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
  - **Physical Equality**
    - Java: `a == b`
    - C: `a == b`
  - **Structural Equality**
    - Ruby: `a.equals(b)`
    - OCaml: `a == b`
    - Explanation: these differences a major goal for 330
    - Will be covering different features in upcoming lectures
### Programming Language Features

<table>
<thead>
<tr>
<th>Paradigm</th>
<th>Declarations</th>
<th>Type System</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Functional</td>
<td>• Explicit</td>
<td>• Typed vs. untyped</td>
</tr>
<tr>
<td>• Imperative</td>
<td>• Implicit</td>
<td>• Static vs. dynamic</td>
</tr>
<tr>
<td>• Object oriented</td>
<td></td>
<td>• Weak vs. strong (type safe)</td>
</tr>
<tr>
<td>• Multi-paradigm</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Higher-order functions</th>
<th>Declarations</th>
<th>Polymorphism</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Closures</td>
<td></td>
<td>• Names &amp; binding</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Namespaces</td>
</tr>
</tbody>
</table>

### Programming Language Features (cont.)

<table>
<thead>
<tr>
<th>Names &amp; binding</th>
<th>Polymorphism</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Static (lexical) scopes</td>
<td>• Ad-hoc</td>
</tr>
<tr>
<td>• Dynamic scopes</td>
<td>• Subtype</td>
</tr>
</tbody>
</table>

| Parameter passing      |                    |
|• Call by value        |                    |
|• Call by reference    |                    |
|• Call by name         |                    |
|• Eager vs. lazy evaluation |                |

| Declarations       |                    |
|• Explicit         |                    |
|• Implicit         |                    |

| Type system               |                    |
|• Typed vs. untyped       |                    |
|• Static vs. dynamic      |                    |
|• Weak vs. strong (type safe) |                |

<table>
<thead>
<tr>
<th>Type vs. Untyped Languages</th>
<th>Static vs. Dynamic Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Typed language</td>
<td>• Static types</td>
</tr>
<tr>
<td>• Operations are only valid for specified types</td>
<td>• Before program is run</td>
</tr>
<tr>
<td>• 2 * 3 = 6</td>
<td>• Type of all expressions are determined</td>
</tr>
<tr>
<td>• &quot;foo&quot; * &quot;bar&quot; = undefined</td>
<td>• Usually by compiler</td>
</tr>
<tr>
<td>• Helps catch program errors</td>
<td>• Disallowed operations cause compile-time error</td>
</tr>
</tbody>
</table>

| Untyped language       |• Static types may be manifest or inferred |
|• All operations are valid for all values |• Manifest – specified in text (at variable declaration) |
|• Treat all values as sequences of 0’s and 1’s |• C, C++, Java, C# |
|• Example               |• Inferred – compiler determines type based on usage |
|• Assembly languages, FORTH |• ML, OCaml |

### Explicit vs. Implicit Declarations

<table>
<thead>
<tr>
<th>Explicit declarations</th>
<th>Implicit declarations</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Variables must be declared before used</td>
<td>• Variables do not need to be declared</td>
</tr>
<tr>
<td>• Examples</td>
<td>• Examples</td>
</tr>
<tr>
<td>• C, C++, Java, OCaml</td>
<td>• Ruby</td>
</tr>
</tbody>
</table>

### Type System Overview

<table>
<thead>
<tr>
<th>Typed vs. untyped</th>
<th>Static vs. dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Helps catch program errors</td>
<td>Weak vs. strong (type safe)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Static vs. Dynamic Types</th>
<th>Static types</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Before program is run</td>
<td>• Before program is run</td>
</tr>
<tr>
<td>• Type of all expressions are determined</td>
<td>• Type of all expressions are determined</td>
</tr>
<tr>
<td>• Usually by compiler</td>
<td>• Usually by compiler</td>
</tr>
<tr>
<td>• Disallowed operations cause compile-time error</td>
<td>• Disallowed operations cause compile-time error</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Static vs. Dynamic Types</th>
<th>Static types</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Manifest – specified in text (at variable declaration)</td>
<td>• Manifest – specified in text (at variable declaration)</td>
</tr>
<tr>
<td>• C, C++, Java, C#</td>
<td>• C, C++, Java, C#</td>
</tr>
<tr>
<td>• Inferred – compiler determines type based on usage</td>
<td>• Inferred – compiler determines type based on usage</td>
</tr>
<tr>
<td>• ML, OCaml</td>
<td>• ML, OCaml</td>
</tr>
</tbody>
</table>
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

Weak vs. Strong Typing

- Weak typing
  - Allows one type to be treated as another
  - ...or provides (many) implicit casts
    - C
      - int i = 1;
      - if (i) // checks for 0
        - print("%d", i);
    - Ruby
      - i = 1
      - if i // checks for nil
        - puts i
        - end;
  - Examples
    - C, C++, Ruby, Perl, Javascript

Weak vs. Strong Typing (cont.)

- Strong typing
  - Prevents one type to be treated as another
  - Also known as type-safe
    - Java
      - int i = 1;
      - if (i) // error, not bool
        - System.out.println();
    - OCaml
      - let i = 1 in
        - if i then // error, not bool
          - print_int i
      - end;
  - Examples
    - Java, 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
- Funargs

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 */
Name Restrictions

- Languages often have various restrictions on names to make lexing 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

Ordering of Bindings

- Languages make various choices for when declarations of things are in scope

Example

```c
void w(int i) {
    ...
}
void x(float j) {
    ...
}
void y(float i) {
    ...
}
void z(void) {
    int j;
    char *i;
    ...
}
```

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 rec length = function
    [] -> 0
  | (h::t) -> 1 + (length t)
```

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

let rec length = function
    [] -> 0
  | (h::t) -> 1 + (length t)
```

```java
let rec length = function
    [] -> 0
  | (h::t) -> 1 + (length t)
```
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 */
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

Mangled Names

- What happens when these names need to be seen by other languages?
  - What if a C program wants to call a C++ method?
    - C doesn’t know about C++’s naming conventions
  - For multilingual communication, names are often mangled into some flat form.
    - E.g., class C { int f(int *x, int y) { ... } } becomes symbol __ZN1C3fEPii in g++
    - E.g., native valueOf(int) in java.lang.String corresponds to the C function
      _Java_java_lang_String_valueOf__I_

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.
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.

```plaintext
{ /* 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 2

Static Scoping and Nested Functions

- To allow arbitrary nested functions with higher-order functions and static scoping, we needed closures.

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

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

Functional Arguments (Funargs)

- **Funarg problem**
  - Difficult to implement functions as first-class objects in stack-based programming languages.

- **Downwards funargs**
  - Passing function as parameter to another function call.
  - Can be implemented efficiently:
    - Since stack frame will still be on stack when funarg is used.
    - Techniques such as access links / displays (see CMSC 430).

- **Upwards funargs**
  - Returning a function from a function call.
  - Implementation requires closures (stored on heap).

Example

```plaintext
let f x = let g y = x + y in app g 3
```

- When *g* is called, *x* is still on the stack.

Dynamic Scope

- In a language with **dynamic scoping**, a name refers to its closest binding at runtime.
  - LISP was the common example.

```plaintext
(define f (lambda () a))
(define a 3) ; bind a to 3
(f) ; calls f and returns 3
(define a 4) ; defines a no-argument function which returns a
(define f (lambda () a)) ; defines a no-argument function which returns a
(f) ; calls f and returns 4
```

Example

```plaintext
let f z = let g y = x + y in app g 3
```

- When *g* is called, *x* is still on the stack.
Nested Dynamic Scopes

- Full dynamic scopes can be nested
  - Static scope relates to the program text
  - Dynamic scope relates to program execution trace

```perl
# Perl (the keyword local introduces dynamic scope)
$l = "global";
sub A {
  local $l = "local";
  B();
}
sub B { print "$l
"; }  # local
global
B(); A(); B();  # global
```

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)