CMSC 330: Organization of Programming Languages

Names & Binding

Type Systems, 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
  - Dynamic typing
  \[ x = 1 \]
  \[ x = 1 : x = \text{"foo"} \]
- OCaml
  - Functional programming
    \[ \text{add} 1 \ (\text{add} 2 \ 3) \]
  - Type inference
    \[ \text{let} \ x = x + 1 \ (x : \text{int}) \]
  - Higher-order functions
    \[ \text{let rec} \ x = \text{fun} \ y \rightarrow x \ y \]
  - Static (lexical) scoping
    \[ \text{let} \ x = \text{let} \ x = \ldots \]
  - Parametric polymorphism
    \[ \text{let} \ x \ y = y \ (\ 'a \rightarrow 'a) \]
  - Modules
    \[ \text{module foo struct} \ldots \text{end} \]

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
      \[ \text{if} (x == 1) \ \{ \ldots \} \ \} \else \{ \ldots \} \]
    > Ruby
      \[ \text{if} \ x == 1 \ \{ \ldots \} \ \text{end} \]
    > OCaml
      \[ \text{if} \ (x == 1) \ \{ \ldots \} \ \text{else} \ { \ldots } \]
  - 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
    - \(2 \times 3 = 6\)
    - \("bo" + "bar"\) = 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 languages, 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

```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

Ordering 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 + 3 in ...
    (* x is not in scope *)

let rec length = function
    [] -> 0
    | (h::t) -> 1 + (length t)  (* ok, length in scope *)
  in ...
```

Ordering 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 old style 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() {
        ...
    }
}
```

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, 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

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

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

```java
C
int i;
void f(float i) {
    char *i = NULL;
    ...
}
```

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 __smic33fpi in C++
    - E.g., native valueOf(int) in java.lang.String corresponds to the C function
      `Java.java_lang_String_valueOf__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

```java
j is bound in scope 1
{
    int j;
    {
        float i;
        j = (int) i;
    }
}
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

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

\[
\text{let add x = (fun y -> x + y)}
\]

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

\[
\text{let add x = (fun y -> x + y)}
\]

Dynamic Scope

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

Example

\[
\text{let add x = (fun y -> x + y)}
\]

Previous OCaml Call Stack Example

\[
\text{let map (f, n) = match n with}
\]

- How to determine value of \( n \) in `add`?
  - Dynamic scope: reads it off the stack \( n = \langle\text{list}\rangle\)
  - Static scope: lexical binding \( n = \text{param n to addN} \)
**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)
$1 = "global";
sub A {
    local $1 = "local";
    B();
}  local global
sub B { print "$1\n"; }  global
B(); A(); B();
```

**Static vs. Dynamic Scope**

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