Programming Languages – Names Bindings and Scopes

Jongwoo Lim

Chapter 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 characterized by attributes
  - To design a type, must consider scope, lifetime, type checking, initialization, and type compatibility

Names

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

Names (continued)

- Length limit
  - If too short, they cannot be connotative
  - Language examples:
    - FORTRAN 95: maximum of 31
    - C99: no limit but only the first 63 are significant; also, 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: variable names that begin with `@` are instance variables; those that begin with `@@` are class variables.

Names (continued)

- Case sensitivity
  - Disadvantage: readability (names that look alike are different)
    - Names in the C-based languages are case sensitive
    - Names in others are not
    - Worse in C++, Java, and C# because predefined names are mixed case (e.g. `IndexOutOfBoundsException`)

Names (continued)

- Special words
  - A **keyword** is a word that is special only in certain contexts.
    - e.g., in Fortran,

```
Integer Apple          Integer Real
Integer = 4            Real Integer
```

  - A **reserved** word is a special word that cannot be used as a user-defined name.
  - Potential problem with reserved words:
    If there are too many, many collisions occur (e.g., COBOL has 300 reserved words! including `LENGTH`, `BOTTOM`, `DESTINATION`, `COUNT`, etc.)

Variables

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

Variables Attributes

- **Name** - not all variables have them.
- **Address** - the memory address with which it is associated
  - A variable may have different addresses at different times during execution
  - A variable may have different addresses at different places in a program
  - 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 (continued)

- **Type** - determines the range of values of variables and the set of operations that are defined for values of that type; in the case of floating point, type also determines the precision

- **Value** - the contents of the location with which the variable is associated
  - The **l-value** of a variable is its address
  - The **r-value** of a variable is its value

    ```
    e.g. int a = 0;  a = a + 1;
    ```

- Abstract memory cell - the physical cell or collection of cells associated with a variable

The Concept of Binding

- A **binding** is an association, such as between an attribute and an entity, or between an operation and a symbol.
- **Binding time** is the time at which a binding takes place.
  - Language design time
    - bind operator symbols to operations, e.g. + : addition.
  - 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 or C++ static variable to a memory cell)
  - Runtime (bind a nonstatic local variable to a memory cell)

Static and Dynamic Binding

- 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.

- Type binding:
  - How is a type specified?
  - When does the binding take place?
  - If static, the type may be specified by either an explicit or an implicit declaration.

Explicit/Implicit Declaration

- An **explicit declaration** is a program statement used for declaring the types of variables
- An **implicit declaration** is a default mechanism for specifying types of variables (the first appearance of the variable in the program)
  - FORTRAN, BASIC, and Perl provide implicit declarations (Fortran has both explicit and implicit)
  - Advantage: writability
  - Disadvantage: reliability (less trouble with Perl - `@` : array, `%` : hash struct)

Dynamic Type Binding

- Dynamic Type Binding
  - e.g. JavaScript and PHP
- Specified through an assignment statement
  - e.g., JavaScript

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

  - Advantage: flexibility (generic program units)
  - Disadvantages:
    - High cost (dynamic type checking and interpretation)
    - Type error detection by the compiler is difficult

Type Inferencing

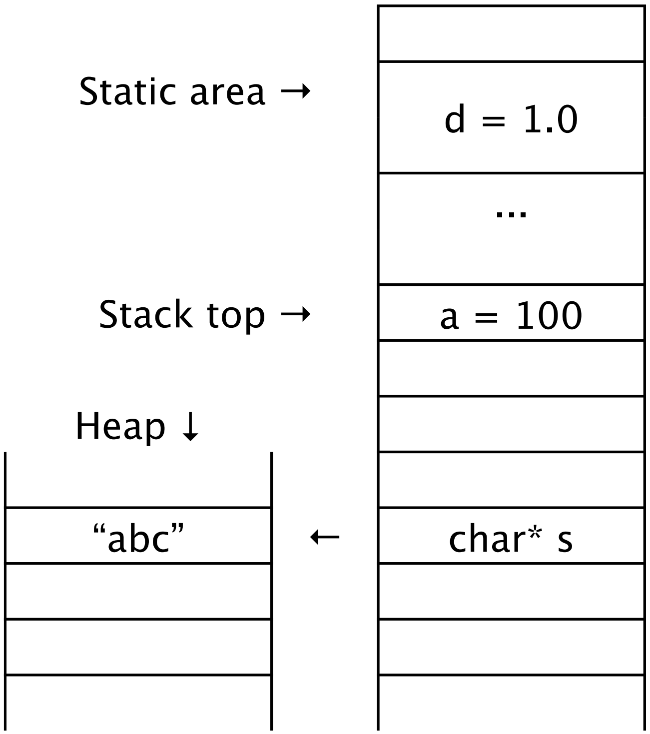
- **Type Inferencing** (ML, Miranda, and Haskell)
  - Rather than by assignment statement, types are determined (by the compiler) from the context of the reference

```
fun circum(r) = 3.141592 * r * r;
fun times10(x) = 10 * x;
fun square(x) = x * x;     -- int

fun square(x) : real = x * x;
fun square(x : real) = x * x;
fun square(x) = (x : real) * x;
fun square(x) = x * (x : real);
```

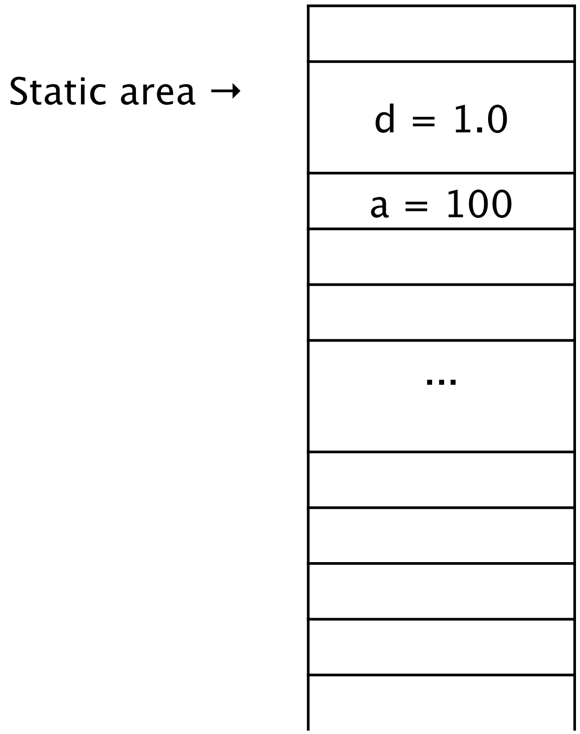
Storage Bindings

- Storage Bindings & Lifetime
  - 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
  - Static
  - Stack-dynamic
  - Explicit heap-dynamic
  - Implicit heap-dynamic



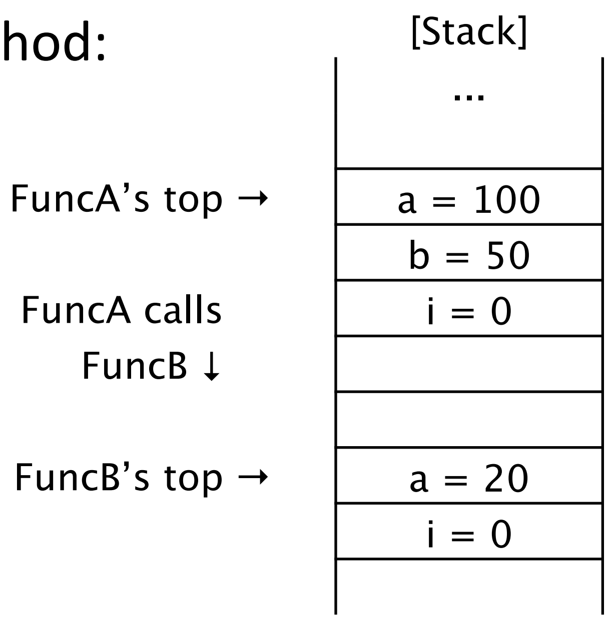
Categories of Variables by Lifetimes

- **Static**: bound to memory cells before execution begins and remains bound to the same memory cell throughout execution
  - e.g., C and C++ `static` variables
  - Advantages: efficiency (direct addressing), history-sensitive subprogram support
  - Disadvantage: lack of flexibility (no recursion)



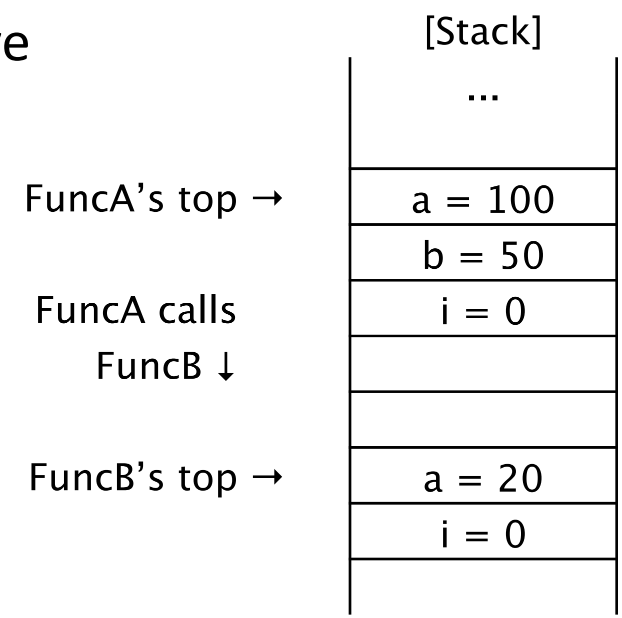
Categories of Variables by Lifetimes

- **Stack-dynamic**: storage bindings are created for variables when their declaration statements are elaborated.
  - A declaration is **elaborated** when the executable code associated with it is executed.
  - Variables are allocated from the runtime stack.
  - Declaration may be in the middle of a method:
    - Storage binding occurs when the method begins execution, but the variable becomes visible at the declaration.



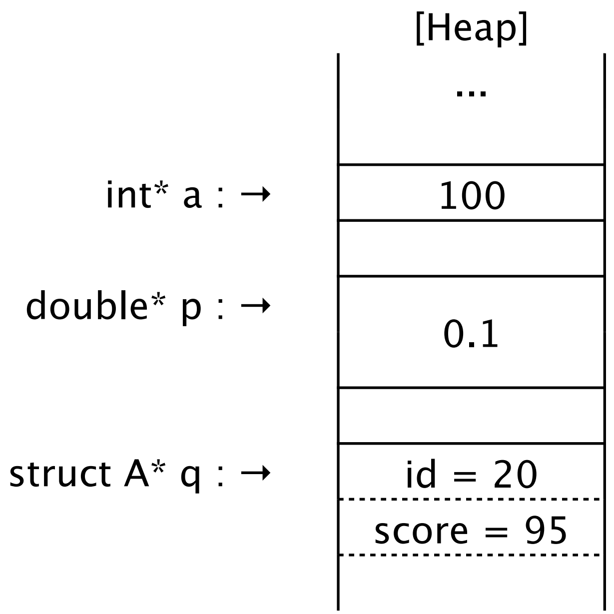
Categories of Variables by Lifetimes

- **Stack-dynamic**: storage bindings are created for variables when their declaration statements are elaborated.
  - Advantage: allows recursion; conserves storage
  - Disadvantages:
    - Overhead of allocation and deallocation
    - Subprograms cannot be history sensitive
    - Inefficient references (indirect addressing)



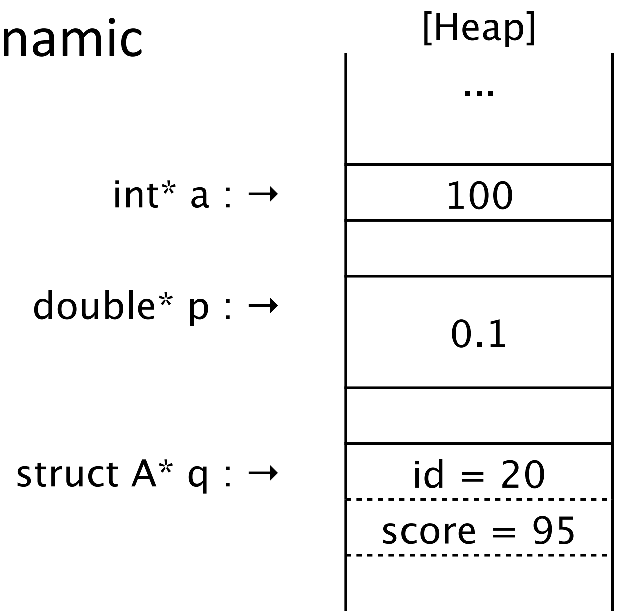
Categories of Variables by Lifetimes

- **Explicit heap-dynamic**: allocated and deallocated by explicit directives, specified by the programmer, which take effect during execution
  - Referenced only through pointers or references, e.g. dynamic objects in C++ (via `new` and `delete`), all objects in Java
  - Advantage: provides for dynamic storage management
  - Disadvantage: inefficient and unreliable



Categories of Variables by Lifetimes

- **Implicit heap-dynamic**: allocation and deallocation caused by assignment statements
  - all variables in APL; all strings and arrays in Perl, JavaScript, and PHP
  - Advantage: flexibility (generic code)
  - Disadvantages:
    - Inefficient, because all attributes are dynamic
    - Loss of error detection



Scope

- The **scope** of a variable is the range of statements over which it is visible.
- The **nonlocal variables** of a program unit are those that are visible but not declared there.
- The scope rules of a language determine how references to names are associated with variables.

Static Scope

- Based on program text (source code)
  - To connect a name reference to a variable, you (or the compiler) must find the declaration.
  - **Search process**: search declarations, first locally, then in increasingly larger enclosing scopes, until one is found for the given name.
  - Enclosing static scopes (to a specific scope) are called its **static ancestors**; the nearest static ancestor is called a **static parent**.
  - Some languages allow nested subprogram definitions, which create nested static scopes (e.g., Ada, JavaScript, Fortran 2003, and PHP).

Static Scope

- Variables can be hidden from a unit by having a "closer" variable with the same name:

```
procedure Big is
  X : Integer;
  procedure Sub1 is
    X: Integer;
    begin        -- of Sub1
    ...
    end          -- of Sub1
  procedure Sub2 is
    begin        -- of Sub2
    ... X ...
    end          -- of Sub2
  begin          -- of Big
  ...
  end            -- of Big
```

  - Ada allows access to these "hidden" variables (e.g. Big::X)

Blocks

- A method of creating static scopes inside program units.

```
if (list[i] < list[j]) {
  int temp;
  temp = list[i];
  list[i] = list[j];
  list[j] = temp;
}
```

Blocks

- A method of creating static scopes inside program units.

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

- Legal in C/C++, but not in Java and C#: too error-prone.

Declaration Order

- C99, C++, Java, and C# allow variable declarations to appear anywhere a statement can appear.
  - In C99, C++, and Java, the scope of all local variables is from the declaration to the end of the block.
  - In C#, the scope of any variable declared in a block is the whole block, regardless of the position of the declaration in the block, however, a variable still must be declared before it can be used.

Declaration Order

- In C++, Java, and C#, variables can be declared in for statements:
  - The scope of such variables is restricted to the for construct.

```
void fun() {
  ...
  for (int count = 0; count < 10; ++count) {
    ...
  }
  ...
}
```

Global Scope

- C, C++, PHP, and Python support a program structure that consists of a sequence of function definitions in a file.
  - These languages allow variable declarations to appear outside function definitions.
- C and C++ have both declarations (just attributes) and definitions (attributes and storage).
  - A declaration outside a function definition specifies that it is defined in another file.

```
extern int sum;
```

Global Scope

- PHP:
  - The scope of a variable (implicitly) declared in a function is local to the function.
  - The scope of a variable implicitly declared outside functions is from the declaration to the end of the program, but skips over any intervening functions
    - Global variables are not visible in any function, but they can be accessed in a function through the `$GLOBALS` array or by declaring it `global`.

Global Scope

- PHP example

```
$day = "Monday";
$month = "January";

function calendar() {
  $day = "Tuesday";
  global $month;
  print "local day is $day <br/>";
  $gday = $GLOBALS['day'];
  print "global day is $gday <br/>";
  print "global month is $month <br/>";
}

calendar();
```

⇒

```
local day is Tuesday
global day is Monday
global month is January
```

Global Scope

- Python:
  - A global variable can be referenced in functions, but can be assigned in a function only if it has been declared to be global in the function

```
day = “Monday”

def tester():
  print “The global day is: “, day

tester()
```

⇒ The global day is: Monday

Global Scope

- Python example:

```
day = “Monday”

def tester():
  print “The global day is: “, day
  day = “Tuesday”
  print “The new value of day is:”, day

tester()
```

⇒ `UnboundLocalError`

Global Scope

- Python example:

```
day = “Monday”

def tester():
  global day
  print “The global day is: “, day
  day = “Tuesday”
  print “The new value of day is:”, day

tester()
```

⇒

```
The global day is: Monday
The new value of day is: Tuesday
```

Evaluation of Static Scoping

- Works well in many situations.
- Problems:
  - In most cases, too much access is possible.
  - As a program evolves, the initial structure is destroyed and local variables often become global; subprograms also gravitate toward become global, rather than nested.

Dynamic Scope

- Based on calling sequences of program units, not their textual layout (temporal versus spatial).
  - References to variables are connected to declarations by searching back through the chain of subprogram calls that forced execution to this point.

Dynamic Scope

- Dynamic scope example:

```
procedure Big is
  X : Integer;
  procedure Sub1 is
    X: Integer;
    begin        -- of Sub1                  Big calls Sub1
    ... Sub2 ...                             Sub1 calls Sub2
    end          -- of Sub1                  Sub2 uses X
  procedure Sub2 is
    begin        -- of Sub2                  vs.
    ... X ...
    end          -- of Sub2                  Big calls Sub2
  begin          -- of Big                   Sub2 uses X
  ... Sub1 ...
  ... Sub2 ...
  end            -- of Big
```

Evaluation of Dynamic Scoping

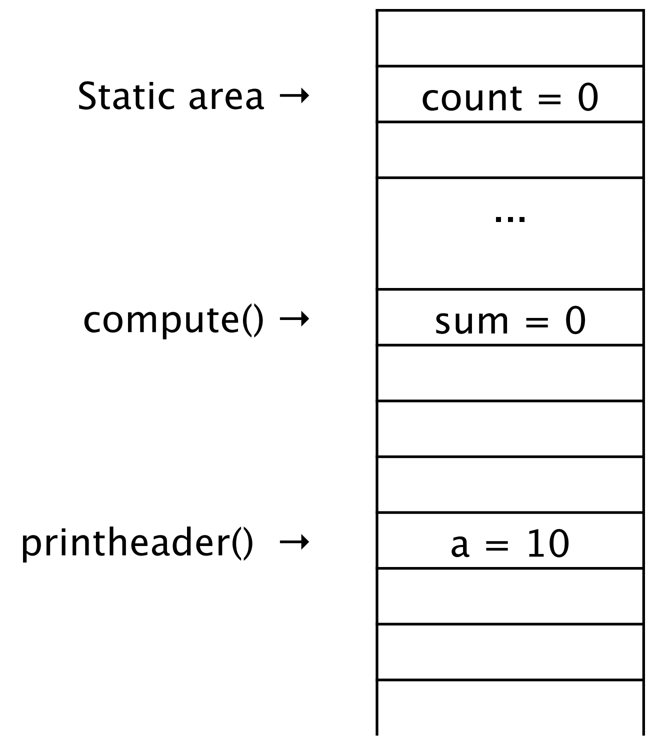
- Evaluation of Dynamic Scoping:
  - Advantage: convenience
  - Disadvantages:
    - While a subprogram is executing, its variables are visible to all subprograms it calls.
    - Impossible to statically type check.
    - Poor readability - it is not possible to statically determine the type of a variable.

Scope and Lifetime

- Scope and lifetime are sometimes closely related, but are different concepts
  - Consider a `static` variable in a C or C++ function.
  - Another example:

```
void printheader() {
  static int count = 0;
  int a = 10;
  ++count;
  ...
}

void compute() {
  int sum = 0;
  ...
  printheader();
  ...
}
```



Referencing Environments

- The **referencing environment** of a statement is the collection of all names that are visible in the statement.
  - In a static-scoped language, it is the local variables plus all of the visible variables in all of the enclosing scopes.
  - A subprogram is **active** if its execution has begun but has not yet terminated.
  - In a dynamic-scoped language, the referencing environment is the local variables plus all visible variables in all active subprograms.

Referencing Environments

- Ada example: static scoped

```
      procedure Example is
        A, B : Integer;
        procedure Sub1 is
          X, Y : Integer;
          begin       -- of Sub1
1 →       ...
          end         -- of Sub1
        procedure Sub2 is
          X : Integer;
          ...
          procedure Sub3 is
            X : Integer;
            begin     -- of Sub3
2 →         ...
            end       -- of Sub3
          begin       -- of Sub2
3 →       ...
          end         -- of Sub2
        begin         -- of Example
4 →     ...
        end           -- of Example
```

```
1: X,Y of Sub1, A,B of Example
2: X of Sub3, A,B of Example
3: X of Sub2, A,B of Example
4: A,B of Example
```

Referencing Environments

- Dynamic scoped example:

```
     void sub1() {
       int a, b;
1 →    ...
     }

     void sub2() {
       int b, c;
2 →    ...
       sub1();
     }

     void main() {
       int c, d;
3 →    ...
       sub2();
     }
```

```
1: a,b of sub1, c of sub2, d of main
2: b,c of sub2, d of main
3: c,d of main
```

Named Constants

- A named constant is a variable that is bound to a value only when it is bound to storage.
  - Advantages: readability and modifiability.
  - Used to parameterize programs.

```
void example() {
  int[] intList = new int[100];
  String[] strList = new String[100];
  ...
  for (idx = 0; idx < 100; ++idx) {
    ...
  }
  for (idx = 0; idx < 100; ++idx) {
    ...
  }
  average = sum / 100;
}
```

```
void example() {
  final int len = 100;
  int[] intList = new int[len];
  String[] strList = new String[len];
  ...
  for (idx = 0; idx < len; ++idx) {
    ...
  }
  for (idx = 0; idx < len; ++idx) {
    ...
  }
  average = sum / len;
}
```

Named Constants

- The binding of values to named constants can be either static (called **manifest constants**) or dynamic.
- Languages:
  - FORTRAN 95: constant-valued expressions.
  - Ada, C++, and Java: expressions of any kind.
  - C# has two kinds, **readonly** and **const**.
    - The values of const named constants are bound at compile time.
    - The values of readonly named constants are 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