**HISTORY AND EVOLUTION OF COMPUTER PROGRAMMING LANGUAGES AND THE MATHEMATICS INVOLVED**

**INTRODUCTION**

Computer programs and computers themselves are all-invasive in today’s world. While many people use several programmed devices and computers on a day-to-day basis, not very many people understand the work underlying their favorite applications or games, or the mathematical foundations that most modern programming languages are based upon. In the course of this paper, I intended to start out with a brief history of computer programming languages in general, followed by a general introduction and explanation of what programming is, what it does, and how it’s classified. To end the report, I’ll show some of the underlying mathematics and mathematical principles used in programming languages that allow them to do what they do.

**HISTORY: BEFORE 1940**

Interestingly enough, credit for the very first “program” actually predates the world’s first modern computer. There are two runners for the world’s first “program”.

**The Jacquard Loom**

In 1801, the Jacquard loom was invented. The Jacquard loom used holes punched in cards to represent the movement of the sewing loom arms, in order to generate decorative patterns automatically. These punched cards are considered by some to be the world’s first “program”, as the cards contained encoded information that told a machine how to do the work for the user.

**Ada Lovelace**

Historically, credit for the world’s first computer program is commonly given to Ada Lovelace, who from 1842 to 1843 translated Italian mathematician Luigi Menabrea’s memoir on Charles Babbage’s newest proposed machine, the **Analytical Engine**. Using the article and the work of Babbage, she added a series of notes which detailed a method for calculating **Bernoulli numbers** with the **Analytical Engine**. These notes, and Ada’s method of calculating the **Bernoulli numbers**, are historically considered to be the world’s first program.

**Other Discoveries Before 1940**

In the first decade of the 20th century, numerical calculations were based on decimal numbers. Eventually, people started to realize that logic could be represented with words as well as numbers. Alonzo Church was able to express the lambda calculus in a formulaic way using this method. The Turing Machine, a theoretical device that manipulates symbols contained on a strip of tape, was an early abstraction of the operation of a tape-marking machine used, for example, in a telephone company. Turing machines helped set the standard for storage of programs as data in the **von Neumann architecture** of computers. Unlike the lambda calculus, Turing’s code didn’t serve well as a basis for higher-level languages. Its principle use lied in rigorous analyses of algorithmic complexity.2

It’s hard to identify what exactly constitutes the “first” programming language. Restrictions in these early days were defined by the limited hardware.

**Herman Hollerith.** In 1890, Herman Hollerith designed a system to record census data via punch cards after watching a conductor use a simpler version of the system on a train. Hollerith invented a mechanical tabulator to rapidly tabulate data from millions of punch cards, which he used to tabulate the results of the 1890 Census in only a year’s time, an efficiency unheard for the time. After leaving the census bureau, he formed a firm using his ideas and inventions to get started. His firm, along with four other companies, merged to form the Computing Tabulating Recording Corporation (CTRC). Later, under the presidency of Thomas J. Walter, it was renamed International Business Machines Incorporated, or IBM. The importance of Hollerith’s contribution to early programming advancements is that his tabulator machine. Originally, it was hard wired to work only on the 1890 census. In later revisions, he installed a control panel allowing it to do different jobs without having to be rebuilt, making it an early stepping stone towards an actual programmable computer.5

Punch cards allowed 80 columns, but some of the columns had to be used for a sorting number on each card. Even the punch holes in a player piano scroll can be considered a limited domain-specific programming language. FORTRAN, one of the earlier languages, included some keywords which were the same as English language words, such as “IF”, “GOTO”, and “CONTINUE”. Early computer programs had to be **interleaved** with the rotations of a magnetic drum, which was used in early computers for memory.

**Alan Turing.** Alan Turing was an English mathematician, logician, cryptanalyst, and computer scientist, and an influential developer of computer science. Among his many achievements, Turing provided a formalization of the concept of the algorithm and computation with the Turing machine, and his Turing test was a significant and provocative contribution to the growing debate of artificial intelligence. During WWII, he worked for the British government’s Code & Cypher School at Bletchley Park, Britain’s code breaking center during the war. He devised a number of techniques at this time for breaking German ciphers, including the method of the bombe, an electromechanical device used by the British cryptologists to help break German Enigma-machine generated signals. After the war, he went to work at the National Physical Laboratory in London, where he created a design for one of the first stored-program computers, the Automatic Computing Engine (ACE). During this time, he began writing his paper on his idea of a “universal computing machine”, or what is now known as a “Universal Turing machine”. Turing met a tragic end when in 1952, he was convicted for what was then a crime in Britain of homosexuality. Rather than face prosecution, he elected to be chemically castrated by the courts. In 1954, he was found dead in his home, having taken a bite from a cyanide-laced apple. Some people believe his death was an accident, as Turing had taken an interest in chemistry towards the end of his life, and like many intelligent men, was a bit of a slob. Others believe it was suicide, as Turing had had a lifelong fixation with the story of Snow White & The 7 Dwarves. In September of 2009, the British government issued a formal apology for the treatment of Alan Turing during the war. As of this writing, there is currently a petition in circulation to grant Alan Turing posthumous Knighthood as well.7

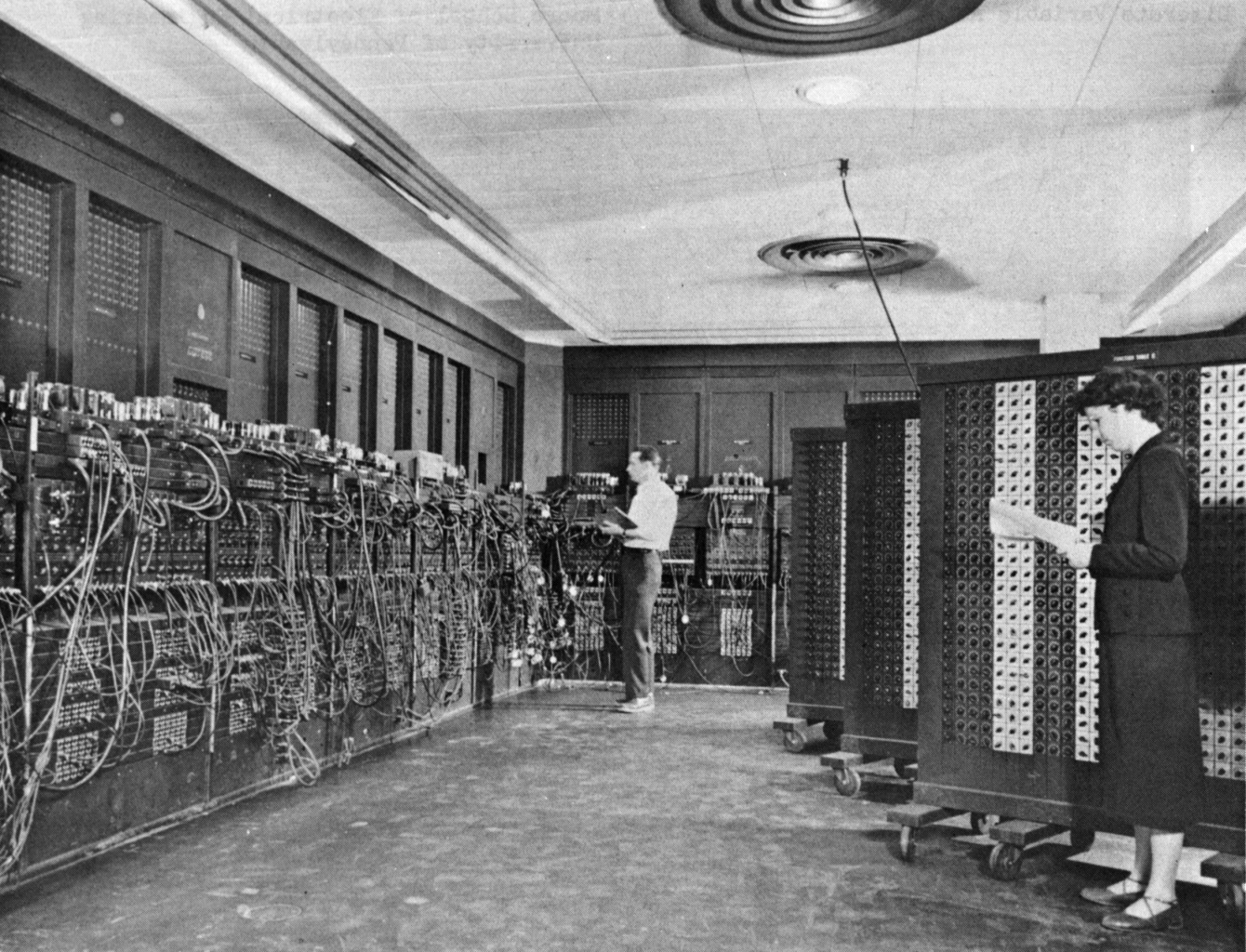
***The Turing Machine.*** A Turing machine is a theoretical device that manipulates symbols contained on a strip of tape. A Turing machine can be adapted to simulate the logic of any computer algorithm, and is particularly useful in explaining the functions of CPU inside of a computer. Turing machines are not intended to be practical computing technology, but rather as a thought experiment representing a computing machine. They’re used to this day to help computer scientists understand the limits of mechanical computation. Turing originally came up with the idea in a publication of his published in 1936. In an 1948 essay entitled “Intelligent Machinery”, Turing wrote that his “Logical Computing Machine”, consisted of:

…”an infinite memory capacity obtained in the form of an infinite tape marked out into squares on each of which a symbol could be printed. At any moment there is one symbol in the machine; it is called the scanned symbol. The machine can alter the scanned symbol and its behavior is in part determined by that symbol, but the symbols on the tape elsewhere do not affect the behavior of the machine. However, the tape can be moved back and forth through the machine, this being one of the elementary operations of the machine. Any symbol on the tape may therefore eventually have an innings” (Turing 1948, p 61)

A Turing machine that can simulate any other Turing machine is called a Universal Turing Machine, or UTM. A more mathematically-oriented definition with a similar “universal” nature was introduced by Alonzo Church, whose lambda calculus intertwined with Turing’s in a formal theory of computation known as the Church-Turing thesis. The thesis states that Turing machines indeed capture the informal notion of effective method in logic and mathematics, and provide a precise definition of an algorithm, or “mechanical procedure”.

**HISTORY: THE 1940s**

The 1940s saw the first recognizably modern, electrically powered computers. The limited speed and memory capacity forced programmers to write hand tuned **assembly language** programs. Assembly language programs required a great deal of mental effort and tended to be very error-prone, however. 1946 saw the introduction of first large-scale, general-purpose electronic digital computer in the form of the Electronic Numerical Integrator and Computer, or ENIAC. The ENIAC weighed 30 tons, occupied a 30x50 foot space, and could perform a then whopping 300 multiplications per second.2 While other computers came before the ENIAC, such as the Colossus or ACE, the ENIAC has the distinction of not only being the first general-purpose computer, as well as being the first **Turing complete** computers.



**Figure 1**: The ENIAC computer, in the Army’s Ballistic Research Laboratory building 328

In 1948, Konrad Zuse published a paper about his programming language, Plankalkül. Plankalkül was a computer language developed for engineering purposes. It was the first high-level, non-von Neumann programming language to be designed for a computer, and was developed between 1943 and 1945. However, it was not implemented in his time (not until 1998!) and his original contributions were isolated from other developments. A quick example here shows a program in Plankalkül that computes the maximum of three variables by calling the function max:

P1 max3 (V0[:8.0],V1[:8.0],V2[:8.0]) => R0[:8.0]

max(V0[:8.0],V1[:8.0]) => Z1[:8.0]

max(Z1[:8.0],V2[:8.0]) => R0[:8.0]

END

P2 max (V0[:8.0],V1[:8.0]) => R0[:8.0]

V0[:8.0] => Z1[:8.0]

(Z1[:8.0] < V1[:8.0]) -> V1[:8.0] => Z1[:8.0]

Z1[:8.0] => R0[:8.0]

END

Several important languages were developed in this period, such as C-10, the ENIAC coding system, which were early attempts at formal assembly languages, and Plankalkül. While none of these are still in use today, they served as a springboard to more modern programming languages.

**HISTORY: THE 1950s AND 1960s**

The 1950s also saw the development of the first three modern programming languages, the descendants of whom are still in use today:

**Fortran**

FORTRAN (the name was capitalized until the release of Fortran 77, after which it was spelled with normal capitalization) is a general-purpose, procedural, **imperative programming** language developed by IBM in the 1950s for scientific and engineering applications. It was developed in 1957 by John Backus and a team at IBM.5

**Lisp**

Lisp, or LISP, or the LISt Processor, was developed in 1958 by John McCarthy at the Massachusetts Institute of Technology (MIT). It was first implemented by Steve Russell on an IBM 705 computer after Russell had read McCarthy’s paper on his new language, and realized that the LISP “eval” function could be implemented in machine code. Since its inception, LISP was closely connected with the artificial intelligence research community, especially on PDP-10 systems. Lisp was used as the implementation of the programming language “Micro Planner” which was used in the famous AI system, SHRDLU.

**Cobol**

COBOL, an acronym for Common Business-Oriented Language, was developed primarily for business, fiancé, and administrative systems for companies and governments. A specification of COBOL was initially created during the second half of 1959 by Grace Hopper. The later specifications that would become the COBOL language were set over a period of time by a series of committees in the US Department of Defense and the Pentagon, completing the specifications in December of 1959. Grace Hopper’s original language was called FLOW-MATIC, and heavily influenced the development of COBOL.

**The ALGOL 60 Report**

Another significant event of the 1950s in computer programming was the publication of the ALGOL 60 report by a committee of American and European computer scientists. The ALGOL 60 Report (the ALGOrithmic Language) was meant to detail specifications for a “new language for algorithms”. The report served to consolidate the many ideas about programming circulating at the time, and featured two new key language innovations: nested block structures, which simply meant meaningful chucks of code could be grouped into “blocks” without having to be turned into separate, explicitly named procedures, and lexical scoping, which meant a block could have its own variables, procedures and functions that code outside the chunk cannot access by name. Keywords in several languages, such as “local”, “private”, “public” and “static” spawned from this initial conception of lexical scoping. The ALGOL 60 report also popularized the use of **Backus-Naur Form** as a means to describe a language’s syntax. Nearly all subsequent programming languages have used a variant of BNF to describe the context-free portion of their syntax.

**HISTORY: THE 1960s AND 1970s**

Two other important languages were developed in the late 1960s: Simula, and BASIC.

**Simula**

Simula was a high-level programming language developed in the mid-to-late 60s at the Norwegian Computing Center in Oslo, by Ole-Johan Dahl and Kristen Nygaard. It is a fairly faithful superset of the Algol 60 language. While some thing that Simula is a legacy language, it still used and taught at some universities, specifically at the University of Malta. Simula holds the distinction of being the first object-oriented programming language. As the name implies, it was designed for doing simulations, and the needs of that domain provided the framework for many of the features of object-oriented languages today.

**Begin**

**OutText** ("Hello World!");

**Outimage**;

**End**;

**Figure 2**: A simple “Hello World” program in Simula

**Basic**

BASIC is an acronym for Beginner’s All-purpose Symbolic Instruction Code. The original BASIC was developed in 1964 by John George Kemeny and Thomas Eugene Kurtz at Dartmouth in New Hampshire to provide computer access to non-science students. At the time of its inception, there was no commercial software packages like we have today, so most computers were operated via custom written programs, which was something only scientists and mathematicians tended to be able to do. Up until this point, languages were designed with specific tasks in mind (see the entries on FORTRAN or COBOL for examples), but as computers got faster and more readily available, the idea of time-sharing idle computer time to people became more and more popular. BASIC was not only developed to allow usage of a computer to a greater range of people, but to be a general-purpose programming language as well. Basic saw a major boom in the 1970s and early 1980s with the advent of personal home computers. Third generation BASIC dialects are still in use today, such as Visual Basic, REALbasic, StarOffice Basic, and BliztMax.

These and other languages spawned an entire family of descendants, and most modern language can count at least one of the languages developed at this time as an ancestor.

**Structured Programming**

The 1960s and 1970s saw considerable debate over the merits of “structured programming”. Structured programming can be seen as a subset or subdiscipline of **imperative programming**. It is most famous for removing or reducing the reliance on the GOTO statement. Several different methodologies have been developed for writing structured programs. Edsger Djikstra’s structured programming, where the logic of a program is a structure composed of similar sub-structures in a limited number of ways, reducing understanding a program to understand each structure on its own. Djikstra’s methodology is less common, due to the advent of object-oriented programming. A view derived from Djikstra’s also advocates splitting programs into sub-sections with a single point of entry, but is strongly opposed to the concept of a single point of exit. Finally, there is Jackson Structured Programming, which is based on aligning data structures with program structures. This approach applied the fundamental structures proposed by Djikstra, but as constructs that used the high-level structure of a program to be modeled on the underlying data structures being processed.

The debate began in 1967, when a letter from Djikstra was published in “Communications of the ACM” with the heading “Goto statement considered harmful.” His letter citied a proof done in a 1966 paper by Boem and Jacopini, which provided the theoretical basis of structured programming. What really got the debate going wasn’t the proof itself, but whether or not structured programming could actually improve software’s clarity, quality, and development time enough to justify training new programmers in it. Donald Knuth accepted the principle, but disagreed with abolishing the GOTO statement. In a 1974 paper, “Structured Programming with Goto Statements”, he gave examples where he believed that a direct jump leads to clearer and more efficient code without sacrificing provability.

**Donald Knuth**

Donald Knuth is a renowned computer scientist and Professor Emeritus of the Art of Computer Programming at Stanford University. Knuth is the author of the seminal multi-volume work “The Art of Computer Programming”, or TAOCP. He has been called the father of the analysis of algorithms, contributing to development of, and systematizing formal mathematical techniques for, the rigorous analysis of the computational complexity of algorithms, and in the process popularizing **asymptotic notation**. Knuth was the recipient of the first ACM Grace Murray Hopper Award, and has received several other awards in his life, such as the Turing Award, the National Medal of Science, the John von Neumann Medal, and the Kyoto Prize. In recognition of his contributions to the field of computer science, in 1990, he was awarded the one-of-a-kind academic title of *Professor of the Art of Computer Programming*, which has since been revised to *Professor Emeritus of the Art of Computer Programming*. Besides his many published works, Knuth is well known for his professional humor, often making mathematical and computer science based jokes in his works.

**Languages Developed In The 1970s**

New programming languages began to abound in the 1970s. Some important languages developed in this period include: Pascal (1970), Forth (1970), Smalltalk (1972), ML (1973), and SQL (1978). Two of the more important languages developed in this time are the following:

**C**. C is a general-purpose programming language developed in 1972 by Dennis Ritchie at the Bell Telephone Laboratories for use with the Unix Operating System. C is one of the most popular programming languages. It is widely used on many different software platforms, and there are few computer architectures for which a C compiler does not exist. C has greatly influenced many other popular programming languages, most notably C++, which originally began as an extension to C. The name “C” came about because many of its features were derived from an earlier language called “B”, which was itself a stripped-down version of the BCPL programming language. This method of naming continued into C++ (the ++ notation in C/C++ languages indicates an increment of a variable. Since C++ was originally designed as an extension to C, the naming convention continued). One consequence of C’s wide acceptance and efficiency is that compilers, libraries, and interpreters of OTHER languages are often implemented in C.

**Prolog**. Prolog is a general purpose logic programming language often associated with artificial intelligence and computational linguistics. Prolog was one of the first **logic programming languages**, and remains among the most popular such languages to this date. The language was first conceived by a group around Alain Colmerauer in Marseille, France, in the early 1970s, and the first Prolog system was developed in 1972 by Alain Colmerauer and Philipe Roussel. Prolog has its roots in formal logic, and unlike many other programming languages, is **declarative** in nature. An example of Prolog in action (that I wrote last year):

accepts(L) :- start(S),

transition(S,L).

transition(X,[A|B]) :-

arrow(X,A,Y), /\* X ---A---> Y \*/

write(X),

write(' '),

write([A|B]),

nl,

transition(Y,B).

transition(X,[]) :-

final(X),

write(X),

write(' '),

write([]), nl.

arrow(q0,0,q0).

arrow(q0,1,q1).

arrow(q1,0,q2).

arrow(q1,1,q1).

arrow(q2,0,q3).

arrow(q2,1,q1).

arrow(q3,0,q0).

arrow(q3,1,q4).

arrow(q4,0,q4).

arrow(q4,1,q4).

start(q0).

final(q4).

**Figure 3:** This is an example program written in Prolog. This program uses a deterministic automaton to test an inputted string vs the language L = {w | w contains 1001}

**HISTORY: THE 1980s**

The 1980s saw relative consolidation in programming languages. C++ came along and combined object-oriented and systems programming. The United States government standardized Ada, a language made for use by defense contractors. Vast sums of money were spent investigating so-called fifth-generation programming languages that incorporated logic programming constructs. The community moved to standardize ML and Lisp. Instead of inventing new paradigms, these movements elaborated upon the ideas invented in the previous decade.

One important trend in language design that cropped up during this time was an increased focus on programming for large-scale systems through the use of modules, or large-scale organizations of code. Another advance forwarded in the 1980s was the RISC movement in computer architecture, which postulated that hardware should be designed for compliers rather than for human assembly programmers.

**Modular programming**. Modular programming is a software design technique that increases the extent to which software is composed from separate parts, called modules. Conceptually, modules represent a separation of concerns, and improve maintainability by enforcing logical boundaries between components through interfaces. A module interface expresses the elements that are provided and required by the module. The elements defined in the interface are detectable by other modules. The implementation contains the working code that corresponds to the elements declared in the interface. Libraries of components built from separately compiled modules can be combined into a whole by using a **linker**.

One of the key aspects of modular programming is the ability to be able to separate concerns such that none or few modules depend upon other modules of the system. It is of upmost importance to have as few dependencies as possible. Another key aspect is that when creating a modular system, instead of creating a monolithic application where the smallest piece is the whole application self, one creates several smaller modules which when composed together will create the whole system. Another key aspect of modular programming is **encapsulation**. What this means is that each module does not directly communicate with other modules or the whole, but rather everything communicates via an interface. Substitutability and reusability are other key aspects to keep in mind when designing modules. A good example is to think of car parts like disk brakes and car stereos. You don’t want to build a car stereo when you are building cars. You want to buy them and plug it in. You also don’t want your braking system affecting the car stereo, or worse, the car stereo affecting the braking system.

**RISC Architecture**. RISC stands for reduced instruction set computer. The idea of RISC-labeled designs has been around for a while now, since 1975 or so. These original design aspects included the observations that the memory restricted compilers of the time were often unable to take advantage of features intended to facilitate manual assembly coding, and that complex addressing inherently takes many cycles to perform, due to the implied additional memory access. It was argued that such functions would better be performed by sequences of simpler instructions, if this could yield implementations simple enough to cope with really high frequencies, and small enough to leave room for many registers, factoring out slow memory accesses. Uniform, fixed length instructions with arithmetics restricted to registers were chosen to ease instruction pipelining in these simple designs, with special load-store instructions accessing memory.

In the mid 1970s, researchers at IBM demonstrated that the majority of instructions in CPU ISAs were not used by most programs generated by compilers available at that time. It proved difficult in many cases to write a compiler with more than limited ability to take advantage of the features provided by conventional CPUs. To make a long story short, by the time the 1980s came around, the idea of RISC architecture was used in order to facilitate the usage of CPUs with compiled programs in mind instead of the old standard of CPUs designed with assembly language usage in mind.

In the 1980s, several important languages were developed as well. Languages like Ada (1983), Eiffel (1985), Erlang (1986), Perl (1987) and FL, or Backus (1989) were introduced during this time. One language we will look at a bit closer, which was developed in 1983, is C++.

**C++.** C++ is a **statically typed**, **free-form**, multi-paradigm, compiled, general-purpose programming language. It is commonly regarded as a middle-level language, as it comprises a combination of both high-level and low-level language features. It was developed by Bjarne Stroustrup starting in 1979 at Bell Labs as an enhancement to the C programming languages. Its original name was “C with Classes”, which referred to the fact that C was not object-oriented in nature, while C++ was. It was renamed to C++ in 1983. C++ is widely used in the software industry to this day, and remains one of the most popular languages ever created.

C++ began its life as an enhancement to the C language developed in the previous decade. It added classes, or object-orientation, to C, later fallowed by **virtual functions**, **operator overloading**, **multiple inheritance**, **templates**, and **exception handling** among other features. After years of development, the C++ programming language standard was ratified in 1998 as ISO/IEC 14882:1998. That standard is still current, but is amended by the 2003 technical corrigendum, ISO/IEC 14882:2003. The next standard version, C++0x, is still in development as of this writing.

**HISTORY: THE 1990s**

In the 1990s, the novelty and innovation of the previous decades slowed, but much recombination as well as maturation of old ideas flourished. A big driving philosophy in the 1990s was programmer productivity. Many “rapid application development”, or RAD, languages emerged during this time. RAD languages were marked by the inclusion of an **IDE** and **garbage collection** among other features, and were usually descendants of older languages. One feature all RAD languages shared was the fact that they were all object-oriented. Some examples of RAD languages include Object Pascal, Visual Basic, and C#. Several other new and important languages were developed during this time, including Haskell (1990), Python (1991), Java (1991), Ruby (1993), Lua (1993), Pike (1994), Javascript (1994), PHP (1994) and C# (2000). These last three languages, much more innovative and provocative than the RAD languages, were known as scripting languages. Unlike RAD languages, scripting languages did not descend from other, earlier languages, and featured new syntaxes and more liberal incorporation of features. Scripting languages quickly became to be the most prominent languages used in connection with the blooming Web.

**Java.** One of the more prominent languages to be developed in the 1990s was Java. Java was originally developed by James Gosling at Sun Microsystems and released in 1995 as a core component of Sun Microsystems’ Java Platform. Much of the syntax of Java is derived from C and C++, but has a simpler object model and fewer low-level facilities. Java programs are compiled into **bytecode** that can run on an Java Virtual Machine, or JVM, regardless of the computer architecture. Java was implemented on the promise of “Write Once, Run Anywhere” (WORA), providing no-cost run-times on popular platforms. In 1997, Sun Microsystems attempted to formalize Java, but soon withdrew from the process. Java remains a de facto standard, controlled through the Java Community Process. Despite the fact that Java was proprietary software, it was available without charge for the most part. Revenue was generated through the selling of licenses for specialized products such as the Java Enterprise System. In 2006, Sun released much of Java as **open source software** under the terms of the GNU General Public License (GPL). There are also special classes of Java code, such as Applets, which are Java programs that are embedded in other applications, typically in Web pages. Java Servlets provide Web developers with simple, consistent mechanisms for extending the functionality of a Web server and for accessing existing business systems. Servlets are server-side Java EE components that generate responses to requests from clients. They can almost be thought of as applets that run on the server side. JavaServer Pages, or JSPs, are server-side Java EE components that generate responses to HTTP requests from clients. JSPs embed Java code in an HTML page by using special delimiters. JSP is compiled into a Servlet the first time it is access. Afterwards, the created Servlet creates the responses.

**Scripting Languages**. Scripting languages are programming languages that allow control of one or more software applications. The “scripts” written are distinct from the core code of the main application, which is usually written in an entirely different language. These scripts are, in turn, usually created or at least modified by the end-user. Scripts are often interpreted from the source or byte code, whereas the applications they control are compiled to native machine code.

Scripts can be considered to be descendants of early shells developed in the 1960s to facilitate remote operation of the first time-sharing computers. The need to maximize time with a time-share lead people to develop macros, or files containing sequences of commands, which eventually developed into shell scripts. Historically, there has been a clear distinction between “real” high speed programs written in languages such as C, and the slower scripts written in languages such as Bourne Shell (1977) or Awk (1977) (Bourne Shell and Awk were languages created specifically to create shell scripts, and as such, were not listed previously with languages of the 1970s). Later languages such as Tcl and Lua were specifically designed as general purpose scripting languages that could be embedded in any application or used on their own. In 1993, the Common Gateway Interface (CGI) came along, allowing scripting languages to control web servers, and thus communicate over the web. Scripting languages that made use of CGI early on included Perl, ASP, and PHP. In today’s world, some software actually incorporates several different scripting languages. Modern web browsers can provide a language for writing extensions to the browser itself (Firefox being a prime example), and several standard embedded languages for controlling the browser, such as JavaScript, CSS, and HTML (CSS and HTML are not technically “programming languages”, but rather markup languages, or systems that annotate text in a way which is syntactically distinguishable from that text).

**2000 AND BEYOND**

In the twinkling twilight of this decade, computers and computer programming continue to evolve. Some current trends in modern computer programming include new mechanisms for adding security and reliability verification to various languages. New alternative mechanisms for modularity have been introduced. A new paradigm, aspect-oriented programming, has been introduced and tinkered with, most prominently in the programming language AspectJ (2001). Metaprogramming and reflection are means by which a computer program can write or manipulate other programs, or themselves, as their data, spawning generative programming, where programs can be generated by something other than human programmers. While several languages support metaprogramming of some level or another, Lisp is probably the current quintessential language with metaprogramming facilities. More and more programs in the 21st century support integration with XML and relational databases, allowing programs to work with massive amounts of data in a humane way for programmers. Finally, a move towards Unicode over the standard ASCII character set in modern times shows a move towards a more international market than ever before.

**XML**. XML stands for Extensible Markup Language (no group of people can work an acronym quite like computer programmers). XML is a set of rules for encoding documents electronically. XML’s design goals emphasize simplicity, generality, and usability over the Internet. It is a textual data format, with strong support via Unicode for the languages of the world. Although XML’s design focus was on documents, it sees wide usage representing arbitrary data structures, like those used in web services. As of the writing of this paper, XML has spun off into hundreds of XML-based languages, such as RSS, Atom, SOAP, and XHTML. Even most office productivity tools today incorporate XML into their design, including Microsoft Office, OpenOffice.org, and Apple’s iWork. XML resembles HTML in form, and in fact, XHTML is actually an extension of HTML that incorporates XML.

<?xml version="1.0" encoding='UTF-8'?>

<painting>

<img src="madonna.jpg" alt='Foligno Madonna, by Raphael'/>

<caption>This is Raphael's "Foligno" Madonna, painted in

<date>1511</date>-<date>1512</date>.</caption>

</painting>

**Figure 4**: An example of a complete XML document, merely containing an image with a caption.

**Unicode**. Unicode is a computing industry standard allowing computers to consistently represent and manipulate text expressed in most of the world’s writing systems. The latest version of Unicode consists of a repertoire of more than 107,000 characters covering 90 scripts, a set of code charts for visual reference, an encoding methodology and set of standard character encodings, an enumeration of character properties such as upper and lower case, a set of reference data computer files, and a number related items, such as character properties, rules for normalization, decomposition, collation, rendering, and bidirectional display order, for the correct display of text containing both right-to-left scripts (such as Arabic or Hebrew) and left-to-right scripts. What all of this basically means is that Unicode is a comprehensive way of defining characters electronically for compatibility throughout the world. The reason for the move to Unicode lies in the fact the much existing software still uses 8 bit characters to encode data, while 16 bit is needed for many countries. The old standard, ASCII, was based solely on the English character set, and lacks any type of support for non-standard characters present in many other languages throughout the world. The first 128 characters of Unicode are, in fact, effectively ASCII.

**CONCLUSION**

Computer science and computer programming, more than any other field one can enter, is one of constant change and evolution. Even the concepts we learn in a college course today may be obsolete by tomorrow, replaced by something newer and better (usually). Newer and newer technologies are cropping up every day. In the next decade, things like distributed processing (where the platform a program may be running on is no longer a single machine or processor) or virtual 3-D environments stand poised to be major points for future programmers. An understanding of where it all came from, however, and how we got to where we are today, gives future programmers a foundation. By seeing how we got to this point, it’s easier to see where we may be going. A common expression amongst C/C++ programmers to newer programmers is “You don’t need to know assembly language to write good programs. But an understanding of assembly will help you to write better programs.” Likewise, understanding the history of where all of this came from isn’t going to necessarily make you a good programmer, just a better one.