Hack Your Language!

Introduction to Compiler Construction

CSE401 Winter 2016

Ras Bodik
Alvin Cheung
Maaz Ahmad
Talia Ringer
Ben Tebbs

Lecture 1: Abstractions

What is a programming language.
Why you will write a few in your life.
Why you will learn many.

Your course staff

Ras

Mazz

Ben

Talia

Alvin

Ras Bodik

Started at UW in 2015 after 12 years at UC Berkeley where a version of this course was developed.

nine week-long projects

together built a browser

- parser generator
- Lua interpreter
- layout engine
- reactive programming

available as extra credit work

Best Project and Fastest Parser winners in "yellow jerseys"

Alvin Cheung

Guess which one he is!

From ACM Spring BBQ 15

Trusty TAs

Maaz Bin Safeer Ahmad

Talia Ringer

Ben Tebbs

Why are we here?

great era again for programming languages

Explosion of industrial languages

Apple	Swift, OpenCL
Facebook	Hack, React
Google	Go, Dart, Angular, GWT, MapReduce
Intel	Ct, Cilk[1]
Microsoft	C#, F#, Rx
Mozilla	JavaScript[2], Rust, asm.js
nVidia	CUDA
Wolfram	Wolfram
Yahoo	Hadoop

[1]acquired; [2]Netscape actually

Explosion of academic languages

EPFL	Scala
Northeastern, Utah, Brown, …	Racket
Edinburgh	Haskell
CMU,	ML, O'Caml
MIT	Julia, Scratch
Stanford	D3
Berkeley	Spark

Explosion of hobby languages

Yukihiro Matsumoto	Ruby
John Resig	jQuery
Walter Bright, Andrei Alexandrescu	D
Rich Hickey	Clojure
your name here	the next big thing

Why are we seeing this growth now?

Unprecedented activity since forefather languages

FORTRAN, ALGOL, Smalltalk, APL, Lisp, Simula (60/70s)

What problems are addressed by recent languages?

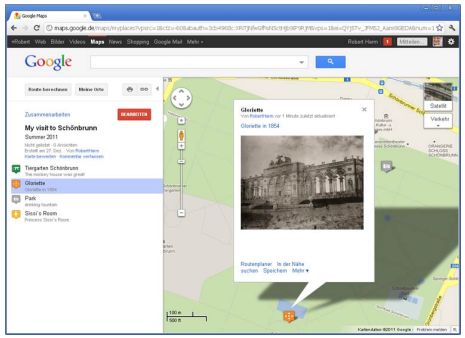
Your answers:

-- end of moore's law → it's all up to the software now (parallelism, efficiency)
-- distributed computing → languages for asynchrony (actors)
-- small form factors → low-energy computing, including smart phones, smart watches

Recent languages were truly revolutionary

without JavaScript, internet would be just hypertext

→



MapReduce enabled parallel programming in the cloud

CUDA made it possible to productively program GPUs

We now want our cake and eat it, too

Before:

- performance (eg C/C++) or
- correctness (eg Java) or
- productivity (eg Python)

Today:

Productivity and correctness:

Hack (Facebook): *gradual typing*, allows adding static type annotations to the program

Performance and correctness:

Rust (Mozilla): C++ performance with strong static type safety (prevent security holes)

Small, domain-specific langauges

The chief subject of this course

Languages created to address a problem

Abstractions design for a domain of programs

Examples:

- text processing with regular expressions
- signal processing
- math
- etc

What graduates of our course have done

Peter designed trans-compiler to build cross-platform projects; took advantage of Haskell to build a scalable concurrent system; DSL for mobile apps

Compiler-style vibrant industrial projects

Industry not only designs new languages. Tons of interesting projects on new compilers and programmer tools.

- LLVM compiler at Apple
- V8 virtual machine at Google
- Multilingual VM at Oracle
- And many more at Mozilla, Intel, Microsoft, …

Abstractions

the building blocks of systems

Hardware abstraction: unix file descriptors

- Abstract all input / output resources

Tapes
Text terminals
Pipes
…

- Representation: non-negative integers in C
- Common functionalities:
  - open / close
  - read / write / seek
- Compositional?

Software abstraction: SQL

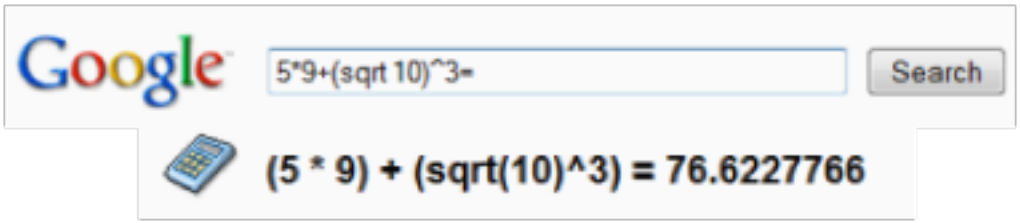
- Abstract all data structures used to implement DB
  - Heap files
  - B-trees
- Representation: relations (think spreadsheets)
- Common functionalities:
  - Create / update / delete relations
  - Create / read / update / delete records from relations
- Compositional:
  - Can combine multiple relations together

Cool abstractions for new problems

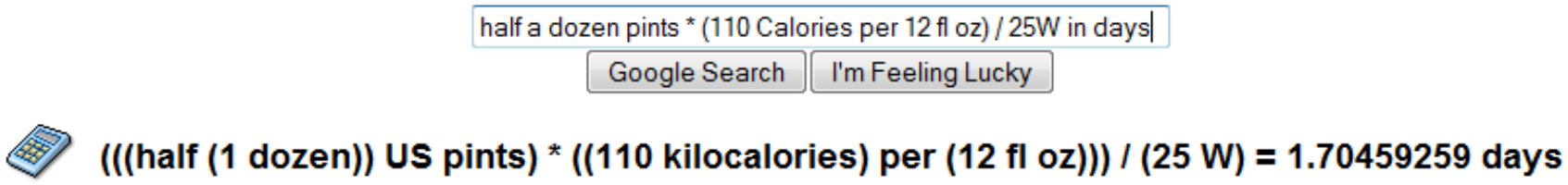
or why you will develop a new language

1. You work in a little web search company

First you add a calculator to the search engine



Then you remember cs164 and add unit conversion …



… allowing the humanity to settle silly bar bets such as:
How long can a brain run on energy from half dozen beers?

2. Then you work in a browser startup

You observe JavaScript programmers and take pity. Instead of

```
var nodes = document.getElementsByTagName('a');
for (var i = 0; i < nodes.length; i++) {
    var a = nodes[i];
    a.addEventListener('mouseover', function(event) { event.target.style.backgroundColor='orange'; }, false );
    a.addEventListener('mouseout', function(event) { event.target.style.backgroundColor='white'; }, false );
}
```

Where is the boilerplate code that we may wish to abstract away?

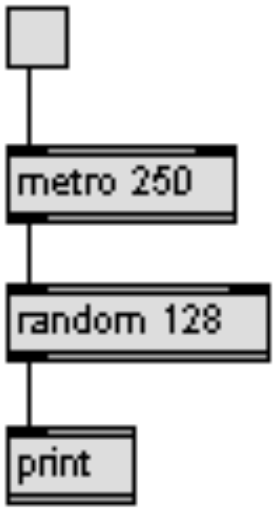
you let them be concise, abstracting node iteration, and plumbing

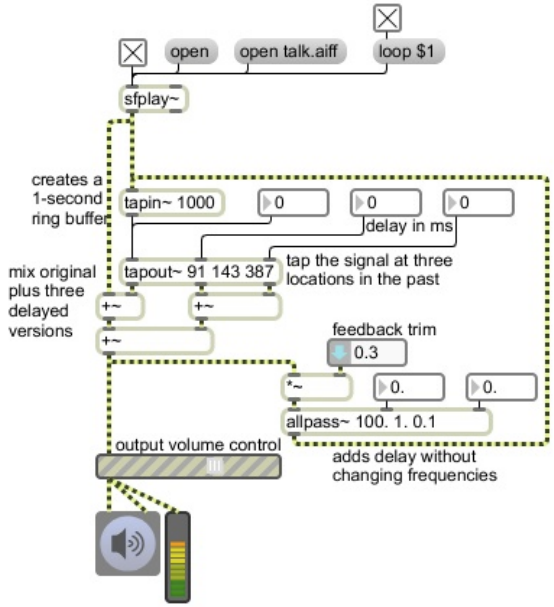
```
jQuery('a').hover( function() { jQuery(this).css('background-color', 'orange'); },
                   function() { jQuery(this).css('background-color', 'white'); } );
```


jQuery was developed by John Resig, now at Mozilla

3. Or you write visual scripting for musicians

Allowing non-programmers produce interactive music by "patching" visual metaphors of electronic blocks:





Max/MSP was created by Miller Puckette and is now developed by Cycling '74.

4. Spark: raise the Hadoop abstraction level

Write data processing applications quickly

in Java, Scala or Python

Spark offers over 80 high-level operators

Spark programs mapped to parallel Hadoop programs

Python | Scala | Java

```
val count = spark.parallelize(1 to NUM_SAMPLES).map{i =>
  val x = Math.random()
  val y = Math.random()
  if (x*x + y*y < 1) 1 else 0
}.reduce(_ + _)
println("Pi is roughly " + 4.0 * count / NUM_SAMPLES)
```

More examples of how 401 will help you

5. rfig, language for slide presentations (Percy Liang)
6. Valgrind, a tool for finding memory leaks
7. Roll your own make/ant in Python (Bill McCloskey)
8. Ruby on Rails (a language on top of Ruby)
9. Custom scripting languages (eg for testing)
10. Custom code generators (eg for new hardware)

Sample of 401 independent final projects

NAtural LAnguage in the Shell

- users express bash commands using natural language

Langauge for Autograder authoring

- checks for correct sequence of calls to the OS

Algorithm visualization

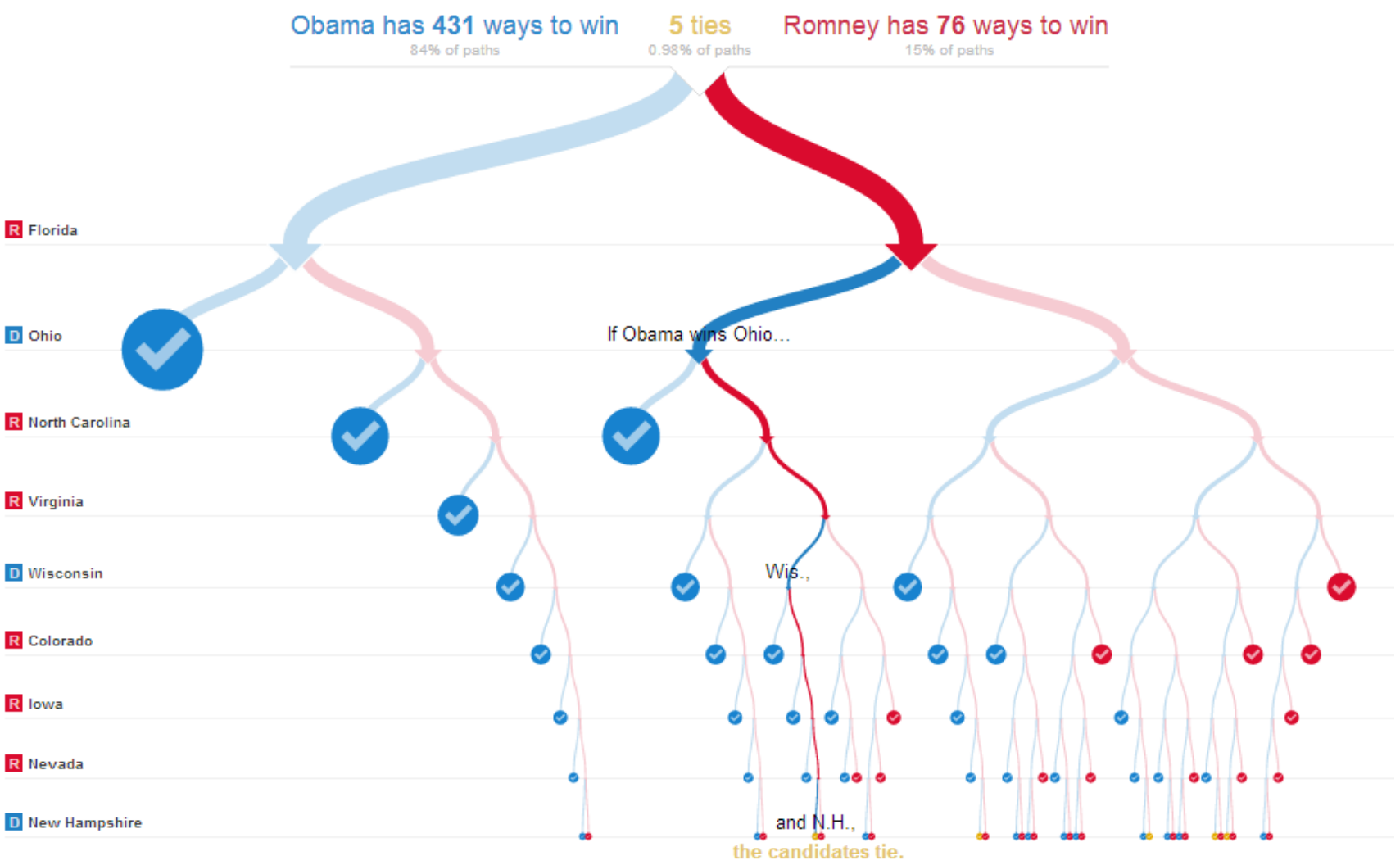
- minimizing the changes to the algorithm

Syncing JS objects via the cloud

- In the presence of asynchronous updates

Abstractions in the D3 visualization language
and introduction to PA1

10. d3 for data visualization



d3.js was developed by Mike Bostock (now NY Times)

PA1: the first programming assignment in 401

Visualization of programming languages

- x: birth year of the language
- y: popularity measured as # repositories created

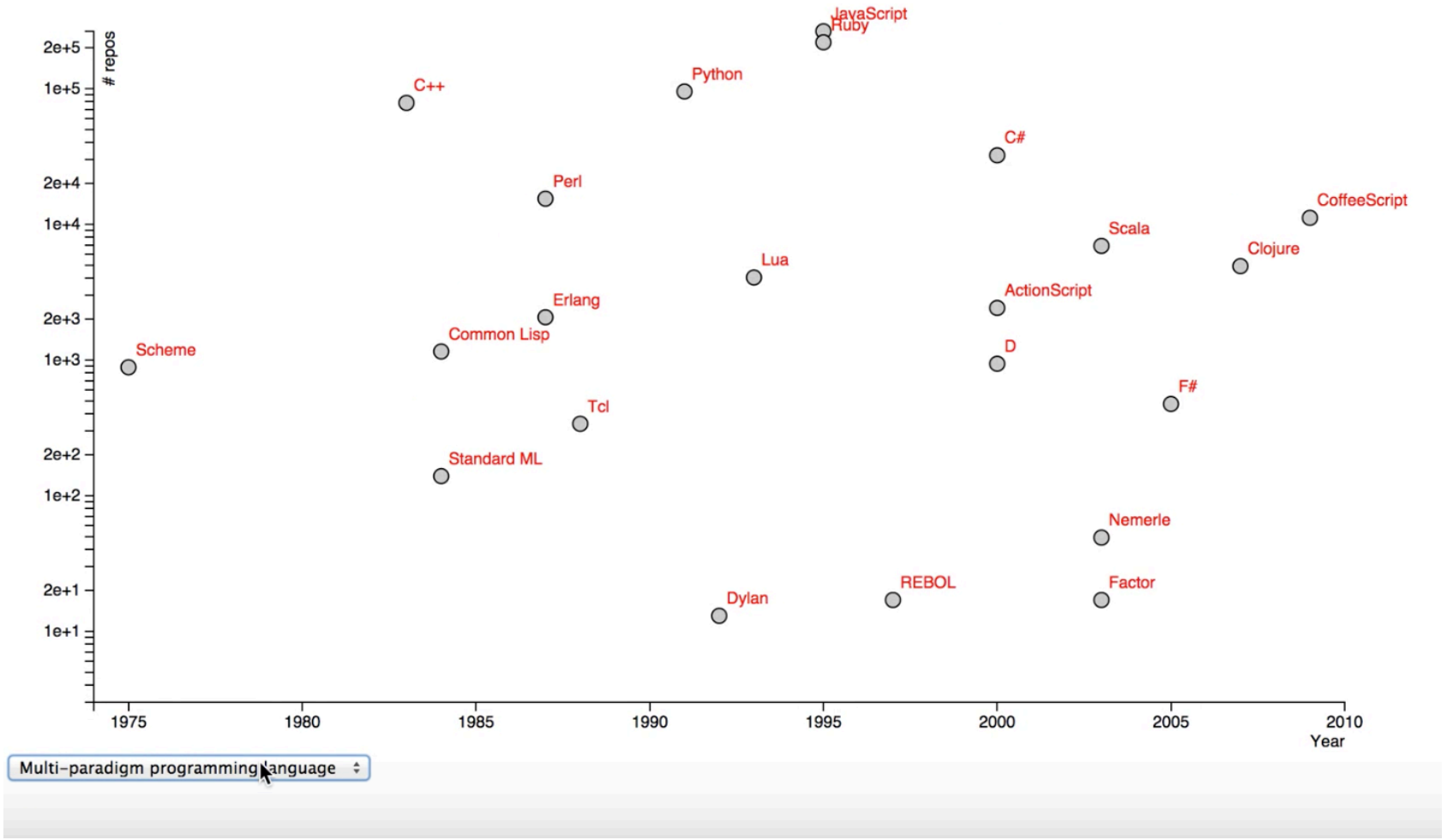
We want to hide/show languages based on attributes

- Eg, functional, procedural

Rescale the x, y axis as languages (dis)appear

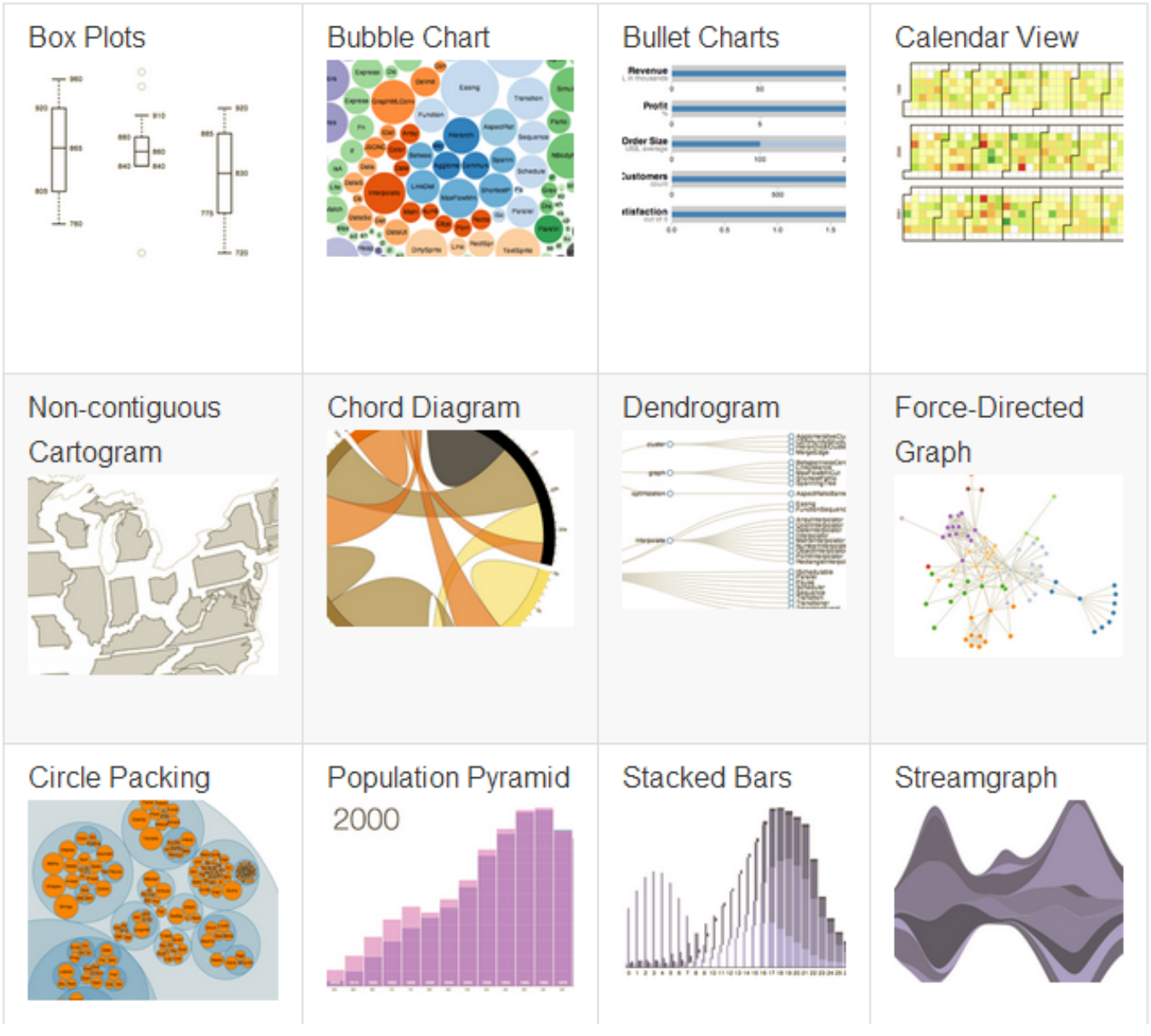
- animate the languages during rescaling

PA1: Scatterplot with animations



https://www.youtube.com/watch?v=qX-dM3OhaxM

D3 gallery (https://github.com/mbostock/d3/wiki/Gallery)



Abstractions and operations that we need

- Visual elements (circles, bars, lines)
- Data (whose values influence visual elements)
- Mapping between data and visual elements
- Changing data on the fly
- … and correspondingly changing visual elements

D3's minimalistic idea

Work directly with the browser's visual elements

- e.g. SVG circles and rectangles,
- organized in the browser DOM tree

Bind data directly to these browser elements!

- Each circle stores the datum that it visualizes

The *selection* type is the key abstraction

- selection: a simple list of DOM nodes (visual elements)
- to name a selection, reuse CSS selectors

Let's explore the selection abstraction

Based on the excellent tutorial by Mike Bostock:

Three Little Circles: http://bost.ocks.org/mike/circles/

Three circles specified as “declarative data”

Once upon a time, there were three little circles.

```
<svg width="720" height="120">
  <circle cx="40" cy="60" r="10"></circle>
  <circle cx="80" cy="60" r="10"></circle>
  <circle cx="120" cy="60" r="10"></circle>
</svg>
```

D3 offers programmatic manipulation of circles

The d3.selectAll method takes a selector string, such as `"circle"`, and returns a *selection* representing all elements that match the selector:

```
var circle = d3.selectAll("circle");
```

With a selection, we can make various changes to selected elements. For example, we might change the fill color using selection.style and the radius using selection.attr:

```
circle.style("fill", "steelblue");
circle.attr("r", 30);
```

The above code sets styles and attributes for all selected elements to the same values.

Anonymous functions per element

We can also set values on a per-element basis by using anonymous functions. The function is evaluated once per selected element. Anonymous functions are used extensively in D3 to compute attribute values, particularly in conjunction with scales and shapes. To set each circle's *x*-coordinate to a random value:

```
circle.attr("cx", function() { return Math.random() * 720; });
```

If you run this code repeatedly, the circles will dance:

Binding Data

More commonly, we use *data* to drive the appearance of our circles. Let's say we want these circles represent the numbers 32, 57 and 112. The selection.data method binds the numbers to the circles:

```
circle.data([32, 57, 112]);
```

Data is specified as an array of values; this mirrors the concept of a selection, which is an array of elements. In the code above, the first number (the first *datum*, 32) is bound to the first circle (the first *element*, based on the order in which they are defined in the DOM), the second number is bound to the second circle, and so on.

Visual appearance from bound data

After data is bound, it is accessible as the first argument to attribute and style functions. By convention, we typically use the name `d` to refer to bound data. To set the radius using the data:

```
circle.attr("r", function(d) { return Math.sqrt(d); });
```

This results in a primitive visualization:

Entering elements

What if we had *four* numbers to display, rather than three? We wouldn't have enough circles, and we would need to create more elements to represent our data. You can append new nodes manually, but a more powerful alternative is the *enter* selection computed by a data join.

When joining data to elements, D3 puts any leftover data — or equivalently "missing" elements — in the enter selection. With only three circles, a fourth number would be put in the enter selection, while the other three numbers are returned directly (in the *update* selection) by selection.data.

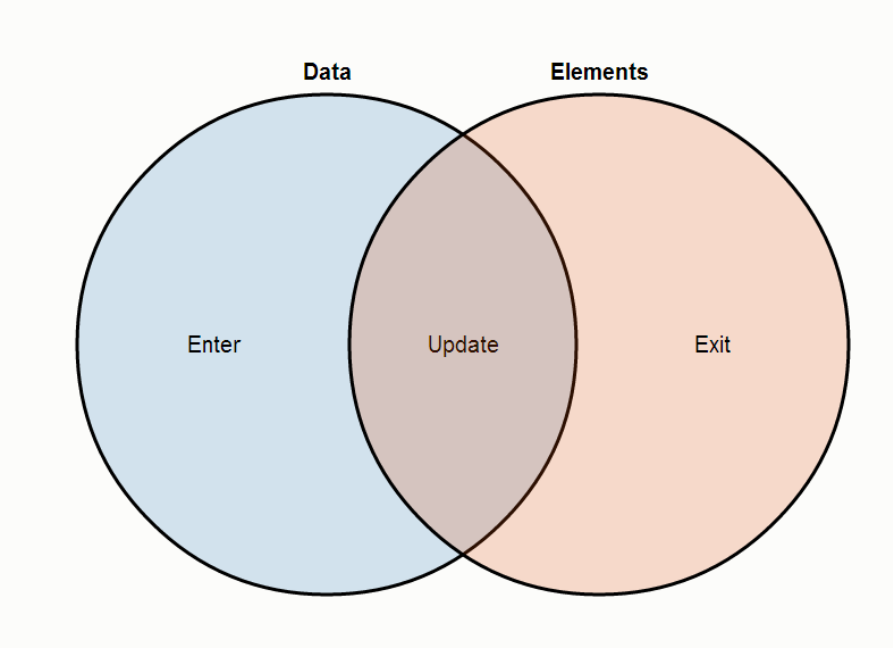
By appending to the enter selection, we can create new circles for any missing data. The new circles will be appended to the element defined by parent selection. So, we select the `"svg"` element first, then select all `"circle"` elements, and then join them to data:

```
var svg = d3.select("svg");

var circle = svg.selectAll("circle")
    .data([32, 57, 112, 293]);

var circleEnter = circle.enter().append("circle");
```

Visually

Putting everything together, consider the three possible outcomes that result from joining data to elements:

1. *enter* - incoming elements, entering the stage.
2. *update* - persistent elements, staying on stage.
3. *exit* - outgoing elements, exiting the stage.



Removing elements

The exit selection is the reflection of the enter selection: it contains the leftover elements for which there is no corresponding data.

```
var circle = svg.selectAll("circle")
    .data([32, 57]);
```

All that's left to do, then, is to remove the exiting elements:

```
circle.exit().remove();
```

And now we have two circles:

Pattern: selectAll + data + enter + append

This pattern is so common, you'll often see the selectAll + data + enter + append methods called sequentially, one immediately after the other. Despite it being common, keep in mind that this is just one special case of a data join.

```
svg.selectAll("circle")
    .data([32, 57, 112, 293])
  .enter().append("circle")
    .attr("cy", 60)
    .attr("cx", function(d, i) { return i * 100 + 30; })
    .attr("r", function(d) { return Math.sqrt(d); });
```

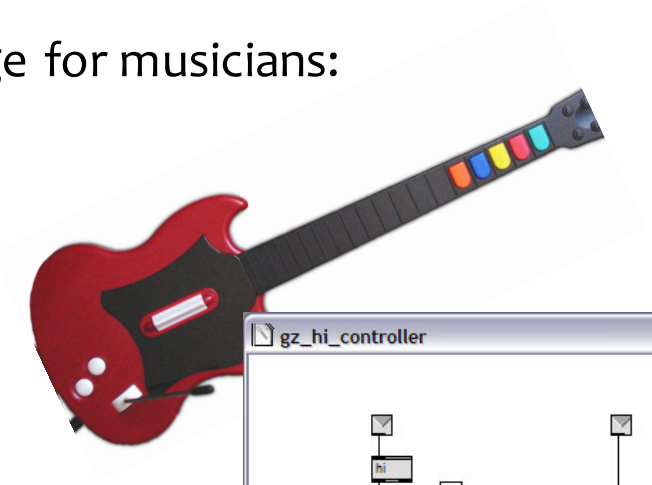
This enter pattern is often used in conjunction with method chaining, another technique for abbreviating code. Because D3 methods returns the selection they act upon, you can apply multiple operations to the same selection.

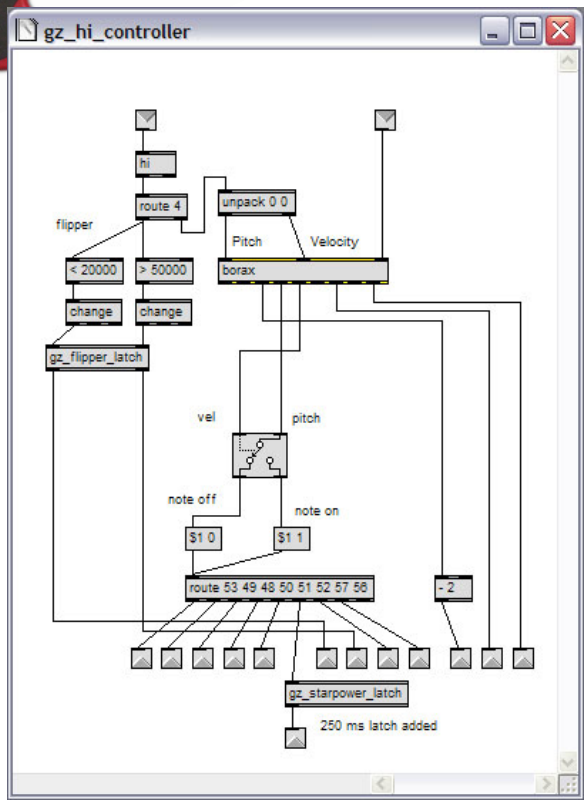
Intermission

We'll show here videos, demos, puzzles
demonstrating benefits of good programming languages

Guitair Zeros, a band enabled by Max/MSP

Recall that Max/MSP is a high-level language for musicians:
Live music synthesis controlled from MIDI





https://www.youtube.com/watch?v=uxzPCt7Pbds

Course logistics

see the course web page for more info

Redesign of the classical CSE401

Not about the usual compiler.

The new 401 will be about:

a) foundations of programming languages
b) but also how to design your own languages
c) how to implement them
d) and about PL tools, such as analyzers and bug finders
e) and also about some classical C.S. algorithms.

This will be a challenging course

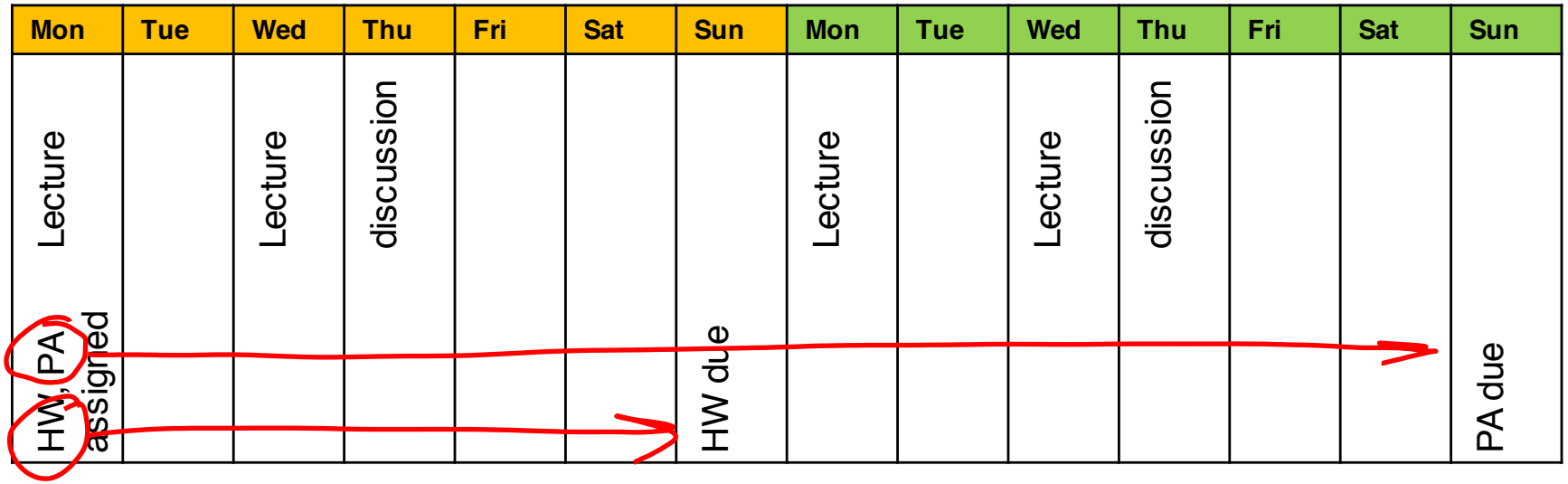
but you'll receive staff support

The HW + PA pattern

Homeworks reinforce concepts for programming assignments.

- PA in teams of three (except PA1)
- HW independent

The PA+HW pattern. It will repeat three times + HW4



We will lecture on Fridays at 2:30 to make up for the Monday holidays.

Programming assignments

PA1: D3 and call chaining

- Learn about a concrete DSL; implement a small DSL construct

PA2: Control abstraction

- bytecode compiler and interpreter, laziness, coroutines

PA3: Reactivity

- Asynchronous programming, events, GUIs

HW4: Parsing and grammars

- Writing small compilers

Final independent project (in teams)

Milestones and deliverables (see course calendar):

- You identify a problem solvable with a language
  - with our help
- You design a small language
  - you get feedback from us and from peers
- You implement it
  - in two weeks, to see small languages can be built rapidly

The grand finale: posters, demos and pizza!

- During finals week, at the time of written final exam

Late policy

PA: up to three late days, 15% penalty for each day

HW: no late days

Final Project deliverables: no late days

Exams

Midterm: Feb 2, 80 minutes, in class

Quiz: March 10, Thursday(!), evening, 60min

Final project posters: Mar 15 (Tues) 2:30 pm, 2 hours

The CSE Academic integrity

We will follow the CSE policy

http://www.cs.washington.edu/students/policies/misconduct:

Rule 1: You must indicate on your submission any assistance you received.

Rule 2: You must not share actual program code with other students.

Rule 3: You must not look at solution sets or program code from other years, nor should you make your own solutions publicly available even after the due date.

Rule 4: You must be prepared to explain any program code you submit.

Rule 5: Modifying code or other artifacts does not make it your own.

Back-to-basic lectures

No laptops and tablets in the classroom.

Exception: taking notes.

- You must email the notes to us after class

If lecture pace is slow:

- ask us for challenge problems

No textbook for this course

- slides, discussion notes, project handouts, simple papers

Add code

Overload Request Link: http://tinyurl.com/hjl3tpj

Your code word is:

[visit the instructors during office hours]

What is a language?

an implementation of an abstraction

Language

Loosely, abstractions materialized in a set of types

type: data + operations + composition

May have a mechanism for error checking (static and dynamic type checks)

May come with an (optimizing) compiler

External vs. Internal languages

The language you'll write will likely be a DSL

- a small, domain-specific language, tailored to a task

Two ways to implement:

- *External DSL*: a dedicated language
  - parsed + compiled or interpreted
- *Internal DSL*: a language created in a *host* language
  - Implemented as a library, potentially using meta-programming features of the host languages, such as macros, overloading, …

Call chaining

one possible language implementation

Call chaining is widespread and versatile

Expresses many abstractions. Embedded in many languages.

jQuery (DOM manipulation in JS)

```
$("#p1").css("color", "red").slideUp(2000).slideDown(2000);
```

Linq (DB queries in C#)

```
hostFileData
.Select(line => new { line, m = FilteringRegex.Match(line) })
.Where(item => item.m.Success)
.Select(item => …
```

Spark (data parallelism in Scala)

```
val counts = file.flatMap(line => line
                  .split(“ ”)) .map(word => (word, 1))
                  .reduceByKey(_ + _)
```

Rfig (slides in Ruby) – uses nesting of call arguments, not call chaining

```
slide!('demo 1',
       itemizeList( 'Point 1', 'Point 2', 'Point 3', nil),
       'And that\'s all for demo 1.', nil)
```

The purpose of call chaining (1)

Encourages fine-grain **decomposition** (Lego-like)

Achieved by uniformity of method signatures:

- `this` and return value special; usually the same (chain) type
- few other arguments used
- forces thinking how to break down into composable elements

For comparison, an API designed less well:

```
BufferedReader br = new BufferedReader(
            new FileReader("D:\\acontent.txt"));
    StringBuilder strBuilder = new StringBuilder();
    String line = br.readLine();
    while(line != null){
        strBuilder.append(line);
        line = br.readLine();
    }
    return strBuilder.toString();
```

The purpose of call chaining (2)

A poor man's **syntax** suggestive of composition

Achieved by variable-free code

```
d3.select("body").append("p").text("hello!");
```

… same with intermediate variables

```
var body = d3.select("body");
var p = body.append("p");
p.text("hello!");
```

The purpose of call chaining (3)

Checking for illegal compositions

Achieved by refining the chain type (open file vs. closed file)

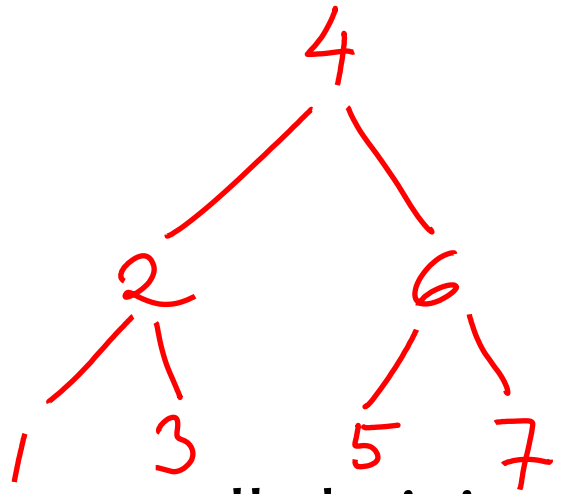
```
open(“log”,append).write(7).close().write(1)

open(“diskFile”,write).connect(“www.uw.edu”);
```

Call chaining is an alternative to what?

Instead, we could design an external language.

A binary search tree in XML:

```
<tree>
 <node id=1, p=2>1</node>
 <node id=3, p=2>3</node>
     <node id=2, p=4>2</node>
 ...
</tree>
```



… same tree data structure in one call-chaining syntax

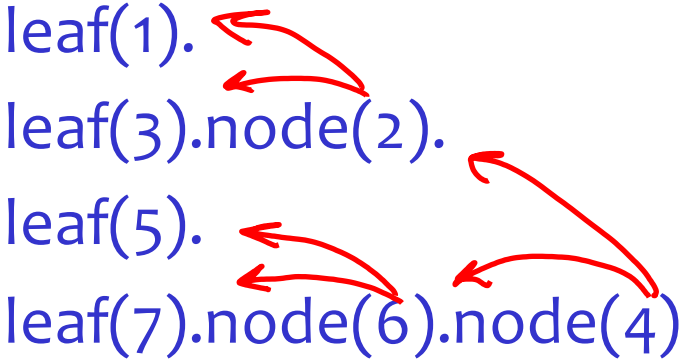
```
var tree =
leaf(1).
leaf(3).node(2).
leaf(5).
leaf(7).node(6).node(4)
```

Implementation (1)

Start with designing the component operations

Our languages for constructing binary trees:

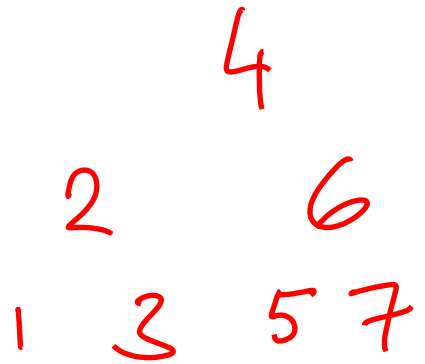
1. create a leaf
2. create an internal node and join two "most recent" subtrees



Implementation (2)

Next, ask what the chain type must be.

leaf(1)
.leaf(3)
.node(2)
.leaf(5)
.leaf(7)
.node(6)
.node(4)



The chain value must be a stack of subtrees.

The final tree is at the top of the stack.

High-order functions (HOF)

HOF: a function that accepts or returns a function.

Uses:

1) Parameterize the operator (iterators, logic)

```
d3.selectAll("p").each(function(p) {
                        console.log(p.source.id);})

circle.attr("r", function(d) { return Math.sqrt(d); });
```

2) Delaying the execution to later events (callbacks)

```
$(document).ready(function() {   … } );
```

Function operators

Passing functions is so common. We will use many non-call-chaining operators. This one is from PA1.

```
function negate(pred) {
   return function() {  return !pred.apply(this, arguments);  };
}
```

Scaling the axes in PA1

```
circles.attr("cy", function(d) { return d.nbRepos; })
    .attr("cx", function(d) { return d.year; })
    .attr("r", r);
```

How do we scale these with an elegant abstraction?

```
var yScale = d3.scale.log()
        .domain([d3.min(langs, function(d) { return parseInt(d.nbRepos); }),
                 d3.max(langs, function(d) { return parseInt(d.nbRepos); })])
         .range([h-r,r]);

circles.attr("cy", function(d) { return yScale(d.nbRepos); })
```

yScale is a function returned by `scale.log().domain().range()`

Introduction to HW1

towards programming language sophistication

HW1, PA1

Both assigned today. Due in 6 and 13 days, resp.

Individual submissions but feel free to discuss

Rationale for HW1 includes language sophistication

- such as the pitfalls of type coercion
- a sample of what programmers need to know is Gary Bernhardt's WAT talk

Conclusion

Take home points.

What is a language?

Why languages help write software.

Examples of small languages and their abstractions.

What problem will your language solve?

There is lots to be invented in programming languages.
You can take a stab at in your final project.

This quote should motivate you:

Millions for compilers, but hardly a penny for understanding human programming language use. Now, programming languages are obviously symmetrical, the computer on one side, the human on the other. In an appropriate science of computer languages, one would expect that half the effort would be on the computer side, understanding how to translate the languages into executable form, and half on the human side, understanding how to design languages that are easy or productive to use. Yet, we do not even have an enumeration of all the psychological functions programing languages serve for the user. Of course, there is lots of programming language *design*, but it comes from computer scientists. And though technical papers on languages contain main appeals to ease of use and learning, they patently contain almost no psychological evidence nor any appeal to psychological science.

- A. Newell and S. Carc

How is this compiler class different?

Not intended for advanced compiler engineers

- there are relatively few among our students

... but for software developers

- Why a developer need a PL class?

Don't be a boilerplate programmer.

Instead, build tools for users and other programmers

- Libraries, frameworks
- small languages (such as configuration languages)
- and maybe also big languages
- we saw at least 10 concrete examples in the lecture

Why a software engineer needs PL

New languages will keep coming

- choose the right one: can reduce code size ten-fold

Write code that writes code

- Compilers, code generators

Develop your own abstractions and languages

- Are you kidding? No.

Learn about compilers and interpreters.

- Programmer's main tools.

Trends in programming languages

programming language and its interpreter/compiler:

- programmer's primary tools
- you must know them inside out

languages have been constantly evolving ...

- what are the forces driving the change?

... and will keep doing so

- new programming problems need new abstractions