CMSC 330: Organization of Programming Languages

Type Systems,
Names & Binding

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	Structural Equality
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 Paradigms

- Imperative programming
  - Assignment statements heavily used
- Functional programming
  - Function calls, higher-order functions
- Object-oriented

- You can do any of these in most languages
  - But, some languages may make this easier/harder

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 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, Scheme
- Most static type systems have some dynamic aspects
  - Null pointers, array bounds, downcasts, etc

Weak vs. Strong Typing

- Weak typing
  - Allows one type to be treated as another
  - …or provides (many) implicit casts
    - C

```
int i = 0xdeadbeef ;
int *p = (int *) i;
*p = 42;    /* write to absolute address */
```

  - Main examples: C, C++
    - Helps make certain kinds of low-level systems programming easier
    - But, pervades language, makes it easy to make mistakes even in code that doesn't need this ability

Weak vs. Strong Typing (cont.)

- Strong typing
  - Prevents one type being treated as another
    - Either statically, dynamically, or both
  - Also known as type-safe
  - Examples
    - Java, OCaml, Ruby, Perl, Javascript, etc
- Consensus: Strong typing is good
  - Most languages have an "escape hatch" for those instances where you need weak typing
    - In OCaml, Obj.magic : 'a -> 'b

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

Polymorphism

- We've seen three kinds of *polymorphism*
  - A feature of type systems in which *one* value can have *many* different types
- Ad-hoc polymorphism (overloading)
  - Ex: + in C, method overloading in Java
- Subtype polymorphism
  - Ex: subclassing in Java
- Parametric polymorphism
  - Ex: OCaml 'a, Java generics

More Language Features Coming Up

- Names and binding
  - Namespaces, scoping
- Parameter passing mechanisms
  - Call-by-{value, reference, name}
- Parallelism support
  - Thread creation
  - Shared-memory concurrency
  - Message passing

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

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 */
```

```
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

```
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

```
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;
  }
}
```

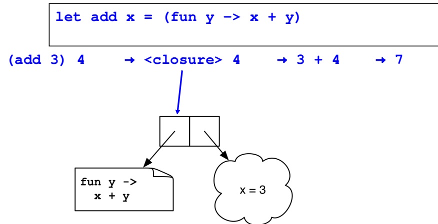
j is bound in scope 1

i is bound in scope 2

j is free in scope 2

Static Scoping and Nested Functions

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

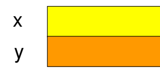


Functional Arguments (Funargs)

- 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

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



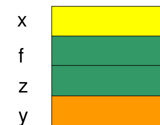
- When g is called, x is still on the stack

when is g called?

Answer: when f is called with parameter x

Example

```
let app f z = f z

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



- Function g is passed as parameter to app
  - I.e., g is a downward funarg
- When g is called, x is still on the stack
  - Closure is not needed

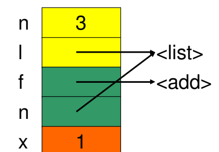
Dynamic Scope

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

```
Scheme (top-level scope only is dynamic)

(define f (lambda () a))
; defines a no-argument function which returns a

(define a 3)    ; bind a to 3
(f)             ; calls f and returns 3
(define a 4)
(f)             ; calls f and returns 4
```


Previous OCaml Call Stack Example

```
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])
```



- How to determine value of n in add ?
  - Dynamic scope: reads it off the stack (n = <list>)
  - Static scope: lexical binding (n = param n to addN)

Nested Dynamic Scopes

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

```
Perl (the keyword local introduces dynamic scope)

$l = "global";

sub A {
  local $l = "local";
  B();
}

sub B { print "$l\n"; }

B(); A(); B();
```

```
global
local
global
```


Static vs. Dynamic Scope

Static scoping	Dynamic scoping
• Local understanding of function behavior	• Can be hard to understand behavior of functions
• Know at compile-time what each name refers to	• Requires finding name bindings at runtime
• A little more work to implement (keep a link to the lexical nesting scope in stack frame)	• Easier to implement (keep a global table of stacks of variable/value bindings)