

CMSC 330: Organization of Programming Languages

Parameter Passing and More on Scoping

Parameter Passing in OCaml

- Quiz: What value is bound to `z`?

```
let add x y = x + y
let z = add 3 4
```

7

```
let add x y = x + y
let z = add (add 3 1) (add 4 1)
```

9

```
let r = ref 0
let add x y = (!r) + x + y
let set_r () = r := 3; 1
let z = add (set_r ()) 2
```

6

Actuals evaluated before call

Call-by-Value

- In *call-by-value* (*cbv*), arguments to functions are fully evaluated before the function is invoked
 - Also in OCaml, in `let x = e1 in e2`, the expression `e1` is fully evaluated before `e2` is evaluated
- C, C++, and Java also use call-by-value

```
int r = 0;
int add(int x, int y) { return r + x + y; }
int set_r(void) {
  r = 3;
  return 1;
}
add(set_r(), 2);
```

Another Puzzle

- Quiz: What value is bound to `z`?

```
let r = ref 0
let add x y = x + y
let set_r () = r := 3; 1
let z = add (!r) (set_r ())
```

- It depends on the *order of evaluation*
 - Usually this is very explicit
 - `e1`; `e2` (* evaluate `e1` before `e2` *)
 - Function arguments is one place it's confusing
 - May be specified in a language, or it may not be
 - May depend on optimization level
 - It's a bad habit to depend on it if you're not sure

Order of Evaluation

- Will OCaml raise a `Division_by_zero` exception?

```
let x = 0

if x != 0 && (y / x) > 100 then
  print_string "OCaml sure is fun"

if x == 0 || (y / x) > 100 then
  print_string "OCaml sure is fun"
```

- No: `&&` and `||` are *short-circuiting* in OCaml
 - `e1 && e2` evaluates `e1`. If false, it returns false. Otherwise, it returns the result of evaluating `e2`
 - `e1 || e2` evaluates `e1`. If true, it returns true. Otherwise, it returns the result of evaluating `e2`

Order of Evaluation (cont'd)

- C, C++, Java, and Ruby all short-circuit `&&`, `||`
- But some languages don't, like Pascal:

```
x := 0;
...
if (x <> 0) and (y / x > 100) then
  writeln('Sure OCaml is fun');
```

- So this would need to be written as

```
x := 0;
...
if x <> 0 then
  if y / x > 100 then
    writeln('Sure OCaml is fun');
```

Call-by-Value in Imperative Languages

- In C, C++, and Java, call-by-value has another feature

- What does this program print?

```
void f(int x) {
  x = 3;
}

int main() {
  int x = 0;
  f(x);
  printf("%d\n", x);
}
```

- Prints 0

Call-by-Value in Imperative Languages, con't.

- Actual parameter is copied to stack location of formal parameter

```
void f(int x) {
  x = 3;
}

int main() {
  int x = 0;
  f(x);
  printf("%d\n", x);
}
```

x	0
x	3

Call-by-Reference

- Alternative idea: Implicitly pass a *pointer* or *reference* to the actual parameter
 - If the function writes to it the actual parameter is modified

```
void f(int x) {
    x = 3;
}

int main() {
    int x = 0;
    f(x);
    printf("%d\n", x);
}
```



Call-by-Reference (cont'd)

- Advantages
 - The entire argument doesn't have to be copied to the called function
 - It's more efficient if you're passing a large (multi-word) argument
 - Can do this without explicit pointer manipulation
 - Allows easy multiple return values
- Disadvantages
 - Can you pass a non-variable (e.g., constant, function result) by reference?
 - It may be hard to tell if a function modifies an argument
 - What if you have *aliasing*?

Aliasing

- We say that two names are *aliased* if they refer to the same object in memory
 - C examples (this is what makes optimizing C hard)

```
int x;
int *p, *q; /*Note that C uses pointers to
            simulate call by reference */
p = &x; /* *p and x are aliased */
q = p; /* *q, *p, and x are aliased */
```

```
struct list { int x; struct list *next; }
struct list *p, *q;
...
q = p; /* *q and *p are aliased */
      /* so are p->x and q->x */
      /* and p->next->x and q->next->x... */
```

Aliasing Example

- What happens in the following function?

```
void f(int *x, int *y, int n) {
    int i;
    for (i = 0; i < n; i++) {
        x[i] += y[i];
        x[i] += y[i]; /* Supposed to add 2*y[i] to x[i] */
    }
}

int a[] = {1, 2, 3, 4, 5};
f(a, a, 5);
```

Call-by-Reference (cont'd)

- Call-by-reference is still around (e.g., C++), but seems to be less popular in newer languages
 - Older languages (e.g., Fortran, Ada, C with pointers) still use it
 - Possible efficiency gains not worth the confusion
 - “The hardware” is basically call-by-value
 - Although call by reference is not hard to implement and there may be some support for it

Call-by-Value Discussion

- Call-by-value is the standard for languages with side effects
 - When we have side effects, we need to know the order in which things are evaluated, otherwise programs have unpredictable behavior
 - Call-by-reference can sometimes give different results
 - Call-by-value specifies the order at function calls
- But there are alternatives to call by value and call by reference ...

Call-by-Name

- *Call-by-name (cbn)*
 - First described in description of Algol (1960)
 - Generalization of Lambda expressions (to be discussed later)
 - **Idea simple:** In a function:

Example:
add (a*b) (c*d) =
(a*b) + (c*d) ← **executed function**

```
Let add x y = x+y  
add (a*b) (c*d)
```

Then each use of *x* and *y* in the function definition is just a literal substitution of the actual arguments, (a*b) and (c*d), respectively
 - **But implementation:** Highly complex, inefficient, and provides little improvement over other mechanisms, as later slides demonstrate

Call-by-Name (cont'd)

- In *call-by-name (cbn)*, arguments to functions are evaluated at the last possible moment, just before they're needed

```
let add x y = x + y  
let z = add (add 3 1) (add 4 1)
```

OCaml; cbv; arguments evaluated here

Haskell; cbn; arguments evaluated here

```
add x y = x + y  
z = add (add 3 1) (add 4 1)
```

Call-by-Name (cont'd)

- What would be an example where this difference matters?

```
let cond p x y = if p then x else y
let rec loop n = loop n
let z = cond true 42 (loop 0)
```

OCaml; cbv; infinite recursion at call

```
cond p x y = if p then x else y
loop n = loop n
z = cond True 42 (loop 0)
```

Haskell; cbn; never evaluated because parameter is never used

But: Call by Name Anomalies

1. $P(x) \{x = x + x;\}$
 $Y = 2;$
 $P(Y);$
 $write(Y)$

← means $Y = Y + Y = 4$

2. But if $F(l) \{l = l + 1; return l;\}$

What is:

$int A[10];$

$l = 1;$

$P(A[F(l)])$

$P(A[F(l)]) \rightarrow A[F(l)] = A[F(l)] + A[F(l)] \rightarrow A[l++] = A[l++] + A[l++]$

$\rightarrow A[2] = A[3] + A[4]$

3. Write a program to exchange values of X and Y: (e.g., $swap(X, Y)$)

Usual way: $swap(x, y) \{t=x; x=y; y=t;\}$

- Cannot do it with call by name. Cannot handle both of following:
 $swap(l, A[l])$ $swap(A[l], l)$
- One of these must fail. Why?

Two Cool Things to Do with CBN

- CBN is also called *lazy evaluation*
 - (CBV is also known as *eager evaluation*)

- Build control structures with functions

```
let cond p x y = if p then x else y
```

- Build "infinite" data structures

```
integers n = n:(integers (n+1))
take 10 (integers 0) (* infinite loop in cbv *)
```

Three-Way Comparison

- Consider the following program under the three calling conventions
 - For each, determine i 's value and which $a[i]$ (if any) is modified

```
int i = 1;

void p(int f, int g) {
    g++;
    f = 5 * i;
}

int main() {
    int a[] = {0, 1, 2};
    p(a[i], i);
    printf("%d %d %d %d\n",
           i, a[0], a[1], a[2]);
}
```

Example: Call-by-Value

```
int i = 1;

void p(int f, int g) {
    g++;
    f = 5 * i;
}

int main() {
    int a[] = {0, 1, 2};
    p(a[i], i);
    printf("%d %d %d %d\n",
        i, a[0], a[1], a[2]);
}
```

i	a[0]	a[1]	a[2]	f	g
1	0	1	2		
				1	1
				5	2

Example: Call-by-Reference

```
int i = 1;

void p(int f, int g) {
    g++;
    f = 5 * i;
}

int main() {
    int a[] = {0, 1, 2};
    p(a[i], i);
    printf("%d %d %d %d\n",
        i, a[0], a[1], a[2]);
}
```

i/g	a[0]	a[1]/f	a[2]
1	0	1	2
2		10	
2		10	

Example: Call-by-Name

```
int i = 1;

void p(int f, int g) {
    g++;
    f = 5 * i;
}

int main() {
    int a[] = {0, 1, 2};
    p(a[i], i);
    printf("%d %d %d %d\n",
        i, a[0], a[1], a[2]);
}
```

i	a[0]	a[1]	a[2]
1	0	1	2
2			10
2			10

The expression `a[i]` isn't evaluated until needed, in this case after `i` has changed.

Other Calling Mechanisms

- *Call-by-result*
 - Actual argument passed by reference, but not initialized
 - Written to in function body (and since passed by reference, affects actual argument)
- *Call-by-value-result*
 - Actual argument copied in on call (like cbv)
 - Mutated within function, but does not affect actual yet
 - At end of function body, copied back out to actual
- These calling mechanisms didn't really catch on
 - They can be confusing in cases
 - Recent languages don't use them

In-out semantics

Most calling conventions (e.g., CBV, CBN) based upon implementation.

Ada parameters based upon semantics of the parameter.

in - argument value will be used in procedure.

out - parameter value will be used in calling program.

in out - both uses of arguments and parameters

P(X in integer; Y out integer; Z in out integer);

begin ... end;

In Ada 83, language definition allowed some latitude in implementation.

But this meant that the same program could give different answers from different standards conforming compilers

In Ada 95, more restricted: **in** integer is cbv, **out** integer is value-result, composite objects (e.g., arrays) is reference.

Simulating CBN with CBV

- Call-by-name is implemented by passing in a *thunk* that, when called, evaluates to the parameter
 - Within body, formal argument thunks are invoked to get actuals
 - A thunk is a function with no arguments

Simulating CBN with CBV (cont'd)

```
let cond p x y = if p then x else y
let rec loop n = loop n
let z = cond true 42 (loop 0)
```

– becomes... **Get 1st arg** **Return 2nd arg**

```
let cond p x y = if (p ()) then (x ()) else (y ())
let rec loop n = loop n (* didn't transform... *)
let z = cond (fun () -> true)
             (fun () -> 42)
             (fun () -> loop 0)
```

Never invoked

CBV versus CBN

- CBN is flexible- strictly more programs terminate
 - E.g., where we might have an infinite loop with cbv, we might avoid it with cbn by waiting to evaluate
- Order of evaluation is really hard to see in CBN
 - Call-by-name doesn't mix well with side effects (assignments, print statements, etc.)
- Call-by-name is more expensive since:
 - Functions have to be passed around
 - If you use a parameter twice in a function body, its thunk will be called twice
 - Haskell actually uses *call-by-need* (each formal parameter is evaluated only once, where it's first used in a function)

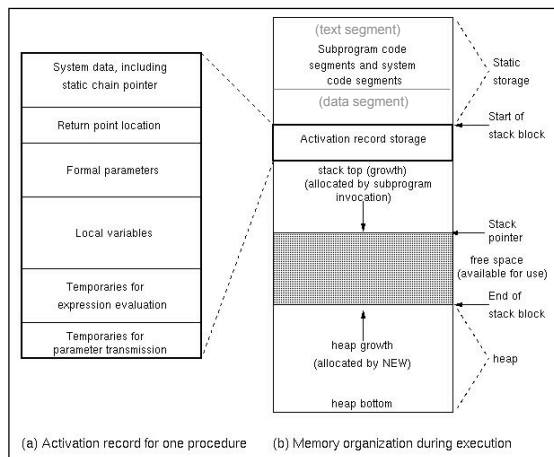
CBV versus CBN (cont'd)

- Call-by-name isn't very “mainstream”
 - Haskell solves these issues by not having side effects
 - But then someone invented “monads” so you can have side effects in a lazy language
- Call-by-name's benefits may not be worth its cost

How Function Calls Really Work

- Function calls are so important they usually have direct instruction support on the hardware
- We won't go into the details of assembly language programming
 - See CMSC 212, 311, 412, or 430
- But we will discuss just enough to know how functions are called

Machine Model (Generic UNIX)



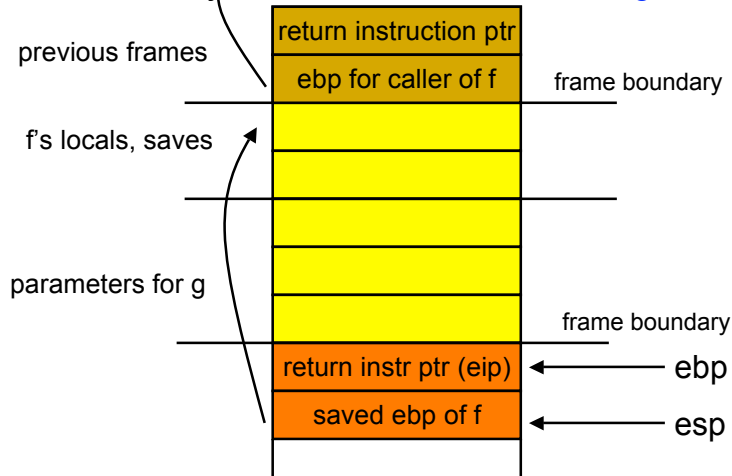
- The *text segment* contains the program's source code
- The *data segment* contains global variables, static data (data that exists for the entire execution and whose size is known), and the heap
- The *stack segment* contains the activation records for functions

Machine Model (x86)

- The CPU has a fixed number of *registers*
 - Think of these as memory that's really fast to access
 - For a 32-bit machine, each can hold a 32-bit word
- Important x86 registers
 - *eax* generic register for computing values
 - *esp* pointer to the top of the stack
 - *ebp* pointer to start of current stack frame
 - *eip* the program counter (points to next instruction in text segment to execute)

The x86 Stack Frame/Activation Record

- The stack just after `f` transfers control to `g`



CMSC 330

Based on Fig 6-1 in Intel ia-32 manual

33

x86 Calling Convention

- To call a function
 - Push parameters for function onto stack
 - Invoke `CALL` instruction to
 - Push current value of `eip` onto stack
 - I.e., save the program counter
 - Start executing code for called function
 - Callee pushes `ebp` onto stack to save it
- When a function returns
 - Put return value in `eax`
 - Invoke `LEAVE` to pop stack frame
 - Set `esp` to `ebp`
 - Restore `ebp` that was saved on stack and pop it off the stack
 - Invoke `RET` instruction to load return address into `eip`
 - I.e., start executing code where we left off at call

CMSC 330

34

Example

```
int f(int a, int b) {
    return a + b;
}

int main(void) {
    int x;

    x = f(3, 4);
}
```

`gcc -S a.c`

```
f:
    pushl   %ebp
    movl    %esp, %ebp
    movl    12(%ebp), %eax
    addl    8(%ebp), %eax
    leave
    ret

main:
    ...
    subl   $8, %esp
    pushl  $4
    pushl  $3
    call   f
1: addl   $16, %esp
    movl   %eax, -4(%ebp)
    leave
    ret
```

CMSC 330

35

Lots More Details

- There's a whole lot more to say about calling functions
 - Local variables are allocated on stack by the callee as needed
 - This is usually the first thing a called function does
 - Saving registers
 - If the callee is going to use `eax` itself, you'd better save it to the stack before you call
 - Passing parameters in registers
 - More efficient than pushing/popping from the stack
 - Etc...
- See other courses for more details

CMSC 330

36

Tail Calls

- A *tail call* is a function call that is the last thing a function does before it returns

```
let add x y = x + y
let f z = add z z (* tail call *)
```

```
let rec length = function
  [] -> 0
  | (_::t) -> 1 + (length t) (* not a tail call *)
```

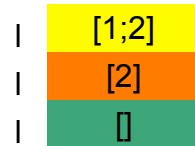
```
let rec length a = function
  [] -> a
  | (_::t) -> length (a + 1) t (* tail call *)
```

Tail Recursion

- Recall that in OCaml, all looping is via recursion
 - Seems very inefficient
 - Needs one stack frame for recursive call
- A function is *tail recursive* if it is recursive and the recursive call is a tail call

Tail Recursion (cont'd)

```
let rec length l = match l with
  [] -> 0
  | (_::t) -> 1 + (length t)
length [1; 2]
```



eax: 2

- However, if the program is tail recursive...
 - Can instead reuse stack frame for each recursive call

Tail Recursion (cont'd)

```
let rec length a l = match l with
  [] -> a
  | (_::t) -> (length (a + 1) t)
length 0 [1; 2]
```



eax: 2

- The same stack frame is reused for the next call, since we'd just pop it off and return anyway

Names and Binding

- Programs use *names* to refer to things
 - E.g., in `x = x + 1`, `x` refers to a variable
- A *binding* is an association between a name and what it refers to

```
- int x;           /* x is bound to a stack
                  location containing an
                  int */
- int f (int) { ... } /* f is bound to a
                  function */
- class C { ... }   /* C is bound to a class */
- let x = e1 in e2   (* x is bound to e1 *)
```

Name Restrictions

- Languages often have various restrictions on names to make lexing and parsing easier
 - Names cannot be the same as keywords in the language
 - OCaml function names must be lowercase
 - OCaml type constructor and module names must be uppercase
 - Names cannot include special characters like `;`, `:` etc
 - Usually names are upper- and lowercase letters, digits, and `_` (where the first character can't be a digit)
 - Some languages also allow more symbols like `!` or `-`

Names and Scopes

- Good names are a precious commodity
 - They help document your code
 - They make it easy to remember what names correspond to what entities
- We want to be able to reuse names in different, non-overlapping regions of the code

Names and Scopes (cont'd)

- A *scope* is the region of a program where a binding is active
 - The same name in a different scope can refer to a different binding (refer to a different program object)
- A name is *in scope* if it's bound to something within the particular scope we're referring to

Example

```
void w(int i) {
  ...
}

void x(float j) {
  ...
}

void y(float i) {
  ...
}

void z(void) {
  int j;
  char *i;
  ...
}
```

- **i** is in scope
 - in the body of **w**, the body of **y**, and after the declaration of **j** in **z**
 - but all those **i**'s are different
- **j** is in scope
 - in the body of **x** and **z**

Ordering of Bindings

- Languages make various choices for when declarations of things are in scope

Order of Bindings – OCaml

- **let x = e1 in e2** – **x** is bound to **e1** in scope of **e2**
- **let rec x = e1 in e2** – **x** is bound in **e1** and in **e2**

```
let x = 3 in
  let y = x + 3 in...   (* x is in scope here *)
```

```
let x = 3 + x in ...   (* error, x not in scope *)
```

```
let rec length = function
  [] -> 0
  | (h::t) -> 1 + (length t)  (* ok, length in scope *)
in ...
```

Order of Bindings – C

- All declarations are in scope from the declaration onward

```
int i;
int j = i;  /* ok, i is in scope */
i = 3;     /* also ok */
```

```
void f(...) { ... }

int i;
int j = j + 3;  /* error */
f(...);       /* ok, f declared */
```

```
f(...); /* may be error; need prototype (or oldstyle C) */

void f(...) { ... }
```

Order of Bindings – Java

- Declarations are in scope from the declaration onward, except for methods and fields, which are in scope throughout the class

```
class C {
    void f(){
        ...g()... // OK
    }

    void g(){
        ...
    }
}
```

Shadowing Names

- *Shadowing* is rebinding a name in an inner scope to have a different meaning
 - May or may not be allowed by the language

```
C
int i;

void f(float i) {
    {
        char *i = NULL;
        ...
    }
}
```

```
OCaml
let g = 3;;
let g x = x + 3;;
```

```
Java
void h(int i) {
    {
        float i; // not allowed
        ...
    }
}
```

Namespaces

- Languages have a “top-level” or outermost scope
 - Many things go in this scope; hard to control collisions
- Common solution seems to be to add a hierarchy
 - OCaml: Modules
 - `List.hd`, `String.length`, etc.
 - `open` to add names into current scope
 - Java: Packages
 - `java.lang.String`, `java.awt.Point`, etc.
 - `import` to add names into current scope
 - C++: Namespaces
 - `namespace f { class g { ... } }, f::g b`, etc.
 - `using namespace` to add names to current scope

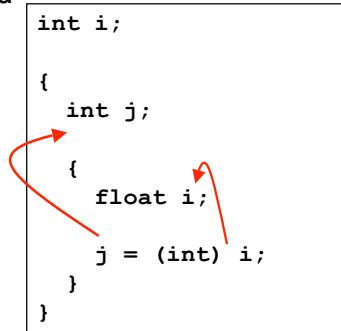
Mangled Names

- What happens when these names need to be seen by other languages?
 - What if a C program wants to call a C++ method?
 - C doesn't know about C++'s naming conventions
- For multilingual communication, names are often mangled into some flat form
 - E.g., `class C { int f(int *x, int y) { ... } }` becomes symbol `__ZN1C3fEPii` in g++
 - E.g., native `valueOf(int)` in `java.lang.String` corresponds to the C function `Java_java_lang_String_valueOf_I`

Static Scope Recall

- In *static scoping*, a name refers to its closest binding, going from inner to outer scope in the program text
 - Languages like C, C++, Java, Ruby, and OCaml are statically scoped

```
int i;
{
  int j;
  {
    float i;
    j = (int) i;
  }
}
```



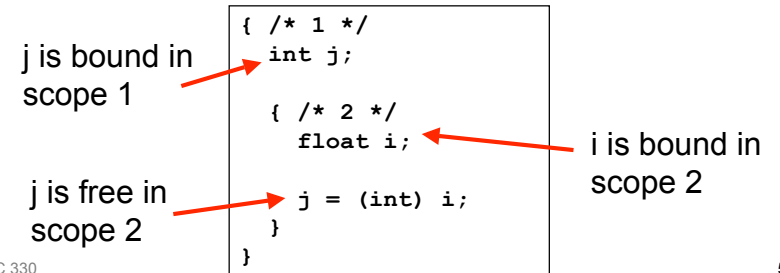
CMSC 330

53

Free and Bound Variables

- The *bound variables* of a scope are those names that are declared in it
- If a variable is not bound in a scope, it is *free*
 - The bindings of variables which are free in a scope are "inherited" from declarations of those variables in outer scopes in static scoping

```
{ /* 1 */
  int j;
  { /* 2 */
    float i;
    j = (int) i;
  }
}
```



CMSC 330

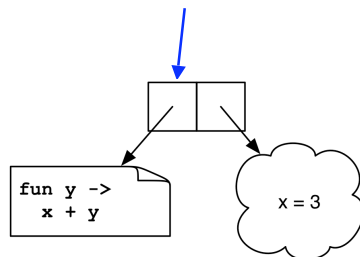
54

Static Scoping and Nested Functions

- To allow arbitrary nested functions with higher-order functions and static scoping, we needed closures

```
let add x = (fun y -> x + y)
```

(add 3) 4 → <closure> 4 → 3 + 4 → 7



CMSC 330

55

Nested Functions (cont'd)

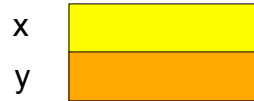
- We need closures for *upward funargs*
 - Functions that are returned by other functions
- If we only have *downward funargs*, then we don't need full closures
 - These are functions that are only passed inward
 - So when they're called, any nonlocal variables they access from outer scopes are still around

CMSC 330

56

Example

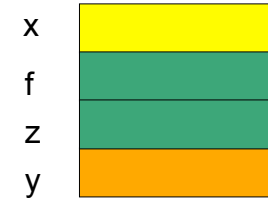
```
let f x =  
  let g y = x + y in  
  g 3
```



- When `g` is called, `x` is still on the stack

Example

```
let app f z = f z  
  
let f x =  
  let g y = x + y in  
  app g 3
```



- When `g` is called, `x` is still on the stack

Downward Funargs

- It turns out that if we only pass functions downward, there are cheaper implementation strategies for static scoping than closures
- They're called *static links* and *displays*, and they're used by
 - Pascal and Algol-family languages
 - gcc nested functions
- We won't go into details, though (CMSC 430 covers these in exciting details.)

Dynamic Scope

- In a language with *dynamic scoping*, a name refers to its closest binding *at runtime*
 - LISP was the common example

Scheme (top-level scope only is dynamic)

```
(define f (lambda () a))  
; defines a no-argument function which returns a  
  
(define a 3) ; bind a to 3  
(f) ; calls f and returns 3  
(define a 4)  
(f) ; calls f and returns 4
```

Nested Dynamic Scopes

- Full dynamic scopes can be nested
 - Static scope relates to the program text
 - Dynamic scope relates to program execution trace

Perl (the keyword `local` introduces dynamic scope)

```
$l = "global";

sub A {
  local $l = "local";
  B();
}

sub B { print "$l\n"; }

B(); A(); B();
```

Static vs. Dynamic Scope

Static scoping

- Local understanding of function behavior
- Know at compile-time what each name refers to
- A bit trickier to implement

Dynamic scoping

- Can be hard to understand behavior of functions
- Requires finding name bindings at runtime
- Easier to implement (just keep a global table of stacks of variable/value bindings)