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. Minor errors can raise significant problems in applications.

4. Security

   Computers are entrusted with great 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 subtle pitfalls.

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

  Its notation should be natural for the application being programed.

- **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*.

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

```
 a = a + 1;
if (a > 10)
      b = 10;
else  b = 15;
a = a * b;
```

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 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.