Bindings and Scope

Programming Languages

William Killian

Millersville University

Lecture Outline

- Bindings
  - Type Inferencing
  - Type Binding Examples
- Lifetime
  - Definition
  - Examples
- Scope
  - Constructs that create scope
  - Static Scope
  - Dynamic Scope
  - Referencing Environments

Bindings

Type Inferencing

Some languages can infer or deduce the type

```
auto v = 4

var w = true

val x = 0

let y = 3.1415

const z = "hello"
```

Dynamic Type Binding

The type of a variable can change via assignment

```
lst = [1, 2, 3]  # does not matter
                 # what lst was before
```

- Advantages:
  - Flexibility
  - Ease of use
- Disadvantages:
  - Costly (must do type checking all the time)
  - Type error deduction can be difficult/impossible

Case Study: Python

```
x = 4

x = 4 + "”

x = [x]

x = set(x)

x = {"x" : x}
```

Case Study: C++

```
auto x = 4;
auto x = static_cast<float>(x);

auto x = 4.0f;
x = 3;

// another example?
```

Case Study: JavaScript

```
var x = 4
x = "hello"

let y = 4
y = "hello"

const z = 4
z = 5
```

Lifetime

Lifetime

- The ***lifetime*** of a variable is the time during which it is bound to a particular memory cell
- Lifetimes can be "created" or "destroyed"
  - **Create:** allocation – getting cell(s) of memory
  - **Destroy:** deallocation – putting cell(s) back
- Four different categories of lifetime
  - Static
  - Stack-dynamic
  - Explicit heap-dynamic
  - Implicit heap-dynamic

Static

`static int x = 4`

- Storage bound *prior* to program execution
- Binding *cannot change*

- *Advantages*
  - Direct addressing (immediately available)
  - History sensitive
- *Disadvantages*
  - Inflexible
  - Only one instance permitted

Stack Dynamic

`int x = 4`

- Storage binding created when declaration is “executed” on the computer.
- Only the address can change

- *Advantages*
  - Enables recursion
  - Conserves storage (local)
- *Disadvantages*
  - Allocation/deallocation overhead
  - Not history sensitive
  - Indirect addressing (extra instruction)

Explicit Heap Dynamic `new`/`delete`

- Allocated and deallocated by explicit directives
- Takes effect during execution
- Referenced only through pointers/references

- *Advantages*
  - Dynamic storage management
- *Disadvantages*
  - Inefficient
  - Unreliable

Implicit Heap Dynamic

- Allocated and deallocated by assignment statement
- All arrays/objects in Javascript, Perl, PHP

- *Advantages*
  - Flexibility (generic code)
- *Disadvantages*
  - Inefficient – all attributes are dynamic
  - Loss of error detection

Scope

Scope

- Range of statements over which a variable is visible
- The **scope rules** of a language determine how references to names are associated with variables
- To connect a name reference to a variable, you (or the compiler) must find the declaration
- Static Scope:
  - Based on the **structure** of the program
- Dynamic Scope:
  - Based on the **execution** of the program

Constructs that Create Scope

- Blocks
  - Present in most imperative languages
  - Any statement can be a block of statements
  - Automatically creates a new scope
  - **Shadowing** permitted in C/C++ but disallowed in Java/C#
- Let
  - Present in most functional languages
  - Composed of two parts:
    - Binding names to values
    - Using the names declared in the first part

Scope Categories

- *Local variables*
  - **Declared** within a particular unit of the program
- *Nonlocal variables*
  - **Visible** within a particular unit of the program
  - **Not declared** in that unit of the program
- *Global variables*
  - Visible within **all units** of a program
  - Declared in the outermost scope of the program

Static Scope Rules

Search process:

- Search declarations, first locally
- Then in increasingly larger enclosing scopes, until one is found for the given name

Terms:

- **Static ancestors**: the enclosing static scopes
- **Static parent**: the nearest static ancestor

Variables can be hidden from a unit by having a "closer" variable with the same name (**shadowing**)

Static Scope Example

```
int x = 3;
int main() {
  std::cout << x << '\n';
  int x = 4;
  {
    int x = 5;
    {
      std::cout << x << '\n';
      int x = 6;
      std::cout << x << '\n';
    }
    std::cout << x << '\n';
  }
  std::cout << x << '\n';
}
```

Dynamic Scope Rules

Search process:

- Based on calling sequences of functions
- Search back through the chain of function calls that forced execution to this point

Notes:

- All visible names must be visible to the function called
- "Temporal" in nature – dependent upon what was most recently accessed

Variables **are hidden** from a unit if one of the same name exists in a closer dynamic scope

Dynamic Scope Example

```
function main() {
  var o = "enter a number: ";
  print();
  var x = input();
  test();
}

function test() {
  if (x % 2 == 0) {
    even();
  } else {
    odd();
  }
}
```

```
function even() {
  o = "even";
  print()
}

function odd() {
  o = "odd";
  print()
}

function print() {
  output(o);
}
```

Static vs. Dynamic Scope

Static

- *Advantages*
  - Possible for compiler to detect errors
  - Type checking guaranteed
- *Disadvantages*
  - Information more difficult to pass

Dynamic

- *Advantages*
  - Convenient
- *Disadvantages*
  - Impossible(?) to analyze without running
  - All variables visible to *subprograms*
  - Poor readability

Static vs. Dynamic Scoping

```
function main() {
  var x = 1;
  function funA() {
    var y = x;
    print(y);
  }
  function funB() {
    var x = 7;
    funA();
  }
  funB();
}
```

What does this program print out with

- *Static Scoping Rules?*
- *Dynamic Scoping Rules?*

Referencing Environments

- All variables/names visible at a current program unit is known as a **referencing environment**
- We have seen how the referencing environment can vary for static and dynamic scoping rules
- Static-Scoped Languages:
  - Local variables plus all the visible variables in **all enclosing scopes**
- Dynamic-Scoped Languages:
  - Local variables plus all visible variables in **all active subprograms**
  - Active subprogram: when running but not terminated