

# CMSC 330: Organization of Programming Languages

---

Names & Binding,  
Type Systems

# Language Features Covered Thus Far

---

## ▶ Ruby

- Implicit declarations
- Dynamic typing

```
{ x = 1 }
```

```
{ x = 1 ; x = "foo" }
```

## ▶ OCaml

- Functional programming
- Type inference
- Higher-order functions
- Static (lexical) scoping
- Parametric polymorphism
- Modules

```
add 1 (add 2 3)
```

```
let x = x+1    ( x : int )
```

```
let rec x = fun y -> x y
```

```
let x = let x = ...
```

```
let x y = y    ( 'a -> 'a )
```

```
module foo struct ... end
```

# Programming Languages Revisited

---

## ▶ Characteristics

- Artificial language for **precisely** describing algorithms
- Used to control behavior of machine / computer
- Defined by its syntax & semantics

## ▶ Syntax

- Combination of meaningful text symbols
  - Examples: if, while, let, =, ==, &&, +

## ▶ Semantics

- Meaning associated with syntactic construct
  - Examples:  $x = 1$  vs.  $x == 1$

# Comparing Programming Languages

---

## ▶ Syntax

- Differences usually superficial

- C / Java                    `if (x == 1) { ... } else { ... }`
- Ruby                        `if x == 1 ... else ... end`
- OCaml                      `if (x = 1) then ... else ...`

- Can cope with differences easily with experience

- Though may be annoying initially

- You should be able to learn new syntax quickly

- Just keep language manual / examples handy



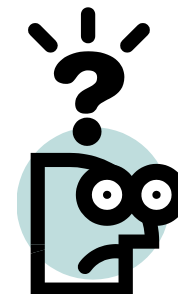
# Comparing Prog. Languages (cont.)

---

## ► Semantics

- Differences may be major / minor / subtle

	Physical Equality	Structural Equality
Java	<code>a == b</code>	<code>a.equals(b)</code>
C	<code>a == b</code>	<code>*a == *b</code>
Ruby	<code>a.equal?(b)</code>	<code>a == b</code>
OCaml	<code>a == b</code>	<code>a = b</code>



- Explaining these differences a major goal for 330
- Will be covering different features in upcoming lectures

# Programming Language Features

---

- ▶ Paradigm
  - Imperative
  - Object oriented
  - Functional
  - Logical
- ▶ Higher-order functions
  - Closures
- ▶ Declarations
  - Explicit
  - Implicit
- ▶ Type system
  - Typed vs. untyped
  - Static vs. dynamic
  - Weak vs. strong (type safe)

# Programming Language Features (cont.)

---

- ▶ Names & binding
  - Namespaces
  - Static (lexical) scopes
  - Dynamic scopes
- ▶ Parameter passing
  - Call by value
  - Call by reference
  - Call by name
    - Eager vs. lazy evaluation
- ▶ Polymorphism
  - Ad-hoc
    - Subtype
    - Overloading
  - Parametric
    - Generics
- ▶ Parallelism
  - Multithreading
  - Message passing

# Names & Binding Overview

---

- ▶ Bindings and declarations
- ▶ Order of bindings
- ▶ Namespaces
- ▶ Static (lexical) scopes
- ▶ Dynamic scopes



# 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 *)`

# Explicit vs. Implicit Declarations

---

- ▶ Explicit declarations identify allowed names
  - Variables must be declared before used

C, Java, C++, etc.

```
void foo(int y) {  
  int x;  
  x = y + 1;  
  return x + y;  
}
```

OCaml

```
let foo y =  
  let x = y + 1 in  
  x + y;;
```

declaration

use

# Explicit vs. Implicit Declarations

---

- ▶ Allowed names also declared implicitly
  - Variables do not need to be declared
    - Implicit declaration when first assigned to

Ruby

```
def foo(y)
  x = y + 1;
  return x + y;
end
```

declared implicitly,  
when assigned

Also: Perl, Python

# Name Restrictions

---

- ▶ Languages often have various restrictions on names to make scanning 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.)

---

- ▶ 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



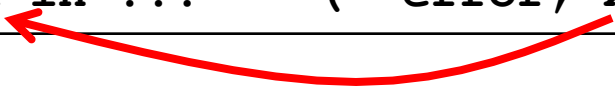
# Order of Bindings – OCaml

---

- ▶ `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 */
```

```
void f(...);  
  
void f(...) { .. 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
  - Methods are mutually recursive, by default

```
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  
        ...  
    }  
}
```

# Scoping, Shadowing, and Declarations

---

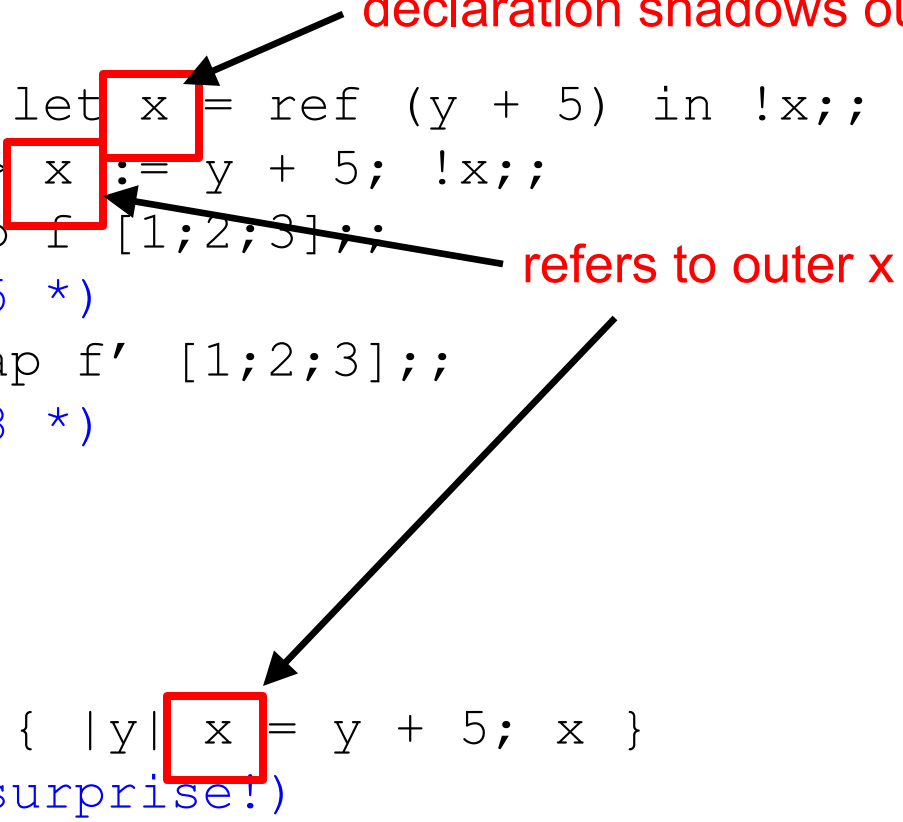
- ▶ Explicit declarations typically made at the outset of a scope
  - `{ int x; .... /* x valid here */ ... } /* x out of scope */`
  - Explicit declaration clarifies shadowing
- ▶ Implicit declarations occur within a scope
  - Not always immediately clear which scope you are in
  - May inadvertently use a name in an outer scope
    - No shadowing

# Shadowing and Implicit/Explicit Decls

---

## OCaml

```
let x = ref 5;;  
let f = fun y -> let x = ref (y + 5) in !x;;  
let f' = fun y -> x := y + 5; !x;;  
let gs = List.map f [1;2;3];;  
!x;; (* returns 5 *)  
let gs' = List.map f' [1;2;3];;  
!x;; (* returns 8 *)
```



## Ruby

```
x = 5  
arr = [1,2,3]  
gs = arr.collect { |y| x = y + 5; x }  
x ## returns 8 (surprise!)
```

# Namespaces

---

- ▶ Languages have a “top-level” or outermost scope
  - Many things go in this scope; hard to control collisions
- ▶ Common solution is to add a hierarchy
  - OCaml: Modules
    - List.hd, String.length, etc.
    - open to add names into current scope
    - Can also nest modules inside of other modules
  - 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

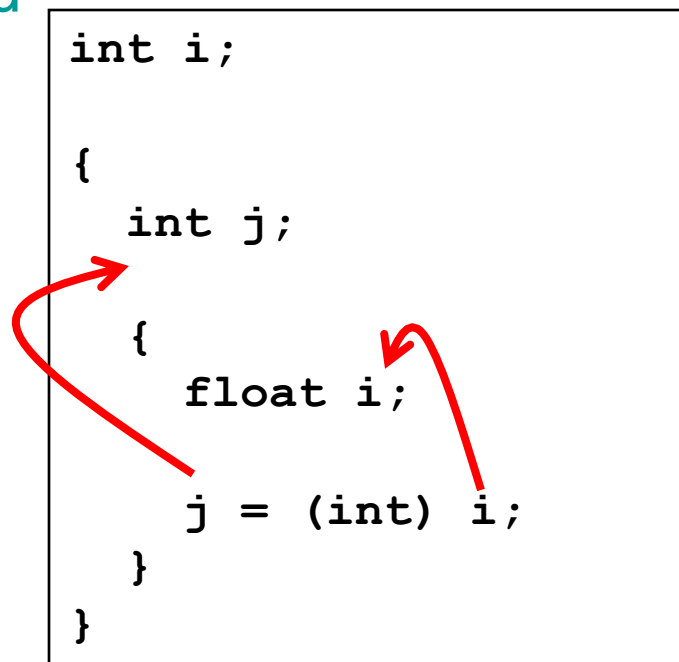
# Static Scoping (revisited)

---

- ▶ 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;
  }
}
```

A diagram illustrating static scoping with nested scopes. The code is enclosed in a black rectangular box. It shows three nested scopes: an outermost scope with 'int i;', a middle scope with 'int j;', and an innermost scope with 'float i;'. A red arrow points from the 'float i;' declaration to the 'j = (int) i;' assignment, indicating that the 'i' in the assignment refers to the 'float i;' in the innermost scope. Another red arrow points from the 'j = (int) i;' assignment to the 'int j;' declaration in the middle scope, indicating that the 'j' in the assignment refers to the 'int j;' in the middle scope.



# 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

j is bound in  
scope 1

j is free in  
scope 2

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

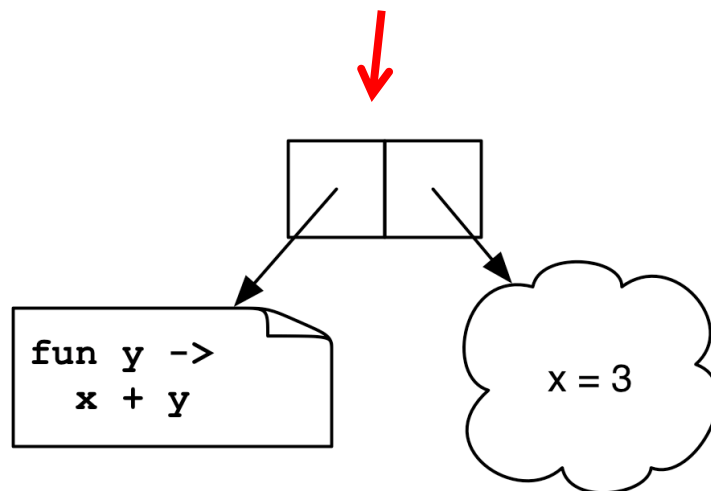
i is bound in  
scope 2

# Static Scoping and Nested Functions

- ▶ Closures needed when
  - Nested function declarations
  - Static scoping
  - Returning a function from function call (upwards funargs)

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

```
(add 3) 4 → <closure> 4 → 3 + 4 → 7
```



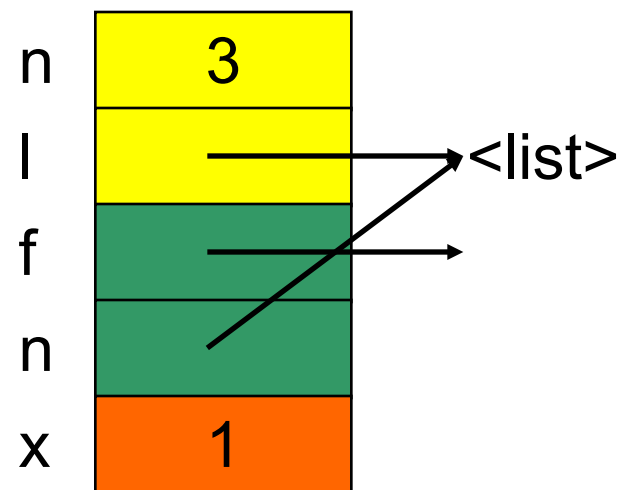
# Dynamic Scoping

- ▶ In a language with **dynamic scoping**, a name refers to its closest binding **at runtime**

```
let map (f, n) = match n with  
  [] -> []  
  | (h::t) -> (f h)::(map (f, t))
```

```
let addN (n, l) =  
  let add x = n + x in  
  map (add, l)
```

```
addN (3, [1; 2; 3])
```



- ▶ Value of `n` in `add`
  - Dynamic scope: reads it off the stack (`n = <list>`)
  - Static scope: lexical binding (`n = param n to addN`)

# Static vs. Dynamic Scoping

---

## Static scoping

- Local understanding of function behavior
- Know at compile-time what each name refers to
- A little more work to implement (keep a link to the lexical nesting scope in stack frame)

## Dynamic scoping

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

# Types

---

- ▶ Typed vs. untyped languages
- ▶ Type safety
- ▶ Static vs. dynamic type checking
- ▶ Weak vs. strong typing
  - Not great terms; mentioned for historical reasons

# Typed vs. Untyped Languages

---

## ▶ Typed language

- Operations are only valid for values of specific types
  - $2 * 3 = 6$
  - “foo” \* “bar” = undefined

## ▶ Untyped language

- All operations are valid for all values
- Treat all values as sequences of 0's and 1's
- Very few languages are untyped
  - Assembly languages, FORTH (maybe)

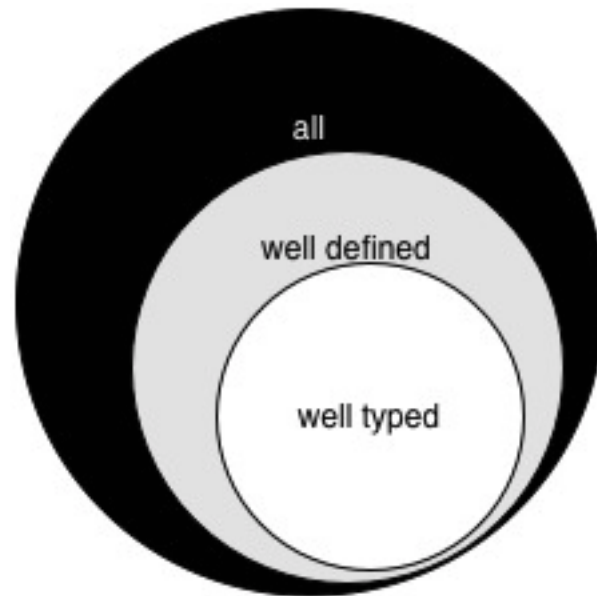
# Type Safety

---

- ▶ Well-typed
  - A **well-typed** program passes the language's type system
    - The “type system” depends on the language
    - Definition is nuanced for dynamically typed languages
- ▶ Going wrong
  - The language definition deems the program nonsensical
    - “Colorless green ideas sleep furiously”
    - If the program were to be run, anything could happen
    - `char buf[4]; buf[4] = 'x'; // undefined!`
- ▶ Type safe = “Well-typed programs never go wrong”
  - Robin Milner, 1978

# Type Safety is Conservative

---



<http://www.pl-enthusiast.net/2014/08/05/type-safety/>



# Static Type Checking

---

- ▶ **Before** program is run
  - Type of all expressions are determined
  - Disallowed operations cause compile-time error
    - Cannot run the program
  
- ▶ Static types are **explicit** (*aka manifest*) or **inferred**
  - Manifest – specified in text (at variable declaration)
    - C, C++, Java, C#
  - Inferred – compiler determines type based on usage
    - OCaml, C# and Go (limited),

# Static Checking, and Type Safe?

---

- ▶ C, C++: **No**.
  - The languages' type systems do not prevent undefined behavior
    - Unsafe casts (int to pointer), out-of-bounds array accesses, dangling pointer dereferences, etc.
- ▶ Java, C#, OCaml: **Yes** (arguably).
  - The languages' type system aim to restrict programs to those that are defined
    - Caveats: Foreign function interfaces to type-unsafe C, bugs in the language design, bugs in the implementation, etc.

# Dynamic Type Checking

---

- ▶ **During** program execution
  - Type of expression determined when needed
    - Values maintain tag indicating type
  - Disallowed operations cause run-time exception
    - Type errors may be latent in code for a long time
- ▶ Dynamic types are not manifest (obviously)
  - Examples
    - Ruby, Python, Javascript, Lisp

# Dynamic Checking, and Type Safe?

---

- ▶ Ruby, Python: **Yes** (arguably).
  - All syntactically correct programs are well defined
    - The meaning of a program can be “throws an exception”
      - E.g., when accessing an array out of bounds, or when trying to call a nonexistent method
  - These languages are **uni-typed**
    - All objects have the same type (sometimes called *Dynamic*) and support all operations
      - For some objects, some operations will throw an exception, while for others they will return a result
    - Requires “type tags” to implement

# Static vs. Dynamic Type Checking

---

- ▶ Static type checking
  - More work for programmer (at first)
    - Catches more errors at compile time
  - Precludes some correct programs
    - May require a contorted rewrite
  - More efficient code (fewer run-time checks)
- ▶ Dynamic type checking
  - Less work for programmer (at first)
    - Delays some errors to run time
  - Allows more programs
    - Including ones that will fail
  - Less efficient code (more run-time checks)

# Type Systems are Not The Same

---

- ▶ OCaml's type system has types for
  - generics (polymorphism), objects, curried functions, ...
  - all unsupported by C
- ▶ Haskell's type system has types for
  - Type classes (qualified types), generalized abstract data types, higher-rank polymorphism, ...
  - All unsupported by Ocaml
- ▶ Added expressiveness ensures more errors prevented before execution
  - Less contorted programs
  - Easier to reason about program correctness

# Weak vs. Strong Typing

---

## ▶ Weak typing

- Allows one type to be treated as another or provides (many) **implicit casts**
- Example (int treated as bool)

➤ C

```
int i = 1 ;  
if (i) // checks for 0  
    printf(“%d”, i);
```

➤ Ruby

```
i = 1  
if i // checks for nil  
    puts i  
end;
```

- Example languages

➤ C, C++, Ruby, Perl, Javascript

# Weak vs. Strong Typing (cont.)

---

## ▶ Strong typing

- Prevents one type from being treated as another, implicitly
- Example (int not treated as bool)

➤ Java

```
int i = 1 ;  
if (i) // error, not bool  
    System.out.println(i);
```

➤ OCaml

```
let i = 1 in  
  if i then // error, not bool  
    print_int i
```

- Example languages

➤ Java (rare exceptions), OCaml



# Terms: Strong vs. Weak Typing

---

- ▶ These terms are not illuminating, or even agreed upon
  - “strong typing” is often confused with “type safety” or “static typing”
  - Supporting implicit casts, or not, is not particularly interesting as a language feature
    - And is confused with features like subtyping
- ▶ Other terms we’ve discussed are more well understood