QUANTIFYING CODE COMPLEXITY AND COMPREHENSION

Michael Hansen

Submitted to the faculty of the University Graduate School

in partial fulfillment of the requirements for the degree

Doctor of Philosophy

in the School of Informatics and Computing

and Cognitive Science Program,

Indiana University

November 2015

Accepted by the Graduate Faculty, Indiana University,

in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Doctoral Committee

Andrew Lumsdaine, Ph.D.

Robert Goldstone, Ph.D.

Raquel Hill, Ph.D.

Chen Yu, Ph.D.

August 28, 2015

Dedicated to Jennifer, Oliver, Mom, Dad, Eric, Michelle, and Alex.

May we all be remembered when Men are just research topics in dissertations written by rabbits.

Michael Edward Hansen

QUANTIFYING CODE COMPLEXITY AND COMPREHENSION

What factors impact how difficult code is to understand? Small, simple programs are often easier to grasp, but can still be locally ambiguous, visually confusing, or violate implicit expectation. The cognitive complexity of a program is something more than its line length, visual layout, and problem domain. It is a gestalt phenomena that arises when a programmer must perform a specific task on a given program or segment of code. Modeling these phenomena is a challenging real-world problem at the intersection of computer science and cognitive science.

Programmers have been the subject of traditional psychology experiments for several decades, correlating code features with task performance. The recent inclusion of eye-tracking in program comprehension experiments, however, has given researchers a unique window into programmers' cognitive processes. Advances in the state of the art for quantitative cognitive modeling have opened the door for a unique confluence of technique and technology: quantitative, cognitive models of human program comprehension.

We present an experiment in which 162 programmers, from beginner to expert, were asked to predict the printed output of ten short Python programs. Different versions of the same program were randomly selected, exposing groups of participants to minor variations in code spacing, naming, and function. A subset of participants performed the experiment in front of an eye tracker, allowing for a combined analysis of reading and response behavior. The observed errors, timings, and keystrokes formed the basis for two program comprehension models: one modeling correct participants' low-level reading and response behavior, and the other modeling incorrect participants' interpretation errors. These models form the foundation for future research in program comprehension, and join the frontiers of quantitative modeling in cognitive science.

Contents

1 Main Introduction

What makes some code harder to understand? The behavior of small, simple programs is often easier to predict than larger programs with many inter-connected components. Yet small programs or snippets of code can still be difficult to understand because they are locally ambiguous, visually confusing, or violate some unwritten rule [16, 89]. The *cognitive complexity* of a program is something more than its length, layout, or even its problem domain; it is a gestalt phenomena that arises when *a programmer* must perform *some task* on a *specific program*. While experience (and tooling) can help programmers reduce this complexity, there are cases where implicit expectations can lead even the most experienced programmers astray [46]. What is the source of these cognitive errors? Can they be predicted using some model of a programmer?

For nearly four decades, psychologists have studied programmers and the kinds of errors they make when reading and writing code. Prior experience, both with the programming language and problem domain has been identified as an important variable when predicting a programmer's performance [29]. But the nuanced relationship between experience, task, code, and performance ultimately depends on internal *cognitive processes* and *mental models*. More recently, eye-tracking has enabled cognitive scientists to observe programmers' low-level reading behaviors, and begin to infer the details of their inner mental processes [4, 14]. At the same time, quantitative cognitive modeling has made great strides towards tackling larger, "real world" tasks using frameworks like ACT-R [3, 78]. A confluence of these techniques and technologies is the focus of this dissertation and line of research, with the goal of producing a *quantitative cognitive model of human program reading and comprehension*.

Two experiments, collectively referred to as the *eyeCode* experiment, run through the heart of this research effort. In eyeCode, we asked 162 programmers of varying experience levels to predict the printed output of ten small Python programs. Our bank of twenty-five programs contained two or three versions of the ten programs each programmer randomly saw, most with subtle notational differences. By observing *errors, timing differences, and keystrokes*, we characterized the effect of these notational differences on the programmers while taking their experience and education levels into account. For the 29 local participants who performed eyeCode in front of an eye-tracker [95], eye movement data was also available.

We developed *two cognitive models*, which quantify program understanding at different levels of abstraction. The first, called **Mr. Bits** (after the Mr. Chips model of text comprehension [57]), predicts the eye movements and keystrokes of a programmer reading and responding to the eyeCode task. Mr. Bits is based on the ACT-R cognitive architecture, and assumes the programmer does not make errors when interpreting their programs. We found that a specific version of Mr. Bits fit our human data the best: one with ACT-R's sub-symbolic functionality enabled (i.e., with memory retrieval times based on recency and frequency) and the inability to remember entire lists of numbers after a single viewing. Although Mr. Bits does not make errors, the simulated time taken by the model to read and respond to a program may be useful as a complexity metric. Mr. Bits' "trial time" was moderately correlated with other common code complexity metrics, such as number of lines and branch points, but did not significantly overlap with any single existing metric. Thus, this model may serve as a means of *ranking programs by their cognitive complexity*.

Our second model, **Nibbles** (a play on the name Mr. Bits), took a very different approach. Using a formalism called *Cognitive Domain Ontologies* (CDOs) and a *constraint solver*, Nibbles is able to generate alternative interpretations of code, some of which may contain errors. A subset of Python's grammar, as well as knowledge about basic variable types and categories of tokens and lines, is included in Nibbles as a set of *constraints*. These constraints, combined with observed code, allow the model to prune its space of possible programs down to a single "best" interpretation. Given some basic structural constraints – e.g., token lengths and counts – Nibbles can also *infer the details of unobserved code*. Due to the large number of possible programs, even for Nibbles' limited subset of Python, a form of *attention* was necessary to limit the scope of inference to nearby code only. We used Nibbles to model three of the most common errors observed in the eyeCode experiment, from incorrect line groupings to deceptively non-functional code. A combined model with Mr. Bits and Nibbles is planned for future work.

This dissertation is broken into three main parts, each of which should be readable in isolation. **Part 1** examines the results of both experiments, including remote participants from Amazon's Mechanical Turk [2]. We quantified participants' performance using metrics like the string-edit distance [79] between their response from the correct one, and how long they took to complete each trial. We found that the correctness of a participant's response for a given trial could be reasonably predicted from the number of (extra) keystrokes they made, the proportion of the trial spent reading versus typing, and how long the trial took from start to finish. The main contributions of Part 1 are (1) defining a family of performance metrics for our output prediction task, (2) quantifying participants' performance differences between different versions of the "same" program, and (3) the identification and categorization of common errors due to expectation violations or aspects of physical notation.

Part 2 focuses on data collected from the 29 eye-tracking participants of the eyeCode experiment. We develop a methodology for analyzing this data, and a Python library (also called eyeCode) that generates useful summary plots of eye movement data *for program comprehension experiments*. Our findings were partially in-line with others', such as the observation that keywords are the least frequently fixated tokens. We observed different values for eye-tracking metrics – e.g., mean fixation duration – and, surprisingly, did not find a significant correlation between a programmer's experience and *any* of our metrics. We hypothesize that this is due to the uniqueness of our output prediction task, and consider the task itself to be a contribution. Additionally, the eyeCode Python library is free and open source [62] for other researchers to use, as well as the complete eye movement dataset. Finally, the use of transition matrices and time-series metrics for identifying import code lines and moments in a trial has contributed to research on computing education [15].

Part 3 describes two models of program comprehension, based on the human errors observed in the eyeCode experiment. Mr. Bits, our lower-level ACT-R model, predicts the eye movements and keystroke timings of an eyeCode participant reading and responding to a particular program. Mr. Bits uses the built-in Python debugger to parse code and generate its evaluation goals, while ACT-R's *declarative memory, visual, procedural, and motor modules* combine to produce human-like timings. Nibbles parses raw text, unlike Mr. Bits, but does not produce timing/keystroke predictions. Instead, a "mental model" of the program is generated after reading each line, based on a formalism called Cognitive Domain Ontologies [33]. Using a LISP-based constraint solver named Screamer [88], Nibbles infers the semantics of tokens, lines, and groups of lines (e.g., loops, function definitions). Both models, Mr. Bits and Nibbles, are contributions to the program comprehension cognitive modeling community. A combined model, spanning the gap from raw text to eye movements and keystroke timings, would be a major step forward for cognitive modeling in general. This dissertation lays the foundation for such a model, and we are excited to see what can be built upon it.

Part 1

What Makes Code Hard to Understand?

Evidence from Output Prediction

Abstract

What factors impact the comprehensibility of code? Intuition and previous research suggests that simpler, expectation-congruent programs should take less time to understand and be less prone to errors. We present an experiment in which participants with varying levels of programming experience predict the printed output of ten short Python programs. We use subtle differences between program versions to demonstrate that seemingly insignificant notational changes can have profound effects on prediction correctness, response times, and keystrokes. Our results show that experience increases performance in most cases, but may hurt performance significantly when underlying assumptions about related code statements are violated. Additionally, we find that response correctness can be predicted using a combination of performance metrics and participant demographics.

2 Part 1: Introduction

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

The design, creation and interpretation of computer programs are some of the most cognitively challenging tasks that humans perform. Understanding the factors that impact the cognitive complexity of code is important for both applied and theoretical reasoning. Practically, an enormous amount of time is spent developing programs, and even more time is spent debugging them. If we can identify factors that expedite these activities, a large amount of time and money can be saved. Theoretically, programming is an excellent task for studying representation, working memory, planning, and problem solving in the real world.

A program is an abstract specification for a set of operations, but its code exists in a concrete notational system made to be read and written by humans. With some exceptions, we intuitively expect programs whose code is **shorter** and **simpler** to be easier to understand, especially for more **experienced** programmers. Given the expressive power of most programming languages, we also expect the exceptions to be fairly nuanced; even operationally simple programs may be difficult when misleading variable names are used or implicit expectations are violated [89].

What are the relationships between a program's source code, the programmer interpreting it, and how difficult the program is to understand? To start, we must quantify each of these components, including what it means to *understand a program*. In this part, we use an **output prediction task** to quantify understanding – we say a programmer understands a program if she can accurately predict its printed output. Using a set of five **performance metrics**, we measure a programmer's prediction accuracy, response time, and individual keystrokes. To quantify features of a program's source code, we employ several commonly-used **code complexity metrics**, such as lines of code, cyclomatic complexity, and Halstead effort. These metrics are often used to predict how error prone or difficult to understand a piece of code is. Lastly, we quantify aspects of the programmers themselves using self-reported **demographics**, such as programming experience and age. Our experiment explores the relationships between these demographics, prediction performance, and code complexity.

The Experiment and Research Questions

We present a web-based experiment in which participants with a wide variety of Python and overall programming experience predicted the output of 10 small Python programs. Most of the program texts were less than 20 lines long and had fewer than 8 linearly independent paths (as measured by cyclomatic complexity [59]). We used ten different program *kinds*, each of which had two or three *versions* with subtle differences. For each participant and program, we collected five different performance measures:

1. **Output distance** - the normalized edit distance between the correct answer and the participant's output prediction.
2. **Duration** - the amount of time taken to predict a single program's output (log scale).
3. **Keystroke coefficient** - the number of keystrokes a participant typed divided by the number of keystrokes needed to type the correct output.
4. **Response proportion** - the amount of time between the first and last keystroke divided by the total trial duration.
5. **Response corrections** - the number of times the participant's output prediction decreased in size (i.e., backtracks or corrections).

The different versions of our programs were designed to test three underlying **research questions**. First, "*How are programmers affected by programs that violate their expectations, and does this vary with expertise?*" Previous research suggests that expectation-violating programs should take longer to process and be more error-prone than expectation-congruent programs. There are reasons to expect this benefit for expectation-congruency to interact with experience in opposing ways. Experienced programmers may show a larger influence of expectations due to prolonged training, but they may also have more untapped cognitive resources available for monitoring expectation violations. In fact, given the large percentage of programming time that involves debugging (it is a common saying that 90% of development time is spent debugging 10% of the code), experienced programmers may have developed dedicated monitors for certain kinds of expectation-violating code.

The second question is: "*How are programmers influenced by physical characteristics of notation, and does this vary with expertise?*" Programmers often feel like the physical properties of notation have only a minor influence on their interpretation process. When in a hurry, they frequently dispense with recommended variable naming, indentation, and formatting as superficial and inconsequential. However, in other formal reasoning domains such as mathematics, apparently superficial formatting influences like physical spacing between operators has been shown to have a profound impact on understanding [43]. Furthermore, there is an open question as to whether experienced or inexperienced programmers are more influenced by similar physical aspects of code notation. Experienced programmers may show less influence of these "superficial" aspects because they are responding to the deep structure of the code. By contrast, in math reasoning, experienced individuals sometimes show more influence of notational properties of the symbols, apparently because they use perception-action shortcuts involving these properties in order to attain efficiency. Our final question is deceptively simple: "*Can task performance be predicted by code complexity metrics and programmer demographics?*" It is common practice in some organizations to use "code complexity" metrics, such as number of lines and control flow statements, to identify potentially error-prone code that is hard to comprehend [36]. Well-defined metrics, including lines of code and cyclomatic complexity [59], have recommended limits for components of large software projects [64]. One failing of these metrics, however, is that they do not take a programmer's experience into account. While experience does not necessarily equal expertise, a veteran programmer and a novice are likely to face different challenges when comprehending the same code. Although our programs and task are fairly simple, it is reasonable to expect that (1) task performance be moderately correlated with code complexity metrics, and (2) performance predictions **improve** when programmer demographics are also considered.

Outline. Section 3 begins by providing the necessary background on research in the psychology of programming. Next, Section 4 provides import details about our experiment (see Appendix A for the source code to all programs). Results are presented in Section 5, broken down by performance/complexity metrics, program versions, and participant expertise. Section 6 discusses the results from the previous section, and relates them to our three research questions. Finally, Section 7 concludes and describes our plans for future work.

3 Part 1: Background and Related Work

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

Psychologists have been studying programmers for at least forty years. Early research focused on correlations between task performance and human/language factors, such as how the presence of code comments impacts scores on a program comprehension questionnaire. More recent research has revolved around the cognitive processes underlying program comprehension. Effects of expertise, task, and available tools on program understanding have been found [29].
Studies with experienced programmers have revealed conventions, or "rules of discourse," that can have a profound impact (sometimes negative) on expert program comprehension [89]. Violations of these rules are thought to contribute directly to how difficult a program is to understand. Much like violating the "rules" of verbal discourse can make a conversation difficult (e.g., awkward phrasing or unconventional uses of words), Soloway and Ehrlich argued that violating discourse rules in code strains communication between programmers. Interestingly, this strained communication affects expert programmers more than novices, suggesting that programming expertise is partially due to learned semantic conventions.
In another psychology of programming study, Soloway and Detienné asked programmers to recall programs that subtly violated particular coding conventions, such as naming the outermost `for` loop index *j* instead of *i*. Expert programmers consistently recalled the loop index as the more canonical *i*, suggesting that storage of code in long-term memory is mediated by a template-like structure [27]. deGroot and Gobet have extended standard short-term memory chunking theory with templates, long-term memory structures with "slots" that allow for rapid memorization of domain items [25]. This extension, however, has only been applied to the domain of chess, where

templates are predicted to correspond to common chess piece positions. For programmers, the precise correspondence between templates and code is currently unknown. Templates may exist for code directly (e.g., `for` loops, `if` statements), or may correspond to higher-level attributes of the code, such as data-flow patterns or variable roles [74].

Our present research focuses on programs much less complicated than those the average professional programmer typically encounters on a daily basis. The demands of our task are still high, however, because participants must predict precise program output. In this way, it is similar to debugging a short snippet of a larger program. Code studies often take the form of a code review, where programmers must locate errors or answer comprehension questions after the fact (e.g., does the program define a Professor class? [10]). Our task differs by asking programmers to mentally simulate code without necessarily understanding its purpose. In most programs, we intentionally use meaningless identifier names where appropriate (variables `a`, `b`, etc.) to avoid influencing the programmer's mental model.

Similar research has asked beginning (CS1) programming students to read and write code with simple goals, such as the Rainfall Problem [44]. To solve it, students must write a program that averages a list of numbers (rainfall amounts), where the list is terminated with a specific value – e.g., a negative number or 999999. CS1 students perform poorly on the Rainfall Problem across institutions around the world, inspiring researchers to seek better teaching methods. Our work includes many Python novices with a year or less of experience (94 out of 162 participants), so our results may also contribute to ongoing research in early programming education.

3.1 Code Complexity Metrics

Code complexity has traditionally been measured using structural or textual features of the code (sometimes called the representational complexity [16]). These complexity metrics are often used as proxies for the maintainability or error-proneness of a piece of code [40]. They are quick and easy to compute, allowing them to be used as tools for identifying potentially problematic code in large software projects. There are drawbacks to using some of these metrics, however, such as their strong statistical correlation with code size [37]. Because of this correlation, it is often difficult to quantify differences between large codebases are that truly complex and ones that are simply repetitive, though metrics inspired by Kolomogorov complexity may overcome this problem [98]. Traditional complexity metrics do not claim to *directly* measure how difficult a program is to understand – its **cognitive complexity** [16]. However, some researchers have claimed they measure something related. It has been argued that having to mentally juggle too many lines of code, branches, or operators and operands strains the capacity limitations of programmers' short-term memories [55]. Therefore, metrics such as number of lines, cyclomatic complexity, and Halstead effort can be used as a proxy for cognitive complexity. While it has been found that short-term memory capacity is indeed limited and approximately the same across individuals, the size of individual items in short-term memory (called chunks) varies greatly [19]. This means we cannot infer that a given piece of code will "overflow" a programmer's short-term memory simply because we cannot assume that a textual measure captures the size of their short-term memory chunks. More complete cognitive models have been proposed to capture the relationships between code text and program understanding, such as the Stores Model [30] and the Cognitive Complexity Model [16]. No quantitative implementation of these models currently exists, however, so their ability to predict performance in a particular programming task (such as output prediction) remains unknown.

4 Part 1: Methodology

One hundred and sixty-two participants were recruited from the Bloomington, IN area (29 participants), on Amazon's Mechanical Turk (130 participants), and via e-mail (3 participants). The local participants in Bloomington were paid $10 each, and performed the experiment in front of an eye-tracker (see Section 7.1 regarding future work). Mechanical Turk participants were paid $0.75, and performed the experiment over the Internet via a web browser. All participants were screened for a minimum competency in Python by passing a basic language test. The mean participant age was 28.4 years, with an average of 2.0 years of self-reported Python experience and 6.9 years of programming experience overall. Most of the participants had a college degree (69.8%), and were current or former Computer Science majors (52.5%). Figure 1 has a more detailed breakdown of the participant demographics.

The experiment consisted of a pre-test survey with questions about demographics and experience,

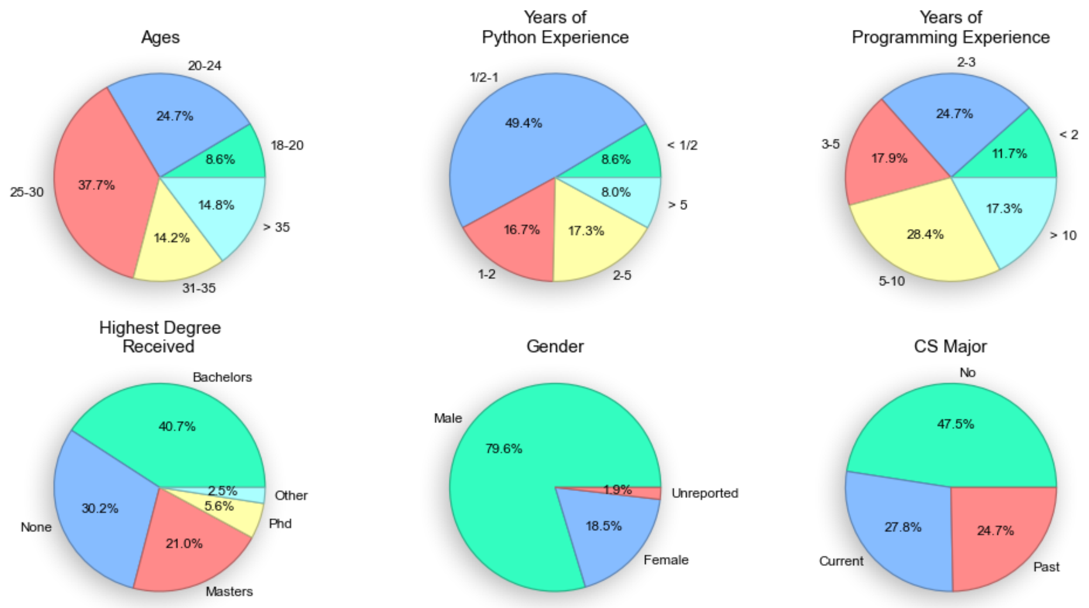
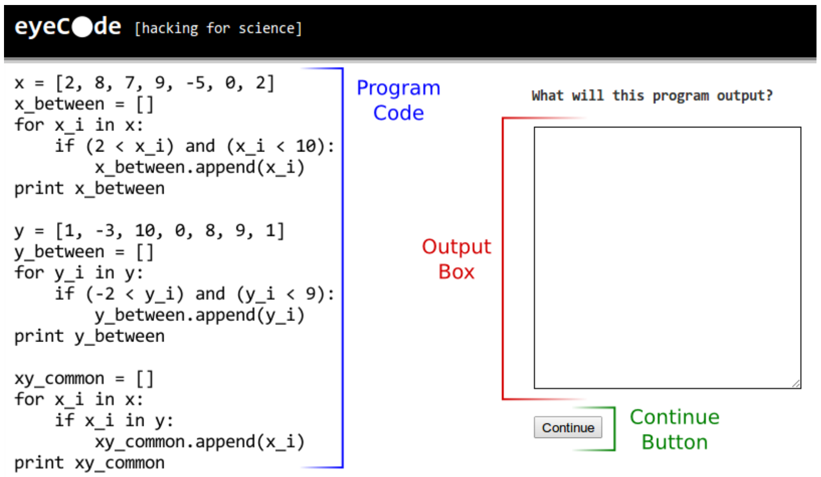


Figure 1: *Demographics of all 162 participants.*

ten trials (one program each), and a post-test survey assessing confidence and requesting feedback. The pre-test survey gathered information about the participant's age, gender, education, Python experience, and overall programming experience. Participants were then asked to predict the printed output of ten short Python programs, one version randomly chosen from each of ten program bases (Figure 2). The presentation order and names of the programs were randomized, and all answers were final (Figure 3). No feedback about correctness was provided and, although every program produced error-free output, participants were not informed of this fact beforehand. The post-test survey gauged a participant's confidence in their answers and the perceived difficulty of the task overall.

We collected a total of 1,620 trials from 162 participants starting November 20, 2012 and ending January 19, 2013. Trial responses were manually screened, and a total of 35 trials were excluded based on the response text [1]. Trial completion times ranged from 12 to 518 seconds. Outliers beyond three standard deviations of the mean (in log space) were discarded (10 of 1,585 trials), leaving a total of 1,575 trials to be analyzed. Participants had a time limit of 45 minutes to complete the entire experiment (10 trials + surveys), but were not constrained to complete individual trials in

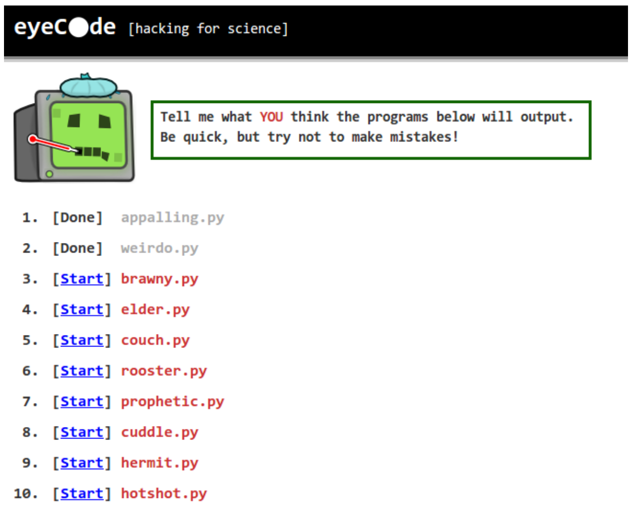
[1]Excluded trial responses were things like "error" or an English description of the code. While these responses may be interesting in their own right, we do not consider them relevant to the intended task.



```
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
```

Figure 2: *Sample trial from the experiment (`between inline`). Participants were asked to predict the exact output of ten Python programs.*

any specific amount of time.



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
```

Figure 3: *Home screen for the experiment. Each program was assigned a random name.*

There were a total of twenty-five Python programs in our experiment belonging to ten different program bases. These programs were designed to be understandable by a wide audience, and therefore did not touch on Python features outside of a first or second introductory programming course (no lambda expressions, generators, etc.). The programs ranged in size from 3 to 24 lines of code, and did not make use of any standard or third-party libraries. Code metrics for each program were calculated using the open source PyMetrics library [18] (see Table 21 in the appendix). Most programs fell well within recommended complexity metric ranges for being “understandable” [64].

4.1 Mechanical Turk

One hundred and thirty participants from Mechanical Turk (MT) completed the experiment online and received \$0.75 each . All MT workers were required to pass a Python pre-test, and could only participate in the experiment once. Although participants were not paid based on performance, some precautions were taken to ensure the task was completed properly. All code was displayed as an image, making it difficult to copy/paste the code into a Python interpreter for quick answers. Responses were manually screened, and any restarted trials [2] or unfinished experiments were discarded (18 out 1,620 trials overall). For reference, the website and associated code for the experiment is freely available online at https://github.com/synesthesiam/eyecode-web.

4.2 Python Programs

Although individual participants were tested on only ten programs, there were a total of twenty-five Python programs in the experiment. We had ten program **bases**, from which one of 2-3 **versions** was randomly selected and given to a participant. Appendix A contains the complete listing of all program source code and printed output. While the programs were small and relatively simple, they were designed to test specific effects of *physical notation* and *expectation violations*.

The `between`, `counting`, `funcall`, `order`, `rectangle`, and `whitespace` programs were designed to test different choices in *physical notation*. For these program bases, all versions produced identical output. We expected differences in notation to result in differences in performance, specifically in the amount of time taken to complete the trial. For example, the `order` programs had 3 functions that were defined and called either in the same order (`inorder`) or a different order (`shuffled`). We expected faster responses in the same order case because the congruence would likely aid visual search..

The `initvar`, `overload`, `partition`, and `scope` programs were designed to violate the programmer’s *expectations*. Our expectations were that experienced programmers would be more

[2] A trial could be restarted by manipulating the web browser’s address field. Our server allowed participants who restarted trials to complete the experiment, but those individual trials were excluded from analysis.

prone to specific mistakes. The `scope` programs, for instance, contained several functions that did not modify their arguments or return any value (an explicit violation of Soloway's maxim "don't include code that won't be used" [89]). We hypothesized that less experienced programmers would interpret the programs literally, giving them an advantage when the code and common expectations diverged. Similarly, two out of three versions of `initvar` contained an off-by-one error in a summation – something an experienced programmer may miss if they aren't careful.

4.2.1 Complexity Metrics

We quantified differences between programs using six **code complexity metrics** (see Table 1). These metrics were computed on the source code with features such as number of lines, independent code paths, and operator / operand counts using the open source PyMetrics library [18]. A complete listing of programs and their associated complexity metrics is available in Table 21 in the Appendix. While these metrics alone do not strongly predict performance in our task (i.e., longer programs are not necessarily more difficult), they are expected to become more useful when combined with additional information about the programmer (e.g., experience, age). Section 5.1.6 explores this in more detail.

One of the most common complexity metrics, **lines of code**, is computed by simply counting up lines in the program's source code (including blank lines, in our case). Counting lines is fast, and the result can be used as a rough indication of a program's overall complexity; much like using page count gives a sense of how complex a book's plot may be. Lines and pages, of course, are not a reliable proxy for how difficult a book or a program is to understand. Hundreds of lines of boilerplate C code, and an extremely clever twenty-line Haskell program will be assigned high and low complexity rankings respectively if lines of code is the only metric.

Another common metric, **cyclomatic complexity**, is computed by counting up control flow statements [59] (e.g., `if`, `for`, `while`). Cyclomatic complexity measures the number of *linearly independent paths* through a program, and is useful when determining the number of tests needed to achieve proper code coverage. More often, however, this metric is used to predict the number of defects in a function or entire program [39]. Cyclomatic complexity has also been found to be highly correlated with lines of code [36]. This is not entirely unexpected, however. It is difficult (or at least uncommon) to write large programs with very few control flow statements.

In 1977, Maurice Halstead introduced several complexity metrics, collectively referred to as the **Halstead metrics** [45]. These metrics depend on dividing the syntax tokens of a programming language into *operators* and *operands*. Given this division, the following quantities can be computed for a program or sub-module: (N_1) the total number of operators, (N_2) the total number of operands, (η_1) the number of *distinct* operators, and (η_2) the number of distinct operands. Halstead proposed the following metrics based on these quantities:

- Difficulty: $(\eta_1 \times N_2)/(2 \times \eta_2)$
- Volume: $V = (N_1 + N_2) \times \log_2(\eta_1 + \eta_2)$
- Effort: $E = V \times D$

Halstead predicted that the volume (V) and effort (E) values for a program would be related to the time needed to produce and review the program. Our preliminary analysis found that these two quantities were almost perfectly correlated for all programs, so we focus on Halstead effort only as a measure of complexity. In addition to being correlated with lines of code, Halstead effort has also been found to be linearly correlated with source code indentation [48].

Because our task is focused on output prediction, we include two additional metrics derived from the output of each program. The number of **output characters** and **output lines** are simply the number of individual characters and lines in the *true* printed output. While these are not directly observable in the source code, they can have an important effect on performance. A program with more characters in its output, for example, may be observed to have more response corrections simply because the chance of typing mistake is higher while responding, not because the program is more difficult to understand.

4.3 Performance Metrics

Participants' performance was quantified with a set of five performance metrics (Table 2). To help explain the details of each metric, consider the sample trial presented in Figure 4. In this trial, the participant is presented with the Python program `print "1" + "2"`. After the initial *reading period* (i.e., before any keystrokes), the participant types a "3". With that first keystroke, the *response period* has begun. Continuing with the trial, the participant realizes their mistake (+ means concatenate in

Metric	Definition
Lines of Code	Number of lines in the code (includes blank lines)
Cyclomatic Complexity	Number of linearly independent paths through the program
Halstead Volume	Program length (N) times the log of the program's vocabulary (n).
Halstead Effort	Effort required to write or understand a program ($D \times V$)
Output Characters	Number of characters in the program's printed output
Output Lines	Number of lines in the program's printed output

Table 1: *Code complexity metrics computed for each program.*

Python when the arguments are strings), and modifies their response before submitting by pressing the delete key (DEL) followed by a "1" and then a "2". The participant then submits their response, concluding the trial.

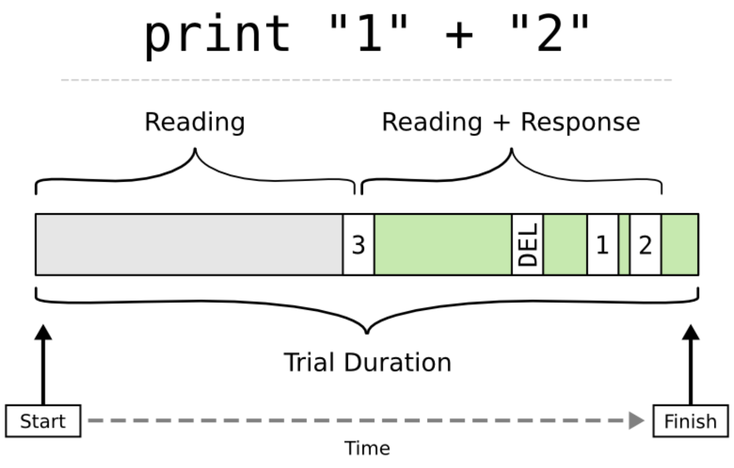


Figure 4: *Anatomy of an individual trial.*

The first performance metric we computed was the **output distance**. This is just the string edit distance [79] between the participant's response and the actual program output, normalized by the length of the longest string. An output distance of 0 means that the participant's response is a perfect match, while a distance of 1 means that all response characters should have been different. For the sample trial in Figure 4, the output distance would be 0 (a perfect score). We ignored whitespace, newlines, and certain formatting characters (square brackets, commas, quotes) when computing the output distance. This was intended to broaden the definition of a "correct" answer by ignoring characters that are only artifacts of the printing process. Out of the 1,575 trials, there were a total of 1,178 trials with correct answers (this number only drops to 1,001 if we require perfect whitespace and formatting characters).

The **duration** of each trial was recorded from start (participant is presented with the program) to finish (participant submitted their response) by the web server. This value was used as the denominator in the response proportion metric (described below). For outlier removal and when computing statistics, we used log duration. Especially within each program base, log duration distributions were approximately normal (Figure 6).

A **keystroke coefficient** was computed by dividing the total number of keystrokes a participant typed (including deletions) by the minimal number of keystrokes required to produce a perfect response. In the sample trial described above, the participant typed a total of four characters (3, DEL, 1, 2) when only two were required (1, 2). This trial, therefore, would have a keystroke coefficient of $4/2 = 2$. A keystroke coefficient less than 1 could be achieved in one of two ways: (1) by providing an incorrect response that was shorter than required, or (2) by copying and pasting text. We explore both of these possibilities in Section 5.

The **response proportion** of a trial is the amount of time between the first and last participant keystroke divided by the trial duration. Conceptually, this represents the proportion of a trial that was spent doing reading *and* responding rather than just reading (at the start) or just reviewing (at the end). In the sample trial, the response proportion would be approximately 0.5. Though there is a relationship between this metric and keystroke coefficient, it captures something different: intermediary results. Even if a participant types the exact number of necessary keystrokes (a coefficient of 1), their response proportion could be low (mostly reading, followed by a quick response) or high (little reading, slow construction of a response).

A **response correction** occurs when the participant's current response size decreases for the first time since the last increase. If we were to plot the number of characters in the participant's response over time, the number of response corrections would correspond to the number of troughs or downward slopes in the graph. While this metric will clearly be correlated with keystroke coefficient, it distinguishes between insertions and deletions. The sample trial only has a single response correction, and this would still be the case even if the participant had typed "33" instead of "3" initially.

Metric	Definition	Range/Units
Output distance	Normalized edit distance between correct answer and participant's output prediction	$[0, 1]$
Duration	Time taken to complete a trial (reading and response)	$[0, +\infty]$ log ms
Keystroke coefficient	Number of participant keystrokes divided by minimal required keystrokes for correct answer	$[0, +\infty]$
Response proportion	Amount of time between first and last keystroke divided by trial duration	$[0, 1]$
Response corrections	Number of times the participant's output decreased in size (backtracks)	$[0, +\infty]$ corrections

Table 2: *Performance metrics computed for each trial.*

4.4 Statistics

When comparing performance metric distributions, we use non-parametric statistical methods. Many of these metric distributions do not fit the assumptions of traditional parametric methods (e.g., ANOVA, t-test), so we use their non-parametric analogues. The Kruskal-Wallis H test serves our analogue of the ANOVA, with pairwise follow-up tests being done with the Mann-Whitney U test – our analogue of the t-test [86]. These pairwise U tests include a Bonferroni correction with a significance level (α) of 0.05. We calculate effect sizes for U tests using the rank-biserial correlation r, a metric whose range is $[-1, 1]$ with 0 meaning no correlation [103]. As a rule of thumb, we take absolute values of r greater than or equal to 0.2 to indicate a meaningful relationship (and $|r| > 0.4$ as a strong relationship). To quantify correlations between two series, we use the Spearman r correlation with a significance level (α) of 0.05. We infer weak, moderate, strong, and very strong relationships when $|r|$ is greater than or equal to 0.2, 0.3, 0.4, and 0.7 respectively.

For models fits, we use three standard approaches. When testing how well a fixed set of predictors predict a response variable, we use either an ordinary least squares (OLS) or logistic fit, depending on whether or not the response variable is binary. In both cases, a significance level of 0.05 is used to ascertain the significance of predictor coefficients. In Section 5.1.6, a LASSO-LARS technique with cross-validation is used to find a "best" fitting model across a set of many predictor variables [34]. This technique avoids significance-testing problems associated with alternative model selection methods, such as stepwise refinement and AIC/BIC ranking. Instead of using p-values, the coefficients of predictors are "shrunk" towards zero if they are not useful when predicting the response variable.

5 Part 1: Results

The results of our experiment are described in detail below. We start by observing the distribution of each performance metric, grouped by program base (Section 5.1). Next, we analyze the results by individual program 5.2. Finally, we consider how performance varies between correct and incorrect trials, as well as between expert and non-expert participants (Section 5.3).

5.1 By Performance Metric

We begin by discussing results at a high level across all trials. For each performance metric, we group results by program base and look for significant patterns. Some of these patterns are intuitively obvious, such as programs with more lines taking longer to read and respond to (Section 5.1.2). Other patterns are not so obvious, like finding trials where participants managed to get the right answer without typing all the necessary characters (Section 5.1.3)!
Below, we analyze each performance metric separately. Section 5.1.1 looks at the normalized output distance between trial responses and the correct response. Section 5.1.2 examines trial duration in log space. Our three keystroke metrics, keystroke coefficient, response proportion, and response corrections are then covered in Sections 5.1.3, 5.1.4, and 5.1.5 respectively. Lastly, we search for correlations between code complexity metrics, participant demographics, and performance metrics across all trials.

5.1.1 Output Distance

In general, participants performed well in their predictions of program output. About 75% of the individual trials results in a "correct" answer – i.e., a match with the true program output when excluding formatting characters. Almost 64% of the trials had perfect output matches (including whitespace and formatting characters). Figure 5 shows the output distance distributions for all trials grouped by program base (see Section 4.3 for details). These distributions make it clear which programs were more difficult: `between`, `counting`, and `scope` stand out with more output distances above zero. The non-zero peaks, such as 0.4 in `counting`, correspond to common errors made participants. Section 5.2 unpacks the results for individual programs, including which errors were most common.

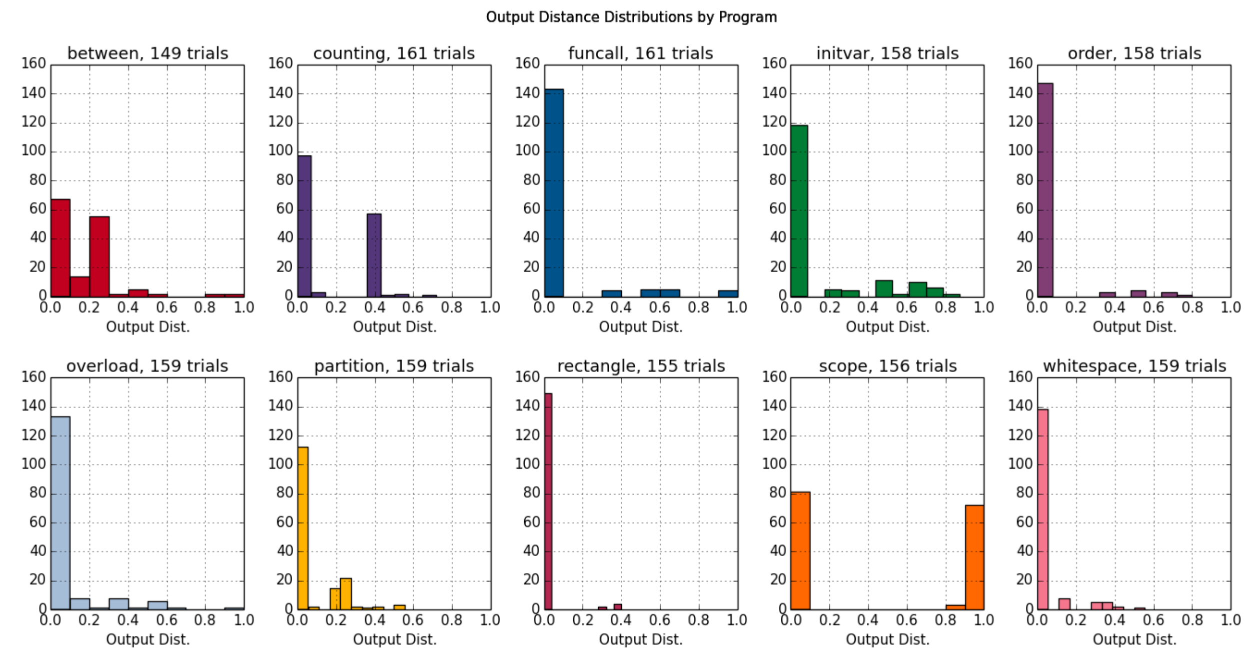


Figure 5: *Normalized output distance distributions by program base (all trials).*

5.1.2 Trial Duration

The median trial duration was 55 seconds, with the full range being 12 to 518 seconds (about 8 1/2 minutes). If we split the trials into those with a correct response and those without, the median trial durations become 55 and 50 seconds respectively. This difference, however, is not statistically significant. When trials durations are grouped by program base and transformed to log space, however, we can see some import timing differences (Figure 6). Many of these visual differences are statistically significant as well. Using an all-pairs Mann-Whitney U test (with a Bonferroni correction, $\alpha = 0.05$), most of the durations distributions differ (Figure 7). The `between`, `rectangle`, and `whitespace` programs stand out here – their duration distributions are different from all others (including each other).

There is a very strong correlation between mean trial duration (again, grouped by program base), and the number of lines in those programs ($r(10) = 0.75$, $p < .05$). This is not terribly surprising; we expect longer programs to take more time on average to read. What *is* surprising, however, is the lack of a significant correlation between mean trial duration and cyclomatic complexity (CC), a common measure of program complexity. So while longer programs took longer to read/response to on average, more *complex* programs (CC-wise) did not. If we use Halstead effort as our measure

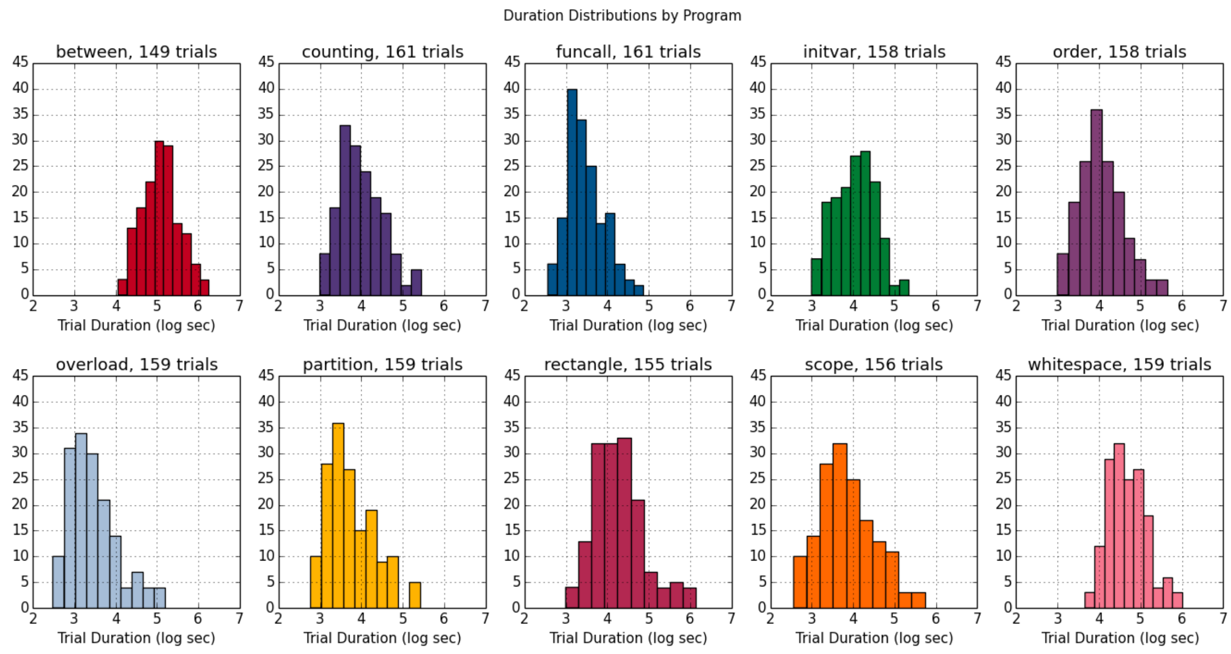


Figure 6: *Duration distributions by program base (all trials).*

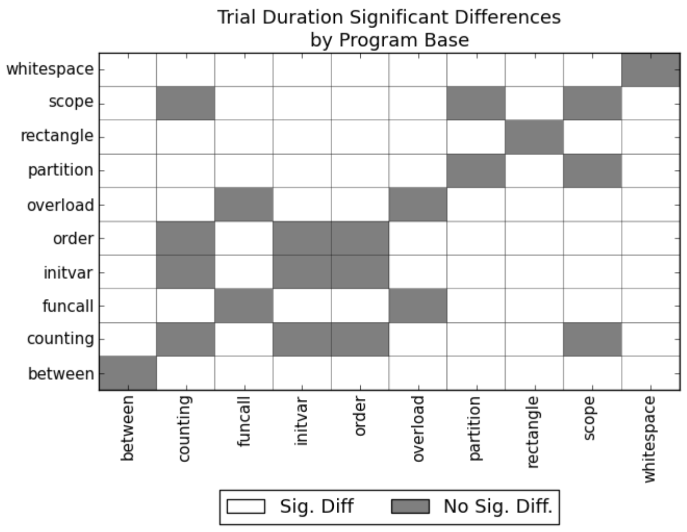


Figure 7: *Significant differences between trial durations grouped by base. Dark cells indicate* ***no*** *significant difference.*

of complexity, however, we again find a very strong correlation with mean trial duration ($r(10) = 0.78, p < .01$).

5.1.3 Keystroke Coefficient

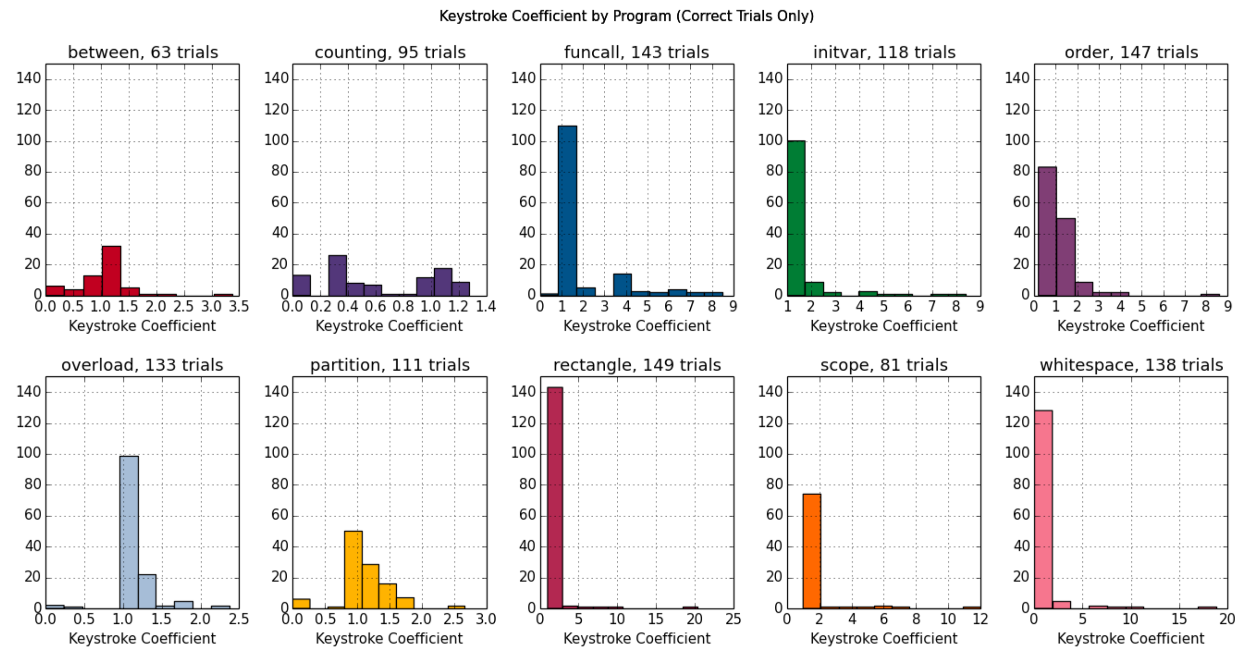


Figure 8: *Keystroke coefficient distributions by program base (correct trials only).*

Analyzing the distributions of keystroke coefficients provided insight into *how* participants were responding, rather than just their final answer (Figure 8). In most correct trials, participants typed only as much as necessary or just a few keystrokes extra (median coefficient was 1, mean 1.4). We were surprised to see keystroke coefficients less than 1 for a few correct trials; this would mean that a participant managed to type **fewer** keystrokes than necessary, yet still produce the correct response! Fortunately, there is a simple explanation: copy and paste.

For the `counting` programs especially, we found that participants were copying and pasting portions of their response (`counting` has a lot of redundant text in the correct response), allowing them to achieve a correct answer without physically typing all the necessary keystrokes. There were a handful of trials in other program types with a keystroke coefficient less than 1, leading us to suspect that one or two participants were typing each program into a Python interpreter and pasting the results into the output box [3]. These trials made up less than half a percent of our dataset, however, so we do no consider them a thread to the validity of our results.

5.1.4 Response Proportion

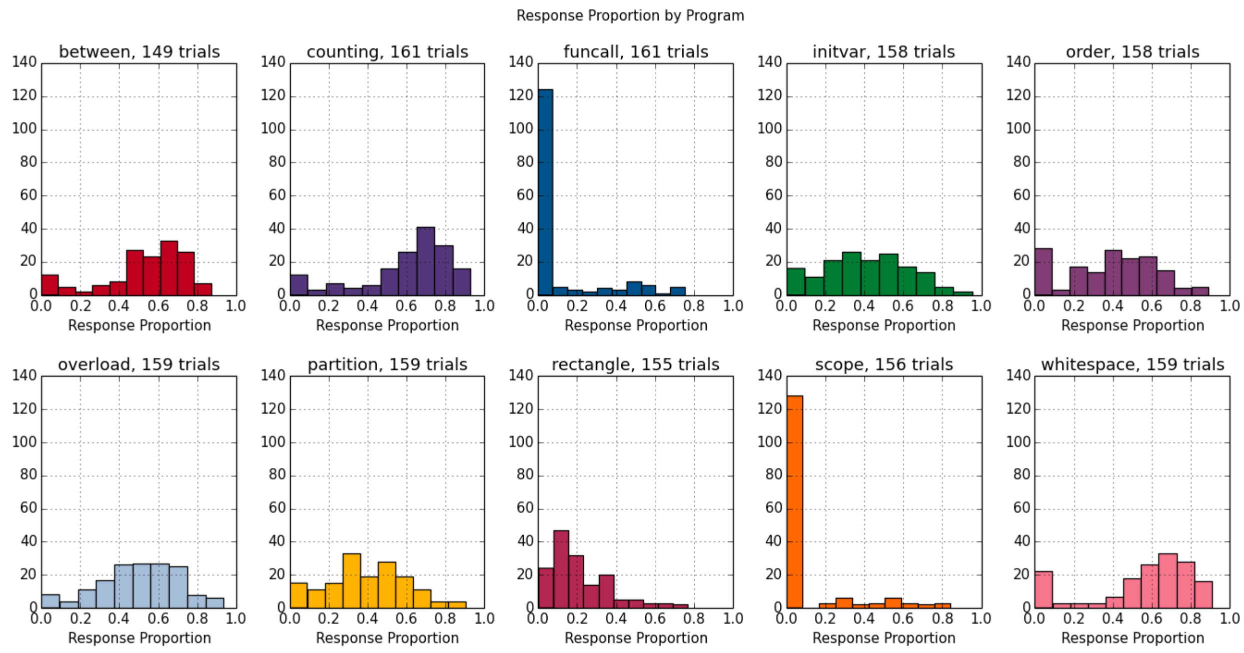


Figure 9: *Response proportion distributions by program base (all trials).*

The response proportion distributions in Figure 9 show how much of the participants' trials were spent responding. Intuitively, we might expect response proportion to increase with the size of the correct response – i.e., programs with longer outputs may take proportionally longer to respond to. An alternative hypothesis, however, is that participants would spend more time up front reading programs with longer outputs (perhaps because they are more complex in some sense), thus maintaining a constant response proportion relative to output size. In this case, the data match intuition: mean response proportion (grouped by program version) is strongly correlated with program output size ($r(25) = 0.54, p < .01$).

Surprisingly, we do not find a correlation between response proportion and the number of lines in a program (or any of our source code complexity metrics). We expected more complex programs, as measured by lines of code, cyclomatic complexity, or Halstead effort, to have higher response

[3]We presented our programs as images rather than text to discourage this behavior.

proportions because of the potential additional mental load on the programmer. A higher response proportion would mean that the participant started typing their output early in the trial, and continued typing late in the trial (presumably to offload mental effort). In Figure 9, we can see two programs that fit this expectation: `between` and `whitespace` both have right-skewed distributions *and* are relatively long programs. But `counting` and `rectangle` buck the trend. The `counting` programs are tiny with lots of output, and the `rectangle` programs are long with little output. Their distributions are also right and left-skewed, respectively – the opposite of what we might expect from program size alone.

5.1.5 Response Corrections

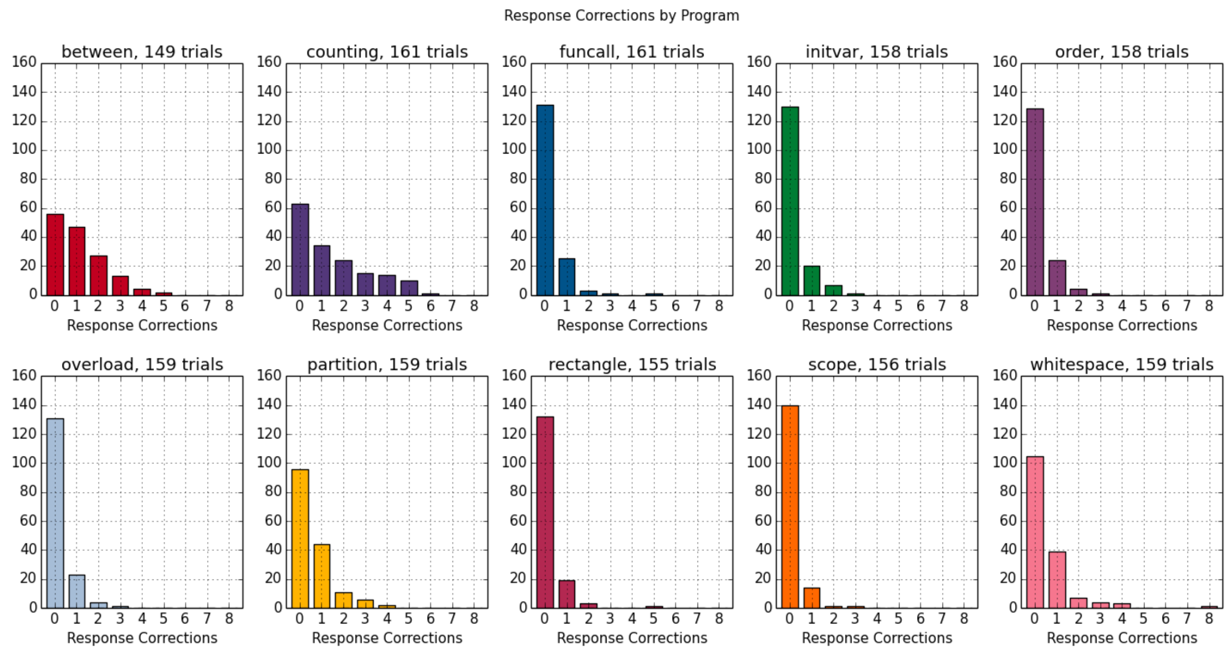


Figure 10: *Response correction distributions by program base (all trials).*

Not all keystrokes are created equal, and Figure 10 shows that some programs resulted in more corrections (or backtracks in response size) than others. The `between` and `counting` programs stand out here, with median response corrections of 1 (all others had medians of 0). This can be largely explained by the intuition that having more characters to type will simply result in more mistakes. And indeed, we find a very strong correlation between mean response corrections (grouped by program) and the number of characters in the correct output ($r(10) = 0.90, p < .001$).

Across all trials, we found a strong positive correlation between response proportion and the number of response corrections ($r(1575) = 0.46, p < .001$). This suggests that when participants spent more of their trial time responding, that time was spent making corrections (as opposed to carefully building their responses piece by piece). We also found weak position correlations between keystroke coefficient/trial duration and response corrections ($r(1575) \approx 0.2, p < .001$). These correlations further support the idea that participants who took longer did so to correct mistakes rather than read the code more thoroughly.

5.1.6 Complexity Metrics and Demographics

Can a participant's performance be predicted by a combination of code complexity/performance metrics and demographics? In this section, we describe several simple models for predicting performance on *individual trials*. These are not intended as **cognitive** models, which would describe the inner mental processes of our participants. Rather, they are **descriptive** models, relating predictor and response variables mathematically. We start by examining correlations between pairs of metrics, both complexity and performance. Next, we use a LASSO-LARS and cross-validation technique to identify the strongest predictors of each performance metric (see Section 4.4 for details). Finally, we focus on the normalized correct output distance and the binary "correct/incorrect" metric for each trial, including the remaining performance metrics as predictors in our models.

Figure 11 shows a Spearman correlation matrix between complexity and performance metrics across all 1,575 trials. Only significant correlations **above threshold** have displayed numeric values. Significance was determined via resampling: we computed the correlation matrix for 1,000 randomly shuffled versions of the data and rejected the null hypothesis for each cell if its correlation was outside the 95th percentile of the corresponding shuffled correlation distribution. Coefficients were also thresholded, with absolute values required to be greater than or equal to 0.2 (our threshold for a weak correlation).

Although the correlation matrix is intended for comparing complexity and performance metrics, we can also see how strongly correlated some of the complexity metrics are with each other. Specifically, lines of code (CodeLn) and Halstead effort (HalEff) are very strongly correlated with

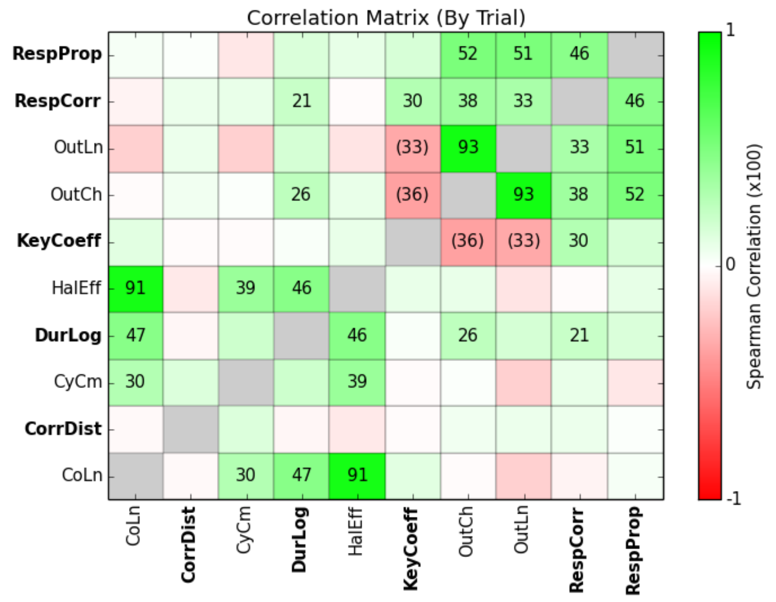


Figure 11: *Spearman correlation matrices between code complexity and performance metrics (bolded). From left to right, code characters (CodeCh), lines of code (CodeLn), correct output distance (CorrDist), cyclomatic complexity (CyCm), log duration in milliseconds (DurLog), Halstead effort (HalEff), keystroke coefficient (KeyCoeff), output characters (OutCh), output lines (OutLn), response corrections (RespCorr), and response proportion (RespProp).*

each other, as are output characters (OutCh) and output lines (OutLn). Cyclomatic complexity (CyCm), however, is only moderately correlated with lines of code ($r(1575) = 0.30$). These correlations are not surprising in and of themselves, but they do reveal important patterns in our programs. First, as our programs get longer, they tend to introduce more variables (Halstead operands) but not many more branches (as measured by cyclomatic complexity). Second, programs with more `print` statements (output lines) also output more characters. This correlation could have been reversed or non-existent had we combined `print` statements at the end of the program (e.g., `print x, y, z` as opposed three separate `print` statements). When interpreting the models below, it's important to keep these patterns in mind.

Another interesting feature of Figure 11 is that the keystroke performance metrics (response proportion, response corrections, keystroke coefficient) are moderately to strongly correlated with the keystroke complexity metrics (output chars/lines), while the log trial duration is strongly correlated with code complexity metrics (lines of code, Halstead effort). This suggests that trial duration may be driven more by the length of the program and number of unique variables than the size of the program's output. Surprisingly, we find no significant correlations between correct output distance and the other metrics! This does not mean that a participant's response correctness cannot be predicted, however. We turn our attention next to slightly more complex models, involving multiple predictors.

Multiple Predictor Models

Most of our performance metrics (excluding correct output distance) are correlated with one or more complexity metrics. But do these metrics equally contribute to performance? Additionally, can we improve predictions by including participant demographics, specifically *age*, years of *Python experience*, and years of *programming experience*? In this section, we examine performance models with multiple predictors.

Figure 12 shows the results of our best model fits, broken down by performance metric. Recall that these are LASSO-LARS fits, so predictors with coefficients of zero have been eliminated from the model. Except for keystroke coefficient and response proportion, the results are somewhat disappointing. The coefficients for all other models are quite small given the range of the corresponding performance metrics (Table 2). Correct output distance ranges from 0 to 1, making the largest coefficient (output lines at 0.06) only a minor contributor. Likewise, log trial duration peaks at 13.15 in our dataset, representing approximately 8 and a half minutes, while the largest coefficient (CyCm - cyclomatic complexity) represents an effect of about 1 and a half **milliseconds**. Response corrections ranges from 0 to 8 in the data, whereas all of the corresponding model coefficients combined do not even reach 1 (a full correction).

Only keystroke coefficient and response proportion have coefficients with meaningful values relative to their metrics' ranges. In both cases, the number of output lines (OutLn) is the most significant predictor. For keystroke coefficient, more output lines predicts a lower metric value, meaning that participants typed fewer unnecessary characters when there were fewer `print` statements. Note that output characters (OutCh) is not a very significant predictor even though it and keystroke coefficient are moderately correlated by themselves (Figure 11). It appears, then, that participant behavior is being driven by the number of `print` statements, rather than the number of characters they need to type.

Response proportion is also best predicted by output lines, but the relationship is positive. This result has a fairly straightforward explanation – participants do not generally read the entire program before beginning to evaluate it and respond. Thus, participants in trials whose programs

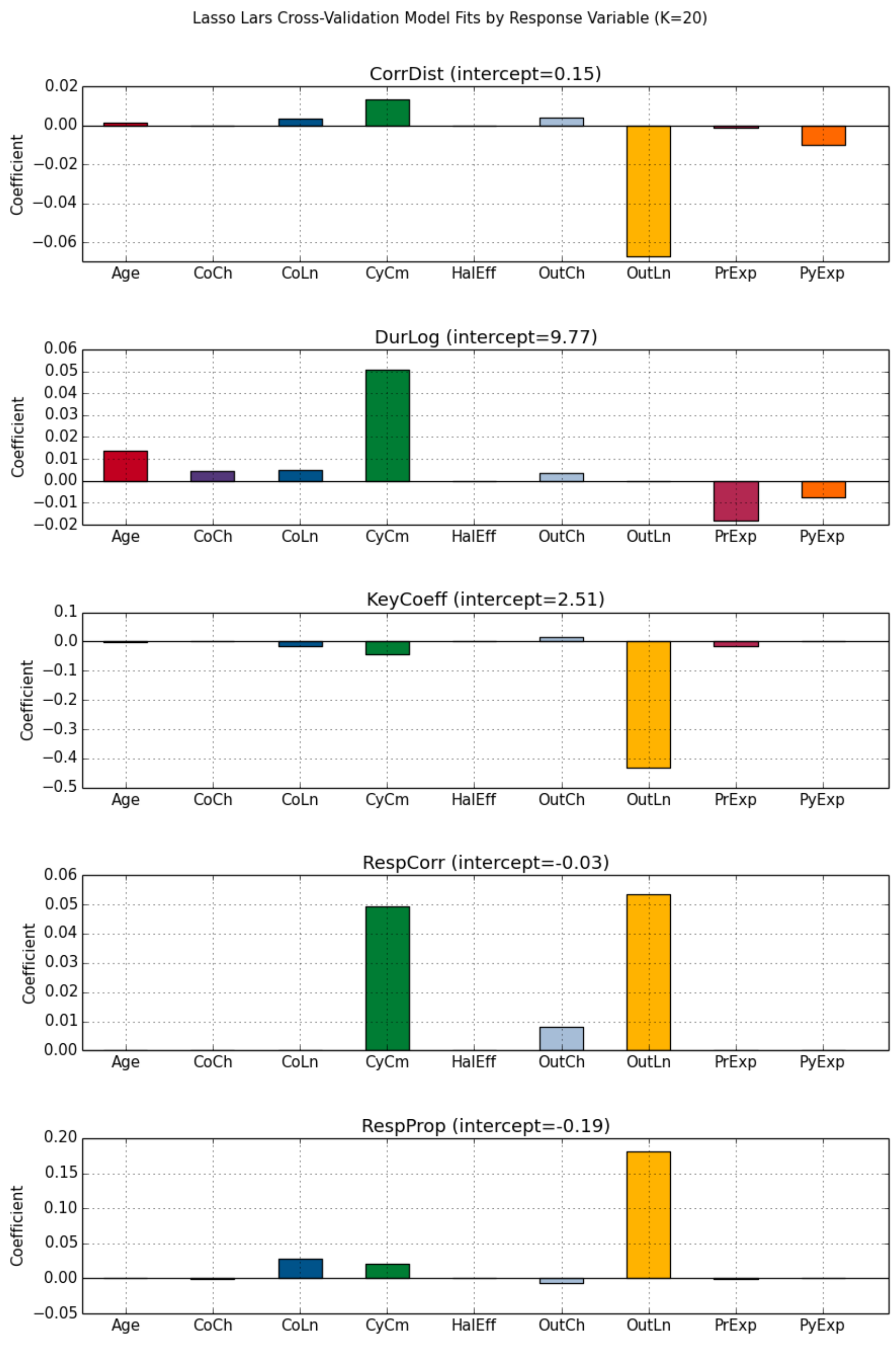


Figure 12: *Coefficients for best-fitting models by performance metric. All models were fit with LASSO-LARS and 20-fold cross validation.*

have more `print` statements (output lines) tend to start responding earlier. A more detailed analysis of individual participants may reveal more nuanced strategies, but evaluating `print` statements as they are read appears to be the dominant one.

Overall, we found that most our performance metrics could **not** be strongly predicted simply by code complexity metrics and participant demographics. In the next section, we focus on correct output distance. This performance metric is unique, as it is computed *after* the participant's response has been submitted. Thus, in addition to complexity metrics and demographics, we also consider other performance metrics as predictors.

Correct Output Distance Models

In this section, we consider models that predict a trial's normalized correct output distance using the corresponding program's code complexity metrics, the participant's demographics, and the *other performance metrics* from the trial. Figure 13 shows the coefficients for a LASSO-LARS model fit. This differs in several important ways from the CorrDist fit in Figure 12. First, the coefficients are larger (correct distance ranges from 0 to 1), giving us more confidence in the real-world power of the model. Second, it's clear that the other performance metrics dominate the model; especially response proportion. In general, it appears that trials with a larger response proportion and longer duration had a more correct response (lower is better for CorrDist), whereas those with a larger keystroke coefficient had less correct responses. This may be because participants that both took more time in a trial, and spent that time slowly constructing their response, did better. If that time was spent typing additional keystrokes – increasing the keystroke coefficient – then the additional characters were likely mistakes.

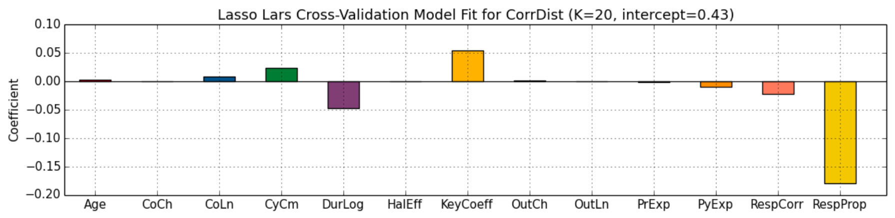


Figure 13: *Coefficients for LASSO-LARS model predicting normalized correct output distance.*

Modeling how close a participant's response was to being correct may be more fine grained than necessary if we're not trying to predict *how* correct they were. Can we get better predictions if we simply ask whether or not a correct response was given? For this (binary) prediction, we employ a set of classifiers from the popular scikit-learn library [69] trained on all trials using K-fold cross-validation ($K = 20$) to predict whether or not the trial response was correct. Figure 14 shows the **A**rea **U**nder the receiver operating characteristics (ROC) **C**urve (AUC) scores for six different classifiers trained on the same dataset. The AUC score is "the probability that the classifier will rank a randomly chosen positive instance higher than a randomly chosen negative instance." [38] An AUC score of 0.5 represents random guessing, and a score of 1.0 represents perfect classification.

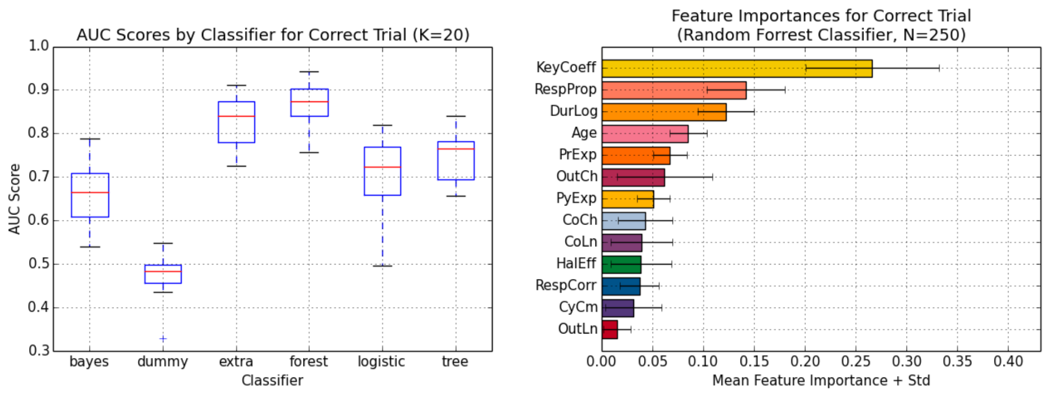


Figure 14: *Area Under the Curve (AUC) scores for $K = 20$ binary classifier runs predicting correct/incorrect trial (left). Feature importances and standard deviations for Random Forest classifier predicting correct/incorrect trial (right).*

The six classifiers in Figure 14 are Naive Bayes (bayes), a simple stratified baseline classifier (dummy), Extra Trees (extra), Random Forest (forest), Logistic Regression, and Decision Tree (tree). Table 3 provides more detailed descriptions of each classifier. The boxplots on the left in Figure 14 show the AUC score distributions over all 20 cross-validation runs. As expected, the "dummy" classifier hovers around 0.5, which is no better than random guessing. The other classifiers perform better on average than dummy, but the Random Forest (forest) classifier outperforms the others. This is an ensemble, or meta, classifier that constructs a "forest" of random decision trees which are each trained on sub-samples of the data. The combined predictions of all trained trees are then used to drive the forest's classification of new instances.

The right-hand plot in Figure 14 dives deeper into the Random Forest classifier results. The so-called "feature importances" are displayed in descending order, with the most important features for prediction accuracy on top. These importances are determined by training $N = 250$ trees on the data, and then compiling the relative importance of each feature across them all. As

Category	Classifier	Description
Naive Bayes	Gaussian	Bayes theorem with assumption of independence.
Dummy		Predicts based on training set's class distribution.
Ensemble	Extremely Randomized Trees	Random thresholds and candidate features.
Ensemble	Random Forest	Constructs classifiers using random features.
Linear	Logistic Regression	Standard logistic regression
Decision Tree		Infers if-then-else decision rules from data features.

Table 3: *Descriptions for all classifiers used to determine utility.*

with the LASSO-LARS model in Figure 13, the three most important predictors are response proportion, keystroke coefficient, and log trial duration. Here, however, keystroke coefficient is consistently ranked higher than the others. Because we are predicting a binary outcome (correct/incorrect) rather than the real-valued output distance, this suggests that keystroke coefficient has a useful, though not strongly linear, relationship with response correctness. It may be the case, for example, that a handful of extra characters is enough to guess that the participant will be submitting an incorrect response.

5.2 By Program Version

In this section, we analyze participant performance by program **version**. Each program **base** had 2-3 versions, one of which a participant randomly received. Below, we describe the code differences between and motivations for each program version. We discuss participants' most common errors, and contrast performance results between versions.

5.2.1 between

The `between` programs were intended to test the effects of pulled-out versus inline functionality (i.e., putting code into functions versus repeating it). In both versions, two lists are filtered and printed, and then the common elements in the original lists are printed. The `functions` version (Appendix A.1.1) contained two functions (between and common), corresponding to the filtering and intersection operations performed on the two lists. The `inline` version (Appendix A.1.2) did not contain any function definitions, repeating code instead.

We found that the `functions` version had a significantly higher median output distance

($U = 2265, p < .05$), though the effect size was small ($r = 0.16$). This means that participants in general gave slightly more correct responses on the `inline` version. We expected that having the filtering and common operations pulled out into reusable functions would benefit experienced programmers, who could quickly chunk the definitions and then interpret the main body of the program. The opposite appears to be the case, as there was a significant interaction between years of programming experience and program version ($F(3, 145) = 3.057, p < .05$), with experience hurting and `inline` helping. It may be that experienced programmers were more likely to make incorrect assumptions in `functions` version because the `between` and `common` operations were abstracted and physically distant. The `inline` version, however, may have facilitated a more literal reading of the program, as a novice was likely to do in both cases. Again, effect sizes were small, so these results may not generalize.

But how did expertise specifically impact response correctness? We found that more experienced programmers were more likely to make a very specific error, regardless of version. In approximately 41% of `between` trials, the response in Listing 1.

```
[8, 7, 9]
[1, 0, 8, 1]
[8]
```

Listing 1: *Incorrect response for `between` programs. The final line should be `[8, 9, 0]`.*

The last line is incorrect, and should be `[8, 9, 0]` instead. More experienced programmers gave this response more often, with a significant logistic regression coefficient of 0.25 ($p < .05$). An informal, post-experiment interview with one participant suggests a plausible explanation: the common error response is correct if we mistakenly assume that the common operation is performed on the *filtered lists* instead of the original lists (`x` and `y`). Experienced programmers may be expecting the high-level behavior of the program to be FILTER LIST, FILTER LIST, INTERSECT FILTERED LISTS rather than what the code really says: FILTER LIST, FILTER LIST, INTERSECT ORIGINAL LISTS. Our interviewee summed up the reason for their mistake: “why would you bother filtering the lists if you don’t use the results?” This was unexpected, and suggests that `between` may be better placed in the “expectation violations” category rather than “physical characteristics of notation”.

Less experienced programmers were expected to take more time on the `functions` than on the `inline` version due to the need to refer back to the functions often. We did not observe this in the data and, in fact, we did not find a significant difference in any of the other performance metrics between versions.

5.2.2 counting

The `counting` programs tested the effects of whitespace on the grouping of statements in a `for` loop. Python does not have a keyword or delimiter for the end of a block because indentation is mandatory. Both versions of `counting` did the same thing: print "The count is *i*" and "Done counting" for $i \in [1, 2, 3, 4]$. While the `nospace` version (Appendix A.2.1) had the `for` loop declaration and the two `print` statements in the body on consecutive lines, the `twospaces` version (Appendix A.2.2) added two blank lines between the first and second `print` statement. Because Python is sensitive to indentation, this did not change the semantics of the program (i.e., both `print` statements still belonged to the `for` loop).

We saw a stark contrast in correct responses between versions. Approximately 60% of the `twospaces` responses failed to group the second `print` statement ("Done counting") with the `for` loop. The effect is easily visible in the output distance distributions (Figure 15), with the peak around 0.4 in `twospaces` corresponding to these errors. Surprisingly, Python experience and overall programming experience did not have a significant effect on the likelihood of making this mistake!

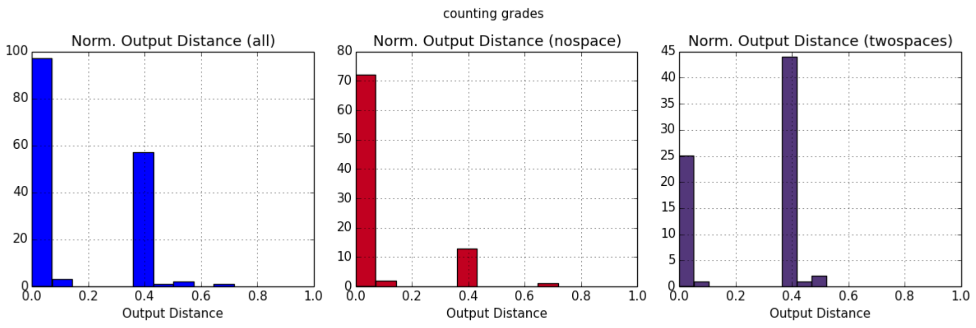


Figure 15: *Grade distributions for `counting` programs.*

Whitespace was not the only contributor to the incorrect grouping of the final `print` statement. Approximately 15% of responses in the `nospace` trial contained this same error, despite there being no whitespace between lines in the `for` loop. Even with no effects of physical notation, there is still an expectation violation in the code: the final `print` statement says "Done counting," but this text is repeated for every loop iteration in the correct output. By mentally moving the `print` statement outside of the loop body, the (incorrect) output makes much more sense:

```
The count is 1
The count is 2
The count is 3
The count is 4
Done counting
```

So we have two likely contributors to the large percentage of errors in the `twospaces` version: whitespace (physical notation) and non-sensical output (expectation violation). While we expected both, we did not expect to have no effect of experience (Python **and** overall programming). For such a simple program, this is quite surprising, and perhaps indicative of the need for an "end of block" keyword or delimiter in the language.

5.2.3 funcall

The `funcall` programs each performed a compound calculation using a single, simple function $f(x) = x + 4$ (Appendix A.3). The calculation, $f(1) \times f(0) \times f(-1)$, either had no whitespace between terms (`nospace` version), a single space between all tokens (`space`), or had each call to $f(x)$ bound to a variable before completing the calculation (`vars`). We expected to find an effect on trial duration and possibly calculation errors, with more whitespace facilitating faster and more correct trials. In the `vars` version especially, we expected having the calculation broken out into multiple, named steps (i.e., `x`, `y`, `z`) would ease participants' mental burden.
Surprisingly, we did not find a difference between any of the three versions for any of the performance metrics. Participants performed equally well despite differences in whitespace and the use of variables for intermediary calculation steps. Approximately 89% of trials were correct for the `funcall` programs, above the average of 75% for all 10 program bases. For the 11% of incorrect trials, the two most common responses were 0 and -60. We hypothesize that these responses correspond to the incorrect assumption by participants that $f(0) = 0$ and $f(-1) = -3$. Together,

though, these two types incorrect responses constituted only a half a percent of the total `funcall` trials, so they do not provide strong evidence for a generalizable pattern in the data.

5.2.4 initvar

The `initvar` programs each contained two accumulation loops: one performing a product, and the other performing a summation (Appendix A.4). In the `good` version, both loops were intended to meet expectations; the product loop had an initial value of 1, and the summation loop had an initial value of 0. The `onebad` version, however, started the summation loop at 1 (an off-by-one error). The `bothbad` version contained the same error, and also started the product loop at 0 (making the final product 0). We expected these "bad" versions to violate participant expectations, possibly more so in experienced programmers. More errors were expected in the `onebad` version relative to `good`, and even more errors were expected in the `bothbad` version.
Our expectations were violated by the actual results. We did not not observe a significant difference in output distance or likelihood of a correct response between program versions. In fact, the only significant difference between versions came from response proportion (Figure 16). After a Kruskal-Wallis H test ($H(2) = 16.72$, $p < .001$), a pair-wise Mann-Whitney U test (with a Bonferroni correction, $\alpha = 0.05$) revealed that both the `good` and `onebad` versions had significantly lower response proportions than `bothbad` (10-20% lower).

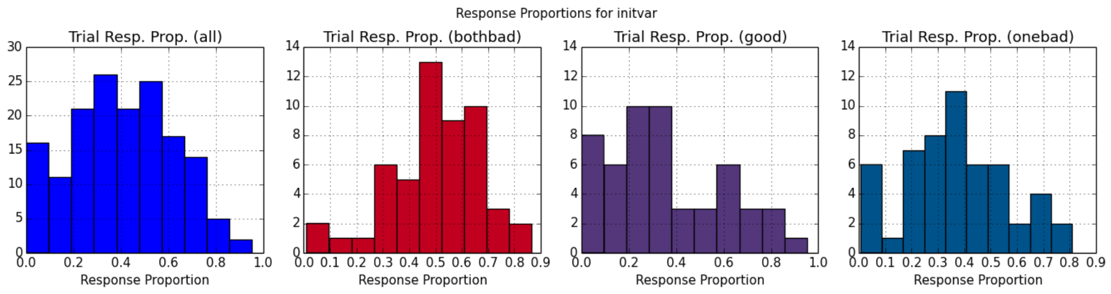


Figure 16: *Response proportion distributions for* `initvar` *programs.*

This result can partially be explained by participants quickly noticing that the first loop will result in *a* being 0, and short-circuiting the calculation. By typing a "0" early in the trial, the response proportion metric would be inflated, *assuming they spent more time elsewhere*. Indeed, we did not observe a significant difference in trial duration between versions, suggesting that `bothbad` participants took the same amount of time as others. The keystroke coefficient and response corrections metrics did not differ either, so `bothbad` participants did not spend their extra response time making corrections. Instead, the data suggest they either took longer on the second loop, or perhaps reviewed their quick response to the first loop before completing the trial. A more in-depth analysis of the individual keystrokes is needed to answer these questions.

5.2.5 order

The `order` programs contained three functions, $f(x)$, $g(x)$, and $h(x)$. While f and g added 4 and doubled x respectively, h computed $f(x) + g(x)$ (Appendix A.5). In the main body, all three functions were called on $x = 1$ in the same order: f, g, h. The order in which the three functions were defined was either in the called order (`inorder`), or in a slightly different order (`shuffled`) – h, f, g. With this simple manipulation of notation, we expected participants to exhibit a different in trial durations. The `shuffled` version had an incongruent definition and call order, so we expected participants to take longer on this version due to a less efficient visual search.

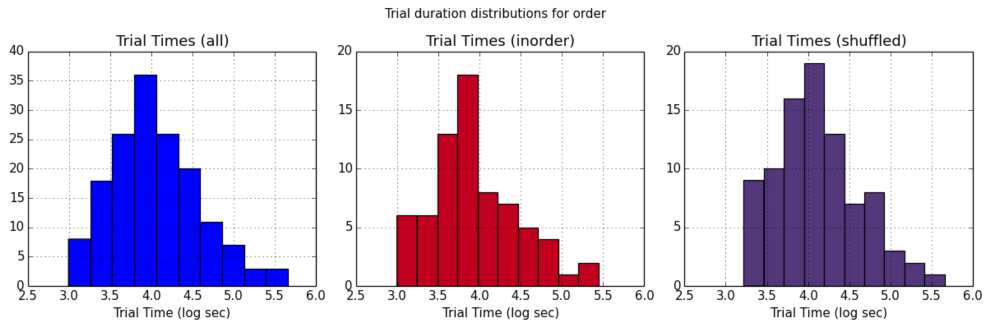


Figure 17: *Duration distributions for `order` programs.*

Our hypothesis was supported by the data, though the effect size was small (Figure 17). We observed a significant difference in log trial durations between the `inorder` and `shuffled` versions ($U = 2530.0$, $p < .05$). The difference in duration medians was fairly small (0.1 log seconds), as was the rank biserial correlation ($r = 0.18$). Despite the small effect size, it is still impressive to observe a significant result with such a small change to the code! We did not observe differences between versions in any other performance metric. Because approximately 93% of participants provided the correct response, we are confident the incongruent definition/call order produced an effect purely on visual search efficiency. An in-depth analysis of the collected eye-tracking data is needed to further support this hypothesis.

5.2.6 overload

The `overload` programs tested the effects of operator overloading (Appendix A.6). Each program had three blocks of code, each with two variable assignments followed by an operation with those two variables (and a printing of the result). The final block always assigned a string "5" and string "3" to variables `e` and `f`, and then printed `e + f`. The preceding two blocks either had exclusively multiplications (`multmixed`), additions (`plusmixed`), or string concatenations (`strings`). We expected `plusmixed` to give participants the most trouble, both in terms of error and additional time, because the `+` operator was used to mean both addition and concatenation. Additionally, we expected participants to be *faster* at the `strings` version because **no** numerical operations were present in the program, thus there should be no priming for the numeric overload of `+`.
The results surprised us. We saw no significant difference between versions in terms of errors or response proportion. Approximately 11% of responses incorrectly computed `5 + 3 = 8` for the final code block instead of `"5" + "3" = "53"`, but any difference between versions was not significant. We did, however, observe the same pattern across trial duration, keystroke coefficient, and response corrections: the `strings` version was significantly different from the `multmixed` version, but not `plusmixed`.
A Kruskal-Wallis test followed by a Mann-Whitney U test (with correction) showed that `strings` had a significantly higher log duration than `multmixed` ($U = 1095.5$, $p < .01$) with a decent effect size ($r = 0.31$). It's tempting to explain this simply by noting that the `strings` version requires more keystrokes to get the correct response than both of the others. However, the duration (and keystroke coefficient, response proportion) metrics are significantly different only between `strings` and `multmixed` (**not** `strings` and `plusmixed`, nor `multmixed` and `plusmixed`). So we can see indirectly that the use of the overloaded `+` in `plusmixed` has had an effect – unlike `multmixed`, this version is not significantly different than `strings`.
How did the use of an overloaded operator impact participants in `plusmixed`? This is difficult to answer precisely because the statistics only allow us to compare with the other versions. We know that `plusmixed` is not significantly different `strings` in terms of trial duration, even though the

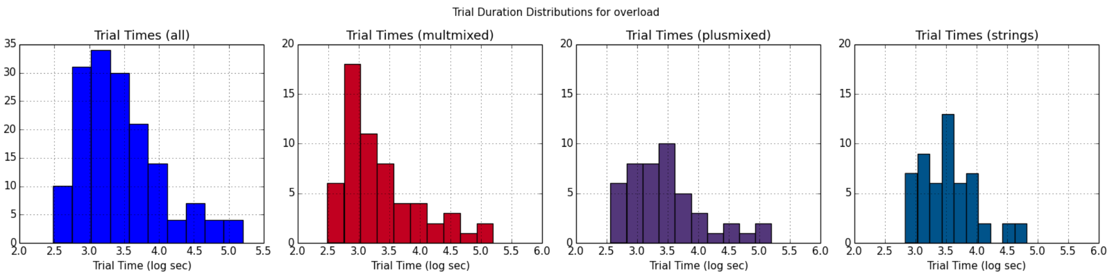


Figure 18: *Duration distributions for* `overload` *programs.*

only difference `multmixed` and `plusmixed` is the numeric operator used in the first two code blocks. This leads us to believe that participants were slowed down *slightly* in the `plusmixed` version, but *not enough* to be significantly different from `multmixed`. Indeed, the median log trial duration for `plusmixed` (10.29) is sandwiched between `multmixed` (9.99) and `strings` (10.43) – see Figure 18. We see the same pattern with keystroke coefficient and response corrections: `plusmixed` is between the other two, but not significantly different (while `multmixed` and `strings` are). Thus, operator overloading appears to decrease performance (excluding response errors), but not enough to completely differentiate it from a nearly identical program without overloaded operators.

5.2.7 partition

The `partition` programs each iterated through a list of numbers and printed the number plus a "high" or "low" designation on each line (Appendix A.7). Numbers less than 3 were *low*, and numbers greater than 3 were *high* (3 itself was skipped). The `balanced` version iterated over $[1, 2, 3, 4, 5]$, producing an equal number of *low* and *high* numbers. In contrast, the `unbalanced` and `unbalanced_pivot` versions iterated over $[1, 2, 3, 4]$, printing two *low* and only one `high`. The `unbalanced_pivot` version used variable `pivot = 3` instead of the constant 3 in its `if` statements. We expected participants to perform better (in terms of error rates) on the `balanced` version because of the symmetric *low* and *high* values. This symmetry was expected to aid in the recognition of this program's purpose: partitioning a list based on a pivot element. For the other versions, we expected `unbalanced` to have the highest error rates, with `unbalanced_pivot` having slightly fewer errors due to the pivot element being explicitly named.

We were surprised to find no significant differences between versions across **all performance metrics**. Approximately 70% of responses were correct, and we did not observe a systematic difference in errors between versions. An ordinary least squares regression did show a very small, but significant, effect of programming experience on output distance ($F(1,157) = 5.07, p < .05$). The coefficient was negative (-0.0038), indicating that more experience decreased errors, but is far too small to make any broad claims.

The most common error across all versions was leaving the number off of each line (i.e., assuming `print i, "XXX"` was just `print "XXX"`). This error occurred in approximately 19% of responses, but did not seem to occur more in any particular version. Around 7% of responses included an extraneous line for 3, but there was not a significant bias for any version or whether this number was marked *high* or *low*.

5.2.8 rectangle

The `rectangle` programs (Appendix A.8) computed the area of two rectangles, represented either as a collection of four x/y variables (`basic`), a `Rectangle` object (`class`), or a pair of `(x, y)` coordinates (`tuples`). We expected the `class` version to take the most time because it is longer and significantly more complex according to our metrics (Figure 19). The `basic` version was expected to take the least amount of time due to its use of simple variables for rectangle representation.

Version	LOC	CC	HE
basic	18	2	18801
class	21	5	43203
tuples	14	2	15627

Figure 19: *Complexity metrics for `rectangle` program versions. Metrics are lines of code (LOC), cyclomatic complexity (CC), and Halstead effort (HE).*

Interestingly, we did not observe a different between versions for any of the performance metrics. In fact, this set of programs had the highest percentage of correct answers (about 96%) of any other program base. This is surprising from a complexity metrics perspective, given that the `rectangle` programs were some of the longest ones in our entire set (see Table 21 in the Appendix). These results are unsurprising, however, when we consider our participant demographics: programmers with a mean age of 28. Our participants have all likely had experience with geometry, and could therefore leverage a large amount of domain knowledge when evaluating the `rectangle` computations. In retrospect, we should have introduced a *calculation bug* into these programs' area functions and seen if there was a 96% error rate instead! As it stands, `rectangle` demonstrates how domain knowledge can overwhelm differences in representation and notation.

5.2.9 scope

The `scope` programs applied two functions to a variable named `added`: one named `add_1`, and the other named `twice` (Appendix A.9). Both of these functions produced no visible effects – they did not actually modify their arguments or return a value. In the `samename` version, we reused the variable name `added` for each function's parameter name. For the `diffname` version, however, we used a different parameter name for both (`num`). Because the `add_1` and `twice` functions had no effect, the main `added` variable retained its initial value of 4 throughout the program (instead of being 22). This directly violates one of Soloway's Rules of Discourse [89]: do not include code that will not be used.

We expected participants to mistakenly assume that the value of `added` was changed more often when the parameter names of `add_1` and `twice` were both also named `added` (i.e., in the `samename` version). The actual results were much more interesting: around 48% of responses were incorrect, regardless of version! There were no significant differences between versions across all performance metrics, but we did observe an effect of experience. A logistic regression predicting a correct response from years of Python experience yielded a significant effect (intercept = -0.37, OR = 1.25, $p < .05$). Because the odds ratio (OR) is greater than 1, we can infer that more Python experience helped increase the odds of a correct response. This same effect was **not** observed for overall programming experience, however.

Although anecdotal, our experience with participants in the eye-tracker may help explain *why* Python experience, and not overall programming experience, aided participants. While evaluating one of the `scope` programs, several participants paused and asked the experimenter (who was behind a partition) whether the Python language was "call by value" or "call by reference." In short, this is the difference between `add_1(x)` only being able to modify a *copy* of `x` (call by value) or the original `x` (call by reference). Python exhibits both behaviors, depending on what `x` contains. In the case of the `scope` programs, `added` was an integer, so neither function *could possibly* modify it[4].

[4]Were `added` to be list or dictionary, then that list or dictionary could be modified by a function. Which object the `added` variable points to, however, could not.

Python's "call by" behavior is not unusual with respect to most commonly-used programming languages. We hypothesize that participants had strong expectations against seeing unused code (Soloway's rule), but also recognized that the functions did not return values. Thus, they had to resolve the conflict by appealing to their knowledge of the Python language itself. The fact that almost half the responses were incorrectly 22, however, demonstrates the power of Soloway's rule. If our hypothesis is correct, many highly-experienced programmers were more willing to bend the language's rules than accept code that does nothing.

5.2.10 whitespace

The `whitespace` programs print the results of three simple linear calculations (Appendix A.10). In the `zigzag` version, the code is laid out with one space between every mathematical operation, so that the line endings have a typical "zig-zag" appearance. The `linedup` version, in contrast, aligns each block of code by its mathematical operators, nicely lining up all identifiers. We expected there to be a speed difference between the two versions, with participants being faster in the `linedup` version. When designing the experiment, most of our pilot participants agreed that this version facilitated reading, but the data did not support this claim.

Approximately 87% of responses were correct, and there were no performance differences between versions. Using an ordinary least squares regression, we did find a significant effect of years of Python experience on log trial duration ($F(1, 157) = 5.247, p < .05$). The coefficient ($-0.043$) is very small, though, even for the log millisecond scale. It is also not terribly surprising that more experienced Python programmers were a bit faster in general.

When inspecting the kinds of errors participants made (only about 13% of responses), we noted that a specific kind of error only occurred in the `zigzag` version. Five participants (about 6% of `zigzag` responses) answered 10 and 15 for the last two y values rather than the correct values of 6 and 11[5]. These are the answers that would be obtained if a participant executed the multiplications before the additions, contra the established of order of operations of Python and mathematics more generally. Effects of spacing on the perceived order of arithmetic operations have been studied before [43], and our results suggest that spacing in code layout may also have an impact on order

[5]Confusingly, two of these participants provided the correct first line (0 1), while the other three were consistently wrong with (0 5).

of executed operations.

5.3 By Experience and Correctness

How does performance vary with expertise, and whether or not the response was correct? We analyzed all trials grouped by the correctness of the response, and separately by whether or not the participant was considered an **expert**. Expertise was defined as having 5 or more years of Python experience, or having 10 or more years of overall programming experience. In general, we expected experts to perform better across individual performance metrics (fewer errors, lower trial durations, etc.). When looking at correct versus incorrect trials, we expected to see distinct differences in *groups* of performance metrics – e.g., longer trial durations, but fewer response corrections.

5.3.1 Correct/Incorrect Trials

While the majority of trials had a correct response (approximately 75%), there were still significant performance differences between correct and incorrect trials. Trial durations did not significantly differ, but we did observe the following differences in other performance metrics:

- **Keystroke Coefficient** - correct trials had significantly *smaller* keystroke coefficients ($U = 216,622.5, p < .05, r = 0.073$).
- **Response Proportion** - correct trials had significantly *smaller* response proportions ($U = 216,770.5, p < .05, r = 0.073$).
- **Response Corrections** - correct trials had significantly *fewer* response corrections ($U = 208,360.0, p < .01, r = 0.11$).

From the above observations, we could infer that participants providing a correct response tended to spend more time reading before responding (smaller response proportion), and less time fixing mistakes (fewer response corrections, smaller keystroke coefficient). Unfortunately, the effect sizes (r) were below our threshold of 0.2 for a meaningful relationship (see Section 4.4), so our confidence in the real-world impact of these inferences is reduced.

5.3.2 Expert/Non-Expert Trials

We divided our participants into **experts** and **non-experts** depending on their Python and overall programming experience. Participants with 5+ years of Python experience or 10+ years of programming experience were considered to be experts *in the scope of our task*. Given that our programs did not stray from material covered in a first year Python course, we are confident in this classification. We had 52 expert participants and 110 non-expert participants in the experiment, and observed the following performance metric differences between expert and non-expert trials:

- **Output Distance** - expert trials had significantly *lower* correct output distances ($U = 255,331.0, p < .01, r = 0.06$).
- **Log Duration** - expert trials had significantly *lower* log trial durations ($U = 238,531.5, p < .001, r = 0.12$).

These observations were surprising to us for several reasons. First, as with the correct/incorrect trial performance differences, the effect sizes were below our threshold for a meaningful relationship (0.2). With such strong conditions for being considered an expert, we expected to observe much stronger performance differences. Second, we did not observe any significant differences in the other performance metrics between expert and non-expert trials (keystroke coefficient, response proportion, response corrections). We hypothesize that this is due to (1) the simple nature of our programs in general, and (2) looking at all trials together rather than separated by program base/version.

When comparing trials by program version, we *do* observe strong differences due to Python/programming experience (Section 5.2). Across all trials however, the effect is likely diminished due to "easy" programs on which experts and non-experts perform similarly. Indeed, half of the programs bases had over 75% correct responses (`overload`, `whitespace`, `funcall`, `order`, and `rectangle`). With more difficult programs, we would expect to observe much stronger distinctions between experts and non-experts *across all trials*.

6 Part 1: Discussion

In Section 5, we analyzed trials in the eyeCode experiment by grouping them in various ways, and then comparing performance metrics between groups. We analyzed trials grouped by program base (Section 5.1), by program version (Section 5.2), and by response correctness/participant expertise (Section 5.3). Below, we relate these results to each of our **three research questions**:

- **RQ1**: How are programmers affected by programs that violate their expectations, and does this vary with expertise?
- **RQ2**: How are programmers influenced by physical characteristics of notation, and does this vary with expertise?
- **RQ3**: Can code complexity metrics and programmer demographics be used to predict task performance?

6.1 Expectation Violations

We hypothesized that expectation-violating programs would result in slower response times and higher error rates, perhaps more-so for experience programmers. Specifically, we expected the following program versions to violate participant expectations:

base	**version(s)**	**violation**
`initvar`	`onebad`, `bothbad`	Wrong starting value and off-by-one error.
`overload`	`plusmixed`	Plus operator used for addition and concatenation.
`partition`	`unbalanced`, `unbalanced_pivot`	Unequal number of "low" and "high" outputs.
`scope`	`diffname`, `samename`	Included code does not produce any effect.

Of these, only `scope` produced an expected effect (high error rates). This was especially interesting because some programmers reported *questioning the language semantics* rather than accepting a program with (effectively) useless code in it (see Section 5.2.9 for details). As predicted by previous research [89], this effect was modulated by experience; more experienced Python programmers less likely to make the mistake. A similar effect was observed in `partition` (programming experience

reduced errors), but there was no difference between versions. A more dramatic effect of experience – specifically domain knowledge – was seen with all three `rectangle` programs. Despite notational differences, we did not observe any difference between versions. In fact, the `rectangle` programs had the highest percentage of correct responses for any program base (96%); a testament to the utility of domain knowledge when the code and domain are congruent[6].
It's not unusual to see experience positively correlated with performance, but this is not a necessary relationship. For example, the most common error made on the `between` programs was significantly more likely to occur for more experienced participants (Section 5.2.1). Distortions of form and content during the recall of programs by experienced participants have been observed in previous experiments [27]. With `between`, experience was likely correlated with a distortion of *content*; participants appeared to ignore the code in front of them and do what "made sense" instead. Something similar may have occurred during trials with the `nospace` version of `counting`. Despite being a perfect example of a Python `for` loop, a minority of responses (15%) contained only a single "Done counting" line at the end. As with `between`, this indicates that participants interpreted the program's *intention* rather than its literal *code*.

6.2 Physical Notation

The physical aspects of notation, often considered superficial, can have a meaningful impact on performance. We expected to see differences in trial duration due to notational effects (except for `counting`). Specifically, the following program versions were expected to produce notational effects:

[6]We expect that introducing a calculation error into the `area` function would strongly increase error rates.

base	**version(s)**	**notation**
`between`	`functions`	Filtering/intersection operations pulled out into functions.
`counting`	`twospaces`	Extra vertical whitespace between loop body statements.
`order`	`shuffled`	Incongruent function definition and call order.
`rectangle`	`basic`, `tuples`, `class`	Different data structure representations for rectangle.
`whitespace`	`linedup`	Code is horizontally aligned by mathematical operators.

The `twospaces` version of `counting` demonstrated that vertical space is more important then indentation to programmers when judging whether or not statements belong to the same loop body. Programmers often group blocks of related statements together using vertical whitespace, but our results indicate that this seemingly superficial space can cause even experienced programmers to internalize the wrong program. As mentioned in the previous section, at least some of this effect can be attributed to an expectation violation due to the words "Done counting" in the final `print` statement. Indeed, our `counting` results could be seen as additional evidence that high-level expectations drive performance more than notation. The `rectangle` programs are another potential example – we did not observe any performance differences between versions, most likely due to the overwhelming influence of domain knowledge.

We observed small notational effects on trial duration in the `order` and `whitespace` programs. Participants were slowed down in the `shuffled` version of `order`, providing evidence of our hypothesis that the congruence of function definition and calling order would aid in visual search. Python experience helped participants on both versions of `whitespace`, but the `zigzag` version contained a handful of incorrect responses indicative of the wrong order of operations. While this result was not statistically significant, it suggests a path for future work in the study of notational effects in programming. We expect to see an overlap with research on how physical spacing influences the perceived order of operations in arithmetic [43].

6.3 Predicting Performance

Our final research question asked if trial performance could be predicted using complexity metrics from the program's source code and demographics from the participant. For basic correlations, we found moderate to strong correlations between our keystroke metrics (keystroke coefficient, response proportion, response corrections) and the number lines/characters in the true program output. Log trial duration was also found to be moderately correlated with lines of code and Halstead effort. Surprisingly, correct output distance was not significantly correlated with any single complexity metric.

Using a multiple-predictor linear model, we found that only keystroke coefficient and response proportion had sizeable coefficients (Section 5.1). In both cases, the number of lines in the true output was the strongest predictor. For keystroke coefficient, the relationship was negative – more output lines decreased the metric (fewer unnecessary keystrokes). Response proportion increased with output lines, which makes sense if participants began evaluating `print` statements as soon as they were encountered.

We were surprised to see such small coefficients for the other performance metrics, especially correct output distance and log trial duration. At a minimum, we expected Python and/or programming experience to be a strong predictor of correct output distance: more experience should reduce errors. Similarly, we expected lines of code to strongly predict log trial duration: more code should take longer to read. In both cases, however, our models did not reveal a strong (linear) relationship. Results improved for correct output distance when we included other performance metrics as predictors, but the coefficients were still small relative to the range of the response variable. It is possible that a non-linear model could provide a better fit to the data, but we do not have a strong theoretical reason to expect specific interactions between predictors.

We investigated a more coarse-grained performance metric in the last part of Section 5.1: whether or not a trial had a correct response. Using a collection of binary classifiers trained on all trials with cross-validation, we found that the Random Forest classifier performed well; achieving a mean AUC score of 0.86 (with a max of 0.94). This means that the probability of ranking a positive instance above a negative instance is approximately 0.86 – much better than chance. We examined the relative importance of each predictor (feature) for this classifier, and found that keystroke coefficient, response proportion, and log trial duration were at the top. Thus, it appears that the correctness of a trial response can be well predicted in our data set, but **not** by complexity metrics and demographics alone[7]. By considering the length of a trial and various keystroke metrics, it appears we can strongly predict whether or not a participant's response was correct.

7 Part 1: Conclusion

In this part, we presented an experiment in which programmers predicted the output of 10 Python programs (drawn from a set of 25 programs). The performance of participants on each trial, a single program prediction, was quantified using a collection of performance metrics. Small differences between versions of each program were predicted to affect participants due to either expectation violations or attributes of physical notation. We observed both kinds of effects, though not always where they were expected.

The `scope` and `counting` programs produced the largest error rates, closely followed by `between`. These errors were driven by expectation violations entirely for `scope`, and partially for `counting` and `between`. While it's clear to an outside observer that the code and the program's intention differ, many participants did not notice. Interestingly, we observed Python/programming experience helping (`scope`), hurting (`between`), and having no effect (`counting`). The `scope` results align with previous research on the so-called "Rules of Discourse" for programming [89], but the other results are more reminiscent of Schank and Abelson's scripts [80]. Participants incorrectly responding to the `between` and `counting` programs appeared to be inferring high-level intentions, and ignoring bottom-up cues (e.g., indentation) that did not fit the "script."

Notational effects were observed in `counting` and `order`, with the strongest effects showing up in the former. Python does not include an ending delimiter for blocks like many other languages (i.e., an `end` keyword or closing brace), so indentation is the only visual cue for statement grouping. Our experiment suggests that this visual grouping can be manipulated by simply adding empty *vertical* whitespace around grouped statements. There are many open questions, however, and we must be cautious in drawing broad conclusions from this one experiment. Though much weaker, the observed notation effect in `order` is also of interest. The two `order` programs are virtually identical,

[7]Training the Random Forest classifier without performance metrics results in a mean AUC score of 0.68.

but the results suggested an implicit expectation that the methods be called and defined in the same order[8]. A more spatially-oriented complexity metric, such as Douce's *FC* metric [32], may be able to quantify these types of nuances.

Lastly, we found that predicting performance could only be done for the binary correct/incorrect response metric, and only if other performance metrics were included as predictors. Were our programs more difficult, or our task different, we would expect complexity metrics and demographics to drive performance more than we observed in this experiment.

7.1 Future Work

During the course of the experiment, Bloomington participants were seated in front of a Tobii X300 eye-tracker. We plan to analyze this eye-tracking data, and correlate it with our findings here. Specifically, we hope to see how code features and experience affect the visual search process and, by proxy, program comprehension. We will also be investigating whether or not our performance metrics can be predicted from eye-tracking metrics, such as mean fixation duration and spatial density [71]. Because we capture keystrokes in real time, we have the opportunity to observe participants' gaze patterns as they are responding and (perhaps) correcting those responses.

For future experiments, we would like to include other languages and more realistic programs (e.g., multiple files and modules). Our work to date has focused exclusively on reading short Python programs, so a similar experiment where participants *write* short programs may prove insightful. Task performance in such an experiment may be more difficult to quantify, however. Our task has a well-defined expectation (predict the program's printed output), whereas asking participants to write a program according to some specification may be too open-ended. Still, observing participants as they read a specification and construct/modify a program would provide a window into the use of *programming plans*: hypothesized mental schemas that programmers use to read and write programs [73].

[8]It's possible that the alphabetic names of the methods provided this implicit ordering.

Part 2

eyeCode: An Eye-Tracking Experimental Framework for Program Comprehension

Abstract

Psychologists and computer scientists have studied the cognitive aspects of programming for nearly thirty years. Eye-tracking has gained popularity recently as a tool for gaining insight into programmer's cognitive processes while they comprehend programs. Both low-level eye movement metrics, such as fixation duration, and high-level metrics like line to line transition probabilities, provide rich alternatives to standard think-aloud experiment protocols. We present an experiment in which programmers predict the output of ten short Python programs, each with alternative versions. We use eye movements to discover important code elements/lines, characterize evaluation strategies, and investigate the source of participant errors. In addition, an open source data analysis library is introduced that provides specialized metrics, plots, and statistics for eye-tracking program comprehension experiments.

8 Part 2: Introduction

The cognitive aspects of programming have been studied by psychologists and computer scientists for nearly thirty years [29], culminating in the development of several **cognitive models** of program comprehension (e.g., [96, 82, 16, 30]). Though useful as tools to describe and reason about program comprehension, these (mostly qualitative) models cannot *quantitatively* predict human behavior when comprehending a specific program. The development of a quantitative cognitive model would represent a milestone in the understanding of program comprehension, and facilitate the semi-automated analysis of design alternatives for programming languages and integrated development environments (IDEs).
Recently, the use of **eye-tracking** has become prevalent in the study of program comprehension [97, 4, 14]. Eye movements are a rich data source, and have been strongly linked with visual attention and cognitive processes [75]. By collecting and analyzing eye movement data from developers during a specific programming task, researchers hope to gain insight into the cognitive processes programmers use to read and understand programs. When combined with other sources of data, such as responses to questionnaires, task timing, response accuracy, and participant experience, the space of possible cognitive models can be strongly constrained. Our programming task, predicting a program's printed output, is a starting point for the development of a cognitive model. This model will predict human behavior when interpreting simple Python programs.

8.1 The Experiment and Research Questions

We present an experiment in which 29 participants with a range of Python and overall programming experience predicted the output of ten small Python programs. Most of the program texts were less than twenty lines long and performed simple calculations (e.g., computed the area of a rectangle). We used ten different program **bases**, each of which had two or three **versions** with subtle differences. Participants were randomly assigned a version of each program, and performed the experiment in front of a Tobii TX300 free-standing eye-tracker (recording at 300Hz). We computed a variety of eye movement metrics, and produced high-level static and dynamic summary statistics of our data. These metrics and statistics, such as time spent looking at particular code elements and transition probabilities between code lines, helped us answer several research questions.

First, **how does the eye movement data from our experiment compare to other eye-tracking program comprehension experiments?** Our task, output prediction, is relatively unique compared to other studies. Locating bugs and answering comprehension questions are the most common tasks (see [14] for an brief survey). Like many other eye-tracking studies, we make use of common fixation metrics and relationships between areas of interest (AOIs) to summarize our data. In addition, we incorporate specific program comprehension measures – e.g., proportion of lines reviewed in the first 30% of a trial [97] – and compute fixation metrics over rolling time windows.

Second, **can aggregate eye movement metrics and summary statistics be predicted from textual/syntactic features of code?** Readability studies of code have found that both textual features (whitespace, word/line length), and syntactic features of code (identifier/keyword count) can influence assessments of the perceived complexity of the code and reading behavior [11, 22]. By linking code features with eye movements, it may be possible to predict how difficult a specific program is to read and comprehend. Another benefit of these studies is the contribution of valuable empirical data towards the investigation of programming language usability [91]. Language usability studies are rare, and the addition of new language features seldom involves controlled human experiments. A quantitative understanding of program comprehension would be a boon to researchers and members of industry who are examining language design alternatives.

Finally, **do differences between versions of the same program, or demographics/performance of the participant, influence eye movements?** Eye movement differences have been previously observed between expert and novice programmers during comprehension and debugging tasks [5, 7]. While more experienced programmers tend to perform better, researchers have noted differences in *strategies* between more and less experienced developers. In our experiment, we consider both a participant's Python and overall programming expertise, as well as which *version of a program* they were asked to interpret. Some program versions were designed to reward syntactic and semantic knowledge of Python (e.g., `counting`, `scope`), while others were intended to expose performance differences (e.g., `rectangle`, `whitespace`). When analyzing our results, we separated trials for some programs by response correctness, and compared eye movement metrics between the two groups.

8.2 Outline

This chapter is organized as follows. Section 9 provides background on eye-tracking in general and as it has been applied specifically to program comprehension. Our experimental methodology is introduced in section 10 along with the metrics and data transformations used in our analysis. Section 12 presents a detailed analysis of our data, with section 12.2 breaking down results by program base. The discussion in section 13 connects the details of our results with the three research questions in this section. Lastly, section 14 concludes and considers future work.

9 Part 2: Background

Although it has only recently been applied to program comprehension, eye-tracking has existed in one form or another for over a century. In this section, we start by reviewing the mechanisms of modern eye-trackers and the assumptions made when analyzing eye movement data. Next, we describe how eye-tracking has been incorporated into studies of program comprehension. Based on these studies, we outline expectations for the results of our experiment

9.1 Eye-tracking Methodology

Researchers have collected eye movement data for over one hundred years, usually to study natural language reading behavior [72]. Modern eye-trackers tend to use an infrared camera and LED to detect the pupil center and corneal reflection. With some calibration and a bit of trigonometry, an individual's "point of regard" can be determined fairly accurately [71].
Eye-tracking hardware can be broadly classified as **free-standing** or **head-mounted**. Free-standing eye-trackers function much like a webcam attached to a monitor. The participant is free to move about, though data accuracy is lost if their head moves too far outside of an expected range [9].
Head-mounted eye-trackers solve this problem by attaching the camera to the participant's head, allowing more range of motion. This increases data accuracy, but may be distracting or otherwise interfere with the experiment.
Raw eye movement data, called gaze points, are processed into **fixations** and **saccades** by software

[9]A chin rest can help keep the participant's head still, but forces them to perform the experiment with a potentially unnatural posture.

outside the eye-tracker – typically provided by the hardware vendor. A fixation occurs when the eye maintains its gaze on a particular location for an extended period of time. Fixations last around a few hundred milliseconds, and are broken up by saccades: rapid jumps from one fixation location to another [10]. For a given task, fixations are mapped to specific **areas of interest** (AOIs): relevant regions of the task environment. AOIs are often individual words for reading tasks, or specific objects in a scene for image-based tasks. A series of fixations from one AOI to another is called a **scanpath**, and can be used to compare participants' task strategies [75].
A variety of eye-tracking metrics exist to quantify eye movement behavior. Fixation counts and duration for each AOI are common metrics for determining which words or objects participants considered most important. There is a crucial assumption being made here, called the **eye-mind hypothesis** – that point of regard and visual attention are strongly correlated. This hypothesis lies at the heart of most eye-tracking research, and is widely accepted by the eye-tracking community. It should still be kept in mind when interpreting results, however. Other eye movement metrics, such as the density of fixations in a given area or the amplitude of saccades, can be used to infer the quality of information cues in the task environment. All of these metrics, of course, depend on the details of the software used to transform gaze points into fixations and saccades. Like the eye-mind hypothesis, these details are potential threats to the validity of experimental results.

9.1.1 Benefits and Potential Problems

Eye movement data provides rich, fine-grained physiological data for visually-intensive tasks. With the eye-mind hypothesis, this allows researchers to gain a unique insight into a participant's cognitive processes with high temporal resolution. Other experimental protocols may provide similar insights, such as "think aloud" in which a participant narrates their activity during the task. The data from such protocols are necessarily mediated by consciousness, however, and therefore may not reflect the details of cognitive processes outside of conscious awareness [65]. To the extent that the task at hand depends on these types of processes, eye movements have an advantage. Additionally, the density of fixation and saccade data make it ideal for statistical methods, both within and between participants.

[10] Besides alternating between fixations and saccades, humans and other animals have a "smooth pursuit" visual mode in which the eyes smoothly following a moving target [17].

There are potential problems in the collection and interpretation of eye movement data. The accuracy and precision of fixations and saccades strongly depends on the eye-tracking hardware and experience of the data collector [66]. In some cases, a participant's eye color, contacts, and eye make-up have been found to decrease data accuracy. Even under ideal collection conditions, there is also some guesswork involved in the transformation of raw gaze data into fixations/saccades, and again in the mapping of fixations to areas of interest (AOIs). Using different methods for each data transformation step may lead to very different conclusions, especially when using a between-subjects experimental design with few participants (common in many eye-tracking studies). Finally, comparing eye movements between participants may require additional data massaging or quantization when looking behaviors are highly individualized. For example, the precise fixation order of code lines in our task varied considerably between participants, leading us to favor the comparison of high-level metric between groups of participants instead (see Section 12.1.3 for details).

9.2 Reading and Program Comprehension

Within the last decade, eye-tracking has become more prevalent in studies of *program comprehension*. In these studies, programmers perform a variety of tasks on source code, such as locating bugs [97], predicting output [46], and answering comprehension questions [14]. Participants' eye movements during program comprehension are used to infer the effects of visualizations [4], expertise [22], pair programming [84], and other environmental factors.

Bednarik et. al performed extensive eye-tracking experiments with expert and novice programmers, focusing on the utility of program visualizations alongside source code [4]. Besides task performance, they found differences between novices and experts in both low-level eye movement metrics (mean fixation duration), and high-level looking behaviors (attention switches between code and visualization). Experts, for example, attended to the source code more than novices, and appeared to integrate more information between the different representations of a program. An earlier paper by Bednarik also found that experts were more affected by being force to use a restricted focus viewer (RFV) when viewing a program's source code and alternative graphical representation [6]. The RFV blurred the contents of the screen except for a small region controlled by the participant. In this paper, Bednarik et. al found that, while task performance did not decrease, more experienced programmers switched visual attention between representations less often when the RFV was active. Novices did not modify their behavior, however, suggesting that experts were making heavier use of peripheral vision when RFV was not active.

Uwano et. al had participants review C source code to locate defects, and analyzed the resulting patterns in their eye movements [97]. They observed a particular pattern, called *scan*, which represents a preliminary reading of the entire program (12-23 lines long) from top to bottom. In their experiment, approximately 73% of code lines tended to be fixated within the first 30% of a trial. Participants who spent more time in this scan pattern were also more likely to detect the defects in the program. We observed a similar scan pattern during some trials, but did not find a correlation with task performance.

Busjahn et. al compared the behavior of programmers reading both Java source code and natural language texts [14]. Significant differences were found in low-level eye movement metrics between text types, such as mean fixation duration and regression rates [11]. Substantial variation was also found between participants and between texts of the same type (source code or natural language). Additionally, the proportional fixation times for different categories of "words" in source code was analyzed, normalized by number of characters [12]. Out of keywords, identifiers, numbers, and operators, Busjahn et. al found that keywords received the least amount of fixation time per character. An earlier study by Crosby found similar results for keywords, though their other source code categories differed [22]. We observed the same pattern for keywords, and found that lines with mathematical operators or comparisons received the most fixation time (Section 12.1.2).

9.2.1 Expectations for Our Data

Based on the body of eye-tracking program comprehension study literature, we can form expectations for our own experiment. These expectations may be violated, however, given the uniqueness of our task. As mentioned previously, many eye-tracking experiments in this area ask participants to debug or answer comprehension questions about programs. In contrast, our participants were tasked with predicting the precise printed output of a program. Within the

[11] A regression occurs when a previously fixated word is fixated again.

[12] As noted by Busjahn et. al, normalizing by syllables or some other unit that can be processed within a single fixation may be more appropriate.

cognitive modeling subsection of program comprehension, programmers have demonstrated significant differences in their recalled representations of programs based on the task at hand [28]. Specifically, a programmer's "mental model" will differ if they're asked to recall relevant portions of a program for the purposes of documenting or summarizing versus modifying or reusing it. An analogous difference may be expected when programmers are asked to answer comprehension questions about, or literally evaluate, a program's source code.

Assuming *some* similarity between our task and others, however, we should expect mean fixation time to be related to experience [4], and to fall within the 300-400 ms range [14]. Regression rates are also predicted to be in the 30% to 40% range, well above the typical 10% to 15% range for natural language text [72]. When analyzing reading behavior, we should also expect the majority of lines to be fixated *in line order* within the first 30% of the trial [97]. Finally, participants are expected to fixate on more "complex" statements about twice as much as "simple" statements, and to spend the least amount of time on keywords [22].

Next, section 10 introduces our experimental methodology. Section 12 then delves into the details of our results. We analyze participants' eye movements in terms of reading behavior, by program base/version, and by participant demographics. Our data conformed to some of the expectations in this section (e.g., keyword fixation time), but violated others (e.g., fixation duration and experience). The discussion in section 13 considers reasons for these violations, and relates the detailed results to our research questions.

10 Part 2: Methodology

We now introduce our experiment design and eye-tracking hardware. Section 10.3 describes the process of transforming raw eye movement data into fixations, and mapping those fixations to areas of interest. Lastly, section 10.4 provides definitions for each of the eye movement metrics used in the analysis.

10.1 Experiment and Participant Demographics

We recruited 29 via e-mail and from an introductory programming class at Indiana University. Participants were paid $10 each, and performed the experiment in front of an eye-tracker. All participants were screened for a minimum competency in Python by passing a basic language test. The mean participant age was 27.3 years, with an average of 2.9 years of self-reported Python experience and 9.4 years of programming experience overall. Most of the participants had a college degree (86.2%), and were current or former Computer Science majors (65.5%). Figure 20 has a more detailed breakdown of the participant demographics.

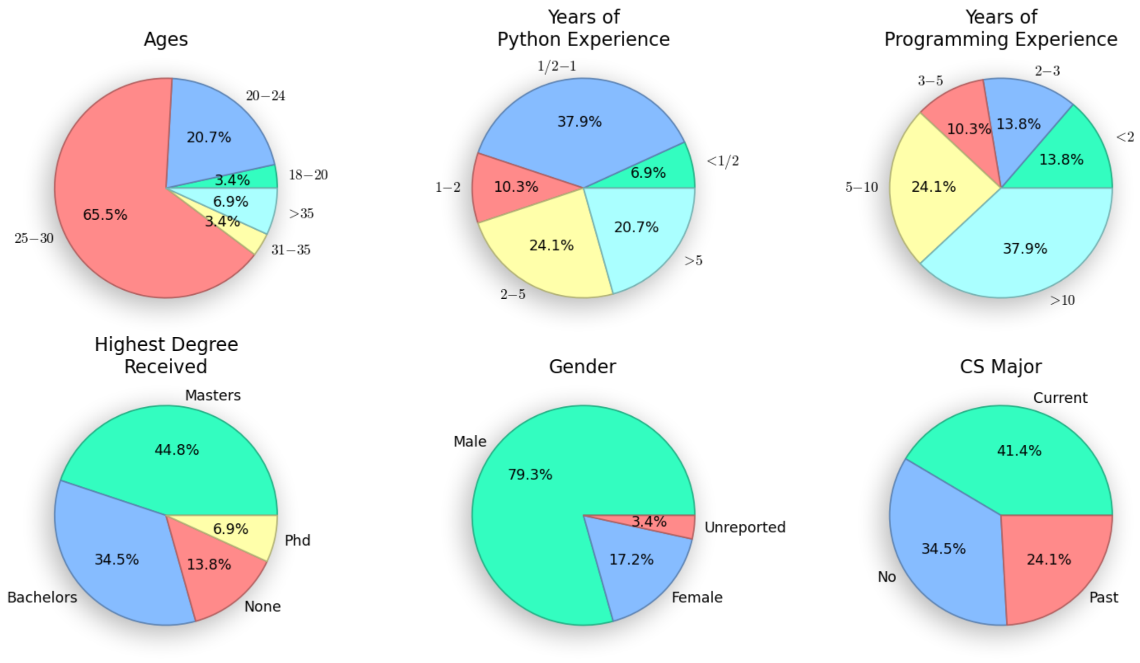
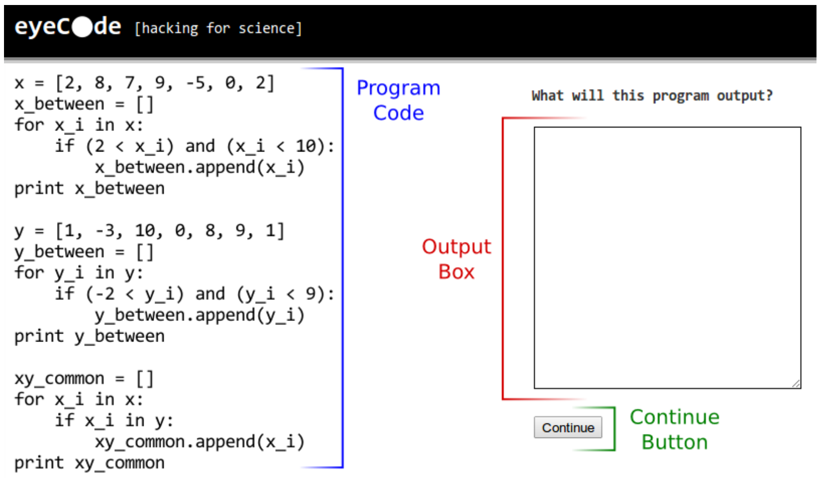


Figure 20: *Demographics of all 29 participants.*

The experiment consisted of a pre-test survey with questions about demographics and experience, ten trials (one program each), and a post-test survey assessing confidence and requesting feedback. The pre-test survey gathered information about the participant's age, gender, education, Python experience, and overall programming experience. Participants were then asked to predict the printed output of ten short Python programs, one version randomly chosen from each of ten program bases (Figure 21). The presentation order and names of the programs were randomized, and all answers were final. No feedback about correctness was provided and, although every program produced error-free output, participants were not informed of this fact beforehand. The post-test survey gauged a participant's confidence in their answers and the perceived difficulty of the task overall.

We collected a total of 288 trials from 29 participants starting November 20, 2012 and ending



```
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
```

Figure 21: *Sample trial from the experiment (`between inline`). Participants were asked to predict the exact output of ten Python programs.*

January 19, 2013. Trial responses were manually screened, and a total of 2 trials were excluded based on the response text. Participants were not constrained to complete individual trials or the experiment in any specific amount of time. There were a total of twenty-five Python programs in our experiment belonging to ten different program bases. The programs ranged in size from 3 to 24 lines of code, and did not make use of any standard or third-party libraries.

10.2 Eye-Tracking Hardware

The Tobii TX300 is a free-standing eye-tracker that collects gaze data at 300Hz, and averages an error of around 0.04 degrees of visual angle for both eyes [13]. The screen is 23 inches in size and measures 557 mm across with a 1920x1080 maximum resolution. Before starting the experiment, participants went through a brief calibration using the vendor-provided Tobii Studio software package. Aside from being asked to sit about 65 cm away from the eye-tracker and not to move their heads as much as possible, participants interacted with the TX300 as if it were a normal computer.

We used Tobii Studio 2.2, a software package provided by Tobii with the eye-tracker, to do calibration and to record/process raw gaze data into fixations and saccades. The TX300 records raw moment-to-moment gaze points, which Tobii Studio collects and processes using a *fixation*

[13]Note, however, that anything within 1 degree may be fixated by the fovea without moving the eyes.

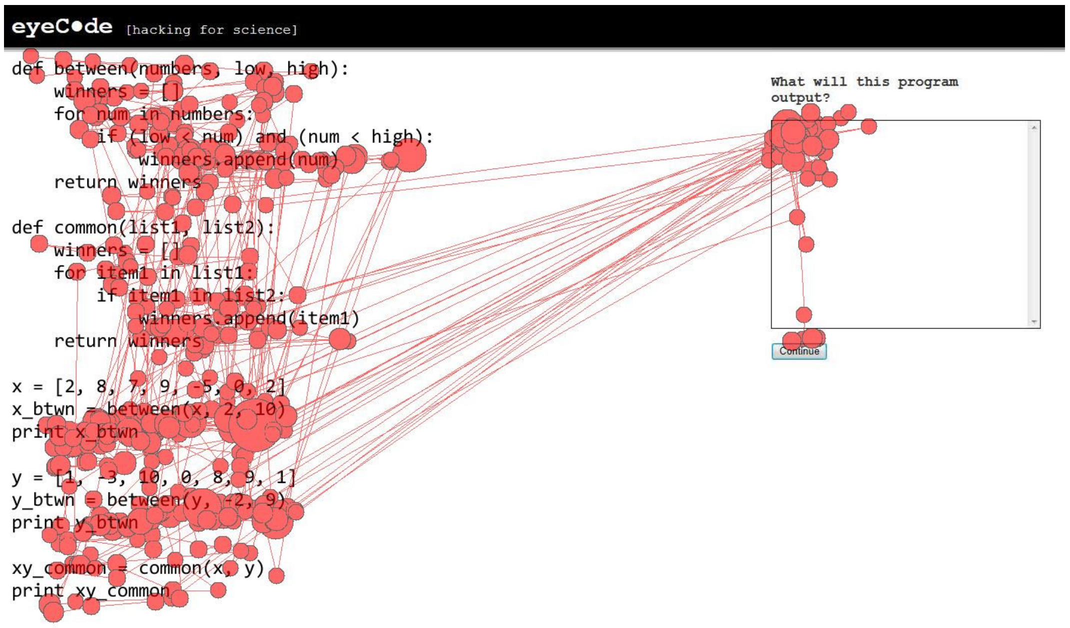
Figure 22: *Tobii TX300 Eye-tracker. Screen size is 23 inches (557 mm wide) with 1920x1080 resolution. When a participant is seated 65 cm away from the screen, it will subtend about a 31 degree visual angle.*

filter. There is no precise means of translating gaze points into fixations, so we relied on Tobii Studio's default I-VT fixation filter to perform this task for us. The Tobii Studio user manual [95] describes this filter as follows:

> *The general idea behind an I-VT filter is to classify eye movements based on the velocity of the directional shifts of the eye. The velocity is most commonly given in visual degrees per second (°/s). If the velocity of the eye movement is below a certain threshold the samples are classified as part of a fixation.*

More technical information about the I-VT fixation filter can be found in a Tobii whitepaper [67]. Per the Tobii Studio user manual, we also filtered out fixations which the eye-tracker marked as potentially invalid (i.e., with a left or right eye validity code higher than 1).

Over the course of 290 trials (10 trials per participant), we collected around 50,000 fixations (average of 172 per trial). Each fixation consists of a two-dimensional screen coordinate, a start time, and a duration in milliseconds. A quick plot of one trial's fixations on a static image of the participant's screen immediately reveals one of the many challenges involved in analyzing gaze data (see Figure 23). Like the gaze point to fixation translation step that Tobii Studio's fixation filter performs, there is some guesswork involved in mapping fixations to *areas of interest* (AOIs). In the next section, we describe our method for assigning fixations to words, lines, and interface regions on the screen.



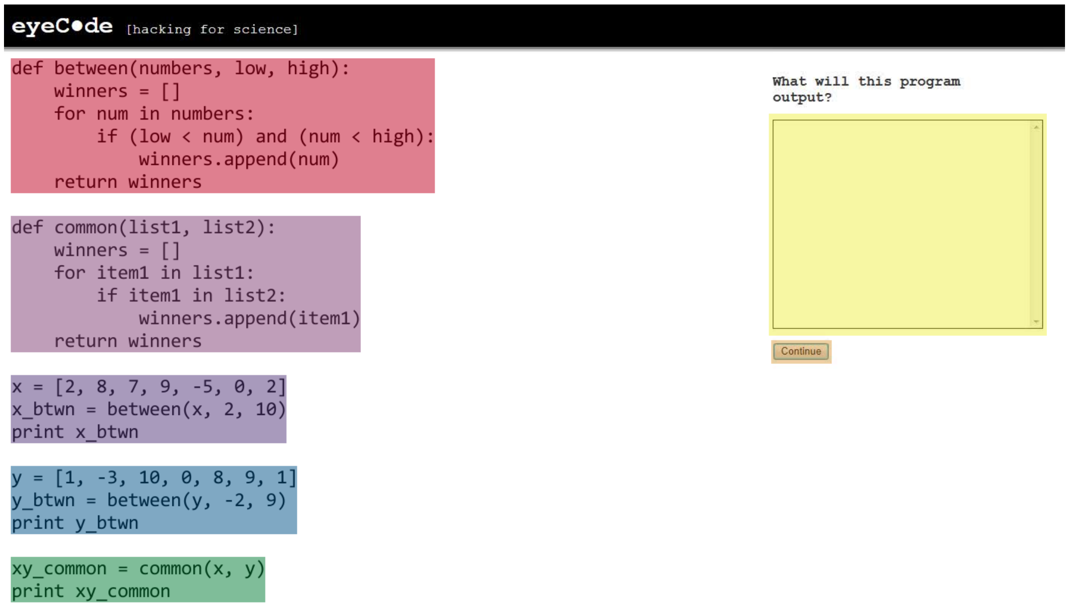
```
def between(numbers, low, high):
    winners = []
    for num in numbers:
        if (low < num) and (num < high):
            winners.append(num)
    return winners

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
```

Figure 23: *Fixation circle plot for a single trial (`between_functions` program). Circle radii are proportional to fixation duration.*

10.3 Areas of Interest

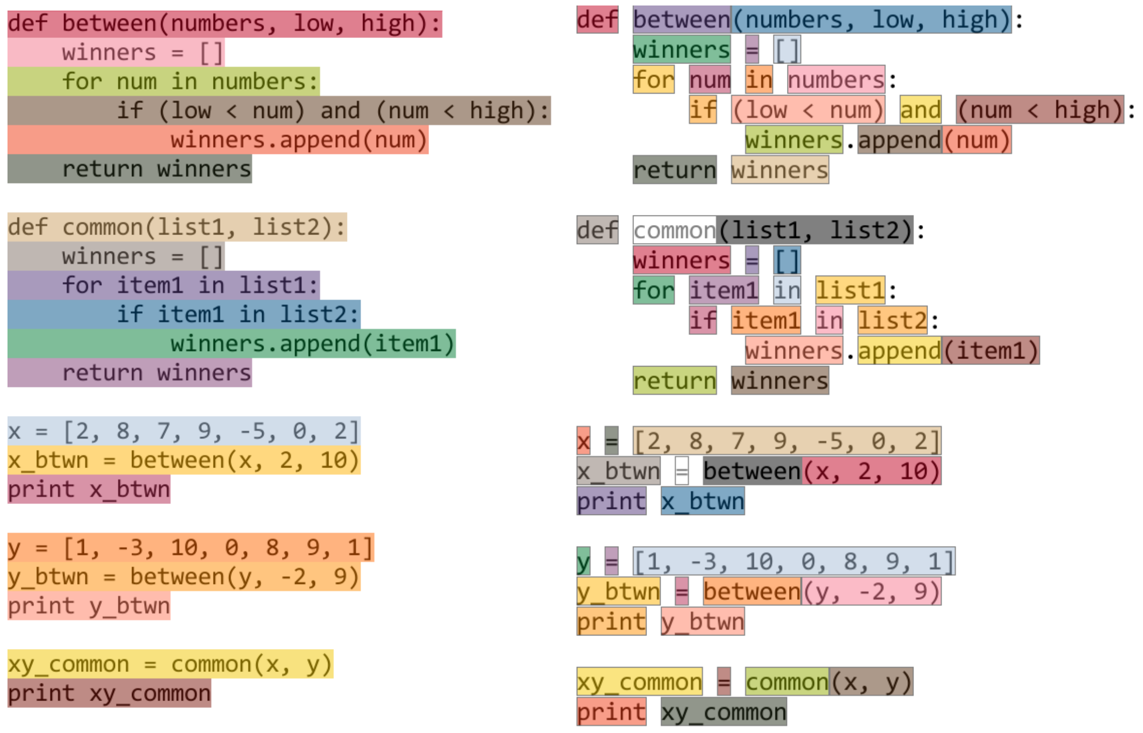
An area of interest (AOI) is a region of the screen where we would like to know when the participant is fixating. We define three kinds of code AOIs (block, line, syntax) and two kinds of interface AOIs (output box, continue button). A **block** AOI is one or more lines of code separated from other blocks by at least one blank line. Figure 24 shows the block AOIs for the `between_functions` program (highlighted on the left side of the screen) as well as the **output box** and **continue button** AOIs (highlighted on the right side of the screen). Figure 25 shows the **line** and **syntax** AOIs for the same program. Line AOIs include indentation because it is semantically relevant to Python. Syntax AOIs are automatically computed with the popular Pygments library [9] and assigned one of the following categories: keyword, identifier, operator (+, −), literal, or comparison (<, >).

Because fixations are essentially timestamped screen coordinates, the easiest way to map a fixation to an AOI is to simply check if the fixation point lies within one of the AOI rectangles (Figure 26). This method, which we will refer to as **point mapping**, is fast and easy to compute. When AOIs are large (relative to the expected error of the eye-tracker) and separated by a sufficient amount of



```
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
```

Figure 24: *Block areas of interest for the (*`between_functions` *program). The output box and continue button are also areas of interest.*



```
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
```

Figure 25: ***Left:*** *line areas of interest in the* `between_functions` *program.* ***Right:*** *syntax areas of interest in the* `between_functions` *program.*

empty space, point mapping can be used to accurately map fixations to the correct AOIs. If AOIs are small and close together, however, point mapping may fail to correctly map fixations due to hardware error or participant movement. Extending the boundaries of an AOI can help, but only if one AOI rectangle does not overlap another. Because our code AOIs are relatively small and close together, we consider a second mapping method that we call **circle mapping**.

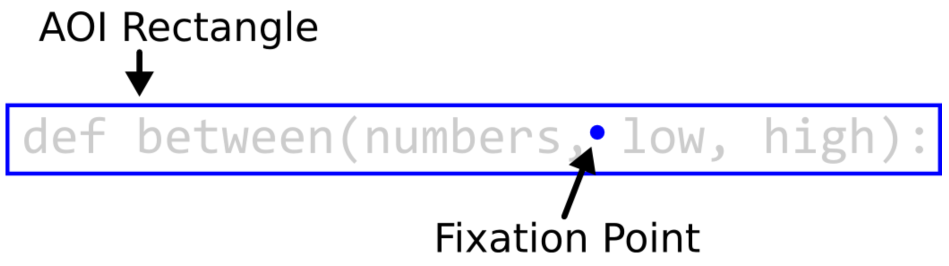


Figure 26: *Point-based AOI mapping. Fixations are mapped to an AOI if the fixation point lies within the AOI rectangle.*

When circle mapping fixations to AOIs, a circle is centered around the fixation point. For our analysis, the radius of the fixation circle (20 pixels) was chosen based on the size of our text and expected error of the eye-tracker. The area of the circle's intersection with every AOI is computed (Figure 27), and the AOI with the largest area of overlap is mapped to the fixation. This mapping method is more computationally intensive, but has an advantage over point mapping when AOIs are close and their boundaries cannot be extended without overlap. When a fixation occurs near two AOIs, the closest one will be mapped to the fixation. Circle mapping could also be used to assign probabilities to multiple AOIs (based on area of overlap), and combined with a Markov model to find the most likely series of fixations over time (see Future Work).

10.3.1 Offset Correction

During the data cleaning step of our analysis, fixation coordinates were manually adjusted in the vertical direction for each participant. A *single vertical offset* was used to correct fixations for each participant's trials, and was chosen to increase the overlap of fixations with non-empty portions of the screen. Figure 28 demonstrates the process: an uncorrected fixation plot is shown on the left. The fixations near the continue button (highlighted in green) suggest that the eye-tracker may be reporting gaze points slightly low for this participant. On the right, the corrected fixation plot lines

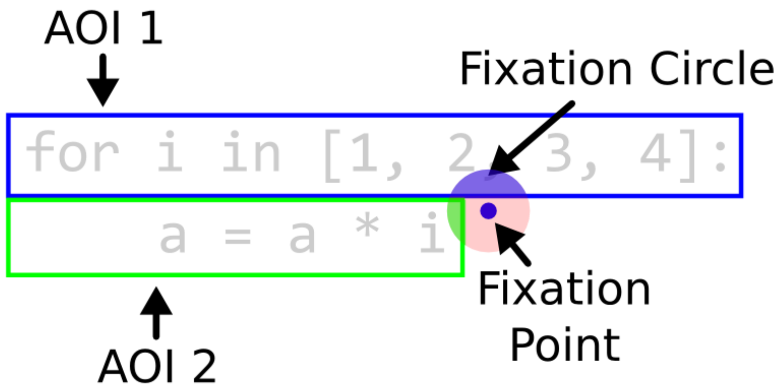
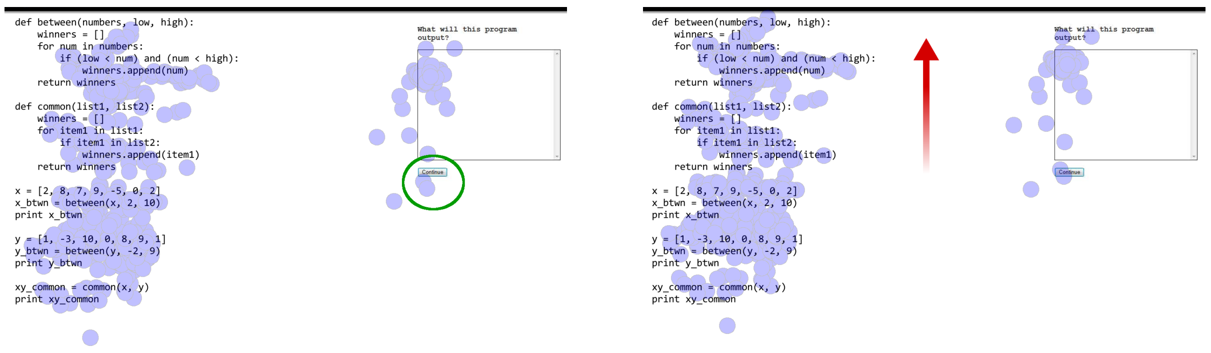


Figure 27: *Circle-based AOI mapping. Fixations are surrounded by circles and mapped to the AOI with the biggest area of overlap (AOI 1 in this case).*

up better visually with the non-empty portions of the screen. All offsets are documented, and the original raw data has been preserved.



```
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
```

Figure 28: *Example of offset correction. Raw fixations (left) were shifted up to better align with known areas of interest (right).*

While an automated offset correction process would have been preferred, conventional methods require knowing where a participant **must** be looking at some point in time. We did not require that participants fixate in a specific spot at the start or end of a trial, so this information was not available. Although it would be reasonable to expect participants to fixate on the continue button before ending a trial, we found several instances where this was clearly not the case. Indeed, some participants hovered their mouse cursor over the continue button, shifted visual attention to the code (perhaps for a final check), and clicked the button without re-fixating it first!

10.3.2 Scanpath Comparisons

With fixations mapped to areas of interest, we can now codify a participant's behavior over time. A **scanpath** is a string representing the order in which AOIs were fixated during a trial. This string does not typically contain any duration information, and adjacent fixations on the same AOIs are usually collapsed into a single symbol. For example, the sequence of fixated AOIs in Figure 29 is `AABCBCCA`. Removing duplicate adjacent fixations, we are left with the scanpath `ABCBCA`.

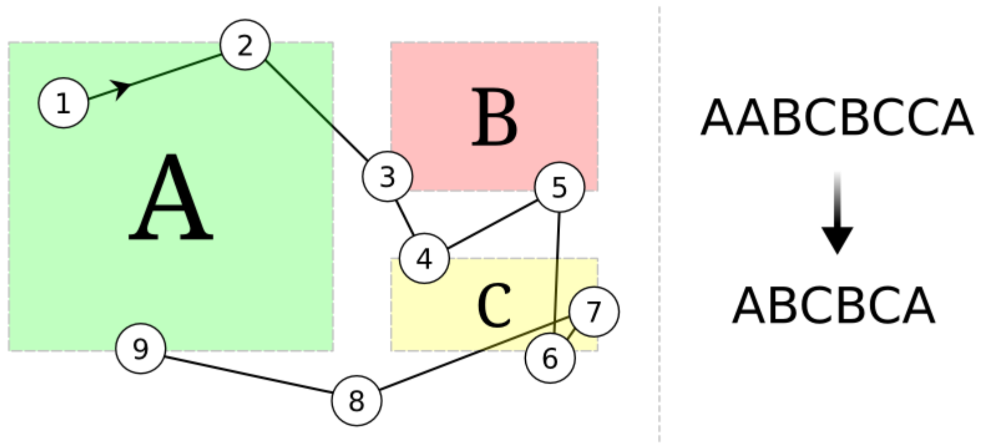


Figure 29: *Creation of an AOI scanpath. Fixations are mapped to an AOI sequence string, often with repeated items removed.*

Scanpaths can be used to generate **transition matrices**, which quantify how often one AOI is fixated after another. Using the scanpath from our example above (`ABCBCA`), the transition matrix in Figure 30 shows that `B` always follows `A`, that `C` always follows `B`, and that both `A` and `B` follow `C` half the time. More sophisticated methods, such as the scanpath successor representation [47] incorporate discounted temporal information (e.g., `C` *eventually* follows `A`). For our analysis, however, we only make use of simple transition matrices.

Metrics like the Levenshtein Distance (commonly know as string edit distance) can be used to compare scanpaths. This distance metric between two strings increases by one for each insertion, deletion, or character change necessary to transform one string into another [79]. The distance between similar strings, such as `123` and `1234` will be small (in this case, just 1). The maximum edit distance between two strings is the length of the largest string, and thus we can use this value as a normalization factor (distance / length of largest string). The normalized edit distance between

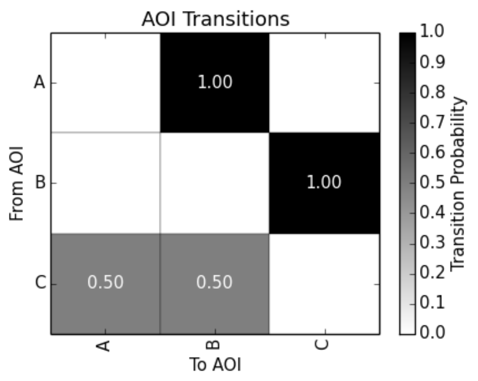


Figure 30: *Transition matrix for the scanpath* `ABCBCA`*.* `B` *always follows* `A`*,* `A` *follows* `C` *half the time, etc.*

scanpaths can be used to cluster trials by **strategy** – common orderings of AOI fixations. This makes sense when there are strong expectations about AOI fixation order, and when scanpaths are close in length. More general algorithms, such as ScanMatch [21], make use of techniques developed for DNA sequence matching. These algorithms allow for more fine-grained control of sequence substitutions (e.g., by penalizing specific character replacements). They are also much better at dealing with scanpaths that have missing segments.

10.4 Eye-Tracking Metrics

Once gaze data has been processed into fixations and saccades, there are a plethora of metrics available to summarize the data [71]. Table 4 lists the handful of metrics we used to summarize participant fixations within and between trials. The **fixation count** is simply the number of fixations, while **mean fixation duration** is the average amount of time a participant spent fixation some area of interest (AOI). **Fixation rate** is the number of fixations per second over the course of a trial. **Spatial density** divides an AOI into a grid, and is higher when more cells in the grid receive at least one fixation. The **normalized saccade length** and **average saccade length** measure the total and average Euclidean distances between fixations. Finally, the **Uwano review percent** is the percentage of lines fixated in the first 30% of a trial.

Name	Description
Fixation Count	Number of fixations in a given time period.
Mean Fixation Duration	Total fixation duration in a given time period divided by the fixation count.
Fixation Rate	Number of fixations divided by the number of seconds in a given time period.
Spatial Density	Number of grid cells containing at least one fixation in a regular $N \times N$ grid covering a given AOI [20].
Normalized Scanpath Length	Total Euclidean distance between fixations in a scanpath divided by time between first and last fixation.
Average Saccade Length	Average euclidean distance between start and end points of each saccade.
Uwano Review Percent	Percentage of code lines reviewed within the first 30% of a trial [97]

Table 4: *Eye-tracking metrics used in our analysis.*

10.4.1 Line Metrics

What properties of a line of code influence a participant's reading behavior? We might expect the position and length of a line to influence when a participant fixates the line and for how long. We define several **textual** and **content-based** metrics to quantify the properties of each line of code. Table 5 describes textual metrics, which could be applied to any text – source code or natural language. These metrics quantify the size and position of a line, as well as the distribution of whitespace within it. Table 6 lists metrics specific to source code, such as the number of keywords and operators.

The **line length** is simply the number of characters in a line, excluding the whitespace at the start of a line. Rather than use line number, we calculate **line number proportion**. This metric tends to correlate better with the time of first fixation (Section 12.1), suggesting that participants perform an initial scan of a program faster for larger programs. The **whitespace proportion** of a line is simply the proportion of spaces to characters, excluding initial indentation. Lastly, the **indentation level** measures the amount of whitespace before a line begins, divided into 4 space blocks (a standard in Python).

For content-based metrics, we define the **keyword count**, **identifier count**, and **operator count**. These are simply the number of keywords, variable/function names, and mathematical/boolean operators in a given line. The **category** of a line was based on a number of factors. For example, lines containing a `for` or `return` keyword were classified as "For Loop" and "Return Statement."

Textual Metric	**Definition**
Line length	Number of characters (excluding indentation)
Line number proportion	Line number divided by total number of lines
Whitespace proportion	Number of spaces (excluding indentation)
Indentation level	Number of 4-space blocks on the left

Table 5: *Textual line metrics*

Content Metric	**Definition**
Keyword count	Number of `class`, `def`, `for`, `if`, `print`, `return`
Operator count	Number of `*`, `+`, `-`, `.`, $<$, `=`, $>$, `and`, `in`
Identifier count	Number of class/function/variable names
Line category	Content of line, one of: • List creation (literal $[1, 2, 3]$) • Comparison ($x < y$) • Math operation ($+$, $*$, $-$) • For loop (`for x in y`) • Function call (`f(x)`) • Function definition (`def f(x)`) • If statement (`if x`) • Print statement (`print x`) • Return statement (`return x`) • Class definition (`class Foo`) • Assignment (`x = y`)

Table 6: *Content-based line metrics*

Some lines, such as `x = [1, 2, 3]`, could have multiple categories (assignment, list creation). We picked the category for each line that was highest in the list given in the final cell of Table 6, so the example line would be classified as a list creation rather than an assignment.

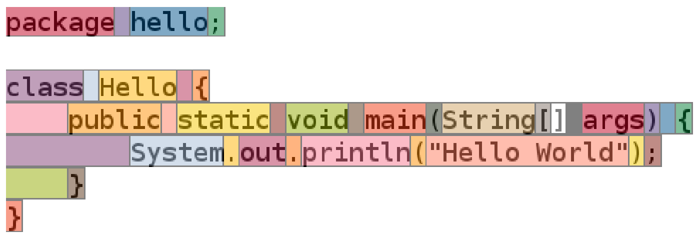
11 Part 2: The eyeCode Library

To facilitate the analysis of eye movement data in the context of program comprehension, we created a Python library called **eyeCode**. Tobii Studio provides basic plots and statistics, but customization is limited to a few menu options. A more sophisticated tool, OGAMA (Open Gaze and Mouse Analyzer), is a freeware package with many more bells and whistles [99]. In addition to standard eye movement plots, OGAMA can be used to create areas of interest and compute the Levenshtein distance between scanpaths. Like Tobii Studio, however, OGAMA is intended to be fairly task neutral, and therefore did not have specific tools for program comprehension experiments.

The eyeCode library is built on top of the `pandas` statistical computing library [60], and contains specialized functions for processing, plotting, and computing metrics over fixations and saccades on a static code display. The source code is freely available at `http://github.com/synesthesiam/eyecode` under a liberal open source license. Example analyses and data from multiple experiments (including this one) are embedded into the library along with participant info and a web-based fixation viewer.

11.1 Areas of Interest

Identifying areas of interest (AOIs) can be done **automatically** for programming languages that are supported by the popular Pygments library [9]. This relieves the researcher of the need to manually create AOIs for each code "word", line, and block. Pygments contains lexers for a variety of languages and, when combined with a monospace font size and line spacing, can be used to generate rectangles for every token in a program. Figure 31 shows the token AOIs identified from a short Java program. Tokens are then merged into lines, and lines are merged into whitespace-separated blocks.(Figure 32). For experiments using variable-width fonts, eyeCode has methods for scanning raw images to locate words and lines.

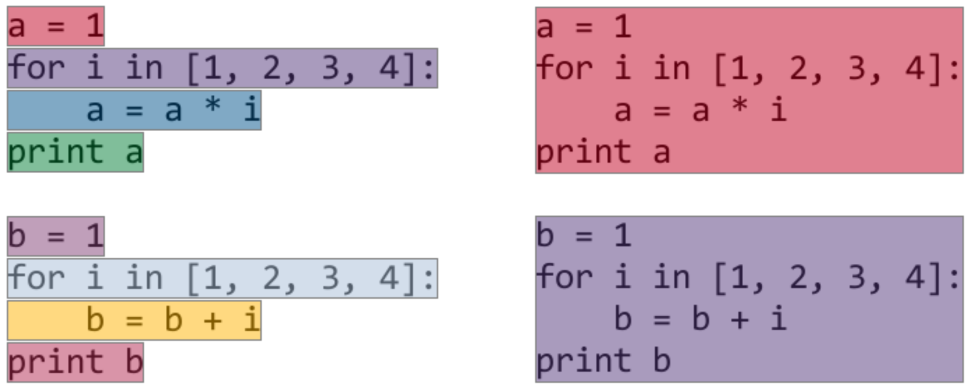


```
package hello;

class Hello {
    public static void main(String[] args) {
        System.out.println("Hello World");
    }
}
```

Figure 31: *Areas of interest for a Java program automatically identified using the Pygments Python library.*

After identifying areas of interest, eyeCode can map fixations to AOIs via a process called **hit testing**. Two methods are currently supported: point mapping and circle mapping (see Section 10 for details). AOIs in eyeCode can also be grouped into layers, allowing overlapping AOIs to be hit-tested simultaneously (only AOIs within a layer must be disjoint). Hit-tested fixations can be easily converted to scanpaths, summarized with metrics, or visualized with one of many plots.

11.2 Metrics and Plots



```
a = 1
for i in [1, 2, 3, 4]:
    a = a * i
print a
```

```
a = 1
for i in [1, 2, 3, 4]:
    a = a * i
print a
```

```
b = 1
for i in [1, 2, 3, 4]:
    b = b + i
print b
```

```
b = 1
for i in [1, 2, 3, 4]:
    b = b + i
print b
```

Figure 32: ***Left***: *Line-based AOIs for the* `initvar - onebad` *program.* ***Right***: *Block-based AOIs for the same program.*

Common eye movement metrics, such as fixation duration, can be computed per area of interest and across trials (Figure 33). Higher-level metrics, like transition matrices and spatial density [20], are available along with many custom visualizations. Because lines are especially relevant for code, eyeCode contains specialized plotting functions for both static and dynamic views of line fixations across and within trials. Figure 34 shows an aggregate view of fixation duration by line for all participants in a single program (top), and a timeline of fixations on each line for a single trial (bottom). Fixations in the timeline above the dotted line occurred in the output box – the text area where participants enter their predictions of program output.
The "Super Code" plot combines fixation duration information about code elements, lines, and blocks into a single plot (Figure 35). The color of each code element represents the relative amount of time spent on that element. Each line has a bar next to it, with the length proportional to the total fixation duration on that line. These bars are also colored, and each block of code shares the same color. This shows the relative amount of time spent on each block as a whole – the darkest red set of bars had the most fixation duration.

11.3 Rolling Metrics

The eyeCode library supports computing some low and high-level metrics over a rolling time window in a trial. Comparing the visualization of these rolling metrics with the line fixation timeline allows for the gathering of additional evidence to support hypothesis about participants'

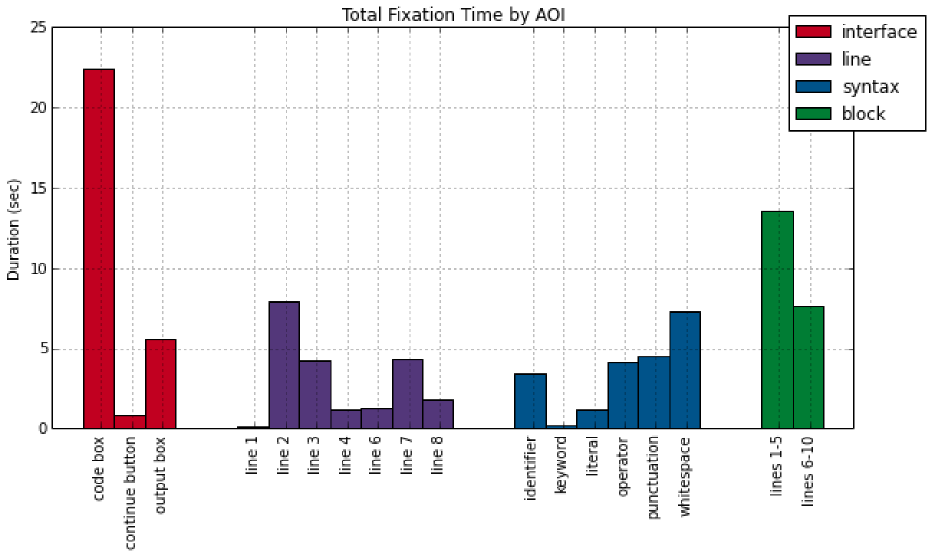
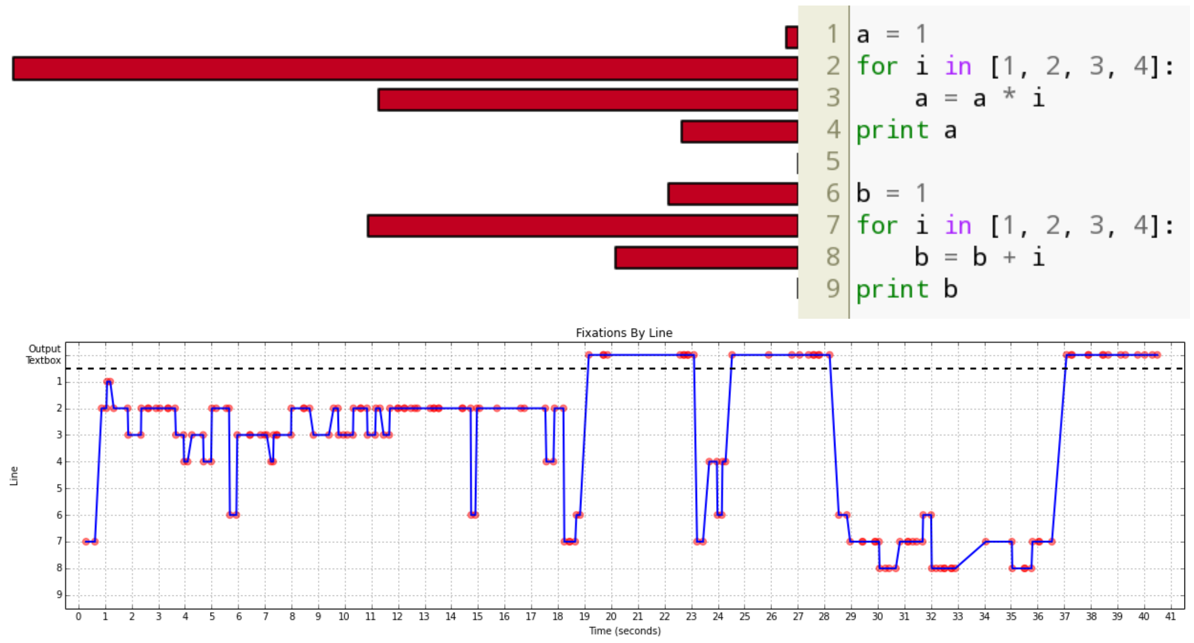


Figure 33: *Total fixation duration by AOI kind and name for* `initvar - onebad` *(all participants).*



```
a = 1
for i in [1, 2, 3, 4]:
    a = a * i
print a

b = 1
for i in [1, 2, 3, 4]:
    b = b + i
print b
```

Figure 34: ***Top***: *Total fixation duration for each line (all participants).* ***Bottom***: *Fixation timeline for experiment 1 trial 1.*

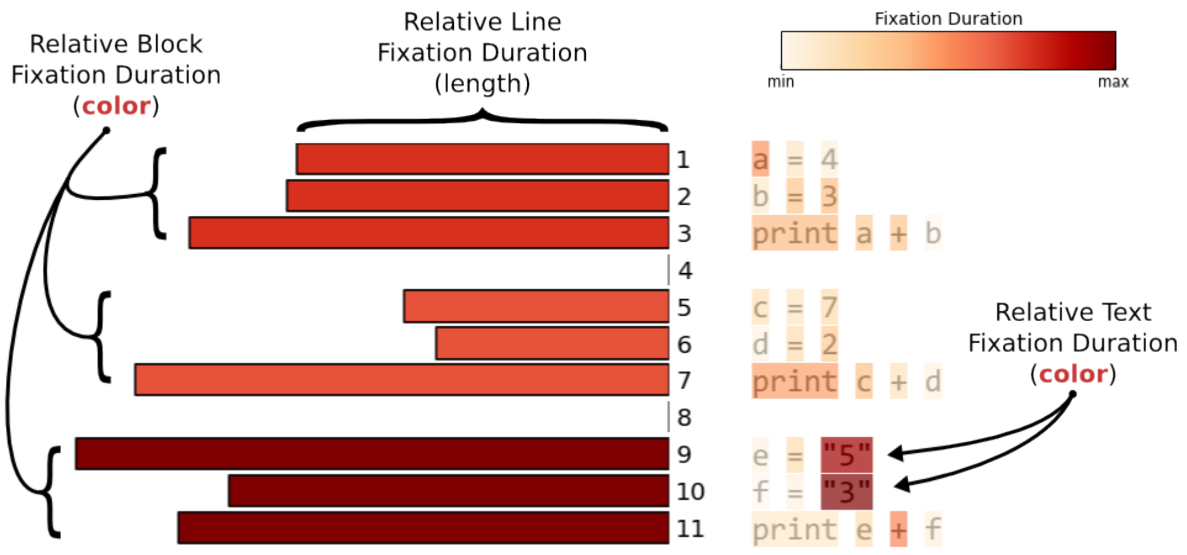


Figure 35: *Relative fixation duration plot for all* `overload plusmixed` *trials. Text color indicates fixation duration relative to other code elements. Bar color indicates relative fixation duration per whitespace-separated block. Bar length is proportional to within-block fixation duration.*

real-time cognitive processes. In Figure 36, for example, both the average length of a saccade and average duration of a fixation are computed every half second across a one second window. Large changes in either metric can be mapped back to the line fixation timeline in Figure 34, and potential causes can be enumerated based on where and when the participant is looking. Some metric spikes, such as the increase just after 25 seconds, may be expected – here, the participant is clearly transitioning back and forth between the output box and the source code. Others, such as the increase in average fixation duration just after 15 seconds, suggest increased focus. Given the subsequent transition to the output box (and that the first response characters are typed afterwards), it's reasonable to guess that the participant is mentally calculating the product $1 * 2 * 3 * 4 = 24$. Indeed, inspecting the participant's response reveals an "a = 24" following by a correction to just "24".

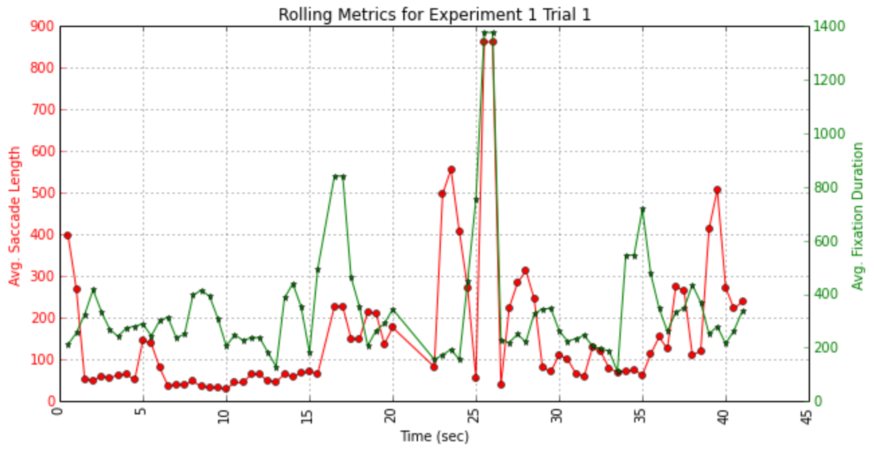


Figure 36: *Rolling metrics for experiment 1 trial 1 (`initvar - onebad`). Average saccade length (red, left) and average fixation duration (green, right) computed over a one second window every half second.*

12 Part 2: Results

We collected approximately 50,000 fixations from 288 trials across 29 participants. Below, we analyze and summarize our data first across all trials (Section 12.1), and then by individual program kind (Section 12.2). Plots and analyses were done using the eyeCode library. Statistics were computed with `scipy` [51], and linear models were fit using the `statsmodels` package [83]. Non-parametric statistics were used to compare groups (Mann-Whitney U test) and determine correlations (Spearman's r) [86]. We calculate effect sizes for U tests using the rank-biserial correlation r, a metric whose range is $[-1, 1]$ with 0 meaning no correlation [103]. As a rule of thumb, we take absolute values of r greater than or equal to 0.2 to indicate a meaningful relationship (and $|r| > 0.4$ as a strong relationship). For correlations, we infer weak, moderate, strong, and very strong relationships when $|r|$ is greater than or equal to 0.2, 0.3, 0.4, and 0.7 respectively.

12.1 Reading Behavior

Across all participants and trials, we observed a mean fixation duration of 273 ms. Fixations typically last 200-300 ms [72], but higher values have been observed in several program comprehension studies. A fixation duration range of 309-408 ms was found in a program comprehension study by Busjahn et. al [14]. Participants in the same study read natural language texts, and a typical fixation duration range of 232-285 ms was recorded. In another study by Bednarik et. al, participants under most conditions had mean fixation durations in the 300-400 ms range when viewing code [4].

When fixating the output box – the text box into which participants typed their responses – fixations were significantly longer, with an average of 331 ms versus 267 ms when fixating source code ($U = 89197473.5, p < .001, r = 0.03$). This suggests that participants were attending to their own predicted output slightly more, but the effect size is too small to be sure. Per trial, we observed an average of 172 fixations and approximately 2.73 fixations per second (Figure 37). At the trial level, and for each line of code, total fixation count and duration were almost perfectly correlated ($r = 0.97, p < .001$). At the interface level, participants produced an average of six transitions between the source code and output box, indicating an on-demand construction of responses. Our analyses of individual programs' code line and output box transition matrices support this hypothesis by correlating output transitions and responses.

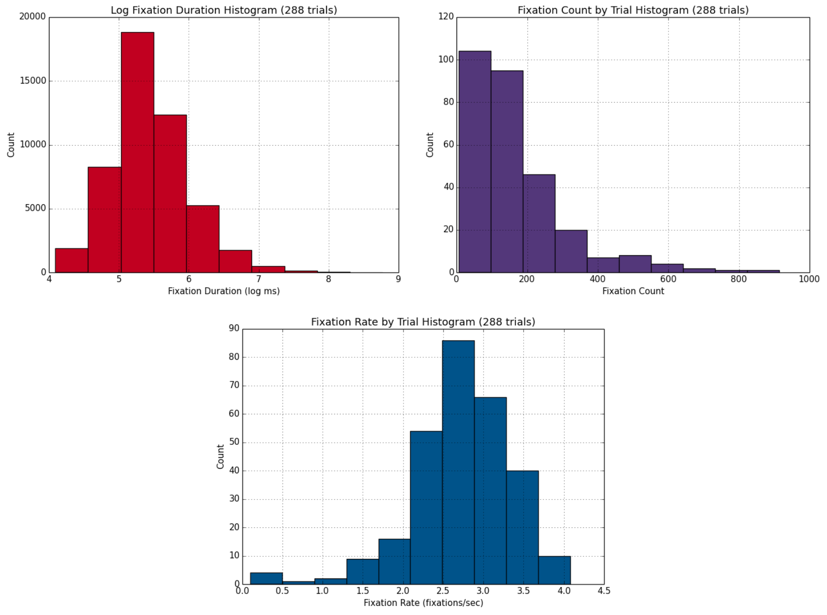


Figure 37: *Fixation metric distributions for all trials.*

Participants tended to read programs in line order (see Figure 38 for an example). The time of their first fixation (proportional to the trial duration) on each line was strongly correlated with the line number (proportional to the total number of lines in the program). In fact, line number proportion alone accounts for approximately half of the variance in a simple linear model predicting first fixation proportion ($r^2 = 0.57$). Textual line metrics, such as line length, indentation, and whitespace proportion did not significantly improve predictions. Additionally, content-based line metrics (e.g., number of identifiers, operators) did not improve the linear model's performance.

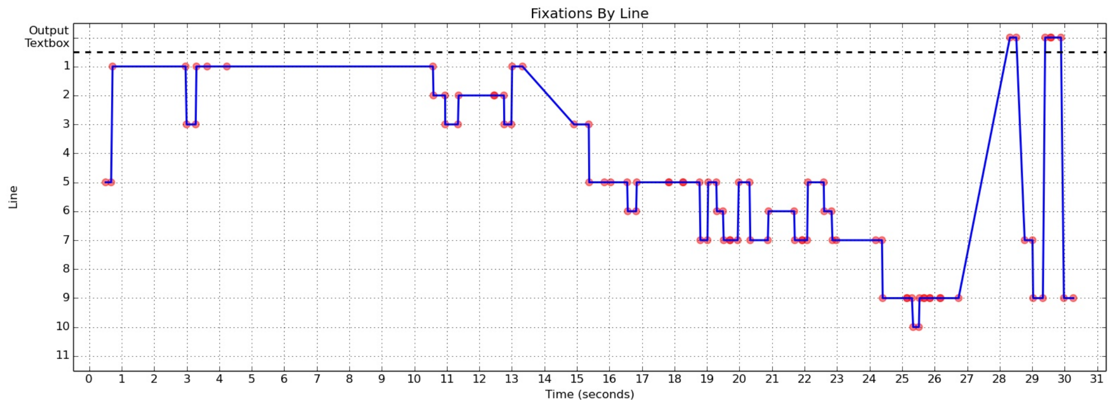


Figure 38: *Timeline for trial 268 (`overload plusmixed`). Participants read programs (mostly) in line order before responding.*

Longer lines tended to receive more time: total fixation duration per line and line length were strongly correlated ($r = 0.41$). Adding the whitespace proportion and length of the line to a simple linear model predicting fixation duration produces an r^2 of 0.54. Surprisingly, additional textual and content-based line metrics to not improve model performance significantly. This may be due to "important" program lines simply being longer, such as lists containing values necessary for calculations. The relationship between category and fixation duration may also be non-linear, resulting in a plateau in linear model performance.

While the category of a line was not a significant predictor of first fixation or total fixation duration, there are potential confounds due to patterns in the our source code. Across all 25 programs, whitespace proportion decreased with line number (i.e., textual density increased). Additionally, operator counts were positively correlated with whitespace proportion and identifiers, but negatively correlated with keyword counts. These patterns may have induced co-variances between textual and content-based metrics, which resulted in a lack of model performance improvement. A meta-study of multiple code repositories would be required to determine whether these patterns hold in general across Python code, or if they are simply an artifact of the programs in our experiment.

Based on other eye-tracking studies, we expected to find a correlation between programmers' experience level and eye movement metrics, especially mean fixation duration. Each participant's distribution of mean fixation durations per trial is shown in Figure 39. While participants are somewhat distinguishable by these distributions, they are heavily skewed. In addition, sorting by years of programming experience on the x-axis does not show evidence of a strong trend. A linear model predicting mean fixation duration from programming experience *does* produce a significant p-value (< 0.05) and a negative coefficient, but the effect size is extremely small ($r^2 = 0.021$).

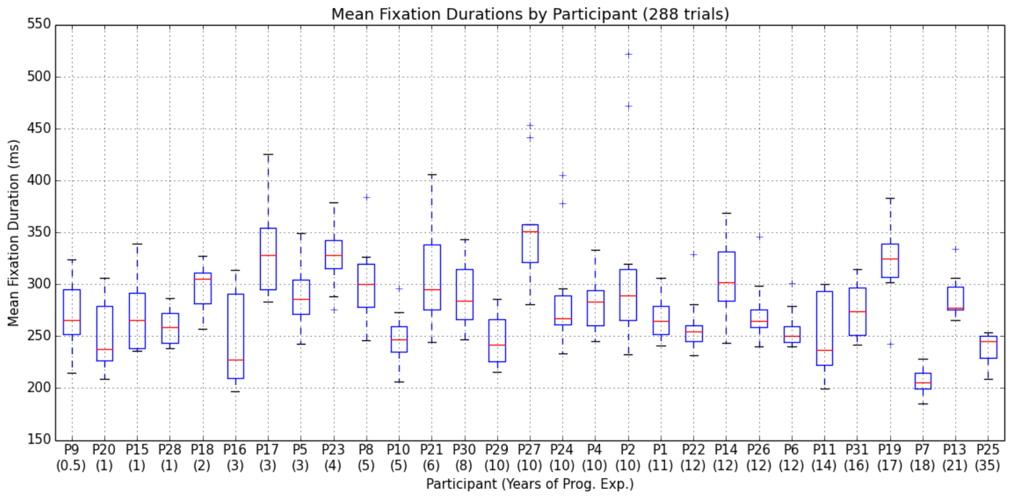


Figure 39: *Mean fixation duration for each trial, grouped by participant. Participants are ordered from least to most programming experience.*

A similar pattern is seen when looking at the number of fixations per second (fixation rate). Figure 40 shows the distributions of mean fixation rates by trial for all participants. The x-axis is again sorted by years of programming experience, but this time there is not even a statistically-relevant trend. The discussion in section 13 provides potential reasons for the mismatch between experience and fixation duration/rate – our unique task and fairly simple programs.

Uwano Review Percent and Spatial Density. Uwano et. al found that programmers tended to fixate 72.8% of code lines during the first 30% of a trial [97]. We refer to this value as the **Uwano Review Percent**. Interestingly, we had a very similar observation: across all trials, an average of 72.5% of code lines were fixated in the first 30% of a trial. However, this number appears to be largely driven by the number of lines in the trial's program. Percentages ranged from 44.4% to

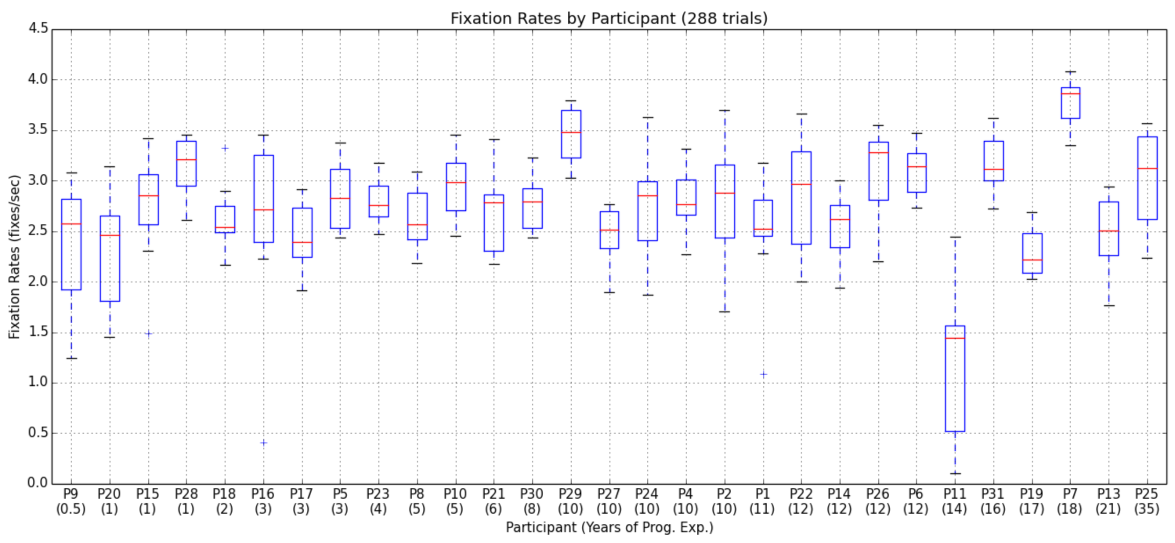


Figure 40: *Mean fixation rate for each trial, grouped by participant. Participants are ordered from least to most programming experience.*

91.1% depending on the program base and version. Additionally, a strong correlation was found between the Uwano Review Percent and the number of lines of code for individual trials ($r = -0.40$, $p < 0.05$). Thus, we cannot infer something general about a programmer's behavior, independent of the program, from the Uwano Review Percent.

We computed the **spatial density** of fixations over an envelope surrounding all lines of code for each trial. This envelope was divided into a grid whose cells were 30 pixels by 30 pixels, and the density was simply the percentage of these cells that had at least one fixation [20]. Across all programs, the mean spatial density was surprisingly consistent with a mean of 0.40 and a standard deviation of 0.07. This value is strongly correlated with the percentage of the code area that has text in it, however, suggesting the obvious – participants are not fixating empty space ($r = 0.57$, $p < 0.01$). Different grid sizes may provide more useful information, but we will leave this exercise for future work.

12.1.1 Saccade Angles

Saccades are the quick jumps between fixations, and can reveal interesting details about a task. The angle and distance between the source and destination fixation of each saccade can be plotted and compared between different tasks. When reading natural language text – e.g., a paragraph of

English – we might expect most saccades to be *short* and to the *right*, with a handful of *long* saccades to the *left* (when jumping to the next line). The right-most polar plot of Figure 41 shows precisely this: saccade angles and distances collected from a reading task in which 12 participants read and answered questions about a paragraph of English text [13]. Each dot represents a single saccade's angle (0°to the right) and length (distance from center, proportional to Euclidean distance of longest saccade).

If we examine the saccade angles between fixations over code, we find large differences with the natural language task. The left-most plot in Figure 41 contains saccades from all participants and trials in our experiment that *occurred over the program text* (i.e., excluding the output box). Angles are significantly more spread out in the vertical direction (90°and 270°), and there appear to be equal numbers of long right and left saccades. While code is often considered text when modeling program comprehension [29], it is clearly read differently than English! This aggregate view may be slightly misleading, however. If we focus on a single program base – `counting` in the middle plot of the same figure – saccades are much less vertical. This makes sense given that the `counting` programs are only 3 and 5 lines long. The English text was 4 lines long, begging the question: would a multi-paragraph natural language task produce a saccade plot like the left-most or right-most plot of Figure 41? We will leave this question for future work.

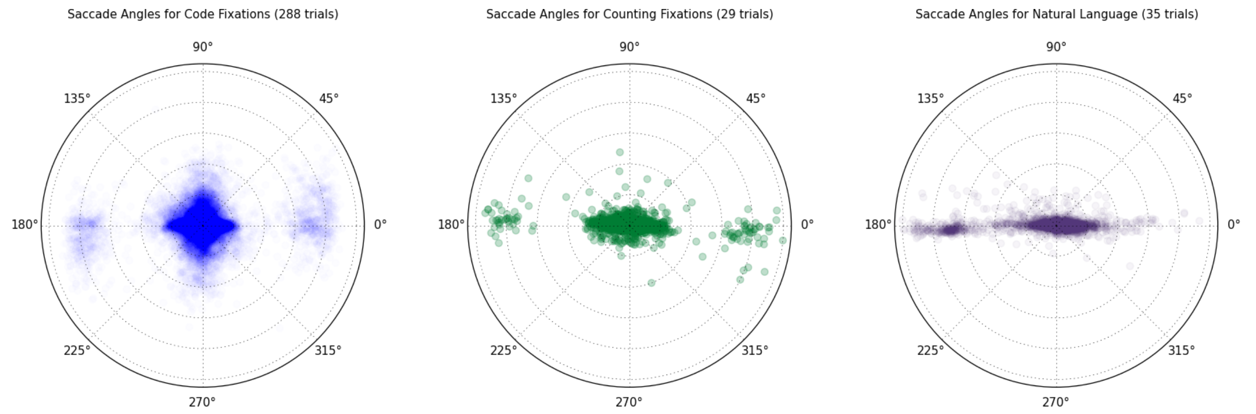


Figure 41: *Saccade angles for all code fixations (left), from just the* `counting` *programs (center), and from fixations in a natural language reading experiment (right). Distance from center is proportional to the longest saccade.*

12.1.2 Code Element Fixations

Using the Pygments lexer, we divided code elements into five categories: identifiers, operators, literals, keywords, and conditions. Identifiers are simply the names of functions and variables. Literals are lists, strings, numbers, and boolean values like `True` and `False`. Operators were either mathematical (e.g., `+`, `*`) and set operations like `in`. Conditions involved numerical comparisons, such as `<` and `>`. Across all participants and trials, Figure 42 shows the total fixation durations for each code element category, both raw totals (left) and normalized by the number of characters in each category (right). We can see that, per character, keywords receive the least amount of fixation time, followed by identifiers and literals. Operators and conditions received the most normalized fixation time – a result we should expect given the prominence of simple calculations and conditional (`if`) statements in our programs. Literals, such as lists and strings, received most of the total fixation duration in individual programs (see Section 12.2), but their large textual size puts them in the middle of the normalized plot. Our results here are mostly in line with other program comprehension experiments, though every experiment categorizes code elements slightly differently (see Discussion for details).

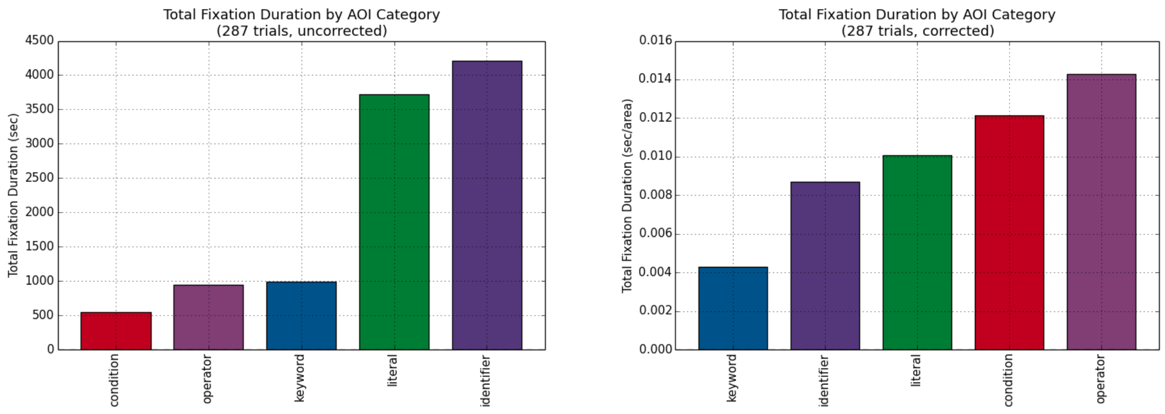
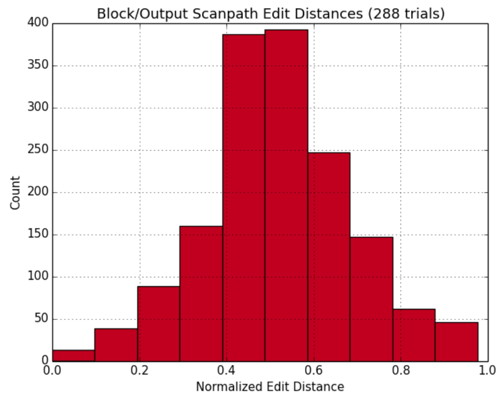


Figure 42: *Total fixation durations by code element category.* ***Left****: raw totals by category.* ***Right****: totals divided by areas of category elements across all programs.*

12.1.3 Scanpath Comparisons

In an attempt to identify common strategies, we compared the scanpaths of participants viewing the same program using the normalized Levenshtein (string-edit) distance [79]. Two kinds of scanpaths were computed: (1) between whitespace-separated blocks of code, and (2) between individual lines of code. The output box was included in both kinds of scanpaths, and repeated visits to the same area of interest were removed. For each program, clusters of scanpaths with low edit distances (e.g., below 0.25) would represent common ways of reading and evaluating the program.

We were surprised to find large differences between participant scanpaths. On average, we observed a normalized edit distance of 0.52 for block/output scanpaths and 0.65 for for line/output scanpaths (Figure 43). These values indicate that participant looking behaviors were highly individualized – i.e., a given pair of scanpaths from the same program tended to differ more than 50%. At the level of individual program versions, the mean and median edit distances were always greater than 0.25. The closest was `funcall space` with a median of 0.26.



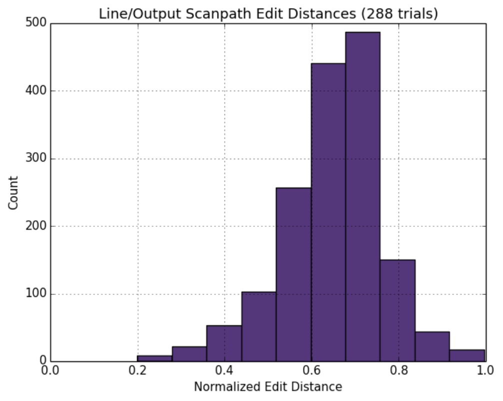


Figure 43: *Normalized edit distances between participant scanpath pairs.* ***Left****: code block and output box scanpaths.* ***Right****: code line and output box scanpaths.*

There are two likely reasons for the large difference between participant scanpaths. First, our task was fairly free-form – participants were not constrained to read or respond in any specific order. Second, there were no time constraints on individual trials, producing scanpaths with variable lengths. The edit distance metric we use is sensitive to both of these complications. Variations of the edit distance exist, with different costs for insertions, deletions, replacements, and even transpositions (e.g., `ba` to `ab`). Methods for normalizing scanpaths also exist, such as removing two character repeats (e.g., `abab` becomes `ab`) to avoid capturing refixations and simply clipping all scanpaths to the same length. Lastly, entirely different comparison algorithms can be used – e.g., ScanMatch [21], scanpath entropy [49]. Because of the large space of possibilities here, we will leave more sophisticated scanpath clustering for future work.

12.2 By Program

In this section, we break results down by individual program base and version. We begin by examining participant performance, graded by how closely their responses approximated the programs true printed output. A *perfect* grade was given if there was a match, character for character. Allowing some room for error, a *correct* grade was given to a response that matched the true output, sans some whitespace and formatting characters (e.g., commas, brackets). Figure 44 shows the proportion of correct (green) and incorrect (red) responses for each program version. It's immediately apparent that `scope`, `counting`, and `between` were the most difficult for participants. Because there were only 29 participants in our experiment, however, a statistical argument for performance differences between program versions is not feasible. Our larger study with Mechanical Turk participants aims to answer these kinds of questions.

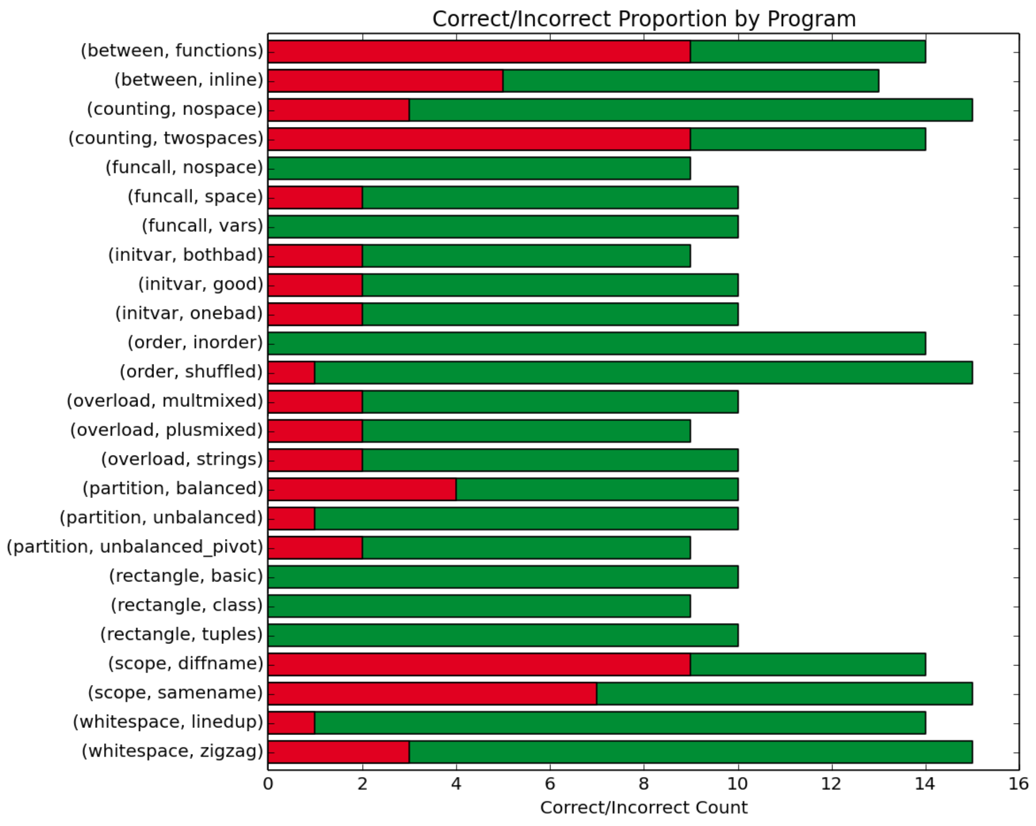
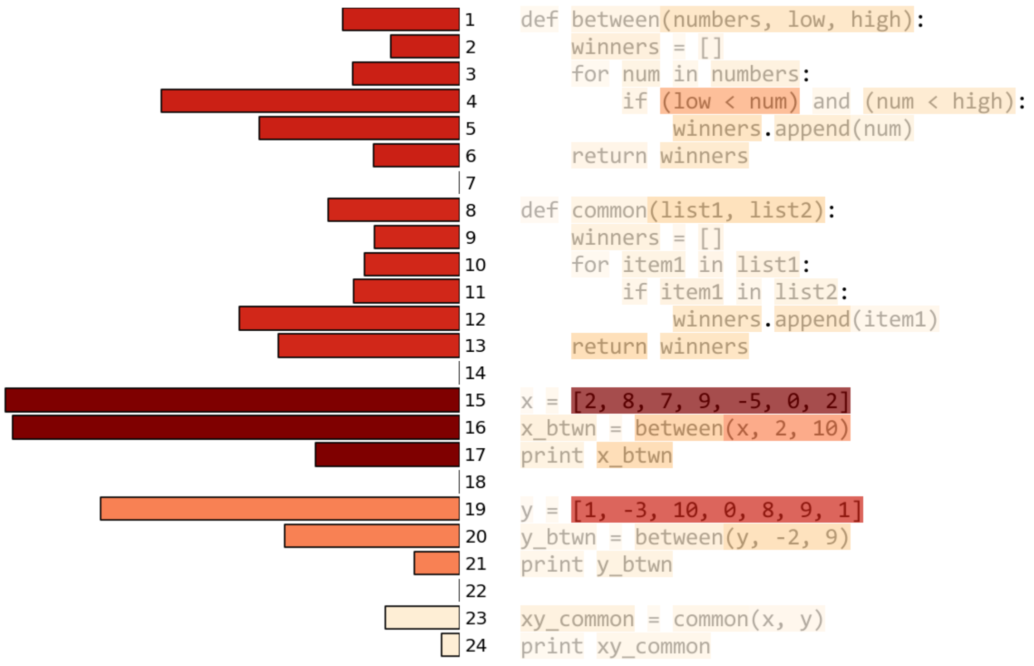


Figure 44: *Correct (green)/incorrect (red) trial proportions and counts by program version.*

12.2.1 between

The `between` programs tested the effects of pulled-out versus inline functionality (i.e., putting code into functions versus repeating it). In both versions, two lists are filtered and printed, and then the common elements in the original lists are printed. We saw several shared looking behaviors between both versions. For example, participants tended to give the majority of their fixation duration to the `x` and `y` lists, and slightly less time to the first comparison in the filtering operation. The `functions` version contained two functions (between and common), corresponding to the filtering and intersection operations performed on the two lists. Figure 45 shows where participants spent their time when looking at the code. As we might expect, most of the time was spent looking at the two lists (lines 15 and 19). Interestingly, we can see that slightly less time was spent on the second list (line 19). This pattern recurs throughout many of the programs, suggesting that participants are *faster* at evaluating the second instance of the `between` function. While this might be expected for a function call, the same pattern is observed in the `inline` version (Figure 48), where the code is duplicated.



```
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
```

Figure 45: *Total fixation duration for* `between functions` *broken down by screen region, line, and token. Dark red regions have the highest concentration of fixation time (all trials).*

Inspecting the transition matrix for `functions`, we can observe some expected behavior. The highest probability transitions *from* the output box are to lines 15-19 – where the two lists are located. Transitions *to* the output box involve these lines, but also include the `common` function call on line 23. Notable transition probabilities also occur between lines 15 and 19, something we would expect if participants were comparing both lists to find the common elements. To see the actual process of evaluation, though, we need to move beyond the aggregate transition matrix and focus on the dynamic behavior within single trials.

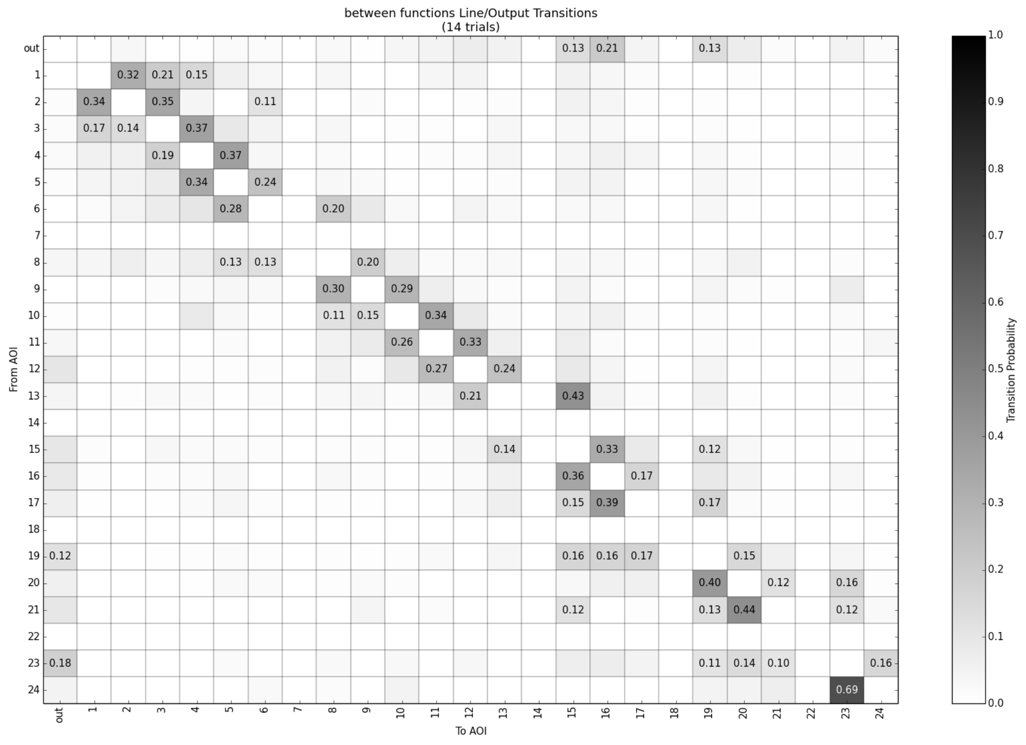


Figure 46: *Transition matrix for* `between functions` *(all participants). Probabilities below 0.1 are not annotated.*

Using a timeline of fixations on each line of code (and the output box), we can observe distinct phases of evaluation at the level of an individual trial. Figure 47 shows a trial timeline with five different sections highlighted in *blue* from left to right, representing what we believe to be the participant (1) reading the `between` function, (2) reading the `common` function, (3) filtering the first list, (4) filtering the second list, and (5) finding the common elements between lists. Additional evidence comes from this participant's intermediary responses (highlighted in yellow and listed in Table 7). These responses line up nicely with our hypothesized order of evaluation by coming at the ends of the last three sections.

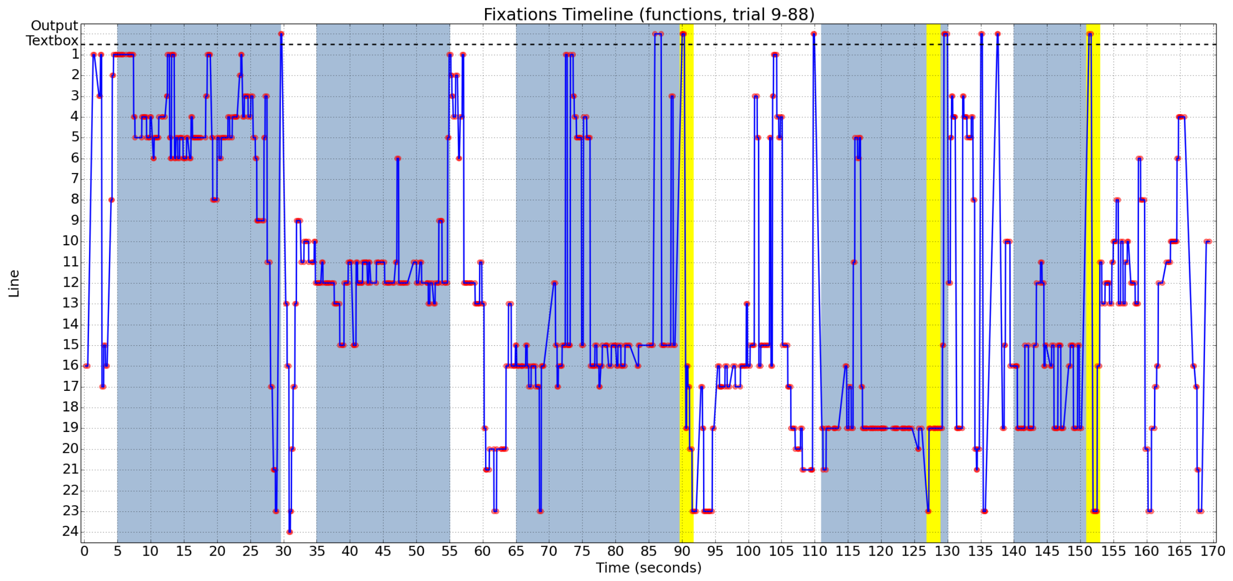
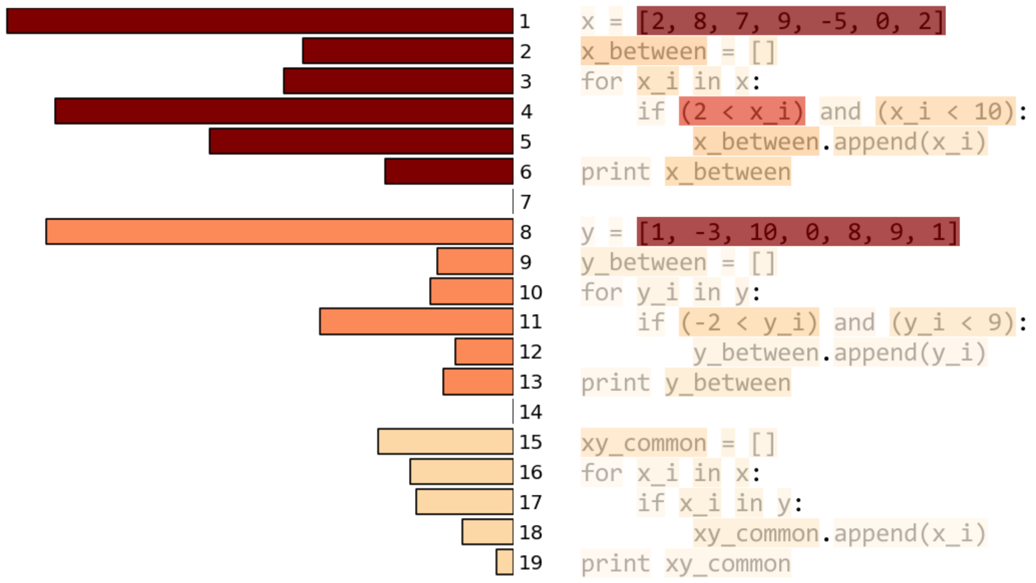


Figure 47: *Timeline of fixations by line for a single* `between functions` *trial. Regions in blue correspond to expected steps in evaluation. Yellow regions match response times from Table 7.*

time_ms	response
90689	[8,7,9]
127881	[8,7,9]●[1,0,8,1]
151920	[8,7,9]●[1,0,8,1]●[8,9,0]

Table 7: *Intermediary responses for trial 9-88 (*`between functions`*). The ● character represents newlines.*

The `inline` version did not contain any function definitions, and instead repeated code for the filtering and intersection operations. Figure 48 shows the relative amount of time spent on each line of this version. Like `functions`, less time is spent on the second instance of the filtering operation (lines 8-13), suggesting that participants recognized the repeated pattern. Unlike `functions`, however, relatively little time was spent on the intersection operation (lines 15-19) compared to the `common` function. Our larger study with participants from Amazon's Mechanical Turk found a slight, though significant, increase in correct responses for `inline` versus `functions`.



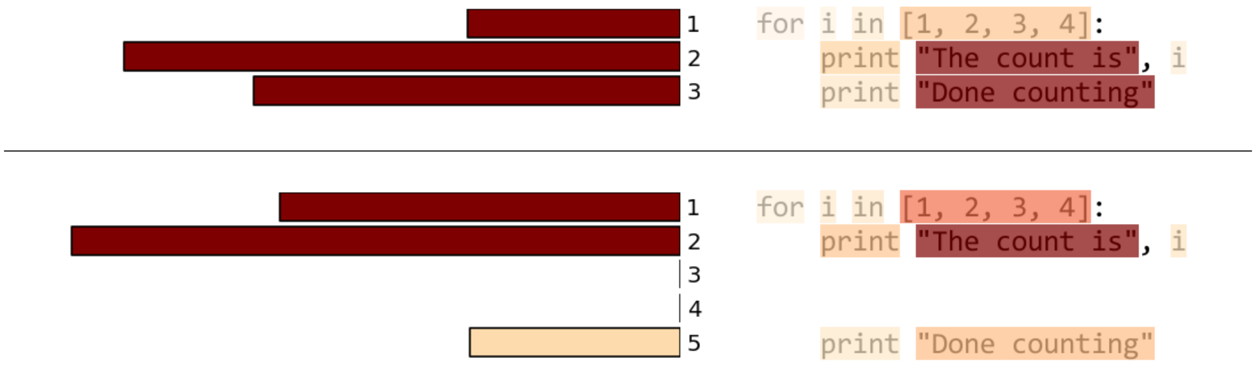
```
x = [2, 8, 7, 9, -5, 0, 2]
x_between = []
for x_i in x:
    if (2 < x_i) and (x_i < 10):
        x_between.append(x_i)
print x_between

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
```

Figure 48: *Total fixation duration for* `between inline` *broken down by screen region, line, and token. Dark red regions have the highest concentration of fixation time (all trials).*

Based on the eye movement data summary here, it seems reasonable that the intersection operation in `inline` was more recognizable than in `functions`. When inspecting response errors in the larger study, however, we found that participants were no more likely to make a mistake on the final line of output (`[8, 9, 0]`) in *either* version. This output line – corresponding to the intersection operation – was five times more likely to contain an error than the first output line, and over twice as likely than the second. A better hypothesis might be that participants simply mistook the intersection operation for something else. Over 40% of participants who answered incorrectly (in both versions) provided an `[8]` as the last output line. This mistake is consistent with assuming that the intersection operation occurs between `x_between` and `y_between` rather than `x` and `y`. Perhaps, then, participants spent less time on the intersection operation because they had (sometimes incorrect) assumptions about what it was “supposed” to do.

12.2.2 counting

The `counting` programs tested the effects of whitespace on the grouping of statements in a `for` loop. Both versions of `counting` did the same thing: print “The count is *i*” and “Done counting” for $i \in [1, 2, 3, 4]$. While the `nospace` version had the `for` loop declaration and the two `print` statements in the body on consecutive lines, the `twospaces` version added two blank lines between the first and second `print` statement. Because Python is sensitive to indentation, this did not change the semantics of the program (i.e., both `print` statements still belonged to the `for` loop). The extra whitespace in the `twospaces` version clearly had an effect: only 36% of participants provided a correct answer (as opposed to 80% correct in the `nospace` version). Proportionally, participants spent more time on the last print statement in the `nospace` version (Figure 49), likely because they were reading it during each evaluation of the loop body. This hypothesis is supported by inspecting the transition matrices for both `counting` versions (Figure 50). Participants were almost twice as likely to fixate the “Done counting” line after the first `print` statement on line 2 in the `nospace` version than in the `twospaces` version.



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

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

Figure 49: *Total fixation durations by line for both versions of the* `counting` *program (all participants). Top:* `nospace`*, bottom:* `twospaces`*.*

We can even see a difference in the `twospaces` trials alone if we split them out by correct and incorrect responses. The transition matrices reveal an interesting difference: participants that provided a correct response were about twice as likely to fixate on the “Done counting” line after line 2 (Figure 51). Examples of individual trials where this behavior was observed are shown in Figure 52. In the correct trial (top), the participant reads all lines of the program before visiting the

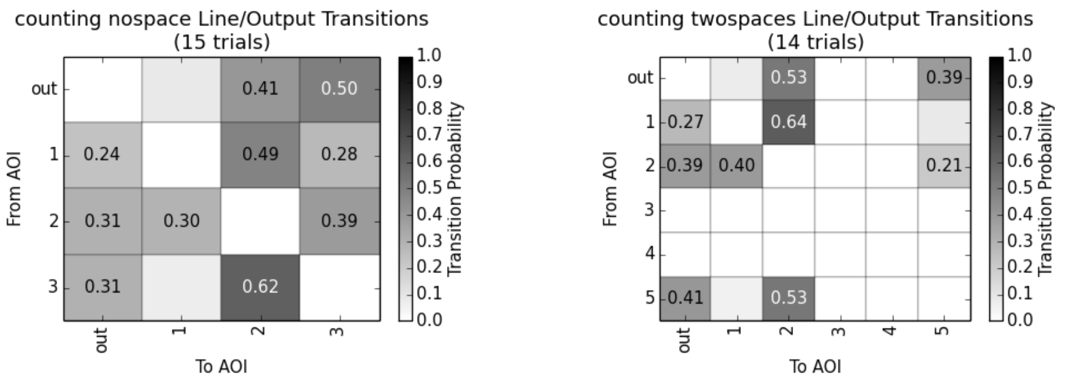


Figure 50: *Transition matrices for* `counting` *programs (all participants). Probabilities below 0.1 are not annotated.*

output box to start typing their response. The participant in the incorrect trial, however, visits the output box after mainly fixating on line 2, and tends to fixate the "Done counting" `print` statement (line 5) in isolation.

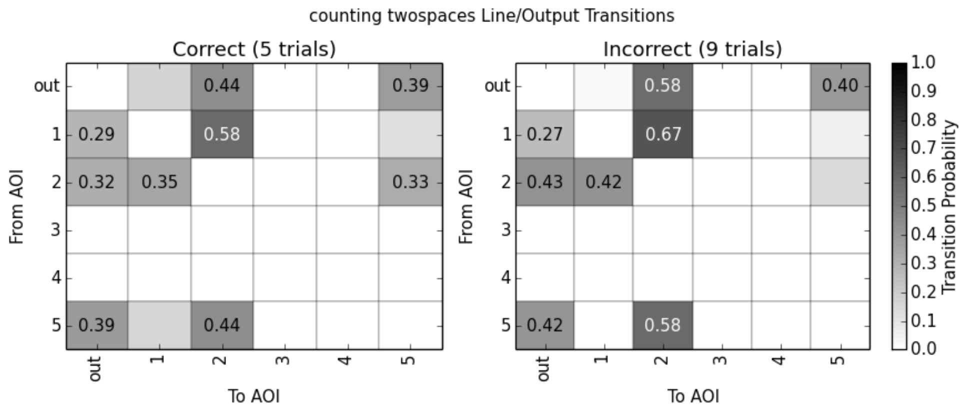


Figure 51: *Transition probabilities between lines and the output box for the* `twospaces` `counting` *program.* ***Left****: trials with correct responses.* ***Right****: trials with incorrect responses.*

The data from `counting` demonstrate that reading behavior and program interpretation are strongly linked. We are not suggesting, however, that simply failing to spend sufficient time looking at the "Done counting" line *causes* a participant to produce an incorrect response. Rather, this is likely a symptom of an underlying error when spatially grouping the `print` statements. Would the same pattern of errors occur if the entire program was embedded in a larger `for` loop or function? Further work is needed to determine which spatial cues programmers use to group code elements, and how `twospaces` is violating the assumptions behind those cues.

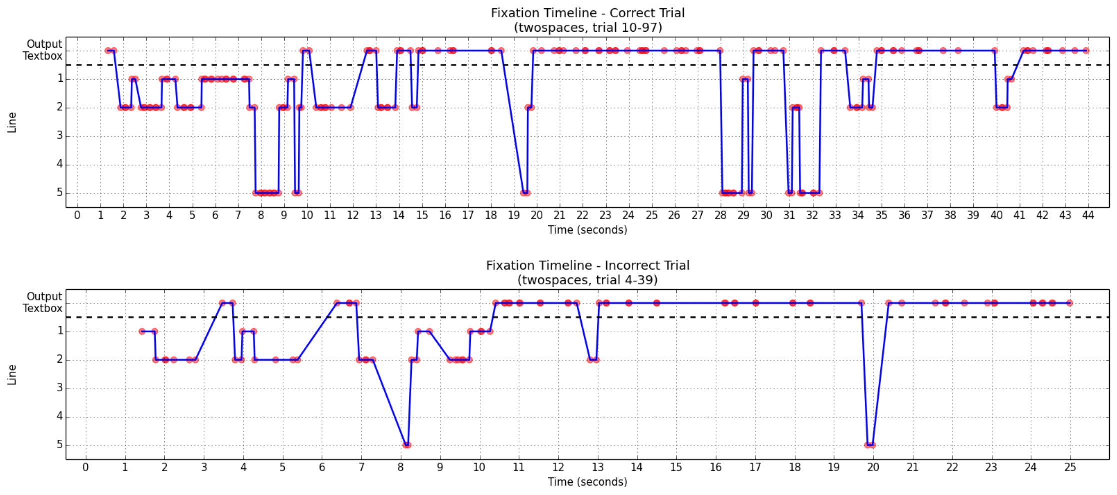


Figure 52: *Fixation timelines for two* `twospaces` *trials. The top trial resulted in a correct response, while the bottom trial did not.*

12.2.3 funcall

The `funcall` programs each performed a compound calculation using a single, simple function: $f(x) = x + 4$. The calculation, $f(1) \times f(0) \times f(-1)$, either had no whitespace between terms (`nospace` version), a single space between all tokens (`space`), or had each call to $f(x)$ bound to a variable before completing the calculation (`vars`). We expected more whitespace to facilitate faster trials and more correct responses, especially in the `vars` versions where the calculation is broken out into multiple, named steps (i.e., `x`, `y`, `z`).

Trial time and response correctness across all versions was effectively the same, violating our performance expectations. In terms of fixation time, the results are also largely the same across versions: the arguments provided to f and the + operator are focal points (Figure 53). The `return` keyword on line 2 is fixated more often than we would expect – keywords received the smallest fixation duration per area across all programs in our experiment (Figure 42). Looking at the transition matrices for each version (Figure 54), we can see that line 2 is a common destination. This is especially true from lines with f calls (line 4 in `nospace` and `space`, lines 4-6 in `vars`). We suspect that the `return` keyword served as a beacon for visual jumps back to the addition on line 2, thus increasing its share of relative fixation time.

The transition matrix for the `vars` version reveals steadily decreasing jump probabilities back to

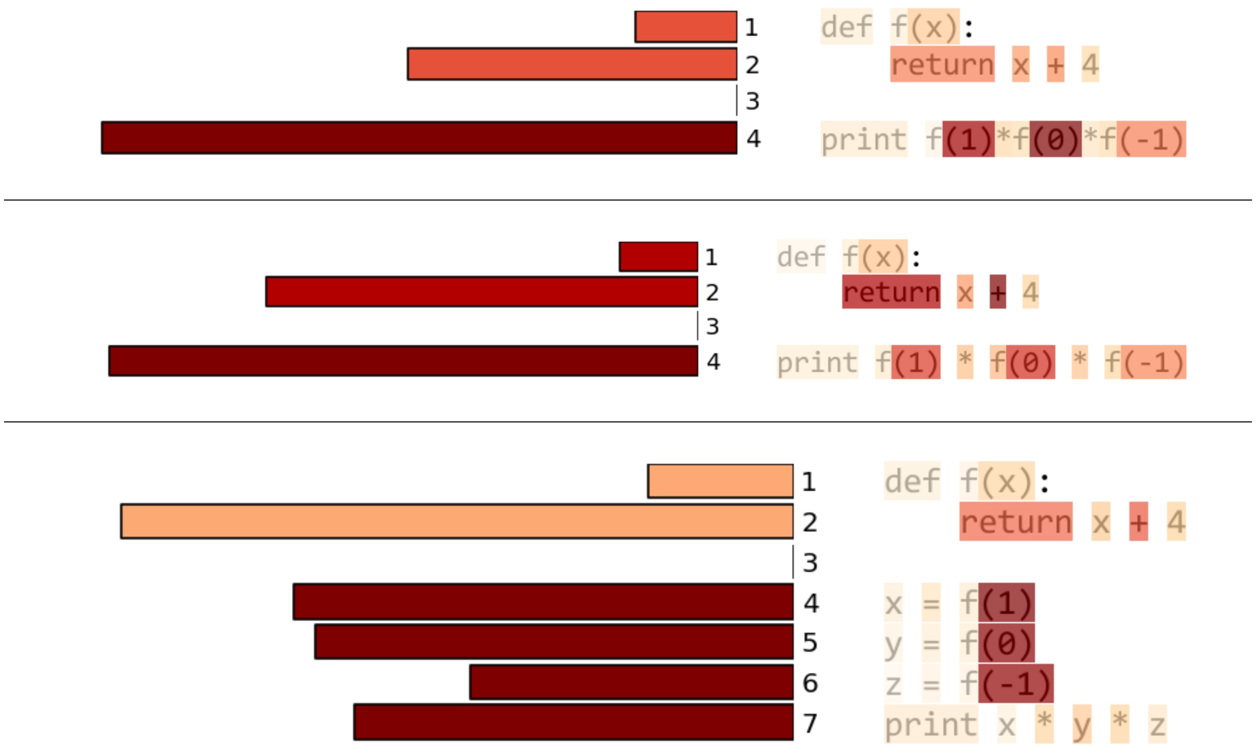
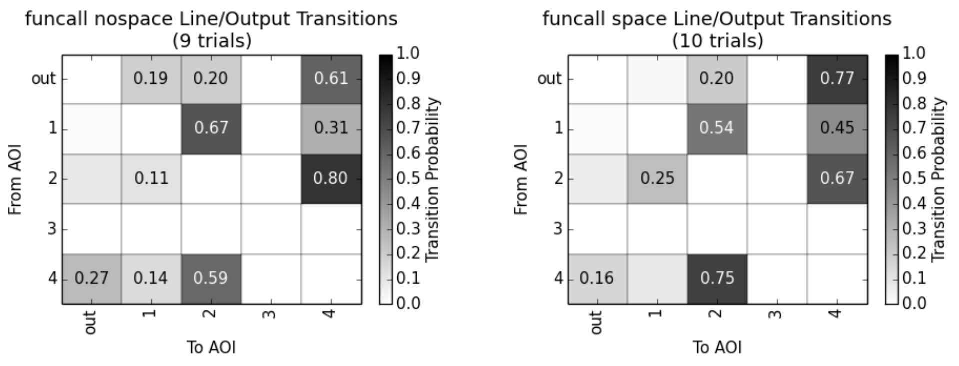


Figure 53: *Total fixation durations by line for all three versions of the* `funcall` *program (all participants). Top:* `nospace`*, middle:* `space`*, bottom:* `vars`*.*

line 2 for each call of f (lines 4-6). This suggests that participants needed to look up the definition of f less and less as they performed the calculation. A good example of this behavior is shown in Figure 55. The participant fixates lines 1 and 2 three times in the first five seconds, and twice in the next five seconds. By the end of the trial, only the output box is being fixated, presumably as the participant calculates the printed product. Looking at the participant's keystrokes partially confirms this hypothesis – an intermediary result of 12×5 is typed first before being replaced by a 60 at about 26.8 seconds.

Let's dig just a little deeper. Looking at the average fixation duration over a rolling window across the same trial, we can see two spikes around 17 and 27 seconds (Figure 56). Given the rolling window delay, these events likely correspond to the initial computation of the three additions ($1 + 4$, $0 + 4$, $-1 + 4$), and the later computation of the final product ($5 \times 4 \times 3$). Similar correspondence between average fixation duration over a rolling window and mental computation was found in the `initvar` programs (Section 12.2.4).



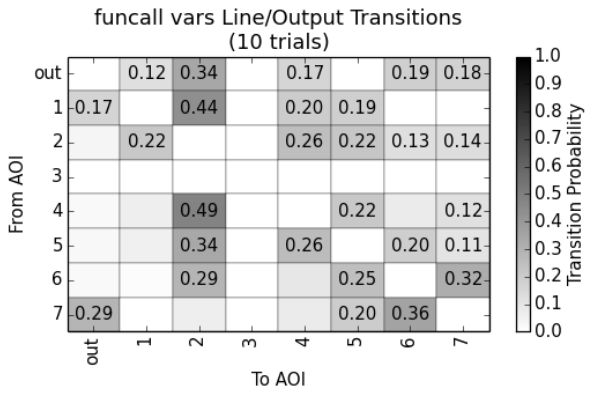


Figure 54: *Transition matrices for* `funcall` *programs (all participants). Probabilities below 0.1 are not annotated.*

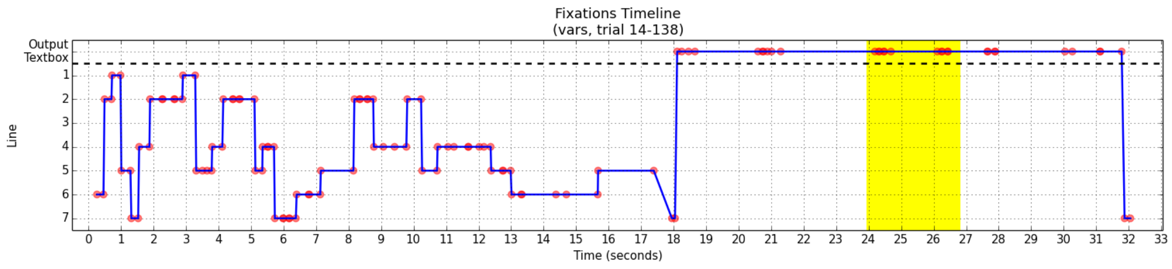


Figure 55: *Fixation timeline for* `vars` *trial. The highlighted region (yellow) goes from the participant's first keystroke to the last.*

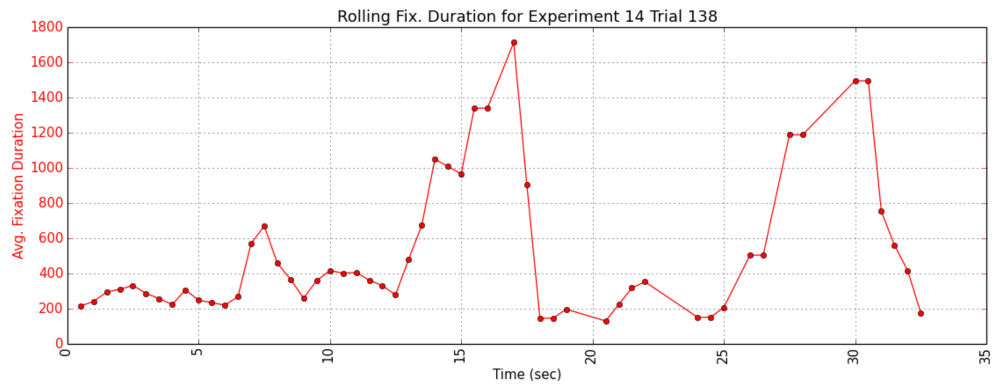


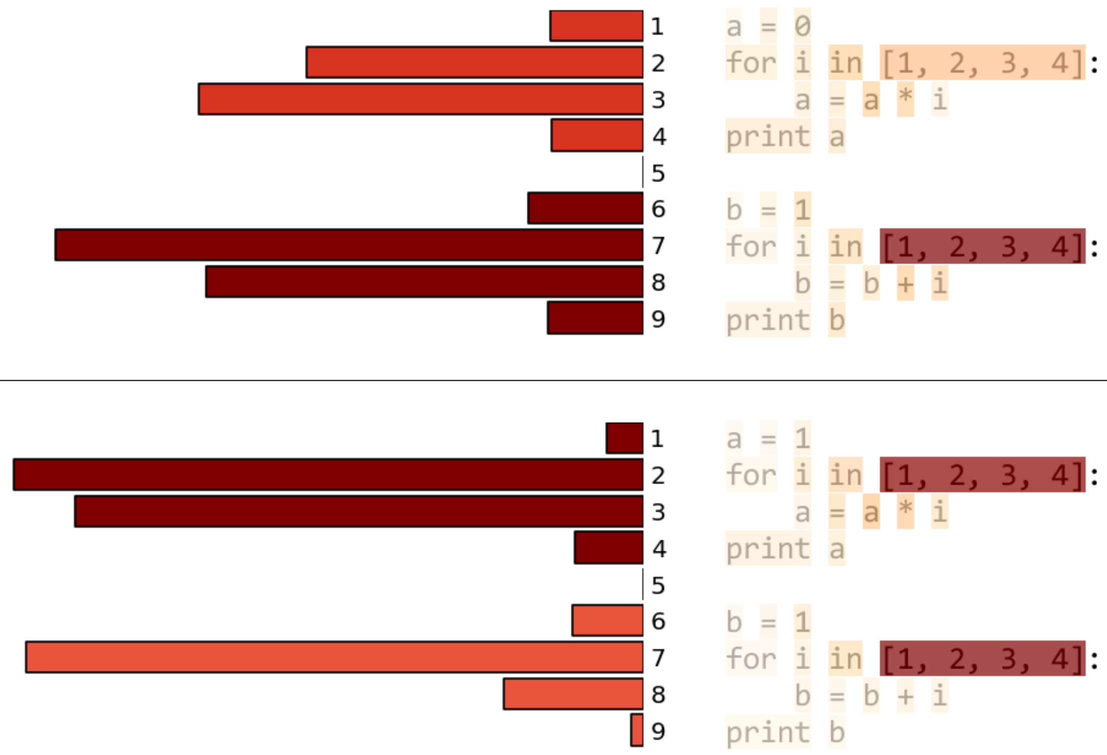
Figure 56: *Average fixation duration over a 1 second rolling window (1/2 second step).*

12.2.4 initvar

The `initvar` programs each contained two accumulation loops: one performing a product, and the other performing a summation. In the `good` version, both loops were intended to meet expectations; the product loop had an initial value of 1, and the summation loop had an initial value of 0. The `onebad` version, however, started the summation loop at 1 (an off-by-one error). The `bothbad` version contained the same error, and also started the product loop at 0 (making the final product 0).

The fixation durations and transition matrices for all versions were fairly similar, except for one major difference: participants in the `bothbad` version spent much less time on the list in the first `for` loop (Figure 57). This provides evidence for our suspicions that participants would "short-circuit" the product calculation by noting that $a = 0$, and thus any product with a would also be 0. Performance-wise, this resulted in a significant increase in response proportion – i.e., a higher proportion of `bothbad` trials were spent responding.

We expected the `bothbad` transition matrix (Figure 58) to have a large transition probability from lines 2 or 3 to the output box, representing a short-circuited calculation (and entry of a "0" by the participant). We did not observe this in the aggregate across all `bothbad` trials, but the behavior is evident in some of the individual trials. For example, Figure 59 shows the timeline of trial 59 (participant 6). The red line shows when the participant first typed a "0" in the output box, which occurs immediately after looking at the first loop. We can be confident this represents a short-circuited calculation because this participant's keystrokes (Table 8) contained intermediary



```
a = 0
for i in [1, 2, 3, 4]:
    a = a * i
print a

b = 1
for i in [1, 2, 3, 4]:
    b = b + i
print b
```

```
a = 1
for i in [1, 2, 3, 4]:
    a = a * i
print a

b = 1
for i in [1, 2, 3, 4]:
    b = b + i
print b
```

Figure 57: *Total fixation durations by line for two versions of the* `initvar` *program (all participants). Top:* `bothbad`*, bottom:* `onebad`*.*

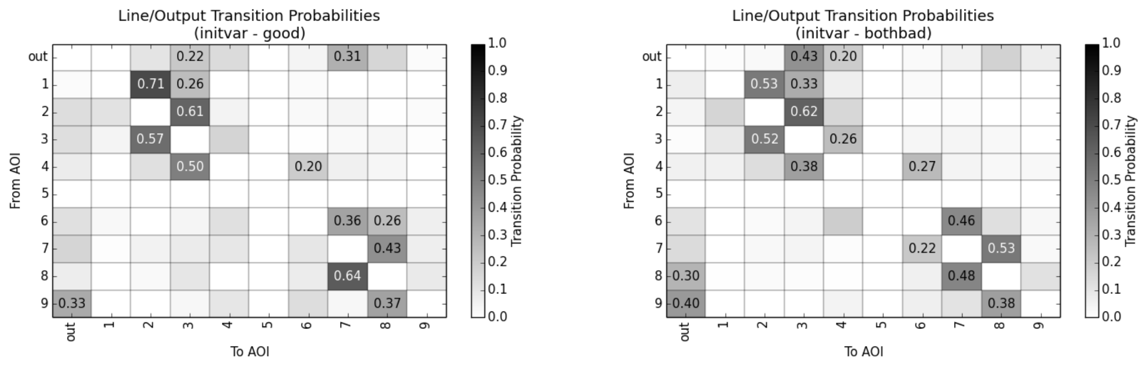


Figure 58: *Transition matrix for* `good` *and* `bothbad` *trials. Probabilities below 0.1 are not annotated.*

time_ms	response
17848	0
19294	0●
40125	0●1
43821	0●1+
45245	0●1+1
46765	0●1+1+
50837	0●1+1+3
51853	0●1+1+2
52205	0●1+1+2+
54669	0●1+1+2+3
55182	0●1+1+2+3+
55645	0●1+1+2+3+4
64188	0●1+1+2+3+4
64332	0●1+1+2+3+4 =
64836	0●1+1+2+3+4 =
70900	0●1+1+2+3+4 = 1
71037	0●1+1+2+3+4 = 11
81000	0●11

Table 8: *Time series of responses for trial 59, participant 6 (`initvar bothbad`). A ● represents a new line.*

results for the remaining calculation.

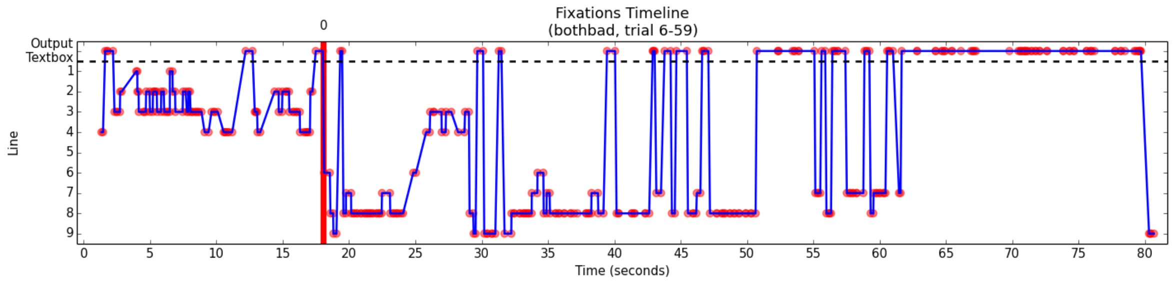


Figure 59: *Fixation timeline for `bothbad` trial. The red line indicates when a "0" response was given.*

Another source of evidence for a short-circuited calculation in the same trial comes from the average fixation duration over a rolling window (Figure 60). The largest spikes occur right around when the participant has finished typing the operands for the final calculation (60-70 sec), and **not** when looking at the first calculation. Interestingly, a small spike occurs around the 50 second mark – right when the participant corrects an error in their response. Average fixation duration may indicate intermediary errors as well as mental calculations.

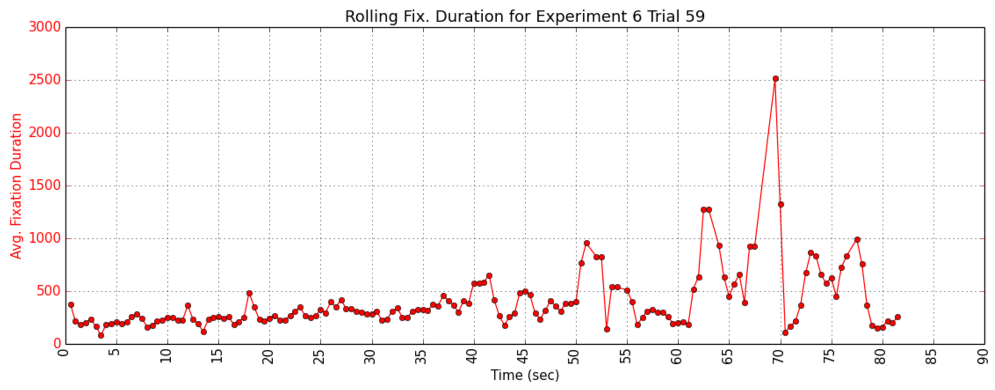


Figure 60: *Average fixation duration over a 1 second rolling window (1/2 second step).*

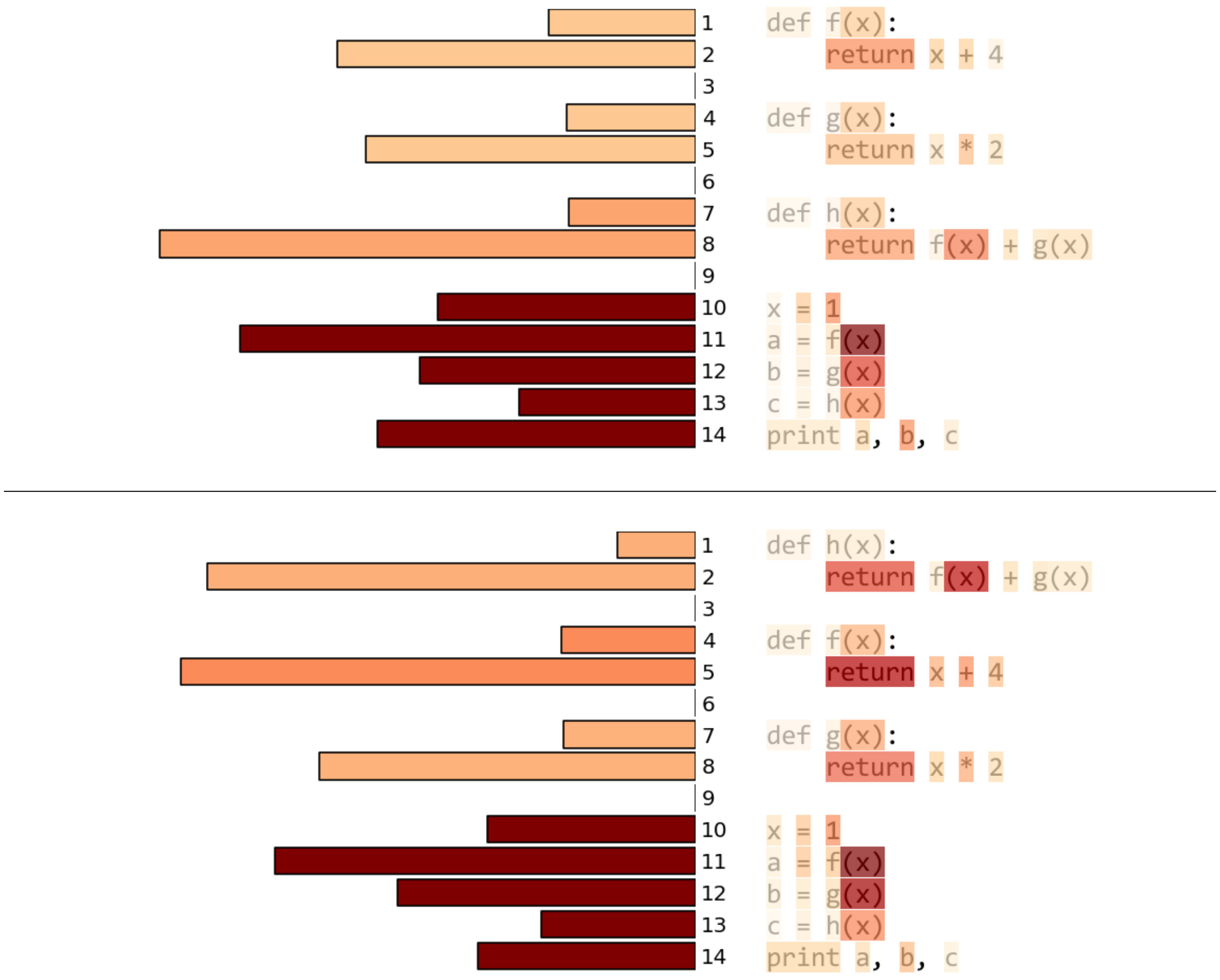
12.2.5 order

The `order` programs contained three functions, $f(x)$, $g(x)$, and $h(x)$. f and g added 4 and doubled x respectively, and h computed $f(x) + g(x)$. In the main body, all three functions were called on $x = 1$ in the same order: f, g, h. The order in which the three functions were defined was either in the called order (`inorder`), or in a slightly different order (`shuffled`) – h, f, g.

We found that participants in the `shuffled` trials took slightly longer to provide responses, presumably because there were ordering expectations for the definitions of f, g, and h. The visualizations of where participants spent their time in both versions are very similar, with the exception that a bit more time was spent in `shuffled` on the `return` keywords inside each function (Figure 61). Like the `funcall` programs, we suspect that these code elements served as beacons for quickly locating the function bodies. The implicit ordering of functions in the `inorder` version may have lessened the need for such beacons. This hypothesis is not quantifiable, however, without further experimentation.

Surprisingly, we did not see differences between versions in the transition matrices, or in metrics related to visual search efficiency, such as normalized scanpath length, fixation rate, spatial density, and average saccade length. Given the observed difference in trial times in our larger study (`inorder` participants were slightly faster), we expect that there exists *some* quantifiable method for distinguishing between participants' eye movements in both versions. We may need more participants, however, to locate a distinguishing feature.

Looking at the timelines for individual trials, similar behaviors can be easily noted (Figure 62). In

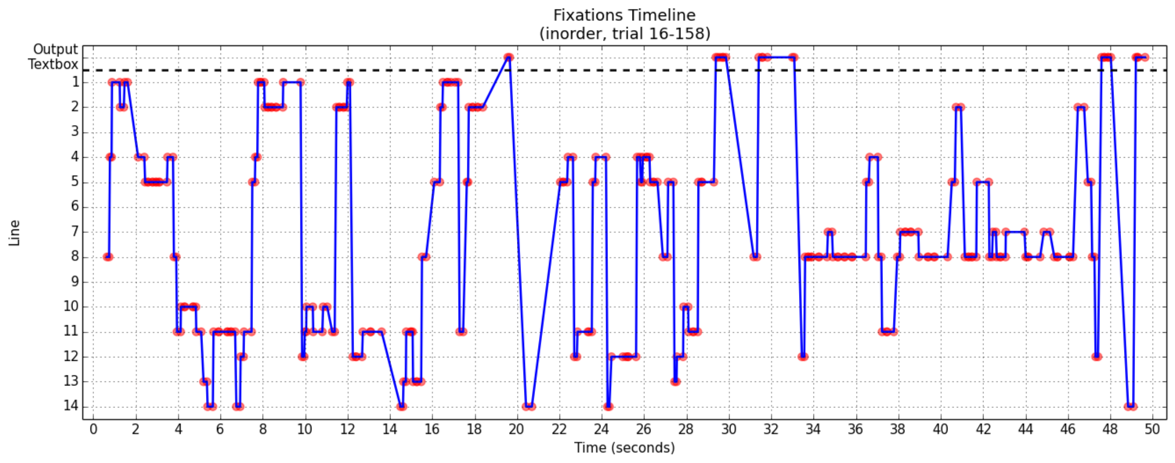


```
def f(x):
    return x + 4

def g(x):
    return x * 2

def h(x):
    return f(x) + g(x)

x = 1
a = f(x)
b = g(x)
c = h(x)
print a, b, c
```

```
def h(x):
    return f(x) + g(x)

def f(x):
    return x + 4

def g(x):
    return x * 2

x = 1
a = f(x)
b = g(x)
c = h(x)
print a, b, c
```

Figure 61: *Total fixation durations by line both versions of the* `order` *program (all participants). Top:* `inorder`*, bottom:* `shuffled`*.*

the `inorder` trial (top), we can see fixations move from lines 1-2 (definition of f) to lines 4-5 (definition of g), and finally to lines 7-8 (definition of h) with occasional jumps back to line 2 (f). Similarly, the `shuffled` trial (bottom) starts on lines 4-5 (f), goes to lines 7-8 (g), and ends with lines 1-2 (h) plus the occasional jump to line 4 (f). The shuffled ordering does not appear to heavily impact visual search or transitions to the output box.



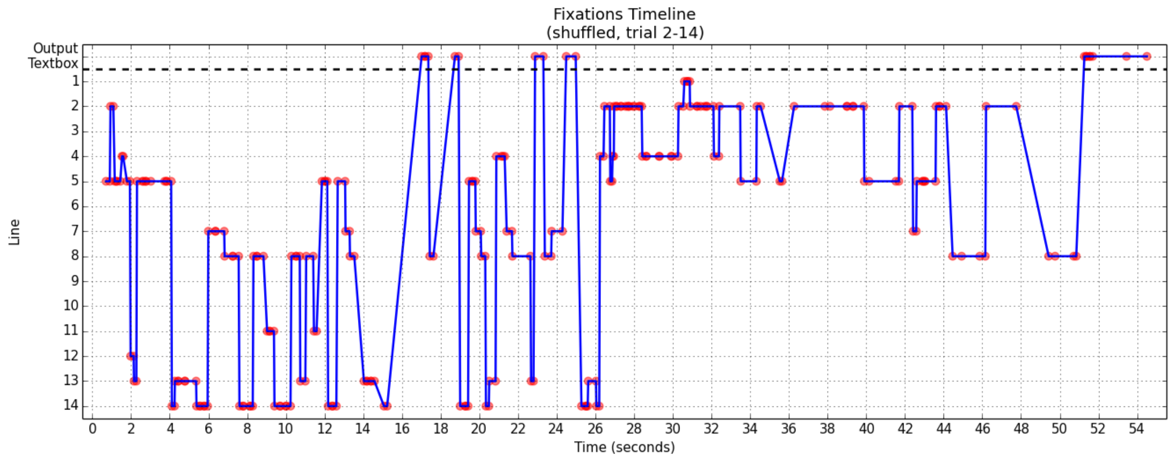
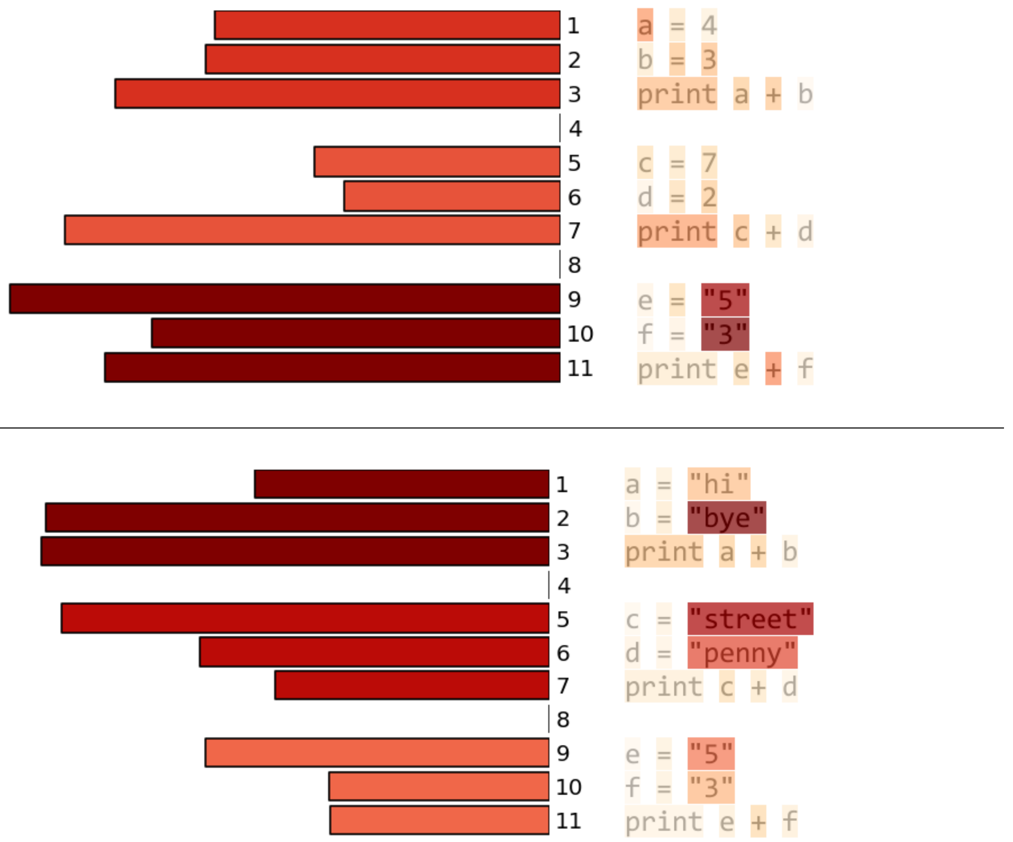


Figure 62: *Timeline of line and output box fixations for two `order` trials. Top: `inorder`, bottom: `shuffled`.*

12.2.6 overload

The `overload` programs tested the effects of operator overloading – i.e., having multiple meanings for the same operator. Each program had three blocks of code, each with two variable assignments followed by an operation with those two variables (and a printing of the result). The final block always assigned a string "5" and string "3" to variables `e` and `f`, and then printed `e + f`. The + operator in Python is *overloaded*, and can either be addition or string concatenation depending on the types of its operands. The preceding two code blocks either had exclusively multiplications (`multmixed`), additions (`plusmixed`), or string concatenations (`strings`).

We expected participants to be "surprised" more by the final + operation (string concatenation) in the `plusmixed` version than the others due to being primed with the mathematical sense of the operator. We saw the largest differences between `plusmixed` and `strings`: participants focused much more on lines 9-11 in `plusmixed`, specifically on the string values and + operator (Figure 63). The results for `multmixed` were similar to `plusmixed`, though not quite as distinct from `strings`.



```
1   a = 4
2   b = 3
3   print a + b
4
5   c = 7
6   d = 2
7   print c + d
8
9   e = "5"
10  f = "3"
11  print e + f
```

```
1   a = "hi"
2   b = "bye"
3   print a + b
4
5   c = "street"
6   d = "penny"
7   print c + d
8
9   e = "5"
10  f = "3"
11  print e + f
```

Figure 63: *Total fixation durations by line for two versions of the* `overload` *program (all participants). Top:* `plusmixed`*, bottom:* `strings`*.*

It's unclear whether participants were more "surprised" by the string numerals ("5" and "3") or the + operator. While the transition matrices for `plusmixed` and `strings` were not drastically different (Figure 64), we did see a slightly higher probability of going from the output box to lines

9-11 in the `plusmixed` version (0.54 versus 0.25). This suggests that participants were more likely to go back and check the string concatenation block of code after responding when it was proceeded by additions. A good example of this behavior is shown in the timeline in Figure 65. After typing the final line of their response (“53”), as indicated by the yellow region, the participant fixates the output box before returning to the final code block. Interestingly, the participant also returns to line 7 (`print c + d`), perhaps to double-check that previous additions were not actually string concatenations. It appears that some priming is going on for the + symbol, and that having multiple senses of an operator in the same small program is enough to affect eye movements.

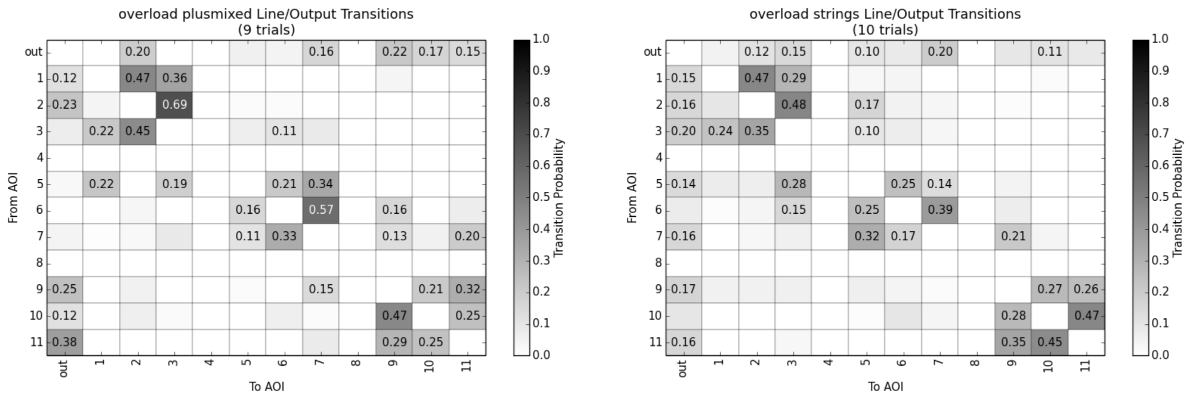


Figure 64: *Transition matrices for* `overload` *programs (all participants). Probabilities below 0.1 are not annotated.*

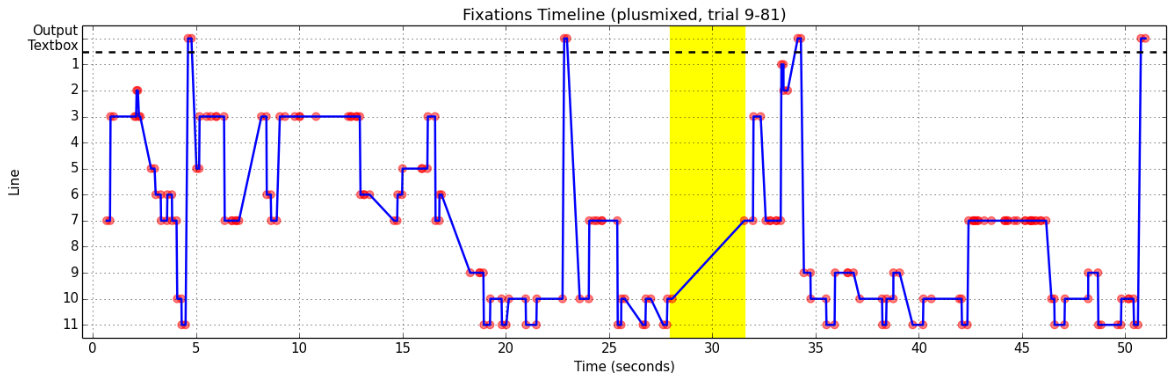
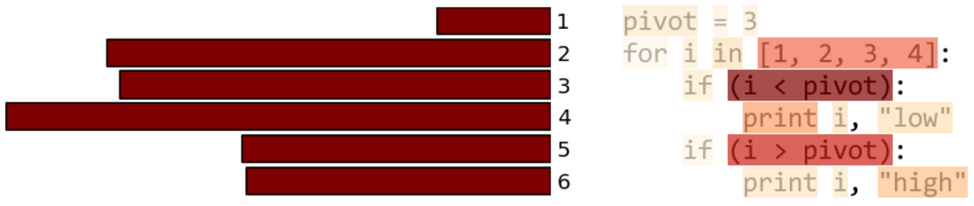


Figure 65: *Fixation timeline for a* `plusmixed` *trial. The yellow region indicates when the participant was typing “53”.*

12.2.7 partition

The `partition` programs each iterated through a list of numbers, and printed the number plus a "high" or "low" designation on each line. Numbers less than 3 were *low*, and numbers greater than 3 were *high* (3 itself was skipped). The `balanced` version iterated over $[1, 2, 3, 4, 5]$, producing an equal number of *low* and *high* numbers. In contrast, the `unbalanced` and `unbalanced_pivot` versions iterated over $[1, 2, 3, 4]$, printing two *low* and only one *high*. The `unbalanced_pivot` version used variable `pivot = 3` instead of the constant 3 in its `if` statements.

We expected participants to make fewer errors in the `balanced` version because of the symmetric *low* and *high* values. No systematic difference between versions was observed, however. In fact, the most common error was simply forgetting to print the number (`i`) alongside the low/high label. There were no major differences between relative fixation durations either – participants in all three versions spent the majority of their time on the list and two conditionals (Figure 66).



```
1  pivot = 3
2  for i in [1, 2, 3, 4]:
3      if (i < pivot):
4          print i, "low"
5      if (i > pivot):
6          print i, "high"
```

Figure 66: *Total fixation durations by line for the* `unbalanced_pivot` *version of* `partition`*. Other versions had similar results.*

The transition matrices for all versions were also very similar, but contained an interesting pattern. Figure 67 shows the transition matrix for the `unbalanced_pivot` trials. Lines 2-4 form a looping core in the matrix, with high probability transitions going to the next and previous lines. From this, we might expect participants to snake their fixations up and down these lines as they evaluate the program. Indeed, this behavior is observable in the fixation timelines – e.g., Figure 68.

This timeline also helps explain the transition probabilities coming out of line 1 (`pivot = 3`). The highlighted portions correspond to the participant's entry of each response line (`1 low`, `2 low`, and `4 high`). We can see several quick jumps up to line 1, one before each entry of the first and second response lines. Line 1 receives few visits, so the high probability transitions from it to lines 2 and 4 are likely due to (1) most participants reading the program in line order, and (2) referencing the variable's value while evaluating the `print` statement on line 4. The latter behavior is referred to as

tracing in Cant et. al's cognitive model of program comprehension [16] and, along with `chunking`, encompasses much of basic program comprehension.

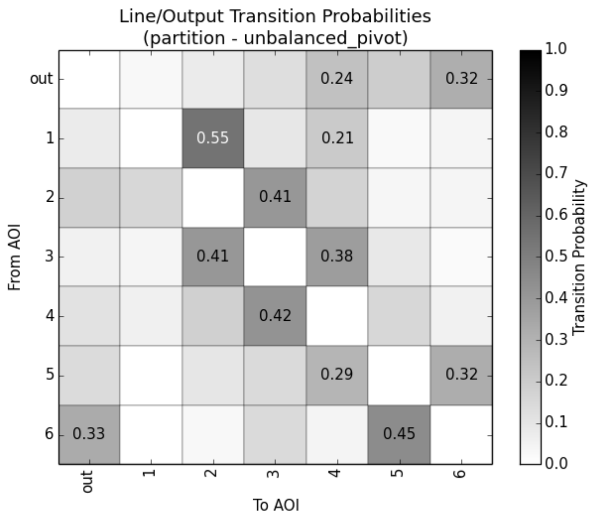


Figure 67: *Transition matrix for all* `unbalanced_pivot` *trials. Probabilities below 0.1 are not annotated.*

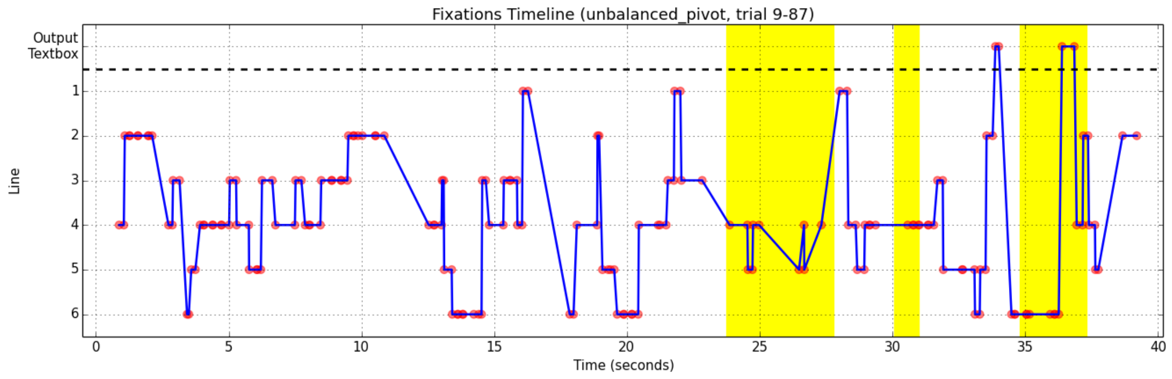
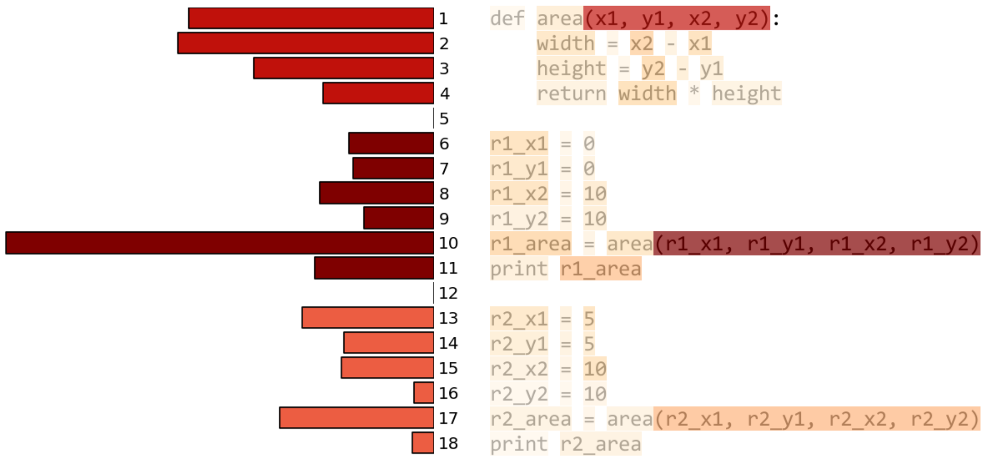


Figure 68: *Fixation timeline for an* `unbalanced_pivot` *trial. The yellow regions correspond to the typing of the first, second, and third response line.*

12.2.8 rectangle

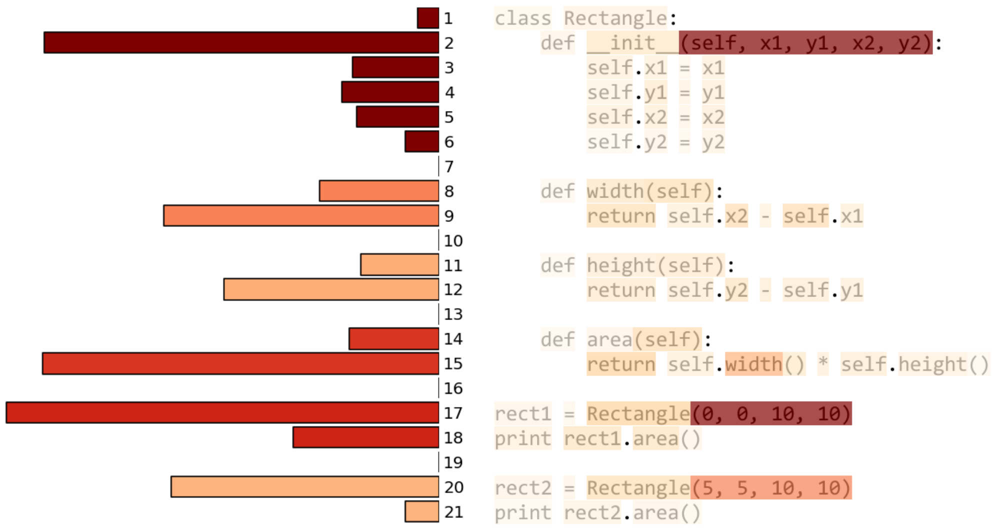
The `rectangle` programs computed the area of two rectangles, represented either as a collection of four `x`/`y` variables (`basic`), a `Rectangle` object (`class`), or a pair of `(x, y)` coordinates (`tuples`). All 29 participants produced a correct response to the `rectangle` programs, and we did not find any significant difference in performance metrics between versions in the present or larger study (with Mechanical Turk participants).

Despite the different forms of the `rectangle` programs, the eye-tracking visualizations reveal strong similarities. In general, participants focused on the ordering of arguments (either to the `area` function or the `Rectangle` class constructor), and on the numeric values for width and height (Figures 69, 70, 71). Interestingly, less time was always spent on the second set of arguments/values, providing evidence for short-term learning of the area operation. Similar behavior was observed for the filtering operation in the `between` programs (Section 12.2.1).



```
def area(x1, y1, x2, y2):
    width = x2 - x1
    height = y2 - y1
    return width * height

r1_x1 = 0
r1_y1 = 0
r1_x2 = 10
r1_y2 = 10
r1_area = area(r1_x1, r1_y1, r1_x2, r1_y2)
print r1_area

r2_x1 = 5
r2_y1 = 5
r2_x2 = 10
r2_y2 = 10
r2_area = area(r2_x1, r2_y1, r2_x2, r2_y2)
print r2_area
```

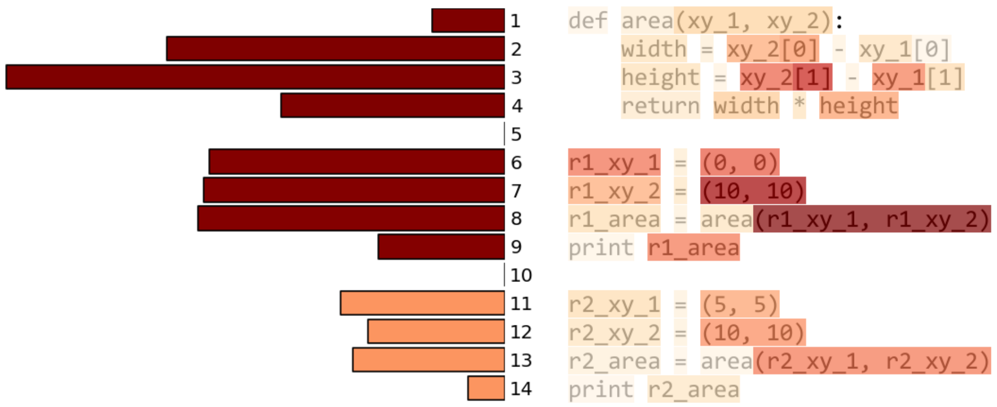
Figure 69: *Total fixation durations by line the* `basic` *version of* `rectangle`*.*



```
class Rectangle:
    def __init__(self, x1, y1, x2, y2):
        self.x1 = x1
        self.y1 = y1
        self.x2 = x2
        self.y2 = y2

    def width(self):
        return self.x2 - self.x1

    def height(self):
        return self.y2 - self.y1

    def area(self):
        return self.width() * self.height()

rect1 = Rectangle(0, 0, 10, 10)
print rect1.area()

rect2 = Rectangle(5, 5, 10, 10)
print rect2.area()
```

Figure 70: *Total fixation durations by line the* `class` *version of* `rectangle`*.*

The `tuples` version was distinguishable by participants' use of their time on lines 2 and 3, especially around the xy_2 operand. The extra time was likely used to verify the calculations of `width` and `height`, which could easily have been wrong if the indices provided to `xy_1` and `xy_2` were reversed. The corresponding lines in the `basic` and `class` versions were not fixated as much, suggesting that the calculation verification was easier with flat variable names (e.g., `x1`, `x2`).



```
1  def area(xy_1, xy_2):
2      width = xy_2[0] - xy_1[0]
3      height = xy_2[1] - xy_1[1]
4      return width * height
5
6  r1_xy_1 = (0, 0)
7  r1_xy_2 = (10, 10)
8  r1_area = area(r1_xy_1, r1_xy_2)
9  print r1_area
10
11 r2_xy_1 = (5, 5)
12 r2_xy_2 = (10, 10)
13 r2_area = area(r2_xy_1, r2_xy_2)
14 print r2_area
```

Figure 71: *Total fixation durations by line the `tuples` version of `rectangle`.*

The transition matrix for `tuples` exemplifies the "block" structure observed in the matrices of all three versions (Figure 72). Clusters of high transition probabilities closely follow the `area` function body and two area computations. Two additional transition probabilities stand out here: (1) from line 9 to 11, and (2) from the output box to line 11. These seem likely to correspond to different evaluation strategies – either reading all the way through the program or responding to the first area computation before continuing. Inspecting individual trial timelines, we can indeed find examples of both! Figure 73 shows two examples, with the highlighted regions corresponding to the participants' response periods. On top, we see a participant read the entire program from top to bottom before responding. On the bottom, there are distinct reading/response phases for both area computations. Note also that the `area` function (lines 1-4) is not fixated during the second computation; another example of short-term learning.

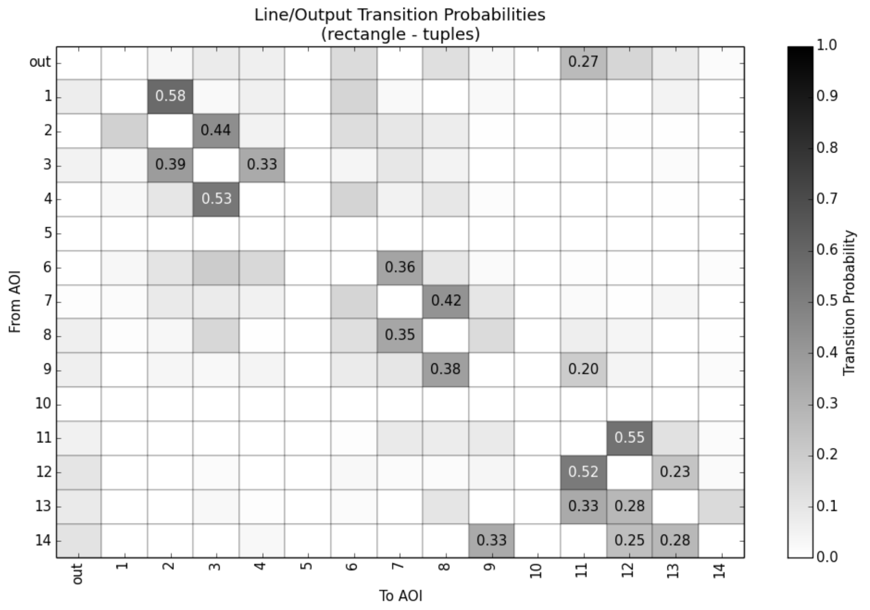
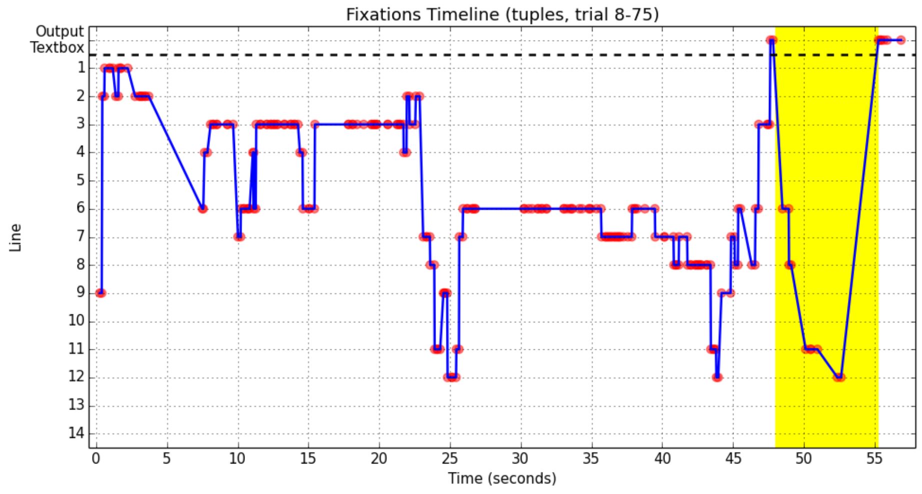


Figure 72: *Transition matrix for all `tuples` trials. Probabilities below 0.1 are not annotated.*



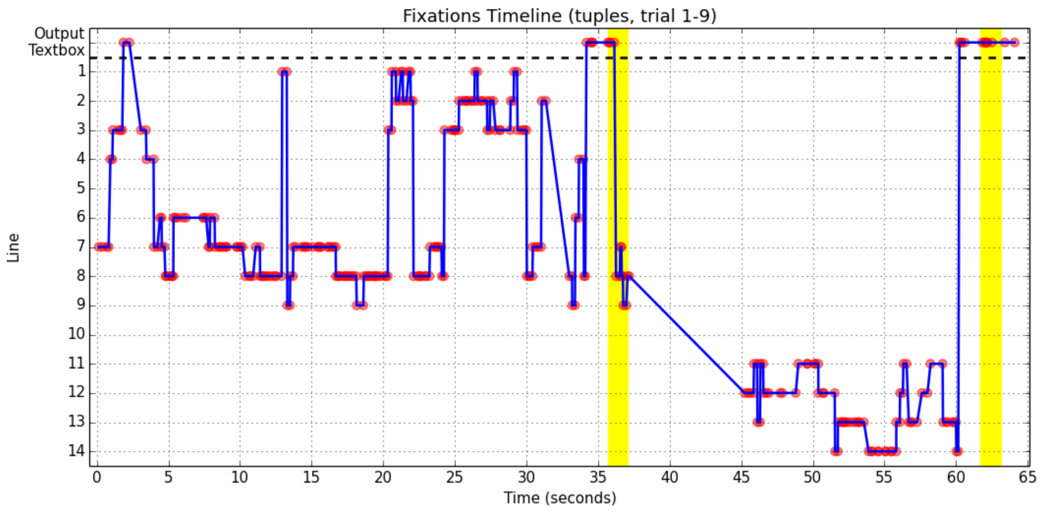


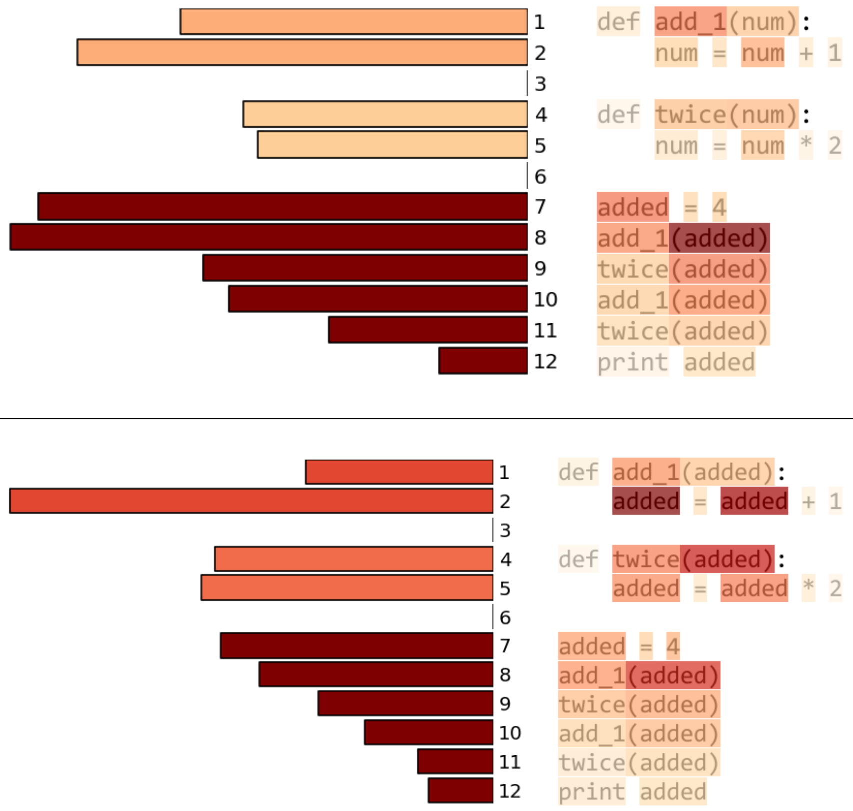
Figure 73: *Fixation timelines for two* `tuples` *trials. The yellow regions correspond to response periods.*

12.2.9 scope

The scope programs applied two functions to a variable named added: one function named add_1, and the other named twice. Both of these functions produced no visible effects – they did not actually modify their arguments or return a value. In the samename version, we reused the variable name added for each function's parameter name. For the diffname version, however, we used a different parameter name (num) for both functions. Because the add_1 and twice functions had no effect, the main added variable retained its initial value of 4 throughout the program (instead of being 22).

Participants made errors on both versions of the scope programs at about the same rate, and we did not observe non-eye-tracking performance differences between versions (trial duration, response proportion, etc.). In the diffname version, however, significantly more fixation time was spent on line 2 relative to the rest of the program (see the bottom of Figure 74). In this version, a variable named added existed in the main program body **and** inside the add_1 and twice functions. The extra time spent on line 2 suggests that having the same variable name for global and local added variables had a measurable effect, though not one that was visible by simply looking at responses and other non-eye-tracking performance metrics.

We did not observe any significant differences between versions in the transition matrices, and only minor differences existed in the transition matrices between correct and incorrect trials (Figure 75). Specifically, we note an increased probability of returning from the output box to line 8, as well as transitioning from line 2 to line 1. The former may simply be an additional check by participants after responding, and the latter may be evidence of recognition that the modified variable (added or num) is locally rather than globally scoped. The true reasons for these differences, however, may be much more complicated. As we mentioned previously, some participants asked the experimenter if functions in the Python language were "call by value" or "call by reference" when evaluating this program. We did not expect such a simple-looking program to induce deep language-related questions, and make our interpretation of the data so difficult!



```
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
```

```
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
```

Figure 74: *Total fixation durations by line for both versions of the* `scope` *program (all participants). Top:* `diffname`*, bottom:* `samename`*.*

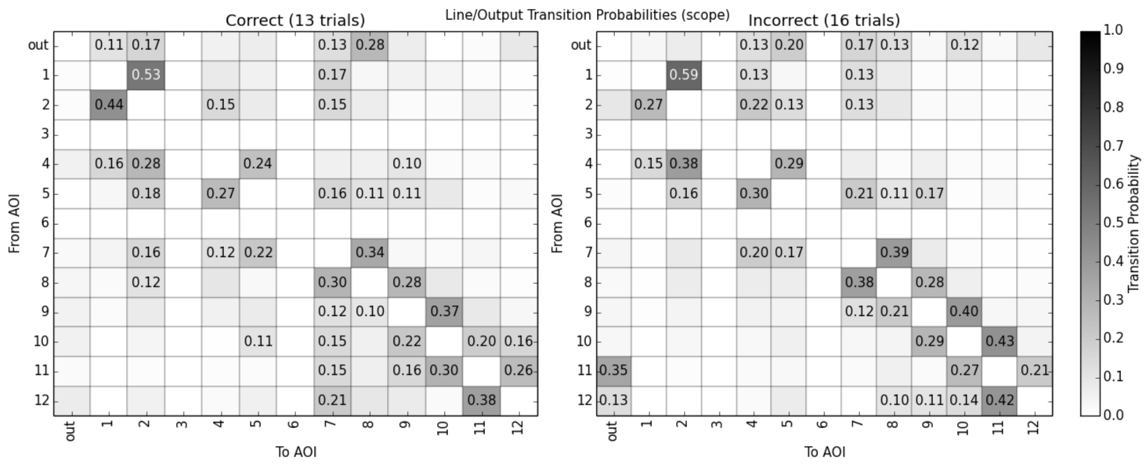
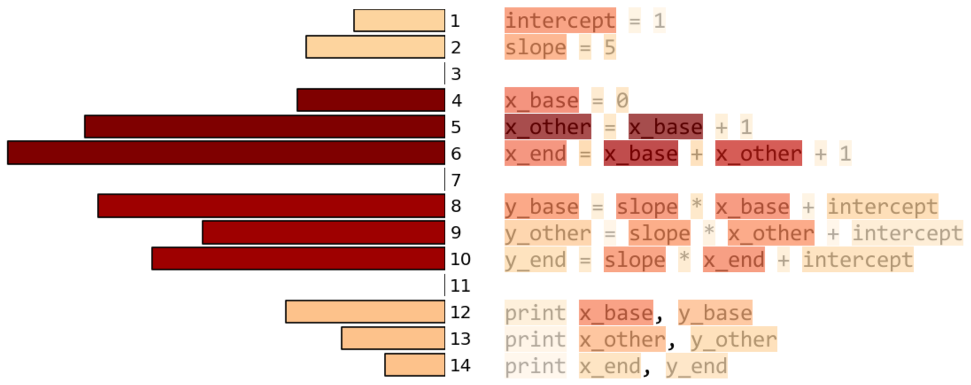


Figure 75: *Transition matrices for correct and incorrect trials in the* `scope` *programs. Probabilities below 0.1 are not annotated.*

12.2.10 whitespace

The `whitespace` programs print the results of three simple linear calculations. In the `zigzag` version, the code is laid out with one space between every mathematical operation, so that the line endings have a typical "zig-zag" appearance. The `linedup` version, in contrast, aligns each block of code by its mathematical operators, nicely lining up all identifiers. We expected there to be a speed difference between the two versions, with participants being faster in the `linedup` version. When designing the experiment, most of our pilot participants agreed that this version facilitated reading, but performance differences were not observed in practice.

We did not find differences in eye movements between versions either. Time spent on each line was approximately the same (Figure 76), and the average lengths of saccades as well as the spatial density of fixations on code were statistically indistinguishable. The transition matrices were also quite similar (Figure 77), leading us to conclude that lining up the text in this small program did not have an impact on eye movements. Because both styles are commonly found in code, programmers may have no trouble switching back and forth. Comparing one or both of these styles to a third, *uncommon* style may reveal differences (e.g., breaking the line in unusual places).



```
intercept = 1
slope = 5

x_base = 0
x_other = x_base + 1
x_end = x_base + x_other + 1

y_base = slope * x_base + intercept
y_other = slope * x_other + intercept
y_end = slope * x_end + intercept

print x_base, y_base
print x_other, y_other
print x_end, y_end
```

Figure 76: *Total fixation durations by line for both versions of the `whitespace` program (all participants). Top: `linedup`, bottom: `zigzag`.*

As with our programs, we observed correlations between fixation duration and specific times in a trial when we expected participants to be performing mental calculations. Figure 78 shows the fixation timeline for a single `zigzag` trial (top) and the participant's average fixation duration over a rolling one second window every half second (bottom). The six highlighted regions correspond to the typing of the six parts of the response: 0, 1, 1, 6, 2, 11 (including spaces and new lines). Note

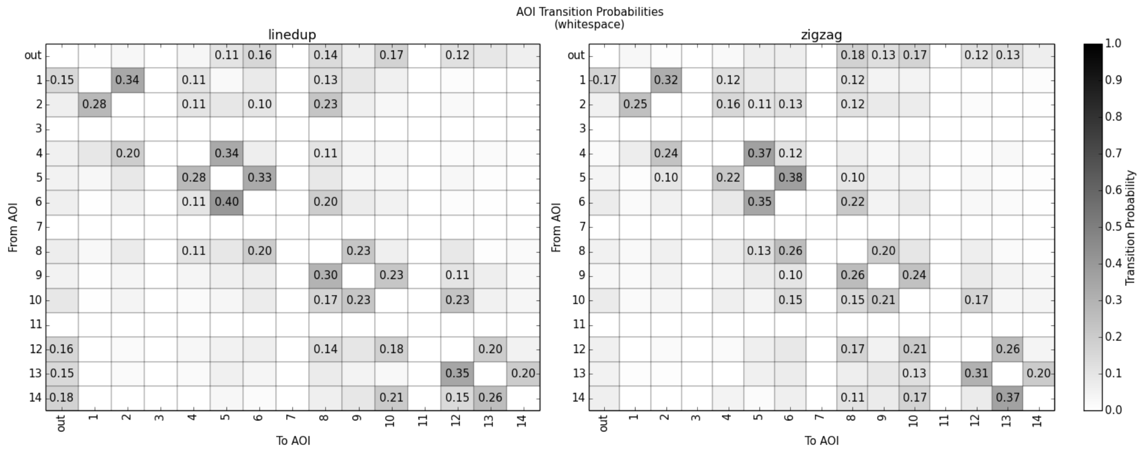
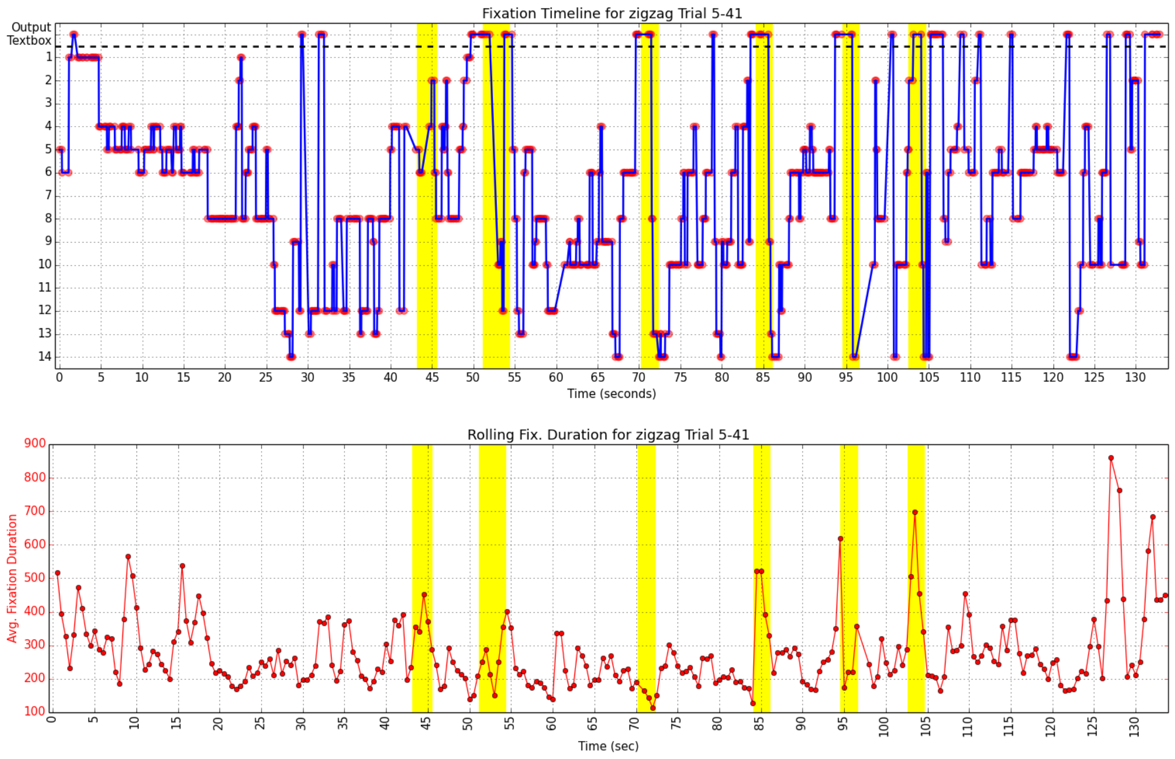


Figure 77: *Transition matrices for both versions of* `whitespace`*. Probabilities below 0.1 are not annotated.*

that most of these regions overlap with local spikes in fixation duration, and are also places where we might expect the participant to be calculating. The `y` component calculations are relatively more difficult, and do indeed correspond to clear spikes. The final `x` component response (for `x_end`, however, also corresponds to a spike despite being a simple addition. We hypothesize that the increase in fixation duration may be due to mental *retrieval* of variable values (`x_base`, `x_other`) rather than calculation. A quantitative cognitive model of program comprehension could predict the underlying source of the fixation duration spikes even though they may produce the same observable phenomena.



Fixation Timeline for zigzag Trial 5-41
Output Textbox
Time (seconds)
Rolling Fix. Duration for zigzag Trial 5-41
Avg. Fixation Duration
Time (sec)

Figure 78

13 Part 2: Discussion

Across 29 participants and 288 trials, we found programmers' eye movements to be strongly associated with the task at hand (output prediction), and highly informative about the relevant parts of each program. We saw evidence for short-term learning, both with repeated function calls and repeated code patterns. Surprisingly, we did not find large differences between the eye movements of more and less experienced programmers. While our larger study with Mechanical Turk did find instances where performance differed for more experienced programmers, this did not show up in our eye-tracking analysis. Despite this, we did observe one case where eye movements and task performance were strongly linked: the `counting` programs. Below, we provide a high-level discussion of our results as they relate to expected task-oriented behavior by participants and our three main research questions.

13.1 Task-Oriented Behavior

Participants' eye movement data revealed strong task-specific signatures for some programs. While the data from all programs were indicative of task-oriented behavior, the distributions of fixation duration in `between`, `initvar`, and `rectangle` were especially insightful. In the `between` programs, participants spent less time looking at the *second instance* of the `between` calculation in both versions. Similar behavior was observed in `scope` and `whitespace` for repeated function calls and calculations, indicating short-term learning and algorithm recognition by participants.

In the `bothbad` version of `initvar`, a decrease in fixation duration on lines associated with the first calculation suggested that participants were short-circuiting the product after noticing the first term was zero. The timeline for one `bothbad` trial in Figure 59 and the corresponding keystrokes in Table 8 provided corroborating evidence, specifically an immediate response of "0" followed by an explicit calculation. On the flip side in `bothbad`, one trial's fixation timeline, average rolling fixation duration, and sequence of keystrokes signaled the participant's mental calculation (Figures 34 and 36). We found a number of cases where spikes in the average rolling fixation duration correlated with times when the participant was expected to be calculating (i.e., between looking at the calculation line and responding).

Fixations in all three versions of `rectangle` were heavily focused on function arguments, either to the `Rectangle` constructor or the `area` function. Because all participants responded correctly to this program, we considered the possibility that little to no verification of the program was done – i.e., the `area` calculation was simply assumed to be correct. The eye movement data tells a different story. **Both** the arguments passed to `area` or `Rectangle` and the arguments of the function/class definition were focused on. In the `tuples` version, focus shifted to the extraction of coordinate components for the rectangle's `width` and `height`, again indicating that participants were intent on verifying the program's behavior. The transition matrices from `rectangle` also hinted at multiple strategies that participants were using to evaluate the program. Further investigation revealed trials that did precisely this (Figure 73), demonstrating the power of high-level, aggregate eye movement statistics.

13.2 Research Questions

Next, we review our three research questions in light of the detailed results in the previous section.

R1. How does the eye movement data from our experiment compare to other eye-tracking program comprehension experiments?

Based on previous program comprehension experiments with eye-tracking, we had specific expectations for our low-level eye movement metrics. Average fixation durations in previous experiments were higher than ours (300-400 ms versus 273 ms), and were also correlated with the participant's programming experience. We did not find programming or Python experience to be even moderately correlated with any of our eye-tracking metrics (both low and high level). We attribute these differences to the uniqueness of our task and the relative simplicity of our programs. Other experiments have had participants debug more complex code, such as binary search algorithms and prime number generators. In contrast, one of our most "complex" programs, `between`, featured list filtering and intersection. Such simple programs may not invoke the same behaviors observed in other experiments, especially in more experienced programmers.

Although our fixation metrics differed from other experiments, the distributions of fixation duration *across areas of interest* produced results more in line with expectations. Keywords, for example, received the least amount of focus, while more complex expressions with conditions and operators received the most. A perfect comparison between experiments is not possible, however, due to differences in how some code elements are categorized. We consider Python's `True` and `False` to be literal values like numbers and strings, but they are categorized as keywords in at least one other Java-based study [14]. Similar problems exist when comparing code complexity metrics like Halstead volume [45] between programming languages. Precisely defining which code elements are "operators", "operands", or neither can be surprisingly difficult – e.g., is the semi-colon in Java an end-of-statement operator?

R2. Can aggregate eye movement metrics and summary statistics be predicted from textual/syntactic features of code?

Across all programs and participants, we found that line-based reading behavior could be moderately predicted from a few simple textual features. The total amount of fixation duration on a line, for example, was decently predicted ($r^2 = 0.54$) from just the length of the line and its proportion of whitespace characters. Similarly, the time of the first fixation on a line (divided by the total trial duration) could be predicted fairly well ($r^2 = 0.57$) from the line number alone. Surprisingly, the category of a line – e.g., function call, `if` statement, `for` loop – did not significantly improved predictions. At the level of individual code elements, however, the category was useful in predicting total fixation duration. A simple linear model with the categories `keyword`, `list`, `identifier`, `operator`, `string`, `integer`, and `tuple` achieved an r^2 value of 0.383. When the line number of the code element was included, model performance increased to $r^2 = 0.576$. Thus, it appears that the category of a code element, and not the entire line, is a better predictor of fixation duration, though line number still plays a significant role.

Several patterns emerged from the AOI transition matrices for individual programs. High probability transitions tended to be clustered around whitespace-separated blocks of code, with transitions up and down the block. The matrix in Figure 72 for `rectangle tuples` is an especially clean example of this pattern. We expected the highest probability transitions from code to the output box to come from `print` statements. While this was simply true for some programs, such as `funcall` (Figure 54), the results were more complicated for most others. The transition matrices for `initvar` shown in Figure 58, for example, require additional explanation. In the `good` version (left), transitions to the output box tend to come from the final `print` statement (0.33 probability). The sum of probabilities from lines 2-4 to the output box, however, accounts for just as much.

Transitions *from* the output box often went to lists, such as those on lines 15 and 19 in `between functions` (Figure 46). This makes sense, given that participants were supposed to filter the lists and type their responses. In fact, we found a moderate correlation ($r = 0.30$, $p < 0.001$) between the transition probability from the output box for a given line and the total fixation duration on that line. If we interpret fixation duration as a measure of line importance, this says that participants tend to return to important lines after providing partial responses (which almost always co-occurred with fixating the output box).

R3. Do differences between versions of the same program, or demographics/performance of the participant, influence eye movements?

We did not observe strong differences in low-level eye movement metrics or high-level statistics between versions for most of our programs. Specifically, the two or three versions of `between`, `funcall`, `order`, `partition`, `scope`, and `whitespace` were very similar in terms of fixation duration distribution and AOI transition probabilities. There were not significant *performance* differences between participants interpreting different versions either in this study (though there were in our larger study).

The different versions of `counting`, `initvar`, `overload`, and `rectangle`, however, were distinguishable by eye movement metrics and statistics. For `counting`, we observed a distinct difference between the AOI transition matrices of correct and incorrect trials in `twospaces` version. Participants who gave correct responses tended to read all lines of the program before responding, while the others waited to read the final line. As mentioned above, the first calculation in the `bothbad` version of `initvar` received less focus, presumably because participants recognized a shortcut in the calculation not present in the other two versions. In the `plusmixed` and `multmixed` versions of `overload`, the final "5" + "3" operation received more time than in the `strings` version where all previous `+` operations were also string concatenations. Finally, the `tuples` version of `rectangle` demonstrated a unique allocation of fixation duration on the calculation of the rectangle's width and height – a point where correct tuple indexing was critical.

Surprisingly, we did not find a strong or even moderate correlation between a participant's programming experience, Python experience, or response correctness and **any** of our low-level eye movement metrics. Programming experience was a statistically significant predictor in a simple linear model predicting average fixation duration. However, the effect size (r^2) was extremely small (< 0.1), so we do not have much confidence in its generalizability to other groups of programmers. As we discussed earlier for research question R1, our unique task and simple programs may account for these results. Other performance metrics, such as trial duration, were trivially correlated with number of fixations in a trial and total scanpath length – longer trials necessarily mean more fixations and saccades. Other eye movement metrics, like average fixation duration, fixation rate, and average saccade length were not correlated with any performance metric or demographic value.

13.3 Scanpath Comparison

We had hoped to use the Levenshtein (string-edit) distance as a means of comparing participant scanpaths on the same program. This proved more difficult than expected, as individual scanpaths were quite different, even at multiple levels of granularity. While scanpaths at the code element level (e.g., keywords, identifiers) will be obviously different and quite noisy, we were surprised to find similar obstacles at the line and whitespace-separated block level. On average, we found differences of over 50% between line and block scanpaths (over 60% for line). These results suggest that alternative techniques may be necessary for meaningful comparisons of programmer scanpaths, even for very simple programs. For example, Cristino et. al have developed a technique based on the Needleman-Wunsch algorithm used to compare DNA sequences in bioinformatics [21]. Hayes et. al have also created a method for converting scanpaths to a type of transition matrix that retains temporal information (called the scanpath successor representation) [47]. These new techniques may provide additional insight, but they are beyond the scope of this work.

14 Part 2: Conclusion

Eye movements are a rich data source, and our analysis has shown just how much information can be extracted from a mere 29 participants. Over the course of 288 trials, our participants predicted the output of 25 total programs (10 programs each). Low-level eye movement metrics, such as fixation duration and saccade length, did not correlate with task performance or programming experience. Fixation duration did serve as an excellent proxy for code line importance, with the task-relevant lines of a program often receiving the most aggregate focus. In line with previous experiments, we found that code elements like keywords and identifiers received the least amount of focus, while more complex statements involving conditional/boolean expressions received more.

High-level eye movement metrics and plots provided the most insight into our programmers' cognitive processes. Area of interest (AOI) transition matrices revealed how often participants transitioned between code lines and the output box. In several cases, these transition probabilities provided hints at the kinds of strategies being used to evaluate the programs. In the `counting` programs, for example, correct and incorrect responses to the `twospaces` version had distinct transition matrices, with the latter having a smaller probability of reading the entire program before starting to respond. The transition matrix for `rectangle tuples` had transitions corresponding to two different strategies (single response, multi-response), and we found trials clearly demonstrating each. Lastly, we combined high and low-level metrics using a rolling time window across individual trials. When looking at combined changes in mean fixation duration, recently fixated lines, and response keystrokes, we found that spikes in fixation duration often co-occurred with times we would expect the participant to perform a mental calculation.

Eye-tracking has recently gained popularity in the program comprehension literature, augmenting or replacing traditional methodologies like think-aloud. Metrics and plots derived from eye movement data can be used to gain invaluable insight into a programmer's cognitive processes. This data could also be used to develop a cognitive model of program comprehension, a step towards the semi-automated analysis of program language design. We plan to use the data collected in this experiment to create a functioning version of Cant et. al's Cognitive Complexity Metric [16]. By incorporating existing components of a *cognitive architecture* – a computational model of human cognitive – we will build on years of modeling research in human memory and perceptual/motor systems.

14.1 Future Work

Aside from developing a cognitive model, there are many other interesting avenues for future work. An obvious extension to our experimental methodology is to include additional programming languages. Java is used in a number of other studies, and so is a likely candidate. More complex programs may elicit differences between more and less experienced programmers, but may require introducing the skeleton of a development environment. A multi-file program, for example, would require additional GUI elements like tabs or a file explorer. Human-computer interface studies on integrated development environments (IDEs) exist (e.g., [56]), and it would be interesting to add eye-tracking to the mix.

We used rolling metrics to correlate spikes in fixation duration with moments where we expected participants to be performing some kind of mental calculation or memory retrieval. Many other eye movement metrics, such as spatial density and saccade length, could be analyzed in a similar manner. Additionally, different window sizes focus on different timescales, providing yet another analysis parameter to vary.

Finally, our comparison of scanpaths was stunted by the brittle nature of the Levenshtein distance. More robust, genomic-inspired methods, like ScanMatch [21] are in our sights for future work. A more radical approach would involve starting at our AOI mapping stage, where fixations are assigned to specific AOIs. Rather than picking a single AOI, the areas of overlap between a circle surrounding the fixation point and proximate AOIs could produce multiple possible scanpaths. Scanpath comparisons would then be between multi-dimensional spaces, rather than one dimensional series, of AOIs.

Part 3

Mr. Bits: A Quantitative Process Model of Simple Program Understanding

&

Nibbles: A Constraint-Based Model of Program Reading and Inference

Abstract

The cognitive complexity of a program, or how difficult it is to understand, is a function of the underlying problem domain, the programming language, and the programmer themselves. In this part, we explore two different modeling formalisms with the goal of producing a quantitative model of human program comprehension. Specifically, we model beginning and experienced Python programmers who are tasked with predicting the printed output of short Python programs. We model the process of reading code using ACT-R, a cognitive architecture, to simulate fixations and keystroke responses to the task. At a higher level of abstraction, Cognitive Domain Ontologies and a constraint solver are used to model code interpretation and human errors. We produce successful, executable models with both formalisms, but note that a comprehensive model of our task will require their combination.

15 Part 3: Introduction

15.1 The Complexity of a Program

What makes some code hard to understand? Intuitively, we might expect the difficulty of the underlying problem, which the program is attempting to solve, to be a major factor. A program designed to simulate the fine-grained physics of an automobile, for example, is likely to be more complex than one that simply computes the average of a set of numbers. The latter program is also bound to be *shorter* and use *fewer operations* than the former [104]. However, short and simple-looking programs can be hard to understand, even for *experienced* programmers, when certain notational and conceptual *expectations* are violated [89]. Therefore, the *cognitive complexity* of a program – e.g., how hard it is to understand – is determined by computational, notational, and psychological factors [16]. To fully understand and predict this kind of complexity, we must create *cognitive models* of programmers that are capable of reasoning about source code and representing programs in a human-like manner. These models could be used to locate unmaintainable code in large codebases, inform design decisions for programming languages, and aid in the automated generation of programs for humans. Additionally, quantitatively modeling such a complex, real world task pushes the frontiers of Cognitive Science by combining existing models of representation, working memory, planning, and problem solving.

In this part, we present two cognitive models, called **Mr. Bits** and **Nibbles**. Mr. Bits is designed to simulate the eye movements and keystroke response times of a programmer who is tasked with reading short Python programs and guessing their printed output. The model is built on top of the **ACT-R cognitive architecture**, a computational simulation of the major components of human cognition and perception, such as declarative/procedural knowledge, vision, and hearing [3]. Mr. Bits is also intended to operationalize aspects of the Cognitive Complexity Metric [16] – a source code metric designed to measure how difficult a program is to *trace* through and mentally *chunk*. A major limitation of the Mr. Bits model, however, is that it *cannot make errors*. Therefore, we explore an alternative formalism called Cognitive Domain Ontologies [33], and produce a second model called **Nibbles**. Unlike Mr. Bits, Nibbles generates multiple possible interpretations of a program's code line-by-line using a constraint solver. After reading each line, Nibbles selects a single (possibly incorrect) interpretation to incorporate into its running "mental model". We evaluate Mr.

Bits by comparing its timing performance to 29 human programmers, each tested on 10 out of a set of 25 short Python programs. For Nibbles' evaluation, we focus on 3 of these programs, and show how the errors observed in our human programmers can be modeled by Nibbles.

15.2 The Psychology of Programming

Psychologists have been studying programmers for at least forty years [23]. Early research focused on correlations between task performance and human/language factors, such as how the presence of code comments impacts scores on a program comprehension questionnaire. More recent research has revolved around the cognitive processes underlying program comprehension. Effects of expertise, task, and available tools on program understanding have been found [28]. Studies with experienced programmers have revealed conventions, or "rules of discourse," that can have a profound impact (sometimes negative) on expert program comprehension [89].
The majority of **qualitative** models of program comprehension are simply "box and arrow" diagrams that connect English descriptions of the models' cognitive components. Von Mayrhauser et al. evaluate five such models in [101], and produce an integrated "metamodel" to capture shared aspects of each individual model (Figure 79). The Integrated Metamodel and others like it incorporate research from many facets of cognitive science: psychology, linguistics, computer science, and even neuroscience [68]. They are useful for making sense of the myriad of results from controlled studies of programmers over the last four decades [29]. Qualitative cognitive models can help emphasize which aspects of cognition are most important for program comprehension, such as language and problem solving skills [87]. Unfortunately, their predictive power is limited. The Stores Model of Code Cognition, for example, emphasizes the importance of the central executive and its relationship to strategic, semantic, and plan knowledge in code problem solving [31]. But predicting whether or not a particular programmer will successfully comprehend a specific program is simply not possible with the model. A *quantitative* model is needed, which formalizes and operationalizes aspects of existing *qualitative* models to emulate the process of reading, interpreting, and understanding a program. In this part, we develop and evaluate two quantitative cognitive models of program comprehension built using different formalisms: the **ACT-R cognitive architecture** [3], and **Cognitive Domain Ontologies** [33].

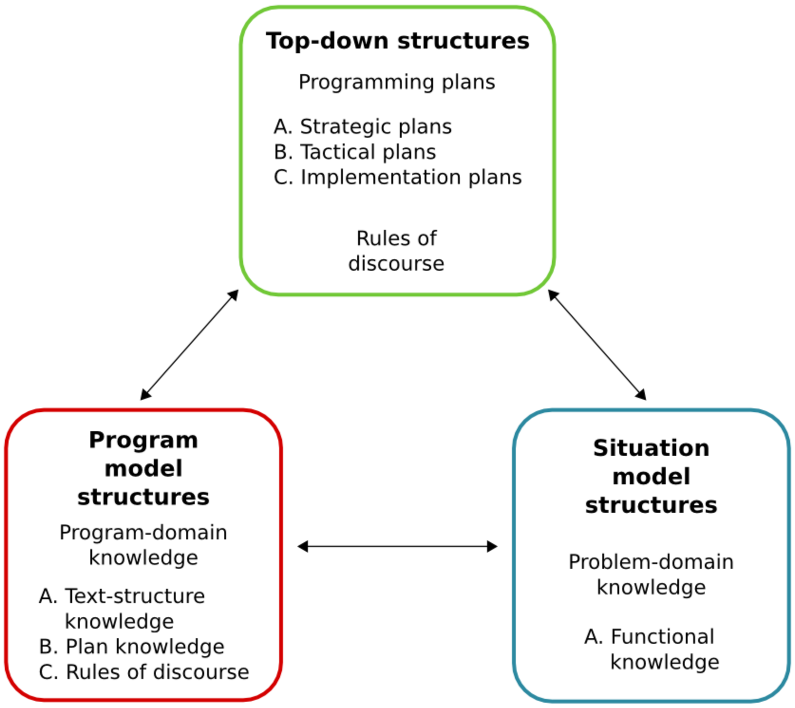


Figure 79: *High-level view of Von Mayrhauser's Integrated Metamodel (re-created from Figure 6 in [101])*

15.3 Why Model a Programmer?

The design, creation and interpretation of computer programs are some of the most cognitively challenging tasks that humans perform. Understanding the factors that impact the cognitive complexity of code is important for both applied and theoretical reasoning. Practically, an enormous amount of time is spent developing programs, and we believe even more time is spent debugging them. If we can identify factors that expedite these activities, then, a large amount of time and money can be saved. Theoretically, programming is an excellent task for studying representation, working memory, planning, and problem solving in the real world.

Our present research focuses on programs much less complicated than those the average professional programmer typically encounters on a daily basis. We base our models on data from the **eyeCode experiment**, a web-based experiment in which Python programmers were tasked with predicting the printed output of 10 short Python programs [46]. This task is similar to debugging a short snippet of a larger program, as our programmers were tasked to predict *precise* output. Code studies often take the form of a code review, where programmers must locate errors

or answer comprehension questions after the fact (e.g., does the program define a Professor class? [10]). Our task differs by asking programmers to mentally simulate code without necessarily understanding its purpose. In most programs, we intentionally used meaningless identifier names where appropriate (variables `a`, `b`, etc.) to avoid influencing the programmer's mental model.
The main product of this research is a realizable (executable) theory of program comprehension that can quantitatively predict human performance in the eyeCode experimental task. We are motivated to produce such a theory in order to (1) inform programming language design, (2) provide a foundation for high-level source code inspection/generation tools, and (3) to push the frontiers of quantitative model in Cognitive Science into more complex tasks.
Similar research has asked beginning (CS1) programming students to read and write code with simple goals, such as the Rainfall Problem [44]. To solve it, students must write a program that averages a list of numbers (rainfall amounts), where the list is terminated with a specific value – e.g., a negative number or 999999. CS1 students perform poorly on the Rainfall Problem across institutions around the world, inspiring researchers to seek better teaching methods. Our experiment included many Python novices with a year or less of experience, so our results may also contribute to ongoing research in early programming education [15].

16 Part 3: Background

Psychologists have been studying the behavioral aspects of programming for at least forty years [23]. In her book *Software Design - Cognitive Aspects*, Françoise Détienne proposed that psychological research on programming can be broken into two distinct periods [29]. The first period, spanning the 1960's and 1970's, is characterized by the importing of basic experimental techniques and theories from psychology into computer science. Early experiments looked for correlations between task performance and language/human factors – e.g., the presence or absence of language features, years of experience, and answers to code comprehension questionnaires. While this period marked the beginning of scientific inquiry into software usability from a programming perspective, results were limited in scope and often contradictory.
The problem is simple: if task performance depends heavily on internal cognitive processes, then it cannot be measured independent of the programmer. Early psychology of programming studies relied exclusively on statistical correlations between metrics like "presence of comments" and "number of defects detected," so researchers were unable to explain some puzzling results. For example, multiple studies in the 1970's sought to measure the effect of meaningful variable names on code understandability. Two studies found no effect [102, 85] while a third study found positive effects as programs became more complex [81]. Almost a decade later, Soloway and Ehrlich provided an explanation for these findings: experienced programmers are able to recognize code *schemas* or *programming plans* [89]. Programming plans are "program fragments that represent stereotypic action sequences in programming," and expert programmers can use them to infer intent in lieu of meaningful variable names. This and many other effects depend on internal cognitive processes, and therefore require a cognitive modeling approach to explain [12].

16.1 The Cognitive Complexity Metric

Developed in the mid-nineties, Cant et al.'s cognitive complexity metric (CCM) attempted to quantify the cognitive processes involved in program development, modification, and debugging [16]. The CCM focuses on the cognitive processes of *chunking* (understanding a block of code) and *tracing* (locating dependencies for the current chunk). Cant et al. provide mathematical definitions for factors that are believed to influence each process. Some these factors are quantified by drawing upon the existing literature at the time, but many definitions are simply placeholders for future empirical studies.

Section 17 describes **Mr. Bits**, a partial implementation of the CCM built using the ACT-R cognitive architecture 16.2. Many of the underlying CCM terms can be simplified or eliminated within ACT-R. This is because the *subsymbolic* layer in ACT-R already contains equations for low-level cognitive processes like rule learning, memory retrieval, and visual attention. Below, we describe the CCM in more detail. In addition to the high-level equations, we briefly discuss each factor in the metric, and how it subsumed within our ACT-R model.

16.1.1 Chunking and Tracing

The cognitive processes of chunking and tracing play key roles in the cognitive complexity metric (CCM). **Chunking** is defined as the process of recognizing groups of related code statements (not necessarily sequential), and recording the information extracted from them as a single mental symbol or abstraction. In practice, programmers rarely read through and chunk every statement in a program. Instead, they **trace** forwards or backwards in order to find relevant chunks for the task at hand [16]. Cant et al. define a chunk as a block of statements that must occur together (e.g., loop + conditional).[14] This definition, however, is intended only for when the programmer is reading code in a line-by-line forward manner. When tracing backwards or forwards across many lines (or files), a chunk is defined as the single statement involving a procedure or variable's definition. In the remainder of this section, care must be taken to disambiguate Cant et al.'s use of the word "chunk" and the ACT-R term (Figure 81). We will clarify instances where there may be ambiguity between these two distinct terms.

16.1.2 Chunk Complexity (C)

The CCM depends on a dozen factors, many of which contain one or more "empirically determined constants". To compute the complexity C_i of chunk i, Cant et al. define the following equation:

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

[14]When operationalizing the definition of a chunk, Cant et al. admit that "*it is difficult to determine exactly what constitutes a chunk since it is a product of the programmer's semantic knowledge, as developed through experience.*"

where R_i is the complexity of the immediate chunk i, C_j is the complexity of sub-chunk j, and T_j is the difficulty in tracing dependency j of chunk i. The definitions of R and T are given as follows:

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

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

Each right-hand side term stands for a particular factor that is thought to influence the chunking or tracing processes (Table 9). The underlying equations for each factor can be found in [16], but are not needed for the discussion below.

16.1.3 Immediate Chunk Complexity (R)

The chunk complexity R is made up of six additive terms (R_S, R_C, R_E, R_R, R_V, R_D) and one multiplicative term R_F. These represent factors that influence how hard it is to understand a given chunk.[15]

$\mathbf{R_F}$ (chunk familiarity) This term captures the increased speed with which a programmer is able to understand a given chunk after repeated readings. In ACT-R, the subsymbolic layer of the declarative memory module handles this, as repeated retrievals of the same ACT-R chunk will increase its retrieval speed. Declarative chunks are not stored independently, however, so the activation of "similar" chunks will potentially cause interference. This means that increased familiarity with one chunk may come at the cost of slower retrieval times for similar chunks.
"size": (1) its structural size (e.g., lines of code) and (2) the "*psychological complexity of identifying a chunk where a long contiguous section of non-branching code must be divided up in order to be understood.*" In other words, R_S should be affected by some notion of short-term memory constraints. Any ACT-R model would be influenced by a chunk's structural size simply because there would be more code for the visual module to attend to and encode (i.e., more sequential productions fired). We do not model the additional "psychological complexity" in Mr. Bits directly, and instead rely on the interplay between vision and memory as code is read and re-read.

[15]More general forms for R and T are discussed in [16], but Cant et al. suggest starting with simple additive representations.

R_C (control structures) The type of control structure in which a chunk is embedded influences R because conditional control structures like `if` statements and loops require the programmer to comprehend additional boolean expressions. In some cases, this might involve mentally iterating through a loop. We expect that boolean expressions are comprehended in much the same way as for the R_E factor (see below). Mr. Bits is not clever enough to chunk an entire loop at once, but it will get faster at evaluating the loop body after each iteration.

R_E (boolean expressions) Boolean expressions are fundamental to the understanding of most programs, since they are used in conditional statements and loops. According to Cant et al., the complexity of boolean expressions depends heavily on their form and the degree to which they are nested. Mr. Bits evaluates boolean expressions in the same manner as numeric additions and subtractions: via declarative memory retrievals (Section 17.4). Thus, it will get faster after repeated uses of the same expression (e.g., $true \wedge false$). Unlike humans, however, Mr. Bits will not *short-circuit* expressions, and avoid unnecessary memory retrievals (e.g., $false \wedge \ldots \wedge \ldots$).

R_R (recognizability) Empirical studies have shown that the syntactic form of a program can have a strong effect on how a programmer mentally abstracts during comprehension [42]. An ACT-R model would show such an effect if its representation of the program was built-up over time via perceptual processes. In other words, the model would need to observe actual code rather than receiving a pre-processed version of the program as input (e.g., an abstract syntax tree). Mr. Bits operates visually on text, though its goals are generated from a processed form of that text. It does not, however, produce an abstract model above the line/token level like the Nibbles model (Section 18).

R_V (visual structure) This term represents the effects of visual structure on a chunk's complexity, and essentially captures how visual boundaries influence chunk identification. While Cant et al. only describe three kinds of chunk delineations (function, control structure, and no boundary), more general notions of textual beacons and boundaries have been shown to be important in code [106]. For example, Détienne found that advance organizers (e.g., a function's name and leading comments) had a measurable effect on programmers' expectations of the code that followed [26]. Biggerstaff et al. have also designed a system for automatic domain concept recognition in code [8]. This system considers whitespace to be meaningful, and uses it to bracket groups of related statements. As with the recognizability term (R_R), it is crucial for an ACT-R model to observe real code instead of an abstract syntax tree. Mr. Bits does not rely on whitespace to delineate chunks, but does depend on function definition headers and `for` loop beacons to accurately shift visual attention.

$\mathbf{R_D}$ (dependency disruptions) There are many disruptions in chunking caused by the need to resolve dependencies. This applies both to remote dependencies (e.g., variable definitions), and to local dependencies (e.g., nested loops and decision structures). Cant et al. cite the small capacity of short-term memory as the main reason for these disruptions. ACT-R does not have a distinct "short-term memory" module with a fixed capacity. Instead, short-term capacity limits are an emergent property of memory decay and interference [92]. Mr. Bits is capable of forgetting variable values and previously-read lines of code if they are not retrieved frequently enough from memory (Section 17.3). Frequent disruptions due to tracing will slow down retrievals when Mr. Bits returns to the source line.

16.1.4 Tracing Difficulty (T)

Most code does not stand alone. There are often dependencies that the programmer must resolve before chunking the code in memory and ultimately understanding what it does. The cognitive complexity metric (CCM) operationalizes the difficulty in tracing a dependency as $T = T_F(T_L + T_A + T_S + T_C)$. For these six terms, the definition of a "chunk" is slightly different than with R. Rather than being a block of statements that must co-occur, a chunk during tracing is defined as a single statement involving a procedure's name or a variable definition.

$\mathbf{T_F}$ (familiarity) The dependency familiarity has a similar purpose to the chunk familiarity (R_F). As with R_F, ACT-R's subsymbolic layer will facilitate a familiarization effect where repeated requests for the same dependency information from declarative memory will take less time. The effect of the available tools in the programmer's environment, however, are also important. An often-used dependency (e.g., function definition) may be opened up in a new tab or bookmarked within the development environment. A more comprehensive version of Mr. Bits would need to interact with the same tools as a human programmer to produce "real world" familiarization effects. As it stands, Mr. Bits simply relies on ACT-R's declarative memory calculus to speed up subsequent retrievals of the same dependencies over time.

T_L (localization) This term represents the degree to which a dependency may be resolved locally. Cant et al. proposed three levels of localization: embedded, local, and remote. An embedded dependency is resolvable within the same chunk. A local dependency is within modular boundaries (e.g., within the same function), while a remote dependency is outside modular boundaries. Mr. Bits follows this model for embedded and local dependencies only as multi-module programs are not yet supported. The resolution of variable names in Mr. Bits is attempted in the immediate function context first, but may ultimately succeed either with an earlier definition of the variable, or in a previous function/global context.

T_A (ambiguity) Dependency ambiguity occurs when there are multiple chunks that depend upon, or are affected by, the current chunk. The CCM considers ambiguity to be binary, so a dependency is either ambiguous or not. Whether ambiguity increases the complexity of a chunk is also dependent on the current goal, since some dependencies do not always need to be resolved (e.g., unused parameters can be ignored). Mr. Bits is designed to evaluate short, simple programs, and therefore does not encounter dependency ambiguity besides variables that share the same name in separate function contexts. Multi-module programs, especially those with class inheritance hierarchies, and outside the scope of Mr. Bits' current abilities.

T_S (spatial distance) The physical distance between the current chunk and its dependent chunk will affect the difficulty in tracing. Lines of code are used in the CCM as a way of measuring distance, though this seems less relevant with modern development environments. Developers today may have multiple files open at once, and can jump to variable/function definitions with a single keystroke. For single-file programs, like those Mr. Bits can evaluate, physical distance may have a noticeable effect. As with the R_S term, ACT-R's visual module produces differential timings for saccades to nearby and distance locations, and therefore no extra effort is required to support this term.

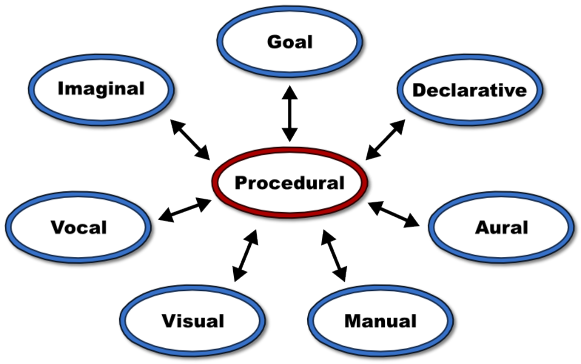
T_C (level of cueing) This binary term represents whether or not a reference is considered "obscure." References that are embedded within large blocks of text are considered obscure, since the surrounding text may need to be inspected and mentally broken apart. This term appears to be related to the effect of visual structure on a chunk's complexity (R_V). Clear boundaries between chunks (e.g., whitespace, headers) play a large role in R_V, and we expect them to play a similar role in T_C. Tracing is presumed to involve a more cursory scan of the code than chunking, however. Because Mr. Bits operates at a line/token level, it cannot (for lack of a better phrase) see the forest for the trees, and is therefore not capable of modeling T_C.

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

Table 9: *Important factors in the cognitive complexity metric. R_x terms affect chunk complexity. T_x terms affect tracing difficulty.*

16.2 The ACT-R Cognitive Architecture

ACT-R is "*a cognitive architecture: a theory about how human cognition works.*" [1] It is a domain-specific programming language built on top of LISP [90], and a simulation framework for cognitive models. There are eight *modules* in ACT-R, which represent major components of human cognition (Figure 80). Module functionality exist at two layers: (1) the *symbolic* layer, which provides a uniform programmatic interface for models, and (2) the *subsymbolic* layer, which hides the implementation details of the module – e.g., the time it takes to retrieve an item from declarative memory. ACT-R models formalize humans performance on tasks using production rules that send and receive messages (chunks) between modules via *buffers*. Models are simulated and evaluated along psychologically relevant dimensions like task accuracy, response times, and simulated BOLD.[16] measures (fMRI brain "activations") These simulations are reproducible, and can provide precise predictions about human behavior. Below, we discuss the ACT-R architecture in more detail, and provide examples of its successful application in a variety of domains.



Module	Purpose
Procedural	Stores and matches production rules, facilitates inter-module communication
Goal	Holds a chunk representing the model's current goal
Declarative	Stores and retrieves declarative memory chunks
Imaginal	Holds a chunk representing the current problem state
Visual	Observes and encodes visual stimuli (color, position, etc.)
Manual	Outputs manual actions like key-presses and mouse movements
Vocal	Converts text strings to speech and performs subvocalization
Aural	Observes and encodes aural stimuli (pitch, location, etc.)

Figure 80: *ACT-R 6.0 modules. Communication between module buffers is done via the procedural module.*

16.2.1 Buffers, Chunks, and Productions

The ACT-R architecture is divided into eight *modules*, each of which has been associated with a particular brain region (see [3] for more details). Every module has its own *buffer*, which may contain a single *chunk*. Buffers also serve as the interface to a module, and can be queried for the

[16]Blood-oxygen-level-dependent contrast. This is the change in blood-flow for a given brain region over time.

module's state. Chunks are the means by which ACT-R modules encode messages and store information internally. They are essentially collections of name/value pairs (called slots), and may inherit their structure from a parent chunk type. Individual chunk instances can be extended in advanced models, but simple modules tend to have chunks with a fixed set of slots (e.g., two addends and a result for a simple model of addition). Modules compute and communicate via *productions*, rules that pattern-match on the slot values of a chunk or the state of a buffer. When a production matches the current system state (all chunks in all module buffers), it "fires" a response. Responses include actions like inserting a newly constructed chunk into a buffer, and/or modifying/removing a buffer's existing chunk.[17]

When it is possible, computations **within** a module are done in parallel. Exceptions include the serial fetching of a single memory from the *declarative* module, and the *visual* module's restriction to only focus on one item at a time. Communication **between** modules is done serially via the *procedural* module (see Figure 80). Only one production may fire at any given time[18], making the procedural model the central bottleneck of the system.

The *visual*, *aural*, *vocal*, and *manual* modules are capable of communicating with the model's external environment, such as a graphical interface with buttons. In ACT-R, this environment is often simulated for performance and consistency. Experiments in ACT-R can be written to support both human and model input, allowing for a tight feedback loop between adjustments to the experiment and adjustments to the model. Lastly, the *goal* and *imaginal* modules are used to maintain the model's current goal and problem state, respectively.

16.2.2 The Subsymbolic Layer

Productions, chunks, and buffers exist at the *symbolic* layer in ACT-R. The ability for ACT-R to simulate human performance on cognitive tasks, however, comes largely from the *subsymbolic* layer. A symbolic action, such as retrieving a chunk from declarative memory, does not return immediately. Instead, the declarative module performs calculations to determine how long the retrieval will take (in simulated time), and the probability of an error occurring. The equations for subsymbolic calculations in ACT-R's modules come from existing psychological models of

[17]By default, chunks that are removed from a buffer are automatically stored in the declarative memory module.

[18]This is a point of contention in the literature. Other cognitive architectures, such as EPIC [53] allow more than one production to fire at a time.

learning, memory, problem solving, perception, and attention [1]. Thus, there are many constraints on models when fitting parameters to human data.

In addition to calculating timing delays and error probabilities, the subsymbolic layer of ACT-R contains mechanisms for learning. Productions can be given *utility* values, which are used to break ties during the matching process.[19] Utility values can be learned by having ACT-R propagate rewards backwards in time to previously fired productions. New productions can also be automatically *compiled* from existing productions (representing the learning of new rules). Production compilation could occur, for example, if production P_1 retrieves stimulus-response pairs from declarative memory and production P_2 presses a key based on the response. If P_1 and P_2 co-occur often, the compiled $P_{1,2}$ production would go directly from stimulus to key press, saving a trip to memory. If $P_{1,2}$ is used enough, it will eventually replace the original productions and decrease the model's average response time.

Term	**Definition**
Chunk	Representation for declarative knowledge
Production	Representation for procedural knowledge
Module	Major components of the ACT-R system
Buffer	Interface between modules and procedural memory system
Symbolic Layer	High-level production system, pattern-matcher
Subsymbolic Layer	Underlying equations governing symbolic processes
Utility	Relative cost/benefit of productions
Production Compilation	Procedural learning, combine existing productions
Similarity	Relatedness of chunks

Figure 81: *ACT-R terminology*

[19]This is especially useful when productions match chunks based on *similarity* instead of equality, since there are likely to be many different matches.

16.2.3 Successful ACT-R Models

Over the course of its multi-decade lifespan, there have been many successful ACT-R models in a variety of domains. Success here means that the models fit experimental data well, and were also considered plausible explanations. We briefly discuss three examples below, though many more are available on the ACT-R website [1].

David Salvucci (2001) used a multi-tasking ACT-R model to predict how different in-car cellphone dialing interfaces would affect drivers [77]. This integrated model was a combination of two more specific models: one of a human driver and another of the dialing task. By interleaving the production rules of the two specific models[20], the integrated model was able to switch between tasks. Salvucci's model successfully predicted drivers' dialing times and lateral deviation from their intended lane.

Brian Ehret (2002) developed an ACT-R model of location learning in a graphical user interface [35]. This model gives an account of the underlying mechanisms of location learning theory, and accurately captures trends in human eye-gaze and task performance data. Thanks to ACT-R's perception/motor modules, Ehret's model was able to interact with the same software as the users in his experiments.

Finally, an ACT-R model created by Taatgen and Anderson (2004) provided an explanation for why children produce a U-shaped learning curve for irregular verbs [93]. Over the course of early language learning, children often start with the correct usage (I went), over-generalize the past-tense rule (I goed), and then return to the correct usage (I went). Taatgen and Anderson's model proposed that this U-shaped curve results from cognitive trade-offs between irregular and regular (rule-based) word forms. Retrieval of an irregular form is more efficient if its use frequency is high enough to make it available in memory. Phonetically post-processing a regular-form word according to a rule is much slower, but always produces something. Model simulations quantified how these trade-offs favor regular over irregular forms for a brief time during the learning process.

The success of these and other non-trivial models in different domains gives us confidence that ACT-R is mature enough for modeling program comprehension. Section 17 describes the Mr. Bits ACT-R model in detail, and compares its performance to human programmers on the same task.

[20]Salvucci notes that this required hand-editing the models' production rules since ACT-R does not provide a general mechanism for combining models. See [78] for a more advanced model of multi-tasking in ACT-R.

16.3 Cognitive Domain Ontologies

A Cognitive Domain Ontology (CDO) is a formal representation of domain knowledge based on System Entity Structure (SES) theory [33]. SES theory is a formal specification framework for describing system aspects and properties [108]. For decades, researchers have used SES theory to automate the exploration of design space alternatives by enumerating the set of all possible system configurations, pruning them according to domain constraints, and then simulating/evaluating each pruned system. CDOs are a theoretical extension to SES in which "system configurations" capture *spaces of behavior or situational knowledge*. A model or agent uses a CDO to explore alternative courses of action, or interpretations of evidence, based on a domain's structure, a-priori constraints, and situational factors. Psychologically, CDOs can be used to represent *mental models*[21], and the exploration of alternative interpretations mimics the reasoning processes of *abduction and deduction*. Deduction produces logical consequences from observation and axioms whereas abduction generates likely explanations for observed evidence. While abduction, unlike deduction, is not logically guaranteed to be true, it is an invaluable tool for knowledge-level modeling [61]. CDOs are formally represented as trees with *entities* as nodes, and one of three *relations* as edges (see next section for details). The pruning of alternatives in a CDO is cast as a constraint satisfaction problem (CSP), and is computationally realized using two extensions to Common LISP [90]. The Screamer [88] and Screamer+ [105] LISP extensions allow for non-deterministic execution of code by adding two special forms. The `either` form takes any number of LISP expressions, and establishes a *choice point*. The value of the first expression is returned, and control flow proceeds as normal until a `fail` form is reached. On each `fail`, Screamer backtracks to the nearest `either`, returning the value of its next expression. Once values are exhausted, evaluation jumps to the next nearest `either` or terminates. Listing 2 provides a LISP code example with `all-values`, a Screamer form that establishes a non-deterministic context and returns a list of all generated values. Screamer+ extends Screamer to allow for more complex data types in `either`, such as Common Lisp Object System (CLOS) objects [52].

[21]Similar in spirit to Johnson-Laird's mental models [50]. Constraint knowledge in CDOs is also implicit – i.e., not available for introspection.

```
> (all-values
    (let ((x (either 'a 'b 'c))
          (y (either 1 2 3)))
      ;; Exclude (B 2)
      (if (and (eq x 'b) (eq y 2))
          (fail))

      (list x y)))

;; ((A 1) (A 2) (A 3) (B 1) (B 3) (C 1) (C 2) (C 3))
```

Listing 2: *Example LISP code with Screamer extensions. The* `either` *and* `fail` *forms are used to generate values.*

16.3.1 Entities, Relations, and Constraints

A CDO consists of a set of *entities*, *relations* between entities, and *constraints*. Entities may also have one or more *attached variables*, which can store values like integers and strings, and may be used in constraints. Every CDO has a top-level, or root, entity and alternating levels of relations and child entities, forming a tree with entities at the leaves (see Figures 82 and 83). Each entity in the tree has a unique name and its own collection of attached variables. In a given *solution* from the domain, an entity may be active or inactive depending on the active state of its parent and the type of relation between them. The root entity is always active.

There are three types of relations between entities: *sub-parts*, *choice-point*, and *instance set*. A sub-parts relation, visually represented as an **and**, is a conjunction of its children. When the parent entity of a sub-parts relation is active, its child entities will necessarily be active. Sub-parts are used to represent required structure in a domain, such as the cost and performance characteristics of a computer component (the Details relation in Figure 83).

A choice-point relation, visually represented as an **xor**, is a disjunction of its children, with only **one child** being active at a time in a solution. Choice points are the source of generativity in CDOs, with all combinations of active/inactive choice point children forming the structure of the complete, unconstrained solution space. Because each choice under a choice point may have sub-structure, it is possible for different solutions from the same CDO to significantly differ in their tree structure.

Lastly, the instances set relation, visually represented as a **0..n**, creates *n* copies of its sub-structure in each solution. Like sub-parts, an instance set's child entities are active when its parent is active. Increasing *n*, the cardinality of the instance set, can exponentially expand the size of the solution space because all child choice points (and nested instance sets) are independent of each other. Constraints can be mapped across all child entities of an instance set, or may target specific instances by ordinal (ord).

Constraints. In addition to the structure of a domain, CDOs contain constraints that serve to prune out non-sensical or irrelevant sections of the solution space. The basic CDO constraint language (Table 10) is based on first-order logic with some additional operators for accessing and comparing entities and variables. Constraints typically set or key off of choice points, shutting down generativity in the Screamer constraint solver. In domains with instance sets, higher-order constraints can be mapped across instances (setting instance variable values), or used to constrain the set (e.g., at most one instance with choice A). Section 16.3.3 provides details on the higher-order constraint operators.

Operator	Description	Example
`==>`	Implication (If)	`(==> p q)`
`<==>`	Biconditional (IFF)	`(<==> p q)`
`not`	Negation (Not)	`(not p)`
`or`	Disjunction (Or)	`(or p q)`
`and`	Conjunction (And)	`(and p q)`
`e@`	Entity in CDO	`(e@ e1 e2)`
`v@`	Variable in CDO entity	`(v@ (e1 e2) v1)`
`equale`	Entity equality	`(equale e1 e2)`
`equalv`	Variable equality	`(equalv v1 v2)`
`let`	Local binding	`(let ((p p')) (equale p p'))`

Table 10: *First order CDO constraint language.*

16.3.2 Ball Example

Figure 82 shows a simple example domain. In this domain, a *ball* is broken down into two components: a *size* and a *sport*. Both components are choice points with three options, making for a total of 9 possible solutions. With no constraints, the solution space consists of all combinations of *size* and *sport* (Table 11). Note that there is an implicit conjunction at the root entity.

This solution space is not very useful, however, as it contains many nonsensical solutions. Golf

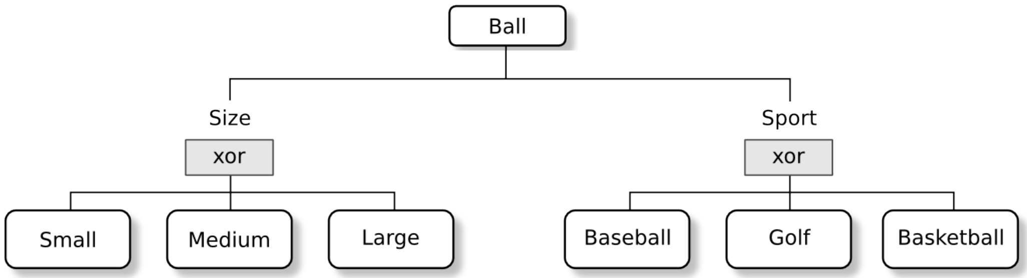


Figure 82: *The ball CDO. A ball has a size and associated sport.*

Solution	Size	Sport
1	Small	Baseball
2	Small	Golf
3	Small	Basketball
4	Medium	Baseball
5	Medium	Golf
6	Medium	Basketball
7	Large	Baseball
8	Large	Golf
9	Large	Basketball

Table 11: *Unconstrained solution space of the ball CDO (9 solutions).*

balls, baseballs, and basketballs should be small, medium, and large respectively. Three simple constraints will serve to constrain our domain:

1. $Small \iff Golf$
2. $Medium \iff Baseball$
3. $Large \iff Basketball$

Note that all three constraints are **bidirectional**. A unidirectional constraint, such as $Small \Rightarrow Golf$ would allow for more solutions, including medium and large golf balls. With these new constraints in place, our solution space size is reduced from 9 to 3 (Table 12), and now matches our intuition about real balls.

Translated into the Common LISP CDO framework, the top-level Ball entity becomes a `ball` function which takes a set of constraints as parameters. Each entity underneath Ball is accessible as a named LISP form that can be referenced in constraints. The `solutions` function explores the solution space defined by the top-level entity and constraints, and enumerates `:all` solutions or

Solution	Size	Sport
1	Small	Golf
2	Medium	Baseball
3	Large	Basketball

Table 12: *Constrained solution space of the ball CDO with all constraints (3 solutions).*

:one solution.

```
> (solutions
    (ball
      (<==> small golf)
      (<==> medium baseball)
      (<==> large basketball))
    :all)

;; 1. ball: small, golf
;; 2. ball: medium, baseball
;; 3. ball: large, basketball
```

16.3.3 Higher-Order Constraints

When a CDO includes one or more instance set relations, it is useful to write constraints that function across sets of entities and variables. Table 13 lists the operators available for instance sets and *higher order* constraints – e.g., constraints that take other constraints as parameters. Operators like `every` and `at-least` apply some constraint to an instance set and require, respectively, that all or at least *n* of them hold. The `mapc` operator is similar, but is used to apply side effects to all entities in an instance set, such as calculating the value of a variable. Any first order constraint can make use of the special `ord` to determine which instance they are being applied to. This allows for higher order constraints to be sensitive to ordering in an instance set; often used when order reflects spatial arrangement.

If the Ball entity in our example above was underneath an instance set (called `balls`), the 3 bidirectional constraints would now need to be applied across each `ball` instance. For example:

```
(every
  (and
    (<==> small golf)
    (<==> medium baseball)
    (<==> large basketball))
```

Operator	Description	Example
`n@`	Entity set under an instance	`(n@ inst)`
`ord`	Special variable for instance ordinal	`(equalv (v@ (e1) ord) 1)`
`exactly`	Exactly *n* instances match	`(exactly 2 cstr (n@ inst))`
`at-least`	At least *n* instances match	`(at-least 1 cstr (n@ inst))`
`at-most`	At most *n* instances match	`(at-most 3 cstr (n@ inst))`
`mapc`	Map constraint with side effects across instances	`(mapc cstr (n@ inst))`

Table 13: *Instance set and higher order CDO constraint operators.*

```
  (n@ balls))
```

The higher order `every` constraint would ensure that all `ball` instances adhered to the size/sport constraints.

16.3.4 Computer Configuration Example

To demonstrate the use of each relation (sub-parts, choice points, and instance sets) and higher order constraints, we will use a CDO that models the configuration of a desktop computer.
Figure 83 shows the CDO structure. At the top level, we say that a computer configuration consists of a set of components (0..*n*). Each component has a type (graphics, memory, sound), a role in the configuration (active, not active), and details of the product, such as its vendor, cost, and performance. Note that cost, performance, and model are *variables* attached to the Product entity. Attached variables can be used in constraints, but unlike choice points, Screamer does **not** generate possible values for them.
With no constraints and 8 components, the space of possible solutions is quite large. The 3 component types (2 memory types), 2 roles, and 3 vendors make up $(3+2) \times 2 \times 3 = 30$ possible choices. Assuming our configuration only has **8 components**, there are a total of 30^8 possible solutions! Clearly, we need some constraints before exploring the solution space.
Table 14 lists the details of the components we will be considering in this example. The first 5 components are new, and may be purchased for the given cost. The last 3 components are currently in use, so they cost nothing to continue using. Performance numbers have been associated with each component, with higher being better. We will say that the cost and performance of an entire configuration is the sum of the costs and performances of its **active** components. To make things

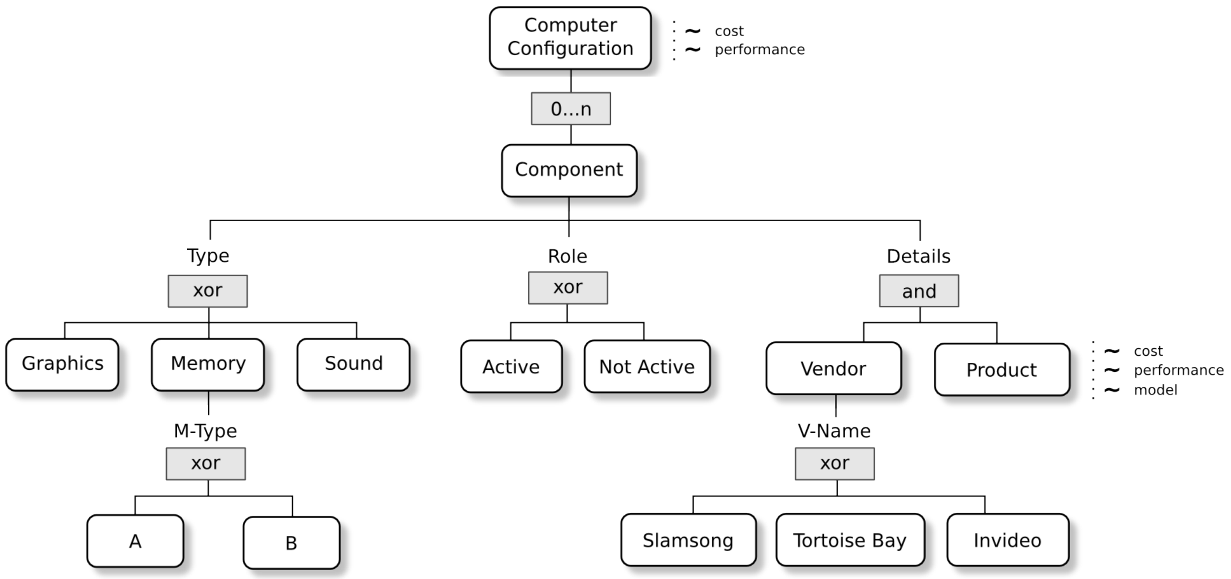


Figure 83: *A CDO representing a computer configuration with any number of components.*

interesting, a performance boost of 5 will be applied to any configuration with multiple memory chips that are all the same type (A or B).

	Type	Vendor	Model	Cost	Perf
1	Graphics	Invideo	D-Force	200	10
2	Memory	Slamsong	DDR9 (B)	20	10
3	Sound	Tortoise Bay	Waves	50	10
4	Memory	Slamsong	DDR7 (A)	10	5
5	Graphics	Invideo	B-Force	100	7
6	Memory	Slamsong	DDR7 (A)	0	5
7	Sound	Slamsong	Puddle	0	1
8	Graphics	Slamsong	A-Force	0	2

Table 14: *Components to consider for computer configuration example. Existing components have zero cost.*

We collect all of the constraints necessary to represent these 8 components into a LISP variable called `*available-components*` (full source code is available in Appendix C). The LISP code below prints the first solution from the constrained search space, including the summed cost and performance of all active components (those with a $*$ next to them).

```
> (solutions
    (computer-configuration
      *available-components*)
    :one)
```

```
;; Configuration (cost:380, perf:50):
;;   1. * [GRAPHICS] INVIDEO D-Force (c:200, p:10)
;;   2. * [MEMORY] SLAMSONG DRR9, B (c:20, p:10)
;;   3. * [SOUND] TORTOISE-BAY Waves (c:50, p:10)
;;   4. * [MEMORY] SLAMSONG DRR7, A (c:10, p:5)
;;   5. * [GRAPHICS] INVIDEO B-Force (c:100, p:7)
;;   6. * [MEMORY] SLAMSONG DRR7, A (c:0, p:5)
;;   7. * [SOUND] SLAMSONG Puddle (c:0, p:1)
;;   8. * [GRAPHICS] SLAMSONG A-Force (c:0, p:2)
```

If we interpret this solution as a recommended configuration, it suggest our computer should have 3 graphics cards, two sound cards, and 3 memory chips of varying types. A typical desktop computer, however, will only have a single graphics/sound card, and one or two memory chips. We will further constrain the solution space by enforcing the constraints in Table 15. Our computer will only have 4 components slots, and must have 1 graphics card, 1 sound card, and 1 or 2 memory chips.

Constraint	**Description**
`max-4-active`	A maximum of 4 components may be active in any configuration.
`only-1-graphics-card`	A configuration must have one graphics card.
`only-1-sound-card`	A configuration must have one sound card.
`1-or-2-memory`	A configuration must have 1 or 2 memory chips.

Table 15: *Default constraints for computer configuration example.*

Collecting the constraints from Table 15 into a LISP variable called `*default-constraints*`, let's look at the first solution again:

```
> (solutions
    (computer-configuration
      *all-components*
      *default-constraints*)
    :one)

;; Configuration (cost:280, perf:35):
;;   1. * [GRAPHICS] INVIDEO D-Force (c:200, p:10)
;;   2. * [MEMORY] SLAMSONG DRR9 B (c:20, p:10)
;;   3. * [SOUND] TORTOISE-BAY Waves (c:50, p:10)
;;   4. * [MEMORY] SLAMSONG DRR7 A (c:10, p:5)
;;   5.   [GRAPHICS] INVIDEO B-Force (c:100, p:7)
;;   6.   [MEMORY] SLAMSONG DRR7 A (c:0, p:5)
;;    7.   [SOUND] SLAMSONG Puddle (c:0, p:1)
;;    8.   [GRAPHICS] SLAMSONG A-Force (c:0, p:2)
```

Now only 4 components are active, and we have the correct number of component types for our system. Note that the performance is precisely the sum of the individual component performances because our memory chips are of different types. This configuration includes all new, high performance components, and may be the most expensive option. What would a solution look like if we didn't want to spend any money (i.e., use only existing components)?

```
;; One solution, force no cost
> (solutions
    (computer-configuration
      *all-components*
      *default-constraints*
      ;; Apply constraint across all components
      (every
        ;; If a component is active...
        (if (equale (e@ component role) active)
            ;; ...its cost must equal zero.
            (equalv (v@ (component details product) cost) 0))
        (n@ components)))
    :one)

;; Configuration (cost:0, perf:8):
;;    1.   [GRAPHICS] INVIDEO D-Force (c:200, p:10)
;;    2.   [MEMORY] SLAMSONG DRR9, B (c:20, p:10)
;;    3.   [SOUND] TORTOISE-BAY Waves (c:50, p:10)
;;    4.   [MEMORY] SLAMSONG DRR7, A (c:10, p:5)
;;    5.   [GRAPHICS] INVIDEO B-Force (c:100, p:7)
;;    6. * [MEMORY] SLAMSONG DRR7, A (c:0, p:5)
;;    7. * [SOUND] SLAMSONG Puddle (c:0, p:1)
;;    8. * [GRAPHICS] SLAMSONG A-Force (c:0, p:2)
```

While this configuration is definitely cheaper than the previous one, its less than a quarter of the performance. Ideally, we'd like to explore the entire (constrained) solution space, and sort all solutions by their cost and performance numbers. The CDO framework supports this type of search via user-defined *utility* and *objective* functions.

16.3.5 Utility and Objective Functions

In addition to `:all` and `:one`, the `solutions` function accepts requests for the `:best` solution(s). The extra keyword arguments `:utility-fun` and `:objective-fun` are user-defined functions for assessing the value of a given solution (its utility), and for sorting the utility values (the objective). The example below defines a new function `performance-utility` that simply extracts a configuration's performance value. The built-in > function serves as our objective function, putting higher utility values (better performing solutions) on top. All solutions with the "best" utility value in the explored solution space are returned.

```
;; Extract the summed performance
(defun performance-utility (cfg)
  (v@ (cfg) performance))

;; Best performance
> (solutions
    (computer-configuration
      *available-components*
      *default-constraints*)
    :best
    :utility-fun #'performance-utility
    :objective-fun #'>)

;;  Configuration (cost:280, perf:35):
;;   1. * [GRAPHICS] INVIDEO D-Force (c:200, p:10)
;;   2. * [MEMORY] SLAMSONG DRR9, B (c:20, p:10)
;;   3. * [SOUND] TORTOISE-BAY Waves (c:50, p:10)
;;   4. * [MEMORY] SLAMSONG DRR7, A (c:10, p:5)
;;   5.   [GRAPHICS] INVIDEO B-Force (c:100, p:7)
;;   6.   [MEMORY] SLAMSONG DRR7, A (c:0, p:5)
;;   7.   [SOUND] SLAMSONG Puddle (c:0, p:1)
;;   8.   [GRAPHICS] SLAMSONG A-Force (c:0, p:2)

;;  Configuration (cost:270, perf:35):
;;   1. * [GRAPHICS] INVIDEO D-Force (c:200, p:10)
;;   2. * [MEMORY] SLAMSONG DRR9, B (c:20, p:10)
;;   3. * [SOUND] TORTOISE-BAY Waves (c:50, p:10)
;;   4.   [MEMORY] SLAMSONG DRR7, A (c:10, p:5)
;;   5.   [GRAPHICS] INVIDEO B-Force (c:100, p:7)
;;   6. * [MEMORY] SLAMSONG DRR7, A (c:0, p:5)
;;   7.   [SOUND] SLAMSONG Puddle (c:0, p:1)
;;   8.   [GRAPHICS] SLAMSONG A-Force (c:0, p:2)

;; Configuration (cost:260, perf:35):
;;   1. * [GRAPHICS] INVIDEO D-Force (c:200, p:10)
;;   2.   [MEMORY] SLAMSONG DRR9, B (c:20, p:10)
;;   3. * [SOUND] TORTOISE-BAY Waves (c:50, p:10)
;;   4. * [MEMORY] SLAMSONG DRR7, A (c:10, p:5)
;;   5.   [GRAPHICS] INVIDEO B-Force (c:100, p:7)
;;   6. * [MEMORY] SLAMSONG DRR7, A (c:0, p:5)
;;   7.   [SOUND] SLAMSONG Puddle (c:0, p:1)
;;   8.   [GRAPHICS] SLAMSONG A-Force (c:0, p:2)
```

Three solutions with a performance value of 35 (the maximum) exist in the solution space. The first solution should look familiar: a recommendation to purchase all new components. The second solution, however, achieves the same performance with a lower cost by reusing the existing DDR7 memory chip. The final solution reduces the cost further by purchasing a new, lower-performing, DDR7 chip instead of the new DDR9 chip. This is due to the memory performance boost mentioned earlier – a bonus 5 points are added to any configuration with multiple memory chips of the same type (A or B). Conceptually (and verbosely), the memory boost constraint looks like this:

```
;; Get the sum of all component performance numbers.
(let ((cfg-performance
        (sum (mapc (v@ (component performance)) (n@ components)))))
  ;; Give +5 to configs with same memory types.
  (if (and
        ;; If at least 2 active memory components...
        (at-least 2 (and (equale (e@ component role) active)
                         (equale (e@ component type) memory)))
        (or
          ;; ...each active memory chip must be type A...
          (every (if (and (equale (e@ component role) active)
                          (equale (e@ component type) memory))
                      (equale (e@ component type memory m-type) A))
                 (n@ components))

          ;; ...or type B.
          (every (if (and (equale (e@ component role) active)
                          (equale (e@ component type) memory))
                      (equale (e@ component type memory m-type) B))
                 (n@ components)))

       ;; If the memory chips are the same type, give the boost...
       (equalv (v@ (configuration) performance)
               (+ 5 cfg-performance))

       ;; ...otherwise, just return the sum of performances.
       (equalv (v@ (configuration) performance)
               cfg-performance))))
```

Multi-Objective Functions. As a last demonstration, we will show how multi-objective functions can be used to order a solution space across multiple dimensions. The previous example used performance-utility to sort by performance alone. When performances are highest and equivalent, we would like to also minimize costs. This can be done manually for our computer configuration example because there are only 3 solutions with the maximum performance, but there may be many more in the general case.

The LISP code below defines a new utility function performance-and-cost-utility that creates a two-element list for each configuration with the performance value first and the cost value second. The next function, better-2, sorts solutions for the highest performance first, and the lowest cost second. With this as our objective function, we should get the best performing configuration with the lowest cost value. Indeed, running solutions with the new function returns a single solution: the configuration with mostly new components that purchases a DDR7 memory chip! A graphical depiction of this solution is shown in Figure 84 with component 6 enlarged to show excluded choice points and the values of attached variables.

```
;; Extract performance and cost as a list
(defun performance-and-cost-utility (cfg)
  (list
   (v@ (cfg) performance)
   (v@ (cfg) cost)))

;; Highest performance (first), lowest cost (second)
(defun better-2 (a b)
  (cond
    ;; Performances equal - compare costs
    ((eq (first a) (first b)) (< (second a) (second b)))
    ;; Compare performances
    (t (> (first a) (first b)))))

;; Best performance with lowest cost
> (solutions
    (computer-configuration
      *available-components*
      *default-constraints*)
    :best
    :utility-fun #'performance-and-cost-utility
    :objective-fun #'better-2)

;; Configuration (cost:260, perf:35):
;;   1. * [GRAPHICS] INVIDEO D-Force (c:200, p:10)
;;   2.   [MEMORY] SLAMSONG DRR9, B (c:20, p:10)
;;   3. * [SOUND] TORTOISE-BAY Waves (c:50, p:10)
;;   4. * [MEMORY] SLAMSONG DRR7, A (c:10, p:5)
;;    5.   [GRAPHICS] INVIDEO B-Force (c:100, p:7)
;;    6. * [MEMORY] SLAMSONG DRR7, A (c:0, p:5)
;;    7.   [SOUND] SLAMSONG Puddle (c:0, p:1)
;;    8.   [GRAPHICS] SLAMSONG A-Force (c:0, p:2)
```

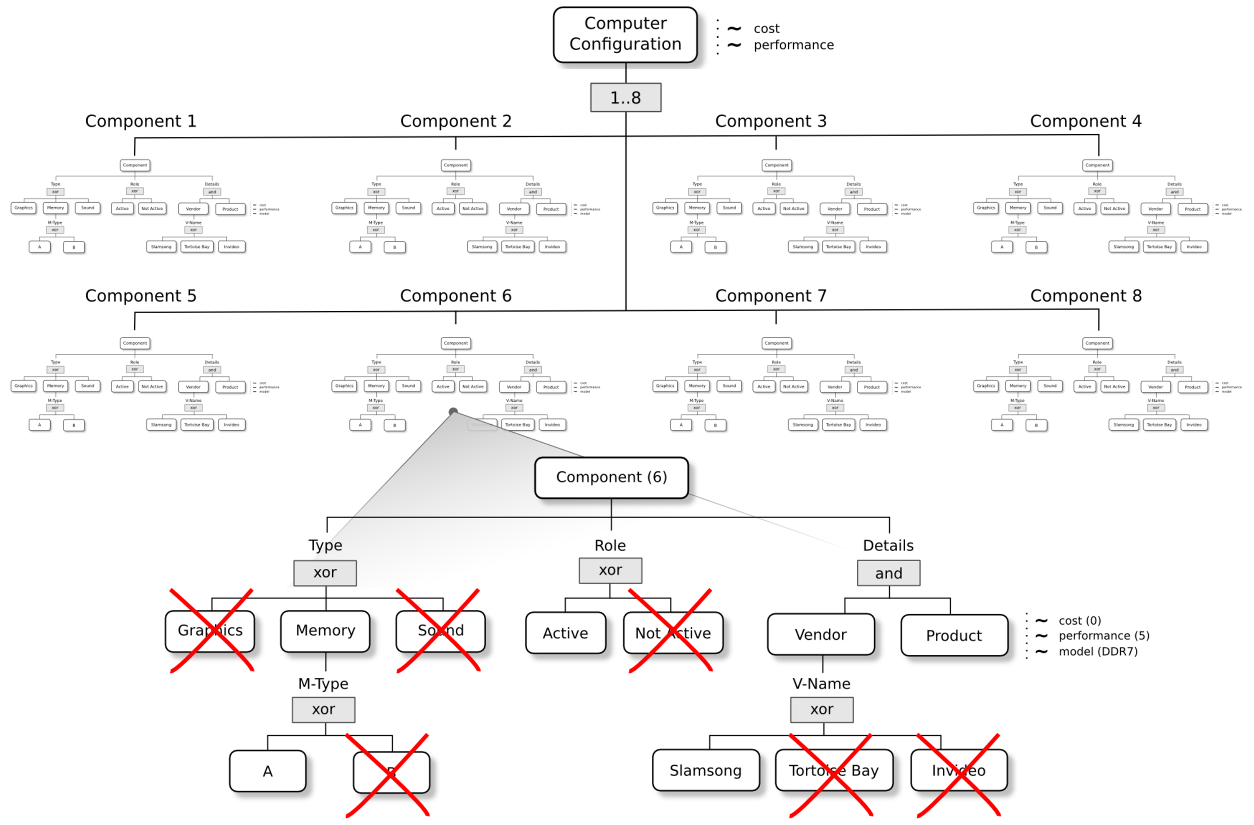


Figure 84: *Graphical representation of a single solution. One component is expanded to show excluded choice points and variable values.*

16.3.6 Constraint Knowledge and Cognition

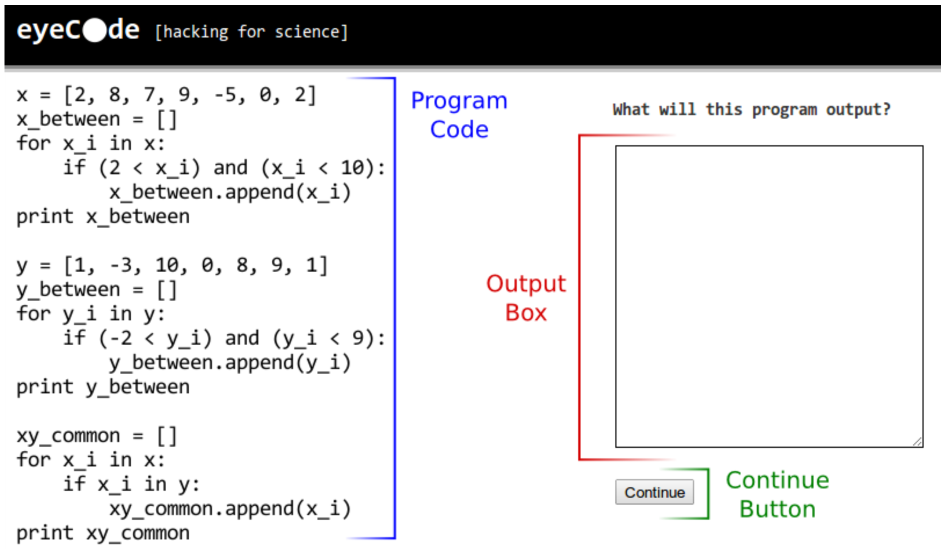
The computer and ball examples above demonstrate interesting ways to use CDOs, but the *cognitive* aspect of these domain ontologies may not be immediately apparent. Depending on how they are designed and searched, CDO solution spaces can be used to abduce likely explanations from evidence, deduce the remainder of incomplete knowledge, and induce possible consequences from observed (or desired) actions.

In *A Framework for Modeling and Simulation of the Artificial*, Douglass et al. describe a situated, autonomous agent that searches rooms in a task environment for a reward [33]. The agent contains a library of behaviors, called *behavior models*, which detail **how** the agent can achieve a specific sub-task. A CDO is used to map situational factors, such as the overall state of the task and current percepts, to specific behaviors. In other words, the agent's CDO helps the agent decide **what** to do next. This same CDO could also be used to go the other direction: from behaviors to percepts. Given a set of asserted behaviors, the CDO's solution space will contain all percepts relevant to their selection. If the agent had determined a candidate set of next behaviors based on some other knowledge, these inferred percepts would tell the agent *what to look for* in the environment in order to exclude candidate behaviors.

Section 18 describes a CDO-based model of program comprehension, called Nibbles. This model represents programs using a CDO, and knowledge of the Python programming language as constraints. Over time, Nibbles builds a "mental model" of a program, line-by-line, and is capable of inferring token, line, and line group semantics from raw text. Importantly, Nibbles is also capable of *making errors* due to code ambiguities and invalid expectations.

17 Part 3: Mr. Bits

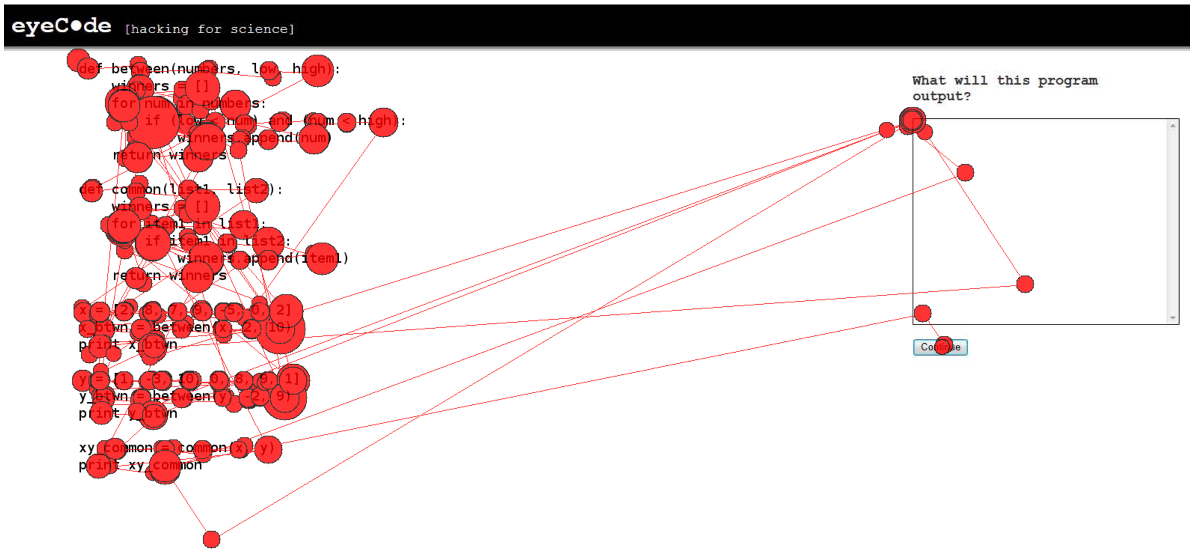
The Mr. Bits model operationalizes components of Cant et. al's Cognitive Complexity Metric [16] (CCM) within the ACT-R cognitive architecture [3]. Mr. Bits gets its name from the Mr. Chips model of text reading [57]: a simple character-based model that minimizes fixations to disambiguate words. Unlike Mr. Chips, Mr. Bits reads small Python programs, and types their printed output. Mr. Bits minimizes fixations by remembering what has been read, and by only fixating novel or forgotten lines of code. Using Python's built-in debugger, Mr. Bits transforms Python programs into ACT-R scripts which simulate a programmer "reading" and evaluating the program within the eyeCode experiment interface (Figure 85). The final output is a series of human-like fixations and keystroke timings which, when plotted on top of the program's code (Figure 86), look similar to human eye-tracking trials from the eyeCode experiment (Section 12). The **chunking** and **tracing** processes described in the CCM are realized through ACT-R's declarative memory and visual modules (specially, EMMA [76]). Keystrokes are also simulated via the ACT-R motor module, which utilizes Fitts Law [41].



```
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
```

Figure 85: *Interface used by human programmers in the eyeCode experiment. Mr. Bits uses the same interface.*

While Mr. Bits *appears* to read and evaluate a program from scratch, the model does not actually parse text or determine the evaluation order of lines, and therefore **cannot make errors**. Despite this limitation, the model serves as a starting point for a truly quantitative, cognitive model of program comprehension. Mr. Bits produces human-like eye movements and keystrokes by "reading" over code tokens, recalling variable values from memory, and performing mental arithmetic. A second model, called Nibbles (Section 18), complements Mr. Bits by transforming raw text into an internal (mental) representation of the underlying program. This mental model can be used to determine what to do next during each step of evaluation. Future work will integrate the Mr. Bits and Nibbles models, removing the need for the Python debugger. The following sections describe the details of Mr. Bits, and Section 17.5 compares the model's output to human data collected from the same 24 Python programs in the eyeCode experiment [46].



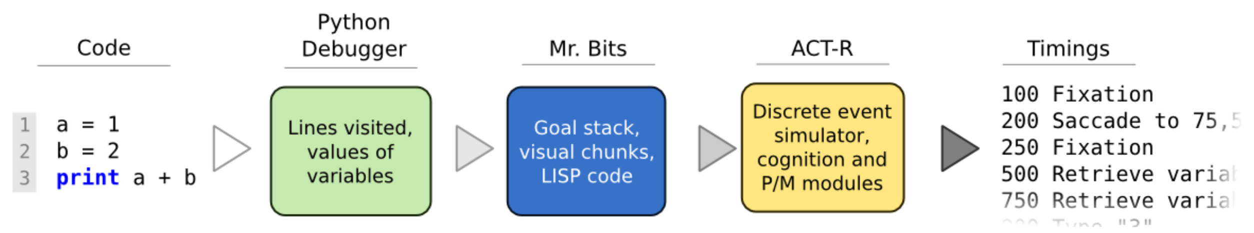
```
def between(numbers, low, high):
    winners = []
    for num in numbers:
        if (low < num) and (num < high):
            winners.append(num)
    return winners

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
```

Figure 86: *Mr. Bits fixation plot for* `between functions` *program.*

17.1 Architecture

Mr. Bits takes a Python program (a text file) as input, and produces a time series of fixations and keystrokes as output. In between, there are three important stages (Figure 87): (1) the built-in Python debugger is used to parse and evaluate the program, (2) an ACT-R LISP script is generated with a program-specific goal stack for each step of the evaluation, and (3) the ACT-R script is executed, and the resulting trace is transformed into fixation/keystroke timings. These data can then be compared to human data to, for example, see if human programmers and Mr. Bits spent the same amount of relative time on each line of the program.

The **first stage** of the Mr. Bits model parses and evaluates a Python program using the built-in


Code
Python Debugger
Mr. Bits
ACT-R
Timings
Lines visited, values of variables
Goal stack, visual chunks, LISP code
Discrete event simulator, cognition and P/M modules

```
a = 1
b = 2
print a + b
```

```
100 Fixation
200 Saccade to 75,
250 Fixation
500 Retrieve varia
750 Retrieve varia
```

Figure 87: *Mr. Bits model workflow. Code is transformed into fixation and response timings via Python and ACT-R.*

Python debugger. The output from this stage is a series of code lines in evaluation order, and the values of each variable during the course of evaluation. Note that the ACT-R portion of Mr. Bits does **not** parse or evaluate code. This is the focus of our second model, Nibbles (see Section 18). As the Python program is parsed, variables are tagged with contextual information, such as which function they are defined in and how many times they are assigned to. When program evaluation is simulated in ACT-R, Mr. Bits uses this information to resolve variable ambiguities, and trace to specific code locations for variable values.

During the **second stage**, a self-contained LISP script is generated with a complete visicon (visual locations of code words), a goal stack with each step of program evaluation, and a collection of general-purpose productions for reading/remembering text, tracing variable values, and typing responses. A complete example LISP script for the `funcall space` program is available in Appendix D. The Visual, Declarative, and Manual ACT-R modules are relied upon to generate human-like timings (see Figure 80 for a description of each module). The goal stack contains four general categories of goals:

1. **Look at/remember line** - The leftmost word of every evaluated code line is first fixated by the Visual module. A declarative memory request for the "details" of the line is made and, if not present, every word in the line is fixated from left to right.

2. **Store variable value/location/reference** - Every assignment statement, `for` loop, or mutable function call (e.g., `list.append`) results in a new declarative memory chunk with the target variable's name, context, and *physical location*. Future attempts to recall the variable's value will require at least one trace to this location before the value is stored in declarative memory. Calls to functions defined in the same program (e.g., `def f(x)`) create a series of variable reference chunks, linking the function's parameters to the supplied arguments. Future requests for a function's parameter value will cause Mr. Bits to follow the reference back to the call site arguments (e.g., `f(1)`).

3. **Recall variable value** - When a variable's value is needed (for a computation, printing, etc.), a declarative memory request is made for the exact name in the context of the current

Goal	ACT-R Module(s)	Description
Go to line	Visual/DM	Moves the virtual eyes to the first character of a line. If the details of the line cannot be retrieved from memory, every token is read.
Remember line	DM	Stores the details of the current line in memory.
Recall variable	Visual/DM	Attempts to retrieve the value of a variable from memory. Failure results in a retrieval of the variable's location, a visual trace to it, and storage of the result in memory.
Store variable	DM	Stores a variable's value in memory.
Compute sum/product	DM	Retrieves a sum or product from memory.
Compare numbers	DM	Retrieves a numeric relationship (less/greater than) from memory.
Evaluate Boolean expression	DM	Retrieves the result of a boolean expression (AND/OR) from memory.
Fixate output box	Visual	Locates the output box on the screen and fixates the upper-left corner.
Type response	Manual	Types a text response, character by character.

Table 16: *Possible goals in the Mr. Bits model. DM stands for Declarative Memory.*

function. If not present, Mr. Bits attempts to recall the variable's physical location, follow references to other variables, or consult previous variable definitions with the same name. If all else fails, the process is repeated within the previous context (either another function or the global context). The values of traced variables are stored in declarative memory, so future retrievals will succeed until the value is forgotten.

4. **Typing responses** - Every `print` statement produces a set of goals, causing Mr. Bits to fixate the experiment's output box, type each character of the response, and re-fixate the previous line. When program evaluation is complete, Mr. Bits will fixate the "continue" button, move its hand to the virtual mouse, and click the button to end the trial (see Figure 85).

State Transition Diagram. Figure 88 shows a state-transition diagram for the ACT-R simulation (final stage) of Mr. Bits. At a high level, goals are processed one by one from the goal stack produced during the previous model stage. When the goal stack is empty, Mr. Bits clicks the continue button to end the trial. Table 16 describes each goal in detail as well as the specific ACT-R modules involved. The declarative memory module is used heavily, with processes like variable look-up and arithmetic cast as memory retrievals (a common practice in ACT-R modeling [3]).

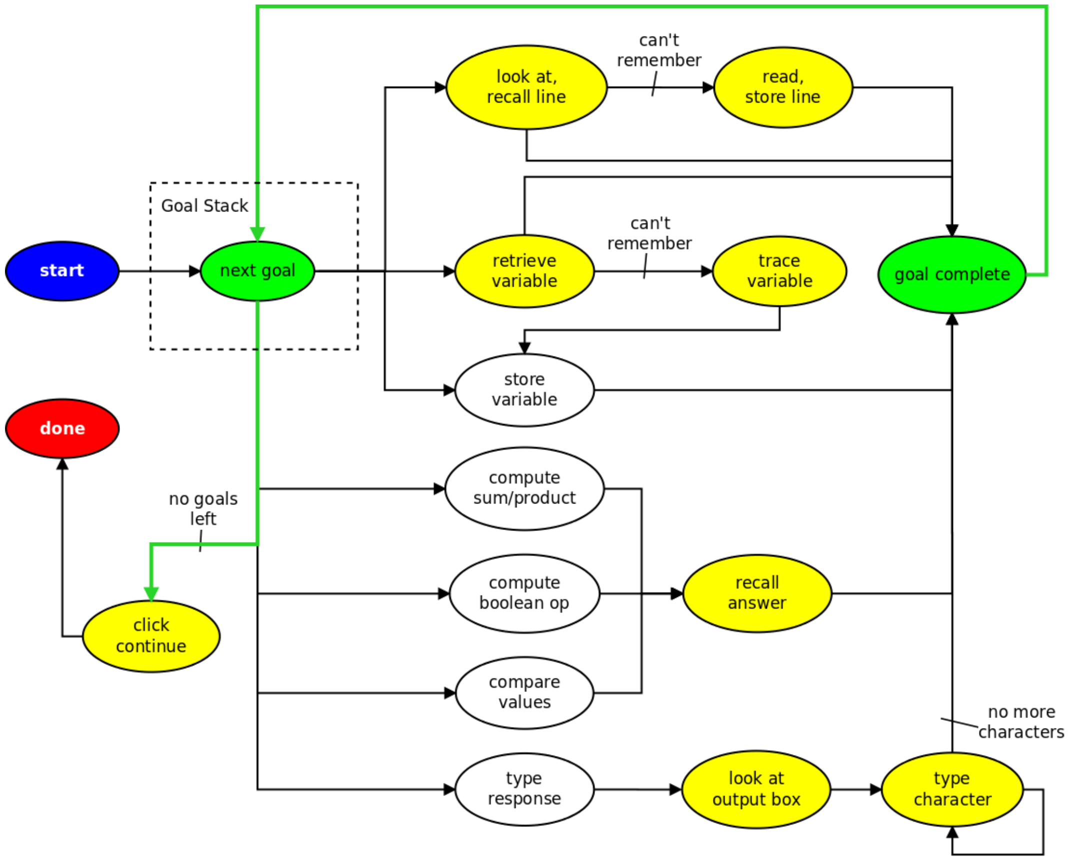
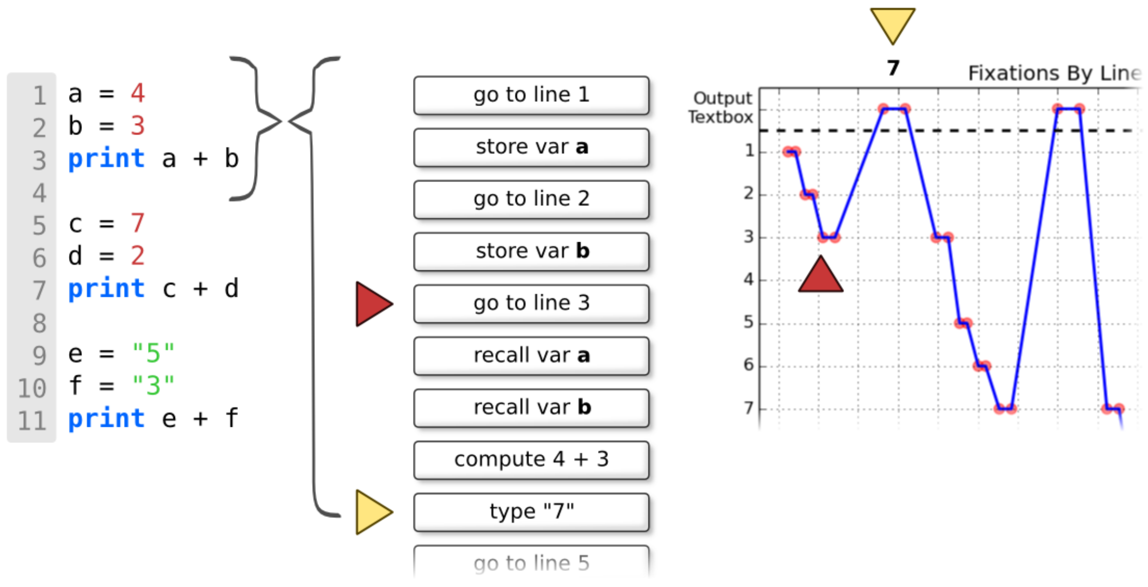


Figure 88: *General state diagram for a Mr. Bits ACT-R script. States highlighted in yellow may have variable timings when ACT-R's sub-symbolic mode is enabled.*

The Goal Stack. A sample program and goal stack produced by Mr. Bits is shown in Figure 89. The fixation timeline on the right-hand side of the figure shows the model fixating specific code lines and the output box when typing responses. While Mr. Bits' goal stack is pre-populated during the parsing stage, the fixations generated during model execution depend on dynamic factors. First, code lines are only read (i.e., each whitespace-separated token is fixated) if they have not been viewed before. Second, variable values are only stored in the model's declarative memory (DM) once they have been traced. As a result, Mr. Bits will shift visual attention to variable values as they are needed. Finally, when ACT-R's sub-symbolic behavior is enabled (described in Section 17.3), declarative memory retrievals for code lines and variable values can fail after enough time has passed and their DM activations have fallen below threshold. In other words, Mr. Bits can forget. On the other side of the coin, sub-symbolic behavior will speed up the retrieval of frequently and recently used chunks. Often-used arithmetic facts, such as $5 \times 8 = 40$, may have increased activation in memory, resulting in faster in-situ retrievals. Existing DM chunks and activations at the start of a trial represent Mr. Bits' *long-term experience*, while accumulated post-trial chunks/activations represent *short-term learning*. We do not currently retain short-term learned information across Mr. Bits trials, though this would be an interesting path for future research.



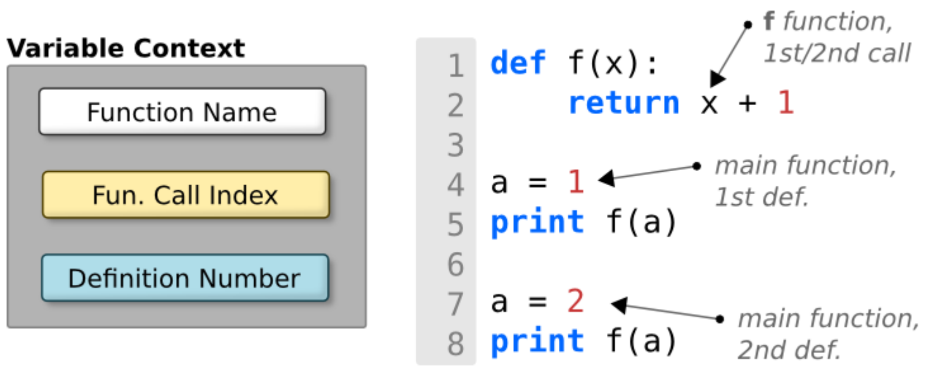
```
 1  a = 4
 2  b = 3
 3  print a + b
 4
 5  c = 7
 6  d = 2
 7  print c + d
 8
 9  e = "5"
10  f = "3"
11  print e + f
```

Figure 89: *Example of a Mr. Bits goal stack (center) for a Python program (left). Processing the goals produces fixations and keystrokes (right).*

17.2 Variables and Context

In the scope of our simple Python programs, a variable has three pieces of information that uniquely identify it. First, the surrounding function name is used to differentiate global and local function scope. Mr. Bits will use the local function scope first when resolving a variable by name. Second, the number of times a function has been called – its "function call index" – helps distinguish between multiple calls to the same function. In Figure 90, `f(x)` is called twice, and so the local `x` variable within `f` will potentially have different values during each call. Mr. Bits considers these two "versions" of `x` to be separate, but related, variables.

Lastly, variables can be reassigned within the same scope and function call index. In Figure 90, the variable `a` (in the global scope) is assigned a value twice. Mr. Bits treats these as two different variable definitions with the same name, and will always trace to the most recent definition first (i.e., the definition with the highest line number in the current scope).



```
def f(x):
    return x + 1

a = 1
print f(a)

a = 2
print f(a)
```

Figure 90: *Variable context*

These three pieces of information: function scope, function call index, and definition number, allow Mr. Bits to uniquely identify variables in each of our sample Python programs. More complex programs, especially those with class definitions, will require extensions to this basic notion of variable context. Presently, Mr. Bits locates values from the right-hand side of assignment statements (e.g., `x = 5`) and passed as function arguments (e.g., `f(5)`). With classes, it will be necessary to maintain the type information for each definition of a variable. Python allows variables to change type when reassigned, so the expression `x.foo(5)` may be dispatched to different code depending on the (current) type of `x`. We leave these extensions as future work.

17.3 Sub-symbolic Chunking and Tracing

ACT-R can operate in *symbolic* or *sub-symbolic* mode. In the former mode, all ACT-R modules (visual, memory, etc.) behave with consistent timings. For example, the declarative memory module in symbolic mode will retrieve chunks immediately, bypassing the activation calculus. When sub-symbolic behavior is enabled, however, retrieval times will depend on the frequency and recency of a chunk's use. Additionally, a chunk can be "forgotten" if its activation falls below a critical threshold (see `rt` in Table 17). Mr. Bits relies heavily on declarative memory for remembering code line details, variable values, traced locations, and arithmetic/boolean facts (see Section 17.4) for details). In sub-symbolic mode, Mr. Bits will get *faster* at remembering a recently or often-used variable value. These values can also be forgotten over time, though none of our sample programs are long enough to demonstrate this behavior.

The activation calculus in ACT-R subsumes the two familiarity parameters in the Cognitive Complexity Metric (Section 16.1). The R_F and T_F parameters, respectively chunk and trace familiarity, are operationalized as retrieval latency in declarative memory. Frequently-used, or familiar, code lines, variable values, and trace locations will be retrieved quickly whereas less familiar items will take more time. If we equate cognitive complexity with the time it takes Mr. Bits to read through a program, this successfully captures the intentions of the familiarity CCM parameters because programs with many unique operations will take longer to evaluate.

A total of 5 ACT-R parameters have been given non-default values in Mr. Bits (Table 17). The latency factor (`lf`) and retrieval threshold (`rf`) parameters were set via an informal search, in order to better fit the human data. These two declarative memory parameters are among the most frequently modified ACT-R parameters, and the chosen values are within the ranges used in other human studies [107]. The `imaginal-delay` parameter was set to 0 in order to quickly move newly created chunks from the imaginal buffer into declarative memory. Mr. Bits does not use the imaginal buffer to hold a representation of the current line or variable, so there is no modeling need for a delay. Lastly, the two motor parameters (feature preparation and burst time delays) were reduced by a factor of 15 in order to account for programmers' above average typing speeds. As with the declarative memory parameters, an informal search was used to produce visibly similar behavior relative to human trials. Neither motor parameter has an entry in the Max Planck

ACT-R Parameter	Default	Value	Description
`lf`	1.0	0.01	Latency factor (F in the retrieval equation).
`rt`	0.0	-2.0	Retrieval threshold. Minimum activation for retrieval.
`imaginal-delay`	0.2	0.0	Delay in seconds for imaginal buffer request.
`motor-feature-prep-time`	0.05	0.001	Time in seconds to prepare each movement feature.
`motor-burst-time`	0.05	0.001	Minimum time in seconds for any motor movement.

Table 17: *ACT-R parameters with non-default values used in Mr. Bits.*

Institute for Human Development's ACT-R parameter database [107], so it is unknown whether our values are cognitively plausible.

17.4 Sums, Comparisons, and Boolean Expressions

Mr. Bits computes sums/products/differences, does numeric comparisons ($x < y$), and evaluates simple boolean expressions ($x \wedge y$) using declarative memory. Each of these "facts" resides in Mr. Bits' declarative memory – e.g., the sum of 2 and 3 is 5 – and has been given a high *base level activation*, representing that they are very well rehearsed. We do not model an explicit *process* of addition, subtraction, etc., such as counting up or down from an anchoring number, because (1) virtually all numeric operations in our simple Python programs are done with small operands, and (2) participants in the eyeCode experiment were experienced enough to have memorized these basic mathematical facts. ACT-R models of basic arithmetic exist [3], but we assume our participants have internalized these facts directly in long-term memory.

When Mr. Bits "computes" the sum of two numbers, x and y, a declarative memory retrieval is initiated with the constraints that it must be a `sum-fact` whose operands are x and y. Operations with more than two operands are serialized into multiple retrievals, each with only two operands. For example, the sum $1 * 2 + 3$ would be transformed into two retrievals:

1. A `prod-fact` with operands 1 and 2 (result: 2),
2. A `sum-fact` with operands 2 and 3 (result: 5)

Mr. Bits relies on the Python debugger to determine the order of operations, and so is not capable of making arithmetical errors as long as its declarative memory facts are consistent.

If ACT-R's subsymbolic behavior is disabled, there is little difference between declarative memory and a database. With subsymbolic effects, however, subsequent retrievals of the same fact will be *faster*, especially when done in quick succession. For example, Mr. Bits will take less time to compute $(2 \times 2) + (2 \times 2)$ than $(2 \times 2) + (2 \times 3)$ because, in the former case, the fact that $2 \times 2 = 4$ will be retrieved quicker the second time. Because chunks in ACT-R's declarative memory have an activation value, the same system can model both long-term and short-term/working memory [92]. Capacity limitations for short-term memory appear in ACT-R as a side effect due to (1) the ability to retrieve only one chunk at a time, and (2) the time costs associated with retrievals and productions (steps in the model). Only a fixed number of chunks can be attended to before activation decay becomes a factor in retrieval. Cognitive theories of program complexity often cite Miller's Magic Number [63] as a potential source of complexity – e.g., having a programmer attend to too many things simultaneously in a snippet of code will make it harder to understand [96, 24, 68]. Mr. Bits will behave as if it has these capacity limitations, but they are an emergent property of declarative memory, not an arbitrary constant.

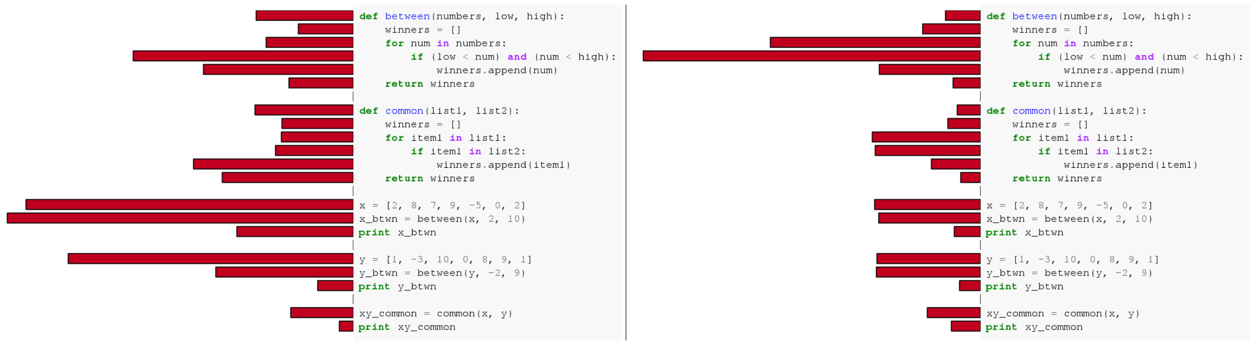
Mr. Bits currently contains sum, product, difference, less-than, and greater-than facts for numbers zero to ten. In order to model more experienced programmers, additional facts could be added to declarative memory. For example, intermediate and advanced programmers will quickly note that $2 \times 2 \times 2 = 2^3 = 8$, and not need to mentally compute the intermediary products.

17.5 Results and Discussion

In order to evaluate Mr. Bits, we compare the model's performance on 24 of the 25 programs used in our eyeCode experiment [46] with data from the 29 human programmers. All model runs used the same ACT-R parameters, as described in Table 17. We considered 4 different versions of Mr. Bits, each with combination of two options. The first option was having ACT-R's subsymbolic computing turned on (SSC) or off (SC). Second, Mr. Bits had the ability to either perfectly remember list variables (RL) or forget list variables (FL). The reasons for this second option are described in more detail below.

17.5.1 Human-Model Comparison

We compared the relative spent on each line of each program by Mr. Bits to the time spent by our human participants. For some programs, a large difference is visually apparent (Figure 91, human data is on the left). If Mr. Bits is allowed to remember the values of **all** variables after tracing them, it will avoid the need to continually refer back to *lists* (like `x` and `y`) as humans do. We created a setting for Mr. Bits that stopped it from remembering non-empty list variable values, forcing the model to trace those variables over and over. With this setting enabled, the shift in where time is spent is again visually apparent (Figure 92), and appears closer to the human data. The two versions of Mr. Bits, with the setting disabled or enabled, are referred to as RL (Remember Lists) and FL (Forget Lists). Combined with the option to place ACT-R in symbolic or sub-symbolic mode, we have four different versions of Mr. Bits to compare against human data:



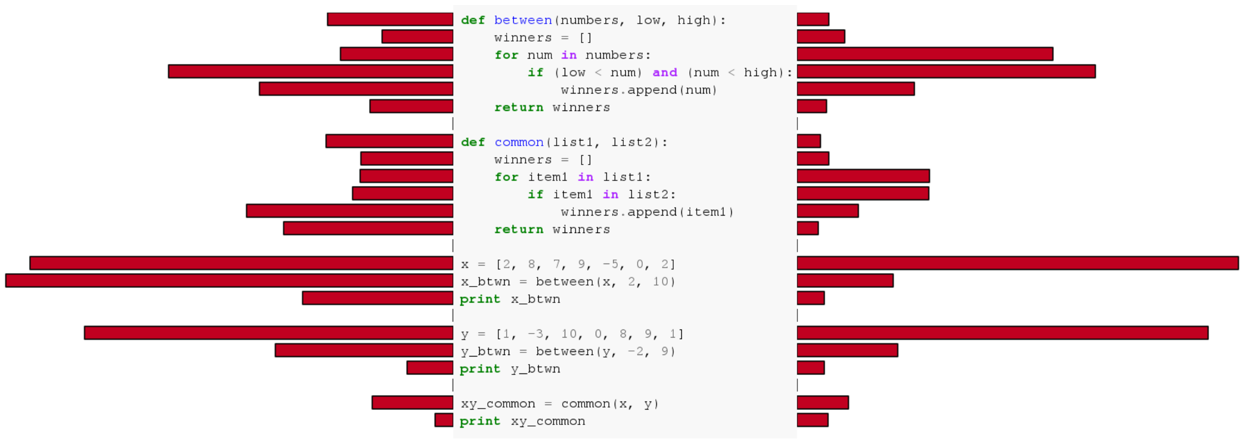
```
def between(numbers, low, high):
    winners = []
    for num in numbers:
        if (low < num) and (num < high):
            winners.append(num)
    return winners

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
```

```
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
```

Figure 91: *Relative time spent on each line of* `between functions` *by human participants (left) and Mr. Bits Remember Lists (right).*

1. **SC + RL** - Symbolic Computing + Remember Lists. All declarative memory retrievals return immediately, and lists can be fully remembered.
2. **SC + FL** - Symbolic Computing + Forget Lists. All declarative memory retrievals return immediately, and lists are traced every time.
3. **SSC + RL** - Sub-Symbolic Computing + Remember Lists. All declarative memory retrievals return immediately, and lists can be fully remembered.
4. **SSC + FL** - Sub-Symbolic Computing + Forget Lists. All declarative memory retrievals return immediately, and lists are traced every time.

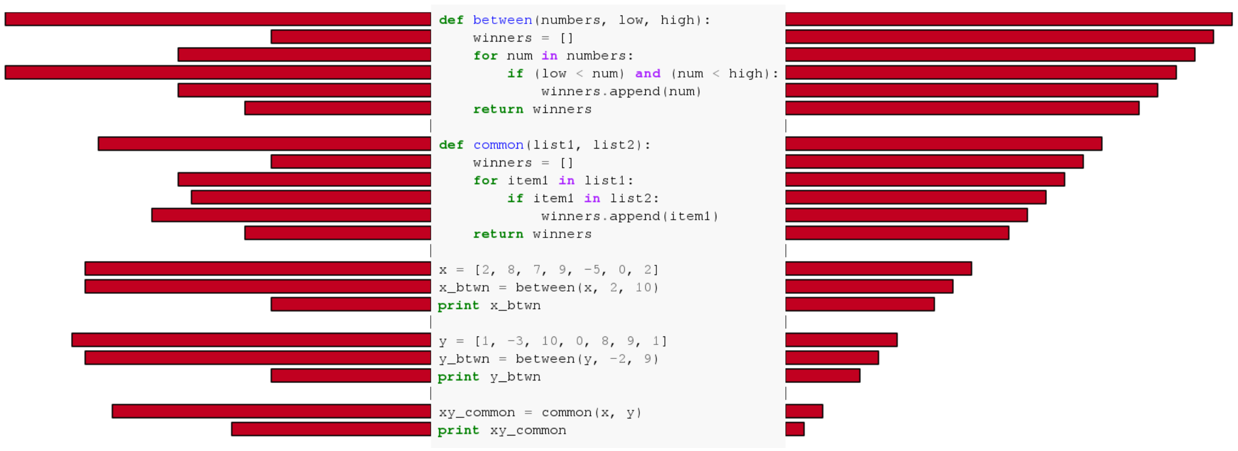
Which version of Mr. Bits correlates best with human data? We compared model runs from each version to human data for each program by computing the Spearman correlation between the



```
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
```

Figure 92: *Relative time spent on each line of `between functions` by human participants (left) and Mr. Bits Forget Lists (right).*

human and model average times spent on each line. In addition, we created two much simpler "null" models for baseline comparison: called **line length** and **line number**. For the line length model, relative time spent on each line is simply proportional to its length, so longer lines will receive more time. In the line number model, line time is inversely proportional to line number, so lines appearing earlier in the program will receive more time. Figure 93 provides an example of each null model using the `between functions` program.



```
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
```

Figure 93: *Examples of relative line times generated from the two null models: line length (left) and line number (right).*

The complete correlation matrix for all four versions of Mr. Bits and the two null models is shown in Figure 94. For each program and each model, the Spearman correlation is computed using the average times spent on each line for the model and our human programmers. A higher, green value indicates a strong positive correlation while a higher, redder value means a strong *negative* correlation with the human data. Correlation values not meeting the $\alpha = 0.05$ significance criteria or below 20 (0.2) are not displayed (though their cells are still colored).

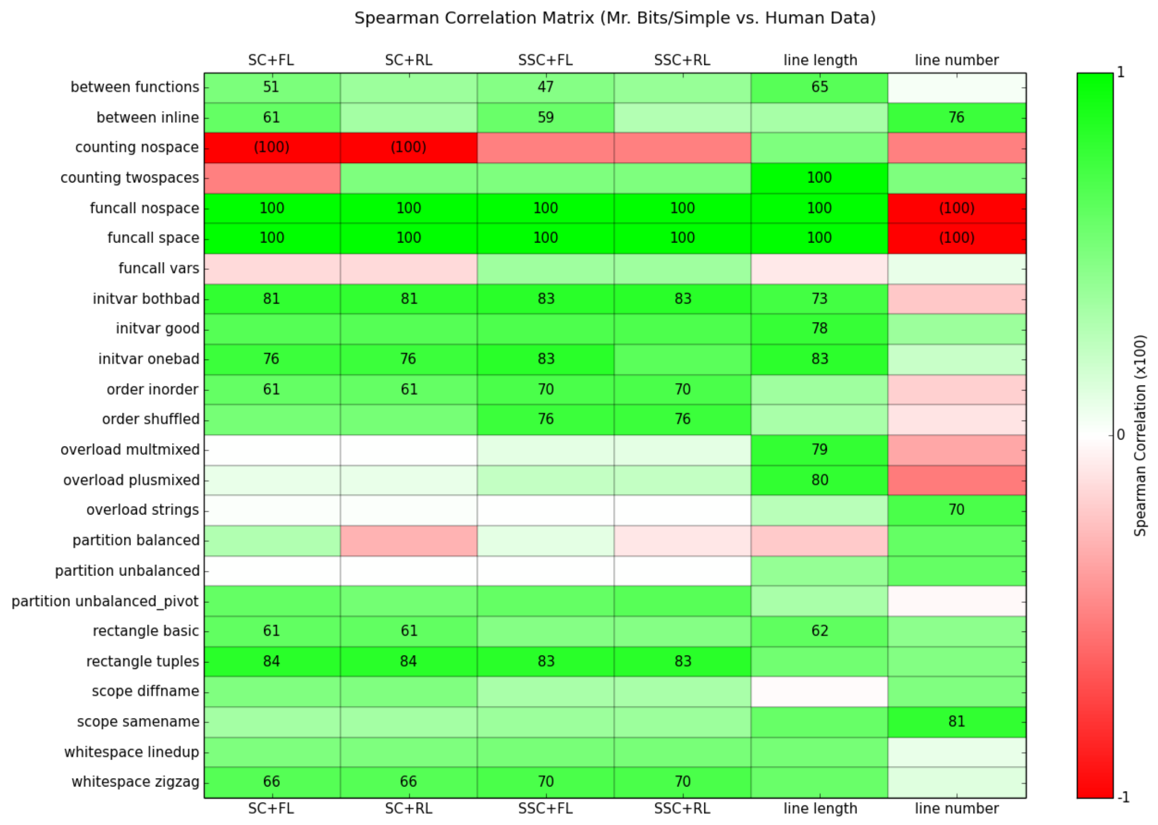


Figure 94: *Correlations between relative time spent on each line for human trials, 4 versions of Mr. Bits, and two simple models based on line length and number.*

If we take the sum of correlation values as our success criteria, then the line length and SSC+FL (subsymbolic computing + forget lists) models come out on top, followed somewhat closely by SC+FL, SSC+RL, and SC+RL [22]. This success criteria and the use of correlations here is not intended to provide a statistical argument for one model over another. Instead, we simply take these results as evidence for two things: (1) forgetting lists is a useful model setting, and (2) line length is a strong contributor to the average time the eyeCode participants spent on each line. While it is common to accept a simpler model in favor of a more complex one (e.g., line length vs. any Mr. Bits version), our goal with Mr. Bits is not *just* to achieve a good fit with the human data. We wish for our model to also provide an **explanation** of the programmer's underlying cognitive processes. The time spent by Mr. Bits on each line is not just a function of its character length, but

[22]The line number model trails *far* behind the others, so we do not consider it further.

also depends on when and how frequently the line is viewed. When combined with the Nibbles model (see Future Work in Section 19.1), it will be possible for the model to spend less time on more *idiomatic* lines – i.e., those fitting a commonly-seen template.

17.5.2 Human Trial Times

We now compare the **total time** taken by Mr. Bits and humans to read and evaluate each program. This "trial time" is a useful summary metric if we assume that more cognitively complex programs should take longer to understand and predict their output. Intuitively, we might assume this is a function of program length, but with the added twist that commonly-used or repeated patterns will speed participants along. Unfortunately, *errors break these assumptions* because failing to properly evaluate portions of the program may result in a smaller trial time (or vice-versa, depending on the program). Therefore, we must be aware of the correctness of a human trial in addition to its length when comparing it to Mr. Bits.

The distributions of human trial times across 24 Python programs is shown in Figure 95. For each program, the individual trial times (bars) have been sorted from shortest to longest, and the trials with incorrect responses have been colored red. The (single) trial times for all four versions of Mr. Bits (with or without subsymbolic, remembering or forgetting lists), are shown as lines, plotted where they would fall in the sorting process. In some program sets, such `funcall`, there is no difference between the remember lists (RL) and forget lists (FL) versions of Mr. Bits because these programs do not contain lists.

With few exceptions, incorrect human trials do not appear strongly correlated with trial time. The two notable exceptions are `counting twospaces` (Appendix A.2.2) and `scope samename` (Appendix A.9.2). The former is easily explainable: the most common error (not including the second `print` statement inside the loop body) significantly reduces the amount of expected output, and therefore the number of response characters the participant must type. The `scope samename` error pattern is not as easy to explain, and may simply be a fluke. Regardless, it is visually apparent that Mr. Bits' trial times tend to fall in the middle of the human distributions. An interesting exception is `overload strings`, where Mr. Bits takes longer than any human participant when ACT-R's subsymbolic mode is engaged. This is likely due to ACT-R's manual

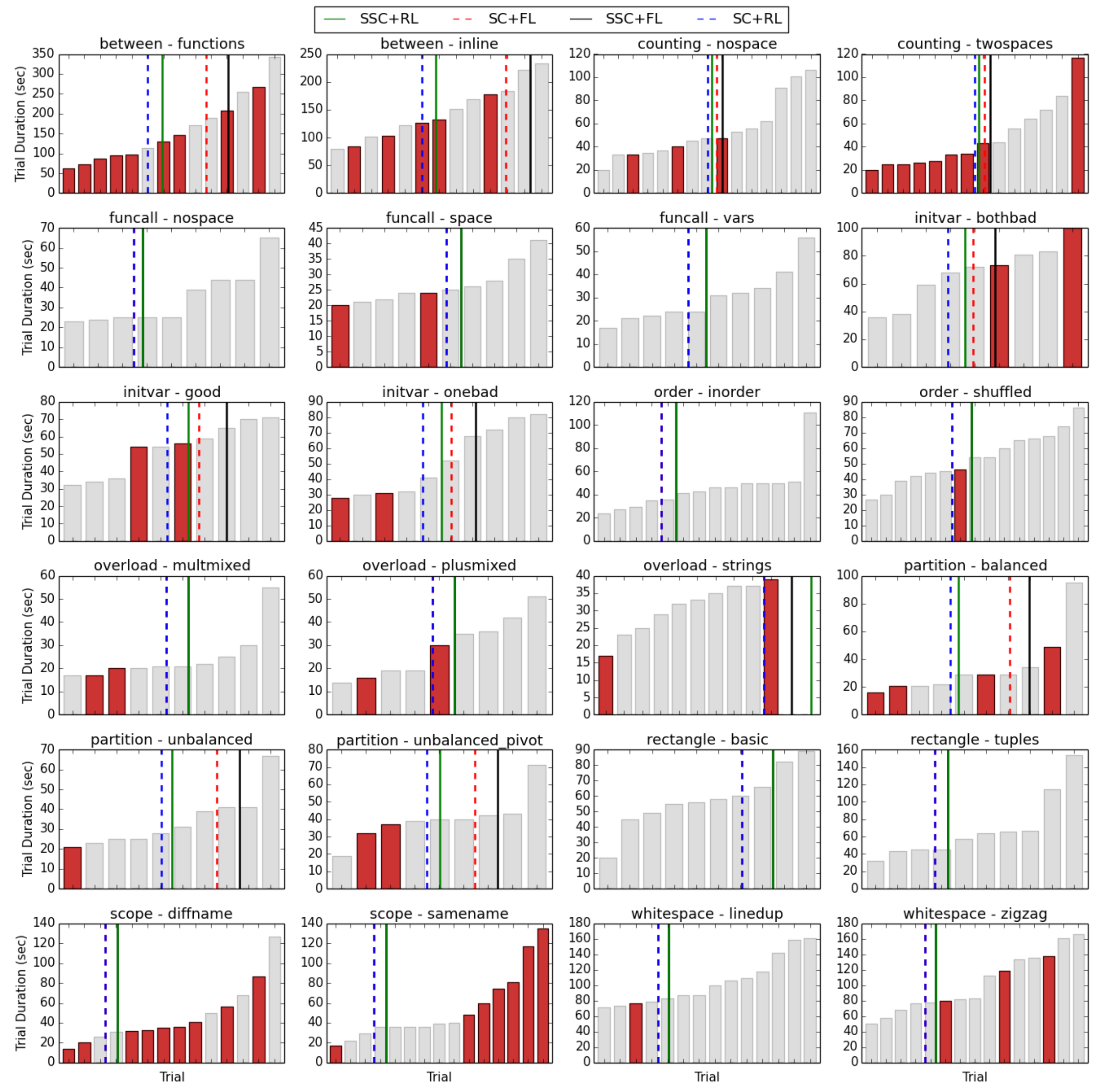


Figure 95: *Mr. Bits trial times (4 versions) compared to human trial time distributions for each program. Incorrect trials are colored red.*

(typing) module; its subsymbolic timings are subject to Fitts Law. Because this program involves typing English words like “bye” and “penny”, participants are likely to be much faster than Mr. Bits, for whom “nypen” is just as slow to type as “penny”.

17.5.3 Trial Time as a Complexity Metric

Because Mr. Bits’ trial times track human performance (with some caveats mentioned above), it may be useful as a complexity metric for ranking programs (or versions of a program) by how difficult they would be to understand. Figure 96 contains a Spearman correlation matrix for the trial times of all four Mr. Bits versions against four commonly-used complexity metrics: lines of code, Cyclomatic Complexity [59], Halstead Volume [45], and number of output characters. Again, non-significant ($\alpha = 0.05$) correlations and correlation values less than 20 (0.2) are not printed.

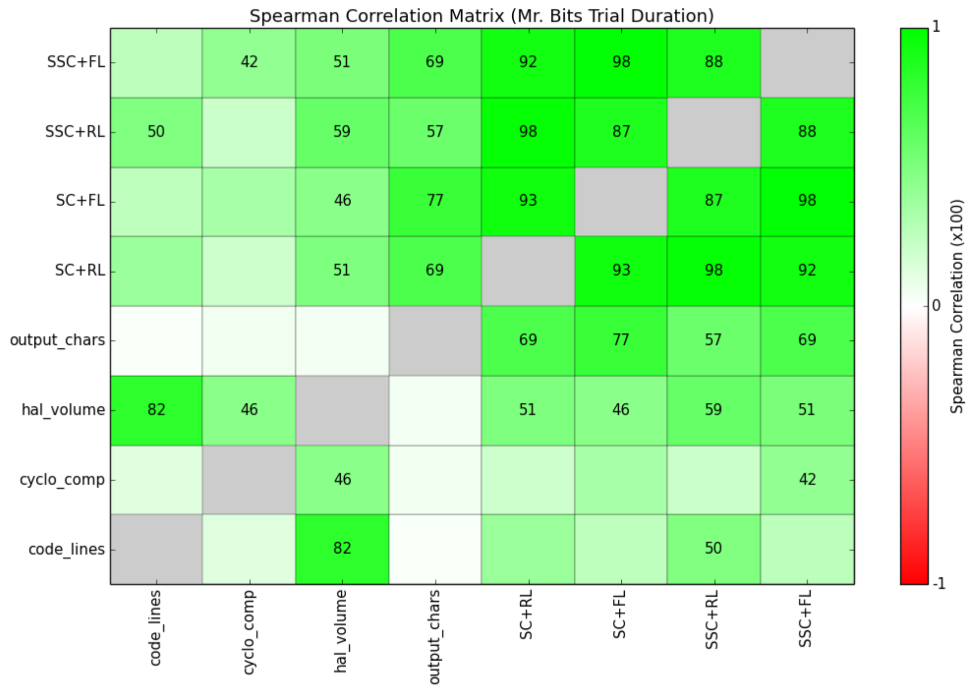


Figure 96: *Correlations between Mr. Bits trial times and code complexity metrics for all programs.*

While the correlation matrix cannot directly tell us whether or not Mr. Bits’ trial time (henceforth referred to as MBTT) is a useful complexity metric, we can at least see that it is not strongly correlated with any of the common source code metrics. In other words, MBTT is not simply restating the fact that a program has many lines or conditional statements (Cyclomatic Complexity).

Interestingly, all versions of MBTT are moderately correlated with Halstead Volume, which is derived from the ratios of unique and total operators and operands. More research is needed to untangle this relationship, but we expect the way Mr. Bits computes sums and products to play a major role (Section 17.4). Lastly, the moderate to strong correlations between MBTT and output chars is expected because Mr. Bits must type a response for each program. It is good news too that the MBTT/output chars correlations are not very strong or perfect. This would suggest that Mr. Bits' trial times are almost entirely driven by its typing speed! Overall, the correlation matrix provides evidence for the potential utility of MBTT as a code complexity metric. An obvious path for future work (Section 19.1) would be to extend Mr. Bits to work on a larger subset of Python and run the model on a large collection of open source programs (e.g., from GitHub or SourceForge).

17.5.4 Limitations and Threats to Validity

Mr. Bits has a number of limitations and specific design choices that may threaten its validity as a model of human program comprehension. First and foremost, the model **cannot make errors**. This is due to the fact that Mr. Bits does not parse code, and therefore cannot internalize an incorrect program. Our other model, Nibbles (Section 18), does just this, but does not predict fixations and keystroke timings like Mr. Bits does. Unlike a human too, Mr. Bits fixates each and every word when reading a line (the line is skipped if it can be recalled from memory). This is not how humans read natural language text [72], and it is clear from our experimental data that this is not how programmers read code either. A more comprehensive model combining Mr. Bits and Nibbles would be capable of inferring unobserved code words or tokens, and avoid the need to fixate them. Eye-tracking studies of programmers have shown that keywords are the least fixated tokens in a program, providing evidence that easily inferred tokens (e.g., the `in` keyword in `for x in y`) are often skipped [14].

Python programs with user-defined classes, generators, and other advanced language features are not currently supported by Mr. Bits. Additionally, the program must be script-like – i.e., compute and print values to the terminal. There are many other kinds of programs, such as those with a graphical interface, but Mr. Bits has been designed to work with a single task: output prediction. Additionally, Mr. Bits' code environment is quite bare-bones compared to modern integrated development environments (IDEs). Syntax highlighting alone could drastically change the model, and is currently being investigated in the context of Nibbles.

Finally, the LISP scripts generated by Mr. Bits will likely give ACT-R modelers pause. Unlike most ACT-R models, Mr. Bits scripts contain dozens of productions. Because each production in an ACT-R model is (in many ways) a free parameter, it is not possible to argue that their specific combination represents the best possible model when there are dozens of productions. While we cannot argue for our individual productions, we can still find value in the high-level behavior of Mr. Bits. ACT-R imposes constraints on how its modules interact, and how long particular actions take (e.g., shifting visual attention). Thus, Mr. Bits will exhibit different macro behaviors when particular programs emphasize different modules by, for example, having information physically more spread out or by including many unrelated calculations. In this way, Mr. Bits serves as a framework for describing human program comprehension and, most importantly, as a model whose high-level predictions can be **falsified by future experiments**.

18 Part 3: Nibbles

The Mr. Bits model builds on top of the ACT-R cognitive architecture to produce human-like fixation and keystroke timings while evaluating simple Python programs (Section 17). Mr. Bits does not, however, actually parse text into an internal representation and determine the evaluation order of program lines. For this purpose, we propose **Nibbles**, a model of program comprehension that makes use of Cognitive Domain Ontologies, or CDOs, to internally represent a program (Section 16.3). The process of searching for the first constraint compliant solution in the CDO's solution space allows Nibbles to handle missing text and local ambiguities. Future work will join the Nibbles and Mr. Bits models to produce an end-to-end model capable of transforming raw text into human-like fixations and keystroke timings.

18.1 Choice Point Uses

Formally, Nibbles represents a program using a Cognitive Domain Ontology, or CDO. A CDO is a tree with entities and relations, together describing a space of possible "solutions" – a specific instance of the CDO tree with all choices fixed. With a set of domain and/or situation-specific constraints, a constraint solver is used to prune the tree and enumerate one or more constraint-compliant solutions. The *choice point* relation is special, in that only one of its child entities may be present in a given solution. Choice points are the source of generativity in a CDO, and Nibbles uses them in four important ways: (1) to classify entities, (2) to determine group membership, (3) to control the size of the solution space, (4) to express a preference or expectation. We expand on each of these uses of choice points below.

(1) Choice points can be used to **classify** or categorize a particular entity. For example, the Line Type choice point in the Nibbles CDO (Figure 99) classifies a Line entity as either a function call, variable assignment, etc. When searching for solutions, each of these choices will be enumerated and checked against the constraints. Every constraint-compliant solution will contain a single choice for every active choice point.

(2) Lines are grouped by Nibbles using spatial and semantic cues. Relative indentation and line type (e.g., a `for` loop), will determine if the current and nearby lines form a cohesive group (e.g., a `for` loop plus its body). The Group Role choice point can either be Active or Inactive for every line

in every line group. Each Line Group contains all lines in the program, initially with an indeterminate Group Role. When Active, Nibbles will enforce additional constraints to ensure that a line is only active in one group, and that the given line type fits within the group. This is not a classification of lines, but is instead an indication of a line's **membership in a group**.

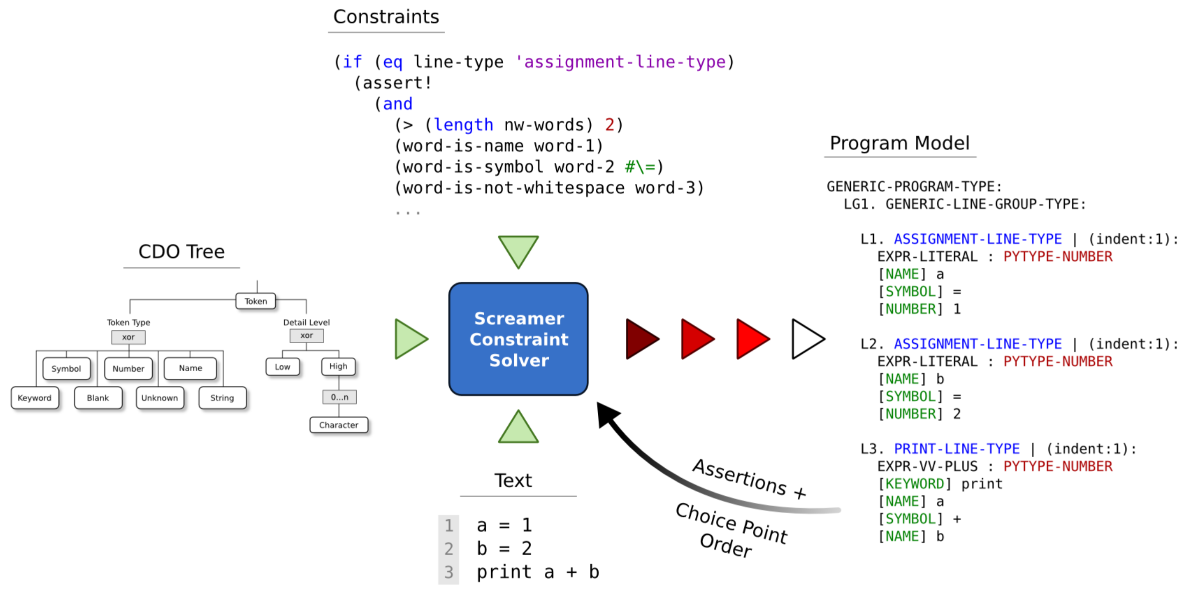
(3) In addition to classification and group membership, Nibbles uses choice points to **limit the size of the CDO solution space**. At every level of the CDO tree, a Detail Level choice point controls whether instances at that level will be High or Low detail. If High detail is set for every Detail Level, the complete solution space is available – down to the character level of every word in the program. Without high performance computing resources, however, it is practically infeasible to enumerate all solutions for programs with only 5 lines or less. Nibbles is able to explore much larger programs by carefully controlling the detail level such that only the currently attended line is set to High detail. Using a form of *attention*, then, Nibbles is able to tame large solution spaces without being exposed to the full brunt of the combinatorics.

(4) Lastly, we use choice points in Nibbles to **express a preference or expectation**. The `scope diffname` example in Section 18.5.3 extends Nibbles' CDO with a Transaction Status choice point at the Line level (Figure 100). The first choice, Complete, enables a set of constraints on the line's tokens. If the tokens survive these constraints, the line will be interpreted as a Complete transaction. If not, however, the line will be marked as an Incomplete transaction, and the current CDO tree will *still be a candidate solution*. This use of choice points allows for "softer" constraints that do not immediately discount a given solution.

18.2 CSP and Text Interpretation

Nibbles parses and interprets text using three components: (1) a CDO tree to describe the structure of the domain, (2) a collection of constraints to exclude non-sensical solutions, and (3) a constraint solver to enumerate constraint compliant solutions from the domain (Screamer [88]). Each line of code is provided to Nibbles with high fidelity, as if it were placed under a virtual fovea.

Figure 97 depicts the transformation of a simple three line program (bottom) into a "mental model" representation in Nibbles (right). Using its existing knowledge of Python (constraints), Nibbles abduces that the three lines form a single group, comprised of two assignment statements and one print statement. Furthermore, the types of variables *a* and *b* are inferred as well as the sense of the overloaded plus (+) operator (numeric sum versus string append).



```
(if (eq line-type 'assignment-line-type)
  (assert!
    (and
      (> (length nw-words) 2)
      (word-is-name word-1)
      (word-is-symbol word-2 #\=)
      (word-is-not-whitespace word-3)
      ...
```

```
a = 1
b = 2
print a + b
```

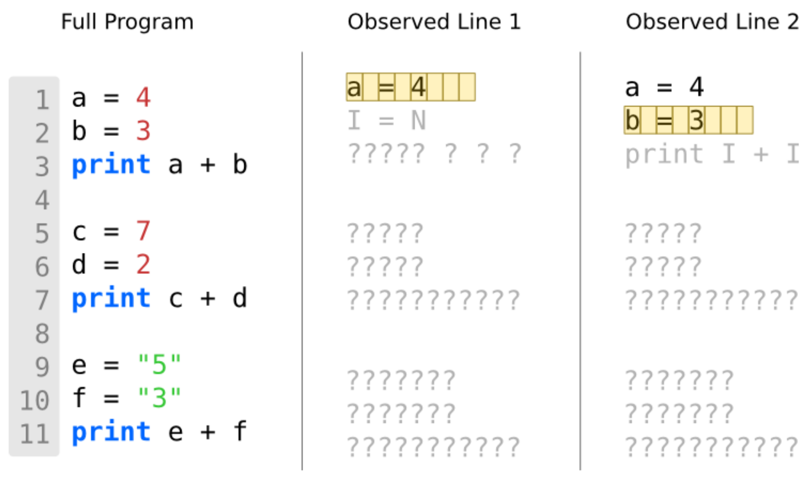
```
GENERIC-PROGRAM-TYPE:
  LG1. GENERIC-LINE-GROUP-TYPE:

    L1. ASSIGNMENT-LINE-TYPE | (indent:1):
      EXPR-LITERAL : PYTYPE-NUMBER
      [NAME] a
      [SYMBOL] =
      [NUMBER] 1

    L2. ASSIGNMENT-LINE-TYPE | (indent:1):
      EXPR-LITERAL : PYTYPE-NUMBER
      [NAME] b
      [SYMBOL] =
      [NUMBER] 2

    L3. PRINT-LINE-TYPE | (indent:1):
      EXPR-VV-PLUS : PYTYPE-NUMBER
      [KEYWORD] print
      [NAME] a
      [SYMBOL] +
      [NAME] b
```

Figure 97: *Information flow for Nibbles model. The interpretation of each line involves a trip to the constraint solver and a subsequent re-ordering of choice points.*

Nibbles reads programs incrementally line-by-line by focusing a character-aligned sensor (fovea) on one line at a time. The observed characters are asserted into the CDO along with the following information:

1. The sizes of unobserved tokens on the current line and the *lines below and above*.
2. The indent level of the line below (relative to the current line).
3. The lengths of unobserved lines.
4. The presence of text above and below the current line.

Figure 98 shows the first few steps of Nibbles reading the `overload plusmixed` program (Appendix A.6.2). The complete program is on the left (highlighted for the reader's benefit). On the immediate right, unobserved characters are replaced with "?". Spaces between blocks of "?" represent the availability of token sizes, while entire lines of "?" reflect knowledge of line lengths. Unobserved code is represented in the Nibbles CDO by setting the line group, line, or token type to Unknown. Additionally, Low Detail choice points are used to dynamically control the size of the



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

c = 7
d = 2
print c + d

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

```
a = 4
I = N
????? ? ? ?

?????
?????
???????????

???????
???????
???????????
```

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

?????
?????
???????????

???????
???????
???????????
```

Figure 98: *Incremental reading of a program. The first two lines of the program are observed, and the third line is inferred. An* `I` *stands for an identifier, and an* `N` *stands for a number.*

CDO solution space. Summary information, such as the size of a line or token, is stored in an unobserved Low Detail choice. This will stop instance sets (of tokens, characters, etc.) under the High Detail choice from being enumerated by the constraint solver, exponentially decreasing the size of the entire space. This serves the dual purpose of speeding up the model and representing a form of **attention**. With every High Detail choice set, exploring the space of even a five line program can be computationally infeasible. See Section 18.1 for more details.

18.3 The Nibbles CDO

The structure of Nibbles' CDO is shown in Figure 99. In it, a program is broken down in the following components:

- **Line Groups** - Contiguous groups of related code lines that may or may not all be at the same indent level.
- **Lines** - A single code line in a line group.
- **Tokens** - A single Python token/word in a line (usually separated by whitespace).
- **Characters** - A single character in a token.

In Nibbles, a program consists of one or more line groups, which are made of one or more lines, etc. As previously described, nested instance sets at each level of the CDO hierarchy fall under a High Detail choice. By toggling the detail level choice points, Nibbles can dynamically contract and expand the solution space. A particular solution for the Nibbles CDO will contain High Detail information for currently observed code, and Low Detail (or Unknown) information for previously/unobserved code.

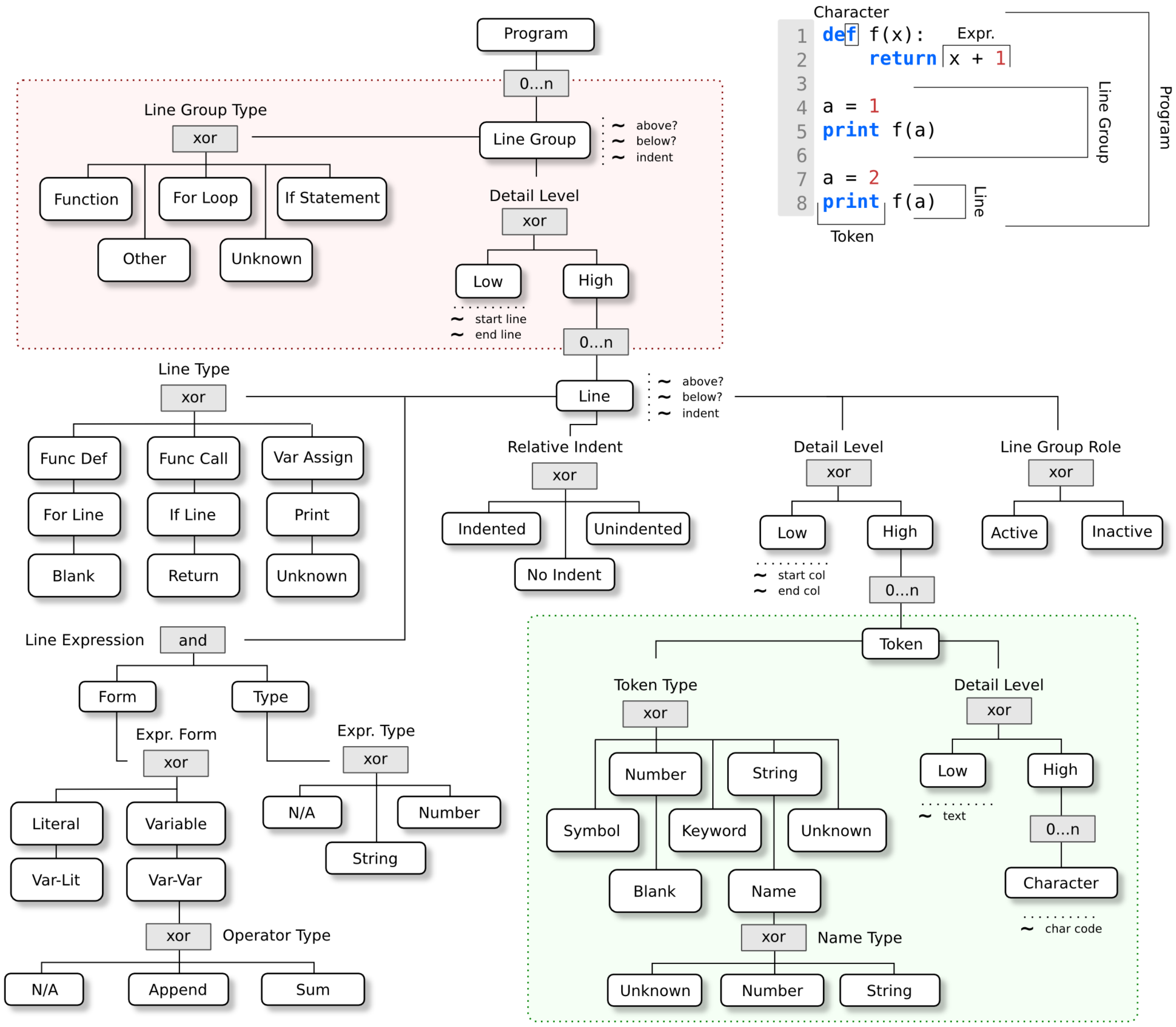

```
def f(x):
    return x + 1

a = 1
print f(a)

a = 2
print f(a)
```

Figure 99: *Structure of the Nibbles CDO.*

Line groups, lines, and tokens are categorized using the Line Group Type, Line Type, and Token Type choice points. Table 18 contains an exhaustive list of each category, and a brief description of how the category is verified by Nibbles' constraints. Lines that contain an *expression*, such as the right-hand side of a variable assignment, make use of the additional Line Expression structure to determine the expression's form and Python type. Table 19 lists the currently supported forms/types, which focus just on single token and simple binary expressions (e.g., `x + 1`). Future work will expand this into a more comprehensive set.

Category	Name	Description
Line Group	Other/Generic	Default line group type (no other type matches).
	Function	A function definition. 1st line must a Function Def line, 2nd line must be indented.
	For Loop	A `for` loop block. 1st line must be a For line, 2nd line must be indented.
	If Statement	An `for` loop block. 1st line must be a For line, 2nd line must be indented.
Line	Function Def	Starts with a `def` Keyword token, followed by a Name and (optional) arguments.
	Function Call	Starts with a Name token, followed by (optional) arguments.
	Variable Assignment	Starts with a Name token, followed by an = Symbol and an expression.
	For Line	Starts with a `for` Keyword token, followed by a Name, `in` Keyword, and either a Name or list literal.
	If Line	Starts with an `if` Keyword token, followed by an expression.
	Print	Starts with a `print` Keyword token, followed by an expression.
	Return	Starts with a `return` Keyword token, followed by an expression.
	Blank	Line contains only Whitespace tokens.
Token	Symbol	A non-alphanumeric token – `=`, `(`, etc.
	Number	A single string of numeric characters – e.g., `123`.
	Name	A single string of alphabetic characters and underscores – e.g., `the_list`.
	Keyword	One of `def`, `for`, `if`, `print`, `in`, `return`.
	String/Text	Any string of characters that are not blank and not one of the other types.
	Whitespace/Blank	A token that is entirely whitespace or of zero length.

Table 18: *Types of tokens, lines, expressions, and line groups in Nibbles.*

18.4 Programming Knowledge as Constraints

A CDO captures the structure of a domain as well as its constraints. Nibbles encodes knowledge of Python into both. The textual structure of a program is represented hierarchically in Nibbles' CDO, as groups of lines, tokens, and characters. Constraints at every level of the hierarchy enforce Python's grammatical rules – e.g., the line after a function `def` must be indented. Table 20 provides

Form	Type	Operator	Description
Literal	Number		Basic numeric literal (`123`)
	String		Empty or non-empty string literal (`""` or `"hello"`)
Variable	Number		Variable name only (`x`).
	String		Variable name only (`x`).
Variable-Literal	Number	Sum	Variable name and numeric literal (`x + 1`).
	String	Append	Variable name and string literal (`x + "hello"`).
Variable-Variable	Number	Sum	Variable names only (`x + y`).
	String	Append	Variable names only (`x + y`).

Table 19: *Expression forms in Nibbles. Used in Print statements, Variable Assignments, etc.*

high-level descriptions of Nibbles' constraints on Line Groups, Lines, and Tokens.

Category	Name	Description
Line Group	line-group-indent	If lines A and B in same group, then B is either in-line with A or indented.
	all-lines-active	Every line must be active in exactly 1 group, in line order.
	lines-to-line-group	Verifies a line group type based on 1st and 2nd active lines.
Line	tokens-to-line	Verifies a line type based on the first few tokens. The Variable Assignment, Print, If Line, and Return line types have their expression forms checked as well.
	whitespace-at-end	If a line contains blank and non-blank tokens, all Whitespace/Blank tokens must be at the end of the line.
Token	chars-to-tokens	Verifies a token type based on its characters. Anything not entirely whitespace and not any other token type is considered Text.

Table 20: *Constraints in the Nibbles model.*

In the Screamer constraint solver, constraints are really just Common LISP functions that inspect proposed sections of the CDO tree and produce failures when specific checks are violated. Because of this, it can be difficult to describe precisely what constraints are doing without using the code itself. While this assumes a familiarity with Common LISP, the Screamer (and Screamer+) constraint language, and the codified Nibbles CDO structure, it is still possible to meaningfully understand the constraints without them. Listing 3, for example, shows a snippet of the constraint the verifies tokens based on characters. In this particular snippet, a `print` keyword has been proposed, and is verified by a set of assertions. Even without the necessary background, this constraint snippet is fairly readable – i.e., it is clear precisely *how* specific characters are being checked.

```
(cond
  ((eq token-type 'keyword-type)
   (assert!
    (or
     ;; print keyword
     (and
      (equalv keyword-kw 'print)
      (equalv (nth 0 char-values) #\p)
      (equalv (nth 1 char-values) #\r)
      (equalv (nth 2 char-values) #\i)
      (equalv (nth 3 char-values) #\n)
      (equalv (nth 4 char-values) #\t)
      (equalv (v@ (token) length) 5)
      (rest-chars-are-whitespace char-values 5))
    ;; ...
))))
```

Listing 3: *Snippet from the "chars-to-token" constraint that verifies a `print` keyword.*

A complete list of constraints at all levels of the CDO tree for Nibbles is available in Appendix E. The constraints are described in English rather than LISP to avoid unnecessary details of Screamer. Note that these constraints represent a small **subset** of Python, not the complete language!

18.5 Results and Discussion

Below we focus on three concrete example programs from the eyeCode experiment [46]. These programs were most difficult for our human participants, resulting in the highest number of errors (incorrect output predictions). We use Nibbles to model the errors observed in humans by generating incorrect mental models for each program. Under these conditions, Nibbles represents either a less experienced programmer or an experienced programmer with invalid assumptions. We then show how, with the re-ordering of specific choice points or the strengthening of particular constraints, these errors can be eliminated.

18.5.1 counting - twospaces

The code for our first example is given in Listing 4. When asked what it would print, about half our participants incorrectly assumed that the final `print` line would only execute once. Because indentation is meaningful in Python, the last line must be part of the `for` loop (despite the misleading text "Done counting").

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

Listing 4: *Code from the `counting - twospaces` program in the eyeCode experiment.*

If the two blank lines (lines 3 and 4) are removed, however, significantly fewer errors of this type are made (Section 5.2.2). This suggests that our participants understood Python's semantic indentation, but may not have been gathering enough evidence before settling on an interpretation of the code. One possible mechanism for this error is if **vertical whitespace primes disjoint grouping** of lines – i.e., a participant is more likely to break code lines into different groups when they are separated by enough whitespace. If a line's indentation is not explicitly (visually) determined, we hypothesize that it will be inferred from the grouping.

Consider the (abbreviated) "mental model" representation given by Nibbles in Listing 5 for `counting twospaces`. At this point, Nibbles is focused on the first line (indicated with a `[*]`). We can see that all three lines (L1, L2, L3) are contained in a single line group (LG1). Because Nibbles is focused on the first line, some details of the second line are available: specifically, token sizes and relative indentation. The > after PRINT-LINE-TYPE indicates that line 2 is indented relative to the line above it. We assume this information is (visually) available to Nibbles because of the proximity of lines 1 and 2. The relative indentation of line 3, however, is **not visually available**, and its value (| - meaning same indent previous line) is inferred *based on the assumption that it is part of line group 1*. While this is the correct interpretation, it is not based any visual evidence.

Listing 6 shows the state of Nibbles' mental model after lines 1 and 2 have been observed. The details of line 3 are now available, and there is enough information to start predicting the

```
LG1. FOR-LOOP-LINE-GROUP-TYPE:
  L1. FOR-LOOP-LINE-TYPE | (indent:1) [*]:
    [KEYWORD-TYPE:3] for
    [NAME-WORD-TYPE:1] i
    [KEYWORD-TYPE:2] in
    ...
    [SYMBOL-WORD-TYPE:1] :

  L2. PRINT-LINE-TYPE > (indent:2):
    EXPR-LITERAL : PYTYPE-NUMBER
    [KEYWORD-TYPE:5] PRINT
    ...
    [NUMBER-WORD-TYPE:1] ?

  L3. UNKNOWN-LINE-TYPE | (indent:2):
```

Listing 5: *Correct mental model for `counting - twospaces`, focus on line 1.*

program's output. Based on this mental model, the **correct output** will be predicted. The line grouping constraint `line-group-indent` is responsible for line 3's relative indentation. If it is part of line group 1, it cannot be indented or unindented relative to line 2 because the first line of the group does not begin a new function or loop block. Nibbles assumes line 3 is part of line group 1, therefore it is inline (|) with line 2.

If we include the assumption that at *least 2 blank lines indicates a likely break in the current line group*, Nibbles will produce the same error observed in our human data. By re-ordering the Line Group Role choice point for lines after a block of whitespace, Nibbles will be primed for solutions that break the code into multiple line groups. Listing 7 shows the incorrect mental model produced by this assumption. As before, lines 1 and 2 are observed and internalized. Line 3, however, is broken out into a separate line group (LG2), and its relative indentation is **unindented relative to line 2**. Again, the inferred indentation for line 3 is based on the (now incorrect) grouping assumption. Were Nibbles to explicitly assert the indentation level for line 3, presumably based on extra visual information, this solution would be discarded and the model in Listing 6 would prevail.

Predictions. Based on results from the eyeCode experiment, we know that moving the final `print` up two lines will significantly reduce errors. We hypothesize that participants who produce the grouping error will also incorrectly believe that the final line is unindented relative to the

```
LG1. FOR-LOOP-LINE-GROUP-TYPE:
  L1. FOR-LOOP-LINE-TYPE | (indent:1):
    [KEYWORD-TYPE:3] for
    [NAME-WORD-TYPE:1] i
    [KEYWORD-TYPE:2] in
    ...
    [SYMBOL-WORD-TYPE:1] :

  L2. PRINT-LINE-TYPE > (indent:2):
    EXPR-LITERAL : PYTYPE-STRING
    [KEYWORD-TYPE:5] PRINT
    ...
    [NAME-WORD-TYPE:1] i

  L3. PRINT-LINE-TYPE | (indent:2) [*]:
    EXPR-LITERAL : PYTYPE-STRING
    [KEYWORD-TYPE:5] print
    [SYMBOL-WORD-TYPE:1] "
    [TEXT-WORD-TYPE:4] Done
    [TEXT-WORD-TYPE:8] counting
    [SYMBOL-WORD-TYPE:1] "
```

Listing 6: *Correct mental model for `counting` - `twospaces`, focus on line 3.*

second line. Additionally, we predict that calling attention to the relative indentation will reduce error rates. For example, adding an `end` token after the final `print` statement or nesting the whole program in another `for` loop may force participants to visually measure the indent level.

Open Questions. Our Nibbles model of the grouping error does not depend on the words "Done counting" in the final line. We suspect that changing these words to something like "Home mounting" will reduce errors slightly (perhaps almost entirely if the blank lines are also removed). A straightforward way of incorporating lexical information into Nibbles is by having words influence the re-ordering of choice points. In this specific instance, "Done counting" could play the same role as whitespace by priming Nibbles to break the current line group. Errors from the `counting nospace` program of the eyeCode experiment (no extraneous whitespace), however, were significantly fewer than the `twospaces` program above. Therefore, some extension to Nibbles' CDO will be required to incorporate realistic text priming.

```
LG1. FOR-LOOP-LINE-GROUP-TYPE:
  L1. FOR-LOOP-LINE-TYPE | (indent:1):
    [KEYWORD-TYPE:3] for
    [NAME-WORD-TYPE:1] i
    [KEYWORD-TYPE:2] in
    ...
    [SYMBOL-WORD-TYPE:1] :

  L2. PRINT-LINE-TYPE > (indent:2):
    EXPR-LITERAL : PYTYPE-STRING
    [KEYWORD-TYPE:5] PRINT
    ...
    [NAME-WORD-TYPE:1] i

LG2. GENERIC-LINE-GROUP-TYPE:
  L3. PRINT-LINE-TYPE < (indent:1) [*]:
    EXPR-LITERAL : PYTYPE-STRING
    [KEYWORD-TYPE:5] print
    [SYMBOL-WORD-TYPE:1] "
    [TEXT-WORD-TYPE:4] Done
    [TEXT-WORD-TYPE:8] counting
    [SYMBOL-WORD-TYPE:1] "
```

Listing 7: *Incorrect mental model for `counting` - `twospaces`, focus on line 3.*

18.5.2 overload - plusmixed

Our next example is given in Listing 8. The `overload` `plusmixed` program performs two summations and one string append at the end. Because Python overloads the + operator, it's possible mistakenly think the final `print` will output 8 instead of "53". A handful of participants made this error (approximately 11%), but we also saw evidence for a slower response time relative to the `multmixed` version where the first two `print` statements used `*` instead of + (Section 5.2.6). How would these two observations be accounted for in the Nibbles model?

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

c = 7
d = 2
print c + d

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

Listing 8: *Code from the* `overload - plusmixed` *program in the eyeCode experiment.*

Modeling the slower response time with Nibbles is fairly straightforward if we assume that the numeric sense of the + operator has been primed by the time the final `print` statement is interpreted. Without additional constraints, and with the Operator Type choice point re-ordered to have Sum first, Nibbles will produce the mental model in Listing 9. Given the assumption that + means "sum", Nibbles infers that *e* and *f* are both numbers.

```
L1. PRINT-LINE-TYPE | (indent:1):
  EXPR-VV-PLUS : PYTYPE-NUMBER (PLUS-SUM)
  [KEYWORD-TYPE:5] print
  [NAME-WORD-TYPE:1] e      (PYTYPE-NUMBER)
  [SYMBOL-WORD-TYPE:1] +
  [NAME-WORD-TYPE:1] f      (PYTYPE-NUMBER)
```

Listing 9: *Incorrect mental model for final line of* `overload - plusmixed`

But after reading the previous two lines (`e = "5"` and `f = "3"`), the fact that *e* and *f* are strings should be known. How can Nibbles represent this knowledge and find the correct mental model, but still be slower than if *e* and *f* were numbers? Adding the following constraint to the **top level (Program)** will produce the desired behavior:

> If a Name token has text "e" or "f", then its type must be String

Now, Nibbles will generate the correct mental model shown in Listing 10. Although the correct solution is the first one found, the search process will take more time. By constraining *e* and *f* to strings at the Program level, as opposed to the Line Group, Line, or Token level, Nibbles will *still generate the previous (incorrect) mental model, but it will be ultimately discarded.* Were the constraint to be pushed further down into the CDO tree, Nibbles would produce the correct solution in fewer steps.

```
L1. PRINT-LINE-TYPE | (indent:1):
  EXPR-VV-PLUS : PYTYPE-STRING (PLUS-APPEND)
  [KEYWORD-TYPE:5] print
  [NAME-WORD-TYPE:1] e     (PYTYPE-STRING)
  [SYMBOL-WORD-TYPE:1] +
  [NAME-WORD-TYPE:1] f     (PYTYPE-STRING)
```

Listing 10: *Correct mental model for final line of `overload - plusmixed`*

With the type constraint on e and f, Nibbles will always produce the correct mental model. How can we account for the observed human error in which the predicted output is 8? Clearly, the 5 and 3 values must have been *read*, but they were either mistakenly interpreted as numbers or those participants believed that Python's + operator would *convert the strings to numbers*. The latter is actually the correct behavior in some languages, such as PHP [94], in which strings are sometimes automatically coerced into integers. Modeling multi-language knowledge is beyond the scope of this work, but it is possible to model the misinterpretation of a string as a number.
The `chars-to-token` and `rhs-expression-pytype` constraints (Appendix E) work together to distinguish `5` (a number) from `"5"` (a string). Nibbles sees the latter as three separated tokens: `"`, `5`, and `"`. At the **Token level**, Nibbles will interpret the `5` in `"5"` as *both* a Number and a Text token. At the **Line level**, the following condition from `rhs-expression-pytype` will discard the numeric interpretation of `"5"`:

If RHS is a Number, it has exactly 1 Number token

Because of the two " Symbol tokens, this constraint will ensure that `"5"` is not a number. We could model the misinterpretation by withholding the " Symbols at first, perhaps based on some notion that digits are more salient than quotes (and therefore visually processed first). Another, simpler option is to slightly tweak the constraint as follows:

If RHS is a Number, it has **at least** 1 Number token

With this updated constraint, Nibbles is now capable of misinterpreting `"5"` as a number *depending on the ordering of other choice points*. If the Number choice is considered first for Token Type, then `rhs-expression-pytype` will happily declare that the expression `"` (Symbol), `5` (Number), `"` (Symbol) is numeric – and therefore, *e* and *f* are numbers. Additional experiments with programmers are needed to disambiguate the cause(s) of the error, and guide the adjustment of Nibbles' constraints in order to more closely match human behavior.

Predictions. Based on the eyeCode experiment results and our Nibbles model, we would expect fewer errors to be made on `overload plusmixed` when the "string append" sense of the `+` operator has been sufficiently primed. We did not observe a statistical difference between the `plusmixed` and `strings` versions of the program in this regard, but the small program lengths and unconstrained reading order make it difficult to pinpoint priming effects. A more controlled, focused experiment would provide more evidence one way or the other. Additionally, we should expect a reduction in errors (or a boost in speed) when either *e* and *f* have no digits (e.g., `"five"` and `"three"` instead of `"5"` and `"3"`) or have been made visually distinct with syntax highlighting.

Open Questions. How can Nibbles be used to inform a discussion on the cognitive complexity of operator overloading? The differences between a Nibbles CDO with and without operator ambiguity can be quantified in a number of potentially useful ways: the sizes of their solution spaces, the number and complexity of their constraints, and by the kinds of invalid solutions that can be generated locally. Without a token for designating a variable's type, such as `string e = "5"`, type inference in the programmer's head hinges entirely on the two quotes. What is the trade-off in adding additional syntax to the language to designate variable types, perhaps via gradual typing [100]? The parsing and compilation costs are not necessarily the same as the cognitive costs, and future work with Nibbles could help predict these costs by modeling the kinds of errors humans make. While this cannot replace experiments with real programmers, it is more objective (and falsifiable) than opinion in the absence of human data.

18.5.3 scope - diffname

The code for our final example is given in Listing 11. Along with `counting twospaces` and its `samename` version, this program was responsible for many of the errors produced in the eyeCode experiment (Section 5.2.9). The program outputs `4`, rather than the expected value of `22`, due to the pass-by-value semantics of Python: given that $added = 4$, the expression `add_1(added)` is equivalent to `add_1(4)`. Many participants, however, were convinced that the `add_1` and `twice` functions must *do something*, and therefore had to modify the contents of `added` (some participants even questioned Python's pass-by-value semantics!). Soloway identified the maxim "Don't include code that won't be used" as an implicit **rule of discourse** for programming [89], and its violation appears to be very potent.

```python
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
```

Listing 11: *Code from the `scope - diffname` program in the eyeCode experiment.*

Nibbles can model the expectation that code should "do something" by extending the CDO and embellishing the existing constraints. For guidance, we turn to Mayer's notion of a **transaction** [58]. Rist describes a Mayer transaction as follows:

> "...The creation of just a single line of a code requires a great deal of reasoning and planning. Mayer (1987) described the conceptual structure underlying a line of BASIC code and found that the smallest piece of knowledge used in program understanding is a transaction. A transaction can be described using three categories: what operation takes place (action), where it takes place (location) and what object is acted upon (object)." [73]

For Nibbles, we will consider a transaction to have **actions**, **inputs**, and **outputs**. This is slightly more general than Mayer's notion, and maps more readily to the lines of a high-level language like Python. For example, the statement `x = y + 1` has two actions (`+`, `=`), two inputs (`y`, `1`), and one output (`x`). We suggest that a line of code can be said to "do something" if it contains a **complete** transaction – that is, with at least one token for an action, input, and output. With this definition, `x = 5` is a complete transaction, while `x == 5` is not. A function call like `f(x)` could be a complete transaction if we consider `f` to be the action, and `x` to be *both the input and the output*. If `x` is a Python list, for instance, it is reasonable for `f` to modify `x` by appending/delete items.

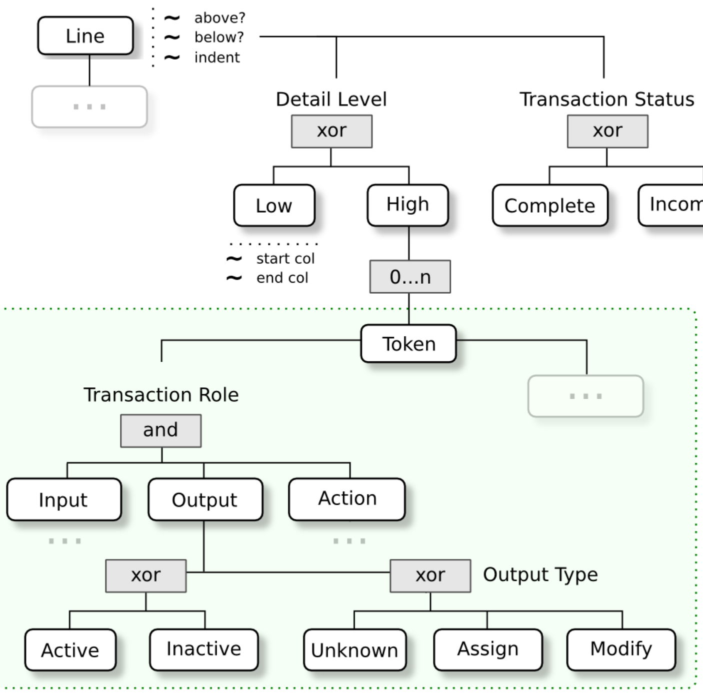


Figure 100: *Extended portion of Nibbles CDO with Mayer-like transactions.*

Figure 100 contains just the extended portion of Nibbles CDO (see Figure 99 for the rest). At the Token level, we add a Transaction Role sub-parts with an Active/Inactive choice point for each of the transaction roles described above. A Token, therefore, can be any combination of Input, Output, and Action. Output tokens have an additional choice point, called Output Type, which differentiates between Assign-ing a Name some new value and Modify-ing the value associated with a Name. A handful of constraints combine to assign appropriate transaction roles to our Tokens:

A Symbol is an Action if and only if it is + or =

A Number or Text token is always an Input

At the Line level, the Transaction Status choice point simply marks whether the line contains a complete or incomplete transaction. The following additional constraints enforce our notion of a complete transaction, and help to assign transaction roles:

A transaction is **complete** if and only if the line has 1+ Action, 1+ Input, and 1+ Output
Any token in a right-hand side expression cannot be an Output
The 1st token of an Assignment Line cannot be an Input or Action
The 1st token of a Function Call Line cannot be an Input or Output

Only the 1st token of an Assignment Line can have Output Type Assign

Example of a Valid Transaction. The Nibbles mental model in Listing 12 is derived from the assignment statement `x = y + 1` with the constraints above applied. At the Line level, a *complete transaction* is marked with a `TX`. For each Token, the transaction role(s) are indicated with an `A` (action), `O` (output), and/or `I` (input) on the right side of each printed line. The Output Type is given as either `O=` (assign) or `O!` (modify). In this example (a complete transaction), `x` is marked as the output, `y` and `1` as inputs, and both + and = as actions.

```
LG1. GENERIC-LINE-GROUP-TYPE:
  L1. ASSIGNMENT-LINE-TYPE | TX (indent:1):
    x : PYTYPE-NUMBER
    EXPR-VL-PLUS : PYTYPE-NUMBER
    [NAME-WORD-TYPE:1] x (PYTYPE-NUMBER) O=
    [SYMBOL-WORD-TYPE:1] = A
    [NAME-WORD-TYPE:1] y (PYTYPE-NUMBER) I
    [SYMBOL-WORD-TYPE:1] + A
    [NUMBER-WORD-TYPE:1] 1 I
```

Listing 12: *Nibbles mental model for the statement `x = y + 1` with transactions.*

Adding Targets. We now turn back to the Python code example in Listing 11. Focusing on lines 7 and 8, the generated Nibbles mental model is given in Listing 13. Both lines are considered complete transactions (marked with a `TX`). However, the `added` variable in line 8 is inferred as *both* *the input and output*! This makes sense, given that a Complete Transaction Status will be tried before an Incomplete at the Line level (mirroring the assumption that programmers expect a complete transaction). With no additional constraints, added is assumed to be modified during the function call despite the fact that it points to a constant (4).

```
LG1. GENERIC-LINE-GROUP-TYPE:
  L7. ASSIGNMENT-LINE-TYPE | TX (indent:1):
    added : PYTYPE-NUMBER
    EXPR-LITERAL : PYTYPE-NUMBER
    [NAME-WORD-TYPE:1] added (PYTYPE-NUMBER) O=
    [SYMBOL-WORD-TYPE:1] = A
    [NUMBER-WORD-TYPE:1] 4 I

  L8. FUN-CALL-LINE-TYPE | TX (indent:1):
    [NAME-WORD-TYPE:5] add_1 A
    [SYMBOL-WORD-TYPE:1] (
    [NAME-WORD-TYPE:5] added (PYTYPE-NUMBER) I O!
    [SYMBOL-WORD-TYPE:1] )
```

Listing 13: *Nibbles mental model for 2 lines from* scope diffname.

We believe this is **one source of human error**: incorrectly assuming that the argument given to add_1 can be modified due to a strong preference for lines that form complete transactions. Nibbles makes this error because it does not carry forward information about the nature of object *targeted* by the variable added. If we simply replace added with its value (4), the error disappears because a Number token cannot be an Output (Listing 14). Although we have not confirmed this with experiments, we would not expect any competent Python programmer to conclude that add_1(4) will modify its argument.

```
LG1. GENERIC-LINE-GROUP-TYPE:
  L1. FUN-CALL-LINE-TYPE | (indent:1):
    [NAME-WORD-TYPE:5] add_1 A
    [SYMBOL-WORD-TYPE:1] (
    [NUMBER-WORD-TYPE:4] 4 I
    [SYMBOL-WORD-TYPE:1] )
```

Listing 14: *Nibbles mental model with the function call's argument name replaced by its value.*

By extending Nibbles' CDO once again, we can prevent an incorrect interpretation of line 8, and attempt to model those participants who did not make this particular error. Figure 101 shows the extended portion of the CDO (relative to Figure 99).

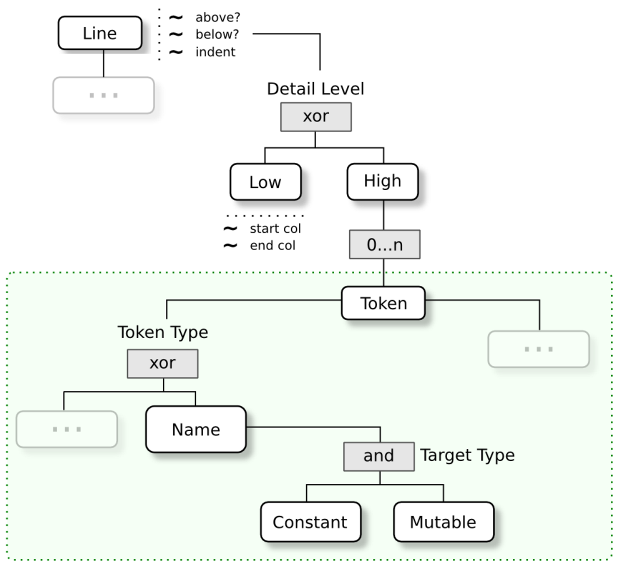


Figure 101: *Extended portion of Nibbles CDO with Name targets.*

We begin by extending Name tokens with the notion of a **target type**. Targets may be a Constant or a Mutable object, like a list or dictionary. Names that target Constants, such as numbers and strings, cannot modify their targets, and therefore should not have an Output Type of Modify. With enough restrictions on its Output Type, a Name token may be *dropped from consideration as a transaction output*. The assignment of a Name Target in Nibbles is done at the Line level via constraints on the Assignment line type. Given a particular right-hand side (RHS) expression, the left-hand side (LHS) variable's Target type can be inferred. The addition of two constraints, one at the Line level and one at the Token level, will help Nibbles avoid the input/output argument error on line 8:

If an Assignment Line RHS is a Number or Text string, the LHS target type is a Constant

A Name that targets a Constant cannot have a Modify Output Type

When interpreting lines 7 and 8 with the extensions above, Nibbles will no longer mistakenly assume that `added` could be modified by `add_1` (Listing 15). In fact, the `added` argument token is no longer considered a transaction Output. Note that this requires propagating the Target information for a Name from one line to the next, much like type information was propagated in Section 18.5.2. For brevity, we do not list these extra constraints.

```
LG1. GENERIC-LINE-GROUP-TYPE:
  L7. ASSIGNMENT-LINE-TYPE | TX (indent:1):
    added : PYTYPE-NUMBER
    EXPR-LITERAL : PYTYPE-NUMBER
    [NAME-WORD-TYPE:1] added (PYTYPE-NUMBER, TARGET-CONSTANT) O=
    [SYMBOL-WORD-TYPE:1] = A
    [NUMBER-WORD-TYPE:1] 4 I

  L8. FUN-CALL-LINE-TYPE | (indent:1):
    [NAME-WORD-TYPE:5] add_1 A
    [SYMBOL-WORD-TYPE:1] (
    [NAME-WORD-TYPE:5] added (PYTYPE-NUMBER, TARGET-CONSTANT) I
    [SYMBOL-WORD-TYPE:1] )
```

Listing 15: *Nibbles mental model for 2 lines from* `scope diffname` *with CDO extensions.*

Besides the mistaken interpretation of line 8, there may be other sources of error in `scope diffname`. Our simple extensions to Nibbles model the notion of code that "does something" at the level of a *single line*. This concept could be applied to an entire Line Group, or even a Program. A Line Group, for example, could be said to do something if it contains a line with at least one complete transaction. Additionally, human programmers likely have a much more nuanced model of transactions. Research on the **roles of variables** in code has uncovered a common set of parts that variables play [74], such as a Fixed Value or Most Recent Holder. A more comprehensive model, perhaps including these variable roles, would boost Nibbles' ability to model human errors.

Predictions. Based on the Nibbles model alone, we might predict that more experienced Python programmers (i.e., those with an extended mental model) would be less likely to make errors on `scope diffname`. This was observed in the eyeCode experiment, though the effect was not large. If our model is correct, we should also expect that some programmers will assume that `added` is modified by `add_1` or `twice` *without seeing the definitions of either function*. The Nibbles model also predicts that changing function names should have no effect – e.g., `foo(added)` instead of `add_1(added)`. We doubt this would hold in a real experiment, and is another opportunity for lexical information to be incorporated (Section 18.5.1).

Open Questions. Do transactions really capture how human programmers determine whether or not a piece of code "does something"? And do programmers use the same principles at and above the line level? Saying that a function does something useful may depend on more factors than whether or not its body contains a complete transaction. We expect that programmers have a sense of whether or not a transaction is *externally visible* outside of a function: for example, a `print` or `return` statement. An entire experiment with variants of `scope diffname` could be constructed to test this hypothesis, having some variants perform externally visible actions in the bodies of `add_1` and `twice`. With at least one of these actions present, would most programmers still assume that `added` is modified by the function calls?

18.5.4 Model Capabilities

Nibbles has a number of interesting capabilities, both as a model of program comprehension and as a cognitive model in general. Most importantly, the model can **make errors** when interpreting a program's code. Unlike Mr. Bits, whose goal stack is pre-populated by the Python debugger, Nibbles may interpret a program differently depending on which choice points are "primed" in the moment. Because we assume the **first solution** given by the constraint solver is the "best" one, learning in the Nibbles model is entirely due to the re-ordering of choice points, which determine how the CDO space is explored. The initial ordering represents pre-existing knowledge of Python – e.g., which types of expressions are more likely than others. As the program is read, line-by-line, Nibbles adjusts the order of choice points to match what it observes. While this learning rule is simple, however, we do not currently have experimental evidence to support its use (see Section 18.5.5 for more threats to validity).

Although the CDO tree for Nibbles (Figure 99) is relatively simple compared to the programs experts comprehend on a daily basis, the *space of possible programs* it represents is quite large. Nibbles cannot practically search the entire space for every token or line that it encounters. A form of **attention** is necessary to reduce the size of the search space, but also adds to the model's plausibility; humans probably do not infer the 47th token of a program after reading the first five. Nibbles' attention is controlled it two ways: first, by reducing the detail level of tokens/lines that are far away from the sensor. Second, by accumulating assertions as the program is read, and fixing the values of choice points. The specifics of how details are degraded with distance from the sensor should be refined with additional experiments. Our design decisions are based loosely on the Mr. Chips model of reading [57], but the possibility exists for a much richer degradation model.

For example, tokens highlighted with a specific color could be considered keywords, regardless of their distance from the sensor. Additionally, classes of characters may be distinguishable just outside the sensor, such as alphabetic, numeric, and punctuation. These additional hints would speed up Nibbles' search and, combined with a more realistic model of character recognition, help language designers quantify the trade-offs inherent in choosing a language's tokens.
Finally, Nibbles is able to infer (or abduce) unobserved code at multiple levels of abstraction: characters, tokens, and lines. At its heart, Nibbles uses the Screamer constraint solver to enumerate possible constraint-complaint solutions. Unobserved code is simply unconstrained by anything other than Nibbles' knowledge of Python and its expectations of complete transactions, etc. (Section 18.5.3). Because of this, Nibbles can both *recognize and generate* programs. While it is possible to parse and generate programs purely using Python's grammar, Nibbles will only recognize/generate *schematic* programs – e.g., those that satisfy **all** of its constraints. These constraints need not always be about Python, and may exclude or discount programs that are well-formed but unconventional (e.g., `scope diffname`).

18.5.5 Limitations and Threats to Validity

Although Nibbles is a step forward for human cognitive models of program comprehension, it has many limitations. Like Mr. Bits, Nibbles is unable to handle larger, "real world" programs with user-defined classes, nested/recursive function calls, and code spread across multiple files. A great deal of future work is needed to take Nibbles from toy programs to those encountered by programmers every day. For these kinds of programs, Nibbles will need to be much more autonomous; behaving as an **agent**. Knowledge from previous code lines (and previous programs) will need to be retained in a declarative memory, and recalled as the program is read. Goals will have to be generated, nested, and pursued, such as evaluating a function call on the right-hand side of an assignment. And accumulated knowledge may need to be discarded or re-evaluated when inconsistencies are noted. The incorrectly inferred type of a variable, for example, could cause Nibbles to return no solutions for downstream code.
A major assumption underlying the Nibbles model is that *the first solution returned is the best interpretation*. This allows us to model priming and expectations simply by reordering choice

points. Of course, other methods are available for obtaining the "best" solution, such as ranking all solutions according to a set of *utility functions* (Section 16.3.5). Enumerating and ranking all of Nibbles' solutions would require more computing resources, and some calculus for combining utility values. It may also be the case that human programmers entertain *multiple or partial solutions*, perhaps blending them to produce a "final" interpretation. For now, we leave these questions to future work with the hope that research into mental models [50] in general will inform future modeling efforts.

Finally, Nibbles' CDO structure and constraints are **engineered** to produce a fit with human behavior. Nibbles does not learn except through the accumulation of assertions about the current token, line, etc. from observation. While the learning of a domain's structure and constraints is a fascinating research area, Nibbles is "hand-built" from experimental observations and expert introspection. With very few program comprehension experiments to draw from [29], we must make some assumptions. Unlike the many **qualitative** cognitive models available (Section 15.2), however, Nibbles can be *executed and empirically tested*.

19 Part 3: Conclusion

In this part, we described and evaluated two quantitative cognitive models of program comprehension: Mr. Bits and Nibbles. Mr. Bits sits at a lower level of abstraction, producing eye movements (fixations) and keystroke timings (responses). The time Mr. Bits spends looking at each line of code correlates strongly with human data from the eyeCode experiment (Section 17.5.1), and the time taken to finish evaluating each program is roughly average when compared to real programmers (Section 17.5.2). With ACT-R's sub-symbolic mode enabled, and without the ability to perfectly remember long lists of numbers, Mr. Bits successfully instantiates most aspects of the cognitive complexity metric [16]. Among its many limitations (Section 17.5.4), however, is the inability to parse raw text into a "mental" model and generate its own goals. Mr. Bits shifts visual attention around the screen and remembers/recalls variable values, but it relies entirely on Python's built-in debugger to tell it *what to do next*.

In the future, we hope to use Nibbles as the bridge between raw text and Mr. Bits. Nibbles can parse raw text into an internal representation, which may or may not contain **errors**. Unlike Mr. Bits, Nibbles transforms raw text into a *program model*, which contains the necessary information for goal generation (e.g., which lines should be evaluated in the body of a `for` loop). Using a constraint solver, Nibbles takes observed characters in the program text and generates likely interpretations of the current token, line, line group, and program. Besides grammatical constraints from Python, Nibbles models human expectations or *rules of discourse* [89]. Schema-compliant code is preferred, such as lines that form complete transactions (Section 18.5.3), though Nibbles can still recognize non-schematic code. Together, Mr. Bits and Nibbles may come close to realizing the majority of Von Mayrhauser's Integrated Metamodel (Figures 79 and 102).

Research Goals. Our main goal for this research was to develop a realizable (executable) theory of program comprehension that (at least) could explain the results of the eyeCode experiment. Mr. Bits and Nibbles have come a long way towards this goal, with Nibbles especially providing specific, testable explanations for the observed human errors. Additional experiments are necessary to validate the many design decisions that were made to match human behavior. Mr. Bits' ACT-R parameters, for example, were hand-picked to ensure realistic trial times (though these parameters are within reasonable human ranges). Nibbles' program model is largely based on the

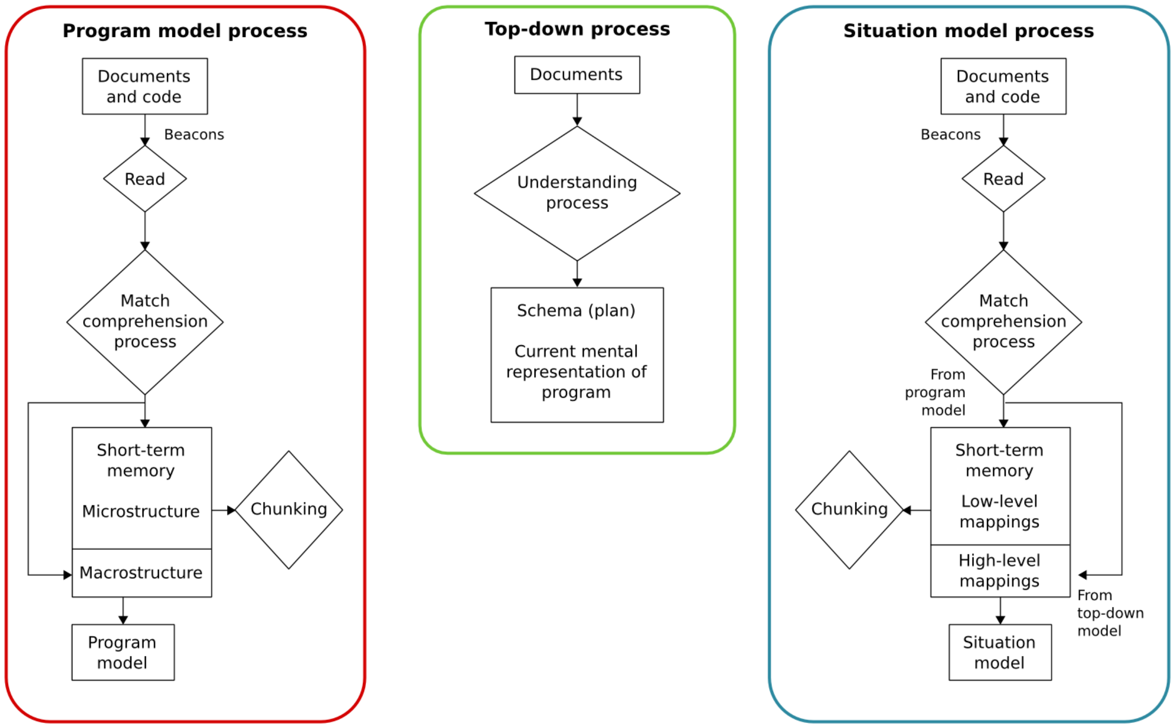


Figure 102: *Low-level view of Von Mayrhauser's Integrated Metamodel (re-created from Figure 6 in [101])*

textual structure of code, as described by Pennington [70], but many of the non-textual aspects, such as (semantic) line grouping, type inference, and transactions are less grounded in empirical research. Research on natural language text understanding may provide more guidance [54], but only insofar as a program's code is like a natural text document. In other words, programmers may construct a *situation model* of a program that bears little resemblance to a person's mental representation of a story or technical document [101].

Model Contributions. How could our models help inform programming language design? Programming languages are rarely designed based on empirical research (Quorum being a notable exception [91]). Instead, they are "evolved" in response to new application areas, requests from their respective user communities, or the emergence of new features in other languages. Predicting how programmers of varying experience levels will be impacted by a change to the language is very difficult without proper, controlled human experiments. With a realistic cognitive model, however, it may be possible to predict the impact of a given language change, and rank order different design alternatives. While not a replacement for human experiments, a more comprehensive version of Nibbles could predict sources of ambiguity or error when performing specific tasks on code written in the modified language. For example, overloading a new operator may cause a Nibbles "test case" to fail because it no longer has enough information available locally to infer a variable's type. Such test cases would call attention to situations where the reasoning methods of the programmer and compiler sharply diverge.

19.1 Future Work

There are many opportunities to extend and improve Mr. Bits and Nibbles, both individually and as a combined model. We briefly discuss potential avenues for future research below (more details are available in Sections 17.5 and 18.5).

19.1.1 Improving Mr. Bits

Mr. Bits was designed to operationalize the cognitive complexity metric [16] in order to rank versions of the same program by their cognitive complexity – i.e., how difficult they would be to understand. We compared Mr. Bits' trial time (MBTT), the time it takes to evaluate and predict the output of a program, to several commonly used source code metrics. Our results were promising, but a thorough comparison of MBTT and more modern metrics, such as Douce's spatial complexity [32] or Buse's readability [11], would help characterize and (hopefully) distinguish this new metric. In the short term, Mr. Bits will also need to be extended to handle user-defined classes, which itself will requiring maintaining type information about variables. Therefore, it may be beneficial to consider any future versions of Mr. Bits as a companion to Nibbles, which does just that.

19.1.2 Nibbles 2.0

The next version of Nibbles will need to not only account for constraints above the line group level, but also incorporate knowledge beyond the scope of Python and programming. Function and variable names, for instance, are not opaque tokens – humans choose specific patterns and parts of speech to convey concepts in their code [8]. Constraints at the module and multi-module level could take Nibbles beyond the eyeCode output prediction task, and into more realistic areas like debugging and code review. Nibbles' simple decision procedure, pick the first solution, may not be sufficient for larger programs where multiple hypotheses need to be tested by gathering additional evidence. Having Nibbles base decisions off of the *entire solution space* would solve the problem, but will require significant computing resources (or a faster constraint solver!).

19.1.3 Combining Mr. Bits and Nibbles

A combination of the Mr. Bits and Nibbles model would form an *end-to-end* model of the eyeCode experiment task, transforming raw text into fixations and response keystrokes. A first step towards this goal would be to modify the early stages of Mr. Bits, and have it generate goals based on Nibbles' mental models. The reading behavior of Mr. Bits could be much improved by only fixating tokens that Nibbles could not infer based on Python's grammar. In the end, the joint Mr. Bits + Nibbles model would need to generate goals on the fly as text is read in order to disambiguate unobserved code and trace the necessary dependencies for evaluation.

20 Main Conclusion

What makes some code harder to understand? In this dissertation, we quantified the *cognitive complexity* of a program via two *cognitive models* of program understanding: **Mr. Bits** and **Nibbles**. These models were informed by data collected from 162 programmers in a large (mostly) online experiment. This experiment, called **eyeCode**, asked programmers to read 10 short Python programs and predict their printed output. We used the errors our participants made to inform Nibbles' "mental model" generation process, and were able to provide *explanations* for these errors. For example, the most common error was made in a version of the `counting` program (Section 5.2.2) involving the mistaken exclusion of a line from the body of a loop. Nibbles is able to produce this error when indentation details for the lines are not explicitly observed, but will correctly group lines in the loop body if local indentation information is available. While the structure of Nibbles' domain and its constraints were engineered specifically to model these errors, the formalism upon which Nibbles is based can be generalized to more complex situations or even other programming languages.

A subset of our 162 participants (29) performed the eyeCode experiment in front of an eye-tracker, allowing us to observe which lines of code were being fixated during the course of the experiment. The Mr. Bits model, built on the ACT-R cognitive architecture, predicts this kind of eye movement data. When comparing Mr. Bits and the eyeCode data, we found that Mr. Bits performed best when ACT-R's sub-symbolic mode was enabled and when the model was not allowed to remember lists of numbers after a single viewing. In isolation, however, Mr. Bits did **not** outperform a simple model (based soley on line length) for predicting relative time spent fixating each line of code. While the line length model is certainly more parsimonious, it cannot also be used to predict keystroke timings, line transition frequencies, and the overall time taken for each trial. Additionally, the Mr. Bits model (like the Mr. Chips model of text reading) can *explain why* its predictions are what they are. The positive correlation between line length and fixation time, for example, is likely an artifact of our simple Python programs. The longest lines in these short, simple programs contained lists of numbers which were crucial for the output prediction task. Were these lines to contain strings with predictable prose or highly schematic boilerplate code, we would expect the correlation to diminish significantly.

Besides the two cognitive models, there were two additional contributions. First, we developed a set of performance metrics for assessing and ranking the eyeCode participants. These metrics quantify the correctness of a trial and categorize both keystrokes and periods of time within a trial. Similarly, we collected a number of eye movement metrics and visualizations for comparing and contrasting our eye-tracking trials. An open source Python data analysis library (also called eyeCode) was developed, and serves as an additional contribution. The most interesting visualization technique made use of rolling time windows, and allowed us to identify points where participants were likely performing mental calculations (Section 12.2.3). Applying these metrics and visualizations to other eye-tracking data sets would be an interesting avenue for future work. Overall, it is clear from the experiments and models provided here that: (1) a cognitive modeling approach to program understanding is necessary for a more complete explanation of observed programmer behaviors, and (2) a great deal of work remains to be done in both the modeling of program understanding and quantitative cognitive modeling in general. Even with decades of research, for example, the ACT-R cognitive architecture quickly becomes unwieldy when modeling phenomena more complex than simple forced-choice tasks or paired associations. New formalisms, like Cognitive Domain Ontologies, facilitate a higher-level of abstraction, but lack strong experimental evidence. Program understanding provides a challenging cognitive modeling opportunity, especially when the desired model produces quantitative predictions. We hope that this field will help to expand the frontiers of Cognitive Science while formalizing one of the more difficult cognitive tasks that humans perform.

21 Acknowledgements

My thanks and appreciation go to Drs. Rob Goldstone and Andrew Lumsdaine: my advisors and (intellectual) parents. They allowed me to pursue a topic somewhat off the beaten path, and to shape it into something uniquely mine. Dr. Goldstone took me in as an undergraduate research programmer, and connected me with Dr. Lumsdaine. I've learned a great amount from both of them, including the amount of dedication it takes to maintain a lab and collection of scruffy grad students. Their support, both personally and financially, will not be forgotten.

I am also thankful for the friendships I found in the Open Systems Lab (now called CREST), and in the Percepts & Concepts Lab. Growing intellectually with my cohort group while heeding the wisdom of the post-docs made for an overall enjoyable graduate school experience.

Lastly, I'd like to thank my colleague and mentor Scott Douglass. Our conversations during my summer internships (and later employment) at the Air Force Research Lab were instrumental in my eye-tracking and modeling work. I've mature as a scientist through our friendship, and his advocacy helped get my career started as a researcher.

Grant R305A1100060 from the Institute of Education Sciences Department of Education and grant 0910218 from the National Science Foundation REESE supported this research.

A Appendix - Programs and Output

A.1 between

A.1.1 between - functions

```python
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
```

```
[8, 7, 9]
[1, 0, 8, 1]
[8, 9, 0]
```

A.1.2 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
```

```
[8, 7, 9]
[1, 0, 8, 1]
[8, 9, 0]
```

A.2 counting

A.2.1 counting - nospace

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

```
The count is 1
Done counting
The count is 2
Done counting
The count is 3
Done counting
The count is 4
Done counting
```

A.2.2 counting - twospaces

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


    print "Done counting"
```

```
The count is 1
Done counting
The count is 2
Done counting
The count is 3
Done counting
The count is 4
Done counting
```

A.3 funcall

A.3.1 funcall - nospace

```
def f(x):
    return x + 4

print f(1)*f(0)*f(-1)
```

```
60
```

A.3.2 funcall - space

```
def f(x):
    return x + 4

print f(1) * f(0) * f(-1)
```

```
60
```

A.3.3 funcall - vars

```
def f(x):
    return x + 4

x = f(1)
y = f(0)
z = f(-1)
print x * y * z
```

```
60
```

A.4 initvar

A.4.1 initvar - bothbad

```
a = 0
for i in [1, 2, 3, 4]:
    a = a * i
print a

b = 1
for i in [1, 2, 3, 4]:
    b = b + i
print b
```

```
0
11
```

A.4.2 initvar - good

```
a = 1
for i in [1, 2, 3, 4]:
    a = a * i
print a

b = 0
for i in [1, 2, 3, 4]:
    b = b + i
print b
```

```
24
10
```

A.4.3 initvar - onebad

```
a = 1
for i in [1, 2, 3, 4]:
    a = a * i
print a

b = 1
for i in [1, 2, 3, 4]:
    b = b + i
print b
```

```
24
11
```

A.5 order

A.5.1 order - inorder

```python
def f(x):
    return x + 4

def g(x):
    return x * 2

def h(x):
    return f(x) + g(x)

x = 1
a = f(x)
b = g(x)
c = h(x)
print a, b, c
```

```
5 2 7
```

A.5.2 order - shuffled

```
def h(x):
    return f(x) + g(x)

def f(x):
    return x + 4

def g(x):
    return x * 2

x = 1
a = f(x)
b = g(x)
c = h(x)
print a, b, c
```

```
5 2 7
```

A.6 overload

A.6.1 overload - multmixed

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

c = 7
d = 2
print c * d

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

```
12
14
53
```

A.6.2 overload - plusmixed

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

c = 7
d = 2
print c + d

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

```
7
9
53
```

A.6.3 overload - strings

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

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

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

```
hibye
streetpenny
53
```

A.7 partition

A.7.1 partition - balanced

```
for i in [1, 2, 3, 4, 5]:
    if (i < 3):
        print i, "low"
    if (i > 3):
        print i, "high"
```

```
1 low
2 low
4 high
5 high
```

A.7.2 partition - unbalanced

```
for i in [1, 2, 3, 4]:
    if (i < 3):
        print i, "low"
    if (i > 3):
        print i, "high"
```

```
1 low
2 low
4 high
```

A.7.3 partition - unbalanced_pivot

```
pivot = 3
for i in [1, 2, 3, 4]:
    if (i < pivot):
        print i, "low"
    if (i > pivot):
        print i, "high"
```

```
1 low
2 low
4 high
```

A.8 rectangle

A.8.1 rectangle - basic

```
def area(x1, y1, x2, y2):
    width = x2 - x1
    height = y2 - y1
    return width * height

r1_x1 = 0
r1_y1 = 0
r1_x2 = 10
r1_y2 = 10
r1_area = area(r1_x1, r1_y1, r1_x2, r1_y2)
print r1_area

r2_x1 = 5
r2_y1 = 5
r2_x2 = 10
r2_y2 = 10
r2_area = area(r2_x1, r2_y1, r2_x2, r2_y2)
print r2_area
```

```
100
25
```

A.8.2 rectangle - class

```
class Rectangle:
    def __init__(self, x1, y1, x2, y2):
        self.x1 = x1
        self.y1 = y1
        self.x2 = x2
        self.y2 = y2

    def width(self):
        return self.x2 - self.x1

    def height(self):
        return self.y2 - self.y1

    def area(self):
        return self.width() * self.height()

rect1 = Rectangle(0, 0, 10, 10)
print rect1.area()

rect2 = Rectangle(5, 5, 10, 10)
print rect2.area()
```

```
100
25
```

A.8.3 rectangle - tuples

```python
def area(xy_1, xy_2):
    width = xy_2[0] - xy_1[0]
    height = xy_2[1] - xy_1[1]
    return width * height

r1_xy_1 = (0, 0)
r1_xy_2 = (10, 10)
r1_area = area(r1_xy_1, r1_xy_2)
print r1_area

r2_xy_1 = (5, 5)
r2_xy_2 = (10, 10)
r2_area = area(r2_xy_1, r2_xy_2)
print r2_area
```

```
100
25
```

A.9 scope

A.9.1 scope - diffname

```python
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
```

```
4
```

A.9.2 scope - samename

```python
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
```

```
4
```

A.10 whitespace

A.10.1 whitespace - linedup

```
intercept = 1
slope     = 5

x_base  = 0
x_other = x_base + 1
x_end   = x_base + x_other + 1

y_base  = slope * x_base  + intercept
y_other = slope * x_other + intercept
y_end   = slope * x_end   + intercept

print x_base,  y_base
print x_other, y_other
print x_end,   y_end
```

```
0 1
1 6
2 11
```

A.10.2 whitespace - zigzag

```
intercept = 1
slope = 5

x_base = 0
x_other = x_base + 1
x_end = x_base + x_other + 1

y_base = slope * x_base + intercept
y_other = slope * x_other + intercept
y_end = slope * x_end + intercept

print x_base, y_base
print x_other, y_other
print x_end, y_end
```

```
0 1
1 6
2 11
```

B Appendix - Program Metrics

Base	Version	Code Ch	Code Ln	CC	HE	HV	Ouput Ch	Output Ln
between	functions	496	24	7	94192.1	830.2	33	3
between	inline	365	19	7	45596.3	660.8	33	3
counting	nospace	77	3	2	738.4	82.0	116	8
counting	twospaces	81	5	2	738.4	82.0	116	8
funcall	nospace	50	4	2	937.7	109.4	3	1
funcall	space	54	4	2	937.7	109.4	3	1
funcall	vars	72	7	2	1735.7	154.3	3	1
initvar	bothbad	103	9	3	3212.5	212.4	5	2
initvar	good	103	9	3	3212.5	212.4	6	2
initvar	onebad	103	9	3	2866.8	208.5	6	2
order	inorder	137	14	4	8372.3	303.1	6	1
order	shuffled	137	14	4	8372.3	303.1	6	1
overload	multmixed	78	11	1	2340.0	120.0	9	3
overload	plusmixed	78	11	1	3428.3	117.2	7	3
overload	strings	98	11	1	3428.3	117.2	21	3
partition	balanced	105	5	4	2896.0	188.9	26	4
partition	unbalanced	102	5	4	2382.3	177.2	19	3
partition	unbalanced_pivot	120	6	4	2707.8	196.2	19	3
rectangle	basic	293	18	2	18801.2	396.3	7	2
rectangle	class	421	21	5	43203.7	620.1	7	2
rectangle	tuples	277	14	2	15627.7	403.8	7	2
scope	diffname	144	12	3	2779.7	188.0	2	1
scope	samename	156	12	3	2413.3	183.6	2	1
whitespace	linedup	275	14	1	6480.0	216.0	13	3
whitespace	zigzag	259	14	1	6480.0	216.0	13	3

Table 21: *Metrics for all 25 Python programs (10 bases, 2-3 versions). From left to right: code characters, code lines, Cyclomatic Complexity, Halstead Effort, Halstead Volume, output characters, output lines.*

C Appendix - Computer Example

```
;;; Author: Michael Hansen

;;; Created on 2015-03-06 14:46:21.715348


(in-package :cdo)


;;; Constraints added by default

(defparameter *computer-configuration-default-constraints* '())


;;; Instance parameters

(defparameter *components-n* 8)

(defparameter *memory-type-boost* 5)


;;; ------------------------------------------------------------------------


(defun print-component (comp)

  (let* ((performance (v@ (comp) performance))

         (comp-type (name^ (e@ comp component-type)))

         (vendor (name^ (e@ comp component-details vendor vendor-choice)))

         (cost (v@ (comp component-details product-info) cost))

         (performance (v@ (comp component-details product-info) performance))

         (model (value-of (v@ (comp component-details product-info) model)))

         (used (name^ (e@ comp component-role)))

         (mem-type (when (equale (e@ comp component-type) memory)

                     (name^ (e@ comp component-type memory memory-type))))

         )

    (with-output-to-string

        (str)

      (format str "~a [~a] ~a ~a~a (c:~a, p:~a)"
              (if (eq used 'used) "*" " ")
              comp-type vendor
              (if (ground? model) model "Unknown Model")
              (cond
                ((eq mem-type 'type-a) ", A")
                ((eq mem-type 'type-b) ", B")
                (t ""))
              (if (ground? cost) cost "?")
              (if (ground? performance) performance "?"))
      )
    ))

(defun print-configuration (cfg)
  (let* ((cost (v@ (cfg) cost))
         (performance (v@ (cfg) performance))
         (components (entities^ (n@ cfg components)))
         )
    (with-output-to-string
        (str)
      (format str "Configuration (cost:~a, perf:~a):~%"
              (if (ground? cost) cost "?")
              (if (ground? performance) performance "?"))
      (dolist (comp components)
        (format str "    ~a~%" (print-component comp))
        )
      (format str "~%")
      )
  ))

;;; ==========================================================================
;;;                               Structure
;;; ==========================================================================


;;; Top-level entities
(defun computer-configuration_ ( &rest constraints)
  (multiple-value-bind (local-constraints relayed-constraints)
      (isolate-constraints :computer-configuration
          (append constraints *computer-configuration-default-constraints*)  )
    `(let* (
            (components
             ,(apply #'dm 'components *components-n*
                     #'component_ relayed-constraints))


            (computer-configuration
             (de 'computer-configuration
                 :r ( components  )
                 :v ( ,(dv 'cost (an-integerv))
                      ,(dv 'performance (an-integerv))   )
                 )
              ))
       ;;
       ,@local-constraints
       ;;
       computer-configuration )))


(defun component_ (n &rest constraints)
  (multiple-value-bind (local-constraints relayed-constraints)
      (isolate-constraints :component  constraints  n)
    `(let* (
            (component-details
             (da 'component-details
                 (de 'product-info
                     :v ( ,(dv 'cost (an-integerv))
                          ,(dv 'performance (an-integerv))
                          ,(dv 'model (a-stringv))  )
                     )
                 ,(apply #'vendor_ relayed-constraints)
                 ))
            (component-role
             (ds 'component-role
                 (de 'used
                     )
                 (de 'not-used
                     )
                 ))
            (component-type
             (ds 'component-type
                 ,(apply #'memory_ relayed-constraints)
                 (de 'graphics
                     )
                 (de 'sound
                     )
                 ))

            (component
             (de 'component
                 :r ( component-details  component-role  component-type  )
                 )
              ))
       ;;
```

```
       ,@local-constraints
       ;;
       component )))

(defun vendor_ ( &rest constraints)
  (multiple-value-bind (local-constraints relayed-constraints)
      (isolate-constraints :vendor  constraints  )
    `(let* (
            (vendor-choice
             (ds 'vendor-choice
                 (de 'invideo
                     )
                 (de 'slamsong
                     )
                 (de 'tortoise-bay
                     )
                 ))

            (vendor
             (de 'vendor
                 :r ( vendor-choice  )
                 )
              ))
       ;;
       ,@local-constraints
       ;;
       vendor )))

(defun memory_ ( &rest constraints)
  (multiple-value-bind (local-constraints relayed-constraints)
      (isolate-constraints :memory  constraints  )
    `(let* (
            (memory-type
             (ds 'memory-type
                 (de 'type-a
                     )
                 (de 'type-b
                     )))

            (memory
             (de 'memory
                 :r ( memory-type  )
                 )
              ))
       ;;
       ,@local-constraints
       ;;
       memory )))


;;; ---------------------------------------------------------------------------


;;; Function to count solutions
(defun computer-configuration-counter_ (&rest constraints)
  (multiple-value-bind (local-constraints relayed-constraints)
      (isolate-constraints :computer-configuration constraints)
    `(let ((count 0))
       (for-effects
        (let ((return-value
               (progn
                (let* (
                       (components
                        ,(apply #'dm 'components *components-n*
                               #'component_ relayed-constraints))

                       (computer-configuration
                        (de 'computer-configuration
                            :r ( components  )
                            :v ( ,(dv 'cost (an-integerv))
                                 ,(dv 'performance (an-integerv))  )
                            )
                         ))
                  ;;
                  ,@local-constraints
                  ;;
                  computer-configuration ))))
          return-value
          (global
           (setf count (1+ count)))))
       count)))

;;; ==========================================================================
;;;                             Constraints
;;; ==========================================================================

(define-ma-constraint-fn component-active (comp)
    (equale (e@ comp component-role) used))

(define-ma-constraint-fn has-graphics (comp)
  (andv
   (equale (e@ comp component-role) used)
   (equale (e@ comp component-type) graphics)))

(define-ma-constraint-fn has-sound (comp)
  (andv
   (equale (e@ comp component-role) used)
   (equale (e@ comp component-type) sound)))

(define-ma-constraint-fn has-memory (comp)
  (andv
   (equale (e@ comp component-role) used)
   (equale (e@ comp component-type) memory)))

(define-constraint max-4-active
    :computer-configuration
    (at-most-ma 4 component-active (n@ components)))

(define-constraint only-1-graphics-card
    :computer-configuration
    (exactly-ma 1 has-graphics (n@ components)))

(define-constraint only-1-sound-card
    :computer-configuration
    (exactly-ma 1 has-sound (n@ components)))

(define-constraint 1-or-2-memory
    :computer-configuration
    (at-least-ma 1 has-memory (n@ components)))
```

```
;; Constrain product types by vendor
(define-constraint vendors-types
    :component
  (let* ((vendor-choice (e@ component component-details vendor vendor-choice))
         (component-type (e@ component component-type))
         )
  (cond
    ((equale vendor-choice invideo) (equale component-type graphics))
    ((equale vendor-choice tortoise-bay) (equale component-type sound))
    ((equale vendor-choice slamsong) (orv
                                       (equale component-type graphics)
                                       (equale component-type sound)
                                       (equale component-type memory)))
    (t t)
    )
  ))

;; Sum component costs for a configuration
(define-constraint config-cost
    :computer-configuration
  (let ((comp-costs (mapcar
                     (lambda (comp)
                       (ifv (equale (e@ comp component-role) used)
                            (v@ (comp component-details product-info) cost)
                            0))
                     (entities^ (n@ components))))
        )
    (equalv (v@ (computer-configuration) cost) (applyv #'+v comp-costs))
    )
  )
```

```
;; Sum component performance for a configuration
;; (add boost for same memory types)
(define-constraint config-perf
    :computer-configuration
  (let* ((mem-count (applyv #'+v
                            (mapcar
                             (lambda (comp)
                               (ifv (andv
                                     (equale (e@ comp component-role) used)
                                     (equale (e@ comp component-type) memory))
                                    1
                                    0))
                             (entities^ (n@ components)))))
         (mem-types (remove-duplicates
                     (remove 'nil
                             (mapcar
                              (lambda (comp)
                                (ifv (andv
                                      (equale (e@ comp component-role) used)
                                      (equale (e@ comp component-type) memory))
                                     (name^ (e@ comp component-type memory
                                                memory-type))
                                     nil))
                              (entities^ (n@ components))))))
         (mem-boost (ifv (andv (>v mem-count 1)
                               (eq (length mem-types) 1))
                         *memory-type-boost*
                         0))
         (comp-perfs (mapcar
                    (lambda (comp)
                      (ifv (equale (e@ comp component-role) used)
                           (v@ (comp component-details product-info)
                               performance)
                           0))
                    (entities^ (n@ components))))
        )
    (equalv (v@ (computer-configuration) performance)
            (+v mem-boost (applyv #'+v comp-perfs)))
    )
  )


;; Force zero cost
(define-constraint no-cost
    :computer-configuration
  (equalv (v@ (computer-configuration) cost) 0))


;; Force new graphics card
(define-constraint new-graphics-card
    :component
  (ifv (andv
        (equale (e@ component component-role) active)
        (equale (e@ component component-type) graphics))
       (>v (v@ (component component-details product-info) cost) 0)
       t
       ))

;;; ===========================================================================
;;;                              Examples
;;; ============================================================================

;; One solution, no constraints

(print
 (soaCDO-solutions
  (computer-configuration_)
  :one
  :print-fun #'print-configuration))

;; Configuration (cost:?, perf:?):
;;      * [MEMORY] INVIDEO Unknown Model, A (c:?, p:?)
;;      * [MEMORY] INVIDEO Unknown Model, A (c:?, p:?)
;;      * [MEMORY] INVIDEO Unknown Model, A (c:?, p:?)
;;      * [MEMORY] INVIDEO Unknown Model, A (c:?, p:?)
;;      * [MEMORY] INVIDEO Unknown Model, A (c:?, p:?)
;;      * [MEMORY] INVIDEO Unknown Model, A (c:?, p:?)
;;      * [MEMORY] INVIDEO Unknown Model, A (c:?, p:?)
;;      * [MEMORY] INVIDEO Unknown Model, A (c:?, p:?)

;; ----------------------------------------------------------------------------

(defparameter *available-components*
  (list
   ;; New components
   (set-ma-instance-properties
    '(:component) '(1)
    :choices '(
               ((component component-details vendor vendor-choice) invideo)
```

```
            ((component component-type) graphics)
            )

 :variables '(
              (((component component-details product-info) model) "D-Force")
              (((component component-details product-info) cost) 200)
              (((component component-details product-info) performance) 10)
              )
 )

(set-ma-instance-properties
 '(:component) '(2)
 :choices '(
            ((component component-details vendor vendor-choice) slamsong)
            ((component component-type) memory)
            ((component component-type memory memory-type) type-b)
            )

 :variables '(
              (((component component-details product-info) model) "DRR9")
              (((component component-details product-info) cost) 20)
              (((component component-details product-info) performance) 10)
              )
 )

(set-ma-instance-properties
 '(:component) '(3)
 :choices '(
            ((component component-details vendor vendor-choice) tortoise-bay)
            ((component component-type) sound)
            )

 :variables '(
              (((component component-details product-info) model) "Waves")
              (((component component-details product-info) cost) 50)
              (((component component-details product-info) performance) 10)
              )
 )

(set-ma-instance-properties
 '(:component) '(4)
 :choices '(
            ((component component-details vendor vendor-choice) slamsong)
            ((component component-type) memory)
            ((component component-type memory memory-type) type-a)
            ;; ((component component-role) used)
            )

 :variables '(
              (((component component-details product-info) model) "DRR7")
              (((component component-details product-info) cost) 10)
              (((component component-details product-info) performance) 5)
              )
 )

(set-ma-instance-properties
 '(:component) '(5)
 :choices '(
            ((component component-details vendor vendor-choice) invideo)
            ((component component-type) graphics)
            )

 :variables '(
             (((component component-details product-info) model) "B-Force")
             (((component component-details product-info) cost) 100)
             (((component component-details product-info) performance) 7)
             )
 )

;; Existing components
(set-ma-instance-properties
 '(:component) '(6)
 :choices '(
            ((component component-details vendor vendor-choice) slamsong)
            ((component component-type) memory)
            ((component component-type memory memory-type) type-a)
            ;; ((component component-role) used)
            )

 :variables '(
             (((component component-details product-info) model) "DRR7")
             (((component component-details product-info) cost) 0)
             (((component component-details product-info) performance) 5)
             )
 )

(set-ma-instance-properties
 '(:component) '(7)
 :choices '(
            ((component component-details vendor vendor-choice) slamsong)
```

```
          ((component component-type) sound)
          )

   :variables '(
            (((component component-details product-info) model) "Puddle")
            (((component component-details product-info) cost) 0)
            (((component component-details product-info) performance) 1)
            )
   )

  (set-ma-instance-properties
   '(:component) '(8)
   :choices '(
          ((component component-details vendor vendor-choice) slamsong)
          ((component component-type) graphics)
          )

   :variables '(
            (((component component-details product-info) model) "A-Force")
            (((component component-details product-info) cost) 0)
            (((component component-details product-info) performance) 2)
            )
   )))

;; ---------------------------------------------------------------------------

;; One solution, just components
(print
  (soaCDO-solutions
   (apply #'computer-configuration_
         (cons config-cost
               (cons config-perf
                     *available-components*)))
   :one
   :print-fun #'print-configuration))

;; Configuration (cost:380, perf:50):
;;      * [GRAPHICS] INVIDEO D-Force (c:200, p:10)
;;      * [MEMORY] SLAMSONG DRR9, B (c:20, p:10)
;;      * [SOUND] TORTOISE-BAY Waves (c:50, p:10)
;;      * [MEMORY] SLAMSONG DRR7, A (c:10, p:5)
;;      * [GRAPHICS] INVIDEO B-Force (c:100, p:7)
;;      * [MEMORY] SLAMSONG DRR7, A (c:0, p:5)
;;      * [SOUND] SLAMSONG Puddle (c:0, p:1)
;;      * [GRAPHICS] SLAMSONG A-Force (c:0, p:2)

;; ----------------------------------------------------------------------------

(defparameter *default-constraints*
  `(
    ,config-cost
    ,config-perf

    ,max-4-active
    ,only-1-graphics-card
    ,only-1-sound-card
    ,1-or-2-memory

    ,vendors-types
    ,@available-components
    ))

;; -----------------------------------------------------------------------------

;; One solution, no additonal constraints
(print
  (soaCDO-solutions
   (apply #'computer-configuration_
          *default-constraints*)
   :one
   :print-fun #'print-configuration))

;; Configuration (cost:280, perf:35):
;;     * [GRAPHICS] INVIDEO D-Force (c:200, p:10)
;;     * [MEMORY] SLAMSONG DRR9 B (c:20, p:10)
;;     * [SOUND] TORTOISE-BAY Waves (c:50, p:10)
;;     * [MEMORY] SLAMSONG DRR7 A (c:10, p:5)
;;       [GRAPHICS] INVIDEO B-Force (c:100, p:7)
;;       [MEMORY] SLAMSONG DRR7 A (c:0, p:5)
;;       [SOUND] SLAMSONG Puddle (c:0, p:1)
;;       [GRAPHICS] SLAMSONG A-Force (c:0, p:2)

;; -----------------------------------------------------------------------------

;; One solution, force no cost
(print
  (soaCDO-solutions
   (apply #'computer-configuration_
          (cons no-cost *default-constraints*))
   :one
   :print-fun #'print-configuration))

;; Configuration (cost:0, perf:8):
;;       [GRAPHICS] INVIDEO D-Force (c:200, p:10)
;;       [MEMORY] SLAMSONG DRR9, B (c:20, p:10)
;;       [SOUND] TORTOISE-BAY Waves (c:50, p:10)
;;       [MEMORY] SLAMSONG DRR7, A (c:10, p:5)
;;       [GRAPHICS] INVIDEO B-Force (c:100, p:7)
;;     * [MEMORY] SLAMSONG DRR7, A (c:0, p:5)
;;     * [SOUND] SLAMSONG Puddle (c:0, p:1)
;;     * [GRAPHICS] SLAMSONG A-Force (c:0, p:2)

;; ---------------------------------------------------------------------------

(defun performance-utility (sol)
  (value-of (v@ (sol) performance)))

;; Best performance
(multiple-value-bind (best-solutions best-value util-values)
    (soaCDO-solutions
     (apply #'computer-configuration_
            *default-constraints*)
     :best
     :utility-fun #'performance-utility
     :objective-fun #'>
     :print-fun #'print-configuration)

  (print best-solutions))
```

```
;; Configuration (cost:260, perf:35):
;;     * [GRAPHICS] INVIDEO D-Force (c:200, p:10)
;;       [MEMORY] SLAMSONG DRR9, B (c:20, p:10)
;;     * [SOUND] TORTOISE-BAY Waves (c:50, p:10)
;;     * [MEMORY] SLAMSONG DRR7, A (c:10, p:5)
;;       [GRAPHICS] INVIDEO B-Force (c:100, p:7)
;;     * [MEMORY] SLAMSONG DRR7, A (c:0, p:5)
;;       [SOUND] SLAMSONG Puddle (c:0, p:1)
;;       [GRAPHICS] SLAMSONG A-Force (c:0, p:2)

;;  Configuration (cost:270, perf:35):
;;     * [GRAPHICS] INVIDEO D-Force (c:200, p:10)
;;     * [MEMORY] SLAMSONG DRR9, B (c:20, p:10)
;;     * [SOUND] TORTOISE-BAY Waves (c:50, p:10)
;;       [MEMORY] SLAMSONG DRR7, A (c:10, p:5)
;;       [GRAPHICS] INVIDEO B-Force (c:100, p:7)
;;     * [MEMORY] SLAMSONG DRR7, A (c:0, p:5)
;;       [SOUND] SLAMSONG Puddle (c:0, p:1)
;;       [GRAPHICS] SLAMSONG A-Force (c:0, p:2)

;;  Configuration (cost:280, perf:35):
;;     * [GRAPHICS] INVIDEO D-Force (c:200, p:10)
;;     * [MEMORY] SLAMSONG DRR9, B (c:20, p:10)
;;     * [SOUND] TORTOISE-BAY Waves (c:50, p:10)
;;     * [MEMORY] SLAMSONG DRR7, A (c:10, p:5)
;;       [GRAPHICS] INVIDEO B-Force (c:100, p:7)
;;       [MEMORY] SLAMSONG DRR7, A (c:0, p:5)
;;       [SOUND] SLAMSONG Puddle (c:0, p:1)
;;       [GRAPHICS] SLAMSONG A-Force (c:0, p:2)
;; ----------------------------------------------------------------------------

;; Highest performance (first), lowest cost (second)
(defun better-2 (a b)
  (cond
    ((eq (first a) (first b)) (< (second a) (second b)))
    (t (> (first a) (first b)))))

(defun performance-and-cost-utility (sol)
  (list
   (value-of (v@ (sol) performance))
   (value-of (v@ (sol) cost))))

;; Best performance with lowest cost
(multiple-value-bind (best-solutions best-value util-values)
    (soaCDO-solutions
     (apply #'computer-configuration_
            *default-constraints*)
     :best
     :utility-fun #'performance-and-cost-utility
     :objective-fun #'better-2
     :print-fun #'print-configuration)

  (print best-solutions))

;; Configuration (cost:260, perf:35):
;;     * [GRAPHICS] INVIDEO D-Force (c:200, p:10)
;;       [MEMORY] SLAMSONG DRR9, B (c:20, p:10)
;;     * [SOUND] TORTOISE-BAY Waves (c:50, p:10)
;;     * [MEMORY] SLAMSONG DRR7, A (c:10, p:5)
;;        [GRAPHICS] INVIDEO B-Force (c:100, p:7)
;;      * [MEMORY] SLAMSONG DRR7, A (c:0, p:5)
;;        [SOUND] SLAMSONG Puddle (c:0, p:1)
;;        [GRAPHICS] SLAMSONG A-Force (c:0, p:2)

;; ---------------------------------------------------------------------------
```

D Appendix - Mr. Bits Code

```
;; ============================================================================
;;                                 MR. BITS
;; ============================================================================

;; Maximum number of seconds the model can run
(defparameter *max-time* 500)

;; ----------------------------------------------------------------------------
;; PROGRAM (funcall - space)
;; ----------------------------------------------------------------------------
;; 1. def f(x):
;; 2.     return x + 4
;; 3.
;; 4. print f(1) * f(0) * f(-1)
;; ----------------------------------------------------------------------------


; -----------------------------------------------------------------------------
; GOAL STACK
; -----------------------------------------------------------------------------

;; Fixed goal stack for model
(defparameter *goal-stack* '(
    go-to-line-1
    remember-line
    go-to-line-4
    link-to-value-f-x-L4-C8-0-1
    go-to-line-2
    do-read-line-2-f-x-1
    compute-sum-line-2-1-1
    remember-line
    go-to-line-4
    remember-line
    link-to-value-f-x-L4-C15-1-1
    go-to-line-2
    do-read-line-2-f-x-2
    compute-sum-line-2-2-1
    remember-line
    go-to-line-4
    remember-line
    link-to-value-f-x-L4-C22-2-1
    go-to-line-2
    do-read-line-2-f-x-3
    compute-sum-line-2-3-1
    remember-line
    go-to-line-4
    compute-prod-line-4-1-1
    compute-prod-line-4-2-1
    remember-line
    fixate-output-box
    type-response
    return-from-output-box
))

;; List of responses that will be typed
(defparameter *responses* (list
    (format nil "60~%")
))
```

```
;; List of tokens to put in visicon
(defparameter *tokens* `(
    (0 0 33 23 "def")
    (44 0 55 23 "f")
    (55 0 55 23 "(x):")
    (44 23 66 23 "return")
    (121 23 11 23 "x")
    (143 23 11 23 "+")
    (165 23 11 23 "4")
    (0 69 55 23 "print")
    (66 69 44 23 "f")
    (77 69 44 23 "(1)")
    (121 69 11 23 "*")
    (143 69 44 23 "f")
    (154 69 44 23 "(0)")
    (198 69 11 23 "*")
    (220 69 55 23 "f")
    (231 69 55 23 "(-1)")
))

; ---------------------------------------------------------------------------

;; Convert text character to a key ACT-R can type
(defun actr-key (c)
  (cond
    ((eq c #\newline) 'return)
    ((eq c #\space) 'space)
    ((eq c #\,) 'comma)
    (t (string c))))
```

```
;; Get the visicon y coordinate from a line number (1-based)
(defun line-to-y (line)
  (let ((y-pos (* 23 (- line 1)))
        (line-height (max (round 23 .8) (+ 23 1))))
    (+ y-pos (round line-height 2))))

;; Get the visicon x coordinate from a column number (0-based)
(defun col-to-x (col)
  (* col 11))

(defmethod rpm-window-key-event-handler ((win rpm-window) key)
  (proc-display))

;; Evalate the goal stack and type responses
(defun do-trial ()
  (reset)

  ; Consider underscores, commas, parens, colons, and brackets as
  ; word separating characters.
  (add-word-characters #\_ #\, #\( #\) #\: #\[ #\])

  ; Create an experiment window
  (let* ((tokens *tokens*)
         (window (open-exp-window "eyeCode Trial" :width 1600 :height 500)))

    ; Add all tokens to the visicon
    (loop for (x y width height text)
          (integer integer integer integer string)
          in tokens
         do (let ((static-text
                      (add-text-to-exp-window :text text :x x :y y
                                              :width width :height height)))
                    ; Use actual width and height of the token text
                    (setf (text-height static-text) 23)
                    (setf (str-width-fct static-text)
                            (lambda (str) (* 11 (length str))))))

    ; Add the continue button (ends the trial)
    (add-button-to-exp-window
      :text "Continue" :x 1115 :y 370
      :width 75 :height 25)

    ; Go, Johnny, Go, Go, Go
    (progn
      (install-device window)
      (proc-display)
      (print-visicon)
      (run *max-time*))))

; ---------------------------------------------------------------------------

(clear-all)

(define-model mr-bits
  ; Fixed random seed for reproducibility
  (sgp :seed (123456 0))

  ; Set ACT-R parameters
  (sgp :v t :needs-mouse nil :show-focus t :esc nil :lf 0.01 :trace-detail high
       :imaginal-delay 0 :act t :bll nil :rt -2
       :motor-feature-prep-time 0.001
       :motor-initiation-time 0.05
       :motor-burst-time 0.001
    )

  ; --------------------------------------------------------------------------
  ; DECLARATIVE MEMORY
  ; --------------------------------------------------------------------------

  ; Define chunk types
  (chunk-type model-state state token line-num response resp-idx resp-char
    last-x last-y trace-line trace-col trace-ctx trace-name trace-call trace-val
    trace-def orig-def last-ctx)

  ; Details of a line of code. A missing chunk for a line forces all tokens
  ; to be read.
  (chunk-type line-info line-num)

  ; Value or reference for a variable
  (chunk-type variable-info
              context      ; function name or "" for global
              name         ; variable name
              call-idx     ; index of current function call
              def-num      ; index of variable definition in context
              has-value    ; :yes if the variable's value is in memory,
                           ; :no otherwise
              ref-context  ; :empty or context of source variable
              ref-name     ; :empty or name of source variable
              ref-call     ; :empty or function call index of source variable
              ref-def      ; :empty or index of source variable definition
              val-line     ; :empty or line number of literal value
              val-col      ; :empty or column offset of literal value
  )

  ; Knowledge of binary sums, differences, products, etc.
  (chunk-type sum-result first second)
  (chunk-type diff-result first second)
  (chunk-type prod-result first second)
  (chunk-type and-result first second)
  (chunk-type less-than-result first second)
  (chunk-type greater-than-result first second)

  (chunk-type (trace-location (:include visual-location))
              context name trace-line trace-col)

  ; Add sums from -10 to 10
  (add-dm-fct
   (loop for i from -10 to 10
      append (loop for j from -10 to 10
                collect (list (intern (string-upcase
                                       (format nil "sum-~a-~a" i j)))
                              'isa 'sum-result
                              'first i
                              'second j))))

  ; Add subtractions from 0 to 10
  (add-dm-fct
   (loop for i from 0 to 10
      append (loop for j from 0 to 10
                collect (list (intern (string-upcase
                                       (format nil "diff-~a-~a" i j)))
                              'isa 'diff-result
                              'first i
                              'second j))))

  ; Add products from -10 to 20
  (add-dm-fct
   (loop for i from -10 to 20
      append (loop for j from -10 to 20
                collect (list (intern (string-upcase
                                       (format nil "prod-~a-~a" i j)))
                              'isa 'prod-result
                              'first i
                              'second j))))

  ; Add less-than comparisons from -10 to 10
  (add-dm-fct
   (loop for i from -10 to 10
      append (loop for j from -10 to 10
                collect (list (intern (string-upcase
                                       (format nil "less-than-~a-~a" i j)))
                              'isa 'less-than-result
                              'first i
                              'second j))))

  ; Add greater-than comparisons from -10 to 10
  (add-dm-fct
   (loop for i from -10 to 10
      append (loop for j from -10 to 10
                collect (list (intern (string-upcase
                                       (format nil "greater-than-~a-~a" i j)))
                              'isa 'greater-than-result
                              'first i
                              'second j))))

  ; Add binary ANDs
  (add-dm-fct
   (loop for a in '(t nil)
      for x = (if a 'True 'False)
      append (loop for b in '(t nil)
                for y = (if b 'True 'False)
                collect (list (intern (string-upcase
                                       (format nil "and-~a-~a" x y)))
                              'isa 'and-result
                              'first x
                              'second y))))

  ; Add visual locations of variables
  (add-dm-fct
    (list
      `(trace-value-4-8
            isa trace-location screen-x ,(col-to-x 8) screen-y ,(line-to-y 4)
            trace-line 4 trace-col 8)
      `(trace-value-4-15
            isa trace-location screen-x ,(col-to-x 15) screen-y ,(line-to-y 4)
            trace-line 4 trace-col 15)
      `(trace-value-4-22
            isa trace-location screen-x ,(col-to-x 22) screen-y ,(line-to-y 4)
            trace-line 4 trace-col 22)
    ))

  (add-dm
    (state isa chunk)

    ; Add chunks for static model states
    (start isa chunk)
    (finish-exp isa chunk)

    (find-next-token isa chunk)
    (search-for-token isa chunk)
    (attend-to-token isa chunk)
    (read-or-remember-line isa chunk)
    (retrieve-line-result isa chunk)

    (check-goal-stack isa chunk)

    ; Location of output box
    (output-box-location isa visual-location
      screen-x 1115 screen-y 80
      width 340 height 270
      color white)

    ; Add chunks for dynamic model states
    (link-to-value-f-x-L4-C8-0-1 isa chunk)
    (compute-prod-line-4-1-1 isa chunk)
    (compute-sum-line-2-2-1 isa chunk)
    (compute-prod-line-4-2-1 isa chunk)
    (compute-sum-line-2-1-1 isa chunk)
    (compute-sum-line-2-3-1 isa chunk)
    (do-read-line-2-f-x-3 isa chunk)

    (do-read-line-2-f-x-2 isa chunk)

    (do-read-line-2-f-x-1 isa chunk)

    (link-to-value-f-x-L4-C15-1-1 isa chunk)

    (link-to-value-f-x-L4-C22-2-1 isa chunk)


    ; Create initial goal

    (goal isa model-state state check-goal-stack line-num 1))


  ; ---------------------------------------------------------------------------

  ; PRODUCTIONS

  ; ---------------------------------------------------------------------------


  ; Try to remember details of line from memory

  (p read-or-remember-line

     =goal>

     isa          model-state

     state        read-or-remember-line

     line-num     =line-num

     ==>

     +retrieval>

     isa          line-info

     line-num     =line-num

     =goal>

     state        retrieve-line-result


     !output!     (retrieve line =line-num)

     )


  ; Line details remembered, continue on without reading
```

```
(p retrieve-line-success
   =goal>
   isa          model-state
   state        retrieve-line-result
   =retrieval>
   isa          line-info
   ==>
   =goal>
   state        check-goal-stack
   )

; Unable to remember line, read each token before continuing
(p retrieve-line-failed
   =goal>
   isa          model-state
   state        retrieve-line-result
   line-num     =line-num
   ?retrieval>
   state        error
   ==>
   =goal>
   state        find-next-token
   !output!     (Reading line =line-num)
   )

; ---------------------------------------------------------------------------

; Shift visual attention to the next visual token
(p shift-attention-to-line
   =goal>
   isa          model-state
   state        shift-attention-to-line
   =visual-location>
   isa          visual-location
   screen-x     =x
   screen-y     =y
   ?visual>
   state        free
   ==>
   +visual>
   isa          move-attention
   screen-pos   =visual-location
   =goal>
   state        attend-first-token
   last-x       =x
   last-y       =y
   )

; Attend to the first token on a line, try to remember
; the line's details.
(p attend-first-token
   =goal>
   isa          model-state
   state        attend-first-token
   =visual>
   isa          text
   value        =token
   ==>
   =goal>
   state        read-or-remember-line
   token        =token
   !output!     (read token =token)
   )

; Shift visual attention to the next visual token
(p search-for-token
   =goal>
   isa          model-state
   state        search-for-token
   =visual-location>
   isa          visual-location
   screen-x     =x
   screen-y     =y
   ?visual>
   state        free
   ==>
   +visual>
   isa          move-attention
   screen-pos   =visual-location
   =goal>
   state        attend-to-token
   last-x       =x
   last-y       =y
   )

; Extract the text from the attended token
(p attend-encoding-token
   =goal>
   isa          model-state
   state        attend-to-token
   =visual>
   isa          text
   value        =token
   ==>
   =goal>
   state        find-next-token
   token        =token
   !output!     (read token =token)
   )

; Look for the next token to the right
(p find-next-token
   =goal>
   isa          model-state
   state        find-next-token
   ==>
   +visual-location>
   isa          visual-location
   kind         text
   ; Don't attend to the continue button
   < screen-x   1115
   > screen-x   current
   screen-x     lowest
   screen-y     current
   =goal>
   state        search-for-token
   token        nil
   )

; Finished reading the current line, record the details
; in memory for later.
(p no-more-tokens-on-line
   =goal>
   isa          model-state
   state        search-for-token
   line-num     =line-num
   ?visual-location>
   state        error
   ==>
   =goal>
   state        check-goal-stack
   !output!     (Done reading line =line-num)
   )

; Commit the details of the current line to memory
(p remember-line
   =goal>
   isa          model-state
   state        remember-line
   line-num     =line-num
   ?imaginal>
   state        free
   ==>
   +imaginal>
   isa          line-info
   line-num     =line-num
   =goal>
   state        check-goal-stack
   !output!     (remembering line =line-num)
   )
```

```
; ---------------------------------------------------------------------------

; Execute the next goal on the stack (done if no goals left)
(p check-goal-stack
   =goal>
   isa          model-state
   state        check-goal-stack
   ==>
   =goal>
   state        =state2

   ; Crucial that we force imaginal to clear.
   ; Otherwise the only thing that will clear it is a new chunk.
   -imaginal>

   !bind!       =state2 (car *goal-stack*)
   !eval!       (setf *goal-stack* (cdr *goal-stack*))
   !output!     (Next goal is =state2)
   )

; No goals left - experiment is over
; Find the continue button and move hand to the mouse
(p find-continue-button
   =goal>
   isa          model-state
   state        nil
   ?manual>
   state        free
   ==>
   +visual-location>
   isa          visual-location
   kind         oval
   +manual>
   isa          hand-to-mouse
   =goal>
   state        click-continue-1
   )

; Look at the continue button
(p click-continue-1
   =goal>
   isa          model-state
   state        click-continue-1
   =visual-location>
   isa          visual-location
   screen-x     =x
   screen-y     =y
   ?visual>
   state        free
   ?manual>
   state        free
   ==>
   +visual>
   isa          move-attention
   screen-pos   =visual-location
   +manual>
   isa          move-cursor
   loc          =visual-location
   =goal>
   state        click-continue-2
   )

; Click the continue button
(p click-continue-2
   =goal>
   isa          model-state
   state        click-continue-2
   ?visual>
   state        free
   ?manual>
   state        free
   ==>
   +manual>
   isa          click-mouse
   =goal>
   state        done
   )


; Report amount of time taken to complete the experiment
(p experiment-is-over
   =goal>
   isa          model-state
   state        done
   ?manual>
   state        free
   ==>
   -goal>
   !bind!       =time (mp-time-ms)
   !output!     (Trial completed at =time)
   )

; ---------------------------------------------------------------------------

; Prepare to type a sequence of characters
(p start-typing-response
   =goal>
   isa          model-state
   state        type-response
   ==>
   =goal>
   state        typing-response
   response     =response
   resp-idx     0
   resp-char    =resp-char

   !bind!       =response (car *responses*)
   !bind!       =resp-char (actr-key (aref =response 0))
   !eval!       (setf *responses* (cdr *responses*))
   !output!     (Typing =response)
   )

; Type the next character
(p typing-response
   =goal>
   isa          model-state
   state        typing-response
   response     =response
   resp-idx     =resp-idx
   - resp-char   nil
   resp-char    =resp-char
   ?manual>
   state        free
   ==>
   +manual>
   isa          press-key
   key          =resp-char
   =goal>
   state        typing-response
   resp-idx     =next-idx
   resp-char    =next-char

   !bind!       =next-idx (+ 1 =resp-idx)
   !bind!       =next-char (if (< =next-idx (length =response))
                               (actr-key (aref =response =next-idx)) nil)
   !bind!       =time (mp-time-ms)
   !output!     (Typed =resp-char at =time)
   )

; No more characters to type
(p done-typing-response
   =goal>
   isa          model-state
   state        typing-response
   resp-char    nil
   ==>
   =goal>
   state        check-goal-stack
   )

; Look at the output box
(p fixate-output-box
   =goal>
   isa          model-state
   state        fixate-output-box
   ?visual>
   state        free
   ==>
   +visual>
   isa          move-attention
   screen-pos   output-box-location
   =goal>
   state        fixating-output-box
   )

; Wait for output box to be fixated
(p output-box-fixated
   =goal>
   isa          model-state
   state        fixating-output-box
   ?visual>
   state        free
   ==>
   =goal>
   state        check-goal-stack
   )

; Find the visual location that was being
; fixated before the output box.
(p return-from-output-box
   =goal>
   isa          model-state
   state        return-from-output-box
   last-x       =last-x
   last-y       =last-y
   ==>
   +visual-location>
   isa          visual-location
   screen-x     =last-x
   screen-y     =last-y
   =goal>
   state        return-from-output-box-move

   !output!     (Returning to =last-x =last-y)
   )

; Move eyes to the previous location
(p return-from-output-box-move
   =goal>
   isa          model-state
   state        return-from-output-box-move
   =visual-location>
   isa          visual-location
   screen-x     =x
   screen-y     =y
   ?visual>
   state        free
   ==>
   +visual>
   isa          move-attention
   screen-pos   =visual-location
   =goal>
   state        return-from-output-box-finish
   )

; Check for the next goal
(p return-from-output-box-finish
   =goal>
   isa          model-state
   state        return-from-output-box-finish
   ?visual>
   state        free
   ==>
   =goal>
   state        check-goal-stack
   )

; ----------------------------------------------------------------------------

;; An explicit value was not found for the variable.
;; Try to find a reference or location in declarative memory.
(p recalling-variable-no-value
   =goal>
   isa          model-state
   state        recalling-variable
   trace-ctx    =trace-ctx
   trace-name   =trace-name
   trace-call   =trace-call
   trace-def    =trace-def
   ?retrieval>
   state        error
   ==>
   +retrieval>
   isa          variable-info
   context      =trace-ctx
   name         =trace-name
   call-idx     =trace-call
   def-num      =trace-def
   has-value    :no
   =goal>
   state        recalling-variable-no-value

   !output!     (Recalling location or reference for =trace-name
                           in context =trace-ctx call =trace-call
                           def =trace-def)
   )

;; No location or reference was found for the variable.
;; Try to find a previous definition in the current context.
(p recalling-variable-no-ref-or-loc
   =goal>
   isa          model-state
   state        recalling-variable-no-value
   trace-ctx    =trace-ctx
   trace-name   =trace-name
   trace-call   =trace-call
   trace-def    =trace-def
   > trace-def  0
   ?retrieval>
   state        error
   ==>
   +retrieval>
   isa          variable-info
   context      =trace-ctx
   name         =trace-name
   call-idx     =trace-call
   def-num      =next-def-num
   has-value    :yes
   =goal>
   state        recalling-variable
   trace-def    =next-def-num

   ; Previous definition
   !bind!       =next-def-num (- =trace-def 1)
   !output!     (Trying again with =trace-name in context =trace-ctx
                        call =trace-call def =next-def-num)
   )

(p recalling-variable-change-context
   =goal>
   isa          model-state
   state        recalling-variable-no-value
   trace-ctx    =trace-ctx
   trace-name   =trace-name
   trace-ctx    =trace-ctx
   last-ctx     =last-ctx
   <= trace-def  0
   ?retrieval>
   state        error
   ==>
   +retrieval>
   isa          variable-info
   context      =last-ctx
   name         =trace-name
   has-value    :yes
   =goal>
   state        recalling-variable

   !output!     (Changing context for =trace-name from
                          =trace-ctx to =last-ctx)
   )

;; Got a reference to another variable (possibly in a different context).
;; Try to retrieve a value, reference, or location for *that* variable.
(p recalling-variable-got-reference
   =goal>
   isa          model-state
   state        recalling-variable-no-value
   =retrieval>
   isa          variable-info
   has-value    :no
   ref-context  =ref-context
   ref-name     =ref-name
   ref-call     =ref-call
   ref-def      =ref-def
   val-line     :empty
   call-idx     =call-idx
   ==>
```

```
  +retrieval>
  isa          variable-info
  context      =ref-context
  name         =ref-name
  call-idx     =ref-call
  def-num      =ref-def
  =goal>
  state        recalling-variable-no-value

  !output!     (Following variable reference to =ref-name in context
                          =ref-context call =ref-call def =ref-def)
  )

;; Got a location (line/column) for a variable's value.
;; Recall the associated visual location, and look there.
(p recalling-variable-got-location
  =goal>
  isa          model-state
  state        recalling-variable-no-value
  trace-val    =trace-val
  =retrieval>
  isa          variable-info
  has-value    :no
  def-num      =def-num
  ref-context  :empty
  val-line     =val-line
  val-col      =val-col
  ==>
  +retrieval>
  isa          trace-location
```

```
   trace-line   =val-line
   trace-col    =val-col
   =goal>
   state        trace-variable-remember
   trace-line   =val-line
   trace-col    =val-col
   trace-def    =def-num

   ; Slip an extra goal into the stack to write the
   ; variable's value after tracing.
   !eval!       (setf *goal-stack*
                      (append (list 'trace-variable-write) *goal-stack*))

   !output!     (Need to trace to line =val-line col =val-col
                      with write =trace-val)
   )

; Remembered the variable's trace location. Find it.
(p trace-variable-remember
   =goal>
   isa          model-state
   state        trace-variable-remember
   =retrieval>
   isa          trace-location
   trace-line   =trace-line
   trace-col    =trace-col
   ?visual>
   state        free
   ==>
   +visual-location>
   isa          visual-location
   :nearest     =retrieval
   =goal>
   state        trace-variable-move

   !output!     (Preparing trace to line =trace-line col =trace-col)
   )

; Move the eyes to the trace location
(p trace-variable-move
   =goal>
   isa          model-state
   state        trace-variable-move
   trace-line   =trace-line
   trace-col    =trace-col
   =visual-location>
   isa          visual-location
   ?visual>
   state        free
   ==>
   +visual>
   isa          move-attention
   screen-pos   =visual-location
   =goal>
   state        trace-variable-moving

   !output!     (Tracing to line =trace-line col =trace-col)
   )

; Start reading at the current location. Reading
; will proceed to the end of the line, and then
; the goal stack will be checked.
(p trace-variable-moving
   =goal>
   isa          model-state
   state        trace-variable-moving
   trace-line   =trace-line
   ?visual>
   state        free
   ==>
   =goal>
   state        find-next-token

   !output!     (Reading line =trace-line)
   )

; Write the value of the traced variable to memory.
; If it's remembered next time, the trace will be
; avoided.
(p trace-variable-write
   =goal>
   isa          model-state
   state        trace-variable-write
   trace-ctx    =trace-ctx
   trace-name   =trace-name
   trace-call   =trace-call
   trace-line   =trace-line
   trace-col    =trace-col
   trace-val    =trace-val
   orig-def     =orig-def
   ?imaginal>
   state        free
   ==>
   +imaginal>
   isa          variable-info
   context      =trace-ctx
   name         =trace-name
   call-idx     =trace-call
   def-num      =orig-def
   has-value    =trace-val
   ref-context  :empty
   ref-name     :empty
   ref-call     :empty
   ref-def      :empty
   val-line     =trace-line
   val-col      =trace-col
   =goal>
   state        check-goal-stack

   !output!     (Wrote traced variable =trace-ctx =trace-name call
                       =trace-call def =orig-def with value =trace-val)
   )

; --------------------------------------------------------------------------
; DYNAMIC PRODUCTIONS
; --------------------------------------------------------------------------


; Shift visual attention to a specific line
(p go-to-line-1
   =goal>
   isa          model-state
   state        go-to-line-1
   ==>
   +visual-location>
   isa          visual-location
   screen-y     =screen-y
   screen-x     lowest
   =goal>
   state        shift-attention-to-line
   line-num     1

   !bind!       =screen-y (line-to-y 1)
   )


; Shift visual attention to a specific line
(p go-to-line-2
   =goal>
   isa          model-state
   state        go-to-line-2
   ==>
   +visual-location>
   isa          visual-location
   screen-y     =screen-y
   screen-x     lowest
   =goal>
   state        shift-attention-to-line
   line-num     2
   !bind!       =screen-y (line-to-y 2)
   )



; Shift visual attention to a specific line
(p go-to-line-3
   =goal>
   isa          model-state
   state        go-to-line-3
   ==>
   +visual-location>
   isa          visual-location
   screen-y     =screen-y
   screen-x     lowest
   =goal>
   state        shift-attention-to-line
   line-num     3

   !bind!       =screen-y (line-to-y 3)
   )



; Shift visual attention to a specific line
(p go-to-line-4
   =goal>
   isa          model-state
   state        go-to-line-4
   ==>
   +visual-location>
   isa          visual-location
   screen-y      =screen-y
   screen-x      lowest
   =goal>
   state         shift-attention-to-line
   line-num      4

   !bind!        =screen-y (line-to-y 4)
   )



; Write the variable's value to memory by
; placing a chunk in the imaginal buffer and
; letting it get flushed.
(p link-to-value-f-x-L4-C8-0-1
   =goal>
   isa           model-state
   state         link-to-value-f-x-L4-C8-0-1
   ?imaginal>
   state         free
   ==>
   +imaginal>
   isa           variable-info
   context       "f"
   name          "x"
   call-idx      0
   def-num       0
   ref-context   :empty
   ref-name      :empty
   ref-call      :empty
   ref-def       :empty
   val-line     4
   val-col      8
   has-value    :no
   =goal>
   state        check-goal-stack
   !output!     (Wrote variable f x 0)
   )


; Try to recall the variable's value from memory.
; A retrieval failure may force a trace.
(p do-read-line-2-f-x-1-retrieve
   =goal>
   isa          model-state
   state        do-read-line-2-f-x-1
   ==>
   +retrieval>
   isa          variable-info
   context      "f"
   name         "x"
   call-idx     0
   def-num      0
   has-value    :yes
   =goal>
   state        recalling-variable
   trace-ctx    "f"
   trace-name   "x"
   trace-call   0
   trace-val    :yes
   trace-def    0
   orig-def     0
   last-ctx     ""

   !output!     (Recalling variable value for "x" in context "f"
                           call 0 def 0)
   )

; Successfully recalled the variable's value.
; Proceed with the next goal.
(p do-read-line-2-f-x-1-success
   =goal>
   isa          model-state
   state        recalling-variable
   =retrieval>
   isa          variable-info
   has-value    :yes
   ==>
   =goal>
   state        check-goal-stack
   )



; Recall the answer for a sum, product, etc.
; This should never fail.
(p compute-sum-line-2-1-1
   =goal>
   isa          model-state
   state        compute-sum-line-2-1-1
   ==>
   +retrieval>
   isa          sum-result
   first        1
   second       4
   =goal>
   state        compute-sum-line-2-1-1-done
   !output!     (Computing sum of 1 4)
   )

; Successfully remembered the answer.
; Proceed with the next goal.
(p compute-sum-line-2-1-1-done
   =goal>
   isa          model-state
   state        compute-sum-line-2-1-1-done
   =retrieval>
   isa          sum-result
   first        1
   second       4
   ==>
   =goal>
   state        check-goal-stack
   )


; Write the variable's value to memory by
; placing a chunk in the imaginal buffer and
; letting it get flushed.
(p link-to-value-f-x-L4-C15-1-1
   =goal>
   isa          model-state
   state        link-to-value-f-x-L4-C15-1-1
   ?imaginal>
   state        free
   ==>
   +imaginal>
   isa          variable-info
   context      "f"
   name         "x"
   call-idx     1
   def-num      0
   ref-context  :empty
   ref-name     :empty
   ref-call     :empty
   ref-def      :empty
   val-line     4
   val-col      15
   has-value    :no
   =goal>
   state        check-goal-stack
   !output!     (Wrote variable f x 1)
   )


; Try to recall the variable's value from memory.
; A retrieval failure may force a trace.
(p do-read-line-2-f-x-2-retrieve
   =goal>
   isa          model-state
   state        do-read-line-2-f-x-2
   ==>
   +retrieval>
   isa          variable-info
   context      "f"
   name         "x"
   call-idx     1
   def-num      0
   has-value    :yes
   =goal>
   state        recalling-variable
   trace-ctx    "f"
   trace-name   "x"
   trace-call   1
   trace-val    :yes
   trace-def    0
   orig-def     0
   last-ctx     ""

   !output!     (Recalling variable value for "x" in context "f"
                           call 1 def 0)
   )

; Successfully recalled the variable's value.
; Proceed with the next goal.
(p do-read-line-2-f-x-2-success
   =goal>
   isa          model-state
   state        recalling-variable
   =retrieval>
   isa          variable-info
   has-value    :yes
   ==>
   =goal>
   state        check-goal-stack
   )


; Recall the answer for a sum, product, etc.
; This should never fail.
(p compute-sum-line-2-2-1
   =goal>
   isa          model-state
   state        compute-sum-line-2-2-1
   ==>
   +retrieval>
   isa          sum-result
   first        0
   second       4
   =goal>
   state        compute-sum-line-2-2-1-done
   !output!     (Computing sum of 0 4)
   )

; Successfully remembered the answer.
; Proceed with the next goal.
(p compute-sum-line-2-2-1-done
   =goal>
   isa          model-state
   state        compute-sum-line-2-2-1-done
   =retrieval>
   isa          sum-result
   first        0
   second       4
   ==>
   =goal>
   state        check-goal-stack
   )


; Write the variable's value to memory by
; placing a chunk in the imaginal buffer and
; letting it get flushed.
(p link-to-value-f-x-L4-C22-2-1
   =goal>
   isa          model-state
   state        link-to-value-f-x-L4-C22-2-1
   ?imaginal>
   state        free
   ==>
   +imaginal>
   isa          variable-info
   context      "f"
   name         "x"
   call-idx     2
   def-num      0
   ref-context  :empty
   ref-name     :empty
   ref-call     :empty
   ref-def      :empty
   val-line     4
   val-col      22
   has-value    :no
   =goal>
   state        check-goal-stack
   !output!     (Wrote variable f x 2)
   )



; Try to recall the variable's value from memory.
; A retrieval failure may force a trace.
(p do-read-line-2-f-x-3-retrieve
   =goal>
   isa          model-state
   state        do-read-line-2-f-x-3
   ==>
   +retrieval>
   isa          variable-info
   context      "f"
   name         "x"
   call-idx     2
   def-num      0
   has-value    :yes
   =goal>
   state        recalling-variable
   trace-ctx    "f"
   trace-name   "x"
   trace-call   2
   trace-val    :yes
   trace-def    0
   orig-def     0
   last-ctx     ""
   !output!     (Recalling variable value for "x" in context "f"
                             call 2 def 0)
   )

; Successfully recalled the variable's value.
; Proceed with the next goal.
(p do-read-line-2-f-x-3-success
   =goal>
   isa          model-state
   state        recalling-variable
   =retrieval>
   isa          variable-info
   has-value    :yes
   ==>
   =goal>
   state        check-goal-stack
   )


; Recall the answer for a sum, product, etc.
; This should never fail.
(p compute-sum-line-2-3-1
   =goal>
   isa          model-state
   state        compute-sum-line-2-3-1
   ==>
   +retrieval>
   isa          sum-result
   first        -1
   second       4
   =goal>
   state        compute-sum-line-2-3-1-done
   !output!     (Computing sum of -1 4)
   )

; Successfully remembered the answer.
; Proceed with the next goal.
(p compute-sum-line-2-3-1-done
   =goal>
   isa          model-state
   state        compute-sum-line-2-3-1-done
   =retrieval>
   isa          sum-result
   first        -1
   second       4
   ==>
   =goal>
   state        check-goal-stack
   )



; Recall the answer for a sum, product, etc.
; This should never fail.
(p compute-prod-line-4-1-1
   =goal>
   isa          model-state
   state        compute-prod-line-4-1-1
   ==>
   +retrieval>
   isa          prod-result
   first        5
   second       4
   =goal>
   state        compute-prod-line-4-1-1-done
   !output!     (Computing prod of 5 4)
   )

; Successfully remembered the answer.
; Proceed with the next goal.
(p compute-prod-line-4-1-1-done
   =goal>
   isa          model-state
   state        compute-prod-line-4-1-1-done
   =retrieval>
   isa          prod-result
   first        5
   second       4
   ==>
   =goal>
   state        check-goal-stack
   )


; Recall the answer for a sum, product, etc.
; This should never fail.
(p compute-prod-line-4-2-1
   =goal>
   isa          model-state
   state        compute-prod-line-4-2-1
   ==>
   +retrieval>
   isa          prod-result
   first        20
   second       3
   =goal>
   state        compute-prod-line-4-2-1-done
   !output!     (Computing prod of 20 3)
   )

; Successfully remembered the answer.
; Proceed with the next goal.
(p compute-prod-line-4-2-1-done
   =goal>
   isa          model-state
   state        compute-prod-line-4-2-1-done
   =retrieval>
   isa          prod-result
   first        20
   second       3
   ==>
   =goal>
   state        check-goal-stack
   )


; ---------------------------------------------------------------------------

; All pre-existing DM facts (sums, etc.) are assumed to be well rehearsed
(set-all-base-levels 100000 -1000)
```

```
  (goal-focus goal)
)
```

E Appendix - Nibbles

E.1 Token Constraints

E.1.1 chars-to-token

Only applies if token is high detail.

Token Type	Constraints
Symbol	Token length is 1
	Character is one of `[ ] ( ) " , : = + *`
Keyword	Must be one of `print`, `def`, `for`, `in`, `return`
	Token lengths must match keyword lengths
Name	All characters are in `a-z _`
Number	All characters are in `0-9`
Whitespace	All characters are whitespace
Text	Does not match other token types
	At least 1 non-whitespace character
Unknown	Must be set via assertion

E.2 Line Constraints

E.2.1 tokens-are-list

True if a collection of tokens form a List literal. The conditions are:

1st and last tokens are `[` and `]` Symbols

Either exactly 2 tokens (empty list),

or more than 2 tokens such that the count of `,` Symbol(s) is 1 less than the count of Number(s) (numeric list)

E.2.2 tokens-to-lines

Only applies if line is high detail.

Line Type	Constraints
Unknown	All tokens are Unknown
Whitespace	All tokens are Whitespace RHS expression form is N/A
Print-Line	More than 1 token 1st token is `print` Keyword RHS expression form is **not** N/A
Assignment-Line	More than 2 tokens 1st token is a Name 2nd token is `=` Symbol 3rd token is **not** Whitespace RHS expression form is **not** N/A
For-Loop-Line	More than 3 tokens 1st token is `for` 2nd token is a Name 3rd token is `in` Keyword Last token is `:` Symbol *Either* 4th token is a Name, *or* all but last tokens are a **List**
Function-Call	More than 2 tokens 1st token is a Name 2nd token is `(` Symbol Last token is `)` Symbol *Either* exactly 3 tokens (no parameters), *or* exactly 4 tokens and 3rd token is a Name (1 parameter)
Function-Def-Line	More than 3 tokens 1st token is `def` Keyword

Line Type	Constraints *(continued)*
	2nd token is `(` Symbol
	2nd-to-last token is `)` Symbol
	Last token is `:` Symbol
	Either exactly 5 tokens (no parameters),
	or exactly 6 tokens and 4th token is a Name (1 parameter)
Return-Line	More than 1 token
	1st token is `return` Keyword
	RHS expression form is **not** N/A

E.2.3 rhs-expression-form

Only applies when RHS expression form is **not** N/A. Constraints apply only to RHS tokens.

Expression Form	Constraints
Literal	*Either* exactly 1 token that is a Number (numeric literal),
	or at least 2 tokens, 1st and last are " Symbols, and rest are Text (string literal)
Variable	Exactly 1 Name token
Literal + Literal	Exactly 3 tokens
	1st and last tokens are Numbers
	2nd token is + Symbol
Variable + Literal	Exactly 3 tokens
	2nd token is + Symbol
	Either 1st token is Name and 3rd token is Number,
	or 1st token is Number and 3rd token is Name
	Operator meaning is **plus**
Variable + Variable	Exactly 3 tokens
	1st and last tokens are Names
	2nd token is + Symbol
	Either operator meaning is **plus** if expression type is Number,
	or operator meaning is **append** if expression type is String

E.2.4 rhs-expression-pytype

Only applies when RHS expression form is **not** N/A. Constraints apply only to RHS tokens.

Expression PyType	Constraints
Number	*Either* exactly 1 Number token,
	or every Token is a Name, Number, or + Symbol
	All Name tokens have type Number
String	*Either* exactly 2 " Symbol tokens,
	or more than 2 tokens, 1st and last are " Symbols, and at least 1 Text token
	or every Token is a Name or + Symbol
	All Name tokens have type String
N/A	Set when RHS expression form is also N/A

E.2.5 whitespace-at-end

Ensures that all Whitespace tokens (if any) occur at the **end** of a line.

E.3 Line Group Constraints

E.3.1 lines-to-line-group

Only applies when line group detail is high.

Line Group Type	Constraints
Function	More than 1 line
	1st line is a Function-Def-Line
	2nd line is indented
For-Loop	More than 1 line
	1st line is a For-Loop-Line
	2nd line is indented
Generic	At least 1 line
	Not any other line type
	All lines are at the same indent level
Whitespace	All lines are Whitespace

E.3.2 all-lines-active

Ensures that every line is active in a single line group, and that lines n and $n + 1$ are either active in the same group or line $n + 1$'s group comes after.

E.3.3 line-group-indent

If lines n and $n + 1$ are in the same line group, they must either be at the same indent level or $n + 1$ must be indented relative to n.

References

[1] ACT-R Research Group. About ACT-R. `http://act-r.psy.cmu.edu/about/`, mar 2012.

[2] Amazon.com. Amazon Mechanical Turk. `https://www.mturk.com`, Jan 2013.

[3] John R. Anderson. *How can the human mind occur in the physical universe?*, volume 3. Oxford University Press, USA, 2007.

[4] Roman Bednarik. *Methods to analyze visual attention strategies: Applications in the studies of programming*. University of Joensuu, 2007.

[5] Roman Bednarik, N. Myller, E. Sutinen, and M. Tukiainen. Program visualization: Comparing eye-tracking patterns with comprehension summaries and performance. In *Proceedings of the 18th Annual Psychology of Programming Workshop*, pages 66–82, 2006.

[6] Roman Bednarik and Markku Tukiainen. Effects of display blurring on the behavior of novices and experts during program debugging. In *CHI'05 Extended abstracts on human factors in computing systems*, pages 1204–1207. ACM, 2005.

[7] Roman Bednarik and Markku Tukiainen. Temporal eye-tracking data: evolution of debugging strategies with multiple representations. In *Proceedings of the 2008 symposium on Eye tracking research & applications*, pages 99–102. ACM, 2008.

[8] T.J. Biggerstaff, B.G. Mitbander, and D.E. Webster. Program understanding and the concept assignment problem. *Communications of the ACM*, 37(5):72–82, 1994.

[9] Georg Brandl and Pygments contributers. Pygments. `http://www.pygments.org`, Sep 2014.

[10] Jean-Marie Burkhardt, Françoise Détienne, and Susan Wiedenbeck. Object-oriented program comprehension: Effect of expertise, task and phase. *Empirical Software Engineering*, 7(2):115–156, 2002.

[11] Raymond Buse and Wes Weimer. Learning a metric for code readability. *Software Engineering, IEEE Transactions on*, 36(4):546–558, 2010.

[12] Jerome R. Busemeyer and A. Diederich. *Cognitive modeling*. Sage Publications, Inc, 2010.

[13] Teresa Busjahn. Personal communication, 2014.

[14] Teresa Busjahn, Carsten Schulte, and Andreas Busjahn. Analysis of code reading to gain more insight in program comprehension. In *Proceedings of the 11th Koli Calling International Conference on Computing Education Research*, pages 1–9. ACM, 2011.

[15] Teresa Busjahn, Carsten Schulte, Bonita Sharif, Simon, Andrew Begel, Michael Hansen, Roman Bednarik, Paul Orlov, Petri Ihantola, Galina Shchekotova, and Maria Antropova. Eye tracking in computing education. In *Proceedings of the Tenth Annual Conference on International Computing Education Research*, ICER '14, pages 3–10, New York, NY, USA, 2014. ACM.

[16] Simon Cant, David Jeffery, and Brian Henderson-Sellers. A conceptual model of cognitive complexity of elements of the programming process. *Information and Software Technology*, 37(7):351–362, 1995.

[17] Roger HS Carpenter. *Movements of the eyes (2nd rev)*. Pion Limited, 1988.

[18] Reginald B. Charney. PyMetrics. `http://pymetrics.sourceforge.net/`, Jan 2013.

[19] William G. Chase and Herbert A. Simon. Perception in chess. *Cognitive Psychology*, 4(1):55 – 81, 1973.

[20] Laura Cowen, Linden Js Ball, and Judy Delin. An eye movement analysis of web page usability. In *People and Computers XVI-Memorable Yet Invisible*, pages 317–335. Springer, 2002.

[21] Filipe Cristino, Sebastiaan Mathôt, Jan Theeuwes, and Iain D Gilchrist. Scanmatch: A novel method for comparing fixation sequences. *Behavior Research Methods*, 42(3):692–700, 2010.

[22] Martha E Crosby, Jean Scholtz, and Susan Wiedenbeck. The roles beacons play in comprehension for novice and expert programmers. In *14th Workshop of the Psychology of Programming Interest Group*, pages 58–73, 2002.

[23] Bob Curtis. Fifteen years of psychology in software engineering: Individual differences and cognitive science. In *Proceedings of the 7th international conference on Software engineering*, pages 97–106. IEEE Press, 1984.

[24] David P Darcy and Chris F Kemerer. Software complexity: Toward a unified theory of coupling and cohesion. In *Friday Workshops, Management Information Systems Research Center, Carlson School of Management, University of Minnesota*, 2002.

[25] Adrian de Groot, Fernand Gobet, and Riekent Jongman. *Perception and memory in chess: Studies in the heuristics of the professional eye*. Van Gorcum & Co, Assen, Netherlands, 1996.

[26] Françoise Détienne. *La compréhension de programmes informatiques par l'expert: un modéle en termes de schémas*. PhD thesis, Université Paris V. Sciences humaines, 1986.

[27] Françoise Détienne. Cognitive ergonomics. chapter Program Understanding and Knowledge Organization: The Influence of Acquired Schemata, pages 245–256. Academic Press Professional, Inc., San Diego, CA, USA, 1990.

[28] Françoise Détienne. What model (s) for program understanding? In *Proceedings of the Conference on Using Complex Information Systems*, Poitiers, France, 1996.

[29] Françoise Détienne and Frank Bott. *Software design–cognitive aspects*. Springer Verlag, 2002.

[30] Christopher Douce. The stores model of code cognition. In *Psychology of Programming Interest Group*, 2008.

[31] Christopher Douce. The stores model of code cognition. 2008.

[32] Christopher Douce, Paul J. Layzell, and Jim Buckley. Spatial measures of software complexity. 1999.

[33] Scott A Douglass and Saurabh Mittal. A framework for modeling and simulation of the artificial. In *Ontology, Epistemology, and Teleology for Modeling and Simulation*, pages 271–317. Springer, 2013.

[34] Bradley Efron, Trevor Hastie, Iain Johnstone, Robert Tibshirani, et al. Least angle regression. *The Annals of statistics*, 32(2):407–499, 2004.

[35] B.D. Ehret. Learning where to look: Location learning in graphical user interfaces. In *Proceedings of the SIGCHI conference on Human factors in computing systems: Changing our world, changing ourselves*, pages 211–218. ACM, 2002.

[36] Khaled El-Emam. Object-oriented metrics: A review of theory and practice. national research council canada. *Institute for Information Technology*, 2001.

[37] Khaled El-Emam, Saida Benlarbi, and Nishith Goel. The confounding effect of class size on the validity ofobject-oriented metrics. *Software Engineering, IEEE Transactions on*, 27:630–650, 1999.

[38] Tom Fawcett. An introduction to ROC analysis. *Pattern recognition letters*, 27(8):861–874, 2006.

[39] Norman E. Fenton and Martin Neil. A critique of software defect prediction models. *Software Engineering, IEEE Transactions on*, 25(5):675–689, 1999.

[40] Norman E. Fenton and Martin Neil. Software metrics: roadmap. *Proceedings of the Conference on The Future of Software Engineering*, pages 357–370, 05 2000.

[41] Paul M Fitts and James R Peterson. Information capacity of discrete motor responses. *Journal of experimental psychology*, 67(2):103, 1964.

[42] DJ Gilmore and TRG Green. The comprehensibility of programming notations. In *Human-Computer Interaction-Interact*, volume 84, pages 461–464, 1985.

[43] Robert L. Goldstone, David H. Landy, and Ji Y. Son. The education of perception. *Topics in Cognitive Science*, 2(2):265–284, 2010.

[44] Mark Guzdial. From science to engineering. *Commun. ACM*, 54(2):37–39, February 2011.

[45] Maurice H. Halstead. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., 1977.

[46] Michael Hansen, Robert L. Goldstone, and Andrew Lumsdaine. What Makes Code Hard to Understand? *ArXiv e-prints*, April 2013.

[47] Taylor R. Hayes, Alexander A. Petrov, and Per B. Sederberg. A novel method for analyzing sequential eye movements reveals strategic influence on raven's advanced progressive matrices. *Journal of vision*, 11(10), 2011.

[48] Abram Hindle, Michael W. Godfrey, and Richard C. Holt. Reading beside the lines: Indentation as a proxy for complexity metric. In *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*, pages 133–142. IEEE, 2008.

[49] Ignace Hooge and Guido Camps. Scan path entropy and arrow plots: capturing scanning behavior of multiple observers. *Frontiers in psychology*, 4, 2013.

[50] Philip N Johnson-Laird. *Mental models*. Number 6. Harvard University Press, 1986.

[51] Eric Jones, Travis Oliphant, Pearu Peterson, and Others. SciPy: Open source scientific tools for Python. `http://www.scipy.org/`, 2001–. [Online; accessed 2014-10-10].

[52] Sonya E Keene, Dan Gerson, and David A Moon. *Object-oriented programming in Common Lisp: A programmer's guide to CLOS*, volume 8. Addison-Wesley Reading, Massachusetts, 1989.

[53] David E. Kieras and David E. Meyer. An overview of the EPIC architecture for cognition and performance with application to human-computer interaction. *Hum.-Comput. Interact.*, 12(4):391–438, December 1997.

[54] Walter Kintsch and Teun A Van Dijk. Toward a model of text comprehension and production. *Psychological review*, 85(5):363, 1978.

[55] Tuomas Klemola. A cognitive model for complexity metrics. In *4th International ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering*, 2000.

[56] A.J. Ko, B.A. Myers, M.J. Coblenz, and H.H. Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *Software Engineering, IEEE Transactions on*, 32(12):971–987, 2006.

[57] Gordon E Legge, Thomas A Hooven, Timothy S Klitz, J Stephen Mansfield, and Bosco S Tjan. Mr. chips 2002: New insights from an ideal-observer model of reading. *Vision research*, 42(18):2219–2234, 2002.

[58] Richard E Mayer. Cognitive aspects of learning and using a programming language. 1987.

[59] Thomas J. McCabe. A complexity measure. *Software Engineering, IEEE Transactions on*, (4):308–320, 1976.

[60] Wes McKinney. Data structures for statistical computing in python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 51–56, 2010.

[61] Tim Menzies. Applications of abduction: knowledge-level modelling. *International Journal of Human Computer Studies*, 45(3):305–336, 1996.

[62] Michael Hansen. The eyeCode Python library. `https://github.com/synesthesiam/eyecode-tools`, May 2015.

[63] George A. Miller. The magical number seven, plus or minus two: some limits on our capacity for processing information. *Psychological review*, 63(2):81, 1956.

[64] NDepend. NDepend Code Metrics Definitions. `http://www.ndepend.com/metrics.aspx`, Jan 2013.

[65] Janni Nielsen, Torkil Clemmensen, and Carsten Yssing. Getting access to what goes on in people's heads?: reflections on the think-aloud technique. In *Proceedings of the second Nordic conference on Human-computer interaction*, pages 101–110. ACM, 2002.

[66] Marcus Nyström, Richard Andersson, Kenneth Holmqvist, and Joost van der Wijer. The influence of calibration method and eye physiology on eyetracking data quality. *Behavior research methods*, 45(1):262–288, 2013.

[67] A Olsen. The tobii i-vt fixation filter: Algorithm description. *Tobii I-VT Fixation Filter–whitepaper [White paper]*, 2012.

[68] Chris Parnin. A cognitive neuroscience perspective on memory for programming tasks. In *In the Proceedings of the 22nd Annual Meeting of the Psychology of Programming Interest Group (PPIG)*. Citeseer, 2010.

[69] Fabian Pedregosa et al. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[70] Nancy Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive psychology*, 19(3):295–341, 1987.

[71] Alex Poole and Linden J. Ball. Eye tracking in HCI and usability research. *Encyclopedia of Human-Computer Interaction, C. Ghaoui (ed.)*, 2006.

[72] Keith Rayner. Eye movements in reading and information processing: 20 years of research. *Psychological bulletin*, 124(3):372, 1998.

[73] Robert S. Rist. Schema creation in programming. *Cognitive Science*, 13(3):389–414, 1989.

[74] Jorma Sajaniemi and Raquel Navarro Prieto. Roles of variables in experts programming knowledge. In *Proceedings of the 17th Annual Workshop of the Psychology of Programming Interest Group (PPIG 2005)*, pages 145–159. Citeseer, 2005.

[75] Dario D. Salvucci. *Mapping eye movements to cognitive processes*. PhD thesis, Carnegie Mellon University, 1999.

[76] Dario D Salvucci. A model of eye movements and visual attention. In *Proceedings of the International Conference on Cognitive Modeling*, pages 252–259, 2000.

[77] Dario D. Salvucci. Predicting the effects of in-car interface use on driver performance: An integrated model approach. *International Journal of Human-Computer Studies*, 55(1):85–107, 2001.

[78] Dario D. Salvucci and N.A. Taatgen. Threaded cognition: An integrated theory of concurrent multitasking. *Psychological Review*, 115(1):101, 2008.

[79] David Sankoff and Joseph B. Kruskal. Time warps, string edits, and macromolecules: the theory and practice of sequence comparison. *Reading: Addison-Wesley Publication, 1983, edited by Sankoff, David; Kruskal, Joseph B.*, 1, 1983.

[80] Roger C. Schank and Robert P. Abelson. Goals and understanding. *Erlbanum: Eksevier Science*, 1977.

[81] B. Schneiderman. Interactive interface issues. *Software Psychology: Human Factors in Computer and Information Systems*, pages 216–251, 1980.

[82] Ben Schneiderman and Richard Mayer. Towards a cognitive model of programmer behavior. Technical report, Indiana University, aug 1975.

[83] J.S. Seabold and J. Perktold. Statsmodels: Economtric and statistical modeling with pythong. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, 2010.

[84] Kshitij Sharma, Patrick Jermann, Marc-Antoine Nüssli, and Pierre Dillenbourg. Gaze evidence for different activities in program understanding. In *24th Psychology of Programming Workshop*, 2012.

[85] S.B. Sheppard, Bob Curtis, P. Milliman, MA Borst, and T. Love. First-year results from a research program on human factors in software engineering. In *Proceedings of the National Computer Conference*, page 1021. IEEE Computer Society, 1979.

[86] Sidney Siegel and N. John Castellan. *Nonparametric statistics for the behavioral sciences*. McGraw–Hill, Inc., second edition, 1988.

[87] Janet Siegmund, André Brechmann, Sven Apel, Christian Kästner, Jörg Liebig, Thomas Leich, and Gunter Saake. Toward measuring program comprehension with functional magnetic resonance imaging. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 24. ACM, 2012.

[88] Jeffrey Mark Siskind and David Allen McAllester. Screamer: A portable efficient implementation of nondeterministic common lisp. *IRCS Technical Reports Series*, page 14, 1993.

[89] Elliot Soloway and Kate Ehrlich. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, (5):595–609, 1984.

[90] Guy L Steele. *Common LISP: the language*. Digital press, 1990.

[91] Andreas Stefik and Susanna Siebert. An empirical investigation into programming language syntax. *ACM Transactions on Computing Education (TOCE)*, 13(4):19, 2013.

[92] N.A. Taatgen. Dispelling the magic: Towards memory without capacity. *Behavioral and Brain Sciences*, 24(01):147–148, 2001.

[93] N.A. Taatgen, John R. Anderson, et al. Why do children learn to say "broke"? A model of learning the past tense without feedback. *Cognition*, 86(2):123–155, 2004.

[94] The PHP Group. Php: Type juggling. `http://php.net/manual/en/language.types.type-juggling.php`, may 2015.

[95] Tobii Eye Tracking Research. Tobii Studio 3.2 User Manual. `http://www.tobii.com/Global/Analysis/Downloads/User_Manuals_and_Guides/Tobii_UserManual_TobiiStudio3.2_301112_ENG_WEB.pdf`, Jun 2014.

[96] W.J. Tracz. Computer programming and the human thought process. *Software: Practice and Experience*, 9(2):127–137, 1979.

[97] Hidetake Uwano, Masahide Nakamura, Akito Monden, and Ken-ichi Matsumoto. Analyzing individual performance of source code review using reviewers' eye movement. In *Proceedings of the 2006 symposium on Eye tracking research & applications*, pages 133–140. ACM, 2006.

[98] Todd L. Veldhuizen. Parsimony principles for software components and metalanguages. *Proceedings of the 6th international conference on Generative programming and component engineering*, pages 115–122, 10 2007.

[99] Adrian Veßkühler. OGAMA. `http://www.ogama.net`, Sep 2014.

[100] Michael M Vitousek, Andrew M Kent, Jeremy G Siek, and Jim Baker. Design and evaluation of gradual typing for python. In *Proceedings of the 10th ACM Symposium on Dynamic languages*, pages 45–56. ACM, 2014.

[101] A. Von Mayrhauser and A.M. Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55, 1995.

[102] L. Weissman. Psychological complexity of computer programs: an experimental methodology. *ACM Sigplan Notices*, 9(6):25–36, 1974.

[103] Hans W. Wendt. Dealing with a common problem in social science: A simplified rank-biserial coefficient of correlation based on the u statistic. *European Journal of Social Psychology*, 2(4):463–465, 1972.

[104] E.J. Weyuker. Evaluating software complexity measures. *IEEE Transactions on Software Engineering*, 14:1357–1365, 1988.

[105] Simon White and D Sleeman. Constraint handling in common lisp. *Department of Computing Science Technical Report AUCS/TR9805, University of Aberdeen, Aberdeen, UK*, 1998.

[106] Susan Wiedenbeck. Beacons in computer program comprehension. *International Journal of Man-Machine Studies*, 25(6):697 – 709, 1986.

[107] Tsunhin John Wong, Edward T Cokely, and Lael J Schooler. An online database of act-r parameters: Towards a transparent community-based approach to model development. In *Proceedings of the Tenth International Conference on Cognitive Modeling, Philadelphia, PA, USA*, pages 282–286. Citeseer, 2010.

[108] Bernard P. Zeigler and Phillip E. Hammonds. *Modeling & Simulation-Based Data Engineering: Introducing Pragmatics into Ontologies for Net-Centric Information Exchange*. Academic Press, 2007.

Michael Hansen

www.synesthesiam.com

mihansen@indiana.edu

Supervisory Control and Cognition Branch (RHCI)

711th Human Performance Wing

Air Force Research Lab

Education

- **Ph.D. in Computer Science and Cognitive Science** (November 2015)
  - Indiana University (IU)
- **M.S. in Computer Science** (2012)
  - Indiana University (IU)
- **B.S. in Computer Science** (2006)
  - University of Wyoming (UW)

Work Experience

- **Computer Scientist** for the Air Force Research Lab (Mar 2014 - present)
  - Working on agent frameworks, autonomy, and intelligence/surveillance/reconnaissance.
- **Contract programmer and Repperger intern** for the Air Force Research Lab (Sep 2012 - Mar 2014)
  - Developed cognitive modeling enhancements to agents in a simulated task environment.
  - Simulated cognitive agents using a distributed networking infrastructure.
  - Transitioned the CECEP cognitively-enhanced complex event processing architecture to a net-centric implementation using ZeroMQ.
  - Designed and implemented a qualitative spatial array capability with ego-centric visibility polygons.

  - Assisted in design of an intelligence/surveillance/reconnaissance (ISR) agent.

- **Research assistant** for CREST at IU (Sept 2009 - Dec 2013)
  - Created computational cognitive model for results of eye-tracking experiment.
  - Designed and ran program comprehension experiment using a Tobii TX300 eye-tracker for local participants and Mechanical Turk for remote participants.
  - Wrote software for processing and rendering plenoptic lightfields using GPU shaders (C#, DirectX).
  - Implemented and optimized algorithms for Schrieber's Transfer Entropy measure, available in the Transfer Entropy Toolbox (MATLAB/C, C++).
  - Implemented algorithms in the Boost Graph Library for McGregor common subgraphs and multi-dimensional grid-graphs (C++).

- **Associate instructor** for Advanced Operating Systems - CSCI-P 536 at IU at IU (Fall 2012)
  - Graded coding assignments and conducted one-on-one student code reviews.
  - Taught weekly lab section and several lectures.

- **Repperger intern** for the Air Force Research Lab (Jun - Aug 2012)
  - Designed and implemented a cognitively-enhanced complex event processing infrastructure using Esper and Scala.
  - Created agents for a checkpoint scenario using Unreal Tournament and Google Maps.
  - Assisted in the design of meta-models for the graphical development of behavioral models in the Generic Modeling Environment.

- **Contract programmer** for Quartermain Inc. (Jan 2006 - Dec 2013)
  - Implemented and maintained the ExcelCube spreadsheet consolidation desktop application (C#, Windows Forms, see link for details).

- **Student programmer** for the Percepts and Concepts Lab at IU (Sept 2008 - Sept 2009)
  - Designed and implemented several cross-platform research games (C#, Mono Framework, OpenGL). See personal web site for details.

- **Contract programmer** for Logical Information Machines (Aug 2007 - Aug 2008)
  - Designed and implemented a desktop application for querying and visualizing stock-market data from an in-house time-series database (C#, Windows Presentation Foundation).

- **Contract programmer** for HappyJack Software LLC (Jan 2007 - Aug 2008)
  - Designed and implemented a student records web management system for the UW School of Nursing (C# ASP.NET, MySQL, 100's of students, 10's of users)

    - Implemented a two-way synchronization plug-in for Microsoft Outlook and the web-based Kalendi product (C$^{\#}$, VB.NET, SyncML)

- **Co-founder and lead programmer** for chapaCode Inc. (Jan 2003 - Aug 2007)
    - Designed, implemented, and maintained web-based student management system for UW College of Education (C$^{\#}$, ASP.NET, SQL Server, 100's of students, 10's of users)
    - Implemented database and reporting website for The Center for Performance Assessment and the state of Nevada (C$^{\#}$, ASP.NET, Sqlite)
    - Designed, implemented, and maintained legal records and reporting system for the Laramie, WY City Attorney's office (C$^{\#}$, Windows Forms, SQL Server, Microsoft Word)
- **Student programmer** for multiple UW departments (Jan 2002 - Dec 2005)
    - **Mechanical Engineering** (2004-2005): Implemented CALISYS program (see Publications).
    - **Student Educational Opportunities** (2004-2005): Administered student database, automated tasks and reports for staff.
    - **Admissions** (2003-2004): Administered database and automated tasks for staff (e.g. detecting duplicate students, assigning e-mail addresses).
    - **Computer Science** (2002-2003): Created utility programs for lab assistants to access Novell Directory Services.

Honors and Organizations

- **Software Carpentry Bootcamp Instructor** (2012-present)
    - Instructor for Software Carpentry bootcamps at Indiana University, Howard Hughes Medical Institute, and Purdue University.
- **Nominated for UW Student Employee of the Year** (2006)
    - Nominated for my work on the Computer-Aided Laboratory Instruction System (CALISYS) project, a virtual lab environment similar to LabView for students to collect, manipulate, and visualize real-time measurement data (C$^{\#}$, Windows Presentation Foundation).
- **Microsoft Most Valuable Professional in Visual C$^{\#}$** (2004-2005)
    - Received for my work with the Wyoming ACM chapter as President and activity organizer.
- **4th place regional winner for Microsoft Imagine Cup** (2004)
    - Received for the ShopNET application, which provided a 3-D multi-user environment for purchasing books from Amazon. Users inhabited a virtual bookstore that was populated with

real products using Amazon Web Services. The application was written in C# and used a custom OpenGL engine which was compatible with Quake 3 maps and models.

- President of the Wyoming Association of Computing Machinery chapter (2003-2004).

Core Courses

- **Principles of Programming Languages** (CSCI-B 521)
  - Daniel Friedman: Fall 2009
- **Advanced Operating Systems** (CSCI-P 536)
  - Andrew Lumsdaine: Fall 2009
- **Introduction to Bayesian Data Analysis** (PSY-P 533/534)
  - John Kruschke: Fall 2009/Spring 2010
- **Natural Language Processing** (CSCI-B 651)
  - Michael Gasser: Fall 2010
- **Integrating Static and Dynamic Typing** (CSCI-B 629)
  - Amal Ahmed: Fall 2010
- **Networks of the Brain** (COGS-Q 610)
  - Olaf Sporns: Spring 2011