maybe use `Environment` for this? Only needs info for local parameters and variables, can leave `parent` field null
- AST of function body

Possibility:

```
struct Function {
  struct Environment env_template;
  struct Node *body_ast;
};
```

Closures

Closures

Many languages support *closures*, a.k.a. anonymous functions, lambdas

Basic idea: the closure retains a pointer to its parent environment, i.e., the environment in which it was created at runtime

- This may imply that the lifetime of the parent environment is extended to be at least as long as the lifetime of the closure

Closure example

```
function make_addn(n) {
  return function(x) { x + n; };
}

var add1, add2;
add1 = make_addn(1);
add2 = make_addn(2);

println(add1(1)); // prints 2
println(add2(1)); // prints 3
```

Implementing closures

A few possible implementation techniques:

- Dynamically-allocate function call environments
  - Closure values retain a pointer to parent environment
  - Could use reference counting to know when to delete dynamically-created environments
    - Environment is destroyed when there are no remaining references to it
- Closure retains a *copy* of its parent environment (and grandparent, etc.)
  - Or, copies of just the variables that are actually referenced by the body of the closure function

Memory management, garbage collection

Dynamically allocated values

- We noted last time that runtime values may require dynamically-allocated storage
  - Strings, vectors, list nodes, objects, etc.
- How to ensure that dynamically allocated memory gets reclaimed when no longer used?
- A couple standard approaches:
  - Reference counting
  - Garbage collection

Reference counting

- All dynamically allocated objects have a *reference count* field
  - Is just an integer indicating how many pointers are pointing to the object
- Language runtime must take care to increment and decrement references counts
  - C++ smart pointers can help a lot with this
- When reference count reaches 0, deallocate
- Problem: object graphs with cycles can't be reclaimed

Garbage collection

- Language runtime keeps track of references to dynamic objects
- Periodically, it determines which objects are reachable
  - Unreachable objects are reclaimed
- There are *many* ways to do this! (Tons of research, we could do an entire course on this topic)
  - We'll briefly discuss two

Root set

- The *root set* of references are the starting point for determining which objects are reachable
- It consists of:
  - Objects referenced by global variables
  - Objects referenced by the activation records (i.e., function call environments) of currently-executing functions (on the call stack)
- Objects not directly or indirectly reachable from the root set can be assumed to be garbage

Assumptions

- The garbage collector can find all of the dynamically allocated objects
- Given a pointer to an object, the garbage collector knows what pointers to other objects are stored in it

Mark and sweep

- Starting from the root set, do a graph traversal to find all reachable objects, and mark them as “live”
- Traverse all dynamically allocated objects, and reclaim the memory of those not marked as alive
  - Also, clear the “live”‘ mark on objects that are still alive

Copying

- Heap is divided into *semispaces*
- New objects are allocated in the current semispace
- To collect garbage:
  - Starting from the root set, do a graph traversal of reachable objects
  - For each live object, copy it into the other semispace (keeping track of mapping from old location to new location)
  - Once all objects are copied, update all pointers in root set and live objects to reflect the updated object locations
  - Switch semispaces