EEECS 490 – Lecture 2

Names and Environments

Announcements

- Homework 1 due 9/15 at 8pm
- Entry survey due at 8pm tonight

Agenda

- Environments and Name Lookup
- Static and Dynamic Scope
- Point of Declaration

Names

- Fundamental form of abstraction
  - Allow entities of arbitrary complexity to be referenced by a single name
- A name is distinct from the entity it names
  - The same name can refer to different entities in different contexts or at different times
  - An entity may have multiple names that refer to it
- Languages define built-in names and also provide a mechanism for users to define their own names

Scope and Frames

- In order to properly implement abstraction, names in general must have a restricted *scope*
  - Avoid conflict between internal names defined in different contexts

- The mapping of names to entities is tracked at runtime in individual *frames* or *activation* records for each region of scope
  - A name is *bound* to an entity in a frame or scope

Frames and Environments

- A piece of code may be located in multiple regions of scope

```
int x = 0;
void foo(int y) {
  cout << (x + y) << endl;
}
```

- It therefore has access to multiple frames that bind names to entities
- Frames are generally ordered by how restricted their corresponding scope regions are
- The set of frames available to a piece of code is called its *environment*

Name Lookup

- Names have a well-defined procedure to look them up in an environment with multiple frames:
  1. Start lookup in the innermost frame
  2. If the name is bound in the current frame, then use that binding
  3. If the name is not bound in the current frame, proceed to the next frame and go to step 2

- Example:

```
int x = 0;
void foo(int x, int y) {
  cout << (x + y);
}
```

Overloading

- A name is *overloaded* if it has multiple bindings in the same frame
- A language that allows overloading must define how overloads are resolved

```
void foo(int x);
int foo(const string &s);
foo(3);
foo("hello");
```

- Some languages, such as Java, use similar rules to disambiguate names in separate frames

```
public static void main(String[] args) {
  int main = 3;
  main(null);  // recursive call
}
```

Blocks

- A *block* is a compound statement that groups together other statements
  ```
  {  statement1; statement2; ...; statmentN; }
  ```

- A block usually defines a region of scope and therefore has its own frame

- Blocks can be associated with a function or be an inline block nested in another block
  ```
  int main(int argc, char **argv) {
    if (argc < 3) {
      int status_code = 1;
      print_usage();
      exit(status_code);
    }
  }
  ```

Suites in Python

- Python does not have inline blocks

- Compound statements can be composed of a *header* followed by a *suite* of statements

- In general, a suite does not have its own frame

```
def foo(x):
  if x < 0:
    negative = True
  else:
    negative = False
  print(negative)
```

Blocks in Scheme

- The *let* forms in Scheme introduce a new frame

```
(let ((x 3) (y 4))
    (display (+ x y))
    (display (- x y)))
```

- This is commonly implemented by translating into a function definition and call:

```
((lambda (x y)
     (display (+ x y))
     (display (- x y)))
 3 4)
```

Anonymous function; more on this in a few weeks

Environments and Nested Blocks

- Nested blocks result in nested frames in the environment

- *Visibility rules* correspond to the lookup procedure

```
{
  int x = 0;
  int y = 1;
  {
    int x = 2;
    int z = 3;
    cout << (x + y + z);
  }
}
```

Binding *hidden* by declaration of x in inner block

Binding *visible* in inner block

Inner x *shadows* outer x

Break Time!

- We'll start again in five minutes.

Functions and Environments

- Functions differ from inline blocks in that the context in which they are defined differs from the context in which they execute

```
int x = 0;

void foo() {
  print(x);
}

void bar() {
  int x = 1;
  foo();
}
```

Which x is printed? Either is a valid choice

Kinds of Environments

- The environment in which a function executes is often divided into three components
  - The *local* environment is the part that is internal to the function
  - The *global* environment is the part defined at the top-level of a program, at global or module scope
  - The *non-local* environment consists of the bindings that are visible to a function but not part of the local or global environment
- The two possibilities for which x is printed correspond to different choices about what constitutes the non-local environment

Static Scope

- In *static* or *lexical* scope, the non-local environment of a function is the environment in which the function is defined
  - Can be determined directly from the program's syntactic structure

```
int x = 0;

void foo() {
  print(x);
}

void bar() {
  int x = 1;
  foo();
}
```

Prints `0`

Not in the environment of `foo()`

Nested Function Definitions

- Nested function definitions result in more complex environments in static scope

```
x = 0

def foo():
    x = 2
    def baz():
        print(x)
    return baz

def bar():
    x = 1
    foo()()  # call baz()

bar()
```

In the environment of baz()

Prints 2

Not in the environment of baz()

Dynamic Scope

- In *dynamic* scope, the non-local environment of a function is the environment in which it is called

```
int x = 0, y = 1;
void foo() {
  print(x);
  print(y);
}
void bar() {
  int x = 2;
  foo();
}
int main() {
  int y = 3;
  bar();
  return 0;
}
```

Prints 2

Prints 3

In the environment of `foo()`

In the environment of `bar()` and `foo()`

Use Before Declaration

- An exact correspondence between blocks, frames, and scope allows code such as the following:

```
int foo() {
  print(x);
  int x = 3;
}
```

- This should be invalid, since **x** is used before it is initialized

Point of Declaration

- In some languages, including the C family, the scope of a name extends from its *point of declaration* to the end of the enclosing block

```
int x = 2;

int foo() {
  print(x);
  int x = 3;
}
```

Prints 2

Scope of inner x starts here

Scope of inner x ends here

Assignments in Python

- Python assumes that an assignment to a variable is intended to target a local variable

- Furthermore, the scope of a local variable starts at the beginning of a function

- Using a variable before it is initialized is an error

```
x = 2

def foo():
    print(x)
    x = 3
```

Scope of local x starts here

Error: use before initialization

Defines local variable x

`global` and `nonlocal` in Python

- A programmer can specify that a name is meant to refer to a global or non-local variable using the **`global`** and **`nonlocal`** statements

```
x = 2

def foo():
    global x
    print(x)
    x = 3

foo()
print(x)
```

Specifies that x refers to the global variable

Prints 2

Assigns 3 to the global variable x

Prints 3

Mutually Recursive Entities

- The C-style point of declaration rules are insufficient for defining mutually recursive entities

```
int foo(int x) {
  return bar(x + 1);
}

int bar(int x) {
  return foo(x - 1);
}
```

Error: use before declaration

Scope of bar starts here

Incomplete Declarations

- C and C++ allow *incomplete declarations* that allow an entity to be declared without being defined

```
int foo(int x) {
  int bar(int);
  return bar(x + 1);
}

int bar(int x) {
  return foo(x - 1);
}
```

Incomplete declaration

Scope of incomplete declaration ends here