Synthesizing Parsing Expression Grammars via Analysis-based Pruning

Seongmin Ko[0009−0003−2556−198X] and Jihyeok Park(✉)[0000−0001−8387−1984]

Korea University, Seoul, South Korea
{2018320221,jihyeok_park}@korea.ac.kr

Abstract. Parsing Expression Grammars (PEGs) are widely adopted for their linear-time parsing and expressive power for complex languages. However, due to their unintuitive semantics, constructing PEGs manually is often error-prone, even for experts, and can lead to subtle bugs that are difficult to detect. This paper proposes SynPEG, the first tool to automatically synthesize general PEGs from input-output examples. SynPEG enumerates candidate PEGs by filling holes in a top-down manner and prunes partial PEGs using two complementary analyses: 1) *grammar analysis* and 2) *example analysis*. For precise and sound example analysis, we introduce an *abstract machine* for PEGs that explicitly captures the *continuation*, which represents the remaining parsing process after halting at a hole. Evaluation on 15 benchmarks shows that SynPEG prunes 99.82% of the search space and successfully synthesizes correct PEGs for all tasks in 0.43s on average.

1 Introduction

Parsing expression grammars (PEGs) [13, 15] have become a mainstream formalism for defining parsers. While PEGs offer unambiguous semantics, they still have the power to express complex patterns and support linear-time parsing via packrat parsing [12]. Such advantages make PEGs highly attractive for modern software development: Python adopted PEGs for its official grammar in version 3.9[1], LPeg established PEGs as a robust foundation for pattern matching in Lua, and libraries such as PEG.js and PyParsing command millions of weekly downloads and dependents across the open-source ecosystem.

CFGs vs. PEGs While context-free grammars (CFGs) are *generative* grammars, PEGs are *recognition-based* systems. A CFG G defines a language as a set of strings $[\![G]\!] : \mathcal{P}(V_T^*)$ with production rules that describe how to generate valid strings. However, a PEG P defines a parsing process as a function $[\![P]\!] : V_T^* \to (V_T^* \uplus \{\mathbf{f}\})$ from input strings to remaining strings after parsing. For instance, a simple CFG $S \to \mathsf{ab}$ represents the singleton language $\{\mathsf{ab}\}$. In contrast, the corresponding PEG $P = \mathsf{ab}$ consumes the prefix ab from any string starting with it (e.g., $P(\mathsf{abbb}) = \mathsf{bb}$) but fails otherwise (e.g., $P(\mathsf{aab}) = \mathbf{f}$). In

[1] PEP 617 – New PEG Parser for CPython (https://peps.python.org/pep-0617/)

addition, PEGs introduce 1) *prioritized choice* (e_1 `/` e_2) for deterministic parsing and 2) *syntactic predicates* (`&`e and `!`e) to increase their expressive power. The expression e_1 `/` e_2 tries to parse with e_1 first and tries e_2 only if e_1 fails to parse. A not-predicate `!`e succeeds without consuming input if e fails and fails if e succeeds, and an and-predicate `&`e is a syntactic sugar for `!!`e. While it is an open problem whether PEGs are strictly more expressive than CFGs, PEGs can express some context-sensitive patterns. For example, the following PEG P' consumes the same number of `a`'s, `b`'s, and `c`'s (at least one of each) in the correct order (e.g., P'(`aabbccabba`) = `abba`), which is impossible to express with a CFG:

$$P' \quad = \quad \&(X\ \texttt{c})\ \texttt{a*}\ Y; \quad X \leftarrow \texttt{a}\ X\ \texttt{b}\ /\ \epsilon; \quad Y \leftarrow \texttt{b}\ Y\ \texttt{c}\ /\ \epsilon$$

It first checks that the input starts with the same number of `a`'s and `b`'s followed by at least one `c` using `&`(X `c`). Then, it consumes all `a`'s using `a`* and the same number of `b`'s and `c`'s using Y.

Unintuitive semantics of PEGs Despite their popularity, constructing correct PEGs is not straightforward because prioritized choice operators introduce unintuitive semantics that significantly differ from CFGs. For example, in the PEG `a`* `/ abb`, the right expression `abb` is never evaluated because the left expression `a`* always succeeds. To fix this issue, we need to carefully order the expression as `abb / a`*. In addition, such orders can lead to more subtle issues when combined with recursion. A CFG $S \rightarrow$ `a` S `a` | `aa` defines the language of even-length `a`'s (i.e., $\{\texttt{a}^{2n} \mid n > 0\}$), but the corresponding PEG $P'' = S$`!a`; $S \leftarrow$ `a` S `a / aa` consumes `a`'s only if the input starts with exactly power of two `a`'s (i.e., $P''(\texttt{a}^{2^n}\texttt{b}w) = \texttt{b}w$ for some $n > 0$). To resolve such unintuitive semantics, users often need to use appropriate syntactic predicates to enforce specific contexts. For instance, the following PEG represents a capture pattern with an identifier for pattern matching statements in Python 3.14 (e.g., a capture pattern with `x` in the match `case x : ...`):

```
pattern_capture_target: !"_" NAME !('.' | '(' | '=')
```

To avoid overlapping with wildcard patterns (e.g., `_`), the Python grammar uses a not-predicate `!"_"` to check the identifier name does not start with an underscore. Similarly, it uses `!('.' | '(' | '=')` at the end of the rule to ensure that the name is not used as a part of attribute, class, or keyword patterns. To appropriately use such predicates, users need to have a deep understanding of PEG semantics and carefully analyze the grammar structure and input strings, which is difficult even for experts and often leads to subtle errors.

Programming-by-Example (PBE) for Grammars PBE is a promising approach to automate PEG construction and mitigate these pitfalls, as it allows users to specify the intended semantics via input-output examples. While Wieczorek et al. [38] present example-based PEG synthesis, they only focus on *non-circular* PEGs, which is a severely limited subset, strictly less expressive than

regular languages. On the other hand, example-based synthesis and oracle-guided inference of *generative* grammars have been extensively studied for regular expressions (REs) [22], CFGs [4, 5, 20, 26], visibly pushdown grammars [17], and semantic REs [7]. However, they are not directly applicable because PEGs are recognition-based rather than generative, requiring reasoning about both acceptance and remaining inputs. Furthermore, the shadowing effect of prioritized choice renders modular synthesis ineffective. The process must consider the entire grammar simultaneously, which significantly inflates the search space.

Our solution To address these challenges, we present the automated synthesis algorithm for PEGs using analysis-based pruning. It enumerates candidate PEGs by filling holes in a top-down manner and prunes them using two complementary analyses: *grammar analysis* and *example analysis*. The grammar analysis prunes partial PEGs that are 1) reducible or 2) ill-formed by analyzing their structure using an abstract interpretation framework [9, 10]. In contrast, the example analysis prunes partial PEGs if their analysis results are 1) invalid for a single example or 2) inconsistent across multiple examples. To achieve precise and sound example analysis, we present an *abstract machine* for PEGs that explicitly captures *continuations*, which represent remaining parsing processes of a given partial PEG when it halts at a hole during parsing an input string. The continuation allows us to check whether the remaining parsing process can satisfy the expected output for each example, or whether two examples that stopped with the same continuation have consistent expected outputs. In addition, we propose a potential score for partial PEGs to guide the search process, alongside a trie-based abstract and incremental parsing to accelerate the example analysis.

We evaluate SynPEG on 15 benchmarks. While the baseline solves only 5 within a 5-minute timeout, SynPEG solves all benchmarks in 0.43s on average by pruning 99.82% of the search space, which achieves a speedup of 389.68×. These results demonstrate the feasibility of automating PEG construction, offering a promising path to reducing the manual effort required to construct PEGs.

The main contributions of this paper are as follows:

- We present the first synthesis algorithm for general PEGs from examples.
- We present *analysis-based pruning* techniques for PEGs that leverage both grammar structure and examples to effectively reduce the search space.
- We design a novel *abstract machine* to capture the semantics of PEGs with explicit *continuations*, enabling precise example analysis for pruning.
- We implement these techniques in SynPEG and evaluate it on 15 benchmarks and show that it successfully synthesizes all benchmarks in 0.43s on average.

2 Background and Overview

This section explains the formal definition of parsing expression grammars (PEGs) (§2.1) and provides a high-level overview of our synthesis approach with analysis-based pruning techniques and potential score-guided search (§2.2).

2.1 Parsing Expression Grammars (PEGs)

A *parsing expression grammar* [13] (PEG) is a formalism for describing recursive-descent parsers. This section defines the syntax and semantics of PEGs.

Syntax A PEG $P = (V_N, V_T, R, e_s)$ is a 4-tuple consisting of:

- a finite set of *nonterminal symbols* V_N
- a finite set of *terminal symbols* V_T
- a finite set of *parsing rules* $R : V_N \to E$
- a *start expression* $e_s \in E$ that is the entry point of the grammar

where $A \leftarrow e$ (or $R(A) = e$) denotes the parsing rule for a nonterminal symbol $A \in V_N$ defined in R. A *parsing expression* $e \in E$ is one of: an empty string ε, a terminal $\mathtt{a}$, the any-terminal $\mathtt{.}$, a nonterminal A, a sequence $e_1\ e_2$, a prioritized choice e_1/e_2, a not-predicate $\mathtt{!}e$, an and-predicate $\mathtt{\&}e$, a repetition $e\mathtt{*}$ or $e\mathtt{+}$, or an optional $e\mathtt{?}$:

$$E \ni e ::= \varepsilon \mid \mathtt{a} \mid \mathtt{.} \mid A \mid e\ e \mid e/e \mid e\mathtt{*} \mid e\mathtt{+} \mid e\mathtt{?} \mid \mathtt{!}e \mid \mathtt{\&}e$$

Semantics The big-step operational semantics of PEGs is defined as:

$$\frac{}{(\epsilon, x) \Downarrow x} \quad \frac{}{(\mathtt{a}, \mathtt{a}x) \Downarrow x} \quad \frac{\nexists y.\ x = \mathtt{a}y}{(\mathtt{a}, x) \Downarrow \mathbf{f}} \quad \frac{R(A) = e \quad (e, x) \Downarrow o}{(A, x) \Downarrow o}$$

$$\frac{(e_1, x) \Downarrow y \quad (e_2, y) \Downarrow o}{(e_1\ e_2, x) \Downarrow o} \quad \frac{(e_1, x) \Downarrow \mathbf{f}}{(e_1\ e_2, x) \Downarrow \mathbf{f}} \quad \frac{(e_1, x) \Downarrow y \quad (e_2, y) \Downarrow \mathbf{f}}{(e_1\ e_2, x) \Downarrow \mathbf{f}}$$

$$\frac{(e_1, x) \Downarrow o}{(e_1/e_2, x) \Downarrow o} \quad \frac{(e_1, x) \Downarrow \mathbf{f} \quad (e_2, x) \Downarrow o}{(e_1/e_2, x) \Downarrow o} \quad \frac{(e_1, x) \Downarrow \mathbf{f} \quad (e_2, x) \Downarrow \mathbf{f}}{(e_1/e_2, x) \Downarrow \mathbf{f}}$$

$$\frac{(e, x) \Downarrow y \quad (e\mathtt{*}, y) \Downarrow o}{(e\mathtt{*}, x) \Downarrow o} \quad \frac{(e, x) \Downarrow \mathbf{f}}{(e\mathtt{*}, x) \Downarrow x} \quad \frac{(e, x) \Downarrow \mathbf{f}}{(\mathtt{!}e, x) \Downarrow x} \quad \frac{(e, x) \Downarrow y}{(\mathtt{!}e, x) \Downarrow \mathbf{f}}$$

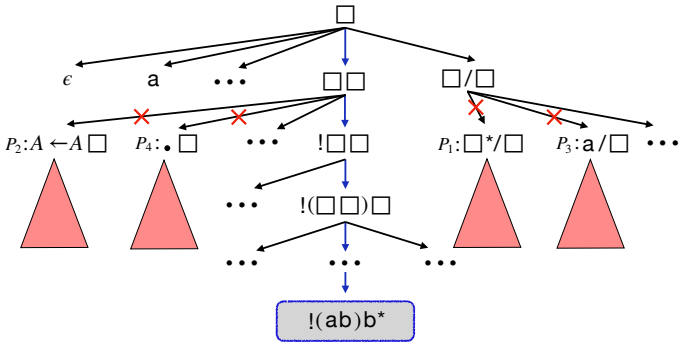
$$\mathcal{D}[\![\mathtt{.}]\!] = \mathtt{a}_1/\mathtt{a}_2/\ldots/\mathtt{a}_n \qquad \mathcal{D}[\![e\mathtt{+}]\!] = e\ e\mathtt{*} \qquad \mathcal{D}[\![e\mathtt{?}]\!] = e/\varepsilon \qquad \mathcal{D}[\![\mathtt{\&}e]\!] = \mathtt{!}\,\mathtt{!}e$$

where $(e, x) \Downarrow o$ denotes the evaluation relation and $\mathcal{D}[\![e]\!]$ denotes the desugaring rule. Given an input string $x = yz$, a parsing expression e either *succeeds* by consuming a prefix y and returning the remaining suffix z (i.e., $(e, yz) \Downarrow z$), or *fails* (i.e., $(e, x) \Downarrow \mathbf{f}$). Note that the original paper [13] defines the result of a successful parse as the prefix consumed by the parsing expression, but we define it as the remaining suffix for easier analysis during synthesis. However, they are semantically equivalent as the consumed prefix can be obtained by subtracting the remaining suffix from the original input string.

Example Consider the PEG $P = (\varnothing, \{\mathtt{a}, \mathtt{b}\}, \varnothing, \mathtt{!(ab)b*})$. It first uses a not-predicate $\mathtt{!(ab)}$ to check whether the input does not start with $\mathtt{ab}$, without consuming any input. Then, it consumes $\mathtt{b}$'s as much as possible using a repetition $\mathtt{b*}$. For example, the input $\mathtt{bbba}$ produces the output $\mathtt{a}$ by consuming three $\mathtt{b}$'s, whereas $\mathtt{aba}$ fails because it starts with $\mathtt{ab}$.

Input	Output
ϵ	$\rightarrow \epsilon$
a	$\rightarrow$ a
ab	$\rightarrow$ f
aba	$\rightarrow$ f
aaab	$\rightarrow$ aaab
b	$\rightarrow \epsilon$
bb	$\rightarrow \epsilon$
bab	$\rightarrow$ ab
bbba	$\rightarrow$ a

(a) Examples



(b) Top-down enumerative synthesis

Fig. 1: An example of a synthesis process for a PEG that consumes b's as much as possible if the input does not start with ab but fails otherwise.

2.2 Overview of Synthesis

Our goal is to automatically synthesize PEGs from input-output examples. This section provides an overview of our synthesis approach and pruning techniques with a synthesis task for the following parser as a running example:

> *A PEG consuming any number of* b*'s if input does not start with* ab *but fails otherwise.*

The input-output examples in Figure 1a specify the desired behavior and Figure 1b depicts the top-down enumerative synthesis process. A hole □ denotes an unspecified subexpression that can be replaced with any PEG expression. A PEG is *complete* if it has no holes, and *partial* otherwise. Synthesis begins with a single top-level hole and incrementally fills holes using predefined rules. For each partial PEG, we design two analyses to prune the search space effectively: 1) *grammar analysis* and 2) *example analysis*. Once a complete PEG is produced, it is verified against all given examples. Figure 2 shows examples of partial PEGs that can be pruned by our techniques for the previous synthesis example. Furthermore, we define the *potential score* of a partial PEG to prioritize when it has more potential to lead to a correct solution but still has a smaller size. We measure the potential by analyzing how many incorrect outputs can be filtered out by the partial PEG across all examples.

Pruning Based on Grammar Analysis Grammar analysis aims to prune partial PEGs that contain ill-formed or reducible constructs.

– *Reducible Expressions.* Some constructs are *reducible*, meaning they can be simplified into equivalent but simpler forms. For example, P_1 is also reducible as □* always succeeds, making the choice's right branch unreachable.

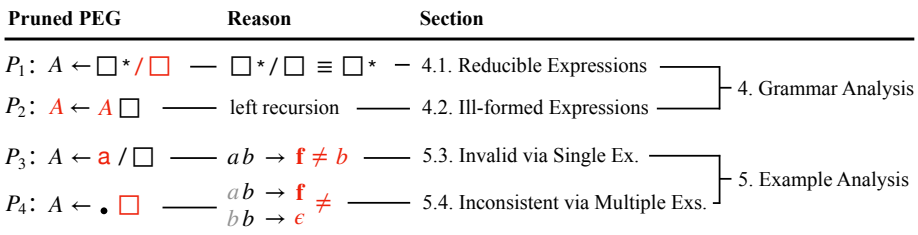


Fig. 2: Examples of partial PEGs pruned during the synthesis process in Figure 1.

– *Ill-formed Expressions.* A PEG is *ill-formed* if it contains constructs that can cause non-termination during parsing. For example, P_2 involves *left-recursive* rule $A \leftarrow A\ \square$, which can lead to infinite recursion.

Pruning Based on Example Analysis We use an *abstract machine* to analyze the partial PEGs for each input string in the specification until they reach a hole or terminate. We prune partial PEGs that produce the invalid output for a single example or inconsistent requirements on holes across multiple examples.

– *Invalid via Single Example.* A partial PEG is invalid if it contradicts an example regardless of hole instantiations. For instance, P_3 is invalid for the example $\mathtt{ab}$, since the expected output is $\mathbf{f}$, yet every instantiation of the PEG produces $\mathtt{b}$.
– *Inconsistent via Multiple Examples.* A partial PEG is pruned if it cannot satisfy multiple examples simultaneously. For example, P_4 is pruned as it cannot satisfy both examples $(\mathtt{ab}, \mathbf{f})$ and $(\mathtt{bb}, \epsilon)$. After consuming one symbol in both $\mathtt{ab}$ and $\mathtt{bb}$ using $\mathtt{.}$, $\square$ requires conflicting outcomes for the same remaining input $\mathtt{b}$: fail ($\mathbf{f}$) for the former and success with ϵ for the latter.

Potential Score-Guided Search To effectively guide the search process during synthesis, we measure the potential of partial PEGs based on their ability to eliminate incorrect outputs across all examples. Consider an input-output example $(\mathtt{bbba}, \mathtt{a})$, which has exactly one correct output ($\mathtt{a}$) and five possible incorrect outputs: four incorrect success cases and one failure case. Now, consider a partial PEG $P = (\mathtt{b}\ /\ \square_1)\ \square_2$. For the given input $\mathtt{bbba}$, P consumes the first $\mathtt{b}$ using the left branch of the prioritized choice and then halts at the second hole $\square_2$ with the remaining input $\mathtt{bba}$. Consequently, any full instantiation of P must either succeed after consuming at least one $\mathtt{b}$ or fail entirely. This guarantees that P cannot produce the incorrect output $\mathtt{bbba}$ (which requires consuming zero symbols). By filtering out one incorrect output for the example $(\mathtt{bbba}, \mathtt{a})$, P achieves a potential score of $1/5 = 0.2$. When evaluated across all examples in Figure 1a, P successfully eliminates 4 out of 29 total incorrect outputs, resulting in a potential score of $4/29 \approx 0.138$. In contrast, another partial PEG $P' = \mathtt{b\ !.\ /}\ \square$ yields a lower potential score of $2/29 \approx 0.069$. We define

the scoring function by combining this potential score with the size of PEGs, and SynPEG selects P over P' for further exploration according to this scoring function. Its detailed definition is explained later in §6.

In the rest of this paper, we formalize the synthesis process (§3) and pruning techniques based on grammar (§4) and example analysis (§5). We then explain the potential score (§6) and optimization (§7). Finally, we evaluate SynPEG (§8), discuss limitations (§9) and related work (§10), and conclude (§11).

3 Synthesis of PEGs Using Top-Down Enumeration

This section formalizes the synthesis process of PEGs using a top-down enumeration algorithm for a given set of input-output examples.

3.1 Normal Form of PEGs

Since the same PEG may have multiple syntactic representations, we define a *normal form* of PEGs and use it for enumeration to avoid redundant searches.

$$P = (V_N, V_T, R, \beta_{\mathsf{start}}) \qquad \begin{array}{ll} e ::= \beta \oplus \beta \mid \ominus\beta & \oplus ::= \cdot \mid / \\ \beta ::= \varepsilon \mid \mathtt{a} \mid \mathtt{.} \mid A \mid \square & \ominus ::= \mathtt{*} \mid \mathtt{?} \mid \mathtt{!} \mid \mathtt{\&} \end{array}$$

A parsing expression is either a binary ($\oplus$) or a unary ($\ominus$) operation applied to *base expressions* (β), i.e., expressions without nested operators. The start expression β_{start} is also defined as a base expression. We desugar all one-or-more repetitions (e+) for simplicity. For example, consider two different syntactic representations of the same PEG:

$$\mathtt{!(ab)b*} \qquad\qquad A; \quad A \leftarrow \mathtt{!}B\ \mathtt{b*}; \quad B \leftarrow \mathtt{ab}$$

Their normal form is as follows:

$$A; \quad A \leftarrow B\ D; \quad B \leftarrow \mathtt{!}C; \quad C \leftarrow \mathtt{a\ b}; \quad D \leftarrow \mathtt{b*}$$

The *size* of a PEG is defined as the number of operators, and it is equal to the number of non-terminal symbols in its normal form. We use $\|P\|$ to denote the size of PEGs. For example, the above PEG has size 4.

3.2 Synthesis Problem

The *synthesis problem* for PEGs is defined as $\mathcal{S} = (V_T, \mathcal{E})$, where V_T is a set of terminal symbols and $\mathcal{E} \subseteq \mathbb{S} \times (\mathbb{S} \uplus \{\mathbf{f}\})$ is a set of input-output examples. Here, $\mathbb{S} = V_T^*$ denotes the set of all strings over V_T. Each *example* (x, o) consists of an input string x and an output o, which is either a remaining string or a failure $\mathbf{f}$. A PEG P *satisfies* an example (x, o) if parsing x with P produces o (i.e., $(\beta_{\mathsf{start}}, x) \Downarrow o$). The synthesis problem $\mathcal{S}$ is to find a PEG P that satisfies all examples in $\mathcal{E}$. Such a PEG is called a *solution* to the synthesis problem $\mathcal{S}$:

$$P \in \mathsf{solution}(\mathcal{S}) \iff \forall (x, o) \in \mathcal{E}.\ (\beta_{\mathsf{start}}, x) \Downarrow o$$

Algorithm 1: Top-Down Enumeration Algorithm

Input: A synthesis problem $\mathcal{S} = (V_T, \mathcal{E})$
Output: A solution $P \in \mathsf{solution}(\mathcal{S})$

$W \leftarrow \{P_\iota\}$ where $P_\iota = (\varnothing, V_T, \varnothing, \square)$ is the most-general partial PEG
while $W \neq \varnothing$ **do**
  $P \leftarrow \mathrm{argmax}_{P' \in W}\, \mathsf{score}(P')$ `// Pick the most promising PEG`
  $W \leftarrow W \setminus \{P\}$ `// Remove from worklist`
  if $P \not\to$ **then**
    if $P \in \mathsf{solution}(\mathcal{S})$ **then**
      return P `// Check if it is a solution`
  else
    $W \leftarrow W \cup \{P' \mid P \to P' \wedge \neg \mathsf{prune}(P')\}$ `// Add non-pruned PEGs`

3.3 Top-Down Enumeration Algorithm

Algorithm 1 presents a *top-down enumeration* algorithm for a synthesis problem $\mathcal{S}$. It maintains a *worklist* W of PEGs to be processed and begins with the *most general* partial PEG, $P_\iota = (\varnothing, V_T, \varnothing, \square)$. For each iteration, it selects the most promising PEG P from W based on a *scoring function* $\mathsf{score}(P)$. If the selected PEG is complete ($P \not\to$), the algorithm evaluates it on all examples in $\mathcal{E}$ to check whether it is a solution. Otherwise, it applies the *hole-filling relation* $\to \subseteq \mathbb{P} \times \mathbb{P}$, which generates PEGs P' by filling a single hole in P. A hole is filled with either 1) a base expression, including a nonterminal already defined in V_N, or 2) a fresh nonterminal whose rule is a binary ($\square \oplus \square$) or unary ($\ominus\square$) operation consisting of holes. We provide the formal definition of the hole-filling rules in the companion report [19]. The resulting PEGs are then added to the worklist W only if they do not satisfy the *pruning* condition (prune). Although the basic algorithm uses only the size of PEGs as the scoring function (i.e., $\mathsf{score}(P) = -\|P\|$) and performs no pruning (i.e., $\mathsf{prune}(P) = \texttt{\#f}$), we later introduce *analysis-based pruning techniques* (§4 and §5) and potential-based scoring function (§6) to accelerate the synthesis process. In addition, we present several optimization techniques (§7) to further improve the performance of the proposed approach.

4 Grammar Analysis

We first introduce a *grammar analysis* to structurally evaluate and prune partial PEGs. Our grammar analysis detects two classes of undesirable partial PEGs: 1) *reducible* (§4.1) and 2) *ill-formed* (§4.2). We formalize these analyses based on the abstract interpretation framework [9, 10] to systematically analyze the structure of partial PEGs in a unified manner. For each analysis, we define:

1. an *abstract domain* $\widehat{D}$,
2. an *initial abstract element* $\widehat{d_\iota} \in \widehat{D}$,
3. a *join operator* $\sqcup$ between two abstract elements, and

4. the *abstract semantics* $\widehat{[\![A \leftarrow e]\!]} : \widehat{D} \to \widehat{D}$ of a rule $(A \leftarrow e) \in R$.

Then, we compute the least fixed point of the abstract transfer function $\widehat{F} : \widehat{D} \to \widehat{D}$, defined with the initial abstract element $\widehat{d_\iota}$ and the abstract semantics $\widehat{[\![A \leftarrow e]\!]}$ of the rules, as follows:

$$\widehat{[\![P]\!]} = \mathbf{lfp}\widehat{F} = \lim_{n\to\infty} \widehat{F}^n(\bot) \qquad \text{where} \qquad \widehat{F}(\widehat{d}) = \widehat{d_\iota} \sqcup \left(\bigsqcup_{(A\leftarrow e)\in R} \widehat{[\![A \leftarrow e]\!]}(\widehat{d}) \right)$$

4.1 Reducible Expressions

A PEG is considered *reducible* if it contains subexpressions that can be simplified without altering the overall semantics of the PEG. For example, as explained in §2, the following PEG is reducible because the left branch of the prioritized choice always succeeds, rendering the right branch strictly unreachable:

$$P_1 : A \leftarrow \square^* \,/\, \square \qquad (\because\ \square^* \,/\, \square \text{ is reducible to } \square^*)$$

Conversely, if the left branch of a choice is guaranteed to fail, the expression can be safely reduced to its right branch. One such case is a *self-sequentially reachable* nonterminal, which reaches itself solely through a chain of sequence operators for any input string. Lacking a successful base case, it inevitably fails on finite inputs. For example, in the following PEG, B is self-sequentially reachable and thus must fail:

$$A \leftarrow B \,/\, \square \qquad B \leftarrow \mathsf{a}\ B \qquad (\because\ B \,/\, \square \text{ is reducible to } \square)$$

To systematically detect such reducible expressions, we formalize a *must-pass/fail analysis* alongside a *sequential-reachability analysis*, whose detailed definitions are provided in the companion report [19].

Must-Pass/Fail Analysis This analysis determines whether a given expression is guaranteed to succeed (*must-pass*) or fail (*must-fail*) using the abstract semantics $\widehat{[\![-]\!]}_{\mathsf{pf}}$. Its abstract domain, $\widehat{D} = \mathcal{P}(V_N) \times \mathcal{P}(V_N)$, maintains a pair of sets tracking the must-pass and must-fail nonterminals, respectively. As established, if a nonterminal A is self-sequentially reachable (i.e., $A \in \widehat{[\![P]\!]}_{\mathsf{reach}}(A)$), it is marked as must-fail. Finally, $\widehat{[\![\beta]\!]}_{\mathsf{pf}}$ returns a pair of boolean values indicating whether the base expression β is must-pass and must-fail, respectively.

Sequential-Reachability Analysis We say a nonterminal A is *sequentially reachable* from another nonterminal B if A is reached from B through a chain of sequence operators. This analysis computes all the sequentially reachable nonterminals from each nonterminal using the abstract semantics $\widehat{[\![-]\!]}_{\mathsf{reach}}$ in the abstract domain $\widehat{D} = V_N \to \mathcal{P}(V_N)$.

Building upon these analyses, we formally define reducible expressions as follows, including not only purely syntactic patterns but also semantic reductions derived from the must-pass/fail analysis.

Definition 1 (Reducible Expressions). *An expression e in a partial PEG P is **reducible** if there exists a smaller expression e' such that $e \equiv e'$ under the following equivalence relations:*

- If β is *must-pass* (i.e., $\widehat{[\![\beta]\!]}_{\mathsf{pf}}(\widehat{[\![P]\!]}_{\mathsf{pf}}) = (\texttt{\#t}, \_)$), the following equivalences always hold:
$$\beta / \beta' \equiv \beta \qquad \beta\texttt{?} \equiv \beta \qquad \texttt{\&}\beta \equiv \varepsilon$$
- If β is *must-fail* (i.e., $\widehat{[\![\beta]\!]}_{\mathsf{pf}}(\widehat{[\![P]\!]}_{\mathsf{pf}}) = (\_, \texttt{\#t})$), the following equivalences always hold:
$$\beta / \beta' \equiv \beta' / \beta \equiv \beta' \qquad \beta\ \beta' \equiv \beta'\ \beta \equiv \beta \qquad \beta\texttt{*} \equiv \beta\texttt{?} \equiv \texttt{!}\beta \equiv \varepsilon \qquad \texttt{\&}\beta \equiv \beta$$
- Other rules for reducible expressions, including rules defined in [13]:
$$\beta / \beta \equiv \beta \qquad \beta_1 / \beta_2 / \beta_1 \equiv \beta_1 / \beta_2 \qquad \beta / \varepsilon \equiv \beta\texttt{?} \qquad \varepsilon\ \beta \equiv \beta\ \varepsilon \equiv \beta$$
$$\texttt{!\&}\beta \equiv \texttt{\&!}\beta \equiv \texttt{!}\beta \qquad \texttt{!!}\beta \equiv \texttt{\&\&}\beta \equiv \texttt{\&}\beta \qquad (\texttt{\&}\beta)\texttt{?} \equiv (\texttt{!}\beta)\texttt{?} \equiv \varepsilon$$
$$\texttt{!}(\beta_1\ \beta_2)\beta_1 \equiv \beta_1\ \texttt{!}\beta_2 \qquad (\beta_1\ \beta_3)/(\beta_2\ \beta_3) \equiv (\beta_1/\beta_2)\ \beta_3$$

Theorem 1 (Soundness of Reducible Expression Pruning). *For a PEG $P = (V_N, V_T, R, \beta_{\mathsf{start}})$, if $\mathsf{prune}_{\mathsf{red}}(P)$ holds, every instantiation of P is reducible:*

$\mathsf{prune}_{\mathsf{red}}(P) = P$ *contains any reducible expressions defined in Definition 1*

We provide the proof of this theorem in the companion report [19].

4.2 Ill-Formed Expressions

A PEG is *ill-formed* [13] if it contains constructs that can cause non-termination during parsing. We can detect such ill-formed PEGs by checking whether they contain any direct or indirect *left-recursive* rules. For example, as explained in §2, any instantiation of the following partial PEG is always ill-formed because of direct left recursion:

$$P_2 : A \leftarrow A\ \square \qquad (\because \text{ direct left recursion in } A)$$

Since the nonterminal A immediately invokes itself without consuming any input symbols, it repeatedly applies A to the same input string and thus diverges (fails to terminate) on certain inputs.

However, we must also consider *indirect* left recursion caused by *nullable* expressions, which succeed without consuming input symbols for at least one input.

$$A \leftarrow \texttt{a?}\ A\ \square \qquad (\because \text{ indirect left recursion in } A \text{ because } \texttt{a?} \text{ is nullable})$$

For input `bb`, `a?` succeeds without consuming any input, causing the nonterminal A to be invoked again on the same input `bb`. On the other hand, the following partial PEG contains an implicit form of left recursion: a repetition operator on a nullable expression.

$$A \leftarrow (\texttt{a?})^* \ \square \qquad (\because \text{ repetition applied to nullable expression } \texttt{a?})$$

Similarly, for input `bb`, the inner nullable expression `a?` succeeds without consumption, causing infinite repetitions of the inner expression on the same input `bb`. To detect such ill-formed expressions, we define 1) *left recursion analysis* and 2) *nullable expression analysis*, whose detailed definitions are provided in the companion report [19].

Left Recursion Analysis This analysis collects reachable nonterminals without consuming any symbols from each nonterminal in the abstract domain $\widehat{D} = V_N \to \mathcal{P}(V_N)$ using the abstract semantics $\widehat{[\![-]\!]}_{\mathsf{lr}}$. Direct and indirect left recursion are identified by checking whether a nonterminal A can reach itself without consuming any input symbols (i.e., $A \in \widehat{[\![P]\!]}_{\mathsf{lr}}(A)$). Note that it utilizes the nullable expression analysis to determine whether the first expression in a sequence is nullable.

Nullable Expression Analysis This analysis collects nullable nonterminals in the domain $\widehat{D} = \mathcal{P}(V_N)$ and identifies nullable expressions using the abstract semantics $\widehat{[\![-]\!]}_{\mathsf{null}}$. Using this analysis, we can detect implicit left recursion caused by repetition expressions whose internal expression is nullable.

Theorem 2 (Soundness of Ill-formed Expression Pruning). *For an irreducible PEG $P = (V_N, V_T, R, \beta_{\mathsf{start}})$, if $\mathsf{prune}_{\mathsf{ill}}(P)$ holds, then every instantiation of P does not terminate on at least one input:*

$$\mathsf{prune}_{\mathsf{ill}}(P) = \begin{cases} (\exists A \in V_N.\ A \in \widehat{[\![P]\!]}_{\mathsf{lr}}(A)) & \vee \quad \textit{(explicit left recursion)} \\ (\exists (A \leftarrow \beta^*) \in R.\ \widehat{[\![\beta]\!]}_{\mathsf{null}}(\widehat{[\![P]\!]}_{\mathsf{null}}) = \texttt{\#t}) & \textit{(implicit left recursion)} \end{cases}$$

We provide the proof of this theorem in the companion report [19].

5 Example Analysis

The second component of our synthesis algorithm is *example analysis*. Unlike the static grammar analysis (§4), this dynamic analysis evaluates partial PEGs against each input string, produces possible outputs, and prunes candidates that contradict the expected outputs. To execute partial PEGs, we design an *abstract machine* that simulates the parsing process and captures the intermediate parsing state exactly when a hole is encountered. We leverage the execution results from this abstract machine to prune partial PEGs that are either: 1) *invalid*

for a single example, or 2) *inconsistent* across multiple examples. The fundamental insight behind both pruning strategies is to explicitly capture and utilize *continuations* within the intermediate parsing states produced by the abstract machine. This section first introduces the abstract machine for PEGs (§5.1), formalizes the computation of possible outputs and state equivalence (§5.2), and then presents the two example-based pruning techniques (§5.3 and §5.4).

5.1 Abstract Machine for PEGs

Since partial PEGs may contain hole expressions with undetermined behaviors, the parsing process must be suspended upon encountering a hole. The example analysis utilizes the intermediate parsing state at this exact moment of suspension, which captures the *continuation* of the parse, to infer the set of possible outputs and check the equivalence of states across examples. However, the big-step operational semantics defined in §2.1 is unsuitable for representing such continuations, as it relates input strings directly to final results without exposing intermediate states. To address this limitation, we define an *abstract machine* that explicitly represents continuations and models the parsing process as a sequence of state transitions.

States A *state* $\sigma \in \Sigma$ represents the current configuration, including the input string, continuation, consumed/seen indices, and pass/fail flag.

$$
\begin{array}{lll}
\text{States} & \sigma \in \Sigma ::= & \mathbb{S} \times \mathbb{K} \times \mathbb{I}_{\mathsf{consume}} \times \mathbb{I}_{\mathsf{seen}} \times \Phi \\
\text{Continuations} & \kappa \in \mathbb{K} \;=\; & \mathbb{C} \times \mathbb{I}_{\mathsf{input}} \times \mathbb{I}^{?}_{\mathsf{mark}} \times \mathbb{K}^{?} \\
\text{Control Points} & c \in \mathbb{C} \;=\; & \blacktriangleright\mathbf{S} \mid \mathbf{S}\blacktriangleleft \mid \blacktriangleright A_{\mathsf{L}} \mid A_{\mathsf{L}}\blacktriangleleft \mid \blacktriangleright A_{\mathsf{R}} \mid A_{\mathsf{R}}\blacktriangleleft \\
\text{Indices} & i \in \mathbb{I} \;=\; & \mathbb{N} \\
\text{Flags} & \phi \in \Phi ::= & \mathbf{p} \mid \mathbf{f}
\end{array}
$$

A *continuation* $\kappa \in \mathbb{K}$ consists of the current control point, input index, optional mark index, and optional further continuation. A *control point* $c \in \mathbb{C}$ specifies the machine's current execution point within the PEG. It is either a pre-evaluation ($\blacktriangleright-$) or post-evaluation ($-\blacktriangleleft$) point of: 1) the start expression ($\mathbf{S}$), or 2) the left (A_{L}) or right (A_{R}) subexpression within the right-hand side of a nonterminal A's rule. If the rule for A is unary, only A_{L} is used. An *index* $i \in \mathbb{I}$ represents a position in the input string, serving one of four distinct roles:

- A *consumed index* $i^{\mathsf{c}} \in \mathbb{I}_{\mathsf{consume}}$ indicates the position up to which the input string has been successfully consumed.
- A *seen index* $i^{\mathsf{s}} \in \mathbb{I}_{\mathsf{seen}}$ indicates the rightmost position accessed in the input string during parsing.
- An *input index* $i^{\mathsf{i}} \in \mathbb{I}_{\mathsf{input}}$ indicates the position in the input string where the evaluation of the current nonterminal began.
- A *mark index* $i^{\mathsf{m}} \in \mathbb{I}_{\mathsf{mark}}$ indicates a position to backtrack to when a choice expression fails or a predicate expression succeeds/fails.

$$A; \quad A \leftarrow B \,/\, \mathtt{a}; \quad B \leftarrow \mathtt{a}\ \mathtt{c}$$

(a) Normalized form of the PEG `ac/a`.

$$\begin{aligned}
&(\blacktriangleright \mathbf{S}, \overset{\mathbf{S}}{\underset{\mathtt{c}\ \mathtt{s}}{|\,|\,|\mathtt{abc}}}, \mathbf{p}) \rightsquigarrow (\blacktriangleright A_{\mathsf{L}}, \overset{\mathbf{S}\,A_{\mathsf{L}}}{\underset{\mathtt{c}\ \mathtt{s}}{|\,|\,|\,|\mathtt{abc}}}, \mathbf{p}) \rightsquigarrow (\blacktriangleright B_{\mathsf{L}}, \overset{\mathbf{S}\,A_{\mathsf{L}}\ B_{\mathsf{L}}}{\underset{\mathtt{m}\qquad\ \ \mathtt{c}\ \mathtt{s}}{|\,|\,|\,|\,|\,|\mathtt{abc}}}, \mathbf{p}) \\
&\rightsquigarrow (B_{\mathsf{L}}\blacktriangleleft, \overset{\mathbf{S}\,A_{\mathsf{L}}\ B_{\mathsf{L}}}{\underset{\mathtt{m}\qquad\ \ \mathtt{c}\ \mathtt{s}}{|\,|\,|\,|\,|\mathtt{a}\,|\,|\mathtt{bc}}}, \mathbf{p}) \rightsquigarrow (\blacktriangleright B_{\mathsf{R}}, \overset{\mathbf{S}\,A_{\mathsf{L}}\ B_{\mathsf{R}}}{\underset{\mathtt{m}\qquad\ \ \mathtt{c}\ \mathtt{s}}{|\,|\,|\,|\,|\,|\mathtt{a}\,|\,|\mathtt{bc}}}, \mathbf{p}) \rightsquigarrow (B_{\mathsf{R}}\blacktriangleleft, \overset{\mathbf{S}\,A_{\mathsf{L}}\ B_{\mathsf{R}}}{\underset{\mathtt{m}\qquad\ \ \mathtt{c}\ \mathtt{s}}{|\,|\,|\,|\,|\,|\mathtt{ab}\,|\,|\mathtt{c}}}, \mathbf{f}) \\
&\rightsquigarrow (A_{\mathsf{L}}\blacktriangleleft, \overset{\mathbf{S}\,A_{\mathsf{L}}}{\underset{\mathtt{m}\quad\ \mathtt{c}\ \mathtt{s}}{|\,|\,|\,|\mathtt{ab}\,|\,|\mathtt{c}}}, \mathbf{f}) \rightsquigarrow (\blacktriangleright A_{\mathsf{R}}, \overset{\mathbf{S}\,A_{\mathsf{R}}}{\underset{\mathtt{c}\quad\ \mathtt{s}}{|\,|\,|\mathtt{ab}\,|\,\mathtt{c}}}, \mathbf{p}) \rightsquigarrow (A_{\mathsf{R}}\blacktriangleleft, \overset{\mathbf{S}\,A_{\mathsf{R}}}{\underset{\mathtt{c}\quad\ \mathtt{s}}{|\,|\,|\mathtt{a}\,|\,\mathtt{b}\,|\,\mathtt{c}}}, \mathbf{p}) \\
&\rightsquigarrow (\mathbf{S}\blacktriangleleft, \overset{\mathbf{S}}{\underset{\mathtt{c}\quad\ \mathtt{s}}{|\,|\mathtt{a}\,|\,\mathtt{b}\,|\,\mathtt{c}}}, \mathbf{p})
\end{aligned}$$

(b) Parsing process of the PEG `ac/a` with input string `abc`.

Fig. 3: An example of the parsing process in the abstract machine.

The indices i^{c}, i^{i}, and i^{m} range from 0 to $|x|$, whereas the seen index i^{s} can extend up to $|x|+1$ to indicate whether the end-of-string (EOS) has been inspected. For example, when parsing `aa` with `a*`, the final seen index is 3, reflecting that the EOS has been inspected. Finally, a *flag* $\phi \in \Phi$ indicates whether the current state is a *pass* ($\mathbf{p}$) or *fail* ($\mathbf{f}$) state.

Example Consider the PEG in Figure 3a and its parsing process on the input string `abc`. It first attempts the left branch of the choice in A, namely $B \leftarrow \mathtt{ac}$, but fails when `c` does not match the second input symbol `b`. It then backtracks and tries the right branch `a`, which succeeds. After B fails, the PEG reaches the sixth state in Figure 3b, which is shown below:

$$\langle \mathtt{abc}, (B_{\mathsf{R}}\blacktriangleleft, 0, \bot, (\blacktriangleright A_{\mathsf{L}}, 0, 0, (\blacktriangleright \mathbf{S}, 0, \bot, \bot))), 2, 2, \mathbf{f}\rangle \quad \equiv \quad (B_{\mathsf{R}}\blacktriangleleft, \overset{\mathbf{S}\,A_{\mathsf{L}}\ B_{\mathsf{R}}}{\underset{\mathtt{m}\qquad\ \ \mathtt{c}\ \mathtt{s}}{|\,|\,|\,|\,|\mathtt{ab}\,|\,|\mathtt{c}}}, \mathbf{f})$$

The right side illustrates a more readable notation for representing these states. Labels `m`, `c`, and `s` placed below the bar represent the mark, consumed, and seen indices, respectively. Labels $\mathbf{S}$, A_{L}, and B_{R} placed above the bar denote the control points and their corresponding input indices within the continuation. We omit the pre/post markers in this visual notation because continuations inherently capture pre-evaluation ($\blacktriangleright -$) points. The mark index 0 associated with A_{L} indicates that the machine backtracks to position 0 upon the failure of B. Thus, in the eighth state, the consumed index reverts to 0 and the pass/fail flag is flipped to $\mathbf{p}$:

$$\langle \mathtt{abc}, (\blacktriangleright A_{\mathsf{R}}, 0, \bot, (\blacktriangleright \mathbf{S}, 0, \bot, \bot)), 0, 2, \mathbf{p}\rangle \quad \equiv \quad (\blacktriangleright A_{\mathsf{R}}, \overset{\mathbf{S}\,A_{\mathsf{R}}}{\underset{\mathtt{c}\quad\ \mathtt{s}}{|\,|\,|\mathtt{ab}\,|\,\mathtt{c}}}, \mathbf{p})$$

Finally, the machine succeeds by parsing the right branch a of the choice expression in A, but the seen index remains 2:

$$\langle \mathsf{abc}, (\mathbf{S}\blacktriangleleft, 0, \bot, \bot), 1, 2, \mathbf{p} \rangle \quad \equiv \quad (\mathbf{S}\blacktriangleleft, \overset{\mathbf{S}}{|\,\underset{\mathsf{c}}{\mathsf{a}}\,|\,\underset{\mathsf{s}}{\mathsf{b}}\,|\,\mathsf{c}}, \mathbf{p})$$

State Transition Relation To model the parsing process as a sequence of state transitions, we define a *state transition relation* $\rightsquigarrow$ that maps one state to the next.

Definition 2 (State Transition Relation). *The **abstract machine** for a partial PEG is governed by a **state transition relation** $\rightsquigarrow \subseteq \Sigma \times \Sigma$:*

$$\sigma \rightsquigarrow \sigma' \iff \sigma = \langle \_, (c, \_, \_, \_), \_, \_, \_ \rangle \wedge \sigma' = [\![c]\!]_c(\sigma)$$

where the transition logic is defined by the semantics of control points $[\![-]\!]_c$ and base expressions $[\![-]\!]_\beta$.

Both the control-point semantics $[\![-]\!]_c$ and the base-expression semantics $[\![-]\!]_\beta$ are defined as compositions of ten *helper functions* that manipulate the components of a state at a finer granularity. For each control point, $[\![-]\!]_c$ performs the actions corresponding to the operator of the nonterminal's rule, whereas $[\![-]\!]_\beta$ performs the actions for a base expression. We provide the full definitions of $[\![-]\!]_c$ and $[\![-]\!]_\beta$ for all control points and base expressions in the companion report [19].

Each helper function manipulates specific components of a state as follows. The read operation reads a symbol from the input string at the current consumed index, updating both the consumed and seen indices accordingly. The operations entry, next, and exit describe how the continuation advances when moving to the entry, next, or exit points of the start expression ($\mathbf{S}$) or the right-hand side of a nonterminal A's rule. To support backtracking over the consumed input, we introduce mark, remove, and revert, which manage the mark index. The entry to and exit from nonterminals are explicitly handled by the call and return operations. Finally, the flip operation updates the pass/fail flag of a state to reflect the outcome of the current computation.

For example, $[\![-]\!]_c$ composes these helper functions to realize the operator of each rule. When the left branch of a choice expression fails (i.e., $c = A_\mathsf{L}\blacktriangleleft$ with $R(A) = \_/\_$ and $\phi = \mathbf{f}$), the machine backtracks to the mark index, flips the fail flag, and proceeds to the right branch of the choice:

$$[\![A_\mathsf{L}\blacktriangleleft]\!]_c(\sigma) = \sigma.\mathsf{revert}.\mathsf{flip}.\mathsf{next}$$

Similarly, $[\![-]\!]_\beta$ composes them to evaluate each base expression. For a nonterminal A, it pushes a new continuation for A onto the continuation stack like a function call ($[\![A]\!]_\beta(\sigma) = \sigma.\mathsf{call}(A)$), whereas for the others it reads the next symbol and determines whether the symbol matches.

Theorem 3 (Correctness of the Abstract Machine). *For a partial PEG $P = (V_N, V_T, R, \beta_{\mathsf{start}})$ and an input string $x \in V_T^*$, the parsing result produced*

by the abstract machine is equivalent to the result derived from the big-step operational semantics defined in §2.1:

$$\langle x, (\blacktriangleright \boldsymbol{S}, 0, \bot, \bot), 0, 0, \mathbf{p}\rangle \rightsquigarrow^* \langle x, (\boldsymbol{S}\blacktriangleleft, 0, \bot, \bot), i^{\mathsf{c}}, i^{\mathsf{s}}, \mathbf{p}\rangle \iff (\beta_{\mathsf{start}}, x) \Downarrow x[i^{\mathsf{c}}:]$$
$$\langle x, (\blacktriangleright \boldsymbol{S}, 0, \bot, \bot), 0, 0, \mathbf{p}\rangle \rightsquigarrow^* \langle x, (\boldsymbol{S}\blacktriangleleft, 0, \bot, \bot), i^{\mathsf{c}}, i^{\mathsf{s}}, \mathbf{f}\rangle \iff (\beta_{\mathsf{start}}, x) \Downarrow \mathbf{f}$$

where $\rightsquigarrow^*$ is the reflexive and transitive closure of $\rightsquigarrow$, and $x[i^{\mathsf{c}}:]$ is the suffix of x starting from the index i^{c}.

We provide the proof of this theorem in the companion report [19].

5.2 Possible Outputs and State Equivalence

The abstract machine halts when no further transitions are possible from the current state. This occurs either when the PEG encounters a hole expression or when it reaches the post-evaluation point of the start expression ($\mathbf{S}\blacktriangleleft$). We define the auxiliary function $\mathsf{halt}(P, x)$ to denote this halting state of the abstract machine when parsing an input string x with a PEG P:

$$\mathsf{halt}(P, x) = \sigma \quad \text{such that} \quad \langle x, (\blacktriangleright \mathbf{S}, 0, \bot, \bot), 0, 0, \mathbf{p}\rangle \rightsquigarrow^* \sigma \not\rightsquigarrow$$

Building on this definition, we first formalize the computation of *possible outputs* from a halted state, and then establish a formal *equivalence* relation between states based on their continuations.

Possible Outputs If the PEG does not encounter a hole, it produces a unique output. Otherwise, the PEG halts at a hole expression and can potentially produce multiple outputs depending on how the hole is instantiated. However, because the remaining parsing process is strictly governed by the continuation of the halting state, some outputs can never be produced regardless of how the hole is instantiated.

We represent these possible outputs using abstract outputs, which consist of an abstract index range and an abstract fail flag:

Abstract Outputs	$\widehat{o} \in \widehat{\mathbb{O}} = \widehat{\mathbb{I}} \times \widehat{\mathbb{F}}$
Abstract Index Ranges	$\widehat{i} \in \widehat{\mathbb{I}} = \{\bot\} \uplus \mathbb{I} \times (\mathbb{I} \uplus \{+\infty\})$
Abstract Fails	$\widehat{\mathbf{f}} \in \widehat{\mathbb{F}} = \{\bot, \top\}$

Operations on abstract outputs, such as partial order ($\sqsubseteq$), join ($\sqcup$), and meet ($\sqcap$), are defined in the standard way. The lifting operator $\widehat{i}\uparrow$ extends the upper bound of the index range to $+\infty$ when $\widehat{i} \neq \bot$ (e.g., $[0,1]\uparrow = [0, +\infty]$), while preserving $\bot$. The size operator $|\widehat{o}|_x$ denotes the number of concrete outputs represented by the abstract output $\widehat{o}$ for an input string x (e.g., $|([0, +\infty], \top)|_{\mathsf{abc}} = 5$).

When the machine halts at a hole, we compute the possible outputs using the *propagate* function $\mathsf{propagate}(P, \kappa, \widehat{o})$, which propagates an abstract output $\widehat{o}$ backward through the continuation κ. It starts from the most general abstract

output $([i^{\mathsf{c}}, +\infty], \top)$ at the hole, which represents that the hole may be instantiated to consume any number of symbols from the current consumed index i^{c} or to fail. Then, traversing the continuation from the innermost control point outward, it refines the abstract output at each control point according to the operator of the corresponding rule (e.g., prioritized choice, repetition, or predicates), excluding the outputs that can never be produced. We provide the full definition of the propagate function in the companion report [19].

Definition 3 (Possible Outputs). *For a partial PEG P and a string $x \in \mathbb{S}$, its **possible outputs** $\mathsf{output}(P, x) \in \widehat{\mathbb{O}}$ is defined as:*

$$\mathsf{output}(P,x) = \begin{cases} ([i^{\mathsf{c}}, i^{\mathsf{c}}], \bot) & \text{if } c = \textbf{\textit{S}}\blacktriangleleft \wedge \phi = \mathbf{p} \\ (\bot, \top) & \text{if } c = \textbf{\textit{S}}\blacktriangleleft \wedge \phi = \mathbf{f} \\ \mathsf{propagate}(P, \kappa, ([i^{\mathsf{c}}, +\infty], \top)) & \text{otherwise} \end{cases}$$

where $\mathsf{halt}(P,x) = (\_, \kappa = (c, \_, \_, \_), i^{\mathsf{c}}, \_, \phi)$.

For example, $\mathsf{output}(\mathsf{a}\square, \mathsf{aab}) = ([1, +\infty], \top)$ indicates that any instantiation either fails or succeeds while consuming at least 1 symbol from aab. Similarly, $\mathsf{output}(\mathsf{a}(\mathsf{b}\ \&\square)?, \mathsf{abab}) = ([1,2], \bot)$ guarantees that any instantiation succeeds while consuming either 1 or 2 symbols from abab.

Equivalence of States During the parsing process, different input strings can reach states that guarantee the same final outcome, regardless of how a hole is instantiated. Specifically, the parsing result is entirely determined by the reachable portion of the input string and the remaining parsing operations captured by the continuation. The reachable string is the suffix of the input starting from the leftmost mark index in the continuation, or from the current consumed index if no mark exists.

Definition 4 (Reachable String). *For a state $\sigma = (x, \kappa, i^{\mathsf{c}}, \_, \_) \in \Sigma$ where the continuation $\kappa = (\_, \_, i^{\mathsf{m}}, \kappa')$, the **reachable string** $\mathsf{reach}(\sigma)$ is defined as:*

$$\mathsf{reach}(\sigma) = \begin{cases} x[i^{\mathsf{c}}:] & \text{if } \mathsf{lm}(\kappa) = \bot \\ x[\mathsf{lm}(\kappa):] & \text{otherwise} \end{cases} \quad \text{where} \quad \mathsf{lm}(\kappa) = \begin{cases} \bot & \text{if } \kappa = \bot \\ i^{\mathsf{m}} & \text{if } \mathsf{lm}(\kappa') = \bot \\ \mathsf{lm}(\kappa') & \text{if } \mathsf{lm}(\kappa') \neq \bot \end{cases}$$

For example, consider the PEG on the left, which contains a hole in the right-hand side of C. When parsing the input string abcd, the PEG halts at the hole, resulting in the state shown on the right:

$$A;\quad A \leftarrow \mathsf{a}\ B;\quad B \leftarrow C\ /\ \square;\quad C \leftarrow \mathsf{b}\ \square \qquad\qquad (\blacktriangleright C_{\mathsf{R}}, \overset{\mathbf{S}\,A_{\mathsf{R}}\ B_{\mathsf{L}}\ C_{\mathsf{R}}}{\mid\ \mid \mathsf{a} \mid\ \underset{\mathsf{m}}{\mid}\ \mid \underset{\mathsf{c\,s}}{\mathsf{b}} \mid \mid \mathsf{cd}}, \mathbf{f})$$

Here, the reachable string is bcd (highlighted in gray). This corresponds to the suffix of the input string starting from the mark index 1, which was established by the choice expression in B after consuming a in A.

We establish the equivalence of two states based on their continuations and reachable strings. If two states possess identical control stacks and equal reachable strings, they will undergo the exact same parsing process regardless of how the hole is instantiated. Since PEG parsing is inherently deterministic, this implies that they will yield identical parsing results. Consequently, this equivalence can be leveraged to prune candidate PEGs that exhibit inconsistent parsing results across different examples that halt at equivalent states.

Definition 5 (Equivalence of States). *Two continuations* $\kappa_1 = (c_1, \_, \_, \kappa_1')$ *and* $\kappa_2 = (c_2, \_, \_, \kappa_2')$ *are* ***weakly equivalent*** *(*$\approx_c$*) if their control stacks are identical:*

$$\kappa_1 \approx_c \kappa_2 \iff (\kappa_1' = \kappa_2' = \bot) \vee (c_1 = c_2 \wedge \kappa_1' \approx_c \kappa_2')$$

Two states $\sigma_1 = (\_, \kappa_1, \_, \_, \_)$ *and* $\sigma_2 = (\_, \kappa_2, \_, \_, \_)$ *are* ***equivalent*** *(*$\equiv$*) if their continuations are weakly equivalent and their reachable strings are equal:*

$$\sigma_1 \equiv \sigma_2 \iff \kappa_1 \approx_c \kappa_2 \wedge \mathsf{reach}(\sigma_1) = \mathsf{reach}(\sigma_2)$$

5.3 Invalid via Single Example

As previously described, the abstract machine determines the *possible outputs* of a partial PEG. If these possible outputs contradict an example's expected result, the PEG is *invalid*, as no hole instantiation can satisfy the example.

For instance, as explained in §2.2, the partial PEG P_3 always produces the output b for the input ab without reaching the hole. This contradicts the expected result $\mathbf{f}$:

$$P_3 : A \leftarrow \mathsf{a} \;/\; \square \qquad (\mathsf{ab}, \mathbf{f}) \in \mathcal{E} \qquad \mathsf{output}(P_3, \mathsf{ab}) = ([1,1], \bot)$$

A more interesting case arises when a partial PEG halts at a hole expression. For example, consider the partial PEG P_3' halting at the hole for the input aaab:

$$P_3' : A \leftarrow \mathsf{a}\ \square \qquad (\mathsf{aaab}, \mathsf{aaab}) \in \mathcal{E} \qquad \mathsf{output}(P_3', \mathsf{aaab}) = ([1, +\infty], \top)$$

As indicated by the abstract output $([1, +\infty], \top)$, the set of possible concrete outputs of P_3' for this input is $\{\mathsf{aab}, \mathsf{ab}, \mathsf{b}, \epsilon, \mathbf{f}\}$. This means the PEG will either consume at least 1 symbol or fail entirely, depending on how the hole is instantiated. However, this set of possible outputs does not contain the expected result aaab, rendering P_3' invalid with respect to this example.

Theorem 4 (Soundness of Validity-Based Pruning). *For a partial PEG* P *and a set of examples* $\mathcal{E}$*, if* $\mathsf{prune}_{\mathsf{ex}}(P)$ *holds, then every instantiation of* P *violates at least one example:*

$$\mathsf{prune}_{\mathsf{ex}}(P) = \exists (x, o) \in \mathcal{E}.\ o \notin \mathsf{output}(P, x)$$

5.4 Inconsistent via Multiple Examples

If two states are *equivalent* within the same PEG, they will always produce the same parsing result for any instantiation of the hole. However, two examples may reach equivalent states with different expected outputs. This contradicts the inherently deterministic property of PEGs and indicates that no single instantiation of the hole can satisfy both examples simultaneously.

For example, consider the partial PEG P_4 from §2.2. For inputs ab and bb, it halts at the hole with the following states:

$$P_4 : A \leftarrow \mathbf{.}\,\square \quad \left\{ \begin{array}{l} (\mathsf{ab}, \mathbf{f}) \\ (\mathsf{bb}, \epsilon) \end{array} \right\} \in \mathcal{E} \quad \begin{cases} (\blacktriangleright \mathbf{S}, \overset{\mathbf{S}}{\underset{\mathsf{c}\,\mathsf{s}}{|\,|\,|\mathsf{ab}}}, \mathbf{p}) \rightsquigarrow^* (\blacktriangleright A_{\mathsf{R}}, \overset{\mathbf{S}\quad A_{\mathsf{R}}}{\underset{\mathsf{c}\,\mathsf{s}}{|\mathsf{a}\,|\,|\mathsf{b}}}, \mathbf{p}) \not\rightsquigarrow \\ (\blacktriangleright \mathbf{S}, \overset{\mathbf{S}}{\underset{\mathsf{c}\,\mathsf{s}}{|\,|\,|\mathsf{bb}}}, \mathbf{p}) \rightsquigarrow^* (\blacktriangleright A_{\mathsf{R}}, \overset{\mathbf{S}\quad A_{\mathsf{R}}}{\underset{\mathsf{c}\,\mathsf{s}}{|\mathsf{b}\,|\,|\mathsf{b}}}, \mathbf{p}) \not\rightsquigarrow \end{cases}$$

After ignoring the unreachable prefixes of the input strings (i.e., a in the first example and the first b in the second), both states share the same reachable string b and identical remaining controls (i.e., $\blacktriangleright\mathbf{S}$ and $\blacktriangleright A_{\mathsf{R}}$). Thus, they are equivalent and must produce the same parsing result for any instantiation of the hole. However, their expected outputs differ ($\mathbf{f} \neq \epsilon$), indicating that no instantiation of P_4 can satisfy both examples simultaneously. We can therefore safely prune P_4 based on this *inconsistency* across examples.

Theorem 5 (Soundness of Consistency-Based Pruning). *For a partial PEG P and a set of examples $\mathcal{E}$, if $\mathsf{prune}_{\mathsf{inc}}(P)$ holds, then every instantiation of P produces inconsistent parsing results for at least one pair of examples in $\mathcal{E}$:*

$$\mathsf{prune}_{\mathsf{inc}}(P) = \exists (x, o), (x', o') \in \mathcal{E}.\ \mathsf{halt}(P, x) \equiv \mathsf{halt}(P, x') \wedge o \neq o'$$

6 Potential Score-Guided Search

To efficiently explore the search space, we define a *potential score* to prioritize partial PEGs that are more likely to lead to a solution.

6.1 Potential Score

A naive enumeration strategy relies solely on the size of the PEG as a scoring function, which fails to account for how well a partial PEG satisfies the given specification. To address this, the *potential score* measures the capability of a partial PEG to filter out incorrect outputs across all examples. For a synthesis problem $\mathcal{S} = (V_T, \mathcal{E})$, the potential score of a partial PEG P is defined as follows:

$$\mathsf{score}_p(P) = 1 - \frac{\sum_{(x,\_)\in\mathcal{E}} (|\mathsf{output}(P, x)|_x - 1)}{\sum_{(x,\_)\in\mathcal{E}} (|x| + 1)}$$

For each input string $(x, \_) \in \mathcal{E}$, there are $|x| + 2$ candidate outputs: $|x| + 1$ prefixes of x and $\mathbf{f}$. Since exactly one of these is the correct expected output,

there are $|x| + 1$ incorrect candidate outputs. Similarly, if P is not pruned by the example analysis on x, exactly one output in the abstract set $\mathsf{output}(P, x)$ corresponds to the expected output. Thus, P fails to filter out $|\mathsf{output}(P, x)|_x - 1$ incorrect outputs, where $|\mathsf{output}(P, x)|_x$ counts the number of possible concrete outputs represented by the abstract output $\mathsf{output}(P, x)$ for the input string x. The potential score $\mathsf{score}_p(P)$ ranges from 0 to 1, where a higher score indicates that P is more effective at filtering out incorrect outputs. The most general PEG $\square$ has a potential score of 0, while a PEG that perfectly isolates the expected outputs for all examples yields a potential score of 1.

6.2 Potential Score-Guided Search

To prevent overfitting, our search primarily relies on the size of the PEG as the baseline scoring function. However, we augment this score by incorporating the potential score to prioritize promising partial PEGs:

$$\mathsf{score}(P) = \alpha \cdot \mathsf{score}_p(P) - \|P\|$$

In the early stages of the search, when partial PEGs are too general, the potential score evaluates to 0, making the search behave like pure size-based enumeration. As the search progresses and partial PEGs begin to filter out incorrect outputs, the potential score increases, allowing the algorithm to prioritize PEGs that are more likely to lead to a valid solution. The hyperparameter α controls the influence of the potential score on the overall score. This formulation allows a larger PEG to outrank a smaller one (by a size difference of up to α) if the larger PEG exhibits a correspondingly higher potential score. For example, if $\alpha = 4$, a PEG with a potential score of 0.5 can be prioritized over another PEG that is up to 2 units smaller in size, since $0.5 \cdot 4 = 2$.

7 Optimization

This section describes three optimization techniques that reduce redundant computations during grammar and example analysis.

1. **abstract parsing** (§7.1): It allows the abstract machine to execute multiple examples simultaneously by abstracting states based on seen prefixes.
2. **Incremental Analysis** (§7.2): Instead of re-analyzing an entire partial PEG after filling a hole, it reuses the analysis results of the previous PEG before filling the hole and analyzes only the newly instantiated subexpression.
3. **Abstract Packrat Parsing** (§7.3): To maintain linear parsing time during abstract parsing, we memoize and reuse results for identical pairs of nonterminals and input indices during the abstract parsing process.

7.1 Abstract Parsing

The example analysis in §5 requires executing the abstract machine for every example in the specification. However, this approach may lead to significant computational overhead when dealing with a large number of examples. To mitigate this issue, we propose *abstract parsing*, which enables the abstract machine to execute multiple examples simultaneously.

Key Observation The key observation is that unseen suffixes of input strings do not affect the execution of the abstract machine until the machine needs to look beyond the currently seen prefix. Thus, if multiple examples halt after seeing the same prefix, their executions are identical up to that point, which allows us to merge their states into a single abstract execution. For instance, in the partial PEG $P_3 : A \leftarrow \mathsf{a}\ /\ \square$, four input strings a, ab, aba, and aaab from Figure 4a all halt after seeing and consuming the same prefix a. Our goal is to merge the executions of these four examples into a single abstract execution:

$$(\blacktriangleright\mathbf{S}, \overset{\mathbf{S}}{\underset{\mathbf{c}\ \mathbf{s}}{|\,|\,|\mathsf{a}}}, \mathbf{p}) \quad \rightsquigarrow^* (\mathbf{S}\blacktriangleleft, \overset{\mathbf{S}}{\underset{\mathbf{c}\ \mathbf{s}}{|\mathsf{a}|\,|}}, \mathbf{p}) \qquad (\blacktriangleright\mathbf{S}, \overset{\mathbf{S}}{\underset{\mathbf{c}\ \mathbf{s}}{|\,|\,|\mathsf{ab}}}, \mathbf{p}) \quad \rightsquigarrow^* (\mathbf{S}\blacktriangleleft, \overset{\mathbf{S}}{\underset{\mathbf{c}\ \mathbf{s}}{|\mathsf{a}|\,|\mathsf{b}}}, \mathbf{p})$$

$$(\blacktriangleright\mathbf{S}, \overset{\mathbf{S}}{\underset{\mathbf{c}\ \mathbf{s}}{|\,|\,|\mathsf{aba}}}, \mathbf{p}) \rightsquigarrow^* (\mathbf{S}\blacktriangleleft, \overset{\mathbf{S}}{\underset{\mathbf{c}\ \mathbf{s}}{|\mathsf{a}|\,|\mathsf{ba}}}, \mathbf{p}) \qquad (\blacktriangleright\mathbf{S}, \overset{\mathbf{S}}{\underset{\mathbf{c}\ \mathbf{s}}{|\,|\,|\mathsf{aaab}}}, \mathbf{p}) \rightsquigarrow^* (\mathbf{S}\blacktriangleleft, \overset{\mathbf{S}}{\underset{\mathbf{c}\ \mathbf{s}}{|\mathsf{a}|\,|\mathsf{aab}}}, \mathbf{p})$$

Example Trie To efficiently group examples by their prefixes and access their expected outputs, we utilize a trie data structure, referred to as the *example trie*. Each path from the root to a node represents a prefix of the input strings in the given examples, and a node is annotated with the expected output (consumed index or failure) if that path corresponds to a complete input string. Figure 4b shows the example trie constructed from the input-output examples in Figure 4a. For instance, the sub-trie rooted at the node for the prefix a represents all inputs sharing that prefix, namely $\{\mathsf{a}, \mathsf{ab}, \mathsf{aba}, \mathsf{aaab}\}$. Their expected outputs are annotated inside the nodes in black: 0 for a and aaab, and $\mathbf{f}$ for ab and aba. The red annotations above the nodes represent the abstract outputs that merge the expected outputs of all inputs under the corresponding sub-trie.

Abstract States We define *abstract states* $\widehat{\Sigma}$ to represent the merged executions of multiple examples as follows:

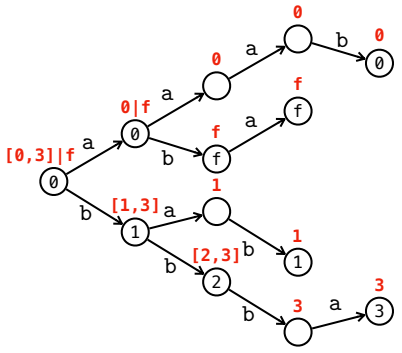
$$\begin{aligned} &\text{Abstract States } \hat{\sigma} \in \widehat{\Sigma} ::= \mathbb{K} \times \mathbb{I}_{\mathsf{consume}} \times \widehat{\mathbb{S}} \times \Phi \\ &\text{Seen Prefixes} \quad \widehat{x} \in \widehat{\mathbb{S}} \ = \ \mathbb{S} \times \{\blacksquare, \$\} \end{aligned}$$

Abstract states utilize *seen prefixes* $\widehat{\mathbb{S}}$ instead of the pair of concrete input string $\mathbb{S}$ and the seen index $\mathbb{I}_{\mathsf{seen}}$ used in concrete states:

- $x\blacksquare$ – The set of all input strings sharing the common prefix x (represented by a sub-trie in the example trie)
- $x\$$ – It represents the single, complete input string x (represented by a single node in the example trie)

In		Out	i^c	ϕ
ϵ	$\to$	ϵ	0	**p**
a	$\to$	a	0	**p**
ab	$\to$	**f**	–	**f**
aba	$\to$	**f**	–	**f**
aaab	$\to$	aaab	0	**p**
b	$\to$	ϵ	1	**p**
bb	$\to$	ϵ	2	**p**
bab	$\to$	ab	1	**p**
bbba	$\to$	a	3	**p**

(a) A set of input-output examples.



(b) An example trie for the given examples.

Fig. 4: A set of input-output examples and an example trie as their abstraction.

Abstract State Transitions We design *abstract state transitions* to parse PEGs across all examples simultaneously. The process 1) starts from an initial state $\hat{\sigma}_0 = (\blacktriangleright\mathbf{S}, \overset{\mathbf{S}}{\underset{\mathtt{c\,s}}{\mid\mid\mid\blacksquare}}, \mathbf{p})$ representing all inputs, 2) progresses linearly until it must look beyond the current seen prefix, and 3) splits according to the example trie. For instance, the analysis of $P_3 : A \leftarrow \mathtt{a}\ /\ \square$ advances from $\hat{\sigma}_0$ to just before reading a, then splits into three abstract states based on Figure 4b:

$$\hat{\sigma}_0 \rightsquigarrow^* (\blacktriangleright A_\mathsf{L}, \overset{\mathbf{S}A_\mathsf{L}}{\underset{\mathtt{m\,c\,s}}{\mid\mid\mid\mid\blacksquare}}, \mathbf{p}) \rightsquigarrow \begin{cases} \nearrow & (A_\mathsf{L}\blacktriangleleft, \overset{\mathbf{S}A_\mathsf{L}}{\underset{\mathtt{m\ c\,s}}{\mid\mid\mid\mathtt{a}\mid\mid\blacksquare}}, \mathbf{p}) \rightsquigarrow^* (\mathbf{S}\blacktriangleleft, \overset{\mathbf{S}}{\underset{\mathtt{c\,s}}{\mid\mathtt{a}\mid\mid\blacksquare}}, \mathbf{p}) & \not\rightsquigarrow \\ \to & (A_\mathsf{L}\blacktriangleleft, \overset{\mathbf{S}A_\mathsf{L}}{\underset{\mathtt{m\ c\,s}}{\mid\mid\mid\mathtt{b}\mid\mid\blacksquare}}, \mathbf{f}) \rightsquigarrow (\blacktriangleright A_\mathsf{R}, \overset{\mathbf{S}A_\mathsf{R}}{\underset{\mathtt{c\,s}}{\mid\mid\mid\mathtt{b}\blacksquare}}, \mathbf{p}) & \not\rightsquigarrow \\ \searrow & (A_\mathsf{L}\blacktriangleleft, \overset{\mathbf{S}A_\mathsf{L}}{\underset{\mathtt{m\,c\ s}}{\mid\mid\mid\mid\mathtt{\$}\mid}}, \mathbf{f}) \rightsquigarrow (\blacktriangleright A_\mathsf{R}, \overset{\mathbf{S}A_\mathsf{R}}{\underset{\mathtt{c\ s}}{\mid\mid\mid\mathtt{\$}\mid}}, \mathbf{p}) & \not\rightsquigarrow \end{cases}$$

This shows how multiple concrete transitions merge into abstract ones. For instance, the top sequence merges four examples (a, ab, aba, and aaab) that share the prefix a■ and produces the same transitions.

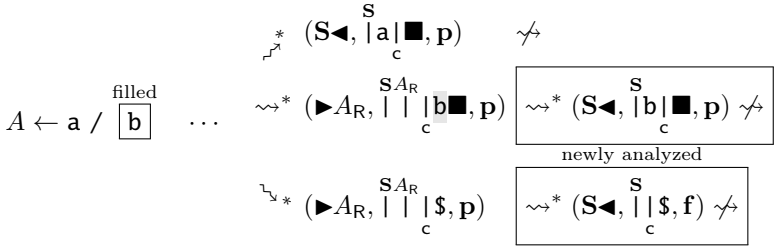
Optimization of Validity-Based Pruning The abstract parsing accelerates the validity-based pruning (§5.3) by checking whether the abstract expected output for a seen prefix is covered by the possible outputs derived from the halting abstract states. For example, considering P_3, the abstract expected output for the prefix a is $([0,0], \top)$. However, the possible output derived from its halting abstract state is $([1,1], \bot)$, which fails to cover the expected output: $([0,0], \top) \not\sqsubseteq ([1,1], \bot)$.

Optimization of Consistency-Based Pruning To apply abstract parsing to consistency checking (§5.4), we compare the sub-tries corresponding to equivalent halting abstract states. Although a naive comparison takes linear time with

respect to the size of the sub-tries, we optimize this process by memoizing previously compared pairs of seen prefixes. This memoization enables amortized constant-time equality checks by simply comparing the roots and consulting the memo table for their children.

7.2 Incremental Analysis

During top-down enumeration, reanalyzing a newly instantiated partial PEG from scratch is redundant. Instead, we leverage our abstract interpretation framework to incrementally reuse prior analysis results. Specifically, the abstract states that previously halted at a hole directly serve as the initial states for evaluating the newly filled subexpression. For instance, after filling the hole in P_4 with b, we reuse the existing transitions and only compute the newly boxed transitions:



7.3 Abstract Packrat Parsing

The packrat parsing [12] is a parsing technique that memoizes the parsing results for each input position and nonterminal to avoid redundant computation. It is powerful in PEGs and supports linear parsing time for any PEGs. To leverage this technique in the abstract interpretation of partial PEGs, we add a memoization to the abstract machine. For each abstract state, we store the memoized results for the nonterminal and the input index $i^{\mathsf{i}} \in \mathbb{I}_{\mathsf{input}}$ pairs. When performing the abstract interpretation, we check the memoized results for the current nonterminal and input index to avoid redundant computation. If the abstract machine requires to look beyond the current seen prefix, the memoized results are also copied to the new abstract states. It allows us to perform packrat parsing optimization in multiple examples at once in the abstract state level.

8 Evaluation

We implemented SynPEG in Scala to answer the following research questions:

RQ1: Can SynPEG synthesize PEGs for our benchmarks?
RQ2: How much do the pruning techniques reduce search space and time?
RQ3: How effective is the potential score-guided search?
RQ4: How do the optimization techniques improve synthesis time?

We ran all experiments single-threaded on a Ryzen 9 7950X (4.5 GHz, 128 GB RAM) with a 300-second timeout (TO), using geometric means to average ratios like speedup and space reduction.

Table 1: Each row shows the description, the terminal set (V_T), the predefined rules, and the number of training examples ($\#e_{\mathsf{train}}$).

No.	Description	V_T	Predefined Rules	$\#e_{\mathsf{train}}$
1	Parse as many ‘a’s as possible or ‘abb’.	a, b	-	15
2	Parse the leading ‘a’s only if there is an odd number of ‘a’s.	a, b	-	15
3	Parse a sequence of ‘a’s followed by an equal number of ‘b’s.	a, b	-	15
4	Parse a sequence of ‘a’s followed by an equal number of ‘b’s and the same number of ‘c’s.	a, b, c	AB ← ‘a’ AB? ‘b’; BC ← ‘b’ BC? ‘c’;	25
5	Parse a sequence of ‘a’s only if its length is a power of two and greater than one.	a, b	-	15
6	Parse until just before the first unmatched closing parenthesis if it exists.	(,)	-	19
7	Parse only sequences of one, two, or three ‘=’.	[a-z], =	-	20
8	Parse only the year of a date written as year/month/day.	[0-9], /	-	14
9	Parse a floating-point number with an optional integer part, or an integer.	[0-9], .	-	20
10	Parse a single italic span in markdown.	[a-z], _, -, s	-	15
11	Parse a block comment.	[a-z], /, *	-	12
12	Parse a variable NAME only when it is not being updated by an assignment.	[a-z], [0-9], =, -, +, ;	NAME ← ALPHA (ALPHA / NUM)*; OP ← ‘-’ / ‘+’ / ‘==’; BASE ← NAME / NUM NUM*; EXPR ← BASE (OP EXPR)* / NAME ‘=’ EXPR ‘;’?;	15
13	Parse an identifier that is not a standalone keyword.	[a-z], [0-9], -, _	NAMECONT ← ‘_’ / ALPHA / NUM; NAME ← (‘_’ / ALPHA) NAMECONT*; KEYWORDS ← ‘if’ / ‘else’ / ‘while’;	30
14	Parse ‘async’ only if it is used as a keyword in a JavaScript program.	[a-z], (,), =, >, s	WS ← ‘ ’*; NAME ← ALPHA ALPHA*; ASYNC ← ‘async ’ WS; FUNC ← ‘function ’ WS; PARAMS ← NAME / ‘()’; / ‘(’ NAME (‘,’ NAME)* ‘)’;	25
15	Parse the keyword ‘static’ only if it is used for static generators in a JavaScript program.	[a-z], (,), *, {, }, s	WS ← ‘ ’*; NAME ← ALPHA ALPHA*; STATIC ← ‘static ’ WS; PARAMS ← ‘()’ / ‘(’ NAME (‘,’ NAME)* ‘)’;	25

Table 2: Comparison of the oracle PEG ($\mathbf{P}_{\mathsf{oracle}}$) with the PEG synthesized by our tool ($\mathbf{P}_{\mathsf{ours}}$), shown side by side with their sizes ($\|P\|$).

No.	$\mathbf{P}_{\mathsf{oracle}}$	$\|\mathbf{P}_{\mathsf{oracle}}\|$	$\mathbf{P}_{\mathsf{ours}}$	$\|\mathbf{P}_{\mathsf{ours}}\|$
1	`'abb' / 'a'*;`	4	`('a' ('bb' / 'a'*))?;`	5
2	`!(('aa')* !'a') 'a'*;`	7	`A; A ← 'a' (!'a' / 'a' A);`	4
3	`A; A ← 'a' A? 'b';`	3	`A; A ← 'a' ('b' / A 'b');`	3
4	`&(AB TC) TA* BC;`	5	`&(AB 'c') A;` `A ← 'a' (BC / A);`	5
5	`A !'a'; A ← 'a' A 'a' / 'aa';`	6	`A !'a'; A ← 'a' (A 'a' / 'a');`	5
6	`A &')'; A ← '(' A ')' A / ;`	6	`A; A ← &')' / B B; B ← . A;`	4
7	`('===' / '==' / '=') !'=';`	7	`'=' A A !'='; A ← '='?;`	5
8	`YEAR &('/' NUM NUM '/' NUM NUM);` `YEAR ← NUM NUM NUM NUM;`	10	`A A &(B B);` `A ← NUM NUM; B ← '/' A;`	6
9	`NUM* '.' NUM NUM* / NUM NUM*;`	8	`A NUM*; A ← '.' NUM / NUM A?;`	6
10	`'_' !'_' . (!'_' .)* '_';`	8	`'_' A; A ← !'_' . ('_' / A);`	5
11	`'/*' (!('*/') .)* '*/';`	8	`'/*' A; A ← '*/' / . A;`	5
12	`NAME !('=' !'=');`	4	`NAME !('=' !'=');`	4
13	`!(KEYWORDS !NAMECONT) NAME;`	4	`(KEYWORDS / &NAME) A;` `A ← NAME / NAMECONT A?;`	6
14	`ASYNC &(FUNC / PARAMS '=>');`	5	`ASYNC &(FUNC / PARAMS '=>');`	5
15	`STATIC &('*' NAME PARAMS '{');`	5	`STATIC &('*' NAME PARAMS '{');`	5

8.1 Benchmarks

Since there are no existing benchmarks for example-based PEG synthesis, we construct a new benchmark suite in Table 1 to evaluate SynPEG on a variety of parsing tasks, covering both formal specifications and practical parsing scenarios.

Benchmarks 1–5 target formal parsing, 6–10 capture simple practical scenarios, and 11–15 parse real-world language fragments. They are designed to expose key PEG characteristics such as prioritized choice, greedy parsing, and lookahead predicates. Predefined rules reflect practical settings where existing grammar components (e.g., character classes like `ALPHA ← [a-z]`) are reused. They are treated as single nonterminals and are *not* counted in $\|P\|$.

Training examples are written manually, reflecting our target scenario in which the user provides them. For each task, we write a small set of examples that covers as many edge cases as possible, so more complex tasks use more examples (12–30 per task, $\#e_{\mathsf{train}}$). They balance successful cases with full or partial consumption and failing cases to accurately capture the intended semantics and prevent over-generalization.

In contrast, synthesized PEGs are validated against 10,000 automatically generated test examples ($\#e_{\mathsf{test}}$). Each test set reuses the training examples and adds strings randomly sampled from the oracle PEG, maintaining an 8:2 ratio of successful to failing cases so that accuracy is not dominated by a single outcome category. The resulting test sets achieve full structural coverage of the oracle PEG, including various repetition counts for iterative or recursive rules.

Table 3: Performance comparison of SynPEG against LLMs ($\mathsf{LLM}_{\mathsf{Ex}}$ and $\mathsf{LLM}_{\mathsf{Ex+Desc}}$) with accuracy (**acc**), execution time (**t**), and generated PEG size ($\|P\|$). The ± values indicate the standard deviation across multiple runs.

	$\mathsf{LLM}_{\mathsf{Ex}}$			$\mathsf{LLM}_{\mathsf{Ex+Desc}}$			SynPEG		
No.	acc (%)	t (s)	$\|P\|$	acc (%)	t (s)	$\|P\|$	acc (%)	t (s)	$\|P\|$
1	39.7±32.0	1.41±0.46	**4.6±0.9**	60.4±30.6	1.36±0.19	4.8±0.4	**100.0**	**0.44**	5.0
2	44.3±25.4	1.82±0.89	8.8±1.3	**100.0±0.0**	1.33±0.25	5.0±0.0	**100.0**	**0.02**	**4.0**
3	84.0±35.7	1.47±0.39	4.4±0.9	**100.0±0.0**	1.10±0.22	4.0±0.0	**100.0**	**0.01**	**3.0**
4	77.5±0.0	1.39±0.43	**3.0±0.0**	90.5±7.3	1.16±0.22	4.8±1.1	**100.0**	**0.62**	5.0
5	97.7±0.0	1.45±0.33	**4.0±0.0**	20.1±0.0	1.84±0.55	12.8±8.2	**100.0**	**1.08**	5.0
6	20.1±0.0	1.45±0.19	7.8±1.3	11.8±10.8	1.52±0.31	9.4±2.3	**100.0**	**0.08**	**4.0**
7	97.5±3.4	1.35±0.42	7.2±3.6	**100.0±0.0**	1.40±0.47	11.0±0.0	**100.0**	**0.03**	**5.0**
8	58.6±38.7	1.20±0.32	**3.4±0.5**	16.2±0.0	1.46±0.82	4.0±0.0	**100.0**	**0.13**	6.0
9	**100.0±0.0**	1.40±0.12	7.0±0.0	**100.0±0.0**	1.31±0.07	10.4±2.6	**100.0**	**0.75**	**6.0**
10	57.2±34.7	1.80±0.34	15.8±4.1	49.8±39.7	2.39±1.65	16.2±5.8	**100.0**	**0.58**	**5.0**
11	**100.0±0.0**	1.84±0.89	9.0±0.0	98.0±1.8	1.34±0.17	13.8±4.4	**100.0**	**0.40**	**5.0**
12	98.8±0.0	1.25±0.21	4.0±0.0	99.4±0.4	1.11±0.27	**2.4±0.9**	**100.0**	**0.06**	4.0
13	99.6±0.6	1.10±0.15	**2.4±0.9**	99.7±0.6	1.32±0.31	4.0±0.0	**100.0**	**0.57**	6.0
14	63.1±0.0	1.42±0.13	9.0±3.9	63.1±0.0	1.23±0.20	6.0±0.0	**100.0**	**0.83**	**5.0**
15	99.9±0.0	1.07±0.19	**4.0±0.0**	99.9±0.0	1.20±0.24	4.4±0.9	**100.0**	**0.90**	5.0
Avg.	75.9	1.43	6.3	73.9	1.41	7.5	**100.0**	**0.43**	**4.9**
Sol.		3.40/15			5.40/15			**15.00/15**	

Table 2 shows the manually written oracle PEG ($\mathbf{P}_{\mathsf{oracle}}$) and its size ($\|\mathbf{P}_{\mathsf{oracle}}\|$) for each benchmark. These oracles are used to generate test examples, never during synthesis. All benchmarks and experimental results are publicly available [1].

8.2 RQ1: Synthesis Performance

The only prior work on PEG synthesis [38] cannot serve as a baseline because it solves a fundamentally different problem. It learns non-circular PEGs from membership labels, where a string is accepted whenever parsing does not fail, regardless of how much input is consumed. Such labels cannot express our benchmarks, whose examples specify the consumed prefix of each accepted input. Conversely, its benchmarks derived from noisy biological datasets provide only membership labels without consumption information, so they cannot be translated into our input-output examples.

Therefore, we instead compare SynPEG with large language models (LLMs) as a natural off-the-shelf baseline. This comparison shows that producing a correct PEG from examples is non-trivial task without a dedicated synthesis procedure. Table 3 compares SynPEG against GPT-5.4 (no-reasoning) under two settings: using only examples ($\mathsf{LLM}_{\mathsf{Ex}}$) and using both examples and descriptions ($\mathsf{LLM}_{\mathsf{Ex+Desc}}$). While we use a no-reasoning model as the natural off-the-shelf baseline, we also provide a comparison against a reasoning LLM in the companion report [19], where it still fails to solve the hardest benchmarks and remains slow and cost-ineffective. Prompts include PEG explanations, the task, and examples (details in [1]). Recognizing the difficulty of pure example-based

synthesis for LLMs, the $\mathsf{LLM}_{\mathsf{Ex+Desc}}$ setting additionally incorporates the specification's description. Since LLMs are stochastic unlike the deterministic SynPEG, we report average LLM results over five independent runs.

Since PEGs represent functions from input strings to parsing outputs rather than binary classifiers, we define *accuracy* (**acc**) as the percentage of test examples where the synthesized PEG's output exactly matches the oracle PEG's output. Since the test sets contain both successful and failing cases (§8.1), achieving 100% accuracy ensures that the synthesized grammar neither overfits nor overgeneralizes. A benchmark is considered *solved* only upon reaching 100% accuracy on all test examples.

As shown in Table 3, SynPEG successfully synthesizes correct PEGs for all benchmarks, with an average size of $\|\mathbf{P}_{\mathsf{ours}}\|$ and synthesis time of 0.43s. Table 2 provides the synthesized PEGs and their respective sizes ($\|\mathbf{P}_{\mathsf{ours}}\|$). In contrast, LLMs perform significantly worse, solving only 3.4 ($\mathsf{LLM}_{\mathsf{Ex}}$) and 5.4 ($\mathsf{LLM}_{\mathsf{Ex+Desc}}$) benchmarks with average runtimes of 1.43s and 1.41s. Despite improvements in the $\mathsf{LLM}_{\mathsf{Ex+Desc}}$ setting, LLMs struggle with PEG semantics; for instance, in Benchmark 1, the LLM fails to capture prioritized choice and greedy parsing, yielding incorrect grammars like `'a'*/'abb'`. Furthermore, LLMs often generate insufficiently restrictive constraints for predicates. In Benchmark 12, the LLM produces `NAME !'='`, failing to reject cases where `NAME` precedes the equality operator `'=='`. Since `'='` successfully consumes the first character of `'=='`, precise handling requires a stricter predicate like `!('=' !'=')` to isolate assignments from equality checks. Ultimately, these results underscore that precise PEG synthesis is a highly non-trivial task, prone to subtle semantic errors even for humans, which SynPEG effectively automates.

8.3 RQ2: Effectiveness of Pruning Techniques

Table 4 shows that grammar-based (G) and example-based (E) pruning significantly accelerate synthesis by filtering invalid candidates. For benchmarks completed within the time limit, the grammar analysis (G) eliminates an average of 66.85% candidates compared to the baseline (B). The example analysis (E) further removes 89.61% candidates, totaling a 96.06% reduction. These techniques yield average speedups of 2.52× and 4.95×, respectively, achieving a combined speedup of 11.37×. Furthermore, pruning enables solving five additional benchmarks that otherwise timeout (TO) under the baseline, proving its critical role in handling practical time constraints.

8.4 RQ3: Effectiveness of Potential Score-Guided Search

Table 4 shows that potential score-guided search (S) successfully prioritizes promising candidates. This search strategy is especially effective for sequential structures even with lookahead predicates, preventing timeouts (TO) by drastically cutting exploration time. On average, it reduces explored candidates by

Table 4: The baseline (B) is progressively enhanced with grammar-based pruning (G), example-based pruning (E), and potential score-guided search (S). **red.** (%) and **spe.** (×) represent the candidate PEG reduction and the synthesis time speedup, relative to B.

	# Candidate PEGs					Synthesis Time (s)				
No.	B (M)	G (M)	E (M)	S (K)	**red.** (%)	B	G	E	S	**spe.** (×)
1	1.07	0.28	0.02	6.48	99.39	7.47	2.58	0.79	0.44	17.10
2	0.12	0.04	0.01	0.54	99.53	0.71	0.32	0.11	0.02	29.58
3	0.05	0.02	0.00	0.12	99.76	0.26	0.15	0.04	0.01	52.80
4	>30.53	>25.12	7.94	24.84	>99.92	TO	TO	151.98	0.62	>485.44
5	>39.28	9.68	0.97	57.79	>99.85	TO	87.24	17.28	1.08	>277.27
6	5.37	1.03	0.07	7.75	99.86	39.29	8.28	0.88	0.08	485.02
7	>17.48	3.82	0.35	1.15	>99.99	TO	70.68	9.15	0.03	>11111.22
8	>29.46	>24.33	>3.41	7.18	>99.98	TO	TO	TO	0.13	>2272.73
9	>28.45	>24.09	2.89	36.72	>99.87	TO	TO	60.31	0.75	>397.88
10	>17.61	13.97	1.64	28.56	>99.84	TO	269.64	125.72	0.58	>518.14
11	>18.46	>16.10	6.89	20.43	>99.89	TO	TO	169.18	0.40	>755.67
12	3.13	1.49	0.19	0.95	99.97	92.84	48.05	8.93	0.06	1657.89
13	>9.61	6.60	0.33	11.45	>99.88	TO	216.03	18.72	0.57	>528.17
14	>10.84	>10.00	>0.96	25.07	>99.77	TO	TO	TO	0.83	>361.89
15	>12.28	>11.41	>1.18	29.20	>99.76	TO	TO	TO	0.90	>333.71
Avg.	>14.91	>9.86	>1.79	17.21	>99.82	209.37	166.87	97.54	0.43	>389.68

94.20% over the previous step (E), yielding a 42.70× speedup. Ultimately, integrating all techniques eliminates 99.82% candidates and achieves a 389.68× speedup over the baseline, ensuring successful synthesis for all benchmarks.

The hyperparameter α in $\mathsf{score}(P) = \alpha \cdot \mathsf{score}_p(P) - \|P\|$ controls the balance between smaller and more promising candidates (§6). Conceptually, an α near zero approximates Breadth-First Search, using the score merely to break ties, whereas an excessively large α shifts the strategy toward Depth-First Search. Moreover, α bounds how far the potential score can reorder the search, which keeps the search systematic rather than greedy. Since the potential score ranges over $[0, 1]$, the potential term raises the priority of a candidate by at most α. Consequently, when k is the size of the smallest PEG satisfying all examples, no candidate of size $k + \alpha$ or larger is expanded before all candidates of size k are explored, so the search never gets trapped in local optima and requires no backtracking. This bound also prevents ad-hoc refinements that enlarge a PEG merely to cover individual examples from dominating the search, as such candidates can be preferred only within the same bounded window.

Empirically, $\alpha = 4$ yields the optimal average synthesis time of 0.43s. While increasing α initially enhances performance by effectively prioritizing promising candidates, it begins to degrade at $\alpha \geq 5$. Specifically, $\alpha \geq 9$ causes timeouts in certain benchmarks, drastically increasing the overall average time. This degradation occurs because an overly large α over-prioritizes small partial PEGs in-

Table 5: Effect of optimization under configuration S. Analysis time is divided into **Grammar Analysis** (§4), **Example Analysis** (§5), and **Total Analysis**. Each reports elapsed time without optimization (wo/), with optimization (w/), and the resulting speedup (**spe.**).

	Grammar Analysis			Example Analysis			Total Analysis		
No.	wo/ (s)	w/ (s)	**spe.** (×)	wo/ (s)	w/ (s)	**spe.** (×)	wo/ (s)	w/ (s)	**spe.** (×)
1	0.18	0.12	1.45	0.09	0.05	1.89	0.27	0.17	1.57
2	0.02	0.01	3.00	0.01	0.00	3.33	0.03	0.01	3.13
3	0.00	0.00	1.00	0.00	0.00	4.00	0.00	0.00	3.00
4	0.76	0.14	5.61	0.35	0.06	5.48	1.10	0.20	5.57
5	0.65	0.20	3.29	0.67	0.22	3.10	1.32	0.41	3.19
6	0.10	0.02	4.95	0.07	0.02	4.29	0.18	0.04	4.66
7	0.07	0.00	33.00	0.01	0.00	2.00	0.07	0.01	12.33
8	0.24	0.02	14.35	0.22	0.02	11.63	0.46	0.04	12.92
9	0.40	0.13	3.07	0.26	0.12	2.13	0.66	0.25	2.62
10	1.63	0.09	18.94	0.23	0.06	3.95	1.86	0.14	12.90
11	1.15	0.06	18.30	0.11	0.04	2.95	1.27	0.10	12.52
12	0.14	0.01	20.14	0.02	0.00	5.25	0.16	0.01	14.73
13	1.18	0.05	23.16	0.70	0.20	3.46	1.88	0.25	7.45
14	2.51	0.08	30.56	0.39	0.07	5.98	2.89	0.15	19.69
15	2.40	0.11	22.46	0.41	0.04	10.84	2.82	0.15	19.41
Avg.	0.76	0.07	8.33	0.24	0.06	4.04	1.00	0.13	6.85

stantiated with terminals, unfairly penalizing larger, more promising candidates that temporarily introduce structural overhead (e.g., choice expressions).

α	0	1	2	3	**4**	5	6	7	8	9	10
Avg. time (s)	97.54	48.73	6.25	0.85	**0.43**	0.68	1.07	2.41	11.73	41.74	41.85

8.5 RQ4: Effectiveness of Optimization

Table 5 demonstrates that our optimizations significantly reduce analysis time without altering the analysis results. Specifically, they yield average speedups of 8.33× and 4.04× for grammar and example analyses, respectively, achieving an overall 6.85× speedup in total analysis time. For grammar analysis, incremental evaluation becomes effective as the size of explored candidates (including predefined rules) grows, as seen in Benchmarks 10–15. For example analysis, speedups are driven by: 1) the number of examples (aiding abstract parsing), 2) example length (aiding incremental analysis), and 3) redundant parsing of the same nonterminals (mitigated by abstract packrat parsing). Consequently, Benchmarks 11 and 13 show modest improvements because these factors are small, whereas Benchmark 15 exhibits substantial speedups because all three are significant.

9 Discussion

Our synthesis tool, SynPEG, explores all possible PEG candidates up to the target size to guarantee soundness. While our pruning, score-based search, and op-

timizations significantly enhance efficiency, the exponential growth of the search space remains a fundamental limitation for synthesizing large grammars from scratch via top-down enumerative search alone.

More Precise Analysis Currently, SynPEG lacks formalized abstraction-based rules to automatically eliminate certain classes of invalid or reducible partial PEGs. Such reducible expressions could be systematically identified through both grammar-based analysis and example-based reasoning. For instance, given the rules $A \leftarrow \mathsf{a} \ / \ B$ and $B \leftarrow \mathsf{a}$, the two syntactically distinct alternatives in A are semantically equivalent across all inputs. Similarly, the expression `&a a` behaves identically to `a` and is therefore trivially reducible. Formalizing these equivalences would further restrict the search space.

Practical Implications Although our benchmarks demonstrate that PEG synthesis is feasible for practical tasks, synthesizing entire large grammars via top-down enumerative search alone remains intractable. A highly promising direction is applying our techniques to *incremental synthesis*. When extending an existing parser with new syntax, the original PEG can serve as a partial PEG with holes, which SynPEG can fill with newly synthesized constructs. Benchmarks having predefined rules demonstrate the viability of this incremental approach.

10 Related Work

Parsing Expression Grammars PEGs [13] have been extensively studied across various dimensions. Prior work explored their expressiveness relative to regular expressions (REs) [32] and restricted context-free grammars (CFGs) [29], including LL(k), right-linear, and LL-regular grammars. Although PEGs can capture certain context-sensitive patterns, whether they strictly subsume CFGs remains an open problem [27]. Subsequent extensions have addressed left recursion [28, 33, 35], backtracking [31], linear-time parsing [6, 12, 37], and domain-specific enhancements [21, 30, 36, 39]. Regarding abstract machines, scaffolding automata [27] and parsing machines [16] have been proposed to characterize their computational power and implement pattern matching, respectively. However, unlike our approach, they do not explicitly capture continuations, which are crucial for the precise and sound analysis of partial PEGs during synthesis.

Parser Synthesis and Grammar Inference To the best of our knowledge, no prior work has studied the synthesis of general PEGs from input-output examples. The closest work [38] infers PEGs from examples, but it differs from ours in two fundamental ways. First, it targets only *non-circular* PEGs representing finite languages, a severely limited subset strictly less expressive than regular languages, which cannot capture repetitive patterns such as $ab*$. Second, it treats PEGs as string recognizers for membership classification, where

a string is accepted whenever parsing does not fail, regardless of the remaining input, and provides no guarantees on the synthesized PEGs. In contrast, we treat PEGs as partial functions from inputs to parsing outputs and guarantee synthesis of a general PEG satisfying all given examples. For general parser generation, prior work [24, 25] has focused on interactive approaches that require users to provide feedback instead of fully automated synthesis from fixed examples. Conversely, generative grammar inference [2–5, 14, 17, 20, 22, 26] and automated repair [8, 11, 23, 34] have been studied extensively. From Gold's formulation [14] to query-based learning with negative examples [2, 3], the field has evolved significantly. Notably, AlphaRegex [22] introduced top-down enumerative synthesis for REs with sound pruning. Because REs are monotone, their pruning analyses can be derived naturally, or even automatically [18]. In contrast, PEGs are inherently non-monotone due to prioritized choice and predicates, necessitating our domain-specific analyses. In software engineering, program input grammar inference has progressed from generalizing examples into CFGs [5] to more deterministic, incremental, and expressive methods [4, 20, 26], including visibly pushdown grammars [17]. However, these approaches typically rely on a black-box oracle to validate inputs and mitigate over-generalization, whereas we synthesize PEGs directly from fixed input-output examples without oracle access.

11 Conclusion

To mitigate the notoriously error-prone nature of manual PEG construction, we proposed SynPEG, the first automated synthesis framework directly using input-output examples. SynPEG combines top-down enumeration with sound pruning, leveraging a novel abstract machine that explicitly captures parsing continuations to precisely analyze incomplete grammars. Evaluations confirm its efficiency: SynPEG successfully eliminates 99.82% of the search space and synthesizes all 15 benchmarks in 0.43s on average, offering a reliable alternative to manual construction.

Acknowledgments. This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (No.RS-2024-00344597).

References

1. Artifact for "Synthesizing PEGs via Analysis-based Pruning". https://doi.org/10.5281/zenodo.21330079 (2026)
2. Angluin, D.: Inductive inference of formal languages from positive data. Information and Control **45**(2) (1980). https://doi.org/10.1016/S0019-9958(80)90285-5
3. Angluin, D.: Learning regular sets from queries and counterexamples. Information and Computation **75**(2), 87–106 (1987). https://doi.org/0.1016/0890-5401(87)90052-6

4. Arefin, M.R., Shetiya, S., Wang, Z., Csallner, C.: Fast deterministic black-box context-free grammar inference. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE) (2024). https://doi.org/10.1145/3597503.3639214
5. Bastani, O., Sharma, R., Aiken, A., Liang, P.: Synthesizing program input grammars. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) (2017). https://doi.org/10.1145/3062341.3062349
6. Blaudeau, C., Shankar, N.: A verified packrat parser interpreter for parsing expression grammars. In: Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP) (2020). https://doi.org/10.1145/3372885.3373836
7. Chen, Q., Banerjee, A., Demiralp, c., Durrett, G., Dillig, I.: Data extraction via semantic regular expression synthesis. Proceedings of the ACM on Programming Languages **7**(OOPSLA2) (2023). https://doi.org/10.1145/3622863
8. Chida, N., Terauchi, T.: Repairing regular expressions for extraction. Proceedings of the ACM on Programming Languages **7**(PLDI) (2023). https://doi.org/10.1145/3591287
9. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming languages (POPL) (1977). https://doi.org/10.1145/512950.512973
10. Cousot, P., Cousot, R.: Abstract interpretation frameworks. Journal of Logic and Computation (JLC) **2**(4), 511–547 (1992). https://doi.org/10.1093/logcom/2.4.511
11. Fok, R.Y.T.: Grammar refinement for grammar-based test input generation. Master's thesis, University of California, Los Angeles (2024), https://escholarship.org/uc/item/1z34f7nx
12. Ford, B.: Packrat parsing: simple, powerful, lazy, linear Time, functional pearl. In: Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP) (2002). https://doi.org/10.1145/581478.581483
13. Ford, B.: Parsing expression grammars: a recognition-based syntactic foundation. In: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL) (2004). https://doi.org/10.1145/964001.964011
14. Gold, E.M.: Complexity of automaton identification from given data. Information and Control **37**(3) (1978). https://doi.org/10.1016/S0019-9958(78)90562-4
15. Grimm, R.: Better extensibility through modular syntax. In: Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) (2006). https://doi.org/10.1145/1133981.1133987
16. Ierusalimschy, R.: A text pattern-matching tool based on Parsing Expression Grammars. Software: Practice and Experience **39**(3), 221–258 (2009). https://doi.org/10.1002/spe.892
17. Jia, X., Tan, G.: V-Star: learning visibly pushdown grammars from program inputs. Proceedings of the ACM on Programming Languages **8**(PLDI) (2024). https://doi.org/10.1145/3656458
18. Johnson, K.J., Krishnan, R., Reps, T., D'Antoni, L.: Automating pruning in top-down enumeration for program synthesis problems with monotonic semantics. Proceedings of the ACM on Programming Languages **8**(OOPSLA2) (2024). https://doi.org/10.1145/3689744
19. Ko, S., Park, J.: Technical Report for "Synthesizing PEGs via Analysis-based Pruning". Tech. rep. (2026). https://doi.org/10.5281/zenodo.21450459

20. Kulkarni, N., Lemieux, C., Sen, K.: Learning highly recursive input grammars. In: Proceedings of the IEEE/ACM 36th International Conference on Automated Software Engineering (ASE) (2021). https://doi.org/10.1109/ASE51524.2021.9678879
21. Kuramitsu, K.: A symbol-based extension of parsing expression grammars and context-sensitive packrat parsing. In: Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering (SLE) (2017). https://doi.org/10.1145/3136014.3136025
22. Lee, M., So, S., Oh, H.: Synthesizing regular expressions from examples for introductory automata assignments. In: Proceedings of the 15th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE) (2016). https://doi.org/10.1145/2993236.2993244
23. Lee, Y., Rajiv, G., Sergey, I.: Grammar repair with examples and tree automata. Proceedings of the ACM on Programming Languages **10** (2026). https://doi.org/10.1145/3798242
24. Leung, A., Lerner, S.: Parsimony: an IDE for example-guided synthesis of lexers and parsers. In: Proceedings of the IEEE/ACM 32th International Conference on Automated Software Engineering (ASE) (2017). https://doi.org/10.1109/ASE.2017.8115692
25. Leung, A., Sarracino, J., Lerner, S.: Interactive parser synthesis by example. In: Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) (2015). https://doi.org/10.1145/2737924.2738002
26. Li, F., Chen, X., Xiao, X., Sun, X., Chen, C., Wang, S., Han, J.: Incremental context-free grammar inference in black box settings. In: Proceedings of the IEEE/ACM 39th International Conference on Automated Software Engineering (ASE). Association for Computing Machinery (2024). https://doi.org/10.1145/3691620.3695494
27. Loff, B., Moreira, N., Reis, R.: The computational power of parsing expression grammars. Journal of Computer and System Sciences **111**, 1–21 (2020). https://doi.org/10.1016/j.jcss.2020.01.001
28. M. Cardoso, E., De Paula, R., Pereira, D., Reis, L., Ribeiro, R.G.: Type-based termination analysis for Parsing Expression Grammars. In: Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing (SAC) (2023). https://doi.org/10.1145/3555776.3577620
29. Mascarenhas, F., Medeiros, S., Ierusalimschy, R.: On the relation between context-free grammars and parsing expression grammars. Science of Computer Programming **89**, 235–250 (2014). https://doi.org/10.1016/j.scico.2014.01.012
30. Medeiros, S., Mascarenhas, F.: Syntax error recovery in parsing expression grammars. In: Proceedings of the 33rd Annual ACM Symposium on Applied Computing (SAC) (2018). https://doi.org/10.1145/3167132.3167261
31. de Medeiros, S.Q., Olarte, C.: A semantic framework for PEGs. In: Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering (SLE) (2020). https://doi.org/10.1145/3426425.3426944
32. Medeiros, S., Mascarenhas, F., Ierusalimschy, R.: From regexes to parsing expression grammars. Science of Computer Programming **93**, 3–18 (2014). https://doi.org/10.1016/j.scico.2012.11.006
33. Medeiros, S., Mascarenhas, F., Ierusalimschy, R.: Left recursion in Parsing Expression Grammars. Science of Computer Programming **96**, 177–190 (2014). https://doi.org/10.1016/j.scico.2014.01.013

34. Pan, R., Hu, Q., Xu, G., D'Antoni, L.: Automatic repair of regular expressions. Proceedings of the ACM on Programming Languages **3**(OOPSLA) (2019). https://doi.org/10.1145/3360565
35. Ribeiro, R., Reis, L.V.S., Feitosa, S., Cardoso, E.M.: Towards typed semantics for Parsing Expression Grammars. In: Proceedings of the XXIII Brazilian Symposium on Programming Languages (SBLP) (2019). https://doi.org/10.1145/3355378.3355388
36. Sobernig, S.: Object parsing grammars with composition. In: Proceedings of the 16th International Conference on Software Technologies (ICSOFT) (2021). https://doi.org/10.5220/0010558303730385
37. Warth, A., Douglass, J.R., Millstein, T.: Packrat parsers can support left recursion. In: Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM) (2008). https://doi.org/10.1145/1328408.1328424
38. Wieczorek, W., Unold, O., Strąk, L.: Parsing Expression Grammars and their induction algorithm. Applied Sciences **10** (2020). https://doi.org/10.3390/app10238747
39. Yamaguchi, D., Kuramitsu, K.: CPEG: a typed tree construction from parsing expression grammars with regex-like captures. In: Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing (SAC) (2019). https://doi.org/10.1145/3297280.3297433