Programming Languages:

Lecture 5

Chapter 5: Names, Bindings, Type Checking, and Scopes

Jinwoo Kim
jwkim@jjay.cuny.edu

Chapter 5 Topics

- Introduction
- Names
- Variables
- The Concept of Binding
- Type Checking
- Strong Typing
- Type Compatibility
- Scope and Lifetime
- Referencing Environments
- Named Constants

Introduction

- Imperative languages are abstractions of von Neumann architecture
  - Memory
  - Processor

- Variables characterized by attributes
  - Type: to design, must consider scope, lifetime, type checking, initialization, and type compatibility

Names

- Design issues for names:
  - Maximum length?
  - Are connector characters allowed?
  - Are names case sensitive?
  - Are special words reserved words or keywords?

Names (continued)

- Length
  - If too short, they cannot be connotative
  - Language examples:
    - **FORTRAN I: maximum 6**
    - **COBOL: maximum 30**
    - **FORTRAN 90 and ANSI C: maximum 31**
    - **Ada and Java: no limit, and all are significant**
    - **C++: no limit, but implementers often impose one**

Names (continued)

- Connectors
  - Pascal, Modula-2, and FORTRAN 77 don't allow
  - Others do

Names (continued)

- Case sensitivity
  - Disadvantage: readability (names that look alike are different)
    - **worse in C++ and Java because predefined names are mixed case (e.g. `IndexOutOfBoundsException`)**
  - C, C++, and Java names are case sensitive
    - **The names in other languages are not**

Names (continued)

- 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, e.g., 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)***
  - A *reserved word* is a special word that cannot be used as a user-defined name

Variables

- A variable is an abstraction of a memory cell
- Variables can be characterized as a six-tuple 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
- *Abstract memory cell* - the physical cell or collection of cells associated with a variable

The Concept of Binding

- The l-value of a variable is its address
- The r-value of a variable is its value
- 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.

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 FORTRAN 77 variable to a memory cell (or a C `static` variable)
- 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, PL/I, BASIC, and Perl provide implicit declarations
  - Advantage: writability
  - Disadvantage: reliability (less trouble with Perl)

Dynamic Type Binding

- Dynamic Type Binding (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**

Variable Attributes (continued)

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

- 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

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., all FORTRAN 77 variables, 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
- If scalar, all attributes except 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* -- 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 and JavaScript
- Advantage: flexibility
- Disadvantages:
  - Inefficient, because all attributes are dynamic
  - Loss of error detection

Type Checking

- Generalize the concept of operands and operators to include subprograms and assignments
- *Type checking* is the activity of ensuring that the operands of an operator are of compatible types
- A *compatible type* is one that is either legal for the operator, or is allowed under language rules to be implicitly converted, by compiler- generated code, to a legal type
  - This automatic conversion is called a coercion
- A *type error* is the application of an operator to an operand of an inappropriate type

Type Checking (continued)

- If all type bindings are static, nearly all type checking can be static
- If type bindings are dynamic, type checking must be dynamic
- A programming language is *strongly typed* if type errors are always detected

Strong Typing

- Advantage of strong typing: allows the detection of the misuses of variables that result in type errors

- Language examples:
  - FORTRAN 77 is not: parameters, `EQUIVALENCE`
  - Pascal is not: variant records
  - C and C++ are not: parameter type checking can be avoided; unions are not type checked
  - Ada is, almost (`UNCHECKED CONVERSION` is loophole)

    (Java is similar)

Strong Typing (continued)

- Coercion rules strongly affect strong typing--they can weaken it considerably (C++ versus Ada)

- Although Java has just half the assignment coercions of C++, its strong typing is still far less effective than that of Ada

Name Type Compatibility

- *Name type compatibility* means the two variables have compatible types if they are in either the same declaration or in declarations that use the same type name

- Easy to implement but highly restrictive:
  - Subranges of integer types are not compatible with integer types
  - Formal parameters must be the same type as their corresponding actual parameters (Pascal)

Structure Type Compatibility

- *Structure type compatibility* means that two variables have compatible types if their types have identical structures

- More flexible, but harder to implement

Type Compatibility (continued)

- Consider the problem of two structured types:
  - Are two record types compatible if they are structurally the same but use different field names?
  - Are two array types compatible if they are the same except that the subscripts are different?

    (e.g. [1..10] and [0..9])
  - Are two enumeration types compatible if their components are spelled differently?
  - With structural type compatibility, you cannot differentiate between types of the same structure (e.g. different units of speed, both float)

Variable Attributes: 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
- 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

Scope (continued)

- Variables can be hidden from a unit by having a "closer" variable with the same name
- C++ and Ada allow access to these "hidden" variables
  - In Ada: **`unit.name`**
  - In C++: **`class_name::name`**

Blocks

- A method of creating static scopes inside program units--from ALGOL 60
- Examples:

C and C++:
```
for (...) {
    int index;
    ...
}
```

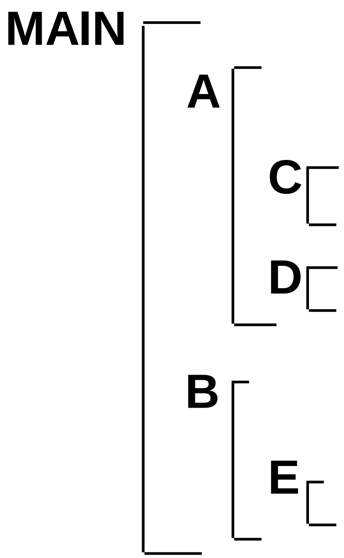
Ada:
```
declare LCL : FLOAT;
begin
    ...
end
```

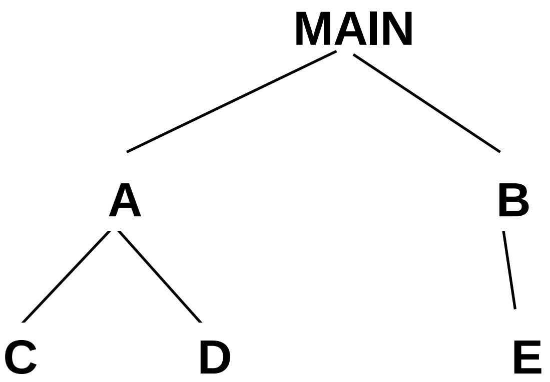
Evaluation of Static Scoping

- Assume MAIN calls A and B

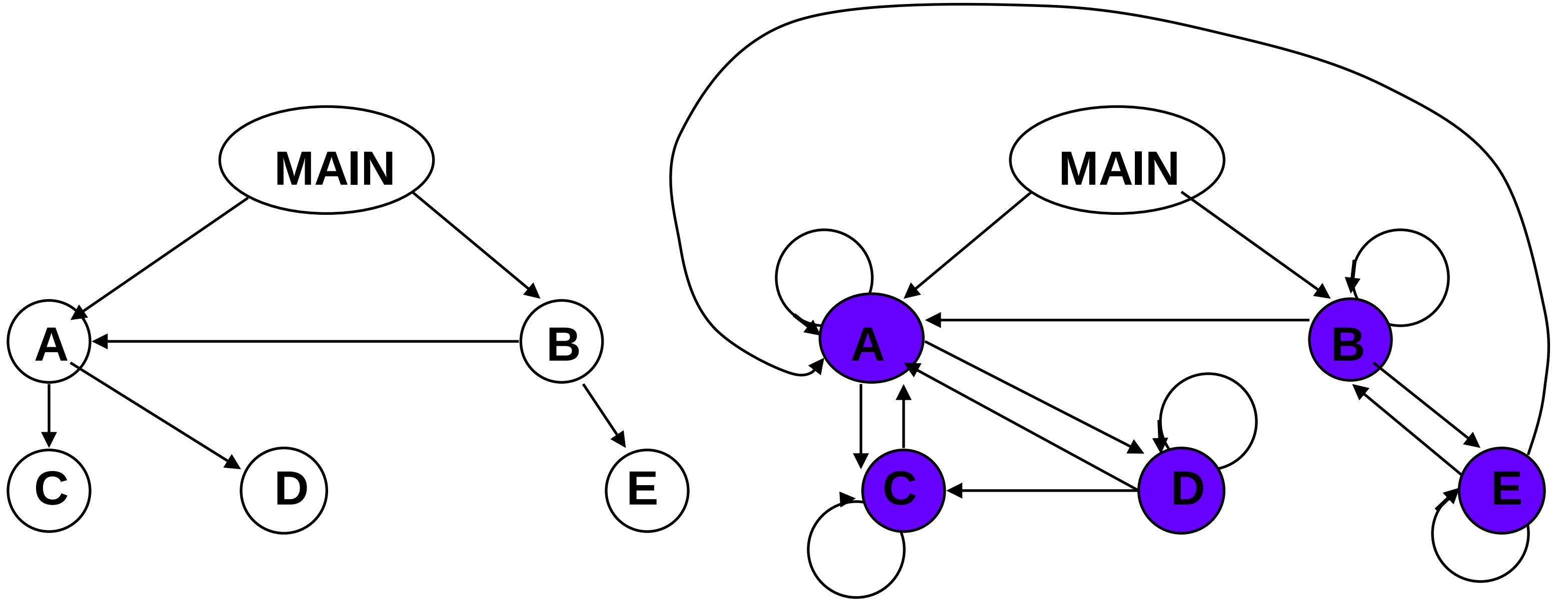
  A calls C and D

  B calls A and E





Static Scope Example



Static Scope (continued)

- Suppose the spec is changed so that D must now access some data in B
- Solutions:
  - Put D in B (but then C can no longer call it and D cannot access A's variables)
  - Move the data from B that D needs to MAIN (but then all procedures can access them)
- Same problem for procedure access
- Overall: static scoping often encourages many globals

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

Scope Example

MAIN
- declaration of x

SUB1
- declaration of x -
...
call SUB2
...

SUB2
...
- reference to x -
...

...
call SUB1
…

MAIN calls SUB1
SUB1 calls SUB2
SUB2 uses x

Scope Example

- Static scoping
  - Reference to x is to MAIN's x
- Dynamic scoping
  - Reference to x is to SUB1's x
- Evaluation of Dynamic Scoping:
  - Advantage: convenience
  - Disadvantage: poor readability

Scope Example 1 (Perl)

- my vs local
  - ***my*** marks a variable as private in a lexical scope
  - ***local*** marks a variable as private in a dynamic scope

```
$x = 1;
sub foo { print "$x\n"; }
sub bar { local $x; $x = 2; foo(); }

&foo; # prints ???
&bar; # prints ???
&foo; # prints ???
```

Scope Example 2 (Perl)

```
$var = 5;
print $var, "\n";
&fun1;
print $var, "\n";

# subroutines
sub fun1 {
        local $var = 10;
        print $var, "\n";
        &fun2; # calling subroutine fun2
        print $var, "\n";
}

sub fun2 { $var ++; }
```

Scope and Lifetime

- Scope and lifetime are sometimes closely related, but are different concepts

- Consider a `static` variable in a C or C++ function

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

- In a dynamic-scoped language, the referencing environment is the local variables plus all visible variables in all active subprograms
  - A subprogram is active if its execution has begun but has not yet terminated

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
- The binding of values to named constants can be either static (called *manifest constants*) or dynamic
- Languages:
  - FORTRAN 90: constant-valued expressions
  - Ada, C++, and Java: expressions of any kind

Variable Initialization

- The binding of a variable to a value at the time it is bound to storage is called *initialization*
- Initialization is often done on the declaration statement, e.g., in Java

```
int sum = 0;
```

Summary

- Case sensitivity and the relationship of names to special words represent design issues of names
- Variables are characterized by the six-tuples: 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

Homework #3 (part 1)

- Programming Exercise

Write three functions in C or C++: one that declares a large array statically, one that declares the same large array on the stack, and one that creates the same large array from the heap. Call each of the subprograms a large number of times (at least 100,000) and output the time required by each. Explain the results.

- Problem Solving (P. 229 of class textbook)
  - 5, 8, 11, 12

- Due date: One week from assigned date
  - Please hand in printed (typed) form
    - **I do not accept any handwritten assignment**
    - **Exception: pictures**