Topics Covered Thus Far

- Programming languages
  - Ruby
  - Ocaml
  - Lambda calculus
- Syntax specification
  - Regular expressions
  - Context free grammars

Language Features Covered Thus Far

- Ruby
  - Implicit declarations
  - Dynamic typing
- Ocaml
  - Functional programming
  - Type inference
  - Higher-order functions
  - Static (lexical) scoping
  - Parametric polymorphism
  - Modules

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
  - Java: a == b, a.equals(b)
  - C: a == b, *a == *b
  - Ruby: a.equal?(b), a == b
  - OCaml: a == b, a = b
- 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

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 \( x \);
  - int \( f \) (int) \{ ... \}
  - class \( C \) \{ ... \}
  - let \( x = e_1 \) in \( e_2 \)

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
  \( x + y \);

- 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

  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

```c
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`

```ocaml
let x = 3 in
let y = x + 3 in...

let x = 3 + x in...
```

Order of Bindings – C

- All declarations are in scope from the declaration onward

```c
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

```java
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; ... /* 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

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

---

Shadowing and Implicit/Explicit Decls

- OCaml
  ```
  let x = ref 5;;
  let f = fun y -> let \x = ref (y + 5) in !x;;
  ```

  - declaration shadows outer x

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

- OCaml
  ```
  open to add names into current scope
  ```

- Java
  ```java
  import to add names into current scope
  ```

- C++
  ```
  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.

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

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.

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

i is bound in scope 2
j is free in scope 2
j is bound in scope 1

Static Scoping and Nested Functions

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

```c
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.

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

```c
let addN (n, l) =
    let map (add, l) =
        match l with
        [ ] -> [ ]
        |(h::t) -> (add h)::(map (add, t))
    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
    - $2 \times 3 = 6$
    - “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 ideals 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
  - In effect, languages have a null type system
    - All syntactically valid programs are well typed
  - Another POV: 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 power 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
      ```c
      int i = 1;
      if (i)
        printf("%d", i);
      ```
    - Ruby
      ```ruby
      i = 1
      if i
        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**
      ```java
      int i = 1;
      if (i) // error, not bool
        System.out.println(i);
      ```
    - **OCaml**
      ```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