Informal Semantics of Data

- semantic specification
- names (identifiers)
- attributes
- binding
- declarations
  - scope rules
  - visibility
Ways to Specify Semantics

• Standards Documents (Language Definition)
• Language Reference Manual e.g. Harbison and Steele
• Translator
• Formal Definition
  – denotational semantics
  – W-grammars (Algol 68)
  – attribute grammars
  – axiomatic semantics
    * Hoare logic
    * Dijkstra’s weakest precondition calculus
  – operational semantics
Names

- A *name* or *identifier* is an abstraction of a memory location.
- Names denote constants, variables, labels, procedure or function names, and formal parameters.
- A step in describing semantics is describing conventions that determine the meaning of each name in a program.
- Design issues for names: maximum length, character set, case sensitivity, reserved vs. keywords, hardware representation.

Attributes

- The meaning of a name is determined by *attributes* such as type, value, location, scope, lifetime, and initialization.
Bindings

- The association of a name with an attribute is a binding.
- These associations are called semantic functions.
  - in compiled languages, static attributes can be handled directly by the compiler by using Symbol Tables:
    ST: Names -> Static Attributes
  - in compiled languages, dynamic attributes are handled during execution by code generated by the compiler. These attributes include locations and values:
    Environment: Names -> Locations
    Memory: Names -> Values
  - in interpreted languages, the interpreter handles everything directly:
    Environment: Names -> Attributes
Binding Times

- **static** binding is done before run time and is unchanged during run time

- **dynamic** binding is done during run time or changes during run time

- language design
  - bind ‘*’ to multiplication

- language implementation
  - bind type integer to a hardware representation

- compile time
  - bind variable to a data type

- link/load time
  - bind function call to code

- run time
  - bind variable to address
Declarations

- Declarations bind attributes to names
  - int x;
    statically binds type integer to name x
  - int x = 0;
    also binds value 0 to name x

- may only bind some attributes
  - function prototype double f(int x);
    specifies type of f but not code
  - recursive definition struct x;
    is an incomplete type used to resolve recursive definitions

- may be implicit
  e.g. Fortran, APL, Snobol, Lisp

- A definition in C/C++ binds all potential attributes
• Declarations are associated syntactically and semantically with language constructs.
  
  – Standard construct is a block, schematically represented as

    begin-marker
decl-seq
stmt-seq
end-marker

  – Declarations associated with a block are local to the block. Others are non-local
    * C declarations appear in compound statements and function bodies
    * C declarations can also appear as globals

  – Declarations may also appear in structured data types and classes.

  – In pure object-oriented languages, all declarations except a class must be inside a class
• Declarations can be collected into larger groups
  – Ada packages and tasks
  – Java packages
  – ML, Haskell modules
  – C++ namespaces
Scope

- The *scope* of a binding is the region of the program over which the binding is maintained.
- Declarations create bindings.
- The *scope* of a declaration is the range of statements in which the bindings established by the declaration are maintained.
- The scope of a declaration includes the program unit in which the declaration is made, but may also include other parts.
• Static, Block, or Lexical Scoping
  – introduced in Algol 60
  – scope can be determined statically, that is, prior to execution
  – programs are typically constructed in nested blocks
  – each block creates its own scope
  – to determine the declaration associated with a variable reference, search outward through enclosing blocks

• Dynamic Scoping
  – Lisp (old versions), APL, SNOBOL
  – scope is determined dynamically, that is, during execution
  – to determine the declaration associated with a variable reference, search upward through call stack
Visibility

- The visibility of a declaration is the region of a program where bindings apply.
- Scope holes are regions where bindings exist but are "hidden" by other bindings.
- Scope and accessibility of declarations can be modified by:
  - scope resolution operators (Java, C++)
  - dot notation (Ada – visibility by selection)
  - keyword modifiers (C - extern)
Symbol Table

- binds names to static attributes
- supports insertion, lookup and deletion
- represents bindings given in declarations
- lexical scope
  - supported by stack operations
  - symbol table is processed by compiler, top to bottom through text
- dynamic scope
  - symbol table is processed at runtime as blocks or functions are executed
- see section 5.3 in text for examples