Names and Binding

In Text: Chapter 4

1

Outline

- Names/Identifiers
- Binding
- Type Checking
- Scope

Names (User-defined 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 implementers often impose one
- Connector characters (e.g., _)
  - Pascal, Modula-2, and FORTRAN 77 don't allow
  - Others do

More Design Issues

- 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
  - Names in most other languages are not
- Special words
  - A **keyword** is a word that is special only in certain contexts
    - Disadvantage: poor readability
  - A **reserved word** is a special word that cannot be used as a user-defined name

Identifiers Have 6 Attributes

- A variable is an abstraction of a memory cell
- A (variable) identifier has 6 attributes:
  - Name
  - Address
  - Type
  - Representation/Value
  - Scope
  - Lifetime

6 Attributes (cont.)

1. Name
   - = identifier
   - can be one-one, many-one, or none-one mapping to memory
2. Address
   - point to a location in memory
   - may vary dynamically
   - Two names for same address = **aliasing**
3. Type
   - range of values + legal operations
   - variable, constant, label, pointer, program, ...

6 Attributes (cont.)

4. Representation/Value
   - interpreted contents of the location
     - l-value (address)
     - r-value (value)
5. Scope
   - Range of statements over which the variable is visible
   - Static/dynamic
6. Lifetime
   - Time during which the variable is bound to a storage location

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
- Possible binding times:
  1. Language design time--e.g., bind operator symbols to operations
  2. Language implementation time--e.g., bind floating point type to a representation
  3. Compile time--e.g., bind a variable to a type in C or Java
  4. Load time--e.g., bind a FORTRAN 77 variable (or a C static variable) to a memory cell
  5. Runtime--e.g., bind a nonstatic local variable to a memory cell

Static and Dynamic Binding

- A binding is **static** if it occurs before run time and remains unchanged throughout program execution
- A binding is **dynamic** if it occurs during execution or can change during execution of the program
- In many ways, binding times for various attributes determine the flavor of a language
- As binding time gets earlier:
  - efficiency goes up
  - safety goes up
  - flexibility goes down

Type Bindings

Two key issues in binding a type to an identifier:

1. How is a type specified?
2. When does the binding take place?

Static Type Binding

- If static, type may be specified by either an explicit or an 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

- Specified through an assignment statement (APL, Smalltalk, etc.)

```
LIST <- 2 4 6 8
LIST <- 17.3
```

- Advantage: flexibility (generic program units)
- Disadvantages:
  - Type error detection by the compiler is difficult
  - High cost (dynamic type checking and interpretation) or low safety
- Type Inferencing (ML, Miranda, and Haskell)
  - Rather than by assignment statement, types are determined from the context of the reference

Storage Bindings

- **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
- Stack-dynamic
- Explicit heap-dynamic
- Implicit heap-dynamic

Static Lifetime

- Bound to memory cells before execution begins and remains bound to the same memory cell(s) throughout execution
- Examples:
  - All FORTRAN 77 variables
  - C and C++ static variables
- Advantages:
  - Efficiency (direct addressing)
  - History-sensitive subprogram support
- Disadvantage:
  - Lack of flexibility (no recursion)

Stack-Dynamic Lifetime

- Storage bindings are created for variables when their declaration statements are elaborated
- If scalar, all attributes except address are statically bound
- Examples:
  - Local variables in Pascal and C subprograms
  - Locals in C++ methods
- Advantages:
  - Allows recursion
  - Conserves storage
- Disadvantages:
  - Overhead of allocation and deallocation
  - Subprograms cannot be history sensitive
  - Inefficient references (indirect addressing)

Explicit Heap-Dynamic Lifetime

- Allocated and deallocated by explicit directives, specified by the programmer, which take effect during execution
- Referenced only through pointers or references
- Examples:
  - Dynamic objects in C++ (via new and delete)
  - All objects in Java
- Advantage:
  - Provides for dynamic storage management
  - Explicit control
- Disadvantage:
  - Potential for human error

Implicit Heap-Dynamic Lifetime

- Allocation and deallocation is implicit, based on language semantics (e.g., caused by assignment statements)
- Ex.: all variables in APL
- Advantage:
  - Flexibility
- Disadvantages:
  - Inefficient, because often all attributes are dynamic
  - May have delay in 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
  - Allowed under language rules to be implicitly converted to a legal type by compiler-generated code
  - This automatic conversion is called a **coercion**

Type Errors

- 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
- A programming language is **strongly typed** if type errors are always detected
- In practice, languages fall on a continuum between strongly typed and untyped

Strong Typing

- Advantage of strong typing: 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 strongly affect strong typing—they can weaken it considerably (C++ versus Ada)

Type Compatibility: Name Equiv.

- **Type compatibility by name** ("name equivalence") 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)
- Predefined or user-supplied coercions can ease restrictions, but also create more potential for error

Type Compatibility: Structural Equiv.

- **Type compatibility by structure** ("structural equivalence") means that two variables have compatible types if their types have identical structures
- More flexible, but harder to implement
- Makes it more difficult to use a type checking to detect certain types of errors (e.g., preventing inconsistent unit usage in Ada)

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
- Scope and lifetime are sometimes closely related, but are different concepts!!
  - Consider a static variable in a C or C++ function

Static (Lexical) 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**

Variable Hiding

- Variables can be hidden from a unit by having a “closer” variable with the same name (closer == more immediate enclosing scope)
- C++ and Ada allow access to these “hidden” variables (using fully qualified names)
- Blocks are a method of creating static scopes inside program units—from ALGOL 60

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
- Effectively, searching “downward” through the call stack looking for an activation record possessing the declaration

Example

```
program foo;
   var x: integer;
   procedure f;
   begin
      print(x);
   end f;
   procedure g;
      var x: integer;
   begin
      x := 2;
      f;
   end g;
begin
   x := 1;
   g;
end foo.
```

What value is printed?

Evaluate with **static scoping**:
x = 1

Evaluate with **dynamic scoping**:
x = 2

Static vs. Dynamic Scoping

- Advantages of static scoping:
  - Readability
  - Locality of reasoning
  - Less run-time overhead
- Disadvantages:
  - Some loss of flexibility
- Advantages of dynamic scoping:
  - Some extra convenience
- Disadvantages:
  - Loss of readability
  - Unpredictable behavior (no locality of reasoning)
  - More run-time overhead

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, that is the local variables plus all of the visible variables in all of the enclosing scopes (see ex., p. 184)
- 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 (see ex., p. 185)

Variable Initialization

- The binding of a variable to a value at the time it is bound to storage is called **initialization**
- Often done on the declaration statement
- An Ada example:

```
sum : Float := 0.0;
```

- Can be static or dynamic (depending on when storage is bound)
- Typically once for static variables, once per allocation for non-static variables