Can We Crowdsource Language Design?

Preston Tunnell Wilson[*]
Brown University
ptwilson@brown.edu

Justin Pombrio
Brown University
justinpombrio@cs.brown.edu

Shriram Krishnamurthi
Brown University
sk@cs.brown.edu

Abstract

Most programming languages have been designed by committees or individuals. What happens if, instead, we throw open the design process and let lots of programmers weigh in on semantic choices? Will they avoid well-known mistakes like dynamic scope? What do they expect of aliasing? What kind of overloading behavior will they choose?

We investigate this issue by posing questions to programmers on Amazon Mechanical Turk. We examine several language features, in each case using multiple-choice questions to explore programmer preferences. We check the responses for consensus (agreement between people) and consistency (agreement across responses from one person). In general we find low consistency and consensus, potential confusion over mainstream features, and arguably poor design choices. In short, this preliminary evidence does not argue in favor of designing languages based on programmer preference.

CCS Concepts **•Software and its engineering → General programming languages; •Social and professional topics →** *History of programming languages;*

Keywords crowdsourcing, language design, misconceptions, user studies

ACM Reference format:
Preston Tunnell Wilson, Justin Pombrio, and Shriram Krishnamurthi. 2017. Can We Crowdsource Language Design?. In *Proceedings of Onward, Vancouver, Canada, October 22–27, 2017 (Onward! '17),* 17 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 Introduction

Programming languages are clearly user interfaces: they are how a programmer communicates their desires to the computer. Programming language design is therefore a form of user interface design.

There are many traditions in interface design, and in design in general. One divisor between these traditions is how design decisions are made. Sometimes, decisions are made by a small number of opinionated designers (think Apple). These have parallels in programming language design, from individual designers to small committees, and this is even codified in (unofficial) titles likes Benevolent Dictator for Life. (Community input processes are clearly a hybrid, but at best they only suggest changes, which must then be approved by "the designers".)

There are fewer examples of language design conducted through extensive user studies and user input, though there are a few noteworthy examples that we discuss in section 11. None of these addresses comprehensive, general-purpose languages. Furthermore, many of these results focus on *syntax*, but relatively little on the *semantics*, which is at least as important as syntax, even for beginners [11, 31].

In this paper, we assess the feasibility of designing a language to match the expectations and desires of programmers. Concretely, we pose a series of questions on Amazon Mechanical Turk (MTurk) to people with programming experience to explore the kinds of behaviors programmers would want to see. Our hope is to find one or both of:

Consistency For related questions, individuals answer the same way.
Consensus Across individuals, we find similar answers.

Neither one strictly implies the other. Each individual could be internally consistent, but different people may wildly disagree on what they expect. At the other extreme, people might not be internally consistent at all, but everyone may agree in their (inconsistent) expectations.

Both properties have consequences for language design. If people generate consensus without consistency, we can still fit a language to their desires, though the language may be unpredictable in surprising ways. On the other hand, if people are internally consistent even though they don't agree with each other, we could imagine creating "personalized" languages (using a mechanism like Racket's `#lang` [9]), though the resulting languages would be confusing to people who don't share their views. (This is already slightly the case: e.g., numerous scripting languages are superficially similar but have many subtle semantic differences.) In an ideal world, people are both consistent and arrive at a consensus, in which case we could fit a language to their views and the language would likely have predictable behavior.

As Betteridge's Law implies [1], we do not live in that world. Indeed, what we find is that in general, programmers exhibit *neither consistency nor consensus.* Naturally, however, this paper is only an initial salvo and the topic requires much more exploration.

Methodology We conducted surveys on MTurk, each focusing on a language feature. Most of these surveys were

[*]Last name is "Tunnell Wilson" (index under "T").

Onward! '17, Vancouver, Canada
2017. 978-x-xxxx-xxxx-x/YY/MM…$15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

open for a week; we kept a few open longer to get enough responses. We focused on mainstream features found in most languages. The tasks asked for previous programming experience, sometimes experience with that specific feature (like object-oriented programming), and asked people to report on their programming experience. Those who did not list any experience were eliminated from the data analysis (as were duplicate respondents). We considered their responses invalid and all others valid.

For each feature we asked respondents—called *workers* or *Turkers*—to tell us what they thought "a NEW programming language would produce". We then presented a series of programs with multiple choices for the *expected* output of each one. (Each question also asked if they would *want* a different answer, but workers rarely selected this option, so we don't discuss it further.) For the programs, we purposely used a syntax that was reminiscent of existing languages, but not identical to any particular one. Every question also allowed "Error" and "Other". All respondents were required to give their reasoning behind their chosen answer; those who picked Error and Other were also asked to state their expectation. In the data analysis, for simplicity we binned all Error answers together, and likewise all Other responses (even though this makes our measures look higher than they really are).

Our discussion will also be presented by language feature.

2 Structure of the Data Analysis

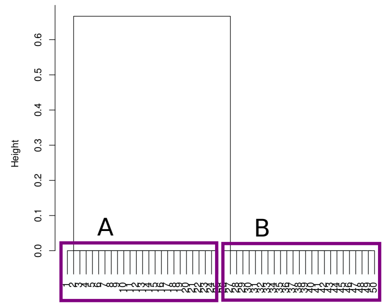
In this section we describe our methods and how we will organize the results. All the questions in one section (with one exception: section 8.2) were answered by the same set of workers, but no attempt was made to solicit the same workers across sections. The final paper will be accompanied by a link to the full questions and data sets.

2.1 Comparisons

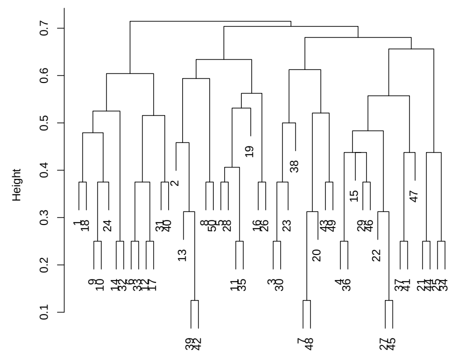
Within each section, we will compute *consensus* and *consistency* for small groups of related questions. We furthermore visualize these measurements as well as the spread of data for most question comparisons.

Summary Tables When comparing answers to just two questions, we will summarize the results in a table. However, for comparisons between three (or more) questions, instead of showing an unwieldy n-dimensional table, we show a cluster tree [16], as described below.

Cluster Trees As an illustrative example of clustering, suppose that there were three questions, and workers fell into two camps, with the first camp always answering "1, 2, 3", and the second camp answering "1, 4, 5". This produces the following cluster tree:



The height of the point at which subtrees join gives the *average distance* between the responses found in the subtrees, where the distance between two responses is the fraction of answers on which they differ (i.e., normalized Hamming distance [13]). In this case, within each cluster the responses do not differ at all, so each cluster is joined at height 0. Between the two clusters, responses always differ in 2 out of 3 answers, so they join at height 2/3. (We have been unable to get our package to always show the full y-axis, i.e., to height 1; sorry.) On the other hand, say everyone chose their answers uniformly at random; the cluster tree might be:[1]



The cluster trees we find for our programming surveys show weak clustering, but there are two incidental factors that may contribute to this. First, these trees are sensitive to the number of questions. When there are few questions (some of ours had fewer than 5), the height exaggerates disagreement. Second, we purposely asked questions with the potential for disagreement: there are plenty of straightforward programs whose behavior is uncontroversial, but they wouldn't make useful survey questions for that very reason.

We give a more technical description of our cluster trees in appendix A.2.

Consensus *Consensus* is how much workers give the same answers as each other. We measure it in two ways. First, we give the *simple agreement* probability $\bar{p}$. This is the probability that two random workers answered a random question the same way as each other. However, it is well understood [6] that $\bar{p}$ overstates agreement, because it does not account for the fact that some agreement may be due to

[1] More precisely, we plotted answers chosen uniformly at random to 8 questions, each with 3 possible answers. These numbers were chosen to be representative of our surveys.

chance. It is therefore standard to use an *inter-coder reliability* metric that accounts for chance. We thus also report a metric called *free-marginal* κ [30], hereafter simply κ. This is the level of agreement *above chance*, where "chance" is calculated assuming that for each question, each answer is equally likely to be chosen. κ scores range from −1 to 1, with negative scores representing less agreement than would be expected by chance, 0 representing plain chance, and 1 representing perfect agreement.

The assumption that all answers are equi-probable is unrealistic, so κ scores may show *higher* agreement than they should (though never lower). Unfortunately, we are unaware of a more appropriate metric of agreement. The notable alternative is Fleiss' κ score [10] (a generalization of Cohen's κ). Fleiss' κ is not applicable for many of our uses, however, because it requires that each question have the same set of possible answers. Fortunately, we will typically be highlighting how *low* the agreement scores are, so free-marginal κ is the conservative direction to err in. We discuss our choice of agreement measures further in appendix A.1.

Consistency Consensus measured how often workers answered a question the same way as each other. *Consistency*, on the other hand, measures how often workers gave *what we consider* to be consistent answers to a set of questions. For example, we asked what the programs `"5" + 2` and `2 + "5"` should produce, and we judged that the answers ought to be the same, even though in some languages they are different.

Similar to consensus, we report for consistency both $\bar{p}$ and (free-marginal) κ. When comparing answers to two questions, $\bar{p}$ for consistency will typically be the sum along the diagonal. In addition, when we provide a consistency score, we will always describe what we took to be consistent responses. Unlike with consensus, which is a pure measurement, here there is room for readers to disagree with our choices; other choices could result in different scores.

2.2 Overviews

Beyond looking at consensus and consistency for small sets of questions, there is also information to be gleaned by looking at workers' overall responses to a survey. Often, clusters of workers can be found who appear to have similar expectations. To explore this, we perform hierarchical clustering analysis on the responses, plot the results as a tree, and label some of the clusters we have found.

Additionally, sometimes Turkers' expectations are at odds with well-established language behaviors such as lexical scope. When they are, their expectations can be classified as *misconceptions*. We will call out these cases. There is a long history of literature on misconceptions that *students* encounter when programming (Sorva [32, pg. 29] provides a detailed summary). We confirm some known misconceptions here, and discover some new and unusual programmer expectations.

With this explanation, we can now begin our study of various language features.

3 Binding and Scope

We surveyed Turkers on their expectations of scope and parameter passing. We obtained 56 valid responses.

3.1 Dynamic Scope

We begin by examining the following two programs and comparing them. Both access identifiers outside their static scope, but in slightly different ways. In a statically-scoped language, both should produce an error.

```
func f():
    a = 14
func g():
    a = 12

f()
g()
print(a)
```

```
func f():
    d = 32
    return g()

func g():
    return d + 3

print(f())
```

	print(f())				
print(a)	3	32	35	Error	Other
12	0.0%	0.0%	28.6%	19.6%	0.0%
14	1.8%	1.8%	1.8%	0.0%	0.0%
26	0.0%	3.6%	7.1%	1.8%	1.8%
Error	1.8%	1.8%	7.1%	16.1%	1.8%
Other	1.8%	0.0%	0.0%	1.8%	0.0%

Sadly, the majority of Turkers expected some form of dynamic scope. Of course, it is impossible to be sure from these studies what mental model they had of binding based on this syntax (e.g., perhaps they assumed that all variables were global and = only performed assignment, or that assignment to undeclared variables lifted those variables to an outer scope). We were aware of these issues and ambiguities but nevertheless used this syntax since it is common to many popular languages that loosely have static scope. Some answers are particularly inexplicable, such as the 19.6% who chose (12, Error): why not pick Error for both? Perhaps they thought that a function's local variables should be available globally, but not from within other functions? Observe that only 16.1% chose (Error, Error).

Consensus We obtained a consensus of $\bar{p}$: 0.339; κ: 0.174.

Consistency For consistency, we binned answers into · accessing another function's variables, · error, and · other, for both of these questions. We obtain $\bar{p}$: 0.536; κ: 0.304.

3.2 Accessing Formal Parameters

The next pair of questions asks Turkers whether the program has access to formal parameters outside a function. These questions are similar to the previous ones, but focus on formal parameters, which are arguably even more clearly "scoped", a perception borne out by the data. Still, only 39.3% chose (Error, Error).

```
func f(b):
    c = 8
    return c + 5

f(4)
print(b)
```

```
func f(e):
    return e + 5

func g(e):
    return e * 2

f(7)
g(3)
print(e)
```

	print(e)						
print(b)	3	6	7	10	24	Error	Other
4	5.4%	0.0%	0.0%	0.0%	0.0%	0.0%	3.6%
8	0.0%	0.0%	0.0%	0.0%	1.8%	0.0%	0.0%
13	1.8%	5.4%	1.8%	1.8%	0.0%	17.9%	7.1%
17	0.0%	0.0%	0.0%	1.8%	3.6%	0.0%	0.0%
Error	0.0%	0.0%	1.8%	0.0%	0.0%	39.3%	1.8%
Other	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	5.4%

Consensus We calculated a consensus of $\bar{p}$: 0.337; κ: 0.223. This low score can be seen in the two largest clusters of 39.3% and 17.9% while the rest of the groups consist of at most 7.1%.

Consistency We binned answers into · the actual argument value, · the return value, · error, and · other. We obtain a consistency of $\bar{p}$: 0.625; κ: 0.5.

3.3 Variable Shadowing

The following question tests whether functions have separate bindings from top-level scope:

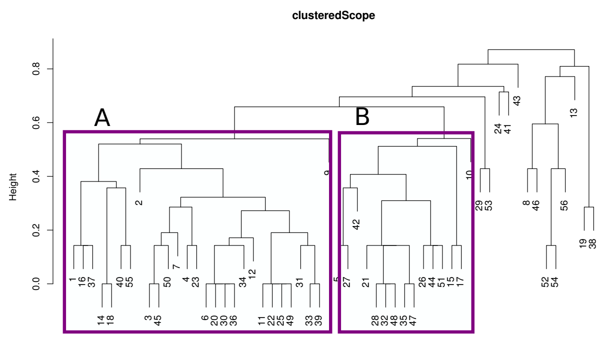
```
o = 3
func f():
    o = 27

func g():
    o = 11

f()
g()
print(o)
```

print(o)					
3	11	27	38	Error	Other
28.6%	53.6%	1.8%	5.4%	8.9%	1.8%

The 11 answer corresponds to the top-level o being the o mutated in the functions. The 3 answer corresponds to introducing new bindings within each function scope.

Consensus We report a consensus of $\bar{p}$: 0.369; κ: 0.243. As there is no question to compare this to, there is no consistency to report.

3.4 Clustering All Responses



The split between the "A" and "B" clusters was entirely due to the program in section 3.3. The workers in the "A" cluster all expected this program to produce 11. The workers in the "B" cluster all expected 3. The remaining workers did not have a majority opinion on this question. Workers in both "A" and "B" were split on whether the first program in section 3.1 would error or not.

4 Single Inheritance

We measured how workers expected field and method access to work in the presence of single inheritance. We prefaced the questions with a quick primer for classes in our made-up language and with class definitions common to the programs. We had 48 valid responses.

	ac.f()							
ac.x	3	30	33	60	300	330	Error	Other
3	4.2%	2.1%	0.0%	0.0%	2.1%	2.1%	0.0%	2.1%
6	0.0%	0.0%	0.0%	2.1%	0.0%	0.0%	0.0%	0.0%
300	2.1%	18.8%	0.0%	0.0%	43.8%	0.0%	2.1%	2.1%
303	0.0%	0.0%	0.0%	2.1%	0.0%	0.0%	0.0%	0.0%
330	0.0%	0.0%	0.0%	0.0%	0.0%	2.1%	0.0%	0.0%
333	2.1%	0.0%	2.1%	0.0%	0.0%	0.0%	0.0%	0.0%
Error	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	6.2%	0.0%
Other	2.1%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%

Table 1. Comparison of Inheritance Answers

4.1 Comparing Field and Method Access

```
class A:
    Int x = 3
    Int f():
        return x
    Int g():
        return x + f()

class B extends A:
    Int x = 30
    Int f():
        return x

class C extends B:
    Int x = 300
    Int g():
        return x + f()
```

Given this preface, we asked for the output for each of:

```
A ac = C()
print(ac.x)
```

```
A ac = C()
print(ac.f())
```

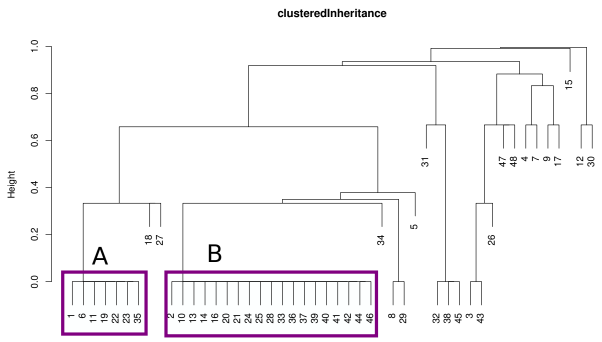
The results of these questions are shown in table 1. Notice that only one person (2.1%) expected these programs to behave as Java would (3 and 30, respectively). We also had a third question:

```
A ac = C()
print(ac.g())
```

No worker answered all three questions consistent with Java’s semantics.

Consensus We report a consensus of $\bar{p}$: 0.373; κ: 0.316. We do not report consistency as field and method access are two separate semantic concepts and Java itself does not treat them consistently.

4.2 Clustering All Responses



The “A” cluster expected ac.f() to produce 30 (Java’s answer). The “B” cluster expected it to produce 300, which corresponds to giving ac the type C instead of A. Both clusters “A” and “B” expected ac.x to produce 300 and ac.g() to produce the sum of these two.

“B”’s answers are what we would get from translating into Python, which would necessarily leave out the type annotation A on ac. Thus, maybe workers simply failed to notice the annotation. We therefore posted another typed inheritance study, this one using two variables, one declared to have type A and the other C. This led 20% to conclude that annotating with A was an *error*. This outcome was independent of whether they listed having Java experience.

It is worth noting that not a single worker expected Java’s behavior across all the questions. We were curious to see how similar but newer languages like C# and Kotlin fare. C# also goes against Turkers’ expectations: field lookup behaves similarly to Java, and method access either gets the “highest” or “lowest” method, depending on whether the superclass’s method is declared with new or virtual. Kotlin, in contrast, behaves according the the majority of workers’ expectations—the fields and methods of the "lowest" class are accessed.[2]

5 Records and Fields

We posed questions to Turkers about object aliasing and dynamic field lookup. We received 64 valid responses.

5.1 Object Aliasing

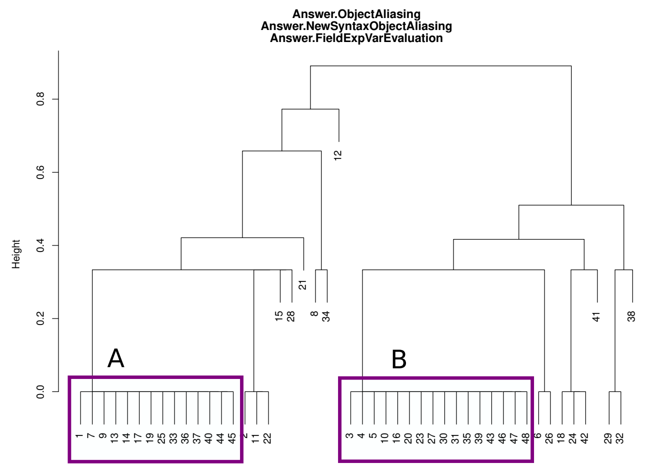
We created programs both in conventional field access syntax (`record.field`) and in one conducive to dynamic field access (`record["field"]`). We can compare between these two syntaxes as a way to validate that workers interpreted them the same way. We did not ask Turkers whether they preferred one over the other.

We compare answers to the following three questions to explore workers' expectations on aliasing:

```
func f(d):
    d.name = "Batman"
c = {"name": "Bruce Wayne", "allowance": 2000}
f(c)
print(c.name)
```

```
func f(e):
    e["type"] = "cocoon"

h = {"name": "Winky", "type": "caterpillar"}
f(h)
print(h["type"])
```

```
func f(k):
    day = "day"
    k["birth" + day] = "today"

l = {"name": "Liam", "birthday": "tomorrow"}
f(l)
print(l["birthday"])
```



About 30% of workers (cluster "B") expected aliasing while another 30% (cluster "A") expected no aliasing (potentially using a pass-by-copy semantics). The remaining workers did not have consistent expectations.

Consensus We report a consensus of $\bar{p}$: 0.410; κ: 0.230.

Consistency We report a consistency of $\bar{p}$: 0.743; κ: 0.679 for binning answers into · nothing, · object aliasing, · no object aliasing, · error, and · other. Though workers might have been split on whether object aliasing occurs naturally or not, few workers changed their opinion between the different syntactical versions of this question. However, despite our instructions on the equivalence of the two syntaxes, some workers thought the `[]` form should error as these records were not arrays. (We also tried this study with parenthetical field access syntax, but this was confused with function calls.)

5.2 Dynamic Field Lookup

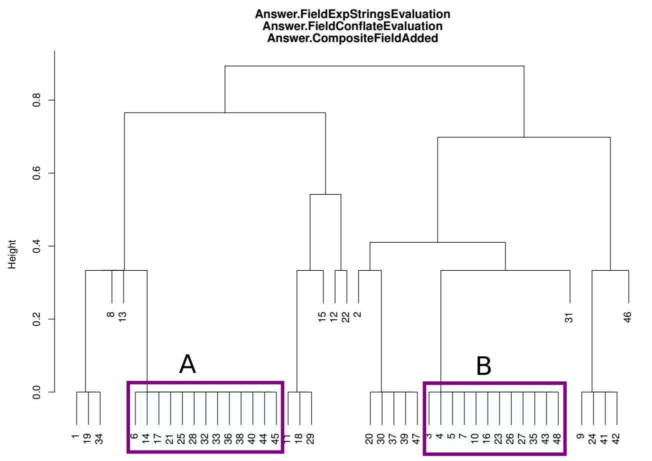
For these questions, we asked what `record["brick" + "house"]` (or `record["dog" + "house"]` for one question) should produce while varying the fields available in `record`.

```
s = {"doghouse": "red", "maker": "Brydon"}
print(s["dog" + "house"])
```

```
w = {"brick": 3, "house": 5}
print(w["brick" + "house"])
```

```
v = {"brick": 1, "house": 4, "brickhouse": 2}
print(v["brick" + "house"])
```

[2] This opens up a potential type soundness issue: if a subclass overrides a fields with a subtype of the original field's type, then assigning to the field using the superclass could violate the field's type signature in the subclass. Kotlin avoids this issue by not allowing a mutable field to be overridden with a subtype.

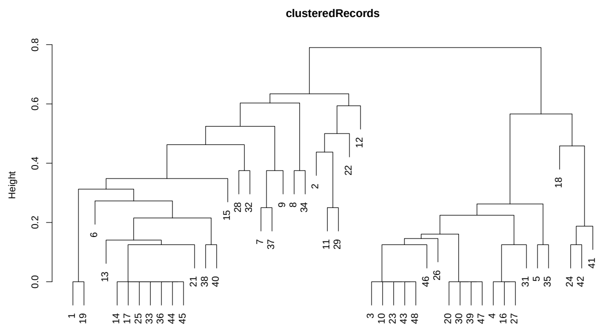


The workers in the “B” cluster evaluated the field-access expression, and then accessed the field of that name (if it existed). The “A” cluster was more interesting: they *distributed* the operation across the fields, thus choosing the answers 8 and 5 (by adding the values together).

Consensus We report a consensus over these three questions of $\bar{p}$: 0.351; κ: 0.153.

Consistency If we consider the A cluster to all be consistent, and similarly the B cluster, we obtain a consistency of $\bar{p}$: .785; κ: 0.677 (these bins are · the distributed fields, · the evaluated field expression value, and · other). We can also check whether workers use the same operation (8 and 5 versus 35 and 14) when distributing. The corresponding bins are · adding the distributed field values, · concatenating the distributed field values, · evaluating the field expression value then looking up the field, and · other. This yields $\bar{p}$: 0.833; κ: 0.778.

5.3 Clustering All Responses



The grouping on the left tended to expect no object aliasing while the grouping on the right thought aliasing occurs. Interestingly, most of the workers on the right evaluated the field-access expression first and then looked up the value; the workers in the left group were split on these questions but tended to distribute. In short, the right group followed conventional language behavior, but the left group did not.

6 Boolean Coercion

We explored which values workers find “truthy” and “falsy”. The basic format of these questions is:

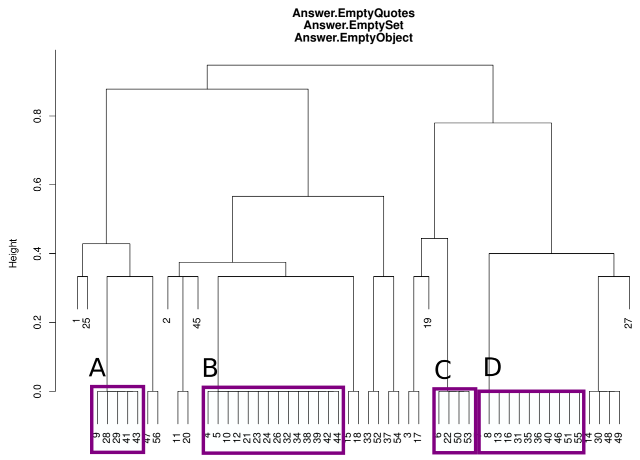
```
if (0) {
    print("blue pill")
} else {
    print("red pill")
}
```

substituting other values for `0`. For each question, the only possible answers are: `blue pill` (corresponding to the value coercing to true), `red pill` (meaning the value coerced to false), and (as always) “Error” and “Other”.

We questioned Turkers on numbers, strings, and empty data structures, and received 56 valid responses. As there are consistency issues with numbers and strings, we delay discussing it until after comparing empty data structures.

6.1 Empty Aggregates: `""`, `[]`, {}

We consider the responses for `""`, `[]`, and {}. The survey explained that `[]` means an empty list and {} means an empty object. We obtain the following cluster tree:



Each of the four largest clusters thought that these values should all behave the same way. Cluster “B” thought that each of these expressions should error in the test expression position of a conditional; cluster “D” believed they should all be falsy; cluster “A” picked Other for each; and cluster “C” believed they should all be truthy.

Consensus No single cluster has a majority of workers, so we obtain low consensus scores of $\bar{p}$: 0.275; κ: -0.033.

Consistency `""`, `[]`, and {} are all empty *aggregates*: of characters, list elements, or field values, respectively. It is therefore plausible that they should all be treated identically as boolean conditionals, and in fact Turkers largely agree. Binning answers into · true, · false, · error, and · other, we get a consistency of $\bar{p}$: 0.726; κ: 0.635.

6.2 The Remaining Questions

The rest of the truthy/falsy questions we asked were essentially independent of one another, so we simply report how many workers believed each value to be truthy/falsy/etc.:

Question	truthy	falsy	Error	Other
`0`	19.6%	48.2%	8.9%	23.2%
`15`	37.5%	23.2%	19.6%	19.6%
`"-1"`	21.4%	28.6%	32.1%	17.9%
`NaN`	12.5%	30.4%	30.4%	26.8%
`"0"`	25.0%	30.4%	25.0%	19.6%
`nil`	14.3%	37.5%	25.0%	23.2%
`true`	51.8%	14.3%	7.1%	26.8%

Notably, workers expected 0—more than any other value—to be falsy, likely from a boolean interpretation of 0 and 1 (as in C).

The number of people who picked Other is uncommonly high for this survey. The majority of these workers reported that they did not expect to see just a constant in the predicate position, and (in effect) expected to see an expression that included an explicit comparison operator. This may be a consequence of many traditional computing curricula, which do not usually show a raw constant in the predicate position, nor use a reduction semantics [7], which would force a learner to confront this possibility.

In response to this, we sent out another survey with a variable assigned to the constant we wanted to test, and the variable placed in the predicate position. This did not remove the original problem entirely (some workers still expected a comparison), and introduced new difficulties due to a variable being in the predicate position (some workers thought the conditional would evaluate to true if the variable inside of it were assigned to anything at all).

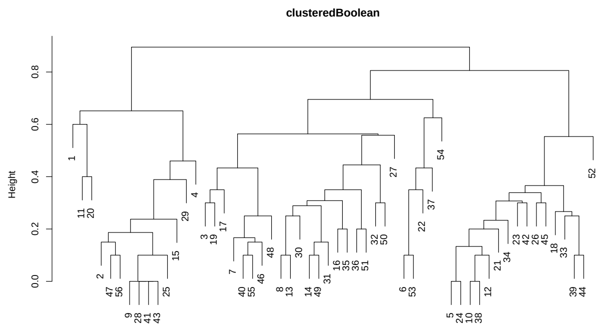
Interestingly, 14.3% of workers expected `true` to be falsy. These workers had similar reasoning as those who picked Other. They gave one of two explanations: (i) since there was no input to this comparison, the conditional would default to false; or (ii) they expected that an implicit input was being compared to `true`, and that this implicit input would most likely be false.

6.3 Clustering All Responses

The full set of 10 questions tested the truthiness of:

```
true, 0, "", NaN, nil, "0", "-1", [], {}, 15
```

We obtained the cluster tree:



Only eight workers agreed with anyone else for all questions—everyone else answered distinctly. Generally, the workers in the left area picked Other for most values, those in the middle area picked `true` or `false` for a majority of questions, and those in the right area picked Error. The overall consensus was correspondingly low: $\bar{p}$: 0.278; κ: 0.036.

7 Numbers

We received 60 valid responses to questions about numbers. We asked three questions, one of which was free-response, so we leave it out of the comparisons below.

Precision

We asked workers to tell us what they expected the programs

```
1 / 3
```

and

```
2.5 + 2.125
```

to produce, with answers that explored a variety of options of precision and presentation. Their answers were as follows:

	`2.5 + 2.125`		
`1 / 3`	`4.6`	`4.625`	Error
`0`	0.0%	10.0%	0.0%
`0.3`	3.3%	5.0%	0.0%
`0.333`	0.0%	8.3%	1.7%
`0.3333333333333`	0.0%	31.7%	0.0%
`0.3333. . .`	0.0%	20.0%	0.0%
$0.\bar{3}$	0.0%	11.7%	0.0%
`1/3`	0.0%	3.3%	0.0%
`NaN`	0.0%	1.7%	0.0%
Error	0.0%	1.7%	0.0%
Other	0.0%	1.7%	0.0%

(The second program had additional answers, such as 4 and 5, which nobody picked.)

Consensus We obtain a consensus of $\bar{p}$: 0.536; κ: 0.467. Virtually every worker expected the answer to the second program to be precise, but for the first program fewer chose any of the precise answers, and they were furthermore split on the preferred output, resulting in a lower overall consensus score.

Consistency We looked for consistency across the questions in choosing precise answers: this means binning together the several precise representations for the first question (e.g., `1/3` with `0.3̄`). Thus, the bins are · precise, · not precise, · error, and · other. This resulted in a score of $\bar{p}$: 0.383; κ: 0.248, which is not very high.

However, this classification conflates two issues. One is the *internal representation* of the quotient, the other is what the computer will *output* [33]. Many of the workers who picked `0.3333333333333` mentioned that the computer would truncate the display. Only three (out of 19) mentioned anything about floating point numbers. Thus, if we instead code `0.3333333333333` as precise in the classification above, we obtain a consistency of $\bar{p}$: 0.700; κ: 0.630.

8 Number and String Operations

We now move on to common binary operations with numbers and strings as operands. We asked 12 questions, and received 45 valid responses.

We constructed most of these questions in analogous pairs. We give the consensus and consistency scores for each question pair below, and will describe the consistency bins next:[3]

		Consensus		Consistency	
Question1	Question2	$\bar{p}$	κ	$\bar{p}$	κ
`"5" + 2`	`2 + "5"`	0.232	0.122	0.933	0.924
`3 * "4"`	`"4" * 3`	0.224	0.138	0.822	0.802
`"123"-"3"`	`"123"-"2"`	0.209	0.121	0.911	0.898
`"xyz"-"z"`	`"xyz"-"y"`	0.352	0.223	0.911	0.893
`"1234"/4`	`"XXXX"/4`	0.358	0.269	0.622	0.528

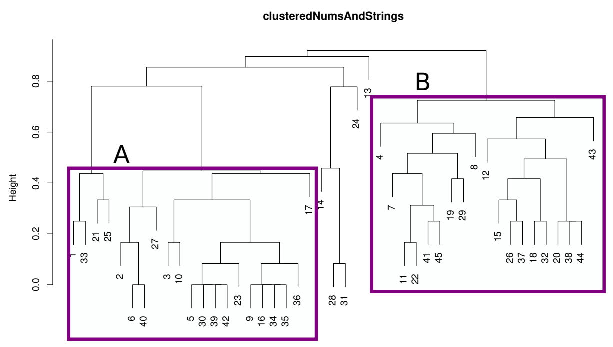
Workers were generally consistent on each question pair. There were a large number of bins, but most workers fell into just these few:

- **+** · error, · addition (converting strings to numbers as necessary), and · string concatenation
- **-** · error, · subtraction, and · removal of the right operand from the left
- ***** · error, · string repetition, · multiplication (converting strings to numbers, and producing a number), and · multiplication (converting strings to numbers, and producing a string)
- **/** · error, · producing the first character of the left operand, and · producing `NaN`

[3] The subtraction and division questions had space around the operator; we omit it here for space.

8.1 Clustering All Responses

Beyond the questions described above, we also asked workers their expectations on `"6" + "7"` and `"8" * "2"`. The overall cluster tree is:



The "A" cluster is characterized by expecting a majority of the operations to error. The "B" cluster expected almost all operations to produce some value.

8.2 Type Annotations

The content here was actually part of the types survey (section 9), so we do not include its responses in the clustering above. We provided the following question:

```
x: String = "5"
y: Int = 3
print(x + y)
```

We include it here to see what effect including type annotations has on producing a value or an Error. Without type annotations, 33% of workers expected this program to error. With type annotations, 41% of workers expected it to error. This difference is not statistically significant (two-sample t-test, p-value = 0.458). Interestingly, a larger percentage of workers in the Types survey than workers in the Number and String Operations survey expected string concatenation to occur: 36% vs 13%. This difference is statistically significant (two-sample t-test, p-value = 0.008).

9 Types

We ask Turkers whether the argument to a function can be of a different type than the formal parameter, and whether an error is produced when incompatible types are used in an operation. We also probed their expectations of what are commonly run-time errors. We have 49 valid responses.

9.1 Re-Assignment or Re-Binding

We reassigned a variable to a value of a different type. However, we used the conventional = syntax, which can be interpreted as either assignment or (re-)binding. We used two variants, distinguished only by a type annotation, to see whether their interpretation of = was changed by the annotation:

```
x: Number = 3
x = "hello"
print(x)
```

```
x: Number = 3
x: String = "hello"
print(x)
```

	x: String = "hello"			
x = "hello"	3	hello	Error	Other
3	0.0%	2.0%	0.0%	0.0%
hello	4.1%	22.4%	4.1%	0.0%
Error	0.0%	36.7%	26.5%	2.0%
Other	0.0%	0.0%	2.0%	0.0%

The 36.7% group suggests that the type annotation made a difference. The workers in this group explained: that the variable was re-declared as a string; that its type was changed to be a string; that it was reassigned to be a string variable; or that a new variable was declared.

Consensus We obtained a consensus of $\bar{p}$: 0.492; κ: 0.323. This low agreement is due to the three large groupings of responses.

Consistency We consider two possible ways to be consistent. It's certainly consistent to report Error in both cases. We bin answers into · the original value, · the mutated value, · error, and · other. This obtains $\bar{p}$ 0.490; κ: 0.320. We are also willing to consider that the annotation makes the second program legal, making Error and `hello` consistent. As these two are now consistent, we can only bin answers into · the original value, · type annotation makes the second program valid, and · other. This scores $\bar{p}$: 0.429; κ: 0.143.

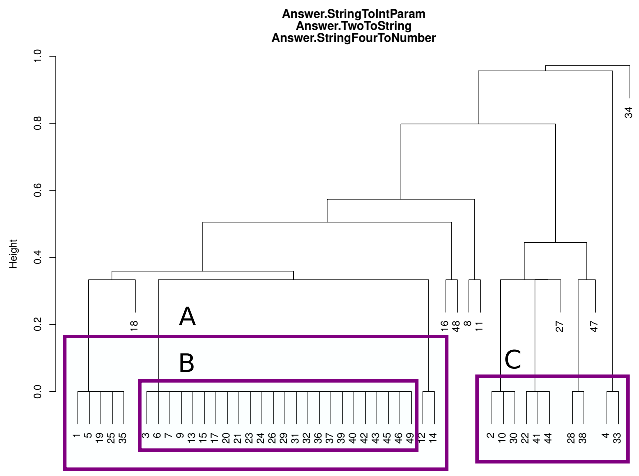
9.2 Incorrect Argument Types

We now examine expectations when actual arguments do not match the formal ones.

```
func f(s: String) -> Number :
    12 / s
print(f(2))
```

```
func f(i: Number):
    print(i + 1)

f("one")
```

```
func f(s: Number) -> Number :
    return 12 / s
print(f("4"))
```



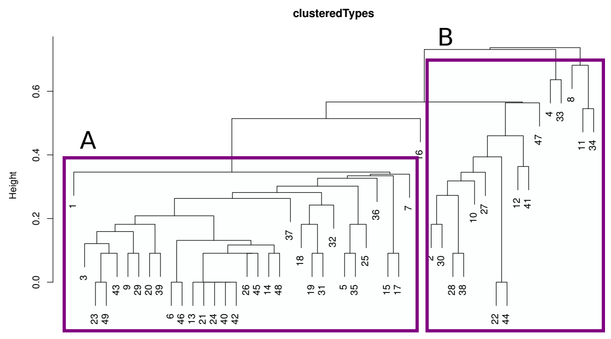
One might expect that Turkers are more likely to think these programs error because the type is explicitly mentioned in the function header. This is in contrast to binary operators, which are often overloaded.

Indeed, the large "A" cluster contains those workers who expected the majority of these programs to error. The inner "B" cluster contains those workers who expected every one of these programs to error. The "C" cluster contains the workers who expected these programs to produce a value, except for three workers in the middle, who expected the first program to error.

Consensus We report $\bar{p}$: 0.505; κ: 0.367. Again, this can be attributed to the number of workers who expected all three programs to error.

Consistency We use a coarse form of consistency: · whether these programs are expected to error or · not or · other. With that loose expectation, we report $\bar{p}$ 0.755; κ: 0.633.

9.3 Clustering All Responses



The large "A" cluster contains workers who thought that the majority of programs should error while the remaining workers, in "B", thought that the majority of these programs should produce a value.

9.4 Runtime Error

To check whether workers who didn't expect annotation errors would expect any errors at all, we included a question that would result in an out-of-bounds error.

```
func getFifthElement(a: Array) -> Int :
    return a[5]

v: Array = [1, 4, 9]
print(getFifthElement(v))
```

About 82% of workers expected the program to error. One worker picked 4; three workers picked 9. None of these workers had a helpful explanation. Five workers picked "Other." Two of the five gave clear explanations. One said that it should error, but mentioned that `Array` is "not further typed as a general container." The other worker expected it to return `null`. (In an earlier version of this study, one respondent picked the answer 9 because they counted the commas as elements…)

9.5 Strictly Typed vs Scripting Languages

We also asked these Turkers whether they preferred "strongly/static typed languages" or "scripting languages". Eighteen responded that they preferred scripting languages while 31 responded that they preferred typed languages.

We correlated these preferences against whether the workers ended up in clusters "A" and "B" (which roughly correspond, respectively, to "typed" and "scripting" responses). We obtained a Pearson's correlation of 0.210, which is—curiously—only a slight correlation.

10 Closures and Lambdas

This survey began by showing an example of how we use lambdas in this language. We received 49 valid responses.

10.1 Top-Level State

We first consider how closures interact with top-level state.

```
z = 5
f = lambda(x): return x + z end
z = 3
print(f(2))
```

```
f = lambda(y): return y * b end
b = 10
print(f(3))
```

These programs resulted in table 2.

Consensus We report $\bar{p}$: 0.579; κ: 0.484.

Consistency We consider the answers 5 and 30 consistent (the same behavior as a language like Scheme). We also consider 7 and Error to be consistent. Binning all other values as having some other semantic model (resulting in bins of · Scheme-like, · closing over the store, and · other), we obtained a consistency of $\bar{p}$: 0.714; κ: 0.571.

	`print(f(3))`			
`print(f(2))`	3	30	Error	Other
5	0.0%	65.3%	2.0%	0.0%
7	2.0%	6.1%	4.1%	2.0%
10	0.0%	4.1%	0.0%	0.0%
Error	0.0%	6.1%	6.1%	0.0%
Other	2.0%	0.0%	0.0%	0.0%

Table 2. Comparison of Top-Level State Answers

10.2 Local Bindings

We next explored Turkers' expectations of nested scopes and lambdas.

```
func make_lambda():
    h = 2
    return lambda(x): return x - h end

h = 13
f = make_lambda()
print(f(20))
```

`print(f(20))`				
5	7	18	Error	Other
2.0%	18.4%	44.9%	26.5%	8.2%

A surprising number of workers expected `make_lambda` to error. Four out of these 13 workers thought that `f` was the same function as `make_lambda`, and were therefore confused to see a parameter (`20`) being passed. Three workers complained that the variable `x` wasn't defined. The remaining workers did not have helpful explanations.

Consensus We report $\bar{p}$: 0.298; κ: 0.123. As this is a singular question, we do not report consistency.

10.3 Closure Creation

```
func hello(name):
    return lambda(): print("Hello, " + name) end

hiJ = hello("Jack")
hiP = hello("Peter")
hiJ()
```

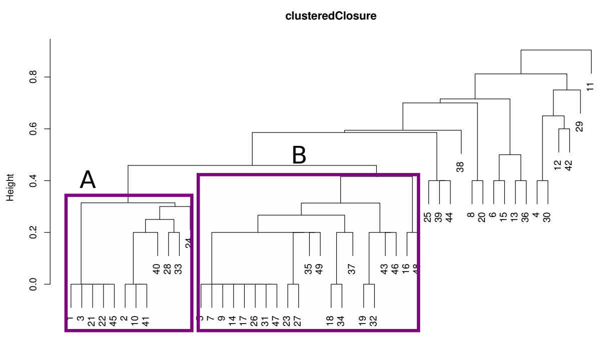
Answer	Percentage
`Jack`	4.1%
`Hello, Jack`	67.3%
`Hello, JackPeter`	2.0%
`Hello, JackHello, Peter`	16.3%
Error	8.2%
Other	2.0%

Fortunately, two-thirds of workers correctly understood how closures should work. Unfortunately, one-sixth thought that the print statement was invoked twice.

Consensus We report a consensus of $\bar{p}$: 0.479; κ: 0.414. Again, there is no question to compare this question to, so there is no consistency to calculate.

10.4 Clustering All Responses

We cluster all questions over all workers.



The "A" and "B" clusters are split on `make_lambda`. Those in "A" expected either 7 or Error while the workers in "B" expected 18. The remaining workers usually only differed by one question.

11 Related Work

Several languages have been designed around user studies. Most of these have focused on novice programmers. For instance, LOGO's turtle geometry was based on systems that children could handle [29]; the natural programming project [28] conducted studies of child programmers to drive the language's design; and Python's predecessor, ABC [27], was designed to be easy to learn for beginners. In contrast, we do not limit ourselves to beginner features.

Other research has focused on more specific topics: Quorum's [34] syntax is based on user studies; Hanenberg and others have studied specific language features [18, 24, 36]; and there have been several studies of programming environments and tools [19, 22] that end up having at least some impact on language design itself. Our studies, on the other hand, focus on overall language design, and in particular on *semantic*, rather than *syntactic* features.

Miller et al. [25] describe how to use programmers' expectations to drive language design. They observe that the semantics of a language as designed, as implemented, and as understood by users tend to all be distinct. They improve Yedalog by making its specification more similar to its users' expectations. Their observations about users are somewhat similar to ours. A major difference is that they are focused on improving one particular feature of an existing language, whereas we are looking at a wide range of language features to identify a holistic language design in the process.

Kaijanaho proposes using results from research papers to design programming languages [15] and offers a useful critique on using Cohen's κ in these kinds of studies. His recommendations, which include Fleiss' κ, all suffer from the problem outlined at the beginning of this paper—namely, they all assume that each question has the same set of possible answers.

There have also been many studies of student learning and misconceptions [2, 4, 31] that have the potential to impact language design. See Sorva [32, pg. 29] for an overview. Our surveys have a similar format of questions and choices of answers (corresponding to common interpretations). We too have found a number of problematic expectations that programmers hold (section 13). However, there are three major differences between that literature and this paper:

- Those papers almost uniquely focus on a certain definition of introductory programming classes. In contrast, MTurk workers are required to be adults (18 and older) and many are working professionals.
- Corresponding to the focus, those papers examine a narrow set of features, such as variables, loops, and function calls. Our work extends beyond this set in both directions, to more advanced features such as the interaction between inheritance and types, and to features taken for granted, like numbers. Similarly, we are also more focused on some of the edge cases of language behavior, which introductory courses would usually not cover and hence not test.
- Finally, there is a major difference in goals. That work is attempting to study whether programmers understand the semantics of the languages they work with. In contrast, we did not start with any fixed language, and instead wanted to explore what workers would want their languages to do. This results in a fairly different set of actual questions.

Nevertheless, we were able to reproduce some of those findings, such as Fleury's observation that novices expected functions to fall back on dynamic scope when a variable was otherwise unbound [11] (section 3.1).

There has been some work on crowdsourcing programming *environments* as well. For instance, Hartmann et al. develop a system that recommends bug fixes based on corrections that peers have made [14], and Mooty et al. crowdsource API documentation [26]. Our study focused entirely on language design, ignoring the programming environment, though it does point out some misconceptions that could limit the effectiveness of crowdsourced explanations.

12 Threats to Validity

Our work has several threats to validity, including these.

First is the use of MTurk itself. Though our criterion for Turkers mentioned programming experience, and we eliminated those who did not select any languages, we did not first verify their programming ability. By providing multiple-choice answers, we may have made it even easier for cheaters to click at random. To prevent this, we did force workers to explain their reasoning for every answer. Most workers provided answers that were not nonsensical (and unlike what we might get from non-programmers or naive bots). Therefore, we have some faith in the responses.

Nevertheless, other researchers have found that up to 40% of Turker responses can be considered uninformative [17], a ratio that may also apply to us. That said, MTurk is known to be a good forum for finding technically qualified workers [23], so while we believe it is worth considering stronger barriers, we do not believe the use of MTurk is itself inherently questionable. To make sure that we only surveyed qualified Turkers, we could have included simple questions by which we could filter out workers who answered incorrectly.

Second is the nature of questions and answers. Three things characterize our quizzes: the sample programs were short, they did not mix many features, and they provided multiple-choice answers. The short programs and multiple-choice answers were intended to make the tasks quick to complete, in keeping with recommendations for MTurk [20]. However, each of these is an important restriction, and lifting them might result in fairly different responses.

Third is that workers might have been biased towards believing that programs were valid and did not error. To mitigate this, we explicitly made "Error" an answer choice and gave it just as much visibility of any of the other answer choices. Furthermore, many workers did expect programs to error. Therefore, we do not believe this is a major threat to validity.

Fourth is that workers might have been influenced by the syntax of our programs. We tried to mitigate this by explicitly choosing a unique syntax. Nevertheless, some individual features (such as indentation, the JavaScript-like map structures) might have biased the workers towards those languages' behaviors.

Fifth is the expertise and qualification of our audience, over which we have no real control. How would these answers vary if we ran a similar study with students at various stages (high school, first-year college, about to graduate), professional programmers, computer science department faculty members, or for that matter participants at a programming languages conference? It would be particularly interesting to explore variations between this last group and everyone else. That said, all these audiences—including the Turkers—use these languages, so we should not be too quick to dismiss the expectations of the MTurk workers.

Finally, we have not yet investigated the stability of these opinions. If we ask many more workers, or ask at different times, do we see a major difference? It is known that the time of day has some impact on MTurk (in part due to the large population of workers from the USA and India) [23], but other, more subtle variables may also be at play. In addition, programmer attitudes are clearly shaped by the languages they use: worker responses in the era of Pascal, C, and Ada may have been very different than they are now when languages like JavaScript, Ruby, and Python—with all their peculiarities [3]—have broad followings.

13 Language Design Consequences

Until this point in the paper, we have mostly only reported findings factually, offering relatively little commentary. The goal of this study, however, is to understand the consequences for language design, which we now discuss.

It is tempting, a priori, to expect that workers would simply choose answers that are consistent with whatever language they use most, i.e., regurgitate existing design decisions. However, we find that even this is not the case, as in many cases workers make choices that are not consistent with any mainstream language: for instance, distributing addition over field access, and accessing function parameters outside the function. Whether it is because they do not understand their languages well enough, or dislike the choices already made, or (in rare cases, like section 4) perhaps simply do not interact in the same way with these features in their daily programming, is a matter for further investigation.

Our overarching conclusion is that we do not see enough consistency or consensus amongst workers to suggest canonical language designs. One could argue that these studies show two camps of workers: those who expect an ML-like or strict language and those who expect a scripting-like language. The workers in the strict camp expected narrow behavior including static scoping, type annotations being respected, minimal operator overloading, and only boolean expressions in the predicate position of conditionals. The workers who fell into the scripting camp wanted rich behavior, but they didn't agree on exactly what that behavior should be.

However, there are still some puzzling clumps of workers. Indeed, even in places where we see large clusters of agreement, they tend to point in directions languages should probably not take: for instance, permitting dynamic scope, and allowing annotations on function parameters to be ignored. Future work needs to explore why programmers want these behaviors, whether they understand the consequences of programming in languages that work this way (especially when some of these features interact), and—most importantly—whether even small amounts of training can dissuade them from making these choices [8]. One hopes that if workers were shown the consequences of their choices then they would be willing to reconsider. However, even so, research is cognitive psychology [5] has shown that misconceptions are difficult to un-learn.

The finding on dynamic scope is particularly disturbing, since it is quite clear that no matter how much programmers may find it appealing for small programs, it is unlikely to return in any carefully-designed language. Indeed, even languages that have included it by accident or lack of forethought (e.g., Python and JavaScript) have evolved to remove it as much as possible. It is certainly worth investigating whether workers even recognize terms like "dynamic scope", have any (negative) associations with it, and can recognize it in a program. It is also unclear whether programmers understand the negative consequences dynamic scope has for the construction of tools. It may be that educating, or even priming, them along these lines may result in different stated preferences.

We also noticed intriguing behavior on other features:

- On *numbers* (section 7), many workers preferred exact arithmetic, and this number grows even higher if we account for the difference between evaluation and printing. This is inconsistent with the behavior of most mainstream languages, which (a) make an artificial distinction between integers and floats, and (b) whose floats fail to obey several well-understood mathematical properties (such as associativity and distributivity). It is unclear whether programmers compartmentalize and mode-shift between "numbers" and "computer numbers", or whether they incorrectly believe numbers in programming languages behave like their mathematical counterparts. Anecdotally, we certainly see evidence—for instance, on programmer discussion fora—of developers suffering from the latter misconception.
- With *inheritance* (section 4), as no worker expected these questions to perform as Java does, one should question how natural Java's approach is. We were also surprised by how brittle their understanding was, with small changes to programs shifting their expectations significantly.
- *Operator overloading* (section 8) showed that workers have consistent interpretations for `+` and `-`, but not so for other binary operations. This suggests that at least all the other cases ought to be made an error.
- In the case of *aliasing* (section 5.1), we were surprised by the number of programmers who did not expect object aliasing across a function call. This is inconsistent with virtually all mainstream languages that have side-effects, and many programs explicitly rely on the aliased behavior (e.g., when passing an object to a `void`-returning method); disturbingly, many workers tend towards assuming a non-existent "safety". We also note that the workers all have experience with primarily-imperative languages. This does suggest investigating other language mechanisms, whether through immutable data or linearity or an outright lack of aliasing [35]. In addition, it suggests that language tutorials and documentation must explicitly teach users about aliasing behavior in the language, especially in the presence of mutation.
- We find a morass when it comes to *truthiness* (and falsehood) (section 6). This may be unsurprising given the variation between languages in the field: just between Python, Ruby, JavaScript, and PHP, there is little to no agreement on which values are truthy/falsy, each having distinct tables—even though each presumably represents what its designers thought was "obvious". The danger of permitting such variation is that programmers are bound to get confused when they switch between languages, especially within the same application (e.g., a Web application that uses a JavaScript front-end and Python back-end). For this reason, it seems safer to allow only values of a distinct Boolean type in the conditional position.

In summary, it is difficult to learn one clear message from these studies, but there do seem to be two themes. One is that workers seem to have difficulty agreeing, and even at being consistent on unusual features. This makes it difficult to design a language around their preferences. The other is that some standard features (such as inheritance and overloading) that have a complex semantics are not easily predictable by programmers. These should therefore either be avoided or at least explained well, with attention paid to corner cases.

Acknowledgments

This work was partially supported by the NSF. We would like to thank Natasha Danas for advice on conducting studies on Mechanical Turk. We would also like to thank our anonymous reviewers for their helpful feedback.

References

[1] 2017. Betteridge's law of headlines. https://en.wikipedia.org/wiki/Betteridge%27s_law_of_headlines. (2017). Accessed: 2017-04-20.

[2] Piraye Bayman and Richard E. Mayer. 1983. A diagnosis of beginning programmers' misconceptions of BASIC programming statements. *Commun. ACM* 26, 9 (1983). DOI: http://dx.doi.org/10.1145/358172.358408

[3] Gary Bernhardt. 2012. Wat. A lightning talk from CodeMash. (2012). https://www.destroyallsoftware.com/talks/wat.

[4] Benedict Du Boulay. 1986. Some Difficulties of Learning to Program. *Journal of Educational Computing Research* 2, 1 (1986). DOI: http://dx.doi.org/10.2190/3LFX-9RRF-67T8-UVK9

[5] M.T.H. Chi. 2005. Common sense conceptions of emergent processes: Why some misconceptions are robust. *Journal of the Learning Sciences* 14 (2005), 161–199.

[6] Jacob Cohen. 1960. A Coefficient of Agreement for Nominal Scales. In *Educational and Psychological Measurement.* SAGE. DOI: http://dx.doi.org/10.1177%2F001316446002000104

[7] Matthias Felleisen. 1992. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science* 103, 2 (1992), 235–271. DOI: http://dx.doi.org/10.1016/0304-3975(92)90014-7

[8] Kathi Fisler, Shriram Krishnamurthi, and Preston Tunnell Wilson. 2017. Assessing and Teaching Scope, Mutation, and Aliasing in Upper-Level Undergraduates. In *Special Interest Group on Computer Science Education*. ACM, New York, NY, USA. DOI: http://dx.doi.org/10.1145/3017680.3017777

[9] Matthew Flatt. 2002. Composable and compilable macros: you want it when? ACM, New York, NY, USA, 12. DOI: http://dx.doi.org/10.1145/581478.581486

[10] Joseph L. Fleiss. 1971. Measuring Nominal Scale Agreement Among Many Raters. In *Psychological Bulletin*. APA. DOI: http://dx.doi.org/10.1037/h0031619

[11] Ann E. Fleury. 1991. Parameter Passing: The Rules the Students Construct. In *Special Interest Group on Computer Science Education*. ACM, New York, NY, USA. DOI: http://dx.doi.org/10.1145/107004.107066

[12] J. C. Gower. 1971. A General Coefficient of Similarity and Some of Its Properties. *Biometrics* 27, 4 (1971), 857–871. DOI: http://dx.doi.org/10.2307/2528823

[13] R. W. Hamming. 1950. Error Detecting and Error Correcting Codes. *The Bell System Technical Journal* 29, 2 (1950), 147–160. DOI: http://dx.doi.org/10.1002/j.1538-7305.1950.tb00463.x

[14] Björn Hartmann. 2010. What Would Other Programmers Do? Suggesting Solutions to Error Messages. In *Special Interest Group on Computer–Human Interaction*. ACM, New York, NY, USA, 1019–1028. DOI: http://dx.doi.org/10.1145/1753326.1753478

[15] Antti-Juhani Kaijanaho. Evidence-Based Programming Language Design. (????).

[16] Leonard Kaufman and Peter J. Rousseeuw. 2008. *Finding Groups in Data: An Introduction to Cluster Analysis*. Wiley.

[17] Aniket Kittur, Ed H. Chi, and Bongwon Suh. 2008. Crowdsourcing User Studies with Mechanical Turk. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '08)*. ACM, New York, NY, USA, 453–456. DOI: http://dx.doi.org/10.1145/1357054.1357127

[18] Sebastian Kleinschmager, Stefan Hanenberg, Romain Robbes, Éric Tanter, and Andreas Stefik. 2012. Do Static Type Systems Improve the Maintainability of Software Systems? An Empirical Study. In *International Conference on Program Comprehension*. IEEE, 153–162. DOI: http://dx.doi.org/10.1109/ICPC.2012.6240483

[19] Andrew J. Ko, Brad A. Myers, and Htet Htet Aung. 2004. Six Learning Barriers in End-User Programming Systems. In *Visual Languages and Human Centric Computing*. IEEE, 199–206. DOI: http://dx.doi.org/10.1109/VLHCC.2004.47

[20] L. Layman and G. Sigurđsson. 2013. Using Amazon's Mechanical Turk for User Studies: Eight Things You Need to Know. In *International Symposium on Empirical Software Engineering and Measurement*. ACM, 275–278. DOI: http://dx.doi.org/10.1109/ESEM.2013.42

[21] Martin Maechler, Peter Rousseeuw, Anja Struyf, Mia Hubert, Kurt Hornik, Matthias Studer, Pierre Roudier, and Juan Gonzalez. 2017. Package 'cluster'. https://cran.r-project.org/web/packages/cluster/cluster.pdf. (2017). v2.0.6. Accessed: 2017-04-20.

[22] Guillaume Marceau, Kathi Fisler, and Shriram Krishnamurthi. 2011. Mind Your Language: On Novices' Interactions with Error Messages. In *Symposium on New ideas, new paradigms, and reflections on programming and software (Onward!)*. ACM. DOI: http://dx.doi.org/10.1145/2048237.2048241

[23] Winter Mason and Siddharth Suri. 2012. Conducting behavioral research on Amazon's Mechanical Turk. *Behavior research methods* 44, 1 (2012), 1–23.

[24] Clemens Mayer, Stefan Hanenberg, Romain Robbes, Éric Tanter, and Andreas Stefik. 2012. An Empirical Study of the Influence of Static Type Systems on the Usability of Undocumented Software. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*. ACM. DOI: http://dx.doi.org/10.1145/2384616.2384666

[25] Mark S Miller, Daniel Von Dincklage, Vuk Ercegovac, and Brian Chin. 2017. Uncanny Valleys in Declarative Language Design. In *LIPIcs-Leibniz International Proceedings in Informatics*, Vol. 71. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[26] Mathew Mooty, Andrew Faulring, Jeffrey Stylos, and Brad A. Myers. 2010. Calcite: Completing Code Completion for Constructors Using Crowds. In *Visual Languages and Human Centric Computing*. IEEE, 15–22. DOI: http://dx.doi.org/10.1109/VLHCC.2010.12

[27] Brad A. Myers, John F. Pane, and Andy Ko. 1976. Natural Programming Languages and Environment. *Mathematisch Centrum, Afdeling Informatica* (1976).

[28] Brad A. Myers, John F. Pane, and Andy Ko. 2004. Natural Programming Languages and Environment. *Commun. ACM* 47, 9 (2004).

[29] Seymour Papert. 1993. *Mind-Storms: Children, Computers, and Powerful Ideas*. Basic Books, New York, NY, USA.

[30] Justus J. Randolph. 2005. Free-Marginal Multirater Kappa: An Alternative to Fleiss' Fixed-Marginal Multirater Kappa. In *Joensuu Learning and Instruction Symposium*. ERIC, 20.

[31] Amber Settle. 2014. What's Motivation Got to Do with It? A Survey of Recursion in the Computing Education Literature. *Technical Report at DePaul University* 23 (2014).

[32] Juha Sorva. 2012. *Visual Program Simulation in Introductory Programming Education*. Ph.D. Dissertation. Aalto University.

[33] Guy L. Steele, Jr. and Jon L. White. 1990. How to Print Floating-point Numbers Accurately. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation (PLDI '90)*. ACM, New York, NY, USA, 112–126. DOI: http://dx.doi.org/10.1145/93542.93559

[34] Andreas Stefik and Susanna Siebert. 2013. An Empirical Investigation into Programming Language Syntax. In *ACM Transactions on Computing Education*. ACM, New York, NY, USA. DOI: http://dx.doi.org/10.1145/2534973

[35] Robert E. Strom, David F. Bacon, Arthur P. Goldberg, Andy Lowry, Daniel M. Yellin, and Shaula Alexander Yemini. 1991. *Hermes: A Language for Distributed Computing*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[36] Phillip Merlin Uesbeck, Andreas Stefik, Stefan Hanenberg, Jan Pedersen, and Patrick Daleiden. 2016. An empirical study on the impact of C++ lambdas and programmer experience. In *International Conference on Software Engineering*. ACM, New York, NY, USA. DOI: http://dx.doi.org/10.1145/2884781.2884849

A Methodology

A.1 Measuring Agreement

Both consistency and consensus are measures of *agreement*. We will first describe how to measure agreement in general, and then state how this applies to consistency and consensus. As stated earlier, our questions were primarily multiple-choice, but had "Other" and "Error" as write-in questions, and we binned all Error answers together, and likewise all Other responses.

In general, agreement can be measured between a set of *raters* that give answers in a set of *cases*. The naive measure of agreement $\bar{p}$ is the probability that two random raters agree on a random case. However, some of this "agreement" may have occurred by chance. This is not accounted for in $\bar{p}$, but it is accounted for in inter-coder reliability metrics, or "kappa" scores. There are two notable options: Fleiss' κ [10] (which is a generalization of Cohen's κ), and free-marginal κ [30].

Fleiss' κ is not applicable for many of our use cases because it requires that each question have the same set of possible answers. Additionally, it gives inappropriately low agreement scores when there are few cases, which is often our situation. In the extreme, when there is only one case, Fleiss' κ cannot be positive (it may be negative, 0, or undefined). This is due to the (unrealistic) assumption in Fleiss' κ that the actual distribution of answers for questions is *exactly* the observed distribution.

Free-marginal κ, on the other hand, assumes that the actual distribution of answers is uniform. This is also an unrealistic assumption. Since we attempted to provide answers for all plausible interpretations for each question (including misinterpretations), "Other" need not have been chosen *at all*. For the same reason, many of the answers we provided were for fringe interpretations of the program, which could plausibly not have been picked at all. As a consequence, free-marginal κ scores are *higher* than they should be (as a uniform distribution minimizes agreement by chance).

Since we argue that agreement scores are *low*, it would be disingenuous to report Fleiss' κ scores even in cases where it *is* technically applicable. We therefore report only free-marginal κ. Of course, since we make our data freely available, readers may perform their own analyses as well.

Consensus For consensus, $\bar{p}$ can be measured as follows: (i) pick a question uniformly at random; (ii) pick two workers uniformly at random; (iii) $\bar{p}$ is the probability that those two workers answered that question the same way.

In terms of inter-coder reliability, the "raters" are workers and the "cases" are questions.

Consistency For consistency, $\bar{p}$ can be measures as follows: (i) pick a worker uniformly at random; (ii) pick two questions (of the subset under consideration) uniformly at random; (iii) $\bar{p}$ is the probability that that worker gave consistent answers to those two questions. (Consistent answers are those we judged to be consistent, as described in each comparison.)

Although it may seem unusual, in terms of inter-coder reliability, the "raters" are questions and the "cases" are workers. In general, the "raters" attempt to categorize each "case", and one looks for agreement *among* the raters *for* each case. For the consistency measure, the questions are being used to categorize the workers in terms of which consistent viewpoint they have, and we look for agreement *among* the questions *for* each worker. This is also the interpretation needed for $\bar{p}$ to be computed as above.

A.2 Clustering Analysis

Hierarchical clustering is performed by first computing all pairwise dissimilarities between observations, and then using those dissimilarities to (recursively) group the observations into clusters. In our case, an *observation* in a survey is the vector of answers given by a worker. We excluded the questions about prior programming experience, and the (rare) open ended questions, since clustering would not be meaningful on them. This left only the questions with categorical answers.

Clustering Algorithm

Our precise method of clustering can be explained concisely in one sentence; however it is ripe with terminology that many readers may be unfamiliar with. We therefore state the sentence and then explain the terminology:

> We used agglomerative hierarchical clustering with a Hamming distance function and an "average" linkage criterion.

Clustering is used to find groups of "similar" observations. Similarity is defined with respect to a *distance function*, in our case normalized Hamming distance.

Hierarchical clustering produces a tree whose leaves are observations and whose subtrees are clusters. Subtrees have a *height* between 0 and 1, and *shorter* subtrees are tighter clusters (i.e., shorter subtrees have smaller distances between their leaves).

Agglomerative hierarchical clustering computes the tree bottom-up. Initially, every observation is a singleton tree. Recursively, two of the most *similar* clusters are joined to form a new tree, whose height is their *dissimilarity*. This metric of similarity between clusters is chosen, and is called the *linkage criterion*. We used the *average* distance between individual observations as the linkage criterion because it seemed more appropriate than the alternatives (such as minimum or maximum distance between clusters).

We computed the plots using the `daisy` and `agnes` functions in R's `cluster` package [21].

Distance Function

We stated that we used Hamming distance. Technically, the library function we invoked uses Gower's distance function [12, equation (5)], but in our case these are equivalent. The exact formula is:

$$d_{ij} = \frac{\sum_{k=1}^{n} s_{ijk}\,\delta_{ijk} w_k}{\sum_{k=1}^{n} \delta_{ijk} w_k}$$

In our case:

- d_{ij} is the computed distance (i.e., dissimilarity) between the answers given by the i'th and j'th workers.
- n is the number of questions.
- s_{ijk} is 1 if worker i and worker j disagreed on question k, or 0 otherwise.
- δ_{ijk} is used to handle questions that were not answered. None of our questions were optional, so $\delta_{ijk} = 1$.
- w_k is the relative weight given to the k'th question. We weighted each question equally, so $w_k = 1$.

Thus, for us d_{ij} is simply the fraction of questions on which workers i and j disagreed:

$$d_{ij} = \frac{\sum_{k=1}^{n} s_{ijk}}{n}$$

Equivalently, this is the Hamming distance [13] divided by the number of questions.