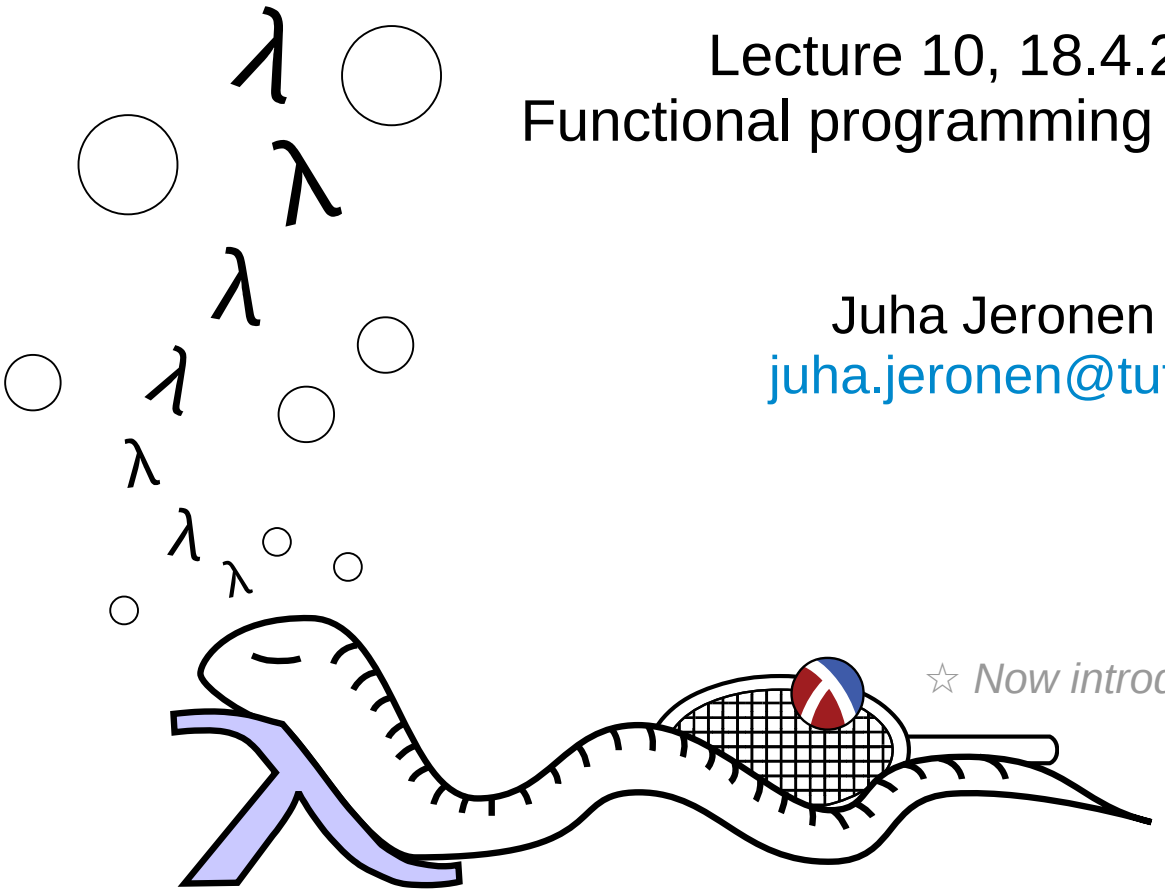
Python 3 for scientific computing ☆

Lecture 10, 18.4.2018
Functional programming (FP), part 1

Juha Jeronen
juha.jeronen@tut.fi



☆ *Now introducing also Racket.*

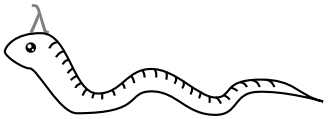
Spring 2018, TUT, Tampere
RAK-19006 Various Topics of Civil Engineering

TAMPERE UNIVERSITY OF TECHNOLOGY

Meta

- Final topic, second to last lecture.
- We will begin with Python, but already today move on to Lisp and Racket.
- **Goals** of these final two lectures; a.k.a. *"why bother?"*:
  - Introduce some basic ideas of FP (with no claims to completeness!)
  - Demystify some programming language constructs (e.g. generators, OOP)
  - Show what Python is still missing, from a Lisp/Racket viewpoint.
    - Not to bash Python! (In many respects it's already an excellent language.)
    - Instead, to show what lies at the horizon – beyond Python.
      - ***Why***? To know only C is to miss out on a lot. To know only C and Python, …
        - Still true if we replace "C" by "MATLAB" and/or "Fortran".
        - But pick your languages wisely:
          - *A language that doesn't affect the way you think about programming, is not worth knowing. –Alan Perlis, Epigrams on Programming (1982)*
      - Languages vary in power, but likely no ultimate programming language exists.
      - Racket and Haskell are both very different from many popular languages.
      - In this course, we have time only for one; we will explore Racket.
- *Be prepared for one final rough ride.*

Python and FP



- Python features **suitable for FP**:
  - Lexical scoping, with closures (recall lecture 2, slide 42)
  - Higher-order functions (lecture 2, slide 41)
    - Fully supported; can make your own
    - **map**, **filter**, the *functools* module (esp. *reduce*, *partial*)
  - **lambda**, to create one-off throwaway operations for higher-order functions
- Python features **not suitable for FP**:
  - No tail-call optimization (TCO)
    - This is by design; the BDFL has stated TCO is unpythonic. [1] [2]
      - Maximum depth of call stack: *sys.getrecursiondepth()*, *sys.setrecursiondepth()*
    - But a TCO library is available, implementing TCO via exceptions and *trampolining* [1] [2].
      - Explanation from the author.
    - From the same author: continuation, adding continuations to Python, also via exceptions.
  - At least from a Racket viewpoint:
    - Focus on imperative programming; no distinction between binding and assignment
    - **lambda** is restricted to a single expression only, severely limiting its potential uses
    - No way to introduce bindings local to a **lambda** (except by abusing another **lambda**)
- So, Python is a *functionally flavored* language, not a *functionally oriented* one (not to mention a *purely functional* one, such as Haskell).
- But this is enough to get started!

⚠ Not to be confused with *procedural programming*, splitting the program into subroutines.

Functional programming (FP)

- A programming paradigm, on the rise as of the 2010s:
  - Popularity of Haskell in academic circles; rise of functionally oriented or flavored languages running on the Java VM (Clojure, Scala, Kotlin).
  - Even Java 8 has introduced some FP features (see e.g. [1] [2]).

- **FP** focuses on **pure functions**, and **avoiding mutable state**.
  - ***Referential transparency***: imperative assignment at least strongly discouraged.
    - *Variables* in the mathematical sense of the word, instead of *assignables*.
  - **Pure FP** (e.g. Haskell) completely prohibits side effects and assignment.
    - Most real-world programs need side effects (esp. I/O); Haskell encapsulates them in *monads*, allowing the rest of the program to remain free of side effects.
  - **Impure FP** (e.g. Lisp), pragmatically, discourages but allows these.
    - No need for a 100% FP program; ***functional style*** where appropriate.

- FP is also associated with:
  - First-class and higher-order functions
  - Recursion as a central problem-solving strategy
  - Immutable objects (such as Python's *tuple*)
  - Type systems with (semi-)automatic inference (e.g. in Haskell, typed/racket)
  - Lazy evaluation of function arguments (e.g. in Haskell, lazy/racket)

- FP is not really new; invented in the 1950s.
  - But nowadays hardware is much faster, making FP more practical. If it buys us powerful abstractions, we can afford to waste CPU cycles – saving developer time.

⚠ Not to be confused with *procedural programming*, splitting the program into subroutines.

Functional programming (FP)

- A **pure function** communicates with the outside world only through its **arguments** and **return values**.
  - Each time it is called with the same argument values, the output is the same.
  - No reference to external state, no I/O, no internal state between invocations.
  - Mathematical functions are pure; in the general case, subroutines are not.
  - (Even Fortran 95 has them; a function can be declared PURE.)

- **Fewer bugs**: pure functions are easier to test and debug, “fewer moving parts”.

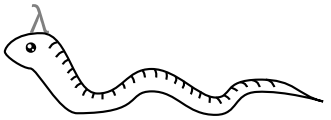
- But a more important advantage of FP is **improved modularity** (Hughes, 1984).
  - Somewhat depends on what language features we have. We'll see examples later.

- ***Composability*** is a major general theme:

  - *A good system should work like Lego. With just a few basic bricks and the means of connecting them, the users are empowered to develop the most unexpected constructions.*
    –*Primitives and means of composition* on the C2 wiki [about]

  - SICP: *primitives*, *means of combination*, *means of abstraction*.
    - *Abstraction*: roughly, being able to name combinations to re-use them.

FP ideas, concepts



- No complete recipe; instead, a sampling of useful “bricks” for your own programs.

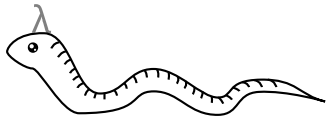
- **Functional update** (exercises 3, 3. d); and lecture 3, slide 51):
  - *Old state in, new state out.*
  - Immutable objects; create the updated instance as a copy, and **return** it.

- **Immutable containers** in Python: *tuple*, *collections.namedtuple*, *frozenset*
  - In *namedtuple*, don't worry about the underscores; the methods are public, just named that way to avoid conflicts with field names.

- Related idea, from Clojure: ***decouple object identity from state***
  - In OOP, object identity = memory address; which tangles the id with state.
  - Keep a central “world” repository, e.g. a Python *dict*, that holds the objects.
  - Dict key = object identity. For a given key, the value is an actual (immutable!) object instance representing the current state of that object.

  - This makes thread safety *almost* trivial: *no instance ever changes state*.
  - A function may have an old version of an object, but never an inconsistent one.
  - Only *almost* trivial; we have pushed the problem to the world update.
    - ⚠ **Single bytecode instructions are atomic, but a pythonic swap isn't.**
    - A typical solution is a mechanism built on top of compare-and-swap. Python is garbage-collected, which would make a lock-free update relatively easy, but there is no a compare-and-swap in Python, so locks are needed.
    - atomos: a library that fakes atomics via locking (hiding irrelevant details).

FP ideas, concepts



- Impure FP allows **a mix of styles** (lecture 3, slide 34):
  - Fully FP – old state in (as read-only!), new state out
  - Hybrid: FP interface to an IP algorithm
    - To the outside world, appears just like fully FP!
  - Fully IP – may mutate its input (not suitable for FP!)

```
def double(lst):    # fully FP
    return [2*x for x in lst]

def sum(lst):       # hybrid
    s = 0
    for x in lst:
        s += x
    return s

def munge(lst):     # fully IP
    for k in range(len(lst)):
        lst[k] = lst[k] ** 2
```

- **First-class functions** (lecture 2, slides 41–43)
  - Functions as arguments
  - Functions as return values
  - Functions as something to store in variables and in containers
  - **lambda**: anonymous functions

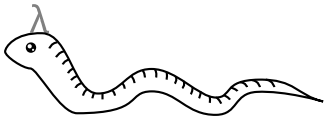
- **Higher-order functions** such as **map** and **filter**:

```
[x for x in lst if pred(x)]   ⇔   filter(pred, lst)
[f(x) for x in lst]           ⇔   map(f, lst)
```

  - "*Higher-order function*" just means it takes a function as an argument. Parameterize the operation to perform, pass around a computation to do later, …

  - In Python, because functions are first-class, you can define custom higher-order functions wherever needed. (And with duck typing, no bureaucracy.)

FP ideas, concepts



- ***Lexical closures*** (lecture 2, slide 42):
  - ***Closure***: a function, together with an *environment* that contains the values for its *free variables*. (Hence a closure is an *instance* of a function.)
  - ***Free variable*** (lecture 3, slide 29): (in Python name/value semantics) A name that is locally unbound, i.e. neither a locally defined name nor a formal parameter.
  - ⚠ **lambda** can be used to *make* a closure, but "a **lambda**" – in the sense of an anonymous function – *is not itself* a closure. A closure can be made, just as well, out of a named function.

```
def make_adder(inc):
    def adder(x):
        return x + inc
    return adder
```

The closure. The returned function instance is *closed over* its free variable "inc".

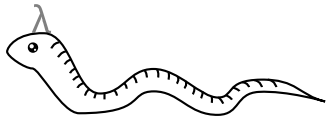
- **Functional loop** (lecture 3, slide 24):

```
def sum(lst):
    def loop(acc, lst):
        if not len(lst):
            return acc
        else:
            first,*rest = lst
            return loop(acc + first, rest)
    return loop(0, lst)
```

  - Based on recursion as a problem solving strategy: until the termination condition is fulfilled, the *loop* function calls itself at the end of each iteration.

⚠ Requires tail call optimization, to work without crashing for arbitrarily long input. Not suitable for Python; but ***the*** way to loop in the Scheme family of Lisps.

FP ideas, concepts



- **Fold** (generalizes to *catamorphism*)
  - *Reduces* a sequence to a single value, with the help of an initial value for the accumulator, and a function that combines the current value in the accumulator with the next element from the sequence.

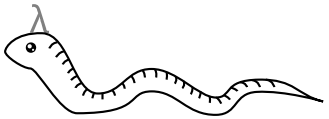
```
# example
from operator import add  # for this example, we need "+" as a regular function
from functools import reduce
def sum(lst):
    return reduce(add, lst, 0)
```

  - Some operations are not commutative; hence both **left** and **right** folds exist.
    - In Python, *functools.reduce* is **foldl**, processing the sequence from the left.
  - Python has no **foldr** (to process the sequence from the right), so let's define one:

```
from functools import reduce as foldl
def foldr(proc, iterable, init):
    return foldl(proc, reversed(iterable), init)
```

    - If *iterable* is random-accessible (such as a list), this should be efficient, because then all *reversed()* needs to do is to set up an iterator to walk backwards.

FP ideas, concepts



- ⚠ In **fold**, the argument order for *proc* depends on the language:
  - **Python**: proc(acc, elt)
  - **Racket**: (proc elt acc) ← Function call notation in Lisps.

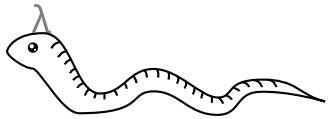
- How to see this in Python: just read the docs – or *reverse-engineer* it, like this:

```
from functools import reduce

lst = (1, 2, 3, 4)

def sum_two(a, b):  # just sum numbers, for easily readable output
    print(a, b)  # DEBUG
    return a + b

reduce(sum_two, lst, 0)
```

- Output (one pair per line; here on one line for brevity): 0 1 1 2 3 3 6 4
  - It is clear from this that “a” is the running total at the start of *sum_two()* – i.e. the current value of the accumulator – and “b” is the next element from the list.
  - *help(reduce)* already states that it processes the input sequence from left to right, but here we see that, too.

- This matters when *proc* is not commutative (example of function composition below).

FP ideas, concepts

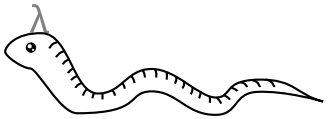


- **Unfold** (generalizes to *anamorphism*)
  - *Generates* a sequence ***corecursively,*** with the help of an initial state, and a function that constructs the next element. The function may either provide a (*value*, *new_state*) pair, or signal to **unfold** that the sequence ends.
  - Essentially (full example on GitHub):

Roughly, ***recursion*** is divide-and-conquer, reducing until it hits a base case, whereas ***corecursion*** is generative, *starting from* the base case.

```
def unfold(proc, state):
    out = []
    while True:
        result = proc(state)
        if result is not None:
            value, state = result  # get latest value, and update state
            out.append(value)
        else:  # sequence ends
            return out

def fibo(state):
    a,b,countdown = state
    if countdown > 0:
        return (a, (b, a+b, countdown - 1))

print(unfold(fibo, (1, 1, 20)))
```

FP ideas, concepts



- **Function composition**
  - Like in mathematics: $(f \circ g)(x) \equiv f(g(x))$
  - In Python (from Mathieu Larose: Function Composition in Python, with minor edit):

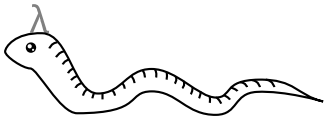
```
from functools import reduce as foldl

def compose(*functions):
    def compose_two(f, g):
        return lambda x: f(g(x))
    return foldl(compose_two, functions, lambda x: x)

double = lambda x: 2 * x
inc    = lambda x: x + 1
dec    = lambda x: x - 1

h = compose(dec, double, inc)
print(h(10))  # 21
```

Exercise: trace the steps to confirm that these conclusions are correct.

In the composition, where does the initial identity function end up?

- **Composition is not commutative**, so both the *fold direction* and the *order of arguments in proc* are important! To get the standard behavior that the *rightmost function in the composition is applied first*:
  - For the above *compose_two()*, with *proc(acc, elt)*, like in Python – use ***foldl***.
  - If it was *proc(elt, acc)*, we would need to use *foldr* – or to use *foldl*, swap the roles of *f* and *g* in the definition of *compose_two()*.

FP ideas, concepts



- ***Partial application*** (of function arguments)

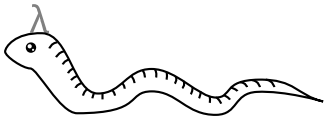
  ⚠ Do not confuse with *partial function (mathematics)*.

  - *Specialize* a function by fixing some of its parameters, starting from the left.
  - Python: *functools.partial*
    - Supports also keyword arguments. For them, the *partial()* call sets up defaults, which can still be overridden when calling the specialized function.
    - If you intend to export a function specialized in this way, may be useful to set the __*doc*__ attribute of the result manually to give it a descriptive docstring.

```
from functools import partial
from operator import add

add_one = partial(add, 1)  # fix leftmost argument to 1, return specialized function
print(add_one(41))  # 42

from_binary = partial(int, base=2)
print(from_binary('00101010'))
print(from_binary('2a', base=16))  # we can still use a non-default base
```

  - Below, we will talk about the related concept of *currying*.

“FP chess”



- Many standard higher-order functions can be derived from others; nice hobby?
- For example, what is **zip**, under the hood?

Maybe at its most useful when learning FP.

  - A particular **map**:

```
def pack(*items):       # vararg tuple constructor...
    return tuple(items)  # ...because this wants one iterable
def zip(*lsts):
    return tuple(map(pack, *lsts))  # tuple() forces the generator

A = (1, 2, 3)
B = (4, 5, 6)
z = zip(A, B)
```

We here use the fact that Python's map accepts multiple inputs (extracting an element from each in lockstep), and terminates on the shortest input.

In other words, **map** is almost **zip**; the only thing we have to do, at each iteration, is to pack the data into a tuple.

  - A particular **unfold** (shown here for two inputs only):

```
def zip_two(state):
    if state is None:
        return None
    (A0, *As), (B0, *Bs) = state
    if len(As) > 0 and len(Bs) > 0:  # continue until shortest input ends
        return ((A0, B0), (As, Bs))
    else:
        return ((A0, B0), None)  # last value; terminate on next call

z = unfold(zip_two, (A, B))
```

Expressive power, in 3 minutes

- **Landmark paper**:
  Matthias Felleisen (1991): *On the Expressive Power of Programming Languages*

- Full text from Rice PLT: Publications.

- **Main ideas**:
  - *All* practical programming languages are *Turing-complete*, hence *in principle* they can express the same programs. However, programming a Turing machine to do any but the very simplest of tasks very quickly becomes infeasible in practice – the *Turing tarpit*.

  - The observation extends to actual programming languages: they vary in power at least in the subjective sense that a competent programmer can accomplish much more in a reasonable timeframe in some languages than in others.

  - Making this idea objective, by a clever trick: constructing small pathological programs, and analyzing in which languages they terminate and in which they enter an infinite loop. Two languages are deemed equally expressive if they have the same sets of terminating and non-terminating programs.

  - Also important: considering whether expressing a given program in another language requires only local modifications, or a complete redesign of the program.
    Leads to the concept of macro-expressibility (important in the context of Lisp).

  - Results: neither pass-by-name nor pass-by-value λ-calculus can be expressed by the other. First-class lazy functions **cannot** be expressed in an eager language, but macros can be used to introduce delayed evaluation as a syntactic element.

λ-calculus, in 3 minutes

- *A formal system in mathematical logic, for expressing computation based on function abstraction and application using variable binding and substitution.* –Wikipedia
  - Developed by the mathematician Alonzo Church in the 1930s.

- On the other hand, *the FP analog of the Turing machine*. A very minimal programming language, embedded in many functional and functionally flavored programming languages.
  - Surprisingly expressive, while near-impossible to use in practice. Some examples:
    - Natural numbers via *Church numerals*
    - Functions representing true, false, if
    - Y combinator: self-reference without direct self-reference, allows recursion
    - Ω: a non-terminating program
  - Indeed, λ-calculus is **equivalent** to a Turing machine. [1] [2]

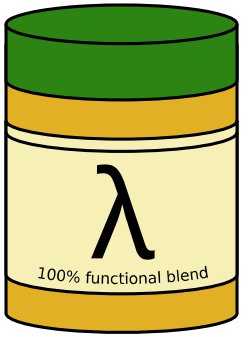
$$Y = \lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))$$

$$\omega = \lambda x.\ x\ x$$
$$\Omega = \omega\ \omega$$

- Both untyped and typed variants exist.
  - Curry's paradox: for mathematical logic, untyped λ-calculus is not a sound deductive system.
    - In practice, it's still useful for programming; these are different domains.
  - Curry–Howard correspondence: typed λ-calculus is.
  - Dynamically typed languages, e.g. Python and most Lisps, embed the untyped variant.
  - Haskell and typed/racket embed the typed variant.

- Materials for the adventurous:
  - The Y Combinator (no, not that one): A crash-course on lambda calculus. (Recommended!)
  - Equational derivations of the Y combinator and Church encodings in Python.
  - Compiling to lambda-calculus: Turtles all the way down.
    The λ-calculus is the assembly language of mathematics. –Matthew Might

Currying



- Seen in functional programming and λ-calculus.
- Named after the mathematician Haskell Curry.
  - Let a, b, c be types (such as int, float, str). A function

    $(a \times b) \rightarrow c$

    can be thought of as first

    $a \rightarrow (b \rightarrow c)$ and then $b \rightarrow c$.

    Curried function; “a” is now specialized.
  - Function arrows are defined to be *right-associative*:

    $a \rightarrow b \rightarrow c \rightarrow d \equiv (a \rightarrow (b \rightarrow (c \rightarrow d)))$
  - If we further define function application to be *left-associative*, we can drop the parentheses. Haskell does this, and curries automatically (which can be mind-bending at first).

    $f\ a\ b \Rightarrow (f\ a)\ b$
    $f\ a\ b\ c \Rightarrow ((f\ a)\ b)\ c$
  - Lisps however make no assumptions; they prefer explicitly marking the expression structure.
    - Automatic currying doesn't mix well with variadic functions and dynamic typing [1] [2] [3].
- Racket: curry. To experiment with *automatic* currying in Racket, try this silly hack.
- See also SRFI-26: Notation for specializing parameters without currying.

Lisp

- The Lisp programming language:
  - LISt Processor (LISP), John McCarthy, MIT, 1958
    - Second-oldest high-level language alive (oldest Fortran, 1957).
    - ⚠ *"High-level language"* means "*anything above assembly*". Just like Python, Lisp operates at a much more abstract level than C or Fortran.
  - Originally a mathematical notation for computer programs, influenced by λ-calculus.
  - Main ideas: ***code is data***; variable-length lists and trees as fundamental data types.
  - Extremely uniform and composable. Lists everywhere; everything is an expression.
  - Homoiconic: *programming by way of raw syntax tree entry*. –seen on the internet
  - Important part of IT/CS history; has influenced many languages, including Python.
    - John McCarthy: History of Lisp
    - Herbert Stoyan: Lisp History
    - LISP 1.5 Programmer's Manual (1962)
  - Books:
    - Abelson & Sussman (1996): Structure and Interpretation of Computer Programs [2e] (Scheme)
    - R. Kent Dybvig (2009): The Scheme Programming Language [4e]
    - Shriram Krishnamurthi (2007): Programming Languages: Application and Interpretation [2e] (Racket)
    - Felleisen, Findler, Flatt, Krishnamurthi (2018): How to Design Programs [2e] (Racket)
    - Peter Norvig (1992): Paradigms of Artificial Intelligence Programming (Common Lisp)
    - Peter Norvig (1993): Tutorial on Good Lisp Programming Style (Common Lisp)
    - Paul Graham (1993): On Lisp (Common Lisp)
    - Peter Seibel (2003): Practical Common Lisp
    - Doug Hoyte (2008): Let Over Lambda – 50 Years of Lisp (Common Lisp)

Maybe start from SICP; doesn't matter if you code in Python, this book teaches also how to think about programming, and demystifies interpreters and compilers.

⚠ **Not** the limit to end all programming languages. Such a thing probably does not exist.

Paul Graham makes excellent points, but consider also Felleisen (1991); from the set of all imaginable features, Lisp lacks at least first-class lazy functions.

Lisp

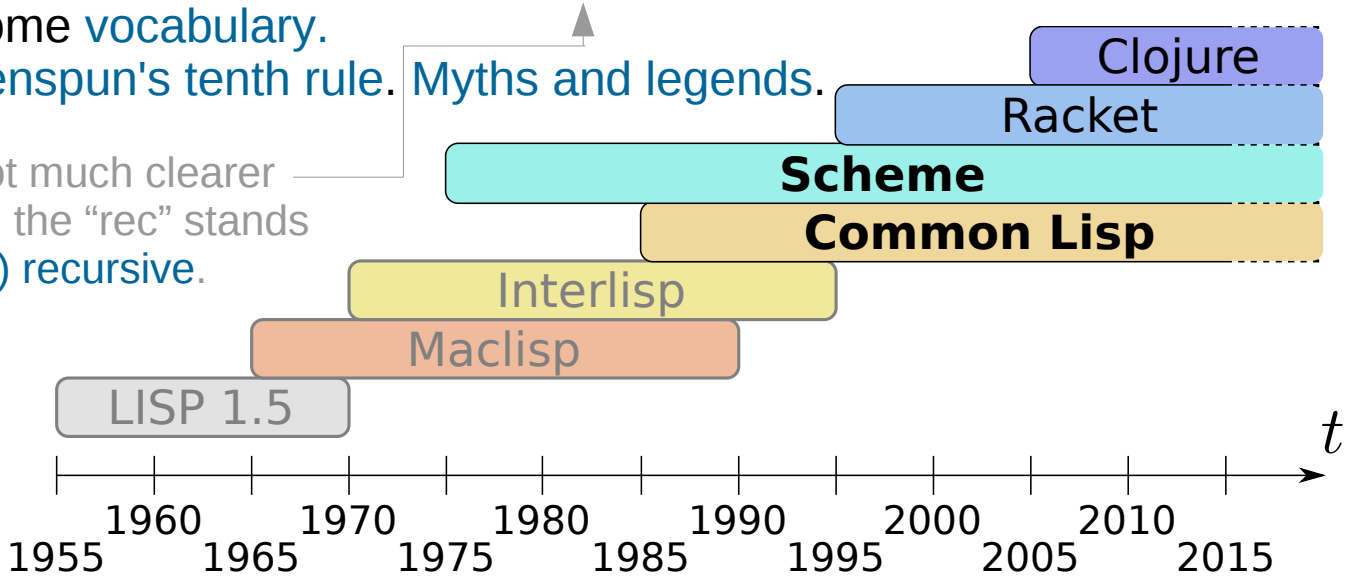
- The Lisp programming language:
  - LISt Processor (LISP), John McCarthy, MIT, 1958
    - Second-oldest high-level language alive (oldest Fortran, 1957).
    - ⚠ *"High-level language"* means "*anything above assembly*". Just like Python, Lisp operates at a much more abstract level than C or Fortran.
  - Originally a mathematical notation for computer programs, influenced by λ-calculus.
  - Main ideas: ***code is data***; variable-length lists and trees as fundamental data types.
  - Extremely uniform and composable. Lists everywhere; everything is an expression.
  - Homoiconic: *programming by way of raw syntax tree entry*. –seen on the internet
  - Important part of IT/CS history; has influenced many languages, including Python.
    - John McCarthy: History of Lisp
    - Herbert Stoyan: Lisp History
    - LISP 1.5 Programmer's Manual (1962)
  - Books:
    - Abelson & Sussman (1996): Structure and Interpretation of Computer Programs [2e] (Scheme)
    - R. Kent Dybvig (2009): The Scheme Programming Language [4e]
    - Shriram Krishnamurthi (2007): Programming Languages: Application and Interpretation [2e] (Racket)
    - Felleisen, Findler, Flatt, Krishnamurthi (2018): How to Design Programs [2e] (Racket)
    - Peter Norvig (1992): Paradigms of Artificial Intelligence Programming (Common Lisp)
    - Peter Norvig (1993): Tutorial on Good Lisp Programming Style (Common Lisp)
    - Paul Graham (1993): On Lisp (Common Lisp)
    - Peter Seibel (2003): Practical Common Lisp
    - Doug Hoyte (2008): Let Over Lambda – 50 Years of Lisp (Common Lisp)

Maybe start from SICP; doesn't matter if you code in Python, this book teaches also how to think about programming, and demystifies interpreters and compilers.

Lisps

- Actually, Lisp is a **family** of programming languages!
- Two main subfamilies:
  - **Scheme** (1975): a minimalist, clean "reboot" (see historical paper; more history)
    - Current standards: R^{6}RS, R^{7}RS; commonly supported: R^{5}RS.
    - Many implementations: CHICKEN, Guile, IronScheme, TinyScheme, …
    - "*Lisp-1*": Functions and variables live in the same namespace, like in Python.
  - **Common Lisp** (1984): a large-ish language to unify by-then successors of Maclisp
    - "*Lisp-2*": Separate namespaces for functions and variables.
    - Archaic function names: *rplaca*, *progn*, *labels*, *mapcar*, *symbolp*, *terpri*, ...
      - In Scheme: *set-car!*, *begin*, *letrec*, *map*, *symbol?*, *newline*, ...
      - Some vocabulary.
    - Greenspun's tenth rule. Myths and legends.

To be fair, not much clearer until you find the "rec" stands for (mutually) recursive.



Wikipedia:
- List
- Timeline

Racket

https://racket-lang.org

- **Racket** (1994):
  - A descendant of Scheme; formerly known as PLT Scheme.
    - Renamed, at v5.0 (2010); diverged sufficiently to be considered a new language.
    - "*Lisp-1*": Functions and variables live in the same namespace, like in Python.
    - ⚠ If you read Common Lisp, in "Lisp-ones" such as Scheme and Racket, #'(…) means (syntax …), not (function …).
    - ⚠ If you read Scheme, *Racket is not Scheme*; e.g. no set-car!, set-cdr!
  - **A large language**, built mostly within itself, on a minimal core. Modules, object system(s) [a] [b], exceptions, comprehensions, generators, promises, contracts, pattern matching, …
    - Highly advanced contracts system [paper]; one of the killer features.
  - **A pythonic Lisp**. Many of the large-language features above, as well as:
    - Multi-paradigm: focused on FP, but IP and OOP also allowed.
      - Also e.g. Racklog: Prolog-style logic programming in Racket.
    - Any script is a program. Has a conditional main construct.
    - Dynamically and strongly typed. Lexically scoped.
    - Keyword arguments, like in Python.
      - But in Racket, each argument is either **positional-only** or **keyword-only**.
    - Very extensive documentation with very nice layout.
      - No docstrings; instead supports in-source documentation similar to Javadoc.
      - No runtime inspection of modules, unlike Python (no *help*, *dir*, *vars*, *inspect*).
    - Unfortunately, concurrent, not parallel – just like Python. Similar solution.

May read [1 2 3 4 5] like a math monograph or "*a transmission from an alien intelligence*".

Racket

https://racket-lang.org

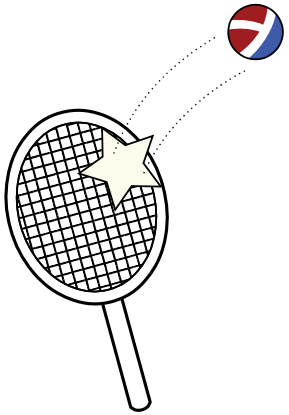
- Why Racket?
  - ***Batteries included***. A modern, serious, production-quality Lisp *with libraries*.
  - The Lisp with an IDE: DrRacket. (F1 to search docs for symbol under text cursor.)
    Includes the Macro Stepper, a proven-correct macro debugger.
  - Actively developed; cross-platform – runs on Windows, Linux, Mac OS X.
  - A platform for *language-oriented programming* (LOP) (Felleisen et al., 2018).
    - A program is a multilingual collection of components, each written in the language most appropriate for the task at hand.
    - One possible future of programming, esp. for wide-spectrum applications?
      - We already saw a light flavoring of LOP in the Python/Cython combination.
  - ***Programmable language***: no boundary between library and language.
    - Even the **for** loop *is imported from the standard library*, it is not a built-in!
    - *Modular syntax system*: a highly advanced version of syntactic macros.
  - Great for thinking about languages: experimentation, understanding.
    - Used for programming language research; developed by CS professors.
  - Timeless, serene beauty of Lisp – "*Maxwell's equations of software*" (Alan Kay).

Rackets

https://racket-lang.org

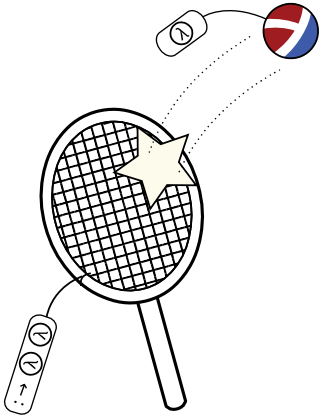
- Racket as much an idea about programming languages as a programming language.
- Philosophy: ***solve problems – make languages***
- Even the base distribution comes with three:

  ...plus the LaTeX-influenced *scribble* for writing documentation.



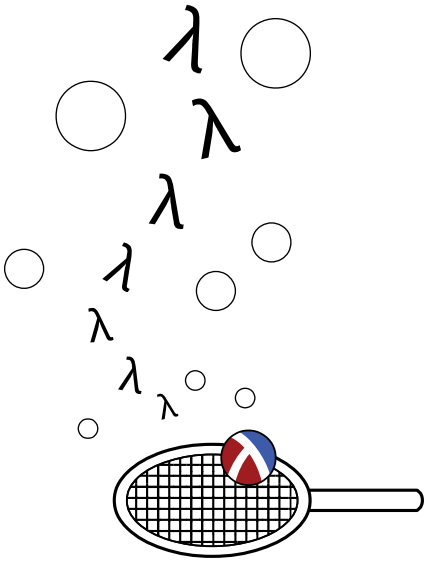
racket

- Dynamically typed
- The default option
- See quick introduction



typed/racket

- Statically typed, with inference
- *Gradually typed*: allows also dynamically typed code
- High performance possible

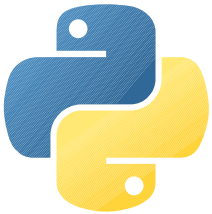


lazy/racket

- Lazy evaluation
- Slow but expressive

Highly unofficial illustration.

Lisp vs. Python



- ***Program***: sequence of (possibly nested) *symbolic expressions* (S-expressions), written as lists.
- Every expression evaluates to a value. No explicit *return* keyword (unlike Python).
  - Racket: value can be (void) if not needed. Logical false (#f) often used like Python's *None*.
  - A λ can contain multiple expressions (unlike in Python). More on this later below.

- ***Prefix notation***. First element of list, function to call. The rest, arguments.
  - Racket like Python: call-by-sharing; eager evaluation, left-to-right.
- Many operations variadic, e.g. (+ 1 2 3 4 5) as well as (+ 1 2) are valid Lisp.

- A ***symbol*** datatype
  - Essentially, an *interned string* (recall lecture 3, slide 18), allowing fast equality comparison by comparing memory addresses (Racket: eq?).
    - The compiler will intern any literal symbols, no runtime overhead.
    - Replaces many (but not all) uses of enum in other languages.
  - Python *almost* has a symbol type (some short strings are interned), but no separate syntax for it (as observed by Paul Graham).

- ***No precedence of operators***; *expression structure must always be written down explicitly*.
  - Syntax close to AST (i.e. *homoiconic*), good for macro processing.
  - The traditional solution is to use (lots and lots of) parentheses to denote the structure.

- Parentheses not the only option; we can use *"something just as bad"* (Paul Graham).
  - Scheme Request For Implementation 110: Sweet-expressions (t-expressions).
    - **Inspired by Python**, replacing many use cases of parentheses by indentation. Adds also infix and natural function call syntax. Backwards-compatible with S-exprs.
  - Racket: sweet-exp. Since this is a Python course, we will mostly *#lang sweet-exp racket*.

Key properties of Lisp syntax, 1/3:

Uniformity

Arithmetic	(+ a b c)
Defining a function	(**define** (f x) (* x x))
Calling a function	(f 17 23)
Conditional	(**if** (eq? a b) 'yes 'no)
Calling a macro	(**for** ([k (in-range 10)]) (displayln k))
Doing **anything**	(operator args …)

Note the Zen minimalism: [*var seq*], no redundant syntactic keywords.

In Racket, () and [] have the same meaning; the square brackets are conventionally used in certain positions to improve readability.

Most Lisps use only ().

The operator is **always** at the start of the list.

Key properties of Lisp syntax, 2/3:

Composability


```
;; E.g. compute inline which function to call:
((if (eq? day-of-week 'sunday) + *) a b c)

;; Or what to put in the arguments of a call:
(loop op-sym
  (cond
    [(eq? prev-op 'none) (list b a op)]
    [(eq? op-sym prev-op) (cons b a)]
    [else (list b (reverse a) op)])
  rest out)

;; ...to DRY out this kind of repetition:
(cond
  [(eq? prev-op 'none)
    (loop op-sym (list b a op) rest out)]
  [(eq? op-sym prev-op)
    (loop op-sym (cons b a) rest out)]
  [else
    (loop op-sym (list b (reverse a) op) rest out)])
```

← I.e., composability of the ***syntax***.

Functions can also be composable:

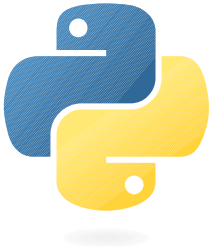
```
(define (compose . functions)
  (define (compose2 f g)
    (λ (x) (f (g x))))
  (foldr compose2 (λ (x) x) functions))

(define (double x) (* 2 x))
(define (inc x) (+ x 1))
(define (dec x) (- x 1))

(define h (compose dec double inc))

(h 10)  ; result: 21
```

Racket's standard library already has compose; an implementation is given here for teaching purposes only.

Key properties of Lisp syntax, 3/3:

Homoiconicity

a.k.a. **code is data**



```lisp
(define (hello name)
  (format "Hello, ~a!" name))
```

AST:

```lisp
'(define (hello name)
  (format "Hello, ~a!" name))
```

The difference.

⚠ **Roughly speaking**, because we have omitted lexical context and source location information.

The point is that given just the text of the source code, it is easy to see the shape of the AST.

```python
def hello(name):
    return "Hello, {}!".format(name)
```

AST (Full example; tool; docs [1 2]):

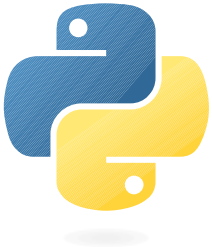
```
FunctionDef(
   name='hello',
   args=arguments(
      args=[arg(arg='name', annotation=None)],
      vararg=None,
      kwonlyargs=[],
      kw_defaults=[],
      kwarg=None,
      defaults=[],
   ),
   body=[
      Return(
         value=Call(
            func=Attribute(
               value=Str(s='Hello, {}!'),
               attr='format',
               ctx=Load(),
            ),
            args=[Name(id='name', ctx=Load())],
            keywords=[],
            starargs=None,
            kwargs=None,
         ),
      ),
   ],
   decorator_list=[],
   returns=None,
)
```

Original example in C# vs. Common Lisp.

Key properties of Lisp syntax, 3/3:

Homoiconicity

a.k.a. **code is data**



```lisp
(define (hello name)
  (format "Hello, ~a!" name))
```

```python
def hello(name):
    return "Hello, {}!".format(name)
```

AST:

```lisp
'(define (hello name)
  (format "Hello, ~a!" name))
```

AST, in a Lisp-like notation:

```lisp
'(FunctionDef (hello name)
  (Return
    ((Attribute
       (Str "Hello, {}!")
      "format")
     name)))
```

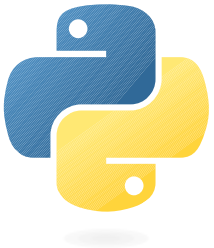
Now that was hardly fair;
how about this for a comparison?

⚠ The main point of homoiconicity is to make it (much) easier to write *syntactic macros*. (We will briefly explain them below.)

Original example in C# vs. Common Lisp.

apply vs. *

A Rosetta pebble



```
define add(a b)
  {a + b}

define L '(1 2)
define s
  apply add L
```

{ } denote infix math in sweet-exp, from SRFI-105.

```
define zip(. lsts)
  apply map list lsts

define A '(1 2 3)
define B '(4 5 6)
define z (zip A B)

define f(a b . more)
  displayln
    format "~a ~a ~a" a b more

f 5 17 23 42 9001
```

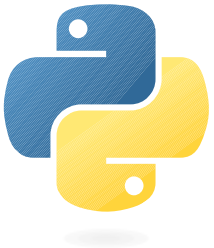
⚠ For a λ with varargs, that takes *only* varargs, the syntax is slightly different.

```
def add(a, b):
    return a + b

L = (1, 2)
s = add(*L)

def pack(*items):       # vararg tuple constructor...
    return tuple(items)  # ←this wants one iterable
def zip(*lsts):    # here tuple() forces the generator
    return tuple(map(pack, *lsts))

A = (1, 2, 3)
B = (4, 5, 6)
z = zip(A, B)

def f(a, b, *more):
    print("{} {} {}".format(a, b, more))

f(5, 17, 23, 42, 9001)
```

To ** in Racket, see keyword-apply (to call) and make-keyword-procedure (to define).

apply vs. *

A Rosetta pebble



apply must go first because of Lisp's prefix notation.

```
define add(a b)
  {a + b}

define L '(1 2)
define s
  apply add L
```

In apply, the list to unpack must be last, and it is always unpacked *to the end* of the argument list.

```
define zip(. lsts)
  apply map list lsts
```

The list function is roughly equivalent to our **pack**.

```
define A '(1 2 3)
define B '(4 5 6)
define z (zip A B)

define f(a b . more)
  displayln
    format "~a ~a ~a" a b more

f 5 17 23 42 9001
```

```
def add(a, b):
    return a + b

L = (1, 2)
s = add(*L)

def pack(*items):       # vararg tuple constructor...
    return tuple(items)  # ←this wants one iterable
def zip(*lsts):    # here tuple() forces the generator
    return tuple(map(pack, *lsts))

A = (1, 2, 3)
B = (4, 5, 6)
z = zip(A, B)

def f(a, b, *more):
    print("{} {} {}".format(a, b, more))

f(5, 17, 23, 42, 9001)
```

⚠ For a λ with varargs, that takes *only* varargs, the syntax is slightly different.

⚠ In this simple example, the Racket code requires inputs of equal lengths; see on GitHub how to make a map() that terminates on the shortest input or on the longest input.

Gotchas

- **Extremely important** which list item each expression goes into, *since you're writing the AST*.
  - Expect many syntax errors in the first few hours/days; most languages paper over this.

- Careful with both **parentheses** and **whitespace**:
  - **Parentheses denote function calls**. E.g. myfunc is just a reference, but (myfunc) is a call!
    - Like *myfunc()* in most languages; just a different placement of parentheses.
    - To instead make a list of items, call list, or quote the list to mark it as data (not to be run).
  - **Whitespace delimits list items**, commas are not used. (In Lisps, a comma means *unquote*.)

- These are standard features of Lisps, not specific to Racket or sweet-exp.

- Sweet-exp allows also myfunc().
  - f(a b …) becomes (f a b ...) in the ***reader layer***, before the rest of Racket even sees the code.

```
(define (f x) (* x x))
```

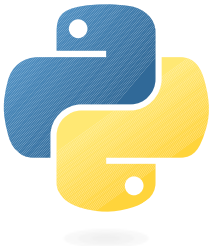
⚠ define binds a yet-unused name. To reassign an existing name, set!.

- **define** is the "function to call" (although here, technically a *special form*; somewhat like a function, but representing special-cased core functionality in the language itself.)
- In the docs, we see its **first argument** specifies **both** the name of the function to define, as well as its arguments – so it must be a list, which we denote by parentheses.
  - Sweet-exp allows also **define** f(x) (* x x), as also leaving out the outermost parentheses.
- All further arguments to **define** comprise the body of the function being defined.

```
(λ (x) (* x x))   ; Ok. See Functions (Procedures): lambda in the Racket guide.
(λ(x) (* x x))    ; Error, no space; in sweet-exp, now the result of (λ x) will be called, instead of λ.
(λ x (* x x))     ; Error, multiplying lists; omitting the parentheses has made x a rest argument.
```

Lisp vs. Python

define and λ vs. **def** and **lambda**



- In Lisps, code like

  (**define** (do-stuff x y) body …)

  at compile time, **quite literally maps to**

  (**define** do-stuff
    (λ (x y) body …))

  so a named function is just a λ bound to a name. In this sense, **in Lisp, all functions are lambdas**!

In Racket, you can use the symbol λ (U+3bb), or the word lambda; they mean the same thing, and either one works. These slides use "λ" simply because it is short and readable.

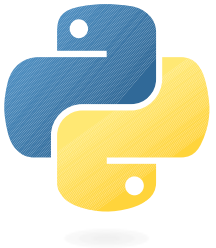
Lisps that are not as Unicode-minded as Racket may accept only "lambda".

- **In almost all other use cases**, in Lisp, think of λ not so much as defining an anonymous function, but rather as a way to encapsulate a code block (as a lexical closure) to pass it around.

- Same thing, different mindset. The point is, unless your background is in FP languages, practical experience with named functions may introduce a certain rigidity in thinking:
  Is this code block worthy of being carved into stone as a named function?
  (Move the code, write the define, add documentation, …)

- Unless one is careful, this rigidity easily carries over when thinking about lambdas:
  Is this code block worthy of being made into a function (albeit by pencil rather than chisel)?

- In Lisp, one can use λ to express the idea of "the following code block" *anywhere*, and *that block may contain arbitrary code*. **Lisp is more like a fluid than a solid**.
  - (As long as the code abides by the expectations of whatever code receives it.
    And that code block still *is a function*; e.g. it must be called to get its result.)



Lists

Python vs. Lisp

- **Python list**:
  - Actually is a variable-length, automatically resizing array – i.e. a ***dynamic array***.
  - Available in Racket as *gvector* (*growable vector*), in C++ as *vector*.

  - Typically the allocated storage is larger than the length of the data, to *amortize* the cost of having to re-allocate a larger storage and copy data around as more elements are added.

  - Many algorithms in Python assume that lists are implemented as arrays, and use the fact that this implementation gives access to any element in O(1) time.

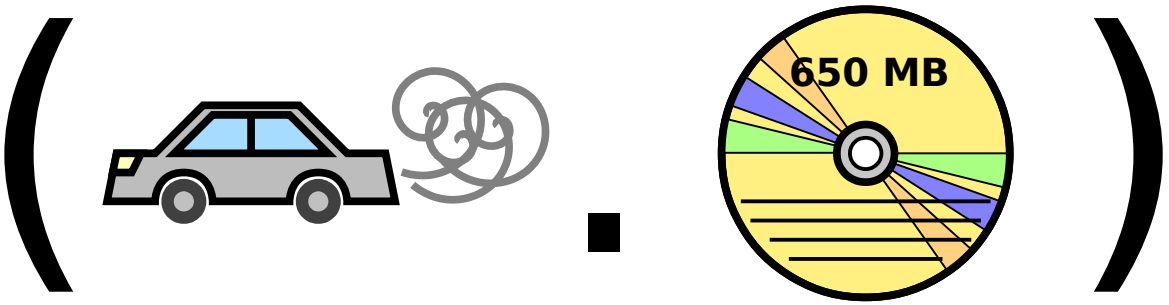
- **Lisp list**:
  - Classical singly ***linked list***
    - In general, non-contiguous memory layout; not as cache-efficient as an array.
    - Sequential access only; accessing the last element takes O(n) time.

  - Consists of ***cons cells***, a.k.a ***pairs***. In Lisp, a pair is denoted **(.)**
    - The **pair**, not the list, is **the fundamental container type in Lisp**; this is why lists are implemented as linked lists.
    - Can build also binary trees out of pairs.

Pairs
Terminology

- A **pair** consists of two elements, the head and the tail.

- Traditionally, the head is **car**, the tail **cdr**.
  - As the Racket Guide rightly points out, *the traditional names are also nonsense*.
  - Abelson and Sussman explain in SICP (2nd ed., footnote on p. 100) that *the names car and cdr persist because simple combinations like cadr are pronounceable.*



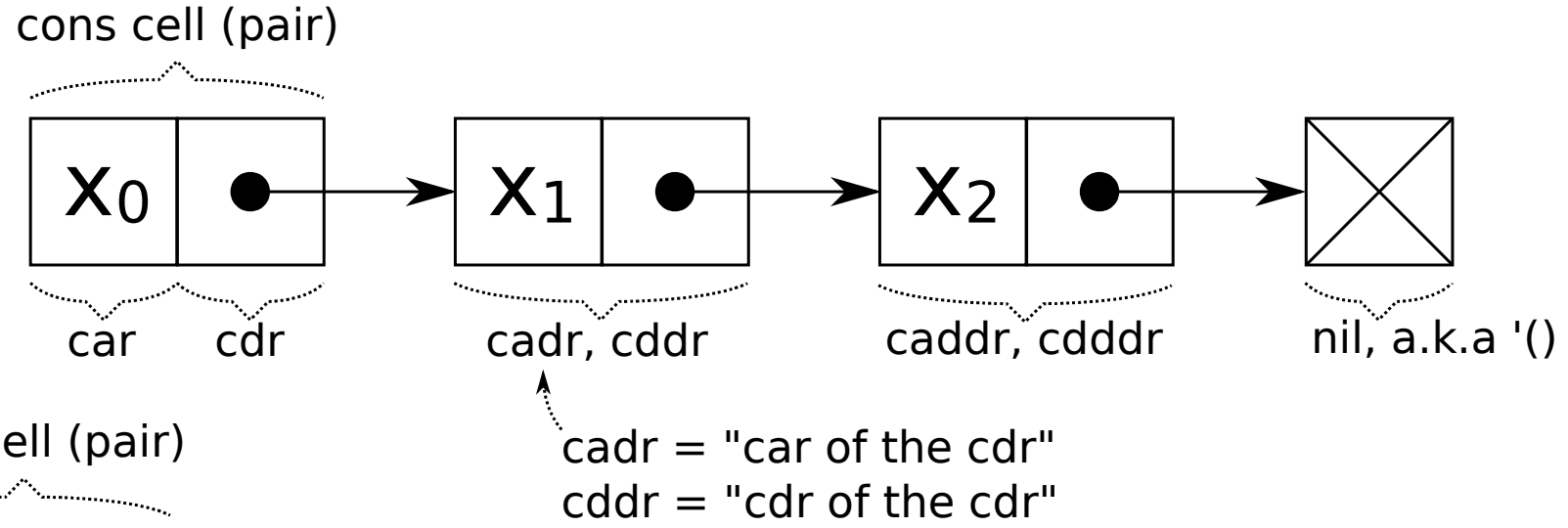
- Etymology [1] [2] : LISP was first implemented on an IBM 704, in the late 1950s.
  - **CAR** = **C**ontents of **A**ddress part of **R**egister
  - **CDR** = **C**ontents of **D**ecrement part of **R**egister
  - ...where “register” means memory location.

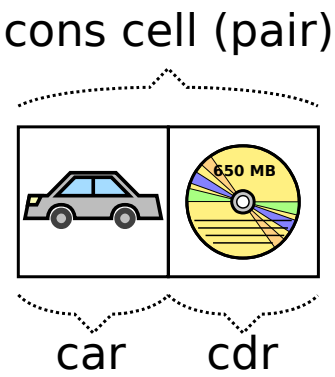
Lisp lists

Box and arrow diagrams

- A **Lisp list** is a *sequence of pairs*, where in each pair the car contains the data element, and the cdr contains a reference pointing to the next pair.
  - The list is terminated by a special value usually called *nil*, *null* or *empty*, which represents the empty list. It can be explicitly written as '()

- Box-and-arrow diagram of a Lisp list:





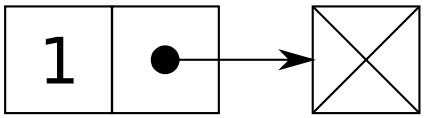
A pair does not *need* to be a list; the cdr can contain anything.

Lisp lists

Sequence of conses

- The function cons creates a *pair*, or *cons cell*:

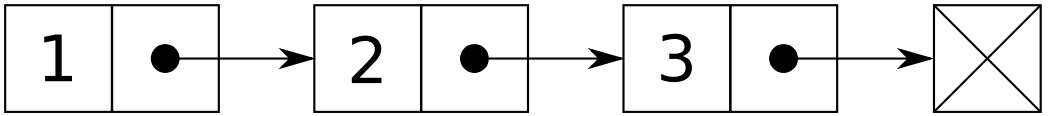
  (cons 1 empty)



empty is not a cons cell

- A list, then, is a sequence of such cells, built by a sequence of conses:

  (cons 1 (cons 2 (cons 3 empty)))



cons cells

- This can be abbreviated as

  (list 1 2 3)

  or

  '(1 2 3)

  The Racket Guide explains quoting.

⚠ (1 2 3) without the quote is an error; 1 is not a function.

- Note that in Lisp lists are built “in reverse”; cons adds the new element to the front. Hence many list-building algorithms finish with (reverse result) to turn it frontwards.

Lisp lists

What's in a cons?

- One final thing: ***data abstraction*** (SICP, sec. 2.1):
  It does not matter how cons, car, cdr internally work, as long as cons makes a pair *somehow*, car returns the head of a consed pair, and cdr returns the tail.

- To illustrate the point, in the exercises of SICP sec. 2.1.3, we find this exotic implementation using *procedures as data* [here shown in #lang sweet-exp racket]:

```
define cons(x y)
  λ (m) (m x y)

define car(z)
  z (λ (p q) p)

define cdr(z)
  z (λ (p q) q)
```

Roughly, LEGB applies here, too; these will shadow Racket's standard definitions.

It is easy to verify that this set of functions does what is needed. (Exercise.)

- Of course, this is inefficient, and not suitable for practical use; the example is meant just as an illustration of data abstraction.
- It also illustrates that even ***data structures are not fundamental***; in principle, they can be built out of lexical closures and λ. See *Church encoding* in λ-calculus.

To be continued…

- In the final lecture, 25.4.:
  - Many more Racket code examples.
  - Syntactic macros (what it means for a language to be programmable)
  - Pattern matching (destructuring bind with a vengeance; overload *by value*)
  - Continuations, call/cc (make your own control flow constructs)
  - How to make **return**, generators, a simple object system.
    - Some pointers to getting started with **syntax-parse**, for the adventurous.
  - Where to get some of these things for Python.
- See you next week!