Multi-modal Program Inference: a Marriage of Pre-trained Language Models and Component-based Synthesis

KIA RAHMANI*, Purdue University, USA
MOHAMMAD RAZA, Microsoft, USA
SUMIT GULWANI, Microsoft, USA
VU LE, Microsoft, USA
DANIEL MORRIS, Microsoft, USA
ARJUN RADHAKRISHNA, Microsoft, USA
GUSTAVO SOARES, Microsoft, USA
ASHISH TIWARI, Microsoft, USA

Multi-modal program synthesis refers to the task of synthesizing programs (code) from their specification given in different forms, such as a combination of natural language and examples. Examples provide a precise but incomplete specification, and natural language provides an ambiguous but more "complete" task description. Machine-learned pre-trained models (PTMs) are adept at handling ambiguous natural language, but struggle with generating syntactically and semantically precise code. Program synthesis techniques can generate correct code, often even from incomplete but precise specifications, such as examples, but they are unable to work with the ambiguity of natural languages. We present an approach that combines PTMs with component-based synthesis (CBS): PTMs are used to generate candidates programs from the natural language description of the task, which are then used to guide the CBS procedure to find the program that matches the precise examples-based specification. We use our combination approach to instantiate multi-modal synthesis systems for two programming domains: the domain of regular expressions and the domain of CSS selectors. Our evaluation demonstrates the effectiveness of our domain-agnostic approach in comparison to a state-of-the-art specialized system, and the generality of our approach in providing multi-modal program synthesis from natural language and examples in different programming domains.

1 INTRODUCTION

In recent years, pre-trained language models (PTMs) have made major breakthroughs in natural language understanding. Models such as Google's BERT [Devlin et al. 2019] and OpenAI's GPT-3 [Brown et al. 2020] demonstrate the potential for *artificial general intelligence* (AGI), in how they provide a powerful basis for creating robust natural language applications without the need for significant domain-specific training. In particular, GPT-3 (generative pre-trained transformer) is a powerful model that can be viewed as an intelligent conversation completion engine: given some text in a so-called *prompt*, the model predicts the "most sensible" text that can follow that prompt. The predicted text tries to maintain the flow of the text in the prompt.

GPT-3 has generated a lot of excitement by enabling a wide variety of tasks through *few-shot learning* [OpenAI 2021]. Few-shot learning refers to the fact that the completion predicted by the model can be tuned by providing only a handful of completion examples in the prompt. For

*The first author worked on this paper during an internship with the PROSE team at Microsoft.

Authors' addresses: Kia Rahmani, Department of Computer Science, Purdue University, West Lafayette, Indiana, USA, rahmank@purdue.edu; Mohammad Raza, Microsoft, USA, moraza@microsoft.com; Sumit Gulwani, Microsoft, USA, sumitg@microsoft.com; Vu Le, Microsoft, USA, levu@microsoft.com; Daniel Morris, Microsoft, USA, Daniel.Morris@microsoft.com; Arjun Radhakrishna, Microsoft, USA, arradha@microsoft.com; Gustavo Soares, Microsoft, USA, Gustavo.Soares@microsoft.com; Ashish Tiwari, Microsoft, USA, astiwar@microsoft.com.

2021. XXXX-XXXX/2021/8-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

example, if the prompt contains some examples of natural language (NL) sentences being followed by the sentiment they convey, and the prompt ends with a sentence, then GPT-3 will predict the sentiment for that last sentence. It is able to do this surprisingly well because it has been trained on the huge amounts of data available on the web.

It is natural to wonder (as many indeed have in various internet discussions and blog posts) if few-shot learning with PTMs can be used to go from NL descriptions to code. For example, if we craft the prompt to include some examples of natural language descriptions followed by code, and then we end the prompt with a natural language description, GPT-3 will predict code that best completes the conversation presented in the prompt. Does that mean that GPT-3 has solved the challenge of generating code from natural language descriptions?

While the generality of PTMs is extremely powerful, it usually comes at the cost of limited precision. We observed that PTMs frequently fail to find exactly the right program from the given NL description, though they may output programs that are very similar to the correct one. We can also configure the PTM to return multiple programs, which it samples from some probability distribution over programs implied by the NL description, but this set also commonly does not contain the desired program due to the many possible variations. As natural language is ambiguous and imprecise, in many cases it is just not possible (even for a human) to infer the precise intent from a natural language description alone. This motivates the need for so-called *multi-modal interaction* paradigms [Chen et al. 2020; Manshadi et al. 2013; Raza et al. 2015], where the user can provide a natural language description together with specific input-output examples to precisely express their intent for how the desired program should behave. Such multi-modal interaction is also natural in human interactions as observed in help forum discussions where users convey their intent with a mixture of natural language descriptions and concrete examples.

If we are given examples in addition to the natural language description, the main question that arises is how the examples can be leveraged to improve the results produced by a language model such as a PTM. What we observe is that although the PTM's candidate programs often do not contain precisely the correct program, the programs in this set often contain many relevant *components* (sub-expressions) and use the relevant *operators* – but that these are just not composed correctly to produce the right program. This leads to the idea that the set of candidate programs produced by a PTM can be effectively leveraged by a *component-based program synthesis* technique to construct the desired program from a multi-modal task specification. Component-based synthesis (CBS) [Alur et al. 2013; Alur et al. 2017; Feng et al. 2017b; Gulwani et al. 2011] is a generic approach for synthesizing a program in a domain-specific programming language (DSL) that satisfies a given formal specification of a task (such as input-output examples). In its simplest form, CBS is a systematic enumerative search in the space of possible programs defined by the DSL. It begins with a set of components that are well-formed expressions in the DSL, and iteratively constructs larger programs by combining these components using the operators of the DSL, until a program that satisfies the specification is found. However, in practice, the main challenges for any CBS technique is to handle the state space explosion due to the exponential growth of the set of possible programs, as well as the challenge of ranking among many possible synthesized programs that may satisfy the given specification.

In this work we address these challenges by introducing a generic approach to multi-modal program inference that is based on a marriage of pre-trained natural language models and component-based program synthesis. Our approach combines the benefits of the two techniques by leveraging the output of a PTM to guide all three key phases of the CBS search: the initialization of components, the iterative synthesis of larger programs, and the ranking of final candidate programs. In this way, the combination of the two approaches serves to address the short-comings of each: the CBS synthesis improves precision by constructing a program that satisfies the examples (which may

<table>
<tr><th>#</th><th>Natural Language</th><th>Ground Truth</th><th colspan="2">Pre-trained Model's Candidates</th></tr>
<tr><td>I</td><td>A line with a "!", a capital, or a lower-case before a character</td><td>(!|[A−Z])|[a−z]. ∗ ..∗</td><td>(!|.|([a−z])|([A−Z]))
[A−Z]!|[a−z]!|[a−z]?
(!|[a−z]|[A−Z][a−z]).∗
([a−z]|[A−Z]). ∗ .</td><td>[A−Z]. ∗ (.|[a−z]).∗
([A−Z][a−z]). ∗ (.|.|.).∗
!|[a−z]|[A−Z]
(.|[A−Z]). ∗ .|[a−z]</td></tr>
<tr><td>II</td><td>Lines with at least 7 of the string "!" or a vowel</td><td>(!|[aAeEiIoOuU]){7, }</td><td>((!∗ 7)|[aAeEiIoOuU])+ .∗
(!+ .∗ |[aAeEiIoOuU]{7})
(. ∗ [aAeEiIoOuU]|vowel){7}
((!!)+ |[aAeEiIoOuU])+</td><td>((!+)|[aAeEiIoOuU])+
(.∗!){7, }|(.∗ [aAeEiIoOuU])
((!+)|.......)+
((((!.?)?)∗)|[aAeEiIoOuU])+</td></tr>
<tr><td>III</td><td>At least one digit followed by character : at most once followed by a digit at least zero times</td><td>[0−9]+:?[0−9]∗</td><td>([0−9]∗..:([0−9]∗)?)+
([0−9]+ :)?[0−9]?
([0−9]? : [0−9]?)∗
([0−9]{1, }(?:.[0−9]{0, }))∗</td><td>([0−9]∗ ([:][0−9]∗))∗(0[0−9]+)
[0−9]{3}
([0−9]∗ .. :∗[0−9]∗ 0∗)∗
([0−9]{3})+</td></tr>
</table>

Fig. 1. Examples of three tasks with natural language descriptions of regular expressions, the intended ground truth program, and a sample of the pre-trained model's top-ranked candidates for the task

otherwise not appear in any of the PTM outputs), while the PTM output tames the complexity of the CBS search space by guiding the search at every stage.

A notable characteristic of our approach is its generality that comes from its *domain-agnostic* design: it has not been designed for any particular domain-specific language and can in principle be applicable to different programming domains. As our primary domain of study we focus on the language of regular expressions and illustrate the benefits of our approach in comparison to the state-of-the-art for multi-modal synthesis techniques designed especially for this domain. We also illustrate the generality of our approach by presenting a concrete instantiation and evaluation in the very different domain of CSS selectors, which is a DSL used in web programming. Note that we do not claim that our approach can be directly applied to any arbitrary programming language off-the-shelf, but only that it is not limited to one particular language by showing its applicability and benefits in at least two very different programming domains. In section 7 we also discuss some limitations and expected improvements as we consider scaling to other domains, but a broader evaluation and extension of these techniques to arbitrary languages is left for future work.

1.1 Motivating examples and overview of approach

Consider a scenario where a user needs help writing a regular expression. Figure 1 shows three such tasks. The first column shows the natural language (NL) description the user provides, and the second column shows the ground truth the user desires. (The first two tasks are from the dataset in [Kushman and Barzilay 2013] and the third one is from a Stack Overflow question.)

The first step in our approach is to use a PTM to generate candidate regular expressions from the NL description. Throughout this work, the PTM we use is Open AI's GPT-3 system [Brown et al. 2020], which is a state-of-the-art pre-trained model for code generation from natural language. To get a PTM to produce regular expressions, we need to provide it with the *right* query (called a prompt). We exploit the few-shot learning capabilities of PTM and provide it with the best possible prompt using our novel dynamic prompt generation algorithm (Section 4), which is inspired by literature on information retrieval. PTMs internally generate a probability distribution on possible completions, and then they sample from this distribution to generate individual candidates. We exploit this fact to configure the PTM to generate a diverse sample. The third column in Fig 1 shows some sample candidates returned by the PTM.

In all three cases, the first observation we can make is that the results of the PTM in general look very similar to the ground truth, but none of them are exactly equivalent to it. This can be expected given the significant ambiguity in the NL descriptions which is difficult even for a human to resolve. For instance, for task I it is not clear if the intent is that any of "!", capital or lower-case can occur before the character, or if only the lower-case is permitted to occur before a character and the other two should occur alone on the line (the ground truth shows that the intent is the latter). It is also not clear if "before a character" should mean immediately before a single character or not. Similar ambiguities can be seen in the other two tasks, e.g. whether "at least 7" refers to just "!" or not in task II, whether "at least zero times" refers to everything before it in task III and whether "followed by" means immediately followed by or not.

Such ambiguities are common in natural language, and a good way to resolve them is by allowing the user to provide concrete examples of the desired behaviour, such as examples of strings the intended regex should or should not match. Such multi-modal intent specification is natural even in human interactions as can be seen in help-forum questions where users often provide a description of the task as well as concrete examples to express their intent. Given such examples in addition to the NL, the challenge is how to generate the correct program. The second step in our approach addresses this challenge using a component-based synthesis (CBS) technique, which is designed to utilize the candidates provided by the PTM at each of the three key phases of the search (initialization, expansion and ranking), which we illustrate next.

Initialization. One important question for any CBS algorithm is how to obtain the initial set of components to begin the search. In an extreme brute force search, one may initialize with a set of concrete values for every terminal symbol of the DSL grammar (e.g. all possible character values in the regex domain), but this is practically untenable for any non-trivial DSL. In our case we observe that the candidate programs provided by the PTM all contain very relevant components that can be used to construct the correct program. For example, for case I in Figure 1 we observe frequently occurring relevant components such as "$[A–Z]$", "$[a–z]$", "!" and "$.*$". For case II, apart from important frequent components such as "!" and the number 7, we can observe even the prominent occurrence of the large sub-expression "$[aAeEiIoOuU]$" that represents the notion of a vowel that the PTM has identified. Such an expression using many occurrences of the class union operator would require prohibitively many iterations and examples to construct if starting from purely atomic components. This leads to the question of how we can obtain these most prominent sub-expressions from the PTM outputs, which we address with the novel notion of *maximal components*. Intuitively, these are the largest sub-expressions that occur in the PTM candidates with high frequency. We demonstrate how starting from such maximal components can help to effectively construct the correct program as compared to the traditional component-based approach that starts from all atomic components.

Expansion. After creating the initial set of components, the CBS approach proceeds by iteratively creating larger programs. At each iteration, this is done by applying the DSL operators to the existing programs to create larger programs. The brute force approach would be to exhaustively apply every operator on every combination of components as permitted by the type system of the DSL, but this leads to a combinatorial blowup in practice. A more tractable option is to employ a *beam search* approach where only a bounded number of new programs are kept at every iteration, but the main question is what criteria to use for which programs should be kept or disregarded.

We address this question again using the PTM candidates, by observing the frequency distribution of operators that is found in these programs and biasing the beam search with respect to this distribution. For instance, for case I in Figure 1 we observe that operators such as alternation (|) and iteration (*) are used about once or twice on average across all candidate programs, while other operators such as quantifiers or character class negation are not used at all. This signals a preference for programs that follow a similar operator distribution pattern as opposed to programs that may use five alternations. Technically, we compute an operator frequency distribution vector from the set of PTM candidates, and at each iteration of the beam search we maintain a bounded set of new programs that most closely follow this distribution. In addition, unlike standard beam search methods, we also maintain semantic variety in the beam exploration by ranking within semantic equivalence classes of programs rather than a global ranking in the search space. Such *condensing* of the set of programs within equivalence classes minimizes redundant syntactic variations of the same program in the search exploration.

Ranking. Eventually, the goal of the CBS algorithm is to return a synthesized program to the user that satisfies the examples. But after a certain number of iterations of CBS in practice, there can be a large number of programs that satisfy the given examples. Hence the main question is how to rank among these programs. This decision can be guided by considering similarity of the synthesized programs to the PTM candidates. The operator frequency distribution as used above is a good signal for guiding the search in terms of which operator applications to explore, and is also a good indicator for the final preference of which program to pick from the set of synthesized programs. However, we also found that for final ranking it is helpful to use additional stronger signals such as direct string similarity of programs to the PTM candidates. We found a combination of these signals more finely distinguishes between the final set of synthesized programs in terms of how different operators are being used in the program.

Contributions. The core contributions we make in this work are summarized as follows.

- We present an abstract domain-agnostic formulation of a multi-modal program inference algorithm that can synthesize programs in an arbitrary DSL when given a natural language description and examples of an intended task. This algorithm uses a novel CBS synthesis technique that utilizes the output of a PTM on the given NL description to generate a program that satisfies the given examples, and we demonstrate the relative completeness of our approach with respect to the PTM output.
- We present a concrete instantiation of our technique for the domain of regular expressions, that has been a popular domain in many works that have explored programming by natural language, examples and multi-modal approaches. We present an evaluation of our approach as compared to the state-of-the-art specialized technique for multi-modal regex synthesis, on both existing and new datasets.
- We present secondary instantiation and evaluation of our approach in the very different domain of CSS selectors for extracting elements from web pages. This illustrates the generality of our approach and its applicability in at least two different practical programming domains.
- We show how the *prompt* provided to pre-trained models such as GPT-3 can significantly impact the quality of results, and present novel techniques for formulating this prompt based on ideas from information retrieval [Jones 1972] to show how GPT-3 results can be significantly improved.

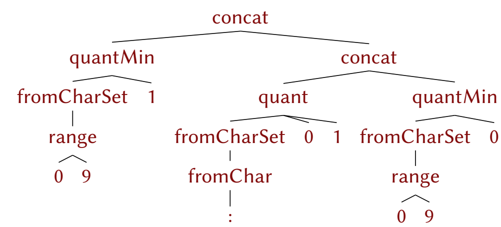
2 DOMAIN SPECIFIC LANGUAGES

Our multi-modal program synthesis algorithm is not designed for a particular programming domain and is parameterized by an arbitrary domain-specific language (DSL) and its execution semantics. In this section, we formally define the general notion of DSLs we use. We also illustrate this by introducing the language of regular expressions, which is the main DSL studied in this paper, and the language of CSS selectors which we also study as a secondary domain.

A DSL is defined as a tuple $\mathcal{L} := (\texttt{Sort}, \texttt{Const}, \texttt{Oper}, \texttt{s}^\circ, \psi_{\texttt{arg}}, \psi_{\texttt{ret}})$ where $\texttt{Sort}$ is a set of *sorts*, $\texttt{Const}$ is a set of *constants*, $\texttt{Oper}$ is a set of *operators*, and $\psi_{\texttt{arg}} : \texttt{Oper} \to \widetilde{\texttt{Sort}}$ and $\psi_{\texttt{ret}} : (\texttt{Oper} \cup \texttt{Const}) \to \texttt{Sort}$ are a pair of *signature* functions. The signature function $\psi_{\texttt{arg}}$ maps a given operator to an ordered sequence of sorts – *of its arguments* – and the signature function $\psi_{\texttt{ret}}$ maps a given operator or a constant to a single sort – *of its return value.* A DSL can be used to build *terms* as follows: every constant is a term, and if $t_1, \ldots, t_n$ are terms of sorts $s_1, \ldots, s_n$ respectively, then $\text{op}(t_1, \ldots, t_n)$ is a term of sort $\psi_{\texttt{ret}}(\text{op})$ if $\psi_{\texttt{arg}}(\text{op}) = \langle s_1, \ldots, s_n \rangle$. We do not distinguish between

$$
\begin{array}{lcl}
\mathsf{i} & := & \{0, 1, 2, 3, \ldots\} \\
\mathsf{c} & := & \{\mathsf{A}, \mathsf{B}, \ldots, \mathsf{a}, \mathsf{b}, \ldots, \#, \$, \%, \ldots, 0, 1, 2, 3, \ldots\} \\
\mathsf{s} & := & \mathsf{fromChar}(\mathsf{c}) \mid \mathsf{range}(\mathsf{c}, \mathsf{c}) \mid \mathsf{union}(\mathsf{s}, \mathsf{s}) \mid \\
 & & \mathsf{negate}(\mathsf{s}) \mid \mathsf{any}() \\
\mathsf{e} & := & \mathsf{quant}(\mathsf{e}, \mathsf{i}, \mathsf{i}) \mid \mathsf{quantMin}(\mathsf{e}, \mathsf{i}) \mid \mathsf{alter}(\mathsf{e}, \mathsf{e}) \mid \\
 & & \mathsf{concat}(\mathsf{e}, \mathsf{e}) \mid \mathsf{fromCharSet}(\mathsf{s})
\end{array}
$$

(a)



(b)

Fig. 2. The DSL $\mathcal{L}_{\text{REG}}$ of regular expressions (left) and the parsed tree of $[0-9]+:?[0-9]*$ (right)

a DSL and the set of terms it generates, and hence, $\mathcal{L}$ will also denote the set of all terms in the language. Each DSL also contains a special sort s° and the goal of the synthesis algorithm is to return a term of this sort. Terms of sort s° will be called *closed* terms or *complete programs*.

As a concrete example, consider Figure 2(a) which presents, $\mathcal{L}_{\text{REG}}$, the DSL of regular expressions. This DSL contains four sorts, $\texttt{Sort} := \{\mathsf{i}, \mathsf{c}, \mathsf{s}, \mathsf{e}\}$, which respectively represent integers, characters, character sets, and expression sorts. Terms of sort e are closed. The set of constants $\texttt{Const}$ includes all non-negative integers (with sort i) and all characters (with sort c). There are also ten operators in the set $\texttt{Oper}$ of operators, whose signatures are shown in Figure 2(a). For instance, quant is an operator with $\psi_{\texttt{arg}}(\mathsf{quant}) = \langle \mathsf{e}, \mathsf{i}, \mathsf{i} \rangle$ and $\psi_{\texttt{ret}}(\mathsf{quant}) = \mathsf{e}$. This DSL encodes a large set of regular expressions that developers commonly write; for example, Figure 2(b) presents the parsed syntax tree of the ground truth expression in Figure 1 (#III).

Definition 2.1 (sub-term and atomic terms). A reflexive and transitive *sub-term* (or *sub-component*) relation, denoted by $\sqsubseteq: \mathcal{L} \times \mathcal{L}$, holds between terms t and t', denoted by $t \sqsubseteq t'$, if t appears as an argument in the syntax tree of t'. We say that a term t is *atomic* if there does not exist any other term t' such that $t' \neq t$ and $t' \sqsubseteq t$.

For example, $\mathsf{fromCharSet}(\mathsf{range}(0, 9))$ is a sub-term of the expression shown in Figure 2(b), and terms 0 and 9 are atomic.

We now formulate a general notion of semantics for terms. We assume that there is an input domain $\Delta_{\texttt{in}}$ and an output domain $\Delta_{\texttt{out}}$ for each sort, and the semantics of a DSL is specified by a function $[[.]] : \mathcal{L} \rightarrow (\Delta_{\texttt{in}} \rightarrow \Delta_{\texttt{out}})$ that given a term in $\mathcal{L}$ returns a function from the input domain to the output domain. Under these assumptions, terms can be viewed as *programs* which transform an input from $\Delta_{\texttt{in}}$ to an output in $\Delta_{\texttt{out}}$. We may also refer to any sub-term of a complete program as a *component* of that program.

For instance, the semantics of closed terms in the DSL of regular expressions is defined over the input domain $\Delta_{\texttt{in}}$ that contains all finite strings and the output domain $\Delta_{\texttt{out}} := \{\bot, \top\}$. A string *str* is said to be *accepted* by the expression r if and only if $[[r]](str) = \top$. We adopt this semantics from the standard regular expression implementations. Informally[1], a term of sort s accepts strings containing a single character c if and only if c satisfies constraints imposed by the root operator of that term. In particular, $\mathsf{fromChar}(c_1)$ only accepts the character c_1, $\mathsf{range}(c_1, c_2)$ accepts any character that lies between c_1 and c_2, $\mathsf{union}(s_1, s_2)$ accepts characters that are accepted by either s_1 or s_2 and $\mathsf{negate}(s)$ accepts characters which are *not* accepted by s. The 0-ary operator $\mathsf{any}()$ accepts all characters.

In a similar fashion, the operator $\mathsf{quant}(e, i, j)$ only accepts strings composed of k sub-strings (for any $i \leq k \leq j$) each of which is accepted by e, and operator $\mathsf{quantMin}(e, i)$ is semantically

[1]The formal definition of this semantics is provided in Appendix A.

$$\begin{array}{lll} \mathsf{s} & := & \text{"a string literal"} \\ \mathsf{i} & := & \text{a number literal} \mid \mathsf{MultipleOffset(i, i)} \\ \mathsf{n} & := & \mathsf{Any()} \mid \mathsf{Union(n, n)} \mid \mathsf{Not(n, n)} \mid \mathsf{TagEquals(n, s)} \mid \mathsf{nthChild(n, i)} \\ & & \mathsf{AttributeEquals(n, s, s)} \mid \mathsf{nthLastChild(n, i)} \mid \mathsf{AttributeContains(n, s, s)} \mid \mathsf{RightSibling(n, n)} \\ & & \mathsf{AttributeStartsWith(n, s, s)} \mid \mathsf{Children(n, n)} \mid \mathsf{AttributeEndsWith(n, s, s)} \mid \mathsf{Descendants(n, n)} \end{array}$$

Fig. 3. The DSL $\mathcal{L}_{\text{css}}$ of CSS expressions.

equivalent to quant(e, i, ∞). Operator alter(e_1, e_2) accepts strings accepted by either e_1 or e_2 and operator concat(e_1, e_2) only accepts strings of the form $str_1; str_2$ if e_1 accepts str_1 and e_2 accepts str_2. Operator fromCharSet does not impose any restriction on the accepted strings and simply *lifts* the terms from sort s to the closed sort e.

For example, the closed term fromCharSet(range(0, 9)) accepts any string composed of a single digit and the term presented in Figure 2(b) accepts all the following four bold-faced strings: **1991:10**, **99999**, **0:1** and **000:**.

As a second target, we now present the domain of Cascading Style Sheets (CSS) selectors. Figure 3 shows, $\mathcal{L}_{\text{css}}$, the DSL for CSS selectors. CSS selectors are expressions for selecting elements from the document object model (DOM) of a webpage. They select nodes based on structural properties that are defined by the HTML source markup of the webpage. For instance, the CSS selector AttributeEquals(TagEquals(Any(), "div"), "class", "row"), call it css_1, selects *all nodes with tag "div" and class "row"*, which is typically written as div.row. Similarly, the CSS selector Children(css_1, AttributeEquals(Any(), "id", "myid")) picks *all nodes that have id "myid" that are immediate child of any node with tag "div" and class "row"*, which is typically written as div.row > #myid, and the CSS selector AttributeEquals(nthChild(TagEquals(Any(), "li"), MultipleOffset(2, 0)), "hidden", "true") represents *all nodes with tag "li" whose attribute "hidden" is set to "true" and that occurs at even positions in the sibling list*, which is typically written as li:nth-child(2n)[hidden = "true"]. The formal semantics is provided in Appendix A. CSS selectors are needed when scraping data from web, or when doing web programming in general. They can be hard to write manually, especially for an occasional user, but they are often easy to describe in natural language.

Having defined the syntax and the semantics of domain specific languages and in particular the DSL of regular expressions and CSS selectors, in the next section, we will formally introduce multi-modal synthesis tasks and describe in detail our generic CBS solution for those tasks.

3 MULTI-MODAL PROGRAM SYNTHESIS ALGORITHM

In this section we present our multi-modal program synthesis algorithm that synthesizes a program to accomplish a task specified in terms of natural language and examples. Our algorithm is *domain-agnostic* and is parameterized by a DSL. Given a DSL $\mathcal{L} := (\texttt{Sort}, \texttt{Const}, \texttt{Oper}, \mathsf{s}^\circ, \psi_{\texttt{arg}}, \psi_{\texttt{ret}})$, we define a multi-modal synthesis task as a tuple (N, E), where N is the natural language description of the task and E is a set of *examples*. We define an example $e \in \Delta_{\texttt{in}} \times \Delta_{\texttt{out}}$ as a pair of values from the input and the output domains. The synthesizer's goal is to find a program $p \in \mathcal{L}$ that is *consistent* with the given examples, defined as follows:

$$p \models E \Leftrightarrow \forall_{<i,o> \in E} [[p]](i) = o$$

Our algorithm, NLX, for multi-modal synthesis from natural language and examples is presented in Figure 4. The main top-level function SYNTHESIZE (Figure 4a) returns a program synthesized from a multi-modal task specification. As the algorithm is domain-agnostic, this function is parameterized by a DSL $\mathcal{L}$ and a PTM $\mathcal{M}$ for this domain. In Section 4 we describe the details of the particular

$\textsc{synthesize}_{\mathcal{L},\mathcal{M}}\ (N, E) :=$
  let $\mathcal{L} = (\mathtt{Sort}, \mathtt{Const}, \mathtt{Oper}, \mathtt{s}^\circ, \psi_{\mathtt{arg}}, \psi_{\mathtt{ret}})$
  $P := \mathtt{exec}(\mathcal{M}, N)$
  $\mathbb{C} := \textsc{initialize}(P)$
  foreach i **in** $\{1, 2, \ldots, \mathtt{SynthDepth}\}$:
    $\mathbb{C} := \textsc{expand}(\mathbb{C}, P, E)$
  $v_1 := \{t \mid t \in \mathbb{C}(\mathtt{s}^\circ) \wedge t \models E\}$
  return $\textsc{rank}(v_1, P).\mathtt{top}(1)$

(a) main function

$\textsc{initialize}\ (P) :=$
  $v_1 := \{t \mid t \sqsubseteq p \wedge p \in P\}$
  $v_2 := \{t \mid t \in v_1 \wedge \mathtt{cnt}(t, P)/|P| \geq \mathtt{PrOcc}\}$
  $v_3 := \emptyset$
  foreach t **in** v_2
    $s := \{t' \mid t' \in v_1 \wedge t \sqsubseteq t'\}$
    $s_r := \{t' \in s \mid t \neq t' \wedge \mathtt{cnt}(t, P) = \mathtt{cnt}(t', P)\}$
    if $|s_r|/|s| \leq \mathtt{PrRed}$ **then**
      $v_3 := v_3 \cup \{t\}$
  $v_3 := v_3 \cup \mathtt{stdComps}(\mathcal{L})$
  foreach s **in** Sort:
    $\mathbb{C}(s) := \{t \mid t \in v_3 \wedge \mathtt{srt}(t) = s\}$
  return $\mathbb{C}$

(b) cache initialization

$\textsc{expand}\ (\mathbb{C}, P, E) :=$
  $\mathbb{C}' := \textsc{prune}(\mathbb{C}, P, E)$
  foreach op **in** Oper :
    $<s_1, \ldots, s_n> := \psi_{\mathtt{arg}}(op)$
    $v_1 := \{op(t_1, \ldots, t_n) \mid \forall_{1 \leq i \leq n}\ t_i \in \mathbb{C}'(s_i)\}$
    $s := \psi_{\mathtt{ret}}(op)$
    $\mathbb{C} := \mathbb{C}[s \mapsto \mathbb{C}(s) \cup v_1]$
  return $\mathbb{C}$

(c) cache expansion

$\textsc{prune}\ (\mathbb{C}, P, E) :=$
  foreach s **in** Sort:
    $v_1 := \mathbb{C}(s)$
    $v_2 := \{t \mid t \in v_1 \wedge \forall_{t' \sqsubseteq t}\ [[t]]_E \neq [[t']]_E\}$
    $v_3 := \{t \mid t \in v_2 \wedge \textsc{HammDist}(P, t) = 0\}$
    $v_4 := \{T \mid T \subseteq v_3 \wedge (\forall_{t,t' \in T}.\ [[t]]_E = [[t']]_E) \wedge (\forall_{t \in v_3} \exists_{T' \in v_4}.\ t \in T')\}$
    $f := \lambda T.\ (|T|/|v_3| \times \mathtt{BeamSize})$
    $v_5 := \bigcup_{T \in v_4} \{t \mid t \in \mathtt{ordEuc}(T, P).\mathtt{top}(f(T))\}$
    $\mathbb{C} := \mathbb{C}[s \mapsto v_5]$
  return $\mathbb{C}$

(d) cache pruning

$\textsc{rank}\ (T, P) :=$
  $f_1 = \lambda t.\ \textsc{EucDist}(P, t)$
  $f_2 = \lambda t.\ (\sum_{p \in P} \mathtt{lev}(t, m))/|P|)$
  return $T.\mathtt{orderBy}(f_1, f_2)$

(e) ranking candidates

$\textsc{EucDist}\ (P, t) :=$
  $<v_1, \ldots, v_n> := \mathtt{opVec}(t)$
  $<v'_1, \ldots, v'_n> := (\sum_{p \in P} \mathtt{opVec}(p))/|P|$
  return $\sqrt{\sum_{1 \leq i \leq n} (v_i - v'_i)^2}$

(f) standard Euclidean distance

$\textsc{HammDist}\ (P, t) :=$
  $<v_1, \ldots, v_n> := \mathtt{opVec}(t)$
  $<v'_1, \ldots, v'_n> := (\sum_{p \in P} \mathtt{opVec}(p))/|P|$
  return $|\{i \mid v_i > \mathtt{OpTH} \wedge v'_i < \mathtt{OpTH} \wedge 1 \leq i \leq n\}|$

(g) customized Hamming distance

SynthDepth :	number of synthesis iterations
PrOcc :	minimum probability of occurrence of a component in programs in P
PrRed :	maximum probability of redundancy of a component in programs in P
BeamSize :	number of programs (of each sort) used to synthesize the next set
OpTH :	threshold that determines low-frequency operator occurrence

(h) constants

$\mathtt{opVec}(t)$:	returns a vector composed of the number of occurrences of each DSL operator in t
$\mathtt{orderBy}(f_1, f_2)$:	returns a lexicographically ordered list based on given score functions f_1 and f_2
$\mathtt{lev}(t_1, t_2)$:	returns the Levenshtein distance between string representations of t_1 and t_2
$\mathtt{top}(n)$:	returns the first n elements from an ordered list
$\mathtt{srt}(t)$:	returns the sort of the root operator in t
$\mathtt{cnt}(t, P)$:	returns number of programs p in P such that $t \sqsubseteq p$
$\mathtt{exec}(\mathcal{M}, N)$:	Runs $\mathcal{M}$ on N and returns the resulting candidate programs
$\mathtt{stdComps}(\mathcal{L})$:	the standard components to include for DSL $\mathcal{L}$
$\mathtt{ordEuc}(T, P)$:	orders terms in T based on their Euclidean distance from average program in P

(i) auxiliary functions

Fig. 4. NLX algorithm for multi-modal program synthesis, parameterized on a DSL $\mathcal{L}$ and PTM $\mathcal{M}$

PTM model we use and how it is configured with few-shot learning for a particular domain, and in this section we assume that such a model $\mathcal{M}$ is given.

We first describe the high-level structure of our NLX algorithm, before describing the key phases in more detail. The algorithm proceeds by first obtaining the top-ranked programs from the PTM for the given natural language query. It then implements a component-based synthesis (CBS) that utilizes the PTM output to guide the CBS search at each of the three key phases: the initialization of components, the iterative expansion to larger programs, and the final ranking of programs.

The SYNTHESIZE function implements this high-level structure of the algorithm. It initially executes the PTM on the given description N and stores the resulting programs in P (line 3), which is used in each of the subsequent phases. The algorithm uses a *cache* object $\mathbb{C}$ to maintain the set of synthesized programs according to their sort in the DSL. A cache, denoted by $\mathbb{C} : \texttt{Sort} \rightarrow \overline{\mathcal{L}}$, is defined as a map from sorts to sets of terms in $\mathcal{L}$. The cache is initialized by extracting components from the PTM candidates P. This initialization phase is defined by the function INITIALIZE (line 4), which we describe in more detail in §3.1. Next, we enter the expansion phase in the main loop of the algorithm at line 5. At each iteration, the cache is updated with new programs synthesized by applying operators of $\mathcal{L}$ on existing components in the cache. As exploring all possible operator and component combinations is intractable in practice, we employ a beam search where such combination choices are guided by the PTM candidates P. This is defined by the EXPAND function that is described in §3.2. This process is repeated up to a tunable constant `SynthDepth`. Finally, the algorithm identifies closed programs in the cache which are consistent with the given examples (line 7) and then performs a ranking to choose the best program to return out of many possible ones. This ranking is based on similarity to the PTM candidates P as defined by the RANK function which we describe in §3.3.

We next describe each of the three key phases of the algorithm that are formally defined by the functions in Figure 4a, and end this section with a discussion of the relative completeness of our algorithm.

3.1 Initialization of components

The first step in the algorithm is to obtain the set of initial components from which to begin the search. As we have discussed, the set of top candidate programs P provided by the PTM contains very relevant components for constructing the correct program, but initializing with a very large set of components can lead to the search complexity becoming intractable. Hence we approach this question with respect to two aspects: how likely a component is to *occur* in the desired program and how likely is it that a component is *redundant* in the initial component set (in the sense that it can already be included as part of another larger component). Both of these questions are addressed using a probabilistic formulation with respect to the distribution of components in the PTM candidates.

The INITIALIZE function in Figure 4b takes as input the set of PTM candidate programs and returns an initialized cache. This function initially extracts the set of all sub-terms of all programs in P and stores it in a variable v_1 (line 2).

Component occurrence. At line 3 we compute the probability of occurrence of each component and keep those above a tunable minimum probability threshold defined by a constant `PrOcc`. The occurrence probability for a term t is computed as $\texttt{cnt}(t, P)/|P|$, which is the proportion of programs in P that contain the component t. For example, in Figure 1 (#II), if we consider the set P to be the 8 candidate PTM programs, and the term t = quantMin(fromCharSet(any()), 0) (printed as .*) appears in five of the programs in P, then we have the occurrence probability of t given by $\texttt{cnt}(t, P)/|P| = 5/8 = 0.625$.

The occurrence probability check ensures that terms that appear more often in the PTM's output have a higher chance of being included in the initial cache, as often times there is noise in the PTM output that includes irrelevant components that occur very infrequently. For example, in Figure 1 (#II), the term printed as *vowel*, which is clearly due to PTM's confusion about the task, only appears once in the candidate programs. Such terms can be easily eliminated from the initial cache by setting the occurrence probability threshold `PrOcc` appropriately. In practice, we found that a value of `PrOcc` = 0.1 worked well in all our evaluations (with usually at least 20 PTM candidates in total).

Component redundancy. The second aspect we consider in the initialization of components is that of *redundant components*. This is important because while many components may occur frequently, many of these may not be useful to include in the initial cache as they may already occur as sub-components of other components. With respect to our PTM candidates, if we find that a component t always appears as a sub-component of another component t' in all of the PTM candidate programs, then that is a strong signal that t is a redundant component as it can already be included as part of t' in the cache. For example, in Figure 1 (#II), the term $t_1 = \mathsf{fromChar(a)}$ *always* appears as a sub-component of the same term t_2 that unions all vowels (printed as $[aAeEiIoOuU]$). Similarly all terms representing subsets of vowel characters always occur only as sub-components of t_2 and can be considered redundant to include by themselves. The term t_2 however, occurs as part of many different components and is important to include as a component by itself. In the same example, the term $t_3 := \mathsf{fromCharSet(fromChar(!))}$ (printed as !) appears in multiple different super-terms, e.g. in $t_4 := \mathsf{quantMin}(t_3, 0)$ (printed as !∗), $t_5 := \mathsf{quantMin}(t_3, 1)$ (printed as !+) and $t_6 := \mathsf{concat}(t_3, t_3)$ (printed as !!). We note that the inclusion of both t_2 and t_3 in the initial cache is important to construct the ground truth program in this case, since none of t_4, t_5 or t_6 directly appear in the ground truth, i.e. $(!|[aAeEiIoOuU])\{7,\}$.

Formally, at lines 6-9 in the algorithm, we compute the probability of redundancy of a component t as the proportion of super-components of t that occur as many times as t in the PTM candidates (note that by definition no super-component can occur more times than any of its sub-components). If the redundancy probability is below a certain maximum threshold given by `PrRed`, then the component is included in the initialization.

Though in general the algorithm permits the redundancy threshold `PrRed` as a tunable constant, we note that the extreme case of `PrRed` = 0 identifies a special case of *maximal components* that work well in practice. These are components that occur more frequently than any of their super-components, and hence represent the PTM's identification of a component that it uses in different ways across different candidate programs, such as the vowel component in Figure 1 (#II). This suggests the PTM's high confidence that the component is useful but lower confidence on *how* it should be used in the final program, and hence makes it a good candidate to include in the CBS search which explores many more combinations for synthesis.

We note that this notion of maximality is not just with respect to size but both size and frequency. Hence components t and t' may both be maximal even if $t \sqsubseteq t'$, if t occurs more frequently than t'. Both would be useful to consider as the PTM candidates indicate that t may be used in other ways outside of t'.

Standard components. Finally, the algorithm permits a fixed set of standard components for the DSL that should always be included (line 10). These may be any terminal or commonly-used special values for the different sorts in the language. For instance, for the regex domain we include the standard components that are the integer values 0,1 and the specially named character classes `\d`,`\s`,`\w`, representing digits, space and word characters. For the CSS domain, we include the any-element selector $\mathsf{Any()}$, the integer value 1 and the empty string attribute value.

3.2 Expansion

In this section we describe the expansion phase of our algorithm, where larger programs are iteratively constructed using the initial components and already synthesized programs. The brute force approach would be to exhaustively apply every operator on every combination of components as permitted by the rules of the DSL, but this leads to intractable complexity in practice. Hence the technique we use is to employ a form of *beam search* where only a bounded number of new programs are kept at every iteration. This beam search is defined by the technique of *pruning* the cache which determines which synthesized programs to keep and which to discard. We design this technique based primarily on the distribution of operators that is found in the PTM candidates and biasing the beam search with respect to this distribution. We first describe the outline of the expansion phase and then describe the pruning technique in more detail in section 3.2.1

The function EXPAND is defined in Figure 4c, which given a cache $\mathbb{C}$ as input returns a new cache expanded with a set of new terms based on existing terms in $\mathbb{C}$. The procedure initially obtains the subset of terms in $\mathbb{C}$ to be considered for expansion using a call to the PRUNE function (line 2). The procedure then iterates over all operators $op \in$ `Oper` and constructs new terms in $\mathcal{L}$ by applying op on existing terms in $\mathbb{C}'$ according the signature of op. Newly constructed terms are then stored in a variable v_1 (line 5). For example, assuming that the following two terms, t_6 and t_7, are in the pruned cache $\mathbb{C}'$, the set of newly constructed terms, v_1, will include terms like $t_8 := \mathsf{alter}(t_6, t_7)$ and $t_9 := \mathsf{concat}(t_6, t_7)$:

$$t_6 := \mathsf{quantMin}(\mathsf{fromCharSet}(\mathsf{fromChar}(\mathsf{a})), 0) \qquad \text{(printed as } a*\text{)}$$
$$t_7 := \mathsf{quantMin}(\mathsf{fromCharSet}(\mathsf{fromChar}(\mathsf{b})), 0) \qquad \text{(printed as } b*\text{)}$$

Finally, the procedure EXPAND updates the original cache $\mathbb{C}$ by adding terms in v_1 to $\mathbb{C}(s)$, where s is the return sort of current operator op (line 7). Once the loop is iterated over all operators, the procedure returns the updated cache $\mathbb{C}$ as its final output (line 8).

3.2.1 Pruning the Cache. Our technique of pruning the search space during the beam search is based primarily on the distribution of operators that is found in the PTM candidates and biasing the beam search with respect to this distribution in a way that maintains semantic variety of the synthesized programs (i.e. minimizes redundant semantically equivalent expressions in the search space). The pruning function PRUNE is defined in Figure 4d, which is used in the beginning of each expansion iteration to bound the number of terms considered for expansion.

Semantically equivalent sub-terms. To avoid semantically redundant states, the first pruning strategy is to remove any term that is semantically equivalent to any of its sub-terms (line 4). For instance, assume that the given set of examples is $E_1 := \{\mathbf{aa}, \mathbf{ccc}\}$ and the term t_8 defined earlier (i.e. $\mathsf{alter}(t_6, t_7)$) is in v_1. Observe that t_8 is not semantically distinguishable from its sub-term t_6 with respect to E_1 (since they both accept **aa** and reject **ccc**). Consequently, any possible use-case of t_8 in the future iterations can also be handled by t_6, and hence, t_8 can be eliminated from v_1.This observation is formalized by defining the *interpretation* of a term t with respect to a set of examples E, denoted by $[\![t]\!]_E$, as a set of input and output pairs, where the input belongs to an example in E and output is generated by running t on that input, i.e. $[\![t]\!]_E := \{<i, [\![t]\!](i)> \mid <i, \_> \in E\}$. In the example discussed above, we have $[\![t_6]\!]_{E_1} = [\![t_8]\!]_{E_1} = \{<\mathbf{aa}, \top>, <\mathbf{ccc}, \bot>\}$. The procedure eliminates all terms in v_1 which share their interpretation (with respect to the given examples) with some of their sub-terms (line 4). The remaining terms are stored in a fresh variable v_2.

Low frequency operators. Our primary signal for pruning is to bias towards the structure of the PTM candidates. The first constraint we consider in this bias is to avoid DSL operators that may occur with a very low frequency (or not at all) across all of the PTM candidates. For example, in Figure 1 (#III), the alternation operator (alter) does not appear in any of the candidate programs generated by the PTM. This signals that the target program does not have many alternation operators (it has in fact none).

We distinguish such low-frequency operators using a tunable constant `OpTH` that defines the threshold for low-frequency operators: operators that on average have fewer occurrences than this threshold are allowed at most `OpTH` occurrences. We implement this using the function `opVec` that given a term t returns an integer vector composed of the number of occurrences of each DSL operator in t. For example, for the term t_6 defined above, `opVec`(t_6) is a vector that has value 1 in the three entries assigned to quantMin, fromCharSet and fromChar and has 0 everywhere else. Using this function, we eliminate terms whose operator vector is different from programs in P. In particular, the procedure PRUNE eliminates terms from v_2 whose *Hamming distance* from programs in P is bigger than 0 (line 5).

The Hamming distance between a term t and the set of programs P is calculated using the function HAMMDIST, defined in Figure 4g. The inputs to the function are a set of programs P to compute the distance from, and a term t. This function first determines the operator vector of t (line 2) and the *average* operator vector of all programs in P (line 3). The final result is defined as the number of entries in the operator vector of t whose value is greater than `OpTH`, and the value of the corresponding entry in the average vector of P is less than `OpTH`.

For example, in Figure 1 (#III), the value assigned to the alter entry in the average operator vector of programs in P is 0, and hence, if `OpTH` is set to 1, their Hamming distance from any term that has more than 1 occurrences of alter is at least 1; such terms will not be included in v_3.

Semantic condensation. The final step of pruning is to implement the beam-based cutoff of the state space based on the final ranking of programs with respect to the operator distribution. Unlike standard beam search methods, we do not perform a global ranking on the search space when considering the beam. Instead, we maintain semantic variety in the beam exploration by ranking *within* semantic equivalence classes of programs. Such *condensing* of the set of programs within equivalence classes minimizes redundant syntactic variations of the same program in the search exploration which can come from a global ranking. At line 6, the prune function classifies terms in v_3 into *semantic classes*. A semantic class is defined as a set of terms which have equal interpretations with respect to E. All terms in v_3 must belong to exactly one semantic class. The set of all semantic classes is stored in variable v_4 (line 6).

Next, using a call to function `ordEuc` (defined in Figure 4i), the procedure orders terms in each semantic class according to their syntactic similarity to the programs in P. The highest ranked programs in each semantic class are then identified and their union is stored in a variable v_5 (line 8). The number of terms selected from each class is determined by the size of that class and a tunable constant `BeamSize` (line 7). For example, assuming `BeamSize` is set to 2000 and there are 5000 terms in v_3, the top 400 terms from a semantic class of size 1000 will be selected to be in the pruned cache.

Finally, once the iteration over all sorts is finished, the procedure returns the fully pruned cache as the final result (line 10).

3.3 Ranking the Synthesized Programs

The eventual goal of the CBS algorithm is to return a synthesized program to the user that satisfies the examples. But after a certain number of iterations of CBS in practice, there can be a large number of programs that satisfy the given examples. As in the other phases, our technique for ranking is also guided by considering similarity of the synthesized programs to the PTM candidates. The operator frequency distribution as used above is a good signal for guiding the search in terms of which operator applications to explore, and is also a good indicator for the final preference of which program to pick from the set of synthesized programs. However, we also found that for final ranking it is helpful to use the additional stronger signal of direct string similarity of programs to the PTM candidates.

The function RANK is defined in Figure 4e. This function is called in the main function SYNTHESIZE (line 8). Given a set T of terms and a set P of programs generated by the PTM, this function returns an ordered list of terms in T according to their syntactic similarity to the programs in P. In particular, terms in T are lexicographically ordered based on two different measures of distance from programs in P. The first measure is the standard Euclidean distance between the operator vector of terms in T and the average operator vector of programs in P (line 2). This is to ensure that the final synthesized program is structurally as close as possible to the PTM's candidate programs.

In order to distinguish terms in T with the same Euclidean distance to P, the procedure next applies the Levenshtein distance [Black 1999] as a more fine-grained measure of distance between (the string representation of) terms. The Levenshtein between two strings is defined as the minimum number of single character modifications required to transform one string into another.

3.4 Completeness

We have described how our NLX algorithm implements a component-based synthesis that is guided by the output of the PTM at every stage. While we empirically evaluate the effectiveness of these techniques in practice, in this section we consider what theoretical guarantees can be provided on completeness: can the algorithm eventually find a program if one exists? As is common for any program synthesis approach based on natural language input, completeness depends on the ability of the underlying language model, which in our case is the PTM, to find relevant programs that match the intended task. Hence, we formulate a *relative completeness* result with respect to the PTM output.

Let P be the candidate programs provided by the PTM. We define the *closure* of P with respect to the DSL $\mathcal{L}$, denoted P_c, as the set of all programs that can be constructed from all atomic components in P. Formally, $t \in P_c$ iff either t is atomic (Definition 2.1) and $t \sqsubseteq p$ for some $p \in P$, or otherwise $t = op(t_1, \ldots, t_n)$ where $op \in \texttt{Oper}$ and $t_i \in P_c$. Hence, P_c includes P and all other programs that can possibly be constructed using components from P. We show that if the correct intended program exists in P_c, then our NLX algorithm can find a semantically equivalent program when given sufficient examples (under the assumption of a condition of compositionality (T) holds for our DSL: for any terms p, t, t', whenever $[[t]]_E = [[t']]_E$, then $[[p[t]]]_E = [[p[t']]]_E$).

Corollary 3.1 (Relative completeness). *Let P be the results of the PTM for a given natural language description N, and assume that the intended ground-truth program p exists in the closure P_c of P. Assume we set algorithm configuration parameter settings* `PrOcc` $= 0$, `PrRed` $= 1$ *and have unbounded* `SynthDepth`, `BeamSize` *and* `OpTH`. *If compositionality condition (T) holds, then there exists a sufficient set of examples E for which the NLX algorithm will return a program p' that is semantically equivalent p.*

This relative completeness result follows from the fact that our NLX algorithm reduces to an exhaustive enumerative search under the extreme parameter settings above. Under the occurrence and redundancy probability settings `PrOcc` $= 0$ and `PrRed` $= 1$ the cache is initialized with all possible components in P, which includes all atomic components. With unbounded iteration depth, unrestricted beam size and no constraints of operator frequency, the expansion phase explores all possible operator applications in the closure P_c. Assuming p requires k synthesis iterations to construct, let $p_1, ..., p_n$ be all programs synthesized in up to k iterations. Let E be a set of examples that distinguishes p from each of $p_1, ..., p_n$ where $p_i \neq p$ (such an example set must exist or else p will be semantically equivalent to some p_i). Then given the example set E, the algorithm will return the desired program p after k iterations. One notable issue is presented by our optimization on line 4 in Figure 4d, where we eliminate any term that is equivalent to any of its sub-terms. In case p contains a term t that is eliminated in favor of some semantically equivalent t', then we will synthesize p' that uses t' instead of t, which will be semantically equivalent to p (by condition (T)).

4 OPTIMIZED USE OF THE PTM

Section 3 describes a generic component-based synthesis technique that uses the top candidate results of a PTM to seed and guide an enumerative search. The effectiveness of this process, however, depends on the quality of the initial results received from the PTM. Getting the right results from the PTM is heavily dependent on *asking the questions in the right way.* In particular, using the PTM effectively involves three distinct steps. First, the task at hand is encoded into a *prompt* that acts as the context for the PTM. Then, the prompt is provided as input to the PTM which then produces a *completion.* Finally, the candidate program from the output completion is *extracted.* In this section, we explain each of these steps in details and conduct a formative study to design a technique to generate high quality prompts to obtain useful initial programs.

Following [Brown et al. 2020], we use the PTM as a *few-shot learner,* i.e. the model is provided a few question-answer pairs that act as examples of the task at hand. Note that we use the term question-answer pair (instead of example) to avoid confusion with the examples required for the synthesis tasks given to NLX algorithm. For instance, Figure 5a shows an example prompt outlining the prompt structure that is used as context for the PTM. The prompt consists of three parts: (i) a high level description of the task domain (lines 1-3), (ii) a sequence of sample question-answer pairs (lines 5-15), and (iii) the question of interest (line 17).

```
(01) Here are some examples of regular expressions
(02) and their descriptions. Use them to generate a
(03) regular expression that matches the description.
(04)
(05) NL: lines which begin with an upper case vowel
(06) Regex: [AEIOU].*
(07)
(08) NL: match lines which contain only consonants
(09) Regex: [^AEIOUaeiou]*
(10)
(11) NL: lines ending with a digit followed by period
(12) Regex: .*[0-9][.]
(13)
(14) NL: dates in ISO 8601 format
(15) Regex: [0-9]{4}-[0-9]{2}-[0-9]{2}
(16)
(17) NL:  lines starting with three upper case vowels
          followed by four digits
(18) Regex:
```

(a) prompt

```
(18) Regex: [AEIOU]{3}[0-9]{4}.*
(19)
(20) NL: lines starting with a digit followed by three
         upper case letters followed by two digits
(21) Regex: [0-9][A-Z][A-Z][A-Z][0-9]{2}
```

(b) completion

Fig. 5. A prompt and the corresponding completion. Line numbers in parenthesis are for illustration only

Structuring the prompt in this way has multiple advantages. First, the question-answer pairs often contain components that increase the probability of the PTM returning results using those components, e.g. `vowel` is used in a question-answer pair (line 5) which is also part of the final question. Additionally, as a small advantage, the structured prompt biases the PTM to produce a response in the same format making the task of extracting the resulting program as simple as picking the right *stop sequence* (here, `NL:`).

For example, Figure 5b presents one of the completions[2] that GPT-3 produces given the prompt in Figure 5a. The completion consists of a candidate program for the task (line 18) and a few additional lines, following the same pattern from the prompt (lines 19-21). It is easy to see how the candidate program can be extracted from the completion using the stop sequences. Although in this example the PTM was able to successfully generate the intended program, that is not always the case. For example, if we remove the first two question-answer pairs from the prompt (lines 5-10), the completion produced by GPT-3 does not solve the task correctly and returns programs like `[A-Z]{3}[0-9]{4}.*`, which does not even include the correct components of the intended program.

[2]Note that GPT-3 is non-deterministic by nature. Completions shown represent typical results for the prompts.

Algorithm 1: Question-Answer pair ranking

Require: Question-answer corpus $\mathrm{QA} = (q_0, a_0), \ldots, (q_n, a_n)$
Require: Natural language description at hand q^*
Require: Relevance metric $\mathcal{R}$
Require: Result size threshold $k \in \mathbb{R}$ and similarity threshold $t \in \mathbb{R}$
1: $\mathsf{RelevantQA} \leftarrow$ empty sequence
2: **while** $\mathsf{QA} \neq \emptyset \wedge |\mathsf{RelevantQA}| < k$ **do**
3: $\quad (q_m, a_m) \leftarrow \mathsf{argmax}_{(q_i, a_i) \in \mathsf{QA}} \mathcal{R}(q^*, q_i)$
4: $\quad \mathsf{QA} \leftarrow \mathsf{QA} \setminus \{(q_m, a_m)\}$
5: $\quad$ **if** $\nexists (q, a) \in \mathsf{RelevantQA}.\ \mathsf{LevenshteinDistance}(a, a_m) < t$ **then**
6: $\quad\quad \mathsf{RelevantQA} \leftarrow \mathsf{RelevantQA}; (q_m, a_m)$
7: $\quad$ **end if**
8: **end while**
9: **return** $\mathsf{RelevantQA}$

The above example highlights the main challenges when using PTMs as program synthesizers. In the remaining of this section, we will present our prompt generation approach to maximize the likelihood of getting the correct programs with right components in the PTM's completion.

4.1 Formative Study on Selecting Question-Answer Pairs

Most PTMs restrict the size of the input prompt they accept. For example, the GPT-3 prompt is restricted to 2048 tokens (i.e. small units that are meaningful and occur more generally). Given this limited prompt size, choosing the right set of question-answer pairs to act as examples for k-shot learning becomes very important for the quality of results. We introduce a technique for choosing relevant question-answer pairs, and study multiple variations to determine the optimal parameters for prompt generation.

Algorithm 1 depicts our question-answer pair selection technique. The primary inputs are (1) a corpus of question answer pairs, $\mathrm{QA} = (q_0, a_0), (q_1, a_1), \ldots (q_n, a_n)$, where each question q_i is a natural language description and the answer a_i is the corresponding program, and (2) a question q^* that represents the task in hand. The procedure returns a sequence $\mathsf{RelevantQA} = (q_{i_0}, a_{i_0}), \ldots, (q_{i_k}, a_{i_k})$ of k question-answer pairs to be used in the prompt. The algorithm is parameterized by a relevance metric $\mathcal{R}$ on questions. A greater $\mathcal{R}(q, q')$ score indicates that question q (and its answer) is more relevant to (answering) q'. At a high level, Algorithm 1 orders the available question-answer pairs in QA based on their relevance to q^* and identifies the highest ranked question-answer pair (line 3). The chosen pair (q_m, a_m) is added to the result sequence if a_m is not "too close" to an already selected answer in RelevantQA (line 5). Here we define closeness of answers as the Levenshtein distance between them, which is a fine-grained measure of distance between strings at the level of characters [Black 1999]. If the distance is less than a threshold t then the answers are considered too close and the question-answer pair is discarded. This is to ensure that the PTM is not biased toward a particular group of tasks and does not produce sub-optimal results.

Below, we introduce two classical metrics of relevance from the information retrieval literature and study the impact of each on the quality of the PTM's completion.

4.1.1 Relevance Metrics. Suppose we are interested in computing $\mathcal{R}(q, q')$, the relevance of question q to question q'. We use $|q|$ and $|q'|$ to denote the number of tokens in q and q', respectively, and define $\mathsf{CT}(q, q')$ to be the set of tokens common to q and q'. We now introduce two different definitions for $\mathcal{R}(q, q')$:

- **Token match:** This metric, $\mathcal{R}_{\mathsf{TM}}$, measures the fraction of the number of tokens in q that are also present in q', i.e. $\mathcal{R}_{\mathsf{TM}(q,q')} = \frac{|\mathsf{CT}(q,q')|}{|q|}$.

- **TF-IDF:** The measure $\mathcal{R}_{\mathsf{TM}}$ treats all tokens identically because we just count the tokens. However, rare tokens are better indicators of relevance. We follow the standard term-frequency inverse document frequency (TF-IDF) technique [Jones 1972] to increase weight of rare tokens. In particular, we define the TF-IDF score of each token and weight them based on this score. The score $\mathsf{TFIDF}(T)$ of a token T is the product of (a) the *term frequency* of T, i.e., the number of times T occurs in q, and (b) the log of the *inverse document frequency* of T, i.e., the negative log of the fraction of questions from the corpus that T appears in. Thus, we have $\mathcal{R}_{\mathsf{TFIDF}}(q, q') = \frac{\sum_{T \in CT(q,q')} \mathsf{TFIDF}(T)}{\sum_{T \in q} \mathsf{TFIDF}(T)}$.

4.1.2 Experimental Results. In this part, we study the impact of the similarity check in Algorithm 1 (line 5) on the recall of the PTM for different relevance metrics. We use GPT-3 as our PTM with the *temperature parameter* set to 0.6. In GPT-3 terminology, temperature 0.0 represents an entirely deterministic value, whereas 1.0 represents output that is fully stochastic. In the domain of regular expression, we aimed to allow the PTM sufficient randomness to generate a varied candidate set, but not an entirely random one such that the similar components between candidates indicated some measure of confidence. After some brief initial trials, 0.6 was selected for temperature and 10 as the threshold on the number of question-answer pairs in the prompt. Our corpus contained 4855 question-answer pairs from the [Locascio et al. 2016] dataset (see Section 5) and our test tasks consisted of 115 questions from the same dataset. For each variant of Algorithm 1, we generated a prompt based on the relevant pairs returned by the variant and measured the recall in top 20 completions, i.e., in what fraction of the cases is the correct answer is in the top 20 completions produced by the PTM. For the baseline, we propose two techniques: First, a straw-man procedure that randomly selects question-answer pairs from the corpus for each question, Second a Hand-Picked context which remains unchanged throughout the entire experiment. First, we fix the threshold k on the size of the result to be 10, and test variants using token match and TF-IDF metrics and with and without the Levenshtein distance based similarity check. The results are summarized in Table 1. The results highlight two key points: (1) Intelligent relevance-based selection of question-answer pairs for the in-context k-shot learning makes a significant difference to the recall of the PTM. (2) Using the Levenshtein distance based similarity check increases the diversity in the question-answer pairs used in the prompt, and thereby increases the recall of the PTM.

Relevance Metric	Similarity Check	Top-20 Recall
Hand-Picked	No	0.32
Random	No	0.33
$\mathcal{R}_{\mathsf{TFIDF}}$	Yes	0.46
$\mathcal{R}_{\mathsf{TFIDF}}$	No	0.44
$\mathcal{R}_{\mathsf{TM}}$	Yes	0.42
$\mathcal{R}_{\mathsf{TM}}$	No	0.40

Table 1. Recall within top 20 completions for variants of Algorithm 1.

Based on the above insights, we chose the best variation with TF-IDF relevance metrics and the similarity check for conducting experiments in Section 5.

5 EVALUATION

This section presents an empirical evaluation of our synthesis approach across two programming domains. First, in §5.1 we present NLX-REG – an implementation of our algorithm for the domain of regular expressions, and compare it to the state-of-the-art regular expression synthesizer. Next, in §5.2, we introduce NLX-CSS for the domain of CSS selectors and evaluate it on a corpus of standard synthesis tasks in this domain.

5.1 Domain of Regular Expressions

We now present our synthesizer for regular expressions from natural language and examples. This synthesizer, named NLX-REG, implements an instance of the domain-agnostic algorithm presented

Question: I want to validate decimal values with up to 18 digits before the decimal and 1 digit after; with the decimal point and the digit after it being optional. For example all the following three numbers should be accepted: `100.1`, `123456789.2`, and `123456789`. But these three numbers should not: `1.01`, `1234567891234567891`, and `1234567891234567891.0`.
I am currently using `([0−9]{1, 18})+ (\.[0−9]{1})?` as my regular expression, however it seems to be accepting things that are more than 18 digits before the decimal point. Does anyone know what I did wrong here?
Answer:
Drop the '+': `([0−9]{1, 18})(\.[0−9]{1})?`

(a)

Natural Language:
A regex which accepts numbers up to 18 digits and an optional decimal point followed by a digit at the end
Examples:
$(\texttt{100.1}, \top)$, $(\texttt{123456789.2}, \top)$, $(\texttt{123456789}, \top)$, $(\texttt{1.01}, \bot)$, $(\texttt{1234567891234567891}, \bot)$, $(\texttt{1234567891234567891.0}, \bot)$

(b)

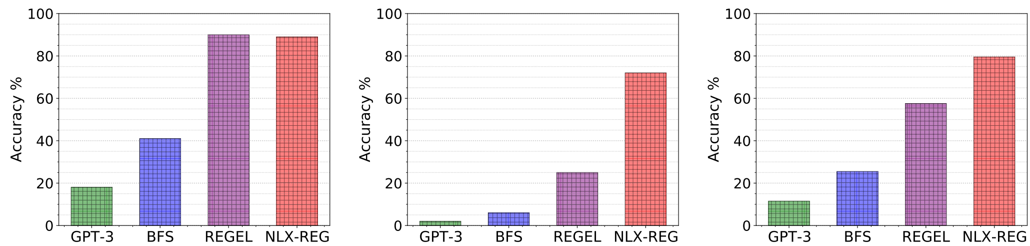
Fig. 6. A StackOverflow post (left) and the extracted task (right)

in section 3, and is written in C# language with about $5k$ lines of code. We apply NLX-REG on a set of synthesis benchmarks adopted from various sources and assess its performance by comparing it to three baselines, including the state-of-the-art synthesizer for regular expressions. We begin by describing these baseline systems:

(1) **REGEL** is a tool developed by Chen et al. [2020], and is the state-of-the-art synthesizer for regular expressions from natural language and examples. REGEL works by first generating a sketch (i.e. a basic scaffolding) of the target expression from the given English description of the task, and then completes the sketch using an enumerative search guided by the given examples. REGEL is designed specifically for the domain of regular expressions and cannot be applied to other DSLs. Chen et al. [2020] report a significantly higher accuracy rate for REGEL (80% vs 43%) compared to DeepRegex [Locascio et al. 2016], the prior state-of-the-art tool for generating regular expressions directly from natural language. The DSL of regular expressions used in DeepRegex and REGEL is similar to ours except that, for implementation reasons, our DSL does not include the `And` (intersection) and `Not` (complement) operators, which are not supported by many standard libraries.
(2) **GPT-3** represents the next baseline system in our setup, which is simply a PTM that is used as an end-to-end synthesis tool. In other words, the top candidate generated for each task by the PTM is compared to the ground-truth without any further processing.
(3) **BFS** represents the *brute force search* approach of component-based synthesis: it implements an exhaustive bottom-up search that starts with the initial set of *all* atomic components found in *any* of the PTM candidates, and applies all DSL operators at every iteration of the search. This baseline represents a simple way of combining the PTM output with component based synthesis, as opposed to the techniques for initialization, expansion and ranking that we introduced in section 3 and are implemented in our NLX-REG system.

We applied NLX-REG and all the above baseline systems on two sets of synthesis tasks and we will report the accuracy of each system on both sets and also per each set separately. Following is a summary of how we curated each of these benchmark sets:

DeepRegex. Chen et al. [2020] originally evaluated REGEL using a set of 200 synthesis tasks sampled from DeepRegex benchmark set [Locascio et al. 2016]. DeepRegex consists of 10000 pairs of natural language descriptions and regular expressions, automatically generated using a small manually-crafted grammar. The artificially created natural language descriptions are then paraphrased through crowd-sourcing. Since tasks in DeepRegex set only include English descriptions, Chen et al. also asked users to provide examples (4 positive and 5 negative on average) for each task. We eliminated 75 tasks where the ground-truth required either `And` or `Not` operators, which



(a) DeepRegex (b) StackOverflow (c) Average

Fig. 7. Evaluation of NLX-REG and baseline systems

are not supported by NLX-REG as mentioned above. We used the remaining tasks for evaluating all systems.

StackOverflow. To complement the DeepRegex benchmarks with more challenging real-world scenarios, we also curated a set of synthesis tasks based on questions submitted to StackOverflow online forum. We initially retrieved posts tagged with keywords "regex" and "regular expression" and identified cases with exactly one regular expression in the question, r_q, and one regular expression in the accepted answer, r_a. Using these expressions, we were able to *automatically* generate positive and negative examples for each task. In particular, we executed both expressions on the body of the question and collected all strings accepted by r_a (i.e. the ground-truth) as positive examples. Similarly, all strings accepted by r_q and rejected by r_a were collected as negative examples.

As a concrete example, consider Figure 6a which presents a StackOverflow post[3] identified using the above procedure. The question in this post explains a task using a combination of natural language and positive and negative examples. It also provides a faulty expression (i.e. r_q) which does not correctly perform that task. The accepted answer includes the correct expression for the task (i.e. r_a). Note that all positive examples provided by the user are accepted by r_a and all negative examples are accepted by r_q and rejected by r_a. The task extracted from this post is shown in Figure 6b. While we were able to automatically extract examples for each task, we relied on users across our institution to read the posts and paraphrase them concisely to eliminate redundancies common in online posts. Using this methodology, we collected a set of 25 tasks with an average of 4.3 positive and 1.4 negative examples per task.

5.1.1 Experimental Setup. For each benchmark set, we computed the PTM's prompt using a subset of tasks (question-answer pairs). For DeepRegex, as there was significant training data available we used 10 tasks for the prompt chosen from the training set as described in section 4. For StackOverflow, where there was significantly less training data, we used 5 out of the 25 tasks in the prompt. The remaining tasks were used for evaluation. Following [Chen et al. 2020], each system was given 60 seconds for each task. A task is considered successfully done, if the output expression is *semantically* equivalent to the ground-truth of that task. We used an off-the-shelf tool, RFixer [Pan et al. 2019], for deciding if two expressions are semantically equivalent. If the synthesized program, p, is not equivalent to the ground truth, g, the synthesizer under test is given another attempt with additional examples (up to 10 iterations). In particular, one negative example accepted by p and rejected by g (if it exists) and another positive example accepted by g and rejected by p (if it exists) are added to the task. We relied on RFixer to automatically generate such examples by comparing p and g semantically. This procedure is in accordance with how real users interact

[3] https://stackoverflow.com/questions/19746891

with program synthesizers; once a candidate program is found, the user either accepts it or provides additional examples to guide the synthesizer to find the intended program.

5.1.2 Comparison to the State-of-the-Art. Figure 7 presents the accuracy of NLX-REG and the baseline systems when applied on the DeepRegex (7a) and on the StackOverflow (7b) data-sets. The average results across both data-sets is provided in Figure 7c. All systems performed considerably better on the DeepRegex data-set, where tasks are relatively less complex than those in StackOverflow. Both NLX-REG and REGEL achieve a high accuracy on the DeepRegex data-set by solving 104 (90%) of the cases; while BFS and GPT-3 systems were less successful and only solved 49 (41%) and 21 (18%) of the cases. On the StackOverflow data-set, however, NLX-REG outperforms all baselines by solving 14 (70%) cases. REGEL solved 5 (25%) and BFS and GPT-3 solved respectively 2 (10%) and 1 (5%) cases on this data-set. This shows the effectiveness of our approach in how, despite being domain-agnostic in nature, it was able to meet the performance of the specialized REGEL system with marginal difference on one dataset, and significantly outperform it on the other. Across both data-sets, NLX-REG solves 80% of the tasks, which is 23% better than the closest baseline, REGEL.

To assess how effectively each system leverages additional examples and converges to the ground-truth, we performed further experiments on the subset of benchmarks that both NLX-REG and REGEL successfully solved, in order to compare the number of examples required by the two systems on tasks where both systems succeeded. Figure 8 presents the results. The y-axis shows the number of iterations in which examples are provided before the correct program is obtained, with 0 meaning that the synthesizer's first guess was correct and no additional examples were needed. The x-axis shows the number of benchmarks in each category. On average NLX-REG required 1.5 rounds and REGEL required 2.3 rounds of additional examples. In particular, NLX-REG successfully guessed the ground-truth at first trial in 22 cases; this number for REGEL was 2. Similarly, 35 cases required only one round of additional examples for NLX-REG; this number was 1 for REGEL. Both systems require 3 or more additional rounds for about one third of all cases.

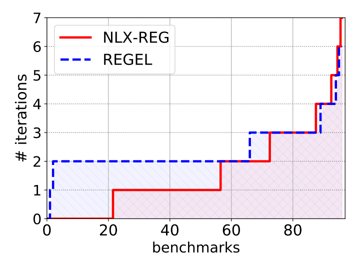
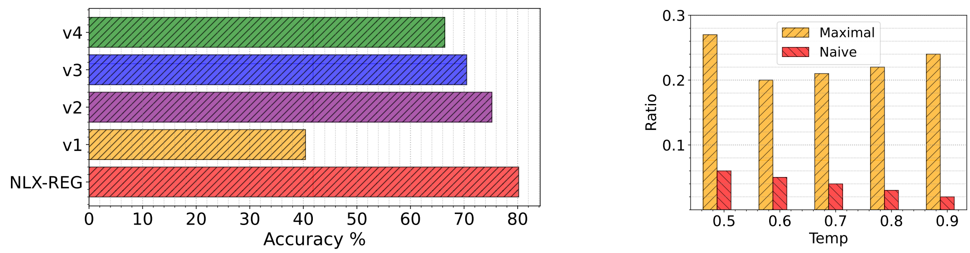


Fig. 8. Number of iterations before success

5.1.3 Ablation Study. In addition to the comparison with the state-of-the-art described above, we also conducted an ablation study [Meyes et al. 2019] with the goal of understanding the impact of the main components of the NLX technique on the overall performance of the system. Specifically, we defined four versions of the system (v1-v4) where each version replaces a specific component of the algorithm with a naive alternative solution. We describe each of these versions below.

In system v1, we initialize the cache with *all* atomic components found in *any* of the PTM candidates, in order to assess the impact of our initialization procedure (discussed in §3.1). In v2 our expansion methodology (§3.2) is replaced with a full application of all DSL operators, i.e. expansion is done by a *complete enumeration* of all valid terms in the DSL. In v3 our ranking procedure (§3.3) is replaced with a function that *randomly* selects the final output of the system. Lastly, v4 represents the full NLX-REG system but where we use a *fixed* set of randomly-chosen cases for the prompt given to the PTM for each task (instead of dynamically choosing the prompt from the full training set that is available). This version is designed to assess the effectiveness of our prompt-generation technique (presented in §4) and also to evaluate the scenario where the user only has a very small amount of training data for the prompt.



(a) performance comparison between different versions (b) ratio of terms appearing in the ground-truth

Fig. 9. Ablation Study of NLX-REG

Figure 9a presents the results from our ablation study. The y-axis is labeled with the versions of the system defined above and the x-axis represents the average success rate across both data-sets. Firstly, we note that all four versions of the system perform more poorly than the full system: specifically, v1 through v4 achieve 40%, 5%, 10% and 14% lower success rates than the full NLX-REG system respectively. This shows how each of the component phases of the NLX technique contributes to the overall performance gain of the NLX-REG system compared to the BFS version defined earlier (the BFS setup can in fact be thought of as an amalgamation of v1, v2 and v3).

We also observe that the most significant degradation of 40% is seen in v1, which highlights the importance of the core initialization phase that uses our maximal components technique. We delved further into this to analyse the *quality* of the PTM's outputs by measuring the ratio of terms appearing in any PTM candidates which also appear in the ground-truth program. Figure 9b presents this analysis for different temperature settings of the GPT-3 model, where temperature is a parameter of the PTM that controls how much randomness occurs in the PTM output. We observe that when considering *all components* from the PTM output, this ratio is generally very small (less than 0.06) for all temperatures and gets smaller as the temperature increases and more randomness occurs in the PTM output. This shows that a large majority of the terms appearing in the PTM's output are redundant and can potentially harm the synthesis procedure. As discussed in §3.1, our initialization technique based on maximal components is designed to address this issue. Figure 9b shows that repeating this experiment but only counting components obtained using our initialization technique, the ratio increases significantly (about 8X) compared to the naive approach. Moreover, there is also not a significant relative decline in the ratio as temperature increases. This illustrates how our initialization procedure effectively declutters the output of the PTM to enhance the overall performance of the system and is also robust to temperature variations that may introduce more randomness.

Finally, we note that the system v4 is not only an experimental instantation for ablation evaluation purposes, but represents the very realistic practical scenario where the user only has a very small amount of training data available (like 10 example pairs only): in such cases our prompt-generation technique of §4 is not applicable as the user can simply provide all the data they have. We observe that while v4 has a significant decrease of 14% compared to the full system, the overall accuracy of v4 is 66%, which is still significantly better than the overall accuracy of the REGEL system which is 57.5% despite it being trained on much more data. This shows that the benefit of the *few-shot learning* capability of PTMs is also exhibited by our multi-modal NLX system because with only a handful of examples it can still perform better than the state-of-the-art system that has been trained on much more data. But of course, our prompt-generation techniques provide significant further gains in the situations where we do have all the training data available.

<table>
<tr><th>#</th><th>Natural Language</th><th>Ground Truth</th><th>Pre-trained Model's Candidates</th></tr>
<tr><td>I</td><td>select just those checkboxes that have values set</td><td>input[value][type="checkbox"]:not([value=""])</td><td>input[type="checkbox"][value]
[type="checkbox"][value]
[checked="true"]
input[type="checkbox"]:checked
[value]</td></tr>
<tr><td>II</td><td>something that matches "(.a or .b) and .c"</td><td>.a.c,.b.c</td><td>.a+.b+.c
.a.b.c
a.b[class*="c"]
[class ="a"][class ="b"] .c
.a|.b[.c]</td></tr>
<tr><td>III</td><td>select the first and the last TD in a row</td><td>tr td:first-child, tr td:last-child</td><td>tr:first-child:last-child
td[last()]
td:first-child:last-child
td:first-child,a:td:last-child
td:not(:first-child):not(:last-child)
td:nth-child(1), a:td:last-child</td></tr>
</table>

Fig. 10. Example tasks for inferring CSS selectors. For each task we show the natural language description, the desired ground truth selector, and a sample of the PTM's top-ranked programs

5.2 Domain of CSS Selectors

Although the main focus in this work is the domain of regular expressions, a notable distinguishing characteristic of our approach is that it is domain-agnostic in nature. This is because the techniques are not designed specifically for the language of regular expressions and can in theory be applicable to other DSLs. Though it is not an extensive exploration of applicability to arbitrary languages, we evaluate the generality aspect of our approach by performing a preliminary evaluation of an implementation of our algorithm in the very different domain of CSS selectors (Cascading Style Sheets) [W3C 2020]. CSS selectors are expressions for selecting elements from the document object model (DOM) of a webpage, based on structural properties that are defined by the HTML source markup of the webpage. We use the language of CSS selectors shown in Figure 3.

Dataset. We collected real-world scenarios from questions about CSS selectors posted on Stack-Overflow. We searched for such questions using the tags "css" and "css-selectors", and as in the previous section, created concise natural language descriptions for the selector based on the description in the question. Some examples of such tasks are shown in Figure 10. Out of 25 such cases we excluded 6 that were using *pseudo-classes* such as `:hover` or `:focus`, which are not static properties of the input webpage and not handled by our CSS parser. This left a total of 19 cases in the dataset. For each of these tasks in this dataset, we also needed a sample input webpage on which one can execute and test the selectors and provide examples of desired elements that should be selected. We synthetically created such a sample webpage by manually examining each of the selectors in the dataset and creating representative HTML structures that contain positive and negative examples for each of the selectors.

System and baselines We implemented our system NLX-CSS for multimodal synthesis of CSS selectors as an instantiation of our generic algorithm from Figure 4 for the CSS domain. The DSL we used was $\mathcal{L}_{\text{CSS}}$ from Figure 3 and the language model $\mathcal{M}_{\text{CSS}}$ was obtained using GPT-3 with few-shot training for the CSS domain. Given the small size of our dataset of only 19 cases, we used 3 of these for the few-shot training examples for GPT-3, and the remaining 16 cases were used as the test set. As we had chosen CSS as a novel domain of study, we are not aware of prior work for multi-modal synthesis of CSS selectors. Hence the two baselines we chose were GPT-3 by itself (the top-ranked program from the PTM model $\mathcal{M}_{\text{CSS}}$ given only the natural language query), and the brute-force multi-modal approach BFS which represents an enumerative search starting from all atomic components of the top-ranked programs from the PTM model $\mathcal{M}_{\text{CSS}}$.

Evaluation We evaluated our system and the two baselines using the test dataset of 16 cases. As in the regex domain, for the multi-modal systems we provided examples iteratively in a CEGIS fashion for a maximum of 10 iterations. A task was considered successfully completed if the synthesized selector is semantically equivalent to the ground truth selector given for that task. As there was no automated equivalence checker for this domain, we performed the equivalence check by manual inspection at every iteration. At each iteration, if the synthesizer under test did not produce the correct selector, then another positive and negative example element was provided from our sample webpage.

The results of our experiments are shown in Figure 11. The relative performance of the three systems are similar to the previous section, with our system NLX-CSS performing the best with 75% accuracy, the brute force approach at 56% and GPT-3 at 20%. As in the regex domain, we observed the benefits of our approach in obtaining relevant components from the PTM candidates and guiding the search based on similarity to these programs. For example, for case I in Figure 10 the initial components included the composite expression `input[type = "checkbox"][value]` which required minor repairs to construct the correct program. In cases II and III we observe the similarity of the operators used in the ground truth and the PTM candidates, even though none of the PTM results were exactly equivalent to the ground truth.

As for the number of examples required by our system to successfully address the task: the average number of examples iterations required to return the correct program was 1.6, with only 2 cases requiring more than 2 iterations.

This is a preliminary evaluation mainly due to the small size of the dataset and the manual work such as equivalence-checking required for experimentation. In particular, using only 3 examples for the few-shot prompt training of GPT-3 was a notable limitation, and we can expect improved performance of all systems with more prompt training examples for GPT-3. This is evident from an examination of the failure cases where the main reason for failure was that the GPT-3 candidates were very different from the ground-truth programs in these cases (often including non-CSS syntax) which meant that many relevant components/operators required for synthesis were missing in these cases. However, while the accuracy of the underlying language model can improve arbitrarily with more prompt-training data or even fine-tuning, the key result of this study is the relative performance of the systems. It demonstrates the added benefit of the synthesis techniques to address the challenging cases that cannot be directly handled by the language model and require further interaction with examples.

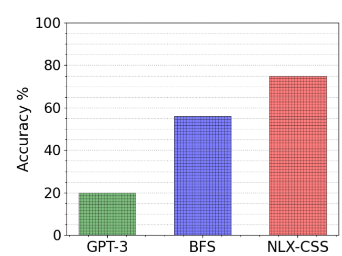


Fig. 11. Evaluation of NLX-CSS

6 RELATED WORK

Regex Synthesis: There is a large body of prior work on synthesizing regular expressions from examples. Angluin presented algorithms to learn finite state automata and regular expressions from a given set of positive and negative examples [Angluin 1978, 1987]. Recently, Mina et.al revisited the problem of learning regular expressions from introductory automata assignments using both positive and negative examples, which leverages ideas of over and under approximation to reduce the search space [Lee et al. 2016]. Since it is hard to construct regular expressions correctly, there is also work on the problem of repairing regular expressions to help developers. Given an incorrect regular expression and a set of positive and negative examples, RFixer returns the closest regex (to the original one) that satisfies the examples [Pan et al. 2019]. Our system is different in that it takes both NL and examples to find the intended program with much fewer interactions.

Researchers have been looking at natural language as a specification to generate regular expressions. Kushman et. al. proposed a semantic parser to synthesize regular expressions from natural language descriptions [Kushman and Barzilay 2013]. With the rise of deep learning, there is recent work that formulates this problem as a machine translation problem (seq2seq) to translate descriptions to regular expressions [Locascio et al. 2016; Zhong et al. 2018]. Unlike these systems, we also allow examples in addition to natural language to refine the intent.

Finally, there is some recent work on regular expression synthesis using a combination of natural language and examples, which we will discuss below.

Multimodal Synthesis: Because different specification modalities have different characteristics (e.g., natural language description is versatile but ambiguous while examples are sound but incomplete), recent work on *multi-modal synthesis* has been leveraging combinations of multiple types of specifications. Manshadi et. al. discussed a probabilistic PBE system to perform string transformation using examples and natural language [Manshadi et al. 2013]. This work extended the version-space-algebra in [Gulwani 2011] by allowing the edges to carry probabilities calculated from both program properties and natural language descriptions. Raza et. al. presented a multi-modal synthesis system that first maps descriptions into various concepts and uses the examples to refine the concepts [Raza et al. 2015]. MARS is a system that synthesizes data wrangling operations from a combination of input-output examples, natural language description, and partial code snippets [Chen et al. 2019]. Their technique uses a combination of sequence to sequence (seq2seq) model to maps the description to an abstract program (sketch) and the apriori algorithm to mine the association rules. The entire problem was then reduced to a Max-SMT problem.

Multimodal Regex Synthesis: Recently, there has been some work on synthesis of regular expressions from a natural language description and input-output examples [Chen et al. 2020; Li et al. 2020]. The Regel system [Chen et al. 2020] first uses a semantic parser to parse a description into a sketch, then completes the sketch using enumerative search guided by the examples. In contrast, we utilize a PTM to generate components from the description and then use a novel CBS synthesis algorithm, which is guided by the PTM output, to generate a program that satisfies the examples. We show higher accuracy of our technique on real-world benchmarks in comparison to the Regel system. Furthermore, while Regel is designed specifically for the regular expression domain, our approach is domain-agnostic and applicable to other programming domains.

Another recent related work in this area is the TransRegex system [Li et al. 2020]. TransRegex is based on two distinct phases of first generating a best regex using an NL model, and then an independent examples-based repair technique to repair this best regex. While we were unable to find an implementation of this system for direct evaluation, their reported accuracy on realistic scenarios from Stack Overflow reaches towards 70% which is similar to our results. The key difference again is that this system is highly specialized for the Regex domain, while our approach is domain-agnostic and applicable in at least one other domain of CSS. In terms of technique, in contrast to the two distinct-phase approach of TransRegex, our approach more tightly integrates synthesis and NL by using the *set* of top candidate programs to guide the synthesis at multiple stages (initialization, expansion and ranking). This tight integration has the benefits that it can "mix and match" likely components that may not all necessarily occur in the top program, and can also analyse patterns of operator occurrences across the top programs to infer the overall shape of the target program. We also note that the major contributions of [Li et al. 2020] are around training of the model to reward syntactically valid regular expressions and to bake in semantic equivalence of regular expressions –

these steps are orthogonal to our work and may even be applied as domain-specific optimizations on the output of GPT-3 to improve our system further.

Enumerative Synthesis: Enumerative search is one of the simplest program synthesis techniques, yet is proven to be effective for synthesizing small programs in complex search space [Alur et al. 2013; Alur et al. 2015, 2017]. Alur et. al. formalized the syntax-guided synthesis (SyGus) problem (where the search space are programs in a CFG) and proposed three different instantiations of the counter-example-guided-inductive-synthesis (CEGIS) strategy [Alur et al. 2013]. Subsequently, [Alur et al. 2015] extends CEGIS with through unification, where the idea is to unify different programs that satisfy different parts of the inputs. EUSolver makes the enumeration process more efficient by employing a divide-and-conquer approach [Alur et al. 2017]. In addition to techniques that are based on program size, researcher also proposed new search techniques such as abstraction-based [Drachsler-Cohen et al. 2017; Feng et al. 2017a; Polikarpova et al. 2016], constraint-based [Jha et al. 2010; Solar-Lezama 2008; Srivastava et al. 2010], deep-learning-based[Balog et al. 2017].

Raza et. al. introduced *predictive synthesis*, in which the synthesizer learns a data wrangling program from just the input (without the output example) [Raza and Gulwani 2017]. Their approach enumerates the program literals bottom-up and has a search strategy that biases conforming programs. Some works have also looked at combining enumerative and deductive synthesis [Huang et al. 2020; Raza and Gulwani 2020]. Our approach also employs enumerative synthesis, but instead of generating components from scratch, we employ a PTM to generate *maximal* components and utilize a novel search technique to synthesize the final regex.

Closed frequent itemset mining: Our goal of finding the most valuable initial components has similarities to the field of frequent pattern mining in databases [Agrawal et al. 1993]. Similar notions of frequency and redundancy are also considered in *closed frequent item-set mining* [Pei et al. 2000] where the goal is to find frequent sets while also avoiding redundancy by not finding subsets with the same support. However, the underlying focus in this field is on “association rules” that follow a flat set-based structure and exist in independent records of the database. The key conceptual difference in our case is that the entities of interest are not sets but structured AST components that may be nested inside one another, and where redundancy comes from the sub-tree rather than subset relation.

Natural Language to Code: There have been numerous proposals to generate different kinds programs from natural language, including SQL queries [Huang et al. 2018; Wang et al. 2020; Yaghmazadeh et al. 2017], smartphone automation scripts [Le et al. 2013], spreadsheet formulas [Gulwani and Marron 2014], bash [Lin et al. 2018]. SQLizer generates a sketch from natural language, then refines it using probabilistic type inhabitation and automated sketch repair [Yaghmazadeh et al. 2017]. SmartSynth combines semantic parser with type-based synthesis to generate phone automation scripts [Le et al. 2013]. NLyze introduced a translation algorithm that utilizes spatial and temporal context in the spreadsheet [Gulwani and Marron 2014]. Recently, RAT-SQL tackles the NL to SQL problem by using relation-aware self-attention to incorporate reasoning that involves both question entities and database schema [Wang et al. 2020]. Unlike the above techniques, our approach does not require explicit supervision with large curated datasets for particular domains, as it leverages the strength of PTMs to provide robust coverage of various domains with few-shot learning.

Natural Language to Code using PTMs: There has not been much published work on code generation using large language models, e.g. GPT-3. There is however plenty ongoing activity and we expect various techniques to emerge in near future. Codex [Chen et al. 2021] is GPT-3 fine-tuned

#	Natural Language	Ground Truth	Pre-trained Model's Candidates
I	Convert a dataframe to a dictionary, where row values of 'HP' are the keys and the other columns are the values	df.set_index('HP') .to_dict(orient='index')	pd.to_dict(df, orient='index') df.to_dict(orient='index', prefix='HP') df.to_dict(orient='index') df.to_dict(orient='index', axis=1) pd.DataFrame(df2).to_dict(orient='records') df.to_dict(orient='index', axis=1)
II	group by 'HP' and 'Type 1' and calculate mean and count for each group	df.groupby(['HP', 'Type 1']) .agg(['mean', 'count'])	df.groupby('HP', 'Type 1').mean().agg(['count', 'sum']).fillna(0) df.groupby(['HP', 'Type 1']).mean() df.groupby('HP','Type 1').apply(np.mean) df.groupby('HP','Type 1').apply(np.count)
III	get list of columns grouped by datatype of column	df.columns.to_series() .groupby(df.dtypes).groups	df.groupby(['dtype']) df['name'].groupby('type').count() Code: df.groupby('col').columns df.groupby('column_type').columns.value_counts() grouped_cols = df.groupby('Type')

Fig. 12. Pandas Examples: The NL description of task, the associated ground truth, and the candidates generated by the PTM model. The last column only shows a few selected candidates.

on code, and generates Python code from docstrings in about 30% of cases. In contrast, for restricted domains, and using synthesis as a post-processor for GPT-3 as described here, we are able to get much higher precision. Going forward, fine-tuned models can be used along with synthesis-based post-processing to build powerful NL to code systems. Hendrycks et al. [2021] introduced a large benchmark set for coding tasks (dubbed APPS), that can be used to systematically evaluate the ability of such techniques in using various data-structures and programming techniques. These benchmarks assume a general purpose language (Python) which is currently not what our NLX system is targeting. However, we think APPS is a valuable framework to track advancements in program synthesis research and would be interesting to explore in our future works.

7 DISCUSSION AND FUTURE WORK

NLX is a general approach for multimodal synthesis that combines the strengths of PTMs and program synthesis. Its effectiveness is based on the underlying hypotheses that (A) the multiple candidates returned by PTMs contain the components of the ground truth, even though they may not contain the whole ground truth, (B) furthermore, the candidates reveal (approximately) the distribution/frequency of the operators in the ground truth, (C) users can provide input-output examples to refine their intent in case the synthesis engine does not return their desired program, and (D) subprograms (subterms) can be executed on inputs (obtained from the examples) to determine (approximate) semantic equivalence of these subprograms.

While our experimental evaluation has focused on regular expressions and CSS selectors, we have observed that some of these assumptions, specifically (A) and (B), hold in general. There exist domains where assumptions (C) and particularly (D) are not easily satisfied, but even in those cases, the NLX approach can be adapted by replacing the steps that rely on examples by alternate steps that rely on other forms of intent specification.

A particularly interesting domain is that of Python's data processing library Pandas. Pandas is popular among data scientists for writing scripts that can be used to ingest data, clean data, reshape and manipulate data, and visualize data. Pandas is a good target for generating code from NL descriptions because (a) it is widely used, including by non-programmers, and (b) it has a very large API, and it is very difficult to remember the API details, especially for an occassional user.

We validated hypotheses (A) and (B) for the Pandas domain by collecting NL descriptions along with ground-truth expressions from stackoverflow posts about Pandas. We then used a PTM with dynamic prompt to generate 25 candidate programs for the NL descriptions. We then analyzed if the candidates have the components used in the ground-truth program. There were around 40% benchmarks where the ground-truth program was present in the 25 candidates. Figure 12 shows three instances when the ground truth was not present in the candidates returned by PTM. There were three classes of instances. In the first class, the candidates contained all components used in the ground truth. For example, consider the ground-truth program df.groupby(['HP', 'Type 1']).agg(['mean', 'count']) that corresponds to the NL description "group by HP and Type 1 and calculate mean and count for each group".Among the generated candidates, we find the maximal components df.groupby, ['HP', 'Type 1'], agg, np.mean, and 'count', which can be combined to give the program df.groupby(['HP', 'Type 1']).agg([np.mean, 'count']), which is equivalent to the ground-truth program. The other two examples in Figure 12 show cases where the candidates do not contain all the components needed to recreate the ground truth, but they contain many of the components. In the first row, only one component, namely set_index is missing, whereas in the last row three are missing, namely to_series, df.dtypes, and groups.

A key difficulty in using NLX for synthesizing Pandas code is that assumptions (C) and (D) are harder to satisfy. In such cases, we need to adapt the approach and extend it with other methods, such as, the use of types to suggest repairs, which we leave for future work. We can also extend NLX by generalizing the notion of components to also include *sketches*, or terms with holes. Most of the steps in our algorithm will generalize to using sketches as components, except for steps that require assumption (D). Adapting NLX to also use sketches as components is an interesting direction for future work. It is also possible to consider fine-tuning the pre-trained models. Fine tuning requires more data, but it also provides more value by giving a good set of initial candidates to the synthesis procedure. There is also the future possibility of employing constrained decoding to guarantee that the pre-trained model only generates valid code in the target language.

While we have discussed applicability to various specialized programming domains, it is also an interesting question to ask if such techniques can be applicable to much more expressive general purpose programming languages such as Python, Java or C#. In practice we do not expect our techniques to directly scale to large programs in such highly expressive languages. However, it is an interesting research direction to build upon our ideas here. For instance, initial experiments on small code snippets in C# suggest that a *compositional* approach to multi-modal synthesis may be valuable: instead of just input-output examples, if the user can provide "traces" of examples over some pseudo-code in natural language then that may more strongly guide the system to scale to more complex programs. These will be interesting explorations for future work.

8 CONCLUSIONS

This paper presents a novel technique for synthesizing programs from natural language descriptions and examples. We introduce a domain-agnostic algorithm that leverages the ability of modern pre-trained language models to provide probability distributions over program components from ambiguous natural language descriptions, and uses them to guide a novel component-based approach for synthesis from examples. We instantiated our algorithm for two programming domains – the domains of regular expressions and CSS selectors. The experimental results suggest effectiveness of this approach on both domains. Most notably, our domain-agnostic synthesizer when specialized to the domain of regular expressions outperforms the state-of-the-art and highly-specialized synthesizer for this domain.

REFERENCES

R. Agrawal, T. Imielinski, and A. Swami. 1993. Mining Association Rules Between Sets of Items in Large Databases. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data (SIGMOD '93)*, Vol. 22. ACM, New York, NY, USA, 207–216. https://doi.org/10.1145/170035.170072

R. Alur, R. Bodik, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. 2013. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*. 1–8. https://doi.org/10.1109/FMCAD.2013.6679385

Rajeev Alur, Pavol Cerny, and Arjun Radhakrishna. 2015. Synthesis Through Unification. In *Computer Aided Verification (CAV)*. https://www.microsoft.com/en-us/research/publication/synthesis-through-unification/

Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. 2017. Scaling Enumerative Program Synthesis via Divide and Conquer. In *Tools and Algorithms for the Construction and Analysis of Systems*, Axel Legay and Tiziana Margaria (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 319–336.

Dana Angluin. 1978. On the complexity of minimum inference of regular sets. *Information and Control* 39, 3 (1978), 337–350. https://doi.org/10.1016/S0019-9958(78)90683-6

Dana Angluin. 1987. Learning Regular Sets from Queries and Counterexamples. *Inf. Comput.* 75, 2 (Nov. 1987), 87–106. https://doi.org/10.1016/0890-5401(87)90052-6

Matej Balog, Alexander Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2017. DeepCoder: Learning to Write Programs. In *Proceedings of ICLR'17* (proceedings of iclr'17 ed.). https://www.microsoft.com/en-us/research/publication/deepcoder-learning-write-programs/

Paul E. Black. 1999. Dictionary of Algorithms and Data Structures [online]. https://www.nist.gov/dads/HTML/Levenshtein.html(Accessed,March2021)

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 1877–1901. https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf

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

Qiaochu Chen, Xinyu Wang, Xi Ye, Greg Durrett, and Isil Dillig. 2020. Multi-Modal Synthesis of Regular Expressions. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 487–502. https://doi.org/10.1145/3385412.3385988

Yanju Chen, Ruben Martins, and Yu Feng. 2019. Maximal multi-layer specification synthesis. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 602–612.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. https://doi.org/10.18653/v1/N19-1423

Dana Drachsler-Cohen, Sharon Shoham, and Eran Yahav. 2017. Synthesis with Abstract Examples. In *Computer Aided Verification*, Rupak Majumdar and Viktor Kunčak (Eds.). Springer International Publishing, Cham, 254–278.

Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017a. Component-Based Synthesis of Table Consolidation and Transformation Tasks from Examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 422–436. https://doi.org/10.1145/3062341.3062351

Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W. Reps. 2017b. Component-based synthesis for complex APIs. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. ACM. https://doi.org/10.1145/3009837.3009851

Sumit Gulwani. 2011. Automating String Processing in Spreadsheets using Input-Output Examples. In *PoPL'11, January 26-28, 2011, Austin, Texas, USA*. https://www.microsoft.com/en-us/research/publication/automating-string-processing-spreadsheets-using-input-output-examples/

Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. 2011. Synthesis of Loop-Free Programs. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*.

Association for Computing Machinery, New York, NY, USA, 62–73. https://doi.org/10.1145/1993498.1993506

Sumit Gulwani and Mark Marron. 2014. NLyze: Interactive Programming by Natural Language for SpreadSheet Data Analysis and Manipulation. In *SIGMOD '14 Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (sigmod '14 proceedings of the 2014 acm sigmod international conference on management of data ed.). Association for Computing Machinery, 803–814. https://www.microsoft.com/en-us/research/publication/nlyze-interactive-programming-by-natural-language-for-spreadsheet-data-analysis-and-manipulation/

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring Coding Challenge Competence With APPS. *CoRR* abs/2105.09938 (2021). arXiv:2105.09938 https://arxiv.org/abs/2105.09938

Kangjing Huang, Xiaokang Qiu, Peiyuan Shen, and Yanjun Wang. 2020. Reconciling Enumerative and Deductive Program Synthesis. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 1159–1174. https://doi.org/10.1145/3385412.3386027

Po-Sen Huang, Chenglong Wang, Rishabh Singh, Wen-tau Yih, and Xiaodong He. 2018. Natural Language to Structured Query Generation via Meta-Learning. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*. Association for Computational Linguistics, New Orleans, Louisiana, 732–738. https://doi.org/10.18653/v1/N18-2115

Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-Guided Component-Based Program Synthesis. In *ICSE '10, May 2-8 2010, Cape Town, South Africa* (icse '10, may 2-8 2010, cape town, south africa ed.). https://www.microsoft.com/en-us/research/publication/oracle-guided-component-based-program-synthesis/

K. Jones. 1972. A statistical interpretation of term specificity and its application in retrieval. *J. Documentation* 60 (1972), 493–502. https://doi.org/10.1108/eb026526

Nate Kushman and Regina Barzilay. 2013. Using Semantic Unification to Generate Regular Expressions from Natural Language. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics, Atlanta, Georgia, 826–836. https://www.aclweb.org/anthology/N13-1103

Vu Le, Sumit Gulwani, and Zhendong Su. 2013. SmartSynth: Synthesizing Smartphone Automation Scripts from Natural Language. In *MobiSys'13, June 25-28, 2013, Taipei, Taiwan* (mobisys'13, june 25-28, 2013, taipei, taiwan ed.). https://www.microsoft.com/en-us/research/publication/smartsynth-synthesizing-smartphone-automation-scripts-natural-language/

Mina Lee, Sunbeom So, and Hakjoo Oh. 2016. Synthesizing Regular Expressions from Examples for Introductory Automata Assignments. *SIGPLAN Not.* 52, 3 (Oct. 2016), 70–80. https://doi.org/10.1145/3093335.2993244

Yeting Li, Shuaimin Li, Zhiwu Xu, Jialun Cao, Zixuan Chen, Yun Hu, Haiming Chen, and Shing-Chi Cheung. 2020. TransRegex: Multi-modal Regular Expression Synthesis by Generate-and-Repair. *CoRR* abs/2012.15489 (2020). arXiv:2012.15489 https://arxiv.org/abs/2012.15489

Xi Victoria Lin, Chenglong Wang, Luke Zettlemoyer, and Michael D. Ernst. 2018. NL2Bash: A Corpus and Semantic Parser for Natural Language Interface to the Linux Operating System. In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018)*. European Language Resources Association (ELRA), Miyazaki, Japan. https://www.aclweb.org/anthology/L18-1491

Nicholas Locascio, Karthik Narasimhan, Eduardo DeLeon, Nate Kushman, and Regina Barzilay. 2016. Neural Generation of Regular Expressions from Natural Language with Minimal Domain Knowledge. In *EMNLP* (emnlp ed.). https://www.microsoft.com/en-us/research/publication/neural-generation-regular-expressions-natural-language-minimal-domain-knowledge/

Mehdi Manshadi, Daniel Gildea, and James Allen. 2013. Integrating programming by example and natural language programming. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 27.

Richard Meyes, Melanie Lu, Constantin Waubert de Puiseau, and Tobias Meisen. 2019. Ablation Studies in Artificial Neural Networks. *CoRR* abs/1901.08644 (2019). arXiv:1901.08644 http://arxiv.org/abs/1901.08644

OpenAI. 2021. GPT-3 powers the next generation of apps. https://openai.com/blog/gpt-3-apps/.

Rong Pan, Qinheping Hu, Gaowei Xu, and Loris D'Antoni. 2019. Automatic Repair of Regular Expressions. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 139 (Oct. 2019), 29 pages. https://doi.org/10.1145/3360565

Jian Pei, Jiawei Han, and Runying Mao. 2000. CLOSET: An Efficient Algorithm for Mining Frequent Closed Itemsets.. In *ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery* (2002-01-03). 21–30. http://dblp.uni-trier.de/db/conf/dmkd/dmkd2000.html#PeiHM00

Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program Synthesis from Polymorphic Refinement Types. *SIGPLAN Not.* 51, 6 (June 2016), 522–538. https://doi.org/10.1145/2980983.2908093

Mohammad Raza and Sumit Gulwani. 2017. Automated Data Extraction Using Predictive Program Synthesis. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence (AAAI'17)*. AAAI Press, 882–890.

Mohammad Raza and Sumit Gulwani. 2020. Web Data Extraction Using Hybrid Program Synthesis: A Combination of Top-down and Bottom-up Inference. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1967–1978. https://doi.org/10.1145/3318464.3380608

Mohammad Raza, Sumit Gulwani, and Natasa Milic-Frayling. 2015. Compositional Program Synthesis from Natural Language and Examples. In *IJCAI 2015* (ijcai 2015 ed.). https://www.microsoft.com/en-us/research/publication/compositional-program-synthesis-natural-language-examples/

Armando Solar-Lezama. 2008. *Program Synthesis by Sketching*. Ph.D. Dissertation. USA. Advisor(s) Bodik, Rastislav. AAI3353225.

Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. 2010. From Program Verification to Program Synthesis. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '10)*. Association for Computing Machinery, New York, NY, USA, 313–326. https://doi.org/10.1145/1706299.1706337

W3C. 2020. CSS Snapshot 2020 [online]. https://www.w3.org/TR/CSS/

Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. 2020. RAT-SQL: Relation-Aware Schema Encoding and Linking for Text-to-SQL Parsers. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Online, 7567–7578. https://doi.org/10.18653/v1/2020.acl-main.677

Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. 2017. SQLizer: Query Synthesis from Natural Language. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 63 (Oct. 2017), 26 pages. https://doi.org/10.1145/3133887

Zexuan Zhong, Jiaqi Guo, Wei Yang, Jian Peng, Tao Xie, Jian-Guang Lou, Ting Liu, and Dongmei Zhang. 2018. SemRegex: A Semantics-Based Approach for Generating Regular Expressions from Natural Language Specifications. In *EMNP'18*. ACL. https://www.microsoft.com/en-us/research/publication/semregex-a-semantics-based-approach-for-generating-regular-expressions-from-natural-language-specifications-2/

A SUPPLEMENTARY DEFINITIONS

In this section we present the formal semantics of the DSLs introduced in the paper. Figure 13 presents the semantics for regular expressions and Figure 14 presents the semantics for the domain of CSS selectors.

$$
\begin{array}{lll}
[[\mathsf{i}]](s) = \bot & & \text{for all } \mathsf{i} \text{ in } \{0, 1, \ldots\} \\
[[\mathsf{c}]](s) = \bot & & \text{for all } \mathsf{c} \text{ in } \{\mathsf{A}, \mathsf{B}, \ldots\} \\
[[\mathsf{fromChar}(\mathsf{c})]](s) = \top & \text{iff} & s == \mathsf{c} \\
[[\mathsf{range}(\mathsf{c}_1, \mathsf{c}_2)]](s) = \top & \text{iff} & s == \mathsf{c} \text{ for some } c \text{ that lies between } \mathsf{c}_1 \text{ and } \mathsf{c}_2 2 \\
[[\mathsf{union}(\mathsf{s}_1, \mathsf{s}_2)]](s) = \top & \text{iff} & [[s_1]](s) = \top \text{ or } [[s_2]](s) = \top \\
[[\mathsf{negate}(s)]](s) = \top & \text{iff} & [[s]](s) = \bot \\
[[\mathsf{any}()]](s) = \top & & \\
[[\mathsf{quant}(\mathsf{e}, \mathsf{i}, \mathsf{j})]](s) = \top & \text{iff} & s = s_1 s_2 \ldots s_k, i \le k \le j, [[e]](s_l) = \top \text{ for all } l \in \{1, \ldots, k\} \\
[[\mathsf{quantMin}(\mathsf{e}, \mathsf{i})]](s) = \top & \text{iff} & s = s_1 s_2 \ldots s_j, j \ge i, [[e]](s_k) = \top \text{ for all } k \in \{1, \ldots, j\} \\
[[\mathsf{alter}(\mathsf{e}_1, \mathsf{e}_2)]](s) = \top & \text{iff} & [[e_1]](s) = \top \text{ or } [[e_2]](s) = \top \\
[[\mathsf{concat}(\mathsf{e}_1, \mathsf{e}_2)]](s) = \top & \text{iff} & s = s_1 s_2, [[e_1]](s_1) = [[e_2]](s_2) = \top \\
[[\mathsf{fromCharSet}(\mathsf{s})]](s) = \top & \text{iff} & [[s]](s) = \top
\end{array}
$$

Fig. 13. The semantics of regular expressions DSL

$[[\mathsf{i}]]$	=	$\{i\}$ for all number literals i
$[[\mathsf{MultipleOffset(i, j)}]]$	=	$\{j, i+j, 2i+j, 3i+j, \ldots\}$
$[[\mathsf{s}]]$	=	s for all string literals s
$[[\mathsf{Any()}]]$	=	the set of all nodes in the input document
$[[\mathsf{Union}(\mathsf{n}_1, \mathsf{n}_2)]]$	=	$[[\mathsf{n}_1]] \cup [[\mathsf{n}_2]]$
$[[\mathsf{Not}(\mathsf{n}_1, \mathsf{n}_2)]]$	=	$[[\mathsf{n}_1]] - [[\mathsf{n}_2]]$ where $-$ denotes set difference
$[[\mathsf{TagEquals(n, s)}]]$	=	$\{\mathsf{node} \in [[\mathsf{n}]] \mid$ the tag of node is "s"$\}$
$[[\mathsf{nthChild}(n, i)]]$	=	$\{\mathsf{node} \in [[\mathsf{n}]] \mid$ node is the k-th child of its parent for some k in $[[\mathsf{i}]]\}$
$[[\mathsf{nthLastChild}(n, i)]]$	=	$\{\mathsf{node} \in [[\mathsf{n}]] \mid$ node is the k-th child for $k \in [[\mathsf{i}]]$, counting from the end, of its parent$\}$
$[[\mathsf{AttributeEquals}(\mathsf{n}, \mathsf{s}_1, \mathsf{s}_2]]$	=	$\{\mathsf{node} \in [[\mathsf{n}]] \mid$ node has an attribute s_1 that is set to $s_2\}$
$[[\mathsf{AttributeContains}(\mathsf{n}, \mathsf{s}_1, \mathsf{s}_2]]$	=	$\{\mathsf{node} \in [[\mathsf{n}]] \mid$ node has an attribute s_1 whose value contains s_2 as a substring$\}$
$[[\mathsf{AttributeStartsWith}(\mathsf{n}, \mathsf{s}_1, \mathsf{s}_2]]$	=	$\{\mathsf{node} \in [[\mathsf{n}]] \mid$ node has an attribute s_1 whose value starts with $s_2\}$
$[[\mathsf{AttributeEndsWith}(\mathsf{n}, \mathsf{s}_1, \mathsf{s}_2]]$	=	$\{\mathsf{node} \in [[\mathsf{n}]] \mid$ node has an attribute s_1 whose value ends with $s_2\}$
$[[\mathsf{RightSibling}(\mathsf{n}_1, \mathsf{n}_2)]]$	=	$\{\mathsf{node} \in [[\mathsf{n}_2]] \mid$ node is preceded by some node in $[[\mathsf{n}_1]]$ with the same parent$\}$
$[[\mathsf{Children}(\mathsf{n}_1, \mathsf{n}_2)]]$	=	$\{\mathsf{node} \in [[\mathsf{n}_2]] \mid$ node is the child of some node in $[[\mathsf{n}_1]]\}$
$[[\mathsf{Descendants}(\mathsf{n}_1, \mathsf{n}_2)]]$	=	$\{\mathsf{node} \in [[\mathsf{n}_2]] \mid$ node is the descendant of some node in $[[\mathsf{n}_1]]\}$

Fig. 14. The semantics of CSS expressions DSL