# Programming Languages

Unlike Friendship, It's Not Magic

# What is a programming language?

(Turing-complete edition)

# The hell does "Turing-complete" mean?

Simple! It's any system which can implement a universal Turing machine -- or, in more modern theoretical parlance, a Turing-Post machine. Duh.

# This is not helping!

I know, I know; I'm sorry.

# But seriously, what's "Turing-complete"?

Basically, it means "capable of performing any possible computation."

Or, to put it another way, "you can use it to solve any problem."

When programmers say "programming language," they almost always actually mean "a Turing-complete programming language."

# What do you need to be Turing-complete?

Surprisingly little:

- a way to store values
- a way to make decisions

Or, in the jargon:

- Variables
- Flow-control

That's it!

# Languages like this are "General Purpose"

C, C++, Java, Javascript, Perl, Python, Ruby, Scala, Clojure, Lisp, Scheme, and hundreds and hundreds more.

All are general-purpose languages, and so all can be used to (given enough time and effort) solve any given problem.

Not all problems can be solved with ease or elegance with all languages, however.

# What isn't a GP programming language?

## Markup languages

HTML, Markdown, XML*

## Data description languages

JSON, YAML

## Query languages

SQL

## Most constraint-based languages

CSS, Prolog

# Under the hood

How languages work

# There are two kinds of languages

- Compiled
- Interpreted

# Except that this is almost entirely false.

It definitely hasn't been true since the 1980s

And I'm not *sure* that it was true even then.

It's far more true to say that there are "languages which compile to native opcodes" and "those which do not."

But jargon aside, this is still rather vague (*e.g.* Java)

# So let's forget about it.

I find this traditional dichotomy pointlessly archaic at best; distracting and confusing at worst.

The same is true of almost all A/B or A/Not A classifications of languages.

# Some commonly "adversarial" paradigms

| | |
|---|---|
| Compiled | Interpreted |
| Procedural | Functional |
| Strongly typed | Weakly typed |
| Object-oriented | (Not) |

# Enough of that; time to look under the hood.

```
total = total + 5
print("Total is {}".format(total))
```

# 1. Source transformed to character stream

```
t, o, t, a, l, [SPC], =, [SPC], t,
o, t, a, l, [SPC], +, [SPC], 5,
[NL], p, r, i, n, t, (, ", T, o, t,
a, l, [SPC], i, s, [SPC], {, }, ",
., f, o, r, m, a, t, (, t, o, t, a,
l, ), [NL]
```

# 2. Stream is parsed into tokens

1. `t -> [o]`
2. `to -> [t]`
3. `tot->[a]`
4. `tota -> [l]`
5. `total -> [[SPC]]`

First token is `"total"`

# 2a. How the hell do we know that?

## The parser has rules for how the different sorts of tokens are constructed.

Identifiers are unlimited in length. Case is significant.
**identifier** ::= id_start | id_continue
**id_start** ::= <all characters in general categories Lu, Ll, Lt, Lm, Lo, Nl, the underscore>
**id_continue** ::= <all characters in id_start, plus characters in the categories Mn, Mc, Nd, Pc>

- *Lu* - uppercase letters
- *Ll* - lowercase letters
- *Lt* - titlecase letters
- *Lm* - modifier letters
- *Lo* - other letters
- *Nl* - letter numbers

- *Mn* - nonspacing marks
- *Mc* - spacing combining marks
- *Nd* - decimal numbers
- *Pc* - connector punctuations

# 2b. Every type of token should be distinct

That is, the rules for determining what type a given token is should have no ambiguity and no dependence on any *other* token definition.

When you achieve this, you have defined a *context-free grammar*.

# 2c. Context-free grammars are great

They are cheap (in terms of time and memory) to parse using *LALR parsers*.

LALR stands for *Look-Ahead Left-right Rightmost derivation.*

This is exactly what we just did with our character stream, and exactly what we did with it.

# 2d. Where does "Rightmost" come in?

`1. t -> [o]`

Could be an identifier, or could be the reserved keyword `"to"`. We have not yet resolved all ambiguity about this token's type.

`2. to -> [t]`

Must be an identifier.

*N.B. In actuality, this specific sort of determination is not made at this stage of source processing, but rather at the next stage, which we are just about to discuss. Please forgive my technical inaccuracy, made for didactic purposes and due to time issues.*

# 3. Tokens are given meaning

The tokens are now reassembled into meaningful chunks and/or subchunks: statements and expressions.

```
total # identifier
=     # operator
total # identifier
+     # operator
5     # numeric literal
```

# 3a. Tokens are given meaning (cont'd)

The = operator means this is an assignment statement.

```
total
=     <- assignment!
total
+
5
```

# 3b. Tokens are given meaning (cont'd)

Assignment statements have two parts: an *lvalue* and an *rvalue*.

```
total  = total + 5
------ - ---------
lvalue =  rvalue
```

The lvalue receives the value of the rvalue, which must itself be a value or an expression which evaluates to a value.

# 3c. Turns out we've been lexxing

Our rvalue is an expression, so at this point the process of *lexical analysis*, or, *lexxing*, which we have been performing on the statement as a whole, must be performed on the expression.

This process continues until all expressions and subexpressions have been lexxed, and the statement as a whole is validated and understood.

# 3d. Lex, rinse, repeat

```
total
+     <- addition operator!
5
```

Our rvalue expression is a simple addition expression. Addition takes two values (or expressions which evaluate to a value) and sums them. We have no sub-expressions; just an identifier (which holds a value) and a numeric literal (which intrinsically has a value).

These values are summed and used as the rvalue of the assignment expression.

# 3e. A statement is born

And so this source statement

```
total = total + 5
```

through lexical analysis of its tokens, becomes a stream of instructions with the meaning:

```
Sum the current value held by the variable
'total' with the value of the numeric
literal '5', and store the resulting value
in the variable 'total'.
```

# 4. From here on out it's details

Piles and piles of details, but for the purposes of this talk, everything that follows can be summed up as "and then you get executable code".

If you're interested in the specifics, check out the wiki articles on Abstract syntax trees, Compilers, Compiler optimization, and just about any entry you find linked from those.

# Compiled vs. "interpreted": compiled

This one really bothers me, so I'll wrap up here.

A compiled language is one which does everything we just talked about to the source code and eventually emits a version of the same program, written in the *assembly* of the CPU you're compiling on.

This asm is then *linked* with a *loader* to produce an *executable*.

# Compiled vs. "interpreted": interpreted

I don't believe there still exist any purely interpreted languages. Certainly not in anything resembling widespread use.

Pretty much everything out there that doesn't compile to native code compiles to bytecode (or an AST, or a combination thereof) for a given *virtual machine*.