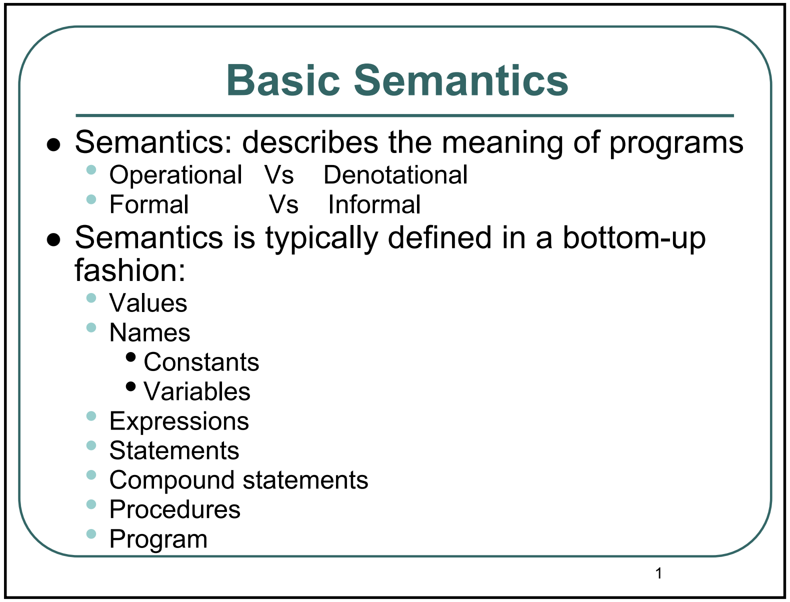
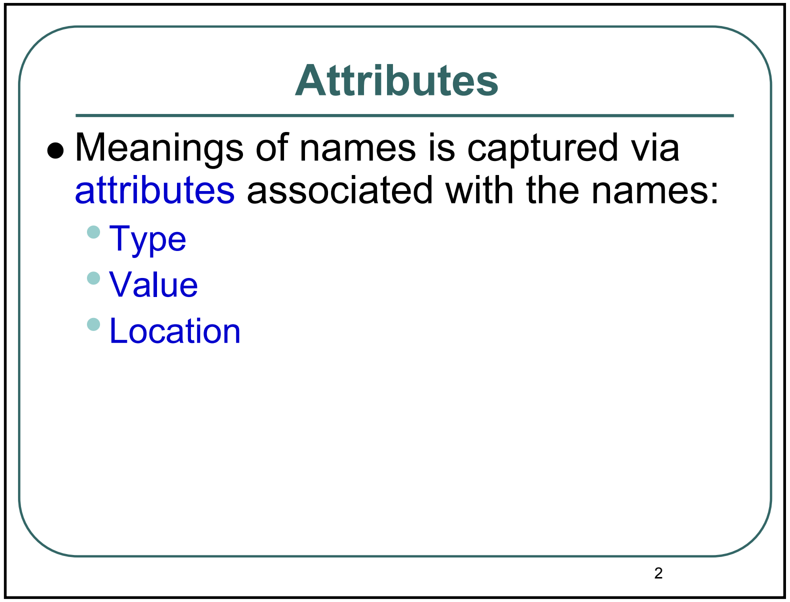


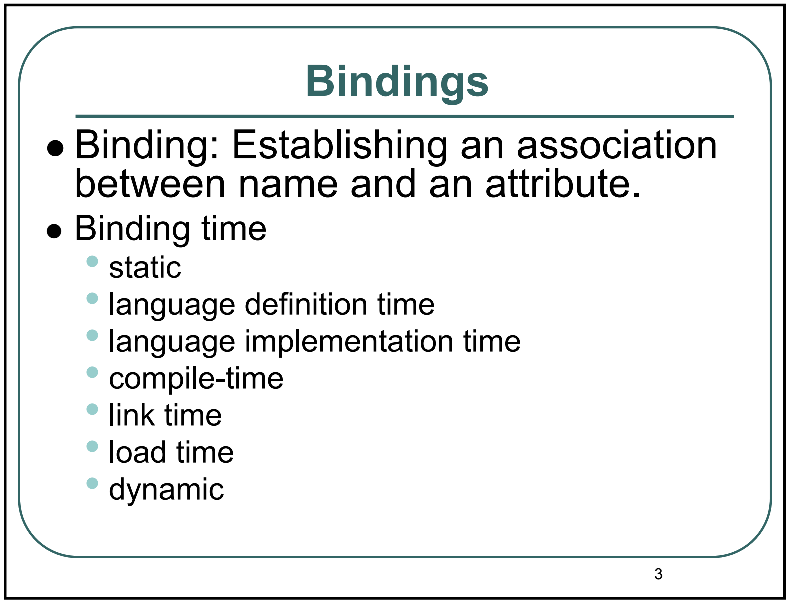
Basic Semantics

- Semantics: describes the meaning of programs
  - Operational Vs Denotational
  - Formal Vs Informal
- Semantics is typically defined in a bottom-up fashion:
  - Values
  - Names
    - Constants
    - Variables
  - Expressions
  - Statements
  - Compound statements
  - Procedures
  - Program



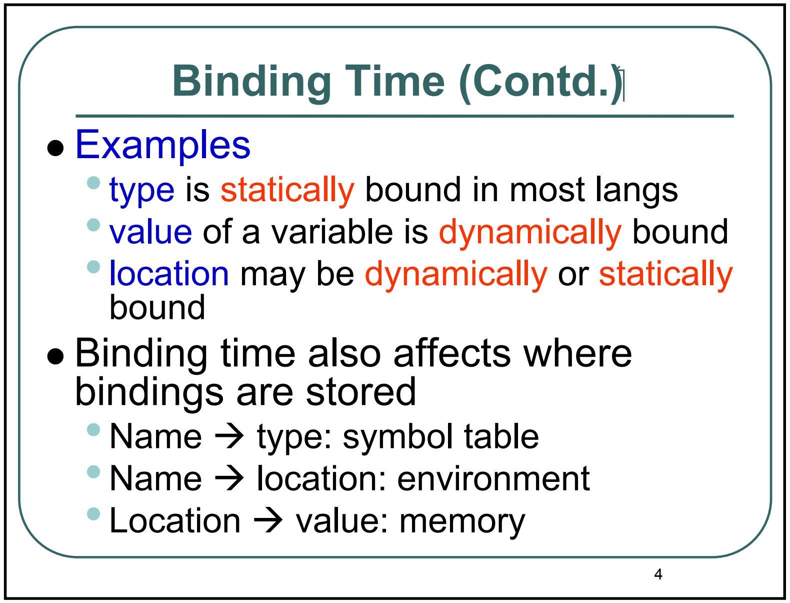
Attributes

- Meanings of names is captured via attributes associated with the names:
  - Type
  - Value
  - Location



Bindings

- Binding: Establishing an association between name and an attribute.
- Binding time
  - static
  - language definition time
  - language implementation time
  - compile-time
  - link time
  - load time
  - dynamic



Binding Time (Contd.)

- Examples
  - type is statically bound in most langs
  - value of a variable is dynamically bound
  - location may be dynamically or statically bound
- Binding time also affects where bindings are stored
  - Name → type: symbol table
  - Name → location: environment
  - Location → value: memory

Scopes

- Region of program over which a declaration is in effect
  - i.e. bindings are maintained
- Possible values
  - Global
  - Package or module
  - File
  - Class
  - Procedure
  - Block

Visibility

- Redefinitions in inner scopes supercede outer definitions
- Qualifiers may be needed to make otherwise invisible names to be visible in a scope.
- Examples
  - local variable superceding global variable
  - names in other packages.
  - private members in classes.

Symbol Table

- Uses data structures that allow efficient name lookup operations in the presence of scope changes.
- We can use
  - hash tables to lookup attributes for each name
  - a scope stack that keeps track of the current scope and its surrounding scopes
    - the top most element in the scope stack corresponds to the current scope
    - the bottommost element will correspond to the outermost scope.

Support for Scopes

lexical scopes can be supported using a scope stack as follows:

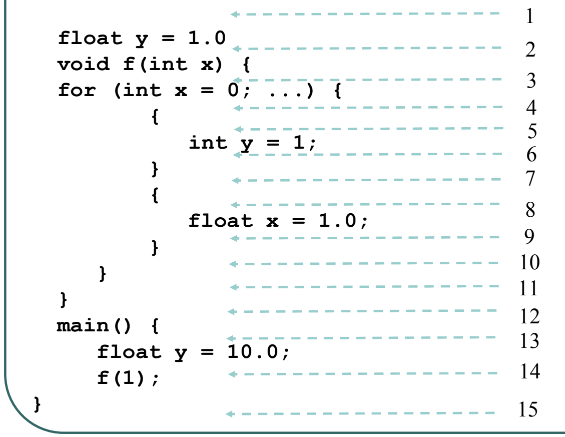
- Symbols in a program reside in multiple hash tables
  - In particular,symbols within each scope are contained in a single hash table for that scope
- At anytime, the scope stack keeps track of all the scopes surrounding that program point.

The elements of the stack contain pointers to the corresponding hash table.

Support for Scopes(contd.)

- To lookup a name
  - Start from the hash table pointed to by the top element of the stack.
  - If the symbol is not found, try hash table pointed by the next lower entry in the stack.
  - This process is repeated until we find the name, or we reach the bottom of the stack.
- Scope entry and exit operations modify the scope stack appropriately.
  - When a new scope is entered, a corresponding hash table is created. A pointer to this hash table is pushed onto the scope stack.
  - When we exit a scope, the top of the stack is popped off.

Example



```
float y = 1.0
void f(int x) {
for (int x = 0; ...) {
        {
           int y = 1;
        }
        {
           float x = 1.0;
        }
    }
}
main() {
   float y = 10.0;
   f(1);
}
```

illustration

- At (1)
  - We have a single hash table, which is the global hash table.
  - The scope stack contains exactly one entry, which points to this global hash table.
- When the compiler moves from (1) to (2)
  - The name y is added to the hash table for the current scope.
  - Since the top of scope stack points to the global table, "y" is being added to the global table.
- When the compiler moves from (2) to (3)
  - The name "f" is added to the global table, a new hash table for f's scope is created.
  - A pointer to f's table is pushed on the scope stack.
  - Then "x" is added to hash table for the current scope.

Static vs Dynamic Scoping

- Static or lexical scoping:
  - associations are determined at compile time
  - using a sequential processing of program
- Dynamic scoping:
  - associations are determined at runtime
  - processing of program statements follows the execution order of different statements

Example

- if we added a new function "g" to the above program as follows:

```
void g() {
    int y ;
    f(); }
```

- Consider references to the name "y" at (3).
  - With static scoping, it always refers to the global variable y defined between (1) and (2).
  - With dynamic scoping
    - if "f" is called from main, "y" will refer to the float variable declared in main.
    - If "f" is invoked from within g, the same name will refer to the integer variable "y" defined in g.

Example (Contd.)

- Since the type associated with "y" at (3) can differ depending upon the point of call, we cannot statically determine the type of "y" .
- Dynamic scoping does not fit well with static typing.
- Since static typing has now been accepted to be the right approach, almost all current languages (C/C++/Java/SML/LISP) use static scoping.

Some Coding Conventions

- Constant names use all upper case letters
- Type names are capitalized
- Variable and function names start with a lowercase letter
- Member variable names end with an '_' to make it easy to distinguish from local vars

Using "const" keyword

- Denotes that a variable does not change
- There may be member variables, local variables or global variables that never change, but this is unusual, so "const" key word is primarily used with function parameters
  - indicates that certain arguments to a function do not change within the function body
  - Member functions take an implicit object argument. If they don't change this argument, then use a "const" after closing parenthesis of declaration
    - `const SymTabEntry& symTab() const;`
  - Note that functions may be overloaded, so same function name could correspond to a const and non-const function
    - `SymTabEntry& symTab();`