Names, Bindings, Types and Scopes

Matt Evett
Dept. Computer Science
Eastern Michigan Univ.
(adapted from Sebesta's slides)

Names (Identifiers)

- Design issues
  - **Maximum length?**
  - **Are connector characters allowed?**
  - **Are names case sensitive?**
  - **Are special words reserved words or keywords?**
- Length
  - **FORTRAN I: maximum 6**
  - **COBOL: maximum 30**
  - **FORTRAN 90 and ANSI C: maximum 31**
  - **Ada: no limit, and all are significant**
  - **C++: no limit, but implementors often impose one**
- Connectors
  - **Pascal, Modula-2, and FORTRAN 77 don't allow**
  - **Others do**

Identifier Case Sensitivity

- ***Disadvantage:* readability (names that look alike are different)**
- **worse in Modula-2 because predefined names are mixed case (e.g. WriteCard)**
- **C, C++, Java, and Modula-2 names are case sensitive**
- **The names in other languages are not**

Special Identifiers

- **Def: A *keyword* is a word that is special only in certain contexts**
  - **Example: Fortran's `REAL APPLE` vs. `REAL = 3.4`**
  - **Disadvantage: poor readability**
- **Def: A *reserved word* is a special word that cannot be used as a user-defined name**
  - C's `switch`, `case`, 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, and scope
  - Name - not all variables have them
  - Address - the memory address with which it is associated.
  - Value - the contents of the location with which the variable is associated

Addresses

- Abstract memory cell - the physical cell or collection of cells associated with a variable
  - **The *l-value* of a variable is its address**
  - **The *r-value* of a variable is its value**
- 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

- Creating aliases:
  - Pointers, reference variables, Pascal variant records, C and C++ unions, and FORTRAN EQUIVALENCE (and through parameters - discussed in Ch 8)
- Some of the original justifications for aliases are no longer valid; e.g. memory reuse in FORTRAN. Replace them with dynamic allocation

Variable Type & Value

- 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

Binding

- Def: A binding is an association, such as between an attribute and an entity, or between an operation and a symbol
- Def: *Binding time* is the time at which a binding takes place.

Binding Times

- Language design time--e.g., bind operator symbols to operations
- Language implementation time--e.g., bind floating point type to an internal representation
- Compile time--e.g., bind a variable to a type in C or Java
- Load time--e.g., bind a FORTRAN 77 variable to a memory cell (or a C static variable)
- Runtime--e.g., bind a nonstatic local variable to a memory cell

Types of Bindings

- Def: A binding is *static* if it occurs before run time and remains unchanged throughout program execution.
- Def: A binding is *dynamic* if it occurs during execution or can change during execution of the program.

Typing Variables

- Type Bindings
  - How is a type specified?
  - When does the binding take place?
    - If static, type may be specified by either an explicit or an implicit declaration
  - Def: An *explicit declaration* is a statement used for declaring the types of variables
  - Def: An *implicit declaration* is a default mechanism for specifying types of variables (at the their first appearance in program)

Example Typing

- FORTRAN, PL/I, BASIC, and Perl provide implicit declarations
  - Advantage: writability
  - Disadvantage: reliability (less trouble with Perl)
    - First char = $ for scalar, @ for array, etc.

Dynamic Type Binding

- Specified through an assignment statement
  - e.g. APL: LIST ⇐ 2 4 6 8 vs. LIST ⇐ 17.3
  - E.g. Lisp: (setq bob "hi") vs. (setq bob 3)
- Advantage: flexibility (generic program units)
- Disadvantages:
  - High cost (dynamic type checking and interpretation)
  - Type error detection by the compiler is difficult

Dynamic Binding via Inference

- Type Inferencing (e.g. ML, Miranda, and Haskell)
  - Rather than by assignment statement, types are determined from the context of the reference
  - E.g. ML: **fun** circ(r) = 3.1415 * r * r
  - E.g. ML: **fun** circ(r) = 10 * r * r

Storage Bindings

- Keeping track of binding of variables to their memory cells.
- Allocation - getting a cell from some pool of available cells
- Deallocation - putting a cell back into the pool
- Def: The *lifetime* of a variable is the time during which it is bound to a particular memory cell

Categories of Variables

- To speak of storage bindings, it is useful to categorize variables by their lifetimes:
  - Inefficient, because all attributes are dynamic
  - Loss of error detection
  - Static
  - Stack-dynamic
  - Explicit heap-dynamic
  - Implicit heap-dynamic

Static Variables

- 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, global variables
- Advantage: efficiency (direct addressing), history-sensitive subprogram support
- Disadvantage: lack of flexibility (no recursion)

Stack-Dynamic Variables

- Storage bindings are created for vars when their declaration statements are elaborated.
  - If scalar, all attributes except address are statically bound
  - e.g. local variables in Pascal and C
- Advantage: allows recursion; conserves storage
- Disadvantages:
  - Overhead of allocation and deallocation
  - Subprograms cannot be history sensitive
  - Inefficient references (indirect addressing)

Explicit Heap-Dynamic Variables

- 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

Implicit Heap-Dynamic Variables

- Allocation and deallocation caused by assignment statements. I.e., when a variable is assigned a value, its cell (and all attributes) are allocated
  - e.g. all variables in APL
- Advantage: flexibility
- Disadvantages:

Type Checking

- Generalize the concept of operands and operators to include subprograms and assignments

Def: *Type checking* is the activity of ensuring that the operands of an operator are of compatible types

Def: 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*.

Type Errors

- Def: A type error is the application of an operator to an operand of an inappropriate type
  - If all type bindings are static, nearly all type checking can be static
  - If type bindings are dynamic, type checking must be dynamic
- Def: A programming language is *strongly typed* if type errors are always detected

Strong Typing

- Advantage: allows the detection of the misuses of variables that result in type errors
- Languages:
  - FORTRAN 77 is not: parameters, EQUIVALENCE
  - Pascal is not: variant records
  - Modula-2 is not: variant records, WORD type
  - 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)
- Coercion rules can strongly weaken strong typing (C++ vs Ada)

Dynamic Type Binding

- Advantage of dynamic type binding: programming flexibility
- Disadvantages:
  - efficiency
  - late error detection (costs more)
- Ex: Lisp

Type Compatibility

- Def: *Type compatibility by name* 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
    - If function parameters are to be a structure type, T, that type must be declared in one, global location. Can't be declared in both formal and actual parameter lists (e.g. Pascal)

Compatibility by Structure

- Def: *Type compatibility by structure* means that two variables have compatible types if their types have identical structures
  - More flexible, but harder to implement

Problems with Structured Types

- Consider the problem of two structured types:
  - Suppose they are circularly defined
  - 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 [-5..4])
  - Are two enumeration types compatible if their components are spelled differently?

More Problems

- With structural type compatibility, you cannot differentiate between types of the same structure (e.g. different units of speed, both float)
  - See Mars Polar Explorer disaster! Fall 1999.

Example Compatibility

- Language examples:
  - Pascal: usually structure, but in some cases name is used (formal parameters)
  - C: structure, except for records
  - C++: name
  - Ada: restricted form of name
    - Derived (sub-)types allow types with the same structure to be different.

```
– type celsius is new FLOAT;
  type fahrenheit is new FLOAT
```

    - Anonymous types are all unique, even in:

```
A, B : array (1..10) of INTEGER:
```

Scope

- Def: The *scope* of a variable is the range of statements over which it is visible.
- Def: 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

- … is based on program text; syntax
  - To connect a name reference to a variable, 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.*

Nested Scopes

- Variables can be hidden (*shadowed*) from a unit by having a "closer" variable with the same name.
  - I.e., identifier refers to the variable with that name in the nearest static ancestor scope.
  - C++, Lisp and Ada allow access to shadowed variables.
    - C++ uses scope operator "::". E.g: `::x` accesses the global variable, x, rather than the local variable x.

Creating static scopes

- Blocks - a method of creating static scopes inside program units--from ALGOL 60
- Examples:
  - C and C++: "{" and "}"

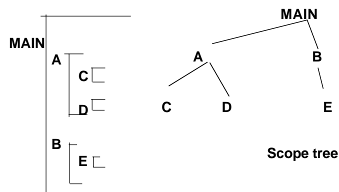
```
for (...) { int index; … }
```

  - Ada: "begin" and "end"

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

Evaluating Static Scopes

Consider the PASCAL-like example:
Assume MAIN calls A and B
A calls C and D
B calls A and E

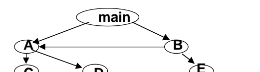


Scope tree

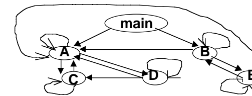
Lexical Program structure (A is def'd within MAIN, etc.)

Evaluating Static Scopes (2)

- Graph of desired potential callability



- Graph of actual potential callability
  - Danger!



Problems with Static Scoping

- 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 (hack to provide access)

Dynamic Scope

- Based on program unit calling sequences, not their textual layout
  - temporal versus spatial scope resolution
- References to variables are connected to declarations by searching back through the chain of subprogram calls that forced execution to this point.
  - Lisp provides dynamic scoping via *special* declarations

Example: Dynamic Scoping, Lisp

- The function FIND-BIGGEST takes a list of positive integers, and returns a dotted pair consisting of the biggest and second-biggest integers in the list. FIND-BIGGEST uses REDUCE in conjunction with another function, BIGGESTYET, and a global variable (boo!!).

```
(defvar second-biggest -1)  ; used in BIGGESTYET
(defun find-biggest (L)
   (setq second-biggest -1)
   (let ((result (reduce #'biggestYet L)))
     (cons result second-biggest)))
(defun biggestYet (a b)
   (let ((max (if (< a b) b a))
         (min (if (>= a b) a b)))
     (if (> min second-biggest)
       (setq second-biggest min))
     max))
USER(20): (find-biggest '(1 3 8 5 2 6 2))
   (8 . 6)
```

Example, (continued)

Now, we will use dynamic scoping (a SPECIAL variable) to solve the same problem without a global variable. In effect, secondB is like a "temporary" global variable, that exists only within the lifetime of FIND-BIGGEST.

```
(defun find-biggest (L)
 (let ((secondB -1))
  (declare (special secondB))
  (let ((result (reduce #'biggestYet L)))
    (cons result secondB))))
(defun biggestYet (a b)
  (let ((max (if (< a b) b a))
        (min (if (< a b) a b)))
    (if (> min secondB)
      (setq secondB min))
    max))
```

Imperative Example

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

Static scoping - reference to x is to MAIN's x

Dynamic scoping - reference to x is to SUB1's x

Evaluating Dynamic Scoping

- Evaluation of Dynamic Scoping:
  - Advantage: convenience
  - Disadvantage: poor readability
- Scope and lifetime are sometimes closely related, but are different concepts!!
  - Consider a static variable in a C or C++ function

Referencing Environments

- Def: The *referencing environment of a statement* is the collection of all names that are visible in the statement
  - In a static scoped language, that is the local variables plus all of the visible variables in all of the enclosing scopes
  - See book example (p. 184)
  - A subprogram is *active* if its execution has begun but has not yet terminated

Referencing Environments with Dynamic Scoping

- In a dynamic-scoped language, the referencing environment is the local variables plus all visible variables in all active subprograms
  - See book example (p. 185)

Named Constants

- Def: A *named constant* is a variable bound to a value only at time it is bound to storage
  - Advantages: readability and modifiability
- The binding of values to named constants can be either static (called manifest constants) or dynamic
- Languages:
  - Pascal: literals only
  - Modula-2 and FORTRAN 90: constant-valued expressions
  - Ada, C++, and Java: expressions of any kind

Variable Initialization

- Def: 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., Ada

    ```
    SUM : FLOAT := 0.0;
    ```
  - C++:

    ```
    int foo = 1;
    ```