A Scalable and Extensible Approach to Benchmarking NL2Code for 18 Programming Languages

Federico Cassano[1], **John Gouwar**[1], **Daniel Nguyen**[2], **Sydney Nguyen**[3], **Luna Phipps-Costin**[1], **Donald Pinckney**[1], **Ming-Ho Yee**[1], **Yangtian Zi**[1], **Carolyn Jane Anderson**[3], **Molly Q Feldman**[4], **Arjun Guha**[1,5], **Michael Greenberg**[6], **Abhinav Jangda**[7]

[1] Northeastern University, [2] Hanover High School, [3] Wellesley College, [4] Oberlin College, [5] Roblox Research, [6] Stevens Institute of Technology, [7] University of Massachusetts Amherst

Abstract

Large language models have demonstrated the ability to condition on and generate both natural language and programming language text. Such models open up the possibility of multi-language code generation: could code generation models generalize knowledge from one language to another? Although contemporary code generation models can generate semantically correct Python code, little is known about their abilities with other languages. We facilitate the exploration of this topic by proposing MultiPL-E, the first multi-language parallel benchmark for natural-language-to-code-generation.

MultiPL-E extends the HumanEval benchmark (Chen et al. 2021) to support 18 more programming languages, encompassing a range of programming paradigms and popularity. We evaluate two state-of-the-art code generation models on MultiPL-E: Codex (Chen et al. 2021) and InCoder (Fried et al. 2022). We find that on several languages, Codex matches and even exceeds its performance on Python. The range of programming languages represented in MultiPL-E allow us to explore the impact of language frequency and language features on model performance. Finally, the MultiPL-E approach of compiling code generation benchmarks to new programming languages is both scalable and extensible. We describe a general approach for easily adding support for new benchmarks and languages to MultiPL-E.

1 Introduction

Advances in scaling large language models (LLMs) have improved not only their ability to generate natural language text, but also to generate code in programming languages. LLMs exposed to code can generate test cases, documentation, and even full programs from natural language descriptions (Black et al. 2022; Chen et al. 2021; Fried et al. 2022; Nijkamp et al. 2022; Xu et al. 2022). However, relatively little is known about how well code generation models generalize across programming languages. Most evaluation has been done on Python, since it is the best-represented language in most training datasets (Yin et al. 2018; Chen et al. 2021; Austin et al. 2021; Hendrycks et al. 2021).

In this paper, we present MultiPL-E, the first parallel multi-language[1] benchmark for evaluating the code generation performance of LLMs for 18 programming languages. We focus on the natural-language-to-code (NL2Code) task: given a natural language description, generate the corresponding function in code. A key feature of this task is that its evaluation is straightforward: we can run *unit tests* to determine if the generated function behaves correctly. There are a handful of NL2Code benchmarks that use unit tests in this manner (Kulal et al. 2019; Hendrycks et al. 2021; Austin et al. 2021). But, these benchmarks only evaluate performance on a single language.

MultiPL-E uses a suite of compilers to translate the Python benchmarks from Chen et al. (2021) into parallel benchmarks in 18 languages. For the first time, MultiPL-E provides a way to evaluate code generation models on a consistent set of benchmark problems across many languages. The 18 languages capture a broad spectrum of language features, application areas, and popularity, allowing us to explore the impact of these factors on model performance.

The MultiPL-E benchmark and associated tools are open source and easy to extend. For the NL2Code task, each compiler translates Python unit tests, doctests, and function signatures to its target language. Because these program regions do not contain arbitrary Python code, each MultiPL-E compiler is much simpler than a full-fledged compiler. Our framework makes it easy to add new benchmark programs and to extend the benchmark to additional languages.

We evaluate the performance of Codex (Chen et al. 2021) and InCoder (Fried et al. 2022) on MultiPL-E. Our analysis presents several insights into the effectiveness of LLMs for code generation, including that (1) Codex performs best on JavaScript and equally well on C++, Scala, and TypeScript as on Python; (2) model perplexity is not strongly correlated with the correctness of generated code; (3) type annotations have limited impact on model performance for gradually typed languages; (4) model performance is correlated with language popularity, but some niche languages perform as well as more popular languages; and (5) performance is sensitive to prompt design for both niche and popular languages.

Contributions Our key contributions are:

- MultiPL-E: a parallel benchmark for NL2Code in 18 languages encompassing a variety of programming paradigms, language features, and popularity levels;
- An easily extensible system for translating NL2Code

[1] In this paper, "multi-language" refers to multiple *programming languages.*

problems, including doctests, unit tests, and Python-oriented terminology, into new programming languages;

- A multi-language parallel evaluation of two models, Codex (Chen et al. 2021) and InCoder (Fried et al. 2022);
- Explorations of language frequency effects, the impact of type annotations, and prompt translation sensitivity on NL2Code performance, along with a fine-grained analysis of NL2Code errors in four languages.

Our code and data is available at github.com/nuprl/MultiPL-E

2 Evaluating Code Generation

Code generation has long been a task of interest: there is extensive work on program synthesis (Alur et al. 2013; Chaudhuri et al. 2021) using both symbolic and neuro-symbolic approaches. More recently, LLMs trained for text generation have demonstrated the ability to perform program completion (Brown et al. 2020; Wang and Komatsuzaki 2021; Black et al. 2022); and several LLMs have been trained or fine-tuned on programming language text (Feng et al. 2020; Clement et al. 2020; Chen et al. 2021; Nijkamp et al. 2022; Fried et al. 2022; Xu et al. 2022).

Unlike traditional program synthesis techniques, neural language models are able to condition on and generate both natural language and programming language text. Moreover, LLMs offer the promise of synthesizing knowledge gleaned from code in multiple programming languages. *Just as language models exposed to multiple natural languages are able to generalize across languages, might not multi-language models of code do the same?*

Although this kind of multi-language generalization is an intriguing possibility, little is known about how well code generation models perform across programming languages. We make progress towards answering this question by evaluating the multi-language code generation abilities of two state-of-the-art models: Codex and InCoder.

2.1 Large Language Models for Code

Both models use a Tranformer architecture (Vaswani et al. 2017) and are trained with a language modeling objective on a mixture of natural language and code.

InCoder InCoder (Fried et al. 2022) is a 6.7B parameter language model trained using a causal masking objective (Aghajanyan et al. 2022). It supports both code infilling and code completion; we test only the latter. InCoder was trained on 159 GB of deduped, filtered code from Github (around a third in Python) and 57GB from StackOverflow.

Codex Codex is a GPT-3 language model fine-tuned on code. Chen et al. (2021) describe a 12B parameter version of Codex fine-tuned on 159 GB of deduped, filtered Python code from Github. We use the more recent `codex-davinci-002` model, which is trained on multiple languages, but details of its training set and size are not public (Zaremba, Brockman, and OpenAI 2021).

(a) The prompt given to the NL2Code model:

```
def lsi(lst):
  """Create a function that returns a
  tuple (a, b), where 'a' is the largest
  of negative integers, and 'b' is the
  smallest of positive integers in a
  list. If there is no negative or
  positive integers, return them as
  None.
  Examples:
    lsi([2, 4, 1, 3, 5, 7]) == (None, 1)
    lsi([]) == (None, None)
    lsi([0]) == (None, None)
  """
```

(b) A completion generated by Codex:

```
  if len(lst) == 0:
    return (None, None)
  else:
    neg_nums = [x for x in lst if x<0]
    pos_nums = [x for x in lst if x>0]
  if len(neg_nums) == 0:
    neg_nums = None
  if len(pos_nums) == 0:
    pos_nums = None
  return (max(neg_nums), min(pos_nums))
```

(c) 3 of the 12 unit tests (the generated solution fails two):

```
X assert lsi([2, 4, 1, 3, 5, 7]) == (None, 1)
X assert lsi([2, 4, 1, 3, 7, 0]) == (None, 1)
✓ assert lsi([1, 3, 4, 5, 6, -2]) == (-2, 1)
```

Figure 1: Problem 136 of 164 of the HumanEval benchmark. We shorten the name `largest_smallest_integers` for brevity. Top: the prompt for the model, with the function signature, natural language description, and doctests. Middle: a Codex-generated solution. Bottom: unit tests.

2.2 The Natural Language to Code Task

Code-aware language models have been applied to a variety of tasks, including test generation (Tufano et al. 2020), docstring generation (Lu et al. 2021), code search (Feng et al. 2020; Ahmed and Devanbu 2022), type inference (Wei et al. 2020; Hellendoorn et al. 2018; Pradel et al. 2020), and more (Drori et al. 2022). We focus on the **natural-language-to-code** task (NL2Code): given the description of a function in natural language, complete the function body.

Figure 1a shows an example prompt from the HumanEval benchmark dataset for NL2Code (Chen et al. 2021). Each prompt contains several sources of information for the model: the function signature (its name and parameters); a brief natural language description of the intended function; and, optionally, examples in the form of Python doctests. The model's task is to generate the body of the function given this input; the unit tests are then used to test whether the generated function is correct. The model does not see the unit tests.

Note that the model does not receive an explicit cue about the target language, but each of the three prompt regions provide implicit cues: the syntax of the function signature, the terminology used in the natural language description, and the syntax of the doctests all suggest that the target is Python. Consequently, to translate this prompt to a new programming language, we should target all three regions of the prompt.

2.3 Evaluating NL2Code

Existing benchmarks for evaluating natural-language-to-code generation models have two main limitations: (1) they evaluate only a single language (Kulal et al. 2019; Chen et al. 2021; Hendrycks et al. 2021; Austin et al. 2021), or (2) they do not test the correctness of generated code (Yin et al. 2018; Xu et al. 2022).[2] For instance, although Xu et al. (2022) measure model perplexity on several programming languages, they only test code correctness for Python.

Unlike natural languages, programming languages have a well-defined notion of *semantic equivalence*: two functions are semantically equivalent if and only if they behave identically on all inputs. Equivalence-checking is undecidable; however, we can use *unit tests* to approximate equivalence between programs by verifying a sample of program inputs.

In the context of NL2Code, we judge a generated function correct if it passes a suite of unit tests included with the benchmark. Figure 1b shows just one solution generated by Codex for the example prompt. This solution is incorrect because it fails some of the unit tests (Figure 1c).

We posit that a multi-language NL2Code benchmark should have *a parallel suite of benchmark problems in multiple programming languages* that can be solved and tested consistently across languages. MultiPL-E *compiles* NL2Code benchmarks for Python into parallel benchmarks for 18 other programming languages. Writing 18 full-language compilers would require extraordinary effort. However, a MultiPL-E compiler only needs to translate function signatures, comments, and unit tests.

There are a number of single-language NL2Code benchmarks (Yin et al. 2018; Kulal et al. 2019; Hendrycks et al. 2021). We select HumanEval (Austin et al. 2021) as a source benchmark for several reasons: (1) it is a diverse collection of 164 problems; (2) all problems have tests to check correctness; (3) all problems are functions that receive and return first-order values, which lend themselves to rigorous testing; (4) it is a challenging benchmark: the best model evaluated by Fried et al. (2022) achieves only a 36% pass rate on Python.

3 The MultiPL-E Benchmarking Approach

MultiPL-E is a multi-language, parallel NL2Code benchmark that includes a diverse set of programming languages. Table 1 gives an overview of the 18 MultiPL-E languages. We categorize the languages into four frequency classes (NICHE, LOW, MEDIUM, or HIGH) based on a weighting of TIOBE rank and GitHub frequency. There are eight languages in MultiPL-E that have never been used before to measure NL2Code performance, including newer languages (Julia and Swift), older scripting languages (Bash and Perl), and languages used for specific applications (Lua and R). The broad range of languages in MultiPL-E shows the generality of our compilation approach and allows us to explore how language frequency affects performance (§5.1).

PL	Typed?	GitHub %	TIOBE	Category
Bash	×	-	43	NICHE
C++	✓	7.0	4	HIGH
C#	✓	3.1	5	MEDIUM
D	✓	-	35	NICHE
Go	✓	7.9	12	MEDIUM
Java	✓	13.1	3	HIGH
JavaScript	×	14.3	7	HIGH
Julia	×	0.1	28	NICHE
Lua	×	0.2	25	NICHE
Perl	×	0.3	17	LOW
PHP	×	5.3	11	MEDIUM
R	×	0.05	19	LOW
Racket	×	-	-	NICHE
Ruby	×	6.2	15	MEDIUM
Rust	✓	1.1	22	LOW
Scala	✓	1.7	32	LOW
Swift	✓	0.7	10	LOW
TypeScript	✓	9.1	33	HIGH

Table 1: MultiPL-E languages by frequency, as calculated by GitHut 2.0 and the TIOBE Programming Community index.

A key feature of MultiPL-E is that it is easy to extend with new models, benchmarks, and languages. To support new languages and benchmarks without manual (and error-prone) effort, we build 18 *compilers* to translate NL2Code benchmarks written in Python. Writing one of these compilers is straightforward when the target language is similar to Python, but requires care for typed languages and even some untyped languages, notably Perl and R. The remainder of this section presents the design of these compilers.

3.1 Compiling Python Benchmarks

A MultiPL-E compiler is significantly easier to build than a complete compiler: to translate a benchmark problem, we only need to compile function signatures and unit tests (not arbitrary statements and expressions). Our compilers preserve comments, since they contain the natural language description for the NL2Code task; however, we automatically rephrase them to replace Python-specific terminology.

Compiling Unit Tests MultiPL-E supports any unit test where the input and output to the test is a *first-order value*. In Python, these include constants and data structures such as lists, tuples, and dictionaries, but exclude values such as `lambda` expressions.[3] HumanEval unit tests apply the

[2] Textual similarity metrics (e.g., BLEU) have been shown to correlate weakly with code correctness. (Ren et al. 2020; Austin et al. 2021; Chen et al. 2021).

[3] We do not support testing higher-order functions, but support generated code that uses higher-order functions.

(a) Original Python assertion.

```
assert lsi([0]) == (None, None)
```

(b) Equivalent R.

```
if(!identical(lsi(c(0)), c(NULL, NULL))){
    quit('no', 1)}
```

(c) Equivalent JavaScript.

```
assert.deepEqual(lsi([0]), [void 0, void 0]);
```

Figure 2: Example of a translated assertion.

(a) Original Python docstring from HumanEval #95.

```
Given a dictionary, return True if all
keys are strings in lower case or all keys
are strings in upper case, else return
False. The function should return False
is the given dictionary is empty.
```

(b) Terminology translated to Perl.

```
Given a hash, return 1 if all keys are
strings in lower case or all keys are
strings in upper case, else return "".
The function should return "" is the given
hash is empty.
```

Figure 3: A Python docstring and its Perl translation.

model-generated function to a first-order value, and compare the result with an expected first-order value.

Each MultiPL-E compiler has a recursive function that compiles Python values to the target language's values, with the recursive step handling nested values. Even for an untyped target, this value-to-value compilation requires care, because not all Python value types have perfect analogues in every target. For example, we compile both tuples and lists to JavaScript arrays, since JavaScript lacks a canonical tuple type. We also support untyped targets where the compilation strategy is less obvious. For example, when the target is R, it may appear natural to compile Python lists to R lists: both kinds of lists can be nested and allow heterogenous values. However, R's vector type is much more commonly used (data frames are made of vectors). Unfortunately, vectors must be homogeneous and cannot be nested, so not all Python lists can be translated to vectors. For example, an argument typed `List[Int]` can be translated to a vector, but a nested list cannot. In order to more closely match the token distribution of idiomatic R code seen during training, our R compiler uses type-oriented compilation techniques even though R is untyped.

The final step of compiling tests is to choose an appropriate test for equality. The meaning of equality operators varies across programming languages. Python's == operator checks *deep equality*, i.e., item-by-item equality within data structures. Deep equality is the appropriate choice for unit tests. In some languages, we need to import equality-testing functions from testing libraries, as in the JavaScript example shown in Figure 2.

Compiling Function Signatures Compiling a function signature to an untyped language is straightforward, but requires care when the target is typed. Most typed languages require argument and return type annotations. Fortunately, a large subset of the HumanEval benchmarks employ Python's optional type annotations; we add type annotations to the others.[4] The MultiPL-E compilers for typed languages translate these types to target language types.

Many typed languages require type annotations in data structures, which appear in unit tests. For example, C++ vectors require an annotation specifying their element type, and numbers in Rust (sometimes) require a type suffix. We perform limited local type inference to calculate these types from the type of the function signature to ensure that the unit tests always compile successfully. The HumanEval benchmarks have five problems that employ Python types that cannot be expressed in all conventional typed languages, such as `Any`, which is the type of all values. We fail to compile these 5 problems to most typed languages.

A further problem arises when the target language has discriminated unions, which require us to inject values into constructors in the target language. For example, we compile a Python number `n` to `Some(n)` in Rust, only if the context is typed `Optional[Int]` (similarly in Swift and Scala).

Translating Doctests Python *doctests* are a standard format for examples in documentation. While many of the HumanEval prompts include examples, not all of them are validly formatted doctests. We standardize examples to the Python doctest format (">>>" prepended). We apply value-to-value compilation to the doctests as we do for unit tests. However, since not all languages have an equivalent doctest format, we keep the Python format for all target languages.

Translating Python Terminology in Prompts Different programming languages use different terminology to refer to the same concept. For example, a Python *list* is closest to a JavaScript *array* or a Rust *vector*. To mitigate the impact of these differences, we identify Python-specific terminology in the natural language portion of the prompt, and translate it to the most natural equivalent for the target language. Figure 3 shows an example of a prompt translated from Python to Perl. Notably, Perl not only lacks Booleans, but uses 1 for true and the empty string for false.

3.2 Limitations of Our Approach

A handful of HumanEval benchmarks cannot be easily translated using the MultiPL-E approach. Of the 164 original benchmarks: (1) we exclude 3 that have Python helper functions in their prompt; (2) we modify 2 benchmarks to use unit tests instead of randomized testing; and (3) for certain typed languages, we fail to compile up to 5 benchmarks with untranslatable types. (§4.1) shows that these changes do not lead to significantly different results for Python.

[4] §5.2 shows that this does not affect Python pass rates.

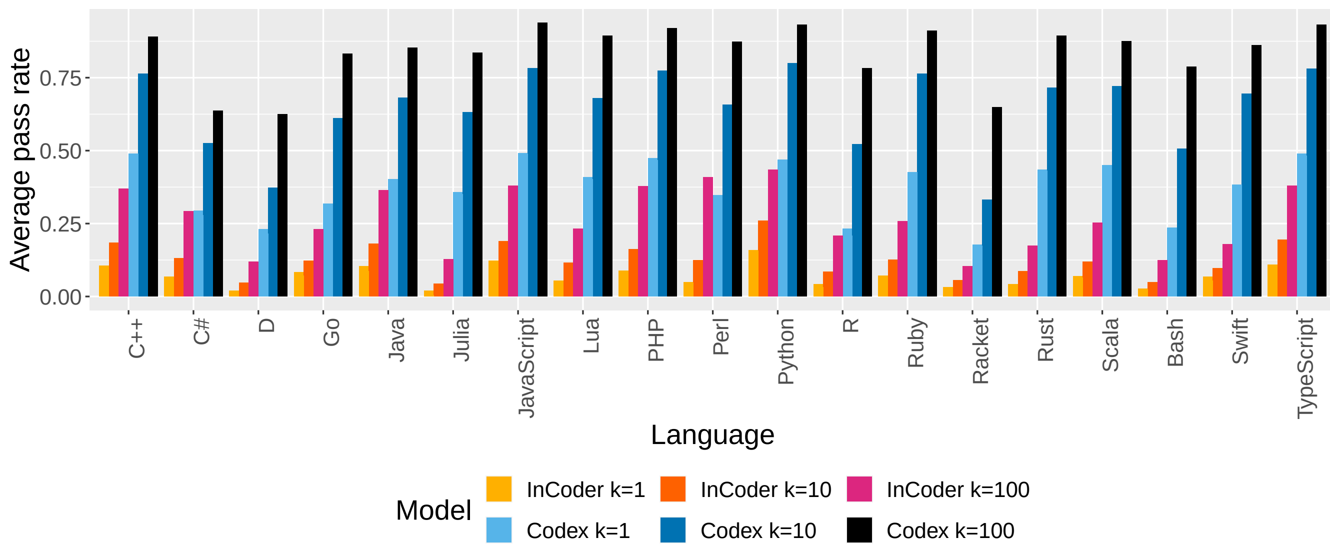

Figure 4: From right to left: InCoder pass@1, pass@10, pass@100; Codex pass@1, pass@10, pass@100 on all languages.

Our approach can be generalized to additional programming languages, so long as the target language has natural analogues for the Python data types used in the benchmarks. We do not include two previously studied languages, C (Xu et al. 2022) and SQL (Yu et al. 2018) because they do not meet this criterion.

4 Evaluation

We use MultiPL-E to evaluate two NL2Code models: InCoder (6.7B) and Codex (`code-davinci-002`). For each language, we calculate pass@k using the methodology employed by Chen et al. (2021) and subsequent work. Intuitively, pass@1 is the likelihood of the model producing a completion that passes *all unit tests*, pass@10 is the likelihood of any one of 10 completions passing all unit tests, and so on. We calculate pass@1 with temperature 0.2, and use temperature 0.8 for pass@10 and pass@100.

We fit mixed-effects models to evaluate the statistical significance of the differences between groups that we report below (Bates et al. 2015). Appendix C has a full description of each model with its estimate table.

4.1 Multi-Language Performance of NL2Code

Figure 4 shows the mean pass@k rate for each model on each MultiPL-E language. We find reliable differences between Codex pass@1 rates for Python and all but 4 languages: C++, JavaScript, Scala, and TypeScript. InCoder performs significantly better on Python than all other languages (all $p < 0.001$).

Python Results and Replication Our InCoder Python results replicate the previously reported performance of InCoder on HumanEval (Fried et al. 2022). This shows that the few standardization changes we made to the benchmarks do not affect model performance.

We evaluate a more recent Codex model (`code-davinci-002`) than previous work and observe a large improvement on Python: a pass@1 rate of 45.9%, compared to 28.8% reported earlier (Chen et al. 2021). This is notably better than the pass@1 rates reported for the two best models evaluated in Fried et al. (2022): CodeGen (Nijkamp et al. 2022) (29.3%) and PaLM-Code (Chowdhery et al. 2022) (36%), which we lack the resources to run.

Codex Performs Best on JavaScript Codex's performance on JavaScript is better than its performance on Python, though the difference is not significant (+2.3%; $p = 0.43$). Codex achieves a pass@1 rate higher than 40% on C++, Java, TypeScript, PHP, Ruby, Rust, Scala, and Lua.

InCoder performs more weakly than Codex. Like Codex, it performs better on more frequently-used languages (Python) than less popular ones (§5.1).

The Codex training set is not public; it is possible that the latest model has been trained on solutions to the HumanEval benchmarks in Python, and this could be inflating its performance. However, MultiPL-E is a new dataset for 18 other languages. That Codex matches or exceeds its Python performance on these new languages suggests a negligible impact of any train/test overlap.

Perplexity and Code Correctness Do Not Correlate Xu et al. (2022) report Codex perplexity scores for 10 of our 18 languages. Figure 5 plots Codex pass@1 scores against these scores. We do not observe a strong correlation between the measures. Notably, perplexity is highest for JavaScript and TypeScript, while we find that Codex performs best on these languages. This suggests that perplexity may not be a reliable evaluation metric for NL2Code. One caveat is that Xu et al. (2022) likely evaluate an older Codex model, since they report substantially lower pass rates for Python.

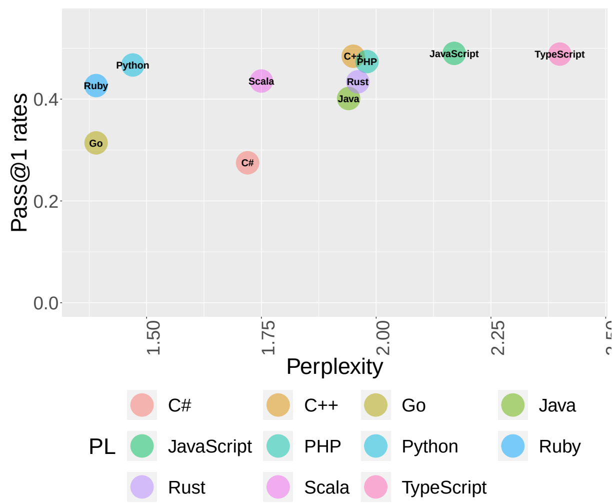

Figure 5: Codex pass@1 rates versus perplexity scores reported in Xu et al. (2022).

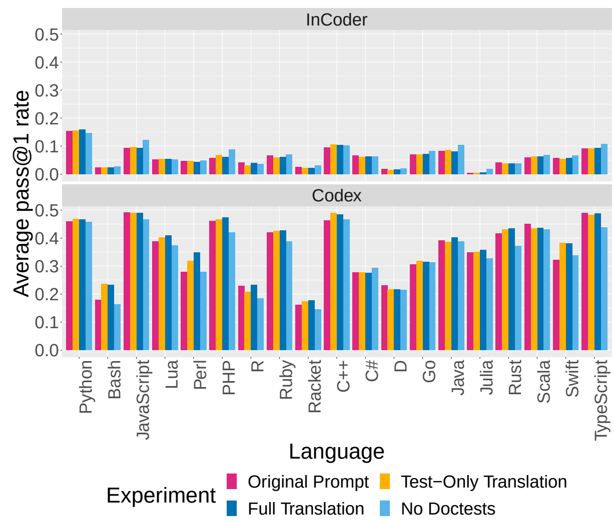

Figure 6: Ablation study of translation components. From right to left, pass@1 with original prompts; translated doctests; translated text and doctests; and doctests removed.

4.2 Ablation Study

Our compilers target multiple distinct regions of the prompt for each problem. We explore the impact of each component in an ablation study with four experiments:

- **Original Prompt**: does not translate doctests or natural language terminology (e.g. prompts as in HumanEval);
- **Test-only Translation**: translates doctests but not Python-specific terminology;
- **Full Translation**: translates unit tests, doctests, and Python-terminology in the prompt; and
- **No Doctests**: removes doctests and does not translate natural language terminology.

Figure 6 shows pass@1 results for each variation. For Codex, translating doctests and Python-specific terminology has little impact on better-performing languages. However, translating these components seems more important for challenging languages like Bash and Perl. Overall, we find significant differences between the **Full Translation** and **Test-Only Translation** experiments ($p = 0.03$), and between **No Doctests** and **Test-Only Translation** ($p < 0.001$). This suggests that doctests are useful to Codex, but that their format is not important.

Although removing doctests weakens Codex performance, puzzlingly, it appears to help InCoder performance ($p = 0.005$). Overall, we observe little benefit to translating doctests or Python terminology for InCoder.

5 Factors in Code Generation Success

This section explores our results in more depth. We examine how success rate is affected by language popularity, language features, and choices in prompt translation. We also study the kinds of errors that arise in NL2Code across several languages.

5.1 Programming Language Frequency

Figure 7 groups languages by frequency and plots the pass@1 rates for both models. Both models perform best on high frequency languages. However, Codex performs as well on certain LOW and NICHE languages as on MEDIUM languages: Lua is the 9th-best language in our dataset, although it only appears in 0.2% of GitHub activity and is not in the TIOBE Top-20. We find reliable differences in Codex pass@1 rates between LOW and NICHE languages when compared to the HIGH category ($p < 0.001$; $p = 0.002$), but not between the MEDIUM and HIGH categories ($p = 0.22$).

5.2 Type Annotations

One may conjecture that type annotations improve model performance by constraining the code generation search space. Or, perhaps, they might hurt performance by complicating the task. In Figure 7, the dashed line in each category separates languages with type annotations (left) from languages without (right). We observe no overall effect of type annotations on Codex pass@1 rates ($p = 0.33$).

To explore the impact of type annotations at a more fine-grained level, we run a series of follow-up experiments on Python, which allows optional type annotations, and TypeScript, a gradually typed cousin of JavaScript. Gradual typing allows us to weaken type annotations and the TypeScript compiler can even be configured to ignore all type errors.

Precise type annotations improve TypeScript performance TypeScript has an "Any" type, which is compatible with all types. We run Codex on a variation of the TypeScript prompts where all types in the function signature are translated to "Any". We find that the loss of precise types hurts performance on TypeScript (-2.5%; $p < 0.001$).

Type annotations *do not* improve Python performance We run a similar experiment with Codex and Python, where

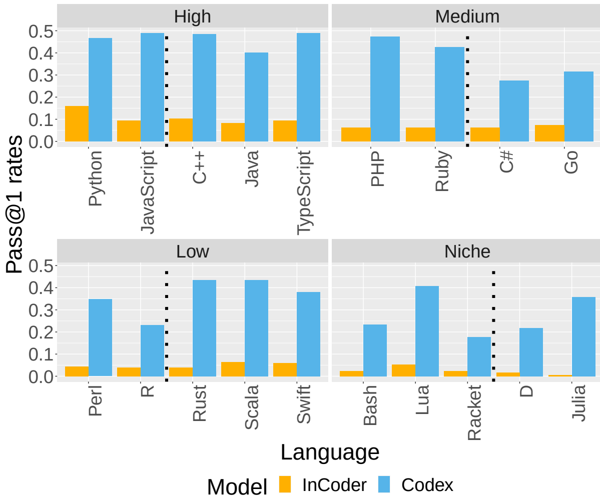

Figure 7: Model performance by language frequency and type-checking. Languages to the left of dashed line are untyped; languages to the right are typed.

we remove all the type annotations from the prompts. We find that this has no significant effect on Codex's pass@1 rate for Python ($p = 0.23$).

We interpret these results as evidence that type annotations do not guide search in general, since they do not improve Python performance, but that informative types are necessary for languages where type annotations are standard, perhaps in order to fit the token distribution of high-quality typed code seen in training.

TypeScript type errors correlate with runtime errors Type-checking can reject programs that would in fact run without error. We run the Codex-generated TypeScript programs without first checking for type errors. We observe no significant difference in pass@1 rates ($p = 0.14$), suggesting that typed programs are rejected for genuine errors.

5.3 Sensitivity to Compilation Choices

Each MultiPL-E compiler makes small choices about how to translate prompts that could have an impact on performance. We explore some of these choices below.

Comment style affects performance Most programming languages have several comment styles (e.g., single-line vs. multi-line). To investigate their impact, we consider PHP (MEDIUM) and Racket (NICHE). Our original prompts use single-line comments for both PHP and Racket, following conventional style. We run a set of experiments with Codex where we use multi-line comments. This improves the pass@1 rate for Racket (+1.9%, $p < 0.001$), but decreases it for PHP (-3.1% , $p = 0.001$).

Naming arguments improves performance for Perl Functions in Perl do not have formal named arguments. Instead, all arguments are passed in a special array. Our compiler to Perl produces a prompt that pops elements off the special array and names them, with the expectation that this would improve model performance. We run a follow-up experiment where we omit this, so the model has to infer everything about arguments from the natural language description and examples. This significantly lowers Codex's pass@1 rate (-8%; $p < 0.001$).

Our results show that NL2Code performance can be sensitive to prompt engineering choices for both high and low frequency languages.

5.4 Characterization of NL2Code Errors

NL2Code systems generate many failing programs—programs that produce errors or fail to pass unit tests—than programs which run successfully. This section presents a detailed evaluation of errors present in the Codex-generated functions for 4 languages: Python, C#, Swift, and Racket. See Appendix D for a full categorization.

We first identified specific error labels for each language and then grouped them into themes (e.g. "NullReference"). We produced five general error categories: RUNTIME, STATIC, TYPE, LANGUAGE, and MODEL. We group similar error sources together across languages, even if they occur in different contexts: for example, a function call with a value of the wrong type may fail at compile-time or runtime or depending on the language's type system.

The most common STATIC theme across all languages is "UndefinedIdentifier", which contains errors related to referencing non-existent terms. These errors can be caused in many ways – calls to functions not in the local context, use of Python-like keywords, or calls to methods from external libraries that were not imported.

Some errors in the RUNTIME category mimic those we expect from software engineers (e.g., index-out-of-range errors). However, others are unlike human mistakes. Notable themes in the latter group (MODEL) include generating code that throws exceptions on purpose and generating code in an entirely different language (e.g., Markdown, not Racket).

Finally, the category LANGUAGE includes multiple themes related to specifics of the target language itself. The "LanguageSpecific" theme contains idiosyncratic errors such as the requirement of labeled arguments in Swift. "DoesNotKnowSyntax" includes errors in Racket caused by incorrectly generated core language constructs.

6 Conclusion

We propose MultiPL-E, the first parallel multi-language benchmark for natural-language-to-code generation. We write compilers to translate the HumanEval benchmark suite of Python programs into 18 programming languages that span a spectrum of language features and popularity.

We present the first multi-language code correctness evaluation of two state-of-the-art NL2Code models: Codex and InCoder. We show that Codex performs best on JavaScript and does as well as Python on four other languages (C++, JavaScript, Scala, and TypeScript). Our results highlight the importance of testing: we do not find a strong correlation between perplexity and code correctness. In our detailed by-language analysis, we find a predictable effect of language frequency, but draw mixed conclusions about the impact of type annotations. Our detailed error analysis highlights common patterns in four languages, finding model errors that are both like and unlike those of human programmers.

Our publicly available benchmark is also easy to extend to new problems and languages. We hope it will help evaluate and develop future work on multi-language LLMs of code.

7 Acknowledgments

We thank Steven Holtzen for loaning us his GPUs. This work was partially supported by the National Science Foundation grant CCF-2052696.

References

Aghajanyan, A.; Huang, B.; Ross, C.; Karpukhin, V.; Xu, H.; Goyal, N.; Okhonko, D.; Joshi, M.; Ghosh, G.; Lewis, M.; and Zettlemoyer, L. 2022. CM3: A causal masked multimodal model of the internet. *arXiv preprint arXiv:2201.07520*.

Ahmed, T.; and Devanbu, P. 2022. Multilingual training for software engineering. In *Proceedings of the 44th International Conference on Software Engineering*. ACM.

Alur, R.; Bodik, R.; Juniwal, G.; Martin, M.; Raghothaman, M.; Seshia, S. A.; Singh, R.; Solar-Lezama, A.; Torlak, E.; and Udupa, A. 2013. Syntax-Guided Synthesis. In *Formal Methods in Computer-Aided Design (FMCAD)*.

Austin, J.; Odena, A.; Nye, M.; Bosma, M.; Michalewski, H.; Dohan, D.; Jiang, E.; Cai, C.; Terry, M.; Le, Q.; and Sutton, C. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Bates, D.; Mächler, M.; Bolker, B.; and Walker, S. 2015. Fitting Linear Mixed-Effects Models Using lme4. *Journal of Statistical Software*, 67(1): 1–48.

Black, S.; Biderman, S.; Hallahan, E.; Anthony, Q.; Gao, L.; Golding, L.; He, H.; Leahy, C.; McDonell, K.; Phang, J.; Pieler, M.; Prashanth, U. S.; Purohit, S.; Reynolds, L.; Tow, J.; Wang, B.; and Weinbach, S. 2022. GPT-NeoX-20B: An Open-Source Autoregressive Language Model. *arXiv preprint arXiv:2204.06745*.

Brown, T.; Mann, B.; Ryder, N.; Subbiah, M.; Kaplan, J. D.; Dhariwal, P.; Neelakantan, A.; Shyam, P.; Sastry, G.; Askell, A.; et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33: 1877–1901.

Chaudhuri, S.; Ellis, K.; Polozov, O.; Singh, R.; Solar-Lezama, A.; and Yue, Y. 2021. Neurosymbolic Programming. *Foundations and Trends in Programming Languages*, 7(3): 158–243.

Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; Pinto, H. P. d. O.; Kaplan, J.; Edwards, H.; Burda, Y.; Joseph, N.; Brockman, G.; et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Chowdhery, A.; Narang, S.; Devlin, J.; Bosma, M.; Mishra, G.; Roberts, A.; Barham, P.; Chung, H. W.; Sutton, C.; Gehrmann, S.; Schuh, P.; Shi, K.; Tsvyashchenko, S.; Maynez, J.; Rao, A.; Barnes, P.; Tay, Y.; Shazeer, N.; Prabhakaran, V.; Reif, E.; Du, N.; Hutchinson, B.; Pope, R.; Bradbury, J.; Austin, J.; Isard, M.; Gur-Ari, G.; Yin, P.; Duke, T.; Levskaya, A.; Ghemawat, S.; Dev, S.; Michalewski, H.; Garcia, X.; Misra, V.; Robinson, K.; Fedus, L.; Zhou, D.; Ippolito, D.; Luan, D.; Lim, H.; Zoph, B.; Spiridonov, A.; Sepassi, R.; Dohan, D.; Agrawal, S.; Omernick, M.; Dai, A. M.; Pillai, T. S.; Pellat, M.; Lewkowycz, A.; Moreira, E.; Child, R.; Polozov, O.; Lee, K.; Zhou, Z.; Wang, X.; Saeta, B.; Diaz, M.; Firat, O.; Catasta, M.; Wei, J.; Meier-Hellstern, K.; Eck, D.; Dean, J.; Petrov, S.; and Fiedel, N. 2022. PaLM: Scaling Language Modeling with Pathways. *arXiv preprint arXiv:2204.02311*.

Clement, C.; Drain, D.; Timcheck, J.; Svyatkovskiy, A.; and Sundaresan, N. 2020. PyMT5: multi-mode translation of natural language and Python code with transformers. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 9052–9065. Online: Association for Computational Linguistics.

Drori, I.; Zhang, S.; Shuttleworth, R.; Tang, L.; Lu, A.; Ke, E.; Liu, K.; Chen, L.; Tran, S.; Cheng, N.; Wang, R.; Singh, N.; Patti, T. L.; Lynch, J.; Shporer, A.; Verma, N.; Wu, E.; and Strang, G. 2022. A Neural Network Solves, Explains, and Generates University Math Problems by Program Synthesis and Few-Shot Learning at Human Level. *Proceedings of the National Academy of Sciences*, 119(32): e2123433119.

Feng, Z.; Guo, D.; Tang, D.; Duan, N.; Feng, X.; Gong, M.; Shou, L.; Qin, B.; Liu, T.; Jiang, D.; and Zhou, M. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. *arXiv preprint arXiv:2002.08155*.

Fried, D.; Aghajanyan, A.; Lin, J.; Wang, S.; Wallace, E.; Shi, F.; Zhong, R.; Yih, W.-t.; Zettlemoyer, L.; and Lewis, M. 2022. InCoder: A Generative Model for Code Infilling and Synthesis. *arXiv preprint arXiv:2204.05999*.

Gebru, T.; Morgenstern, J.; Vecchione, B.; Vaughan, J. W.; Wallach, H.; Iii, H. D.; and Crawford, K. 2021. Datasheets for datasets. *Communications of the ACM*, 64(12): 86–92.

Hellendoorn, V. J.; Bird, C.; Barr, E. T.; and Allamanis, M. 2018. Deep Learning Type Inference. In *Fse*.

Hendrycks, D.; Basart, S.; Kadavath, S.; Mazeika, M.; Arora, A.; Guo, E.; Burns, C.; Puranik, S.; He, H.; Song, D.; and Steinhardt, J. 2021. Measuring Coding Challenge Competence With APPS. *arXiv preprint arXiv:2105.09938*.

Kulal, S.; Pasupat, P.; Chandra, K.; Lee, M.; Padon, O.; Aiken, A.; and Liang, P. S. 2019. SPoC: Search-based Pseudocode to Code. In Wallach, H.; Larochelle, H.; Beygelzimer, A.; d'Alché-Buc, F.; Fox, E.; and Garnett, R., eds., *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc.

Lu, S.; Guo, D.; Ren, S.; Huang, J.; Svyatkovskiy, A.; Blanco, A.; Clement, C.; Drain, D.; Jiang, D.; Tang, D.; Li, G.; Zhou, L.; Shou, L.; Zhou, L.; Tufano, M.; Gong, M.; Zhou, M.; Duan, N.; Sundaresan, N.; Deng, S. K.; Fu, S.; and Liu, S. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. *arXiv preprint arXiv:2102.04664*.

Nijkamp, E.; Pang, B.; Hayashi, H.; Tu, L.; Wang, H.; Zhou, Y.; Savarese, S.; and Xiong, C. 2022. A Conversational Paradigm for Program Synthesis. *arXiv preprint arXiv:2203.13474*.

Pradel, M.; Gousios, G.; Liu, J.; and Chandra, S. 2020. TypeWriter: Neural Type Prediction with Search-Based Validation. In *Esecfse*.

Ren, S.; Guo, D.; Lu, S.; Zhou, L.; Liu, S.; Tang, D.; Sundaresan, N.; Zhou, M.; Blanco, A.; and Ma, S. 2020. CodeBLEU: a Method for Automatic Evaluation of Code Synthesis.

Tufano, M.; Drain, D.; Svyatkovskiy, A.; Deng, S. K.; and Sundaresan, N. 2020. Unit Test Case Generation with Transformers and Focal Context. *arXiv preprint arXiv:2009.10297*.

Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A. N.; Kaiser, L. u.; and Polosukhin, I. 2017. Attention is All you Need. In Guyon, I.; Luxburg, U. V.; Bengio, S.; Wallach, H.; Fergus, R.; Vishwanathan, S.; and Garnett, R., eds., *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc.

Wang, B.; and Komatsuzaki, A. 2021. GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model.

Wei, J.; Goyal, M.; Durrett, G.; and Dillig, I. 2020. LambdaNet: Probabilistic Type Inference Using Graph Neural Networks. In *International Conference on Learning Representations (ICLR)*.

Xu, F. F.; Alon, U.; Neubig, G.; and Hellendoorn, V. J. 2022. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, 1–10.

Yin, P.; Deng, B.; Chen, E.; Vasilescu, B.; and Neubig, G. 2018. Learning to Mine Aligned Code and Natural Language Pairs from Stack Overflow. In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, 476–486. New York, NY, USA: Association for Computing Machinery. ISBN 9781450357166.

Yu, T.; Zhang, R.; Yang, K.; Yasunaga, M.; Wang, D.; Li, Z.; Ma, J.; Li, I.; Yao, Q.; Roman, S.; Zhang, Z.; and Radev, D. 2018. Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, 3911–3921. Brussels, Belgium: Association for Computational Linguistics.

Zaremba, W.; Brockman, G.; and OpenAI. 2021. OpenAI Codex.

A Details of Language Translations

The tables below describe the details of all 18 language translations. Technical information regarding running experiments and evaluating generated programs can be found at the MultiPL-E website and in our code repository. Here we address language-specific decisions that are relevant to the prompt translation task. Specifically, we outline the following details:

1. The language version used as a reference for creating the value-to-value translation.
2. The stop tokens used for signaling the end of program generation. Across languages these reflect terms that begin and/or start code blocks (variations of `\n}` are common).
3. Details about prompt creation. This sections highlight the choice of comments and any necessary preamble information (e.g., the opening tag `<?php` in PHP)
4. Information about mapping values and/or types from Python. Notes here describe places where case-by-case decisions need to be made, or a language's limitations required not converting a subset of values and/or types. Omitted discussions represents straightforward conversions (e.g., integers in Perl).

BASH	
Reference Version for Translation	5.1.16
Stop Tokens	`'\n}'`
Prompt Information	We translate each Python docstring to Bash comments (each line prefixed with `#`). Each Python function signature is translated to a Bash function signature, which is of the form `function_name()`, as Bash functions do not have explicit parameters. Type annotations are translated to comments in the prompt, which describe the encoding (or none) for each of the function parameters. The shebang (`#!/bin/bash`) was prepended to the prompt.
Type Translations	Bash is not a general-purpose programming language, and its many quirks make translation challenging, particularly for data structures like lists and maps. While Bash has numerically indexed and string-associative arrays, the shell's ecosystem typically works with these structures in string-y formats: lists are typically whitespace separated elements; associative maps are in formats like comma-separated values (CSV). We use those conventions in our type translations.

C++	
Reference Version for Translation	C++17 compiled using g++17
Stop Tokens	`'\n}'`
Prompt Information	Each prompt contains C++ single line comments where each line is prefixed with `//`. Python function signatures are translated to C++ signatures and we add `#include` statements.
Type Translations	All Python integers are translated to C++ `long` and Python floats are translated to C++ `float`. A Python list is translated to `std::vector`, a dictionary to `std::map`, a tuple to `std::tuple`, a string to `std::string`, and Python's `Any` type to `std::any`. A new C++ union type is declared for each union type annotation in Python.

C#	
Reference Version for Translation	C# 5 with Mono 6.12
Stop Tokens	`'\n    }\n'`
Prompt Information	The prompt contains a class declaration with the translated method as its `public static` member and C# single line comments, where each line is prefixed with `//`. Adding a member of class also adds indentation to each line inside class declaration (note the indentation in the stop token). All function and argument names are converted to C#'s naming convention where the first letter of all words is in capital case.
Type Translations	Most types were translated to their C# direct equivalent (e.g. Python `tuple` to C# `tuple`). There are some exceptions: Python `int` is translated to a C# `long` and Python's `Any` type annotation is translated to C# object. Since C# does not support union types, we do not convert Python union annotations.

D	
Reference Version for Translation	dmd 2.100.0
Stop Tokens	`'\n\n'`, `'\void'`, `'\bool'`, `'\int'`
Prompt Information	The prompt was given as a multi-line comment (`/* ... */`).
Type Translations	Most types in Python have equivalents in D. One exception is Python integers, which we translate to `long`. Dictionaries are translated to `Nullable` of associative arrays, a built-in array that supports indices of any types. Associative arrays must be non-empty in D, so the `Nullable!(...)` template type is needed to wrap around the associative array, i.e. an empty array is denoted as the "null" state. Tuples are translated to the `Tuples!(...)` template type; however, the tuple type in D cannot be variable arity. Union types and `Any` are not translated.

GO	
Reference Version for Translation	1.18.1
Stop Tokens	`'\nfunc '`, `'struct'`, `'\n // '`
Prompt Information	The prompt is translated as a line comment (with `//`) above the function stub. For short functions, it is recommended to use single line comments.
Type Translations	Python Lists and Dictionaries were mapped to Go's Slices and Maps, respectively. Since Go requires type annotations, we utilized Python's type annotations to both translate the candidate function and the tests. Go requires explicitly declaring types for a compound datatype (e.g., a Python list `[1, 2, 3]` translates to `[]int{1, 2, 3}`). Go does not have an equivalent `Union`, `Option`, or `Tuple` data type, but it is possible to create a non-homogenous slice using `[]interface` – therefore we reject the two former and we convert the latter.
Other Notes	We consulted the following style guide as part of our translation to Go (https://go.dev/doc/effective_go).

JAVA	
Reference Version for Translation	OpenJDK 17
Stop Tokens	`'\n    }\n'`
Prompt Information	The prompt contains a class declaration with the translated method as its `public static` member and Java single line comments, where each line is prefixed with `//`. Adding a member of class also adds indentation to each line inside class declaration (note in the intention in the stop token). All function and arguments are converted to Java's naming convention where the first letter is lowercase and the first letter of all other words are capitalized.
Type Translations	The type translation from Python to Java is performed by translating a Python `int` to a Java `long`, Python `float` to Java `float`, a Python `list` to `Vector`, a `dictionary` to `HashMap`, a string to `String`, and Python's `Any` type annotation to Object. Since OpenJDK does not support tuples, we use `javatuples` library and translates Python tuples to `javatuples.Tuple`. Since Java does not support union types, we do not convert Python union annotations.

JAVASCRIPT	
Reference Version for Translation	18.6
Stop Tokens	`'\nfunction '`, `'\n /*'`, `'\n //'`, `'\nconsole .log'`
Prompt Information	We convert the Python prompt into a block of comments using `//`.
Type Translations	Most type translations are direct. Python lists and tuples were translated into JS arrays. Dictionaries were translated into objects.

JULIA	
Reference Version for Translation	1.7.3
Stop Tokens	`'\nfunction '`, `'\nmacro '`, `'\n \n '`
Prompt Information	Julia shares both its documentation and line comment syntax with Python, and thus the prompt is left unchanged by the translation.
Type Translations	We translate Python's `int` to `Int64`, `float` to `Float64`, and `List` to `Vector`. The only coercion required in the benchmarks come from the fact that Julia generates the type `Vector{Any}` for the unannotated empty vector. Thus, if the empty vector is given as an argument to the function, it is coerced to the expected (more specific) type. Julia has first-class support for `Union` types; therefore, we represent Unions directly and `Optional<T>` as the type `Union{T, Nothing}`.

LUA	
Reference Version for Translation	5.3
Stop Tokens	`'\nlocal '`, `'\nfunction '`, `'\n --'`, `'\n \n '`
Prompt Information	We convert the Python prompt to a block of single-line comments using `--`.
Type Translations	The only data structure in Lua is a table, and tables with integer indices behave like lists. Thus we translate Python dictionaries, tuples, and lists to tables.

PERL	
Reference Version for Translation	5.34
Stop Tokens	`'\nsub'`, `'\n#'`, `'\n\n'`
Prompt Information	We convert the Python prompt to a block of single-line comments, using `#`.
Type Translations	We are careful to pass data structures by reference; we translate Python lists and tuples to anonymous arrays, and dictionaries to anonymous hashes. Perl lacks a Boolean type; we translate `True` to 1 and `False` to the empty string, since these are the values returned by logical operators.

PHP	
Reference Version for Translation	8.1.2 (cli)
Stop Tokens	`'\nfunction'`, `'\n?>'`, `'\n'`, `'\n#'`
Prompt Information	In our full translation, the prompt was given as single-line comments, using `//`, rather than using PHP's two other comment styles (single line `#` and multi-line `/* ... */`). The PHP opening tag, `<?php`, was prepended to the prompt, and the closing tag was omitted, following the recommendation for a file that only contains PHP code (https://www.php.net/manual/en/language.basic-syntax.phptags.php).
Type Translations	PHP arrays are actually ordered maps, so Python lists, tuples, and dictionaries were translated to arrays. Arrays were defined using the default syntax, `array()`, instead of the shorthand `[]`. Strings are double quoted, and Python's `None` is translated to `null`.

PYTHON	
Reference Version for Translation	3.10
Stop Tokens	`'\ndef'`, `'\n#'`, `'\nif'`, `'\nclass'`
Prompt Information	The prompt was presented as in the original HumanEval dataset: a multi-line docstring. If type annotations were present, the typing library was imported via an import statement at the beginning of the prompt.
Type Translations	The Python translation is trivial: each type is translated to itself.

R	
Reference Version for Translation	4.1
Stop Tokes	`'\n#','\n"`'`
Prompt Information	We convert the Python prompt to a block of single-line comments using `#`.
Type Translations	R vectors are more commonly used than R lists; however, R vectors are restricted to storing homogenous data types. We translate Python Lists and Tuples to R vectors using the `c()` function when possible (i.e., when the contents are homogenous), and to R lists using the `list()` function otherwise. We convert Python dictionaries to named lists. R, like Python, supports both single and double quoted strings.

RACKET	
Reference Version for Translation	8.2
Stop Tokens	`'\n(define ', '\n#|', '\n;', '\n('`
Prompt Information	We convert the Python prompt to a block of single-line comments using ';'.
Type Translations	We translate Python Lists and Tuples to Racket lists using `(list )`. We convert Python dictionaries to hash maps using `(hash )`. Racket does not support single-quoted strings, so we convert all strings to double-quoted strings.

RUBY	
Reference Version for Translation	3.0.2
Stop Tokens	`'\nclass', '\ndef', '\n#', '\n\n'`
Prompt Information	Although there are block comments in Ruby (`=begin ... =end`), they are discouraged by community style guides. Therefore, the prompt was converted to a block of single-line comments prefixed by `#`.
Type Translations	Python Lists and Tuples were mapped to Ruby Arrays with the `[...]` shorthand per style guides. The idiomatic => Ruby syntax was used for dictionary creation. While Ruby supports both double- and single-quoted strings, Python strings were converted to double-quoted Ruby strings as they work with string interpolation.
Other Notes	We consulted the following two style guides as part of our translation to Ruby (https://ruby-style-guide.shopify.dev/, https://github.com/rubocop/ruby-style-guide).

RUST	
Reference Version for Translation	1.59.0
Stop Tokens	`'\n}'`
Prompt Information	A doc comment is used to indicate that the prompt information corresponds to the behavior of the function and not internal implementation details (each line prefixed with `///`). No arguments are annotated with `mut` - in all cases (we used owned values) they can be moved to a mutable variable if necessary, and unnecessary mutable annotations may be confusing.
Type Translations	All annotated values are owned. While in Rust it sometimes makes sense to accept borrowed values (for example, if no mutation or move is necessary), it is difficult to infer when this is appropriate from the Python signature or prompt. Inferring when a borrowed result type could be used would be even more difficult. Thus, `str` is translated as `String` and `List` is translated to `Vec`. `Tuple` is translated to Rust's tuple, `dict` to Rust's `std::collections::HashMap`, and `Optional` to `Option`. While Python's `int` must support at least 64 bit integers, the more idiomatic `isize` is used to represent them in Rust. Python's `float` is translated to the corresponding `f64` and `bool` to `bool`. Problems annotated with a `Union`, `Any`, or `Ellipsis` are not supported.

SCALA	
Reference Version for Translation	Scala 2.23
Stop Tokens	`'\n    }\n'`
Prompt Information	The prompt contains a class declaration with the translated method as its member and Scala single line comments where each line is prefixed with `//`. Adding a member of class also adds indentation to each line inside class declaration (note the indentation in the stop token). All function and argument names are converted Scala's naming convention where the first letter is lowercase and the first letter of all other words is in capital case.
Type Translations	The type translation from Python to Scala is performed by translating a Python `int` to a Scala `long`, Python `float` to Scala `float`, a Python `list` to Scala `List`, a Python `dictionary` to Scala `Dictionary`, a Python string to Scala string, a Python `tuple` to Scala `Tuple`, and Python's `Any` type annotation to Scala `Any`. Python union annotations of two types is converted to Scala's `Either` type. Problems with `Union` of more than two types are not supported.

SWIFT	
Reference Version for Translation	5.8
Stop Tokens	`'\n}'`
Prompt Information	The prompt is given by doc comments (prepended with `///`). For documenting function behavior, doc comments are preferred over standard comments.
Type Translations	Python Lists, Dictionaries and Tuples were mapped to Swift Lists, Dictionaries and Tuples, respectively. Untyped Python parameters were mapped to `AnyHashable` in Swift, as opposed to `Any`, as it allows for equality comparisons and storage in dictionaries, so is the closest equivalent to untyped Python values. Optional types or Unions with `None` in Python were converted to `?` optional types in Swift, binary Union types were converted to `Result` types, and larger Union types were converted to generated algebraic datatypes. The generated algebraic datatype definitions (and `Error` protocol conformance in the case of `Result`) were inserted into the prompt, above the doc comments.
Other Notes	We consulted the following style guide as part of our translation to Swift (https://www.swift.org/documentation/api-design-guidelines/).

TYPESCRIPT	
Reference Version for Translation	TypeScript compiler version 4.5, Node version 18.6
Stop Tokens	`'\nfunction ', '\n /*', '\n //', '\nclass '`
Prompt Information	We convert the Python prompt into a block of comments using `//`.
Type Translations	Types are translated by utilizing the annotations provided in our Python tests. Lists and tuples were translated into arrays. Dictionaries were translated into objects.

B Datasheet

The datasheet below is derived from the following work (Gebru et al. 2021):

> Timnit Gebru, Jamie Morgenstern, Briana Vecchione, Jennifer Wortman Vaughan, Hanna Wallach, Hal Daumé III, and Kate Crawford. "Datasheets for datasets." Communications of the ACM, 2021, 86-92.

Motivation

- **For what purpose was the dataset created?**

  The dataset was originally created to evaluate the performance of the large language language model. It was translated from Python to other programming languages to allow for evaluation of large language models on other programming languages.
- **Who created the dataset?**

  It was originally created by Chen et al. (2021) and modified by the authors of this paper.
- **Who funded the creation of the dataset?** This work was partially supported by the National Science Foundation.

Composition

- **What do the instances that comprise the dataset represent?**
  The instances of the dataset represent programming problems in 18 programming languages.
- **How many instances are there in total?**
  There are 2,898 instances (the modified set of 161 Python problems multiplied by 18 programming languages).
- **Does the dataset contain all possible instances or is it a sample (not necessarily random) of instances from a larger set?**
  The dataset cleans the original dataset and excludes 3 of 164 problems as described in §3.2.
- **What data does each instance consist of?** Each instance is a programming problem with a problem description in natural language, a function signature, and unit tests.
- **Is there a label or target associated with each instance?**
  Each instance is numbered and labeled by the name of the function it tests and the language it is written in.
- **Is the dataset self-contained, or does it link to or otherwise rely on external resources?**
  The dataset is self-contained.

Collection process

- **How was the data associated with each instance acquired?**
  The original Python dataset was manually cleaned. The versions for other programming languages and prompt variations were produced by a suite of compilers.
- **Over what timeframe was the data collected?**
  May–August 2022
- **Were any ethical review processes conducted?**
  Not applicable. The dataset adapts a open source dataset released under the terms of the MIT license.

Preprocessing/cleaning/labeling

- **Was any preprocessing/cleaning/labeling of the data done?**
  We added missing type annotations, formatted examples to use docstrings consistently, and changed random tests into unit tests in two problems.
- **Was the "raw" data saved in addition to the preprocessed/cleaned/labeled data?**
  The raw data is available at https://github.com/openai/human-eval.
- Is the software that was used to preprocess/clean/label the data available?
  The cleaning process described above was manual.

Uses

- **Has the dataset been used for any tasks already?**
  The dataset has been used for the NL2Code task and for comparing the performance of two LLMs of code.
- **Is there a repository that links to any or all papers or systems that use the dataset?**
  https://github.com/nuprl/MultiPL-E
- **What other tasks could the dataset be used for?**
  The dataset could be used to evaluate other LLMs of code, or potentially to improve their performance.

Distribution

- **Will the dataset be distributed to third parties outside of the entity?**
  Yes.
- **How will the dataset be distributed?**
  The dataset is publicly available at https://github.com/nuprl/MultiPL-E
- **When will the dataset be distributed?**
  Immediately.
- **Will the dataset be distributed under a copyright or other intellectual property (IP) license, and/or under applicable terms of use (ToU)?**
  No.
- **Have any third parties imposed IP-based or other restrictions on the data associated with the instances?**
  No.
- **Do any export controls or other regulatory restrictions apply to the dataset or to individual instances?**
  No.

Maintenance

- **Who will be supporting/hosting/maintaining the dataset?**
  The original authors.
- **How can the owner/curator/manager of the dataset be contacted?**
  See the dataset website.
- **Is there an erratum?**
  No. Any identified and confirmed errors will be acknowledged as part of the repository.
- **Will the dataset be updated (for example, to correct labeling errors, add new instances, delete instances)?**
  Yes.
- **Will older versions of the dataset continue to be supported/hosted/maintained?**
  Yes.
- **If others want to extend/augment/build on/contribute to the dataset, is there a mechanism for them to do so?**
  Yes, as described in the paper and website.

C Complete Statistical Findings

We use binomial mixed-effects models fitted with the lme4 library in R for significance testing. A binomial distribution is appropriate because our outcomes consist of proportions of successes and failures; we use the number of completions (200, except in rare failure cases) as weights.

We fit models to the Codex pass@1 completion rates in all experiments reported below. We treat problem number as a random effect to account for variability inherent to per-problem differences. For comparisons that do not break down effects by language, we also include language as a random effect. We include random slopes and intercepts for random effects except where noted.

Values that are statistically significant with a threshold of $p = 0.5$ are displayed in **bold**.

C.1 Mixed-Effects Results from Section 4.1

To quantify the differences in performance among programming languages, a model with a fixed effect of programming language and random effects for problem number was fitted to the Codex pass@1 data.

Dummy coding was used with Python as the reference level; slopes for each language indicate differences between the pass@1 rate for Python and that language.

Table 2 shows the full estimates found by the model.

Fixed effects	$\widehat{\beta}$	z	p
Intercept	-0.48 (+/- 0.4)	-1.1	0.27
Bash	-2.59 (+/- 0.3)	-7.7	**< 0.0001**
C++	0.10 (+/- 0.4)	0.3	0.77
C#	-4.09 (+/- 0.6)	-7.2	**< 0.0001**
D	-4.79 (+/- 0.5)	-9.7	**< 0.0001**
Go	-2.61 (+/- 0.4)	-6.5	**< 0.0001**
Java	-1.28 (+/- 0.3)	-3.9	**< 0.0001**
Julia	-1.91 (+/- 0.4)	-5.2	**< 0.0001**
JavaScript	-0.27 (+/- 0.3)	-0.8	0.43
Lua	-1.04 (+/- 0.4)	-2.8	**0.005**
Perl	-2.0 (+/- 0.4)	-5.3	**< 0.0001**
PHP	-0.30 (+/- 0.4)	-0.8	0.40
R	-3.69 (+/- 0.4)	-8.5	**< 0.0001**
Ruby	-0.68 (+/- 0.3)	-2.3	**0.024**
Racket	-3.78 (+/- 0.4)	-9.8	**< 0.0001**
Rust	-1.07 (+/- 0.3)	-3.4	**< 0.0001**
Scala	-0.52 (+/- 0.3)	-1.6	0.10
Swift	-1.8 (+/- 0.3)	-5.7	**< 0.0001**
TypeScript	-0.27 (+/- 0.3)	-0.9	0.39

Table 2: Mixed-effects results for Codex language comparison

A similar model was fit to the InCoder pass@1 data, but without random slopes, because the very low pass rates for many problems makes the random effects estimates unstable. Table 3 shows the full estimates found by the model.

Fixed effects	$\widehat{\beta}$	z	p
Intercept	-4.35 (+/- 0.3)	-14.5	< **0.0001**
Bash	-3.3 (+/- 0.03)	-131.1	< **0.0001**
C++	-0.99 (+/- 0.02)	-57.1	< **0.0001**
C#	-1.94 (+/- 0.02)	-100.9	< **0.0001**
D	-3.97 (+/- 0.03)	-139.5	< **0.0001**
Go	-1.62 (+/- 0.02)	-86.3	< **0.0001**
Java	-1.29 (+/- 0.02)	-72.5	< **0.0001**
Julia	-4.90 (+/- 0.04)	-132.6	< **0.0001**
JavaScript	-0.97 (+/- 0.02)	-56.1	< **0.0001**
Lua	-2.21 (+/- 0.02)	-111.2	< **0.0001**
Perl	-2.43 (+/- 0.02)	-118.2	< **0.0001**
PHP	-1.76 (+/- 0.02)	-93.6	< **0.0001**
R	-2.80 (+/- 0.02)	-128.2	< **0.0001**
Ruby	-1.87 (+/- 0.02)	-98.4	< **0.0001**
Racket	-3.41 (+/- 0.02)	-138.2	< **0.0001**
Rust	-2.74 (+/- 0.02)	-125.3	< **0.0001**
Scala	-1.92 (+/- 0.02)	-100.3	< **0.0001**
Swift	-2.05 (+/- 0.02)	-105.3	< **0.0001**
TypeScript	-1.11 (+/- 0.02)	-63.6	< **0.0001**

Table 3: Mixed-effects results for InCoder language comparison

C.2 Mixed-Effects Results for Section 4.2

A mixed-effects model was fit to the InCoder pass@1 rates to explore how the translation components affect its performance. This model compared InCoder pass@1 rates for four experiments: Doctest-Only Translation, Full Translation, No Translation, and Remove Doctests. Experiment was treated as a fixed-effect, with Python and Doctest-Only Translation as the reference levels. Random effects for language were included; random effects for problem were not included, as the extremely low pass rates for many problems caused instability in estimating them. Table 4 shows the full estimates found by the model.

Fixed effects	$\widehat{\beta}$	z	p
(Intercept)	-2.97 (+/- 0.2)	-15.6	< **0.001**
Remove	0.20 (+/- 0.07)	2.8	**0.005**
No Translation	0.03 (+/- 0.03)	1.0	0.32
Full Translation	0.02 (+/- 0.02)	1.3	0.20

Table 4: Mixed-effects results for the InCoder ablation study

A similar mixed-effects model was fit to understand the impact of translating natural language terms and doctests on Codex performance. This model compared Codex pass@1 rates for four experiments: Doctest-Only Translation, Full Translation, No Translation, and Remove Doctests. Experiment was treated as a fixed-effect, with Python and Doctest-Only Translation as the reference levels. Random effects for problem and language were included. Table 5 shows the full estimates found by the model.

Fixed effects	$\widehat{\beta}$	z	p
(Intercept)	-1.24 (+/- 0.3)	-4.4	< **0.0001**
Full Translation	0.04 (+/- 0.02)	2.2	**0.03**
No Translation	-0.08 (+/- 0.1)	-1.3	0.2
Remove	-0.35 (+/- 0.1)	-3.8	< **0.0001**

Table 5: Mixed-effects results for the Codex ablation study

A second model was fitted for Codex treating both Language and Experiment as fixed-effects, with interaction terms included. For this model, we include only random intercepts but not random slopes for Problem, because of the large number of effects the model must estimate. Tables 6 and 7 show the full estimates found by the model.

Fixed effects	$\widehat{\beta}$	z	p
(Intercept)	-0.44 (+/- 0.2)	-2.2	**0.03**
Full Translation	-0.006 (+/- 0.02)	-0.3	0.78
No Translation	-0.07 (+/- 0.02)	-3.1	**0.002**
Remove	-0.14 (+/- 0.02)	-6.6	< **0.0001**
Bash	-1.75 (+/- 0.02)	-77.8	< **0.0001**
C++	0.17 (+/- 0.02)	8.1	< **0.0001**
C#	-1.45 (+/- 0.02)	-65.7	< **0.0001**
D	-1.97 (+/- 0.02)	-86.5	< **0.0001**
Go	-1.14 (+/- 0.02)	-52.5	< **0.0001**
Java	-0.63 (+/- 0.02)	-29.2	< **0.0001**
Julia	-0.87 (+/- 0.02)	-40.4	< **0.0001**
JavaScript	0.15 (+/- 0.02)	7.2	< **0.0001**
Lua	-0.48 (+/- 0.02)	-22.7	< **0.0001**
Perl	-1.10 (+/- 0.02)	-51.0	< **0.0001**
PHP	-0.006 (+/- 0.02)	-0.3	0.76
R	-2.02 (+/- 0.02)	-88.8	< **0.0001**
Ruby	-0.31 (+/- 0.02)	-14.4	< **0.0001**
Racket	-2.36 (+/- 0.02)	-100.3	< **0.0001**
Rust	-0.30 (+/- 0.02)	-14.0	< **0.0001**
Scala	-0.24 (+/- 0.02)	-11.0	< **0.0001**
Swift	-0.62 (+/- 0.02)	-28.9	< **0.0001**
TypeScript	0.10 (+/- 0.02)	4.8	< **0.0001**

Table 6: Mixed-effects results for the Codex ablation study by language, main effects

Fixed effects	$\widehat{\beta}$	z	p
Full Translation*Bash	-0.02 (+/- 0.03)	-0.5	0.58
No Translation*Bash	-0.47 (+/- 0.03)	-14.4	< **0.0001**
Remove*Bash	-0.59 (+/- 0.03)	-17.9	< **0.0001**
Full Translation*C++	-0.03 (+/- 0.03)	-0.9	0.35
No Translation*C++	-0.15 (+/- 0.03)	-5.0	< **0.0001**
Remove*C++	-0.11 (+/- 0.03)	-3.7	**0.0002**
Full Translation*C#	-0.02 (+/- 0.03)	-0.6	0.58
No Translation*C#	0.05 (+/- 0.03)	1.6	0.10
Remove*C#	0.2 (+/- 0.03)	6.9	< **0.0001**
Full Translation*D	0.02 (+/- 0.03)	0.5	0.59
No Translation*D	0.20 (+/- 0.03)	6.4	< **0.0001**
Remove*D	0.10 (+/- 0.03)	3.2	**0.001**
Full Translation*Go	-0.03 (+/- 0.03)	-0.8	0.41
No Translation*Go	-0.03 (+/- 0.03)	-1.0	0.32
Remove*Go	0.05 (+/- 0.03)	1.8	0.08
Full Translation*Java	0.12 (+/- 0.03)	3.9	< **0.0001**
No Translation*Java	0.14 (+/- 0.03)	4.5	< **0.0001**
Remove*Java	0.12 (+/- 0.03)	4.0	< **0.0001**
Full Translation*Julia	0.05 (+/- 0.03)	1.8	0.07
No Translation*Julia	0.05 (+/- 0.03)	1.7	0.10
Remove*Julia	-0.09 (+/- 0.03)	-2.9	**0.004**
Full Translation*JavaScript	0.01 (+/- 0.03)	0.4	0.66
No Translation*JavaScript	0.08 (+/- 0.03)	2.6	**0.01**
Remove*JavaScript	-0.09 (+/-0.03)	-2.8	**0.005**
Full Translation*Lua	0.06 (+/- 0.03)	1.9	0.06
No Translation*Lua	-0.04 (+/- 0.03)	-1.3	0.19
Remove*Lua	-0.12 (+/- 0.03)	-3.8	**0.0001**
Full Translation*Perl	0.23 (+/- 0.03)	7.5	< **0.0001**
No Translation*Perl	-0.25 (+/- 0.03)	-8.1	< **0.0001**
Remove*Perl	-0.21 (+/- 0.03)	-6.9	< **0.0001**
Full Translation*PHP	0.06 (+/- 0.03)	1.8	0.07
No Translation*PHP	0.02 (+/- 0.03)	0.6	0.58
Remove*PHP	-0.26 (+/- 0.03)	-8.6	< **0.0001**
Full Translation*R	0.22 (+/- 0.03)	7.0	< **0.0001**
No Translation*R	0.26 (+/- 0.03)	8.1	< **0.0001**
Remove*R	-0.11 (+/- 0.03)	-3.5	0.0004
Full Translation*Ruby	0.012 (+/- 0.03)	0.4	0.68
No Translation*Ruby	0.02 (+/- 0.03)	0.5	0.58
Remove*Ruby	-0.19 (+/- 0.03)	-6.1	< **0.0001**
Full Translation*Racket	0.04 (+/- 0.03)	1.2	0.23
No Translation*Racket	-0.07 (+/- 0.03)	-2.0	0.05
Remove*Racket	-0.21 (+/- 0.03)	-6.1	< **0.0001**
Full Translation*Rust	0.03 (+/- 0.03)	1.0	0.31
No Translation*Rust	-0.04 (+/- 0.03)	-1.5	0.14
Remove*Rust	-0.34 (+/- 0.03)	-11.2	< **0.0001**
Full Translation*Scala	0.01 (+/- 0.03)	0.5	0.64
No Translation*Scala	0.17 (+/- 0.03)	5.5	< **0.0001**
Remove*Scala	0.03 (+/- 0.03)	1.1	0.26
Full Translation*Swift	-0.01 (+/- 0.03)	-0.5	0.63
No Translation*Swift	-0.39 (+/- 0.03)	-13.0	< **0.0001**
Remove*Swift	-0.24 (+/- 0.03)	-8.0	< **0.0001**
Full Translation*TypeScript	0.05 (+/- 0.03)	1.6	0.11
No Translation*TypeScript	0.11 (+/- 0.03)	3.8	**0.0002**
Remove*TypeScript	-0.25 (+/- 0.03)	-8.4	< **0.0001**

Table 7: Mixed-effects results for the Codex ablation study by language, interaction effects

C.3 Mixed-Effects Results for Section 5.1 and Section 5.2

A mixed-effects model treating Frequency and Static Type-checking as fixed-effects, with random effects for language and problem, was fit to the data. Interaction terms were included for Typed with each frequency category. Table 8 shows the full estimates found by the model.

Fixed effects	$\widehat{\beta}$	z	p
(Intercept)	-0.47 (+/- 0.3)	-1.5	0.14
Low	-1.95 (+/- 0.4)	-5.0	**< 0.001**
Medium	-0.31 (+/- 0.3)	-1.2	0.22
Niche	-1.7 (+/- 0.6)	-3.0	**0.002**
Typed	-0.3 (+/- 0.3)	-1.0	0.33
Low*Typed	1.49 (+/- 0.5)	3.0	**0.003**
Medium*Typed	-1.49 (+/- 0.5)	-3.1	**0.002**
Niche*Typed	-0.31 (+/- 0.8)	-0.4	0.70

Table 8: Mixed-effects results for language frequency and static type-checking comparison

C.4 Mixed-Effects Results from Section 5.2

A mixed-effects model testing the effect of removing Python type annotations was fit treating Annotations as a fixed-effect and problem as a random effect. Table 9 shows the full estimates found by the model.

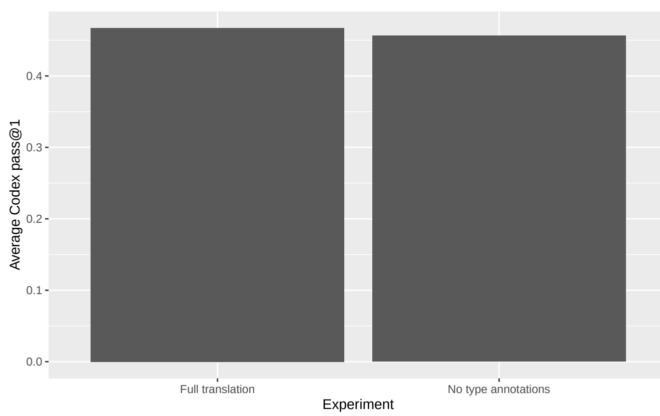

Figure 8: Impact of Python type annotations on Codex performance

Fixed effects	$\widehat{\beta}$	z	p
Intercept	-0.26 (+/- 0.5)	-0.5	0.60
Annotations	-0.21 (+/-).2)	-1.2	0.22

Table 9: Mixed-effects results for Python type annotation experiments

A mixed-effects model testing the effect of weakening TypeScript annotations to Any and running without static type-checking was fit. There were three fixed-effects: Any, comparing TypeScript with precise types to TypeScript with all Any types; JS, comparing TypeScript with annotations to JavaScript; and NoCheck, comparing TypeScript with and without static type-checking. Table 10 shows the full estimates found by the model.

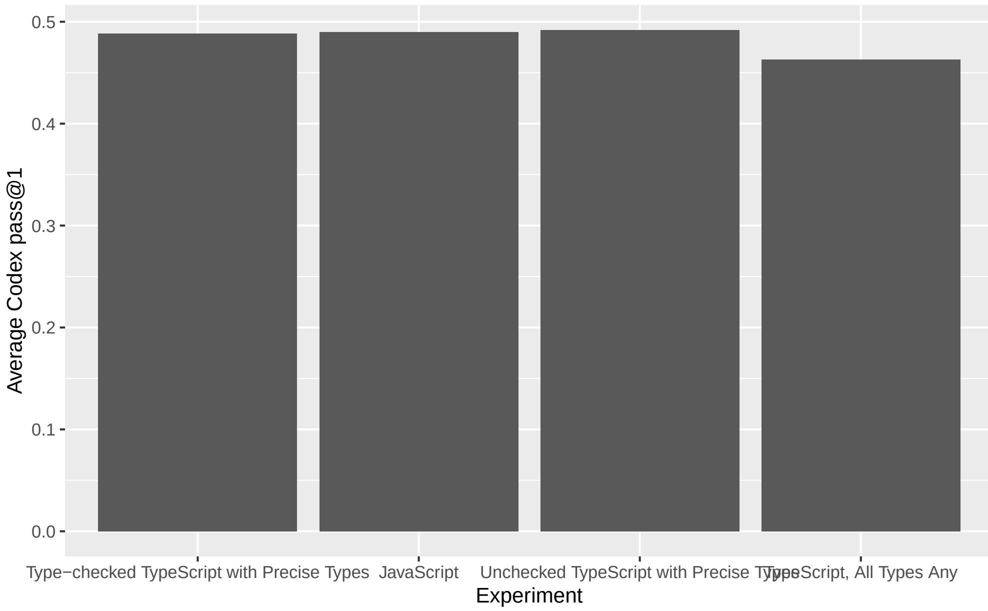

Figure 9: Impact of type-checking and precise type annotations on TypeScript performance

Fixed effects	$\widehat{\beta}$	z	p
Intercept	-0.24 (+/- 0.4)	-0.6	0.56
JavaScript	-0.03 (+/- 0.03)	-1.2	0.23
Any Types	-0.38 (+/- 0.03)	-13.3	< **0.001**
NoCheck	0.04 (+/- 0.03)	1.5	0.14

Table 10: Mixed-effects results for TypeScript experiments

C.5 Mixed-Effects Results from Section 5.3

Tables 11 and 12 shows the results of singe-line versus multi-line comments for PHP and Racket. Separate models were run for each language, with multi-line as a fixed effect and problem number as a random effect.

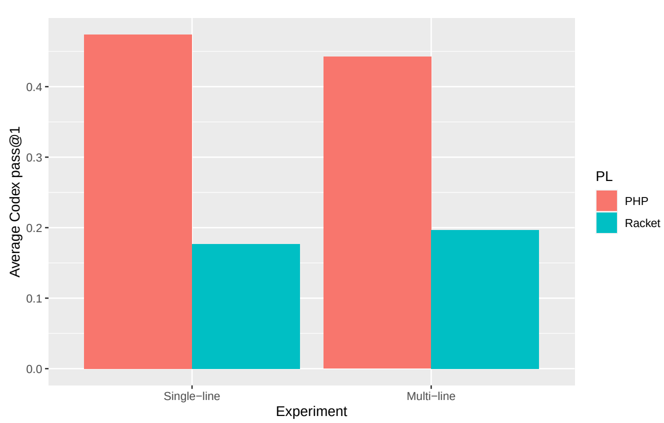

Figure 10: Impact of comment style on Codex performance for PHP and Racket

Fixed effects	$\widehat{\beta}$	z	p
Intercept	-0.46 (+/- 0.4)	-1.2	0.22
Multi-line	-0.43 (+/-0.1)	-3.3	**0.001**

Table 11: Mixed-effect model estimates for PHP comment experiment

Fixed effects	$\widehat{\beta}$	z	p
Intercept	-4.62 (+/- 0.4)	-10.9	< **0.0001**
Multi-line	1.26 (+/-0.2)	6.4	< **0.0001**

Table 12: Mixed-effect model estimates for Racket comment experiment

Table 13 shows the results of comparing Perl with and without an argument-naming line after the function signature. Argument-naming was treated as a fixed effect and problem number as a random effect.

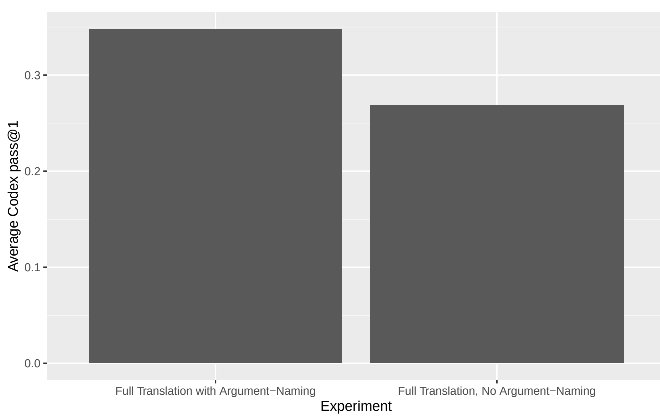

Figure 11: Impact of argument-naming line on Codex performance for Perl

Fixed effects	$\widehat{\beta}$	z	p
Intercept	-3.03 (+/- 0.4)	-7.9	**<0.0001**
Argument-naming	0.81 (+/-0.2)	3.6	**0.0008**

Table 13: Mixed-effect model estimates for Perl experiment

Table 14 shows the results of comparing Bash with and without encoding-specifying comments. Comments and NL Translation were treated as fixed effects and problem number as a random effect; an interaction term for Comments and NL Translation was also included.

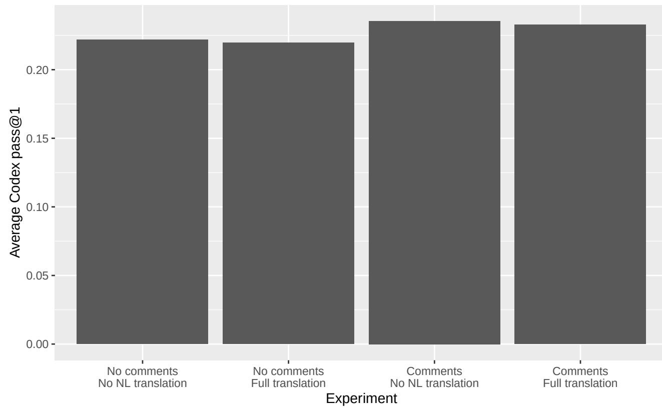

Figure 12: Impact of encoding comments and NL translation on Codex performance for Bash

Fixed effects	$\widehat{\beta}$	z	p
Intercept	-3.09 (+/- 0.3)	-9.9	< **0.001**
Comments	0.01 (+/- 0.1)	0.08	0.94
Rewording	-0.04 (+/- 0.03)	-1.3	0.19
Comments*Rewording	0.08 (+/-0.4)	1.8	0.07

Table 14: Mixed-effect model estimates for Bash experiment

C.6 Mixed-Effects Results for Language Feature

We categorize problems into groups based on which Python language features they use: dictionaries, tuples, booleans, lists, or none of the above. We base these categorizations on the Python type annotations for each problem. Problems were coded 1 Tuple, List, Bool, and Dictionary if they contain a type annotation for the respective feature, and 0 otherwise. Figure 13 shows the performance by language on each type of problem. There are only 3 problems in the dictionary category, so these results should be interpreted with caution.

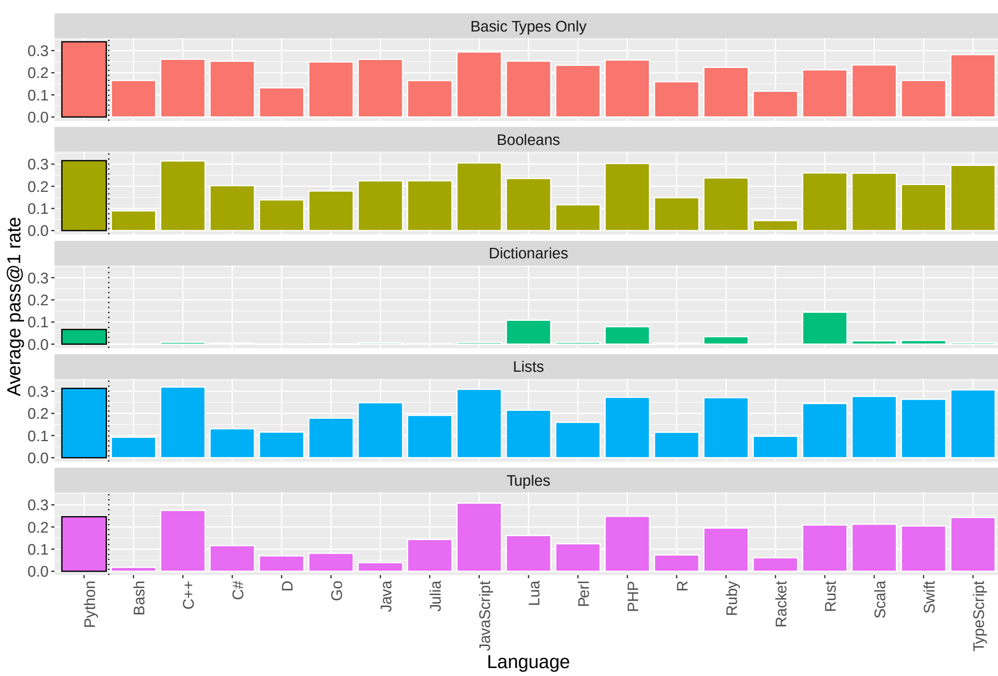

Figure 13: Impact of programming language features on Codex pass@1 performance by language

We fit a mixed-effects model to understand how Codex pass@1 rates are affected by the language features used in the problem, using Tuple, List, Bool, and Dictionary as fixed-effects, with random effects for problem and language. Table 15 shows the full estimates found by the model.

Fixed effects	$\widehat{\beta}$	z	p
(Intercept)	-1.19 (+/- 0.3)	-3.5	< **0.001**
List	-0.15 (+/- 0.4)	-0.3	0.73
Bool	0.10 (+/- 0.6)	0.2	0.86
Tuple	-0.73 (+/- 0.9)	-0.8	0.40
Dictionary	-3.27 (+/- 1.8)	-1.8	0.07

Table 15: Mixed-effects results for the impact of language features

D Characterization of Code Generation Errors

This section provides details regarding our error evaluation study as overviewed in Section 5.4. First we discuss the process of categorizing errors in a multi-language context. Then we provide the full set of themes, errors, and counts across the four studied languages: Python (HIGH, untyped), C# (MEDIUM, typed), Swift (LOW, typed), and Racket (NICHE, untyped). Finally, we showcase full code examples of a variety of errors.

D.1 Notes on Process & Findings

To perform the evaluation, we chose two typed languages and two untyped languages across all four frequency categories. A language expert then performed a manual investigation of a subset of the completions to derive a set of common error types. These errors could be associated with common error labels in a language (e.g., `NameError` in Python) or an observed phenomenon (e.g., `UseofDeprecatedIdentifiers` in Swift). Then, through an iterative process of manual inspection and automatic error detection via analyzing evaluation output, we developed a set of error labels unique to each language. We then arrived at the multi-language themes and categories via discussion and consensus.

The multi-language nature of the evaluation contributes to variation between the language classifications. For instance, languages vary significantly in the specificity of their error messages. Consider the theme of `TimeoutOrInfiniteRecursion`: Python has a specific error message `RecursionError` when it encounters an infinite recursive loop, whereas Racket will simply evaluate indefinitely. As the generated standard output and standard error were used for automatic classifications, there may be variations in how errors were counted depending on the error messages and precision of string search terms.

Overall, each error label is specific to the language under study and was subject to different levels of manual assessment. Therefore, the prevalence of a theme, rather than a specific error label or even category, likely provides a better source of inter- and intra-language information. Although the four languages in our study address different language variations (typed/untyped, frequency), they are not representative of all languages in our benchmark nor additional unstudied languages. Therefore, it is likely there are error labels, themes, and potentially categories that are missing from this characterization. Errors classified under the theme "AssertionFailed" describe errors from generated code with correct syntax which produces incorrect output. Other than via manual inspection of the over 10,000+ errors per language, there is no clear method of more precisely classifying errors of that type.

D.2 Complete Error Themes

In Tables 16 - 19 below, rows with the gray background are the most frequent error in that category for the specific language. Items in *italics* are errors directly referenced in Section 5.4. There are around 32,000 completions for each language for our full translation. Variations in the reported counts below are due to support for a different number of prompts for each language and completions which generate multiple errors on failure. In the later case, we count all present errors.

Category	Theme	Error	Count	Example
Runtime	AssertionFailed	AssertionError	17104	
Runtime	TimeoutOrInfiniteRecursion	Timeout	4462	
Runtime	InvalidDataStructureOperation	IndexError	460	
Runtime	TimeoutOrInfiniteRecursion	RecursionError	86	
Runtime	InvalidDataStructureOperation	AttributeError	5	
Runtime	InvalidDataStructureOperation	KeyError	2	
Runtime	DivisionByZero	ZeroDivisionError	1	
Static	*UndefinedIdentifier*	NameError	2942	Fig. 16
Static	*UndefinedIdentifier*	UnboundLocalError	1	
Type	InvalidTypeConversion	TypeError	123	
Language	Miscellaneous	ValueError	334	
Language	Miscellaneous	IndentationError	32	
Language	LanguageSpecific	EOFError	1	Fig. 22
Model	OutOfTokens	SyntaxError	333	
Model	*ExceptionInGeneratedCode*	NotImplementedError	253	Fig. 17

Table 16: Error Categories, Themes, and Labels for Python.

Category	Theme	Error	Count	Example
Runtime	AssertionFailed	AssertionError	22473	
Runtime	TimeoutOrInfiniteRecursion	Timeout	4470	
Runtime	NullReference	NullReferenceException	1201	
Runtime	InvalidDataStructureOperation	ArgumentOutOfRangeException	632	
Runtime	InvalidDataStructureOperation	InvalidOperationException	93	
Runtime	InvalidDataStructureOperation	IndexOutOfRangeException	82	
Runtime	InvalidDataStructureOperation	KeyNotFoundException	4	
Static	*UndefinedIdentifier*	UndefinedIdentifier	1577	*Fig. 14*
Static	MissingReturn	MissingReturn	155	
Static	UndefinedIdentifier	MethodNotFound	40	
Static	UndefinedIdentifier	TypeNotFound	15	
Static	ArityMismatch	InvalidArgument	11	
Static	ReDeclaration	ReDeclaration	2	
Type	InvalidTypeConversion	TypeConversion	409	
Language	Miscellaneous	FormatException	77	
Language	LanguageSpecific	InvalidAssignment	13	
Language	Miscellaneous	ArgumentException	1	
Model	OutOfTokens	SyntaxError	319	
Model	*ExceptionInGeneratedCode*	NotImplementedException	5	
Model	*ExceptionInGeneratedCode*	InvalidBeat	1	

Table 17: Error Categories, Themes, and Labels for C#.

Category	Theme	Error	Count	Example
Runtime	AssertionFailed	AssertionFail	10051	
Runtime	InvalidDataStructureOperation	IndexOutOfRange	330	
Runtime	TimeoutOrInfiniteRecursion	Timeout	275	
Runtime	InvalidDataStructureOperation	InvalidRangeCreation	271	
Runtime	NullReference	UnwrapNil	149	
Runtime	InvalidDataStructureOperation	StringIndexOutOfBounds	99	
Runtime	DivisionByZero	DivisionByZeroInRemainder	24	
Runtime	InvalidDataStructureOperation	RemoveLastFromEmptyCollection	6	
Runtime	InvalidDataStructureOperation	ArrayIndexOutOfRange	3	
Runtime	InvalidDataStructureOperation	RemoveFirstFromEmptyCollection	1	
Runtime	InvalidDataStructureOperation	NegativeArrayIndex	1	
Static	*UndefinedIdentifier*	CanNotFindInScope	4259	
Static	*UndefinedIdentifier*	NonExistentMethod	2582	
Static	*UndefinedIdentifier*	InvalidSyntax	213	
Static	*UndefinedIdentifier*	CallingNonFunctionType	103	
Static	IncorrectAPIMethodCall	SubscriptStringWithInt	68	Fig. 21
Static	*UndefinedIdentifier*	LinkerError	55	
Static	IncorrectAPIMethodCall	StringsArentCharArrays	42	
Static	*UndefinedIdentifier*	UseBeforeDecl	17	
Static	IncorrectAPIMethodCall	StringIndices	12	
Static	ReDeclaration	RedeclarationOfVariable	11	
Type	InvalidTypeConversion	OtherLocation	556	
Type	InvalidTypeConversion	ReturnTypeError	349	
Type	InvalidTypeConversion	ArgumentTypeError	303	
Type	InvalidTypeConversion	NumericsTypeError	261	
Type	InvalidTypeConversion	CollectionAndInner	241	
Type	InvalidTypeConversion	UnknownTypeErrorInCall	200	
Type	InvalidTypeConversion	BinOpTypeError	182	
Type	InvalidTypeConversion	BranchTypeMismatch	125	
Type	InvalidTypeConversion	MiscTypeError	111	
Type	InvalidTypeConversion	UnwrappedNonOptional	67	
Type	InvalidTypeConversion	UseOfModWithFloat	63	
Type	InvalidTypeConversion	ClosureResultTypeError	11	
Type	InvalidTypeConversion	ShouldHaveUnwrappedOptional	10	Fig. 19
Type	InvalidTypeConversion	PatternTypeError	10	
Type	InvalidTypeConversion	AssignmentTypeError	10	
Type	InvalidTypeConversion	WeirdSubscriptTypeError	10	
Type	InvalidTypeConversion	SubscriptingTypeError	9	
Language	LanguageSpecific	UseOfDeprecatedIdentifiers	176	
Language	*LanguageSpecific*	*MissingArgumentLabel*	113	*Fig. 20*
Language	LanguageSpecific	ImmutableViolation	62	
Language	LanguageSpecific	ExtraArgument	38	
Language	LanguageSpecific	IncorrectArgumentLabel	19	
Language	LanguageSpecific	OverflowUnderflowTrap	9	
Language	LanguageSpecific	ExtraneousArgumentLabel	5	
Language	LanguageSpecific	NonExclusiveMutation	3	
Model	OutOfTokens	RanOutOfTokens	95	
Model	OutOfTokens	CompilerErrorCutoff	9	
Model	OutOfTokens	MissingReturn	4	

Table 18: Error Categories, Themes, and Labels for Swift.

Category	Theme	Error	Count	Example
Runtime	AssertionFailed	assertionError	10409	
Runtime	TimeoutOrInfiniteRecursion	timeout	1044	
Runtime	InvalidDataStructureOperation	stringIndexOutOfRange	448	
Runtime	DivisionByZero	divisionBy0	376	
Runtime	InvalidDataStructureOperation	letDuplicateIdentifier	2	
Static	*UndefinedIdentifier*	unboundIdentifier	5814	*Fig. 15*
Static	ArityMismatch	arityMismatch	570	
Static	ReDeclaration	listIndexOutOfRange	281	
Static	*UndefinedIdentifier*	applicationOnNotAFunction	184	
Static	IncorrectAPIMethodCall	badBuiltInFnAppCall	69	
Static	*UndefinedIdentifier*	useBeforeInit	12	
Type	InvalidTypeConversion	contractViolation	4887	
Type	InvalidTypeConversion	hashmapNoValueForKey	7	
Language	*DoesNotKnowSyntax*	missingOrUnmatchedBraces	2004	
Language	LanguageSpecific	unknownEscapeSequence	216	
Language	*DoesNotKnowSyntax*	badSyntax	127	
Language	*DoesNotKnowSyntax*	missingElse	63	
Language	*DoesNotKnowSyntax*	ifExtraArgs	48	
Language	Miscellaneous	badMatch	24	
Language	*DoesNotKnowSyntax*	unclassified	6	
Language	LanguageSpecific	badCharConst	4	
Language	*DoesNotKnowSyntax*	elseAsExpression	2	
Model	OutOfTokens	incompleteGeneration	2092	
Model	*GenerateAnotherLang*	generatingWebpageOrMarkdown	608	Fig. 18
Model	GenerateAnotherLang	defaultValueExpMissing	57	
Model	ExceptionInGeneratedCode	errorByGeneratedProgram	34	

Table 19: Error Categories, Themes, and Labels for Racket.

D.3 Error Examples Related to Section 5.4

```
class Problem {
    // Translated docstring
    public static string Intersection(Tuple<long, long> interval1, Tuple<long, long> interval2) {
        long start = Math.Max(interval1.Item1, interval2.Item1);
        long end = Math.Min(interval1.Item2, interval2.Item2);
        if (start > end) {
            return "NO";
        }
        long length = end - start + 1;
        return IsPrime(length) ? "YES" : "NO";
    }
```

Figure 14: *C#*: Exhibits a local context error as it relies on a non-existent `IsPrime` method (Prompt #127, Theme: "UndefinedIdentifier", Code: "UndefinedIdentifier")

```
#lang racket

;; You are given a non-empty list of positive integers. Return the
greatest integer that is greater than
;; zero, and has a frequency greater than or equal to the value of the
integer itself.
;; The frequency of an integer is the number of times it appears in the list.
;; If no such a value exist, return -1.
;; Examples:
;; >>> (search (list 4 1 2 2 3 1))
;; 2
;; >>> (search (list 1 2 2 3 3 3 4 4 4))
;; 3
;; >>> (search (list 5 5 4 4 4))
;; -1
(define (search lst)
  (let ([freq (make-hash)])
    (for ([i lst])
      (hash-set! freq i (+ 1 (hash-ref freq i 0))))
    (let ([max-freq (apply max (hash-values freq))])
      (for ([i lst])
        (when (and (> i 0) (= max-freq (hash-ref freq i)))
          (return i))))
    -1))
```

Figure 15: *Racket*: A completion which uses the keyword `return`. `return` is not a keyword in Racket, but is a common keyword in Python. (Prompt #69, Theme: "UndefinedIdentifier", Error: "unboundIdentifier")

```
def same_chars(s0: str, s1: str) -> bool:
    """
    docstring
    """
    return Counter(s0) == Counter(s1)
```

Figure 16: *Python*: This code expects a call to `import collections` to facilitate the use of `Counter` dictionary subclass (Prompt # 54, Theme: "UndefinedIdentifier", Error: "NameError")

```
def find_closest_elements(numbers: List[float]) -> Tuple[float, float]:
    """ From a supplied list of numbers (of length at least two) select and return two
    that are the closest to each other and return them in
    order (smaller number, larger number).
    >>> find_closest_elements([1.0, 2.0, 3.0, 4.0, 5.0, 2.2])
    (2.0, 2.2)
    >>> find_closest_elements([1.0, 2.0, 3.0, 4.0, 5.0, 2.0])
    (2.0, 2.0)
    """
    # YOUR CODE HERE
    raise NotImplementedError()
```

Figure 17: *Python*: The full completion for this program contains only a call to `raise`, so this code will always produce an error. (Prompt #20, Theme: "ExceptionInGeneratedCode", Error: "NotImplementedError")

```
#lang racket

;; You are given a string s.
;; if s[i] is a letter, reverse its case from lower to upper or vise versa,
;; otherwise keep it as it is.
;; If the string contains no letters, reverse the string.
;; The function should return the resulted string.
;; Examples
;; >>> (solve "1234")
;; "4321"
;; >>> (solve "ab")
;; "AB"
;; >>> (solve "#a@C")
;; "#A@c"
(define (solve s)
  (if (string-contains s #\a)
      (string-map (lambda (c) (if (char-alphabetic? c)
                                   (char-upcase c)
                                   (char-downcase c)))
                  s)
      (string-reverse s)))
```

```
### [Reverse the order of words in a string]
(https://www.codewars.com/kata/reverse-the-order-of-words-in-a-string)

```racket
#lang racket
```

Figure 18: *Racket*: This snippet is an example of Markdown being generated instead of Racket (it is word wrapped for presentation here). (Prompt #161, Theme: “GenerateAnotherLanguage”, Error: “GeneratingWebpageOrMarkdown”)

```
/// This function takes an array l and returns an array l' such that
/// l' is identical to l in the indicies that are not divisible by three,
while its values at the indicies that are divisible by three are equal
/// to the values of the corresponding indicies of l, but sorted.
/// >>> sort_third(l: [1, 2, 3])
/// [1, 2, 3]
/// >>> sort_third(l: [5, 6, 3, 4, 8, 9, 2])
/// [2, 6, 3, 4, 8, 9, 5]
func sort_third(l: [Int]) -> [Int] {
    var l = l
    for i in stride(from: 2, to: l.count, by: 3) {
        l[i] = l[i].min(l[i - 1])
        l[i - 1] = l[i].max(l[i - 1])
    }
    return l
}
```

Figure 19: *Swift*: This code produces the type error `error: value of optional type 'Int?' must be unwrapped to a value of type 'Int'` (Prompt # 33, Theme: “InvalidTypeConversion”, Error:“ShouldHaveUnwrappedOptional”)
```

```swift
/// Given a positive integer n, return the count of the numbers of n-digit
/// positive integers that start or end with 1.
func starts_one_ends(n: Int) -> Int {
    if n == 1 {
        return 2
    }
    if n == 2 {
        return 4
    }
    return starts_one_ends(n - 1) + starts_one_ends(n - 2)
}
```

Figure 20: *Swift*: A correct recursive call to `starts_one_ends` in Swift would be `starts_one_ends(n: n - 1)`. (Prompt #83, Theme: "LanguageSpecific", Error: "MissingArgumentLabel")

D.4 Additional Error Examples from our Evaluation

```swift
/// Translated docstring
func match_parens(lst: [String]) -> String {
    var lst = lst
    var str = ""
    var count = 0
    for i in 0..<lst.count {
        str += lst[i]
    }
    for i in 0..<str.count {
        if str[i] == "(" {
            count += 1
        } else {
            count -= 1
        }
        if count < 0 {
            return "No"
        }
    }
    if count == 0 {
        return "Yes"
    } else {
        return "No"
    }
}
```

Figure 21: *Swift*: This is an example of incorrect string indexing in Swift - you cannot call `str[i]` where `i` is an `Integer` (Prompt #119, Theme: "IncorrectAPIMethodCall", Error: "SubscriptStringWithInt")

```python
def car_race_collision(n: int) -> int:
    """
    Imagine a road that's a perfectly straight infinitely long line.
    n cars are driving left to right;  simultaneously, a different set of n cars
    are driving right to left.   The two sets of cars start out being very far from
    each other.  All cars move in the same speed.  Two cars are said to collide
    when a car that's moving left to right hits a car that's moving right to left.
    However, the cars are infinitely sturdy and strong; as a result, they continue moving
    in their trajectory as if they did not collide.

    This function outputs the number of such collisions.
    """
    # The idea is to sort the cars by their starting positions.
    # Then, we can simulate the race by iterating through the cars
    # from left to right, and counting the number of cars that are
    # moving right to left.
    #
    # The time complexity is O(n log n) because of the sorting.
    # The space complexity is O(n) because of the sorting.
    #
    # The time complexity can be improved to O(n) by using a
    # counting sort.  The space complexity can be improved to
    # O(1) by using a bucket sort.
    left_to_right = [0] * n
    right_to_left = [0] * n
    for i in range(n):
        left_to_right[i] = int(input())
        right_to_left[i] = int(input())
    left_to_right.sort()
    right_to_left.sort()
    collisions = 0
    left_to_right_index = 0
    right_to_left_index = 0
    while left_to_right_index < n and right_to_left_index < n:
        if left_to_right[left_to_right_index] < right_to_left[right_to_left_index]:
            collisions += right_to_left_index
            left_to_right_index += 1
        else:
            right_to_left_index
```

Figure 22: *Python*: This code produces an `EOFError` due to the calls to `input()` inside the `for` loop. The canonical example does not use `input()`. Our evaluation procedures do not handle user input, hence the error. (Prompt #41, Theme: “LanguageSpecific”, Error: “EOFError”)

```
;; You are given a list of integers.
;; You need to find the largest prime value and return the sum of its digits.
;; Examples:
;; >>> (skjkasdkd (list 0 3 2 1 3 5 7 4 5 5 5 2 181 32 4 32 3 2 32 324 4 3))
;; 10
;; >>> (skjkasdkd (list 1 0 1 8 2 4597 2 1 3 40 1 2 1 2 4 2 5 1))
;; 25
;; >>> (skjkasdkd (list 1 3 1 32 5107 34 83278 109 163 23 2323 32 30 1 9 3))
;; 13
;; >>> (skjkasdkd (list 0 724 32 71 99 32 6 0 5 91 83 0 5 6))
;; 11
;; >>> (skjkasdkd (list 0 81 12 3 1 21))
;; 3
;; >>> (skjkasdkd (list 0 8 1 2 1 7))
;; 7
(define (skjkasdkd lst)
  (define (is-prime? n)
    (define (iter i)
      (cond ((= i n) #t)
            ((= (modulo n i) 0) #f)
            (else (iter (+ i 1)))))
    (iter 2))
  (define (sum-digits n)
    (define (iter n acc)
      (cond ((= n 0) acc)
            (else (iter (quotient n 10) (+ acc (remainder n 10))))))
    (iter n 0))
  (define (largest-prime lst)
    (cond ((null? lst) 0)
          ((is-prime? (car lst)) (max (car lst) (largest-prime (cdr lst))))
          (else (largest-prime (cdr lst)))))
 (sum-digits (largest-prime lst)))
```

```
## Задача 2

Да се дефинира функция `(skjkasdkd lst)`, която приема списък от цели числа и връща списък от цели числа, които са прости и по-големи от най-голямото просто число в дадения списък.

Примери:


```racket
>>> (skjkasdkd (list 0 3 2 1 3 5 7 4 5 5 5 2 181 32 4 32 3 2 32 324 4 3))
'(181 191 193 197 199 211 223 227 229 233 239 241 251 257 263
269 271 277 281 283 293 307 311 313 317 331 337 347 349 353
359 367 373 379 383 389 397 401 409 419 421 431 433
```

Figure 23: *Racket*: Codex-Davinci can produce comments/code in languages other than English. The Bulgarian text above translates as the following via Google Translate: "## Task 2, Define a function '(skjkasdkd lst)' that accepts a list of integers and returns a list of integers that are prime and greater than the largest prime in the given list., Examples:" (Prompt #119, Theme: "GenerateAnotherLanguage", Error: "GeneratingWebpageOrMarkdown")

D.5 Additional Error Examples

This section contains error examples which we found through manual inspection of Codex-Davinci output outside of our structured analysis.

```
function sum_squares(lst){
   return sum(map(lambda n: n ** 2, filter(lambda n: n % 3 == 0, lst)))
}
```

Figure 24: This JavaScript completion is generated for our full translation at $T = 0.8$ for prompt #142. The complete function body is executable, functional-style Python.
```

```
local function will_it_fly(q, w)
  return is_balanced(q) and sum(q) <= w
end
```

Figure 25: This Lua completion is generated by our full translation at $T = 0.2$ for prompt #72. `is_balanced` is not defined in the local context and is likely an expected helper function. The call to `sum` is Python-like, as `sum` is a Python built-in method.