Bindings

Outline

- Preliminaries
- Scope
  - Block structure
  - Visibility
- Static *vs.* dynamic binding
- Declarations and definitions
- More about blocks
- The *Qualification Principle*

Bindings, Declarations, and Environments

- **Binding**: the programmer's ability to *bind* identifiers to entities
  - Constants
  - Variables
  - Procedures
  - Functions
  - Types
- **Declaration**: the operation of binding an identifier and an entity
- **Environment:** a set of bindings
  - Each expression and each command must be interpreted in a particular environment
  - All identifiers in an expression/command must have a binding in that environment
  - An environment is nothing but a partial *mapping* from identifiers to entities

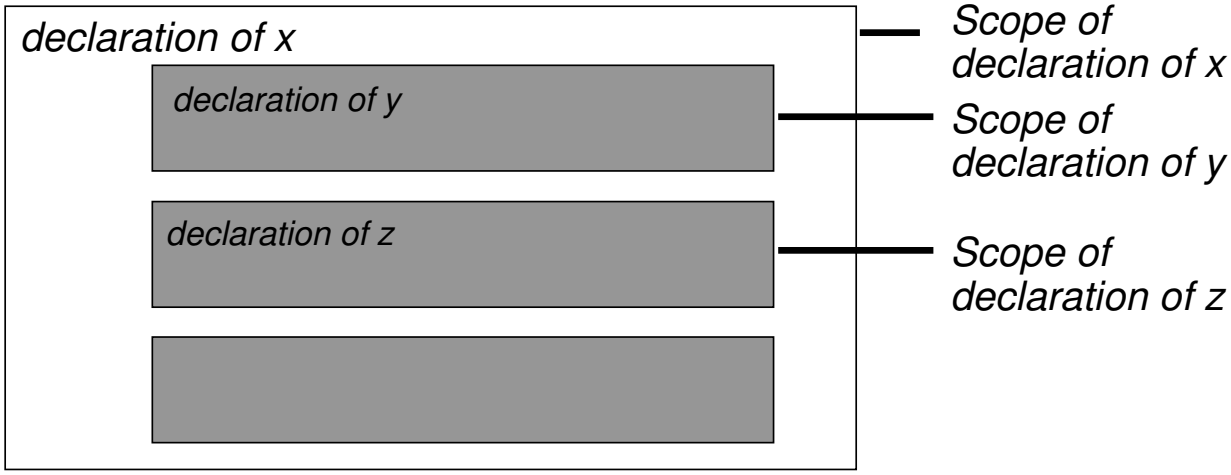
Scope

- **Scope:** the portion of the program text in which a declaration (a binding) is effective
  - Only in very primitive languages each declaration affects the environment of the whole program; in such languages, there is only one environment
  - **Block:** a program phrase that delimits the scope of any declarations that it may contain
- Scope relates to names bound to variables, functions, etc.; lifetime relates to *when* variables "exist"
- **Block structure:** a textual relationship between blocks
  - Can blocks be nested?

Monolithic Block Structure

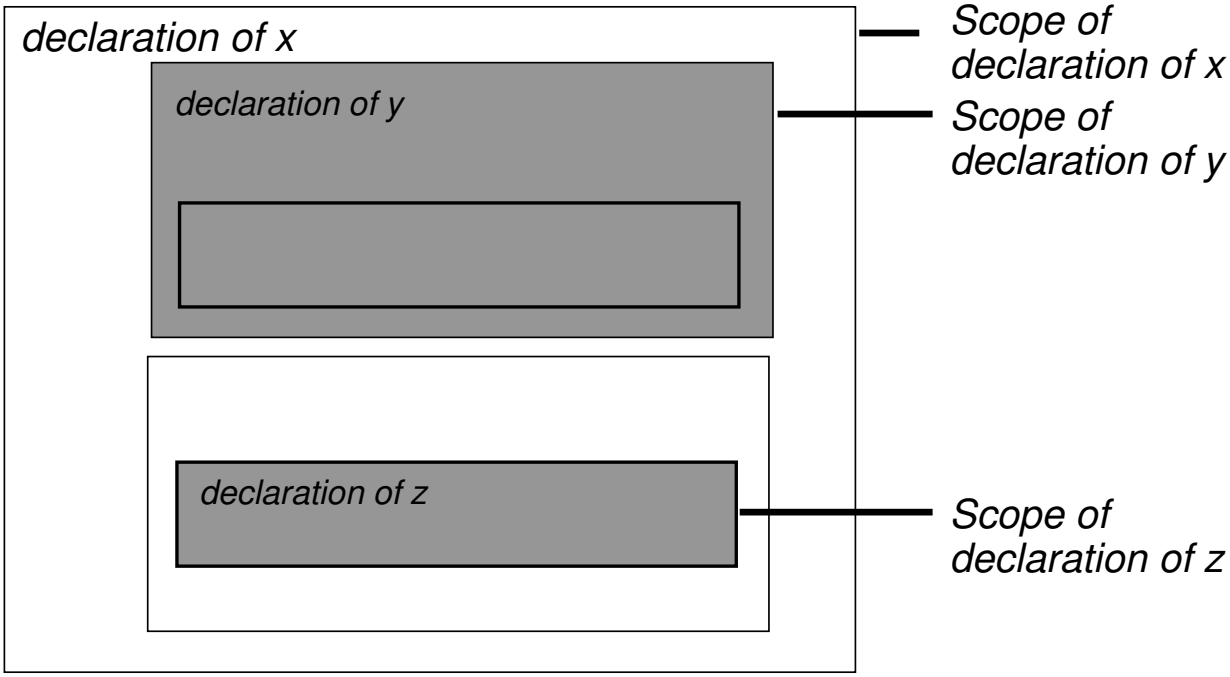
- The entire program is a single block
  - Older versions of *Cobol* and *Basic*
- Too crude, especially for large programs
  - All declarations are in one place
  - Awkward in teamwork

Flat Block Structure



- Program is partitioned into blocks (as in *Fortran*)
  - Block declarations are local to the block
  - Global declarations are truly global
- Disadvantages
  - All sub-programs must have distinct names
  - If it can't be local, then it must be global!

Nested Block Structure

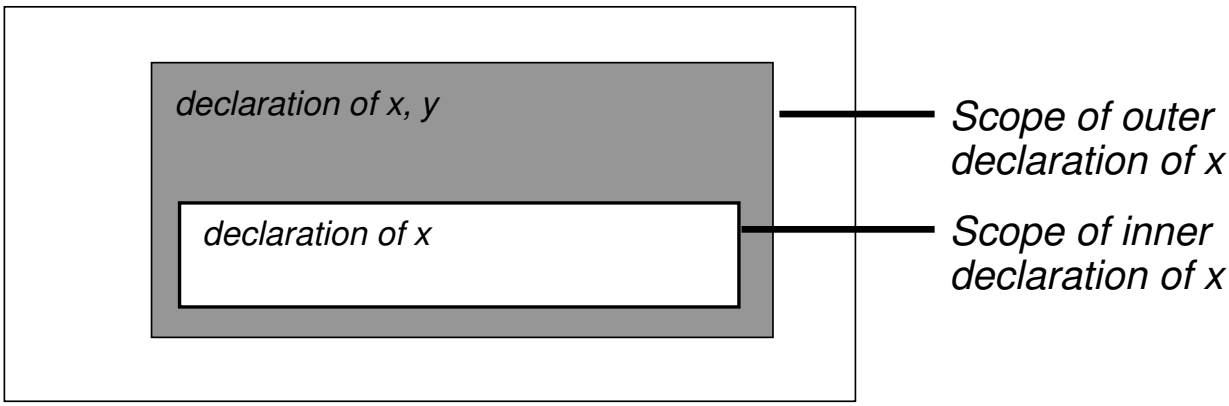


- *Algol*-like languages. Most popular.

Scope and Visibility

- It is possible to bind the same identifier to different entities in different blocks
- What happens if the same identifier is declared in two *nested* blocks?
- The answer (in the next foil) assumes:
  - Static binding
  - Scope of declaration is *the entire enclosing block*
  - No overloading or polymorphism

Hiding Declarations



If the scope of an identifier I encompasses a block *B*, then

- If *B* has no declaration of I, then all occurrences of I in *B* refer to the outer declaration (like y above)
- If *B* has a declaration of I, (like x) then the outer declaration is *hidden*/*invisible*
  - all *applied occurrences* (=uses) of I in *B* refer to the declaration in *B*
  - all *binding occurrences (=declarations)* of identifiers in *B,* prior to the declaration of I, refer to the outer declaration of I

Hiding and Overloading

- In any language that allows both overloading and nested block structure, there is an intriguing question:

  When does an internal declaration hide an external one, and when does it overload it?

- **Example:** C++
  - Nested classes
  - Classes defined in functions
  - Member functions defined in classes defined in classes
  - Overloading can occur at any level: raises tough questions
- **Overloading *vs.* Overriding:** in C++, a class can override a function defined in its base class
  - **What happens if the re-definition looks like overloading?**
    - **Tough question;** correct but inexplicable answer for C++: *if there is overloading then there is also hiding*

Static *vs.* Dynamic Binding

- How does an applied occurrence (a use of a name) associate with a binding (a declaration of that name)?
  - **Static binding** (done at *compile-time,* on program text):
    - Fortran, Algol, Pascal, Ada, C
    - Find the *smallest containing* block with a declaration
  - **Dynamic binding** (done at *run-time,* on history of execution):
    - Lisp, Smalltalk
    - Find the *most recent* declaration inside a currently active block
- Dynamic binding
  - Prevents static typing of variable names
  - Run-time type errors (even missing binding declarations)
  - Makes procedures and functions harder to understand

Static and Dynamic Binding

```
const  s = 2;

   function scaled (d: integer): integer;

   begin
         scaled := d * s;
   end;

   procedure P;
      const s = 3;
      begin
         ... scaled(5) ...
      end;

begin
   ... scaled(5) ...
end
```

Static binding: *Value is 10*
Dynamic binding: *Value is 15*

Value is 10

Declarations

- A *declaration* is a program phrase that will be *elaborated* to produce a binding
- Kinds of declarations:
  - Definitions
  - Collateral declarations
  - Sequential declarations
  - Recursive declarations
- Moreover
  - Type declaration: may create a new type
  - Variable declaration: may create a new variable

Definitions

- A *definition* is a declaration that produces nothing but bindings
  - Bind once, use many times
- Pascal *constant-definition*:
  - `const minint = - maxint; (*` This is as general as it can be! `*)`
  - But not

```
const letters = ['a'.. 'z', 'A' .. 'Z'];
      minchar = chr(0);
      halfpi = 0.5 * pi;
```

- Algol-68, Ada, ML, and C allow any value of any type to be bound
- ML's *value definition* **`val`** Id = E permits any expression, including one which cannot be evaluated at compile time

Collateral vs. Sequential Declarations

- **Collateral Declaration:** Composition of declarations where components are independent of each other. None can use an identifier bound in the other. Quite rare, but exists in ML.

```
val  Pi = 3.14159
and  sin = fn (x: real) => …
and  cos = fn (x: real) => …
```

- **Sequential Declaration:** A component may use identifiers bound in a previous component.
  - The following would *not* work:

```
val Pi = 3.14159 and TwoPi = 2 * Pi;
```

  - But this would:

```
val Pi  = 3.14159;
val TwoPi = 2 * Pi;
```


Recursive Declarations

A declaration that uses the very binding it produces

```
struct Node {
  int data;
  struct Node *next;
};
```

In Pascal, a sequence of type definitions (function and procedure definitions) is automatically recursive:

```
type
   IntList = ^ IntNode;
   IntNode = record
      head: Integer;
      tail: IntList;
   end;
```

However, constant definitions and variable declaration are always non-recursive

Forward Declarations

Forward is nothing but a compiler directive, which does not affect meaning:

```
procedure Statement; forward;

procedure BeginEnd;
begin
   ...
   Statement;
   ...
end;
procedure Statement;
begin
   ...
   BeginEnd;
   ...
end;
```

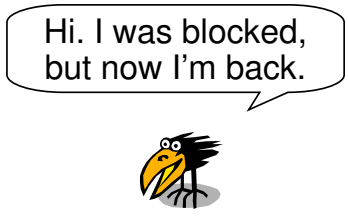
```
struct husband;

struct Wife {
   char *name;
   struct Husband *husband;
   ...
};
struct Husband {
   char *name;
   struct Wife *wife;
   ...
};
```


Blocks

Block: a program phrase delimiting the scope of the declarations in it.

- Block Commands
- Block Expressions
- The *Qualification Principle*



Block Commands

- A command containing declarations
  - The bindings of the declarations are used *only* for executing that command
  - The only net effect is to update variables
- **Pascal:** *D* `begin` *C* `end`;
  - *C* is executed in an environment in which the bindings produced by *D* override the binding of the outside environment
  - Can only occur as a program body or procedure body
  - Irregularity in the language:
    - Cannot place variables where they are needed!
- **C:** `{ D C }`
  - Semantics as in Pascal
  - Similar to Algol-60:

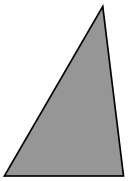
    `begin` *D; C* **`end`**;

```
for (i = 0; i < N-1; i++)
 for (j = i + 1; j < N; j++)
  if (a[i] > a[j]) {
    int temp = a[i];
    a[i] = a[j];
    a[j] = temp;
  }
```


Block Expressions

- An expression containing declarations
  - The declarations are used only for evaluating this expression
  - Net effect is to yield a value
- Pascal: none
  - Closest thing is a function body
- ML:
  - **How?** **`let`** ***D*** **`in`** ***E*** **`end`**
  - **Where?** anywhere an expression can occur
  - **What?** evaluate the sub-expression ***E*** in the outside environment overridden by the declarations in ***D***

```
let
   val
      s = (a + b + c) * 0.5
   in
      sqrt( s * (s-a) * (s-b) * (s-c) )
end
```



The Qualification Principle

- Summary:
  - **Block Command:** local declaration used in *executing* the command
  - **Block Expression:** local declaration used in *evaluating* the expression
- Generalization:

> It is possible to include a block in any syntactic class, provided that the phrases of that class specify some kind of computation.

Block Declaration

- A local declaration whose bindings are used only for *elaborating* the block declaration

```
local
  fun multiple(n: int, d: int) = (n mod d = 0)
in
  fun leap(y: int) = (multiple(y, 4)
    andalso not multiple(y, 100))
    orelse multiple(y, 400)
end
```

- Different from nested functions/procedures in e.g., Pascal:
  - No parameters to block declaration
  - Nested procedures are mostly concerned with code