Parsel 🐍: A (De-)compositional Framework for Algorithmic Reasoning with Language Models

Eric Zelikman [1] **Qian Huang** [1] **Gabriel Poesia** [1] **Noah D. Goodman** [1] **Nick Haber** [1]

Abstract

Despite recent success in large language model (LLM) reasoning, LLMs struggle with hierarchical multi-step reasoning tasks like generating complex programs. For these tasks, humans often start with a high-level algorithmic design and implement each part gradually. We introduce Parsel[2], a framework enabling automatic implementation and validation of complex algorithms with code LLMs, taking hierarchical function descriptions in natural language as input. We show that Parsel can be used across domains requiring hierarchical reasoning, including program synthesis, robotic planning, and theorem proving. We show that LLMs generating Parsel solve more competition-level problems in the APPS dataset, resulting in pass rates that are over 75% higher than prior results from directly sampling AlphaCode and Codex, while often using a smaller sample budget. We also find that LLM-generated robotic plans using Parsel as an intermediate language are more than twice as likely to be considered accurate than directly generated plans. Lastly, we explore how Parsel addresses LLM limitations and discuss how Parsel may be useful for human programmers.

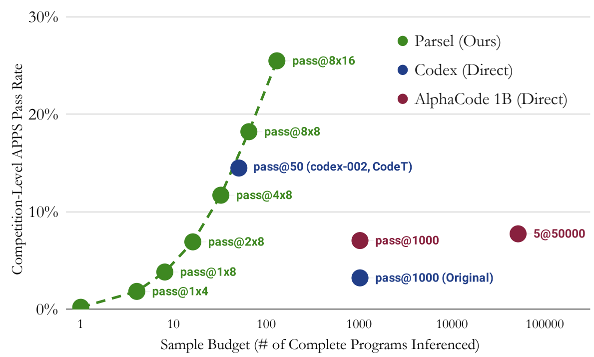

Figure 1. **Competition-level problem pass rate**. Comparison of Parsel's pass rate on competition-level APPS problems (Hendrycks et al., 2021) against direct code generation on multiple Codex versions (Chen et al., 2021; 2022) and AlphaCode (Li et al., 2022).

1. Introduction

To a language model for code (as for a human), each new token is a new chance to break the program. Chen et al. (2021) highlighted this issue in a toy experiment: when asked to generate a program with a series of simple string transformations, the performance of their code LLM, Codex, drops dramatically with the number of steps. As they pointed out, a human who can implement a few building blocks should be able to compose these blocks with arbitrary length.

Unlike other token generators, human programmers have (mostly) learned to break down complex tasks into manageable parts that work alone (modular) and work together (compositional). And, when human-generated tokens break functions, the functions can ideally be rewritten independently of the rest of the program. In contrast, naively, we expect code LLMs to generate token sequences that are correct in their entirety. Motivated by this, we study how to leverage LLMs to decompose problems and assemble their compositional solutions.

We propose Parsel: the Parsel compiler takes in a specification with 1) **natural language function descriptions** and 2) **constraints** that specify expected behavior for the function implementations. For each function, a code LLM is prompted to generate implementations from that function's description and the descriptions and function signatures of its direct dependencies. The compiler then searches over combinations of implementations until it finds one that satisfies all the constraints (e.g. unit tests).

Thus, generating and implementing Parsel mirrors a pattern in human reasoning – decomposing an abstract plan until it can be solved automatically (Simon & Newell, 1971) – and this compositional structure also benefits language models. We show that LLMs can generate Parsel with only a few examples, and their solutions outperform prior work on competition-level problems in the APPS dataset (Hendrycks et al., 2021), including AlphaCode (Li et al., 2022) and both versions of Codex (Chen et al., 2021; 2022). LLMs also propose robotic plans that are more accurate than in prior work (Huang et al., 2022). Indeed, we formulate Parsel as a general-purpose framework for algorithmic reasoning and show that it can even be used for tasks like theorem proving.

[1]Stanford University. Correspondence to: Eric Zelikman <ezelikman@cs.stanford.edu>.

[2]Named for Parseltongue, the language used to speak to serpents like pythons in Harry Potter.

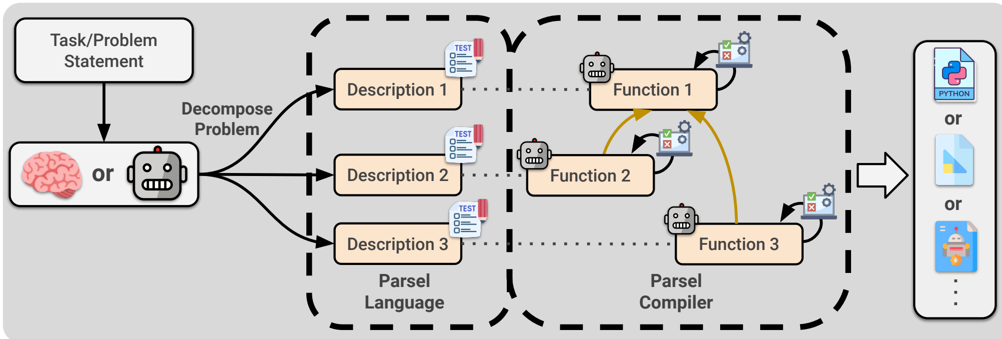

Figure 2. **Parsel overview**. A human or LLM writes a task decomposition in the Parsel language, which is split into its strongly-connected components (SCC), and then the Parsel compiler uses a code LLM and a constraint solver to implement and compose each SCC. When all functions have constraints, and there is no recursion, each function is its own SCC. We provide a more detailed figure in Appendix D.

2. Specifying Programs in Parsel

To specify a high-level algorithmic design formally, we develop a simple language underlying Parsel. We design it considering programmers, code LLMs, and students, as discussed in Appendix A and inspired by many works, noted in Section 5. In Parsel, each line contains a description, a constraint, or a reference to a description. They all obey scoping rules and have some nuances per target language.

Descriptions. A function description is represented as a function name followed by comma-separated input arguments in parentheses, and optionally an arrow followed by what the function returns[3], then a colon and text describing the function to be implemented. For example, as part of Conway's Game of Life, one might write

```
count_living_neighbors(grid, i, j): count the number of living neighbors of the cell at the index (i, j)
```

A function generated from a description can call either the functions defined directly below the description in the function graph (indicated with indentation) or references directly below the description[4], both shown in Fig. iii.

Constraints. A constraint is represented as a function's input values comma-separated, optionally followed by an arrow and the expected output of the function. Constraints are provided at the same indentation as the preceding description. For example, after the definition of count_living_neighbors one can write

```
[[1, 0], [0, 1]], 0, 0 -> 1
[[1, 0], [0, 1]], 0, 1 -> 2
```

This indicates that count_living_neighbors should return 1 when called with the arguments `[[1, 0], [0, 1]], 0, 0` and 2 when called with `[[1, 0], [0, 1]], 0, 1`. Notably, to apply complex constraints on functions, one can describe and constrain higher-order functions. For example:

```
type_fn_output(fn, args): returns the type of the output of a function called with args
count_living_neighbors, ([[1, 0], [0, 1]], 0, 0) -> int
```

This indicates that the function count_living_neighbors should return an integer when called with the input arguments `[[1, 0], [0, 1]], 0, 0`.

What it means to satisfy constraints to validate a program varies from language to language: in Python, one can check that a program passes certain assert statements by evaluating them; however, in a theorem-proving language like Lean, where the ability to run a program (without skipping steps by using `sorry` or `oops` lines) shows that a proof holds, one would instead represent the formal proof statement as the specified constraint (that is, that you are actually proving what you set out to prove). For languages where correctness can be checked without any unit tests, their functions can be treated as also having implicit constraints.

References. A reference is simply the name of a function defined in the current scope (see the next paragraph for details) within the function graph. A reference allows and encourages (via prompt) the parent function to use the referenced function. This allows for recursive function definitions and functions called by multiple functions. For example, one can define an (overly verbose) version of the Collatz conjecture (a well-known recursive open question in mathematics) as shown in Figure iii, where the final line is a reference. We visualize the corresponding call graph and its strongly connected components (SCC) in Figure 4. In the Collatz functions, `base_case` is implemented first as the `collatz_recursion` SCC depends on it.

Scope. Scope in Parsel is defined by indentation. The scope S of a function f includes the set of functions that can be used as a reference for a given function – that is, all functions where the indentations between the current function to the referenced function are strictly decreasing.

[3] Note that in Parsel, for Python one can also indicate in the description that a function should yield a value or is a generator.

[4] A nuance here is the optional ability for undefined/out-of-scope functions which are generated by the code LLM to also be implemented automatically.

```
task_plan(): return a list of strings that represents an action plan to put a mug on the stall and bread on the desk.
-> "executable"
  put_object_on(object, place): return a list of strings that represents an action plan to put an object in a place.
  "mug", "stall" -> "executable"
```

⇒

```python
# return a list of strings that represents an action plan to put a mug on the stall and bread on the desk.
def task_plan():
   return put_object_on("mug", "stall") + put_object_on("bread", "desk")
# return a list of strings that represents an action plan to put an object in a place.
def put_object_on(object, place):
   return [
      'find␣' + object,
      'grab␣' + object,
      'walk␣to␣' + place,
      'put␣' + object + '␣on␣' + place
   ]
from execute_virtual_home import test_script; assert test_script(task_plan()) == "executable"
from execute_virtual_home import test_script; assert test_script(put_object_on("mug", "stall")) == "executable"
```

Figure i. Parsel to VirtualHome (robotic planning)

```
and_commute(p q: Prop): the and operator is commutative
show (p ∧ q → q ∧ p) ∧ (q ∧ p → p ∧ q)
 p_q_implies_q_p(p q: Prop): if p ∧ q, then q ∧ p
 q_p_implies_p_q(p q: Prop): if q ∧ p, then p ∧ q
```

⇒

```lean
-- if p ∧ q, then q ∧ p
lemma p_q_implies_q_p(p q: Prop):

   p ∧ q → q ∧ p :=
begin
   intro h,
   cases h with hp hq,
   split,
      exact hq,
      exact hp,
end
-- Description: if p ∨ q, then q ∨ p
-- if q ∧ p, then p ∧ q
lemma q_p_implies_p_q(p q: Prop):

   (q ∧ p) → (p ∧ q) :=
begin
 intro h,
 split,
  exact h.right,
  exact h.left,
end
/-
 Theorem:
  If q ∧ p, then p ∧ q
-/
-- the and operator is commutative
lemma and_commute(p q: Prop):
 (p ∧ q → q ∧ p) ∧ (q ∧ p → p ∧ q) :=

begin
 apply and.intro,
 { apply p_q_implies_q_p },
 { apply q_p_implies_p_q }
end
```

Figure ii. Parsel to Lean (theorem proving)

```
collatz_recursion(num, cur_list=list()): Calls base_case if 1, otherwise recursion_rule
19 -> [19, 58, 29, 88, 44, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1]
  base_case(num, cur_list): Returns the list with the number appended to it
  recursion_rule(num, cur_list): Add num to list, collatz with 3n + 1 if odd or n / 2 if even
    collatz_recursion
```

⇒

```python
# Calls base_case if 1, otherwise recursion_rule
def collatz_recursion(num, cur_list=list()):
  if num == 1:
     return base_case(num, cur_list)
  else:
     return recursion_rule(num, cur_list)
# Returns the list with the number appended to it
def base_case(num, cur_list):
  cur_list.append(num)
  return cur_list
# Add num to list, collatz with 3n + 1 if odd or n / 2 if even
def recursion_rule(num, cur_list):
   cur_list.append(num)
   if num % 2 == 0:
      return collatz_recursion(num / 2, cur_list)
   else:
      return collatz_recursion((3 * num) + 1, cur_list)
assert collatz_recursion(19) == [19, 58, 29, 88, 44, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1]
```

Figure iii. Parsel to Python

Figure 3. Examples using Parsel to compile programs for VirtualHome (Puig et al., 2018), Lean, and Python. Note the columns on the generated examples are only there to allow them to fit compactly – each program implementation is one contiguous solution. Colors for Parsel code on the left are used to indicate constraints and references. All other Parsel lines shown are definitions.

Variations due to target language requirements. Certain aspects of the implementation are still target-language specific. As discussed above, the meaning and representation of a constraint may vary by language. Moreover, every language has a different evaluation function: executing Python is different than compiling and running C++ code, which is different than checking a proof with Lean. Further, every language will likely require a different prompt for the language model. Thus, we detail these particularities in language-specific configuration files.

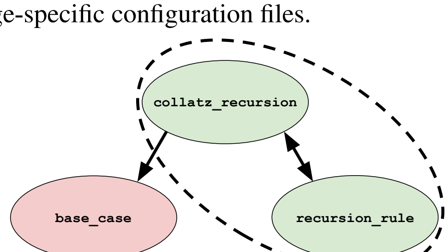

Figure 4. `collatz_recursion` call graph. There is a strongly connected component formed by `collatz_recursion` and `recursion_rule`.

3. Implementing Programs in Parsel

There are several steps to compiling a Parsel program, which we explicate in this section, and provide a high-level pseudocode capturing the details in Fig. 5 and visualize in Fig.2.

3.1. Constrained Implementation

Every approach here relies on the same principle: by generating functions with a code LLM and testing them, we can identify an implementation satisfying all of the constraints (e.g. in Python, we use assert statements as constraints). An implementation satisfying all constraints is treated as "fixed" and leveraged in functions calling it. Note this creates a responsibility for the user – if constraints are specified and Parsel finds an implementation passing the constraints, we assume the implementation is correct! To lighten this effect, we support backtracking: if we cannot find a correct implementation for a function (or functions evaluated together), we can automatically re-implement its descendants.

3.2. Generating Function Implementations

A function is implemented by first aggregating the descriptions and function signatures of its (potentially zero) children as part of a language model prompt (for Python, we generate a prompt as if the child functions are imported and use their descriptions as comments). Crucially, this allows us to easily interchange child implementations. We then query the code LLM using the description text as a docstring and the description's function name and arguments for the signature, with prompts in Appendix G.

```
parsel(program, target_language): compile a program from a string specifying a Parsel program.
 parse_program(program): parse the Parsel program string to a call graph
  create_root_node(): create a root node as the current function node, without any constraints
  parse_line(line, current_node, current_indent) -> function_graph: for each step up in indentation, set the current node to its
    ↪ parents. then, parse the definition, reference, or constraint.
   parse_definition(line): create a new function node, make it a child of the current node's parent, then assign it as current node.
   parse_reference(line): add reference as a child of current node if reference is an ancestor or a direct child of an ancestor
   parse_constraint(line): add the constraint to the current node's constraints.
 get_dependency_graph(function_graph) -> dependency_graph: taking the function graph, create a copy where all nodes without asserts
   ↪ also depend on their parents unless the target language implicitly tests all functions.
 identify_strongly_connected_components(dependency_graph): return SCCs of the dependency graph and the edges between the SCCs.
 compile_scc(scc, scc_graph): find an implementation string solving a given SCC, starting with SCC dependencies, then generating
   ↪ possible implementations of SCC functions, then finding an implementation combination satisfying the functions' constraints
  compile_children(scc, scc_graph): compile any SCCs this SCC depends on and add them to the implementation string.
   compile_scc
  generate_implementations(scc, n, children_implementation_str): for each function in the SCC, prompt the language model to generate n
    ↪  implementations of each function starting with the implementation string of the SCC's children.
  solve_constraints(scc, fn_implementations): taking the provided constraints of each function in the scc, evaluate a shuffled list of
    ↪  the direct product of implementations with the constraints until one passes all of them
   direct_product_implementations(fn_implementations): return the direct product of the list of lists of fn_implementations
   generate_constraints(fn_node): translate each of the constraints into an evaluation string idiomatic to the target language and add
     ↪  these to the list of combined implementations
   eval_str(scc, implementation_str): evaluate an implementation including constraints by running it in a target-language executor
   on_fail(scc, scc_graph): raise an error highlighting the scc which could not be compiled
```

Figure 5. Pseudocode in the style of Parsel describing how Parsel compiles programs. A detailed version including automatic decomposition and automatic infilling is in Figure A.10 of Appendix C. Constraints are left out for clarity – e.g. one could define a test function and validate the compilability (or lack thereof) of a set of reference Parsel programs.

3.3. Strongly Connected Components of Functions

Sequential case. Perhaps the most straightforward and simplest case of a Parsel program is one where all functions have constraints (e.g. unit tests), and no functions have recursive dependencies (e.g. Fig. A.9). We start by considering this case. This defines a clear topological function order so they can be implemented sequentially. In this situation, Parsel implements functions with post-order traversal from the root, generating implementations and finding one passing the specified constraints for each function. In other words, without any cycles in the call graph, we can start by implementing the leaf functions first, then their parents, etc. until the program is implemented. But, in practice, many programs have more complex structures and constraints.

Recursion. One can imagine a strict version of Parsel where we require all functions to have accompanying constraints. However, even with this requirement, as long as recursion is possible, it is sometimes necessary to implement functions jointly – with cyclic dependencies, none can be tested alone. Further, there are many contexts in which one may only have access to constraints for a subset of functions. In particular, to support automatic decomposition where a language model may automatically construct the Parsel subfunctions for a given function, it is necessary to allow for functions to be defined without constraints. This raises a key question: how?

One solution would be to consider all possible implementations of all functions defined in the Parsel program and then to iterate through combinations until a valid implementation is found. However, this is intractable for large programs. Specifically, the number of possible implementations of a program with k functions and n implementations per function is $O(n^k)$, exponential in the number of functions[5].

Instead, we propose a more efficient solution, inspired by Cocke & Kennedy (1977): for cases of recursion, we reference the function call graph and identify all of its SCCs (e.g. Fig 4). In other words, these are the sets of functions where their definitions depend on one another. For these cases, we apply the above-described technique of considering all possible sets of function implementations until one set satisfies all of the constraints. Notably, we make the simplifying assumption that any statefulness across functions does not interfere with dependent asserts.

We evaluate each element of a shuffled list corresponding to the function implementation sets arising from the direct product of all the sets of function implementations in the SCC. In other words, for possible implementations of functions f, g, h which form an SCC of a call graph, and $I(f)$ corresponds to the language-model-generated implementations of f, we evaluate uniformly random samples without replacement from $I(f) \times I(g) \times I(h)$.

Functions with no constraints. For functions with no constraints, we can conveniently use the same approach as above by reformulating the call graph as a "test dependency graph." That is, if a function has no constraints, it depends on its parents to enforce constraints on its implementations. This also allows us to automatically introduce new children via automatic decomposition, as it is nontrivial to generate constraints for those children (see Subsection 3.6).

[5] With the caveat that we remove exact duplicate implementations for a specific function. This could be improved by identifying semantically equivalent implementations that might differ syntactically, as typically done with e-graphs (Ellis et al., 2021)

3.4. Headers

We also support program headers, allowing global contexts, used when implementing all new functions within a program. This is indicated by a line containing an optional string of special characters (e.g. "#*#*#") separating the body and the text and is passed as a prefix to all prompts.

3.5. Automatic Generation

In order to generate Parsel programs from arbitrary natural language descriptions, we use a language model to generate Parsel code. We provide a few examples of Parsel programs and treat Parsel generation as a translation task. There are two primary approaches for generating Parsel: for problems where explicitly writing out a preliminary plan may be helpful, we first ask the model to generate a zero-shot plan in natural language and then prompt the model to translate the plan to Parsel. For problems where an intermediate plan is not necessary, we directly ask the model to translate the problem description to a Parsel solution. Although we would hope that with more widespread adoption, code LLMs will be able to automatically generate Parsel programs without any reference examples, as this is a newly proposed language, it is currently necessary.

3.6. Automatic Decomposition

As indicated by a rapidly growing number of papers (Brohan et al., 2022; Huang et al., 2022), the task of decomposing a task into steps in natural language is one that language models are surprisingly capable of. As explored in concurrent work (Stuhlmüller et al., 2022), using language models to automatically and recursively decompose difficult open-ended problems to arbitrary depth is a powerful tool. Thus, we treat the ability to automatically decompose a Parsel function as a key feature of Parsel. This is an optional flag that prompts a language model to generate Parsel code corresponding to any additional subfunctions necessary when Parsel fails to implement a function. These proposed subfunctions are then added as child nodes to the decomposed function node. However, an additional consequence is that Parsel can thus be used to recursively decompose tasks into steps, by repeatedly identifying descriptions that cannot be directly implemented and attempting to decompose them.

4. Experiments

We explore the ability of Parsel to generate programs for various tasks, including program synthesis, robotic task planning, and theorem proving. These three categories represent related but distinct kinds of algorithmic reasoning, with varying levels of abstractness and generalizability. By evaluating Parsel on this breadth of tasks, we hope to better understand its generality as a framework for algorithmic reasoning.

4.1. Python Code Generation

Solving competition-level problems. We evaluated Parsel on the competition-level subset of the APPS (Hendrycks et al., 2021) dataset as follows: first, we zero-shot prompt GPT3 (text-davinci-002) with the problem statement, requiring it to first propose a high-level solution and explain why it is correct, asking it to "think step by step to come up with a clever algorithm" (Kojima et al., 2022). We then prompt Codex (Chen et al., 2021) to translate the generated solution into Parsel code with a few examples of valid code. We then attempt to compile the Parsel code, evaluating implementations to measure their pass rate. We compare to directly sampling Codex (Chen et al., 2021; 2022) and AlphaCode (Li et al., 2022). We visualize the pass rate in Figure 1, finding that this pipeline improves over prior work from 14.5% to 25.5%. However, we note this prior result was not zero-shot, unlike our sketch generation and Parsel compilation.

We report "pass@$n \times k$", where n is the number of Parsel programs generated and k is the number of complete program implementations generated. Prior work has emphasized performance as a function of the sample/inference budget (Chen et al., 2021; Li et al., 2022), and we do the same. Note, however, that since we are able to "mix-and-match" the function implementations across complete implementations of a given Parsel program, each additional complete program implementation sampled multiplies the number of distinct programs available to us. In other words, the number of programs to test grows exponentially with k, while the inference cost is linear. The program generation prompts depend on the task (e.g. competitive coding, robotic task planning, etc.), and we include these, as well as more sampling/evaluation details, in Appendix J.

While an advantage of Parsel is that the number of potential programs under consideration grows exponentially with the sample budget, when few functions have constraints (as in this evaluation) or there are complex recursive dependencies, it becomes infeasible to exhaustively evaluate all possible programs. Thus, we also report the performance of Parsel as a function of the number of evaluations, as shown in Fig. 7. We find that the performance improves with the number of evaluations, but that many of the gains come early in terms of number of evaluations, with diminishing returns.

We also perform an ablation to show the impact of Parsel as an intermediate language: we provide the same high-level plans to Codex, but instead of translating them to Parsel, we translate them to Python directly. We generate 16 Python implementations per high-level plan on 100 randomly sampled problems and find that the performance drops to 6%. Sampling the longer solutions resulted in us hitting the rate limit more often, even when using the same number of total tokens per prompt, so scaling up to a 128-solution comparison on 1000 problems would likely take months.

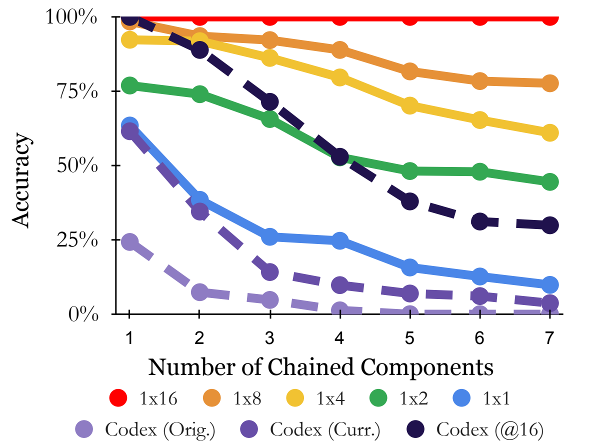

Figure 6. **Pass rate vs number of chained components.** Parsel performance is shown as 1xk where k is the number of complete programs sampled. Orig and Curr correspond to the original (Chen et al., 2021) and current Codex results, respectively. Codex (@16) corresponds to the Codex pass rate with 16 programs.

Comparison to expert. In our fully automated pipeline, an LLM generates a Parsel program, which the Parsel attempts to compile into a solution. We also evaluated how well an expert Parsel user would perform by directly writing Parsel and interacting with the Parsel compiler. As a case study, one of our authors, with significant prior experience in competitive programming was presented with 10 randomly-selected competition-level Codeforces problems from the APPS dataset. The participant successfully solved 5 of the problems within a 6-hour time frame. In comparison, when GPT-3 generated Parsel solutions to these problems, it had a success rate of 2 out of 10 problems with 8 attempts per problem. The participant's solutions and the associated problems are included in Appendix L. This experience suggests that, given a suitable decomposition, the Parsel compiler can effectively generate complete correct solutions to hard problems – thus, a major point for improvement in the fully automated pipeline lies in the first stage (Parsel generation).

Chained components. Chen et al. (2021) highlight in their discussion of limitations that language models fail to produce code that requires the simple composition of building blocks. We replicate this experiment with Parsel, using the same docstring and providing the same descriptions as function descriptions. In Fig. 6, we visualize how Parsel solves this. We find that even interchanging the parts of two complete programs (1x2) solves more problems than Codex with 16 programs as the number of components grows.

4.2. Robotic Planning in VirtualHome

We also perform a VirtualHome (Puig et al., 2018) study to demonstrate that Parsel can also be used for complex robotic planning. VirtualHome is a simulation environment consisting of households with various interactive objects and agents. There are a small set of permissible actions with a strict grammar – tasks like "paint ceiling" may require multiple levels of decomposition, e.g. finding and collecting specific painting supplies, finding and using a ladder, and using the painting supplies, each requiring further decomposition.

To test the effectiveness of Parsel in this domain, we investigate whether Parsel could generate programs to solve tasks in the VirtualHome environment, while using the environment to provide feedback on whether the plan is executable. Specifically, we use Parsel to generate a Python program that can generate action plans in natural language similar to ones used in Huang et al. (2022). In each specified constraint, the produced natural language action plan is translated to formal VirtualHome instructions with minimal regex matching and tested executability. If the instructions can be successfully executed, they are considered valid – however, one could describe object-relational constraints on the state of the world after instructions execution. We include an example of a Parsel program that successfully executed and decomposed a task in VirtualHome in Figure 3. Note we also used a header describing a valid action plan, shown in Figure A.43.

However, as pointed out by (Huang et al., 2022), while it is easy to check plan executability, it is harder to check correctness, because there are generally many valid ways of accomplishing realistic tasks. Thus, like Huang et al. (2022), in order to evaluate the accuracy of the executable plans, we perform a human study. Specifically, we ran two surveys on Prolific (Palan & Schitter, 2018) where we asked 20 participants to make five rankings each, for a total of 100 rankings per survey. In one survey, we asked participants to rank a set of solutions by how accurately each solution accomplishes the task while in another we asked about how understandable the solution was. Two versions of distinct Parsel solutions were shown (where multiple executable Parsel solutions were available), one which included the indented function names used to solve the tasks alongside their step-by-step solutions, and one which only included step-by-step output. We compared both Parsel solutions to an also-executable baseline solution, based on Huang et al. (2022). We include more details about the survey format in Appendix K.

Humans ranked the Parsel solutions as more accurate than the baseline. In each ranking, both the indented and non-indented Parsel solutions were consistently ranked higher than the baseline. In accuracy, standard Parsel solutions (as well as indented solutions) were identified as more accurate than the baseline solutions in 69% of comparisons, more than twice as often as the baseline. In clarity, the standard Parsel solution was 70% more likely to be preferred over the baseline while the indented Parsel solution was 50% more likely to be preferred compared to the baseline. There was no notable difference between indented and non-indented Parsel solutions in either accuracy or clarity.

4.3. Theorem Proving in Lean

With the same framework, we can generate proofs in formal theorem-proving languages such as Lean, as in Figure ii. We include the translated version in the appendix. Note a nuance of Lean and theorem-proving languages is that the ability to run Lean on proof with no errors/warnings indicates the proof is correct (but is not a guarantee that the proof statement matches our claim in language). Thus, each function in a Lean Parsel proof has an "implicit constraint." This makes it straightforward to identify which informal parts of a proof are most difficult to explicate. Generally, we believe Parsel can be a powerful tool for theorem proving.

Yet, we observed important challenges in this context, which we believe are avenues for future work and can be resolved. For example, in datasets such as MiniF2F (Zheng et al., 2022), many proofs require explicit calculations in intermediate steps. That is, many proofs are similar to "Find the minimum value of $\frac{9x^2 \sin^2 x + 4}{x \sin x}$ for $0 < x < \pi$. Show that it is 012." (from the informal MiniF2F introduced by Jiang et al. (2022)). We believe that a dataset of proof statements (in natural and formal language), requiring complex proofs that are more abstract and less dependent on explicit calculations would allow us to better measure progress towards solving difficult theorems – we leave this to future work.

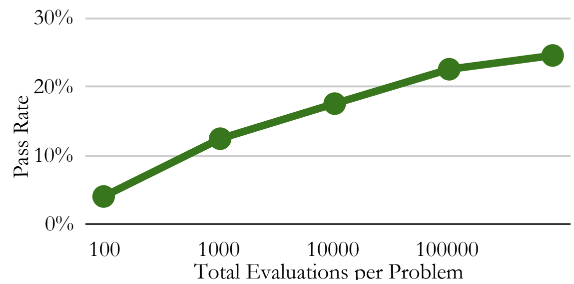

Figure 7. **Pass rate vs number of evaluations.** Parsel generates many programs to evaluate with a small inference budget, so we care about performance vs evaluations. We evaluate 8 LLM-generated Parsel programs with 16 implementations (8x16), on a random 200-problem competition-level subset of APPS.

5. Related Works

Step-by-step problem solving with LLMs. Many works show that step-by-step reasoning benefits LLM performance (Rajani et al., 2019; Shwartz et al., 2020; Nye et al., 2021; Wei et al., 2022; Marasović et al., 2021; Lampinen et al., 2022) and correspondingly, that this performance can be improved with guidance and tool use (Zhou et al., 2022a; Zelikman et al., 2022; Yao et al., 2022; Uesato et al., 2022; Dua et al., 2022). Acquaviva et al. (2022) encouragingly showed that humans, when asked to explain how to solve problems in the Abstract Reasoning Corpus (Chollet, 2019), tended to provide step-by-step hierarchical descriptions with many verification steps. Moreover, Wies et al. (2022) presents a theoretical argument showing problems that can be learned efficiently if decomposed but require exponentially many examples w.r.t. length if not decomposed.

Program synthesis. Program synthesis is the long-standing challenge of generating programs from high-level specifications (Gulwani et al., 2017), such as input-output examples (Bauer, 1979; Gulwani, 2016) and/or natural language descriptions (Raza et al., 2015; Yin & Neubig, 2017; Desai et al., 2016). Program synthesizers typically search the exponentially large space of programs. Consequently, synthesizing large, complex programs remains an open challenge. Recently, *library learning* has shown a way to make progress: even complex programs can be short in terms of the right high-level library. In turn, this library can be progressively induced from solutions to simpler synthesis problems. This idea is embodied in DreamCoder (Ellis et al., 2021; Bowers et al., 2023). Library learning requires a rich distribution of related tasks so that patterns emerge from solutions to simple problems. The idea is that patterns can be abstracted into useful library functions, enabling short solutions to more complex problems. Parsel similarly aims to synthesize complex programs by decomposing them into smaller functions. In Parsel, however, the user specifies the decomposition, so a family of related tasks is not required.

LLMs for formal environment multi-step planning. Also encouragingly, several existing works can be expressed in Parsel. For example, Huang et al. (2022) and Brohan et al. (2022) showed that language models can be used to generate step-by-step algorithms for robotic agents in language automatically. In both cases, the generated language corresponds directly to pre-implemented low-level robotic abilities. This could be expressed by providing a description of the task and constraints that evaluate that the high-level task was completed successfully. In addition, Jiang et al. (2022) proposed a framework to generate formal proofs in formal theorem-proving languages from informal proofs by first generating an intermediate natural language proof sketch. This could be expressed in Parsel by generating each sketch step as a function and then using formal verification for each lemma as the Parsel validation step.

Programming languages and frameworks incorporating language models. Other works have explored programming languages that incorporate language models. For example, Cheng et al. (2022) explored the introduction of a language-model-based evaluation function, which would allow `f('North America?', 'U.S.')` to automatically return 'yes' by referencing the knowledge of the language model, and showed that they could also generate programs using this tool with a language model and Beurer-Kellner et al. (2022) explored a related SQL-style LLM-querying language. In addition, Dohan et al. (2022) presents an

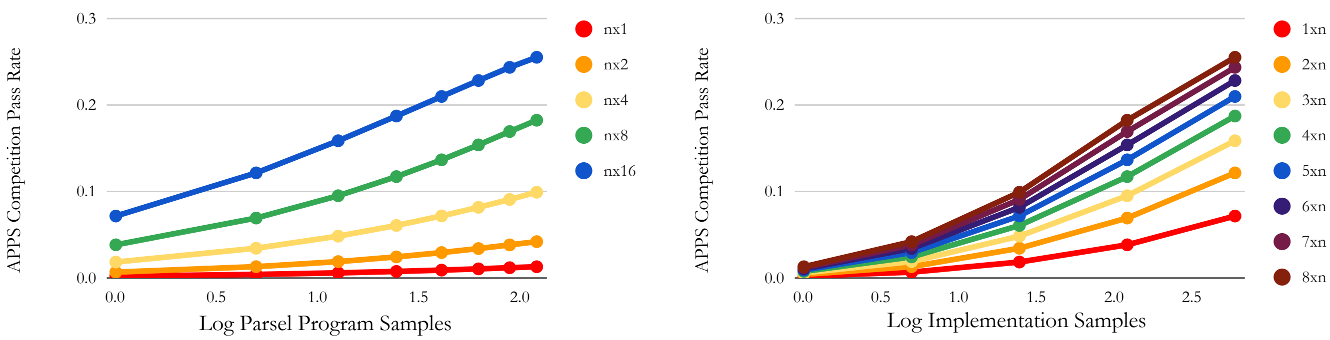

Figure 8. **Scaling performance.** The probability of passing an APPS competition-level problem increases quadratically with respect to the log of the number of Parsel programs sampled (left) and the number of implementations sampled (right).

inference-focused framework for language models more broadly for probabilistic graphical models composed of language model actions. Unlike LM Cascades (Dohan et al., 2022), we primarily focus on the constrained generation of programs, instead of leveraging language models as functions within a particular program.

Testing Code Language Model Outputs. Related works have explored the capacity of assert statements to constrain the generation space of LLMs for code on individual functions (Austin et al., 2021; Chen et al., 2021; Li et al., 2022). In particular, Merrill et al. (2021) proves essential constraints on what can be learned from assertions alone, and more crucially, what cannot.

6. Limitations

There are several limitations to the current implementation of Parsel. First, Parsel relies on a code LLM to generate implementations of individual functions, and the quality of these implementations can vary depending on the specific model used and the complexity of the function descriptions. In particular, Parsel may struggle to generate correct code for individual functions with complex behavior (i.e. functions that Codex cannot implement). However, this can be mitigated by decomposing the complex functions into simpler ones that can be implemented more easily.

The current implementation of Parsel may struggle to generate correct code when there are many functions with complex dependencies or without constraints. This is because the number of implementation combinations to consider grows exponentially with the size of the largest strongly connected components. As discussed, this can limit Parsel's performance on some programs. However, approaches like Chen et al. (2022) may be able to mitigate this.

Code LLMs, unfortunately, do not perform well on languages underrepresented in their training data – with few examples to learn from, LLMs may struggle to generate correct code in these languages (Athiwaratkun et al., 2022). However, some LLMs can learn new languages in context, allowing them to generate code in languages not in their training data (Athiwaratkun et al., 2022). These limitations can impact the quality and reliability of the code generated with Parsel. In addition, because code LLMs have never been trained on Parsel, this harms their ability to generate it. While we could wait for Parsel to gain widespread adoption, it should also be possible to translate many existing codebases to Parsel. We include a proof-of-concept backtranslation/decompilation study in Appendix H.

In addition, the best open-source code LLMs currently available e.g. PolyCoder (Xu et al., 2022) substantially underperform Codex, while Codex is competitive with other traditional LLMs on reasoning tasks (Liang et al., 2022). However, this dependence on closed models creates a vulnerability, as the providers of closed LLMs can change behavior (e.g. rate limits or model implementations) without warning.

However, despite these limitations, the current Parsel implementation has shown promising results in generating correct code for a variety of functions and languages. Many limitations will likely be ameliorated as code LLMs improve.

7. Discussion

Despite the limitations and the challenges of using a language model to generate a language that it has never seen before, the generated Parsel was able to implement robust algorithmic reasoning. There are a few natural questions that arise from this work. How robust is Parsel to the language used to specify the problem? When solving hard problems, is it better to increase the number of Parsel programs sampled or the number of program implementations sampled? We visualize Parsel's performance on the explored subset of APPS in Figure 8 to try to answer some of these questions.

In general, we found that the probability of passing increased quadratically ($R^2 \geq 0.99$ for all curves) with respect to the log of both the number of Parsel programs sampled and the number of implementations sampled. Obviously, since the pass rate is bounded between 0 and 1, this trend cannot continue indefinitely. However, given the rate limit associated to calls to the Codex API (officially, 40,000 tokens or 20 requests per minute, but in practice, we consistently ran into the rate limit with roughly 10% of that usage), we were not able to identify the inflection point.

Thus, we believe that Parsel is already a powerful tool that can compile larger and more complex programs. For reference, we include a working Lisp compiler in Appendix E. We started with the Lisp interpreter from Norvig (2010) as a starting point and then translated it to Parsel. We make sure to include no explicit references to lisp in the Parsel code and provide a header to tell Parsel to keep track of environments as dictionaries.

We hope that Parsel provides a broadly useful framework for several groups: for programmers, Parsel should provide a language for robust code generation without the need to evaluate the underlying code; for students, Parsel should allow the teaching of algorithmic reasoning with less emphasis on syntax and more emphasis on problem-solving, similarly to a mathematics curriculum; for language models, Parsel should facilitate hierarchical task decomposition.

8. Future Work

In the future, we also hope to integrate automatic unit test generation (Daka & Fraser, 2014; Chen et al., 2022). One method would be to identify edge cases and check whether the set of functions that successfully solve all existing tests disagree on any new tests. This could permit automatic decomposition without exponential growth in implementation combinations. Techniques like those proposed in Zhang et al. (2022), which would allow us to rerank a set of solutions, could also allow us to search the combinatorial space of solutions more quickly. Relatedly, for the robotic task planning, incorporating asserts at the execution level (e.g. checking whether the agent is close to the microwave, as in Singh et al. (2022)) is a promising research direction.

In addition, we plan to incorporate ways of varying the "confidence threshold" of the language model. Ensuring that the descriptions are straightforward and unambiguous is important for more critical programs and parts of programs. In addition, when teaching students simpler concepts, requiring them to decompose the task further may be useful.

We would like to integrate value functions to allow decomposition to be done more methodically where no verification is possible. Specifically, automatically decomposing all functions that have not yet been implemented in an SCC is suboptimal and could be improved with a model of expected improvement due to expansion, as done for proof expansion in Polu & Sutskever (2020). In addition, when decomposing functions, we would like to permit the model to reference already-defined functions (rather than to just define new ones). We might even use the code language model to determine which function to evaluate next. Further, we aim to support more general reward functions for function implementations where multiple may be valid but we rank implementations based on a desired feature. These "soft" constraints may also allow new Parsel uses, e.g. planning stories in natural language (Ye et al., 2022).

Finally, we hope it would be possible to use Parsel as a framework for bootstrapping increasingly complex program generation (e.g. Anthony et al. (2017); Zelikman et al. (2022); Odena et al. (2021)). That is, by 1) generating Parsel examples from a purely natural language specification and then reinforcing those which successfully compile, and 2) by reinforcing the model with each successfully compiled component, we would likely be able to iteratively improve performance with an arbitrarily large dataset of examples.

Another feature that would be valuable would be the ability to incorporate multiple base tools with different kinds of specialized models, inspired by Ibarz et al. (2022) and Dohan et al. (2022). That is, it would be valuable to allow a model to determine which target language to use, possibly combining them. For example, for large parts of the Tensorflow and PyTorch libraries, while their interfaces are written in Python, they depend heavily on large C++ codebases (Paszke et al., 2019; Abadi et al., 2015). Relatedly, Cobbe et al. (2021) showed that giving language models access to a calculator allowed them to solve more complex math word problems. This, combined with the observation that Parsel could also compile programs by generating language model prompts to be used as part of the program, may potentially allow the automatic generation of task-specific language model cascades (Dohan et al., 2022).

Another noteworthy addition would be the integration of Synchromesh (Poesia et al., 2022), ensuring that each new word or token generated by the model is actually possible within the grammar of the given formal language and does not violate other semantic constraints.

Ultimately, we hope that this specification for Parsel is a jumping-off point for a new way of thinking about programming and reasoning.

Acknowledgements

We would like to thank Jesse Mu, Tianyi Zhang, Rose E. Wang, Allen Nie, Ben Prystawski, Xindi Wu, Fan-Yun Sun, Isaac Kauvar, Xi Jia Zhou, John Thickstun, and Shikhar Murty for their helpful feedback, as well as Rishi Bommasani for highlighting highly relevant works. We also thank the Stanford HAI-Google collaboration for their Cloud Credit grant.

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL https://www.tensorflow.org/. Software available from tensorflow.org.

Acquaviva, S., Pu, Y., Kryven, M., Sechopoulos, T., Wong, C., Ecanow, G. E., Nye, M., Tessler, M. H., and Tenenbaum, J. B. Communicating natural programs to humans and machines. *NeurIPS*, 2022.

Anil, C., Wu, Y., Andreassen, A. J., Lewkowycz, A., Misra, V., Ramasesh, V. V., Slone, A., Gur-Ari, G., Dyer, E., and Neyshabur, B. Exploring length generalization in large language models. In *Advances in Neural Information Processing Systems*.

Anthony, T., Tian, Z., and Barber, D. Thinking fast and slow with deep learning and tree search. *Advances in Neural Information Processing Systems*, 30, 2017.

Athiwaratkun, B., Gouda, S. K., Wang, Z., Li, X., Tian, Y., Tan, M., Ahmad, W. U., Wang, S., Sun, Q., Shang, M., et al. Multi-lingual evaluation of code generation models. *arXiv preprint arXiv:2210.14868*, 2022.

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

Bauer, M. A. Programming by examples. *Artificial Intelligence*, 12(1):1–21, 1979.

Beurer-Kellner, L., Fischer, M., and Vechev, M. Prompting is programming: A query language for large language models. *arXiv preprint arXiv:2212.06094*, 2022.

Bowers, M., Olausson, T. X., Wong, C., Grand, G., Tenenbaum, J. B., Ellis, K., and Solar-Lezama, A. Top-down synthesis for library learning. *POPL*, 2023.

Brohan, A., Chebotar, Y., Finn, C., Hausman, K., Herzog, A., Ho, D., Ibarz, J., Irpan, A., Jang, E., Julian, R., et al. Do as i can, not as i say: Grounding language in robotic affordances. In *6th Annual Conference on Robot Learning*, 2022.

Chen, B., Zhang, F., Nguyen, A., Zan, D., Lin, Z., Lou, J.-G., and Chen, W. Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397*, 2022.

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. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Cheng, Z., Xie, T., Shi, P., Li, C., Nadkarni, R., Hu, Y., Xiong, C., Radev, D., Ostendorf, M., Zettlemoyer, L., et al. Binding language models in symbolic languages. *arXiv preprint arXiv:2210.02875*, 2022.

Chollet, F. On the measure of intelligence. *arXiv preprint arXiv:1911.01547*, 2019.

Cobbe, K., Kosaraju, V., Bavarian, M., Hilton, J., Nakano, R., Hesse, C., and Schulman, J. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Cocke, J. and Kennedy, K. An algorithm for reduction of operator strength. *Communications of the ACM*, 20(11): 850–856, 1977.

Daka, E. and Fraser, G. A survey on unit testing practices and problems. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pp. 201–211. IEEE, 2014.

Denny, P., Kumar, V., and Giacaman, N. Conversing with copilot: Exploring prompt engineering for solving cs1 problems using natural language. *SIGCSE*, 2022.

Desai, A., Gulwani, S., Hingorani, V., Jain, N., Karkare, A., Marron, M., and Roy, S. Program synthesis using natural language. In *Proceedings of the 38th International Conference on Software Engineering*, pp. 345–356, 2016.

Dohan, D., Xu, W., Lewkowycz, A., Austin, J., Bieber, D., Lopes, R. G., Wu, Y., Michalewski, H., Saurous, R. A., Sohl-Dickstein, J., et al. Language model cascades. *arXiv preprint arXiv:2207.10342*, 2022.

Dua, D., Gupta, S., Singh, S., and Gardner, M. Successive prompting for decomposing complex questions. *arXiv preprint arXiv:2212.04092*, 2022.

Ellis, K., Wong, C., Nye, M., Sablé-Meyer, M., Morales, L., Hewitt, L., Cary, L., Solar-Lezama, A., and Tenenbaum, J. B. Dreamcoder: Bootstrapping inductive program synthesis with wake-sleep library learning. In *Proceedings of the 42nd acm sigplan international conference on programming language design and implementation*, pp. 835–850, 2021.

Games, M. The fantastic combinations of john conway's new solitaire game "life" by martin gardner. *Scientific American*, 223:120–123, 1970.

Gulwani, S. Programming by examples. *Dependable Software Systems Engineering*, 45(137):3–15, 2016.

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

Hendrycks, D., Basart, S., Kadavath, S., Mazeika, M., Arora, A., Guo, E., Burns, C., Puranik, S., He, H., Song, D., et al. Measuring coding challenge competence with apps. *Advances in Neural Information Processing Systems*, 2021.

Huang, W., Abbeel, P., Pathak, D., and Mordatch, I. Language models as zero-shot planners: Extracting actionable knowledge for embodied agents. *arXiv preprint arXiv:2201.07207*, 2022.

Ibarz, B., Kurin, V., Papamakarios, G., Nikiforou, K., Bennani, M., Csordás, R., Dudzik, A., Bošnjak, M., Vitvitskyi, A., Rubanova, Y., et al. A generalist neural algorithmic learner. *LoG*, 2022.

Jiang, A. Q., Welleck, S., Zhou, J. P., Li, W., Liu, J., Jamnik, M., Lacroix, T., Wu, Y., and Lample, G. Draft, sketch, and prove: Guiding formal theorem provers with informal proofs. *arXiv preprint arXiv:2210.12283*, 2022.

Kojima, T., Gu, S. S., Reid, M., Matsuo, Y., and Iwasawa, Y. Large language models are zero-shot reasoners. *arXiv preprint arXiv:2205.11916*, 2022.

Kulal, S., Pasupat, P., Chandra, K., Lee, M., Padon, O., Aiken, A., and Liang, P. S. Spoc: Search-based pseudocode to code. *Advances in Neural Information Processing Systems*, 32, 2019.

Lampinen, A. K., Dasgupta, I., Chan, S. C., Matthewson, K., Tessler, M. H., Creswell, A., McClelland, J. L., Wang, J. X., and Hill, F. Can language models learn from explanations in context? *arXiv preprint arXiv:2204.02329*, 2022.

Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Dal Lago, A., et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.

Liang, P., Bommasani, R., Lee, T., Tsipras, D., Soylu, D., Yasunaga, M., Zhang, Y., Narayanan, D., Wu, Y., Kumar, A., et al. Holistic evaluation of language models. *arXiv preprint arXiv:2211.09110*, 2022.

Marasović, A., Beltagy, I., Downey, D., and Peters, M. E. Few-shot self-rationalization with natural language prompts. *arXiv preprint arXiv:2111.08284*, 2021.

Merrill, W., Goldberg, Y., Schwartz, R., and Smith, N. A. Provable limitations of acquiring meaning from ungrounded form: What will future language models understand? *Transactions of the Association for Computational Linguistics*, 9:1047–1060, 2021.

Nijkamp, E., Pang, B., Hayashi, H., Tu, L., Wang, H., Zhou, Y., Savarese, S., and Xiong, C. A conversational paradigm for program synthesis. *arXiv preprint arXiv:2203.13474*, 2022.

Norvig, P. How to write a Lisp interpreter (in Python), 2010. URL http://norvig.com/lispy.html.

Nye, M., Andreassen, A. J., Gur-Ari, G., Michalewski, H., Austin, J., Bieber, D., Dohan, D., Lewkowycz, A., Bosma, M., Luan, D., et al. Show your work: Scratchpads for intermediate computation with language models. *arXiv preprint arXiv:2112.00114*, 2021.

Odena, A., Shi, K., Bieber, D., Singh, R., Sutton, C., and Dai, H. Bustle: Bottom-up program synthesis through learning-guided exploration. *ICLR*, 2021.

Palan, S. and Schitter, C. Prolific. ac—a subject pool for online experiments. *Journal of Behavioral and Experimental Finance*, 17:22–27, 2018.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

Perry, N., Srivastava, M., Kumar, D., and Boneh, D. Do users write more insecure code with ai assistants? *arXiv preprint arXiv:2211.03622*, 2022.

Poesia, G., Polozov, A., Le, V., Tiwari, A., Soares, G., Meek, C., and Gulwani, S. Synchromesh: Reliable code generation from pre-trained language models. In *International Conference on Learning Representations*, 2022.

Polu, S. and Sutskever, I. Generative language modeling for automated theorem proving. *arXiv preprint arXiv:2009.03393*, 2020.

Puig, X., Ra, K., Boben, M., Li, J., Wang, T., Fidler, S., and Torralba, A. Virtualhome: Simulating household activities via programs. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 8494–8502, 2018.

Rajani, N. F., McCann, B., Xiong, C., and Socher, R. Explain yourself! leveraging language models for commonsense reasoning. *ACL*, 2019.

Raza, M., Gulwani, S., and Milic-Frayling, N. Compositional program synthesis from natural language and examples. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

Sandoval, G., Pearce, H., Nys, T., Karri, R., Dolan-Gavitt, B., and Garg, S. Security implications of large language model code assistants: A user study. *arXiv preprint arXiv:2208.09727*, 2022.

Shwartz, V., West, P., Bras, R. L., Bhagavatula, C., and Choi, Y. Unsupervised commonsense question answering with self-talk. *EMNLP 2020*, 2020.

Simon, H. A. and Newell, A. Human problem solving: The state of the theory in 1970. *American psychologist*, 26(2): 145, 1971.

Singh, I., Blukis, V., Mousavian, A., Goyal, A., Xu, D., Tremblay, J., Fox, D., Thomason, J., and Garg, A. Progprompt: Generating situated robot task plans using large language models. *arXiv preprint arXiv:2209.11302*, 2022.

Stuhlmüller, A., Reppert, J., and Stebbing, L. Factored cognition primer. https://primer.ought.org, 2022.

Tabachnyk, M. and Nikolov, S. Ml-enhanced code completion improves developer productivity, 2022.

Uesato, J., Kushman, N., Kumar, R., Song, F., Siegel, N., Wang, L., Creswell, A., Irving, G., and Higgins, I. Solving math word problems with process-and outcome-based feedback. *arXiv preprint arXiv:2211.14275*, 2022.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Chi, E., Le, Q., and Zhou, D. Chain of thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903*, 2022.

Wies, N., Levine, Y., and Shashua, A. Sub-task decomposition enables learning in sequence to sequence tasks. *arXiv preprint arXiv:2204.02892*, 2022.

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

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., and Cao, Y. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

Ye, A., Cui, C., Shi, T., and Riedl, M. O. Neural story planning, 2022. URL https://arxiv.org/abs/2212.08718.

Yin, P. and Neubig, G. A syntactic neural model for general-purpose code generation. *ACL*, 2017.

Yin, P., Deng, B., Chen, E., Vasilescu, B., and Neubig, G. Learning to mine aligned code and natural language pairs from stack overflow. In *2018 IEEE/ACM 15th international conference on mining software repositories (MSR)*, pp. 476–486. IEEE, 2018.

Yin, P., Li, W.-D., Xiao, K., Rao, A., Wen, Y., Shi, K., Howland, J., Bailey, P., Catasta, M., Michalewski, H., Polozov, A., and Sutton, C. Natural language to code generation in interactive data science notebooks, 2022. URL https://arxiv.org/abs/2212.09248.

Zelikman, E., Wu, Y., Mu, J., and Goodman, N. STar: Bootstrapping reasoning with reasoning. In Oh, A. H., Agarwal, A., Belgrave, D., and Cho, K. (eds.), *Advances in Neural Information Processing Systems*, 2022. URL https://openreview.net/forum?id=_3ELRdg2sgI.

Zhang, T., Yu, T., Hashimoto, T. B., Lewis, M., Yih, W.-t., Fried, D., and Wang, S. I. Coder reviewer reranking for code generation. *arXiv preprint arXiv:2211.16490*, 2022.

Zheng, K., Han, J. M., and Polu, S. Minif2f: a cross-system benchmark for formal olympiad-level mathematics. *ICLR*, 2022.

Zhou, D., Schärli, N., Hou, L., Wei, J., Scales, N., Wang, X., Schuurmans, D., Bousquet, O., Le, Q., and Chi, E. Least-to-most prompting enables complex reasoning in large language models. *arXiv preprint arXiv:2205.10625*, 2022a.

Zhou, H., Nova, A., Larochelle, H., Courville, A., Neyshabur, B., and Sedghi, H. Teaching algorithmic reasoning via in-context learning. *arXiv preprint arXiv:2211.09066*, 2022b.

```
select_airport_cities(city_road_cost, city_airport_cost): given a matrix representing the cost of
  ↪ building a road between any two cities, and a list representing the cost of building an airport
  ↪ in a city (where any two cities with airports are connected), return a list of the cities that
  ↪ should have airports built in them to minimize the total cost of building roads and airports such
  ↪  that all cities are connected. The list should be sorted in ascending order.
[[0,3,3],[3,0,3],[3,3,0]],[0,0,0] -> [0,1,2]
[[0,3,3],[3,0,3],[3,3,0]],[10,10,10] -> []
[[0,10,3],[10,0,11],[3,11,0]],[1,4,5] -> [0,1]
   sky_city_cost(city_road_cost, city_airport_cost): given a list of lists representing the cost of
     ↪ building a road between any two cities, and a list representing the cost of building an
     ↪ airport in a city, return a new cost matrix with a new node corresponding to the sky.
   [[1,2,3],[1,2,3],[1,2,3]],[4,5,6] -> [[1,2,3,4],[1,2,3,5],[1,2,3,6],[4,5,6,0]]
   minimum_spanning_tree(cost_matrix): given a list of lists representing the cost of each edge,
     ↪ return an adjacency matrix corresponding to the minimum spanning true.
   [[0,1,3,4],[1,0,2,100],[3,2,0,5],[4,100,5,0]] -> [[0,1,0,1],[1,0,1,0],[0,1,0,0],[1,0,0,0]]
   final_node_connectors(adjacency_matrix): given a list of lists representing an adjacency matrix,
     ↪ return a list of the nodes connected to the final node. However, if only one node is connected
     ↪  to the final node, return an empty list.
   [[0,1,0,1],[1,0,1,0],[0,1,0,0],[1,0,0,0]] -> []
   [[0,1,0,1],[1,0,1,0],[0,1,0,1],[1,0,1,0]] -> [0,2]
```

Figure A.9. A potential programming assignment focused on problem-solving rather than implementation. The top-level function and asserts would be the assigned problem (which Codex (Chen et al., 2021) does not seem to be able to solve directly), while the other functions would be the student solution.

A. Implications

Parsel is a natural language compiler framework that bridges the gap between natural language and programming language by allowing programmers to write high-level algorithmic designs in natural language and automatically compiling them into valid code. This has potential benefits for programmers, students, and code language models.

A.1. For Programmers

A.1.1. Current Limitations

First, programming generation language models like Codex continue to be constrained primarily to individual functions, rarely exceeding a few dozen lines in practice (Chen et al., 2021; Tabachnyk & Nikolov, 2022). This is still a dramatic shift from foundational earlier works, which focused on the association between one line of natural language pseudocode with one line of code (Kulal et al., 2019) or a line of text to a StackOverflow snippet (Yin et al., 2018). Yet, these models perform worse the more unusual the desired functions are, and recent research suggests that people using these language models are more likely to introduce buggy code (Perry et al., 2022), although this is not yet conclusive (Sandoval et al., 2022).

A.1.2. Potential Benefits

On the other hand, results from Google and others indicate that professionals can write code more efficiently with large language models, and the benefits will likely only improve as they improve (Tabachnyk & Nikolov, 2022). Since Parsel requires constraints that ensure functions behave as expected, this should encourage bug-free programs and avoid the need for manually checking that specific underlying functions are correct. Furthermore, a function written in Parsel is likely to be more resilient to breaking changes in the target language, especially syntactic changes (e.g. Python2 to Python3). In addition, a natural extension would draw on work on automatic unit testing (Daka & Fraser, 2014) to suggest additional constraints where behavior is ambiguous between implementations of a function.

A.2. For Students

A.2.1. Current Limitations

In addition, these language models pose serious challenges for programming pedagogy – existing introductory programming classes rely extensively on teaching syntax and how to implement algorithms over how to solve problems with them. Free language model-based tools like Copilot can essentially solve many of these introductory assignments directly, function by function. Those which cannot be solved currently will be increasingly solved (Denny et al., 2022).

A.2.2. Potential Benefits

Many students currently introduced to programming struggle with learning syntax and debugging unclear compiler or interpreter errors. However, abstracting away these details with a natural-language coding language will likely make learning to code more accessible to students who are just beginning to code. In addition, stepping away from implementation-focused assignments will allow a focus on higher-level problem-solving assignments earlier. These will allow for assignments that are more like those in mathematics. For example, for a problem like Figure A.9, instead of choosing between requiring students to manually implement a problem-solving focused question like the top-level description of, or requiring teaching assistants to manually evaluate the reasoning for correctness, one could ask them to implement a solution in Parsel.

A.3. For Code Language Models

A.3.1. Current Limitations

Traditional programming languages result in some unique challenges for language models. For example, unlike natural languages, traditional programming languages are far less robust to slight variations in wording. In addition, traditional programming languages require many tokens for syntactic details and in some cases, may take many lines to express what can be expressed far more simply in language. For example, referring to a shortest-path algorithm or Conway's game of life takes far fewer tokens than actually implementing them. However, even with fairly nonstandard problems, LLMs have shown remarkable algorithmic generalization ability (Liang et al., 2022; Xu et al., 2022; Anil et al.; Zhou et al., 2022b). One alternative that has been explored is conversational code generation (Nijkamp et al., 2022; Yin et al., 2022). However, these approaches have primarily focused on highly imperative programming structures. Moreover, they still require having the full program in context and do not clearly generalize to complex hierarchical programs with many functions.

A.3.2. Potential Benefits

Parsel allows code language models to stay closer to natural language when generating code, which corresponds more closely to their primary source of training data. Moreover, it allows complex but standard methods to be described concisely, requiring fewer tokens to generate. One exciting additional benefit is the potential to generate solutions recursively: if the Parsel compiler is unable to find a solution for a set of functions, it should be possible to prompt the model to define new helper functions. In fact, we find that often the model attempts to reference undefined auxiliary functions when defining complex functions (e.g. "count_living_neighbors(grid, i, j)" in Conway's game of life), and as a result support an optional argument where the model can attempt to resolve NameErrors automatically by attempting to implement functions.

B. Optimizations

B.1. Caching

We cache responses from the language model with respect to the prompt and language model decoding parameters 1) to reduce the number of queries necessary and 2) to keep the programs generated mostly stable (i.e. a working function should continue working unless it or its children change). To this end, when the number of desired implementations increases for a pre-existing query with all other arguments fixed (temperature, number of decoding tokens, etc), we append the additional ones to those already generated.

B.2. Automatic Function Infilling

Sometimes, a function generated by a language model may call a function that is not yet implemented. In this case, we can (optionally) attempt to automatically generate and implement it based on its usage. The function is then incorporated into the call graph as a unit-test-less child of the function which calls it. To avoid infinite recursion and inefficient use of language model quota, we limit the number of times that this process can be applied to a function.

B.3. Multiprocessing

We use multiprocessing with a user-specified timeout to test many implementation sets in parallel to allow for many fast solutions to be tested alongside slower solutions[6].

[6]As anticipating the number of steps that a solution will take universally is a version of the halting problem and thus intractable.

C. Parsel Pseudocode

We include a longer-form Parsel pseudocode in the style of Parsel. Note this pseudocode does not include backtranslation.

```
parsel(program, target_language, allow_autofill=False, allow_autodecomp=False): compile a program from a string specifying a Parsel program.
 parse_program(program): parse the Parsel program string to a call graph
  create_root_node(): create a root node as the current function node, without any constraints
  parse_line(line, current_node, current_indent) -> function_graph: for each step up in indentation, set the current node to its parents. then, parse the definition, reference, or constraint.
   parse_definition(line): create a new function node, make it a child of the current node's parent, then assign it as current node.
    parse_line_to_fn(line) -> name, args, rets, description: extract the function name, arguments, optionally returned variables, and description of the form "name(args) -> rets: description" if return variables are present else "name(args): description".
    populate_fn_node(name, args, rets, description): populate the new node's name, arguments, description, and optionally a list of returned variables.
   parse_reference(line): add reference as a child of current node if reference is an ancestor or a direct child of an ancestor
   parse_constraint(line): add the constraint to the current node's constraints.
 get_dependency_graph(function_graph) -> dependency_graph: taking the function graph, create a copy where all nodes without constraints also depend on their parents unless the target language implicitly tests all functions.
 identify_strongly_connected_components(dependency_graph): return SCCs of the dependency graph and the edges between the SCCs.
 compile_scc(scc, scc_graph, allow_autofill, allow_autodecomp): accumulate a implementation string which solves the current function
  compile_children(scc, scc_graph, allow_autofill, allow_autodecomp): compile any SCCs this SCC depends on and add them to the implementation string.
   compile_scc
   direct_product_implementations(fn_implementations): return the direct product of the list of lists of fn_implementations
  generate_implementations(scc, n, children_implementation_str): for each function in the SCC, generate n implementations of each function starting with the implementation string of the SCC's children.
   fn_implementation
  fn_implementation(fn_node, n): prompt the language model to generate n implementations of a function
   generate_prompt(fn_node): first prepend a string with all descriptions, names, arguments, and returns of fn_node's direct children, in a style idiomatic for the target language. then, add fn_node's description and function signature.
  solve_constraints(scc, fn_implementations, n, allow_autofill, allow_autodecomp): taking the provided constraints of each function in the scc, evaluate a shuffled list of the direct product of implementations with the constraints until one passes all of them
   generate_constraints(fn_node): translate each of the constraints into an evaluation string idiomatic to the target language
   eval_str(scc, implementation_str, allow_autofill): evaluate an implementation including constraints by running it in a target-language executor. if allow_autofill and the execution fails due to an undefined reference, attempt autofill
    exec_implementation(implementation_str): run the implementation, including constraints/tests, in a target-language-specific executor, returning whether it was successful
    attempt_autofill(scc, implementation_str, undefined_fn_use_example): create a new function node for the referenced function, then re-attempt to execute autofill
     add_undefined_fn(scc, implementation_str, undefined_fn_caller, undefined_fn_use_example): create a new function node for the undefined function as a child of the function which calls it and add it to the scc and implementation string. prompt the language model with the usage example as the description to generate a set of implementations.
      fn_implementation
     eval_str
   on_fail(scc, scc_graph, allow_autofill, allow_autodecomp): if allowing autodecomposition, attempt to decompose. otherwise, raise an error highlighting the scc which could not be compiled
    attempt_autodecomp(scc, scc_graph, allow_autofill, allow_autodecomp): prompt the language model to decompose each unimplemented function node.
     prompt_model(fn_node): prompt the language model, asking it to generate a "fn_name(arg): desc" for each subfunction necessary to implement the function node. add those functions to the scc, including a set of possible implementations for each.
      fn_implementation
     compile_scc
    raise_error(scc): raise an error that Parsel could compile the scc
```

Figure A.10. Longer pseudocode of Parsel, including automatic infilling and automatic decomposition.

D. Parsel Overview (Detailed)

We include a more detailed figure outlining Parsel.

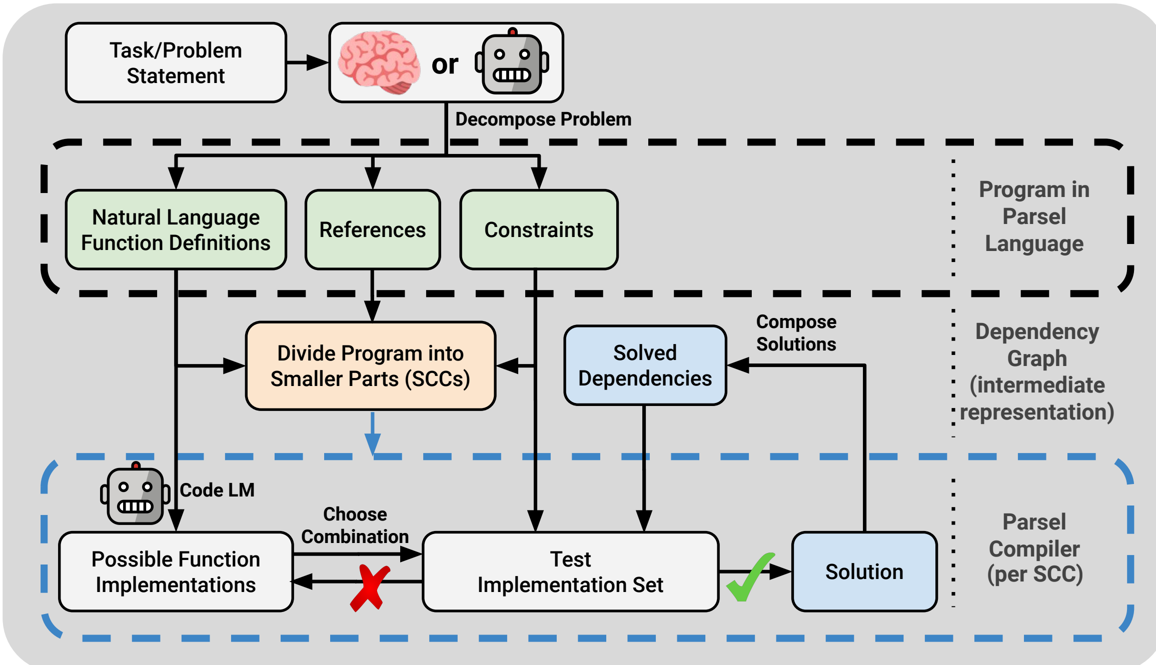

Figure A.11. **Parsel overview (detailed)**.

E. Lisp Interpreter

We include the Parsel code for a minimal Lisp interpreter.

```
An env is a dictionary of {'var':val} pairs, with a link to its outer environment in env['_outer'].
A procedure is a lambda expression, with parms, body, and env which calls eval_exp on the body.
#*#*#
evaluate_program(program): Initialize a standard environment. Parse and evaluate a list of expressions, returning the final result.
['(define square (lambda (r) (* r r)))', '(square 3)'] -> 9
  get_env(parms, args, env=None): Return a new env inside env with parms mapped to their corresponding args, and env as the new env's outer env.
  [], [] -> {'_outer': None}
  ['a'], [1] -> {'a': 1, '_outer': None}
  standard_env(includes=['math','ops','simple_math']): An environment with some Scheme standard procedures. Start with an environment and update it with standard functions.
  [] -> {'_outer': None}
    get_math(): Get a dictionary mapping math library function names to their functions.
    get_ops(): Get a dictionary mapping operator symbols to their functions: +, -, *, /, >, <, >=, <=, =.
    get_simple_math(): Get a dictionary mapping 'abs', 'min', 'max', 'not', 'round' to their functions.
    apply_fn_dict_key(fn_dict_generator, key, args_list): Return the value of fn_dict_generator()[key](*args_list) in standard_env.
    get_math, 'sqrt', [4] -> 2.0
    get_ops, '+', [1, 2] -> 3
    get_simple_math, 'abs', [-1] -> 1
      get_math
      get_ops
      get_simple_math
  parse_and_update(expression, env): Parse an expression, return the result.
  "(+ 1 (* 2 3))", {'+': (lambda x, y: x + y), '*': (lambda x, y: x * y), '_outer': None} -> 7
    eval_exp(x, env): Evaluate an expression in an environment and return the result. Check if x is a list, a string, or neither, and call the corresponding function.
    1, {'_outer': None} -> 1
      find(env, var): Find the value of var in the innermost env where var appears.
      {'a':4, '_outer':None}, 'a' -> 4
      {'_outer':{'a':4, '_outer':None}}, 'a' -> 4
      {'a':3, '_outer':{'a':4, '_outer':None}}, 'a' -> 3
      string_case(x, env): Return find(env, x).
      'a', {'a':4, '_outer':None} -> 4
        find
      list_case(x, env): Handle the function specified by the first value of x. Handle the first value of x being quote, if, define, set!, lambda, or otherwise. Return the result.
      ['quote', 'a'], {'_outer': None} -> 'a'
      ['if', True, 1, 2], {'_outer': None} -> 1
      ['define', 'a', 1], {'_outer': None} -> None
        get_procedure(parms, body, env): Return a procedure which evaluates body in a new environment with parms bound to the args passed to the procedure (in the same order as parms).
          eval_procedure(parms, body, env, args): Gets a procedure and returns the result of evaluating proc(*args) in env. Should not be called directly.
          ['r'], ['*', 'pi', ['*', 'r', 'r']], {'*': (lambda x, y: x * y), 'pi': 3, '_outer': None}, [1] -> 3
            get_procedure
            get_env
            eval_exp
        otherwise_case(x, env): Get the procedure by evaluating the first value of x. Then, evaluate the arguments and apply the procedure to them. Return the result.
        ['+', 1, 2], {'+': (lambda x, y: x + y), '_outer': None} -> 3
          eval_exp
        eval_exp
      not_list_case(x, env): Return x
      1, {} -> 1
    parse(program): Read a Scheme expression from a string.
    '(1 + (2 * 3))' -> [1, '+', [2, '*', 3]]
      tokenize(s): Convert a string into a list of tokens, including parens.
      "1 + 2" -> ['1', '+', '2']
      "1 + (2 * 3)" -> ['1', '+', '(', '2', '*', '3', ')']
      read_from_tokens(tokens): Translate tokens to their corresponding atoms, using parentheses for nesting lists.
      ['(', '1', '+', '(', '2', '*', '3', ')', ')'] -> [1, '+', [2, '*', 3]]
        atom(token): Numbers become numbers; every other token is a string.
        "1" -> 1
        "a" -> "a"
        "1.2" -> 1.2
    nested_list_to_str(exp): Convert a nested list into a string with nesting represented by parentheses.
    1 -> "1"
    [1, '+', [2, '*', 3]] -> "(1 + (2 * 3))"
```

Figure A.12. Full Lisp interpreter implementation in Parsel, including constraints.

F. Case Study

We include a simple example function we could not generate with Codex (Chen et al., 2021) directly from the top-level description in Figure A.13. The corresponding Python code (included in the appendix) is exactly 58 non-whitespace lines of code, including 17 lines of comments (3 corresponding to the descriptions), 2 asserts, and 39 lines implementing the three functions described as well as an automatically generated `get_number_of_active_cells_around_cell` function. In fact, using automatic decomposition, as discussed in Subsection 3.6, it is not necessary to provide any of the function descriptions besides the top one. The model is (unsurprisingly) able to understand that `game_of_life_inversion_iteration` can be broken down into `invert_array` and `game_of_life_iteration`.

```
game_of_life_inversion_iteration(array_at_time_t): Takes a board and returns the next iteration of the game of life, but with all values flipped
[[0,0,1],[1,0,0],[1,0,0]] -> [[1,1,1],[1,0,1],[1,1,1]]
[[0,1,0,0],[1,0,1,0],[1,0,0,1],[0,1,1,0]] -> [[1,0,1,1],[0,1,0,1],[0,1,1,0],[1,0,0,1]]
   game_of_life_iteration(array_at_time_t) -> array_at_time_t_plus_1: Takes a board with active and inactive cells as a list of lists and returns the next iteration of the game of life
   array_inversion(array) -> inverted_array: Invert a square array by flipping 0's and 1's
```

Figure A.13. An example Parsel program for Python that takes in a list of lists representing a state of Conway's game of life (Games, 1970) and returns the next state, with all the values inverted.

G. Parsel Prompts

```
# Description: given a list of lists representing the cost of each edge, return an adjacency matrix corresponding to the minimum spanning true.
def minimum_spanning_tree(cost_matrix):
```

Figure A.14. Codex Prompt for an example leaf node

```
# Description: given a list of lists representing the cost of building a road between any two cities, and a list representing the cost of building an airport in a city, return a new cost matrix with a new node corresponding to the sky.
# Signature: sky_city_cost(city_road_cost, city_airport_cost)
from helpers import sky_city_cost

# Description: given a list of lists representing the cost of each edge, return an adjacency matrix corresponding to the minimum spanning true.
# Signature: minimum_spanning_tree(cost_matrix)
from helpers import minimum_spanning_tree

# Description: given a list of lists representing an adjacency matrix, return a list of the nodes connected to the final node. However, if only one node is connected to the final node, return an empty list.
# Signature: final_node_connectors(adjacency_matrix)
from helpers import final_node_connectors

# Description: given a matrix representing the cost of building a road between any two cities, and a list representing the cost of building an airport in a city (where any two cities with airports are connected), return a list of the cities that should have airports built in them to minimize the total cost of building roads and airports such that all cities are connected. The list should be sorted in ascending order.
# Uses: sky_city_cost, minimum_spanning_tree, final_node_connectors
def select_airport_cities(city_road_cost, city_airport_cost):
```

Figure A.15. Codex Prompt for an example merge node

```
# Reviewer:
# Please explain the above function in one sentence with as much detail as possible.
# In your one-sentence description, specify the range and domain of your function precisely.
# Your description should be clear enough that someone could reimplement the function from it.
# Author:
# Sounds good, here’s my one-sentence explanation of {name}:
# {name}
```

Figure A.16. Prompt format to generate descriptions for backtranslation

H. APPS Backtranslation

H.1. Backtranslation / decompiling.

We anticipate that there are many programs that LLMs can implement by first generating Parsel code. But, as Parsel is a new framework, while language models can sometimes generate Parsel programs with few-shot prompts, it is not a syntax they have previously encountered. Thus, we may want to use existing code in other languages to construct datasets of Parsel programs from other languages. This requires us to first extract the call graph from the code, generate descriptions for each of the functions, and then generate Parsel programs from the graph. This call graph representation is convenient, so it is useful to have a bidirectional method to produce a graph from Parsel code and to produce Parsel code from the graph.

We filter the dataset to problems with starter code (providing the name of the evaluated function) and unit tests (provided as input-output pairs). For those tasks, we select solutions that define and call at least three functions, with at least one over 4 lines long and none over 15 lines.

As a proof of concept, we show 10 Parsel solutions which we could automatically generate from the APPS solutions. We generated the descriptions by prompting Codex to explain each function and its inputs and outputs. From this, we use backtranslation to attempt to implement these solutions in Python. We then verify that they are correct by applying the original unit tests as constraints on the root function. As mentioned in Section 1, the Parsel code is substantially shorter in terms of lines of code. We include these in Appendix H.

H.2. Examples

We exclude the asserts in these examples for brevity - they correspond to those in the original dataset.

```
longest_palindrome(s): longest_palindrome takes a string s and returns the longest palindrome in s.
 is_palindrome(s): is_palindrome returns True if the string s is the same forwards and backwards, and
   ↪ False otherwise.
 check(li, ri, s): check takes a string s, a left index li, and a right index ri, and returns the
   ↪ longest palindrome that starts at or before li and ends at or after ri.
  is_palindrome
```

Figure A.17. Train Problem 1638, Solution 2

```
# longest_palindrome takes a string s and returns the longest palindrome in s.
def longest_palindrome(s):
   if len(s) <= 1:
      return s
   else:
      longest = s[0]
      for i in range(len(s)):
         for j in range(len(s)):
            if is_palindrome(check(i, j, s)) and len(check(i, j, s)) > len(longest):
               longest = check(i, j, s)
      return longest

# is_palindrome returns True if the string s is the same forwards and backwards, and False otherwise.
def is_palindrome(s):
   if len(s) <= 1:
      return True
   else:
      return s[0] == s[-1] and is_palindrome(s[1:-1])

# check takes a string s, a left index li, and a right index ri, and returns the longest palindrome that starts at or before li and ends at or after ri.
def check(li, ri, s):
   while li >= 0 and ri < len(s) and s[li] == s[ri]:
      li -= 1
      ri += 1
   return s[li+1:ri]
```

Figure A.18. Train Problem 1638, Solution 2

```
case_id(c_str): case_id takes a string and returns a string that is either "kebab", "snake", "camel", or "none" depending on whether the input string is in kebab case, snake case, camel case, or none of the above.
 is_snake(s): is_snake takes a string and returns True if the string is snake_case and False otherwise.
 is_kebab(s): is_kebab takes a string and returns True if the string is a kebab-case string, and False otherwise.
 is_camel(s): is_camel returns True if the string s is not lowercase, does not contain dashes, and does not contain underscores.
```

Figure A.19. Train Problem 2892, Solution 7

```python
# case_id takes a string and returns a string that is either "kebab", "snake", "camel", or "none" depending on whether the input string is in kebab case, snake case, camel case, or none of the above.
def case_id(c_str):
    if is_snake(c_str) == True:
        return "snake"
    elif is_kebab(c_str) == True:
        return "kebab"
    elif is_camel(c_str) == True:
        return "camel"
    else:
        return "none"

# is_snake takes a string and returns True if the string is snake_case and False otherwise.
def is_snake(s):
    if s[0].isalpha() and s[0].islower() and len(s) > 1:
        for char in s:
            if char.isalpha():
                if char.isupper():
                    return False
            elif char == '_':
                pass
            else:
                return False
        return True
    else:
        return False

# is_kebab takes a string and returns True if the string is a kebab-case string, and False otherwise.
def is_kebab(s):
    # if s is empty, False
    if s == '':
        return False
    # if s is not a string, False
    if type(s) != str:
        return False
    # if s is not lowercase, False
    if s != s.lower():
        return False
    # if s contains anything other than a-z or -, False
    for c in s:
        if not (c.isalpha() or c == '-'):
            return False
    # if s contains a - at the beginning or end, False
    if s[0] == '-' or s[-1] == '-':
        return False
    # if s contains more than one - in a row, False
    for i in range(len(s)-1):
        if s[i] == '-' and s[i+1] == '-':
            return False
    # otherwise, True
    return True


# is_camel returns True if the string s is not lowercase, does not contain dashes, and does not contain underscores.
def is_camel(s):
    return s != s.lower() and s.find('_') == -1 and s.find('-') == -1
```

Figure A.20. Train Problem 2892, Solution 7

```
find_2nd_largest(arr): find_2nd_largest takes a list of numbers and returns the second largest number in the list.
 filter_int(arr): filter_int takes an array of integers and returns an array of integers.
 sec_big(a, b): sec_big takes two numbers and returns the smaller of the two.
 sort(arr): sort takes an array of numbers and returns a sorted array of numbers.
 is_diff(arr): is_diff takes an array of numbers and returns True if there are any two numbers in the array that are different, and False if all the numbers in the array are the same.
```

Figure A.21. Train Problem 3090, Solution 9

```python
# find_2nd_largest takes a list of numbers and returns the second largest number in the list.
def find_2nd_largest(arr):
   if len(arr) == 1:
      return None
   arr = filter_int(arr)
   if len(arr) == 0:
      return None
   if len(arr) == 1:
      return arr[0]
   arr = sort(arr)
   if is_diff(arr) == False:
      return None
   res = arr[len(arr) - 2]
   return res

# filter_int takes an array of integers and returns an array of integers.
def filter_int(arr):
   return list(filter(lambda x: type(x) == int, arr))

# sec_big takes two numbers and returns the smaller of the two.
def sec_big(a, b):
   if a > b:
      return b
   else:
      return a

# sort takes an array of numbers and returns a sorted array of numbers.
def sort(arr):
   return sorted(arr)

# is_diff takes an array of numbers and returns True if there are any two numbers in the array that are different, and False if all the numbers in the array are the same.
def is_diff(arr):
   for i in range(len(arr) - 1):
      if arr[i] != arr[i + 1]:
         return True
   return False
```

Figure A.22. Train Problem 3090, Solution 9

```
happy_numbers(n): happy_numbers takes a positive integer n and returns a list of all the happy numbers between 1 and n, inclusive.
 _is_happy_number(number): _is_happy_number takes a positive integer and returns True if the number is a happy number, False otherwise.
  _sum_squares(number): _sum_squares takes a non-negative integer and returns the sum of the squares of its digits.
```

Figure A.23. Train Problem 3140, Solution 7

```python
# happy_numbers takes a positive integer n and returns a list of all the happy numbers between 1 and
↪ n, inclusive.
def happy_numbers(n):
    return [i for i in range(1, n + 1) if _is_happy_number(i)]

# _is_happy_number takes a positive integer and returns True if the number is a happy number, False
↪ otherwise.
def _is_happy_number(number):
    # We want to make sure that the number is positive
    if number < 0:
        return False
    # We want to make sure that the number is not 1
    if number == 1:
        return True
    # We want to keep track of the numbers we have used
    used_numbers = []
    # We want to loop through the number
    while number not in used_numbers:
        # We want to add the number to the list of used numbers
        used_numbers.append(number)
        # We want to find the sum of the squares of the digits of the number
        number = _sum_squares(number)
        # We want to check if the number is one
        if number == 1:
            # We want to return True
            return True
    # We want to return False
    return False

# _sum_squares takes a non-negative integer and returns the sum of the squares of its digits.
def _sum_squares(number):
    if number < 0:
        raise ValueError
    else:
        number = str(number)
        sum = 0
        for i in number:
            sum += int(i) ** 2
        return sum
```

Figure A.24. Train Problem 3140, Solution 7

```
am_i_wilson(n): am_i_wilson(n) returns True if n is a prime number between 2 and 563, inclusive, and
↪ False otherwise.
 is_prime(n): is_prime takes a positive integer n and returns True if n is prime and False otherwise.
 factorial(n): factorial(n) returns the product of all integers from 1 to n, inclusive.
```

Figure A.25. Train Problem 3229, Solution 26

```python
# am_i_wilson(n) returns True if n is a prime number between 2 and 563, inclusive, and False
 ↪ otherwise.
def am_i_wilson(n):
   if is_prime(n) and 2 <= n and n <= 563:
      return (factorial(n-1) + 1) % (n**2) == 0
   else:
      return False

# is_prime takes a positive integer n and returns True if n is prime and False otherwise.
def is_prime(n):
   if n == 2:
      return True
   if n == 3:
      return True
   if n % 2 == 0:
      return False
   if n % 3 == 0:
      return False
   i = 5
   w = 2
   while i * i <= n:
      if n % i == 0:
         return False
      i += w
      w = 6 - w
   return True


# factorial(n) returns the product of all integers from 1 to n, inclusive.
def factorial(n):
   if n == 0:
      return 1
   else:
      return n * factorial(n-1)
```

Figure A.26. Train Problem 3229, Solution 26

```
am_i_wilson(n): am_i_wilson takes a positive integer n and returns True if n is prime and (n-1)! + 1
 ↪ is divisible by n^2, and False otherwise.
  fac(n): fac is a function that takes a positive integer n and returns the product of all integers
   ↪ from 1 to n.
  is_prime(n): is_prime takes a positive integer n and returns True if n is prime and False otherwise.
```

Figure A.27. Train Problem 3229, Solution 71

```
# am_i_wilson takes a positive integer n and returns True if n is prime and (n-1)! + 1 is divisible by n^2, and False otherwise.
def am_i_wilson(n):
    return is_prime(n) and (fac(n-1) + 1) % n**2 == 0

# fac is a function that takes a positive integer n and returns the product of all integers from 1 to n.
def fac(n):
    if n == 0:
        return 1
    return n * fac(n-1)
# is_prime takes a positive integer n and returns True if n is prime and False otherwise.
def is_prime(n):
    if n == 2:
        return True
    elif n < 2 or n % 2 == 0:
        return False
    for i in range(3, int(n**0.5)+1, 2):
        if n % i == 0:
            return False
    return True
```

Figure A.28. Train Problem 3229, Solution 71

```
evil(n): evil(n) returns "It's Evil!" if n is an evil number, otherwise it returns "It's Odious!" The range of evil is the set of all integers, and the domain is the set of all strings.
 evilometer(n): evilometer(n) is a generator that yields n times if n is even, and yields n // 2 times if n is odd.
```

Figure A.29. Train Problem 3321, Solution 33

```
# evil(n) returns "It's Evil!" if n is an evil number, otherwise it returns "It's Odious!" The range of evil is the set of all integers, and the domain is the set of all strings.
def evil(n):
    gen = evilometer(n)
    if sum(list(gen)) % 2 == 0:
        return "It's␣Evil!"
    else:
        return "It's␣Odious!"

# evilometer(n) is a generator that yields n times if n is even, and yields n // 2 times if n is odd.
def evilometer(n):
    while n:
        yield n
        if n % 2:
            n //= 2
        else:
            n -= 1
```

Figure A.30. Train Problem 3321, Solution 33

```
circular_prime(number): circular_prime takes a number and returns True if it is a circular prime, and False otherwise.
 rotate(l, n): rotate takes a list and an integer and returns a new list with the last n elements of the original list moved to the front.
 is_prime(n): is_prime returns True if n is a prime number, and False otherwise.
```

Figure A.31. Train Problem 3368, Solution 5

```python
# circular_prime takes a number and returns True if it is a circular prime, and False otherwise.
def circular_prime(number):
    if is_prime(number) == False:
        return False
    else:
        for i in range(len(str(number))-1):
            number = rotate(str(number), 1)
            if is_prime(int(number)) == False:
                return False
        return True

# rotate takes a list and an integer and returns a new list with the last n elements of the original
    ↪ list moved to the front.
def rotate(l, n):
    return l[-n:] + l[:-n]

# is_prime returns True if n is a prime number, and False otherwise.
def is_prime(n):
    if n < 2:
        return False
    for i in range(2, int(n ** 0.5) + 1):
        if n % i == 0:
            return False
    return True
```

Figure A.32. Train Problem 3368, Solution 5

```
circular_prime(n): circular_prime(n) returns True if n is a circular prime, and False otherwise.
  is_prime(n): is_prime takes a number n and returns True if n is prime and False if n is not prime.
  circul_num(n, l): circul_num takes an integer n and an integer l, and returns the integer that
    ↪ results from moving the first digit of n to the end of n, and then padding the result with zeros
    ↪  until it has length l.
```

Figure A.33. Train Problem 3368, Solution 6

```
# circular_prime(n) returns True if n is a circular prime, and False otherwise.
def circular_prime(n):
    # Check if n is a prime number.
    if not is_prime(n):
        return False
    else:
        # Get the length of n.
        length = len(str(n))
        # Get the first digit of n.
        first_digit = str(n)[0]
        # Get the number of rotations.
        rotations = length
        # Rotate n.
        while rotations > 0:
            # Get the new number.
            n = circul_num(n, length)
            # Check if n is a prime number.
            if not is_prime(n):
                return False
            # Decrement the number of rotations.
            rotations -= 1
        return True
# is_prime takes a number n and returns True if n is prime and False if n is not prime.
def is_prime(n):
    if n == 2 or n == 3:
        return True
    if n % 2 == 0 or n < 2:
        return False
    for i in range(3, int(n**0.5)+1, 2): # only odd numbers
        if n % i == 0:
            return False
    return True

# circul_num takes an integer n and an integer l, and returns the integer that results from moving the first digit of n to the end of n, and then padding the result with zeros until it has length l.
def circul_num(n, l):
    n = str(n)
    digit = n[0]
    n = n[1:] + digit
    n = n.ljust(l, "0")
    return int(n)
```

Figure A.34. Train Problem 3368, Solution 6

```
list_squared(m, n): list_squared(m, n) returns a list of lists of integers, where each sublist contains an integer and its sum of squared divisors, and the integers are in the range [m, n] (inclusive).
 divisors_list(num): divisors_list takes a number and returns a list of all the divisors of that number.
 sum_squares(nums): sum_squares takes a list of numbers and returns the sum of the squares of those numbers.
 isSquarable(num): isSquarable takes a number and returns True if the number is a perfect square, and False otherwise.
```

Figure A.35. Train Problem 3393, Solution 35

```python
# list_squared(m, n) returns a list of lists of integers, where each sublist contains an integer and
    ↪ its sum of squared divisors, and the integers are in the range [m, n] (inclusive).
def list_squared(m, n):
    result = []
    for num in range(m, n + 1):
        divisors = divisors_list(num)
        sum_of_squares = sum_squares(divisors)
        if isSquarable(sum_of_squares):
            result.append([num, sum_of_squares])
    return result
# divisors_list takes a number and returns a list of all the divisors of that number.
def divisors_list(num):
    divisors = []
    for i in range(1, num + 1):
        if num % i == 0:
            divisors.append(i)
    return divisors


# sum_squares takes a list of numbers and returns the sum of the squares of those numbers.
def sum_squares(nums):
    sum = 0
    for i in range(len(nums)):
        sum += nums[i]**2
    return sum

# isSquarable takes a number and returns True if the number is a perfect square, and False otherwise.
def isSquarable(num):
    root = num ** (1/2)
    if root % 1 == 0:
        return True
    else:
        return False
```

Figure A.36. Train Problem 3393, Solution 35

I. Compiled Examples

```python
# Takes a board and returns the next iteration of the game of life, but with all values flipped
def game_of_life_inversion_iteration(array_at_time_t):
    # Your code here
    #return game_of_life_iteration(invert_array(array_at_time_t))
    return invert_array(game_of_life_iteration(array_at_time_t))

# Takes a board and returns the next iteration of the game of life
def invert_array(array_at_time_t):
    return [list(map(lambda x: 1-x, row)) for row in array_at_time_t]

# Takes a board and returns the board with all values flipped
def game_of_life_iteration(array_at_time_t):
    # The array that will be returned
    array_at_time_t_plus_1 = []

    # Iterate through the rows of the array
    for i in range(0, len(array_at_time_t)):
        # The array that will contain the next row
        next_row = []

        # Iterate through the columns of the array
        for j in range(0, len(array_at_time_t[i])):
            # The number of neighbors
            num_neighbors = 0

            # Iterate through the neighbors of the cell
            for k in range(-1, 2):
                for l in range(-1, 2):
                    # Don't count the cell itself
                    if k == 0 and l == 0:
                        continue

                    # Check if the neighbor is valid
                    if i + k >= 0 and i + k < len(array_at_time_t) and j + l >= 0 and j + l < len(
                     ↪ array_at_time_t[i]):
                        # If the neighbor is alive, increment the number of neighbors
                        if array_at_time_t[i + k][j + l] == 1:
                            num_neighbors += 1

            # If the cell is alive, check if it should die
            if array_at_time_t[i][j] == 1:
                if num_neighbors < 2 or num_neighbors > 3:
                    next_row.append(0)
                else:
                    next_row.append(1)
            # If the cell is dead, check if it should become alive
            else:
                if num_neighbors == 3:
                    next_row.append(1)
                else:
                    next_row.append(0)

        # Add the next row to the array
        array_at_time_t_plus_1.append(next_row)

    # Return the next array
    return array_at_time_t_plus_1

assert game_of_life_inversion_iteration([[0, 0, 1], [1, 0, 0], [1, 0, 0]]) == [[1, 1, 1], [1, 0, 1],
 ↪ [1, 1, 1]]
assert game_of_life_inversion_iteration([[0, 1, 0, 0], [1, 0, 1, 0], [1, 0, 0, 1], [0, 1, 1, 0]]) ==
 ↪ [[1, 0, 1, 1], [0, 1, 0, 1], [0, 1, 1, 0], [1, 0, 0, 1]]
```

Figure A.37. Compilation of the program taking Conway's game of life (Games, 1970) and returning the next state, with the values inverted. In this case, we use automatic decomposition to generate the two subfunctions of `game_of_life_iteration`.

```python
# Takes a board with active and inactive cells and returns the next iteration of the game of life, but with all the values flipped
def game_of_life_inversion_iteration(array_at_time_t):
    return array_inversion(game_of_life_iteration(array_at_time_t))

# Takes a board with active and inactive cells as a list of lists and returns the next iteration of the game of life
def game_of_life_iteration(array_at_time_t):
    array_at_time_t_plus_1 = []
    for row in range(len(array_at_time_t)):
        array_at_time_t_plus_1.append([])
        for column in range(len(array_at_time_t[row])):
            array_at_time_t_plus_1[row].append(0)
    for row in range(len(array_at_time_t)):
        for column in range(len(array_at_time_t[row])):
            if array_at_time_t[row][column] == 1:
                if count_neighbors(array_at_time_t, row, column) < 2:
                    array_at_time_t_plus_1[row][column] = 0
                elif count_neighbors(array_at_time_t, row, column) > 3:
                    array_at_time_t_plus_1[row][column] = 0
                else:
                    array_at_time_t_plus_1[row][column] = 1
            else:
                if count_neighbors(array_at_time_t, row, column) == 3:
                    array_at_time_t_plus_1[row][column] = 1
    return array_at_time_t_plus_1

# Invert a square array by replacing all 0's with 1's and vice versa
def array_inversion(array):
    inverted_array = []
    for i in range(len(array)):
        inverted_array.append([])
        for j in range(len(array[i])):
            inverted_array[i].append(1 - array[i][j])
    return inverted_array

#
def count_neighbors(array_at_time_t, row, column):
    count = 0
    for i in range(row-1, row+2):
        for j in range(column-1, column+2):
            if i == row and j == column:
                continue
            if i < 0 or j < 0:
                continue
            if i >= len(array_at_time_t) or j >= len(array_at_time_t[0]):
                continue
            if array_at_time_t[i][j] == 1:
                count += 1
    return count


assert game_of_life_inversion_iteration([[0, 0, 1], [1, 0, 0], [1, 0, 0]]) == [[1, 1, 1], [1, 0, 1], [1, 1, 1]]
assert game_of_life_inversion_iteration([[0, 1, 0, 0], [1, 0, 1, 0], [1, 0, 0, 1], [0, 1, 1, 0]]) == [[1, 0, 1, 1], [0, 1, 0, 1], [0, 1, 1, 0], [1, 0, 0, 1]]

assert game_of_life_iteration([[0, 0, 1], [1, 0, 0], [1, 0, 0]]) == [[0, 0, 0], [0, 1, 0], [0, 0, 0]]
assert game_of_life_iteration([[0, 1, 0, 0], [1, 0, 1, 0], [1, 0, 0, 1], [0, 1, 1, 0]]) == [[0, 1, 0, 0], [1, 0, 1, 0], [1, 0, 0, 1], [0, 1, 1, 0]]

assert array_inversion([[1]]) == [[0]]
assert array_inversion([[0, 1], [1, 0]]) == [[1, 0], [0, 1]]
```

Figure A.38. Compilation of the program taking Conway's game of life (Games, 1970) and returning the next state, with the values inverted. In this case, we use automatic infilling to generate the `count_neighbors` function.

```python
# Calls base_case if 1, otherwise recursion_rule
def collatz_recursion(num, cur_list=list()):
    if num == 1:
        return base_case(num, cur_list)
    else:
        return recursion_rule(num, cur_list)

# Returns the list with the number appended to it
def base_case(num, cur_list):
    cur_list.append(num)
    return cur_list


# Add num to list, collatz with 3n + 1 if odd or n / 2 if even
def recursion_rule(num, cur_list):
    cur_list.append(num)
    if num % 2 == 0:
        return collatz_recursion(num / 2, cur_list)
    else:
        return collatz_recursion((3 * num) + 1, cur_list)


assert collatz_recursion(19) == [19, 58, 29, 88, 44, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8,
    4, 2, 1]
```

Figure A.39. Compilation of the program generating a list corresponding to the Collatz conjecture.

```
-- if p ∧ q, then q ∧ p
lemma p_q_implies_q_p(p q: Prop):

   p ∧ q → q ∧ p :=
begin
   intro h,
   cases h with hp hq,
   split,
      exact hq,
      exact hp,
end

-- Description: if p ∨ q, then q ∨ p
-- if q ∧ p, then p ∧ q
lemma q_p_implies_p_q(p q: Prop):

   (q ∧ p) → (p ∧ q) :=
begin
  intro h,
  split,
   exact h.right,
   exact h.left,
end

/-
  Theorem:
   If q ∧ p, then p ∧ q
-/
-- the and operator is commutative
lemma and_commute(p q: Prop):
  (p ∧ q → q ∧ p) ∧ (q ∧ p → p ∧ q) :=

begin
  apply and.intro,
  { apply p_q_implies_q_p },
  { apply q_p_implies_p_q }
end

-- Description: if p ∧ q, then p
-- Signature: p_and_q_implies_p(p q: Prop)

-- show (p ∧ q → q ∧ p) ∧ (q ∧ p → p ∧ q)
```

Figure A.40. Generated proof of `and_commute`. We trim the post-proof comments elsewhere.

```python
# given a list of lists representing the cost of building a road between any two cities, and a list representing the cost of building an airport in a city, return a new cost matrix with a new node corresponding to the sky.
def sky_city_cost(city_road_cost, city_airport_cost):
    """
    :param city_road_cost: list of lists representing cost of road between any two cities
    :param city_airport_cost: list representing cost of an airport in a city
    :return: new cost matrix with a new node corresponding to the sky
    """
    # add new node for sky to cost matrix
    num_cities = len(city_road_cost)
    sky_city_cost = [[0 for _ in range(num_cities + 1)] for _ in range(num_cities + 1)]
    for i in range(num_cities):
        for j in range(num_cities):
            sky_city_cost[i][j] = city_road_cost[i][j]
    for i in range(num_cities):
        sky_city_cost[i][-1] = city_airport_cost[i]
        sky_city_cost[-1][i] = city_airport_cost[i]
    return sky_city_cost

# given a list of lists representing the cost of each edge, return an adjacency matrix corresponding to the minimum spanning true.
def minimum_spanning_tree(cost_matrix):
    # This is a list of the vertices that have been added to the MST
    visited = [0]
    # This is a list of the vertices that have not been added to the MST
    unvisited = [i for i in range(1, len(cost_matrix))]
    # This is a list of edges that are part of the MST
    edges = []
    # This is the adjacency matrix corresponding to the MST
    adjacency_matrix = [[0 for i in range(len(cost_matrix))] for j in range(len(cost_matrix))]
    while len(unvisited) > 0:
        # Get the index of the minimum edge
        min_edge_index = -1
        min_edge_value = float('inf')
        for i in range(len(visited)):
            for j in range(len(unvisited)):
                if cost_matrix[visited[i]][unvisited[j]] < min_edge_value:
                    min_edge_index = (visited[i], unvisited[j])
                    min_edge_value = cost_matrix[visited[i]][unvisited[j]]
        # Add the minimum edge to our MST
        edges.append(min_edge_index)
        # Add the unvisited vertex to the list of visited vertices
        visited.append(min_edge_index[1])
        # Remove the unvisited vertex from the list of unvisited vertices
        unvisited.remove(min_edge_index[1])
    # Add edges to the adjacency matrix
    for edge in edges:
        adjacency_matrix[edge[0]][edge[1]] = 1
        adjacency_matrix[edge[1]][edge[0]] = 1
    return adjacency_matrix

# given a list of lists representing an adjacency matrix, return a list of the nodes connected to the final node. However, if only one node is connected to the final node, return an empty list.
def final_node_connectors(adjacency_matrix):
    final_node = len(adjacency_matrix) - 1
    final_node_connectors = []
    for i in range(len(adjacency_matrix) - 1):
        if adjacency_matrix[i][final_node] == 1:
            final_node_connectors.append(i)
    if len(final_node_connectors) == 1:
        return []
    return final_node_connectors

# given a matrix representing the cost of building a road between any two cities, and a list representing the cost of building an airport in a city (where any two cities with airports are connected), return a list of the cities that should have airports built in them to minimize the total cost of building roads and airports such that all cities are connected. The list should be sorted in ascending order.
def select_airport_cities(city_road_cost, city_airport_cost):
    cost_matrix = sky_city_cost(city_road_cost, city_airport_cost)
    adjacency_matrix = minimum_spanning_tree(cost_matrix)
    return final_node_connectors(adjacency_matrix)

assert repr(str(select_airport_cities([[0, 3, 3], [3, 0, 3], [3, 3, 0]], [0, 0, 0]))) == repr(str([0, 1, 2]))
assert repr(str(select_airport_cities([[0, 3, 3], [3, 0, 3], [3, 3, 0]], [10, 10, 10]))) == repr(str([]))
assert repr(str(select_airport_cities([[0, 10, 3], [10, 0, 11], [3, 11, 0]], [1, 4, 5]))) == repr(str([0, 1]))

assert repr(str(sky_city_cost([[1, 2, 3], [1, 2, 3], [1, 2, 3]], [4, 5, 6]))) == repr(str([[1, 2, 3, 4], [1, 2, 3, 5], [1, 2, 3, 6], [4, 5, 6, 0]]))

assert repr(str(minimum_spanning_tree([[0, 1, 3, 4], [1, 0, 2, 100], [3, 2, 0, 5], [4, 100, 5, 0]]))) == repr(str([[0, 1, 0, 1], [1, 0, 1, 0], [0, 1, 0, 0], [1, 0, 0, 0]]))

assert repr(str(final_node_connectors([[0, 1, 0, 1], [1, 0, 1, 0], [0, 1, 0, 0], [1, 0, 0, 0]]))) == repr(str([]))
assert repr(str(final_node_connectors([[0, 1, 0, 1], [1, 0, 1, 0], [0, 1, 0, 1], [1, 0, 1, 0]]))) == repr(str([0, 2]))
```

Figure A.41. Generated solution for `select_airport_cities`.

```python
# chess_game takes a board and returns the winner of the game.
def chess_game(board):
    if weight(board) == 0:
        return 'Draw'
    if weight(board) > 0:
        return 'White'
    if weight(board) < 0:
        return 'Black'

# weight takes a board and returns the weight of the board.
def weight(board):
        weight = 0
        for row in range(len(board)):
                for col in range(len(board[row])):
                        if board[row][col] != '.':
                                weight += weight_helper(board, row, col, 0)
        return weight

# weight_helper takes a board, a row, a column, and a weight and returns the weight of the board.
def weight_helper(board, row, col, weight):
    if board[row][col] == '.':
        return 0
    elif board[row][col] == 'Q':
        return weight + 9
    elif board[row][col] == 'R':
        return weight + 5
    elif board[row][col] == 'B':
        return weight + 3
    elif board[row][col] == 'N':
        return weight + 3
    elif board[row][col] == 'P':
        return weight + 1
    elif board[row][col] == 'q':
        return weight - 9
    elif board[row][col] == 'r':
        return weight - 5
    elif board[row][col] == 'b':
        return weight - 3
    elif board[row][col] == 'n':
        return weight - 3
    elif board[row][col] == 'p':
        return weight - 1
    else:
        return weight

assert repr(str(chess_game('...QK...\n........\n........\n........\n........\n........\n........\n...rk...'))) == repr('White')
assert repr(str(chess_game('rnbqkbnr\npppppppp\n........\n........\n........\n........\nPPPPPPPP\nRNBQKBNR'))) == repr('Draw')
assert repr(str(chess_game('rppppppr\n...k....\n........\n........\n........\n........\nK...Q...\n........'))) == repr('Black')
assert repr(str(chess_game('....bQ.K\n.B......\n.....P..\n........\n........\n........\n...N.P..\n.....R..'))) == repr('White')
...
```

Figure A.42. Generated solution for Problem 368 of the APPS test set, identifying the leader of a chess game from the board.

J. APPS Decomposition Prompts and Evaluation Hyperparameters

We slightly loosen the requirements for Parsel programs generated by language models, treating redundant function definitions as references instead of raising errors. We sample everything with temperature=0.6, except the translations which we sample with temperature=0.2, a presence penalty of 0.1, and a logit bias to prevent it from generating the text "`def`", as Codex has a tendency to degenerate to producing Python even when prompted with Parsel examples. We allow at most 500 tokens per function, but in practice found that they typically used less than half of them.

For evaluation, we use a timeout of 0.04 seconds per solution and evaluate at most 100,000 implementations per generated Parsel program.

For the Codex-only ablation, we allow it to generate at most 1000 tokens, in large part due to the rate limit. In particular, there is a heuristic rate limit that rejects any calls requesting more than 4,000 tokens. As a result, any larger number of samples per problem would prevent batching more than 3 samples at a time.

```
"""An action plan is a list of strings that describes a sequence of steps to accomplish a task, To be correctly parsed, an action plan must be syntactically correct and contain only allowed actions and recognizable simple objects. Allowed actions: 'close' <arg1>, 'cut' <arg1>, 'drink' <arg1>, 'drop' <arg1>, 'eat' <arg1>, 'find' <arg1>, 'grab' <arg1>, 'greet' <arg1>, 'lie on' <arg1>, 'look at' <arg1>, 'open' <arg1>, 'plug in' <arg1>, 'plug out' <arg1>, 'point at' <arg1>, 'pour' <arg1> 'into' <arg2>, 'pull' <arg1>, 'push' <arg1>, 'put' <arg1> 'on' <arg2>, 'put' <arg1> 'in' <arg2>, 'put back' <arg1>, 'take off' <arg1>, 'put on' <arg1>, 'read' <arg1>, 'release', 'rinse' <arg1>, 'run to' <arg1>, 'scrub' <arg1>, 'sit on' <arg1>, 'sleep', 'squeeze' <arg1>, 'stand up', 'switch off' <arg1>, 'switch on' <arg1>, 'touch' <arg1>, 'turn to' <arg1>, 'type on' <arg1>, 'wake up', 'walk to' <arg1>, 'wash' <arg1>, 'watch' <arg1>, 'wipe' <arg1>. To satisfy the common-sense constraints, each action step in this action plan must not violate the set of its pre-conditions (e.g. the agent cannot grab milk from the fridge before opening it) and post-conditions (e.g. the state of the fridge changes from "closed" to "open" after the agent opens it)."""
#*#*#
task_plan(): return a list of strings that represents an action plan to put a mug on the stall and bread on the desk.
-> "executable"
   put_object_on(object, place): return a list of strings that represents an action plan to put an object in a place.
   "mug", "stall" -> "executable"
```

Figure A.43. Full Parsel program including header for Fig. 3 example, with the #*#*# as the header seperator. Note that we essentially just took the executability definition in (Huang et al., 2022) and added the list of available actions.

```
"""-----Solution-----

Propose a clever and efficient high-level solution for this problem. Consider all edge cases and failure modes.

Some common strategies include:
Constructive algorithms, Binary search, Depth-first search (DFS) and similar algorithms, Dynamic programming, Bitmasks, Brute force, Greedy algorithms, Graphs, Two pointers, Trees, Geometry, Graph matchings, Hashing, Probabilities, Data structures, Sortings, Games, Number theory, Combinatorics, Divide and conquer, Disjoint set union (DSU), Expression parsing

Write out your reasoning first, and then describe your high-level solution and explain why it is correct.
\"\"\"
Let's think step by step to come up with a clever algorithm."""
```

Figure A.44. High-level sketch prompt for APPS programs

```
"""-----Translation-----
# Here is an example calculating the probability of landing on the same character in a random shift of an input string, based on the following problem:
# Vasya and Kolya play a game with a string, using the following rules. Initially, Kolya creates a string s, consisting of small English letters, and uniformly at random chooses an integer k from a segment [0, len(s) - 1]. He tells Vasya this string s, and then shifts it k letters to the left, i.e. creates a new string t = s_k + 1s_k + 2... s_ns_1s_2... s_k. Vasya does not know the integer k nor the string t, but he wants to guess the integer k. To do this, he asks Kolya to tell him the first letter of the new string, and then, after he sees it, open one more letter on some position, which Vasya can choose.
# Vasya understands, that he can't guarantee that he will win, but he wants to know the probability of winning, if he plays optimally. He wants you to compute this probability.
# Note that Vasya wants to know the value of k uniquely, it means, that if there are at least two cyclic shifts of s that fit the information Vasya knowns, Vasya loses. Of course, at any moment of the game Vasya wants to maximize the probability of his win.
\"\"\"
generate_cyclic_shifts(input_str): Calculates the average number of unique characters in the substrings of the input string that start with each character.
 parse_input(input_str): Takes a string and returns the input string
 compute_a_and_letter_pos(input_str): Generates the str_as_number_list and letter_pos lists. str_as_number_list is a list of integers that is used to store the character values of the input string. str_as_number_list is initialized as a list of 0s for twice the length of the input string. The values are calculated by taking the ASCII value of each character in the string and subtracting the ASCII value of the character 'a'. letter_pos is a list of lists, with each sublist containing the indices at which a particular character appears in the input string.
 compute_unique_characters(c, str_as_number_list, letter_pos) -> ans: Calculates the maximum number of unique characters in all substrings (for k=1 to length) that start with the character represented by c. letter_pos is a list of lists, with each sublist containing the indices at which a character appears in the input string. str_as_number_list is a list of integers that is used to store the character values of the input string.
  compute_unique_characters_for_k(c, k, str_as_number_list, letter_pos): Create a counts list of zeros for each of the 26 alphabetical characters. For each i in the sublist of positions of letter_pos[c], increment counts at str_as_number_list[i + k]. Return the number of counts which are exactly one.
 to_output_str(ans, input_str): Returns a string representation of ans divided by the length of the input string.
\"\"\"
(6 lines)

# And here is an example identifying the largest binary number according to the following rules:
# The Little Elephant has an integer a, written in the binary notation. He wants to write this number on a piece of paper.
# To make sure that the number a fits on the piece of paper, the Little Elephant ought to delete exactly one any digit from number a in the binary record. At that a new number appears. It consists of the remaining binary digits, written in the corresponding order (possible, with leading zeroes).
# The Little Elephant wants the number he is going to write on the paper to be as large as possible. Help him find the maximum number that he can obtain after deleting exactly one binary digit and print it in the binary notation.
\"\"\"
largest_binary_number(input_str): Returns the largest binary number that can be made by removing at most one digit from the input string.
 parse_input(input_str): Takes a string and returns the input string
 remove_zero(binary_str): Remove the first zero from the input string.
 to_output_str(bigger_str): Returns the bigger string.
\"\"\"
(4 lines)

# Here is an example of the format applied to identifying the winner of the following game:
# It is so boring in the summer holiday, isn't it? So Alice and Bob have invented a new game to play. The rules are as follows. First, they get a set of n distinct integers. And then they take turns to make the following moves. During each move, either Alice or Bob (the player whose turn is the current) can choose two distinct integers x and y from the set, such that the set doesn't contain their absolute difference |x - y|. Then this player adds integer |x - y| to the set (so, the size of the set increases by one).
# If the current player has no valid move, he (or she) loses the game. The question is who will finally win the game if both players play optimally. Remember that Alice always moves first.
\"\"\"
identify_winner(input_str): Returns the winner of the game.
 parse_input(input_str): Takes a string containing the length on the first line and the integers on the second and returns the list of integers
 num_moves(l): The number of moves is the largest element in the list divided by the greatest common divisor of all elements in the list, minus the length of the list.
  all_gcd(l): Returns the greatest common divisor of all elements in the list
 to_output_str(num_moves): Returns the string 'Alice' if the number of moves is odd and 'Bob' if the number of moves is even
\"\"\"
(5 lines)

# Limak is a little bear who loves to play. Today he is playing by destroying block towers. He built n towers in a row. The i-th tower is made of h_i identical blocks. For clarification see picture for the first sample.
# Limak will repeat the following operation till everything is destroyed.
# Block is called internal if it has all four neighbors, i.e. it has each side (top, left, down and right) adjacent to other block or to the floor. Otherwise, block is boundary. In one operation Limak destroys all boundary blocks. His paws are very fast and he destroys all those blocks at the same time.
# Limak is ready to start. You task is to count how many operations will it take him to destroy all towers.
\"\"\"
destroy_towers(input_str): Returns the number of operations it takes to destroy all towers.
 parse_input(input_str): Takes a string containing the number of towers on the first line and the heights of the towers on the second and returns the list of heights
 side_ones(heights_list): From a list of ints, set the first and last elements to 1 and return the list
 destroy_from_left(side_list): Copy the list and set each each element to the minimum of itself and one more than the element to its left, starting from the second element
 destroy_from_right(side_list): Copy the list and set each each element to the minimum of itself and one more than the element to its right, starting from the second to last element
 min_list(l1, l2): Return a list of the minimum of the corresponding elements of l1 and l2
 to_output_str(min_list): Return the string representation of the maximum element in the list
\"\"\"
(7 lines)

# Alex decided to go on a touristic trip over the country.
# For simplicity let's assume that the country has $n$ cities and $m$ bidirectional roads connecting them. Alex lives in city $s$ and initially located in it. To compare different cities Alex assigned each city a score $w_i$ which is as high as interesting city seems to Alex.
# Alex believes that his trip will be interesting only if he will not use any road twice in a row. That is if Alex came to city $v$ from city $u$, he may choose as the next city in the trip any city connected with $v$ by the road, except for the city $u$.
# Your task is to help Alex plan his city in a way that maximizes total score over all cities he visited. Note that for each city its score is counted at most once, even if Alex been there several times during his trip.
\"\"\"
max_score(input_str): Simple function returning the maximum score Alex can get.
 parse_input(input_str): Takes a string containing the number of cities and roads on one line, the scores of the cities on the next line, the roads on the next lines besides the last (1-indexed, make 0-indexed), and the starting city on the last line. It returns the city scores, the roads as an edge list, and the starting city.
 get_neighbors(edges): Returns a dictionary of the neighbors of each city, defaulting to an empty set.
 get_degrees_and_leaves(neighbors, root): Returns a dictionary of the degrees of each city, and a set of the leaves (excluding the root).
 remove_leaves(scores, neighbors, degrees, leaves, root): Create a 0-initialized defaultdict of total_extra, and an int of max_extra. Pop leaves until it is empty . Update total_extra and max_extra based on the parent's total_extra vs the leaf's score plus its total_extra, whichever is greater. Return max_extra.
  pop_leaf(neighbors, degrees, leaves, root): Pop off a leaf. Set parent to sole neighbor of the leaf and delete the leaf from the neighbors dictionary. Decrement the parent's degree. If the parent is not the root and has degree 1, add it to the leaves. Return the leaf and parent.
 to_output_str(scores, neighbors, root, max_extra): Returns the string of the maximum score Alex can get. If the root isn't in neighbors, return the score of the root. Otherwise, this is the sum of the scores of the cities left in neighbors, plus the returned encountered max_extra.
\"\"\"
(7 lines)

# Translate the following solution plan into the above format:
{solution_start}{solution_text}

TRANSLATE to Parsel.
\"\"\"
"""
```

Figure A.45. Translation prompt for APPS programs

K. Questionnaire

Our questionnaire closely follows that of Huang et al. (2022). We provide a figure with the directions for the accuracy version of the survey in the first image of Fig A.46. We include an example question in the second image. Note that each participant was shown a random 5 questions with their answers in random order. The clarity survey instead asks “For every question below, evaluate how easy it is to understand how the provided steps accomplish the task. Please rank the planned steps for each question from most understandable to least understandable (with 1 as the best and 3 as the worst).” In addition, for the clarity survey, each question text instead said “Rank the following plans based on which is the most understandable (1 = most understandable, 3 = least understandable).”

On the next (and only) survey page, you will be presented with 5 questions.

For each question you will see an everyday task and a plan to accomplish it. Please determine whether the task can be completed using the provided steps. In other words, can the task be decomposed into these steps? Note that simply restating the task does not mean completing it. Please rank the planned steps for each question from most accurate to least accurate (with 1 as the best and 3 as the worst).

There is no correct answer to each question. Please just use your first intuition to determine the answers.

→

Rank the following plans based on how accurately they accomplish the task (1 = most accurate, 3 = least accurate):

1 2 3

Task: Wash teeth
walk to bathroom
walk to sink
find faucet
switch on faucet
find toothbrush
grab toothbrush
find toothpaste
grab toothpaste
put toothpaste on toothbrush
put back toothpaste
grab toothbrush
wash teeth
rinse toothbrush
switch off faucet
grab toothbrush
find toothbrush holder
put toothbrush on toothbrush holder
grab toothbrush
put toothbrush on toothbrush holder
walk to bedroom

Task: Wash teeth
```
task_plan:
   wash_teeth:
      find toothbrush
      plug in sink
      pour water into sink
      pour toothpaste into toothbrush
      scrub teeth with toothbrush
      pour water into toothbrush
      rinse toothbrush
      plug out sink
      drop toothbrush
      find towel
      wipe teeth with towel
      drop towel
```

Task: Wash teeth
find toothbrush
walk to toothbrush
grab toothbrush
open toothbrush
grab toothbrush
find bathroom sink, walk to bathroom sink
grab toothbrush
run to bathroom sink
put toothbrush on bathroom sink
release toothbrush
put back toothbrush
walk to toothbrush

Figure A.46. Screenshot of survey directions and example survey question. In this figure, the first answer was generated by the baseline, the second was the indented Parsel version, and the third was the unindented Parsel version. However, note that the order is randomized for each participant.

L. Human Case Study

```
main(input_string): Takes an input line. First, splits it at newline, and stores the second line in a variable s and the third in t. Splits s in two parts at the asterisk, and checks whether the string t starts with the first part of s and ends with the second part. Returns the string "YES" if that condition is met, otherwise "NO". Also, it should always return "NO" if the length of t is smaller than the sum of the length of the parts of s.
"6 10\ncode*s\ncodeforces" -> "YES"
"6 10\ncodeforces\ncodeforces" -> "YES"
"6 10\ncode*morces\ncodeforces" -> "NO"
"6 10\na*a\na" -> "NO"
"6 10\na*a\naa" -> "NO"
```

Figure A.47. Solution to https://codeforces.com/problemset/problem/1023/A

```
main(input_string): Parses the input and returns the minimum area of the input.
"3\n10 1\n20 2\n30 3" -> 180
"3\n3 1\n2 2\n4 3" -> 21
   parse_input(input_string): Takes the input line and first splits on newline. Ignores the first line, and parses each of the remaining lines as a tuple of two numbers, which give a list L of tuples. Returns L.
   "3\n10 1\n20 2\n30 3" -> [[10, 1], [20, 2], [30, 3]]
      parse_line(l): Splits l on space, converts each element to int, and returns the result of converting the result to a list.
      "10 1" -> [10, 1]
   enumerate_subsets_at_most_k(L, k): Returns all subsets of L with sizes ranging from 0 to k, inclusive.
   [1, 2, 3], 2 -> [[], [1], [2], [3], [1, 2], [1, 3], [2, 3]]
      enumerate_subsets(L, k): recusively enumerates the subsets of size k of the list L. Base cases: if k = 0, returns a list containing the empty list. If k > len(L), returns the empty list. Otherwise, first construct the subsets that contain the first element, then those that do not, and return their concatenation.
      [1, 2, 3], 2 -> [[1, 2], [1, 3], [2, 3]]
   minimum_area(whs): First, calls enumerate_subsets_at_most_k passing whs and half the length of whs rounded down. Returns the minimum result of calling compute_area on the list given by apply_inversions with whs and the subset.
   [[10, 1], [20, 2], [30, 3]] -> 180
   [[3, 1], [2, 2], [4, 3]] -> 21
      enumerate_subsets_at_most_k
      compute_area(whs): takes a list of pairs (width, height). Computes the sum of the widths and the maximum of the heights. Returns the product of those two numbers.
      [[1, 2], [3, 5]] -> 20
      [[10, 1], [20, 2], [30, 3]] -> 180
      apply_inversions(whs, subset): Takes a list of pairs of form (w, h) and a subset of indices to invert. Returns a list where the elements of whs whose index is in the subset are inverted to (h, w), and the others appear as given.
      [[1, 2], [3, 5]], [1] -> [[1, 2], [5, 3]]
      [[1, 2], [3, 5]], [] -> [[1, 2], [3, 5]]
```

Figure A.48. Solution to https://codeforces.com/problemset/problem/529/B

```
main(input): Converts the input to an integer and returns the value of f of n.
"1" -> 1
"2" -> 3
"3" -> 10
   f(n): First pre-computes the Pascal triangle up to n+1 using compute_pascal_triangle. Then, returns the value of dp(n, pascal_triangle)
   1 -> 1
   2 -> 3
   3 -> 10
      compute_pascal_triangle(N): returns a matrix with N + 1 rows where m[i][j] corresponds to "i choose k", i.e., the Pascal triangle. It is computed using dynamic programming: m[i][j] = m[i-1][j] + m[i-1][j-1]. All elements are modulo (10**9 + 7). The i-th row has only i columns.
      2 -> [[1], [1, 1], [1, 2, 1]]
      3 -> [[1], [1, 1], [1, 2, 1], [1, 3, 3, 1]]
      dp(n, pascal_triangle): first creates a list with (n + 1) zeros called L. Then fills it in with the following dynamic programming relation: base case is L[0] = 1; then, L[i] = sum (j in [1, i]) pascal_triangle[i-1][j-1] * L[i - j]. Finally, returns the following answer: sum (k in [1, n]) pascal_triangle[n][k] * L[n - k]. After each of these assignments, take modulo 10**9 + 7 to avoid big numbers.
      1, [[1], [1, 1], [1, 2, 1]] -> 1
      2, [[1], [1, 1], [1, 2, 1]] -> 3
      3, [[1], [1, 1], [1, 2, 1], [1, 3, 3, 1]] -> 10
```

Figure A.49. Solution to https://codeforces.com/problemset/problem/568/B

```
main(input): Reads the input line and counts how many pairs of elements pass the test.
"4 2\n2 3\n1 4\n1 4\n2 1" -> 6
"8 6\n5 6\n5 7\n5 8\n6 2\n2 1\n7 3\n1 3\n1 4" -> 1
   parse_input(input): Splits input as a sequence of lines. Each line is parsed as a list of two
     ↪ space-separated integers. The first line of input contains N and P, and the second to last
     ↪ lines are aggregated in a list L. Returns a list with three values: N, P and L.
   "4 2\n2 3\n1 4\n1 4\n2 1" -> [4, 2, [[2, 3], [1, 4], [1, 4], [2, 1]]]
   count_valid_pairs(L, p): for each distinct pair (i, j) both ranging from 0 to the length of L,
     ↪ counts how many of those pairs have score at least p in L given by compute_pair_score.
   [[2, 3], [1, 4], [1, 4], [2, 1]], 2 -> 6
   [[5, 6], [5, 7], [5, 8], [6, 2], [2, 1], [7, 3], [1, 3], [1, 4]], 6 -> 1
      compute_pair_score(a, b, L): receives two integers, a and b, and a list of pairs L. Returns how
        ↪  many elements of L contain either a + 1 or b + 1.
      1, 2, [[2, 3], [1, 4], [1, 4], [2, 1]] -> 2
      1, 1, [[2, 3], [1, 4], [1, 4], [2, 1]] -> 2
      0, 1, [[2, 3], [1, 4], [1, 4], [2, 1]] -> 4
      4, 5, [[2, 3], [1, 4], [1, 4], [2, 1]] -> 0
```

Figure A.50. Solution to https://codeforces.com/problemset/problem/420/C

```
main(input): parses the input as two space-separated integers, n and m. Return 2 * f(n, m) modulo
  ↪ 10**9 + 7
"2 3" -> 8
"3 2" -> 8
   f(n, m): computes fib(n) + fib(m) - 1
   2, 3 -> 4
      fib(m): computes the m-th fibonacci number modulo 10**9 + 7 using dynamic programming starting
        ↪ with dp[0] = 1 and dp[1] = 1, then dp[n] = (dp[n-1] + dp[n-2]) % (10**9 + 7)
      1 -> 1
      2 -> 2
      3 -> 3
      4 -> 5
      5 -> 8
      6 -> 13
```

Figure A.51. Solution to https://codeforces.com/problemset/problem/1239/A