Chapter 5
Basic Semantics

A fundamental step in describing the semantic of a language is to describe the conventions that determine the meaning of each name used in the program.

Chapter Content
• Name Binding
  – Name Resolution and Overloading
• Type Binding
• Scope Binding
• Location Binding
  – Allocation Lifetime
  – Aliases, Dangling References, and Garbage
• Value Binding
  – Variables and Constants
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 time

Possible binding times:
1. Language design time--e.g., bind operator symbols to operations
2. Language implementation time--e.g., bind fl. pt. type to a representation
3. Compile time--e.g., bind a variable to a type in C or Java
4. Link time e.g., bind a the body of an externally defined function
5. Load time--e.g., bind a FORTRAN 77 variable to a memory cell (or a C static variable)
6. Runtime--e.g., bind a nonstatic local variable to a memory cell

**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.
Name Binding

Names

- *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
Name Binding

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 words

Def: A *keyword* is a word that is special only in certain contexts

- Disadvantage: poor readability

Def: A *reserved word* is a special word that cannot be used as a user-defined name
Type Binding

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.

Type Bindings
1. How is a type specified?
2. 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 program statement used for declaring the types of variables.
Def: 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).
Type Binding

Dynamic Type Binding
- Specified through an assignment statement
e.g. APL
  LIST <- 2 4 6 8
  LIST <- 17.3

Advantage: flexibility

Disadvantages:
1. High cost (dynamic type checking and interpretation)
2. Type error detection by the compiler is difficult
Scope Binding

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

Def: The *non-local* 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 Binding

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

Blocks - a method of creating static scopes inside program units--from ALGOL 60

Examples:

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

Ada:  
```ada
declare LCL : FLOAT;
begind
    ...
end
```
Scope Binding

Evaluation of Static Scoping

Consider the example:
Assume MAIN calls A and B
A calls C and D
B calls A and E
Scope Binding

Suppose the spec is changed so that D must now access some data in B

Solutions:

1. Put D in B (but then C can no longer call it and D cannot access A's variables)

2. 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 Binding

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
Scope Binding

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

- 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
Location Binding

A variable is an abstraction of a memory cell

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 or more can be used to access the same memory location, they are called aliases
  - Aliases are harmful to readability

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
Location Binding

Categories of variables by lifetimes

1. **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
   - *Advantage*: efficiency (direct addressing),
     - history-sensitive subprogram support
   - *Disadvantage*: lack of flexibility (no recursion)

2. **Stack-dynamic**--Storage bindings are created for variables when their declaration statements are elaborated.
   - If scalar, all attributes except address are statically bound
   - e.g. local variables in Pascal and C subprograms

   - *Advantage*: allows recursion; conserves storage
   - *Disadvantages:*
     - Overhead of allocation and deallocation
     - Subprograms cannot be history sensitive
     - Inefficient references (indirect addressing)
Location Binding

3. 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

4. Implicit heap-dynamic--Allocation and deallocation caused by assignment statements  
e.g. all variables in APL

*Advantage:* flexibility  
*Disadvantages:*  
- Inefficient, because all attributes are dynamic  
- Loss of error detection
Value Binding

**Value** - the contents of the location with which the variable is associated

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

Def: A *named constant* is a variable that is bound to a value only when 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
- **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;
```