# COS 360 Programming Languages

# Prof. Briggs Background IV : von Neumann, etc.

### Learning Objectives

1. To see how the durable von Neumann machine architecture has lead to the imperative programming paradigm domination of computer software development.
2. To briefly review some of the milestones in the history of programming languages.
3. To discuss some of the desirable features we look for in programming languages.

### The Influence of the von Neumann Architecture

The history of electronic computing machinery in the mid 20th century had a variety of contributors, and some of the contributions were obscured by the second world war(Zuse in Germany and Turing in England were both working for their governments). We are not concerned here with correctly assigning chronological priority to the sources of the ideas but rather to characterize some of the architectural features that have had a profound influence on computer programming languages.

The first electronic computers developed in the 1940's in the United States required that the machine wiring itself be modified to perform one calculation or another. John von Neumann, drawing on Turing's suggestion of a universal Turing machine, proposed a machine architecture wherein the machine had a set of instructions that could be stored in its memory. The hardware architecture would read the instructions from memory and carry them out. In this way the computing machine itself was universal in that it could(ignoring the memory bounds) realize any computable function as specified by the instructions of the stored program.

You are very likely familiar with computer organization from your earlier coursework, but we will review it briefly here. The von Neumann architecture divides the machine into three parts.

1. the CPU, which has the logic to fetch instructions from memory, decode them, and perform them. The CPU has a "Program Counter"(PC) which identifies the address of the next instruction to fetch and execute
2. the random access memory itself, where both the program instructions and data are stored; memory is connected to the CPU by a bus

3. i/o devices which enable the internal electronics of the computer to both receive input values and display results

Because program instructions are stored in read/write memory, it is possible for a program to modify its own instructions. There may be some applications that really call for this, but so called self-modifying code is so difficult to reason about that the practice is not recommended and rarely employed. Code that is not self-modifying is called "re-entrant" and can be shared by multiple applications.

There are a couple of observations to make about this architecture's connection to programming languages.

1. Because the machine is the translation target of the programming language, that is, the machine is the actual vehicle that carries out the computation, this architecture favors imperative languages with their execution model of a state (defined by the values of the variables and a position within the instruction list) transition sequence. That paradigm is more directly like the von Neumann model.

   John Backus, one of the principals involved in the development of Fortran, titled his Turing Award lecture "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs", which indicates the connection between the hardware architecture and the prevalent languages. Functional languages are higher level, more "declarative", which leads to more rapid implementation. There were some alternative data flow architectures proposed for functional languages, but the von Neumann machine has been remarkably durable.

2. Regarding stored programs as data contributes to the idea of operations that act on programs, such as assemblers, compilers, pretty printers, and symbolic debuggers. Even so, it is usually difficult to write a program in an imperative language that dynamically constructs other programs in the same language for immediate execution. It is usually much easier to dynamically construct a program of the same language and immediately execute it in a functional language.

## A Very Brief History of Programming Languages

We want to consider some of the trends in the history of programming languages Generally the movement has been away from the machine towards higher level conceptual modeling. I do not claim to base this on an exhaustive knowledge of computing history; it's more of a personal impression of some of the big ideas that led to features in programming languages. I am leaving out a lot of languages and in particular Prolog.

1. 1940's and early 1950's

   a. first electronic computers and von Neumann machines

   b. assembly languages introduce symbolic op codes and symbols for addresses, but op codes are still in 1-1 correspondence with the instruction set, so assembly language is still very tied to a specific machine architecture; of course, the various hardware companies produced machines with different architectures, and assembly language programs would not port from one machine to a different one

2. late 1950's

   a. first higher level languages, Fortran and Lisp
   b. Fortran(arithmetic expressions, control abstraction with loops and if statements and a Goto; second version included subroutines) is independent of a specific architecture
   c. McCarthy's Lisp is based on Church's Lambda calculus and is the first functional language to work on electronic computers

3. early to mid 1960's

   a. "Structured programming" revolution provoked by Dijkstra's "Goto considered harmful" letter to CACM; emphasis on fundamental control abstractions of sequencing, selection, and iteration;
   b. ALGOL(block structure, support for recursion); a much more thoughtfully designed language
   c. context free grammars and science of translation developed; Knuth's bottom up parsing algorithm replaces the top down recursive descent approach

4. late 1960's to mid 1970's

   a. early object-oriented languages Simula and Smalltalk; they emphasize modeling;
   b. ML(has an early theoretically sound generic type system)
   c. Scheme(cut back version of Lisp)
   d. C developed by Dennis Ritchie at Bell Labs; low level but efficient; heavily associated with the Unix operating system, most of which was written in C
   e. Niklaus Wirth designs Pascal and it replaces Fortran as the language taught in introductory courses at most universities
   f. efforts at formally defining language semantics
      i. Vienna Definition Language(operational)
      ii. Assertion Based(R. W. Floyd, C.A.R. Hoare)
      iii. Denotational Semantics(Dana Scott and Chris Strachey)

5. late 1970's and 1980's

    a. data abstraction, abstract data types, data encapsulation as illustrated by the languages Modula, CLU, and Ada
    b. more generic data type constructors for lists, sets, etcetera
    c. more development of object-oriented languages with subtyping and inheritance, as illustrated by languages C++ and Eiffel
    d. structural operational semantics introduced by Gordon Plotkin is an improvement on Vienna Definition Language
    e. functional language Miranda includes lazy evaluation

6. 1990's

    a. Java(includes garbage collection, relieving programmer of low level memory management)
    b. scripting languages(often not strongly typed, and so not suitable for large scale softward development, in my view)
    c. languages for combining different software components
    d. growth of the internet and the JavaScript language for executing in web browsers

7. since 2000(mostly from the Wikipedia article)

    a. more support for functional language features in imperative languages, in Java, for example
    b. more attention to concurrent and distributed constructs
    c. more integration of programming languages and databases, either the relational model or XML model
    d. who knows what is next?

There is a Wikipedia article at `en.wikipedia.org/wiki/History_of_programming_languages` with more details.

The trend is clearly towards more richly expressive and powerful languages. Compiler optimization techniques have developed so that compilers can generate fairly efficient code from high level constructs. Architectures have, of course, grown by leaps and bounds since the early days so that a computer you hold in your hand is way more powerful than the most powerful machine of the 1950's. The clock cycle is at the nanosecond level and the parallel use of cheap graphics processing units for machine learning algorithms appears be revolutionizing what can be accomplished mechanically.

Also, the trend has been towards using more theoretical results in the design and realization of programming languages. Type systems and compiler development have been most significantly impacted by theoretical studies, but research into semantics of programming languages and formal methods of software development have also had important effects.

**Desirable Features for Programming Languages**

The threshold of being "Turing complete", that is, being able to calculate the same functions of the natural numbers that can be computed by a Turing machine, is surprisingly easy to reach. Very simple arithmetic operations and an unbounded loop structure, and an unbounded amount of memory are all that is really needed. Evidently, the cellular automaton model of Conway's Game of Life is Turing complete

`stackoverflow.com/questions/394957`
`/why-can-conway-s-game-of-life-be-classified-as-a-universal-machine`

Ethan Standel indicated to me that HTML 5 is Turing complete, but I gather that is somewhat controversial.

To be a convenient vehicle for complex software development, however, there are many other characteristics a language should have. Our textbook mentions

1. Efficiency(of the language): can be translated into efficient machine code; details of this would take us into compilation and machine architecture, but we will mention the C pre- and postfix unary operators `++` and `--`, which translate into a single intruction with a side effect on some machine architectures, in particular on the machine in use at Bell Labs when C was designed there.

2. Efficiency(with respect to programmer time: easy to understand and use; the other attributes given below contribute to this; in particular, regularity reduces learning time, expressiveness reduces initial coding time and security reduces debugging time;

3. Expressiveness: has a rich collection of powerful operators and types

4. Regularity/Uniformity: constructs exhibit syntactic and semantic consistency in different contexts; comparable situations have comparable appearance; few special cases or exceptions to general rules

5. Generality: many constructs have multiple uses that are regular, thereby reducing the number of constructs to be learned

6. Orthogonality/Independence: constructs that refer to different aspects can be combined; an example of this would be the Java modifiers for visibility(public, protected, private) and the Java modifiers for class-wide versus instance specific(static), which can be combined in all combinations

7. Security/Reliability: the language has safeguards against errors, for example automatic array bounds checking or dynamic memory management; Java is a big winner here

8. Extensibility: typically this means the language supports procedural and data abstraction by allowing the programmer to define encapsulated data types with enforced protection and to define functions as invokable units;

These are high level properties in that the connection between the property and whether a programming language has it or not is not always crystal clear, and often mixed and debatable. Moreover, even the goals themselves are sometimes conflicting.

For example, to be efficient of a programmer's time a language should be very expressive and have a lot of constructs, but on the other hand it should also be easy to understand, which argues for simplicity.

The regularity area has to do with, among other things, there being some consistency in the way punctuation and operators are used in the language. The reduction in the number of special cases that have to be explained eases the cognitive burden of learning and applying a language.

In my view the advances in compilation and code optimization have more or less settled the issue of efficiency of the compiled code. It is still possible to write code that is inefficient, and of course some problems are inherently costly anyway. Unless a language includes constructs that are directly translated as searches over extremely large spaces(like the set of all tours in a TSP problem), this kind of efficiency seems to me to be less important than the programmer's time. Programmers should be aware of operations that are costly and avoid them. In Java one should avoid calling a method more than once if the same result will be returned, since it's unlikely any compiler optimizer would remove the call.

To support efficient use of programmer time, the language should have constructs that are easy to grasp and automate tasks that are typically needed. An example is Java's iterator loop construct.

```
for (T x: exp){
 ...
}
```

where `T` is some type and `exp` is a subtype of `Iterable<T>`. This is general, easy to code, and useful in many contexts.

Syntax should be easy but plain; structure should be free format(Sorry, Python!) to allow for different pretty printing styles.

Very simple syntactic changes can make a huge difference. For example, explicit reserved word boundaries for tests and governed statements in loops and selections would save a lot of mistaken code.

```
if test then
   statement list
endif
```

```
while test do
   statement list
endwhile
```

are improvements over what C, C++, and Java offer, because they avoid mistakes. Some have proposed fi and od for the end markers, mimicing ( and ). The syntax of C, C++, and Java requires less typing, to be sure, but is not as clear. Modest changes like these would make it very difficult to mistake the boundaries of the governed statements, which can happen with (assume Java)

```
if (test)
   statement1;
   statement2;
   statement3;
```

The compiler will not report that as an error, and a human reader might imagine all three statements are controlled. The C designers chose a syntax to reduce the amount of typing.

There are a number of syntactic design mistakes that we are stuck with. Using = for destructive assignment was unfortunate, but goes back to Fortran and was continued in C and its descendants. Unfortunately, = already had a mathematical meaning, and since assignment is not the same as that earlier meaning, = should have been reserved for its more conventional meaning and a new symbolization used for assignment.

We represent a variety of sorts of information, so we need a rich type systems that includes numbers(distinguishing floating point types from whole number types seems a concession most languages make to the hardware, which treats them differently), strings, and the capability of defining additional types, which can be encapsulated so the object/value of those types can only be manipulated in prescribed ways.

Strong typing is critical for a language used for large software. One wants as much machine support for debugging as one can get. JavaScript and Python are poor in that regard, as is C. Strong typing is also important for security.

The object oriented subtyping, inheritance and dynamic dispatch features are why these languages have gained so much popularity. They really do support code reuse and maintenance. These build on the data abstraction and data encapsulation capacities that were recognized in the late '60's and through the 1970's, the years of Ada, Modula, CLU, etc..

Java is much more secure than C++ and C, which are practically disqualified by the gaping holes in their arrays(including character strings) and their typing systems. The lint tool might be able to help with this issue, but it's better to have the language itself enforce array bounds checking, pointer dereferencing, etc.. It is, of course, possible to write secure programs in these languages, but doing so puts a heavy burden on the programmer.

The fact is, programmers make a lot of mistakes. Not you and me, of course, but all those **OTHER** programmers make a lot of mistakes. The more you can use library code that has been written by experts for garbage collection, array index checking, binary search, etc., the less likely the code is to have hidden little bugs because some overworked programmer working under horrific time pressure implemented something incorrectly.

GOTO's should be done away with in favor of more restrictive branches like break, continue, return, and throwing exceptions. These constructs are harder to reason about than the basic three structures of sequence, selection, and iteration, but are not prone to the abuses that GOTO's are prone to, and provide adequate support for the sorts of things die-hard GOTOers say require the construct.

The ban on GOTO's is part of the general issue that programming language semantics should be clear and accessible. We need to be able to reason about what the code is going to do. The biggest problem for programmers is passing from the static code as it sits on the page to the dynamic behavior of the executing program, and language constructs that are difficult to reason about(like dynamically scoped variables as opposed to statically scoped ones) should not be included. Such constructs are like live wires in the language: dangerous to touch.

Direct support for modularity and abstraction, is of course absolutely a requirement, and has been known to be important for decades. You cannot build large software systems without breaking them into pieces. Included in this is separately compilable system units. Most modern languages have constructs to support procedural and data abstraction. Generic type constructors are part of this trend. Data encapsulation and careful attention to representation invariants can ensure that no ADT object every is in an invalid state.

There have been some advances in automated reasoning about code behavior, so-called "formal methods", and these results should be used whenever possible. Inclusion of assertion checking is a good idea, if only to encourage programmers to think in terms of class invariants, loop invariants, and properties that should be true at specific control points.

The trend has been to provide more powerful constructs that are then translated into code at a lower level. We see this also in database query languages. The SQL language owes more to set theory and logic than it does to the von Neumann architecture. I expect this to continue, and for perhaps some special purpose languages that are geared for algorithms of a certain sort (genetic algorithms for example?) to be developed to make it convenient for programmers working in the paradigm to express their intentions.

We will be looking at ML later, which is higher level than the imperative languages you have studied.