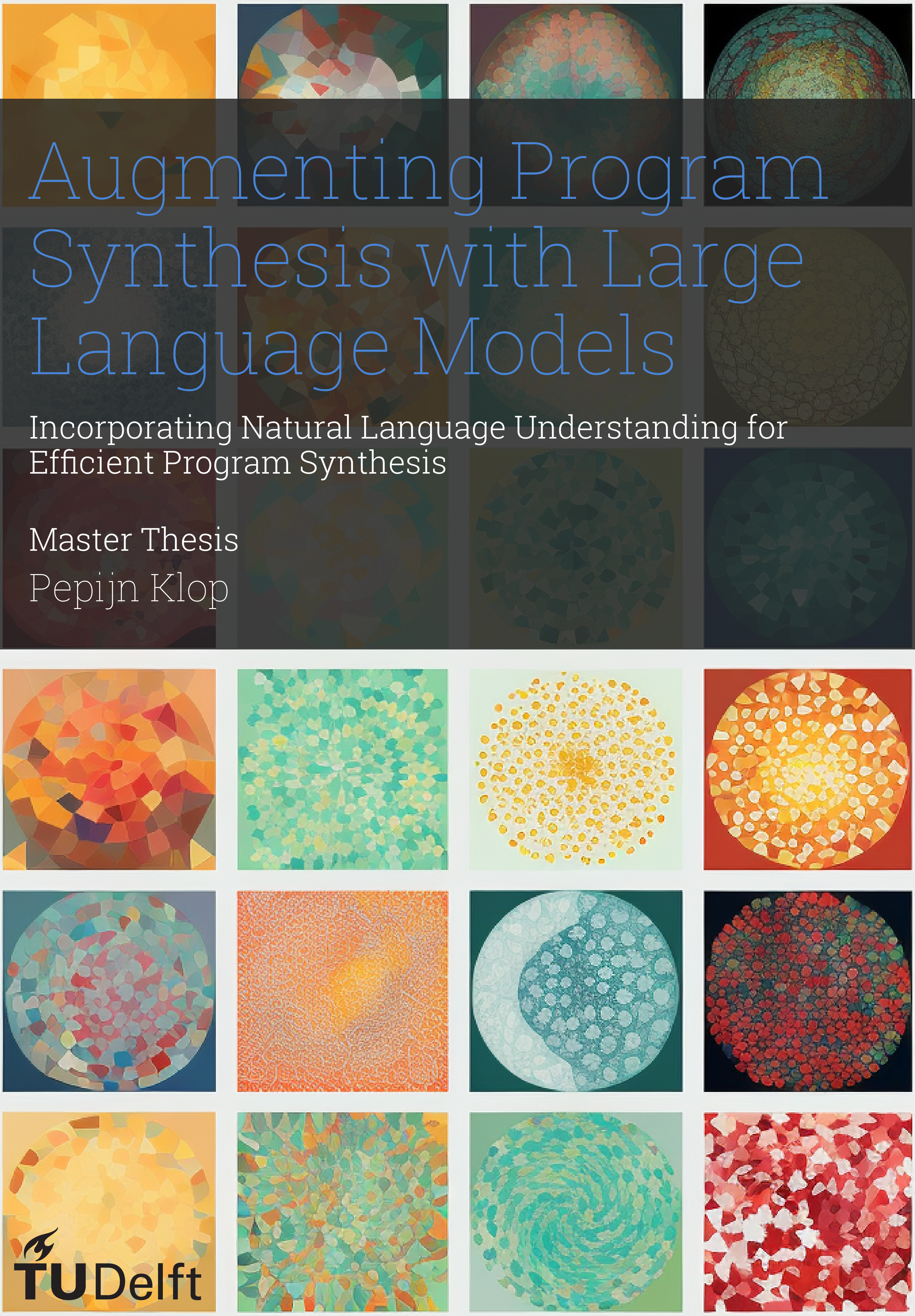

Augmenting Program Synthesis with Large Language Models

Incorporating Natural Language Understanding for Efficient Program Synthesis

Master Thesis

Pepijn Klop

TU Delft

Augmenting Program Synthesis with Large Language Models

Incorporating Natural Language Understanding for Efficient Program Synthesis

by

Pepijn Klop

Student Name	Student Number
Pepijn Klop	4684311

Instructor:	S. Dumančić
External Expert:	G. Verbruggen (Microsoft)
Project Duration:	November, 2022 - June, 2023
Faculty:	Faculty of Electrical Engineering, Mathematics and Computer Science, Delft

Cover: Generated with Midjourney (prompt: neural networks flat art, simple, minimalistic art, large language models, machine, patterns)

Summary

This research introduces a Language Model Augmented Program Synthesis (LMAPS) workflow to enhance traditional Programming by Example (PBE). PBE is a method to automatically generate a program that satisfies a specification that consists of a set of input-output examples. These program specifications are often defined by a few examples, which can lead to multiple programs that satisfy the given examples. In addition, PBE synthesisers have to explore a huge inefficient search space to solve these problems. The LMAPS workflow incorporates three components to overcome these limitations of PBE by using the language understanding capabilities of Large Language Models (LLM). LLMs can assist in generating a well-defined specification to mitigate the ambiguity issue inherent in PBE. The core component of LMAPS leverages the capabilities of LLMs to generate programs. These programs can be decomposed into building blocks to create a concise grammar for an inductive program synthesiser. This optimized grammar makes it able to synthesise correct programs at lower depths, make the workflow more efficient. LLMs can also aid in understanding the automatically generated programs, as these programs can be hard to interpret by humans. We compare LMAPS to a traditional PBE workflow in the task of synthesising regular expressions across four data sets. The results demonstrated that LMAPS can significantly reduce the search space for program synthesis and achieve up to 40% higher accuracy than PBE-only systems. Our research indicates that integrating LLMs into a typical PBE workflow shows significant improvements because of their combined strengths, resulting in a more accurate, efficient, and human-aligned workflow.

Contents

Nomenclature

Abbreviations

Abbreviation	Definition
AI	Artificial Intelligence
APE	Automatic Prompt Engineer
DFA	Deterministic Finite Automata
DSL	Domain Specific Language
GPT	Generative Pre-trained Transformers
IO	Input-Output
LMAPS	Language Model Augmented Program Synthesis
LLM	Large Language Model
NLP	Natural Language Processing
PBE	Programming By Example
Regex	Regular Expression
t-SNE	t-Distributed Stochastic Neighbor Embedding

1 Introduction

In an era where machines are increasingly proficient at interpreting human language, our research introduces a method that harnesses the synergy between Large Language Models (LLM) and program synthesis. This approach is designed to generate programs that meet the given specification and are aligned with human intentions.

Program synthesis is the field of computer science that aims to generate computer programs from high-level specifications automatically. One approach to program synthesis is programming by example (PBE), which offers a simplified way of creating computer programs by illustrating tasks using input-output examples. For instance, imagine a programmer needs a program capable of dividing a person's full name into their first and last names. A program synthesiser can create such a program by providing it with input-output examples that illustrate how to split the full names. This can drastically reduce the time and effort involved in software development, optimisation, and debugging. However, PBE systems also face various challenges and limitations as 'ambiguity', 'generalizability' and 'computational efficiency'.

Ambiguity: PBE inherently suffers from under-definition because the problems are outlined using a limited set of examples. This limited context can lead to ambiguity, resulting in various plausible interpretations. Consequently, the synthesised program may not necessarily align with the intended functionality. For most tasks is impractical to create an exhaustive list of input-output (IO) examples, particularly one that sufficiently represents all potential edge cases. The more IO examples a system gets as input, the better it can understand the general rules or pattern to be applied, thereby producing more accurate programs.

Generalizability: PBE frequently encounters the dual issues of overfitting and underfitting. Overfitting is observed when the synthesised program excessively aligns with the provided examples, hindering its capacity to generalise to unseen cases. On the contrary, underfitting arises when the produced program is overly generalised, failing to accurately encapsulate the nuanced characteristics of the examples. Both these issues occur due to the fact that there are multiple valid interpretations and synthesised programs for a given set of examples. Note that ambiguity and generalizability look very similar. The main difference between the two is that ambiguity refers to programs with wrong behaviour, whereas overfitting and underfitting can be seen as finding the right balance between correct programs based on the context.

Computational Efficiency: The task of synthesising a program that perfectly aligns with all the given examples can be computationally intensive, especially for complex and large tasks. The PBE system must search through many potential programs to find the one that accurately matches the user its intention. This search space grows exponentially with the size of the programs, making it infeasible to explicitly enumerate and check every possible program. This computational inefficiency may limit the practical applicability of PBE, making it slow or even unusable in certain applications.

Explainability: Programs synthesised through PBE often lack clarity, making understanding, debugging, and maintaining these generated programs particularly challenging. PBE systems are designed to create a program that correctly handles the provided examples, not necessarily a program that is easy to comprehend. In some cases, the generated program might use complex logic or take an approach that a human wouldn't typically use. Since users only provide input-output pairs, they may not fully understand the underlying logic of the generated code. This can make it challenging to predict how the program will behave in situations not covered by the examples. If something goes wrong or if changes need to be made, it can be tough to debug or modify the generated code due to its complexity and the user's limited understanding of its logic.

Limited Modality: In traditional PBE workflows, only one type of information is used, which means that potential information is not being fully utilised. Using different types of information can provide additional insights into a problem, making it easier for the user to control the program synthesis process. Showing user intent can be challenging when relying solely on input-output examples. By incorporating different modalities, users can make a richer specification of the problem.

The potential of integrating LLMs into PBE systems offers a promising solution to address these limitations. By leveraging their capacity to generate human-like text, large language models can assist in generating more diverse IO examples, creating programs with multiple modalities and explaining the program afterwards. In this thesis, we harness the power of LLMs to address various tasks related to program synthesis. LLMs can assist in creating more effective program specifications by generating IO example suggestions based on both a natural language description and the already existing IO examples. Initially, the LLM itself tries to synthesise a program using these specifications. If the LLM cannot find a correct program, parts of the generated programs are utilised to assist the inductive program synthesiser. Once a program has been generated, the LLM can explain the workings of this program to help users understand it better.

Let's consider an example to illustrate these limitations of programming by example and show how LLMs can help to overcome these limitations. Imagine that we want a program to check whether an email address is valid by using regular expression pattern matching. First, we need to create a list of IO examples, as seen in Table 1.1. These IO examples are then given as input to a program synthesis engine.

Input	**Output**
`john.doe@example.com`	✓
`jane_doe@example.com`	✓
`example.com`	×

Table 1.1: IO examples for a valid email address check

A program that checks if the input ends with ’@example.com’ or a program that checks if the input starts with a ’j’ would both satisfy all examples. However, both programs do not capture the complete rules for a valid email address. This happens due to the ambiguous nature of the examples. An LLM can help to generate more diverse examples and assists in finding edge cases that the user did not think of.

Depending on the interpretation of the examples, the program could potentially overfit or underfit. For example, an extremely overfitted program would only check if the input is exactly "john.doe@example.com" or "jane_doe@example.com", which perfectly matches the examples but fails to generalize to all other valid email addresses. An underfitted program could only check if the input contains an ’@’ symbol. This can be solved by adding a negative example which contains the ’@’ symbol. LLMs could potentially overcome this limitation because they can understand a task description as well as the IO examples. For example, if a user only wants email addresses ending with ’.com’, this constraint can be clearly communicated in the description of the task. This helps to strike the right balance between overfitting and underfitting.

Exploring all possible programs that match the given examples can be computationally intensive. A program that matches email addresses contains different parts (username, "@" symbol, domain name, domain extension), and each part can have many possible variations. The search space can quickly become large and complex. The search space can be greatly reduced by using partially correct LLM solutions as a starting point which is further explained in Section 4.

In the current example, the user is only providing input-output pairs. They do not have a way to express that the email address should not start with a special character or end with a certain domain. The inability to supplement the examples with more nuanced multimodal instructions makes the process less human-aligned. An LLM can ’understand’ text-based modalities, making it useful to combine text with IO examples to guide the program synthesis process.

Synthesised programs can be hard to understand as they are automatically generates and may contain illogical elements from a user perspective. For instance, the regular expression `"[A-Za-z0-9._+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}"` is a correct program that matches email addresses, but it might be challenging for a user to comprehend it. An LLM can explain the program to help the user understand the generated program.

This research explores the use of LLMs to tackle the challenges of program synthesis, with a specific focus on synthesizing regular expressions. Specifically aiming to provide valuable insights into the capabilities of LLMs in program synthesis workflows. We will investigate the benefits of using LLMs in program synthesis workflows, analyze their performance compared to traditional approaches, and assess their effectiveness in addressing the challenges inherent in programming by example. We explore how LLMs can be used to augment the program synthesis workflow. By doing so, we hope to contribute to the broader understanding of the potential of LLMs in program synthesis and their practical applications in the software development process.

LLMs can be used in programming synthesis tasks that satisfy the following three requirements. First, it is important that the task its core components and constraints can be effectively communicated using textual descriptions. LLMs excel at understanding and generating natural language, making them well-suited for problems that can be framed in this manner. Sometimes it is needed to convert the data to a textual format. For example, tabular data can often be converted to a textual representation. Second, it should be feasible to give a textual representation of the problem as the problem must be comprehensively described using natural language. This allows the LLM to understand the problem's intricacies, objectives, and constraints, enabling it to generate accurate and relevant solutions. Third, it should be feasible to give a small subset of positive and negative examples. Providing a limited set of positive and negative examples allows the LLM and program synthesizer to distinguish between correct and wrong solutions. This helps the model to generalize and generate new solutions that adhere to the given specifications.

Regular expressions are often used to evaluate PBE synthesis due to a combination of characteristics that test the capabilities of these systems in various ways. One of the critical features of regular expressions is their varying complexity, from straightforward to highly intricate. This allows us to assess the full spectrum of a PBE system's capabilities. Secondly, it is relatively easy to check if the regular expression matches all IO examples. Regular expressions are widely recognized in almost all programming languages. This ensures that evaluating models can often be done across different PBE systems. The ability of a PBE system to effectively synthesize regular expressions gives a strong indication of its practical utility. Lastly, equivalent regular expressions can be written in many ways. This tests the capacity of a PBE system to generate a variety of correct solutions, mirroring the reality that multiple valid approaches often exist for a given problem. Thus, regular expressions present a robust, flexible, and contextually relevant way of benchmarking PBE synthesis. They offer a comprehensive test of a system its performance and provide an indication of its likely performance in tackling practical programming tasks.
This research proposes a workflow called Language Model Augmented Program Synthesis (LMAPS) that combines Large Language Models with PBE to enhance the program synthesis process. The primary aim of the research is to mitigate the limitations of PBE-only systems, such as ambiguity, overfitting, underfitting, efficiency, explainability, and single modality issues. LMAPS combines three LLM components, "Specification Suggestion", "LLM Program Synthesis", and "Explain Program Step", with a traditional PBE workflow. Each component

addressing a specific limitations of the PBE-only approach. The 'LLM Program Synthesis' is the core component of the LMAPS workflow, which uses the capabilities of LLMs to generate programs. These programs, if initially incorrect, can be repaired with an inductive program synthesis engine. These partially correct programs can be used to reduce the program synthesis search space immensely. Incorrect programs generated by the LLM can be dissected into building blocks, contributing to the grammar of the program synthesis engine and boosting its efficiency. In addition to the efficiency gains, the LMAPS workflow achieves significantly higher accuracy than LLM-only and PBE-only approaches.

2
Background

In this research thesis, we explore a novel approach called LMAPS. To understand the methodology and rationale behind our approach, it is crucial to have a basic understanding of several key topics. This background section provides an overview of the fundamental concepts required to comprehend the subsequent chapters.

2.1. Programming By Example

Programming By Example (PBE) is a powerful approach in program synthesis that aims to generate computer programs from user-provided examples automatically. The core idea behind PBE is to infer the intended program behaviour based on input-output (IO) examples rather than requiring the user to specify the desired algorithm explicitly. PBE systems can analyze these examples, identify patterns, and synthesize a program that captures the underlying logic.

Input-output examples are essential in PBE because they serve as a simple and intuitive way to communicate the desired behaviour of a program to the PBE system. By providing specific examples of inputs and their corresponding expected outputs, users can effectively convey the intended pattern or transformation without having to write explicit code. PBE systems can generalize from the given examples, automatically synthesize a program that adheres to the desired behaviour, and ultimately save time and effort for users.

Let's say we have a set of input-output pairs, $E = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$, where each (x, y) pair denotes an example provided by the user, such that x is the input, and y is the output. The task of PBE is to infer a program P from these examples, such that for all $(x, y) \in E$, when we execute the program P with input x, it produces output y. This can be represented by the equation:

$$\forall (x, y) \in E, P(x) = y$$

The real challenge lies in generating the most appropriate program P given the examples, considering there could be multiple programs that satisfy the input-output examples.

To generate the program, a PBE system generally follows a search-based approach. This search is often performed in the space of all possible programs, represented as $\mathcal{P}$, that can be generated using the constructs of a given grammar. The search process can be highly sophisticated and could involve constraint solving, machine learning, and probabilistic reasoning. The goal is to find a program $P^* \in \mathcal{P}$ such that it is consistent with all provided examples and has the user its intended behaviour. It's important to note that PBE often relies on user interaction to help refine and correct the inferred program. Users may need to provide additional examples, validate the correctness of inferred programs, or directly guide the search process.

The critical part of this process is the grammar that guides the conversion of these examples into the actual program. The grammar is the set of rules that describe how expressions of a particular language can be formed. These rules provide a formal definition of the syntax of the language. It's essentially the "vocabulary" that the system uses to express programs. This grammar guides the system on the kind of programs it can generate. The PBE engine uses the grammar, in conjunction with the provided examples, to search for a valid program. The grammar can for example define how to generate Python code or regular expressions.

The task of synthesizing a program based on examples entails exploring a vast search space that consists of all potential programs that can be constructed using the given grammar. The complexity or depth of a program directly influences the size of the search space. For instance, in the scenario where the maximum depth is limited to one, the search space encompasses all possible programs of depth one. As the depth is incremented, the search space expands to include all feasible programs up to the given depth, thus leading to a considerable increase in the search space size. The size of the grammar also impacts the size of the search space. A compact grammar consisting of only a few elements yields a relatively smaller search space compared to a more expansive grammar with a diverse set of operators.

To illustrate these concepts, consider the task of regular expression synthesis, where the objective is to synthesize a regular expression to match a specified string pattern. Suppose we intend to synthesize a regex that identifies strings representing positive integers or, in other words, sequences composed of one or more digits. For this task, the grammar might consist of elements like individual digits, the dot-star operator '`.*`' matching any character zero or more times, the '+' operator that matches one or more instances of the preceding character or group, and character classes such as `[0-9]` denoting any digit.

A basic regex program which can be constructed with a depth of one is `[0-9]`. This pattern matches a string containing a singular digit, for example, '5', but not multi-digit strings like '12' or '345'. The depth limit of one means that the program synthesiser can only use one element from the grammar, in this case `[0-9]`. A more sophisticated regex of depth two could be `[0-9]+`. This pattern matches one or more digits, therefore identifying any positive integer.

Introducing new operators to our grammar, such as `{n}`, which matches 'n' occurrences of the preceding character or group, significantly enlarges the search space. We can now construct expressions like `[0-9]{3}` to match exactly three digits and `[0-9]{3}+` to match one or more sequences of three digits.

This example scenario demonstrates the growth of the search space both with the depth of programs and the number of elements in the grammar. One of the principal challenges in PBE lies in efficiently traversing this space to identify the most appropriate program for a given set of input-output examples.

The are numerous benefits of using PBE to generate programs. For one, it allows for intuitive communication, as IO examples enable users to show their intentions without the necessity of writing explicit code. This makes it user-friendly even for non-programmers to create customized solutions tailored to their specific needs. Another advantage is that it saves time and effort since only needing to provide IO examples simplifies the program creation process, thus reducing the time and effort required by users. Finally, PBE engines are capable of synthesising complex programs that solve a certain specification, which is especially useful when having a large list of strict requirements for the program. PBE engines also reduce the likelihood of errors and inconsistencies, as the synthesized code is aligned with the provided examples.

2.2. Large language models

Large Language Models (LLMs) have recently gained significant attention due to their remarkable performance in various natural language processing tasks. An LLM is a transformer-based neural network architecture designed to understand and generate human-like text by learning from vast amounts of textual data. These models are trained on diverse sources, allowing them to capture intricate patterns, relationships, and structures within the language. Notable LLMs at the time of writing include GPT-4 by OpenAI [19], LLaMa by Meta [23], and PaLM 2 by Google [2].

While there exist specialized LLMs dedicated to code synthesis, such as OpenAI's Codex model, it has been reported that general-purpose LLMs like GPT-4 exhibit even better coding abilities [20]. This demonstrates the incredible adaptability and competence of these models in a wide range of tasks. LLMs can perform tasks from simple tweet sentiment analysis to more complex tasks like language translation and code generation. LLMs are particularly useful in problems with a large search space. LLMs have 'learned' to detect patterns in the huge amount of data they are trained on, making them useful for identifying patterns or trends within extensive search spaces.

A transformer is composed of several layers, each of which consists of self-attention mechanisms and feed-forward neural networks. However, the inner workings of the LLM architecture are out of the scope of this research.
Tokens represent the smallest units of input that the model can understand and process. When given an input prompt, it is converted to its corresponding numerical token representation. This representation is then passed through the model its layers, where it is manipulated and transformed to generate output.
A token is typically a word or a part of a word. The precise nature of a token, however, can vary based on the tokenization strategy used. For instance, some models treat words as whole tokens (e.g., "apple" would be a single token), while others use subword tokenizers which break words into smaller units (e.g., "ap" and "ple" could be separate tokens). This latter approach can help models handle rare and out-of-vocabulary words more effectively. The

choice of tokenization strategy and the number of tokens a model can process simultaneously play significant roles in the model's performance, capabilities, and computational requirements.

LLMs are autoregressive models which essentially only predict the next token given its preceding context. Given an input sequence of tokens, $[x_1, x_2, \ldots, x_n]$, the model computes the likelihood of the next token, x_{n+1}. The objective of the model is to maximize this likelihood.

$$\max P(x_{n+1}|x_1, x_2, \ldots, x_n) \tag{2.1}$$

The concept of a 'prompt' in the context of LLMs is the sequence of tokens $[x_1, x_2, \ldots, x_n]$ that are fed as input to the model. For instance, in a text generation task, a prompt could be the start of a sentence, and the LLM's job would be to predict the rest of the sentence.
Given a prompt, e.g. 'What is the capital of France?', the LLM calculates the probability distribution over all possible next tokens in its vocabulary V, and selects the next token x_{n+1} based on these probabilities. In this case, there is clearly one correct answer, which is 'Paris'. However, for less factual questions, the simple method of always selecting the token with the highest probability may not yield diverse or realistic output. Therefore, we need a way to control the randomness of the token selection process, and this is where certain parameters come into play.

2.2.1. Large Language Model parameters

The two main parameters to control the randomness and diversity of the output generated by LLMs are the 'temperature' and 'top-p' parameters. These parameters allow us to tweak the sampling process of the next token based on the output probability distribution. By adjusting these parameters, we can influence the trade-off between diversity and quality in the model its output and thereby adjust its "creativity". A higher temperature or top-p value typically leads to more diverse but potentially less coherent output, which is useful in creative writing tasks. In contrast, lower temperature and top-p values result in more coherent and predictable text, something that is useful in tasks like fact retrieval or code generation. The optimal setting for these parameters can depend on the specific use case and desired outcome. These parameters must be set carefully to get the desired balance between quality and diversity in the output.
The temperature parameter uses an adjusted softmax function, which converts logits (raw model output) into probabilities. The softmax function with a temperature parameter is defined as

$$\text{softmax}(z_i) = \frac{e^{z_i/T}}{\sum_j e^{z_j/T}} \tag{2.2}$$

where z_i are the logits and T is the temperature. When $T = 1$, the function becomes the standard softmax function. For $T < 1$, the model will output the token with the highest probability more often. When $T > 1$, the output becomes more uniform, hence adding more diversity in the choice of the next token.

Top-p sampling offers another way to sample from the model's output distribution. Instead of always picking the most probable next token or sampling from the full distribution, we sample from the smallest set of tokens S whose cumulative probability exceeds a threshold p. This is the algorithm to find the top-p tokens:

1. Sort the probabilities in descending order
2. Keep adding tokens to the set S while the cumulative probability of the set is lower than the specified p.
3. Then sample the next token from the distribution over S

2.2.2. Prompting techniques

Prompting is the way to communicate with LLMs to get the model to generate the desired output. The quality and specificity of the prompt significantly influence the model its subsequent text generation. Prompt engineering is the practice of carefully crafting and optimizing prompts with the goal of generating useful and accurate outputs. Optimized prompts can guide the model towards a specific type of output or to elicit a certain kind of knowledge. This is particularly relevant for factual or technical questions, where precision and detail are essential. A slight change in the formulation of the prompt can make a large difference in the the quality of the response.

Different prompting techniques can maximize the model its performance for a given task. The two most used prompting techniques are zero-shot and few-shot prompting.

Zero-shot prompting requires the model to perform a task without any prior explicit examples. The prompt should precisely describe the task that the model is required to perform. The effectiveness of zero-shot prompting is highly dependent on the specific structure of the prompt and the data the LLM was trained on.
For instance, to generate a regular expression that matches strings ending with "dog", a zero-shot prompt may look like:

```
User: Generate a regular expression that matches any string ending with 'dog'.
Model: '.*dog$'
```

Prompt 2.1: Example of a zero-shot prompt, where 'User' refers to the user input prompt and 'Model' refers to the output generated by an LLM.

Few-shot prompting provides the model with a few examples of the task before the task itself. This approach aims to offer the model a clearer understanding of the specific task, by showing the desired output for similar inputs.
For instance, a few-shot prompt for the same regular expression generation task could be:

```
User:
Task: "Generate a regular expression that matches any string ending with 'cat'."
Output: '.*cat$'

Task: "Generate a regular expression that matches any string ending with 'ball'."
Output: '.*ball$'

Task: "Generate a regular expression that matches any string ending with 'dog'."
Output: <<let LLM complete>>
Model: '.*dog$'
```

Prompt 2.2: Example of a few-shot prompt, which includes examples of similar problems with their corresponding solution.

Few-shot prompting most of time achieves a higher accuracy than zero-shot prompts as it gives the model more context. However, it few-shot prompting requires high-quality example data for each task. Both prompting techniques are used to guide the model to generate the intended output. This process often involves trying different prompts, seeing how the model responds, and then making adjustments to improve the outcomes.

2.2.3. LLM language understanding

LLMs are designed to process and generate human language in a way that often appears to demonstrate understanding. However, the type of 'understanding' these models exhibit is fundamentally different from human understanding. These models are based on recognizing patterns in the data they were trained on. For example, if the model was trained on a lot of books, it might recognize that the phrase "Once upon a time" is often followed by the beginning of a story. The "understanding" exhibited by current generation LLMs is a form of pattern recognition based on statistical associations in the data they were trained on. Unlike humans, these models don't understand the meaning of the words or sentences they process. For instance, if a model generates a reasonable response to the statement "The sun is cold," it's not because the model understands that this statement contradicts the fact that the sun is actually hot. It's because the model has learned from its training data that the phrase "The sun is cold" is often followed by responses indicating disagreement or correction.

2.3. Regular Expressions

Regular expressions (regex) are a powerful and versatile tool used in text processing, enabling efficient pattern matching and manipulation of strings. Regular expressions are used in various fields, including data processing, programming, and natural language processing. We evaluate our LMAPS methodology on multiple regular expression data sets.

A regular expression is a sequence of characters that specifies a search pattern in text, commonly used in "find" or "replace" operations on strings and input validation. Regex patterns consist of a series of atoms, which are the smallest matching units within a pattern. Atoms can be literals, metacharacters, and groups of characters enclosed in parentheses and square brackets. Metacharacters play an essential role in shaping regex patterns as they define all the functionalities of the regular expression.

From a mathematical standpoint, a regular expression is a way to describe a language (set of strings). Regular expressions use operations that are based on the algebraic structure known as Kleene algebra. Examples of regular expressions can be seen in Table 2.1

Regular Expression	Description
`a+`	Matches one or more consecutive 'a' characters.
`(ab|cd)`	Matches either 'ab' or 'cd'.
`a(b|c)d`	Matches either 'abd' or 'acd'.
`[0-9]`	Matches a single digit.
`[A-Z]2`	Matches two uppercase letters.

Table 2.1: Examples of regular expressions

Regular expressions are often perceived as difficult to read and write due to their concise and cryptic syntax. The compact representation can be challenging for those unfamiliar with the regex syntax. Furthermore, the flexibility in formulating regex patterns allows for multiple representations of equivalent patterns, which can lead to confusion when comparing different expressions. As a result, many people can benefit from tools and techniques that can simplify the process of generating and validating regex, and make regex more accessible to a broader audience. This means that regex synthesis could benefit from the "Explain Program" component of the LMAPS workflow.

2.4. Multimodal program synthesis

Multimodal program synthesis is a method of automatically generating computer programs by combining different types of user inputs or specifications, such as natural language descriptions and input-output examples. Multimodal program synthesis often works better than NLP-based or example-based approaches because it leverages the strengths of both modalities while mitigating their individual weaknesses.

NLP-based synthesis relies solely on natural language descriptions, which are often ambiguous and can lead to multiple interpretations. Although modern natural language processing techniques can capture the meaning to some extent, they struggle to generate precise and accurate code when faced with ambiguous descriptions.

Example-based synthesis uses only input-output examples to generate code. While these examples provide precise constraints on the desired program's behaviour, they often represent an incomplete specification of the user's intent. Users would have to come up with a huge amount of examples to cover all edge cases of a program, which is often infeasible. Therefore, program synthesis tasks are usually defined through only a few examples which makes them under-defined.
Multimodal synthesis combines multiple modalities to generate programs. In PBE the most used multimodal combination is a natural language description combined with IO examples, benefiting from the complementary nature of these two modalities. Natural language descriptions provide a more "complete" task description, capturing the user's intent in a way that examples alone cannot. For instance let's say we want to match United Airlines flight numbers with the positive examples from Table 2.2.

Positive examples
`UA12`
`UA34`

$\longrightarrow$

Candidate programs
`(UA12 | UA34)`
`[A-Z]{2}[0-9]{2}`
`UA[0-9]{2}`

Table 2.2: There are often multiple interpretations possible of a list of positive examples. An inductive program synthesiser generates candidate programs that satisfy all positive examples. By only using these positive examples it is not possible to discriminate between those candidate programs to find the user intended one.

Any of the generated programs could be the intended program by the user, as they all satisfy all IO examples. From a program synthesis perspective, there is no way to discriminate between them. When the user adds the following description "strings starting with 'UA' and ending with two digits" it is immediately clear which of the three programs best fits the user intent. In this case it becomes clear that `UA[0-9]{2}` is the correct solution. `(UA12 | UA34)` overfits as it does not match any other input than the positive examples. `[A-Z]{2}[0-9]` underfits as it also matches inputs that are not United Airways flight numbers, for example 'AB12'. Only the synergy of task descriptions and IO examples made it possible to find the intended program. Descriptions of a task can help to discriminate between multiple potential solutions to find the user intended one.

Natural language description help to find the right balance between overfitting and underfitting. Whereas IO-examples offer precise constraints that help disambiguate the natural language description and guide the search towards the correct program. Suppose a user wants to generate a program that matches numbers in a string using only a natural language description. Even for this simple case, there are many interpretations of the word 'numbers'. For example, should it match integer numbers, decimal numbers, negative numbers and numbers in a word? In this cases, IO examples are helpful to clarify the user intention as seen in Table 2.3.

Pattern	Positive example
Integer numbers	`1`
Decimal numbers	`1.2`
Negative numbers	`-1`
Numbers in a word	'56' in 'UA56'?

Table 2.3: Potential patterns which a user could have intended. IO examples are helpful to show the intended behaviour.

Multimodal synthesis techniques can achieve higher accuracy and efficiency than NLP-based or example-based approaches alone. They are capable of narrowing down the search space effectively, handling the ambiguity of natural language, and providing a more comprehensive specification of the user its intent. This makes multimodal program synthesis a powerful approach to efficiently generate programs across a wide range of tasks.

2.5. Vector Embeddings & Cosine Similarity

Vector embeddings are used in machine learning and natural language processing, serving as a compact representation of high-dimensional data. This form of data representation is used to capture the semantic and syntactic features of the data. When considering natural language data, each string is represented by a unique vector in a multidimensional space.
Cosine similarity is a metric used to measure the similarity between two vectors. It is defined as the cosine of the angle between the vectors, providing a measure of their orientation, regardless of their magnitude. Mathematically, the cosine similarity between two vectors A and B can be calculated using the following formula:

$$Sim(A, B) := \cos(\theta) = \frac{A \cdot B}{\|A\|\|B\|} \tag{2.3}$$

The result of this calculation ranges between -1 and 1. A value closer to 1 indicates that the vectors are similar, 0 indicates that the vectors are not similar, and -1 indicates that the vectors are opposites.
For example given the following three sentences and embeddings:

Name	Description	Embedding
Vector A	The dog is chasing its tail	[1, 2, 3]
Vector B	A canine is pursuing its own tail	[1.1, 2.1, 2.9]
Vector C	I ate pasta last night	[-5, 6, -2]

Table 2.4: An example of descriptions with their corresponding embedding vector. The sentence "The dog is chasing its tail" is similar to "A canine pursuing its own tail", which can be seen by the relatively similar embedding vectors.

The cosine similarity between Vector A and Vector B is $Sim(A, B) = 0.999$, which means they are very similar. Whereas, the cosine similarity between Vector A and Vector C is $Sim(A, B) = 0.033$, which means they are not similar.

Finding similar prompt-answer pairs
In the context of few-shot learning, cosine similarity can be used to find prompt-answer pairs in the training data that are similar to a new prompt. The embeddings of all data points in the training data need to be stored in a vector database to efficiently search for similar prompts. New unseen prompts can then be compared with the already existing prompts in the vector database to find the most similar prompts-answer pairs. These pairs can then be given as extra information to an LLM to predict the response to the new prompt.

Data set similarity
To evaluate the similarity of data sets, the cosine similarities between all pairs of vectors are then calculated and compared with the other data set. For example, a high average cosine similarity would indicate that the data sets have a high degree of overlap in their semantic content.

3 Literature Review

In the rapidly evolving domain of artificial intelligence, the synthesis of programming through LLMs has emerged as an intriguing field of study. LLM program synthesis presents a paradigm shift from traditional coding by generating code from high-level language specifications. Concurrently, the established concept of PBE remains vital, as PBE systems can guarantee that generated programs satisfy the given specification, whereas LLMs-generated programs can not. Understanding the intricacies of these two areas is crucial towards combining them into a streamlined, synergistic workflow, with the goal of a more efficient program synthesis workflow.

It is essential to mention that multimodal program synthesis, while holding immense potential, is currently under-explored in academic research. This form of synthesis, incorporating multiple types of data inputs, can profoundly impact the efficacy and adaptability of program synthesis systems. This research aims to contribute to this sparse body of knowledge, seeking to foster new insights and advancements in this multifaceted area.

Finally, a comprehensive understanding of prompt generation techniques for LLMs is another integral part of this study. The process of getting user-intended responses from LLMs using suitably defined prompts is critical in maximizing the efficiency and utility of these models. The ability to generate effective prompts automatically would significantly enhance the interaction with the program synthesis workflow.

This literature review provides an overview of the current state of programming by example, LLM program synthesis, multimodal program synthesis, and prompt generation. This study aims to improve our knowledge in these fields with the goal of combining these methods into a stronger and more effective approach.

3.1. Programming by Example

Programming by Example (PBE) is a powerful program synthesis method that allows users to create programs by providing examples of the intended program behaviour [7]. This technique has been successfully applied to various domains, including string transformations [11] and regular expression synthesis [4]. In PBE, the interaction between the user and synthesizer is limited to examples for both the initial specification and refinements.

One of the first and most well-known applications that use PBE on a large scale is FlashFill [11], which was developed by Microsoft in 2011. FlashFill revolutionized the way users perform data manipulation tasks by automating the process of generating data transformation functions based on provided examples in Microsoft Excel.

FlashFill's success lies in its ability to simplify repetitive and time-consuming tasks that often arise when working with spreadsheets, such as formatting, extracting, or concatenating data. By merely providing a few input-output examples, users can teach FlashFill the desired transformation pattern, and the system will generate a function that can be applied to the entire dataset. The integration of FlashFill in Excel has made PBE accessible to a broad audience, showcasing its potential to empower users with little or no programming expertise. By reducing the need for manual data manipulation and complex formulas, FlashFill has significantly improved productivity and efficiency for countless Excel users worldwide.

One of the earliest works in PBE was on inferring LISP programs from examples [22]. Over the years, the field has evolved, with researchers exploring novel ways to automate string processing in spreadsheets, a real-world scenario of such a PBE engine is FlashFill in Microsoft Excel [11].

A common approach in PBE techniques is to combine enumerative search with lightweight deductive reasoning, significantly pruning the search space [1, 8]. This method has been employed in various domains, such as synthesizing data structure transformations from input-output examples [10], developing a framework for data extraction by examples [14].
Each paper mentioned expanded the horizon of possible applications within the domain of programming by example. From early explorations in inferring LISP programs from examples to the practical implementation of PBE techniques like FlashFill in Microsoft Excel. Each individual study has, therefore, contributed to a cumulative increase in our understanding and application of PBE techniques. For our research, we examined the methods employed across various domains to get crucial insights into how inductive program synthesis can be effectively and efficiently executed. These insights have provided a valuable context to our research, helping us understand the limitations of program synthesis. Moreover, by understanding the history and evolution of PBE, we can better position our own research within this broader context. This enables us to build upon existing methods, learn from past failures, and make more meaningful contributions to the field.

3.2. Program Synthesis with Large Language Models

Code generation has become a hot topic in AI research because of transformer-architecture models. Mainly because the transformers-architecture allowed for models that can understand the relationship between sequential elements that are far from each other. LLMs have been central to this effort, leveraging their natural language processing capabilities to generate human-like text and code. Several papers have investigated LLMs, providing valuable insights into the capabilities and limitations of LLMs in code generation. There is still a significant amount of research required in the pursuit of understanding LLMs. Remarkably, the transformer architecture has only existed since 2017 [24], underscoring the novelty and rapid evolution of LLMs.

One of the first papers that used the transformer-based architecture to generate code is by Feng et al. (2020) [9] who introduced a model called CodeBERT. CodeBERT is one of the first models that were able to generate complex programs based on natural language. The study highlighted the capability of language models to extract programming language semantics from large-scale data, paving the way for more advanced models.

A subsequent paper, from Austin et al. (2021) [3], conducted an evaluation of LLMs' capability in synthesizing short Python programs. The models evaluated ranged from 244 million to 137 billion parameters, with performance scaling log-linearly with model size. Notably, this study found that engaging the model in dialogue about code and incorporating human feedback could halve the error rate. Despite these advancements, the study also highlighted some limitations of these models. One major limitation is that these models are heavily input dependent, which means that these models sometimes fail on very easy tasks when given a specific input.

Chen et al. (2021) [5] introduce Codex, a Generative Pretrained Transformer (GPT) model fine-tuned on publicly available code from GitHub. Codex showed significant capabilities in writing Python code based on HumanEval, a new evaluation set designed to measure the functional correctness of programs. Nonetheless, the study again showed the limitations of these models, including their struggles with longer chains of operations and more complex reasoning tasks.

Before Codex, language models struggled to generate consistent and coherent long passages of text. The development and success of GPT-based models made it possible to generate programs with much higher accuracy and with far better natural language understanding capabilities. The natural language understanding and code generation capabilities of LLMs have paved the way for generating programs in innovative ways, which our research builds upon.

A more recent development in this area is AlphaCode from Li et al. (2022) [17]. They showed that LLMs still perform poorly when evaluated on more complex and unseen problems that require problem-solving skills beyond simply translating instructions into code. They introduce AlphaCode, a system for code generation that can create solutions to more complex problems that require a deeper understanding of the problem. This model demonstrated the ability to perform competitively in programming competitions on the Codeforces platform [1]. AlphaCode utilized an encoder-decoder transformer architecture and a strategy of generating diverse programs and then filtering them. This achievement marks a notable milestone in the application of AI in code generation, suggesting that AI could perform at a level comparable to human programmers under certain circumstances.

AlphaCode only uses natural language descriptions to generate programs. To increase the accuracy, they generate multiple programs and then cluster and filter them to only get the

[1]Codeforces website: `https://codeforces.com/`

potential best programs. However, a drawback of this approach is that clustering and filtering are not straightforward to implement for new domains of problems. On top of that, generating multiple programs also significantly increases computational intensiveness.

Our approach combines the use of natural language descriptions and input-output examples, which simplifies the process of identifying the correct solution. Furthermore, we have eliminated the need for complex filtering and clustering techniques. Instead, our workflow integrates a program synthesizer that can repair partially correct responses from an LLM.

The ongoing research on program synthesis by LLMs has revealed promising results, with models showing increasing competence in generating functional code. However, these studies also show the challenges that remain, such as handling complex programming tasks, reasoning tasks and biased outputs. Further research is needed to overcome these limitations and fully realize the potential of AI in program synthesis.

3.3. Multimodal program synthesis

The number of papers that address multimodal program synthesis is limited, particularly in the context of regular expression synthesis. The papers in this section present different methods and techniques for synthesizing regex from multimodal inputs to improve the accuracy and efficiency of regex generation. In these papers 'multimodal' always refers to natural language descriptions and input-output examples,

Chen et al. (2020) [6] presented Regel, which parses the natural language description into a hierarchical sketch to guide the PBE engine. The evaluation shows that Regel outperforms NLP-only and PBE-only baselines. Regel uses semantic parsing instead of deep learning methods, as it requires less labelled training data. Regel is limited by its needs for specific parsing which takes a significant amount of time and knowledge to build. This constraint makes it less suitable for diverse use cases and makes it hard to use models in different domains. Although Regel improves over previous methods, it is still not optimal with an accuracy of 75.5% on a relatively easy data set.

Our methodology is similar to Regel in the sense that partially correct programs are used to create the full program. However, Regel requires a semantic parsing of the English description to create regex sketches, whereas our method leverages an LLM to perform this task, eliminating the need for specific parsing methods. This flexibility makes our approach useable outside of the domain of regular expressions generation. Regel utilizes sketches as the initial input for the program synthesizer. Whereas, our method involves extracting building blocks from partial solutions, which we then input into the program synthesizer. Nonetheless, both methodologies use the underlying principle of synthesising solutions with a program synthesiser from partially correct programs.

Ye et al. (2020) [26] propose a neural synthesis approach by using a recurrent neural model that aims to find a program satisfying user-provided constraints. The paper demonstrates that this approach outperforms prior techniques in terms of accuracy, efficiency, and frequency of finding model-optimal programs on the StructuredRegex data set (section 5.2).

This paper also creates partial programs with an abstract syntax tree and non-terminal nodes. These non-terminal nodes are then filled by a synthesiser to create a full program with only terminal nodes. In our approach, we first try to solve the problem with an LLM. We extract the building blocks from the partially correct program if the LLM did not find a correct solution.

Li et al. (2021) [16] introduce TransRegex. The paper treats the problem as NLP-based synthesis with regex repair and presents novel algorithms for the combination of NLP-based synthesis and regex repair. TransRegex is currently the state-of-the-art model and achieves much higher accuracy than previous multi-modal techniques. However, TransRegex is heavily optimized and finetuned for regex synthesis and specific data sets. They specifically analyzed the most made mistakes from the regex synthesiser and created a list of 10 transformations to repair those mistakes.
TransRegex is very similar to our approach as it first tries to find a complete program using one technique. If the first program is not correct they use a second algorithm to fix the incorrect program based on the provided IO examples. However, TransRegex uses a synthesiser in the first step and an exact highly-specific and optimized algorithm approach in the second step, whereas our approach first uses an LLM and then tries to repair the programs by using an inductive program synthesiser with a simple grammar.

In conclusion, these papers present various approaches to multimodal program synthesis, focusing on the synthesis of regular expressions. They combine natural language processing, neural networks, and program synthesis techniques to improve the accuracy and efficiency of regex generation.

3.4. Prompt generation

Prompt engineering is an essential area to enhance the performance of LLMs. Precise construction of prompts can result in more specific and better-performing model responses, but this precision often requires intensive manual involvement or gradient-based tuning of prompts. While these approaches have their benefits, they can be limited in scalability and practical application.

Zhou et al. (2023) [27] present a simple method to automatically generate prompts based on IO examples. This is one of the first methods in the new field of automated prompt generation. The method is called Automatic Prompt Engineer (APE), which automatically generates and selects instructions based on input-output examples. It uses an LLM to generate, score, and optimize prompts. This strategy reduces the dependency on human input and prompt creation, thereby streamlining the process and increasing efficiency. The authors treat LLMs as black-box optimizers, contrary to other methods like backpropagation [15] which uses the weights of the LLM to find the best prompt. The algorithm generates and scores a pool of instruction candidates, from which it selects the most appropriate instruction. APE can automatically generate high-quality instructions that can significantly enhance zero-shot and few-shot learning performance. This makes APE effective in directing LLMs towards desired behaviours without the need for human prompt creation.

Reynolds and McDonell (2021) [21] benchmarked multiple prompt programming techniques to generate higher-quality outputs with LLMs. It specifically evaluates the capability of GPT-3 in tasks through a zero-shot approach versus a few-shot approach. The authors also propose new perspectives on how to approach prompt engineering, including the exploration of metaprompts. Metaprompts, are a way to let LLMs generate their own natural language prompts for a range of tasks. Significant human effort is needed to create good prompts since non-specific prompts tend to be less efficient than those tailored for a certain task. This highlights the value of creating automated ways of generating prompts dedicated to specific tasks. This method is nearly identical to the Automatic Prompt Engineer approach. This research also emphasises the importance of treating prompts as an integral part of LLMs prompt tuning for generating desired outputs.

Both papers conclude that prompt engineering is an important aspect of getting the right output from LLMs. In our methodology, we use prompt engineering techniques to improve the accuracy of the generated programs. Prompt engineering techniques not only enhance the output of LLMs but also expands their potential by using the APE method to let LLMs generate their own prompts. Automatic prompt generation can make LLMs easier to work with by eliminating the need for human-made prompts. Prompts can be automatically generated by using the IO examples, this potentially removes the need for natural language descriptions. In our research, we use the APE method to create prompts that can generate programs.

4 Methodology

The creation of the Language Model Augmented Program Synthesis (LMAPS) workflow is primarily driven by the need to overcome the limitations associated with Programming by Example (PBE). The goal of the LMAPS workflow is to use LLMs to augment traditional program synthesis, helping to tackle these challenges and improving the effectiveness of programming by example. This new workflow contains three new LLM-based components, called "Specification Suggestion", "LLM Program Synthesis", and "Explain Program".

By introducing the "Specification Suggestion" step, we are leveraging the natural language understanding capabilities of the LLM to reduce the ambiguity of the problem specifications. This is done by generating relevant examples and a task description with the LLM. The "LLM Program Synthesis" step overcomes the issues of generalizability and efficiency by generating programs with an LLM based on the specifications. These generated programs are often correct or partly correct. A program synthesis engine can make variations of these nearly correct programs to find a correct program. This approach reduces the search space and thereby increases the efficiency. Lastly, the "Explain Program" step enhances explainability. The generated program is explained to the user, which helps the user understand the program.

By including natural language problem descriptions alongside IO examples, we have introduced a second modality to the traditionally single modality of PBE. This allows the system to use this additional information for better results. The ultimate goal is to build a more accurate, efficient, and user-friendly method for programming by example, one that leverages the capabilities of large language models to generate, enhance, and explain program solutions.

To create the new LLM augmented workflow, we first need to understand the traditional program by example workflow. After that, we show the program synthesis workflow with the new LMAPS components.

4.1. Traditional PBE workflow

The first step in a traditional PBE workflow begins with the user. The user is the actor in the process who has a specific need or problem that they wish to address through a program.

The user defines the problem as a program specification, which shows what the desired program is expected to achieve. Note that the specification can also be indirectly defined, for example, FlashFill in Microsoft Excel, where the user does not know that program synthesis is used under the hood.

The quality of the program specification heavily influences the success of the inductive program synthesis. The more accurately and completely a problem is defined, the better the chances that a solution will be generated that meets the user's needs. Therefore, it is crucial to focus on the quality of the specification. The user can provide as many IO examples as necessary to express the general and edge behaviour of the desired program.

Once the program specification represents the intended behaviour, it can be given to an inductive program synthesis engine. This engine uses the provided specification to synthesise a program that satisfies the given IO examples. The goal is to generate a program that not only meets the IO examples given but is also general enough to handle other unspecified inputs correctly. However, in traditional program synthesis workflows, it is hard to distinguish between multiple 'correct' programs.

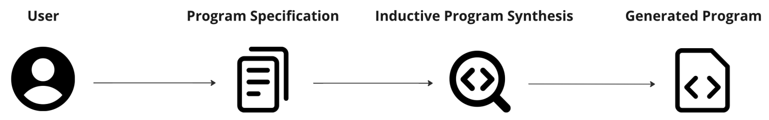

Figure 4.1: The traditional program synthesis workflow. A user creates a specification of a problem, which an inductive program synthesiser can use to generate a program that solves it.

4.2. LLM augmented program synthesis workflow

The LMAPS workflow overcomes the limitations of PBE by using the capabilities of LLMs. By incorporating LLMs into the process, LMAPS gains the ability to understand and generate code based on both natural language descriptions and specifications. This integration allows programmers to provide high-level instructions and requirements, and the LLM assists in automatically synthesizing the corresponding code that fulfils those requirements. It is important to note that the 'Specification Suggestion' and 'Explain Program' steps of the LMAPS workflow are optional. This means the workflow can be tailored to specific needs. For instance, the 'Explain Program' can be excluded without affecting the other components.

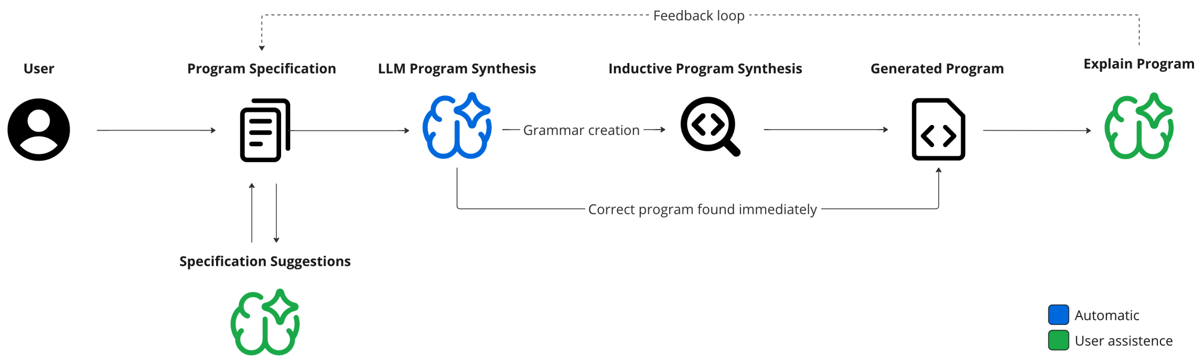

Figure 4.2: The LLM augmented workflow consists of the Specification Suggestion, LLM Program Synthesis and Explain Program steps on top of the traditional workflow. The Specification suggestion and Explain Program steps assist the user in the program synthesis workflow, whereas the LLM Program Synthesis works by generating programs based on the given specification automatically.

Program Specification

To generate the intended program, a user first needs to formulate the problem. The program specification consists of a natural language description of the problem and a set of IO examples. This specification is important to check the correctness of generated programs and works as guidance for the next steps in the workflow. The LLM can help to improve the specification by formulating the task description and generating diverse IO examples. This assists users in creating a diverse set of IO examples. These examples then need to be labelled by the user to indicate if they are correct or not, based on the desired behaviour. Defining a representative set of the problem is crucial in overcoming the ambiguity limitation of programming by example.

By leveraging the LLM its ability to understand and generate contextually relevant examples, users can obtain a diverse set of IO examples that cover various aspects of the desired program. The LLM-generated examples can aid in identifying edge cases, ensuring the synthesized program is robust and reliable. Even incorrect input-output examples generated by LLMs are valuable as negative examples, as they often contain partially correct patterns. These negative examples are important for multiple reasons. First, negative examples can also help users refine their natural language description by pointing out ambiguities or inconsistencies in their initial description. Secondly, generated negative examples can expose edge cases or unusual patterns the user may not have considered. By taking these negative examples into account, the PBE system can better handle such edge cases and generate a more reliable program. Lastly, generated negative examples can help discriminate between programs as PBE systems often find multiple programs that satisfy all IO examples.

LLM Program Synthesis

The LLM gets as input a prompt that contains a natural language description of the task to generate a list of programs. This prompt contains the problem specification to show the LLM what kind of program it has to generate. For regular expression generation, the prompt could look like this: "Write a list of regular expressions that satisfies the following description and IO examples. <`description`> and <`io examples`>". This generates a list of one or more candidate programs that might solve the problem. The generated candidate programs are then tested against the IO examples to check if they contain a correct program. A candidate program is correct if the program satisfies all IO examples. When all candidate programs are incorrect, they are given to the grammar creation step to extract valuable building blocks from them. These building blocks make it able to create a concise grammar, thereby improving the overall effectiveness and efficiency of the workflow. Therefore, LLM program synthesis is crucial to overcome the efficiency and generalizability issues of PBE-only workflows.

Grammar Creation

We need to create a grammar for the inductive program synthesis, as the grammar defines the syntax of the language in which the program is to be synthesized. We can utilize valuable insights from partially correct programs generated by the LLM by extracting their building blocks. These building blocks, being part of a partially correct solution, have a high likelihood of being useful in a correct solution. Using these building blocks, we construct a grammar that is used as input for the inductive program synthesis engine. The LMAPS process can be seen as a program repair mechanism, transforming partially correct LLM-generated programs into beneficial building blocks for further program synthesis. Instead of dismissing the incorrect outputs, this methodology harnesses them to help steer the program synthesizer's search process, leading to more accurate program generation.

Inductive Program Synthesis

The inductive program synthesis engine gets the IO examples and building blocks grammar as input with the goal of finding a program that satisfies all IO examples. The search space includes all potential programs that can be generated using the provided grammar and constraints. This step can be seen as a repair mechanism based on the grammar with building blocks from partially correct programs. The benefit of inductive program synthesis is that they can quickly generate and evaluate programs to find a correct one. The engine can use various search strategies to find a program within the search space that satisfies the specification. This search can be conducted using different approaches, including enumerative search, stochastic search, and constraint-solving, among others. Every program that the synthesiser generates is verified against the specification. If the program satisfies the IO examples, it is considered a valid program. If not, the engine continues the search. Finally, if the synthesiser finds one or more valid programs, the programs are given to the user. The programs can then be used, evaluated, or further refined.

Inductive program synthesis, combined with the constructed grammar from candidate programs and natural language descriptions, showcases the power of program synthesis. This concise grammar, instead of a complete grammar, makes finding a correct program more efficient. However, no correct program will be found if the solution consists of different building blocks than those present in the grammar.

Explain Program

Once an inductive program synthesizer generates a correct program, it only guarantees that the program meets all IO examples. However, understanding the program can be challenging for the user, especially since it's created automatically. Synthesized programs often suffer from a lack of clarity, posing considerable obstacles when it comes to interpreting, debugging, and maintaining the resulting software.

The main aim of inductive program synthesis systems is to produce a program that can accurately handle the given IO examples rather than focusing on the readability of the generated program. This oversight could potentially lead to significant risks when trying to predict the behaviour of the program in situations not covered by the provided examples. LLMs can bridge this gap of understanding by generating human-friendly explanations of the program. For instance, given a regular expression such as `^a[0-9]*b$`, an LLM could explain that this pattern will match any string that starts with 'a' (`^a`), followed by any number of digits (`[0-9]*`), and ends with 'b' (`b$`). LLMs can also provide examples, further aiding in comprehending the generated regular expressions. For instance, the LLM might say that the regular expression `^a[0-9]*b$` will match strings like 'a123b', 'a4567b', or 'ab', but not 'a123', '123b', or 'ba123b'. However, it is always important to check the output of an LLM, as it is not guaranteed to be correct.

Feedback loop

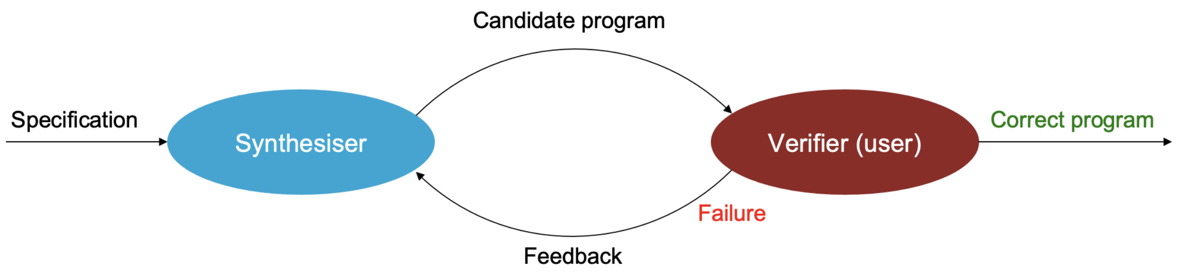

Figure 4.3: Programs that satisfy all IO examples are not necessarily the user-intended program. The feedback loop can be used to update the specification of the problem to synthesize a more user-aligned program. A verifier checks if the generated program works as intended. If not, the program specification has to be updated which acts as feedback for the synthesiser.

Including a feedback loop in our approach potentially allows for iterative refinement of the program specification based on user inputs and synthesized results. This could enable the continuous improvement of program synthesis. When the generated program does not align with the user intent, the user can update the specification by updating the natural language description or by adding a counter-example to the set of IO examples. This feedback loop can be seen in Figure 4.3. The complete workflow has to be rerun after the user updates the specification.
In conclusion, we introduce the LMAPS workflow, designed to address the limitations of traditional PBE workflows. LMAPS aims to enhance various facets of PBE, improving its accuracy, efficiency and user experience. Our proposed methodology includes an innovative process of extracting beneficial building blocks from partially correct programs generated by LLMs. We hypothesize that these insights could aid in creating a more refined grammar for program synthesis, thereby leading to a more accurate and efficient program generation. We evaluate the effectiveness of this approach in the results section.

4.3. Prompt enhancing components

The quality of the prompt has an important factor in the accuracy of the workflow. To enhance the prompt generation, we use two methods. The first method can aid to generate a prompt based on the provided IO examples, which can help to reduce the time that is needed to manually construct prompts. The second component can improve few-shot prompts by including similar description-answer pairs. Few-shot often reaches a higher accuracy than zero-shot prompts, especially with similar description-answer pairs. It is important to note that both these components are optional to use but can help to enhance prompts.

4.3.1. Automatic prompt generation

Prompt creation by humans can take a lot of time. This time could potentially be saved by letting LLMs create their own prompt based on IO examples. This is done by asking the model what the instruction should be to create the output from the input as explained in 3.4. So essentially, an LLM uses a standard prompt and IO examples to generate a task description prompt. This generated task description prompt can then be used to generate programs.

For example, by using '1' and '3' as positive examples, and '2' as a negative example for a program that can check if an integer is odd.

```
User: Write a task instruction prompt for the following IO examples:
Input: 1
Output: True

Input: 2
Output: False

Input: 3
Output: True

Model: Please write a function that takes an integer as input. The function should return
    True if the input is an odd number, and False if it's an even number.
```

Prompt 4.1: An example of a prompt to automatically generate a task description prompt by using the IO examples.

The model output "Please write a function that takes an integer as input. The function should return True if the input is an odd number, and False if it's an even number." can then be used to create programs without a human ever needing to write a task description. These automatically generated prompts can then be used to synthesize programs by giving the task instruction prompt combined with the natural language description and IO examples to an LLM. The complete prompt that is used in the Automatic Prompt Engineering method can be found in Appendix A.3.

4.3.2. Similar few shot examples lookup

In the context of Few-Shot Prompting, vector databases make it able to retrieve similar tasks, where we already know the answer, efficiently. This can be useful to add more relevant examples to the prompt, which can improve the performance of the prompt.

In our case, the task-answer pairs consist of a description that describes a regular expression. These descriptions are then converted to vector embedding, which makes it able to calculate the similarity between descriptions in a high-dimensional vector space called the "embedding space." The embedding model essentially learned to map semantically similar descriptions to proximate points in the embedding space. When a new description is presented, it is first transformed into a vector representation using an embedding model. This vector is then compared to the vectors in the database by cosine similarity. The system can quickly identify similar descriptions and their corresponding regular expressions. These descriptions and regular expressions can then be used to give more relevant IO examples in the few-shot prompts.

4.4. Regular expression components

To evaluate LMAPS on regular expressions, we use two more components. The first component is used to create a valid regex grammar. The grammar creation step is necessary to incorporate the building blocks from candidate programs in the grammar so that the inductive program synthesiser can use it. The second component is needed to check if a generated regular expression is equivalent to the one used in the evaluation regex data sets. This extra validation step is important as programs that satisfy all IO examples are not necessarily the intended program.

4.4.1. Regex grammar creation

The goal of the grammar creation step is to create a concise regular expression grammar that contains all building blocks to synthesize a correct program that satisfies all IO examples.

Building block extraction from candidate programs

The modularity of regular expressions allows expressions to be broken down into simpler builder blocks. The building blocks of regular expressions can be divided into literal characters, and quantifiers, which dictate how many times a pattern should repeat.

Literals are a string of characters that match exactly as they are. Everything inside parentheses or block quotes becomes a single literal. These elements can then be modified with quantifiers. For example, the regular expression `(abc)+[A-Z]d{2}` consists of the literals `(abc)`, `[A-Z]` and `d`.

Quantifiers specify how many instances of an element we are looking for. For example, in the regular expression `a*`, the asterisk is a quantifier, and it signifies zero or more 'a'.

Extracting building block elements and quantifiers from a regular expression is a task that can be solved by using regular expression patterns themselves. First, the '?', '+' and '*' quantifiers have to be converted to `{0,1}`, `{1,}` and `{0,}` respectively. After that, all elements and quantifiers can be extracted by using three regular expressions. One to extract parentheses, one to extract block quotes and one to extract quantifiers surround by curly brackets.

Building block extraction from natural language description

Natural language descriptions also contain information about potential building blocks and elements included in the regex solution. Descriptions often contain explicit digits, which can be used to specify quantifiers in regular expressions. These digits may appear as single or composite numbers and denote specific repetitions or ranges. A regex pattern can be used to detect and isolate these numbers. After that, the numbers can be converted to quantifiers. For example, given the description "matches exactly 5 alphabetic characters," we can extract the digit '5' and convert it into a quantifier in the regular expression, resulting in "5".

Numbers may also be represented as words in natural language descriptions, such as "one", "two", "three", etc. These word counts can be translated into numeric quantifiers for regular expressions. By using a number lookup table, we can extract those numbers. For instance, the description "matches three consecutive digits" can be translated into a regular expression using the quantifier "3".

It is common in natural language descriptions to specify elements of interest enclosed in quotes. Elements surrounded by quotes can also be found by using a regular expression. For example, in the description "matches the word 'apple'", the text within quotes 'apple' can then be used as an element in the regular expression grammar.

Extracting elements and quantifiers can also be done by prompting an LLM. This saves time for users as they don't need to develop a parsing algorithm. For regular expressions, a parsing prompt could look like this: "Extract all literals and quantifiers from this regular expression: «regular expression»". However, exact parsing gives more control over the building blocks extraction. In general parsing methods are also faster than prompting LLMs.

Grammar

The building blocks are inserted as regex elements and quantifiers. The program synthesis starts with a 'RegexConnector' as its root. This grammar is then given to the inductive program synthesis engine to find a correct program based on these building blocks and quantifiers. Small variations can be made to the building blocks if no solution is found with the current building blocks.

```
RegexConnector = Regex + RegexConnector
RegexConnector = Regex
Regex = Literal or Literal + Quantifier
Literal = <List of literals>
Quantifier = <List of quantifers>
```

Grammar 4.2: Simplified regular expression grammar which can incorporate building blocks, consisting of literals and quantifiers.

4.4.2. Program validation

A crucial phase in the program validation is the regex equivalence check. This check involves the generation of positive examples and negative examples. First, a set of positive examples is sampled from the regular expression program. Sampling matching string from the regular expressions is easy to do as the regular expression itself defines a language. These positive examples can then also be used to construct near-miss negative examples as more than one random negative example is not useful.

Regex equivalence

The flexibility of regular expressions allows for multiple representations of equivalent patterns. Therefore, checking the equivalence of regexes is not as straightforward as a simple string comparison. Two regexes that look dissimilar can match the same language, making them functionally equivalent.
The regex equivalence check determines the equivalence of two regular expressions, which is needed to remove duplicate candidate programs or to detect whether a synthesized candidate program is a solution (if the solution is known).

By determining regex equivalence, we can eliminate the computational redundancy of evaluating functionally identical candidate programs. This reduction makes the search process more efficient.
There are two ways to determine regex equivalence. One way is by transforming both regexes into their respective deterministic finite automata (DFAs). These DFAs can prove whether the regexes are functionally equivalent by checking if the DFAs are the same. The DFA equivalence check can be performed using a graph equivalence algorithm. However, two functionally equivalent regular expressions can still have different DFAs. That is the reason we need another method for determining regex equivalence. This is done by random sampling 'n' positive examples from the languages of both regular expressions and constructing 'n' negative examples from those positive examples. In a case where two regular expressions are equivalent, they match the same set of positive examples and do not match any of the negative examples.

Negative examples are crucial for ensuring the regular expressions are not overgeneralized, for example, a regular expression which matches everything. The drawback of the random sampling method for checking equivalence is that it does not guarantee that the regexes are equivalent, whereas equivalent DFAs prove that the regexes are equivalent. As 'n' increases, the likelihood of making an accurate prediction becomes more certain. It can be guaranteed that the two expressions are equal if 'n' equals the length of the whole language for both regular expressions.

Positive example generation
Generating positive examples works by sampling strings with different complexities and lengths from the language. It is important to create a diverse set of positive examples to cover as many edge cases as possible for the equivalence check. However, generating too many positive examples makes the regex equivalence computationally intensive, which slows down the workflow as the equivalence check has to be done many times.

Negative example generation
Negative example generation is also necessary to check the equivalence of two regular expressions in a program synthesis workflow. The task involves crafting 'n' negative examples from the positive examples provided for both regular expressions. However, this is not a trivial process as it requires some knowledge of the specific regular expressions to create valuable and nearly matching negative examples.

Generating random strings as negative examples is a straightforward solution but does not lead to robust evaluations. Random strings are almost always far from any plausible match for a given regular expression, so they contribute little to the effective testing of the expression. These random negative examples do not provide a proper boundary test and, therefore, may be unable to catch all possible errors or edge cases. An edge case could, for example, be a single character off, a misplaced special character or slightly different quantifiers.

The workflow uses mutation of the positive examples to create such near-miss negative examples. This involves replacing, adding, deleting or swapping one or more characters. The goal is to create examples that are close to the regular expression but still fail to match it, thereby pushing the boundaries of the test case.

Additionally, to further improve the testing quality, some random strings can also be added to the pool of negative examples. While these may not present the challenge of the near-miss examples, they can still provide additional assurance that the regular expression functions as expected in more generic cases, further ensuring the validity of the equivalence check. It is important to balance the number and types of mutations to apply in order to create the most effective negative examples.

5 Experiments & Results

This chapter presents the results of our research on the Language Model Augmented Program Synthesis (LMAPS) workflow. We compare LMAPS with a traditional inductive program synthesis workflow on four regular expression synthesis data sets.

We evaluate the usefulness of augmenting inductive program synthesis with LLMs by answering the following questions:
Q1: Can LLMs improve the accuracy of traditional program synthesis?
Q2: Can information from LLM-generated programs improve the efficiency of traditional program synthesis by reducing the search space?
Q3: Can LLMs aid in the enhancement of problem specifications through example generation?
Q4: To what extent are LLMs capable of explaining the synthesized programs?

5.1. Research tools

The workflow is written in Python using the OpenAI package to interact with the ChatGPT 3.5 model with version identifier 'gpt-3.5-turbo-0301'. For the inductive program synthesis, we use Herb.jl [12]. Herb.jl is a program synthesis framework written in Julia. It uses an enumerative search to synthesize a program, or in this case, a regular expression that aligns with the IO examples. However, Herb is currently in development and not fully optimized. To still be able to test the LMAPS performance potential, we do a depth and search space analysis.

5.2. Data sets

To evaluate our workflow, we use four regex data sets: KB-13, NL-RX, StructuredRegex and KB-13-emoji. Each dataset contains a collection of descriptions along with their corresponding regex solutions. These data sets are often used to benchmark models on the task of regular expression generation. In the following section, we provide an overview of each data set, including its creation process and limitations.

KB-13

The KB-13 data set [13], introduced in 2013, is created by crowdsourcing pattern finding in random texts. Participants were asked to provide descriptions of patterns they observed, and programmers then manually crafted the corresponding regex solutions based on these descriptions. The descriptions provided in the data set tend to be relatively simple and similar. For example, the data set contains the descriptions 'lines that have at least 3 words' and 'lines which have 3 words.

NL-RX
The NL-RX dataset [18], created in 2016, was developed through a combination of manual crafting and crowdsourcing. A manually crafted grammar was used to translate regular expressions into natural language descriptions. These natural language descriptions were then paraphrased by crowdsourcing to create a more diverse set of descriptions.

StructuredRegex
StructuredRegex [25] was introduced in 2020 to overcome some of the limitations of the previous data sets. The regex solutions in this data set were generated using a probabilistic grammar based on StackOverflow posts. The data set its descriptions were created through crowdsourcing, with participants writing descriptions for the regular expressions. The data set contains incorrect descriptions for some of the regex solutions, which may introduce challenges when using this data set for evaluation purposes. This data set is specifically made to be more diverse and complex than the KB-13 and NL-RX data sets.

Shared limitations
KB-13, NL-RX and StructuredRegex all share similar limitations. The main limitation of those data sets is that they use 'NOT' and 'AND' operators in their domain-specific language (DSL), which cannot be trivially converted into valid regex. We filtered the data sets to only keep the data points that could be converted to valid regular expressions. Additionally, the regex solutions in this data set are often unnecessarily complex. Many regex solutions can be replaced by an equivalent simpler regular expression. However, we did not filter based on this limitation as the regex solution is only used to check if a candidate program is equivalent.

KB-13-emoji
We introduce a new data set called KB-13-emoji, as KB-13, NL-RX and StructuredRegex could potentially be part of the training data of ChatGPT 3.5. Our modified KB-13 data set is used to evaluate our LMAPS approach with a data set that ChatGPT 3.5 is not trained on. This approach helps to understand how the model generalizes from its training data and tackles unforeseen inputs, which our proposed workflow relies on. We modified the existing KB-13 data set in a unique manner, by incorporating emojis into its natural language descriptions and regular expressions. The idea behind this modification comes from our limited knowledge of the training data used for ChatGPT 3.5. Without concrete knowledge of the corpus utilized in its training, there is an inherent risk of our chosen data set overlapping with the training data set, which could potentially skew the results of our experiment. By infusing our data set with emojis, we aim to minimize this risk. An example of a description: "sentences that contain «emoji» or «emoji»". It is important to note that the KB-13-emoji is a simpler subset of KB-13 as the description always contains the sequence it has to match in the natural language description. However, this approach also works with descriptions like "sentences that contain an apple emoji", where the emoji is described in the text.

Our modification process involved replacing specific parts of the KB-13 data set with emojis, chosen at random, yet maintaining the underlying structure and content of the data set. We are confident about the uniqueness of our modified data set because emojis are seldom used as serious linguistic elements in academic databases. Thereby reducing the chance of pre-existing similar regex data sets that include emojis.

In summary, KB-13, NL-RX and StructuredRegex provide valuable resources for research in the field of regex generation from natural language. However, they exhibit limitations related to the DSL operators and the correctness of descriptions. It is important to keep these limitations in mind when evaluating new regex generation methods. An overview of the data sets can be found in Table 5.1.

Name	Year	Descriptions created by	Regexes created by	Datapoints
KB-13	2013	Crowdsourcing	Programmers	824
NL-RX	2016	Crowdsourcing	Manual crafted grammar	10,000
StructuredRegex	2020	Crowdsourcing	Probabilistic grammar	3,520
KB-13-emoji	2023	Modified KB-13 description	Modified KB-13 regexes	200

Table 5.1: Overview of the regular expression data sets.

Combined, these data sets contain 14544 data points. We use a subset of these data sets for the evaluation for two reasons. Firstly, it is not feasible to make 14544 requests to the ChatGPT API for all of the 14 prompts. However, making a subset of the data sets does not change the underlying concepts of the data sets, as the data points per data set are relatively similar to each other. Secondly, the StructuredRegex contains many wrong descriptions. Some of the wrong descriptions can be useful to make our LMAPS method robust to slightly wrong descriptions. We exclude data points which contained completely wrong descriptions as they can not be used as valid input for an LLM. For the evaluation, we use 200 data points from the KB-13 data set, NL-RX and KB-13-emoji data sets, and 111 manually checked data points from the StructuredRegex data set.

5.2.1. Data set similarity

Using multiple data sets prevents our research from being overly sensitive to the unique characteristics of a single data set, thereby ensuring the findings can be applied in a wider range of contexts. Different data sets can introduce different challenges that test the robustness and versatility of our methodology. Using multiple data sets also helps to benchmark the results against different methods and algorithms.

By selecting a variety of data sets and ensuring they are not overly similar, we can better evaluate the generalizability and robustness of our methodology. We use the cosine similarity the measure the similarity of the datasets. For every data point in the data set, we calculated the embedding of the natural language description with the 'text-embedding-ada-002' model of OpenAI. These vector embeddings can then be used to compare the different data sets. A lower cosine similarity value implies the data sets are distinct from each other, ensuring the data sets chosen for testing the methodology are not too similar. As all data sets are regular expression related, they still have a high base similarity. The cosine similarity of a data set with itself is not equal to 1, as the similarity is calculated as the average pairwise cosine similarity. The similarity of a data set with itself measures the variety of the descriptions within the data, where a higher value indicates that the data set is less diverse.

	KB-13	KB-13-emoji	NL-RX	StructuredRegex
KB-13	0.834	0.818	0.828	0.777
KB-13-emoji	0.818	0.903	0.810	0.770
NL-RX	0.828	0.810	0.874	0.808
StructuredRegex	0.777	0.770	0.808	0.872
Mean	0.814	0.825	0.830	0.807

Table 5.2: Cosine similarity between all data sets. A higher value means that the data sets are more similar.

We use t-Distributed Stochastic Neighbor Embedding (t-SNE) to represent the similarity between data sets visually. The t-SNE algorithm models each high-dimensional vector embedding by a two-dimensional point in such a way that similar objects are modelled by nearby points and dissimilar objects are modelled by distant points with high probability. Using t-SNE to visualize data set similarity is particularly useful because it allows us to intuitively understand the relationships between data sets in a way that is difficult to achieve with raw numerical measures like cosine similarity. By projecting the data sets into a lower-dimensional space, we can visually inspect whether and how the data sets cluster together, providing a visual means of understanding their degree of similarity.

If the data sets we use to test our approach are too similar, they would group together closely in the t-SNE visualization. However, if they are diverse enough, we would see these data sets form distinct clusters within the t-SNE plot. In Figure 5.1 can be seen that every data set has its own cluster, which means that the natural language descriptions of different data sets do not have much overlap. This confirms that it is useful to evaluate our workflow on multiple data sets, to make sure that our workflow works on a larger variety of regex descriptions. The KB-13-emoji data set has its own cluster and is the closest related to the KB-13 data set, which is as expected since KB-13-emoji is a modification of the data set. Furthermore, it can be seen that the StructuredRegex data set is the least similar to any other data set, which was the goal of the creators of the data set.

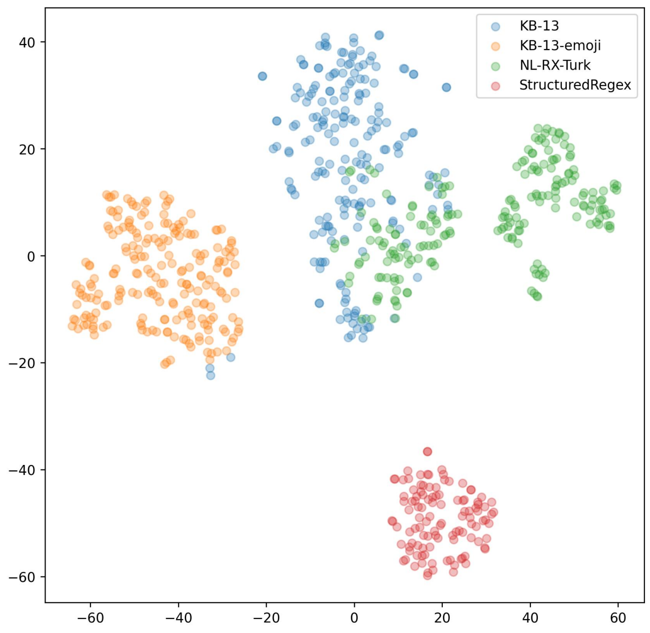

Figure 5.1: t-SNE analysis of the natural language description embeddings. The analysis shows that every data set has its own cluster, indicating that the data sets contain diverse regular expression descriptions.

5.3. LMAPS accuracy

To answer the question if LLMs can improve the accuracy of traditional program synthesis, we create an experiment to test the full LMAPS workflow. The first step in this experimental setup is creating a program specification for every task in the regex data sets. These tasks consist of a description and a regular expression solution. These solutions are used to generate random positive and negative examples that can be used in the LLM prompt. We use multiple prompts that can generate regular expressions as the complexity of program synthesis lends itself to diverse approaches in the prompt formulation. These prompts are then ran on all the data sets, to generate the base LLM prediction programs. Building blocks are extracted from these programs to create an optimized grammar for the inductive program synthesiser. This grammar is then compared to a traditional regex grammar to measure the accuracy improvement. The accuracy of the LMAPS and PBE grammars are calculated by checking if the solution can be synthesized with a depth of 6 or less.

The PBE-only method reached an average accuracy of 31.9% on the KB-13, NL-RX and StructuredRegex data sets. In Table 5.3 can be seen that LMAPS has an average accuracy of 57.0, which is 25.1% higher than the PBE-only approach. The LMAPS improves the base LLM prediction by 30.7%, this show that the inductive program synthesiser can improve the base LLM prediction by synthesising a program using an optimized grammar with building blocks from the LLM predictions. The LMAPS grammar is on average 8 times smaller than the traditional regex grammar, so, in most cases, can search deeper than the traditional regex grammar. This means that the accuracy difference between these methods is potentially even higher than 25.1%.

All prompts are created and tested by humans except for the APE prompts. An analysis of the prompts used in this research is done in Section 5.7. The exact prompts can be found in Appendix A.

Prompt identifier	LLM-only	LMAPS	Δ LLM-only	Δ PBE-only
Baseline 1	22.0	54.8	32.8	22.9
Baseline 2	19.4	52.5	33.1	20.6
APE 1	22.4	55.2	32.8	23.3
APE 2	19.4	53.6	34.2	21.7
Task-based	17.5	53.2	35.7	21.3
Requirements-based instruction	18.5	52.5	34.0	20.6
Requirements-based completion	19.3	53.2	33.9	21.3
Simple instruction + positive examples	25.3	57.4	32.1	25.5
Simple instruction + quoted positive examples	23.0	55.6	32.6	23.7
Positive examples	27.5	56.6	29.1	24.7
Positive and negative examples	25.0	56.2	31.2	24.3
Few-shot baseline	18.3	51.1	32.8	19.2
Few-shot similarity lookup	55.2	73.5	18.3	41.6
Few-shot similarity lookup + positive examples	54.2	71.9	17.7	40.0
Mean	26.2	57.0	30.7	25.1

Table 5.3: The LLM-only columns shows the accuracy (%) per prompt on the KB-13, NL-RX and StructuredRegex data set of the generated programs by using an LLM. The LMAPS accuracy is compared to the PBE-only and LLM-only by calculating the accuracy increase. The LMAPS workflow has, on average, a 25.1% higher accuracy than the PBE-only approach and a 30.7% higher accuracy over the base LLM prediction.

5.3.1. Accuracy per data set

Comparing LMAPS accuracy on individual data set is important to evaluate if the workflow works for different types of regex problems. We evaluate this by using the 'Positive examples' prompt as it is a representative prompt for many use cases, as programming by example tasks need to have some positive examples. From Figure 5.2 can be seen that LMAPS has a higher accuracy than PBE-only and the base LLM predictions. However, just combining two methods will in most cases increase the accuracy as both methods solve slightly different problems than the other. To measure the synergy effect we compare the accuracy of LMAPS with a combined LLM and PBE method that checks if either LLM or PBE can solve it, without using an optimised grammar. This synergy effect can be seen in Figure 5.3, where the LMAPS approach performs on average 11.3% better than the methods combined.

The KB-13-emoji data set has a higher accuracy than the other data sets for the base LLM prediction, PBE-only and LMAPS. This is mainly because the data set is less complex than the other data sets. However, it does show the capabilities of LLMs to generalize to new domains, like regular expressions with emojis.

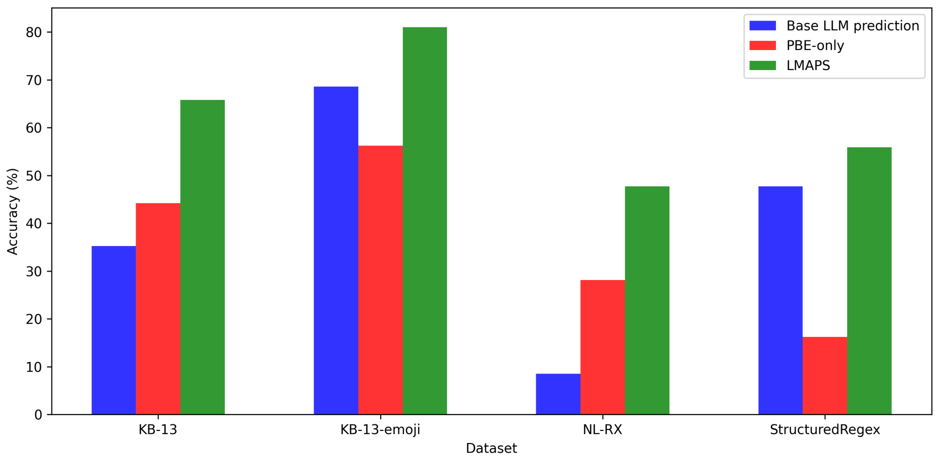

Figure 5.2: Accuracy per data sets for different methods for the **Positive examples** prompt.

Combining two methods will in most cases reach a higher accuracy, as only one of the models has to be correct. When combined, they often complement each other because if one model can not find the solution, there is still a chance that the other model can find it. We analyzed if LMAPS performs better than the combination of the base LLM prediction and PBE-only approach. This is done to analyze if the LMAPS workflow has a synergy effect of combining the strengths of LLMs and programming by example. The combined accuracy of the 'Positive examples' prompt is 54.0% and 64.7% on the KB-13, KB-13-emoji, NL-RX and StructuredRegex data sets. LMAPS, on average, has 9.7% higher accuracy than the methods combined, which shows the synergy of LLMs and the PBE engine.

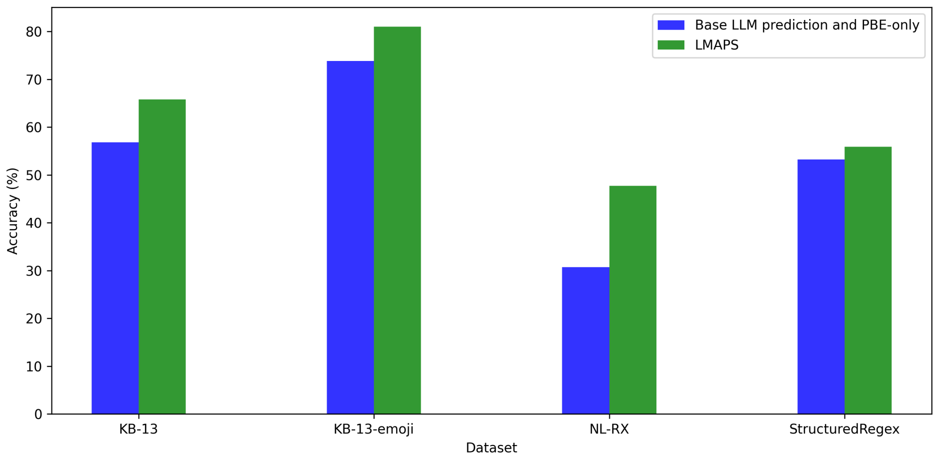

Figure 5.3: LMAPS reaches a higher accuracy than the combination of the base LLM prediction and PBE-only approach.

We also analyze the relationship between the base accuracy of different prompts and the accuracy of LMAPS. A higher base LLM accuracy implies that the prompt is better at the given tasks, which in most cases means better building blocks that can be given to the inductive program synthesis grammar. Our experiments show that prompts that have a higher base accuracy tend to produce better building blocks, which contribute to increased accuracy of the synthesised programs, as can be seen in Figure 5.4.

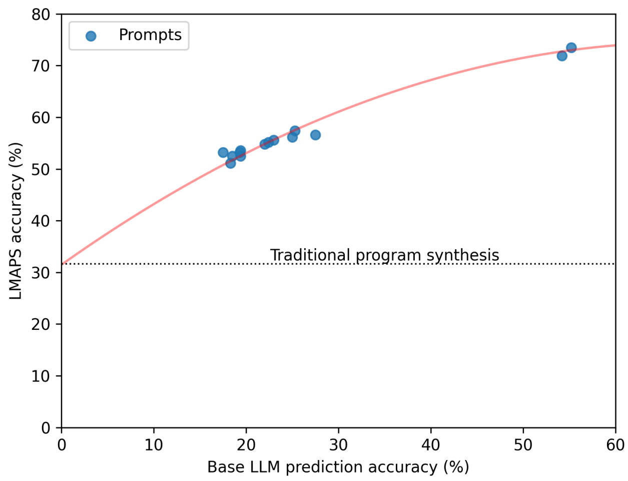

Figure 5.4: Higher accuracy LLM base predictions produce higher-quality building blocks, which in turn increases the accuracy of the program synthesisers. The curve flattens as there are some tasks in the data sets that require a high depth to be solved even with well-defined building blocks, which makes these tasks still unsolvable by LMAPS.

The results of this study show a clear improvement in the accuracy of program synthesis when using LMAPS compared to traditional PBE methods. The experiments demonstrated that LMAPS outperforms both PBE and LLM methods individually on various data sets, as it leverages the strengths of both. Another insight from the study is that the quality of the building blocks extracted from LLM programs is strongly related to the base accuracy of the prompts. Prompts with higher base accuracy tend to produce superior building blocks, thereby increasing the accuracy of the synthesised programs.

5.4. LMAPS efficiency

In program synthesis, the depth needed to find a solution is highly related to the complexity of the synthesized program logic. Without enough depth, the model might fail to solve problems that require a more elaborate logic. When the depth is increased, the model its search space broadens, enabling it to synthesize a wider variety of programs. The benefits of increased depth come at the cost of a larger search space, translating into more programs the model must explore. That is why enhancing the grammar to reduce the depth needed to reach a solution makes the search more efficient.

Another critical aspect of efficient inductive program synthesis lies in the formulation of a well-structured grammar. An optimized grammar can find the correct program at a comparatively lower depth. Our optimized grammar with building blocks can significantly limit the combinatorial explosion of the search space typically seen with increased depth. This reduction enables the synthesizer to find the correct program more efficiently, thus enhancing the overall efficiency of the synthesis process. Furthermore, synthesizers often operate within strict computational constraints, particularly in scenarios where it is not feasible to allow extensive search durations. For practical reasons, a timeout is commonly implemented to limit the computational resources. Given this timeout, the efficiency gains from an optimized grammar can significantly impact the synthesizer its ability to produce a correct program within the allotted time, thereby also increasing the accuracy.

An optimized grammar that includes the building blocks of candidate LLM programs has a smaller search space and can find programs at lower depths. As many synthesis processes are time-constrained, the ability to find a solution faster boosts the likelihood of finding a correct program before hitting the time limit. Therefore, optimization of the grammar not only enhances the speed and efficiency of the synthesis process but also has an impact on its accuracy.

We do three experiments to analyze if LLM-generated programs can improve the efficiency of traditional program synthesis by reducing the search space. First, we check how many solutions can be generated with the new grammar. Secondly, we analyse the depth to reach a solution with the new grammar compared to traditional regex grammar, as a lower depth to reach a solution makes the search more efficient. Lastly, we compare the search space of traditional PBE systems with the new workflow, as in general it is harder to find a solution when the search space is larger. Based on these experiments, the efficiency of LMAPS is evaluated.

A drawback of more concise optimized grammar is that it no longer can create all possible regular expressions. In this experiment, we measure how many optimized grammar still contain all the necessary elements to synthesize the solution, regardless of the depth. This is done by analysing every problem from regex data sets to determine if the solution can potentially be reached. The common quantifiers and common literals are added to improve the grammar as the extract building blocks are only able to synthesize the solution in 31.2%. In Table 5.4 can be seen that adding common literals and quantifiers to the building blocks increases the percentage of solutions the grammar can synthesize from 32.4% to 95.6%. We define grammar size as the total number of literals and quantifiers used in the grammar. The grammar size changes based on the number of building blocks extracted from the LLM programs. The 'Extracted building blocks + common literals + common quantifiers' on average has a grammar size of 21.0, of which, on average, 14.5 building blocks and 6.5 quantifiers. In comparison, a traditional regex grammar contains 100+ literals and quantifiers, as it has to include all lower-case and upper-case letters, digits and special characters. However, a traditional regex grammar can synthesize all solutions as it contains all the elements a regular expression can consist of.

Common literals = **[A-Z], [A-Za-z], [a-z], [0-9], [AEIOUaeiou], .**
Common quantifiers = **{0,}, {1,}, {0,1}**

These common and building block literals and quantifiers are then used in the following grammar:

```
RegexConnector = Regex + RegexConnector
RegexConnector = Regex
Regex = Literal or Literal + Quantifier
Literal = <List of literals>
Quantifier = <List of quantifers>
```

Grammar 5.1: Simplified regular expression grammar which can incorporate building blocks, consisting of literals and quantifiers.

Grammar	Possible to synthesize program
Common literals + common quantifiers	7.7%
Extracted building blocks	32.4%
Extracted building blocks + common quantifiers	34.6%
Extracted building blocks + common literals	78.4%
Extracted building blocks + common literals + common quantifiers	95.6%

Table 5.4: A grammar with extracted building blocks from LLM candidate programs can only find a solution in 32.4% of the cases. Adding often-used regex building blocks to the grammar makes it possible to synthesize a program in 95.6% of the cases.

The number of building blocks that get extracted from the LLM predictions increases with the complexity of the task. This means that when a larger depth is needed to construct a solution, the LLM, on average, also generates programs that contain more building blocks. In Figure 5.5 can be seen that a smaller depth to reach the solution also requires fewer building blocks. This show that the complexity of the problem has a huge effect on the search space, as more complex problems need a larger depth and a larger grammar size to be solved.

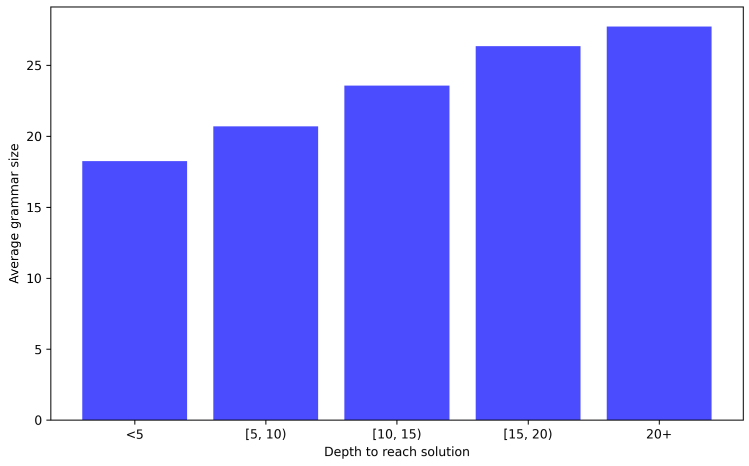

Figure 5.5: The average grammar size compared to the depth to reach a solution. The generated LLM programs contain fewer building blocks for less complex programs, resulting in a small grammar size.

Being able to find correct programs at lower depths improves the efficiency significantly as the search space grows exponentially with the depth. Solutions that require less depth are therefore, in general, much faster to compute. Creating a well-structured, comprehensive grammar is crucial for optimizing the depth to reach a solution. An optimized grammar can essentially provide 'shortcuts' to the desired solution. A grammar designed to find solutions at lower depths provides a more constrained and directed search space for the solution. The depth to reach a solution is calculated by finding the minimum number of literals and quantifiers from the grammar needed to construct the regular expression solution. In Figure 5.6 can be seen that LMAPS can find solutions at much lowers depths than a traditional regex grammar. The LMAPS grammar contains fewer elements and can reach solutions at lower depths. This makes the search much more efficient than a traditional regex grammar. It is important to note that the new grammar can not reach a solution in 5.4% of the cases, as seen in Table 5.4. In these cases, a traditional regex grammar still has to be used.

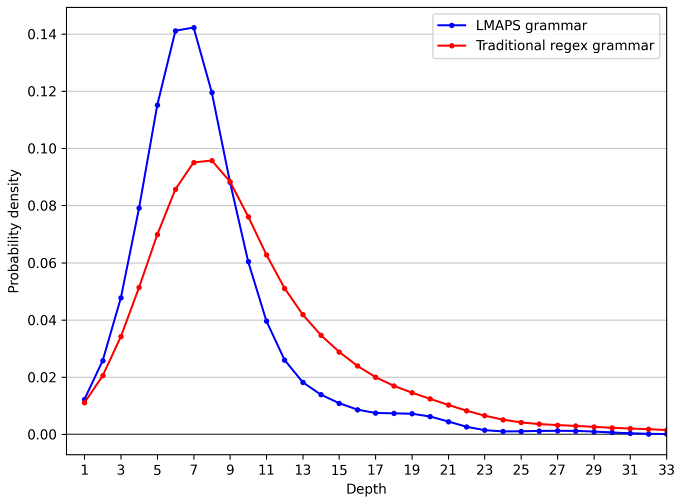

Figure 5.6: Comparison of the depth required to reach a solution for the LMAPS and traditional regex grammars. LMAPS can solve programs at a lower depth than traditional regex grammars.

Search space

The number of components in a grammar is a determining factor for the quantity of potentially synthesizable programs, hence presenting a direct correlation with the size of the search space. We compare the LMAPS grammar with a traditional regex grammar to analyze the search space reduction, as can be seen in Figure 5.7. At a depth of three, the search space of the LMAPS grammar is on average 545 times smaller compared to a traditional regex grammar. This huge reduction in the search space has a positive effect on the computational resource requirements and time efficiency in the PBE synthesis.

The advantage of the LMAPS grammar becomes even more apparent when tackling problems that need a higher depth to be solved due to the combinatorial explosion of the search space. The LMAPS grammar has a 43686 times smaller search space at a depth of five. This clearly shows the potential of the LMAPS grammar in a more efficient and effective PBE workflow.

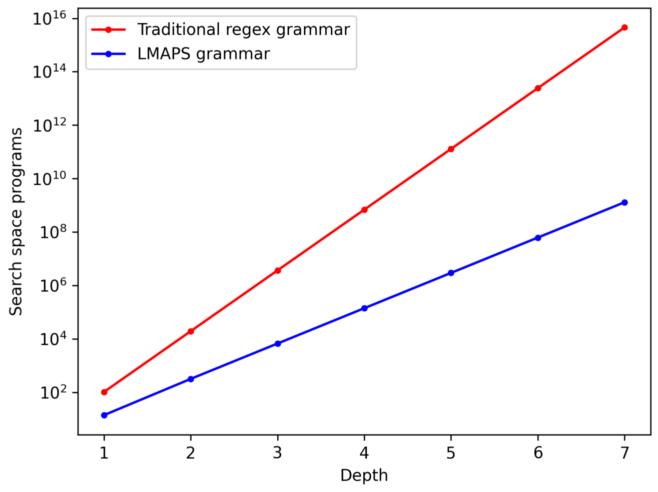

Figure 5.7: Search space comparison of the traditional regex grammar with the LMAPS grammar. The concise LMAPS grammar has a smaller search space than the traditional regex grammar, which becomes even more apparent at higher depths.

5.5. Generating examples

Creating a representative program specification of the problem is crucial in getting the program synthesiser to produce the intended program. LLM can potentially help to improve the program specification by generating examples that provide additional information to specification. A key difficulty in measuring the additional information provided by each example in PBE is the inherent variety of possible examples. Examples can vary significantly in terms of their complexity, structure, and relevance to the problem at hand. Some examples might offer unique insights into the problem, while others might only confirm already understood patterns. Another difficulty arises from the inherent complexity of the underlying problem. If the problem requires complex logic, even many examples may not suffice to capture the entire problem space. Consequently, an example that seems to provide little information might actually be vital for solving a complex problem.

Given the aforementioned challenges, we quantify the relative effectiveness of adding examples by measuring the reduction in programs that satisfy them. With these experiments, we evaluate if LLMs can aid in the enhancement of problem specifications through example generation. When new examples are added to a program specification, they often eliminate a number of incorrect programs from the set of programs that did satisfy the specification before the example was added. A reduction in the number of programs that satisfy the new program specification means that added examples are effective. Conversely, if introducing a new example does not affect the number of programs that satisfy the program specification, it implies that the example has offered no additional information to the already existing set.

To understand the effectiveness of LLMs in generating examples for PBE, we create an experimental setup to compare the LLM approach against two alternative methods. The experimental setup and the obtained results are presented herein. We consider three distinct example-generation methods. The first method generates examples with an LLM as used in LMAPS. The second method samples random positive examples from the regex solution and generates completely random negative examples. It's important to note that in real-world scenarios, the solution isn't known in advance. Therefore, if our proposed method achieves similar or better results compared to this approach, it can already be considered beneficial. Nevertheless, a limitation of randomly generated examples is that they may not effectively cover edge cases. The third method involves sampling random positive examples from the solution and creating near-miss negative examples based on these positive examples using algorithms that make small variations. This approach effectively covers a lot of edge cases, as near-miss negative examples, in general, help to create a well-defined program specification. However, it costs human knowledge and time to create an algorithm to generate high-quality near-miss negative examples and other meaningful variations. To keep the comparison fair across the experimental setup, we generated the same number of positive and negative examples across each method, matching the quantity generated by the LLM for each problem.

To measure the reduction in correct programs, we first generate all correct programs that satisfy a certain specification based on the tasks from the regex data sets. The initial program specification is made by sampling three random positive examples from the regex solution and generating one random negative example. This specification is then given to a program synthesiser to find all programs that match it. After that, we update the program specification with the newly generated examples per method and count the reduction of programs that satisfy the specification. The more programs that are no longer 'correct', the more information the examples gave to the program specification.

From the results in Figure 5.8, it is clear that the LLM-generated examples outperformed the random examples, with a program count reduction improvement of 9.4%. This suggests that LLMs have an inherent capability of generating meaningful examples. However, they did not surpass the effectiveness of the smart negative example variation examples, which require human knowledge to create valuable variations. The superior performance of the smart negative example variation algorithm emphasizes the value of human intuition and expertise in defining near-miss examples to cover edge cases.

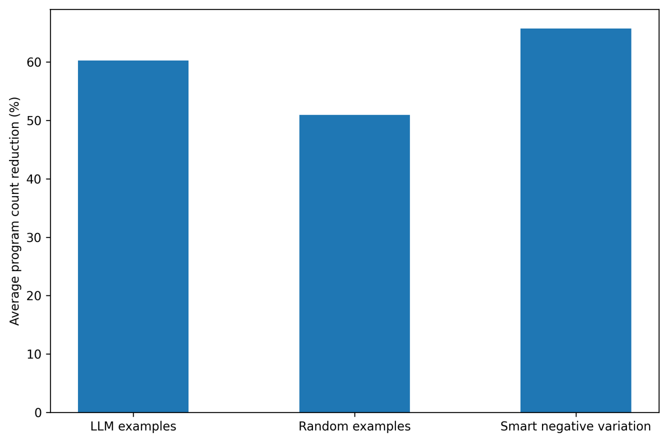

Figure 5.8: The average search space reduction for three different example generation methods. LLM-generated examples are better at reducing the number of programs that satisfy the specification than randomly generated ones. However, the human-crafted negative example variation algorithm still outperforms the LLM.

Despite not reaching the performance of human-crafted algorithms, the LLM its relative success over random examples underscores its potential to create well-defined program specifications. As advancements in AI continue, we anticipate that the gap between LLMs and human-crafted examples will continue to narrow.

5.6. Explaining programs

Our research investigated the capability of an LLM to explain regular expressions to humans as part of the Language Model Augmented Program Synthesis workflow. In this experiment, we analyze to what extent LLMs are capable of explaining the synthesized programs. This is done by manually analyzing if the explanation of the regex solutions in the KB-13-emoji data set were correct. This experiment showed that ChatGPT accurately explains 94% of the regular expression of the KB-13-emoji data set. LLMs can thus help to understand the generated program in the LMAPS workflow. Nevertheless, users always have to be careful when they rely on the output of LLM, given that in 6% of the cases, ChatGPT provided inaccurate or incomplete explanations. A thorough, independent understanding of the underlying principles of programming and regex remains essential. This balance between reliance on LLMs and a solid foundational understanding of programming ensures that users can effectively harness the power of PBE while mitigating the risks of misinterpretation or misuse of generated programs. The full prompt can be found in Appendix A.3.

It is important to note that regular expressions are commonly used in programming which is part of the reason why LLMs are so good in explaining regular expressions. Another reason is that regular expressions often contain the same building blocks like'.*' and [A-Z], which means that ChatGPT probably has seen many of these. This experiment does not show that LLMs can also explain programs for different programming or domain-specific languages. It does show its usefulness in LMAPS for regular expression and its potential for new domains.

5.7. Prompt analysis

By analysing varied prompt methodologies, we can better understand how to approach and optimize prompt formulation in the field of program synthesis. Better prompts increase the accuracy of LMAPS and help to overcome the 'efficiency' limitation as shown in Section 5.3 and 5.4. The performance is measured in terms of the base LLM prediction and the final LMAPS accuracy. Table 5.5 provides an overview of the performance of different prompt strategies in the LMAPS method. These strategies range from basic prompts without any examples to more advanced few-shot prompts that give more context to the prompt by providing similar examples.

Prompt descriptions:

- **Baseline 1** and **Baseline 2** are basic prompts with no other information like positive examples.
- **APE 1** and **APE 2** are generated by the Automatic Prompt Engineer method.
- **Task-based**, **Requirements based instruction**, and **Requirements based completion** use more detailed instructions than the baselines.
- **Simple instruction + positive examples**, **Simple instruction + quoted positive examples**, and **Positive examples** consist of a task description and positive examples. This will

be an often used prompt because it is a good performing and relatively simple prompt, which does need any other components like similar example lookup. The **Positive and negative examples** prompt also includes negative examples in this same prompt.

- **Few-shot baseline** uses a few-shot prompt with random description-regex solution pairs from any of the data sets. The **Few-shot similarity lookup** and **Few-shot similarity lookup + positive examples** prompts use a more advanced method to look up similar description-regex pairs based on the cosine similarity of the embeddings.

In Table 5.5 can be seen that the Few-shot similarity lookup method shows a significant leap in accuracy, especially in the base LLM predictions, reaching accuracies of 54.2% and 55.2%. The LMAPS accuracy for these strategies is also the highest among the evaluated strategies. Overall, the results indicate that prompts enriched with similarity lookups or positive examples enhance both LLM predictions and LMAPS accuracy, underscoring the utility of these strategies in LMAPS. The variety in the results of different prompt strategies shows the importance of prompt design and selection for the LMAPS method.

	Base LLM prediction	LMAPS	Δ Improvement
Baseline 1	22.0	54.8	32.8
Baseline 2	19.4	52.5	33.1
APE 1	22.4	55.2	32.8
APE 2	19.4	53.6	34.2
Task-based	17.5	53.2	35.7
Requirements-based instruction	18.5	52.5	34.0
Requirements-based completion	19.3	53.2	33.9
Simple instruction + positive examples	25.3	57.4	32.1
Simple instruction + quoted positive examples	23.0	55.6	32.6
Positive examples	27.5	56.6	29.1
Positive and negative examples	25.0	56.2	31.2
Few-shot baseline	18.3	51.1	32.8
Few-shot similarity lookup	55.2	73.5	18.3
Few-shot similarity lookup + positive examples	54.2	71.9	17.7
Mean	26.2	57.0	30.7

Table 5.5: The average accuracy per prompt on the KB-13, NL-RX and StructuredRegex data sets for the base LLM prediction and LMAPS method. The 'Improvement' column shows the difference between the LMAPS accuracy and the base LLM prediction accuracy. The prompts can be found in Appendix A.

A more in-depth analysis is done to compare the difference between different types of prompts. Analyzing prompts can give insights into how changes in the prompts affect the model its output behaviour. This can aid in improving the model responses and can help in mitigating risks associated with unintended outputs. By comparing the outputs of similar prompts, we can analyze the effect of adding positive, negative and few-shot examples to the prompt, which is useful for future prompt design and makes it easier to decide which prompt type best suits a specific use case.

Automatic prompt generation component

The performance of the APE prompts is analyzed to measure if automatically generated prompts perform worse than human-created ones. The prompt generation component (4.3.1), which uses the Automatic Prompt Engineer (APE) method, performs as well as human-created baselines. Zhou et al. (2023) [27] concluded that APE-generated prompts could perform comparably to human-designed prompts. Our results show that APE-generated regular expression generation prompts indeed perform on par with human-created baseline prompts, as shown in Table 5.6. This finding suggests that the time and effort typically required to craft high-quality manually created prompts could be saved by using APE without sacrificing the performance of the system. The APE system can generate and test many prompts in the time it takes a human to create a single one. However, for more advanced prompts, human prompt engineering is still required.

	Base LLM prediction	LMAPS	Δ Improvement
APE 1	22.4	55.2	32.8
APE 2	19.4	53.6	34.2
Baseline 1	22.0	54.8	32.8
Baseline 2	19.4	52.5	33.1

Table 5.6: Generated prompts with the APE method perform as well as human-created baseline prompts.

Negative examples in prompts

To measure the effect of adding negative examples to a prompt, we compare the prompt with only positive examples to the one with both positive and negative examples. The accuracy of prompts with only positive examples is higher than that of prompts with both positive and negative examples in both the base prediction as LMAPS workflow, as can be seen in Table 5.7. This suggests that adding negative examples may not be beneficial for the accuracy of these prompts and can even decrease the accuracy.

	Base LLM prediction	LMAPS	Δ Improvement
Positive examples	27.5	56.6	29.1
Positive and negative examples	25.0	56.2	31.2

Table 5.7: The accuracy of the prompts with only positive examples is higher than the prompt with both positive and negative examples.

Few-shot prompts

Few-shot prompts are used to provide context that guides the model in generating appropriate responses, especially when dealing with tasks that require specific knowledge or a particular style of response. They essentially act as a primer, helping the model to better interpret and adapt to the desired task. Our experiment compares the performance of three different few-shot prompts to check the effect of using a similarity lookup to add similar description-regex pairs to the prompt, as seen in Table 5.8.

- **Few-shot base:** This prompt includes randomly chosen description-regex pairs from any of the data sets.
- **Few-shot similarity lookup:** This prompt similar description-regex pairs from the data sets, which are calculated via cosine similarity. This ensures the selection of more relevant few-shot examples, resulting in significantly improved accuracy over the few-shot base propmpt.
- **Few-shot similarity lookup + positive examples:** This prompt is the same as the 'Few-shot similarity lookup' prompt with the addition of positive examples.

Using relevant description-regex pairs significantly enhances the accuracy of few-shot prompts, with the prompt using the similar example lookup performing considerably better than the one using random examples. Adding both positive examples and similar description-regex pairs slightly decreases the prompt accuracy.

It is important to note that the similarity lookup often uses data points from the same data set as the current description. This could potentially inflate the accuracy of this method in comparison with real-world unseen data points as data points from the same data set are often similar, as shown in Section 5.2.1. In diverse real-world data, the looked-up description-regex pairs are potentially less similar, which would result in lower accuracy.

	Base LLM prediction	LMAPS	Δ Improvement
Few-shot baseline	18.3	51.1	32.8
Few-shot similarity lookup	55.2	73.5	18.3
Few-shot similarity lookup + positive examples	54.2	71.9	17.7

Table 5.8: Comparison of the average accuracy of the few-shot prompts. The few-shot prompt with similarity lookup performs significantly better than the few-shot baseline. Including positive examples in the similarity lookup prompt does not increase its accuracy.

6 Discussion

The results from our study show that LMAPS is a potent method for multimodal program synthesis, providing improvements over traditional regular expression generation methods. Even though little research is done in this area, our results indicate that combining LLMs with PBE can address several of the limitations of traditional PBE workflows.

By harnessing the power of LLMs in three distinct stages of program synthesis, our approach manages to mitigate the challenges of ambiguity, overfitting, underfitting, computational efficiency, explainability, and single modality associated with PBE. The significant improvements of our LMAPS method signal the untapped potential that lies in the synergy of LLMs and PBE.

While the LMAPS workflow showed significant improvement over traditional PBE workflow, we recognize it does have limitations. First, creating a fully unambiguous specification using LLMs is not a trivial task. Even a well-formulated problem description can result in ambiguous or non-optimal programs due to the inherent nature of language itself. Additionally, the model its capacity to generate diverse IO examples is not without faults. Human review is needed to create a well-defined program specification and to check whether the generated program works as intended. Secondly, the results can vary significantly depending on the complexity of the task, the capabilities of the LLM, and the efficiency of the inductive program synthesis engine. This should be included in the consideration of using LMAPS. Thirdly, although our method introduced multi-modality to the process, there still exists a question of how well the LLM can comprehend and integrate information from various modalities. We know current models are relatively robust in the text modality but might not be as strong in other modalities, such as interpreting equations and tables. Further research is necessary to realize a fully multimodal workflow. Lastly, the added complexity introduced by the LLM in the workflow also has to be considered. While the LLM assists in the generation of programs, diverse IO examples, and explaining the generated programs, it presents an additional layer of complexity that must be managed.

LLM development limitation
One significant limitation is the intensive computational power required by LLMs which is not available or affordable for most researchers and organizations. The development, training,

and deployment of LLMs involve significant financial investment. The resources necessary for model training, including computational hardware and energy, can be costly. This expense can pose a barrier to research and application development. Furthermore, comparing LMAPS performance to previous programming synthesis benchmarks may not always be fair due to differences in computational requirements. LLMs require huge amounts of diverse and high-quality data to be trained effectively. The necessity for such large and complex data sets may limit the accessibility and usability of these models, particularly for researchers and organizations that do not have access to such data resources. The computational resources, high costs and data requirements make it infeasible to create their own LLM for most people. This creates a dependency issue when relying on a third-party LLM, as users have to trust the security, availability and data control of another company. Sending sensitive data through a third-party model can raise privacy concerns.

LLM capabilties limitation
LMAPS its effectiveness is inherently bound to the capabilities of the LLM used, limiting the universal applicability of our method. Our research is mainly focused on regular expression synthesis, and more research has to be done to evaluate the effectiveness of LMAPS on new domains. Certain tasks that are seemingly simple and straightforward for humans are very difficult for LLMs due to their architecture. For instance, tasks such as counting the number of words in a text are challenging for current LLMs. This limitation shows the importance of understanding and testing the capabilities of an LLM before using it in real-world scenarios. As LLMs are black-box models, it can be difficult to understand why they generate something. Any limitations or shortcomings intrinsic to the LLM, such as biases in training data or inability to handle certain task types, negatively affect the LMAPS workflow. This lack of transparency can be problematic in certain domains where interpretability is important.

As current-generation LLMs are only trained on text, they have no real-world experience or sensory understanding. They do not understand or know anything about the world in the way humans do. The model generates responses based on patterns learned from training data, which may contain errors, biases, or misinformation. Because of this, LLMs sometimes generate inaccurate or misleading information, even when asked factual questions. Additionally, LLMs do not always comprehend nuanced or complex contexts, leading to nonsensical or irrelevant responses.

Future work should aim to refine the integration of LLMs in the programming by example workflow and further investigate the challenges of ambiguity, generalizability, efficiency, explainability, and multi-modality in this context. A closer look at the capabilities of LLMs in program synthesis and understanding how these models work can lead to significant advances in the field of program synthesis.

Practical applications
Practical applications of LMAPS extend across various sectors. For example, LMAPS could be integrated into programming environments to assist developers in writing code, debugging, and maintaining software systems. It can help reduce the time spent on coding and debugging by generating code snippets, refactoring existing code, and providing suggestions based on

the problem description and input-output examples. LMAPS would potentially work well in Test-Driven Development, a programming technique that involves writing tests before you write the code to fulfil them. Each test essentially provides a specific example of how your code should behave under a given set of conditions, effectively acting as the program specification for the LMAPS workflow.

LMAPS could also help professionals who are not trained programmers, such as data analysts or consultants, but do need to write scripts for automation. LMAPS could provide significant assistance by synthesizing code based on their problem descriptions and examples, making programming more accessible to a wider audience. This is possible because the PBE engine and LLMs both provide an intuitive interface, consisting of IO examples and a language description, that can be used without programming knowledge.

Future of LLMs in Program Synthesis
The immense potential for the growth and development of LLMs, particularly in the domain of program synthesis, holds a promising future. As these models continue to evolve, we anticipate multiple improvements. The next generation of LLMs will likely enhance their grasp of context, leading to a better understanding of human intent, which is key to synthesizing accurate, efficient programs. We foresee these improved models being capable of handling more complex programming tasks, demanding a deeper understanding of the problem and advanced reasoning capabilities. As our understanding and refinement of LLMs progress, we can expect to see models that better generalize to new domains. Our LMAPS workflow would greatly benefit from improved language understanding capabilities of LLMs, resulting in a better example, program and explanation generation. This could lead to a significant improvement in the final generated programs by LMAPS. The future of programming looks set to be an exciting blend of human creativity and AI efficiency, leading to faster development times, fewer errors, and potentially, more innovative programs.

7 Conclusion

Our research explored the use of LLMs to augment PBE workflows, aiming to overcome the inherent limitations of PBE. LLMs have been shown to be effective in augmenting program synthesis workflows by generating IO examples, programs and explaining the generated programs on the task of generating regular expression. Using four regular expression data sets, we conducted an in-depth analysis to evaluate the efficacy of our LMAPS method. By leveraging the natural language processing capabilities of LLMs, we were able to create a workflow that can address the key limitations of programming by example, notably ambiguity, overfitting, underfitting, computational efficiency, explainability and single modality.

The results of our experiments show that LMAPS significantly improves the accuracy of program synthesis compared to traditional PBE methods. LMAPS outperforms both PBE and LLM methods individually, leveraging the strengths of both. The accuracy improvement achieved by LMAPS is substantial, with an average increase of 25.1% compared to the PBE-only approach, up to a 40% increase by using more advanced prompts. By utilizing LLM-generated programs to extract building blocks, an optimized grammar is created. This optimized grammar allows for finding correct programs at lower depths, resulting in more efficient synthesis. The LMAPS workflow effectively reduces the search space and improves the efficiency of program synthesis.

LLMs also show the potential in assisting in the process of creating well-defined problem specifications through example generation. The LLM-generated examples contribute valuable information to the program specification, resulting in a reduction of programs that satisfy the specification. While LLM-generated examples demonstrate effectiveness, they do not surpass the performance of human-crafted algorithms for generating near-miss negative examples. Human intuition and expertise in creating near-miss examples to cover edge cases still achieve superior results. However, the relative success of LLM-generated examples does highlight their potential to enhance program specifications. Lastly, LLMs are also capable to explain synthesized programs, as demonstrated by the accurate explanations of regular expressions of the KB-13-emoji data set. The LLM explanations were correct in 94% of the cases, providing a valuable tool for understanding the generated programs. However, LLM should never be blindly trusted, as in 6% of the cases the LLM explanations were inaccurate.

The synergy of LLMs and traditional program synthesis holds great promise for advancing the field of program synthesis by providing an effective way to overcome the limitations of programming by example.

References

[1] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. “Recursive program synthesis”. In: *Computer Aided Verification: 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings 25*. Springer. 2013, pp. 934–950.

[2] Rohan Anil et al. *PaLM 2 Technical Report*. 2023.

[3] Jacob Austin et al. “Program synthesis with large language models”. In: *arXiv preprint arXiv:2108.07732* (2021).

[4] Alberto Bartoli et al. “Automatic synthesis of regular expressions from examples”. In: *Computer* 47.12 (2014), pp. 72–80.

[5] Mark Chen et al. “Evaluating large language models trained on code”. In: *arXiv preprint arXiv:2107.03374* (2021).

[6] Qiaochu Chen et al. “Multi-modal synthesis of regular expressions”. In: *Proceedings of the 41st ACM SIGPLAN conference on programming language design and implementation*. 2020, pp. 487–502.

[7] Kevin Ellis and Sumit Gulwani. “Learning to Learn Programs from Examples: Going Beyond Program Structure.” In: *IJCAI*. 2017, pp. 1638–1645.

[8] Yu Feng et al. “Program synthesis using conflict-driven learning”. In: *ACM SIGPLAN Notices* 53.4 (2018), pp. 420–435.

[9] Zhangyin Feng et al. “Codebert: A pre-trained model for programming and natural languages”. In: *arXiv preprint arXiv:2002.08155* (2020).

[10] John K Feser, Swarat Chaudhuri, and Isil Dillig. “Synthesizing data structure transformations from input-output examples”. In: *ACM SIGPLAN Notices* 50.6 (2015), pp. 229–239.

[11] Sumit Gulwani. “Automating string processing in spreadsheets using input-output examples”. In: *ACM Sigplan Notices* 46.1 (2011), pp. 317–330.

[12] *Herb Program Synthesis Framework*. `https://github.com/Herb-AI/Herb.jl`. 2023.

[13] Nate Kushman and Regina Barzilay. “Using semantic unification to generate regular expressions from natural language”. In: North American Chapter of the Association for Computational Linguistics (NAACL). 2013.

[14] Vu Le and Sumit Gulwani. “Flashextract: A framework for data extraction by examples”. In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2014, pp. 542–553.

[15] Brian Lester, Rami Al-Rfou, and Noah Constant. “The Power of Scale for Parameter-Efficient Prompt Tuning”. In: *CoRR* abs/2104.08691 (2021).

[16] Yeting Li et al. “TransRegex: Multi-modal Regular Expression Synthesis by Generate-and-Repair”. In: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE. 2021, pp. 1210–1222.

[17] Yujia Li et al. “Competition-level code generation with alphacode”. In: *Science* 378.6624 (2022), pp. 1092–1097.

[18] Nicholas Locascio et al. “Neural generation of regular expressions from natural language with minimal domain knowledge”. In: (2016).

[19] OpenAI. *GPT-4 Technical Report*. 2023.

[20] OpenAI. *OpenAI Models Codex*. 2023. URL: `https://platform.openai.com/docs/models/gpt-3` (visited on 05/03/2023).

[21] Laria Reynolds and Kyle McDonell. “Prompt programming for large language models: Beyond the few-shot paradigm”. In: *Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems*. 2021, pp. 1–7.

[22] David E Shaw, William R Swartout, and C Cordell Green. “Inferring LISP Programs From Examples.” In: *IJCAI*. Vol. 75. 1975, pp. 260–267.

[23] Hugo Touvron et al. *LLaMA: Open and Efficient Foundation Language Models*. 2023.

[24] Ashish Vaswani et al. “Attention Is All You Need”. In: *CoRR* abs/1706.03762 (2017).

[25] Xi Ye et al. “Benchmarking multimodal regex synthesis with complex structures”. In: (2020).

[26] Xi Ye et al. “Optimal neural program synthesis from multimodal specifications”. In: *arXiv preprint arXiv:2010.01678* (2020).

[27] Yongchao Zhou et al. *Large Language Models Are Human-Level Prompt Engineers*. 2023.

A Prompts

A.1. Example generation

```
Create a list of examples that fully match this regular expression ‘{regex}‘? Only answer
    with a comma-separated list of examples without any other information.
```

Prompt A.1: Regex to examples

```
Create a list of examples that fully matches the following regular expression description ’{
    nl_description}‘? Only answer with a comma-separated list of examples without any other
    information.
```

Prompt A.2: Natural language to examples

A.2. Regex generation

```
Write a list of regular expressions that match the following pattern:
{nl_description}
```

Prompt A.3: APE 1

```
Create a list of regular expression patterns for the given input description:
{nl_description}
```

Prompt A.4: APE 2

```
List of regular expression patterns that match {nl_description}:
```

Prompt A.5: Baseline 1

```
Generate a list of regular expression patterns that match {nl_description} without giving an
    explanation or other information.
```

Prompt A.6: Baseline 2

```
Generate a list of matching regular expression patterns based on a description of the regex
    pattern.

Requirements:
- The list of matching regular expressions must be diverse to compensate for ambiguous
    descriptions.
- The regular expression patterns must fully match the strings described by the description.

Description: A sentence that contains the sequence 'dog12'
Regular expression: .*dog12.*

Description: A text that starts with "hello" and ends with "world"
Regular expression: hello.*world

Description: A string that consists of exactly 5 uppercase letters
Regular expression: [A-Z]{5}

Description: Lines that contain only letters and numbers, and are between 3 and 6 characters
    long
Regular expression: [A-Za-z0-9]{3,6}

List of regular expressions that match "{nl_description}":
```

Prompt A.7: Few-shot baseline

```
Generate a list of matching regular expression patterns based on a description of the regex
    pattern.

Requirements:
- The list of matching regular expressions must be diverse to compensate for ambiguous
    descriptions.
- The regular expression patterns must fully match the strings described by the description.

{io_pairs}

List of regular expressions that match {nl_description}:
```

Prompt A.8: IO examples

```
Generate a list of matching regular expression patterns based on a description of the regex
    pattern.

Requirements:
- The regular expressions must fully match all positive examples
- The list of matching regular expressions must be diverse to compensate for ambiguous
    descriptions.
- The regular expression patterns must fully match the strings described by the description.

{io_pairs}
{pos_examples}

List of regular expressions that match {nl_description}:
```

Prompt A.9: IO examples and positive examples

```
Generate a list of matching regular expression patterns based on a description of the regex
    pattern.

Requirements:
- The regular expressions must fully match all positive examples
- The list of matching regular expressions must be diverse to compensate for ambiguous
    descriptions.
- The regular expression patterns must fully match the strings described by the description.

{pos_examples}

List of regular expressions that match "{nl_description}":
```

Prompt A.10: Positive examples

```
Generate a list of matching regular expression patterns based on a description of the regex
    pattern.

Requirements:
- The regular expressions must fully match all positive examples
- The regular expressions do not match any negative examples.
- The list of matching regular expressions must be diverse to compensate for ambiguous
    descriptions.
- The regular expression patterns must fully match the strings described by the description.

{pos_examples}
{neg_examples}

List of regular expressions that match "{nl_description}":
```

Prompt A.11: Positive and negative examples

```
Generate a list of regular expression patterns that match {nl_description}.

Requirements:
- The list of matching regular expressions must be diverse to compensate for ambiguous
    descriptions.
- The regular expression patterns must fully match the strings described by the description.
- Only respond with a list of regular expressions without any explanations.

List of regular expressions that match {nl_description}:
```

Prompt A.12: Requirements based completion

```
Generate a list of regular expression patterns that match {nl_description}.

Requirements:
- The list of matching regular expressions must be diverse to compensate for ambiguous
    descriptions.
- The regular expression patterns must fully match the strings described by the description.
- Only respond with a list of regular expressions without any explanations.
```

Prompt A.13: Requirements based instruction

```
Generate a list of potentially matching regex based on a description without giving an
    explanation or other information. The regular expressions must match all positive
    examples.

{pos_examples}

List of regular expressions that match "{nl_description}":
```

Prompt A.14: Simple instruction + positive examples

```
Generate a list of potentially matching regex based on a description without giving an
    explanation or other information. The regular expressions must match all positive
    examples.

{pos_examples_quotes}

List of regular expressions that match "{nl_description}":
```

Prompt A.15: Simple instruction + quoted positive examples

```
Your task is to provide regular expression patterns based on the given description without
    any explanations.
List of regular expressions that match "{nl_description}":
-
```

Prompt A.16: Task-based

A.3. Program explaining

```
Task: Explain regular expression patterns.
This is a regular expression that matches the following description: {nl_description}.
Please explain how all parts of the regular expression work in detail: {regex}
```

Prompt A.17: A prompt to generate an detailed explanation of a regular expression