Lecture 6

Names, Bindings, and Scopes

Dr. Kebande

Lecture 6 Topics

- Introduction
- Names
- Variables
- The Concept of Binding
- Scope
- Scope and Lifetime
- Referencing Environments
- Named Constants

Introduction

- Imperative languages are abstractions of von Neumann architecture
    - Memory
    - Processor
- Variables are characterized by attributes
- The design of a type requires several issues to be considered

Names

- Design issues for names:
  - Are names case sensitive?
  - Are special words reserved words or keywords?

Names (continued)

- Length
  - If too short, they cannot be connotative
  - Language examples:
    - FORTRAN 95
      - A maximum of 31
    - C99
      - No limit but only the first 63 are significant
      - External names are limited to a maximum of 31
    - C#, Ada, and Java
      - No limit, and all are significant
    - C++
      - No limit, but implementers often impose one

Names (continued)

- Special characters
  - PHP
    - All variable names must begin with dollar signs
  - Perl
    - All variable names begin with special characters, which specify the variable's type
  - Ruby
    - Instance variable names begin with `@`
    - Class variable names begin with `@@`

Names (continued)

- Case sensitivity
  - Disadvantages:
    - Readability
      - Names in the C-based languages are case sensitive
        - Names that look alike are actually different
      - Names in other languages are not
    - Writability
      - In C++, Java, and C# predefined names are mixed case
      - For example, `IndexOutOfBoundsException`
      - Programmer has to remember the case for these names

Names (continued)

- Special words
  - Aid to readability by naming actions
  - Also separate parts of statements & programs
  - <u>Keyword</u> is special only in certain contexts
    - In Fortran
      - `Real VarName`

        `Real` is a keyword, denoting a data type
      - `Real = 3.4`

        `Real` is a variable
  - <u>Reserved word</u> cannot be a user-defined name
    - Potential problem
      - If there are too many, many collisions occur
      - COBOL has 300 reserved words!

Variables

- A variable
  - An abstraction of a memory cell
  - Can be characterized by 6 attributes:
    - Name
    - Address
    - Value
    - Type
    - Lifetime
    - Scope

Variables Attributes

- Name
  - Not all variables have them (more on this later)
- Address
  - The memory address with which a variable is associated
  - May have different addresses at different execution times
  - If two variable names can be used to access the same memory location, they are called aliases
    - Created via pointers, references, C and C++ unions
    - Harmful to readability because program readers must remember all of them

Variables Attributes (continued)

- Type
  - Determines
    - Range of values of variables
    - Set of operations that are defined for values of that type
    - In the case of floating point, type also determines the precision
    - How a value is stored in memory
- Value
  - Contents of the location with which variable is associated
  - Represents an abstract memory cell
    - The <u>physical cell</u> or <u>collection of cells</u> associated with a variable
    - Example: a float occupies 4 bytes, but 1 abstract memory cell
- The l-value of a variable is its address
- The r-value of a variable is its value

The Concept of Binding

- A <u>binding</u>: an association
  - Between an attribute and an entity
    - Example: between a variable and its type or value
  - Or between an operation and a symbol
    - Example: between a multiplication operation and *
- <u>Binding time</u>
  - Time that binding takes place
  - Several are possible

Possible Binding Times

- Language design time
  - Bind operator symbols to operations

- Language implementation time
  - Bind floating point type to a representation
- Compile time
  - Bind a variable to a type in C or Java
- Load time
  - Bind a C/C++ `static` variable to memory cell
- Runtime
  - Bind non-static local variable to a memory cell

Static and Dynamic Binding

- Static binding
  - Occurs before run time and remains unchanged throughout program execution
- Dynamic binding
  - Occurs during execution or can change during execution of the program

Type Binding

- How is a type specified?
- When does the binding take place?
  - Static
  - Dynamic

Static Type Binding

- Explicit declaration
  - A program statement used for declaring the types of variables
- Implicit declaration
  - A default mechanism for specifying variable types
    - Involves the first appearance of a variable in the program
    - Naming conventions may determine type
  - Used in BASIC, Perl, Ruby, JavaScript, and PHP
  - Fortran has explicit and implicit declarations
  - Advantage: writability
    - A minor convenience
  - Disadvantage: reliability
    - Typographic errors can't be picked up by the compiler
    - Less trouble with Perl (why?)

Static Type Binding (continued)

- Some languages use <u>type inferencing</u>
  - Determine types of variables from context
  - For example:
    - C# optionally uses type inferencing
      - Variable can be declared with **`var`** and initial value
      - The initial value sets the variable type

```
var value = 12;
```

    - Also used in
      - Visual BASIC 9.0+
      - Haskell
      - ML
      - F#

Dynamic Type Binding

- JavaScript, Python, Ruby, PHP, C# (limited)
- Type specified by assignment statement
- For example, in JavaScript

```
list = [2, 4.33, 6, 8];
list = 17.3;
```

- Advantage
  - Flexibility (generic program units)
- Disadvantages
  - Cost (dynamic type checking & interpretation)
  - Type error detection by the compiler is difficult

Variable Attributes (continued)

- Storage bindings and Lifetime
  - Allocation
    - Getting a cell from a pool of available memory cells
  - Deallocation
    - Putting a memory cell back into the pool
  - Variable lifetime
    - The time during which a variable is bound to a particular memory cell

Categories of Variables by Lifetimes

- Static variables
  - Bound to memory cells before execution begins and stays bound to the same memory cell throughout execution
  - For example, C and C++ `static` variables
  - Advantages
    - Efficiency because of direct addressing
    - History-sensitive subprogram support
  - Disadvantages
    - Lack of flexibility (do not allow for recursion)
    - Storage cannot be shared amongst variables

Categories of Variables by Lifetimes

- Stack-dynamic variables
  - Storage bindings are created for variables when their declaration statements are elaborated
    - When the executable code associated with it is executed
  - Scalars (variables that hold one value at a time)
    - All attributes except storage & address are statically bound
    - Local variables in C subprograms and Java methods
  - Advantage
    - Allows recursion
    - Conserves storage
  - Disadvantages
    - Overhead of allocation and deallocation
    - Subprograms cannot be history sensitive
    - Inefficient references (indirect addressing)

Categories of Variables by Lifetimes

- Explicit heap-dynamic variables
  - Allocated and deallocated by explicit directives, specified by the programmer, which take effect during execution
  - These variables are nameless
    - Referenced only through separate pointers or references
  - For example
    - Dynamic objects in C++ (via `new` and `delete`)
    - All objects in Java
  - Advantage
    - Provides for dynamic storage management
  - Disadvantage
    - Often unreliable (difficult to use correctly)
    - Inefficient
    - Difficult to implement

Categories of Variables by Lifetimes

- Implicit heap-dynamic variables
  - Allocation and deallocation caused by assignments
    - What kind of type binding does this imply?
  - For example
    - All variables in APL
    - All strings and arrays in Perl, JavaScript, and PHP
  - Advantage
    - Flexibility (generic code)
  - Disadvantages
    - Inefficient (because of attribute binding)
    - Loss of error detection

Variable Attributes: Scope

- The scope of a variable
  - The range of statements over which it is visible
- The local variables of a program unit
  - Variables declared in the unit
- The nonlocal variables of a program unit
  - Variables that are visible but not declared in the unit
- Global variables
  - Special category of nonlocal variables
- Language scope rules
  - Define how references to names are associated with variables

Static Scope

- Based on program text
- To connect a name reference to a variable
  - You (and the compiler) must find the declaration
- Search process
  - Search declarations, starting locally
  - Continue search in increasingly larger enclosing scopes
  - Stop when one is found for the given name
- Enclosing static scopes are called <u>static ancestors</u>
  - Nearest static ancestor is called a <u>static parent</u>
- Some languages allow nested subprogram definitions, which create nested static scopes
  - For example: Ada, JavaScript, Common LISP, Scheme, Fortran 2003, F#, and Python

Scope (continued)

- If a variable in a “closer” scope has the same name as variable in a static ancestor
  - Variable in static ancestor is “hidden”
- Ada allows access to “hidden” variables
  - For example, **`unitName.variableName`**

Blocks

- Method of creating static scopes inside program units
- From ALGOL 60
- Example in C:

```
void sub() {
        int count;
    while (...) {
        int count;
        count++;
    }
}
```

- Variable hiding in blocks is legal in C and C++
- No variable hiding in blocks for Java and C# (error-prone)

The **`LET`** Construct

- Most functional languages
  - Include some form of **`let`** construct
- A **`let`** construct has two parts
  - The first part binds names to values
  - The second part uses the names defined in the first part
- In Scheme:

```
(LET (
  (top (+ a b))
  (bottom (- c d))
  (/ top bottom)
)
```

The `LET` Construct (continued)

- In ML:

```
let
  val name1 = expression1
  …
  val namen = expressionn
in
 expression
end;
```

The **LET** Construct (continued)

- In F#:
  - Left side of **let** is either
    - A name
    - A tuple pattern (a sequence of names)

```
let n1 =
   let n2 = 7
   let n3 = n2 + 3
   n3;;
let n4 = n3 + n1;;
```

Declaration Order

- In C99, C++, Java, and C#
  - Variable declarations can appear anywhere a statement can
- In C99, C++, and Java
  - Scope of local variables is from declaration to end of the block
- In C#
  - Variable scope is the whole block it appears in
  - However a variable can only be used after a declaration

```
{
    { int x; }     ←    Illegal
    int x;
}
```

- In C++, Java, and C#
  - Variables can be declared in `for` statements
  - Variable scope is restricted to the `for` construct

Global Scope

- In C, C++, PHP, and Python
  - Allow a sequence of function definitions, with
  - Variable declarations outside function definitions, and visible to multiple functions
- In C and C++
  - Global declarations (just attributes, no storage)
    - Global variable (possibly with the `extern` keyword)
  - Global definitions (attributes and storage)
    - Global variable with an assignment
  - A declaration outside a function definition means that the variable is defined in another file

Global Scope (continued)

- In PHP
  - Programs in XHTML markup documents
    - Statements and function definitions can be mixed
  - Variable implicitly declared in a function
    - Scope is local to the function
  - Variable implicitly declared outside a function
    - Scope is from the declaration to end of the program
    - Skips over any intervening functions
    - Global variables can be accessed in a function
      - Through the `$GLOBALS` array
      - By declaring it `global` in the function

Global Scope (continued)

- In Python
  - A global variable can be referenced in functions
  - BUT, a global variable can only be assigned to in a function if declared as `global` in the function

Evaluation of Static Scoping

- Advantage
  - Works well in many situations
- Disadvantages
  - In most cases, too much access is possible
  - As program evolves local variables tend to gravitate towards becoming global
  - Subprograms also gravitate toward become global, rather than nested

Dynamic Scope

- Based on calling sequences of program units, not their textual layout
  - In other words, temporal rather than spatial
- To connect variables to declarations
  - Searching back through chain of subprogram calls that brought execution to this point

Scope Example

```
Big
{
     declare X

     Sub1
     {
          declare X
          call Sub2
     }

     Sub2
     {
          refer to X
     }

     call Sub1
}
```

Call order

- **Big** calls **Sub1**
- **Sub1** calls **Sub2**
- **Sub2** uses **X**

Static scoping

- Reference to **X** is to the **X** in **Big**

Dynamic scoping

- Reference to **X** is the **X** in **Sub1**

Evaluation of Dynamic Scoping

- Advantage
  - Convenience (eliminates need for parameters)
- Disadvantages
  - While a subprogram is executing, its variables are visible to all subprograms it calls
  - Impossible to statically type check (why?)
  - Programs more difficult to read (why?)
  - Access to non-local variables is slower

Scope and Lifetime

- Scope and lifetime
  - Sometimes closely related
    - Local variable in Java method
      - What is the scope?
      - What is the lifetime?
  - But are <u>different concepts</u>
    - A `static` variable in a C or C++ function
      - What is the scope?
      - What is the lifetime?

Referencing Environments

- The referencing environment of a statement
  - Collection of all names that are visible in the statement
- In a static-scoped language
  - The local variables plus all of the visible variables in all of the enclosing scopes
- In a dynamic-scoped language
  - The local variables plus all the visible variables in all of the active subprograms
  - Active subprograms have started execution and have not yet terminated

Referencing Environments Example

```
Main() {
    def A, B;

    Sub1 {
       def B, C;
       <--- (2)
       call Sub2;
    }

    Sub2 {
       def C, D;
       <--- (3)
    }

    <--- (1)
    call Sub1;
}
```

- Referencing environment for static scoping:
  - (1): A & B in Main
  - (2): B & C in Sub1, and A in Main
  - (3): C & D in Sub2, and A & B in Main
- Referencing environment for dynamic scoping:
  - (1): A & B in Main
  - (2): B & C in Sub1, and A in Main
  - (3): C & D in Sub2, B in Sub1, and A in Main

Named Constants

- Definition
  - Variable bound to a value only once
  - Used to parameterize programs (e.g. `SIZE` for C arrays)
- Advantages
  - Readability and reliability
- Binding of values to named constants
  - Static (manifest constants)
  - Dynamic
- Languages:
  - FORTRAN 95
    - Constant-valued expressions (static or dynamic?)
  - Ada, C++, and Java
    - Expressions of any kind (static or dynamic?)
  - C# has two kinds
    - Values of `const` named constants bound at compile time
    - Values of `readonly` named constants dynamically bound

Summary

- Case sensitivity and the relationship of names to special words represent design issues of names
- Variables are characterized by the sextuples: name, address, value, type, lifetime, scope
- Binding is the association of attributes with program entities
- Scalar variables are categorized as: static, stack dynamic, explicit heap dynamic, implicit heap dynamic
- Strong typing means detecting all type errors