Lisp (List Processing Language)

Example:

```lisp
((lambda (x) (* x x)) 10)
```

Developed in the early 60s.

A radical departure from earlier programming languages.

Programs and data are represented in a uniform list format.

Types are a property of data values, not variables or parameters.

A program can build and run new functions as it executes.

Data values were not fixed in size.

Memory management was automatic.

A formal semantics was developed to define precisely what a program means.
Simula 67 (Simulation Algol)

Example:

Class Rectangle (Width, Height);
Real Width, Height;
Boolean Procedure IsSquare;
    IsSquare := Width=Height;
End of Rectangle;

Developed about 1967.

Introduced the notion of a class (for simulation purposes).

Included objects, a garbage collector, and notions of extending a class.

C++ was originally C with classes (as Simula was Algol with classes).
C and C++

C was developed in the early 70’s; C++ in the mid-80s.

These languages have a concise, expressive syntax; they generate high quality code sufficient for performance-critical applications.

C, along with Unix, provided the viability of platform-independent languages and applications.

C and C++ allow programmers a great deal of freedom in bending and breaking rules.

Raises the issue of whether one language can span both novice and expert programmers.
Interesting issue—if most statements and expressions are meaningful, can errors be readily detected?

```c
if (a==b)
    a=0;
else a = 1;
```
Java

Developed in the late 90s.
Cleaner object-oriented language than C++.
Introduced notions of dynamic loading of class definitions across the Web.
Much stronger emphasis on secure execution and detection of run-time errors.
Extended notions of platform independence to system independence.
What Drives Research into New Programming Languages?

Why isn’t C or C++ or C+++ enough?

1. Curiosity
   
   What other forms can a programming language take?
   
   What other notions of programming are possible?

2. Productivity
   
   Procedural languages, including C, C++ and Java, are very detailed.
   
   Many source lines imply significant development and maintenance expenses.
3. Reliability
   Too much low-level detail in programs greatly enhances the chance of minor errors. Even minor errors can raise significant problems in applications.

4. Security
   Computers are entrusted with ever-increasing responsibilities. How can we know that a program is safe and reliable enough to trust?

5. Execution speed
   Procedural languages are closely tied to the standard sequential model of instruction execution. We may need radically different programming models to fully exploit parallel and distributed computers.
Desirable Qualities in a Programming Language

Theoretically, all programming languages are equivalent (Why?)
If that is so, what properties are desirable in a programming language?
• It should be easy to use.

Programs should be easy to read and understand.

Programs should be simple to write, without unexpected pitfalls.

It should be orthogonal, providing only one way to do each step or computation.

Its notation should be natural for the application being programmed.

• The language should support abstraction.

You can’t anticipate all needed data structures and operations, so adding new definitions easily and efficiently should be allowed.
• The language should support testing, debugging and verification.

• The language should have a good development environment.
  
  Integrated editors, compilers, debuggers, and version control are a big plus.

• The language should be portable, spanning many platforms and operating systems.
• The language should be inexpensive to use:
  
  Execution should be fast.
  Memory needs should be modest.
  Translation should be fast and modular.
  Program creation and testing should be easy and cheap.
  Maintenance should not be unduly cumbersome.
  Components should be reusable.
Programming Paradigms

Programming languages naturally fall into a number of fundamental styles or paradigms.

Procedural Languages

Most of the widely-known and widely-used programming languages (C, Fortran, Pascal, Ada, etc.) are procedural in nature.

Programs execute statement by statement, reading and modifying a shared memory.

This programming style closely models conventional sequential processors linked to a random access memory (RAM).
Question:

Given

\[
\begin{align*}
    a &= a + 1; \\
    \text{if } (a > 10) & \quad b = 10; \\
    \text{else} & \quad b = 15; \\
    a &= a \times b;
\end{align*}
\]

Why can’t 5 processors each execute one line to make the program run 5 times faster?
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 can be treated uniformly. In Java you can print any object that provides the `toString` method. You can iterate through the elements of any Java object that implements the `Enumeration` interface.
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 the control flow of a program (how to 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:

\[
\begin{align*}
\text{inOrder}(\;[\;]\;). \\
\text{inOrder}(\;[\;\_\;]\;). \\
\text{inOrder}([a,b|c]) & :-(a<b), \\
& \quad \\text{inOrder}([b|c]).
\end{align*}
\]

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.