Parsel: Algorithmic Reasoning with Language Models by Composing Decompositions

Authors - Eric Zelikman, Qian Huang, Gabriel Poesia,
Noah D. Goodman, Nick Haber (Stanford University)

Presented by Alex Mathai, Deepak Surya

Background

- **Intuition**: A framework to enable automatic implementation and validation of complex algorithms with code LLMs.
- **Problem**:
  - Code language models, such as Codex, struggle with generating code for tasks involving multiple steps or complex transformations.
  - Human programmers excel at breaking down complex tasks into manageable and modular parts.
  - Unlike humans, code language models lack the ability to effectively decompose problems and assemble solutions from modular components.
  - As a result, the performance of code language models drops dramatically with an increase in the number of steps or complexity of the task.
- **Solution**: Automatically decompose algorithmic tasks into hierarchical natural language function descriptions and then search over combinations of possible function implementations using tests

Contribution

Parsel is a framework that helps break down, create, and put together complex programs using natural language.

The process has three main steps:

- A language model creates a Parsel program by breaking down a task described in natural language into smaller parts, called function descriptions.
- Each function description, another language model makes small, separate pieces of the program, called implementations.
- Parsel synthesizer combines these pieces together to make the whole program. It does this by testing small groups of implementations to make sure they fit together and meet certain requirements, like giving the right outputs for given inputs.

Method - Overview

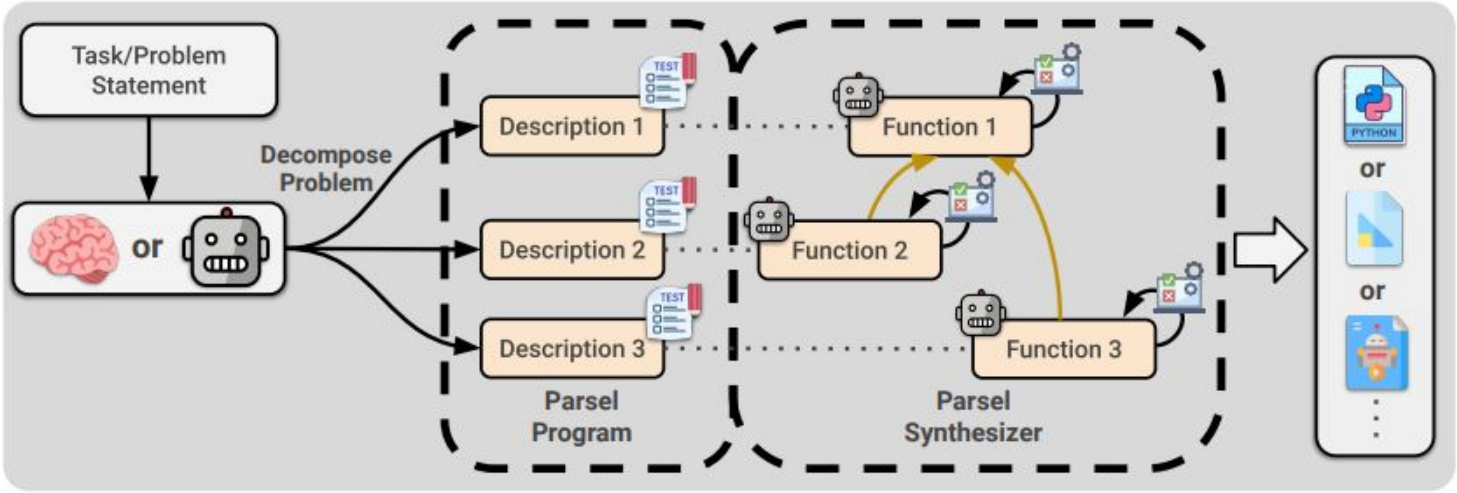

Figure 1: **Parsel overview**. A human or LLM writes a task decomposition (in Parsel), which is split into its strongly-connected components (SCC), and then the Parsel synthesizer uses a code LLM and a constraint solver to implement and compose each SCC. When all functions have constraints and there is no recursion, each function is its own SCC. We provide a more detailed figure in Appendix G.

High Level Methodology

Task → High-level Sketch → Parsel Program → Program

The grand museum has just announced a large exhibit on jewelry from around the world. In the hopes of his potential future prosperity, the world-renowned thief and master criminal Edward Terrenando has decided to attempt the magnum opus of his career in thievery.

Edward is hoping to purloin a large number of jewels from the exhibit at the grand museum. But alas! He must be careful with which jewels to appropriate in order to maximize the total value of jewels stolen.

Edward has k knapsacks of size 1, 2, 3, up to k, and would like to know for each the maximum sum of values of jewels that can be stolen. This way he can properly weigh risk vs. reward when choosing how many jewels to steal. A knapsack of size s can hold items if the sum of sizes of those items is less than or equal to s. If you can figure out the best total value of jewels for each size of knapsack, you can help Edward pull off the heist of the century!
… [continued in appendix]

First, we need to figure out how to calculate the maximum sum of values of jewels for each size of knapsack. To do this, we can use a dynamic programming approach.

We can create a two-dimensional array of size n×k where each row represents the different jewels and each column represents the different sizes of knapsack. The value of each cell in the array will represent the maximum sum of values of jewels that can be stolen for a given knapsack size.

We can then iterate through the array, starting at the beginning. For each row, we can calculate the maximum sum of values of jewels that can be stolen for each knapsack size by taking the maximum of the value of the current cell and the value of the previous cell plus the value of the current jewel.

At the end, we will have an array that contains the maximum sum of values of jewels that can be stolen for each knapsack size. We can then output these values as our answer.

… [continued in appendix]

max_sum_of_jewels(input_str): Returns the maximum sum of values of jewels that can be stolen for each knapsack size.

parse_input(input_str): Takes a string containing the number of jewels and knapsack sizes on the first line, the jewels on the next lines, and returns the number of jewels, the knapsack sizes, and the jewels.

max_sum_of_jewels_for_size(jewels, size): Returns the maximum sum of values of jewels that can be stolen for each knapsack size.

max_sum_of_jewels_for_size_and_jewel(jewels, size, jewel): Returns the maximum sum of values of jewels that can be stolen for each knapsack size.

to_output_str(max_sum_of_jewels_for_size): Returns the string of the maximum sum of values of jewels that can be stolen for each knapsack size.

```
# Takes a string containing the number of jewels and knapsack sizes on the first line, the jewels on the next lines, and returns the number of jewels, the knapsack sizes, and the jewels.
def parse_input(input_str):
    # [Omitted for space.]
    return num_jewels, knapsack_sizes, jewels

# Returns the maximum sum of values of jewels that can be stolen for each knapsack size.
def max_sum_of_jewels_for_size_and_jewel(jewels, size, jewel):
    if size < 0:
        return 0
    elif jewel < 0:
        return 0
    elif jewel == 0:
        return jewels[jewel][1] if jewels[jewel][0] <= size else 0
    elif jewels[jewel][0] > size:
        return max_sum_of_jewels_for_size_and_jewel(jewels, size, jewel-1)
    else:
        return max(max_sum_of_jewels_for_size_and_jewel(jewels, size, jewel-1),
                   max_sum_of_jewels_for_size_and_jewel(jewels,
size-jewels[jewel][0], jewel-1) + jewels[jewel][1])

# Returns the maximum sum of values of jewels that can be stolen for each knapsack size.
def max_sum_of_jewels_for_size(jewels, size):
    result = []
    for s in range(1, size + 1):
        result += [max_sum_of_jewels_for_size_and_jewel(jewels, s, len(jewels) -
1)]
    return result

# Returns the string of the maximum sum of values of jewels that can be stolen for each knapsack size.
def to_output_str(max_sum_of_jewels_for_size):
    return " ".join(map(str, max_sum_of_jewels_for_size))

# Returns the maximum sum of values of jewels that can be stolen for each knapsack size.
def max_sum_of_jewels(input_str):
    number_of_jewels, knapsack_sizes, jewels = parse_input(input_str)
    return to_output_str(max_sum_of_jewels_for_size(jewels, knapsack_sizes))
```

Parsel Program Example

```
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
```

⇒

```
# 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 ii: Parsel to Python

Parsel Programs contain one/more of (A) Description - NL description and function signature with arguments, (B) Reference to a function (either recursion or otherwise) and (C) A constraint like a test case.

Post-Order generation of programs

When implementing the parent node, we first provide the LLM with the description & signature of each helper child function.

Then we provide the description of the parent function and tell the LLM to explicitly use the children helper functions in the implementation.

J 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.16: 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.17: Codex Prompt for an example merge node

Composing the entire/complete program

1. After generating the implementations for each sub-function, we need to compose these functions for the complete implementation.
    a. Say there are N helper functions, we ask the LLM to generate Top-K implementations for each function. Hence K^N combinations - too much.
    b. Instead partition the function dependency graph into strongly connected components (SCC) and evaluate all combinations within a SCC.
        i. So for example if F, G an H form a SCC, Evaluate K^3 solutions and pick a solution that satisfies all constraints.
        ii. Time complexity is $O(K^{MAX\_SCC\_SIZE})$ where MAX_SCC_SIZE < N.
2. If every function has its own test case and there is no recursion/cycle, then each function along with its test case is in its own SCC.
3. If there is no test case for a function, it's added to the parent function's SCC.
4. If the parent function does not have a test case, ask the LLM to create a test case.

Evaluation

- Parsel is tested for generating programs across different tasks:
  1. **Competitive coding:** It's a standard benchmark for evaluating code generation tools and heavily relies on breaking down tasks into smaller parts.
  2. **Robotic task planning:** Involves planning and executing tasks in a robotic environment.
  3. **A standard coding benchmark**

- These tasks represent different forms of algorithmic reasoning, each with varying levels of complexity and generality.
- Evaluating Parsel across these tasks helps assess its versatility and effectiveness as a framework for algorithmic reasoning.

Evaluation Setup

Setup:
-Tested Parsel on competition-level problems from the APPS dataset.
- Utilized GPT-3 to propose high-level solutions to the problems.
- Translated these solutions into Parsel code using Codex.
- Synthesized Parsel code and evaluated implementations to measure pass rates.

Metrics and Comparison:
- Pass rate reported as "pass@n × k" where n is the number of Parsel programs generated and k is the number of implementations per function.
- Compared Parsel's performance to direct solutions from Codex and AlphaCode.
- Visualized pass rates, showing Parsel's substantial improvement over prior methods.

Scalability and Computational Cost:
- Parsel's approach allows for a vast number of candidate programs due to "mix-and-match" function implementations.
- While the generation cost for components is linear, the number of distinct candidate programs grows exponentially with k.
- Generating each program may require significant computational resources, but testing complete programs is relatively less costly.
- Performance of Parsel improves log-linearly with the number of evaluations, justifying computational expenditure for evaluating all possible programs.

Evaluation

Evaluation on HumanEval:

- Parsel was tested on HumanEval, which consists of simpler problems than competitive coding.
- This allowed for evaluating the performance at pass@1 by automatically generating tests.

Improvement with GPT-4 and Parsel:

- GPT-4 combined with Parsel showed a significant improvement in pass@1 performance, from 67% to 85%.
- GPT-4 was able to translate natural language plans into Parsel format effectively, surpassing GPT-4 alone on HumanEval.

Pass Rate Analysis:

- Pass rates with Parsel improved regardless of whether filtered by generated tests.
- Sampling until three Parsel programs had one implementation passing a generated test resulted in an 85.1% pass rate.
- Parsel outperformed GPT-4 alone both in CodeT score and pass@any evaluations.

Effect of Parsel on Pass Rates:

- Allowing up to 8 Parsel programs with 8 implementations each increased the pass rate to 91.5% compared to directly generating programs.
- When allowing up to 128 programs, pass rate improved from 84.1% to 93.9% with Parsel.

Solving Complex Problems:

- Parsel helped solve problems with more functions that GPT-4 alone couldn't solve.
- Problems solved with Parsel required more functions on average compared to those solved by GPT-4 alone.
- Some problems remained unsolved, and Parsel enabled solving problems that GPT-4 alone couldn't tackle, showcasing its efficacy in handling complex tasks.

Evaluation

Comparison to Human Expert:

- An author experienced in competitive programming successfully solved 5 out of 10 randomly-selected Codeforces problems from the APPS dataset within 6 hours.
- In comparison, GPT-3 generated Parsel solutions had a success rate of 2 out of 10 problems with 8 attempts per problem, albeit in a much shorter time frame.
- This suggests that while Parsel can effectively generate complete correct solutions to hard problems, improving the ability of models to decompose tasks into Parsel programs is crucial for future enhancements.

Participant Feedback:

- The participant found some aspects of working with Parsel counterintuitive.
- Writing additional tests was more effective for ensuring a working solution than clarifying function descriptions.
- Concerns about changing the description of a child function breaking its parent were rare in practice.

Chained Components Experiment:

- Replicated Chen et al.'s experiment highlighting limitations of language models in producing code that requires simple composition of building blocks.
- Visualized Parsel's performance in solving problems as the number of required components grows.
- Even interchanging the parts of two complete programs solved more problems than Codex with 16 programs.
- This illustrates the advantage of Parsel in supporting increasingly complex and abstracted programs as language models improve.

Robotic Planning in VirtualHome:

Robotic Planning in VirtualHome:

- Utilizing VirtualHome, a simulated environment with interactive elements, researchers delved into household task automation.
- Tasks within VirtualHome adhere to strict grammatical structures, requiring intricate decomposition.
- Parsel, a tool, was employed to generate Python programs capable of devising action plans resembling natural language instructions.
- These plans were then translated into formal instructions for VirtualHome and assessed for feasibility.

Comparison of Robotic Plans:

- The clarity and accuracy of Parsel-generated plans were compared to directly created plans, with Parsel's plans emerging as superior.
- Human studies were essential for assessing plan correctness due to the subjective nature of task execution.

Human Study Details:

- Two surveys, each involving 20 participants on Prolific, generated 100 rankings per survey.
- Participants ranked solutions based on accuracy and comprehensibility.
- Two versions of Parsel solutions were presented: one with indented function names alongside step-by-step solutions, and another with only step-by-step output.
- Both Parsel solutions were compared against a baseline solution derived from prior research.

Scaling Performance:

- Solving complex problems became increasingly likely with the logarithm of the number of sampled Parsel programs.
- Human evaluations favored Parsel solutions over the baseline, with standard Parsel solutions outperforming the baseline in 69% of comparisons.
- In terms of clarity, standard Parsel solutions were preferred over the baseline by 70%, while indented Parsel solutions were favored 50% more often.
- No significant difference was observed between indented and non-indented Parsel solutions in terms of accuracy or clarity.

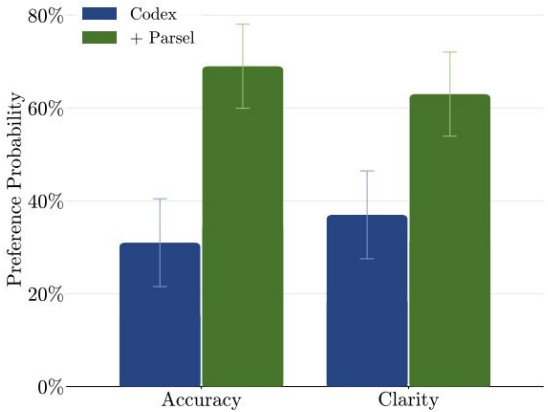

Figure 7: **Robotic plan comparison.**

Scaling Performance

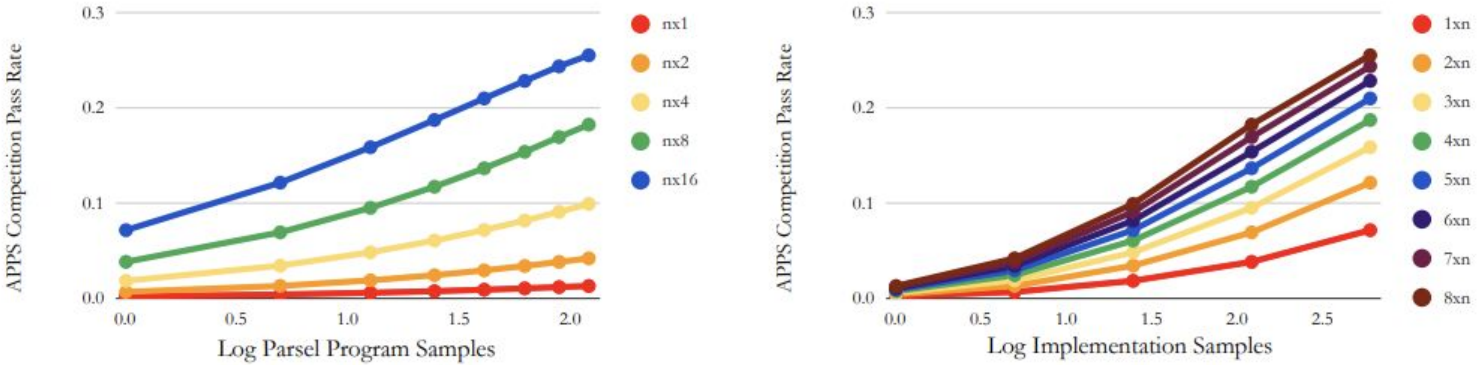

Figure 8: **Scaling performance.** The probability of passing an APPS competition-level problem increases quadratically with respect to the log of the number of Parsel programs sampled (left) and the number of implementations sampled (right).

Conclusion

For Programmers:
- Parsel offers a robust framework for code generation, eliminating the need to manually evaluate the underlying code.

For Students:
- Provides a platform for teaching algorithmic reasoning with less focus on syntax and more emphasis on problem-solving, akin to a mathematics curriculum.

For Language Models:
- Facilitates hierarchical task decomposition, enabling language models to break down complex tasks into manageable parts.

Extensions and Future Work:

- Parsel's potential extensions include integrating automatic test case generation, reranking solutions based on model-judged quality, implementing more intelligent recursive decomposition, bootstrapping increasingly complex programs, and generating language model prompts as part of a target language.

- Additionally, Parsel may find utility in theorem proving, bridging the gap between reasoning and execution in complex, hierarchical reasoning tasks for language models.

Questions

- Given that Parsel is completely automated, what would happen if we trained/fine-tuned a LLM on a synthetically created dataset with Parsel ?
- What more could we do to enforce logic training into LLMs ?
- This paper is really a trade-off between querying the LLM vs invoking the execution environment
    - Would this method be practical in cases where invoking the execution environment is expensive ?