A bit of history and classification.

In the beginning there were tally marks...a lot of time passed...and zeros got interspersed...and the binary system came to be.

The idea of a programming language is as old as the idea of computation. Unfortunately, it took a long time to understand what we were trying to talk about.

The description of algorithms in some notation goes back to the beginning of arithmetical computation, and can be followed in many languages and historical documents, from the Rhynd papyrus and Mesopotamian clay tablets on (2000BC and earlier).

Or more relevance to us are the ideas of formalization that started with Classical Greece, and continued through the intervening centuries to culminate with some ideas of Leibnitz (the co-inventor of the Calculus, and a major philosopher of the end of the seventeenth-beginning of the eighteenth century). His idea involved the association of a unique number to each idea and the ability to reason by doing arithmetical computations: translate sentences into arithmetic, use rules of formal computation on the arithmetical quantities and translate back.

Computational aids (abaci, etc.) are quite old and descriptions for the carrying out of computations (including foreign exchange) on any such aids could qualify as protoprogramming languages.

The whole thing didn't go very far at the time, although the first mechanical device to carry out logical inferences (syllogisms and some inferences in probability) was constructed and demonstrated by the third Earl of Stanhope (1753-1816).

The Jaquard Loom - a child of that industrial revolution whose textile beginnings were spurred by the invasion of Britain on the part of inexpensive Indian cotton clothing that undersold the locally produced hand loomed fabrics - was invented in the early 1800s and allowed for programming the sequence of steps that created the carpet and the patterns. It used "punched cards". The largest program devised for it, a rather complex tapestry - took over 25,000 cards. The French textile workers went on strike, but didn't succeed in stopping mechanization.
The next series of attempts came in the middle of the nineteenth century, with Charles Babbage (and Ada Lovelace) and the Analytical Engine covering the engineering and programming aspects and George Boole (boolean algebra) - also an Englishman - covering the formal mathematical development. Babbage had devised an earlier (non-programmable) device known as the Difference Engine, and it was for delivery of a working copy of that device that he received various grants from the British Government (they had subsidized an earlier technology race: the design and construction of the Marine Chronometer - accurate to within half a second a day). In true scientist fashion, he used all the money to study the next idea - that of the fully programmable Analytical Engine - while not delivering what promised. He ultimately lost his grant...

The last quarter of the nineteenth century saw a very strong movement directed to the algorithmization of Mathematics: a large number of mathematicians were looking for effective procedures to construct all kinds of mathematical entities and were attempting to equate proof with effective procedure. Others were busily working on understanding the infinite - and therefore showing that the notion of effective procedure was too limited in principle. James Hollerith was busy convincing the US Census Bureau to buy his tabulating machines. He was successful, and I.B.M. is the offshoot of his efforts.

This state of affairs continued until 1929, when Kurt Goedel showed that any system that contains arithmetic also contains undecidable questions. Or, you didn't need fancy infinities to run into the limits of computation.

Within the next few years, several mathematicians attacked the problem of what IS computable?, and came up with formal definitions of computability. The two major ideas came from Alan Turing - with his Turing Machine - and Alonzo Church, with the lambda calculus. The two ideas appear quite different. The first involves variants of a tape, a pen and a finite state machine, while the second involves formalizing the idea of function. As it turns out, both ideas are equivalent in the computational processes they can describe, although they seem superficially different. Other ideas followed - mostly extending the power of the method by adding oracles (interactive computing is an example of this) or, in the last 20 years or do, by adding the notion of quantum processes and quantum computing.

From the point of view of Programming Languages it was the Turing Machine idea that led to the first programming languages. It lent itself to an easy realization (approximation, to be a bit more precise) in hardware and to a suggestive interpretation of the underlying processes. The Von Neumann model of computation - winner hands down up to now - was the realization of the Turing Machine model.

Assembly Language followed (a classified paper about automatic programming from the late '40ies surfaced around 1975: it talked about Assembly Language - or, how to automate putting in all those zeros and ones of machine language) and then FOR(mula)TRAN(slation). Konrad Zuse, the German computer pioneer (designed and built the first true digital computer out of telephone relays, using punched celluloid tape for
I/O, in the 1930’s) whose work to design and build the first electronic computer had been stopped by Hitler because “the war will be won in two years anyway”, had also worked out a “high level language” by 1950, but nobody read about it until the mid-1970s or so. COBOL was introduced around 1960 so that managers could read the code their underlings wrote. The work that the linguist Noam Chomsky of MIT did in the early 1950s began to be understood, and the pioneers of Artificial Intelligence were developing list processing languages, since they discovered that much of the work they were trying to do could be best carried out with the idea of list. Logicians like John McCarthy kept trying to understand the work of Alonzo Church, and trying to come up with a computational realization of the lambda calculus. Around 1960 they did, and LISP was invented - both as a list processing language (there is where it gets its name) and as a (partial) realization of the lambda calculus. The understanding of Formal Language Theory led to ALGOL 60 (John Backus, the major force behind the development of FORTRAN, was also one of the major contributors to ALGOL 60) and to all the modern imperative languages which are its intellectual descendants (including FORTRAN 90).

The period of the 1960s was almost frenetic in the exploration of high level languages. PL/I (IBM's answer to ALGOL 60: it was supposed to supersede both FORTRAN and COBOL, but the Department of Defense said NO) and the fiasco of OS360 (the last large Operating System written completely in Assembly Language: finally stabilized at 10,000 known bugs) led to Multics (the first Operating System written more than 95% in a high level language: only 5% or so of the code is really so speed critical that you need to write it on the metal) which led - as a reaction to gigantism - to the discarded PDP8 in a hallway of Bell Labs, to UNIX and A, B, C... APL, SNOBOL, BASIC, etc... appeared then. Simula 67 came out of Norway as a simulation language containing the fundamental ideas of class and object; Conniver and MicroPlanner were developed at MIT as relational languages in which you could carry out computations of predicate calculus, based on the Unification Algorithm and Resolution Theorem Prover of J. Alan Robinson (1965). The decade closes with that most baroque artifact known as ALGOL 68 - a committee designed language that tried to embody everything known at the time and only managed to be unimplementable.

One could argue that the 1970s were a time of consolidation. Pascal was a reaction to PL/I and ALGOL 68 - smaller, running on nearly anything (2000 lines of Pascal could give you a Pascal compiler and PDP 11/03s, Intel 8080s, Z80s, 6502s would run it with acceptable speed). Prolog resulted from Alain Colmerauer of Marseilles finally making sense of what the MIT guys had been unable to think all the way through, and of D. H. D. Warren of Edinburgh being able to write an efficient compiler for what Colmerauer had devised. Smalltalk came from Alan Kay and others taking the ideas of Simula 67 and really designing a full-blown object-oriented programming environment. Alan Kay was also one of the early pioneers of the bit-mapped display and the Macintosh user interface (ten years before the Macintosh). Scheme came from trying to understand the power of the lambda calculus in terms of many of these developments.

Another development of the 1970s was the realization that there was a software crisis: too few people writing code that did what it was supposed to do. The US Department
of Defense started a project that culminated in the design of the Ada language (they had 500+ different implementation languages to deal with in all their projects, and that was a REAL headache). Unfortunately, having been designed by a committee (like ALGOL 68) to be all things to all people (again, like ALGOL 68) it has had, roughly, the same success as ALGOL 68 - and similar problems of implementation. Other people went down the object oriented route. John Backus, a certifiable genius (he already had TWO proven good ideas - FORTRAN and a lot of the work on ALGOL 60, including the invention of the Backus-Naur language description notation) decided to go in a different way. He became a champion of Functional Programming, beginning to lay the groundwork for a move away from the Von Neumann model of computation, to a functional model. His contention was that the granularity of the existing model was just too fine, it intruded too deeply into the thought processes of the application programmer and a new class of languages (regardless of what happened to the underlying hardware architecture) needed to be invented. Actually, the class already existed: the languages based on the lambda calculus - of which LISP and Scheme are two (impure) examples. The pure functional languages do away with the notion of assignment, and use functional composition as the way in which parts of a program are glued together. Some of these ideas had already surfaced in Prolog, one could certainly program that way in LISP, but there were major drawbacks: there were processes nobody knew how to express in a purely functional manner, and the efficiency of all the potential candidates was, to put it mildly, atrocious - acceptable for research prototypes, but either too slow, or too demanding of memory, or unable to deal with real-time events (or all of them) to be usable in the real world.

There are very good reasons why people were not deterred: the pure functional languages (no assignment) are the only ones for which one can even hope formally to prove correctness of large programs. The provability of correctness could have considerable importance in environments where people could lose their lives (or a lot of money - take your pick) because of a software error. Furthermore, the absence of state information could make the recovery of a distributed computation possible - or maybe even easy. The problems to be overcome were thus both practically and theoretically interesting.

The 1980s saw continuation of development in all these directions - and a major disaster (this may be a slightly biased view, but it is not extreme). Ada had substantial backing (the US DOD) and a syntax that would have made an easy learn for a Pascal programmer. Instead of flooding the universities with inexpensive and small compilers for subsets - which would have allowed the easy conversion of all those Pascal programmers to Ada programmer, the DOD refused to certify anything but FULL compilers. And nobody knew how to write a full Ada compiler for several years after the specs were finalized. The first certified Ada compiler came out of NYU, was written in SETL (a language developed at NYU for the very high level description of mathematical objects and functions) and compiled at the rate of a line a minute on a supercomputer. Things did NOT get better fast. In the meantime, AT&T had been giving UNIX and C away to universities - including source code (most importantly, including source code) and the number of UNIX-C hackers had been growing like Kudzu, porting OS and language to virtually every piece of hardware known to man. The rest is history. When a dreadful kludge by the name of C++ appeared to add some
object-oriented capabilities on top of C, Ada was finished for good (the latter statement may provoke debate).

Some of the post mortems on Ada concentrated on the size and complexity of the language. That was probably irrelevant, since Common Lisp has a specification manual well over 1000 pages long to Adas 350 or so... It was just that complex languages (or any kind of really complex system) can grow best only by long term accretion and evolution: you simply cannot figure all the problems out BEFOREHAND.

Prolog had a flourishing in the mid-80s, and has found a niche. Its not-having-been-invented-here has been the usual drawback. LISP had a flourishing in the same period - partly due to one of the periodic hype-frenzies that AI gets into - and maintains its niche in both the AI community and in fast prototyping situations. EMACS is still written in LISP. Smalltalk seems to be taking over in what were typical COBOL environments - its fast prototyping capabilities have made it a very effective language in the financial community, where the window to reproduce the functionality of a competitor's software is on the order of weeks, and billions of dollars are at stake.

The functional languages are now reaching reality. People have understood how to describe any computational process in them, including interaction with a user. Their efficiency - based on optimization and code rewriting techniques not available 20 years ago - has improved so that, speedwise, the most recent ones are well within a factor of 10 of highly optimized C or FORTRAN (and, sometimes, they beat them). They are used in various production environments, including telephone switching systems (Ericsson of Sweden). They also come in various levels of purity - i.e., the way they allow a limited form of explicit assignment for efficiency purposes. Some names are Hope, Miranda, ML, Standard ML, Haskell, Concurrent Clean. US developed ones are SISAL and Id. They are, mostly, strongly typed, but they often incorporate recent developments in type inferencing mechanisms that relieve the programmer from most of the drudgery connected, for example, with Pascal (and with the fact that discussing Abstract Data Types in the context of Pascal is self-contradictory...WHY?), and allow easy generic functions without giving up any of the advantages of strong typing.

One aspect we have not discussed relates to parallelism. Ideas about speeding up computations by breaking them up in pieces running on different processors are as old as mankind - there was a French scheme of the time of the Revolution that tried to construct a pyramid with fancy mathematicians at the top directing less skilled computers, all the way down to barely numerate peasants who would just handle simple additions. It didn't work.

It was probably Seymour Cray who, in the early 1960s, working for the Control Data Corporation, incorporated multiple arithmetic units in his processors. The CDC machines were, essentially, FORTRAN machines, and the development of FORTRAN parallelizing compilers started. To make a long story short, most reasonable processors of today have multiple execution units, and most reasonable compilers support them. It is the Object Oriented and Functional Languages who make it easiest to describe and execute computations.
in parallel. If you need raw speed, nothing beats hand-crafted code in Assembly Language BUT...it is expensive to develop, it is expensive to maintain and its advantages are being slowly eroded by better algorithms, better hardware architectures and smarter compilers. Ada, FORTRAN, Linda, and PVM (the latter two are, strictly speaking, NOT languages, but parallel extensions to other languages, e.g., C), Modula2 and Modula3, Occam, Concurrent Prolog, Concurrent Clean, many extensions of C++, all support the description of parallel algorithms at the language level (rather than at the OS level).