12

HOW DID WE GET HERE? WHERE DO WE GO FROM HERE?

Why is the chapter on history at the end of the text? Shouldn't it be first? Not necessarily. The purpose of this chapter is to consider "the past as prologue" and "possible histories from this point on."

It would be tempting to think that the current state of the art in programming languages is the pinnacle of development. It would be tempting to believe that the object paradigm is the last improvement in programming language design. Technology has a way of making a mockery of such thoughts. The purpose of this chapter is to consider "What's past is prologue" (William Shakespeare, *The Tempest*).

This chapter models how historical studies are conducted. You might call this the "follow the string" approach. I will choose some ends of the string by making observations about the state of programming languages and follow these strings backward. If we were going to write a paper on the subject, we would then rewrite the timeline forward. There is a danger in this forward exposition because the reader never quite knows where the author is going or why a certain point might be germane.

12.1 LOOKING BEHIND

In order to make sense of the roots of computing (which is, after all, the reason for programming) we start with the first programming language and work backward. We start with the question, "Why was FORTRAN even invented?" We can actually trace it back to the Hindus in 2000 BCE.

In choosing historical material, it is tempting to take 1954 as the beginning of the Universe; November 1954 is given as the date of the introduction of FORTRAN. However, this makes little sense, as FORTRAN, which

stands for *Formula Translator*, did not spring *de novo* from out of the ether. FORTRAN was put in place to give scientists a mechanism to implement numerical routines on computers. From our investigation of linguistics, we can see that *pragmatics* was the driver of language development. The form of FORTRAN is not surprising to anyone who has studied numerical analysis texts of the 1950s and 1960s.

12.1.1 The Difference Engine

Tracking this "scientific programming" heritage backward, we next find Ada Byron. In the late 1970s, the programming language Ada was all the rage. The foundation of the pragmatics of Ada was that of embedded systems, such as guidance systems on missiles. The reason to bring Ada up is the historical note that Ada was named for Ada Byron, Countess of Lovelace, and the daughter of the poet Lord Byron. Lady Byron was a very complex person who teamed with Charles Babbage, who invented the Difference Engine. The purpose of the Difference Engine was to solve differential equations for commerce. Today's computers are legitimate heirs of the Difference Engine, which was successfully implemented at Manchester University in 1948 by the engineers F. C. Williams and Tom Kilburn.

12.1.2 The Golden Age of Mathematics

Some of the issues that confronted Babbage and Lovelace were first explored by Isaac Newton and his contemporaries. There are so many famous contemporaries of Newton that it is impossible to list them all. For our search, though, we are interested in two: Colin Maclaurin and Brook Taylor. Newton, Maclaurin, and Taylor developed the fundamental principles for deriving polynomials for computing functions. The simple realization that infinitely presented numbers (π, for instance) must be finitely presented requires us to redevelop continuous mathematics with discrete mathematics. In particular, the computation world requires algebraic, not analytic (calculus), representations. Another important link at this juncture is the Bernoulli family.

Viète began the use of symbols in his book *In artem analyticam isagoge* in 1591.*

12.1.3 Arab Mathematics

The history of algebra is what we want to track, apparently. Most computer scientists should know that the word "algorithm" is of Arabic origin; in fact, so is the word "algebra." *Algebra* comes from the title of

* http://www-groups.dcs.st-and.ac.uk/ history/Mathematicians/Viete.html

a book on the subject, *Hisab al-jabr w'al muqabala*,* written about 830 by the astronomer/mathematician Mohammed ibn-Musa al-Khowarizmi. *Algorithm* is a corruption of al-Khowarizmi's name. However, the Arabian study of algebra came late, following the Hindus, Greeks, and Egyptians. The most advanced studies of algebra in antiquity were carried out by the Hindus and Babylonians.

12.1.4 Hindu Number Systems and Algebra

Although the Hindu civilization dates back to at least 2000 BCE, their record in mathematics dates from about 800 BCE. Hindu mathematics became significant only after it was influenced by the Greeks. Most Hindu mathematics was motivated by astronomy and astrology. Hindu mathematics is credited with inventing the base ten number system; positional notation system was standard by 600 CE. They treated zero as a number and discussed operations involving this number.

The Hindus introduced negative numbers to represent debts. The first known use is by Brahmagupta about 628. Bhaskara (b. 1114) recognized that a positive number has two square roots. The Hindus also developed correct procedures for operating with irrational numbers.

They made progress in algebra as well as arithmetic. They developed some symbolism that, though not extensive, was enough to classify Hindu algebra as almost symbolic and certainly more so than the syncopated algebra of Diophantus. Only the steps in the solutions of problems were stated; no reasons or proofs accompanied them.

The Hindus recognized that quadratic equations have two roots, and included negative as well as irrational roots. They could not, however, solve all quadratics because they did not recognize square roots of negative numbers as numbers. In indeterminate equations the Hindus advanced beyond Diophantus. Aryabhata (b. 476) obtained whole number solutions to $ax \pm by = c$ by a method equivalent to the modern method. They also considered indeterminate quadratic equations.

The Greek mathematician Diophantus (200 CE? 284 AD?) represents the epitome of a long line of Greek mathematicians (Archimedes, Apollonius, Ptolemy, Heron, Nichomachus) away from geometrical algebra to a treatment that did not depend upon geometry either for motivation or to bolster its logic. He introduced the syncopated style of writing equations, although, as we will mention below, the rhetorical style remained in common use for many more centuries.

Diophantus' claim to fame rests on his *Arithmetica*, in which he gives a treatment of indeterminate equations—usually two or more equations

* Often translated as *Restoring and Simplification* or *Transposition and Cancellation*.

in several variables that have an infinite number of rational solutions. Such equations are known today as "Diophantine equations." He had no general methods. Each of the 189 problems in the *Arithmetica* is solved by a different method. He accepted only positive rational roots and ignored all others. When a quadratic equation had two positive rational roots he gave only one as the solution. There was no deductive structure to his work.

12.1.5 Greek Geometrical Algebra

The Greeks of the classical period, who did not recognize the existence of irrational numbers, avoided the problem thus created by representing quantities as geometrical magnitudes. Various algebraic identities and constructions equivalent to the solution of quadratic equations were expressed and proven in geometric form. In content there was little beyond what the Babylonians had done, and because of its form geometrical algebra was of little practical value. This approach retarded progress in algebra for several centuries. The significant achievement was in applying deductive reasoning and describing general procedures.

12.1.6 Egyptian Algebra

Much of our knowledge of ancient Egyptian mathematics is based on the Rhind Papyrus. This was probably written about 1650 BCE and is thought to represent the state of Egyptian mathematics of about 1850 BCE. Egyptian mathematicians could solve problems equivalent to a linear equation in one unknown. Their method was what is now called the "method of false position," which is also a modern numerical algebra technique. Their algebra was rhetorical, that is, it used no symbols; problems were stated and solved in full language.

The Cairo Papyrus of about 300 BCE indicates that by this time the Egyptians could solve some problems equivalent to a system of two second-degree equations in two unknowns. Egyptian algebra was undoubtedly retarded by their cumbersome method of handling fractions. Egyptian fractions are a favorite programming project in beginning programming projects and data structures classes.

12.1.7 Babylonian Algebra

The mathematics of the Old Babylonian Period (1800–1600 BCE) had a sexagesimal (base 60) system of notation that led to a highly developed algebra system. The Babylonians had worked out the equivalent of our quadratic formula. They also dealt with the equivalent of systems of two equations in two unknowns. They considered some problems involving more than two unknowns and a few equivalent to solving equations of higher degree.

Like the Egyptians, their algebra was essentially rhetorical, meaning that there were few symbols and that problems and procedures were described in natural language. The procedures used to solve problems were taught through examples and no reasons or explanations were given. Also like the Egyptians they recognized only positive rational numbers, although they did find approximate solutions to problems which had no exact rational solution.

12.1.8 What We Can Learn

The very nature of algebra and algebraic notation changed over its 40-century history. From approximately 1800 BCE to around 200 BCE, algebra was primarily rhetorical, meaning that it was written in natural language. From 200 BCE to 1500 CE, algebra was syncopated, meaning that it was comprised of both natural language and symbols. The modern *symbolic form* appeared in the sixteenth century. The modern *abstract* version appeared only in the nineteenth century.

Algebra was driven by linguistically pragmatic issues. For example, the "lowly" quadratic formula appeared early but generalized polynomial notation arrived much later, requiring the symbolic version of algebra to appear.

The general theory of the algebra of real numbers began to appear in its modern form in the mid-1850s. Modern algebraic studies were greatly expanded by the use of algebraic methods with logic. Modern algebraic approaches use *category theory*, an approach that emphasizes functions over conditions. Category theory is heavily used in theoretical work in computer science.

12.2 THE ROLE OF THE λ-CALCULUS

We have called the definition of how a program language works its semantics. There have been three approaches to defining semantics: (1) operational, (2) denotational, and (3) axiomatic or connotational. Operational semantics approaches posit a hypothetical machine that is then used to describe the operation of a program. Denotational semantics is based on λ-calculus, and axiomatic semantics approaches are based on logic.

In 1963, John McCarthy (the "father" of Lisp) wrote an article that would change the way computation was perceived. Until McCarthy's article, computation, and by implication programming and programming languages, was tied to Turing machines. Even today, Turing machines (not λ-calculus) are the basis of algorithms research in the *NP*-complete investigations. McCarthy's article instead proposed λ-calculus as the foundation (McCarthy 1963).

In 1956, Noam Chomsky developed what is now called the *Chomsky Hierarchy*, which showed how Turing machines and grammars were connected (Chomsky 1956, 1959). In effect, Turing machines were shown to be equivalent to the so-called Type-0 Grammars. Chomsky's work tied together three major languages for computation: the λ-calculus, Turing machines, and grammars/languages. Turing machines and λ-calculus were already linked.

Turing machines were introduced by Turing in 1936 in a paper, "On Computable Numbers, with an Application to the *Entscheidungsproblem*." In the paper, Turing tried to captured a notion of algorithm with the introduction of Turing machines. The *Entscheidungsproblem* is the German name for Diophantine problems. Turing showed that the general Diophantine equations could not be solved by algorithms.

A few months earlier Alonzo Church had proven a similar result in "A Note on the Entscheidungsproblem" but he used the notions of recursive functions and lambda-definable functions to formally describe what the logicians call *effective computability* (Church 1936). Lambda-definable functions were introduced by Alonzo Church and Stephen Kleene (1936), and recursive functions by Kurt Gödel and Jacques Herbrand. These two formalisms describe the same set of functions, as was shown in the case of functions of positive integers by Church and Kleene (Church 1936, Kleene 1936).

The upshot of Church, Gödel, Kleene, and Turing was that although there were many different languages to describe algorithms, they were equivalent, in the sense that any computable function coded in one language could be coded in any of the other languages.

It is important to note that the issue of computation in the 1920s and 1930s was an issue in logic, not computing as we know it today. The study of equivalence of algorithms is an issue in the theory of computation or computability.

The λ-calculus grew out of Church's earlier work published in 1927. Church was interested in a single question: how to formally describe how variables work. In the 400 plus years since Viète, the actual, precise rules on the use of variables and parameters had not been formalized; the λ-calculus provided such a basis. Interestingly, the actual semantics for the λ-calculus did not get worked out until the 1980s.

In summary, the role of functionality in programming is undeniable. The notation used in programming languages borrows heavily from the standard algebraic notation. This leads to a class of languages, called functional languages, that include Lisp and ML. ML's pragmatic beginnings were support of an extensive type system and a demonstration that certain principles of typing could be efficiently implemented. Functional languages are developed on a different basis than the C-based systems, the latter being called imperative to emphasize the use of commands such as `while`. Functional

language aficionados maintain that recursion *pragmatically* better captures `while`s.

12.2.1 The Influence of Logic

Modern logic systems are effectively symbolic languages with no specific meaning assigned to the nonlogical symbols. Meaning (semantics) is supplied by assigning variables, constants, and functions a specific meaning based on the use at hand. For example, mathematical theories, such as group theory, outline the logical properties of groups, but leave the specific actions to the semantic interpretation; hence the role of semantics in programming languages.

Another use of logic in programming languages is the development of axiomatic semantics, proposed primarily by Floyd, Hoare, Gries, and Dijkstra. Let's start with Hoare's "An Axiomatic Basis for Computer Programming" (1969). Operational and functional approaches describe how something is computed; the axiomatic approach is to consider under what conditions something is defined. We now call the notational concept *Hoare triples*, denoted:

{*pre-condition*} {*statement*} {*post-condition*}

where *pre-conditions* and *post-conditions* are logical statements about the *state* of the program.

These ideas are the foundation of formal methods, such as Communicating Sequential Processes and program synthesis methods. Students will recognize the software engineering terms *pre-* and *post-*conditions that got their start in Hoare logics.

To summarize, on the surface, one would think there is no connection between logic and programming languages. Clearly this is not true. In fact, it is hard to separate modern logic and modern functional approaches at the foundation level—they all grew out of the larger issue of how to specify logical semantics.

12.2.2 Computability

A central issue in computation is the question, "What are the limits of computational devices?" In other words, what sorts of problems are solvable by computation? Turing showed there are some problems that are *inherently unsolvable*. We can see that this problem has dogged mathematics since Diophantus in the third century CE. The focus on computation was an outgrowth of two controversies: Russell's paradox and L. E. J. Brouwer's criticism of mathematics.

Russell's paradox was identified after the German logician Gottlob Frege was preparing to publish a book on the newly developed subject of

set theory. In the book, Frege allowed for a *set of all sets*. Basically, Russell showed this to be a paradox. There are many types of paradoxes; two are the logical and semantic paradoxes. A logical paradox is a statement that is an argument, based on acceptable premisses and using justifiably valid reasoning that leads to a conclusion that is self-contradictory. The second type of paradox is a semantic paradox, which involves language and not logic. A famous semantic paradox is "This sentence is not true"; it is a semantic paradox because the paradox relies on the meaning of the words. Russell's statement is a logical paradox because it is a logical statement (it has premises and reasoning) that Russell showed was oxymoronic. Let A be the set of all sets. Then

(1) "$A \in A$. A cannot be in A because A is the set of all sets.

(2) "$A \notin A$. But A is the set of all sets, so it must be in A.

(3) These are the only possibilities.

Russell's paradox set the mathematics and logic world on a hunt for absolute "knowability" in logic. Until this point, it had been assumed that paradoxes were oddities; now the foundations of mathematics were in jeopardy.

12.3 MOVING FORWARD FROM 1954

The first recognizable programming language is taken to be first described in 1954, in *Preliminary Report: Specifications for the IBM Mathematical FORmula TRANslating System*. FORTRAN is credited to John Backus and was released commercially in 1957. The specifications of FORTRAN have been standardized since 1966. This standardization is controlled by the J3 subcommitte of the International Committee for Information Technology Standards (INCITS), formerly known as the National Committee for Information Technology Standards (NCITS).

> J3 developed the FORTRAN 66, FORTRAN 77, FORTRAN 90, FORTRAN 95 and FORTRAN 2003 standards. FORTRAN 2003, published 18 November 2004, is an upwardly-compatible extension of FORTRAN 95, adding, among other things, support for exception handling, object-oriented programming, and improved interoperability with the C language. Working closely with ISO/IEC/JTC1/ SC22/WG5, the international FORTRAN standards committee, J3 is the primary development body for FORTRAN 2008. FORTRAN 2008 is planned to be a minor revision of FORTRAN 2003.*

* Taken from the J3 Web site,www.j3-FORTRAN.org

Why bring this up? The point is that even such an "ancient" language as FORTRAN is an evolving system, despite the hubris of different programming language communities.

Let's look at the evolution of programming language concepts over the past 50-plus years.

12.4 FROM 1954 UNTIL 1965

Programming language development was rampant in the time frame 1954 to 1964. Virtually all the programming language paradigms that survived until today have their basis in this time period.

12.4.1 Imperative Languages from 1954 to 1965

Early languages were, not unexpectedly, somewhat ad hoc in their design. The years 1957 to 1965 were very eventful, with the introduction of many of the languages still in use today. Of the imperative style programming language, the three main languages of the period are

1. FORTRAN: FORTRAN II in 1957 and ending with FORTRAN IV in 1962.
2. Algol: Algol 58 and Algol 60 in those years; a variant of Algol, called Jovial, was introduced, too.
3. COBOL: COBOL versions in 1958, 1961, and 1962.

FORTRAN and Algol were primarily conceived as numerical processing languages but rapidly became general purpose languages. COBOL was the product of Rear Admiral Grace Murray Hopper. Whereas Algol and FORTRAN were inspired by scientific computing, COBOL was developed to meet business needs. The difference in language was presentation: although FORTRAN and Algol looked like C-based mathematical languages, COBOL looked like English. FORTRAN and COBOL both have modern, current standards in effect.

Three important offshoots of Algol were Pascal, CPL, and Simula I. Pascal began as an attempt to produce a language that could be used to teach programming; it would become an important commercial success as a system development language. CPL would be extended to BCPL, then B and finally C. Simula I (1964) would become Simula 67, the beginning of the object-oriented paradigm. Simula-based systems were developed not for general programming, but for discrete-event simulation.

Late in this timeframe came another venerable system: Basic in 1964. The reader probably has used either Visual Basic or its modern counterpart VB.NET.

PL/I was first produced in 1964 and was hailed by IBM as its preferred mode of development languages. PL/I was a very rich language that tried to cover the entire landscape of applications. A major difference with PL/I and C-based languages was that the input/output sublanguage was massive, covering access methods that are now subsumed by database management systems. In many ways, C was developed as the antithesis of PL/I.

12.4.2 Lisp and the Functional Paradigm

Lisp 1 was produced in 1958 by John McCarthy, then at the Massachusetts Institute of Technology. McCarthy developed Lisp because he disagreed with the approach taken in FORTRAN. Lisp is important for three major reasons: functional programming, introducing the λ-calculus, and supporting artificial intelligence research.

1. Functional programming. The hallmark of Lisp is that it was the first language to rely completely on recursion and functions. Lisp programmers of the 1960s and 1970s spoke of pure Lisp, which disallowed the use of programming structures such as `while` and `for` loops of imperative languages.
2. The λ-calculus. As part of the Lisp development, McCarthy (1963) wrote "A Basis for a Mathematical Theory of Computation," in which he laid out the case for moving programming from the Turing machine model then in vogue to the λ-calculus version. His view would ultimately become the dominant view, and for that reason the λ-calculus basis is used in this text.
3. Artificial intelligence. Until Lisp came along, there were no adequate languages for artificial intelligence research. Lisp became, and perhaps remains, the preeminent language in artificial intelligence research. Among its attributes, the inclusion of garbage collection made many artificial intelligence algorithms practical. Lisp 1.5 came out in 1962 from which sprang many different Lisp-like systems.

12.4.3 SNOBOL and Pattern Matching

SNOBOL was introduced in 1962 and it was based on the idea of pattern matching. At a time when built-in string manipulation operations were nonexistent, SNOBOL had a full repertoire of string operations, including full pattern-matching capability. It was dynamically typed and interpretive (as were the first Lisps). While not a direct descendent, Prolog shares SNOBOL's pattern-matching and searching focus.

12.4.4 APL and Matrices

Finally, a completely different language was developed: APL, short for "A Programming Language." APL was the invention of Kenneth Iverson and to say that it was unique is probably an understatement. Iverson had invented a notation for describing matrix computations, which he turned into a programming system in 1960. It was so unique that a special keyboard was required. Although completely unique, APL represents how far thinking ranged in the early 1960s as to what programming and programming languages were all about.

12.5 PROGRAMMING LANGUAGES FROM 1965 TO 1980

Although many programming languages were invented and many of the original languages underwent extensive development in this period, there were three events of notice: the development of structured programming, the implementation of Smalltalk, and the implementation of ML.

12.5.1 Structured Programming

Early programming languages and programming methodologies were dominated by a construct called the `goto`, something that is rarely seen in more contemporary languages. The construct does not even exist in Java, for example. Structured programming was based on a theorem in computability but it took the famous paper by Dijkstra (1968), "Go To Statement Considered Harmful," to popularize the notion. It is not worth recanting the details here, but suffice it to say that the elimination of `goto`s was seen as immensely important.

The upshot is that the languages of the day worked hard to remove, or at least minimize, the use of the `goto` construct. Perhaps the best-known of the "non-goto" languages was BLISS. BLISS is a system programming language developed at Carnegie Mellon University by W. A. Wulf, D. B. Russell, and A. N. Habermann around 1970. It was used extensively to develop systems until C. BLISS is a typeless block-structured language based on expressions rather than statements, and does not include a goto statement.

While not as radical as BLISS, Pascal made its appearance in 1970, along with C in 1971. These were noted earlier.

12.5.2 The Advent of Logical Programming

Prolog was introduced in 1970. Compared to either the imperative style or the functional style, Prolog is a distinct departure. Imperative and functional languages are based on λ-calculus; Prolog is based on logic. Hence, Prolog represents the archtypal logic programming language.

Prolog was initially designed to support artificial intelligence applications (recall Lisp). Prolog programs have two readings: a declarative one and an operational one. Prolog programmers place a heavy declarative emphasis on thinking of the logical relations between objects or entities relevant to a given problem, rather than on procedural steps necessary to solve it. The system decides the way to solve the problem, including the sequences of instructions that the computer must go through to solve it. It solves problems by searching a knowledge base (database) that would be greatly improved if several processors are made to search different parts of the database.

12.5.3 ML and Types

ML was originally developed as an experiment in type-checking. ML is a general-purpose functional programming language developed by Robin Milner and others in the late 1970s at the University of Edinburgh. ML stands for *metalanguage* as it was conceived to develop proof tactics in the LCF theorem prover. LCF stands for logic of computable functions and is important in its own right. The language of LCF was pplamda, for which ML was the metalanguage. ML is known for its use of the Hindley–Milner type inference algorithm, which can infer the types of most values without requiring the extensive annotation often criticized in languages such as Java.

12.5.4 Our Old Friend: Forth

Forth was introduced in 1968. Its history was given in Milestone I and it is mentioned here for completeness.

12.6 THE 1980s AND 1990s: THE OBJECT-ORIENTED PARADIGM

The major innovation in this timeframe was not an innovation as much as the popularization of the object-oriented paradigm. In these two decades we see the rise of C++, Objective C, Objective Pascal, Objective CAML (an offshoot of ML), Python (and its non-object base, Perl), and, of course, Java. The object paradigm derives its usefulness by naturally associating real-world objects with software models through the language. The popularity of the paradigm can be seen by reading the "help wanted" advertisements.

12.7 THE PARALLEL PROCESSING REVOLUTION

A true revolution, certainly in the United States, was the *HPCC*, the High Performance Computing and Communications Act, in 1991. This Act

provided funds and momentum to develop parallel and distributed computing and what would become the Internet. Now things get interesting.

12.7.1 Architecture Nomenclature

One classification scheme of architectures is based on the number of instruction streams by the number of data streams, as shown in Figure 12.1. For exmaple, a simple von Neumann style chip has one instruction stream (as pointed to by the program word) and one data stream (it retrieves one piece of data at a time). Architectures meant to process mathematical matrix programs are often SIMD; the same instruction is simultaneously executed on many different data streams. SIMD architectures gain one order of magnitude in execution time; they effectively remove the innermost loop in matrix algorithms. SIMD systems can, and have, made good use of the classical programming languages, especially FORTRAN.

Things are more interesting when we consider multiple instruction streams. MISD architectures have been developed (Halaas et al. 2004), but by far the most familiar architectures are the Multiple Instruction, Multiple Data architectures. The Internet can be thought of as a massive MIMD system.

12.7.2 Languages for Non-SISD Architectures

One of the original languages suitable for MIMD work was proposed in 1974 by C. A. R. Hoare for operating systems. Hoare's proposal was for *message passing primitives* of `send` and `receive`. In order to control concurrency, `receive` was a blocking primitive—it caused a wait. Occam was the system used by the INMOS transputer (now defunct), but occam is still available.*

The development of operating systems has been a driver of programming language development, C being the canonical example. Language features, such as `monitors` (a type of object), was introduced by Hoare in 1974. Monitors encapsulated semaphores, but were not the complete langauge as was occam.

The programming language UNITY was developed by K. Mani Chandy and Jayadev Misra in their book *Parallel Program Design: A Foundation* (1988). Although it was considered a theoretical language, it focused on what, instead of where, when, or how. The peculiar thing about the language is that it has no flow control. The statements in the program

* One of the lessons of programming language history is that, once developed, a language will have adherents forever.

Online Resources

Online descriptions of the the following language systems are available

General Purpose Imperative Languages				
Ada	Algol	Basic	C	C++
C#	COBOL	FORTRAN	Modula-2	Pascal
Perl				
Object-Oriented Languages				
C++	Eiffel	Java	Objective-C	Python
Ruby	Simula	Smalltalk		
Functional Languages				
Caml	Common Lisp	Haskell	ML	Ocaml
Scheme				
Logic Programming Languages				
Prolog				
Web Development Languages				
ColdFusion	Delphi	JavaScript	PHP	PL/SQL
PowerBuilder	Visual Basic	VB.NET		
Database Programming				
ABAP	Clipper	MUMPS	T-SQL	

Miscellaneous

APL: Matrices and linear algebra
AWK: Pattern Matching
Tcl: Command scripting language
RPG: Venerable report writing language
SAS: Statistical processing
Logo: Educational language of the turtle
CLIPS: Rule-based programming
Maple: Computer Algebra System
Mathematica: Computer Algebra System
Rational Rose: System Development System

Architecture Types

	Single data	Multiple data
Single Instruction	SISD	SIMD
Multiple Instruction	MISD	MIMD

Figure 12.1 Architecture Classification by Instructions and Data

run in a random order until none of the statements cause change if run. To discrete event simulation practitioners, UNITY was a natural view of parallel/distributed processing. About the same time, Ian Foster and Stephen Taylor (1990) developed Strand, a more functionally oriented system.

Eventually, it seems, every language paradigm was moved to the parallel and distributed world, demonstrating once again Chomsky's Hierarchy.

12.8 CRITIQUE OF THE PRESENT

When we consider a lineage diagram, such as `levenez.com/lang/`, it is clear that there are very few unique brands of programming languages. And although there seems to be no end of argument over the "best" programming language, there also appears no way to scientifically establish "best." In fact, one might claim that the focus on programming languages only that one sees in an undergraduate curriculum is very short sighted.

Focusing on "conventional" programming is also a dead end. For example, programming systems, such as Maple™, Mathematica®, and Matlab®, are based on inputting information in an algebraic format, yet these languages often do not make use of what we've learned about language because the user base does not want to change to wit, types.

As the computer has become more ubiquitous, the need for better interfaces with the systems becomes more important. A classical case is the destruction of the Iranian airliner by the *USS Vincennes* in 1988. At least part of the issue was the inability to adequately display information in order for the captain of the *Vincennes* to make an informed decision.

We should even expect programming itself to change. For example, the concept of visual programming has arisen several times over the course of the last 50 years. Can programming be eliminated via automatic programming?