Names, Bindings and Scopes

R. W. Sebesta, Concepts of Programming Languages, Chapter 5

M. L. Scott, Programming Language Pragmatics, Chapter 3

Introduction

- ‘High level’ programing languages
  - Machine independence
- Design of modern languages
  - Machine independence remains important
  - Ease of programming
- We will address the core issues in language design.

Names

- A ***name*** is a mnemonic character string used to represent something else.
  - Most names are identifiers (alphanumeric tokens)
  - Other symbols (like '+') can also be names
  - Names allow us to refer to variables, constants, operations, types, and so on using symbolic identifiers.
- Design issues for names:
  - Case sensitivity
  - Special words – keywords, reserved words

Names

- **Case sensitivity**
  - Names in the C-based languages are case sensitive
    - Ex: rose, ROSE, and Rose are distinct in C++
  - There are programming languages where names are not case sensitive
    - Ex: Ada, Fortran
  - Disadvantage (for some people): readability (names that look alike are different)
    - Ex: Rose and rose look similar, there is no connection between them.

Names

- **Case sensitivity (cont.)**
  - Not everyone agrees that case sensitivity is bad for names.
  - In C, the problems of case sensitivity are avoided by the convention that variable names do not include uppercase letters.
  - In Java and C#, predefined names include both uppercase and lowercase letters.
    - Ex: IndexOutOfBoundsException
    - This is a problem of writability rather than readability.
      - Because the need to remember specific case usage makes it more difficult to write correct programs.

Names

- **Special words**
  - An aid to readability; used to delimit or separate statement clauses.
  - A ***keyword*** is a word that is special only in certain contexts.
    - Ex: in Fortran
      - `Real VarName` (Real is a data type followed with a name, therefore Real is a keyword)
      - `Real = 3.4` (Real is a variable)
      - Allowed but not readable:
        - `INTEGER REAL`
          `REAL INTEGER`

Names

- **Special words**
  - A ***reserved word*** is a special word that cannot be used as a user-defined name.
    - Can't define `for` or `while` as function or variable names.
    - Good design choice
  - Potential problem: If there are too many, many collisions occur
    - Ex: COBOL has 300 reserved words!

Names

- **Length**
  - If too short, they cannot be connotative
  - Language examples:
    - Earliest languages : single character
    - FORTRAN 95: maximum of 31 characters
    - C99: no limit but only the first 63 are significant; also, external names are limited to a maximum of 31
    - C# and Java: no limit
    - C++: no limit, but implementers often impose one

Variables

- A ***variable*** is an abstraction of a memory cell or collection of cells.
- Variables can be characterized as a sextuple of attributes:
  - Name
  - Address
  - Type
  - Value
  - Lifetime
  - Scope

Variables Attributes

- **Name**
  - Most variables have names
- **Address** - the memory address with which it is associated
  - A variable may have different addresses at different times during execution.
    - For a variable declared in a recursive procedure, in different steps of recursion it refers to different locations.
  - A variable may have different addresses at different places in a program.
    - A program can have two subprograms `sub1` and `sub2` each of defines a local variable that use the same name, e.g. `sum`.

Variables Attributes

- **Address** (cont.)
  - If two variable names can be used to access the same memory location, they are called ***aliases.***
  - Aliases are created via pointers, reference variables, C and C++ unions.
  - Aliases are harmful to readability (program readers must remember all of them).

Variables Attributes

- **Type**
  - determines the range of values of variables
  - determines the set of operations that are defined for values of that type
  - in the case of floating point, type also determines the precision
  - Ex: The int type in Java specifies a value range of −2147483648 to 2147483647 and arithmetic operations for addition, subtraction, multiplication, division, and modulus.

Variables Attributes

- **Value** - the contents of the location with which the variable is associated
- **Lifetime**
- **Scope**

The Concept of Binding

- A ***binding*** is an association between two things, such as a name and the thing it names.

- A ***binding*** is an association between an *entity* and an *attribute*, such as between
  - a *variable* and its *type* or *value*, or
  - an *operation* and a *symbol*

- ***Binding time*** is the time at which a binding takes place.

- ***Binding*** and ***binding times*** are prominent concepts in the semantics of programming languages.

Possible Binding Times

- **Language design time**
- **Language implementation time**
- **Compile time**
- **Load time**
- **Runtime**

Possible Binding Times

- **Language design time**
  - bind operator symbols to operations
  - * is bound to the multiplication operation
  - pi=3.14159 in most PL's
- **Language implementation time**
  - bind floating point type to a representation
  - `int` in C is bound to a range of possible values

Possible Binding Times (Cont.)

- **Compile time**
  - bind a variable to a type in C or Java
- **Load time**
  - a variable is bound to a specific memory location
  - bind a C or C++ static variable to a memory cell
- **Runtime**
  - bind a nonstatic local variable to a memory cell
  - a variable is bound to a value through an assignment statement

Binding Time

- The terms STATIC and DYNAMIC are generally used to refer to things bound *before* run time and *at* run time, respectively.
- A binding is ***static*** if it first occurs before run time and remains unchanged throughout program execution.
- A binding is ***dynamic*** if it first occurs during execution or can change during execution of the program.
- In general, early binding times are associated with greater efficiency.
- Later binding times are associated with greater flexibility.

Binding Time

- Compiled languages tend to have early binding times.
  - Ex: A compiler analyzes the syntax and semantics of global variable declarations once, before the program ever runs.
  - They tend to be more efficient.
- Interpreted languages tend to have later binding times.
  - Ex: A pure interpreter must analyze the declarations every time the program begins execution.
  - They tend to be more flexible.

Type Binding

- Before a variable can be referenced in a program, it must be bound to a data type.
- The two important aspects of this binding:
  - How is the type specified?
  - When does the binding take place?
- Types can be specified statically or dynamically.
  - Static Type Binding
  - Dynamic Type Binding
- If static, the type may be specified by either an explicit or an implicit declaration.

Static Type Binding: Variable Declarations

- **Explicit declaration** *(by statement)*
  - A program statement used for declaring the types of variables.
  - Lists variable names and specifies that they are a particular type
  - Ex: `int sum`, `float average`
- **Implicit declaration** (*by first appearance*)
  - Means of associating variables with types through default conventions, rather than declaration statements. First appearance of a variable name in a program constitutes its implicit declaration.
  - Ex: In Fortran, if not explicitly declared, an identifier starting with I,J,K,L,M,N are implicitly declared to integer, otherwise to real type.

Static Type Binding: Variable Declarations

- **Implicit Declaration** (cont.)
  - Not good for reliability and writability because misspelled identifier names cannot be detected by the compiler.
  - Some problems of implicit declarations can be avoided by requiring names for specific types to begin with a particular special characters.
    - Ex: in Perl
      - $ceng: scalar
      - @ceng: array
      - %ceng: hash structure

Dynamic Type Binding

- Type of a variable is not specified by a declaration statement, nor can it be determined by the spelling of its name. (JavaScript, Python, Ruby, PHP, and C# (limited))
- The variable is bound to a type when it is assigned a value in an assignment statement.
- Advantage: Allows programming flexibility
- Ex: In JavaScript
  - list = [10.2, 3.5];
    - list is a single dimensioned array of length 2.
  - list = 47;
    - list is a simple integer

Dynamic Type Binding

- Disadvantages:
  1. Less reliable: Compiler cannot check and enforce types
     - Ex: Suppose I and X are integer variables, and Y is a floating-point.
       - The correct statement is I := X
       - But by a typing error I := Y is typed.
       - In a dynamic type binding language, this error cannot be detected by the compiler.
         - I is changed to float during execution.
         - The value of I becomes erroneous.

Dynamic Type Binding

- Disadvantages:
  2. High cost: Type checking must be done at run-time.

- Languages that use dynamic type bindings are usually implemented as interpreters (LISP is such a language)

Lifetime and Storage Management

- **Allocation** - getting a cell from some pool of available cells
- **Deallocation** - putting a cell back into the pool
- The ***lifetime*** of a variable is the time during which it is bound to a particular memory cell.
- Categories of **Variables** by **Lifetimes**
  - Static Variables
  - Stack-Dynamic Variables
  - Explicit Heap-Dynamic Variables
  - Implicit Heap-Dynamic Variables

Static Variables

- Are bound to memory cells before execution begins and remains bound to the same memory cell throughout execution.
- Applications: globally accessible variables, to make some variables of subprograms to retain values between separate execution of the subprogram
- Such variables are history sensitive
- Examples:
  - All variables in FORTRAN I, II, and I
  - Static variables in C, C++, Java
- Advantage: Efficiency (Direct addressing - no run-time overhead for allocation and deallocation)
- Disadvantage: Lack of flexibility (no recursion)

Stack-Dynamic Variables

- Storage bindings are created for variables when their declaration statements are *elaborated (in runtime).*
  - A declaration is elaborated when the executable code associated with it is executed.
  - Ex: The variable declarations that appear at the beginning of a Java method are elaborated when the method is called, and the variables defined by those declarations are deallocated when the method completes its execution.
- In C and C++, local variables are, by default, stack- dynamic
- In Java, C++, and C#, variables defined in methods are by default stack dynamic
- Advantage: Allows recursion; conserves storage
- Disadvantage: Runtime overhead for allocation and deallocation

Explicit Heap-Dynamic Variables

- Nameless variables
- Allocated and deallocated by explicit run-time instructions written by the programmer
- Referenced only through pointers or references
- Ex: Dynamic objects in C++ (via **`new`** and **`delete`**), all objects in Java

Explicit Heap-Dynamic Variables

```
int *intnode;          // Create a pointer
intnode = new int;     // Create the heap-dynamic variable
  . . .
delete intnode;        // Deallocate the heap-dynamic variable
                       // to which intnode points
```

- Advantage: Required for dynamic structures (e.g., linked lists, trees)
- Disadvantage: Difficult to use correctly, costly to refer, allocate, deallocate

Implicit Heap-Dynamic Variables

- Are bound to heap storage only when they are assigned values.
- Allocation and deallocation caused by assignment statements
  - *Storage binding* is done when they are assigned values.
- Ex: All variables in APL; all strings and arrays in Perl, JavaScript, and PHP
- Advantage: Highest degree of flexibility (generic code)
- Disadvantages:
  - Runtime overhead for allocation and deallocation
  - Loss of error detection by compiler

Variable Attribute: Scope

- The ***scope*** of a variable is the range of statements in which the variable is visible.
- A variable is visible in a statement if it can be referenced or assigned in that statement.
- The **scope rules** of a language determine how references to names are associated with variables
  - Static (Lexical) Scope Rules
  - Dynamic Scope Rules

Static (Lexical) Scope Rules

- A scope is defined in terms of the physical (lexical) structure of the program.
- Based on program text
- All bindings for identifiers can be resolved by examining the program.
- Typically, we choose the most recent, active binding made at compile time.
- The determination of scopes can be made by the compiler.

Static (Lexical) Scope Rules

- Ex: We can look at a C program and know which names refer to which objects at which points in the program based on purely textual rules.

- Search process:
  - search declarations,
    - first locally,
    - then in increasingly larger enclosing scopes,
    - until one is found for the given name

Dynamic Scope Rules

- With dynamic scope rules, bindings depend on the current state of program execution.
  - They cannot always be resolved by examining the program because they are dependent on calling sequences.
  - To resolve a reference, we use the most recent, active binding made at run time.
- Scope is determined in run-time.
- Dynamic scope rules are usually encountered in interpreted languages.

Scope Rules
Example: Static vs. Dynamic

```
function big() {
 function sub1()
    var x = 7;
 function sub2(){
    var y = x;
 }
 var x = 3;
}
```

```
big calls sub1
sub1 calls sub2
sub2 uses x
```

- Static scoping
  - Reference to x in sub2 is to big's x
- Dynamic scoping
  - Reference to x in sub2 is to sub1's x

An Example

a) What does the function test print if the language uses static scoping?

b) What does it print with dynamic scoping?

https://www.cs.virginia.edu/~asb

```
int  n = 1;    // global

print_plus_n(int x) {
     cout << x + n;
}
increment_n() {
     n = n + 2;
     print_plus_n(n);
}

test() {
     int n;
     n = 200;
     print_plus_n(7);

     n = 50;
     increment_n();
     cout << n;

}
```

An Example

a) What does the function test print if the language uses static scoping?

8 6 50

a) What does it print with dynamic scoping?

207 104 52

https://www.cs.virginia.edu/~asb

```
int  n = 1;    // global

print_plus_n(int x) {
     cout << x + n;
}
increment_n() {
     n = n + 2;
     print_plus_n(n);
}

test() {
     int n;
     n = 200;
     print_plus_n(7);

     n = 50;
     increment_n();
     cout << n;

}
```

Scope Rules
Example: Static vs. Dynamic

```
function big() {
 function sub1()
    var x = 7;
 function sub2(){
    var y = x;
 }
 var x = 3;
}
```

```
big calls sub1
sub1 calls sub2
sub2 uses x
```

- Static scoping
  - Reference to x in sub2 is to big's x
- Dynamic scoping
  - Reference to x in sub2 is to sub1's x

Scope Rules
Example: Static vs. Dynamic

```
program scopes (input, output);
var a : integer;
procedure first;
        begin a := 1; end;
procedure second;
      var a : integer;
      begin first; end;
begin
      a := 2; second; write(a);
end.
```

Scope Rules
Example: Static vs. Dynamic

- If static scope rules are in effect (as would be the case in Pascal), the program prints *a* 1
- If dynamic scope rules are in effect, the program prints *a* 2
- Why the difference? At issue is whether the assignment to the variable *a* in procedure *first* changes the variable a declared in the main program or the variable *a* declared in procedure *second*

Scope Rules
Example: Static vs. Dynamic

Static Scoping

"Physical (Lexical) Structure"

- Global *a* becomes 2.
- Procedure *second* is called.
- Inside procedure *second*, procedure *first* is called.
  - Which *a* is referred?
    - Global *a*, because *first* is nested into program scopes.
    - Global *a* becomes 1.
- Global *a* is printed.
  - 1 is printed.

Scope Rules
Example: Static vs. Dynamic

Dynamic Scoping

"The most recent active binding"

- Global *a* becomes 2.
- Procedure *second* is called.
- Inside procedure *second*, procedure *first* is called.
- Which *a* is referred?
  - Procedure *second*'s local variable *a* becomes 1.
- Global *a* is printed.
  - 2 is printed.