Programming Languages: If you're not tired, you're not doing it right

Professor Keith Sullivan

Basic Terminology

- Name: A character string to represent something
- Binding: An attachment of a value to a name
- Scope: The part of code where a binding is active
- Referencing Environment: The set of active bindings at the point of an expression
- Allocation: Setting aside space for an object
- Lifetime: The time when an object is in memory

Naming Issues: Example 1

We need to know what thing a *name* refers to in our programs.

Consider, in Perl:

```
$x=1;
sub foo() { $x = 5; }
sub bar() { local $x = 2;
            foo();
            print $x,"\n"; }
bar();
```

What gets printed for x?

Naming Issues: Example 2

We need to know what thing a *name* refers to in our programs.

Consider, in Scheme:

```
(define x 1)
(define (foo x)
  (lambda () (display x)))
((foo 5))
(display x)
```

What gets printed for *x*?

Naming Issues: Example 3

We need to know what thing a *name* refers to in our programs.

Consider, in C++:

```
char* foo() {
  char s[20];
  cin >> s;
  return s;
}

void bar (char* x) { cout << x << endl; }

int main() { bar(foo()); return 1; }
```

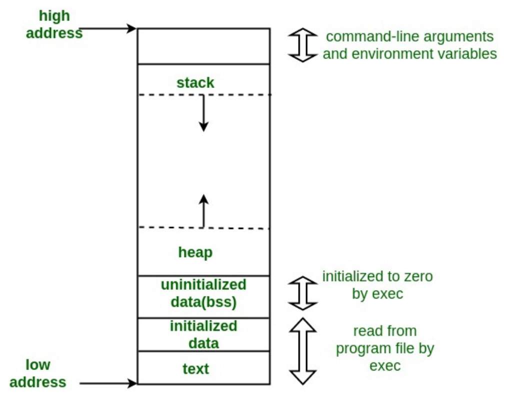
What gets printed for *x*?

When to bind?

- Language design: language semantics
- Language implementation
  - Precision, coupling of I/O
- Program writing
- Compile time
- Linking
- Loading
- Run time

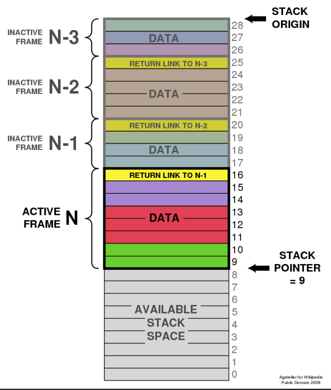
Earlier binding times have greater efficiency, while later binding times have greater flexibility.

Memory Layout


high address
command-line arguments and environment variables
stack
heap
uninitialized data(bss)
initialized to zero by exec
initialized data
read from program file by exec
text
low address

Pointers

- Stack frame: a call to a function (and associated information) that has not yet returned
- Stack pointer: points to the top of the stack
- Frame pointer: points to the top of the frame
- Program counter: address which points to the next instruction



Allocation

Defined by the compiler/interpreter writer, not the language specification.

- Static Allocation
  Allocation fixed at compile time
- Stack Allocation
  Follows function calls
- Heap Allocation
  Done at run time as objects created and destroyed

Static Allocation

- Objects are given an absolute memory address at compile time
- Our program can then access them really quickly
- Memory size does not change
- No memory re-usability (objects stay in memory until program terminates)

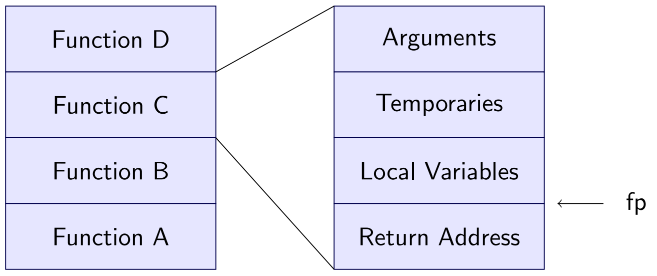
Examples:

- Global variables
- Literals (e.g., numbers, strings)
- *Everything* in Fortran77

What about recursion?

Stack Allocation

Run-time stack is used for function calls.
Contiguous block of memory
No decisions about where to allocate



Stack example

What does the stack look like for this C code?

```
int g(int x) { return x*x; }

int f(int y) {
    int x = 3 + g(y);
    return x;
}

int main() {
    int n = 5;
    f(n);
}
```

Heap Allocation

- A heap is a pile of memory that can used as needed for runtime memory allocation.
- No specific limit on memory size, randomly allocated
- Manual allocation and de-allocation
- Slowest method of allocation
- How to manage?
  - First fit
  - Best Fit
  - Worst Fit

Fragmentation is a problem!

Garbage Collection

- De-allocation of objects
  - Explicitly done by user (C, C++, Rust)
  - Done by language when object is no longer reachable
- Garbage collection identifies and reclaims unreachable objects
- Manual de-allocation errors are common and costly
  - Dangling reference
  - Memory leaking

Scoping

Scoping

- Single Global Scope
  Just one symbol table
- Dynamic Scope
  Stacks of scopes; depends on run-time behavior
- Lexical Scope
  Scope is based on the syntaxical (lexical) structure of the code

What is a scope?

Certain language structures create a new scope. For example:

```
int temp = 5;

// Sorts a two-element array.
void twosort(int A[]) {
  if (A[0] > A[1]) {
    int temp = A[0];
    A[0] = A[1];
    A[1] = temp;
  }
}

int main() {
  int arr[] = {2, 1};
  twosort(arr);
  cout << temp;
}
```

Nested Scopes

In C++, nested scopes are made using curly braces ({ and }).
The scope resolution operator :: allows jumping between scopes manually.

In most languages, function bodies are a nested scope.
Often, *control structure* blocks are also (e.g. **for**, **if**, etc.)

Lexical scoping follows the nesting of scopes in the actual source code (as it is parsed).
Dynamic scoping follows the nesting of scopes as the program is executed.

Declaration Order

In many languages, variables must be declared before they are used. (Otherwise, the first use constitutes a declaration.)

In C/C++, the scope of a name starts at its declaration and goes to the end of the scope. Every name must be declared before its first use, because names are resolved as they are encountered.

C++ and Java make an exception for names in class scope.
Scheme doesn't resolve names until they are evaluated.

Declaration Order and Mutual Recursion

Consider the following familiar code:

```
void exp() { atom(); exptail(); }

void atom() {
  switch(peek()) {
    case LP: match(LP); exp(); match(RP);
    break;
    ...
  }
}
```

Mutual recursion in C/C++ requires forward declarations,
i.e., function prototypes.

These wouldn't be needed within a class definition or in Scheme.
C# and Pascal solve the problem in a different way...

Dynamic vs. Lexical Scope

Dynamic Scope

- Bindings determined by *most recent declaration* (at run time)
- The same name can refer to many different bindings!
- Examples:

Lexical Scope

- Bindings determined from lexical structure at compile-time
- The same name always refers to the same binding.
- More common in "mature" languages
- Examples:

Dynamic vs. Lexical Example

```
n : integer

procedure first()
  n := 1

procedure second()
  n : integer
  first()

n := 2
if read_integer() > 0
  second()
else
  first()
write_integer(n)
```

How does the behavior differ between a dynamic or lexically scoped language?

Problem with Dynamic Scoping

```
max_score : integer

function scaled_score(raw_score : integer)
  return raw_score / max_score * 100

function foo()
  max_score : real := 0
  foreach student in class
    student.percent :=
          scaled_score(student.points)
    if student.percent > max_score:
      max_score := student.percent
```

Implementing Dynamic Scoping

A Central Reference Table is used to implement dynamic scope.

This **global** object contains:

- A mapping of names to *stacks of values*
  Declaring a new binding pushes onto the stack; exiting that binding's scope pops off the top of the stack.
- A stack of sets of names. Each set stores the names declared in some scope (so we know what bindings to pop).

Example

```
{
  new x := 0;
  new i := -1;
  new g := lambda z { ret := i; };
  new f := lambda p {
    new i := x;
    ifelse (i > 0) { ret := p@0; }
    {
      x := x + 1;
      i := 3;
      ret := f@g;
    }
  };
  write f@(lambda y {ret := 0});
}
```

Central Reference Table Example

Names in Scope	x	i	g	f
{x,i,g,f}	0	-1	$z \to i$	$p \to \ldots$

```
{
  new x := 0;
  new i := −1;
  new g := lambda z { ret := i; };
  new f := lambda p {
    new i := x;
    ifelse (i > 0) { ret := p@0; }
    {
      x := x + 1;
      i := 3;
      ret := f@g;
    }
  };
  write f@(lambda y {ret := 0});
}
```

Central Reference Table Example

Names in Scope	x	i	g	f	p	ret
{x,i,g,f}	0	-1	$z \to i$	$p \to \ldots$	$y \to 0$	(unset)
{p, ret}						

```
{
  new x := 0;
  new i := −1;
  new g := lambda z { ret := i; };
  new f := lambda p {
    new i := x;
    ifelse (i > 0) { ret := p@0; }
    {
      x := x + 1;
      i := 3;
      ret := f@g;
    }
  };
  write f@(lambda y {ret := 0});
}
```

Central Reference Table Example

Names in Scope	x	i	g	f	p	ret
{x,i,g,f}	0	-1	$z \to i$	$p \to \ldots$	$y \to 0$	(unset)
{p, ret, i}		0				

```
{
  new x := 0;
  new i := −1;
  new g := lambda z { ret := i; };
  new f := lambda p {
    new i := x;
    ifelse (i > 0) { ret := p@0; }
    {
      x := x + 1;
      i := 3;
      ret := f@g;
    }
  };
  write f@(lambda y {ret := 0});
}
```

Central Reference Table Example

Names in Scope	x	i	g	f	p	ret
{x,i,g,f}	1	-1	z → i	p → . . .	y → 0	(unset)
{p, ret, i}		3			z → i	(unset)
{p, ret}						

```
{
  new x := 0;
  new i := −1;
  new g := lambda z { ret := i; };
  new f := lambda p {
    new i := x;
    ifelse (i > 0) { ret := p@0; }
    {
      x := x + 1;
      i := 3;
      ret := f@g;
    }
  };
  write f@(lambda y {ret := 0});
}
```

Central Reference Table Example

Names in Scope	x	i	g	f	p	ret	z
{x,i,g,f}	1	-1	z → i	p → . . .	y → 0	(unset)	0
{p, ret, i}		3			z → i	(unset)	
{p, ret, i}		1				(unset)	
{z, ret}							

```
{
  new x := 0;
  new i := −1;
  new g := lambda z { ret := i; };
  new f := lambda p {
    new i := x;
    ifelse (i > 0) { ret := p@0; }
    {
      x := x + 1;
      i := 3;
      ret := f@g;
    }
  };
  write f@(lambda y {ret := 0});
}
```

Central Reference Table Example

Names in Scope	x	i	g	f	p	ret	z
{x,i,g,f}	1	-1	z → i	p → . . .	y → 0	(unset)	0
{p, ret, i}		3			z → i	(unset)	
{p, ret, i}		1				1	
{z, ret}							

```
{
  new x := 0;
  new i := −1;
  new g := lambda z { ret := i; };
  new f := lambda p {
    new i := x;
    ifelse (i > 0) { ret := p@0; }
    {
      x := x + 1;
      i := 3;
      ret := f@g;
    }
  };
  write f@(lambda y {ret := 0});
}
```

Central Reference Table Example

Names in Scope	x	i	g	f	p	ret	z
{x,i,g,f}	1	-1	z → i	p → ...	y → 0	(unset)	
{p, ret, i}		3			z → i	1	
{p, ret, i}		1					

```
{
  new x := 0;
  new i := −1;
  new g := lambda z { ret := i; };
  new f := lambda p {
    new i := x;
    ifelse (i > 0) { ret := p@0; }
    {
      x := x + 1;
      i := 3;
      ret := f@g;
    }
  };
  write f@(lambda y {ret := 0});
}
```

Central Reference Table Example

Names in Scope	x	i	g	f	p	ret	z
{x,i,g,f}	1	-1	z → i	p → ...	y → 0	1	
p, ret, i		3					

```
{
  new x := 0;
  new i := −1;
  new g := lambda z { ret := i; };
  new f := lambda p {
    new i := x;
    ifelse (i > 0) { ret := p@0; }
    {
      x := x + 1;
      i := 3;
      ret := f@g;
    }
  };
  write f@(lambda y {ret := 0});
}
```

Lexical Scope Tree

Name resolution in lexical scoping follows a scope tree

- Every (nested) scope is a node
- The root is the global scope
- Nodes contain names defined in that scope
- To determine active bindings, follow the tree up until you find the name

Example

```
{
  new x := 0;
  new i := −1;
  new g := lambda z { ret := i; };
  new f := lambda p {
    new i := x;
    ifelse (i > 0) { ret := p@0; }
    {
      x := x + 1;
      i := 3;
      ret := f@g;
    }
  };
  write f@(lambda y {ret := 0});
}
```

The class of functions

- Third Class: never treated as a variable
- Second Class: Passed as parameters to other functions
- First Class: Also returned from a function and assigned a variable

Implementing Lexical Scope

With lexical scoping, binding rules get more complicated when functions have more flexibility:

- Third Class Functions
  "Static links" into function call stack
- Second Class Functions
  "Dynamic links" into function call stack
- Third Class Functions
  Must use Frames

Lexical Scope with 1st Class Functions

What happens here?

```
{
  new f := lambda x {
    new g := lambda y {
      ret := x * y;
    };
  };

  new h:= f@2;
  write h@3;
}
```

Frames

A frame is a data structure that represents the referencing environment of part of a program. It contains:

- A link to the parent frame
  This corresponds to the enclosing scope
- A symbol table

Looking up a name means checking the current frame, then recursively looking in the parent frame.

Function calls create new frames.

Closures

How are functions represented as values? With a **closure**

Recall that a closure is a function definition with its referencing environment. In a frame, we represent this as a pair:

- The function definition (parameters and body)
- Link to the frame where the function was defined.

Example

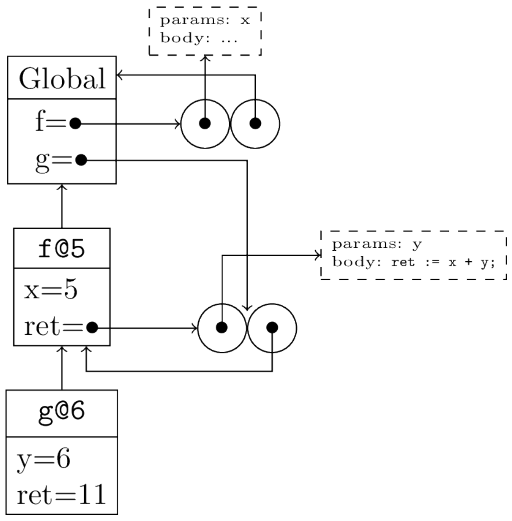
```
new f := lambda x {
  ret := lambda y {
    ret := x + y;
  };
;

new g := f@5;
write g@6;
```

Example

```
new f := lambda x {
  ret := lambda y {
    ret := x + y;
  };
;

new g := f@5;
write g@6;
```


params: x
body: ...
Global
f=
g=
params: y
body: ret := x + y;
f@5
x=5
ret=
g@6
y=6
ret=11