Is this a Programming Language?

Why (not)?

What is a Programming Language?

"A set of conventions for communicating an algorithm" (Horowitz)

These conventions differ very greatly — broadly speaking, three basic paradigms today:

1. Procedural / Imperative (e.g., C, Fortran),
2. Functional (e.g., ML, LISP),
3. Logic (e.g., Prolog).

But many paradigmatic conventions cut through these distinctions, such as:

- message-passing / object-orientation,
- event-handling,
- concurrency / threading,
- domain-specificity,
- security.

What is a Programming Language?

"A set of conventions for communicating an algorithm" (Horowitz)

These conventions differ very greatly — broadly speaking, three basic paradigms today:

1. Procedural / Imperative (e.g., C, Python),
2. Functional (e.g., ML, OCaML, Haskell),
3. Logic (e.g., Prolog).

In this course, we will focus on functional and logic programming languages . . . as well as illustrate a lot of the principles behind the design of any PL.

So Why then is (practically) Everybody using Imperative PLs?

For a few reasons:

- Inertia: lots of code out there to maintain already.
- Lack of competent programmers: lots of the maintainers finished university a long time ago.
- Efficiency: there has been progress ...
  - some functional languages can stay within a factor of 2 of C,
  - almost every language can beat C at certain kinds of programs,

  but this is still a big problem.

And What's Wrong with Imperative PLs anyway?

- *not* expressive power — plenty of that;
- they specify *how* as well as *what* to compute — in many cases, *how* can be inferred;
- many "hows" can be subsumed under the description of a single "what," e.g., database access:
  - retrieve telephone number of Gerald Penn,
  - retrieve name of person at 978-7390;
- imperative programming languages are often (but incidentally) naïve in the methods they provide for articulating how;

And What's Wrong with Imperative PLs anyway?

- more advanced methods can result in code that is:
  - shorter,
  - better captures the intuitions of the designer(s),
  - easier to prove correct, e.g.:

$$fib(N) = \begin{cases} 0 & N = 0 \\ 1 & N = 1 \\ fib(N-1) + fib(N-2) & N > 1 \end{cases}$$

Declarative Programming

By contrast to imperative PLs, functional and logic PLs are more "declarative," e.g. in this linear system:

$$
\begin{aligned}
x + y &= 1 \\
x - y &= 2
\end{aligned}
$$

the solutions for x and y are implicit in these equations — even if we don't define determinants, implement Gaussian elimination, etc.

Both functional and logic PLs have extensions ("constraint" functional/logic programming) that allow you to specify these equations as your program — with the implicit request to find solutions for all of the variables.

Declarative Programming

Pure declarative languages don't even care about order. In an imperative language ...

`x:= 1;`	vs.	`x:= x + 1;`
`x:= x + 1`		`x:= 1`

Variables in pure declarative languages are *logical*, not nicknames for machine registers.

Properties of a Good PL

- Code should be easy to read and understand.
- Reflects intuitions of the programmer.
- No synonyms.
- Not many primitive concepts to master.
- *Orthogonality*: primitives combine cleanly and systematically — no exceptions.
- Meaning of construct (control and data) independent of context.
- Natural for intended applications
- Easy to learn.
- Efficient.
- Portable.
- …and more technical properties that we will discuss later.

Examples of lousy languages: BASIC, C++, Perl

Properties of a Good Programming Environment

- A good PL.
- Graphical IDE.
- Version control system.
- Profiler (and tools for diff'ing profiles).
- Issue tracking system.
- Dashboard: monitors status of builds, regular tests, team discussions, issue tracker, etc.
- Unit testing system and test suite creation.
- Coverage analyser.
- Source-code analyser.
- GUI testing system.

On this point, modern programming languages have lagged *way* behind until quite recently, in part thanks to better open-source collaboration.

In What Sense are PLs really Languages?

A *language* is an arbitrary assocation of a collection of forms with their meanings.

Syntax: the specification of the forms.
Semantics: the specification of the meanings.

We're actually not going to say much to formalize meaning in this course, but we've already seen a few different kinds:

- *denotational*: a declaration of what an expression means, e.g., $x + y = 1$ means that the value of x added to the value of y is the same as the value of this expression: 1.
- *operational*: an elucidation of what the programmer is asking us to do, e.g., $x := x + 1$ means we should look up the value stored in the location called x, add 1 to it, and store the result in the location called x.

In What Sense are PLs really Languages?

A *language* is an arbitrary assocation of a collection of forms with their meanings.

Syntax: the specification of the forms.
Semantics: the specification of the meanings.

We're actually not going to say much to formalize meaning in this course, but we've already seen a few different kinds:

But *remember*: both kinds of statements have both kinds of semantics. It's just that some PLs emphasize one more than another in how they're used.

Syntax

There are a few ways to think about syntax, too ...

- Grammars for string languages (e.g., regular grammars), or
- Specifications of form that abstract away from their realization as strings, e.g.:

(Infix) arithmetic:	$3 + (2 * 4) - 7$
Reverse Polish notation:	$3\ 2\ 4\ * + 7\ -$

Let's start with the former, using *context-free grammars*.