# I Speak, You Verify: Toward Trustworthy Neural Program Synthesis

**Anonymous authors**
Paper under double-blind review

## Abstract

We develop an approach for improving the trustworthiness and overall accuracy of program synthesizers based on large language models for source code. Given a natural language description of a programming problem, our method samples both candidate programs as well as candidate predicates specifying how the program should behave. We learn to analyze the agreement between programs and predicates to judge both which program is most likely to be correct, and also judge whether the language model is able to solve the programming problem in the first place. This latter capacity allows favoring high precision over broad recall: fostering trust by only proposing a program when the system is certain that it is correct.

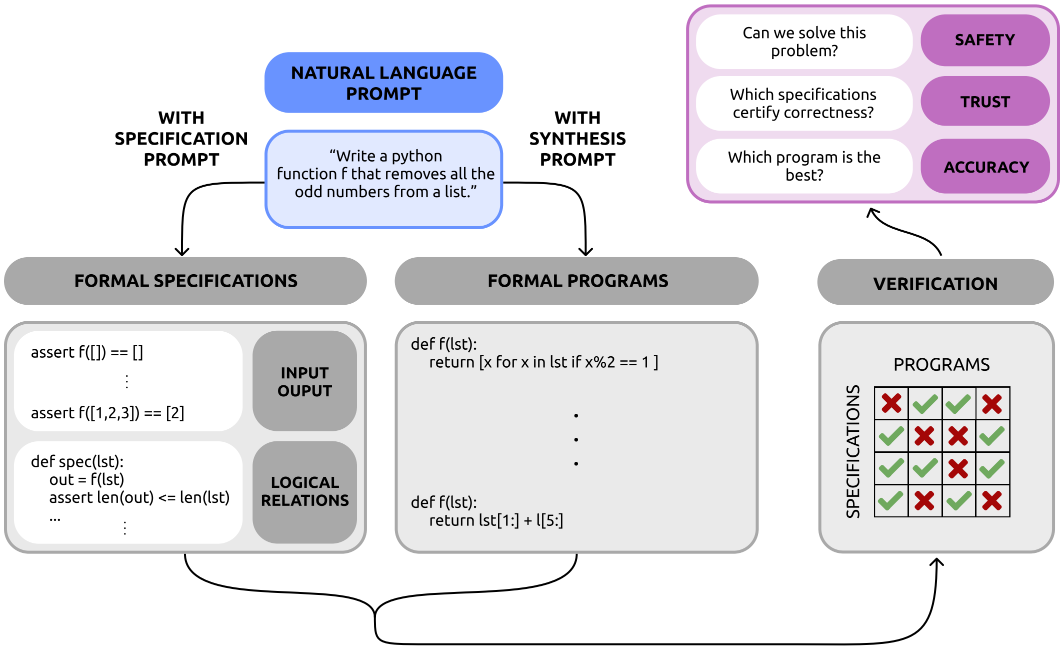
## 1 The importance of trust

Picture a future where AI systems attempt to close GitHub issues by generating source code given only the natural language of the GitHub issue. Such systems might not come out next year, but when or if they ever do, they will likely leverage large neural network language models for source code (Chen et al., 2021; Austin et al., 2021). These neural systems are good, but not perfect. Suppose 75% of the time, such systems propose a correct fix to the GitHub issue. The other 25% of the time, they produce plausible looking code containing subtle bugs. Would you use this system?

Most engineers would be reluctant to use such a system, because it fails to build trust with the user. When it fails, it cannot detect its own failure. When it succeeds, it cannot construct a human-comprehensible certificate of its success. In this paper we seek steps towards rectifying this lack of trust. Concretely, our goal is to build natural-language conditioned neural program synthesizers that are more trustworthy along several dimensions:

- We want systems that, when they cannot solve a programming problem, simply return no answer, rather than return a (possibly subtly) incorrect program. We conjecture that it is better to fall back on the human programmer, rather than risk introducing bugs. Contrast the situation with natural language translation: Unlike natural language, programs are brittle, and so must be exactly correct. And debugging bad code, unlike proofreading language, can be more difficult then just writing it yourself.
- We want systems that can produce a *human-understandable certificate* of the correctness of the synthesized code. This activity is common among human engineers, who often write test harnesses for new code. Similarly, our system proposes predicates testing its solutions, which act as a human-comprehensible signal of the code's (in)correctness.
- Ideally, trustworthy systems should be more accurate overall, solving more programming problems. This goal would seem to be in tension with the previous two. Surprisingly we find our methods for building trust also serve to boost overall accuracy on natural language to code generation problems as well.

Our high-level approach has a neural network propose candidate program solutions and independently propose predicates that correct solutions should satisfy, known as **specifications** ('specs', Fig. 1). We consider two kinds of specs: (1) input-output test cases, and (2) test harnesses specifying logical relations between inputs and outputs, also known as *functional specifications* (Solar-Lezama, 2008). In general, a spec can be any mechanically checkable property. We check the programs against the specs, and learn to use this checking to predict if the system knows how to solve the problem at all, and if so, which program(s) are probably the right solution. Intuitively, we ask the language model to 'check its work' by generating specs. We call our approach `speculyzer`, short for 'Specification Synthesizer', because in addition to synthesizing programs, it synthesizes specs.


NATURAL LANGUAGE PROMPT
"Write a python function f that removes all the odd numbers from a list."
WITH SPECIFICATION PROMPT
WITH SYNTHESIS PROMPT
Can we solve this problem?
SAFETY
Which specifications certify correctness?
TRUST
Which program is the best?
ACCURACY
FORMAL SPECIFICATIONS
FORMAL PROGRAMS
VERIFICATION
INPUT OUPUT
LOGICAL RELATIONS
PROGRAMS
SPECIFICATIONS

```
assert f([]) == []
assert f([1,2,3]) == [2]
```

```
def spec(lst):
    out = f(lst)
    assert len(out) <= len(lst)
    ...
```

```
def f(lst):
    return [x for x in lst if x%2 == 1 ]
```

```
def f(lst):
    return lst[1:] + l[5:]
```

Figure 1: Our `speculyzer` system inputs a natural language description of a programming problem. It uses large language models to independently sample candidate programs, and candidate specifications of what the program should do. Because natural language is informal, we cannot verify programs against it, but logical relations and input-outputs can be mechanically checked against. The result of this verification is fed to a learned model which predicts whether the problem can be solved; if so, which program is correct; and which specs best certify whether that program is correct or incorrect.

## 2 Related Work

**Program synthesis.** Automatically constructing software has been a longstanding goal of computer science (Manna & Waldinger, 1979; Gulwani et al., 2017). Classic program synthesizers input a formal specification of what the program should do, and then either search or logically derive a program guaranteed to satisfy that formal specification (Alur et al., 2013). This formal specification could come from a rich, expressive logic (e.g. Polikarpova et al. (2016)) or less precise, but still formal modalities such as input-output examples (Gulwani, 2011). Classic program synthesizers assume it is possible to verify the correctness of a candidate program. This verification assumption allows a generate-and-test approach to always eventually find a satisfying program, though practical program synthesizers are more clever in how they search for programs (Solar-Lezama, 2008), including incorporating guidance from neural networks (Chaudhuri et al., 2021; Ellis et al., 2021).

**Large language models for source code.** Our work uses large language models for source code (Chen et al., 2021; Austin et al., 2021). These neural networks generate source code conditioned or 'prompted' by a mix of code and natural language (the natural language is usually represented as code comments). Such language models are typically implemented as very large transformers (Vaswani et al., 2017; Brown et al., 2020).

Following the introduction of large transformer-based language models for source code, there has been work on how to boost the accuracy of those models. Here, accuracy means the probability of sampling a correct program conditioned on a natural-language prompt. Accuracy is often measured by functional correctness with the *pass@k* metric, which considers drawing $k$ IID samples from the language model and testing if any of those samples pass a set of holdout test cases. Toward boosting *pass@k*, researchers have considered *clustering* sampled programs according to the outputs they produce on test inputs (Shi et al., 2022; Li et al., 2022). For example, AlphaCode prioritizes large 'clusters' of samples with the exact same input-output behavior (Li et al., 2022), effectively reranking the samples from the language model according to how likely they are to solve the task. A complementary reranking strategy is to train a second neural network to predict program correctness, as explored in Inala et al. (2022). Another approach is to ask the language model to 'show its work' by prompting it to generate/use intermediate evaluation states, known as 'chain-of-thought-prompting' (Wei et al., 2022) and 'scratch pads' (Nye et al., 2021). Our goal of having our model not make predictions when it doesn't think it can get it right is related to the recent Grammformer introduced by Guo et al. (2022), which is a Codex-like model outputting regex patterns containing wildcards where the model is uncertain.

The closest work to ours is the concurrently developed CodeT system (Chen et al., 2022). CodeT independently proposed generating programs as well as input-output test cases, with the goal of boosting *pass@k*. The qualitative difference between our systems is that we designed `speculyzer` to build trust by synthesizing specifications–only boosting *pass@k* as a side effect–and incorporated input-output test cases as a special case of specs in general.

Engineering safe, trustworthy language models has received considerable attention by the AI safety (Thoppilan et al., 2022) and AI alignment communities (Kadavath et al., 2022). These works find that one can train classifiers which predict the truthfulness or safety of language outputs by inspecting the hidden activations of the model or even by simply 'asking' the model if its output is correct or safe. We see this family of efforts as complementary: For programs, it is possible to formally specify correctness properties, which is not generally true in NLP, so we focus on formal properties (specifications) here. Nonetheless, one can train statistical predictors of program correctness (Inala et al., 2022), and in fact these synergize with formal notions of correctness (Chen et al., 2022). Broadly however, we think that program synthesis offers unique opportunities for building trust through symbolic methods. Although statistically reranking language model outputs via a second neural network improves raw performance, we believe it is a suboptimal trust-builder: an inscrutable neural network cannot guarantee the correctness of another inscrutable network. Here we advocate that properties which are symbolically verifiable and human-comprehensible should play a role, and examine certain specifications as basic examples of such properties.

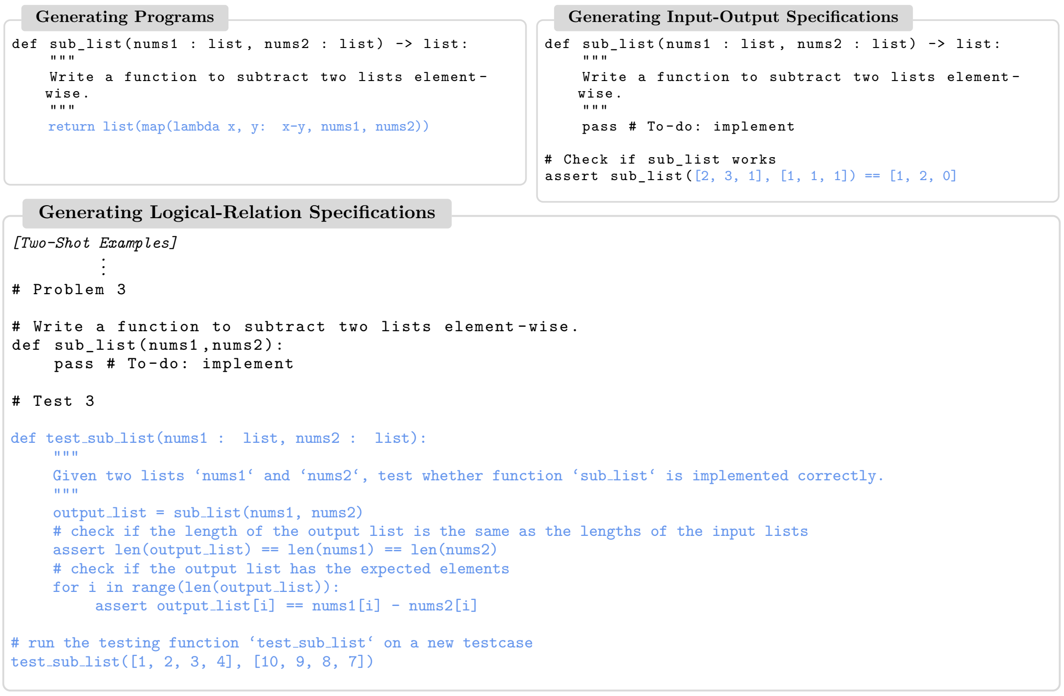
**Verification.** Specifying and certifying the correctness of software is the traditional goal of formal verification methods (Pierce et al., 2022; Baier & Katoen, 2008). We seek trust in slightly different ways: There are no prospects of truly verifying against natural language, so we use less precise, but more human-understandable, kinds of specifications. Rather than specifying exact program semantics in a rich logic, we use unit-test harnesses. A language model generates inputs on which to run those harnesses, instead of verifying across all possible inputs. In principle nothing precludes applying more sophisticated verification techniques to the specifications our system elicits. See Appendix A.2 for further discussion.

## 3 Methods

Given a natural-language prompt describing a programming problem, our goal is to construct a ranked list of candidate program solutions, *or* to output an empty list whenever the system cannot solve the programming problem. Our approach independently samples a set of candidate programs $\mathcal{P}$ and a set of candidate specs $\mathcal{S}$. Specs can be either input-output testcases, or logical relations (Fig. 1). We write $\mathcal{T}$ for the set of test cases and $\mathcal{R}$ for the set of logical relations, so $\mathcal{S} = \mathcal{T} \cup \mathcal{R}$. Each program $p \in \mathcal{P}$ is checked against each spec $s \in \mathcal{S}$, and basic statistics of program-spec agreement are computed. These statistics are aggregated by a learned model into a **confidence score** for each program. Programs whose confidence score falls below a threshold are discarded. Any remaining programs are sorted by confidence score and returned to the user as possible solutions, together with certain specs they pass. Returning specifications allows the user to verify that the code has the intended behavior. This architecture lets the system learn how to predict when it cannot solve a problem, and also learn to rank candidate solutions and their corresponding specs.

### 3.1 Sampling programs and tests

Given a string `prompt` describing a programming problem, we sample $n = 100$ candidate programs (the set $\mathcal{P}$) and candidate specs (the set $\mathcal{S}$). Both sets are sampled using a pretrained language model, which can probabilistically generate program source code conditioned on a prompt. We write $P^{\text{LM}}(\cdot|\texttt{prompt})$ for the conditional distribution over programs, given `prompt`. If a program $p \in \mathcal{P}$, then $p \sim P^{\text{LM}}(\cdot|\texttt{prompt})$. To sample specs, we deterministically transform the prompt as in Fig. 2 and Appendix A.10, then draw iid samples from the language model to construct relations $\mathcal{R}$ and input-output test cases $\mathcal{T}$.



**Generating Programs**

```
def sub_list(nums1 : list, nums2 : list) -> list:
    """
    Write a function to subtract two lists element-
   wise.
    """
    return list(map(lambda x, y:  x-y, nums1, nums2))
```

**Generating Input-Output Specifications**

```
def sub_list(nums1 : list, nums2 : list) -> list:
    """
    Write a function to subtract two lists element-
   wise.
    """
    pass # To-do: implement

# Check if sub_list works
assert sub_list([2, 3, 1], [1, 1, 1]) == [1, 2, 0]
```

**Generating Logical-Relation Specifications**

```
[Two-Shot Examples]
        ⋮
# Problem 3

# Write a function to subtract two lists element-wise.
def sub_list(nums1,nums2):
    pass # To-do: implement

# Test 3

def test_sub_list(nums1 :  list, nums2 :  list):
    """
    Given two lists ‘nums1‘ and ‘nums2‘, test whether function ‘sub_list‘ is implemented correctly.
    """
    output_list = sub_list(nums1, nums2)
    # check if the length of the output list is the same as the lengths of the input lists
    assert len(output_list) == len(nums1) == len(nums2)
    # check if the output list has the expected elements
    for i in range(len(output_list)):
        assert output_list[i] == nums1[i] - nums2[i]

# run the testing function ‘test_sub_list‘ on a new testcase
test_sub_list([1, 2, 3, 4], [10, 9, 8, 7])
```

Figure 2: Our systems uses different prompts to generate programs, input-output tests, and logical relations. Here we also show the example completion from the model in blue.

### 3.2 Scoring and analyzing test coverage

Given programs $\mathcal{P}$ and specs $\mathcal{S}$, we compute a confidence score for each $p \in \mathcal{P}$ measuring how likely $p$ is correct. Assuming, on average, specs correctly formalize the informal natural-language intention, satisfying more specs should increase our confidence in a program.

Additionally, if many sampled programs exhibit identical behavior on the specs, then we should increase our confidence in those programs, because this indicates high marginal probability of that behavior under $P^{\text{LM}}(\cdot|\texttt{prompt})$. This ‘clustering’ of candidate solutions according to their execution behavior, and prioritizing large clusters, has been successfully used by AlphaCode (Li et al., 2022), Minerva (Lewkowycz et al., 2022), and others (Shi et al., 2022). It is also related to *observational equivalence* from classic program synthesis (Udupa et al., 2013), which treats programs as identical if they have the same outputs on test inputs.

While cluster size and spec pass rate give confidence in individual programs, global features of the distribution of sampled programs can indicate whether this system might be able to solve the problem in the first place. So, we also compute the entropy over cluster assignments: diffuse clusterings could suggest lack of confidence.

Finally, we compute a real-valued score for each program $p \in \mathcal{P}$ using a logistic regressor over features $\phi(p, \mathcal{P}, \mathcal{S})$.[1] The features $\phi(p, \mathcal{P}, \mathcal{S})$ include **testcase pass rate** (fraction of input-output specifications passed), **relation pass rate** (fraction of logical-relation test-harness specifications passed), **cluster size** (fraction of other programs with the same behavior on the specifications), the ordinal rank[2] and logarithms of the preceding features, and the entropy of cluster assignment distributions for both input-output and logical-relation specifications, for a total of 18 features. We clarify the meaning of those features below:

$$\begin{aligned}
&\text{score}_\theta(p|\mathcal{P}, \mathcal{S}) = \theta \cdot \phi(p, \mathcal{P}, \mathcal{S}) + \theta_0 && \text{learned } \theta\text{, features } \phi \\
&\underline{\text{Components of } \phi(p, \mathcal{P}, \mathcal{S})\text{, plus logs and ordinal ranks:}} && \qquad (1) \\
&\text{testPass}(p, \mathcal{P}, \mathcal{T} \cup \mathcal{R}) = \frac{1}{|\mathcal{T}|} \sum_{s \in \mathcal{T}} \mathbb{1}\left[p \vdash s\right] && p \vdash t \text{ means prog. } p \text{ satisfies } s \\
&\text{relationPass}(p, \mathcal{P}, \mathcal{T} \cup \mathcal{R}) = \frac{1}{|\mathcal{R}|} \sum_{s \in \mathcal{R}} \mathbb{1}\left[p \vdash s\right] && \\
&\text{clusterSize}(p, \mathcal{P}, \mathcal{S}) = \sum_{p' \in \mathcal{P}} \prod_{s \in \mathcal{S}} \mathbb{1}\left[(p \vdash s) = (p' \vdash s)\right] && \#\text{ progs. w/ same spec behavior} \\
&\text{clusterEntropy}(p, \mathcal{P}, \mathcal{S}) = \mathbb{H}\left[A\right] && \text{cluster assignment } A : \mathcal{S} \rightarrow \{0, 1\} \\
&\qquad \text{where } \mathbb{P}\left[A\right] \propto \sum_{p' \in \mathcal{P}} \prod_{s \in \mathcal{S}} \mathbb{1}\left[(p \vdash s) = A(s)\right] && \text{cluster assignment distribution}
\end{aligned}$$

We fit $\theta$ via maximum likelihood on a corpus $\mathcal{D}$ containing triples $\langle \mathcal{P}, \mathcal{S}, \mathcal{G} \rangle$ of programs $\mathcal{P}$ and specifications $\mathcal{S}$, both sampled from the same prompt, and ground-truth testcases $\mathcal{G}$, which serve as a proxy for program correctness. The ground-truth testcases $\mathcal{G}$ are assumed to be unavailable at test time, because our goal is synthesis from informal specifications like natural language. We use gradient ascent to maximize the log likelihood, $\mathcal{L}$:

$$\mathcal{L} = \sum_{\substack{\langle \mathcal{P}, \mathcal{S}, \mathcal{G} \rangle \in \mathcal{D} \\ p \in \mathcal{P}}} \mathbb{1}\left[p \vdash \mathcal{G}\right] \log \sigma\left(\text{score}_\theta(p|\mathcal{P}, \mathcal{S})\right) + \mathbb{1}\left[p \nvdash \mathcal{G}\right] \log\left(1 - \sigma\left(\text{score}_\theta(p|\mathcal{P}, \mathcal{S})\right)\right) \qquad (2)$$

where $\sigma(\cdot)$ is the logistic sigmoid function.

### 3.3 Test time metrics

**Precision-Recall.** Ultimately our goal is a trustworthy system that proposes program solutions whenever it can, but avoids proposing buggy code. Toward those ends, we seek high *precision* without sacrificing *recall*. High precision means that when the system suggests a program, it is probably correct. Precise systems foster trust because they don't propose wrong answers, though they may decline to provide an answer in the first place. High recall means a correct program achieves the top rank: In other words, the system can solve a lot of programming problems, though it might make more mistakes in the process.

The tradeoff between precision and recall can be tuned by a thresholding parameter, $\tau$. A candidate program is discarded if its score falls below the threshold $\tau$. If all programs are discarded, the system declines to provide an output for the programming problem, and otherwise the system outputs a ranked list of programs sorted by score.

We define Precision@$k$ and Recall@$k$, which respectively measure (1) whether a correct program is in the top $k$ whenever any program scores above $\tau$ and (2) how often a correct

[1]We also tried a small multilayer perceptron, which underperformed logistic regression (A.7)
[2]Ordinal rank compared to other clusters

program scoring above $\tau$ is in the top $k$:

$$\text{Precision@}k = \frac{\text{TruePositives@}k}{\text{PredictedPositives}} \qquad \text{Recall@}k = \frac{\text{TruePositives@}k}{\text{ActualPositives}} \tag{3}$$

$$\text{TruePositives@}k = \sum_{\langle \mathcal{P},\mathcal{S},\mathcal{G}\rangle \in \mathcal{D}} \mathbb{1}\left[\begin{array}{ll} \exists p \in \mathcal{P}: & p \vdash \mathcal{G} \text{ and } \tau \leq \text{score}_\theta(p|\mathcal{P},\mathcal{S}) \text{ and} \\ & p \in \text{top-k}_{p' \in \mathcal{P}}\text{score}(p'|\mathcal{P},\mathcal{S}) \end{array}\right] \tag{4}$$

$$\text{PredictedPositives} = \sum_{\langle \mathcal{P},\mathcal{S},\mathcal{G}\rangle \in \mathcal{D}} \mathbb{1}\left[\exists p \in \mathcal{P}:\ \tau \leq \text{score}_\theta(p|\mathcal{P},\mathcal{S})\right] \tag{5}$$

$$\text{ActualPositives} = \sum_{\langle \mathcal{P},\mathcal{S},\mathcal{G}\rangle \in \mathcal{D}} \mathbb{1}\left[\exists p \in \mathcal{P}:\ p \vdash \mathcal{G}\right] \tag{6}$$

We sweep possible values for $\tau$ to compute a precision-recall curve. Generically, there is no 'true' best trade-off between these desiderata.

**Pass rate.** The *pass@k* metric (Austin et al., 2021; Chen et al., 2021) measures the probability of $k$ samples from $P^{\text{LM}}(\cdot|\texttt{prompt})$ passing the ground-truth test cases, $\mathcal{G}$:

$$\textit{pass@k} = \mathbb{E}_{p_1,\cdots,p_k \sim P^{\text{LM}}(\cdot|\texttt{prompt})} \mathbb{1}\left[\exists p_i :\ p_i \vdash \mathcal{G}\right] \tag{7}$$

Note that *pass@k* is proportional to ActualPositives (Eq. 6): The (fraction of) problems where there is at least one correct answer in the sampled programs.

It is also conventional to combine *pass@k* with a scoring function that reranks the sampled programs. This generalizes *pass@k* to *pass@k,n*, which measures the probability that, after generating $n$ candidate programs, a correct one is in the top-$k$ under our scoring function:

$$\textit{pass@k,n} = \mathbb{E}_{\langle \mathcal{P},\mathcal{T},\mathcal{G}\rangle \sim \mathcal{D}} \mathbb{1}\left[\exists p \in \text{top-k}_{p' \in \mathcal{P}}\text{score}_\theta(p'|\mathcal{P},\mathcal{T}) \text{ where } p \vdash \mathcal{G}\right] \tag{8}$$

**Ranking and clustering.** When the programs are grouped into clusters, we can also define *pass@k,n* by ranking the clusters and predicting a program from each of the top $k$ clusters. This works well when all programs in each cluster have the exact same score, and when clusters tend to be either 100% correct or 0% correct. We report *pass@k,n*, when reranking clusters, but also analyze scoring/reranking individual programs in Appendix A.8.

## 4 Results

We study our approach on two popular datasets and use the Codex Davinci model (Chen et al., 2021), seeking to answer the following research questions:

- How does our learned reranking impact raw rate of success (*pass@k,n*)?
- How trustworthy and safe can we make the system (precision), and how much does that require sacrificing coverage (recall)?
- How does our learned scoring function generalize across datasets?
- How can we use the synthesized specifications to certify program correctness?

We evaluate on programming problems from the Mostly Basic Python Problems (MBPP:Austin et al. (2021), sanitized version) and HumanEval datasets (Chen et al., 2021). Each of these datasets contains natural language descriptions of programming problems, and holdout tests to judge program correctness. An important difference between them is that HumanEval sometimes includes example input-outputs as part of the natural language description, while MBPP does not. Having I/O examples in the problem description makes spec generation easier: some specs are given for free. On the other hand, humans sometimes spontaneously mix natural language and examples (Acquaviva et al., 2021). Therefore, using both MBPP and HumanEval gives a more robust evaluation, but we note this qualitative difference between them. Appendix Sec. A.1 gives further experimental setup details, such as hyperparameters and example prompts.

### 4.1 Raw accuracy improvement from reranking

To understand how well `speculyzer` learns to predict the best program—independent of predicting when it doesn't know the answer—we measure *pass@k,n* (Fig. 3). We assess our system using cross validation, and consider ablations (1) using only input-output testcases; and (2) using only logical relations. We use as baselines (1) AlphaCode's ranking function, which ranks based on cluster size ("cluster"); (2) CodeT's ranking function, which ranks based on (testcase pass rate)$\sqrt{\text{cluster size}}$; (3) a random baseline, which ranks all programs equally; and (4) an oracle, which always chooses a correct programs, if it exists. We also quote numbers from Inala et al. (2022) (which ranks programs using a separate neural net) and Chen et al. (2022) (CodeT). We see that both varieties of specification are valuable, but that input-outputs work better on their own than logical relations on their own.

Overall, `speculyzer` achieves 78.05% *pass@1* on HumanEval and 67.13% *pass@1* on MBPP. To the best of our knowledge this is the highest *pass@1* rate reported so far on HumanEval (12% absolute improvement over CodeT, 65.8%→78.05%), and within 1% of the best reported number on MBPP (Chen et al., 2022); it also represents a larger improvement relative to Inala et al. (2022). Directly comparing these numbers is subtle however, because all three works (including ours) use different sampling parameters (Holtzman et al., 2019), causing Inala et al. (2022) to underperform our random baseline and Chen et al. (2022) to outperform our oracle at $k = 10$. Hence we replicate CodeT using our sampling parameters.

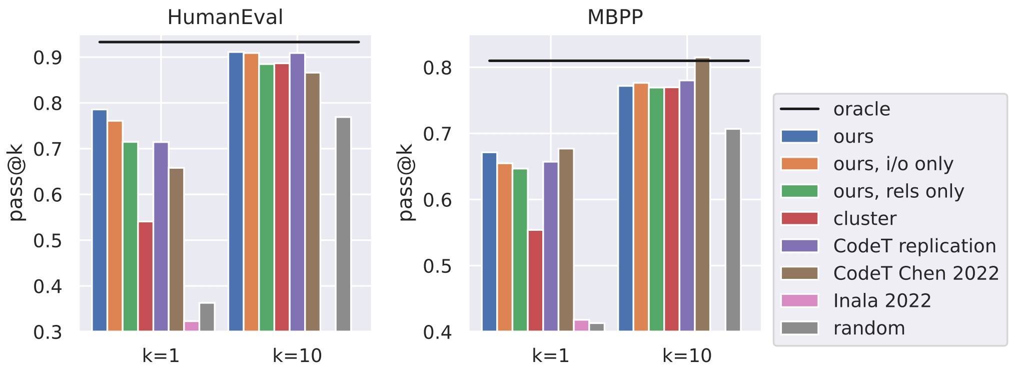
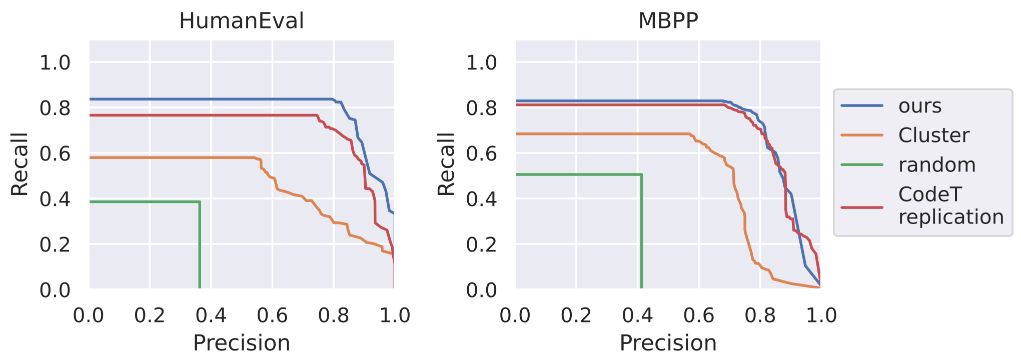


Figure 3: Solve rate of different models while varying $k$, the # guesses allowed per problem. Calculated using $n = 100$ samples. Inala et al. (2022) does not report $k = 10$.

### 4.2 Trading off between trustworthiness and coverage

Trustworthy systems should avoid predicting any programs at all when they cannot solve a problem. Doing so increases precision, the fraction of outputs which are correct, but at the expense of recall, the fraction of solvable problems where we output a correct solution. Fig. 4 illustrates how one can adjust this trade-off. For example, we can achieve 100% precision on HumanEval (*zero* error rate), in exchange for dropping our recall from 82% to 33%. Note this zero error rate does not come from our learned score function memorizing the data: we use cross validation to test each program using weights trained on other folds. Less extreme tradeoffs are possible, such as 90% precision in exchange for 51% recall.

Our method quantitatively outperforms the alternatives on precision/recall statistics (Fig. 4, bottom), but as with *pass@k*, our advantage is largest on HumanEval. We hypothesize that this is because HumanEval is a much cleaner dataset compared to MBPP, and not because HumanEval often includes input-outputs in the prompt. Our CodeT replication controls for that difference across datasets, because it too sees input-outputs in HumanEval prompts, yet it underperforms `speculyzer` by a wider margin on HumanEval compared to MBPP. Although the best balance between precision and recall depends on the downstream application, we hope that our study invites further progress on systems that optimize this trade off, in addition to optimizing the popular *pass@k*.



| | AUC | max F1 | R@ P=.9 | R@ P=1 |
|---|---|---|---|---|
| ours | **0.79** | **0.82** | **0.51** | **0.33** |
| CodeT | 0.68 | 0.75 | 0.44 | 0.19 |
| Cluster | 0.47 | 0.57 | 0.21 | 0.16 |
| random | 0.14 | 0.37 | 0.00 | 0.00 |

| | AUC | max F1 | R@ P=.9 | R@ P=1 |
|---|---|---|---|---|
| ours | **0.73** | **0.78** | **0.42** | **0.014** |
| CodeT | 0.71 | 0.76 | 0.31 | 0.001 |
| Cluster | 0.51 | 0.63 | 0.006 | 0.006 |
| random | 0.21 | 0.45 | 0.00 | 0.00 |

Figure 4: Top: Precision-Recall curves with $k = 1$. Bottom: Statistics of these curves, measuring Area Under Curve (AUC), max F1 (harmonic mean of precision and recall), recall in the high-trust regime: R@P=.9 is recall when precision=90%, and R@P=1 is recall at perfect precision. Left: HumanEval. Right: MBPP.

## 4.3 Generalization across datasets

Unlike recent heuristics for reranking solutions proposed by a large language model, our scheme involves learning real-valued parameters ($\theta$ in Eq. 2). To understand how learned parameters generalize across datasets, we compute the *pass@1* rate and precision-recall stats for models trained on MBPP, but tested on HumanEval (and vice versa). These statistics are essentially unchanged by training on different datasets (Fig. 5), varying by 4% absolute and 6% relative, indicating generalization across similar, but not identical, data distributions.

| | test on HumanEval | | test on MBPP | |
|---|---|---|---|---|
| | train HumanEval | train MBPP | train HumanEval | train MBPP |
| pass@1 | 0.78 | 0.76 | 0.65 | 0.67 |
| AUC | 0.79 | 0.75 | 0.70 | 0.73 |
| max F1 | 0.82 | 0.80 | 0.75 | 0.78 |

Figure 5: Comparing generalization when test and train data are drawn from the same corpus of problems (evaluated with cross validation), vs. transferred across corpora.

## 4.4 Certifying (In)correctness

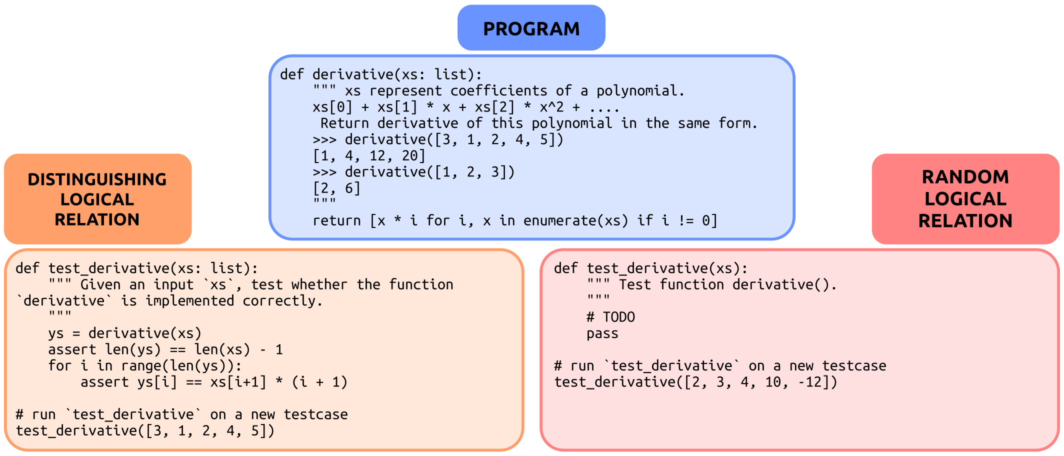
No natural language program synthesizer will always produce correct programs: Therefore, the system needs to communicate what a synthesized program $p$ computes, so that the user can confidently accept or discard it. `speculyzer` does this by outputting a specification that certifies $p$'s (in)correctness while being maximally informative as to $p$'s behavior.

Whenever `speculyzer` ranks $p^* \in P$ as the best solution to a problem, it selects a spec $s^* \in \mathcal{S}$ to certify the behavior of $p^*$. The certificate $s^*$ must be a true fact about $p^*$, so $p^* \vdash s^*$, but should also constrain the behavior of $p^*$. For example, the specification $\forall x : p^*(x) = p^*(x)$ is vacuously true for any $p^*$, and so makes a poor certificate.

We formalize this as a rational-communication model of program synthesis (Pu et al., 2020), which means first defining a joint probability distribution over programs and specifications: $\mathbb{P}[p, s] \propto \mathbb{1}[p \vdash s]\, \mathbb{1}[p \in \mathcal{P}]\, \mathbb{1}[s \in \mathcal{S}]$. Then, we score each specification $s$ by the conditional probability of $p^*$ given $s$, i.e. $\mathbb{P}[p^*|s]$. Applying Bayes' Rule and simplifying, we find that this is equivalent to ranking specs by how few other programs satisfy them, i.e. their selectivity:

$$s^* = \arg\max_{s \in \mathcal{S}} \mathbb{P}[p^*|s] = \arg\min_{\substack{s \in \mathcal{S} \\ p^* \vdash s}} \sum_{p \in \mathcal{P}} \mathbb{1}[p \vdash s] \tag{9}$$

Fig. 6 illustrates a representative programming problem and its top specification compared with a random specification. Appendix A.11 illustrates 20 further examples.



```
def derivative(xs: list):
    """ xs represent coefficients of a polynomial.
    xs[0] + xs[1] * x + xs[2] * x^2 + ....
     Return derivative of this polynomial in the same form.
    >>> derivative([3, 1, 2, 4, 5])
    [1, 4, 12, 20]
    >>> derivative([1, 2, 3])
    [2, 6]
    """
    return [x * i for i, x in enumerate(xs) if i != 0]
```

```
def test_derivative(xs: list):
    """ Given an input `xs`, test whether the function
`derivative` is implemented correctly.
    """
    ys = derivative(xs)
    assert len(ys) == len(xs) - 1
    for i in range(len(ys)):
        assert ys[i] == xs[i+1] * (i + 1)

# run `test_derivative` on a new testcase
test_derivative([3, 1, 2, 4, 5])
```

```
def test_derivative(xs):
    """ Test function derivative().
    """
    # TODO
    pass

# run `test_derivative` on a new testcase
test_derivative([2, 3, 4, 10, -12])
```

Figure 6: Representative certificate of program (in)correctness. Our probabilistic model favors selective specs as certificates, which we contrast with a random spec.

## 5 Contributions and Outlook

We have contributed a program synthesizer that learns to predict when it cannot solve a problem and learns to construct its own specifications that communicate what each program does. We intend for these elements of `speculyzer` to increase the trust and safety of neural program synthesis and to serve as a modest step toward program synthesizers that could better collaborate with software engineers. For this idea of better human-machine collaboration, writing correct code and documenting its correctness are paramount. In the process, we have also improved the state-of-the-art *pass@k* accuracy for the HumanEval dataset, and we have laid out basic trust and safety statistics, namely recall at high precision, which we hope springboards further investigation in language model safety for source code.

Our work has important limitations. Because `speculyzer` wraps around a large language model, it inherits some of their limitations, such as expensive sampling times. Since we also run the executed code, we incur additional cost and impose security risks if that execution is not appropriately sandboxed. Fundamentally, an approach like ours can never truly provide the same level of trust as a classic program synthesizer working from human-crafted formal-logic specifications; however, formal logic is less accessible than natural language.

Many directions remain open. Conceptually, the idea of formal specifications as a liaison between programs and informal natural language opens up the possibility of using richer kinds of specs and verifiers. This would allow tapping many years of effort from the programming languages community (D'silva et al., 2008; Baldoni et al., 2018), at least if we can interface such formalisms with large language models. Using a sophisticated verifier instead of executing candidate programs could also address the security concerns and performance hit from our additional code executions. Another direction is to combine our ideas with recent advances in HCI for program synthesis, such as Peleg et al. (2020), which develops powerful human interaction paradigms for program synthesis.

## References

Samuel Acquaviva, Yewen Pu, Marta Kryven, Catherine Wong, Gabrielle E. Ecanow, Maxwell I. Nye, Theodoros Sechopoulos, Michael Henry Tessler, and Joshua B. Tenenbaum. Communicating natural programs to humans and machines. *CoRR*, abs/2106.07824, 2021. URL `https://arxiv.org/abs/2106.07824`.

Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. *Syntax-guided synthesis.* IEEE, 2013.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Christel Baier and Joost-Pieter Katoen. *Principles of model checking.* MIT press, 2008.

Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 51 (3):1–39, 2018.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *CoRR*, abs/2005.14165, 2020. URL `https://arxiv.org/abs/2005.14165`.

Swarat Chaudhuri, Kevin Ellis, Oleksandr Polozov, Rishabh Singh, Armando Solar-Lezama, Yisong Yue, et al. Neurosymbolic programming. *Foundations and Trends® in Programming Languages*, 7(3):158–243, 2021.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397*, 2022.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021. URL `https://arxiv.org/abs/2107.03374`.

Vijay D'silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7):1165–1178, 2008.

Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sablé-Meyer, Lucas Morales, Luke Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B Tenenbaum. Dreamcoder: Bootstrapping inductive program synthesis with wake-sleep library learning. In *Proceedings of the 42nd acm sigplan international conference on programming language design and implementation*, pp. 835–850, 2021.

Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices*, 46(1):317–330, 2011.

Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119, 2017.

Daya Guo, Alexey Svyatkovskiy, Jian Yin, Nan Duan, Marc Brockschmidt, and Miltos Allamanis. Learning to complete code with sketches. In *ICLR 2022*, April 2022. URL `https://www.microsoft.com/en-us/research/publication/learning-to-complete-code-with-sketches/`.

Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration. In *International Conference on Learning Representations*, 2019.

Jeevana Priya Inala, Chenglong Wang, Mei Yang, Andres Codas, Mark Encarnación, Shuvendu K Lahiri, Madanlal Musuvathi, and Jianfeng Gao. Fault-aware neural code rankers. *arXiv preprint arXiv:2206.03865*, 2022.

Saurav Kadavath, Tom Conerly, Amanda Askell, Tom Henighan, Dawn Drain, Ethan Perez, Nicholas Schiefer, Zac Hatfield Dodds, Nova DasSarma, Eli Tran-Johnson, et al. Language models (mostly) know what they know. *arXiv preprint arXiv:2207.05221*, 2022.

Aitor Lewkowycz, Anders Andreassen, David Dohan, Ethan Dyer, Henryk Michalewski, Vinay Ramasesh, Ambrose Slone, Cem Anil, Imanol Schlag, Theo Gutman-Solo, et al. Solving quantitative reasoning problems with language models. *arXiv preprint arXiv:2206.14858*, 2022.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *arXiv preprint arXiv:2203.07814*, 2022.

Z. Manna and R. Waldinger. Synthesis: Dreams $\rightarrow$ programs. *IEEE Transactions on Software Engineering*, SE-5(4):294–328, 1979. doi: 10.1109/TSE.1979.234198.

Maxwell I. Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, Charles Sutton, and Augustus Odena. Show your work: Scratchpads for intermediate computation with language models. *CoRR*, abs/2112.00114, 2021. URL `https://arxiv.org/abs/2112.00114`.

Hila Peleg, Roi Gabay, Shachar Itzhaky, and Eran Yahav. Programming with a read-eval-synth loop. *Proc. ACM Program. Lang.*, 4(OOPSLA), nov 2020. doi: 10.1145/3428227. URL `https://doi.org/10.1145/3428227`.

Benjamin C Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Catalin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. Software foundations. electronic textbook. version 6.2, 2022.

John Platt et al. Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods. *Advances in large margin classifiers*, 10(3):61–74, 1999.

Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program synthesis from polymorphic refinement types. *ACM SIGPLAN Notices*, 51(6):522–538, 2016.

Yewen Pu, Kevin Ellis, Marta Kryven, Josh Tenenbaum, and Armando Solar-Lezama. Program synthesis with pragmatic communication. *Advances in Neural Information Processing Systems*, 33:13249–13259, 2020.

Freda Shi, Daniel Fried, Marjan Ghazvininejad, Luke Zettlemoyer, and Sida I Wang. Natural language to code translation with execution. *arXiv preprint arXiv:2204.11454*, 2022.

Armando Solar-Lezama. *Program synthesis by sketching*. PhD thesis, 2008.

Romal Thoppilan, Daniel De Freitas, Jamie Hall, Noam Shazeer, Apoorv Kulshreshtha, Heng-Tze Cheng, Alicia Jin, Taylor Bos, Leslie Baker, Yu Du, YaGuang Li, Hongrae Lee, Huaixiu Steven Zheng, Amin Ghafouri, Marcelo Menegali, Yanping Huang, Maxim Krikun, Dmitry Lepikhin, James Qin, Dehao Chen, Yuanzhong Xu, Zhifeng Chen, Adam Roberts, Maarten Bosma, Yanqi Zhou, Chung-Ching Chang, Igor Krivokon, Will Rusch, Marc Pickett, Kathleen S. Meier-Hellstern, Meredith Ringel Morris, Tulsee Doshi, Renelito Delos Santos, Toju Duke, Johnny Soraker, Ben Zevenbergen, Vinodkumar Prabhakaran, Mark Diaz, Ben Hutchinson, Kristen Olson, Alejandra Molina, Erin Hoffman-John, Josh Lee, Lora Aroyo, Ravi Rajakumar, Alena Butryna, Matthew Lamm, Viktoriya Kuzmina, Joe Fenton, Aaron Cohen, Rachel Bernstein, Ray Kurzweil, Blaise Aguera-Arcas, Claire Cui, Marian Croak, Ed H. Chi, and Quoc Le. Lamda: Language models for dialog applications. *CoRR*, abs/2201.08239, 2022. URL `https://arxiv.org/abs/2201.08239`.

Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M.K. Martin, and Rajeev Alur. Transit: Specifying protocols with concolic snippets. *SIGPLAN Not.*, 48(6):287–296, jun 2013. ISSN 0362-1340. doi: 10.1145/2499370.2462174. URL `https://doi.org/10.1145/2499370.2462174`.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (eds.), *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL `https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf`.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed H. Chi, Quoc Le, and Denny Zhou. Chain of thought prompting elicits reasoning in large language models. *CoRR*, abs/2201.11903, 2022. URL `https://arxiv.org/abs/2201.11903`.

# A Appendix

## A.1 Experimental setup

**Sampling from language models.** We used Codex Davinci with Temperature = 0.8 and TopP = 1 and a max-token size of 580 for our generation of programs and test cases for both HumanEval and MBPP. We used `"\ndef"`, `"\n#"`, `"\nclass"`, `"\nif"`, `"\nassert"`, and `"\nprint"` as stop tokens for our generation of programs and input-output test cases, and we used `"\n# Problem"` as the stop token for our generation of the logical relations test cases. We used zero-shot prompting for program and input-output test case generation, and few-shot prompting for the logical relations specifications generation.

**Logistic regressor.** We used the Adam optimizer with $10^{-3}$ as the learning rate and $10^{-4}$ as the weight decay. We used 10-fold cross-validation for in-domain testing (training on HumanEval and evaluating on HumanEval, training on MBPP and evaluating on MBPP), and we trained for 1500 epochs. We trained for 2000 epochs for domain generalization testing (training on HumanEval and evaluating on MBPP, training on HumanEval and evaluating on MBPP). For training, we standardize all input features to have a mean of 0 and a standard deviation of 1.

**Verification.** We verify input-output specifications to see if they hold for each program by executing the program on the input and comparing it with the output. Logical relations specifications require sample input(s) on which to test the relation; our prompt for logical relations causes the language model to construct such inputs, but in general one could use a fuzzer or verification tool. This causes verification of logical relation specifications to also reduce to program execution. We executed generated programs and test case in a Oracle Virtual Machine as a sandbox.

## A.2 Supplemental Discussion: Soundness & Completeness

The formal methods community has traditionally sought methods that are sound and complete Baier & Katoen (2008). Soundness means that whenever a program satisfies a spec, we identify it as such. Completeness means that every spec we identify as valid for a program is indeed entailed by that program. Generally, `speculyzer`'s verification step to overapproximates the set of specs a program satisfies, achieving completeness at the expense of soundness. This overapproximation only occurs for logical relations: we're sound and complete for input-outputs, because they assert a property of the program only on a single input. But because the language model constructs specific inputs on which to run the logical relations, we cannot assert that they hold generically for every single possible input.

The practical impact of this unsoundness for logical relations is that a human user has to inspect the candidate inputs on which the language model probes the relation. In practice, we find that the neural network generates fairly representative inputs (Appendix A.11), something that AlphaCode (Li et al., 2022) also found.

In principle, nothing precludes running a model checker or solver to check that the logical relations hold over every single possible input, which would make the method sound and complete w.r.t. the programs and specifications. However, this would not eliminate the need for a human to examine the AI-generated specifications: ultimately, the true specification is in natural language, and there is no sound and complete way of verifying against an informal language.

### A.3 Dataset Statistics

| | Input-Output | | Logical Relations | |
|---|---|---|---|---|
| | HumanEval | MBPP | HumanEval | MBPP |
| cluster size (# of test cases) | 4.55 | 4.27 | 4.46 | 5.02 |
| stddev | 10.64 | 10.75 | 10.15 | 12.10 |
| average # of test cases per program | 102.33 | 230.62 | 94.73 | 94.77 |
| stddev | 62.11 | 93.95 | 3.54 | 2.72 |
| % of programs that satisfy at least one test | 84.2% | 82.8% | 98.1% | 96.3% |

### A.4 Example Zero-Shot Prompts for program generation

For MBPP, to generate programs, we converted the natural language prompt to a function by adding in the prompt as a docstring for a function with the name of the function called in the ground-truth test cases. We used the HumanEval prompts as is.

Two examples of zero-shot prompts used for program generation are as follows:

#### A.4.1 HumanEval

First example:

```
def is_happy(s):
    """You are given a string s.
    Your task is to check if the string is happy or not.
    A string is happy if its length is at least 3 and every 3 consecutive
    ↪  letters are distinct
    For example:
    is_happy(a) => False
    is_happy(aa) => False
    is_happy(abcd) => True
    is_happy(aabb) => False
    is_happy(adb) => True
    is_happy(xyy) => False
    """
```

Second example:

```
def fix_spaces(text):
    """
    Given a string text, replace all spaces in it with underscores,
    and if a string has more than 2 consecutive spaces,
    then replace all consecutive spaces with -

    fix_spaces("Example") == "Example"
    fix_spaces("Example 1") == "Example_1"
    fix_spaces(" Example 2") == "_Example_2"
    fix_spaces(" Example   3") == "_Example-3"
    """
```

#### A.4.2 MBPP

First example:

```
def sum_range_list(list1 : list, m : int, n : int) -> int:
    """
    Write a function to find the sum of numbers in a list within a range
    ↪  specified by two indices.
    """
```

Second example:

```
def diff_even_odd(list1 : list) -> int:
    """
    Write a function to find the difference of the first even and first
    ↪  odd number of a given list.
    """
```

### A.5 Example Zero-Shot Prompts for input-output generation

We extracted input-output test cases by generating $n = 100$ times per HumanEval/MBPP prompt, then extracting each distinct single-line test case from each generation. We do this because each generation may produce multiple test cases, and we aimed to test each program on a single test case. For our test case prompts, we used the prompts to generate programs from MBPP and HumanEval, and we added in a `pass # To-do: implement` statement, a line with a comment asking Codex to `# Check if func_name works` and another line to asking Codex to `assert func_name(`.

#### A.5.1 HumanEval

First example:

```
def is_happy(s):
    """You are given a string s.
    Your task is to check if the string is happy or not.
    A string is happy if its length is at least 3 and every 3 consecutive
    ↪  letters are distinct
    For example:
    is_happy(a) => False
    is_happy(aa) => False
    is_happy(abcd) => True
    is_happy(aabb) => False
    is_happy(adb) => True
    is_happy(xyy) => False
    """

    pass # To-do: implement

# Check if is_happy works
assert is_happy(
```

Second example:

```
def fix_spaces(text):
    """
    Given a string text, replace all spaces in it with underscores,
    and if a string has more than 2 consecutive spaces,
    then replace all consecutive spaces with -

    fix_spaces("Example") == "Example"
    fix_spaces("Example 1") == "Example_1"
    fix_spaces(" Example 2") == "_Example_2"
    fix_spaces(" Example   3") == "_Example-3"
    """

    pass # To-do: implement
```

```
# Check if fix_spaces works
assert fix_spaces(
```

### A.5.2 MBPP

First example:

```
def sum_range_list(list1 : list, m : int, n : int) -> int:
    """
    Write a function to find the sum of numbers in a list within a range
    ↪  specified by two indices.
    """
    pass # To-do: implement

# Check if sum_range_list works
assert sum_range_list(
```

Second example:

```
def diff_even_odd(list1 : list) -> int:
    """
    Write a function to find the difference of the first even and first
    ↪  odd number of a given list.
    """
    pass # To-do: implement

# Check if diff_even_odd works
assert diff_even_odd(
```

## A.6 Few-Shot Prompt for logical relations spec generation

We use two-shot examples prompting to guide the model to tests various kinds of properties.

### A.6.1 HumanEval

```
# Problem 1

from typing import List

def filtered_even_integers(input_list: List([int]) -> List[int]:
    """ Given a list of integers, return a list that filters out the even
    ↪  integers.
    >>>  filtered_even_integers([1, 2, 3, 4])
    [1, 3]
    >>> filtered_even_integers([5, 4, 3, 2, 1])
    [5, 3, 1]
    >>> filtered_even_integers([10, 18, 20])
    []
    """
    # TODO
    pass

# Test 1

def test_filtered_even_integers(input_list: List()):
    """ Given an input `input_list`, test whether the function
    ↪  `filtered_even_integers` is implemented correctly.
    """
    output_list = filtered_even_integers(input_list)
    # check if the output list only contains odd integers
    for integer in output_list:
        assert integer % 2 == 1
    # check if all the integers in the output list can be found in the
    ↪  input list
    for integer in output_list:
        assert integer in input_list

# run the testing function `test_filtered_even_integers` on a  new testcase
test_filtered_even_integers([31, 24, 18, 99, 1000, 523, 901])

# Problem 2

def repeat_vowel(input_str: str) -> str:
    """ Return a string where the vowels (`a`, `e`, `i`, `o`, `u`, and
    ↪  their capital letters) are repeated twice in place.
    >>> repeat_vowel('abcdefg')
    'aabcdeefg'
    >>> repeat_vowel('Amy Emily Uber')
    'AAmy EEmiily UUbeer'
    """
    # TODO
    pass

# Test 2

def test_repeat_vowel(input_str: str) :
    """ Given an input `input_str`, test whether the function
    ↪  `repeat_vowel` is implemented correctly.
    """
    output_str = repeat_vowel(input_str)
    vowels = ['a', 'A', 'e', 'E', 'i', 'I', 'o', 'O', 'u', 'U']
    # check if the number of vowels in the  output string is doubled
    # First get the number of vowels in the input
    number_of_vowels_input = sum([input_str.count(vowel) for vowel in
    ↪  vowels])
    # Then get the number of vowels in the output
    number_of_vowels_output = sum([output_str.count(vowel) for vowel in
    ↪  vowels])
    assert number_of_vowels_input * 2 == number_of_vowels_output

# run the testing function `test_repeat_vowel` on a new testcase
test_repeat_vowe('ABCDEabcdeABCDE YOUUOY')
```

### A.6.2 MBPP

```
# Problem 1

from typing import List

# Given a list of integers, return a list that filters out the even
↪  integers.
def filtered_even_integers(input_list: List([int]) -> List[int]:
    pass # To-do: Implement

# Test 1

def test_filtered_even_integers(input_list: List()):
    """ Given an input `input_list`, test whether the function
    ↪  `filtered_even_integers` is implemented correctly.
    """
    output_list = filtered_even_integers(input_list)
    # check if the output list only contains odd integers
    for integer in output_list:
        assert integer % 2 == 1
    # check if all the integers in the output list can be found in the
    ↪  input list
    for integer in output_list:
        assert integer in input_list

# run the testing function `test_filtered_even_integers` on a  new testcase
test_filtered_even_integers([31, 24, 18, 99, 1000, 523, 901])

# Problem 2

# Return a string where the vowels (`a`, `e`, `i`, `o`, `u`, and their
↪  capital letters) are repeated twice in place
def repeat_vowel(input_str: str) -> str:
    pass # To-do: Implement

# Test 2

def test_repeat_vowel(input_str: str) :
    """ Given an input `input_str`, test whether the function
    ↪  `repeat_vowel` is implemented correctly.
    """
    output_str = repeat_vowel(input_str)
    vowels = ['a', 'A', 'e', 'E', 'i', 'I', 'o', 'O', 'u', 'U']
    # check if the number of vowels in the  output string is doubled
    # First get the number of vowels in the input
    number_of_vowels_input = sum([input_str.count(vowel) for vowel in
    ↪  vowels])
    # Then get the number of vowels in the output
    number_of_vowels_output = sum([output_str.count(vowel) for vowel in
    ↪  vowels])
    assert number_of_vowels_input * 2 == number_of_vowels_output

# run the testing function `test_repeat_vowel` on a new testcase
test_repeat_vowe('ABCDEabcdeABCDE YOUUOY')
```

### A.7 Multilayer Perceptron Model

In the main text we describe a linear model that computes confidence scores for each program based on features of the verification results. We also tried a multilayer perceptron with a single hidden layer and five hidden units: a small model, because we have a low dimensional problem with at most a few hundred training problems. As shown in Fig. 7, the multilayer perceptron does not actually do better on held out data, as measured by pass@k.

### A.8 Ranking individual programs vs ranking clusters

In the main text we describe *pass@k* results based on first clustering the programs according to which specifications they pass, and drawing $k$ programs from the top $k$ clusters. A simpler approach is to simply return the top $k$ ranked programs. Overall this gives inferior *pass@k* for $k > 1$, and by design has no effect when $k = 1$ (Fig. 8).

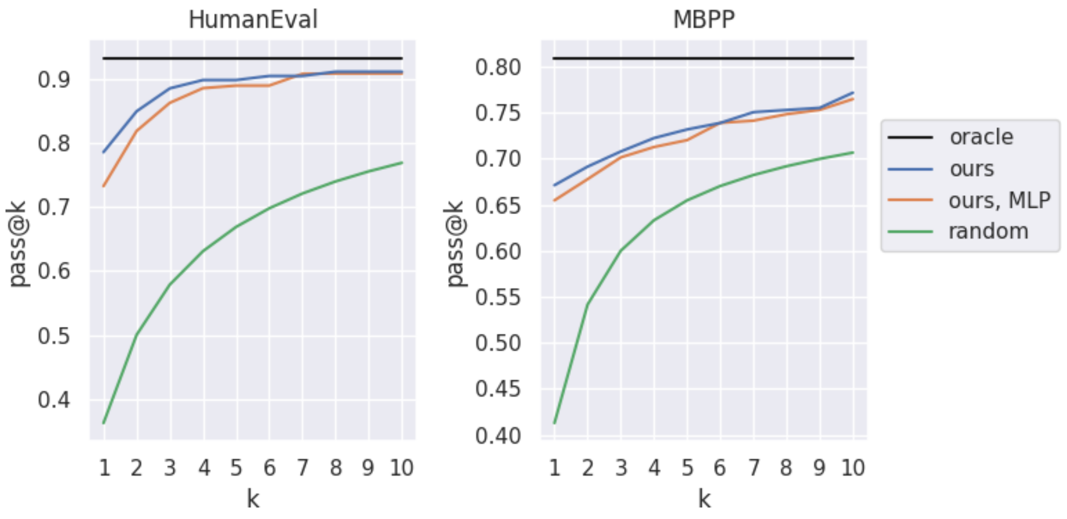


Figure 7: Comparing cross-validated pass@k for our linear model and a small MLP.

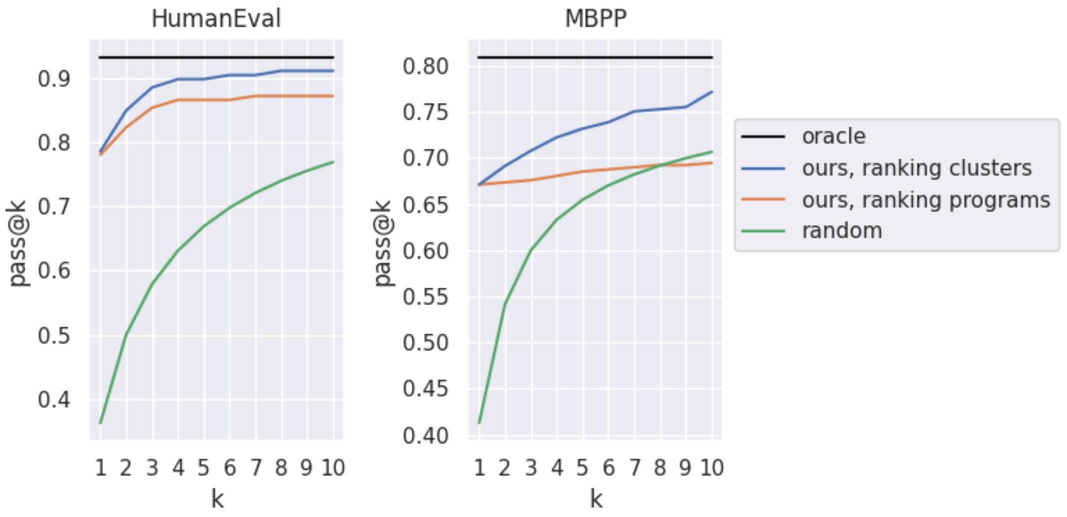


Figure 8: Comparing cross-validated pass@k for our model ranking clusters compared to our model ranking programs.

## A.9 Threshold Calibration

Our scoring function comes from logistic regression, which is a probabilistic discriminative model. Maximum likelihood training encourages it to be well-calibrated. Calibration means that the classifier not only discriminates positive/negative examples correctly but also, whenever it predicts probability $x$ of a positive label, approximately $x$ of the examples are actually positive examples. In our setting, this means when the model predicts 90% confidence that a program is correct, then about 90% of the time the program actually is correct. We experimentally confirmed this calibration property, which allows tuning the threshold $\tau$ to achieve the desired precision. The free parameter $\tau$ acts as a threshold on the confidence score needed to output a program. Because our scoring function comes from logistic regression, the threshold $\tau$ also acts as a threshold on how high the predicted probability that a program is correct has to be before we consider it to be a candidate solution. In particular, the logistic regressor predicts the probability of program correctness as $\sigma(\text{score})$, so thresholding score by $\tau$ corresponds to thresholding probability by $\sigma^{-1}(\tau)$. Thus, if our classifier is well-calibrated, we can set the threshold $\tau$ to the desired precision. Indeed, our model is out-of-the-box well-calibrated, as illustrated in Fig. 9.

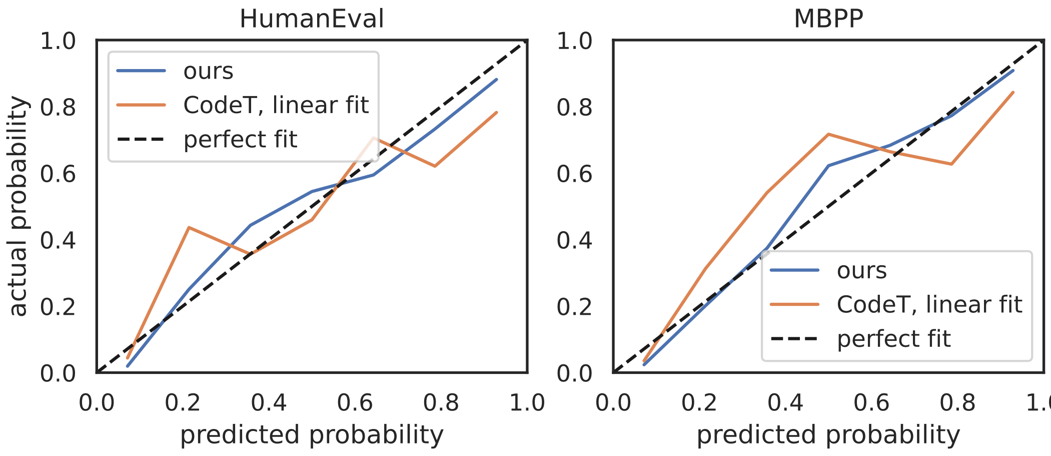


Figure 9: Comparing the calibration of our probabilistic scoring function against CodeT's scoring function. Note that, at high predicted certainty (rightward on horizontal axis), our method remains well calibrated, while the non-probabilistic scoring approach adopted by CodeT overestimates its success probability. `speculyzer` natively estimates probabilities, so we plot the raw output of our system. CodeT's ranking function is mapped to probabilities via the Platt transform–a linear mapping followed by sigmoid–which is standard practice for calibration curves (Platt et al., 1999). Because the Platt transform introduces two free parameters for its linear mapping, cross validation is used to estimate CodeT's calibration curve.

## A.10 Transformation of input problems to logical relations prompts

Here we show how to transform the input problem to the prompt used for generating logical relations.

**MBPP transformation** First we parse the input problems from MBPP dataset and get the string representation of library imports, function name, function parameters, return type, and English problem description. We denote them as `imports`, `func_name`, `parameter_format`, `return_type`, and `description` respectively.

Then we use the template shown in Figure 11 and Figure 10 for input-output and logical relations respectively. The parsed string from the input problem would then be inserted to the placeholder accordingly.

```
# Problem 3
{imports}
def {func_name}({", ".join(parameter_format)}) -> {return_type}:
    """
    {description}
    """
    pass # To-do: implement

# Test 3

def test_{func_name}(
```

Figure 10: Template for MBPP logical relation prompt

```
{imports}

def {func_name}({", ".join(parameter_format)}) -> {return_type}:
        """
        {description}
        """
        pass # To-do: implement

    # Check if {func_name} works
    assert {func_name}('
```

Figure 11: Template for MBPP input-output prompt

Then, for the logical relations, we prepend the resulting string with the few shot example string shown in A.6.2. For the input-output, we strip out the blank lines prefix if import is empty.

**HumanEval Transformation** Similar to the above MBPP transformation, we parse the input problems from HumanEval dataset and get the string representation of function definition plus English description and function name. We denote them as `function_definition_with_description` and `func_name`. Then we insert these into the template shown in Figure 12 and Figure 13 for input-output and logical relations respectively.

```
{function_definition_with_description}
    pass

# Check if {func_name} works
assert {func_name}(
```

Figure 12: Template for HumanEval input-output prompt

```
# Problem 3
{function_definition_with_description}
    pass # To-do: implement

# Test 3

def test_{func_name}(
```

Figure 13: Template for HumanEval logical relation prompt

Finally, we prepend the resulting string with the few shot example string shown in A.6.1.

## A.11 Examples of Top Certificates versus Random Certificates

Below we show example HumanEval problems, our top ranked sample, our top ranked specification certifying the (in)correctness of that sample, and a random specification. There are 20 such examples, randomly chosen; half show input-output specifications, and the other half show logical relations.

**Example synthesized program 1 (sample is correct)**

```
from typing import List


def mean_absolute_deviation(numbers: List[float]) -> float:
    """ For a given list of input numbers, calculate Mean Absolute
   Deviation
    around the mean of this dataset.
    Mean Absolute Deviation is the average absolute difference between
     each
    element and a centerpoint (mean in this case):
    MAD = average | x - x_mean |
    >>> mean_absolute_deviation([1.0, 2.0, 3.0, 4.0])
    1.0
    """
    mean = sum(numbers) / len(numbers)
    mad = 0
    for number in numbers:
        mad += abs(number - mean)
    return mad / len(numbers)
```

**Top ranked relational spec for program 1**

```
def test_mean_absolute_deviation(numbers: List[float]):
     """ Given an input list `numbers`, test whether the function `
    mean_absolute_deviation` is implemented correctly.
     """
     output = mean_absolute_deviation(numbers)
     # check if the result of mean_absolute_deviation is the same as
    numpy.mean(np.abs(numbers - np.mean(numbers)))
     try:
         import numpy as np
         # use this boolean variable to determine whether the test is
    passed
         test_pass = output == np.mean(np.abs(numbers - np.mean(numbers
    )))
     except ImportError:
         print('make sure you have numpy installed')
         # if numpy is not installed, the test will not be run
         test_pass = False
     assert test_pass

# run the testing function `test_mean_absolute_deviation` on a new
    testcase
test_mean_absolute_deviation([48.35, 48.45, 48.85, 49.15, 49.35,
    49.55, 50.15, 50.55, 51.55, 54.45])
```

**Random relational spec for program 1**

```
def test_mean_absolute_deviation(numbers: List[float]) -> float:
     """ Given an input `numbers`, test whether the function `
    mean_absolute_deviation` is implemented correctly.
     """
     # TODO
     pass

# run the testing function `test_mean_absolute_deviation` on a new
    testcase

test_mean_absolute_deviation([1.0, 2.0, 3.0, 4.0])
```

**Example synthesized program 2 (sample is correct)**

```
from typing import List


def filter_by_substring(strings: List[str], substring: str) -> List[
    str]:
     """ Filter an input list of strings only for ones that contain
    given substring
     >>> filter_by_substring([], 'a')
     []
     >>> filter_by_substring(['abc', 'bacd', 'cde', 'array'], 'a')
     ['abc', 'bacd', 'array']
     """
     return [s for s in strings if substring in s]
```

**Top ranked relational spec for program 2**

```
def test_filter_by_substring(input_list: List[str], input_substring:
    str):
    """ Given an input `input_list` and `input_substring`, test
    whether the function `filter_by_substring` is implemented
    correctly.
    """
    output_list = filter_by_substring(input_list, input_substring)
    # check if all the elements in the output list do contain the
    given substring
    for string in output_list:
        assert input_substring in string

# run the testing function `test_filter_by_substring` on a new
    testcase
test_filter_by_substring(['ABCDE', 'BACDE', 'CDE', 'ARRAY'], 'BC')
```

**Random relational spec for program 2**

```
def test_filter_by_substring(strings: List[str], substring: str) ->
    List[str]:
    """ Given a list of `strings` and an input `substring`, test
    whether the function `filter_by_substring` is implemented
    correctly.
    """
    output_strings = filter_by_substring(strings, substring)
    # check if every element returned in the output list contains the
    given substring
    for element in output_strings:
        assert substring in element

# run the testing function `test_filter_by_substring` on a new
    testcase
test_filter_by_substring(['abc', 'bcd', 'cde', 'adb', 'aed'], 'ae')
```

**Example synthesized program 3 (sample is incorrect)**

```
from typing import List, Tuple


def rolling_max(numbers: List[int]) -> List[int]:
    """ From a given list of integers, generate a list of rolling
    maximum element found until given moment
    in the sequence.
    >>> rolling_max([1, 2, 3, 2, 3, 4, 2])
    [1, 2, 3, 3, 3, 4, 4]
    """
    result: List[int] = []
    current_max: int = numbers[0]
    for i in numbers:
        if i > current_max:
            current_max = i
        result.append(current_max)
    return result
```

**Top ranked relational spec for program 3**

```
def test_rolling_max(input_list: List[int]):
    """ Given an input list `input_list`, test whether the function `
    rolling_max` is implemented correctly.
    """
    output_list = rolling_max(input_list)
    # check if the length of the output list is correct
    assert len(output_list) == len(input_list)
    # check if the last element of the output list is the maximum of
    the input list
    assert output_list[-1] == max(input_list)

    # Check if the the the element at index i (starting from 0) in the
     output list
    # is the maximum of the elements at index 0 through index i (
    inclusive) of the input list.
    for i in range(len(output_list)):
        assert output_list[i] == max(input_list[0: i + 1])

# run the testing function `test_rolling_max` on a  new testcase
test_rolling_max([-1, 2, 7, 10, -5, 10, 9, 5, 4, 8, 9, 8, -10])
```

**Random relational spec for program 3**

```
import random

def test_rolling_max(numbers: List[int]):
    """ Given an input `numbers`, test whether the function `
    rolling_max` is implemented correctly.
    """
    rolling_max_list = rolling_max(numbers)
    # check if the length of the output list is correct
    assert len(rolling_max_list) == len(numbers)
    # check if the output list is correct
    for i in range(len(rolling_max_list)):
        assert rolling_max_list[i] == max(numbers[:i + 1])

# run the testing function `test_rolling_max` on a new testcase
test_rolling_max(numbers = [random.randint(0, 1000) for i in range
    (100)])
```

**Example synthesized program 4 (sample is correct)**

```
def count_upper(s):
    """
    Given a string s, count the number of uppercase vowels in even
    indices.

    For example:
    count_upper('aBCdEf') returns 1
    count_upper('abcdefg') returns 0
    count_upper('dBBE') returns 0
    """
    upper_count = 0
    upper_vowels = ['A', 'E', 'I', 'O', 'U']
    for i in range(0, len(s), 2):
        if s[i] in upper_vowels:
            upper_count += 1
    return upper_count
```

**Top ranked relational spec for program 4**

```
def test_count_upper():
    """
    Test whether the count_upper function is implemented correctly
    """
    assert count_upper('aBCdEf') == 1
    assert count_upper('abcdefg') == 0
    assert count_upper('dBBE') == 0
    assert count_upper('Aa') == 1
    # add your own test cases here!

# run the testing function `test_count_upper`
test_count_upper()
```

**Random relational spec for program 4**

```
def test_count_upper(s):
    """
    Given a string s, test whether the function `count_upper` is
   implemented correctly.
    """
    # TODO
    pass

# run the testing function `test_count_upper` on a new testcase
test_count_upper('ABCDEabcdeABCDE YOUUOY')
```

**Example synthesized program 5 (sample is incorrect)**

```
def get_max_triples(n):
    """
    You are given a positive integer n. You have to create an integer
   array a of length n.
        For each i (1 <= i <= n), the value of a[i] = i * i - i + 1.
        Return the number of triples (a[i], a[j], a[k]) of a where i <
    j < k,
    and a[i] + a[j] + a[k] is a multiple of 3.

    Example :
        Input: n = 5
        Output: 1
        Explanation:
        a = [1, 3, 7, 13, 21]
        The only valid triple is (1, 7, 13).
    """
    if n < 3:
        return 0
    cache = {}
    for i in range(3, n+1):
        residue = i*i - i + 1 % 3
        if residue == 0:
            cache[residue] = 1
        else:
            if residue in cache:
                cache[residue] += 1
            else:
                cache[residue] = 1
    res = 0
    for i in cache:
        if cache[i] >= 2:
            res += get_combinations(cache[i], 2)
    return res
```

**Top ranked relational spec for program 5**

```
def test_get_max_triples(n):
    """ Given an input `n`, test whether the function `get_max_triples
   ` is implemented correctly.
    """
    output = get_max_triples(n)
    # check if the output is of the correct type
    assert isinstance(output, int)
    a = [i * i - i + 1 for i in range(1, n+1)]
    # check if the output is of the correct value
    number_of_valid_triples = 0
    for i in range(1, n):
        for j in range(i+1, n):
            if a[i] + a[j] % 3 == 0:
                number_of_valid_triples += 1
    assert number_of_valid_triples == output

# run the testing function `test_get_max_triples` on a new testcase
test_get_max_triples(n=10)
```

**Random relational spec for program 5**

```
def test_get_max_triples(n: int):
    """ Given an input `input_str`, test whether the function `
    get_max_triples` is implemented correctly.
    """
    # TODO
    pass

test_get_max_triples(34) # should print 92
test_get_max_triples(24) # should print 37
test_get_max_triples(14) # should print 18
test_get_max_triples(23) # should print 39
test_get_max_triples(35) # should print 96
test_get_max_triples(43) # should print 124
test_get_max_triples(2) # should print 0
test_get_max_triples(32) # should print 78
test_get_max_triples(15) # should print 18
test_get_max_triples(21) # should print 28
test_get_max_triples(36) # should print 99
```

**Example synthesized program 6 (sample is incorrect)**

```
from typing import List


def separate_paren_groups(paren_string: str) -> List[str]:
    """ Input to this function is a string containing multiple groups
    of nested parentheses. Your goal is to
    separate those group into separate strings and return the list of
    those.
    Separate groups are balanced (each open brace is properly closed)
    and not nested within each other
    Ignore any spaces in the input string.
    >>> separate_paren_groups('( ) (( )) (( )( ))')
    ['()', '(())', '(()())']
    """
    return []
```

**Top ranked relational spec for program 6**

```
def test_separate_paren_groups(input_str: str) -> None:
    """ Given an input `input_str`, test whether the function `
   separate_paren_groups` is implemented correctly.
    """
    output_str_list = separate_paren_groups(input_str)
    # Check if all the entries in the output list are valid
    for input_str in output_str_list:
        assert validate_paren_group(input_str)

# Test 3
def test_separate_paren_groups(input_str: str) -> None:
    """ Given an input `input_str`, test whether the function `
   separate_paren_groups` is implemented correctly.
    """
    output_str_list = separate_paren_groups(input_str)
    # Check if all the entries in the output list are valid
    for input_str in output_str_list:
        assert validate_paren_group(input_str)

def validate_paren_group(input_str: str) -> bool:
    """ Check if an input string is a valid parenthesis group.
    """
    # check if the number of opening and closing parentheses are the
   same
    # check that the character at each position is a closed
   parentheses if the number of opening parentheses so far
    # is greater than the number of closing parentheses so far
    # check that the character at each position is a opening
   parentheses if the number of opening parentheses so far
    # is greater than the number of closing parentheses so far
    pass

# run the testing function `test_separate_paren_groups` on a  new
   testcase
test_separate_paren_groups('()((())(()))')
```

**Random relational spec for program 6**

```
def test_separate_paren_groups(paren_string: str) -> List[str]:
    """ Given an input `paren_string`, test whether the function `
   separate_paren_groups` is implemented correctly.
    """
    output_list = separate_paren_groups(paren_string)
    # first check if the output is a list
    assert isinstance(output_list, list)
    for individual_group in output_list:
        # check if each individual group has the same number of open
   and close brackets
        assert individual_group.count('(') == individual_group.count('
   )')
        # check if separate groups are not nested within each other
        assert individual_group.count('()') == 0



# run the testing function `test_separate_paren_groups` on a new
   testcase

test_separate_paren_groups('( ) (( )) (( )( ))')
```

**Example synthesized program 7 (sample is correct)**

```
def compare_one(a, b):
    """
    Create a function that takes integers, floats, or strings
    representing
    real numbers, and returns the larger variable in its given
    variable type.
    Return None if the values are equal.
    Note: If a real number is represented as a string, the floating
    point might be . or ,

    compare_one(1, 2.5) -> 2.5
    compare_one(1, "2,3") -> "2,3"
    compare_one("5,1", "6") -> "6"
    compare_one("1", 1) -> None
    """
    # create a list of all variables without types
    list_of_vars = [a, b]
    list_of_vars = [str(i) for i in list_of_vars]
    # remove commas
    for i, val in enumerate(list_of_vars):
        if ',' in val:
            list_of_vars[i] = val.replace(",", ".")
    # compare variables and return
    if float(list_of_vars[0]) < float(list_of_vars[1]):
        return b
    elif float(list_of_vars[0]) > float(list_of_vars[1]):
        return a
    else:
        return None
```

**Top ranked relational spec for program 7**

```
def test_compare_one():
    """
    Test whether the function `compare_one` is implemented correctly.
    """
    assert compare_one(1, 2.5) == 2.5
    assert compare_one(1, "2,3") == "2,3"
    assert compare_one("5,1", "6") == "6"
    assert compare_one("1", 1) == None

test_compare_one()
```

**Random relational spec for program 7**

```
def test_compare_one(a, b):
    """ Given an inputs `a` and `b`, test whether the function `
    compare_one` is implemented correctly.
    """
    output = compare_one(a, b)
    if a > b:
        assert output == a
    elif a < b:
        assert output == b
    elif a == b:
        assert output == None

# run the testing function `test_compare_one` on a new testcase
test_compare_one(1, 2.5)
```

**Example synthesized program 8 (sample is correct)**

```
def is_prime(n):
    """Return true if a given number is prime, and false otherwise.
    >>> is_prime(6)
    False
    >>> is_prime(101)
    True
    >>> is_prime(11)
    True
    >>> is_prime(13441)
    True
    >>> is_prime(61)
    True
    >>> is_prime(4)
    False
    >>> is_prime(1)
    False
    """

    if n == 2:
        return True

    if n < 2 or (n % 2) == 0:
        return False

    return not any(n % i == 0 for i in range(3, int(n ** 0.5) + 1, 2))
```

**Top ranked relational spec for program 8**

```
def test_is_prime(n):
    """ Given an input `n`, test whether the function `is_prime` is
    implemented correctly.
    """
    # check if the function return value is either `True` or `False`
    assert is_prime(n) in [True, False]
    # check if the function returns `True` if `n` is a prime number,
    otherwise returns `False`.
    if n < 2:
        assert not is_prime(n)
    else:
        for i in range(2, n):
            if n % i == 0:
                assert not is_prime(n)
            else:
                assert is_prime(n)

# run the testing function `test_is_prime` on a new testcase
test_is_prime(1579)
```

**Random relational spec for program 8**

```
def test_is_prime(n):
    """ Given an input `n`, test whether the function `is_prime` is
    implemented correctly.
    """
    # test if `n` is a prime number
    if is_prime(n):
        # check if `n` is divisible by any number other than 1 and `
    num`
        for i in range(2, n):
            assert n % i == 0
    else:
        # check if `n` is not a prime number
        assert n == 1 or n % 2 == 0
    pass

# run the testing function `test_is_prime` on a new testcase
test_is_prime(10)
```

**Example synthesized program 9 (sample is incorrect)**

```
def closest_integer(value):
    '''
    Create a function that takes a value (string) representing a
   number
    and returns the closest integer to it. If the number is
   equidistant
    from two integers, round it away from zero.

    Examples
    >>> closest_integer("10")
    10
    >>> closest_integer("15.3")
    15

    Note:
    Rounding away from zero means that if the given number is
   equidistant
    from two integers, the one you should return is the one that is
   the
    farthest from zero. For example closest_integer("14.5") should
    return 15 and closest_integer("-14.5") should return -15.
    '''
    return round(float(value))
```

**Top ranked relational spec for program 9**

```
def test_closest_integer(value):
    '''
    Given a value, test whether the function `closest_integer` is
    implemented correctly.
    '''
    output_val = closest_integer(value)
    assert type(output_val) is int and round(float(value)) ==
   output_val
# run the testing function `test_closest_integer` on a  new testcase
test_closest_integer("14.5")
test_closest_integer("-14.5")
test_closest_integer("14.4")
test_closest_integer("-14.4")
```

**Random relational spec for program 9**

```
def test_closest_integer(value):
    '''
    Test whether the function `closest_integer` is implemented
   correctly.
    '''
    # TODO
    pass

test_closest_integer("10.1")
```

**Example synthesized program 10 (sample is correct)**

```
def can_arrange(arr):
    """Create a function which returns the largest index of an element
     which
    is not greater than or equal to the element immediately preceding
   it. If
    no such element exists then return -1. The given array will not
   contain
    duplicate values.

    Examples:
    can_arrange([1,2,4,3,5]) = 3
    can_arrange([1,2,3]) = -1
    """
    for i in range(len(arr)-1, 0, -1):
        if arr[i-1] > arr[i]:
            return i
    return -1
```

**Top ranked relational spec for program 10**

```
def test_can_arrange(arr):
    """Given an input `arr`, test whether the function `can_arrange`
   is implemented correctly.
    """
    output = can_arrange(arr)
    # Check if the function returns -1 when there are no elements that
     are not greater than or
    # equal to the element immediately preceding it.
    if (all(arr[i] >= arr[i-1] for i in range(1, len(arr)))):
        assert output == -1
    # Check if the function returns the largest index of an element
   which is not greater
    # than or equal to the element immediately preceding it.
    else:
        assert output == max([i for i in range(1, len(arr)) if arr[i]
   < arr[i-1]])

# run the testing function `test_can_arrange` on a new testcase
test_can_arrange([1, 2, 4, 3, 5])
test_can_arrange([1, 2, 3])
test_can_arrange([5, 2, 3])
test_can_arrange([5, 2, 2, 2, 2])
test_can_arrange([9, 8, 7, 6, 5, 4, 3, 2, 1])
```

**Random relational spec for program 10**

```
def test_can_arrange(arr):
    """Given an array `arr`, test whether the function `can_arrange`
   is
    implemented correctly.
    """
    output = can_arrange(arr)
    if output == -1:
        assert len(arr) == 0 or all(arr[i + 1] <= arr[i] for i in
   range(len(arr) - 1))
    else:
        assert 0 <= output < len(arr)
        assert output == 0 or arr[output] < arr[output - 1]
        assert output == len(arr) - 1 or arr[output] >= arr[outpus +
   1]

# run the testing function `test_can_arrange` on a new testcase
test_can_arrange([7, 6, 5, 4, 3, 2, 1])
```

**Example synthesized program 11 (sample is correct)**

```
def specialFilter(nums):
    """Write a function that takes an array of numbers as input and
   returns
    the number of elements in the array that are greater than 10 and
   both
    first and last digits of a number are odd (1, 3, 5, 7, 9).
    For example:
    specialFilter([15, -73, 14, -15]) => 1
    specialFilter([33, -2, -3, 45, 21, 109]) => 2
    """
    return len([num for num in nums if (num > 10 and (num % 10 in {1,
   3, 5, 7, 9} and num // 10 ** (len(str(num)) - 1) in {1, 3, 5, 7,
   9}))])

    return len(list(filter(lambda x: bool(x > 10 and (x % 10 in {1, 3,
    5, 7, 9} and x // 10 ** (len(str(x)) - 1) in {1, 3, 5, 7, 9})),
   nums)))
```

**Top ranked input-output spec for program 11**

```
assert specialFilter([33, 23, 75, 21, -1, 109]) == 3
```

**Random input-output spec for program 11**

```
assert specialFilter([1, 2, 3, 4]) == 0
```

**Example synthesized program 12 (sample is correct)**

```python
def make_a_pile(n):
    """
    Given a positive integer n, you have to make a pile of n levels of
     stones.
    The first level has n stones.
    The number of stones in the next level is:
        - the next odd number if n is odd.
        - the next even number if n is even.
    Return the number of stones in each level in a list, where element
     at index
    i represents the number of stones in the level (i+1).

    Examples:
    >>> make_a_pile(3)
    [3, 5, 7]
    """
    stones = []
    for level in range(1, n + 1):
        stones.append(n + 2 * level - 2)
    return stones
```

**Top ranked input-output spec for program 12**

```python
assert make_a_pile(9) == [9, 11, 13, 15, 17, 19, 21, 23, 25]
```

**Random input-output spec for program 12**

```python
assert make_a_pile(4) == [4, 6, 8, 10], "Wrong implementation"
```

**Example synthesized program 13 (sample is correct)**

```
def do_algebra(operator, operand):
    """
    Given two lists operator, and operand. The first list has basic
   algebra operations, and
    the second list is a list of integers. Use the two given lists to
   build the algebric
    expression and return the evaluation of this expression.

    The basic algebra operations:
    Addition ( + )
    Subtraction ( - )
    Multiplication ( * )
    Floor division ( // )
    Exponentiation ( ** )

    Example:
    operator['+', '*', '-']
    array = [2, 3, 4, 5]
    result = 2 + 3 * 4 - 5
    => result = 9

    Note:
        The length of operator list is equal to the length of operand
   list minus one.
        Operand is a list of of non-negative integers.
        Operator list has at least one operator, and operand list has
   at least two operands.

    """
    if len(operator) > 0 and len(operand) > 1:
        expression = [str(operand[0])]
        for i in range(len(operator)):
            expression.append(operator[i])
            expression.append(str(operand[i+1]))
        expression = ' '.join(expression)
        return eval(expression)
    else:
        return None
```

**Top ranked input-output spec for program 13**

```
assert do_algebra(['+', '*'], [2, 3, 4, 5]) == 14
```

**Random input-output spec for program 13**

```
assert do_algebra(['+', '*', '-'], [2, 3, 4, 5]) == 9
```

**Example synthesized program 14 (sample is correct)**

```
def modp(n: int, p: int):
    """Return 2^n modulo p (be aware of numerics).
    >>> modp(3, 5)
    3
    >>> modp(1101, 101)
    2
    >>> modp(0, 101)
    1
    >>> modp(3, 11)
    8
    >>> modp(100, 101)
    1
    """
    # Explanation of the strategy:
    # if n == 0: return 1 because 2^0 = 1
    # if n == 1: return 2 because 2^1 = 2 % p = 2
    # if n == 2: return 4 because 2^2 = 4 % p = 4
    # if n == 3: return 3 because 2^3 = 8 % p = 3 (p is prime, so 2^p
   = 2)
    # if n == 4: return 1 because 2^4 = 16 % p = 1
    # if n == 5: return 2 because 2^5 = 32 % p = 2
    # if n == 6: return 4 because 2^6 = 64 % p = 4
    # if n == 7: return 3 because 2^7 = 128 % p = 3 (p is prime, so 2^
   p = 2)
    # if n == 8: return 1 because 2^8 = 256 % p = 1
    # if n == 9: return 2 because 2^9 = 512 % p = 2
    # ...
    # if n == 2k: return 2^{2k} = 2^{k}^2 % p = (2^k % p)^2 % p
    # if n == 2k + 1: return 2^{2k+1} = 2 * 2^{2k} = 2 * (2^k % p)^2 %
     p = 2 * (2^k % p)^2 % p
    # if n == 2k + 1: 2^n % p = 2 * 2^{2k} % p = 2 * (2^k % p)^2 % p
    # if n == 2k: 2^n % p = (2^k % p)^2 % p
    if n == 0:
        return 1
    else:
        if n % 2 == 0:
            half = modp(n / 2, p)
            return (half ** 2) % p
        else:
            return (2 * modp(n - 1, p)) % p
```

**Top ranked input-output spec for program 14**

```
assert modp(3**3**3, 20) == 8
```

**Random input-output spec for program 14**

```
assert modp(0, 101) == 1
```

**Example synthesized program 15 (sample is correct)**

```
def triangle_area(a, h):
    """Given length of a side and high return area for a triangle.
    >>> triangle_area(5, 3)
    7.5
    """
    return (a * h) / 2
```

**Top ranked input-output spec for program 15**

```
assert triangle_area(1.5, 1.5) == 1.125
```

**Random input-output spec for program 15**

```
assert triangle_area(5, 3) == 7.5, 'Test failed'
```

**Example synthesized program 16 (sample is correct)**

```
def fibfib(n: int):
    """The FibFib number sequence is a sequence similar to the
    Fibbonacci sequnece that's defined as follows:
    fibfib(0) == 0
    fibfib(1) == 0
    fibfib(2) == 1
    fibfib(n) == fibfib(n-1) + fibfib(n-2) + fibfib(n-3).
    Please write a function to efficiently compute the n-th element of
     the fibfib number sequence.
    >>> fibfib(1)
    0
    >>> fibfib(5)
    4
    >>> fibfib(8)
    24
    """
    if n < 2:
        return 0
    elif n == 2:
        return 1
    else:
        f0 = 0
        f1 = 0
        f2 = 1
        for i in range(3, n+1):
            f0, f1, f2 = f1, f2, f0+f1+f2
        return f2
```

**Top ranked input-output spec for program 16**

```
assert fibfib(8) == 24, "fibfib 8" ''' 6. Write a function to return
    the most common character in a string. Inputs: str: string to be
    evaluated Outputs: character that occurs most frequently in the
    string Notes: This problem can be solved in O(n) time using a hash
     map. Examples: - "hello world" -> 'l' - "aabbcc" -> 'a' - "
    mississippi" -> 'i' '''
```

**Random input-output spec for program 16**

```
assert fibfib(5) == 4, "Did you implement fibfib(n) correctly?"
```

**Example synthesized program 17 (sample is correct)**

```
def fib(n: int):
    """Return n-th Fibonacci number.
    >>> fib(10)
    55
    >>> fib(1)
    1
    >>> fib(8)
    21
    """
    "*** YOUR CODE HERE ***"
    a, b = 0, 1
    while n > 0:
        a, b = b, a + b
        n -= 1
    return a
```

**Top ranked input-output spec for program 17**

```
assert fib(1000) ==
   434665576869374564356885276750406258025646605173717804024817290895365554179490518904
```

**Random input-output spec for program 17**

```
assert fib(10) == 55, fib(10)
```

**Example synthesized program 18 (sample is incorrect)**

```
def prime_length(string):
    """Write a function that takes a string and returns True if the
   string
    length is a prime number or False otherwise
    Examples
    prime_length('Hello') == True
    prime_length('abcdcba') == True
    prime_length('kittens') == True
    prime_length('orange') == False
    """
    length_str = len(string)
    for i in range(2, length_str - 1):
        if length_str % i == 0:
            return False
    return True
```

**Top ranked input-output spec for program 18**

```
assert prime_length('v') == True, "Failed prime_length test 5"
```

**Random input-output spec for program 18**

```
assert prime_length('orange') == False, 'Check function definition!'
```

**Example synthesized program 19 (sample is correct)**

```
from typing import List


def remove_duplicates(numbers: List[int]) -> List[int]:
    """ From a list of integers, remove all elements that occur more
    than once.
    Keep order of elements left the same as in the input.
    >>> remove_duplicates([1, 2, 3, 2, 4])
    [1, 3, 4]
    """
    num_count = {}
    for num in numbers:
        num_count[num] = num_count.get(num, 0) + 1

    new_list = []
    for num in numbers:
        if num_count[num] == 1:
            new_list.append(num)

    return new_list
```

**Top ranked input-output spec for program 19**

```
assert remove_duplicates([1, 2, 1, 2, 1, 2, 3]) == [3]
```

**Random input-output spec for program 19**

```
assert remove_duplicates([1, 2, 1, 2]) == []
```

**Example synthesized program 20 (sample is incorrect)**

```
def is_bored(S):
    """
    You'll be given a string of words, and your task is to count the
   number
    of boredoms. A boredom is a sentence that starts with the word "I
   ".
    Sentences are delimited by '.', '?' or '!'.

    For example:
    >>> is_bored("Hello world")
    0
    >>> is_bored("The sky is blue. The sun is shining. I love this
   weather")
    1
    """
    boredoms = 0
    # replace . or ! or ? with . to simplify this problem
    S = S.replace('.','. ')
    S = S.replace('!','! ')
    S = S.replace('?','? ')
    sentences = S.split(' ')
    for sentence in sentences:
        if sentence.startswith('I'):
            boredoms = boredoms + 1
    return boredoms
```

**Top ranked input-output spec for program 20**

```
assert is_bored("I have no idea what I'm doing") == 2
```

**Random input-output spec for program 20**

```
assert is_bored("I love this weather.") == 1
```