Topics Covered Thus Far

- Programming languages
  - Ruby
  - OCaml
- Syntax specification
  - Regular expressions
  - Context free grammars
- Implementation
  - Finite automata (scanners)
  - Recursive descent parsers
Language Features Covered Thus Far

- **Ruby**
  - Implicit declarations: `{ x = 1 }
  - Dynamic typing: `{ x = 1 ; x = “foo” }

- **OCaml**
  - Functional programming: `add 1 (add 2 3)`
  - Type inference: `let x = x+1 ( x : int )`
  - Higher-order functions: `let rec x = fun y -> x y`
  - Static (lexical) scoping: `let x = let x = …`
  - Parametric polymorphism: `let x y = y ( ‘a -> ‘a )`
  - Modules: `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

<table>
<thead>
<tr>
<th></th>
<th>Physical Equality</th>
<th>Structural Equality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td>a == b</td>
<td>a.equals(b)</td>
</tr>
<tr>
<td>C</td>
<td>a == b</td>
<td>*a == *b</td>
</tr>
<tr>
<td>Ruby</td>
<td>a.equal?(b)</td>
<td>a == b</td>
</tr>
<tr>
<td>OCaml</td>
<td>a == b</td>
<td>a = b</td>
</tr>
</tbody>
</table>

- 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
Explicit vs. Implicit Declarations

- Explicit declarations
  - Variables must be declared before used
  - Examples
    - C, C++, Java, OCaml

- Implicit declarations
  - Variables do not need to be declared
  - Examples
    - Ruby

Type System Overview

- Typed vs. untyped
- Static vs. dynamic
- Type safety
  - Weak (not type safe) vs. strong (type safe)
Type vs. Untyped Languages

- **Typed language**
  - Operations are only valid for specified types
    - 2 * 3 = 6
    - “foo” * “bar” = undefined
  - Helps catch program errors
    - Either at compile or run time

- **Untyped language**
  - All operations are valid for all values
  - Treat all values as sequences of 0’s and 1’s
  - Example
    - Assembly languages, FORTH

Static vs. Dynamic Types

- **Static types**
  - Before program is run
    - Type of all expressions are determined
      - Usually by compiler
      - Disallowed operations cause compile-time error

- Static types may be manifest or inferred
  - Manifest – specified in text (at variable declaration)
    - C, C++, Java, C#
  - Inferred – compiler determines type based on usage
    - ML, OCaml
Static vs. Dynamic Types (cont.)

- **Dynamic types**
  - While program is running
    - Type of all expressions determined
    - Values maintain tag indicating type
    - Disallowed operations cause run-time exception

- Dynamic types are not manifest (obviously)
  - Examples
    - Ruby, Python, Javascript, Lisp

Type Safety

- Determined by extent programming language allows type errors
- Language should only allow operations on values that are permitted by their type
  - Non-type safe code example
    - `printf("%d", 3.12)` // Allows float to be printed as int

Definitions
- Type safe language → strong type system
- Non-type safe language → weak type system
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) // checks for 0
        printf("%d", i);
    ```
  - Ruby
    ```ruby
    i = 1
    if i // checks for nil
        puts i
    end;
    ```
- Example languages
  - C, C++, Ruby, Perl, Javascript

Strong typing

- Prevents one type from being treated as another (also known as type safe)
- 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
Weak/Strong vs. Static/Dynamic Types

- How do these properties interact?
  - Weak/strong & static/dynamic are orthogonal
  - Some literature confuse strong & static type
- Strong / static types
  - More work for programmer
  - Catches more errors at compile time
- Weak / dynamic types
  - Less work for programmer
  - More errors occur at run time

Names & Binding Overview

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

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... (* x is in scope here *)

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

```ocaml
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

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

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

```c
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.

```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
        ...
    }
}
```
**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

**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

```latex
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.

```
{ /* 1 */
  int j;

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

- j is bound in scope 1.
- j is free in scope 2.
- i 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)

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

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

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

```ocaml
define addN (n, l) =
  define add x = n + x in
  map (add, l)
define map (f, n) =
  match n with
  | [] -> []
  | (h::t) -> (h :: (map (f, 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 Scope

**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)