Hack Your Language!

CS164: Introduction to Programming Languages and Compilers, Spring 2012

UC Berkeley

Ras, Ali, and Mangpo

Lecture 1: Why Take CS164?

Today

What is a programming language

Why you will write a few in your life

How you will learn language design skills in cs164

Outline

- Ten examples of languages you may write
- A puzzle
- The puzzle solved in Prolog
- Evolution of programming abstractions
- Why you will write a small language, sw.eng. view
- The course project
- The final project: your small language
- HW1
- Other course logistics

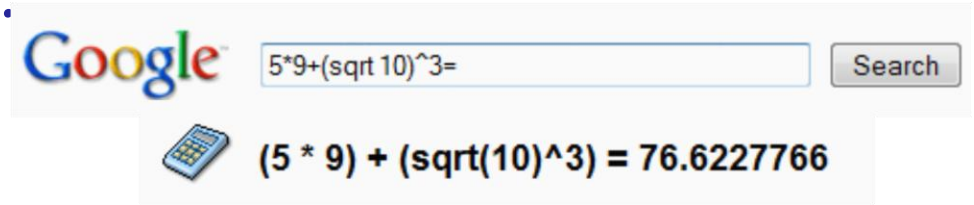
The life of a CS164 graduate

or, why you will develop a language

1. You work in a little web search company

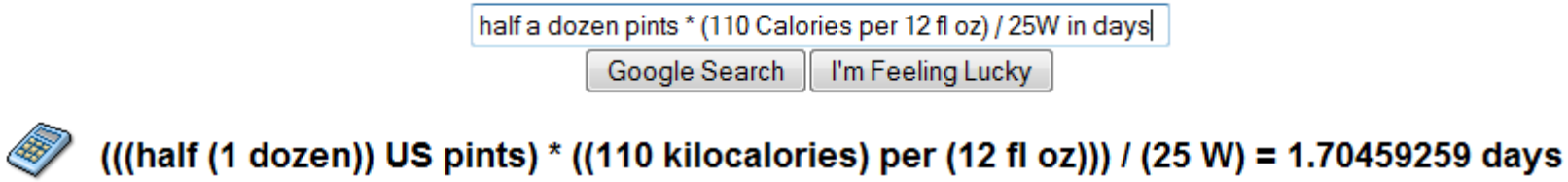
Boss says: *"We will conquer the world only if our search box answers all questions the user may ask."*

You build a calculator:



Then you remember cs164 and easily add unit conversion.

You can settle bar bets such as: How long a brain could function on 6 beers? (provided alcohol energy was not converted to fat):



You are so successful that Yahoo and Bing imitate you.

2. Then you work in a tiny browser outfit

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 );
}
```

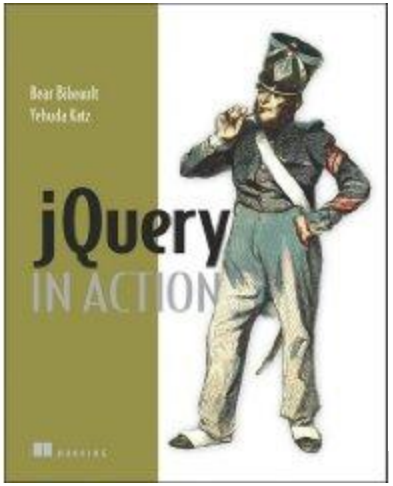
Where is the boilerplate code that we may wish to abstract away?

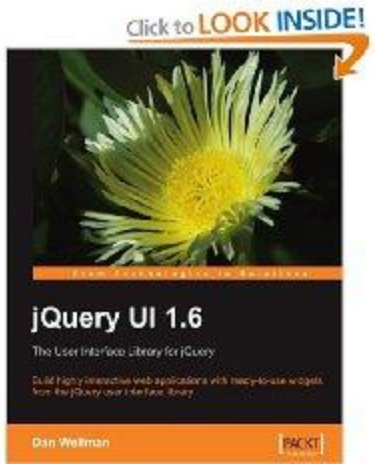
you let them be concise, abstracting node iteration, and plumbing

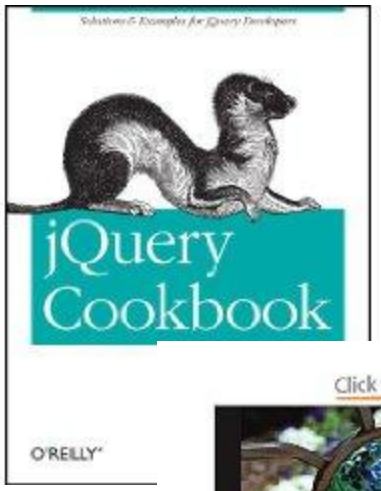
```
jQuery('a').hover( function() { jQuery(this).css('background-color', 'orange'); },
                   function() { jQuery(this).css('background-color', 'white'); } );
```

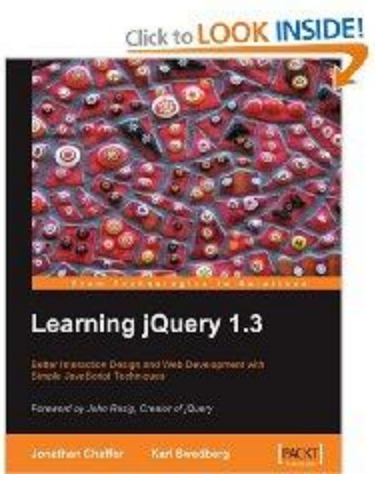

jQuery was developed by John Resig, now at Mozilla

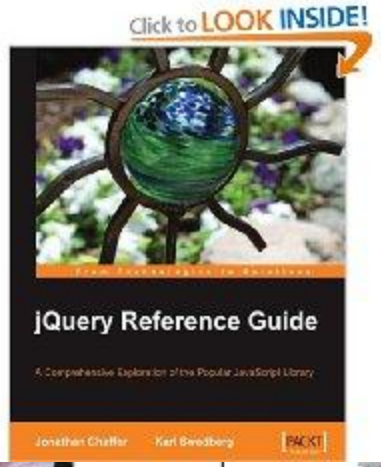
… and the fame follows

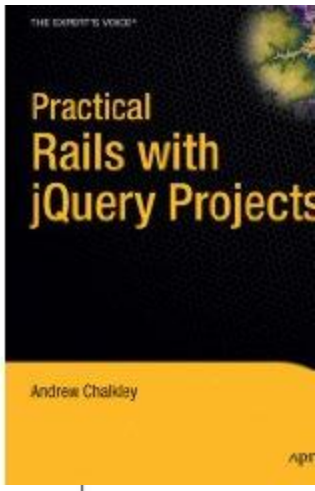


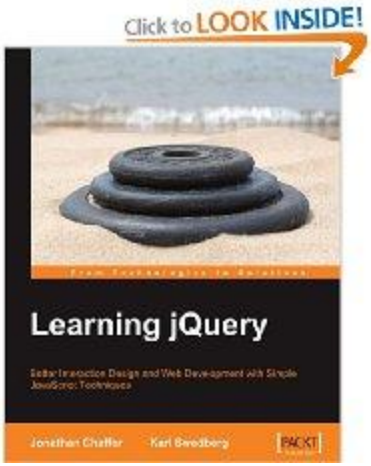


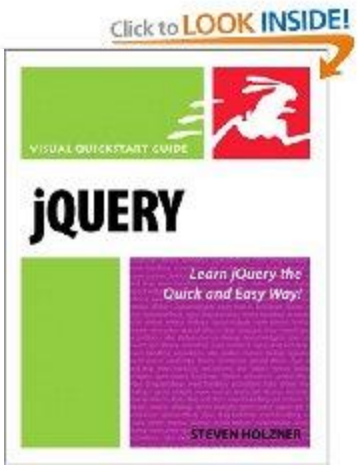


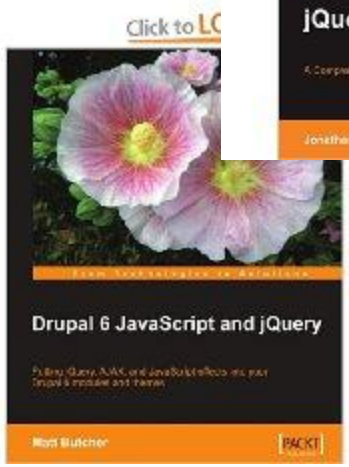






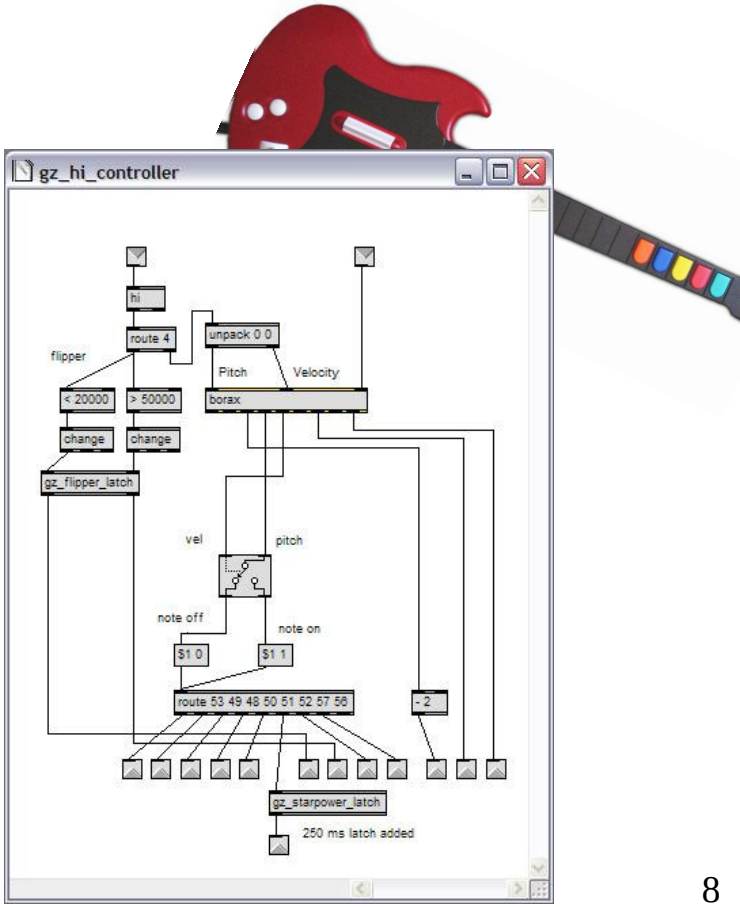






3. Or you write visual scripting for musicians

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



Guitair Zeros: a S.F. band enabled by the Max/MSP language.

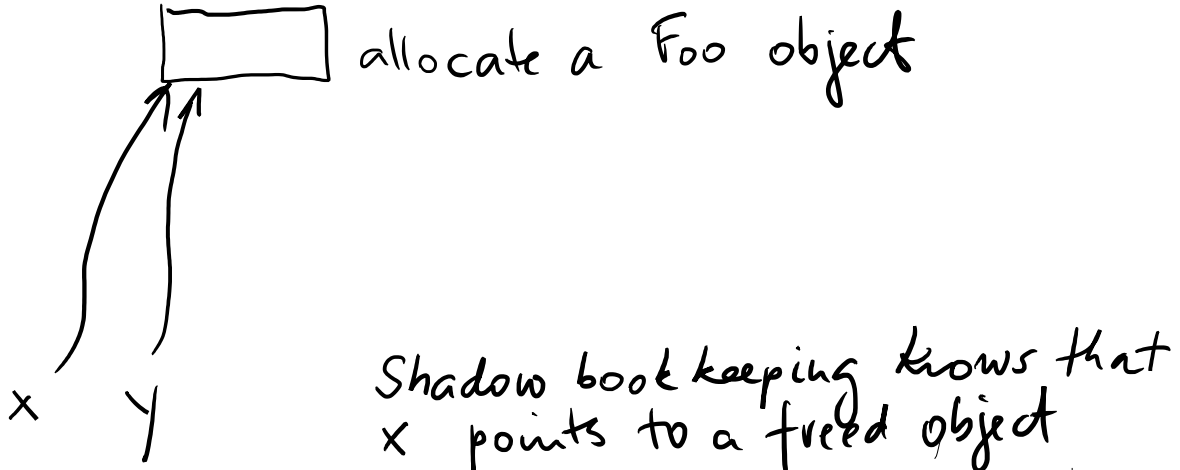
http://www.youtube.com/watch?v=uxzPCt7Pbds

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

4. Then you live in open source space

You see Linux developers suffer from memory bugs, eg buffer overruns and dangling pointers (accesses to freed memory).

```
x = new Foo()
y=x
…
free(y)
…
x.f = 5
```



You design a tool that associates each byte in memory with a shadow "validity bit", set by new and reset by free. When a memory location is accessed, you check its validity bit.

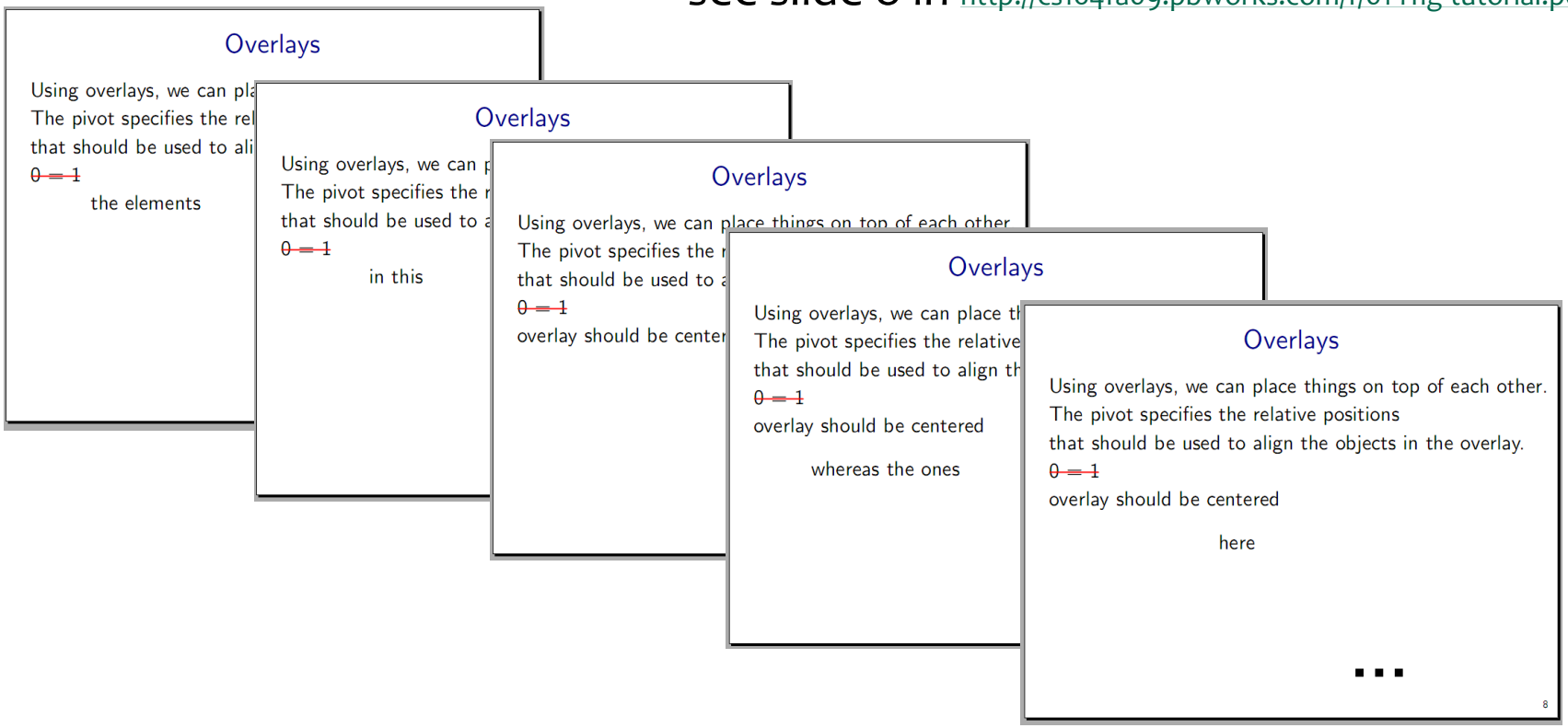
To add these checks, the implementation rewrites the binary of the program, and adds shadow memory.

Valgrind was developed by Julian Seward

5. Then you decide to get a PhD

You need to give talks but get tired of PowerPoint.
Or you realize you are not a WYSIWYG person.
You embed a domain-specific language (DSL) into Ruby.

see slide 8 in http://cs164fa09.pbworks.com/f/01-rfig-tutorial.pdf



The animation in rfig, a Ruby-based language

```
slide!('Overlays',
  'Using overlays, we can place things on top of each other.',
  'The pivot specifies the relative positions',
  'that should be used to align the objects in the overlay.',

  overlay('0 = 1', hedge.color(red).thickness(2)).pivot(0, 0),

  staggeredOverlay(true, # True means that old objects disappear
    'the elements', 'in this', 'overlay should be centered', nil).pivot(0, 0),

  cr, pause,            # pivot(x, y): -1 = left, 0 = center, +1 = right

  staggeredOverlay(true,
    'whereas the ones', 'here', 'should be right justified', nil).pivot(1, 0),

  nil) { |slide| slide.label('overlay').signature(8) }
```

call chaining: convenient for setting multiple attributes

rfig was developed by Percy Liang, a Berkeley student

More examples of how cs164 will help

6. ProtoVis, a DSL for data visualization
7. Roll your own make/ant in Python (Bill McCloskey)
8. Ruby on Rails (another system on top of Ruby)
9. Custom scripting languages (eg for testing)
10. Custom code generators (eg for new hardware)

11. Your language or tool here.

Additional reasons to take cs164

Choose the right language

can reduce code size 10x

Language Sophistication expected of programmers

- understand Java generics
  (what are variant, co-variant types?)

- understand pitfalls of coercion
  (see Gary Bernhardt's WAT talk)

Summary

Summary: Don't be a boilerplate programmer.

Instead, build tools for users and other programmers

- libraries
- frameworks
- code generators
- small languages (such as configuration languages)
- and maybe also big languages
- we just saw 10 concrete examples

Summary

Take historical note of textile and steel industries:

do you want to design machines and tools

or

do you want to operate those machines?

be a designer

Take cs164. Become unoffshorable.

"We design them here, but the labor is cheaper in Hell."

How is this PL/compiler class different?

Not really a compiler class.

It's about:

a) foundations of programming langauges
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.

a 5-minute intermission with a puzzle

solve with your neighbor

Puzzle: Solve it. It's part of the lecture.

9 • A New "Colored Hats" Problem

Three subjects—A, B, and C—were all perfect logicians. Each could instantly deduce all consequences of any set of premises. Also, each was aware that each of the others was a perfect logician. The three were shown seven stamps: two red ones, two yellow ones, and three green ones. They were then blindfolded, and a stamp was pasted on each of their foreheads; the remaining four stamps were placed in a drawer. When the blindfolds were removed, A was asked, "Do you know one color that you definitely do not have?" A replied, "No." Then B was asked the same question and replied, "No."

Is it possible, from this information, to deduce the color of A's stamp, or of B's, or of C's?

From The Lady or the tiger, R. Smulyan

Solution

9 ◆ The only one whose color can be determined is C. If C's stamp were red, then B would have known that his stamp was not red by reasoning: "If my stamp were also red, then A, seeing two red stamps, would know that his stamp is not red. But A does not know that his stamp is not red. Therefore, my stamp cannot be red."

This proves that if C's stamp were red, then B would have known that his stamp was not red. But B did not know that his stamp was not red; therefore, C's stamp cannot be red.

The same argument, replacing the word *red* with *yellow*, shows that C's stamp cannot be yellow either. Therefore, C's stamp must be green.

most of us got it wrong

=> can we ask a computer to do the search thru possible alternatives?

yes, let's write a program.

Languages as thought shapers

http://soft.vub.ac.be/~tvcutsem/whypls.html>

Language as a thought shaper

We will cover less traditional languages, too.

The reason:

> *A language that doesn't affect the way you think about programming, is not worth knowing.*
>
> *an Alan Perlis epigram <http://www.cs.yale.edu/quotes.html>*

One of thought-shaper languages is Prolog.

You will both program in it and implement it.

Solving the puzzle with Prolog (part 1)

We'll use integers 1..7 to denote the stamps.

Numbers 1 and 2 denote red stamps.

Yellow stamps are 3 and 4 …

```
red(1).
red(2).
yellow(3).
yellow(4).
green(5).
green(6).
green(7).
```

facts

:- red(3)? No

:- yellow(X)?

X = 3

X = 4

Solving the puzzle with Prolog (part 2)

S is variable that can be bound to a stamp, ie to a number 1, 2, ..7.

With the following, we say that a stamp is either a red stamp or a yellow stamp or a green stamp.

```
stamp(S) :- red(S) ; yellow(S) ; green(S).
```

; OR

We now state what three stamps, A, B, C, we may see on the three heads.
(We are saying nothing more than no stamp may appear on two heads.)

```
valid(A,B,C) :- stamp(A), stamp(B), stamp(C), A\=B, B\=C, A\=C.
```

Solving the puzzle with Prolog (part 3)

Now to the interesting parts. Here we encode the fact that logician **a** can't rule out any color after seeing colors B and C on **b**'s and **c**'s heads.

We define the so-called *predicate* `a_observes(B,C)` such that it is true iff all of red, yellow, green are valid colors for **a**'s head given colors B and C.

```
a_observes(B,C) :- red(R),    valid(R,B,C),
                   yellow(Y), valid(Y,B,C),
                   green(G),  valid(G,B,C).
```

that is, a_observes is false if a can rule out some color

Solving the puzzle with Prolog (part 4)

How to read this rule: a_observes(B,C) is called with B and C bound to colors on **b** and **c**'s heads, say B=3 and C=6. The rule the "reduces" to

```
a_observes(3,6) :- red(R),    valid(R,3,6),
                   yellow(Y), valid(Y,3,6),
                   green(G),  valid(G,3,6).
```

The Prolog interpreter then asks:

Does there exist a red stamp that I can assign to R such that it forms valid stamps assignment with 3 and 6? Same for the yellow Y and the green G.

Solving the puzzle with Prolog (part 5)

Similarly, we encode that the logician **b** can't rule out any color either.

Careful, though: In addition to making deductions from the stamps **b** can see (A,C), **b** also considers whether the logician **a** would answer "No" for each of the possible colors of B and C.

```
b_observes(A,C) :- red(R),    valid(A,R,C), a_observes(R,C),
                   yellow(Y), valid(A,Y,C), a_observes(Y,C),
                   green(G),  valid(A,G,C), a_observes(G,C).
```

Finding solution with Prolog (part 6)

Predicate `solution(A,B,C)` will be true if stamps A,B,C meet the constraints of the problem statement (ie, **a** and **b** both answer "No").

```
solution(A,B,C) :- stamp(A), stamp(B), stamp(C),
                   a_observes(B,C),
                   b_observes(A,C).
```

We can ask for a solution, ie an assignment of stamps to heads

```
?- solution(A,B,C).
A = 1
B = 2
C = 5
Yes
```

Finding solution with Prolog (part 7)

Finally, we can ask whether there exists a solution where a particular stamp has a given color. Here, we ask if **c**'s color can be red, and then green:

```
?- solution(A,B,C),red(C).
No

?- solution(A,B,C),yellow(C).
No

?- solution(A,B,C),green(C).
A = 1
B = 2
C = 5
Yes
```

Prolog summary

Forces you to think differently than a, say, object-oriented language. Sometimes this way of thinking fits the problem well, as in the case of our puzzle.

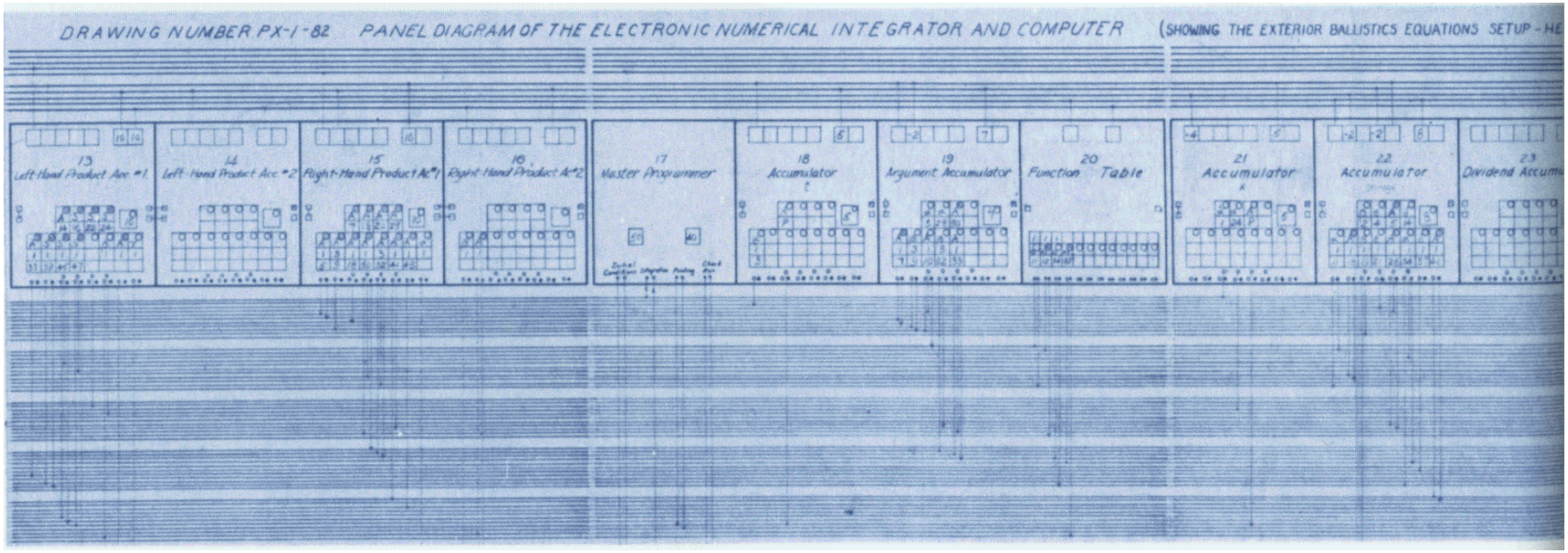
You will implement Prolog, program in it, and write a simple translator from a simple natural language to Prolog programs --- a puzzle solver!

Programming Abstractions

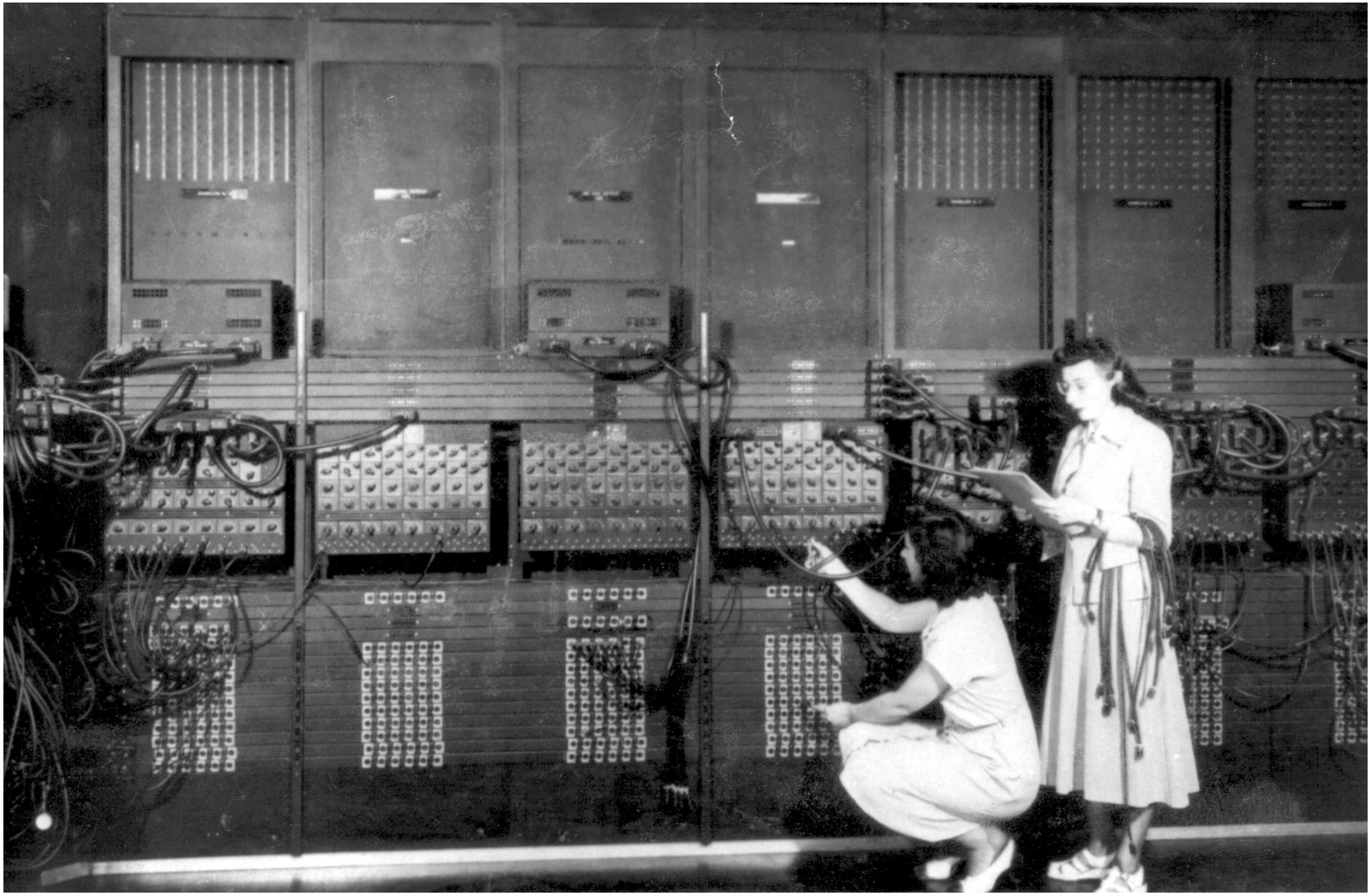
what are they?

ENIAC (1946, University of Philadelphia)

ENIAC program for external ballistic equations:



Programming the ENIAC



ENIAC (1946, University of Philadelphia)

programming done by

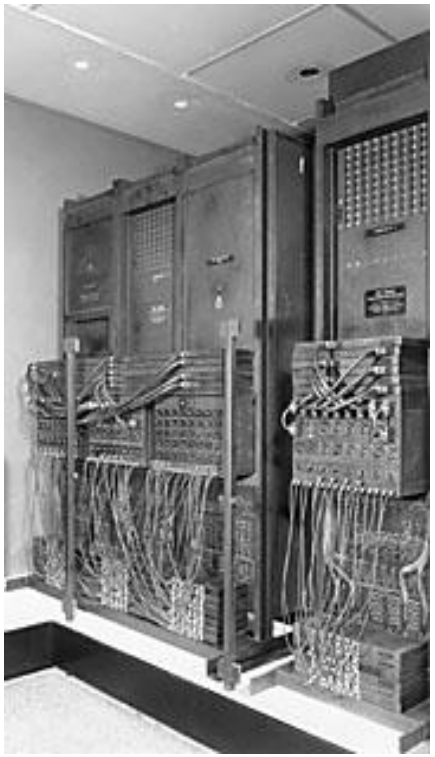
- rewiring the interconnections
- to set up desired formulas, etc

Problem (what's the tedious part?)

- programming = rewiring
- slow, error-prone

solution:

- **store the program in memory!**
- birth of ***von Neuman*** paradigm



UDSAC (1947, Cambridge University)

the first real computer

- large-scale, fully functional, **stored-program** electronic digital computer (by Maurice Wilkes)

problem: Wilkes realized:

- "a good part of the remainder of my life was going to be spent in finding errors in ... programs"

solution: procedures (1951)

- procedure: abstracts away the implementation
- reusable software was born

Assembly – the language (UNIVAC 1, 1950)

Idea: mnemonic (assembly) code

- Then translate it to machine code by hand (no compiler yet)
- write programs with mnemonic codes (add, sub), with symbolic labels,
- then assign addresses by hand

Example of symbolic assembler

clear-and-add a

add b

store c

translate it by hand to something like this (understood by CPU)

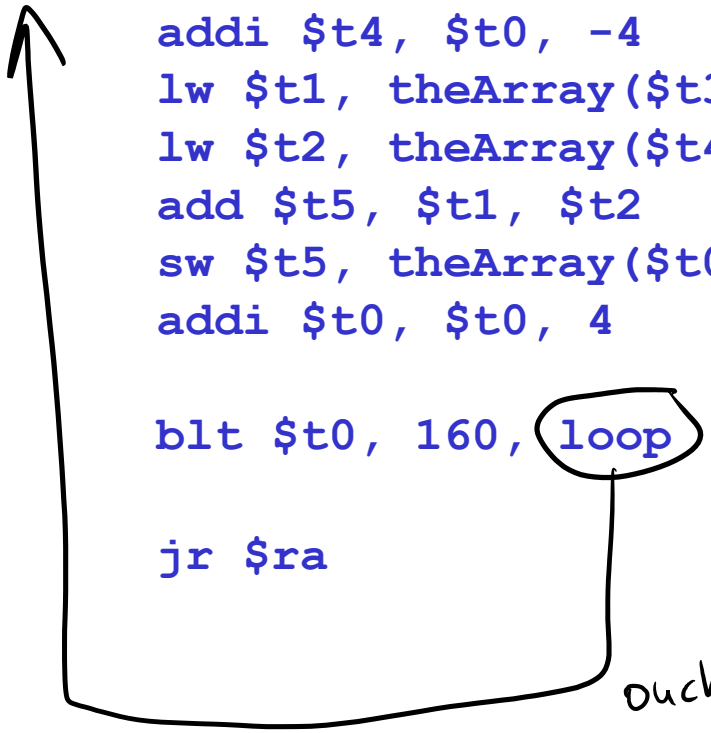
B100 A200

C300

Assembler – the compiler (Manchester, 1952)

- it was assembler nearly as we know it, called AutoCode
- a loop example, in MIPS, a modern-day assembly code:

```
loop: addi $t3, $t0, -8
      addi $t4, $t0, -4
      lw $t1, theArray($t3)        # Gets the last
      lw $t2, theArray($t4)        #  two elements
      add $t5, $t1, $t2            # Adds them together...
      sw $t5, theArray($t0)        # ...and stores the result
      addi $t0, $t0, 4             # Moves to next "element"
                                   # of theArray
      blt $t0, 160, loop           # If not past the end of
                                   # theArray, repeat
      jr $ra
```



ouch, why do labels need to be visible to programmers?

Assembly programming caught on, but

Problem: Software costs exceeded hardware costs!

John Backus: "Speedcoding"

- An interpreter for a high-level language
- Ran 10-20 times slower than hand-written assembly
  - way too slow

FORTRAN I (1954-57)

Langauge, and the first compiler

- Produced code almost as good as hand-written
- Huge impact on computer science (laid foundations for cs164)
- Modern compilers preserve its outlines
- FORTRAN (the language) still in use today

By 1958, >50% of all software is in FORTRAN

Cut development time dramatically

- 2 weeks → 2 hrs
- that's more than 100-fold

FORTRAN I (IBM, John Backus, 1954)

Example: nested loops in FORTRAN

- a big improvement over assembler,
- but annoying artifacts of assembly remain:
  - labels and rather explicit jumps (CONTINUE)
  - lexical columns: the statement must start in column 7

- The MIPS loop from previous slide, in FORTRAN:

```
      DO 10 I = 2, 40
      A[I] = A[I-1] + A[I-2]
10    CONTINUE
```

Side note: designing a good language is hard

Good language protects against bugs, but lessons take a while.
An example that caused a failure of a NASA planetary probe:

buggy line:

```
DO 15 I = 1.100
```

what was intended (a dot had replaced the comma):

```
DO 15 I = 1,100
```

because Fortran ignores spaces, compiler read this as:

```
DO15I = 1.100
```

which is an assignment into a variable DO15I, not a loop.

This mistake is harder to make (if at all possible) with the modern lexical rules (white space not ignored) and loop syntax

```
for (i=1; i < 100; i++) { … }
```

Goto considered harmful

L1: statement

if expression goto L1

statement

Dijkstra says: gotos are harmful

- use structured programming
- lose some performance, gain a lot of readability

how do you rewrite the above code into structured form?

Object-oriented programming (1970s)

The need to express that more than one object supports draw() :

```
draw(2DElement p) {
    switch (p.type) {
      SQUARE: … // draw a square
                break;
       CIRCLE: … // draw a circle
                break;
    }
}
```

Problem:

unrelated code (drawing of SQUARE and CIRCLE) mixed in same procedure

Solution:

Object-oriented programming with inheritance

Object-oriented programming

In Java, the same code has the desired separation:

```
class Circle extends 2DElement {
  void draw() { <draw circle> }
}
class Square extends 2DElement {
  void draw() { <draw circle> }
}
```

the dispatch is now much simpler:

```
p.draw()
```

Review of historic development

- wired interconnects → stored program (von Neuman machines)
- lots of common bugs in repeated code → procedures
- machine code → symbolic assembly (compiled by hand)
- tedious compilation → assembler (the assembly compiler)
- assembly → FORTRAN I
- gotos → structured programming
- hardcoded "OO" programming → inheritance, virtual calls

Do you see a trend?

- Removal of boilerplate code
  - also called **plumbing**, meaning it conveys no application **logic**
- Done by means of new abstractions, such as procedures
  - They abstract away from details we don't wish to reason about

Where will languages go from here?

The trend is towards higher-level abstractions

- express the algorithm concisely!
- which means hiding often repeated code fragments
- new constructs hide more of these low-level details.

Haven't we abstracted most programming plumbing?

- no, history repeats itself
- it is likely that new abstractions will be mostly domain specific ==> small language will prevail

New Languages will Keep Coming

A survey: how many languages did you use?

Let's list the more unusual here:

Be prepared to program in new languages

Languages undergo constant change

- FORTRAN 1953
- ALGOL 60 1960
- C 1973
- C++ 1985
- Java 1995

Evolution steps: 12 years per widely adopted language

- are we overdue for the next big one?

... or is the language already here?

- *Hint:* are we going through a major shift in what computation programs need to express?
- your answer here: JS

Develop your own language

Are you kidding? No. Guess who developed:

- PHP
- Ruby
- JavaScript
- perl

Done by hackers like you

- in a garage
- not in academic ivory tower

Our goal: learn good academic lessons

- so that your future languages avoid known mistakes

Another reason why you'll develop a language

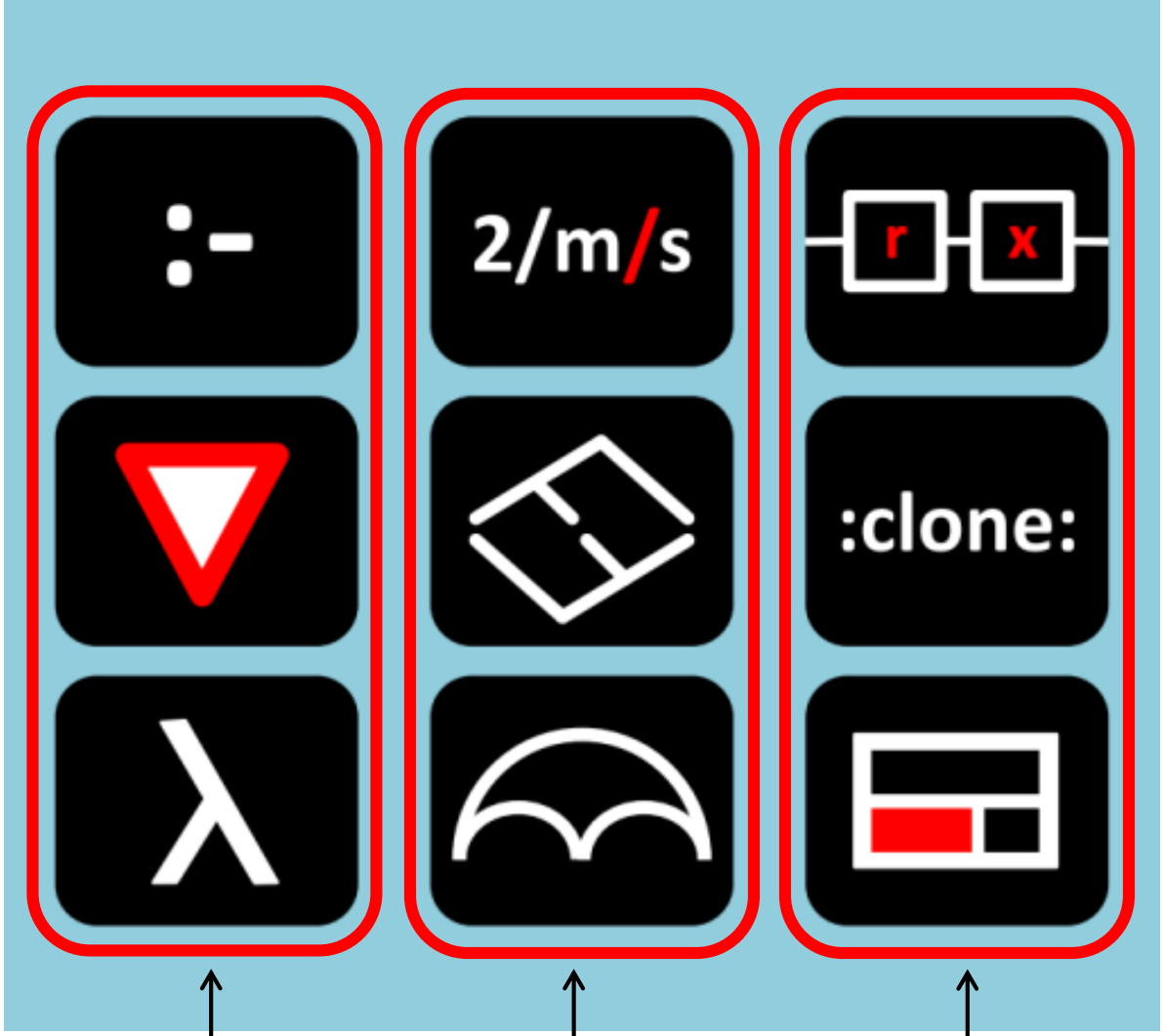
Software architectures evolved from

- libraries
- frameworks
- small languages

Read in the lecture notes about MapReduce evolution

The course project assignments

Nine weekly projects

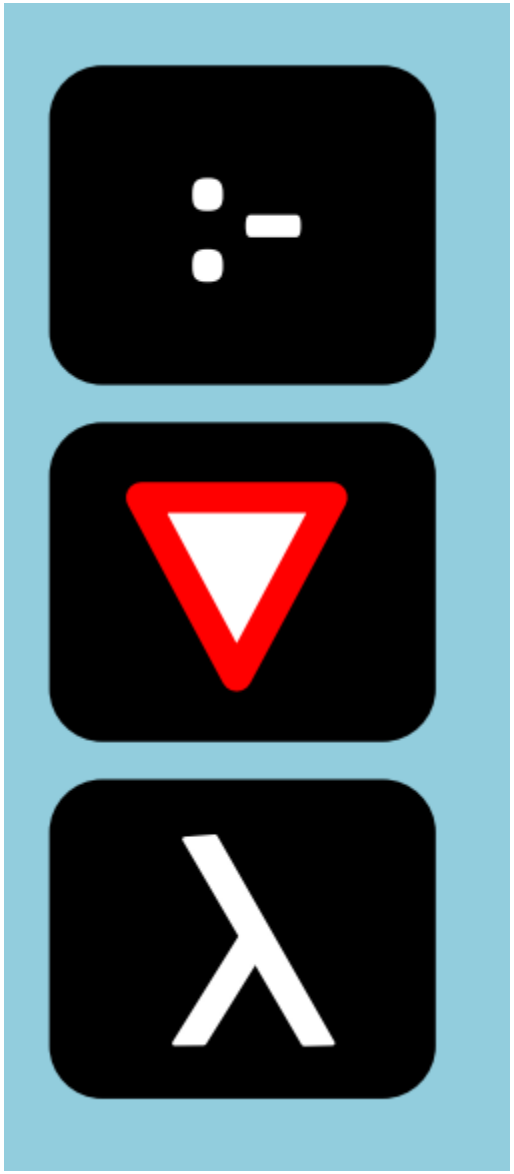


interpreter (abstractions)

parser (syntax-directed xslation, external DSLs)

web browser (embedded DSLs, concurrency)

Language abstractions



prolog

with coroutines, it's a few lines;

later used

yield: full coroutines

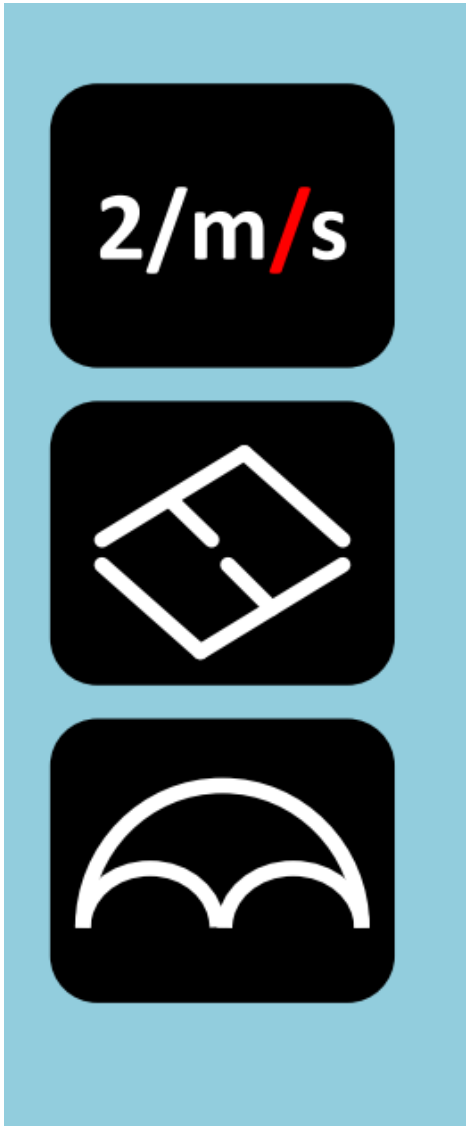
bytecode compiler, regex, backtracking

interpreter

closures, lexical scoping,

desugaring, iterators

Parsing and external DSLs



google calculator

half a dozen pints * (110 Calories per 12 fl oz) / 25W in days

Google Search | I'm Feeling Lucky

syntax-directed translation

disambiguation (%left, %prec)

Earley parser

Web browser: layout and scripting

Rx

from events to high-order dataflow; reactive programming a`la RxJS

jQuery

embedded DSL for declarative DOM manipulation

layout engine

from constraints to attribute grammars; add OO to language

Contest winners in yellow jerseys

Testimonials

We are just about done with PA6, and I am still marveling at what we have created.

I think this class was an amazing first choice at an upper-div CS class. (Although the workload is pretty brutal)

final project

Final project

Replaces the written final exam

- show-and-tell with posters, demos and pizzas and t-shirts

You identify a problem solvable with a language

- with our help

You design a small language

- we give you feedback + you give each other feedback

You implement it

- in two weeks, to see small languages can be built rapidly

See course calendar for milestones

course logistics

(see course info on the web for more)

Back-to-basic Thursdays

No laptops in the classroom on Thursdays

HW1: A mashup with GreaseMonkey

Assigned **Today,** due Monay

Get familiar with the languages of the browser

HTML, JavaScript, regexes, DOM, GreaseMonkey

Ger familiar with a DSL and its constructs

d3, a language for visualizations

Observe that we program with multiple languages

each used for a distinct task

Reflect on the flaws of these languages

and suggest ideas improvements

Calendar (will be updated on the web page)

CS164 Spring 2012 Schedule

week	lect	week		Lecture	Projects	HW
1		16-Jan	Mo			
	1	17-Jan	Tu	why take cs164		HW: greasemonkey + regex (medium hard)
		18-Jan	We			
	2	19-Jan	Th	interpreters: gcalc		
		20-Jan	Fr			
2		23-Jan	Mo			
	3	24-Jan	Tu	closures and scoping	PA1: basic **interpreter,** with closures, iterators, a some constructs via desugaring.	
		25-Jan	We			
	4	26-Jan	Th	control abstraction		
		27-Jan	Fr			
3		30-Jan	Mo			
	5	31-Jan	Tu	coroutines and bytecode interp	PA2: **Coroutines** and bytecode compiler. More powerful iterators. Implement backtracking-based regex matching with coroutines.	
		1-Feb	We			
	6	2-Feb	Th	logic programming		
		3-Feb	Fr			
4		6-Feb	Mo			
	7	7-Feb	Tu	Prolog	PA3: Build a **Prolog interpreter** on top of your coroutines and streams. Use your Prolog to implement a simple parser and solve puzzle problems.	
		8-Feb	We			
	8	9-Feb	Th	parsing		
		10-Feb	Fr			
5		13-Feb	Mo			
	9	14-Feb	Tu	SDT		HW: making Earley parser optimal (hard)
		15-Feb	We			
	10	16-Feb	Th	compilation: re		
		17-Feb	Fr			

Project assignment timing and logistics

- assigned Tuesday morning
- due Monday night (11:59pm)

- four free late day per semester
- see more details on the course web page

- HWs are individual
- Projects are in groups of three
- Collaboration via bitbucket.org (mercurial)

Administrativia

Newsgroup

- Piazza.com

Waitlist

- Talk to Michael-David Sasson in CS Office

Accounts

- Pick up accounts in Thursday discussion

Academic (Dis)honesty

Read the policy at:

- http://www.eecs.berkeley.edu/Policies/acad.dis.shtml

We'll be using a state-of-the art plagiarism detector.

- Before you ask: yes, it works very well.

You are allowed to discuss the assignment

- but you must acknowledge (and describe) help in your submission.

Conclusion

How is this PL/compiler class different?

Not intended for future compiler engineers

- there are few among our students

... but for software developers

- raise your hand if you plan to be one

But why does a developer or hacker need a PL class?

- we'll take a stab at this question today

Why a software engineer/developer needs PL

New languages will keep coming

- Understand them, choose the right one.

Write code that writes code

- Be the wizard, not the typist.

Develop your own language.

- 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

- to predict the future, let's examine the history…

Summary. Take home points.

What is a language?

Why languages help write software.

How does Prolog solve the puzzle?

Examples of small languages and their abstractions.

History of abstractions in programming languages.

An optional exercise

List three new languages, or major features added to established major languages, that have appeared in the last seven years. For each language, answer with one sentence these questions. (Use your own critical thinking; do not copy text from the Web.)

- *Why did the languages appear? Or, why have these features been added?* Often, a new language is motivated by *technical* problems faced by programmers. Sometimes the motivation for a new feature is *cultural*, related to, say, the educational background of programmers in a given language.

- *Who are the intended users of this language/feature?* Are these guru programmers, beginners, end-users (non-programmers)?

- *Show a code fragment that you find particularly cool.* The fragment should exploit the new features to produce highly readable and concise code.

Links that may help you start your exploration of the programming language landscape:

- http://lambda-the-ultimate.org/
- http://bit.ly/ddH47v
- http://www.google.com

Side notes

Where will languages go from here?

Another trend is to detect more bugs in programs when the program is compiled or run

- with stricter type checking
- with tools that look for bugs in the program or in its execution