



Quantifying Program Complexity and Comprehension

Michael Hansen, Andrew Lumsdaine, Rob Goldstone, Raquel Hill, Chen Yu

Dissertation Proposal

Indiana University, November 22 2013

Big Question

- How do we quantify the psychological or **cognitive complexity** of a program?

Motivation

- Explicit psychological theory of programming
- Automated identification of error-prone or potentially confusing code
- More objective design decisions for tools, programs, libraries, and languages
- Constrain code generators to produce less complex programs

Medium-sized Questions

- What is cognitive complexity in the context of programming?
- Which aspects of a program/programmer should affect this cognitive complexity?
- How might we quantify a program's cognitive complexity?
- Knowing a program's cognitive complexity, what could we predict?

Cognitive Complexity in the Context of Programming

Software complexity is "a measure of resources expended by a system [human or other] while interacting with a piece of software to perform a given task."

– Basili, 1980

One feature which all of these [theoretical] approaches have in common is that they begin with certain characteristics of the software and attempt to determine what effect they might have on the difficulty of the various programmer tasks.

A more useful approach would be first to analyze the processes involved in programmer tasks, as well as the parameters which govern the effort involved in those processes. From this point one can deduce, or at least make informed guesses, about which code characteristics will affect those parameters.

– Cant et. al, 1995

Of Models and Metrics

Cognitive complexity is...

- **Function of source code** (complexity metrics)
 - $\vec{c} = f(\text{code})$
 - $c_i > \text{threshold}_i$ is bad
- **Poor support for user activities** (Cognitive Dimensions of Notation)
 - Resistance to change, hidden dependencies, etc.
 - Programming languages are used to try out ideas
- **Unfamiliar schemas/implicit rule violations** (Détienne/Soloway)
 - Explains novice/expert differences
 - Don't X , IF/THEN rules
- **Properties of a cognitive model trace** (Cognitive Complexity Metric, Mr. Bits)
 - Cognitive resource constraints + effects of notation
 - Task time, eye movement metrics, contents of memory, etc.

Thesis Contributions

1. Thorough review of relevant literature in software complexity and the psychology of programming
2. Analysis of code/cognitive/demographic factors affecting programmer output predictions
3. Methodology and Python library for analyzing programmers' responses and eye movements
4. Analysis of collected eye movement data
5. Design, prototype, and evaluation of quantitative process model (output prediction task)

Presentation Overview

1. Measuring Software Complexity ([link](#))

- Kinds of complexity

2. Psychology of Programming ([link](#))

- Cognitive models of program comprehension

3. Experiments ([link](#))

- Aspects of code/programmer that affect comprehension

4. Modeling: Mr. Bits ([link](#))

- Quantifying resource expenditure

5. Conclusion and Research Timeline ([link](#))

- Finish by Spring 2015 at the latest

1. Measuring Software Complexity

- **Completed work**
 - Literature review
 - Complexity vs. reuse experiment
- **Proposed work**
 - Cohesive write-up
 - Readability vs. complexity (Buse)

Kinds of Software Complexity

- **Problem/computational complexity**
 - Complexity of underlying problem or domain
 - Usually considered fixed
- Representational complexity
- Cognitive/psychological complexity

Computational Complexity

- Bounds on computing resources as a function of input size
- $\mathcal{O}(c) < \mathcal{O}(n) < \mathcal{O}(n^c) < \mathcal{O}(c^n)$

Kolmogorov Complexity

- Computational resources needed to specify an object
- Size of smallest program in language \mathcal{L}
- Not computable in the general case

Kinds of Software Complexity

- Problem/computational complexity
- **Representational complexity**
 - Physical form of the program
 - Language, formatting, naming, etc.
 - Problem representation
- Cognitive/psychological complexity

Source Code Metrics

- Syntactic - Size/Spatial/Graph/Counter-Factual
 - Lines of Code
 - Function Complexity
 - Inheritance Depth
 - Minimum Description Length
- Readability
 - Line length
 - Number of identifiers/identifier length
 - Indentation/blank lines
- Concepts and beacons (stack, queue, etc.)
 - Formal Concept Analysis (lattice)
 - Concept Identification (Biggerstaff)

Weyuker's Properties (1988)

- Proposed properties of syntactic software complexity measures
- $|P|$ is the complexity of program P

Property	Description
$(\exists P, Q)(P \neq Q)$	Not all programs should have the same complexity
$(\forall c)(\{P \mid P = c\} \text{ is finite})$	The set of programs whose complexity is c is finite
$(\exists P, Q)(P = Q \text{ and } P \neq Q)$	Some programs share the same complexity
$(\exists P, Q)(P \equiv Q \text{ and } P \neq Q)$	Functional equivalence does not imply complexity equivalence
$(\forall P, Q)(P \leq P; Q \text{ and } Q \leq P; Q)$	Concatenation cannot decrease complexity
$(\exists P, Q, R)(P = Q \text{ and } P; R \neq Q; R)$	Context matters for complexity after concatenation
$(\exists P)(P \neq \text{permute}(P))$	The order of statements matters
$(\forall P)(P = \text{rename}(P))$	Identifier and operator names do not matter
$(\exists P, Q)(P + Q < P; Q)$	Concatenated programs may be more complex than the sum of their parts

Kinds of Software Complexity

- Problem/computational complexity
- Representational complexity
- **Cognitive/psychological complexity**
 - Influenced by problem, representational complexity
 - Function of programmer experience, mental resource constraints
 - Task dependent: reuse vs. debugging vs. modification

Qualitative Models

- Integrated Metamodel (von Mayrhauser, 1995)
 - Program/situation models + top-down planning
- Cognitive Dimensions of Notation (Blackwell and Green, 1995)
 - Programming languages are used to try out ideas
 - Hidden dependencies, viscosity, consistency, ...
- Rules of Discourse (Soloway and Ehrlich, 1984)
 - Unwritten rules internalized by experts
 - Expectations that drive understanding process

Kinds of Software Complexity

- Problem/computational complexity
- Representational complexity
- **Cognitive/psychological complexity**
 - Influenced by problem, representational complexity
 - Function of programmer experience, mental resource constraints
 - Task dependent: reuse vs. debugging vs. modification

Quantitative Models

- Cognitive Weights (Chhabra, 2011; Shao et. al, 2003)
 - Assign weights to syntactic & semantic elements
 - Complexity = $f(weights)$
- Cognitive Complexity Metric (Cant et. al, 1995)
 - "Process" model based on chunking & tracing
 - Terms for chunk size, control structures, boolean expressions, etc.
 - Complexity = $f(chunking) + g(tracing)$
- **Mr. Bits**
 - Embodied process model based on eye movements, memory, spatial reasoning, inference
 - Task is to predict printed output
 - Complexity = time spent, steps taken, representation, etc.

Readability vs. Complexity

- Readability is "accidental" while complexity is "essential"
 - Problem/computational complexity
- Readability is local, line-by-line (Buse, 2010)
 - Number of identifiers
 - Line length
 - Indentation
- Software Readability Ease Score (SRES)
 - Like Flesch score (FRES)
 - Tokens = syllables, statements = words, units = sentences

2. Psychology of Programming

- **Completed work**
 - Literature review
 - Onward! workshop paper (cognitive architectures)
- **Proposed work**
 - Review literature on text understanding models (Kintsch, 1978)
 - Consider recent eye-tracking studies of programming

Many claims are made for the efficacy and utility of new approaches to software engineering - structured methodologies, new programming paradigms, new tools, and so on. Evidence to support such claims is thin and such evidence, as there is, is largely anecdotal. Of proper scientific evidence there is remarkably little.

Furthermore, such as there is can be described as "black box", that is, it demonstrates a correlation between the use of certain technique and an improvement in some aspect of the development. It does not demonstrate how the technique achieves the observed effect.

— Software Design - Cognitive Aspects (Détienne, 2001)

Periods of Research

Early: 1960-1980

- Importing of experimental techniques to CS
- Correlations between task performance and PL/human factors
- Novice participants on toy programs
- Contradictory and confusing results

Later: 1980-Present

- Use of cognitive models to explain internal processes
- Verbal reports, real-time code changes, gaze patterns, etc.
- Experienced/professional participants on real-world programs
- Models are largely qualitative

Early Study Example

- Effect of variable naming on code understanding
 - No effect for simple programs, positive effect for complex programs
 - Experienced programmers recognize *schemas* (Soloway and Ehrlich, 1984)

Important Factors

- **Knowledge**

- Experienced programmers represent at multiple levels of abstraction: syntactic, semantic, and schematic
- Conventions and common programming plans allow experts to quickly infer intent and avoid unnecessary details

- **Strategies**

- Experienced programmers use more design strategies (top-down, bottomup, breadth-first, depth-first)
- Current strategy is chosen based on factors like familiarity, problem domain, and available language features

- **Task**

- Current task or goal will change which kinds of program knowledge and reading strategies are advantageous
- Experienced programmers read and remember code differently depending on whether they intend to edit, debug, or reuse it

- **Environment**

- Programmers use their tools to off-load mental work and to build up representations of the current problem state
- The benefits of specific tools, such as program visualization, also depend on programming expertise

Models of Text Understanding

- **Structural**

- Understanding = constructing a network of relations
- Top-down identification of structural schemas
- Bottom-up construction of propositional network
- Stages
 1. Morpho-syntactic decoding
 2. Sentence parsing and proposition construction
 3. Connecting propositions (micro and macro structure)

- **Mental Model**

- Understanding = constructing a representation of the situation
- Levels of representation
 1. Surface representation
 2. Propositional
 3. Situational model (optional)
- Situational model
 - Content-rich (vs. structural)
 - Invocation of knowledge schemas (including domain knowledge)

Cognitive Models of Programming

[A cognitive model] seeks to explain basic mental processes and their interactions; processes such as perceiving, learning, remembering, problem solving, and decision making.

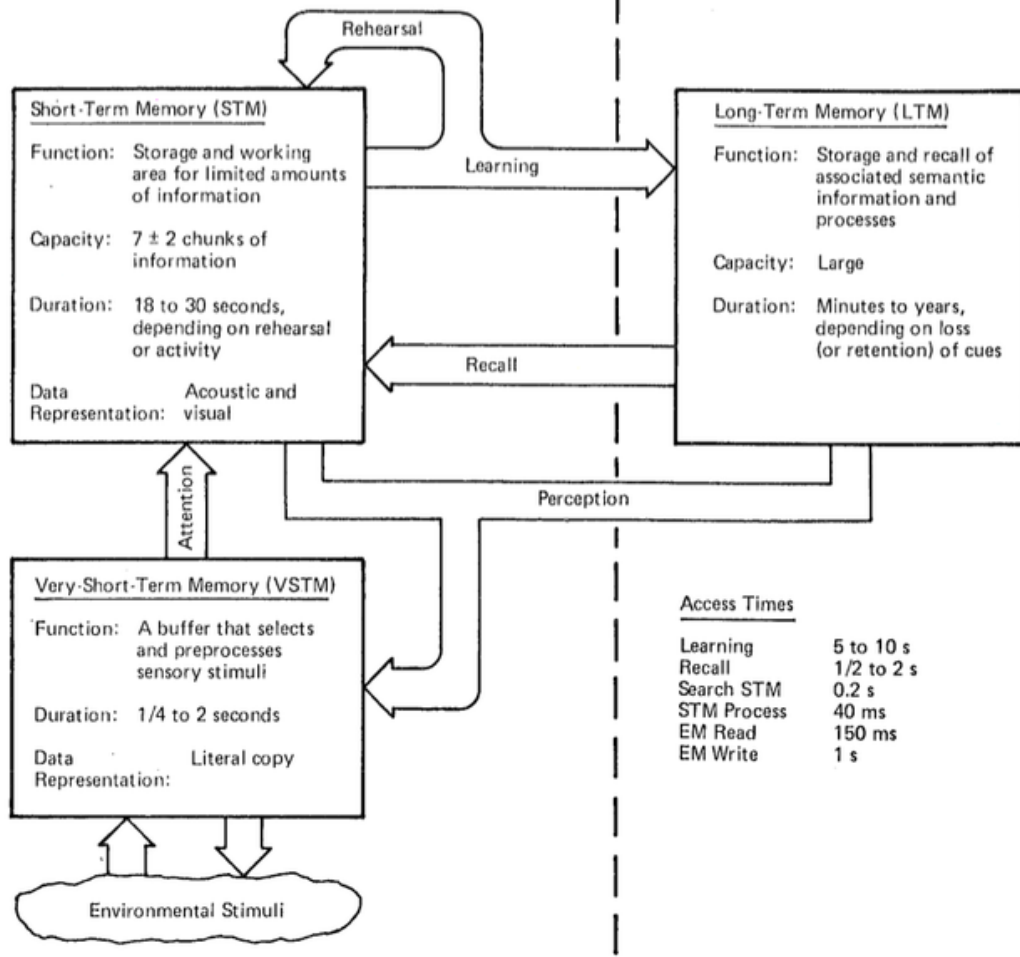
– Busemeyer and Diederich, 2010

Experimental Approaches

- **Comprehension tests** - read code and answer questions
 - Questions about control flow are easier than data flow
- **Code recall** - reproduce code after reading
 - Experts more likely to recall prototypical schema values (*i* instead of *j* for iterator)
- **Debugging** - look for errors and fix
 - Review time linked to success
- **Completion** - fill in the blank
 - Experts do better than novices when rules of discourse are not violated
- **Create** - write a program according to some spec
 - Experts are top-down for known problems, bottom-up otherwise

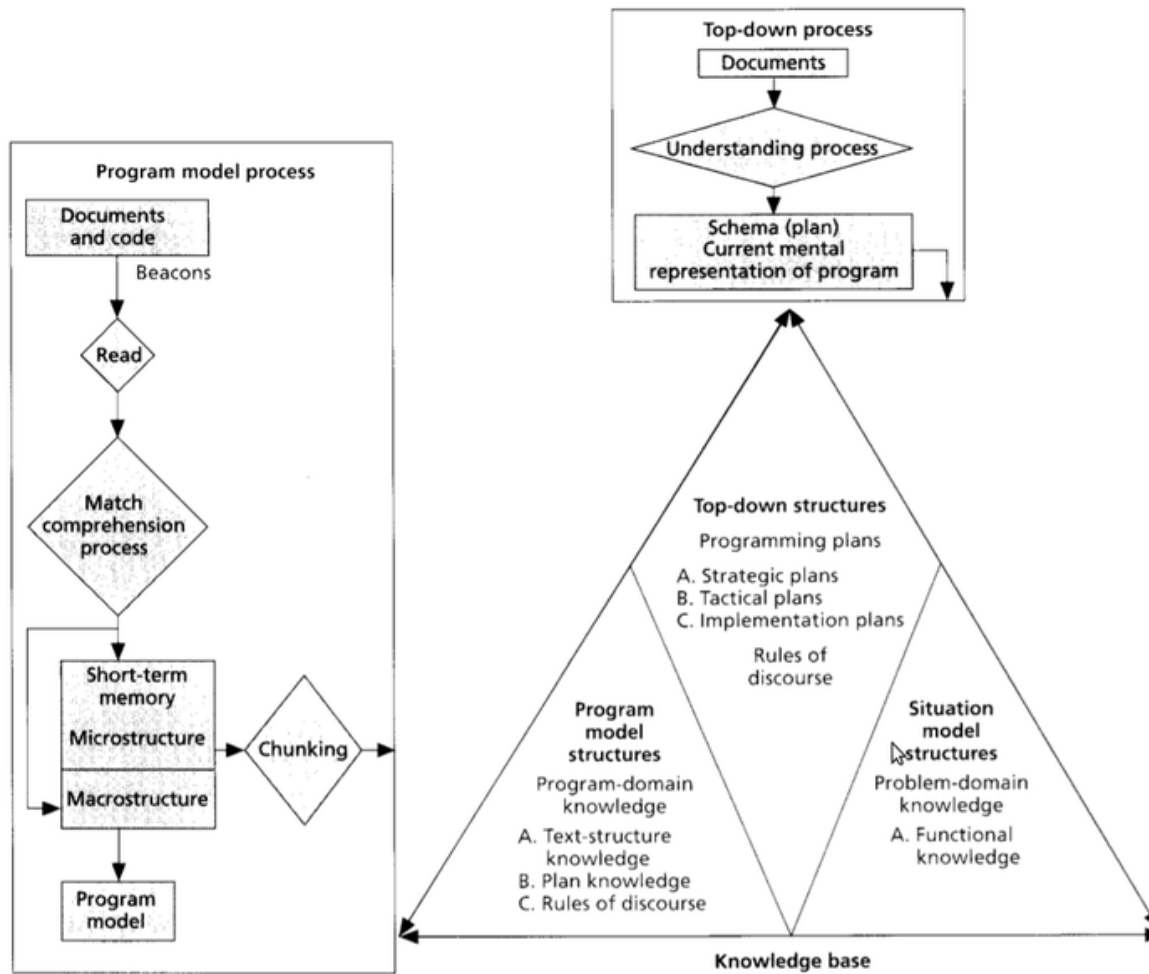
Tracz's Human Information Processing System (HIPS)

CONSCIOUSNESS



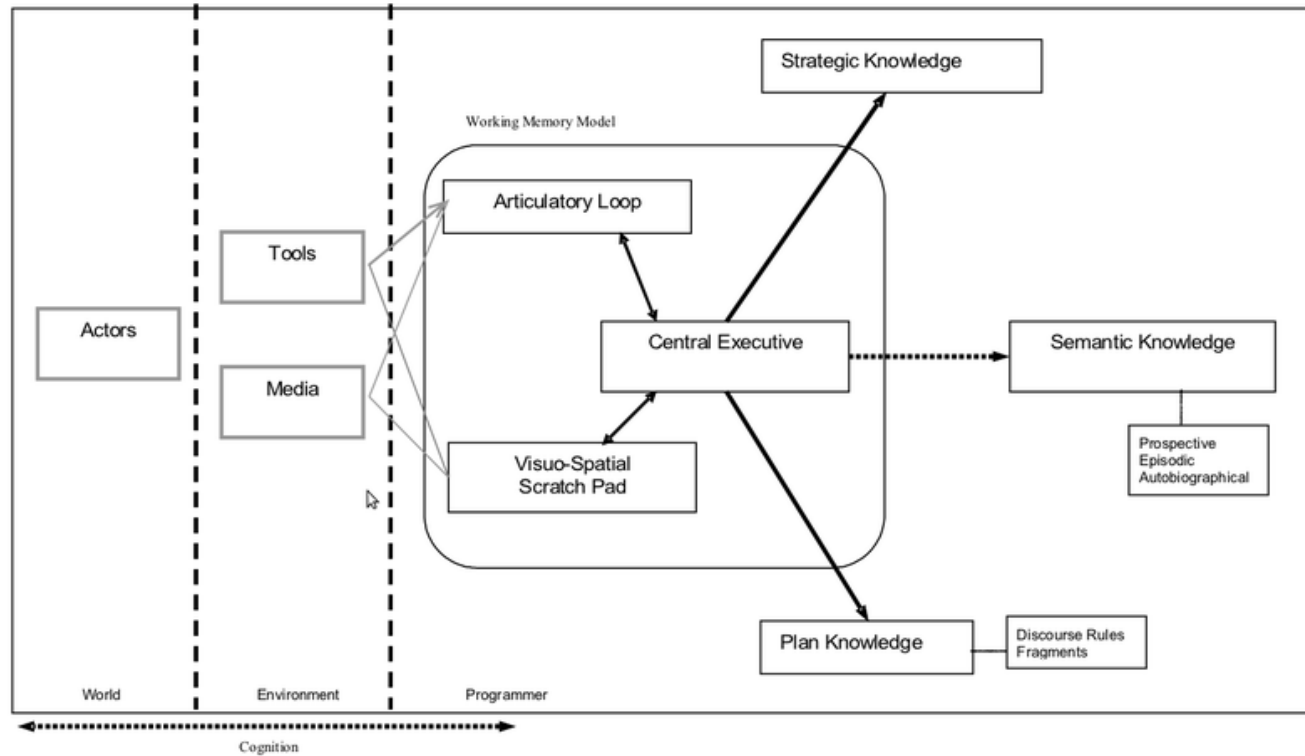
● Image from Tracz, 1979

von Mayrhauser's Integrated Metamodel



● Image from von Mayrhauser and Vans, 1995

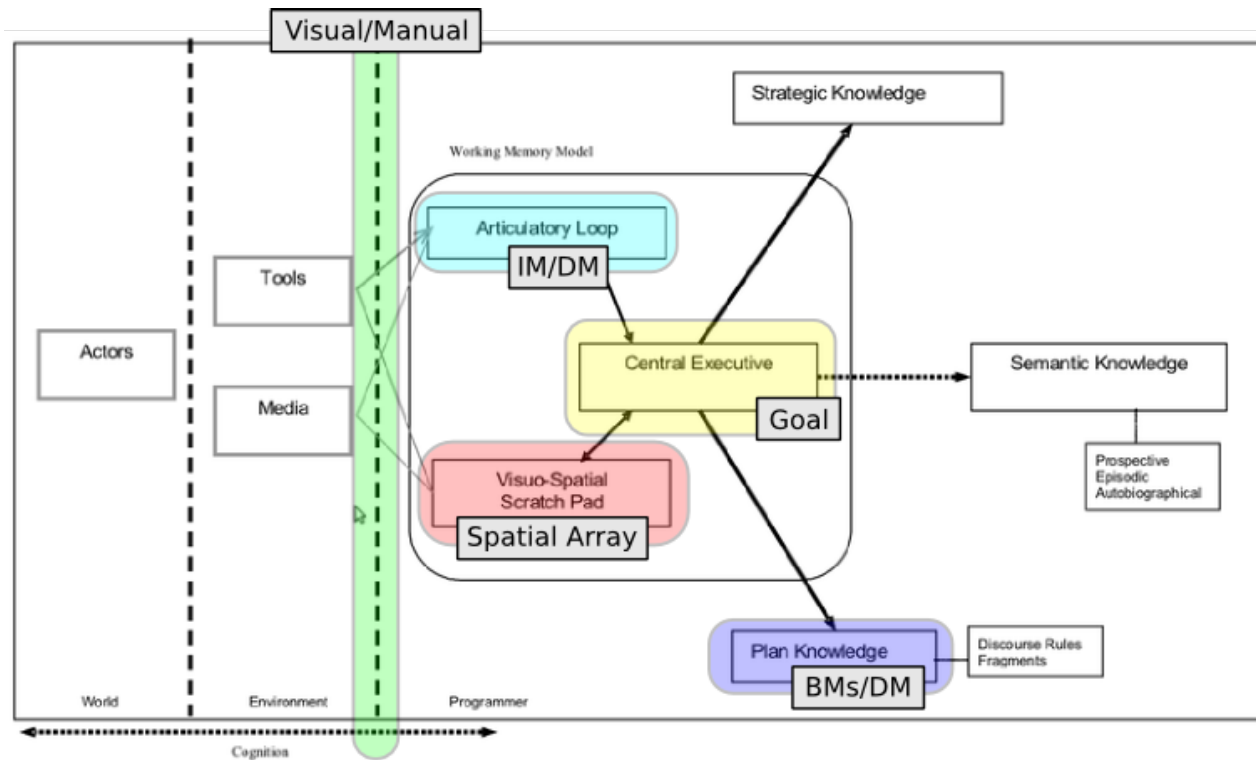
Douce's Stores Model of Code Cognition



● Image from Douce, 2008

Douce's Stores Model of Code Cognition + Mr. Bits

- IM = imaginal buffer, DM = declarative memory (ACT-R)
- BMs = behavior models (productions + state)

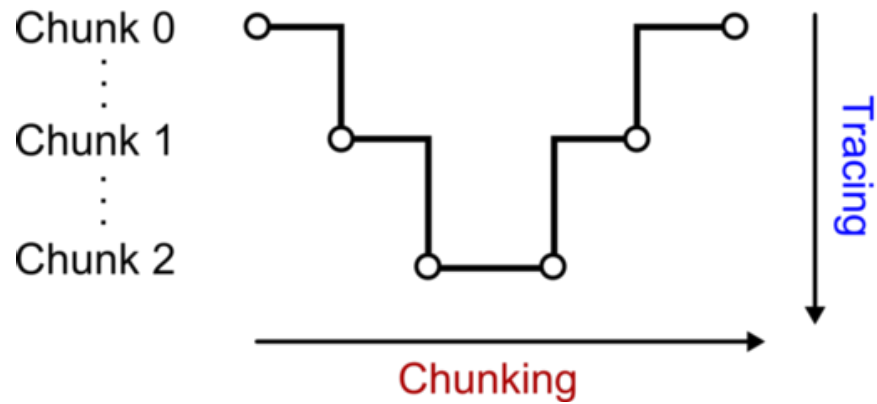


- Base image from Douce, 2008

Cant's Cognitive Complexity Metric (CCM)

- Immediate chunk complexity (R_i)
- Sub-chunk complexity (C_j)
- Tracing difficulty (T_j)

$$C_i = \boxed{R_i} + \sum_{j \in N} C_j + \sum_{j \in N} \boxed{T_j}$$



CCM Terms (Chunking)

- $R = R_F(R_S + R_C + R_E + R_R + R_V + R_D)$

Term	Description
R_F	Speed of recall or review (familiarity)
R_S	Chunk size
R_C	Type of control structure in which chunk is embedded
R_E	Difficulty of understanding complex Boolean or other expressions
R_R	Recognizability of chunk
R_V	Effects of visual structure
R_D	Disruptions caused by dependencies

CCM Terms (Tracing)

- $T = T_F(T_L + T_A + T_S + T_C)$

Term	Description
T_F	Dependency familiarity
T_L	Localization
T_A	Ambiguity
T_S	Spatial distance
T_C	Level of cueing

Mr. Chips (Legge, 2002)

- Ideal observer model of text reading based on EZ-Reader
- Single, long line of text
- Make the saccade that minimizes uncertainty

Retina

- Rectangular, discrete fovea + para-fovea
- Only characters/whitespace distinguishable in para-fovea

Eye Movement Model

- Gaussian variability in saccade length
- Model is aware of noise

Lexicon

- Words and relative frequencies

Information Available to Mr. Chips

Visual

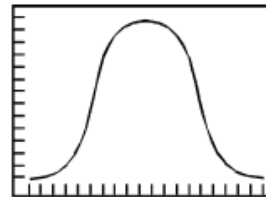
*_*_at_is_abo**_*

"Normal" Retina

*_*_at_*_*_abo**_*

Central Scotoma Retina

Eye Movement Accuracy



Lexical

Word	Freq(%)
a	3.7
about	.37
above	.068
at	.963
•	•
is	1.2
•	•
yes	.072
you	2.8

- Image from Legge et. al, 2002

A Model of Human Behavior?

...Mr. Chips is not proposed as a model of human behavior, and is not falsifiable by human reading data. Its value in studying human reading should be judged on its claim to optimality (see Chips97), the reasonableness of its assumed informational constraints, and the insights it generates into human reading.

– Legge et. al, 2002

- **Mr. Bits**

- Model of human programmer?
- Model of task with resource constraints
 - Sensor (eye), DM, manual

3a. Experiment 1: Output Prediction

- **Completed work**
 - Data collection from Mechanical Turk and Bloomington (162 participants)
 - arXiv paper with response data analysis
- **Proposed work**
 - Journal article with additional complexity/performance metrics

The eyeCode Experiment

Research Questions

1. How are programmers affected by programs that violate their expectations, and does this vary with expertise?
2. How are programmers influenced by physical characteristics of notation, and does this vary with expertise?
3. Can code complexity metrics and programmer demographics be used to predict task performance?

Task

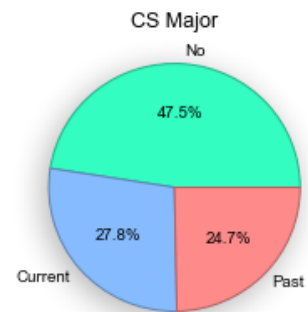
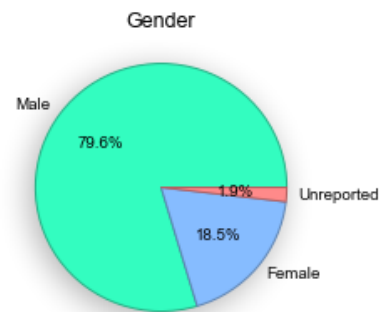
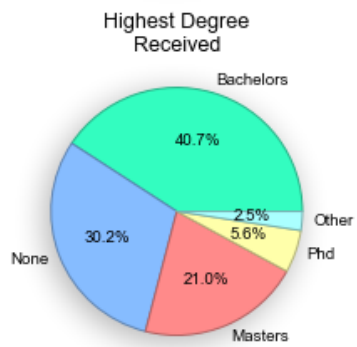
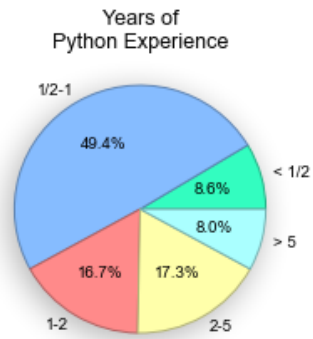
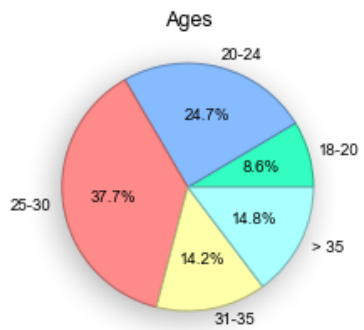
- Predict printed output of 10 short Python programs
- 2-3 versions of 10 programs, randomly assigned
- Pre/post surveys
- No feedback, syntax highlighting

Participants

- 162 total participants
 - 29 Bloomington (\$10)
 - 130 Mechanical Turk (\$0.75)
 - 3 E-mail
- 1,602 trials
 - 18 trials discarded

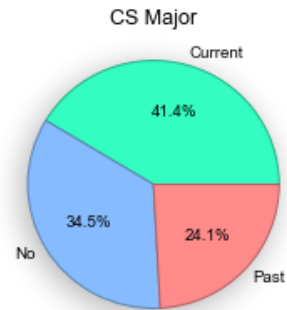
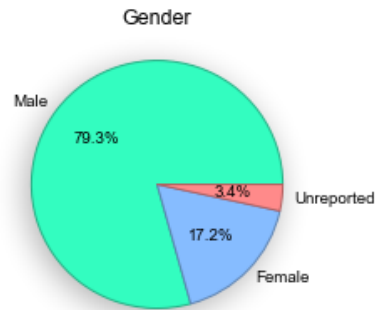
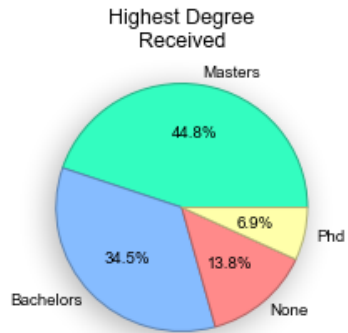
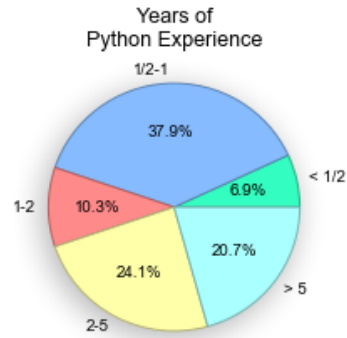
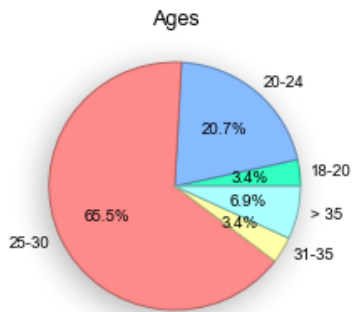
The screenshot shows the Amazon Mechanical Turk interface. At the top, there is a navigation bar with the Amazon Mechanical Turk logo and the text "Artificial Intelligence". To the right of the logo are three buttons: "Your Account", "HITS", and "Qualifications". Below the navigation bar, there is a search bar with the text "Find HITS containing" and a search button. To the right of the search bar, there are links for "All HITS", "HITS Available To You", and "HITS Assign". Below the search bar, there is a section titled "All HITS" with the text "1-10 of 2960 Results". To the right of this text is a dropdown menu for "Sort by:" set to "HITS Available (most first)" and a "GO" button. To the right of the dropdown menu are links for "Show all details" and "Hide all details". Below the search bar, there are three HIT cards. The first card is titled "Copy Text from Business Card" and has a requester of "Oscar Smith", a HIT Expiration Date of "Feb 15, 2013 (1 hour 59)", and a Time Allotted of "10 minutes". The second card is titled "Inv. B. 2" and has a requester of "rohzi0d", a HIT Expiration Date of "Mar 16, 2013 (4 weeks)", and a Time Allotted of "48 minutes". The third card is titled "Tag actors from a piece of adult content" and has a requester of "NetMSi", a HIT Expiration Date of "Feb 18, 2013 (2 days 23)", and a Time Allotted of "7 minutes".

Demographics (All Participants)



Demographics (Bloomington Participants)

- Younger participants, more experienced programmers



Home Screen

- Program order is randomized

eyeCode [hacking for science]



Tell me what **YOU** think the programs below will output.
Be quick, but try not to make mistakes!

1. [Done] `appalling.py`
2. [Done] `weirdo.py`
3. [Start] `brawny.py`
4. [Start] `elder.py`
5. [Start] `couch.py`
6. [Start] `rooster.py`
7. [Start] `prophetic.py`
8. [Start] `cuddle.py`
9. [Start] `hermit.py`
10. [Start] `hotshot.py`

Trial Screen

- Images instead of text, no feedback

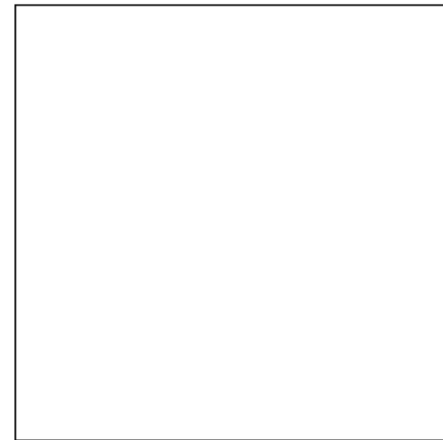
eyeCode [hacking for science]

```
x = [2, 8, 7, 9, -5, 0, 2]
x_between = []
for x_i in x:
    if (2 < x_i) and (x_i < 10):
        x_between.append(x_i)
print x_between

y = [1, -3, 10, 0, 8, 9, 1]
y_between = []
for y_i in y:
    if (-2 < y_i) and (y_i < 9):
        y_between.append(y_i)
print y_between

xy_common = []
for x_i in x:
    if x_i in y:
        xy_common.append(x_i)
print xy_common
```

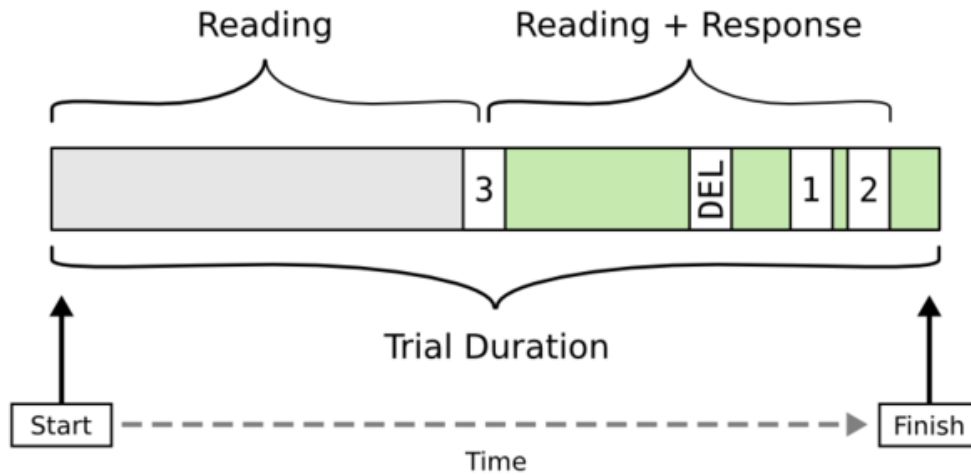
What will this program output?



Continue

Anatomy of a Trial

```
print "1" + "2"
```



- Response proportion ≈ 0.5
- Keystroke coefficient = $4/2 = 2$
- Keystroke count = 4
 - True output characters = 2
- Response corrections = 1
- Grade = 10 (perfect)

Programs (1/2)

10 categories, 2-3 versions each (25 total)

- between - filter two lists, intersection
 - functions - between/common in functions (24 lines)
 - inline - no functions (19 lines)
- counting - simple for loop with bug
 - nospace - no blank lines in loop body (3 lines)
 - twospaces - 2 blank lines in loop body (5 lines)
- funcall - simple function call with different values
 - nospace - calls on 1 line, no spaces (4 lines)
 - space - calls on 1 line, spaced out (4 lines)
 - var s - calls on 3 lines, different vars (7 lines)
- overload - overloaded + operator (number strings)
 - multmixed - numeric *, string + (11 lines)
 - plusmixed - numeric +, string + (11 lines)
 - strings - string + (11 lines)
- partition - partition list of numbers
 - balanced - odd number of items (5 lines)
 - unbalanced - even number of items (5 lines)
 - unbalanced_pivot - even number of items, pivot var (6 lines)

Programs (2/2)

10 categories, 2-3 versions each (25 total)

- `initvar` - summation and factorial
 - `bothbad` - bug in both (9 lines)
 - `good` - no bugs (9 lines)
 - `onebad` - bug in summation (9 lines)
- `order` - 3 simple functions called
 - `inorder` - call order = definition order (14 lines)
 - `shuffled` - call order \neq definition order (14 lines)
- `rectangle` - compute area of 2 rectangles
 - `basic` - `x,y,w,h` in separate vars, `area()` in function (18 lines)
 - `class` - `x,y,w,h,area()` in class (21 lines)
 - `tuples` - `x,y,w,h` in tuples, `area()` in function (14 lines)
- `scope` - function calls with no effect
 - `diffname` - local/global var have same name (12 lines)
 - `samename` - local/global var have different name (12 lines)
- `whitespace` - simple linear equations
 - `linedup` - code is aligned on operators (14 lines)
 - `zigzag` - code is not aligned (14 lines)

Code Complexity Metrics

- `code_lines` - number of lines in the program (includes blank lines)
 - Correlated with CC (0.46) and E (0.78)
- `cyclo_comp` - McCabe's Cyclomatic Complexity
 - $CC = E - N + 2P$
 - E = edges
 - N = nodes
 - P = connected components
 - Upper bound for branch coverage
 - Lower bound for paths
- `hal_effort` - Halstead's Effort
 - n_1 = unique operators, n_2 = unique operand
 - N_1 = total operators, N_2 = total operands
 - Program Length: $N = N_1 + N_2$
 - Program Vocabulary: $n = n_1 + n_2$
 - Volume: $V = N \log_2 n$
 - Difficulty: $D = \frac{n_1}{2} \times \frac{N_2}{n_2}$
 - Effort: $E = D \times V$

		code_chars	code_lines	cyclo_comp	hal_effort	output_chars	output_lines
base	version						
between	functions	496	24	7	94192	33	3
	inline	365	19	7	45596	33	3
counting	nospace	77	3	2	738	116	8
	twospaces	81	5	2	738	116	8
funcall	nospace	50	4	2	937	3	1
	space	54	4	2	937	3	1
	vars	72	7	2	1735	3	1
initvar	bothbad	103	9	3	3212	5	2
	good	103	9	3	3212	6	2
	onebad	103	9	3	2866	6	2
order	inorder	137	14	4	8372	6	1
	shuffled	137	14	4	8372	6	1

		code_chars	code_lines	cyclo_comp	hal_effort	output_chars	output_lines
base	version						
initvar	bothbad	103	9	3	3212	5	2
	good	103	9	3	3212	6	2
	onebad	103	9	3	2866	6	2
overload	multmixed	78	11	1	2340	9	3
	plumixed	78	11	1	3428	7	3
	strings	98	11	1	3428	21	3
partition	balanced	105	5	4	2896	26	4
	unbalanced	102	5	4	2382	19	3
	unbalanced_pivot	120	6	4	2707	19	3
rectangle	basic	293	18	2	18801	7	2
	class	421	21	5	43203	7	2
	tuples	277	14	2	15627	7	2
scope	diffname	144	12	3	2779	2	1
	samename	156	12	3	2413	2	1
whitespace	linedup	275	14	1	6480	13	3
	zigzag	259	14	1	6480	13	3

Performance Metrics

- **Grade**
 - A grade of 7 or higher (out of 10) is correct
 - More complex programs should result in a lower grade
- **Trial duration**
 - Time from start to finish (reading + responding)
 - More complex programs should take longer to read and respond to (higher duration)
- **Response proportion**
 - Time spent responding / trial duration
 - More complex programs should require more reading time up front (higher proportion)
- **Keystroke coefficient**
 - Number of actual keystrokes / required keystrokes
 - More complex programs should require more keystrokes due to mistakes/corrections (higher coefficient)
- **Response Corrections**
 - Number of decreases in response size
 - More complex programs should require more corrections (higher number)

Grades

- 0 to 10 (perfect)
- ≥ 7 correct modulo formatting

```
print "1" + "2"  
print 4 * 3
```

True Output

```
12  
12
```

Common Error (4)

```
3  
12
```

Correct (7)

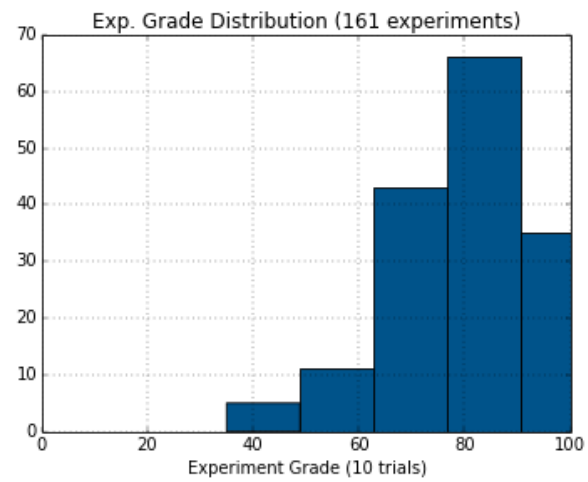
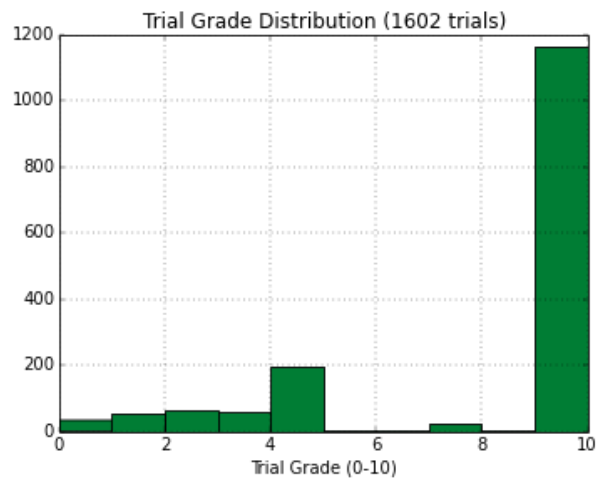
```
"12", 12
```

Incorrect (0)

```
barney
```

Grades

- 0 to 10 (perfect)
- ≥ 7 correct modulo formatting



- Median trial grade = 10
- Median experiment grade = 81

scope - samename

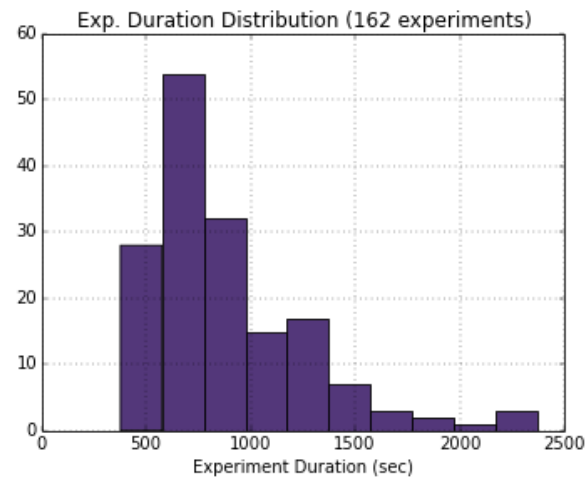
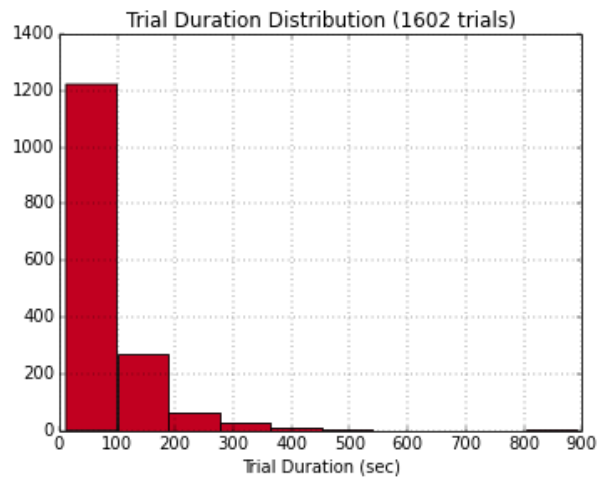
```
def add_1(added):  
    added = added + 1  
  
def twice(added):  
    added = added * 2  
  
added = 4  
add_1(added)  
twice(added)  
add_1(added)  
twice(added)  
print added
```

scope - diffname

```
def add_1(num):  
    num = num + 1  
  
def twice(num):  
    num = num * 2  
  
added = 4  
add_1(added)  
twice(added)  
add_1(added)  
twice(added)  
print added
```


Trial Duration

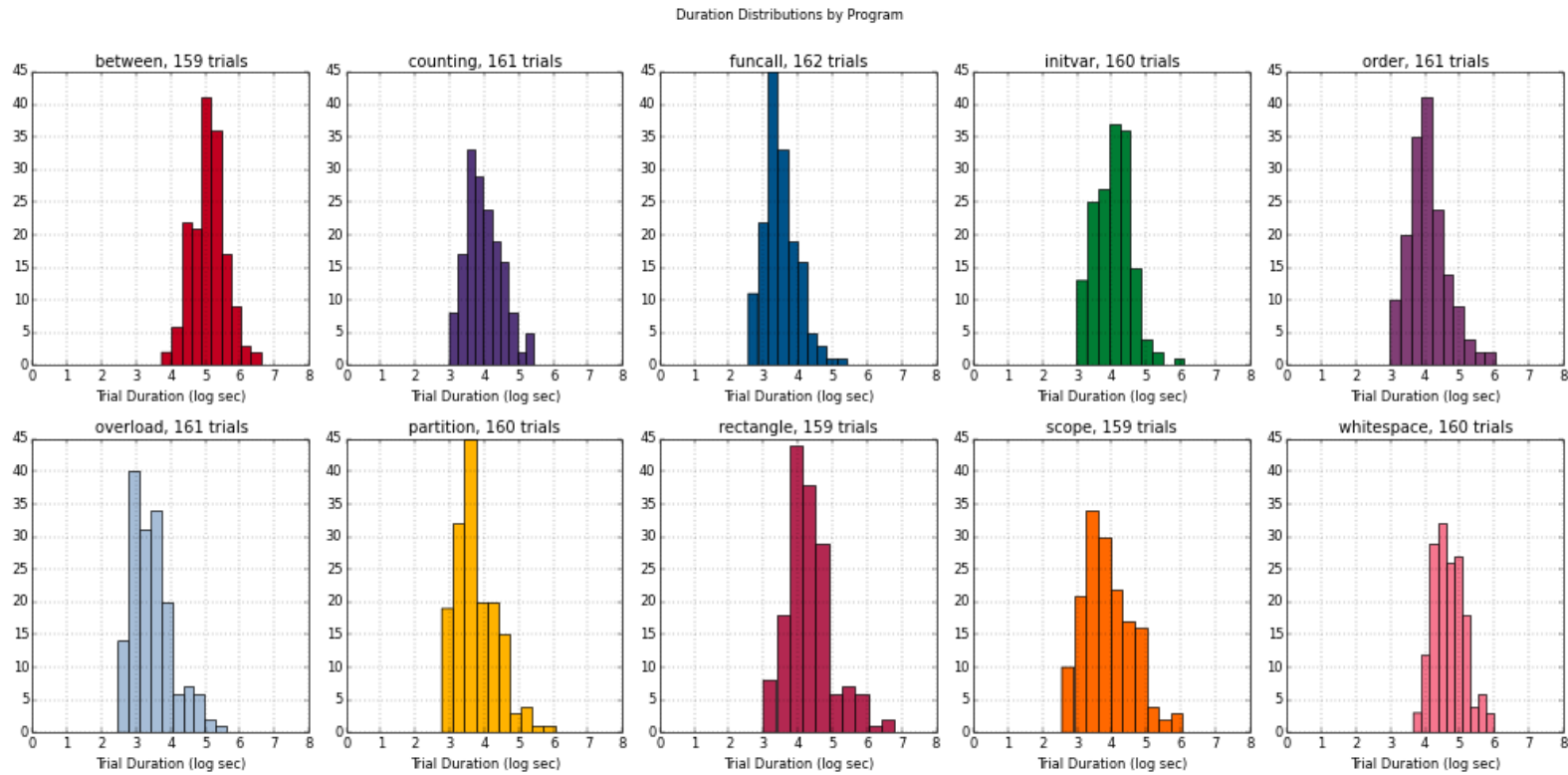
- 45 minutes for entire experiment
- No time limit on individual trials



- Median trial duration: 55 sec
- Median experiment duration: 773 sec (12.9 min)

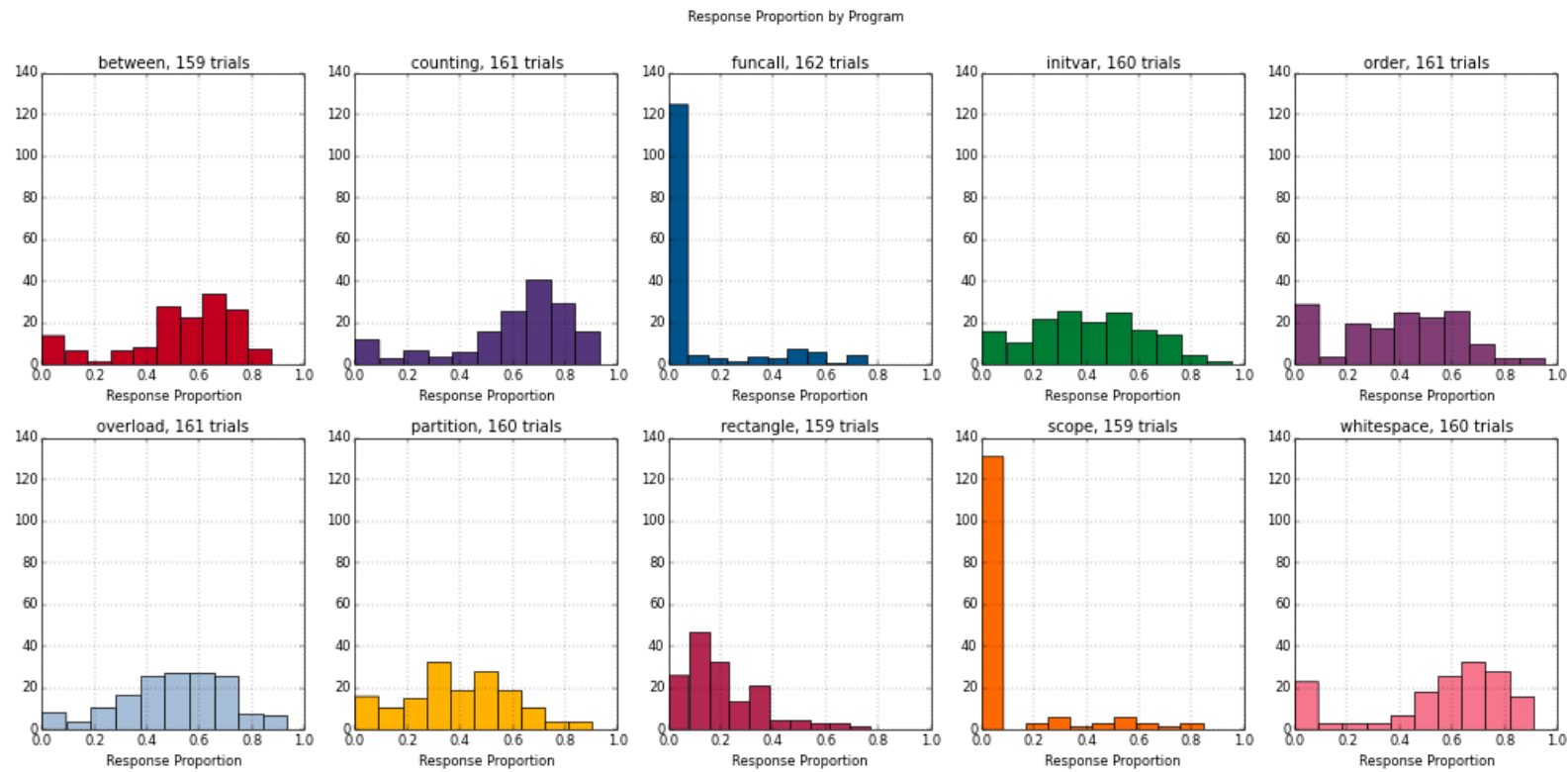
Duration Distributions by Program

- Log scale, strong positive correlation with lines of code (0.48)



Response Proportions by Program

- Time spent responding / trial time



between - functions

```
def between(numbers, low, high):
    winners = []
    for num in numbers:
        if (low < num) and (num < high):
            winners.append(num)
    return winners

def common(list1, list2):
    winners = []
    for item1 in list1:
        if item1 in list2:
            winners.append(item1)
    return winners

x = [2, 8, 7, 9, -5, 0, 2]
x_btwn = between(x, 2, 10)
print x_btwn

y = [1, -3, 10, 0, 8, 9, 1]
y_btwn = between(y, -2, 9)
print y_btwn

xy_common = common(x, y)
print xy_common
```

between - inline

```
x = [2, 8, 7, 9, -5, 0, 2]
x_between = []
for x_i in x:
    if (2 < x_i) and (x_i < 10):
        x_between.append(x_i)
print x_between

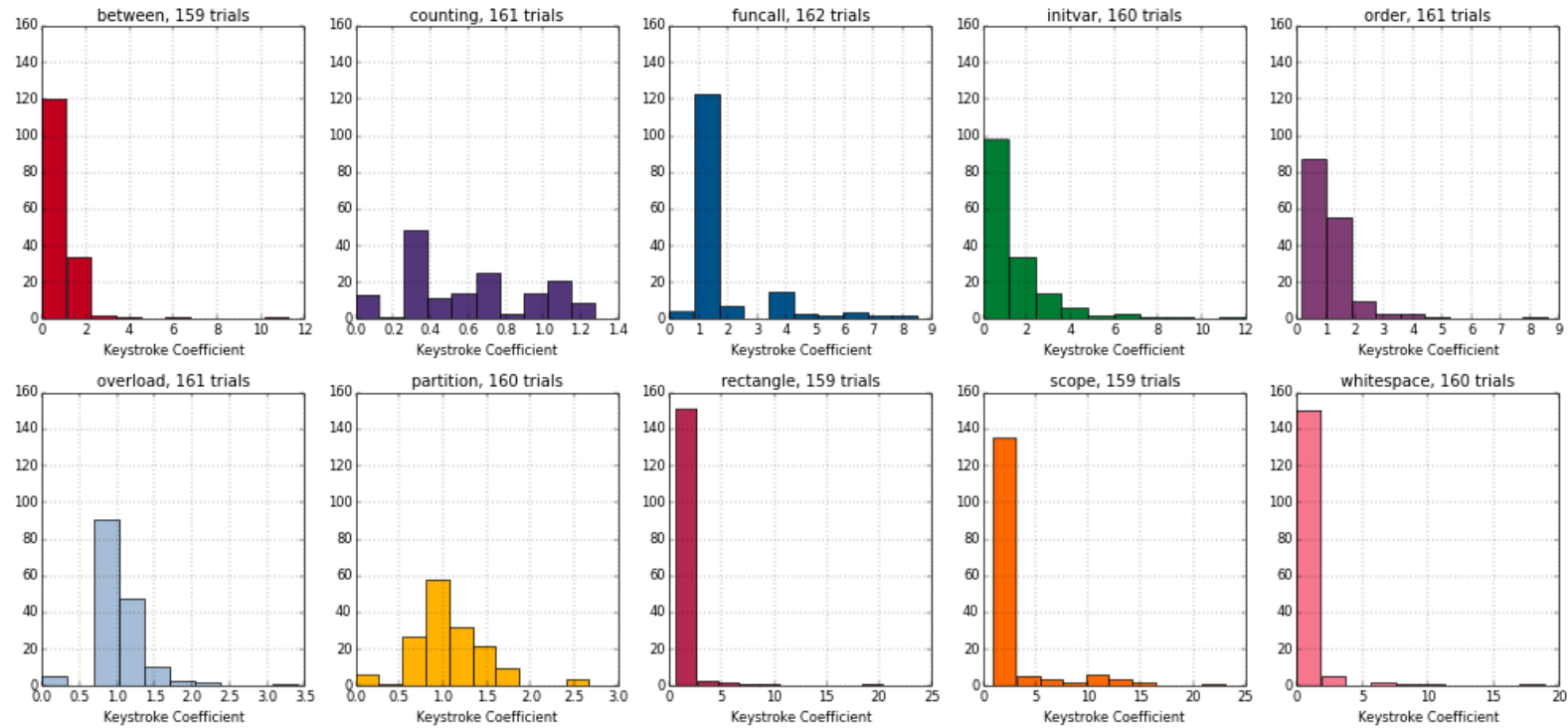
y = [1, -3, 10, 0, 8, 9, 1]
y_between = []
for y_i in y:
    if (-2 < y_i) and (y_i < 9):
        y_between.append(y_i)
print y_between

xy_common = []
for x_i in x:
    if x_i in y:
        xy_common.append(x_i)
print xy_common
```

Keystroke Coefficient

- Number of keystrokes / characters in true output
- > 1 is less efficient

Keystroke Coefficient by Program



counting - nospace

```
for i in [1, 2, 3, 4]:  
    print "The count is", i  
    print "Done counting"
```

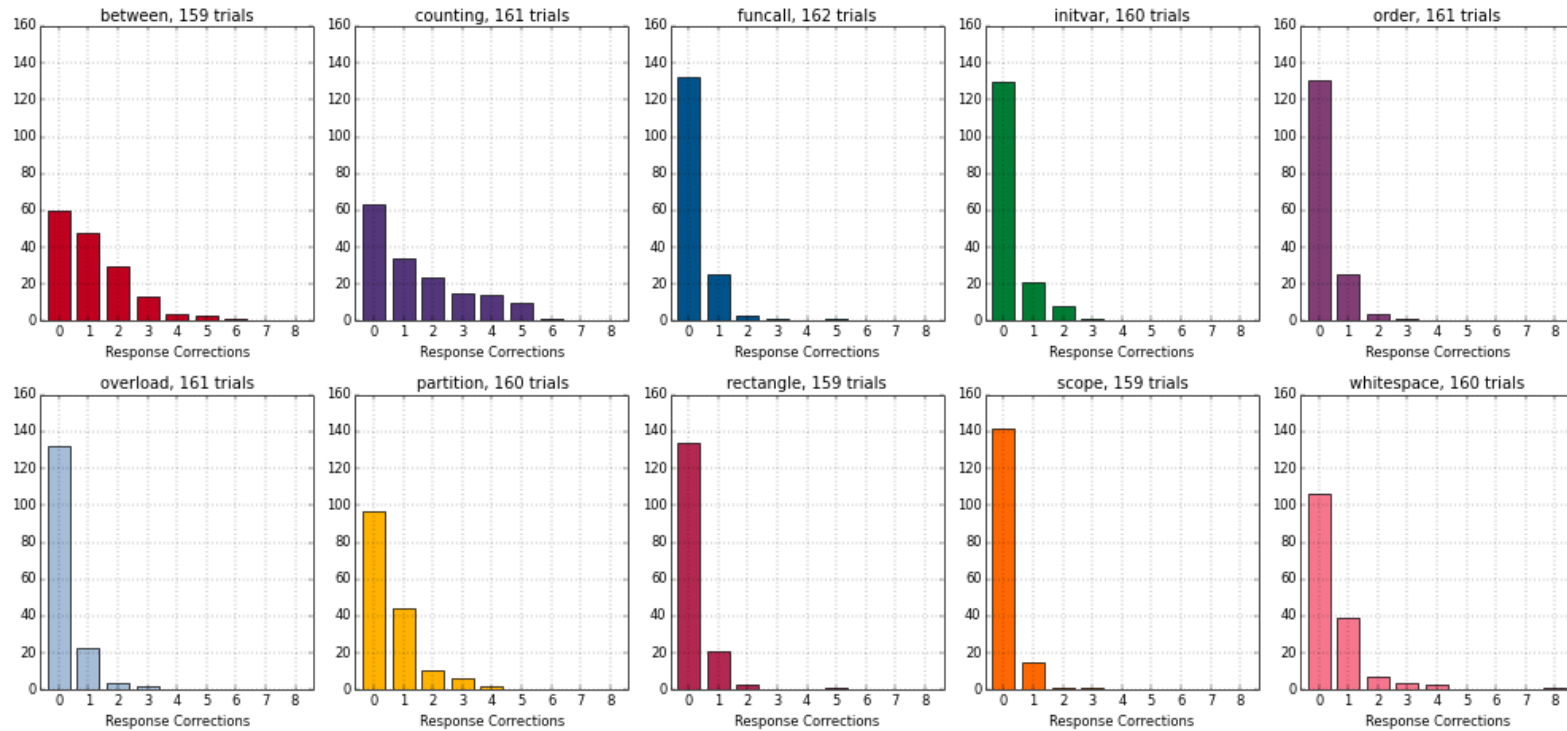
counting - twospaces

```
for i in [1, 2, 3, 4]:  
    print "The count is", i  
  
    print "Done counting"
```

Response Corrections

- Number of decreases in response size
- Higher number means more corrections

Response Corrections by Program



Results

1. How are programmers affected by programs that violate their expectations, and does this vary with expertise?
 - More response errors (scope, between)
 - Varies with expertise sometimes (scope - Python)
2. How are programmers influenced by physical characteristics of notation, and does this vary with expertise?
 - More response errors, longer trials (counting, overload)
 - No significant effect of expertise
3. Can code complexity metrics and programmer demographics be used to predict task performance?
 - Yes, significantly better than chance (binary metrics)
 - Cyclomatic complexity (↓) + years of Python experience (↑) best for correct grade
 - Code lines (↑) + years of programming experience (↓) best for trial duration

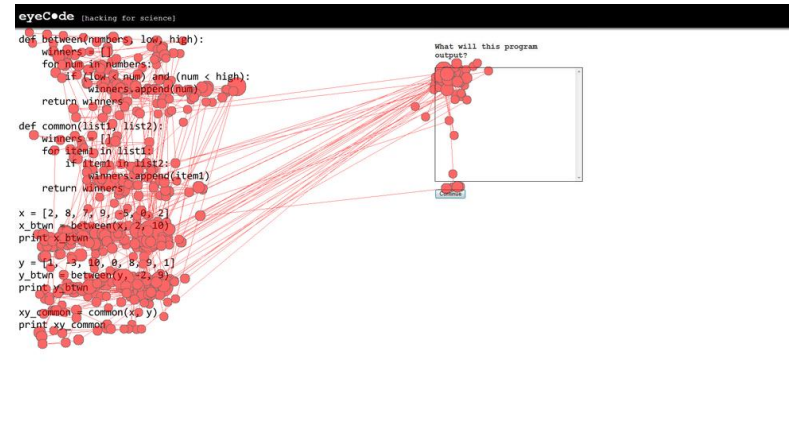
3b. Experiment 2: Eye Tracking

- **Completed work**
 - Data collection from Bloomington (29 participants, 5.5 hours of video)
 - Videos and preliminary analysis available via web
 - Koli Calling workshop paper with automated coding
 - Alpha version of eyeCode Python library
- **Proposed work**
 - Follow-up Koli Calling publications (automated coding, visualization - abstract rendering?)
 - Paper with fixation metric and scanpath comparisons
 - Release stable eyeCode library, data, and complete analyses

Eye-Tracking Hardware



- Tobii TX300 - 300Hz
- 23 in. screen, 1920x1080
- Free-standing (no chinrest)
- Tobii Studio 2.2



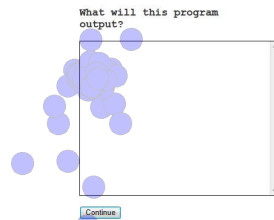
- Fixations from single trial
- between_functions program
- Radii proportional to duration

Data Processing and Correction

- Tobii Studio default fixation filter
- Fixations were manually correct by experiment (vertical shifts only)

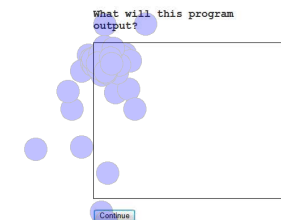
Uncorrected

```
def between(numbers, low, high):  
    winners = []  
    for num in numbers:  
        if (low < num) and (num < high):  
            winners.append(num)  
    return winners  
  
def common(list1, list2):  
    winners = []  
    for item1 in list1:  
        if item1 in list2:  
            winners.append(item1)  
    return winners  
  
x = [2, 8, 7, 9, -5, 0, 2]  
x_btwn = between(x, 2, 10)  
print x_btwn  
  
y = [1, -3, 10, 0, 8, 9, 1]  
y_btwn = between(y, -2, 9)  
print y_btwn  
  
xy_common = common(x, y)  
print xy_common
```



Corrected

```
def between(numbers, low, high):  
    winners = []  
    for num in numbers:  
        if (low < num) and (num < high):  
            winners.append(num)  
    return winners  
  
def common(list1, list2):  
    winners = []  
    for item1 in list1:  
        if item1 in list2:  
            winners.append(item1)  
    return winners  
  
x = [2, 8, 7, 9, -5, 0, 2]  
x_btwn = between(x, 2, 10)  
print x_btwn  
  
y = [1, -3, 10, 0, 8, 9, 1]  
y_btwn = between(y, -2, 9)  
print y_btwn  
  
xy_common = common(x, y)  
print xy_common
```



Line-based AOs

- Indentation is part of line AOI

```
def between(numbers, low, high):  
    winners = []  
    for num in numbers:  
        if (low < num) and (num < high):  
            winners.append(num)  
    return winners
```

```
def common(list1, list2):  
    winners = []  
    for item1 in list1:  
        if item1 in list2:  
            winners.append(item1)  
    return winners
```

```
x = [2, 8, 7, 9, -5, 0, 2]  
x_btwn = between(x, 2, 10)  
print x_btwn
```

```
y = [1, -3, 10, 0, 8, 9, 1]  
y_btwn = between(y, -2, 9)  
print y_btwn
```

```
xy_common = common(x, y)  
print xy_common
```

Syntax-based AOs

- Current data is too noisy to use syntax AOs

```
def between(numbers, low, high):  
    winners = []  
    for num in numbers:  
        if (low < num) and (num < high):  
            winners.append(num)  
    return winners
```

```
def common(list1, list2):  
    winners = []  
    for item1 in list1:  
        if item1 in list2:  
            winners.append(item1)  
    return winners
```

```
x = [2, 8, 7, 9, -5, 0, 2]  
x_btwn = between(x, 2, 10)  
print x_btwn
```

```
y = [1, -3, 10, 0, 8, 9, 1]  
y_btwn = between(y, -2, 9)  
print y_btwn
```

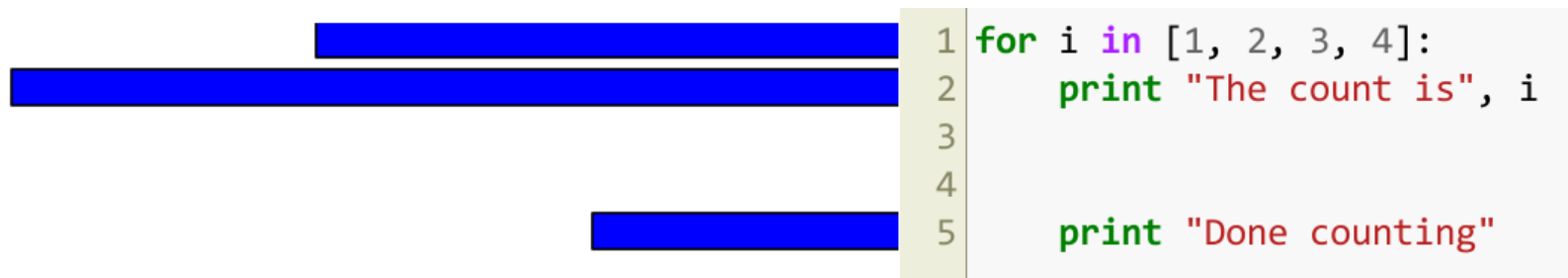
```
xy_common = common(x, y)  
print xy_common
```

Time Spent on Each Line

- Proportions of total fixation times (all participants)



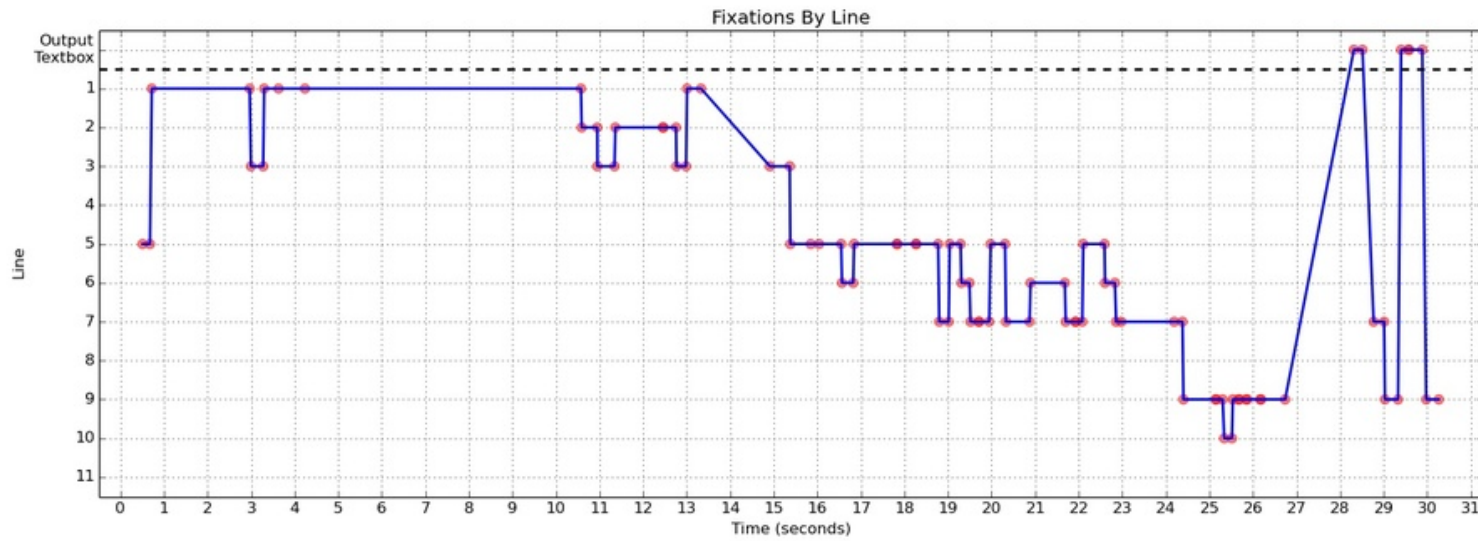
- Median grade = 10



- Median grade = 4

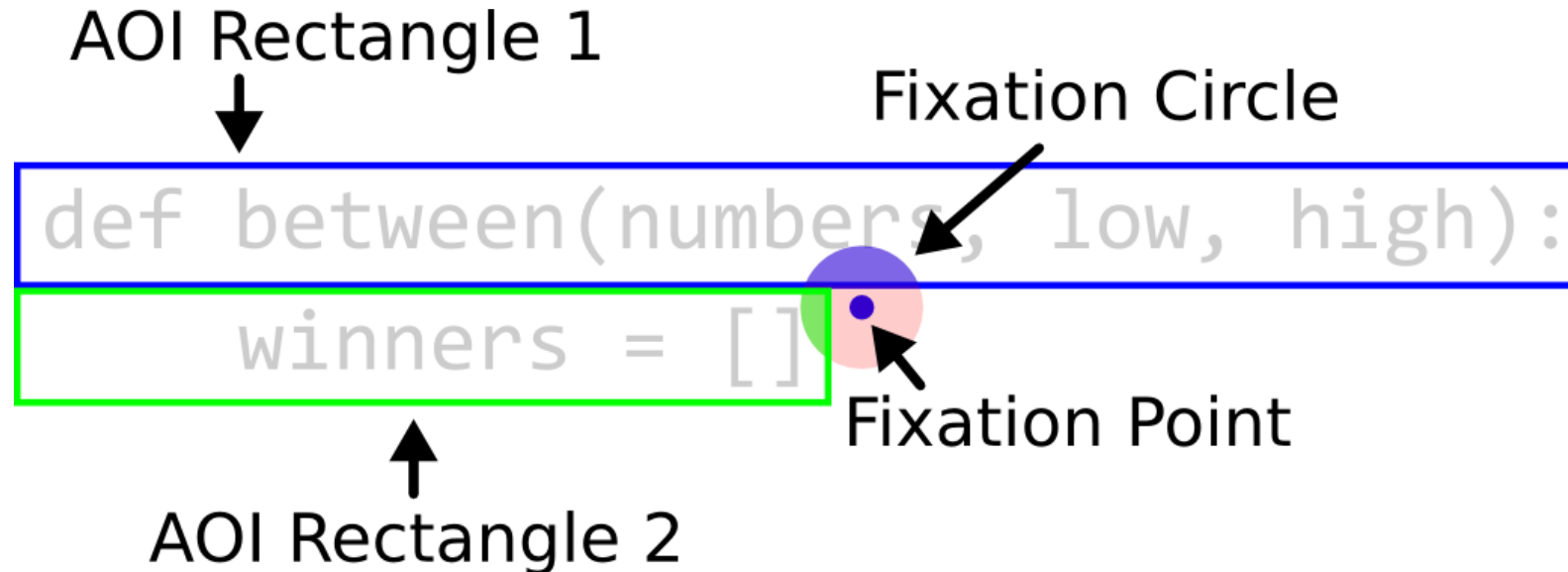
Timeline from Fixations and Areas of Interest

- By line and output box



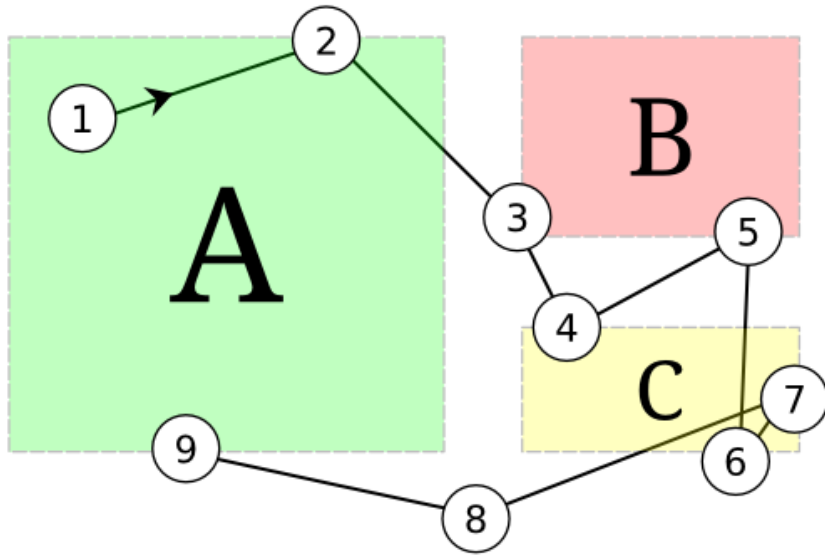
Mapping Fixations to Areas of Interest

- Multiple layers of AOIs, disjoint intra-layer
- In each layer, fixation → 0 or 1 AOI
- Circle around fixation point, AOI with largest area overlap



Scanpath Comparisons

- Levenshtein distance (string edit distance)
- Needleman-Wunsch (DNA sequence matching)



AABCBCCA

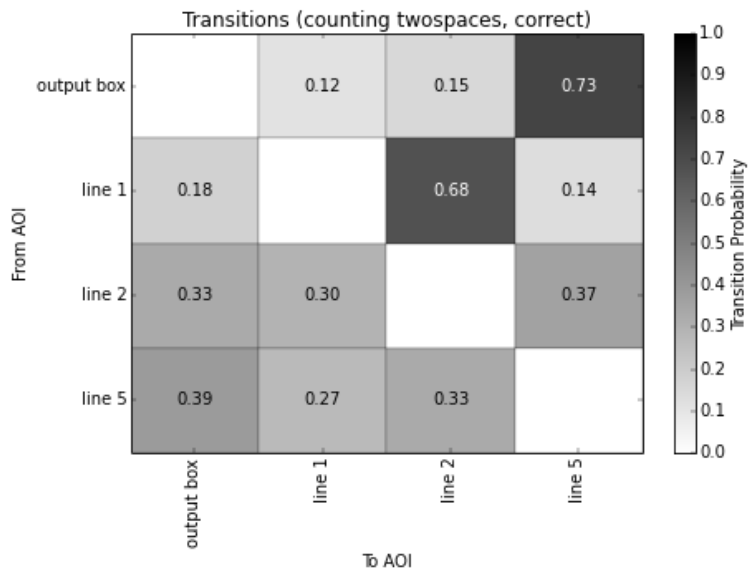


ABCBCA

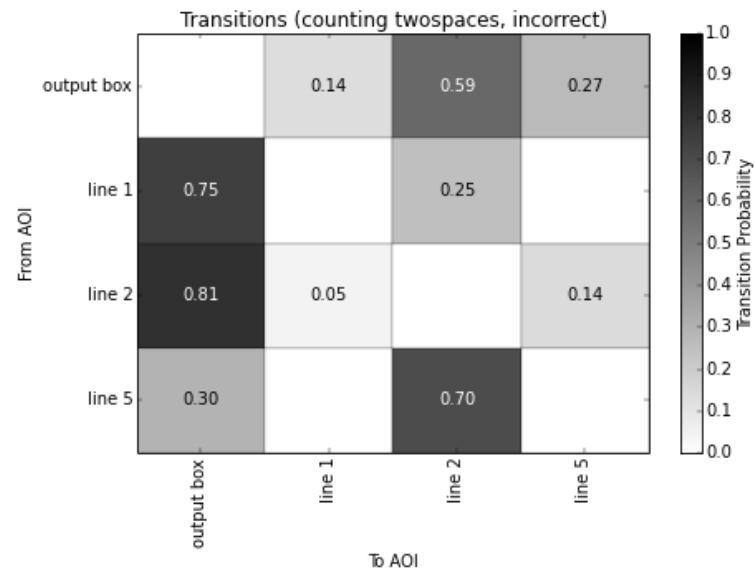
AOI Transition Matrix

```
1 for i in [1, 2, 3, 4]:
2     print "The count is", i
3
4
5     print "Done counting"
```

Correct Trials



Incorrect Trials



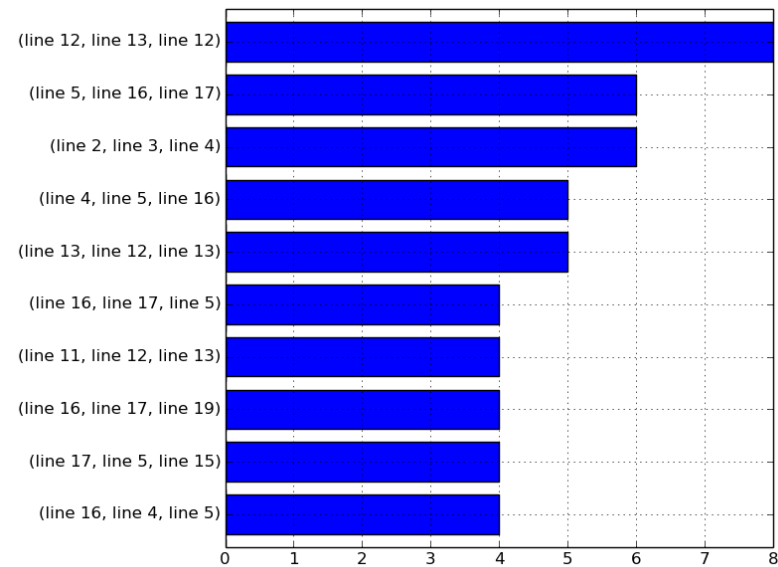
The eyeCode Library

- Data importing/cleaning
- Fixations → AOIs
- Scanpath construction/comparison
- Visualization, automated coding
- Mr. Bits models

```
# Load library and experiment data  
from eyecode import data, aoi  
fixes = data.hansen_2012.all_fixations()  
aois = data.hansen_2012.areas_of_interest()
```

```
# Filter down to a single trial  
trial_id = 17  
t_fixes = fixes[fixes.trial_id == trial_id]  
t_aois = aois[aois.trial_id == trial_id]
```

```
# Compute scanpath and plot top 10 tri-grams  
line_scan = aoi.scanpath_from_fixations(  
    t_fixes, repeats=False,  
    aoi_names = { "line": [] })  
tri_grams = nltk.util.ngrams(line_scan, 3)  
pandas.Series(tri_grams)\  
    .value_counts()[:10].plot(kind="barh")
```

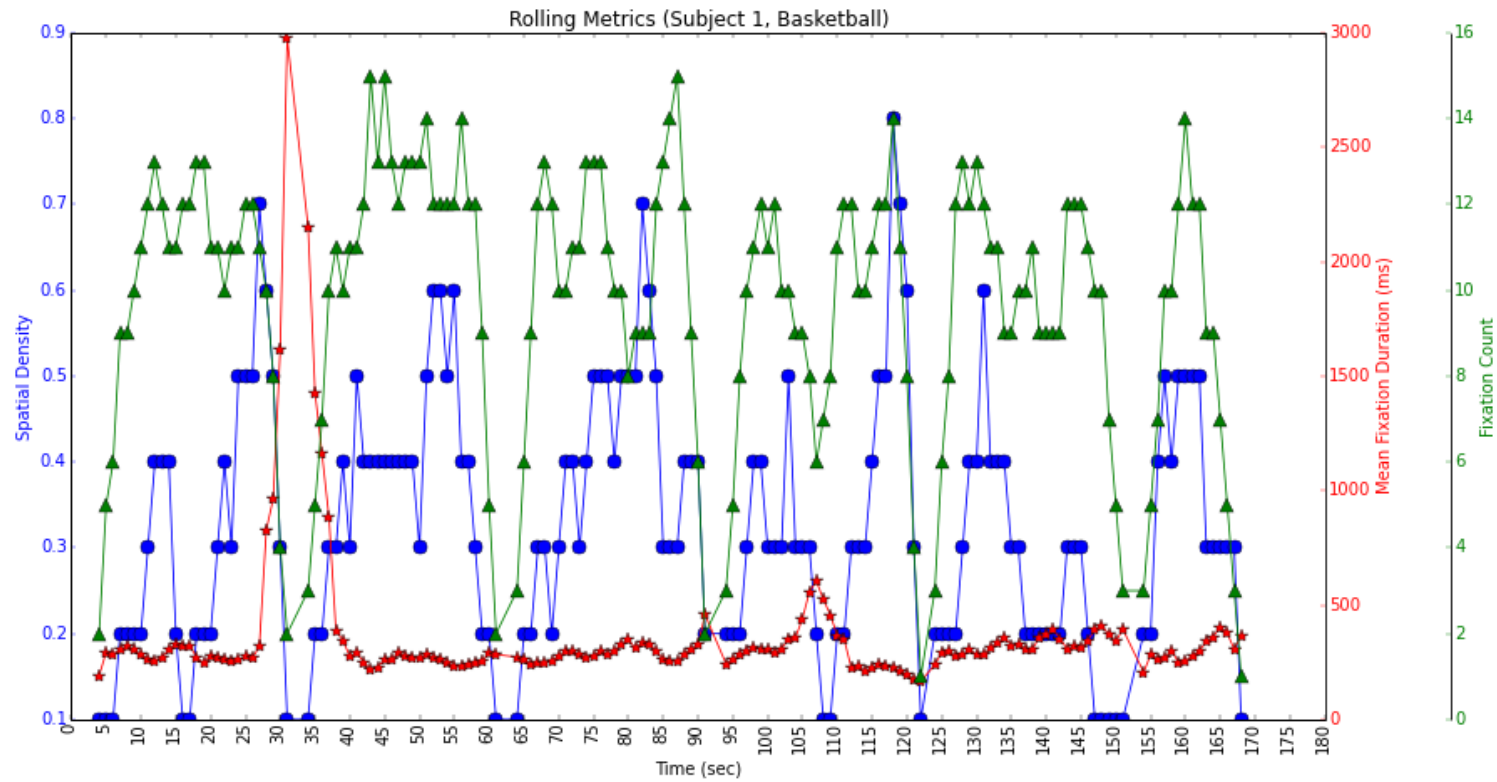


between - functions

```
1 def between(numbers, low, high):
2     winners = []
3     for num in numbers:
4         if (low < num) and (num < high):
5             winners.append(num)
6     return winners
7
8 def common(list1, list2):
9     winners = []
10    for item1 in list1:
11        if item1 in list2:
12            winners.append(item1)
13    return winners
14
15 x = [2, 8, 7, 9, -5, 0, 2]
16 x_btwn = between(x, 2, 10)
17 print x_btwn
18
19 y = [1, -3, 10, 0, 8, 9, 1]
20 y_btwn = between(y, -2, 9)
21 print y_btwn
22
23 xy_common = common(x, y)
24 print xy_common
```

Rolling Metrics (Koli Calling)

- Metrics computed over a 4 second rolling window



Results

Analysis is on-going

- Mean fixation durations are about 50ms above normal reading
- Connections between performance and fixations (counting, overload)
- About 75% of code lines are fixated in the first 30 sec (Uwano et. al, 2006)
- Edit-distance scanpaths are on average 75-80% different

3c. Experiment 3: Follow-Up

- **Completed Work**
 - Experiment design and software
 - New programs
- **Proposed Work**
 - Run new Mechanical Turk experiment (Software Carpentry students?)

Updated Design/Interface

- Only two versions of each program, **same output**
- Time limit for individual trials
- Record start/end timestamps in Javascript
- Discourage copy/paste in output box

Other Ideas

- Add tabs to separate helper functions and main code?
- Ask confidence level after each trial?

New and Updated Programs

- between
 - Focus on pulled-out vs. inline functionality
- counting
 - Use something besides "Done counting"
- scope
 - Return a value in one version
- order and whitespace
 - Larger changes in notation
- Variable names without implicit order
 - red, green, blue instead of a, b, c
- Drop `funcall` and `partition`
 - Nothing significant from previous experiment

counting

```
for i in [1, 2, 3, 4]:  
    print "The count is", i  
  
print "Done counting"
```

```
for i in [1, 2, 3, 4]:  
    print "The count is", i  
  
print "Today is Friday."
```

order

```
def green(x):  
    return x + 4  
  
def blue(x):  
    return x * 2  
  
def orange(x):  
    return green(x) + blue(x)  
  
def purple(x):  
    return orange(x) * blue(x)  
  
x = 1  
a = green(x)  
b = blue(x)  
c = orange(x)  
d = purple(x)  
print a, b, c, d
```

```
def purple(x):  
    return orange(x) * blue(x)  
  
def blue(x):  
    return x * 2  
  
def orange(x):  
    return green(x) + blue(x)  
  
def green(x):  
    return x + 4  
  
x = 1  
a = green(x)  
b = blue(x)  
c = orange(x)  
d = purple(x)  
print a, b, c, d
```

overload

```
a = 4  
b = 3  
print a * b
```

```
c = 7  
d = 2  
print c * d
```

```
e = "5"  
f = "3"  
print e + f
```

```
a = 4  
b = 3  
print a * b
```

```
c = 7  
d = 2  
print c * d
```

```
e = "x"  
f = "y"  
print e + f
```

scope

```
def add_1(added):  
    added = added + 1  
  
def twice(added):  
    added = added * 2  
  
added = 4  
add_1(added)  
twice(added)  
add_1(added)  
twice(added)  
print added
```

```
def add_1(added):  
    added = added + 1  
    return added  
  
def twice(added):  
    added = added * 2  
    return added  
  
added = 4  
add_1(added)  
twice(added)  
add_1(added)  
twice(added)  
print added
```

whitespace

```
intercept = 1  
slope = 5
```

```
x_base = 0  
y_base = slope * x_base + intercept  
print x_base  
print y_base
```

```
x_other = x_base + 1  
y_other = slope * x_other + intercept  
print x_other  
print y_other
```

```
x_end = x_base + x_other + 1  
y_end = slope * x_end + intercept  
print x_end  
print y_end
```

```
intercept = 1  
slope = 5
```

```
x_base = 0  
y_base = slope * x_base + intercept  
print x_base  
x_other = x_base + 1
```

```
print y_base  
y_other = slope * x_other + intercept  
print x_other  
print y_other
```

```
x_end = x_base + x_other + 1  
print x_end  
y_end = slope * x_end + intercept  
print y_end
```

4. Modeling: Mr. Bits

- **Completed work**
 - Basic model design
 - Preliminary model based on Python ACT-R
 - Dagstuhl talk on inductive programming (December)
- **Proposed work**
 - Line-based model with eye, DM, BM components
 - Qualitative comparison with human data

The basis of any informed discussion is a mathematical model. The best way to think of a mathematical model is a way to force everyone to clearly enumerate all assumptions being made, and to accept all logical reasoning that follows from those assumptions.

Given a model, everyone involved in a discussion can agree that either the conclusions of the model are correct, or one of the assumptions going into the model must be false.

— Chris Stucchio, http://www.chrisstucchio.com/blog/2013/basic_income_vs_basic_job.html

Model Overview

- **Computational process model with active vision**
 - Software complexity is resource expenditure
- **Reading and predicting printed output of code**
 - Same task as human programmers
- **Implemented on top of Python ACT-R**
 - Part of the eyeCode library

Limitations

- Subset of Python
 - for, if, def, print
- Single file program, one screen
- Basic text I/O schema, no OO
- Discrete retina, line-based reading
- No **production** learning (learning happens in DM)

Comparison to KLM and GOMS

Keystroke-Level Model (KLM)

- Task defined in terms of key presses, mouse clicks, mental preparation
- Fixed times for pressing keys, pointing mouse
- Skilled vs. unskilled timings possible

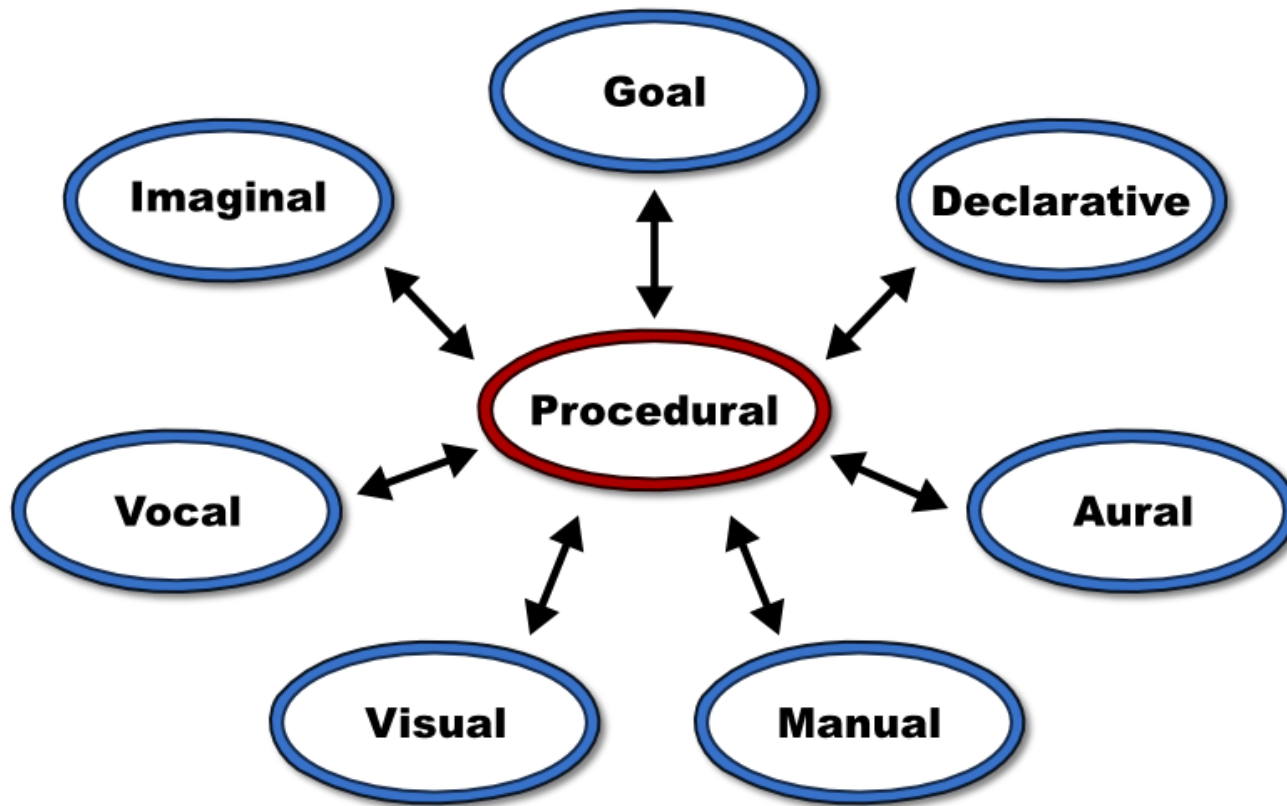
Goals, Operators, Methods, and Selection rules (GOMS)

- Task defined by goals/methods/rules, realized by operators
 - Eye movements, perceptual/cognitive/motor processors
- Fast, medium, slow times for operators
- Opaque cognition, no possibility of errors

ACT-R

- **Adaptive Control of Thought - Rational**
 - Atomic Components of Thought
-
- Cognitive architecture (CMU)
 - Implemented in LISP, Java, Python
 - Defines atomic perceptual/cognitive actions
 - Eye movements/encoding, memory retrieval, motor movements
 - Production-based
 - IF-THEN rules + current state determine next state
 - Subsymbolic layer
 - Declarative memory noise, manual "jamming"

ACT-R Modules



Mr. Bits and ACT-R

- **Goal**
 - High-level strategy: skim vs. predict
 - Sub-goal: chunk or trace
- **Visual**
 - Line-aligned sensor
 - Whitespace-separated tokens into Imaginal
- **Imaginal**
 - Context of current line/branch/loop
 - Type of line, role of variables
- **Declarative**
 - Short and long-term memory for variables/locations
 - Forgetting causes a re-trace
- **Procedural**
 - Behaviors to categorize lines, update DM, drive sensor
- **Manual**
 - Type response

Mr. Chips Retina

-	-	-	-	x		=		[1	,	2	,	-	*	*	
---	---	---	---	---	--	---	--	---	---	---	---	---	---	---	---	--

- **Open Questions**

- Add noise to saccades?
- Distinguish numbers/letters/operators in para-foveal?
- Peripheral vision, sensor shape?

ACT-R Declarative Memory

- Chunks (key/value pairs) with spreading activation
- Decay and retrieval latency predictions (Bayesian calculus)

Questions

- Probe by identifier prefix? p_y when p_x is seen
- Summary versus details for variables/functions
- Existing programming ontology?

Productions (Behaviors)

- LHS: patterns/guards, RHS: updates to buffers/modules
- Many ways to accomplish one behavior (within time constraints)

```
(p encode-letter
  =goal>
    isa      read-letters
    state    attend
  =visual>
    isa      text
    value    =letter1
  ?imaginal>
    buffer   empty
==>
  =goal>
    state    wait
  +imaginal>
    isa      array
    letter1  =letter1
)
```

Prototype Model: Mr. Bits 0.1a

overload - plusmixed

```
a = 4  
b = 3  
print a + b
```

```
c = 7  
d = 2  
print c + d
```

```
e = "5"  
f = "3"  
print e + f
```

```
a = 4  
b = 3  
print a + b
```

1. Move eye to line 1
2. Parse `a = 4`
3. Store `a` in DM with type `int` and value 4
4. Move eye to line 2
5. Parse `b = 3`
6. Store `b` in DM with type `int` and value 3
7. Move eye to line 3
8. Parse `print a + b`
9. Retrieve `a` from DM
10. Retrieve `b` from DM
11. Look up `4 + 3` in DM
12. Type 7

...

Example: Overload

multimixed

```
a = 4  
b = 3  
print a * b
```

```
c = 7  
d = 2  
print c * d
```

```
e = "5"  
f = "3"  
print e + f
```

plumixed

```
a = 4  
b = 3  
print a + b
```

```
c = 7  
d = 2  
print c + d
```

```
e = "5"  
f = "3"  
print e + f
```

strings

```
a = "hi"  
b = "bye"  
print a + b
```

```
c = "street"  
d = "penny"  
print c + d
```

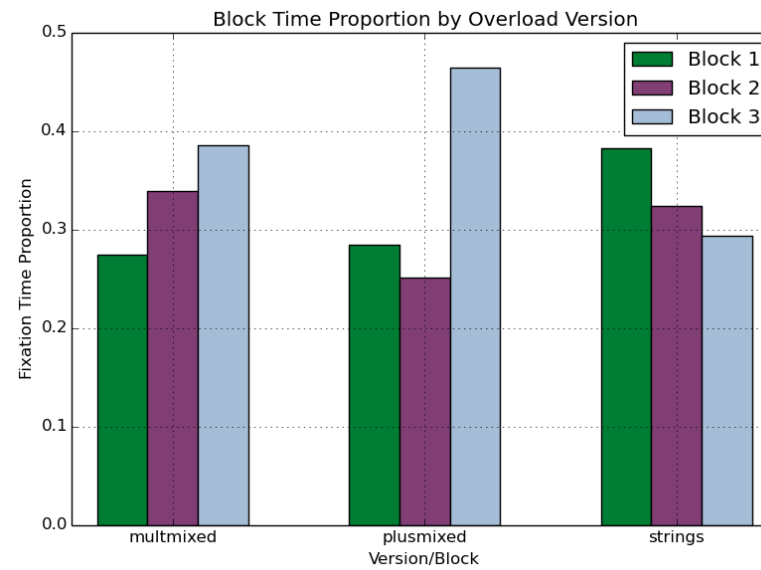
```
e = "5"  
f = "3"  
print e + f
```

Example: Overload

```
a = 4  
b = 3  
print a + b
```

```
c = 7  
d = 2  
print c + d
```

```
e = "5"  
f = "3"  
print e + f
```



Prototype Model: Mr. Bits 0.1a

overload - plusmixed

```
a = 4  
b = 3  
print a + b
```

```
c = 7  
d = 2  
print c + d
```

```
e = "5"  
f = "3"  
print e + f
```

```
a = 4  
b = 3  
print a + b
```

1. Move eye to line 1
2. Parse `a = 4`
3. Store `a` in DM with type `int` and value 4
4. Store `int` in DM
5. Move eye to line 2
6. Parse `b = 3`
7. Store `b` in DM with type `int` and value 3
8. Store `int` in DM
9. Move eye to line 3
10. Parse `print a + b`
11. Retrieve `a` from DM
12. Retrieve `b` from DM
13. Retrieve `+` from DM for `int` and `int`
14. Look up `4 + 3` in DM
15. Type 7

...

A Model of Human Programmers?

- Perhaps not exactly, but
 - Framework for testing a family of models
 - Provides timing and output predictions for task
 - Makes important questions explicit
 - First attempt at a program comprehension process model based on a cognitive architecture

5. Conclusion and Future Work

- Spatial reasoning
- Cognitive domain ontologies
- Research plan

Glasgow Spatial Array (1/2)

- Qualitative spatial reasoning (left-of, contains, etc.)
- **Goal:** better timing predictions for tracing



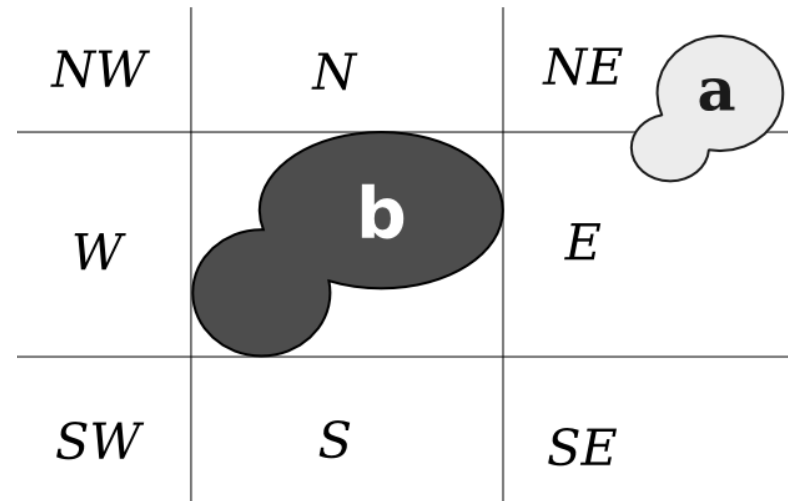
		rocks	
	grass		
	tree		
	lake		
well			
	hut		beach

Glasgow Spatial Array (2/2)

- Qualitative spatial reasoning (left-of, contains, etc.)
- Augmented with cardinal directions (NE-of and E-of)

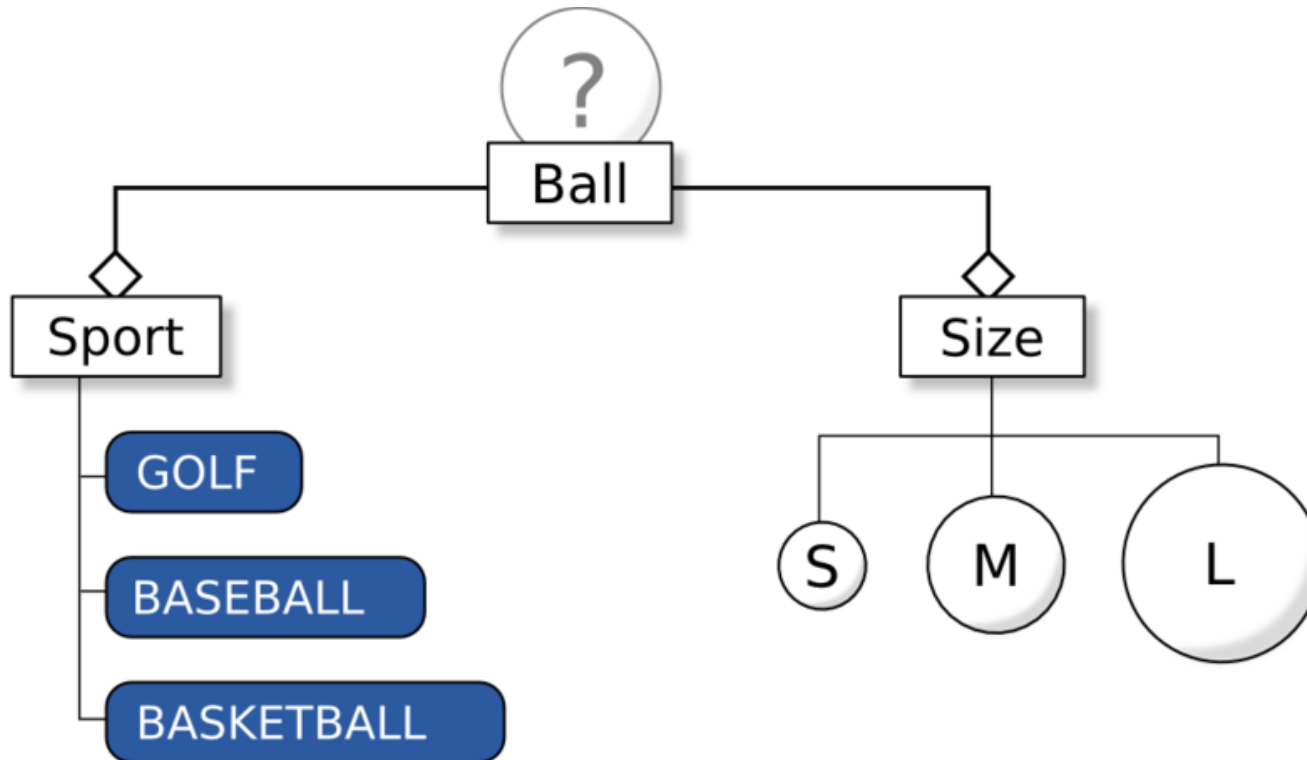
Questions

- Time cost for array inspection and reasoning?
- How to represent data flow? Time \approx space?



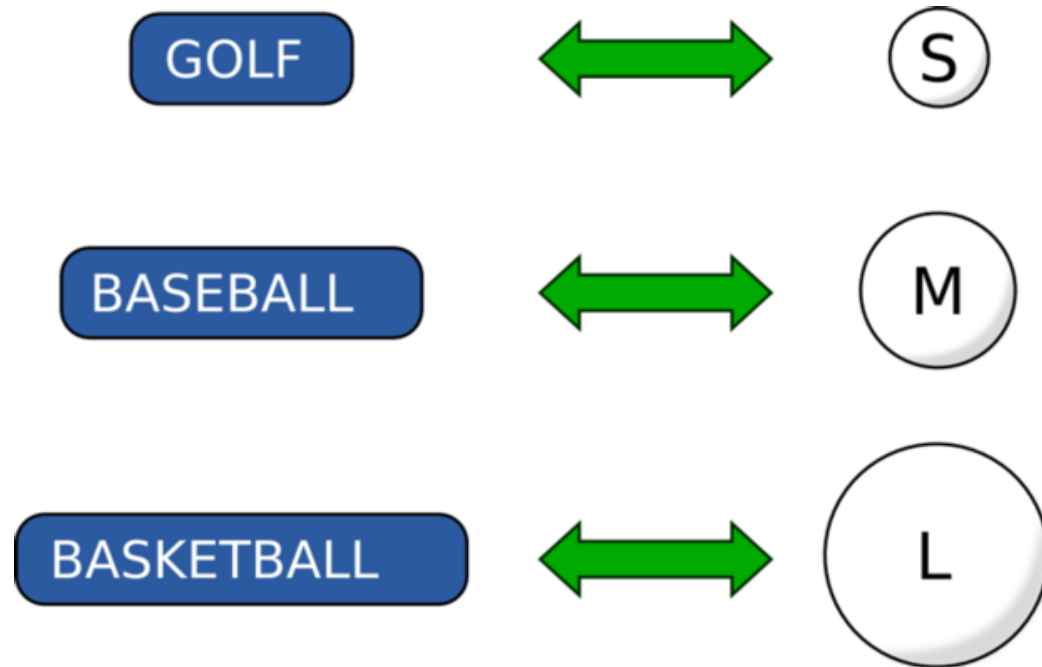
Cognitive Domain Ontologies (1/2)

- Domain knowledge represented as entities, relationships, constraints
- Constraint solver solutions are "possible worlds"
- **Goal:** recognize variable roles, algorithms



Cognitive Domain Ontologies (2/2)

- Top-down: entity is asserted, evidence is searched for
- Bottom-up: evidence is found, possible entities are considered



Research Timeline

Project	Planned Dates	Status
Literature review of Psychology of Programming	Spring-Summer 2011	Complete
Mechanical Turk and eye-tracking experiments	Spring-Fall 2012	Complete
Data analysis and publication of results	Fall 2013-Fall 2014	In Progress
Cognitive model development and follow-up experiment	Spring 2014-Spring 2015	In Progress
Final results	Spring 2015	Incomplete

Thank you!

