Functional Languages

Lisp, Scheme and ML are *functional* in nature.

Programs are expressions to be evaluated.

Language design aims to *minimize* side-effects, including assignment.

Alternative evaluation mechanisms are possible, including

Lazy (Demand Driven)

Eager (Data Driven or Speculative)

Object-Oriented Languages

C++, Java, Smalltalk, Pizza and Python are object-oriented.

Data and functions are encapsulated into Objects.

Objects are active, have persistent state, and uniform interfaces (messages or methods).

Notions of inheritance and common interfaces are central.

All objects that provide the same interface are treated uniformly. In Java you can print any object that provides the **`toString`** method. Iteration through the elements of any object that implements the **`Enumeration`** interface is possible.

Subclassing allows to you extend or redefine part of an object's behavior without reprogramming all of the object's definition. Thus in Java, you can take a **`Hashtable`** class (which is fairly elaborate) and create a subclass in which an existing method (like **`toString`**) is redefined, or new operations are added.

Logic Programming Languages

Prolog notes that most programming languages address both the logic of a program (what is to be done) and its control flow (how you do what you want).

A logic programming language, like Prolog, lets programmers focus on a program's logic without concern for control issue.

These languages have no real control structures, and little notion of "flow of control."

What results are programs that are unusually succinct and focused.

Example:

```
inOrder( [] ).
inOrder( [ _ ] ).
inOrder([a,b|c]) :- (a<b),
   inOrder([b|c]).
```

This is a *complete*, *executable* function that determines if a list is in order. It is naturally polymorphic, and is not cluttered with declarations, variables or explicit loops.

Review of Concepts from Procedural Programming Languages

Declarations/Scope/Lifetime/ Binding

Static/Dynamic

- Identifiers are *declared*, either explicitly or implicitly (from context of first use).
- Declarations *bind* type and kind information to an identifier. Kind specifies the grouping of an identifier (variable, label, function, type name, etc.)
- Each identifier has a *scope* (or range) in a program—that part of the program in which the identifier is visible (i.e., may be used).

- Data objects have a *lifetime*—the span of time, during program execution, during which the object exists and may be used.
- Lifetimes of data objects are often tied to the scope of the identifier that denotes them. The objects are created when its identifier's scope is entered, and they may be deleted when the identifier's scope is exited. For example, memory for local variables within a function is usually allocated when the function is called (activated) and released when the call terminates. In Java, a method may be loaded into memory when the object it is a member of is first accessed.

Properties of an identifier (and the object it represents) may be set at

- **Compile-time**

  These are *static* properties as they do not change during execution. Examples include the type of a variable, the value of a constant, the initial value of a variable, or the body of a function.

- **Run-time**

  These are *dynamic* properties. Examples include the value of a variable, the lifetime of a heap object, the value of a function's parameter, the number of times a while loop iterates, etc.

Example:

In Fortran

- The scope of an identifier is the whole program or subprogram.
- Each identifier may be declared only once.
- Variable declarations may be implicit. (Using an identifier implicitly declares it as a variable.)
- The lifetime of data objects is the whole program.

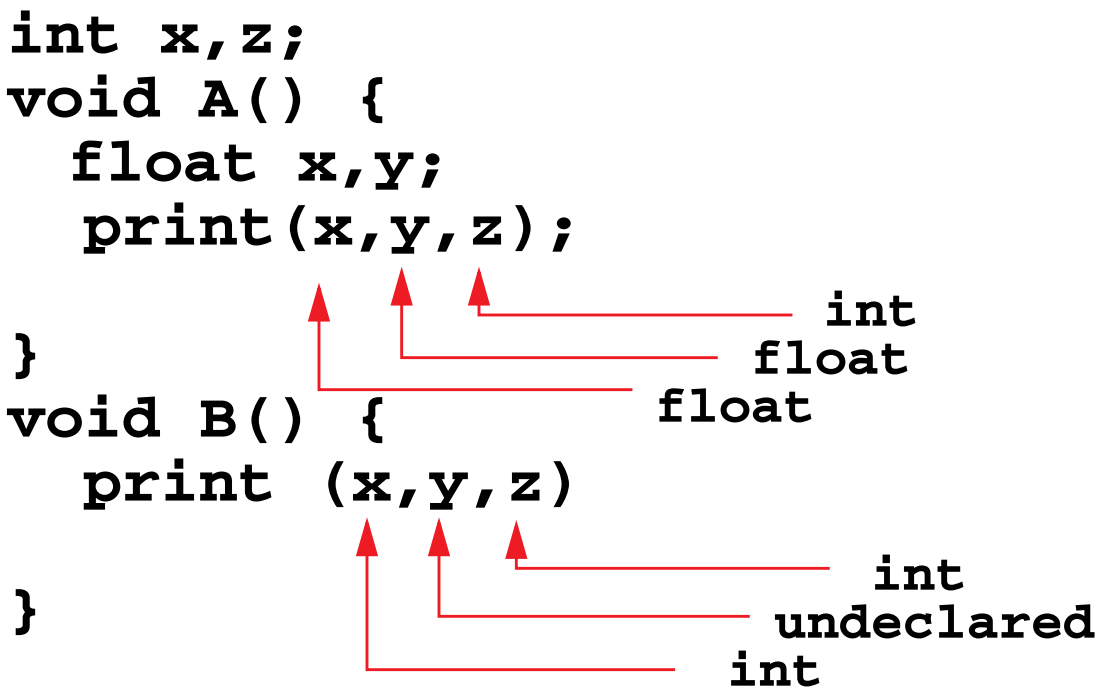
Block Structured Languages

- Include Algol 60, Pascal, C and Java.
- Identifiers may have a non-global scope. Declarations may be *local* to a class, subprogram or block.
- Scopes may *nest*, with declarations propagating to inner (contained) scopes.
- The lexically *nearest* declaration of an identifier is bound to uses of that identifier.

Binding of an identifier to its corresponding declaration is usually static (also called lexical), though dynamic binding is also possible.

Static binding is done prior to execution—at compile-time.

Example (drawn from C):



Block Structure Concepts

- Nested Visibility

  No access to identifiers outside their scope.

- Nearest Declaration Applies

  Static name scoping.

- Automatic Allocation and Deallocation of Locals

  Lifetime of data objects is bound to the scope of the Identifiers that denote them.

Variations in these rules of name scoping are possible.

For example, in Java, the lifetime of all class objects is from the time of their creation (via **`new`**) to the last visible reference to them.

Thus

```
... Object o;...
```

creates an *object reference* but does not allocate any memory space for `o`.

You need

```
... Object o = new Object(); ...
```

to actually create memory space for `o`.

Dynamic Scoping

An alternative to static scoping is *dynamic scoping*, which was used in early Lisp dialects (but not in Scheme, which is statically scoped).

Under dynamic scoping, identifiers are bound to the dynamically closest declaration of the identifier. Thus if an identifier is not locally declared, the call chain (sequence of callers) is examined to find a matching declaration.

Example:

```
int x;
void print() {
  write(x); }
main () {
  bool x;
  print();
}
```

Under static scoping the **x** written in **print** is the lexically closest declaration of **x**, which is as an **int**.

Under dynamic scoping, since **print** has no local declaration of **x**, **print**'s caller is examined. Since **main** calls **print**, and it has a declaration of **x** as a **bool**, that declaration is used.

Dynamic scoping makes type checking and variable access harder and more costly than static scoping. (Why?)

However, dynamic scoping does allow a notion of an "extended scope" in which declarations extend to subprograms called within that scope.

Though dynamic scoping may seen a bit bizarre, it is closely related to *virtual functions* used in C++ and Java.

Virtual Functions

A function declared in a class, C, may be redeclared in a class derived from C. Moreover, for uniformity of redeclaration, it is important that *all* calls, including those in methods within C, use the new declaration.

Example:

```
class C {
  void DoIt()(PrintIt();}
  void PrintIt()
     {println("C rules!");}
}
class D extends C {
  void PrintIt()
      {println("D rules!");}
  void TestIt() {DoIt();}
}
D dvar = new D();
dvar.TestIt();
D rules! is printed.
```