Intro

Awesomeness begins

What is programming?

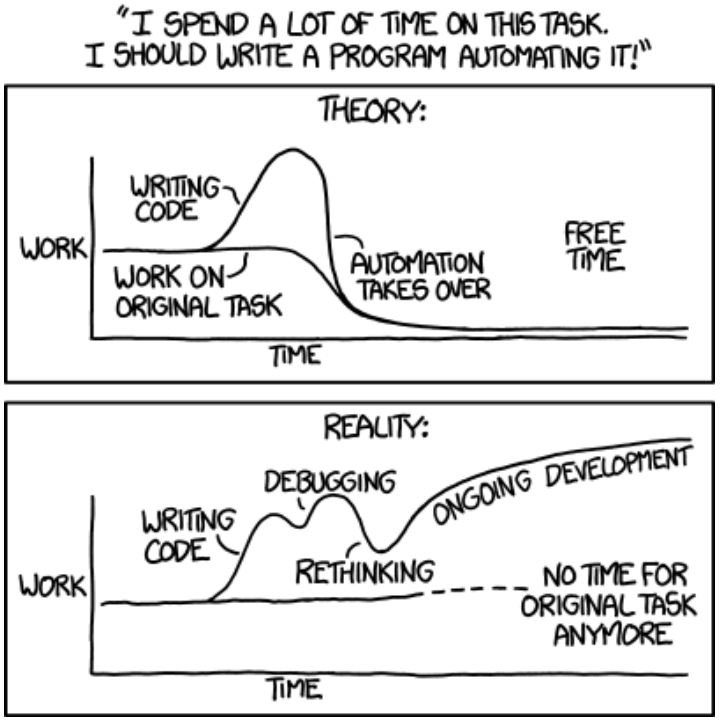
> “ Programming is the act of giving a machine a set of instructions to enable it to perform a certain task.

The computer mindlessly does only and exactly what it's told, provided it's told in a language it understands.

Why do we program?

Computers can do basic computations very fast (much faster than humans, billions per second). That's useful for:

- executing tedious tasks automatically ("booooring, let the machine do it")
  - counting words in a book
  - searching in databases
  - writing personalized email to a thousand people
  - solving sudoku
- solving problems that we know *how* to solve, but we never *actually* solved because it would take more than a lifetime
  - finding the quadrillionth decimal of π (spoiler: it's a 0)
  - enumerating mathematical objects (like trees)
  - cracking cyphers


"I SPEND A LOT OF TIME ON THIS TASK.
I SHOULD WRITE A PROGRAM AUTOMATING IT!"
THEORY:
WORK
WRITING CODE
WORK ON ORIGINAL TASK
AUTOMATION TAKES OVER
FREE TIME
TIME
REALITY:
WORK
WRITING CODE
DEBUGGING
RETHINKING
ONGOING DEVELOPMENT
NO TIME FOR ORIGINAL TASK ANYMORE
TIME

Why do we program? (2)

Computers also have interesting input/output devices, which give them access to "non-human" powers, and by extension, gives humans non-human powers:

- video games
- computer generated imagery
- sound synthesis
- networking

Anything you can think of, you can program and *make it happen*. Through language you can make worlds appear and set them in motion.

> I'm a god!!

What can't we program?

Computers are very fast for very simple tasks, which can be combined in less simple tasks that are necessarily slower. Very complex or "unnatural" tasks are typically slow, if even possible. Brains are very parallel in nature, and some tasks that are trivial for a human being are typically hard for computers:

- recognizing shapes in a picture
- finding an object in a collection in an instant
- doing anything "intelligent"

If you can break up a problem in simple sub-problems, odds are you can write a program to solve it. Otherwise, you'll probably have to cheat.

Programming languages

Basically, we tell the computer what to do. To "tell" we need to write. To write we need a common language.

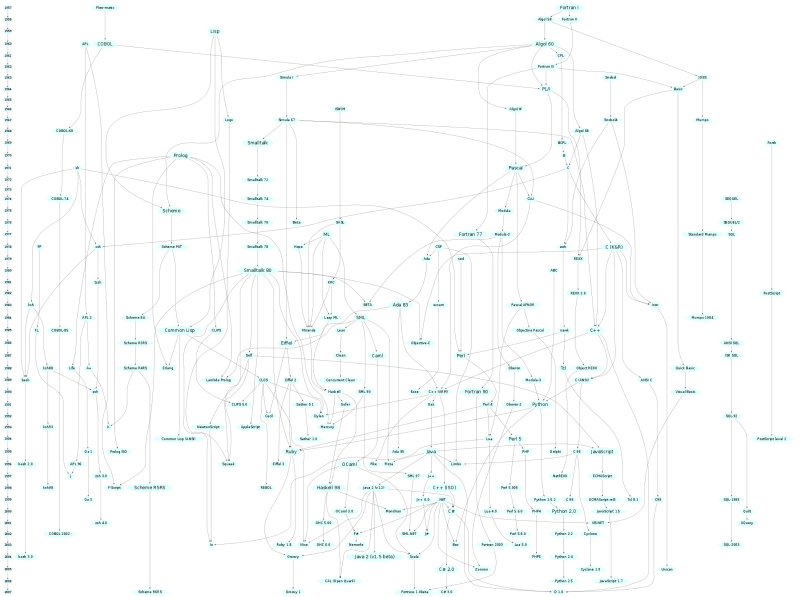
Wikipedia has a list of:

> all notable programming languages in existence, both those in current use and historical ones, in alphabetical order, except for dialects of BASIC and esoteric programming languages.

It's 644 items long. There are many way to categorize and compare these languages. Wikipedia's list of programming languages by category has 42 categories.

You can also sort them by family, which influenced which, which merged to give birth to others, etc.

So, where do we start?



General purpose languages

We want to make programs. Native desktop or mobile applications. For that we'll focus on general purpose languages. We can already remove:

- web oriented languages (JavaScript, PHP)
- scientific languages (MATLAB, R, Prolog)
- application specific (VBA)

There are many programming paradigms, or philosophies behind the design of languages. Imperative, functional, object-oriented, concurrent, logic, etc. Not mutually exclusive!

Imperative: statements that change a program state ("do this, do that, if this then do that, etc.")

High level, low level

One of the many ways to measure and classify languages is "how close to the machine" they are.

At the lowest level:

- machine code (the *only* language a processor actually understands)
- assembly (machine code but in human readable form. Very difficult to write what you mean, very easy to make mistakes. Basically impossible to maintain.)

```
D01H    equ 00DE0B6B3h
D01L    equ 0A7640000h
QtoA:
    sub esp,12
    xor ecx,ecx
    sub eax,D01L
    sbb edx,D01H
    jb    @@a01f
    sub eax,D04L
    sbb edx,D04H
    jb    @@a05
    mov cl,05h
```

High level, low level (2)

Higher (disguised assembly, plus bonuses to make life easier):

- C, C++ (compiled languages, quite verbose, strict syntax, still very lax in terms of memory safety, error prone, but have many higher level constructs that makes coding more "logical" and closer to thought processes)

```c
int main()
{
    int a, b;
    srand(time(NULL));
    while (1) {
        a = rand() % 20;
        printf("%d\n", a);
        if (a == 10) break;
        printf("%d\n", b);
    }
    return 0;
}
```

High level, low level (3)

At the highest level:

- Python, Ruby, Lua (interpreted languages, concise syntaxes, very dynamic, they allow to do pretty much anything and it works, memory safe)

```
repeat
    k = math.random(19)
    print(k)
    if k == 10 then break end
    print(math.random(19))
until false
```

At first sight, why bother with lower languages, they seem horrible, while the higher level ones are dreamy? Speed. High level languages have more layers of things going on, taking care of all the things "behind the scene" for you, and that takes ressources. They can be as much as 10~100 times slower than low-level languages.

High level, low level (4)

What's a good compromise?

- new modern languages try to combine efficiency of low-level languages with ease of use of higher level languages (D, Rust, Go). Still a work in progress.
- big projects combine and use different languages for different purposes. World of Warcraft is mainly in C++, with probably a bit of assembly for very speed critical code, and Lua for scripting and user interface.

Compare solutions for the N-queens problem on Rosetta Code.

Compiled vs. interpreted

Compiled language

Development:

- write source code
- give source code to compiler (a program on the developer's machine)
- compiler converts source to machine code, thus creating a program
- distribute program

Use:

- ask OS to run program
- machine code is loaded into memory, processor executes instructions directly

(C, C++, D, Rust, Go...)

Compiled vs. interpreted (2)

Interpreted language ("scripting language")

Development:

- write source code
- distribute source code

Use:

- give source code to interpreter (a program on the user's machine)
- interpreter builds in memory a high-level representation of the source
- interpreter follows this representation and executes commands

Interpreter = simulation of computer (slow). Often embedded in another program (game, application). Fast enough for most purposes.

(Lua, Python, Ruby, JavaScript...)

Compiled vs. interpreted (3)

Bytecode

Development:

- write source code
- give source code to compiler
- compiler converts source to bytecode, creating a "program"
- distribute "program"

Use:

- give "program" to interpreter ("virtual machine", VM)
- interpreter executes bytecode instructions one by one

(Java, C#, ActionScript...)

Compiled vs. interpreted (4)

Bytecode

Advantage:

- separate compilation: can optimize
- portable "program"
- bytecode instructions "almost" machine code: still pretty fast

In practice, most interpreted languages also compile to bytecode, but there is no separate compilation step: program are distributed in source form (Lua, Python, Ruby...)

The final choice

"I want to learn to program. What language should I learn?"

Impossible to answer. Many programmers end up knowing several (many!), each adapted to different purposes. However, it is probably a lot harder to go from high-level to lower levels than the opposite. Lots of idioms in high level languages can be though of as "bad habits" that just won't work in lower level.

A compromise

Go through an introduction to the C language to:

- understand the memory model of a computer
- learn discipline in programming (because in C you can't do without)
- know what happens behind the scenes of higher level languages (and thus program better with them)
- realize that some seemingly trivial tasks are a pain in the butt

A compromise (2)

In parallel, or after, learn Lua:

- can be learned in an afternoon (and mastered in a few more)
- seemingly complex tasks can in fact be trivial!
- popularity increasing very fast
- gorgeous construct: the Lua table combines goodies from other languages
- the fastest of the "slow languages"
- available frameworks for desktop applications and video games
- my personal favorite :)

"That was fun, I want more."

- master the core concepts of C. You are now a real programmer.
- learn an object-oriented derivative of C
  - C++ is *the* most popular language to date
  - Objective-C for anything Apple related
  - C# for anything Microsoft related
  - D is "C++ done right", but is not so popular yet
- if nerd pulsions, take a look at assembly. Just for curiosity.
- whatever happens, don't learn Java.
- gratuitously diss Java programmers. If you don't know why, they do.
- create own programming language.
- abandon all hopes of having a social life again.
- create own operating system.
- reach singularity and access higher level of existence.

> I'm a god!!