Evolution of Programming Languages

In the beginning, ...

programs were written in absolute machine code—a sequence of bits that encode machine instructions.

Example:

```
34020005
0000000c
3c011001
ac220000
```

This form of programming is

- Very detailed
- Very tedious
- Very error-prone
- Very machine specific

Symbolic Assemblers

Allow use of symbols for operation codes and labels.

Example:

```
li $v0,5
syscall
sw $v0,a
```

Far more readable, but still very detailed, tedious and machine-specific.

Types are machine types.

Control structures are conditional branches.

Subprograms are blocks of code called via a “subroutine branch” instruction.

All labels are global.

Fortran (Formula Translator)

Example:

```
do 10 i=1,100
10 a(i)=0
```

Developed in the mid-50s.

A major step forward:

- Programming became more “problem oriented” and less “machine oriented.”
- Notions of control structures (ifs and do loops) were introduced.
- Subprograms, calls, and parameters were made available.
- Notions of machine independence were introduced.
- Has evolved into many new variants, including Fortran 77, Fortran 90 and HPF (High Performance Fortran).

Cobol (Common Business Oriented Language)

Example:

```
multiply i by 3 giving j.
mov j to k.
write line1 after advancing 1 lines.
```

Developed in the early 60s.

The first widely-standardized programming language.

Once dominant in the commercial world; still important.

Wordy in structure; designed for non-scientific users.

Raised the issue of who should program and how important readability and maintainability are.
Algol 60 (Algorithmic Language)

Example:

```algol
real procedure cheb(x,n);
value x,n;
real x; integer n;
cheb := if n = 0 then 1
      else if n = 1 then x
      else 2 × x ×
           cheb(x,n-1)−cheb(x,n-2);
```

Developed about 1960.
A direct precursor of Pascal, C, C++ and Java.
Introduced many ideas now in wide use:
- Blocks with local declarations and scopes.
- Nested declarations and control structures.
- Parameter passing
- Automatic recursion.
But,
- I/O wasn’t standardized.
- IBM promoted Fortran and PL/I.

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:

```algol
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?

```java
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.