CMSC 330: Organization of Programming Languages

Type Systems, More on Scoping, and Parameter Passing, con't.

Static vs. Dynamic Types

- With static typing the types of all expressions are determined *before* a program is run (usually by the 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.)

- With dynamic typing the types of all expressions are determined while a program is running
  - Disallowed operations cause run-time exception
  - Values maintain a tag indicating their type
- 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 boolean)
    - C
      ```
      int i = 1;
      if (i)   // checks for 0
         printf("%d", i);
      ```
    - Ruby
      ```
      i = 1
      if i   // checks for nil
         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 (also known as type-safe)
  - Example (int not treated as boolean)
    - Java
      ```
      int i = 1;
      if (i)   // error, not boolean
         System.out.println(i);
      ```
    - OCaml
      ```
      let i = 1 in
         if i then   // error, not boolean
            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 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 i) { ... }` `/* 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, con't.

- 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
  - these are different j's

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 + 4 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 */
```


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 adds names into current scope
  - Java: Packages
    - java.lang.String, java.awt.Point, etc.
    - import adds names into current scope
  - C++: Namespaces
    - namespace f { class g { ... } }, f::g b, etc.
    - using namespace adds 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 Java, C, C++, 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

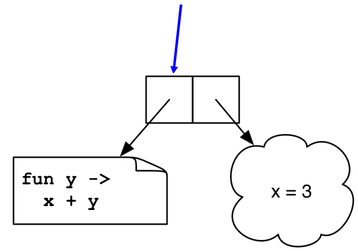
j is free in scope 2

Static Scoping and Nested Functions

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

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

`(add 3) 4 → <closure> 4 → 3 + 4 → 7`

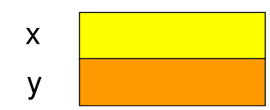


Nested Functions, con't.

- We need closures for *upward funargs*
  – Functions that are returned by other functions
- If we only have *downward funargs*, then we don't need full closures
  – These are functions that are only passed inward (as parameters)
  – So when they're called, any nonlocal variables they access from outer scopes are still around

Example

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

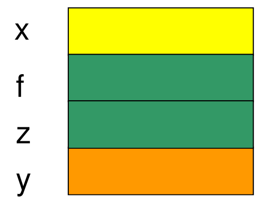


- When g is called, x is still on the stack

Example

```
let app f z = f z

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



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

Downward Funargs

- It turns out that if we only pass functions downward, there are cheaper implementation strategies for static scoping than closures
- They're called *static links* and *displays*, and they're used by
  - Pascal and Algol-family languages
  - gcc nested functions
- We won't go into details

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

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


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();
```

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 bit trickier to implement	– Easier to implement (just keep a global table of stacks of variable/value bindings)