Chapter 3

(3.1, 3.3)

Informal Semantics:
Names, Bindings and Scope

Informal Semantics

Semantics describes the *meaning* of a construct

Formal semantic definitions are precise but difficult to understand

Informal descriptions of semantics are common in reference manuals

Names

An abstraction mechanism

Semantics describe the meaning of a name
- Its attributes

Design Issues
- Maximum length
- What characters can be used
- Case sensitivity
- Special words

Names - Design Issues

Length
- Originally 1 character
- Fortran I – 77 → 6 characters
- COBOL → 30 characters
- Fortran 90 & ANSI C → 31 characters
- Ada & C++ → no limit
  But a compiler may limit significance

Most modern languages allow underscore (_)

Case sensitivity
- Prior to 1970 computers couldn't distinguish

Java camelcase example:
`ArrayIndexOutOfBoundsException`

Special words

Reserved Words
- Cannot be used as a user defined name

Keyword
- Context determines whether it is a special word

Predefined
- Language has a meaning, but can be changed by the programmer (`main` in C, **…**)

Variables

An association between a name and a memory location

Attributes include:
- Name – How it is referenced
- Address – Memory location (*l*-value)
- Value – Contents of memory location (*r*-value)
- Type – Range of acceptable values
- Scope – Where the name can be referenced
- Lifetime – Time period when memory is allocated

Variables Attributes

Type

Determines possible values of a variable and the set of operations that are defined that type; in the case of floating point, type also determines the precision

Value

The contents of the location with which the variable is associated

Abstract memory cell

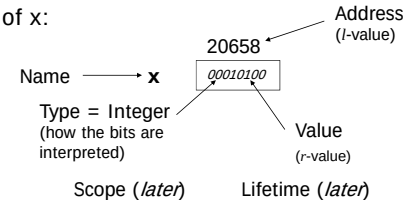
The physical cell or collection of cells associated with a variable

Variables (Pascal)

Example

```
var x : Integer;
x := 20;
x := x + 1;
```

Attributes of x:



Functions (C++)

Example

```
int f(int x) {
   return 2 * x;
}
```

Attributes of **f**:

Signature:
- Name: **f**
- Parameters: one **int** passed by value

Return type: **int**

Body of code

Binding

An association between an entity and an attribute

Examples:
- Value ↔ Memory cell
- Memory cell ↔ Name
- Name ↔ Type

Binding is a central concept in programming language semantics

Binding Times

Language definition time
- Reserved words, syntax rules, types, operator symbols

Language implementation time
- Size & bit pattern of floating point, maximum integer, runtime exception handling

Compile time
- Relative address & type of a variable, high-level constructs to machine code

Link/Load time
- Relative address for var's & subprograms in separate modules
- Absolute addresses of global variables

Runtime
- Values of var's, addresses for parameters & local vars

Binding Times

Static – Occurs before runtime and does not change during program execution

Dynamic – Occurs during program execution and may change according to language specific rules

Early binding supports efficient implementation & reliable code (compilers)

Late binding provides flexibility (interpreters)

Declarations

A method for establishing bindings

Attributes bound to names by declarations include: var, type, constant, function, etc.

Explicit declaration - a statement for declaring the types of variables

Implicit declaration – a default mechanism for specifying types of variables (the first appearance of the variable in the program)

Declarations have an attribute called *scope*…

Scope of a Declaration

Section of program text where the bindings established by a declaration are in effect

In *block structured* languages, scope of a declaration is limited to the block where it occurs

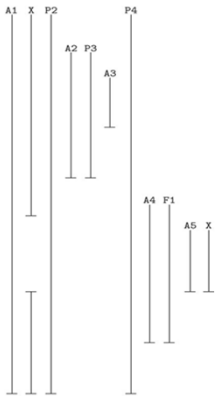
Declarations in nested blocks have precedence over earlier declarations

- This produces a "hole" in the scope of a declaration
- The binding exists, but is hidden from view
- Visibility – Region where a binding applies

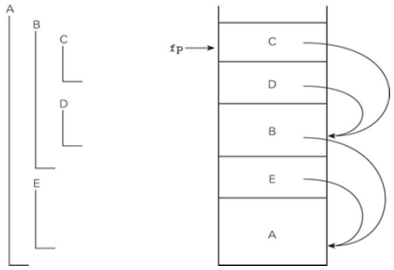
Note: Java does NOT use this (declaration in nested block cannot override a declaration *preceding* it)

Nested subroutines in Pascal

```
procedure P1(A1 : T1);
var X : real;
    ...
    procedure P2(A2 : T2);
        ...
        procedure P3(A3 : T3);
        ...
        begin
            ...        (* body of P3 *)
        end;
        ...
    begin
        ...            (* body of P2 *)
    end;
    ...
    procedure P4(A4 : T4);
        ...
        function F1(A5 : T5) : T6;
        var X : integer;
        ...
        begin
            ...        (* body of F1 *)
        end;
        ...
    begin
        ...            (* body of P4 *)
    end;
    ...
begin
    ...                (* body of P1 *)
end
```



Static Links for Non-local Access



Static & Dynamic Scoping

Static Scoping

- Bindings defined by the structure of a program
- Determined prior to program execution
- Most common scoping method

Dynamic Scoping

- Bindings determined at run time based on the calling sequence of subprograms
- Complex & rarely used
  - One important use - exception handling

Static Scope

To connect a name reference to a variable, find its declaration

- First look locally (same block)
- If not found try increasingly larger enclosing scopes

Variables can be "hidden" from a unit by having a "closer" variable with the same name

Dynamic Scoping

References to variables are connected to declarations by searching back through the chain of subprogram calls

- Temporal versus spatial

Static vs. Dynamic

```
program scopes (input, output );
var a : integer;
  procedure first;
  begin a := 1; end;
  procedure second;
     var a : integer;
  begin first; end;
begin
   a := 2; second; write(a);
end.
```

Static Scoping → prints 1

Dynamic Scoping → prints 2

Ruby Scope Weirdness

```
x = 5
for x in 1..4
   y = 2 + x
end
puts x, y
```

for loop doesn't create a local scope

```
[1,2,3].each do |x|
   y = x + 1
end
puts x, y
```

Error - x and y out of scope

```
x = nil
y = nil
[1,2,3].each do |x|
   y = x + 1
end
puts x, y
```

Works

Referencing Environment

The collection of all names that are visible to a statement

Static Scope

- Local variables + visible variables in all of the enclosing scopes

Dynamic Scope

- Local variables + visible variables in all active subprograms

Scoping with Namespaces

Used to logically arrange classes, etc. in groups

C# example declaration

```
namespace MyStuff {
   public class MyClass1 {
    …
   }
   public class MyClass2 {
    …
   }
}
```

Scoping with Namespaces

C# example client

```
class Client {
  static void main() {
    MyStuff.MyClass1 x =
               new MyStuff.MyClass1();
    MyStuff.MyClass2 x =
               new MyStuff.MyClass2();
  }
}
```

Scoping with Namespaces

C# example client

```
using MyStuff;
Class Client {
  static void main() {
    MyClass1 x = new MyClass1();
    MyClass2 x = new MyClass2();
  }
}
```

Scoping with Namespaces

Namespaces can be used to avoid names clashes

```
namespace Stuff1 {
   public class MyClass { }
}
namespace Stuff2 {
   public class MyClass { }
}
```

```
class Client {
  static void main() {
    Stuff1.MyClass x = new Stuff1.MyClass();
    Stuff2.MyClass y = new Stuff2.MyClass();
  }
}
```

Scoping with Namespaces

- Namespaces can be nested
- XML uses namespaces to avoid name clashes in documents
- Java imports are similar to namespaces, but are directly related to the hierarchy of the file system containing class files
- Namespaces are one of the most confusing concepts in modern programming languages