Programming, Languages

A programming language is a language; an abstraction of thought applied to a specific domain.

Alan Turing pioneered the idea of a programming language in his 1936 paper “On Computable Numbers, with an Application to the Entscheidungsproblem”. There he developed the idea of a notational system that could represent any other system by writing, erasing and storing a limited number of symbols; a 'universal computing machine'.
A programming language is a way to abstract the features of a problem from the implementation of the solution. It is a way of telling a machine what to do. The need to precisely tell a computer what to do and the desire to make the instructions easily readable to a human being are difficult to reconcile.

Today, programming languages are used and developed mostly for specific applications, but research continues into novel forms of thought representation. Programming languages have also entered the cultural domain. With the ability to wrestle with the language in which they are written, just as much contemporary literature does, programming languages are recognized as a form of cultural expression.
The only language that a computer can understand is machine language. Machine language is a set of basic operations whose execution is implemented in the hardware of the processor.

High level programming languages provide a machine-independent level of abstraction that is higher than the machine language. They are more adapted to a human-machine interaction. But this also implies that there is a sort of translator between the high level programming language and the machine languages. There exists two sorts of translators:

Compiler: A Compiler is a program that translates code of a programming language in machine code. Machine code can be executed directly on the machine where it was compiled.

Interpreter: An Interpreter is a program that implements or simulates a virtual machine using the base set of instructions of a programming language as its machine language.
Compiler (top) versus Interpreter (bottom)
Programming, Languages

The main two programming approaches of interest in this course are imperative vs declarative programming concepts.

We will use arduino (frontend in java, programs in c++) and python as examples of imperative languages and SQL (Structured Query Language) as an example of a declarative programming concept.
Procedural Programming

Procedural programming (often used as synonym for imperative programming) implements the idea of procedures/subroutines and variables local to a procedure. This approach limits access of other procedures to these variables and allows the building of reusable libraries and modular programming code.

Common procedural programming languages include C, Fortran, Java, Visual Basic.
Declarative Programming

A declarative program is context independent; it only specifies WHAT the program is to accomplish, but not how. Declarative languages consider programs as an expression of logic.

The declarative programming paradigm stipulates that an operation, called with the same arguments, returns the same results independent of any other computation state.

Declarative languages do not have looping control structures, declarative expressions express only the logical relationships of the constituent parts (not changeable variables).

Common declarative languages include those of database query languages (SQL), HTML, regular expressions and logic programming.
Functional Programming

Functional programming is a special kind of declarative programming that emphasizes mathematical functions. These produce results that depend only on their inputs and not on the program state.

Furthermore, the output value of a function depends only on the arguments input to the function; repeated calls to a function \( f \) with the same value for an argument \( x \) will produce the same result \( f(x) \). As the declarative programming paradigm, functional programming avoids state and mutable data.
Object Oriented Programming

OOP adds modularity to code design with the introduction of 'objects': a combination of data and functions. Other important concepts in OOP include:

- Inheritance (extending one object through another)
- Polymorphism (multiple forms of one function)
- Encapsulation (restricting access to some parts of an object's components)
- Dynamic dispatch (selecting which function to call at runtime)
The principal programming paradigms

"More is not better (or worse) than less, just different."

Data structures only
Turing equivalent

Observable nondeterminism? Yes No

First-order functional programming

+ procedure

Functional programming

+ closure

+ unification (equality)

Deterministic logic programming

+ search

Relational & logic programming

+ solver

Constraint (logic) programming

+ search

Prolog, SQL embeddings

+ by-need synchron.

Lazy functional programming

Concurrent constraint programming

Concurrent constraint programming

LIFE, AKL

+ by-need synchron.

Lazy concurrent concurrent programming

Lazy dataflow programming

+ thread

Monotonic dataflow programming

Declarative concurrent programming

Lazy declarative concurrent programming

Nonmonotonic dataflow programming

Concurrent logic programming

Weak synchronous programming

Strong synchronous programming

Instantaneous computation

Functional reactive programming (FRP)

Esterel, Lustre, Signal

Dataflow and message passing

Named state

Message passing

Message-passing concurrent programming

Multi-agent dataflow programming

Oz, Alice, AKL

FrTime, SL

CSP, Occam, E, Oz, Alice, publish/subscribe, tuple space (Linda)

Software transactional memory (STM)

SQL embeddings

Shared state

Sequential object-oriented programming

Stateful functional programming

CLU, OCaml, Oz

+ port (channel)

Event-loop programming

Oz in one vat

+ thread

Multi-agent functional programming

Oz, Alice, Curry

+ thread

Pipes, MapReduce

+ single assignment

Functional

+ thread

Lazy dataflow programming

Functional and constraints

+ solver

Concurrent constraint programming

CLP, ILOG Solver

+ thread

Lazy declarative concurrent programming

Lazy functional programming

Deterministic logic programming

+ by-need synchron.

Unnamed state (seq. or conc.)

Nondet. state

Named state

Less declarative

More declarative
Most programming languages are designed for clarity and efficiency. Weird programming languages are not designed for any real-world application; rather, they are intended to test the boundaries of programming language design. Brainfuck is one such example: an esoteric programming language famous for its minimalism and borderline illegibility.

Brainfuck consists of only eight commands. The commands are executed sequentially; an instruction pointer begins at the first command, and each command it points to is executed, after which it normally moves forward to the next command. The program terminates when the instruction pointer moves past the last command.
Brainfuck

> increment the data pointer (to point to the next cell to the right).
< decrement the data pointer (to point to the next cell to the left).
+ increment (increase by one) the byte at the data pointer.
- decrement (decrease by one) the byte at the data pointer.
. output the byte at the data pointer.
, accept one byte of input, storing its value in the byte at the data pointer.
[ if the byte at the data pointer is zero, then instead of moving the instruction pointer forward to the next command, jump it forward to the command after the matching ] command.
] if the byte at the data pointer is nonzero, then instead of moving the instruction pointer forward to the next command, jump it back to the command after the matching [ command.

Source:
http://www.muppetlabs.com/~breadbox/bf/
The following program prints "Hello World!" and a newline to the screen:

```
+++++ ++++
 initialize counter (cell #0) to 10
[ 
    > ++++ +
    > ++++ ++++
    > ++++
    > +
    <<< -
    decrement counter (cell #0)
]
> ++
> + ,
+++++ ++ ,
. 
+++ ,
> ++ ,
<< ++++ ++++ ++++
> . 
++++ ,
----- - ,
----- --- ,
> + ,
> . 

------------[>+++++<++++++++++><+++><<<<<-]>++.>+.++++++++.+++>.+++.<<<<<<<<<<<<<<<+>.>++.------------.--.----.----.--.--.--.--.--.--.--.--.>.
```
Brainfuck

Line 1 initializes $a[0] = 10$ by incrementing ten times from 0.
Line 2 sets the initial values for the array: $a[1] = 70$ (close to 72, the ASCII 'H')
Line 3 sets $a[2] = 100$ (close to 101 or 'e')
Line 4 sets $a[3] = 30$ (close to 32, the code for space) and

The loop works by adding 7, 10, 3, and 1, to cells $a[1]$, $a[2]$, $a[3]$ and $a[4]$ respectively each time through the loop; 10 additions for each cell in total. After the loop is finished, $a[0]$ is zero. The command '>++' then moves the pointer to $a[1]$, which holds 70, adds two to it, totaling 72, which is the ASCII character code of a capital H, and outputs it.....

...you get the idea....