Names, Scopes, Bindings

CMPSC 461
Programming Language Concepts
Penn State University
Fall 2016

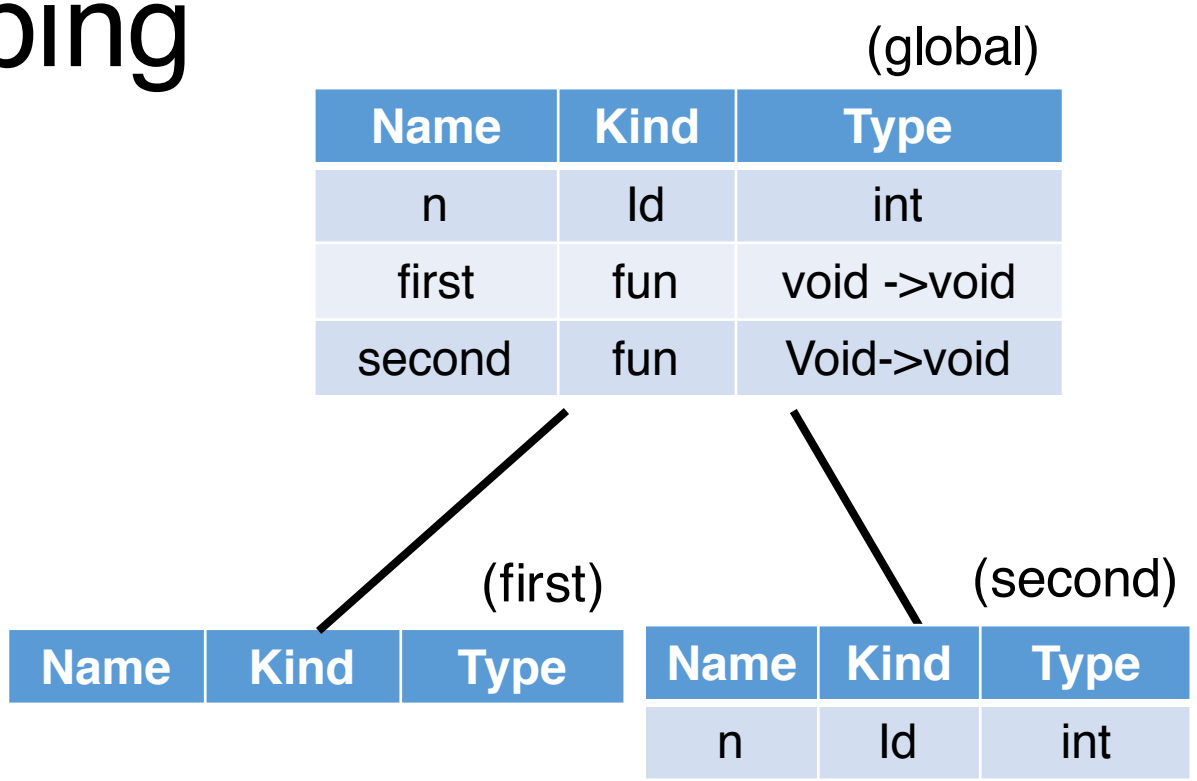
Static Scoping

```
int n=2;

void first() {
 n = 1
}

void second() {
 int n=0;
 first();
}

first();
second();
```



Symbol tables determined at compile time
Search for the binding from the table for the current scope

Dynamic Scoping

```
int n=2;

void first() {
 n = 1
}

void second() {
 int n=0;
 first();
}

first();
second();
```

(global)

Name	Kind	Type
n	Id	int
first	fun	void ->void
second	fun	Void->void

(first)

Name	Kind	Type

Tables when control flow reaches the ***first execution*** of function first

=>

Modifies global n

```
int n=2;

void first() {
 n = 1
}

void second() {
 int n=0;
 first();
}

first();
second();
```

(global)

Name	Kind	Type
n	Id	int
first	fun	void ->void
second	fun	Void->void

(second)

Name	Kind	Type
n	Id	int

(first)

Name	Kind	Type

Tables when control flow reaches the ***second execution*** of function first (from function second)

=>

modifies n defined in second

Dynamic Scoping

```
int n=2;

void first() {
 n = 1
}

void second() {
 int n=0;
 first();
}

first();
second();
```

Symbol tables changes at run time!
Always use most recent, active binding

Scope vs. Lifetime

Scope refers to visibility of a binding/name

Lifetime refers to creation/destruction of storage

Scope and lifetime are usually connected, but not always:

- Functions returned as values

```
(let ((x 17))
 (lambda (z) (+ z x)))
```

- Static variables

If a subroutine is passed as a parameter, when are dynamic/static rules apply?

```
function F(int x) {
    function G(fx) {
        int x = 13;
        fx();
    };
    function H() {
        print x;
    };
    G(H);
};
```

which x?

What's the parent of H's symbol table?

Who's the parent of H's symbol table?

```
function F(int x) {
    function G(fx) {
        int x = 13;
        fx();
    };
    function H() {
        print x;
    };
    G(H);
};
```

Shallow Binding: when the subroutine is called

Deep Binding: when reference is created

Shallow Binding: use the environment at call time

```
function F(int x) {
    function G(fx) {
        int x = 13;
        fx();
    };
    function H() {
        print x;
    };
    G(H);
};
```

(F)

Name	Kind	Type
x	para	int
G	fun	fun ->void
H	fun	void->void

(G)

Name	Kind	Type
fx	fun	fun
x	Id	int

Established at call time

(H)

Name	Kind	Type

Deep Binding: use the environment when ref. is created

```
function F(int x) {
    function G(fx) {
        int x = 13;
        fx();
    };
    function H() {
        print x;
    };
    G(H);
};
```

(F)

Name	Kind	Type
x	para	int
G	fun	fun ->void
H	fun	void->void

Established when ref. is created

(H)

Name	Kind	Type

Impl. of Deep Binding: Function Closures

```
function F(int x) {
    function G(fx) {
        int x = 13;
        fx();
    };
    function H() {
        print x;
    };
    G(H);
};
```

Pass in a function closure

A function closure contains:

- A pointer to the function
- The current symbol table (or all symbol tables, depending on implementation)

Shallow, Deep Binding and Static, Dynamic Scoping

Dynamic scoping

- Both shallow and deep binding are implemented
- Shallow binding has a higher cost at run time

Static scoping

- Shallow binding has never been implemented

So far, one-to-one mapping between names and visible objects

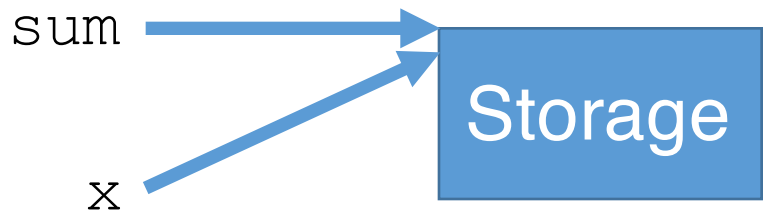
In many languages, we also have:

Multiple names to same object

Multiple objects the same name

Alias

```
double sum, sum_of_squares;
void accumulate(double& x) {
    sum += x;
    sum_of_squares += x * x;
}
accumulate(sum);
```



Overloading

So far, two functions with same name cannot coexist (one hides the other)

Overloading: uses the number or type of parameters to distinguish functions

```
void print (int a) {…};
void print (boolean a){…};
void print (char a){…};
Print (3);
```

Overloading

```
void print (int a) {…};
void print (boolean a){…};
void print (char a){…};
print (3);
```

Symbol Table

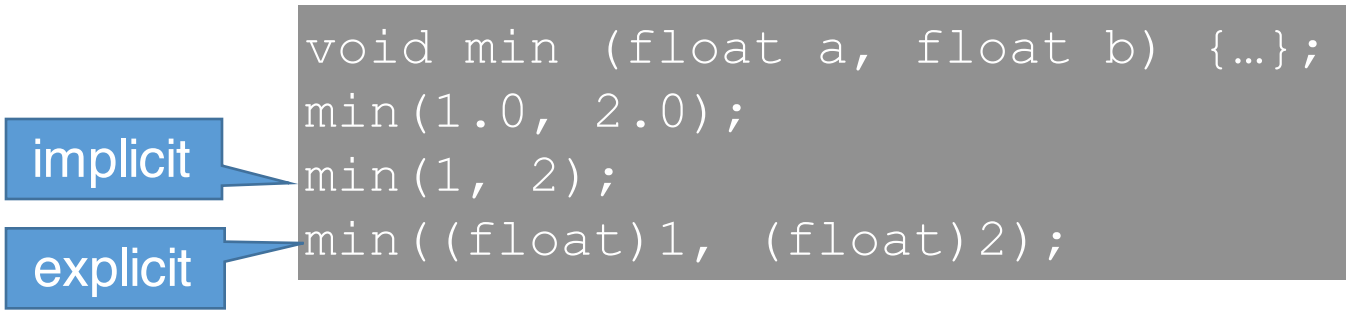
Name	Kind	Type
print	fun	int->void
print	fun	boolean->void
print	fun	char->void

When check statement `print(3)`:

1. All def. of “print” are returned
2. The proper def. picked based on number or type of para. (3)

Coercion

Coercion: the process by which a compiler automatically converts a value of one type to a value with another type



Coercion vs. Overloading

Coercion converts parameters to fit function def.

Overloading allows compiler to pick function implementation that fits

Coercion

```
void min (float a, float b)
min(1.0, 2.0);
min(1, 2);
```

Overloading

```
void min (float a, float b)
void min (int a, int b)
min(1.0, 2.0);
min(1, 2);
```

Polymorphism: function with multiple forms

Parametric polymorphism (fun. with a set of types)

- Explicit para. polymorphism (Genericity, aka. Templates in C++)

```
template <class T>
T GetMax (T a, T b) {
  T result;
  result = (a>b)? a : b;
  return (result);
}
```

Polymorphism

Parametric polymorphism (fun. with a set of types)

- Explicit para. polymorphism
- Implicit para. polymorphism (Lisp, Scheme, ML)

```
(define (min a b) (if (< a b) a b))
```

No mention of type

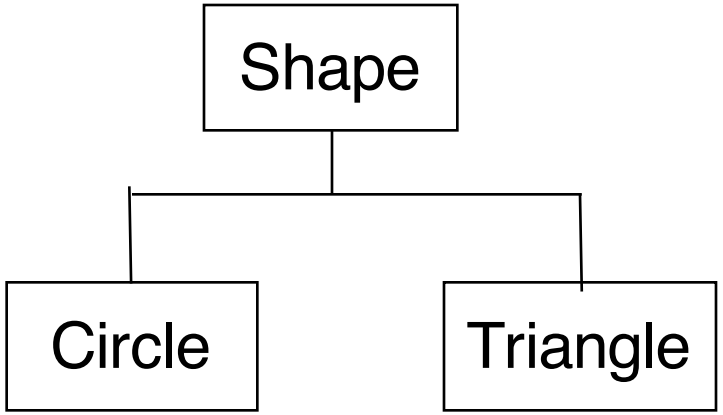
In Haskell

```
min a b = if a < b then a else b
```

Polymorphism

Subtype polymorphism (fun. with one type, and its refinements)

```
Circle c;
Triangle t;
int height (Shape s)
height (c);
height (t);
```



Coercion, Overloading, Polymorphism

Coercion converts parameters to fit function def.

Overloading allows compiler to pick function implementation that fits

Polymorphism allows one function with multiple uses