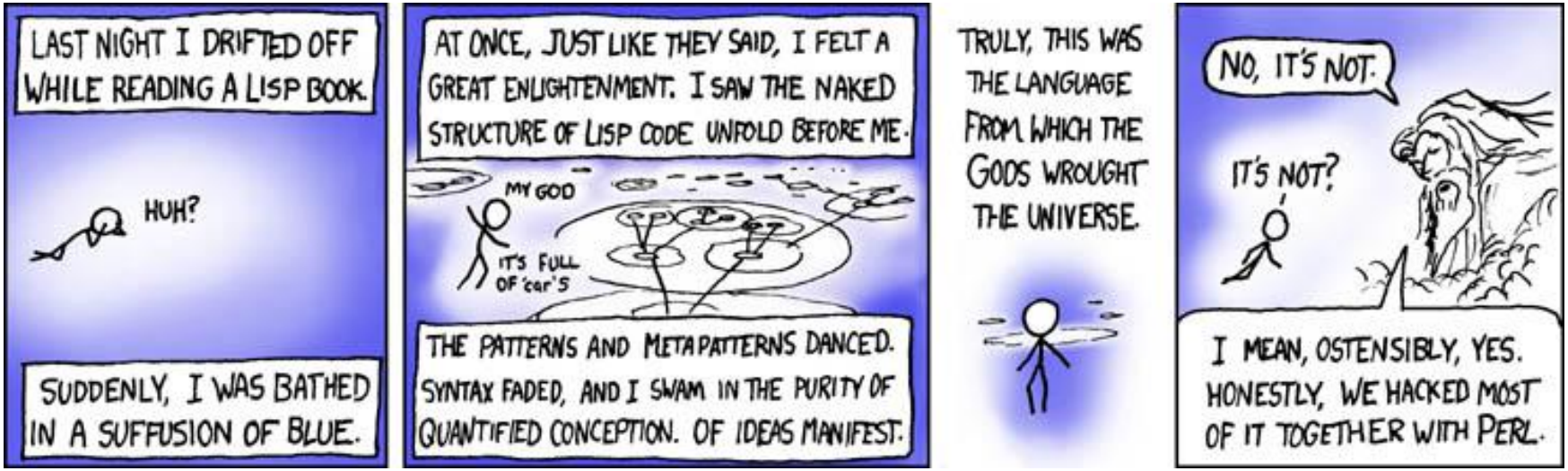
LISP

(Based on *The Roots of LISP*, Paul Graham, Jan 2002)



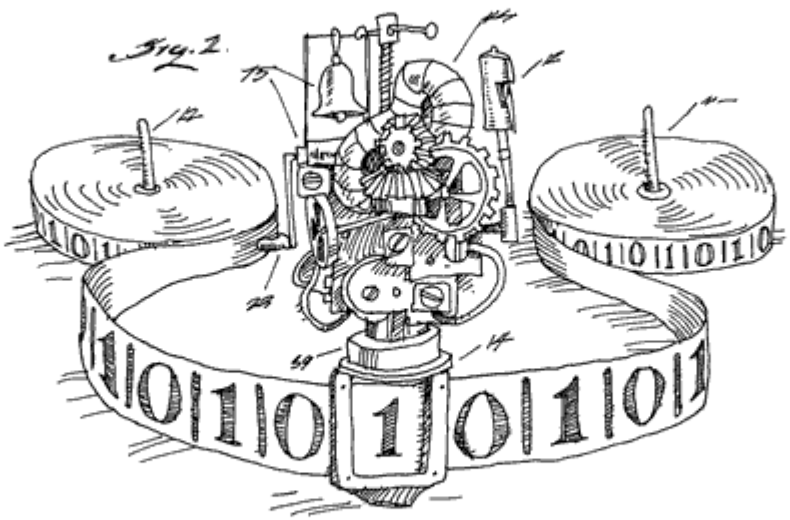
http://xkcd.com/224/

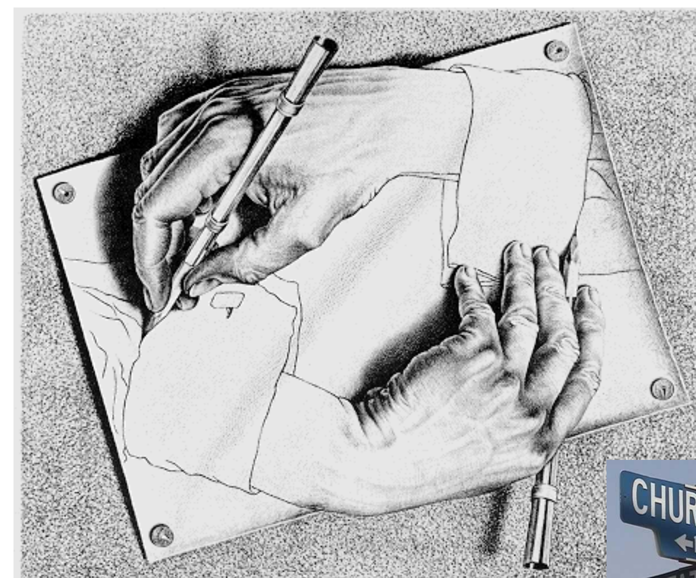
See also: http://xkcd.com/297/

The Church-Turing Thesis

- Already mentioned in the first lecture...
- All programming language have equal "power" (can represent the same set of computable functions).
- But in software engineering, we also care about a different meaning of "power".
  - Mostly, language expressiveness.
  - How different is the code from the concept?
  - Still other meanings I care less about: performance, ability to manipulate raw iron, etc.

The Church vs. Turing Shootout





Turing machine power == Lambda Calculus power

The Discovery of LISP

- "Programs with Common Sense"
  - John McCarthy, Nov. 1958
  - Symposium on the Mechanization of Thought Processes, National Physical Laboratory, Teddington, England.

- "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I"
  - John McCarthy, April 1960
  - Communications of the ACM

"Discovery," in the same sense of Pythagoreans discovering irrational numbers or Newton discovering calculus. Core LISP is a notation for mathematical objects.

Expressions

- An expression is either an **atom** or a **list** of zero or more expressions.

- `foo`
- `()`
- `(foo)`
- `(foo bar)`
- `(a b (c) d)`

Truth or Consequence

- The value "True" in LISP is represented using the atom `t`.
- The value "False", using the empty list: `()`

- These are just conventions -- many modern dialects use `t` and `nil`, or other alternatives.

"Evaluating" Expressions

If an expression is a list:

- The first element is the **operator**.
- Remaining elements (if any) are the **arguments**.

Seven Primitive Operators

- The basic blocks that make computing possible.
- The names of some of these operators are historic accidents, based on the computer architecture of the time.
  - IBM 704 CPU.

1. quote

- `(quote` *x*`)` returns *x*.
- Abbreviation: `'x`

```
> (quote a)
a
> 'a
a
(quote (a b c))
(a b c)
```

2. atom

- `(atom` *x*`)` returns the atom `t` if *x* is an atom or the empty list, `()` otherwise.

```
> (atom 'a)
t
> (atom '(a b c))
()
> (atom '())
t
```

Why do we need quote?

```
> (atom 'a)
t
> (atom a)
t
> (atom '(atom a))
()
> (atom (atom a))
t
> (atom '(a))
()
> (atom (a))
Invalid function call: a
```

3. eq

- `(eq` *a* *b*`)` returns `t` if *a* and *b* are both the same atom, or both the empty list; `()` otherwise.

```
> (eq 'a 'a)
t
> (eq '() '())
t
> (eq 'a 'b)
()
> (eq '(a) '(a))
()
```

4. car
5. cdr

- If *x* is a list, then `(car` *x*`)` returns the first element of the list, while `(cdr` *x*`)` returns everything after the first element.

```
> (car '(a b c))
a
> (cdr '(a b c))
(b c)
```

- car = content of address part of register; cdr = content of decrement part of register.

6. cons

- `(cons` *x* *y*`)`, where *x* is an atom and *y* is a list, returns a list composed of *x* followed by the elements of *y*.

```
> (cons 'a '(b c))
(a b c)
> (cons 'a (cons 'b (cons 'c '())))
(a b c)
> (car (cons 'a '(b c)))
a
> (cdr (cons 'a '(b c)))
(b c)
```

7. cond

- `(cond (`p_1 e_1`) ... (`p_n e_n`))` starts evaluating p_1, p_2, ... until some p_i returns `t`. Then the whole expression returns e_i.

```
> (cond ((eq 'a 'b) 'first)
        ((atom 'a) 'second))
second
```

Denoting Functions

- A *lambda expression* denotes a function. The expression structure is:

  `(lambda (`p_1 ... p_n`) e)`
- The expression *e* can refer to the parameters p_1...p_n.

```
(lambda (x) (cons x '(b)))
```

Function Calls

- If *f* is a lambda expression (with parameters $p_1 ... p_n$ and expression *e*), we can use it as the operator in a list.
- The basic syntax is: $(f \ a_1 \ ... \ a_n)$
  - Just as with the built-in primitive operators.
- Arguments $a_1 ... a_n$ are evaluated first, then *e* is evaluated with p_i assigned the value of a_i for each *i*.
- But how can we make *f* a lambda expression?

Function Calls

- One way is obvious:

```
> ((lambda (x) (cons x '(b))) 'a)
(a b)
```

- But how about:

```
> ((lambda (f) (f '(b c)))
        (lambda (x) (cons 'a x)))
(a b c)
```

Enabling Recursion with Labels

- The syntax

  `(label` *f* `(lambda (`p_1 ... p_n`)` *e*`))`

  allows the expression *e* to refer to the lambda expression itself, using the name *f*.

Example Using Label

```
(label subst (lambda (x y z)
    (cond ((atom z)
           (cond ((eq z y) x)
                 ('t z)))
          ('t (cons (subst x y (car z))
                    (subst x y (cdr z)))))))
```

- Note the use of `'t` as "else" in `cond` expressions.

Some Syntactic Sugar

- As a shorthand for *f* = `(label` *f* `lambda` `(`p_1 ... p_n`)` *e*`)`, we can write:
  `(defun` *f* `(`p_1 ... p_n`)` *e*`)`
- We can now refer to functions by name even when not needed for recursion.
  - Note that this does not increase the computation power of the language.

And... That's It.

- The language we've defined can be used to describe (and calculate) any function that can be computed by a Turing Machine.
- Still missing a lot:
  - No side effects. (Modern LISP is *not* pure Lambda calculus.)
  - No sequential calculation (only useful with side-effects, anyway).
  - Only has dynamic scoping (modern LISP has both).
  - No numbers.
    - Can be expressed using Church Numerals, or a unary system (e.g., list of *n* `t`s), or a binary

The Universal Function

- So, we have a simple notation for working with lists.
- But the notation itself is composed of lists.
- So, we can write a function to evaluate functions!
- Universal function / UTM theorem.
- McCarthy defined `(eval` *e a*`)`, which accepts an expression *e* and a mapping *a* from names to values, and returns the (evaluated) value of *e*.

The Universal Function

- `eval`'s definition takes only a few dozen lines of code.
- Was implemented by McCarthy's student Steve Russell in IBM 704 machine code.
  - The result *is* a LISP interpreter.
- Try doing something similar with C-style languages.
  - How do you represent C's syntax as a data structure?
  - How complex would a C program, which takes a C-code-structure and "executes" it, be?
  - Now try answering the same with C++11 syntax...

No Code/Data Boundary

- The super-simple evaluation is possible because there's little to no boundary between code and data in LISP.
- Note that this has little to do with the fact that the language is functional, and indeed modern LISP has functions with side-effects, true variables, and more.
  - Implementing `eval` still remains equally simple.

No Code/Data Boundary

- The above also means that the boundary between development-time, compile-time, and run-time is blurred.
- Code can easily generate more code (it's just data!) and execute it.
  - The generated code is much more likely to have proper structure than an attempt to generate valid strings that contain C code, for example.
- *These* are real macros.
- Make the language super-extensible.
  - e.g., object-orientation as a library... (CLOS).

“LISP ia a Programmable Programming Language”

(John Foderaro)

- Bottom Up development: Adapt the language to your specific project.
- Somewhat like designing project-specific libraries, but on a much deeper level.

What Made LISP Different (in 1960)

- Paul Graham lists nine things that made LISP very different from other languages at the time (well, assembly and FORTRAN):

1. **Conditionals**. Prior to that, all we had was a conditional jump.
2. **A function type**. Functions as first-class values.
3. **Recursion**.
4. **Variables are references**. Values, not variables, have types.

What Made LISP Different (in 1960)

5. **Garbage collection**.
6. **Program composed of expressions**. The following two are equivalent:

   ```
   (if foo '(= x 1) '(= x 2))
   ```

   and `(= x (if foo 1 2))`
7. **A symbol type**. Different than strings, since equality can be tested by pointer comparison.
8. **A notation for code** as runtime data.
9. **Whole language always available**. Little distinction between edit/compile/run-time.

What Makes LISP Different Today

	C (1972)	Java (1991)	Python (1995)
Conditionals	✓	✓	✓
Function type		½	✓
Recursion	✓	✓	✓
Reference variables	✓	✓	✓
Garbage collection		✓	✓
Program composed of expressions			½
Symbol type			
Notation for code			
Language always available			½

Is LISP Used in the Real World™?

- Yes, but apparently mostly in in-house programs rather than off-the-shelf software.
  - EMACS.
  - MACSYMA (symbolic math).
  - AutoCAD's scripting language (AutoLisp).
  - Card processing program used by American Express.
  - Mirai (animation program; Gollum in *LotR*).
  - Yahoo! Shops.
  - Industrial design programs used by many car and airplane manufacturers.
  - ITA's (now Google's) QPX air-ticket pricing program.
  - (Historically) AI research.
    - Mostly because this was McCarthy's research area.
  - Many, many more...

Is LISP Used in the Real World™?

- There are dozens of modern dialects of LISP.
- Dialects in actual use in the industry:
  - ANSI Common LISP (ANSI X3J13 [Steele], 1994)
  - Scheme (Steele and Sussman, 1975)
  - AutoLISP and VisualLISP (AutoCAD, AutoCAD 2000 and later)
  - Clojure (Hickey 2007).

((((But What About The Parens?))))

```
<program>
  <include lib="stdio.h"/>
  <define-function name="main">
    <args>
      <arg name="argc" type="int">
      <arg name="argv" type="char**">
    </args>
    <code>
      <invoke function="printf">
        <arg type="char*">Yeah, like XML is better.</arg>
      </invoke>
    </code>
  </define-function>
</program>
```

(If you think this isn't a realistic example, read about XSLT.)

Philip Greenspun's Tenth Rule of Programming

“Any sufficiently complicated C or FORTRAN program contains an ad-hoc, informally-specified, bug-ridden, slow implementation of half of Common LISP.”

[Robert Morris: "Including Common LISP."]

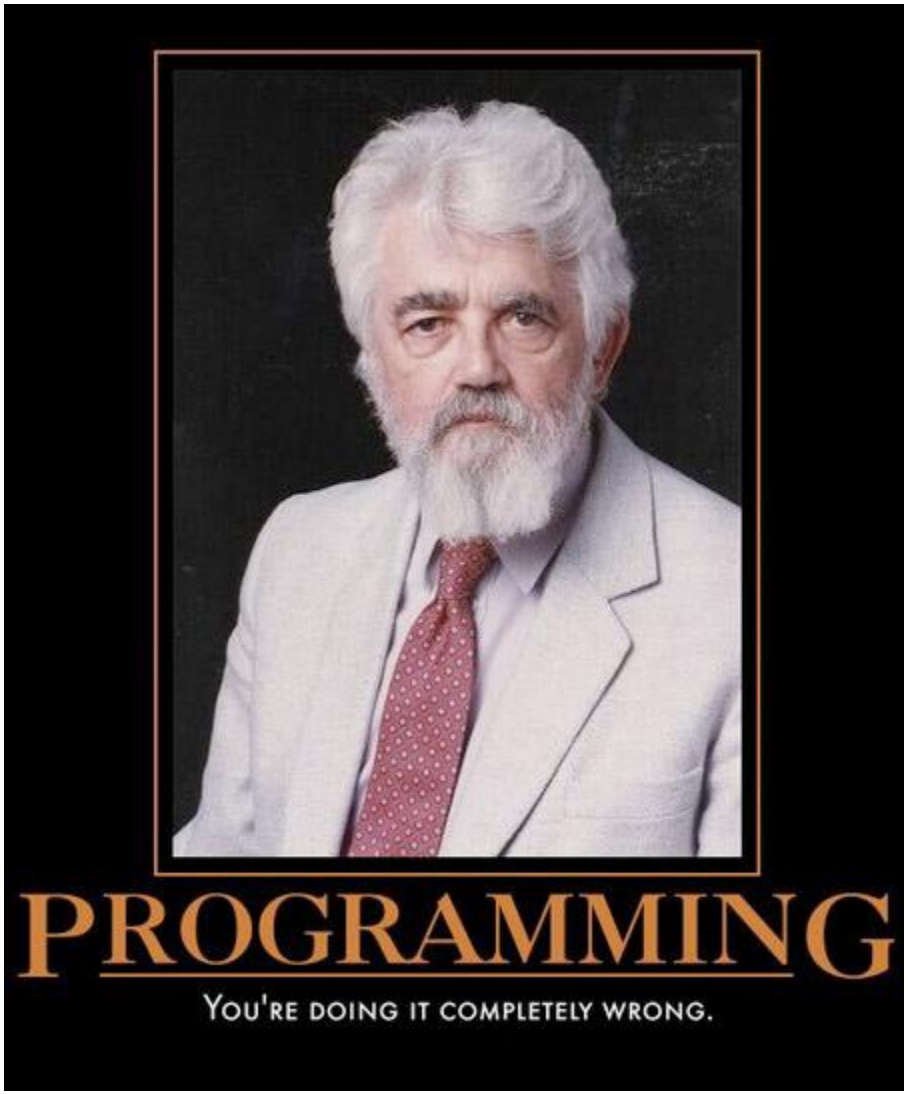
So, Why Doesn't *Everyone* Use LISP?

- Super-high developer productivity *for good developers* ("LISP hackers").
- Does *not* work well for mediocre or bad developers.
  - Easy to write C programs in LISP, but then you'd complain about the syntax and performance.
- Bad rep.
  - Historically, bad (slow) GC.
    - No longer an issue, e.g. on JVM.
  - Code x0.9-1.7 times as fast as C++ code for common cases.
    - cf. Python: x2.6 to x84; Java: x2.6-7.
    - (Numbers from Norvig, 2002.)

Some Recommended Reading

- *The Little LISPer* (3e, Friedman and Felleisen, 1989)
  - About solving problems with recursive functions.
- *Common LISPcraft* (Wilensky, 1986)
  - Basic introduction to LISP.
- *On Lisp* (Graham, 1993)
  - Advanced topics; how to become a LISP hacker.
- Peter Norvig's essays on LISP on norvig.com
- Paul Graham's essays on LISP on paulgraham.com
- "Why Lisp macros are cool, a Perl perspective" (Dominus, 2005).
  - Short online post by the author of *Higher Order Perl*.

- Avoid: *The Joy of Clojure* (Fogus and Houser, 2011).



John McCarthy (1927-2011)