Theme 3

Algorithms and programming languages

Subjects:

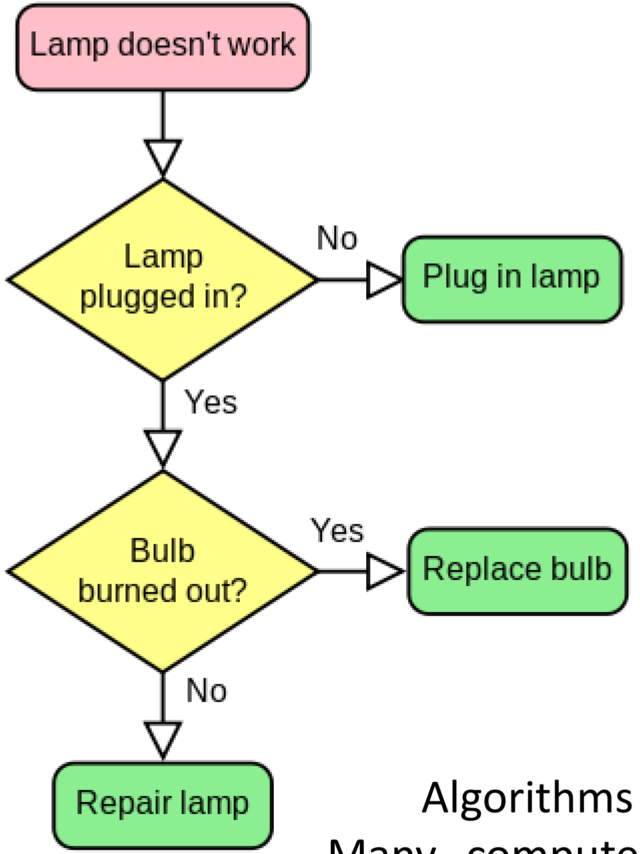
-Basic concepts
-Turing machine
-Expressing algorithms
-Classification of algorithms
-Programming languages
-Generations of programming languages
-Application Programming Interface

Duration - 2 ac.h.

Basic concepts

In mathematics, computing, linguistics, and related subjects, an **algorithm is an effective method for solving a problem using a finite sequence of instructions**.

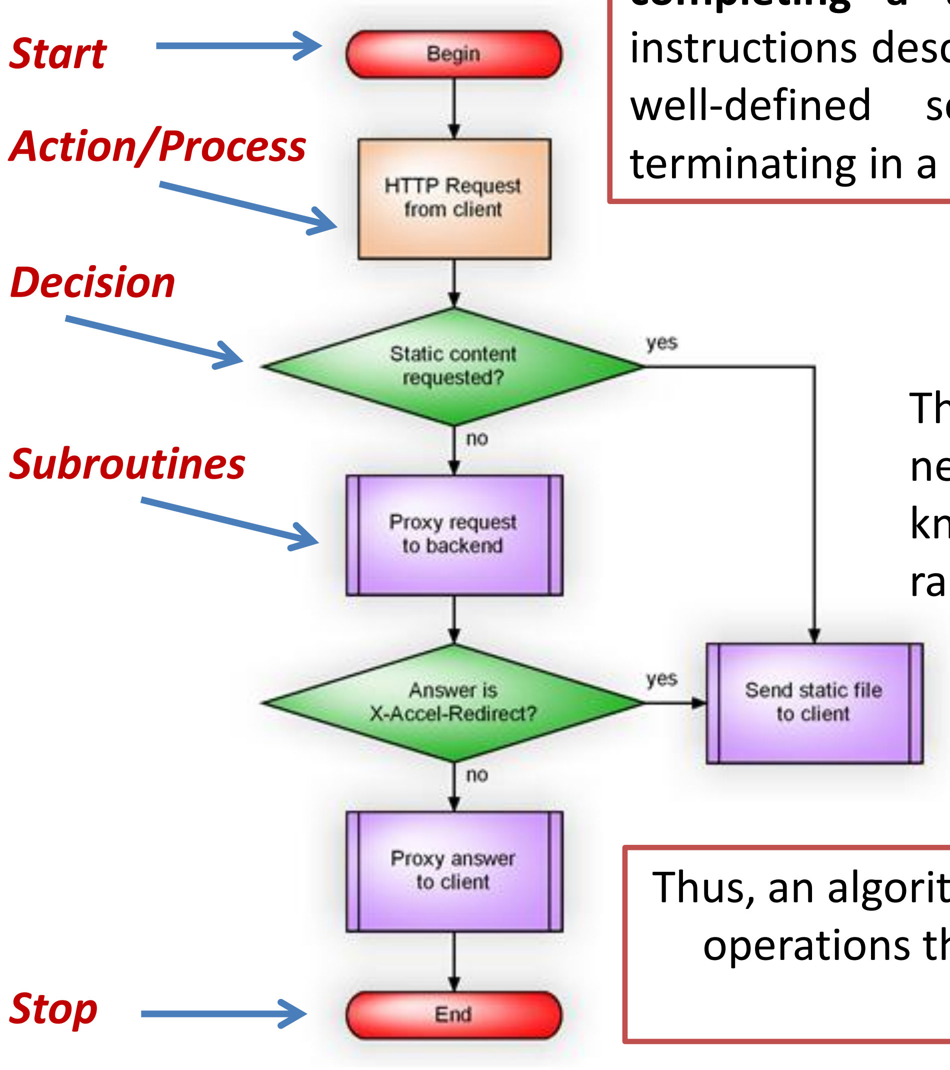
Algorithms are used for calculation, data processing, and many other fields.



An algorithm is a specific set of instructions for carrying out a procedure or solving a problem, usually with the requirement that the procedure terminate at some point. Specific algorithms sometimes also go by the name method, procedure, or technique. *The word "algorithm" is a distortion of al-Khwārizmī, a Persian mathematician who wrote an influential treatise about algebraic methods*. **The process of applying an algorithm to an input to obtain an output is called a computation**.

Algorithms are essential to the way computers process information. Many computer programs contain algorithms that specify the specific instructions a computer should perform (in a specific order) to carry out a specified task, such as calculating employees’ paychecks or printing students’ report cards.

Basic concepts



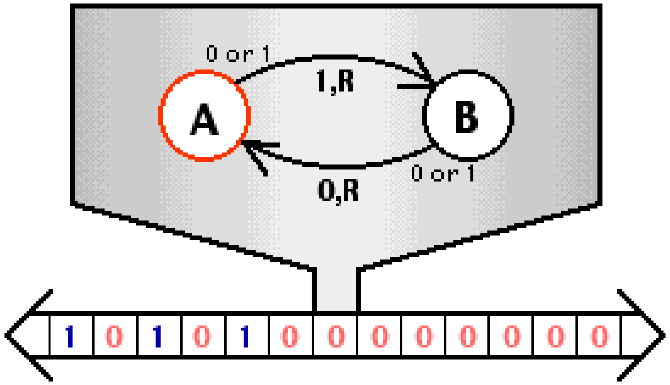
Each algorithm is a list of **well-defined instructions** for **completing a task**. Starting from an **initial state**, the instructions describe a computation that proceeds through a well-defined series of **successive states**, eventually terminating in a final **ending state**.

The transition from one state to the next is not necessarily deterministic; some algorithms, known as **randomized algorithms**, incorporate randomness.

Thus, an algorithm can be considered to be any sequence of operations that can be simulated by a **Turing-complete system**.

Turing machine

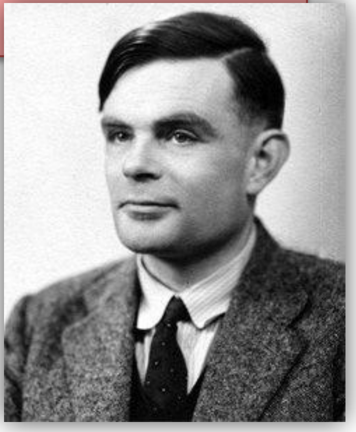
What is the Turing machine? It is **hypothetical computing device** proposed by ***Alan M. Turing***; it is not actually a machine, it is **an idealized mathematical model that reduces the logical structure of any computing device to its essentials**.



It consists of **an infinitely extensible tape**, **a tape head** that is capable of performing various operations on the tape, and **a modifiable control mechanism** in the head that can store instructions. As envisaged by Turing, it performs its functions in a **sequence of discrete steps**. His extrapolation of the essential features of information processing was instrumental in the development of modern digital computers, which share his basic scheme of:

- an input/output device (tape and tape reader),
- central processing unit (CPU, or control mechanism),
- and stored memory.

Alan Turing visualized a "**state machine**" that was capable of basic computing. He never actually built his Turing Machine but people have paid homage to him by creating modern examples of the machine that he imagined. **A state machine is a device which is controlled by the "current state" and a set of instructions which determines the "next state"**. In other words, a prototype for the computers of today.



Expressing algorithms

Algorithms can be expressed in many kinds of notation, including **natural languages**, **pseudocode**, **flowcharts**, and **programming languages**.

Natural language expressions of algorithms tend to be verbose and ambiguous, and are rarely used for complex or technical algorithms.

Pseudocode **and** **flowcharts** are structured ways to express algorithms that avoid many of the ambiguities common in natural language statements, while remaining independent of a particular implementation language.

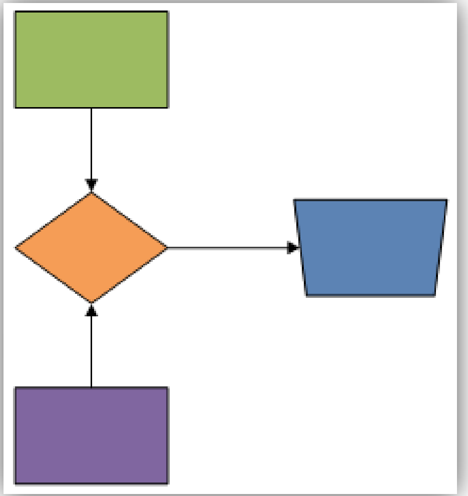
Pseudocode is a computing notation resembling a simplified programming language, used in program design.

Euclidian algorithm

```
Input: Positive integers $a$, $b$, $a \geq b$.
Output: $\gcd(a,b)$, $x$, $y$, with agreement $ax + by = \gcd(a,b)$.

1. $U \leftarrow (a,1,0)$, $V \leftarrow (b,0,1)$.
2. while $(v_1 \neq 0)$ do
3.     $q \leftarrow u_1/v_1$;
4.     $T \leftarrow (u_1 \bmod v_1,\ u_2 - qv_2, u_3 - qv_3)$;
5.     $U \leftarrow V, V \leftarrow T$.
6. end
7. return $U = (\gcd(a,b), x, y)$
```

!



Flow chart is a graphical representation of a computer program in relation to its sequence of functions (as distinct from the data it processes).

Expressing algorithms

Programming languages are primarily intended for expressing algorithms in a form that can be executed by a computer, but are often used as a way to define or document algorithms.

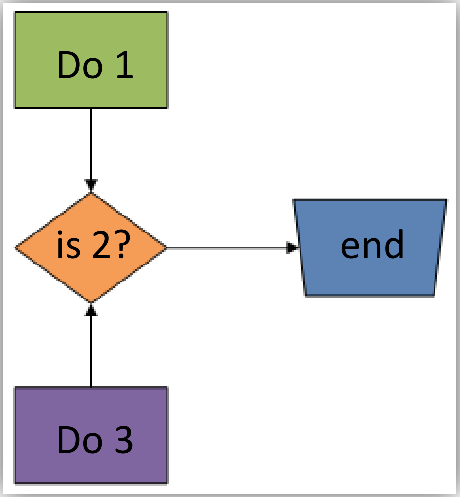
```
int Euclid(int value, int module)
{
 int u1=value, u2=1, u3=0, v1=module, v2=0, v3=1, q, t1, t2, t3;

 while (v1 != 0) {
   q = u1 / v1;
   t1 = u1%v1; t2 = u2-q*v2; t3 = u3-q*v3;               // T = U
   u1 = v1; u2 = v2; u3 = v3; v1 = t1; v2 = t2; v3 = t3; // U = V, V = T
   if ((u1 % v1) == 0) {
     if (v2 < 0) return (v2+module);
     return v2;
   }
 }

 return 0;
}
```

There is a wide variety of representations possible and one can express a given Turing machine program as a **sequence of machine tables**, as flowcharts, or as a form of **rudimentary machine code** or **assembly code** called "sets of quadruples".

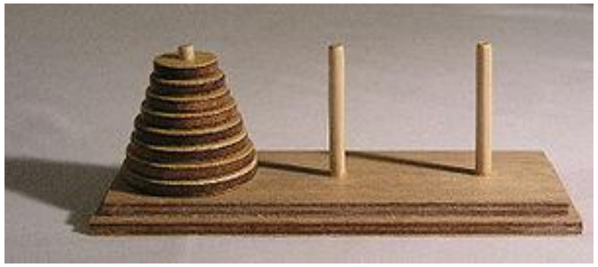
Sometimes it is helpful in the description of an algorithm to supplement small "flow charts" (state diagrams) with natural-language and/or arithmetic expressions written inside "block diagrams" to summarize what the "flow charts" are accomplishing.



Classification of algorithms (by implementation)

Recursion or iteration: A recursive algorithm is one that invokes (makes reference to) itself repeatedly until a certain condition matches, which is a method common to functional programming. Iterative algorithms use repetitive constructs like loops and sometimes additional data structures like stacks to solve the given problems. Some problems are naturally suited for one implementation or the other.

Every recursive version has an equivalent (but possibly more or less complex) iterative version, and vice versa.



Logical: An algorithm may be viewed as **controlled logical deduction**. This notion may be expressed as: **Algorithm = logic + control**. The logic component expresses the axioms that may be used in the computation and the control component determines the way in which deduction is applied to the axioms.

Serial or parallel or distributed: Algorithms are usually discussed with the assumption that computers execute one instruction of an algorithm at a time. Those computers are sometimes called **serial computers**. An algorithm designed for such an environment is called a serial algorithm, as opposed to parallel algorithms or distributed algorithms. **Parallel algorithms** take advantage of computer architectures where several processors can work on a problem at the same time, whereas distributed algorithms utilize multiple machines connected with a network. **Parallel or distributed algorithms** divide the problem into more symmetrical or asymmetrical subproblems and collect the results back together.

Classification of algorithms (by design paradigm)

Another way of classifying algorithms is by their **design methodology** or **paradigm**. There is a certain number of paradigms, each different from the other. Furthermore, each of these categories will include many different types of algorithms.

Brute-force or **exhaustive search**. **This is the naive method of trying every possible solution to see which is best**.

Divide and conquer. This algorithm repeatedly reduces an instance of a problem to one or more smaller instances of the same problem (usually recursively) until the instances are small enough to solve easily. One such example of divide and conquer is merge sorting. Sorting can be done on each segment of data after dividing data into segments and sorting of entire data can be obtained in the conquer phase by merging the segments.

Dynamic programming. If the optimal solution to a problem can be constructed from optimal solutions to subproblems, and overlapping subproblems, meaning the same subproblems are used to solve many different problem instances, a quicker approach called **dynamic programming**. It avoids recomputing solutions that have already been computed. *For example, the shortest path to a goal from a vertex in a weighted graph can be found by using the shortest path to the goal from all adjacent vertices*.

Linear programming. When solving a problem using linear programming, specific inequalities involving the inputs are found and then an attempt is made to maximize (or minimize) some linear function of the inputs. Many problems (such as the maximum flow for directed graphs) can be stated in a **linear programming way**, and then be solved by a 'generic' algorithm such as the simplex algorithm. A more complex variant of linear programming is called **integer programming**, where the solution space is restricted to the integers.

Classification of algorithms (by design paradigm)

Reduction. This technique involves solving a difficult problem by transforming it into a better known problem for which we have (hopefully) asymptotically optimal algorithms. The goal is to find a reducing algorithm whose complexity is not dominated by the resulting reduced algorithm's. For example, one selection algorithm for finding the median in an unsorted list involves first sorting the list (the expensive portion) and then pulling out the middle element in the sorted list (the cheap portion). This technique is also known as **transform and conquer**.

Search and enumeration. Many problems (such as playing chess) can be modeled as problems on graphs. A graph exploration algorithm specifies rules for moving around a graph and is useful for such problems. This category also includes search algorithms, branch and bound enumeration and backtracking.

The probabilistic and heuristic paradigm. Algorithms belonging to this class fit the definition of an algorithm more loosely.

1.Randomized algorithms are those that make some choices randomly (or pseudo-randomly); for some problems, it can in fact be proven that the fastest solutions must involve some randomness. There are two large classes of such algorithms:

- **Monte Carlo algorithms** return a correct answer with high-probability. E.g. RP is the subclass of these that run in polynomial time)
- **Las Vegas algorithms** always return the correct answer, but their running time is only bound in probaility, e.g. ZPP.

2.In optimization problems, heuristic algorithms do not try to find an optimal solution, but an approximate solution where the time or resources are limited. They are not practical to find perfect solutions. An example of this would be local search, tabu search, or simulated annealing algorithms, a class of heuristic probabilistic algorithms that vary the solution of a problem by a random amount.

Other classification of algorithms

By field of study. Every field of science has its own problems and needs efficient algorithms. Related problems in one field are often studied together. Some example classes are **search** algorithms, **sorting** algorithms, **merge** algorithms, **numerical** algorithms, **graph** algorithms, **string** algorithms, **computational geometric** algorithms, **combinatorial** algorithms, **machine learning**, **cryptography**, **data compression** algorithms and **parsing** techniques.

By complexity. Algorithms can be classified by the amount of time they need to complete compared to their input size. There is a wide variety: some algorithms complete in **linear time relative to input size**, some do so in an **exponential amount of time** or even worse, and some **never halt**. Additionally, some problems may have multiple algorithms of differing complexity, while other problems might have no algorithms or no known efficient algorithms. There are also mappings from some problems to other problems. Owing to this, it was found to be more suitable to **classify the problems** themselves instead of the algorithms into equivalence classes based on the complexity of the best possible algorithms for them.

By computing power. Another way to classify algorithms is by **computing power**. This is typically done by considering some collection (class) of algorithms. A **recursive class of algorithms** is one that includes algorithms for all Turing computable functions. Looking at classes of algorithms allows for the possibility of restricting the **available computational resources** (time and memory) used in a computation. A subrecursive class of algorithms is one in which not all Turing computable functions can be obtained. For example, the algorithms that run in polynomial time suffice for many important types of computation but do not exhaust all Turing computable functions. The class of algorithms implemented by **primitive recursive functions** is another subrecursive class.

Programming languages

!

A programming language is an artificial language designed to express computations that can be performed by a machine, particularly a computer. Programming languages can be used to create programs that control the behavior of a machine, to express algorithms precisely, or as a mode of human communication.

A **programming language** is a notation for writing **programs**, which are specifications of a computation or algorithm. Some, but not all, authors restrict the term "programming language" to those languages that can express all possible algorithms. ***Traits often considered important for what constitutes a programming language include***:

Function and target: A computer programming language is a language used to write computer programs, which involve a computer performing some kind of computation or algorithm and possibly control external devices such as printers, disk drives, robots, and so on. For example **PostScript** programs are frequently created by another program to control a computer printer or display.

Abstractions: Programming languages usually contain abstractions for defining and manipulating data structures or controlling the flow of execution. The practical necessity that a programming language support adequate abstractions is expressed by the abstraction principle; this principle is sometimes formulated as recommendation to the programmer to make proper use of such abstractions.

Expressive power: The theory of computation classifies languages by the computations they are capable of expressing. All Turing complete languages can implement the same set of algorithms. **ANSI/ISO SQL** and **Charity** are examples of languages that are not Turing complete, yet often called programming languages.

Generations of programming languages

Programming languages have been classified into several **programming language generations**. Historically, this classification was used to indicate increasing power of programming styles. Later writers have somewhat redefined the meanings as distinctions previously seen as important became less significant to current practice.

A **first-generation programming language** **is a machine-level programming language**. Originally, no translator was used to compile or assemble the first-generation language. The first-generation programming instructions were entered through the front panel switches of the computer system.

The **main benefit** of programming in a first-generation programming language is that **the code a user writes can run very fast and efficiently**, since it is directly executed by the CPU. However, machine language is a lot more difficult to learn than higher generational programming languages, and it is far more difficult to edit if errors occur.

Second-generation programming language is a generational way to categorize assembly languages. The term was coined to provide a distinction from higher level third-generation programming languages (3GL) such as COBOL and earlier machine code languages. Second-generation programming languages have the following ***properties***:

- **The code can be read and written by a programmer. To run on a computer it must be converted into a machine readable form, a process called assembly.**
- **The language is specific to a particular processor family and environment.**

Second-generation languages are sometimes used in kernels and device drivers (though **C** is generally employed for this in modern kernels), but more often find use in extremely intensive processing such as games, video editing, graphic manipulation/rendering.

Generations of programming languages

A **third-generation programming language (3GL)** is a refinement of a second-generation programming language. Whereas a second generation language is more aimed to fix logical structure to the language, a third generation language aims to refine the usability of the language in such a way to make it more user friendly. This could mean **restructuring categories of possible functions to make it more efficient, condensing the overall bulk of code via classes** (eg. Visual Basic). A third generation language improves over a second generation language by having more refinement on the usability of the language itself from the perspective of the user.

First introduced in the late 1950s, Fortran, ALGOL and COBOL are early examples of this sort of language. Most "modern" languages (BASIC, C, C++, C#, Pascal, and Java) are also third-generation languages. Most 3GLs support **structured programming**.

A **fourth-generation programming language** (1970s-1990) (abbreviated 4GL) is a programming language or programming environment designed with a specific purpose in mind, such as the development of commercial business software. In the evolution of computing, the 4GL followed the 3GL in an upward trend toward **higher abstraction** and **statement power**. The 4GL was followed by efforts to define and use a 5GL.

The natural-language, block-structured mode of the third-generation programming languages improved the process of software development. However, 3GL development methods can be slow and error-prone. It became clear that some applications could be developed more rapidly by adding a higher-level programming language and methodology which would generate the equivalent of very complicated 3GL instructions with fewer errors. In some senses, **software engineering arose to handle 3GL development**. 4GL and 5GL projects are more oriented toward problem solving and systems engineering.

Generations of programming languages

A **fifth-generation programming language** (abbreviated 5GL) is a programming language based around solving problems using constraints given to the program, rather than using an algorithm written by a programmer. Most constraint-based and logic programming languages and some declarative languages are fifth-generation languages.

While fourth-generation programming languages are designed to build specific programs, **fifth-generation languages are designed to make the computer solve a given problem without the programmer**. This way, the programmer only needs to worry about what problems need to be solved and what conditions need to be met, without worrying about how to implement a routine or algorithm to solve them. Fifth-generation languages are used mainly in **artificial intelligence research**. Prolog, OPS5, and Mercury are examples of fifth-generation languages.

Most common used programming paradigms

Imperative programming. In computer science, **imperative programming** is a programming paradigm that **describes computation in terms of statements that change a program state**. In much the same way that imperative mood in natural languages expresses commands to take action, imperative programs define sequences of commands for the computer to perform.

Procedural programming can sometimes be used as a synonym for imperative programming (specifying the steps the program must take to reach the desired state), but can also refer to a programming paradigm **based upon the concept of the procedure call**. **Procedures, also known as routines, subroutines, methods, or functions** (not to be confused with mathematical functions, but similar to those used in functional programming) **simply contain a series of computational steps to be carried out**. Any given procedure might be called at any point during a program's execution, including by other procedures or itself. A procedural programming language provides a programmer a means to define precisely each step in the performance of a task.

Object-oriented programming (OOP) is a programming paradigm that uses "objects" – data structures consisting of datafields and methods – and their interactions to design applications and computer programs. Programming techniques may include features such as information hiding, data abstraction, encapsulation, modularity, polymorphism, and inheritance. It was not commonly used in mainstream software application development until the early 1990s. Many modern programming languages now support OOP.

Concurrent computing is a form of computing in which **programs are designed as collections of interacting computational processes that may be executed in parallel**. Concurrent programs can be executed sequentially on a single processor by interleaving the execution steps of each computational process, or executed in parallel by assigning each computational process to one of a set of processors that may be close or distributed across a network.

Generations of programming languages

Event-driven programming. In computer programming, event-driven programming or event-based programming **is a programming paradigm in which the flow of the program is determined by events** – i.e., sensor outputs or user actions (mouse clicks, key presses) or messages from other programs or threads.

Functional programming is a programming paradigm that **treats computation as the evaluation of mathematical functions and avoids state and mutable data**. It emphasizes the application of functions, in contrast to the imperative programming style, which emphasizes changes in state.

Functional programming languages, especially purely functional ones, have largely been emphasized in academia rather than in commercial software development. However, prominent functional programming languages such as **Scheme**, **Erlang**, and **Haskell**, have been used in industrial and commercial applications by a wide variety of organizations. Functional programming also finds use in industry through domain-specific programming languages like **R** (statistics), **Mathematica** (symbolic math), and **XSLT** (XML). **Spreadsheets** can also be viewed as functional programming languages.

Metaprogramming **is the writing of computer programs that write or manipulate other programs (or themselves) as their data, or that do part of the work at compile time that would otherwise be done at runtime**. In many cases, this allows programmers to get more done in the same amount of time as they would take to write all the code manually, or it gives programs greater flexibility to efficiently handle new situations without recompilation.

The language in which the **metaprogram** is written is called the **metalanguage**. The language of the programs that are manipulated is called the object language. The ability of a programming language to be its own metalanguage is called **reflection** or **reflexivity**.

Programming languages

Language	Intended use	Paradigm	Failsafe
ActionScript 3.0	Web, client-side	imperative, object-oriented, event-driven	Yes
Ada	Application, Embedded, System and Realtime	imperative, procedural, concurrent, distributed, generic, object-oriented	Yes
ALGOL 60	Application	imperative	Yes
Assembly language	General		No
BASIC	Application, Education	imperative, procedural	Yes, stop
BlitzMax	Application, Game	imperative, procedural, object-oriented	Yes
C	System	imperative, procedural	No
C++	Application; System	imperative, procedural, object-oriented, generic	No
C#	Application	imperative, object-oriented, functional, generic, reflective	Yes
COBOL	Application, Business	imperative, object-oriented	Yes
Common Lisp	General	imperative, functional, object-oriented	Not def
Erlang	Application, Distributed and Telecom	functional, concurrent, distributed	Yes
FORTRAN	Application, scientific and engineering	imperative, procedural, object-oriented	Yes
Game Maker Lang	Application, games	imperative, object-oriented, event-driven	Yes

Programming languages

Language	Intended use	Paradigm	Failsafe
Java	Application, Web	imperative, object-oriented, generic, reflective	Yes
JavaScript	Web, client-side	imperative, object-oriented, functional, reflective	Yes
Mathematica	Highly domain-specific, Math	procedural, functional	Yes
MATLAB M-code	Highly domain-specific, Math	imperative, object-oriented	Yes
Object Pascal (Delphi)	Application	imperative, object-oriented, generic, event-driven	Yes
Oxygene	Application	imperative, object-oriented, generic	Yes
Pascal	Application, Education	imperative, procedural	Yes
Perl	Application, Text processing, Scripting, Web	imperative, procedural, reflective, functional, object-oriented, generic	Optional
PHP	Web, server-side	imperative, procedural, object-oriented, reflective	Yes
Prolog	Application, Artificial intelligence	logic	Yes
Visual Basic	Application, Education	imperative, component-oriented, event-driven	Yes

Application Programming Interface

Application programming interface (API) **is an interface in computer science that defines the ways by which an application program may request services from libraries and/or operating systems**. An API determines the vocabulary and calling conventions the programmer should employ to use the services. It may include specifications for routines, data structures, object classes and protocols used to communicate between the requesting software and the library.

An API may be:

- **Language-dependent**; that is, **available only in a given programming language**, using the syntax and elements of that language to make the API convenient to use in this context.
- **Language-independent**; that is, **written in a way that means it can be called from several programming languages (typically an assembly or C interface)**. This is a desired feature for a service-style API that is not bound to a given process or system and is available as a remote procedure call.

An API itself is largely abstract in that it specifies an interface and controls the behavior of the objects specified in that interface. The software that provides the functionality described by an API is said to be an implementation of the API. An API is typically defined in terms of the programming language used to build the application. The related term **application binary interface** (ABI) is a lower level definition concerning details at the assembly language level.

For example, the Linux Standard Base is an ABI, while POSIX is an API.

Thanks for attention