

Synthesizing Parsing Expression Grammars via Analysis-based Pruning (Technical Report)

Seongmin Ko and Jihyeok Park

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

Abstract. This is the companion technical report of the paper “Synthesizing Parsing Expression Grammars via Analysis-based Pruning”, which presents **SynPEG**, the first tool to automatically synthesize general PEGs from input-output examples. To keep the main paper concise, several detailed definitions and all proofs are omitted from it and instead provided in this report. We first recall the syntax and semantics of PEGs and formalize the synthesis problem with its top-down enumeration algorithm. We then present the full definitions of the *grammar analysis* and the *example analysis*, including the abstract machine for PEGs, together with the complete proofs of all theorems stated in the main paper. Finally, we report an additional experiment comparing **SynPEG** against a reasoning LLM, which extends the evaluation of RQ1 in the main paper.

1 Parsing Expression Grammars (PEGs)

A *parsing expression grammar* [3] (PEG) is a formalism for describing recursive-descent parsers. This section defines the syntax and semantics of PEGs.

1.1 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* $R : V_N \rightarrow E$
- 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 **a**, the any-terminal $\cdot$, a nonterminal A , a sequence $e_1 e_2$, a prioritized choice e_1/e_2 , a not-predicate $!e$, an and-predicate $\&e$, a repetition e^* or $e+$, or an optional $e?$:

$$E \ni e ::= \varepsilon \mid \mathbf{a} \mid \cdot \mid A \mid e e \mid e/e \mid e^* \mid e+ \mid e? \mid !e \mid \&e$$

1.2 Semantics

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

$$\begin{array}{c}
\frac{}{(e, x) \Downarrow x} \quad \frac{}{(a, ax) \Downarrow x} \quad \frac{\nexists y. x = ay}{(a, x) \Downarrow \mathbf{f}} \quad \frac{R(A) = e \quad (e, x) \Downarrow o}{(A, x) \Downarrow o} \\
\frac{(e_1, x) \Downarrow y \quad (e_2, y) \Downarrow o}{(e_1 e_2, x) \Downarrow o} \quad \frac{(e_1, x) \Downarrow \mathbf{f}}{(e_1 e_2, x) \Downarrow \mathbf{f}} \quad \frac{(e_1, x) \Downarrow y \quad (e_2, y) \Downarrow \mathbf{f}}{(e_1 e_2, x) \Downarrow \mathbf{f}} \\
\frac{(e_1, x) \Downarrow o \quad (e_1, x) \Downarrow \mathbf{f} \quad (e_2, x) \Downarrow o}{(e_1/e_2, x) \Downarrow o} \quad \frac{(e_1, x) \Downarrow \mathbf{f} \quad (e_2, x) \Downarrow \mathbf{f}}{(e_1/e_2, x) \Downarrow \mathbf{f}} \\
\frac{(e, x) \Downarrow y \quad (e^*, y) \Downarrow o}{(e^*, x) \Downarrow o} \quad \frac{(e, x) \Downarrow \mathbf{f}}{(e^*, x) \Downarrow x} \quad \frac{(e, x) \Downarrow \mathbf{f} \quad (e, x) \Downarrow y}{(!e, x) \Downarrow x} \quad \frac{(e, x) \Downarrow y}{(!e, x) \Downarrow \mathbf{f}}
\end{array}$$

$$\mathcal{D}[\cdot] = \mathbf{a}_1/\mathbf{a}_2/\dots/\mathbf{a}_n \quad \mathcal{D}[e+] = e e^* \quad \mathcal{D}[e?] = e/\varepsilon \quad \mathcal{D}[\&e] = !e$$

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 [3] 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.

2 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.

2.1 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.

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

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. 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.

Algorithm 1: Top-Down Enumeration Algorithm

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

```
1  $W \leftarrow \{P_i\}$  where  $P_i = (\emptyset, V_T, \emptyset, \square)$  is the most-general partial PEG
2 while  $W \neq \emptyset$  do
3    $P \leftarrow \operatorname{argmax}_{P' \in W} \text{score}(P')$            // Pick the most promising PEG
4    $W \leftarrow W \setminus \{P\}$                        // Remove from worklist
5   if  $P \not\rightarrow$  then
6     if  $P \in \text{solution}(\mathcal{S})$  then
7       return  $P$                                      // Check if it is a solution
8   else
9      $W \leftarrow W \cup \{P' \mid P \rightarrow P' \wedge \neg \text{prune}(P')\}$  // Add non-pruned PEGs
```

2.2 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., $(\beta_{\text{start}}, x) \Downarrow o$). 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 $\mathcal{S}$:

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

2.3 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, $P_i = (\emptyset, V_T, \emptyset, \square)$. For each iteration, it selects the most promising PEG P from W based on a *scoring function* $\text{score}(P)$. If the selected PEG is complete ($P \rightarrow$), the algorithm evaluates it on all examples in $\mathcal{E}$ to check whether it is a solution. Otherwise, it applies the *hole-filling relation* $P \rightarrow P'$ defined in Figure 1, which generates PEGs P' by filling a single hole in P . 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., $\text{score}(P) = -\|P\|$) and performs no pruning (i.e., $\text{prune}(P) = \#\mathbf{f}$), we introduce *analysis-based pruning techniques* (§3 and §4) to accelerate the synthesis process. The potential score-guided search and further optimization techniques are described in the main paper.

3 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:

$$\begin{array}{c}
\frac{A \notin \text{dom}(R)}{(R, \square) \rightarrow (R[A \mapsto \square \oplus \square], A)} \quad \frac{A \notin \text{dom}(R)}{(R, \square) \rightarrow (R[A \mapsto \ominus \square], A)} \quad \frac{A \in \text{dom}(R)}{(R, \square) \rightarrow (R, A)} \\
\frac{}{(R, \square) \rightarrow (R, \varepsilon)} \quad \frac{}{(R, \square) \rightarrow (R, \mathbf{a})} \quad \frac{}{(R, \square) \rightarrow (R, \cdot)} \quad \frac{(R, \square) \rightarrow (R', \beta)}{(R, \ominus \square) \rightarrow (R', \ominus \beta)} \\
\frac{(R, \square) \rightarrow (R', \beta_1)}{(R, \square \oplus \square) \rightarrow (R', \beta_1 \oplus \square)} \quad \frac{\beta_1 \neq \square \quad (R, \square) \rightarrow (R', \beta_2)}{(R, \beta_1 \oplus \square) \rightarrow (R', \beta_1 \oplus \beta_2)} \\
\frac{(R, \square) \rightarrow (R', \beta)}{(V_N, V_T, R, \square) \rightarrow (\text{dom}(R'), V_T, R', \beta)} \\
\frac{\beta \neq \square \quad (A \leftarrow e) \in R \quad e \text{ contains } \square \quad (R, e) \rightarrow (R', e')}{(V_N, V_T, R, \beta) \rightarrow (\text{dom}(R'), V_T, R'[A \mapsto e'], \beta)}
\end{array}$$

Fig. 1: Hole-Filling Rules $\rightarrow \subseteq \mathbb{P} \times \mathbb{P}$ for PEGs

1) *reducible* (§3.1) and 2) *ill-formed* (§3.2). We formalize these analyses based on the abstract interpretation framework [1, 2] 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* $\widehat{\llbracket A \leftarrow e \rrbracket} : \widehat{D} \rightarrow \widehat{D}$ of a rule $(A \leftarrow e) \in R$.

Then, we compute the least fixed point of the abstract transfer function $\widehat{F} : \widehat{D} \rightarrow \widehat{D}$, defined with the initial abstract element $\widehat{d}_\iota$ and the abstract semantics $\widehat{\llbracket A \leftarrow e \rrbracket}$ of the rules, as follows:

$$\widehat{\llbracket P \rrbracket} = \text{lfp} \widehat{F} = \lim_{n \rightarrow \infty} \widehat{F}^n(\perp) \quad \text{where} \quad \widehat{F}(\widehat{d}) = \widehat{d}_\iota \sqcup \left(\bigsqcup_{(A \leftarrow e) \in R} \widehat{\llbracket A \leftarrow e \rrbracket}(\widehat{d}) \right)$$

3.1 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, the following PEG is reducible because the left branch of the prioritized choice always succeeds, rendering the right branch strictly unreachable:

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

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

$$\begin{aligned}
& \widehat{\llbracket A \leftarrow e \rrbracket}_{\text{pf}}(\widehat{d}) = (X^p, X^f) \text{ where } \begin{cases} X^p = \begin{cases} \{A\} & \text{if } p = \#t \\ \emptyset & \text{otherwise} \end{cases} \\ X^f = \begin{cases} \{A\} & \text{if } f = \#t \vee A \in \widehat{\llbracket P \rrbracket}_{\text{reach}}(A) \\ \emptyset & \text{otherwise} \end{cases} \\ \widehat{\llbracket e \rrbracket}_{\text{pf}}(\widehat{d}) = (p, f) \end{cases} \\
& \widehat{\llbracket \beta_1 \beta_2 \rrbracket}_{\text{pf}}(\widehat{d}) = (p_1 \wedge p_2, f_1 \vee f_2) \quad \widehat{\llbracket \beta_1 / \beta_2 \rrbracket}_{\text{pf}}(\widehat{d}) = (p_1 \vee p_2, f_1 \wedge f_2) \\
& \widehat{\llbracket -^* \rrbracket}_{\text{pf}}(\widehat{d}) = \widehat{\llbracket -? \rrbracket}_{\text{pf}}(\widehat{d}) = (\#t, \#f) \quad \widehat{\llbracket !\beta_1 \rrbracket}_{\text{pf}}(\widehat{d}) = (f_1, p_1) \quad \widehat{\llbracket \&\beta_1 \rrbracket}_{\text{pf}}(\widehat{d}) = (p_1, f_1) \\
& \widehat{\llbracket \beta_1 \rrbracket}_{\text{pf}}(\widehat{d}) = \begin{cases} (A \in X^p, A \in X^f) & \text{if } \beta_1 = A \\ (\#t, \#f) & \text{if } \beta_1 = \varepsilon \\ (\#f, \#f) & \text{otherwise} \end{cases} \quad \text{where } \begin{cases} (p_1, f_1) = \widehat{\llbracket \beta_1 \rrbracket}_{\text{pf}}(\widehat{d}) \\ (p_2, f_2) = \widehat{\llbracket \beta_2 \rrbracket}_{\text{pf}}(\widehat{d}) \\ (X^p, X^f) = \widehat{d} \end{cases} \\
\text{(a) Must-pass/fail analysis with } \widehat{D} = \mathcal{P}(V_N) \times \mathcal{P}(V_N), \widehat{d}_i = (\emptyset, \emptyset), \sqcup = \cup, \text{ and } \widehat{\llbracket - \rrbracket}_{\text{pf}}.
\end{aligned}$$

$$\begin{aligned}
& \widehat{\llbracket e \rrbracket}_{\text{reach}}(\widehat{d}) = \begin{cases} \widehat{\llbracket \beta_1 \rrbracket}_{\text{reach}}(\widehat{d}) \cup \widehat{\llbracket \beta_2 \rrbracket}_{\text{reach}}(\widehat{d}) & \text{if } e = \beta_1 \beta_2 \\ \emptyset & \text{otherwise} \end{cases} \\
& \widehat{\llbracket A \leftarrow e \rrbracket}_{\text{reach}}(\widehat{d}) = \{A \mapsto \widehat{\llbracket e \rrbracket}_{\text{reach}}(\widehat{d})\} \quad \widehat{\llbracket \beta \rrbracket}_{\text{reach}}(\widehat{d}) = \begin{cases} \widehat{d}(A) \cup \{A\} & \text{if } \beta = A \\ \emptyset & \text{otherwise} \end{cases} \\
\text{(b) Sequential-reachability with } \widehat{D} = V_N \rightarrow \mathcal{P}(V_N), \widehat{d}_i = \lambda A. \emptyset, \sqcup = \dot{\cup}, \text{ and } \widehat{\llbracket - \rrbracket}_{\text{reach}}.
\end{aligned}$$

Fig. 2: Grammar analysis to detect *reducible* expressions in PEGs.

on finite inputs. For example, in the following PEG, B is self-sequentially reachable and thus must fail:

$$A \leftarrow B / \square \quad B \leftarrow \mathbf{a} B \quad (\because B / \square \text{ is reducible to } \square)$$

To systematically detect such reducible expressions, we formalize a *must-pass/fail analysis* alongside a *sequential-reachability analysis*, as detailed in Figure 2.

Must-Pass/Fail Analysis This analysis determines whether a given expression is guaranteed to succeed (*must-pass*) or fail (*must-fail*), as formally defined in Figure 2a. 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., $A \in \widehat{\llbracket P \rrbracket}_{\text{reach}}(A)$), it is marked as must-fail. Finally, $\widehat{\llbracket \beta \rrbracket}_{\text{pf}}$ 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 sequential reachability

nonterminals from each nonterminal, as formally defined in Figure 2b using the abstract domain $\widehat{D} = V_N \rightarrow \mathcal{P}(V_N)$.

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., $\widehat{\llbracket \beta \rrbracket}_{\text{pf}}(\widehat{\llbracket P \rrbracket}_{\text{pf}}) = (\#t, _)$), the following equivalences always hold:

$$\beta/\beta' \equiv \beta \quad \beta? \equiv \beta \quad \&\beta \equiv \varepsilon$$

- If β is *must-fail* (i.e., $\widehat{\llbracket \beta \rrbracket}_{\text{pf}}(\widehat{\llbracket P \rrbracket}_{\text{pf}}) = (_, \#t)$), the following equivalences always hold:

$$\beta/\beta' \equiv \beta'/\beta \equiv \beta' \quad \beta \beta' \equiv \beta' \beta \equiv \beta \quad \beta^* \equiv \beta? \equiv !\beta \equiv \varepsilon \quad \&\beta \equiv \beta$$

- Other rules for reducible expressions, including rules defined in [3]:

$$\begin{aligned} \beta/\beta &\equiv \beta & \beta_1/\beta_2/\beta_1 &\equiv \beta_1/\beta_2 & \beta/\varepsilon &\equiv \beta? & \varepsilon \beta &\equiv \beta \varepsilon \equiv \beta \\ !\&\beta &\equiv \&!\beta \equiv !\beta & !!\beta &\equiv \&\&\beta \equiv \&\beta & (\&\beta)? &\equiv (!\beta)? \equiv \varepsilon \\ !(\beta_1 \beta_2)\beta_1 &\equiv \beta_1 !\beta_2 & (\beta_1 \beta_3)/(\beta_2 \beta_3) &\equiv (\beta_1/\beta_2) \beta_3 \end{aligned}$$

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

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

Proof. We first show that a base expression β is *must-pass* (or *must-fail*) if $\widehat{\llbracket \beta \rrbracket}_{\text{pf}}(\widehat{\llbracket P \rrbracket}_{\text{pf}})$ yields $(\#t, _)$ (or $(_, \#t)$); that is, parsing β on any input $x \in \mathbb{S}$ always succeeds (or fails). The abstract semantics $\widehat{\llbracket - \rrbracket}_{\text{pf}}$ defines such base cases and propagates them through nonterminals. In particular, a self-sequentially reachable nonterminal is *must-fail*: if A is reachable from itself through sequencing, the parser may consume unbounded input without a backtracking mark, and thus cannot succeed on any finite input. The reachability semantics $\widehat{\llbracket - \rrbracket}_{\text{reach}}$ precisely captures such sequential reachability, starting from sequence expressions and propagating through nonterminals. We now prove the two cases separately:

- **Case 1 (must-pass).** If β is must-pass, the right operand β' of the ordered choice β/β' is unreachable, making the whole expression equivalent to β . Likewise, $\beta?$ is equivalent to β . Since β always succeeds, $\&\beta$ also succeeds without consuming input and is therefore equivalent to ε .
- **Case 2 (must-fail).** If β is must-fail, the alternative β' in β'/β or β/β' is always selected, so both are equivalent to β' . Regardless of β' , both $\beta\beta'$ and $\beta'\beta$ fail since β always fails, hence they are equivalent to β . Similarly, β^* , $\beta?$, and $!\beta$ always succeed without consuming input, making them equivalent to ε . Finally, $\&\beta$ always fails and is thus equivalent to β .

3.2 Ill-Formed Expressions

A PEG is *ill-formed* [3] 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, any instantiation of the following partial PEG is always ill-formed because of direct left recursion:

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

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.

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

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

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

Similarly, for input $\mathbf{bb}$, the inner nullable expression $\mathbf{a?}$ succeeds without consumption, causing infinite repetitions of the inner expression on the same input $\mathbf{bb}$. To detect such ill-formed expressions, we define 1) *left recursion analysis* and 2) *nullable expression analysis*.

Left Recursion Analysis This analysis collects reachable nonterminals without consuming any symbols from each nonterminal in the abstract domain $\widehat{D} = V_N \rightarrow \mathcal{P}(V_N)$, as defined in Figure 3a. Direct and indirect left recursion are identified by checking whether a nonterminal A can reach itself without consuming any input symbols (i.e., $A \in \widehat{\llbracket P \rrbracket}_l(A)$). Note that it utilizes the nullable expression analysis to determine whether the first expression in a sequence is nullable.

Nullable Expression Analysis Figure 3b defines this analysis, which collects nullable nonterminals in the domain $\widehat{D} = \mathcal{P}(V_N)$ and identifies nullable expressions. 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 $P = (V_N, V_T, R, \beta_{\text{start}})$, if $\text{prune}_{\text{ill}}(P)$ holds, then every instantiation of P does not terminate on at least one input:*

$$\text{prune}_{\text{ill}}(P) = \begin{cases} (\exists A \in V_N. A \in \widehat{\llbracket P \rrbracket}_l(A)) & \vee & (\text{explicit left recursion}) \\ (\exists (A \leftarrow \beta^*) \in R. \widehat{\llbracket \beta \rrbracket}_{\text{null}}(\widehat{\llbracket P \rrbracket}_{\text{null}}) = \#t) & & (\text{implicit left recursion}) \end{cases}$$

$$\begin{aligned}
\widehat{\llbracket A \leftarrow e \rrbracket}_r(\hat{d}) &= \{A \mapsto \widehat{\llbracket e \rrbracket}_r(\hat{d})\} & \widehat{\llbracket A \rrbracket}_r(\hat{d}) &= \hat{d}(A) \cup \{A\} & \widehat{\llbracket \ominus \beta \rrbracket}_r(\hat{d}) &= \widehat{\llbracket \beta \rrbracket}_r(\hat{d}) \\
\widehat{\llbracket \varepsilon \rrbracket}_r(\hat{d}) &= \widehat{\llbracket \mathbf{a} \rrbracket}_r(\hat{d}) = \widehat{\llbracket \cdot \rrbracket}_r(\hat{d}) = \widehat{\llbracket \square \rrbracket}_r(\hat{d}) = \emptyset & \widehat{\llbracket \beta_1 / \beta_2 \rrbracket}_r(\hat{d}) &= \widehat{\llbracket \beta_1 \rrbracket}_r(\hat{d}) \cup \widehat{\llbracket \beta_2 \rrbracket}_r(\hat{d}) \\
\widehat{\llbracket \beta_1 \beta_2 \rrbracket}_r(\hat{d}) &= \begin{cases} \widehat{\llbracket \beta_1 \rrbracket}_r(\hat{d}) \cup \widehat{\llbracket \beta_2 \rrbracket}_r(\hat{d}) & \text{if } \widehat{\llbracket \beta_1 \rrbracket}_{\text{null}}(\hat{d}) = \#t \text{ (i.e., } \beta_1 \text{ is nullable)} \\ \widehat{\llbracket \beta_1 \rrbracket}_r(\hat{d}) & \text{otherwise} \end{cases}
\end{aligned}$$

(a) Left recursion analysis with $\hat{D} = V_N \rightarrow \mathcal{P}(V_N)$, $\hat{d}_\varepsilon = \lambda A. \emptyset$, $\sqcup = \cup$, and $\widehat{\llbracket - \rrbracket}_r$.

$$\begin{aligned}
\widehat{\llbracket A \leftarrow e \rrbracket}_{\text{null}}(\hat{d}) &= \begin{cases} \{A\} & \text{if } \widehat{\llbracket e \rrbracket}_{\text{null}}(\hat{d}) = \#t \\ \emptyset & \text{otherwise} \end{cases} & \widehat{\llbracket A \rrbracket}_{\text{null}}(\hat{d}) &= A \in \hat{d} \\
\widehat{\llbracket \mathbf{a} \rrbracket}_{\text{null}}(\hat{d}) &= \widehat{\llbracket \cdot \rrbracket}_{\text{null}}(\hat{d}) = \widehat{\llbracket \square \rrbracket}_{\text{null}}(\hat{d}) = \#f & \widehat{\llbracket \varepsilon \rrbracket}_{\text{null}}(\hat{d}) &= \widehat{\llbracket \ominus \beta \rrbracket}_{\text{null}}(\hat{d}) = \#t \\
\widehat{\llbracket \beta_1 \beta_2 \rrbracket}_{\text{null}}(\hat{d}) &= \widehat{\llbracket \beta_1 \rrbracket}_{\text{null}}(\hat{d}) \wedge \widehat{\llbracket \beta_2 \rrbracket}_{\text{null}}(\hat{d}) & \widehat{\llbracket \beta_1 / \beta_2 \rrbracket}_{\text{null}}(\hat{d}) &= \widehat{\llbracket \beta_1 \rrbracket}_{\text{null}}(\hat{d}) \vee \widehat{\llbracket \beta_2 \rrbracket}_{\text{null}}(\hat{d})
\end{aligned}$$

(b) Nullable expression analysis with $\hat{D} = \mathcal{P}(V_N)$, $\hat{d}_\varepsilon = \emptyset$, $\sqcup = \cup$ and $\widehat{\llbracket - \rrbracket}_{\text{null}}$.

Fig. 3: Grammar analysis to detect *ill-formed* expressions in PEGs.

Proof. The abstract semantics $\widehat{\llbracket - \rrbracket}_{\text{null}}$ determines whether an expression is *nullable*, that is whether there exists at least one input string that can be parsed successfully without consuming any symbols. It begins from base cases (e.g., ε is nullable, whereas $\mathbf{a}$ is not) and propagates nullable information through non-terminals. As a must-analysis ensuring the existence of at least one such input, holes are conservatively treated as non-nullable. The semantics also tracks all nonterminals reachable without consuming symbols. If β is nullable in β/β' , then β' becomes reachable without consuming input. From this information, we detect (1) *left recursion* when a nonterminal is reachable from itself without consuming symbols, and (2) *infinite loops* when a nullable expression appears as the operand of a repetition operator. Therefore, if $\text{prune}_{\text{ill}}(P)$ holds, there exists at least one input string that causes the parsing process of P to diverge.

4 Example Analysis

The second component of our synthesis algorithm is *example analysis*. Unlike the static grammar analysis (§3), 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 (§4.1), for-

malizes the computation of possible outputs and state equivalence (§4.2), and then presents the two example-based pruning techniques (§4.3 and §4.4).

4.1 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 §1.2 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.

States	$\sigma \in \Sigma ::= \mathbb{S} \times \mathbb{K} \times \mathbb{I}_{\text{consume}} \times \mathbb{I}_{\text{seen}} \times \Phi$
Continuations	$\kappa \in \mathbb{K} = \mathbb{C} \times \mathbb{I}_{\text{input}} \times \mathbb{I}_{\text{mark}}^? \times \mathbb{K}^?$
Control Points	$c \in \mathbb{C} = \blacktriangleright \mathbf{S} \mid \mathbf{S} \blacktriangleleft \mid \blacktriangleright A_L \mid A_L \blacktriangleleft \mid \blacktriangleright A_R \mid A_R \blacktriangleleft$
Indices	$i \in \mathbb{I} = \mathbb{N}$
Flags	$\phi \in \Phi ::= \mathbf{p} \mid \mathbf{f}$

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

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* (**p**) or *fail* (**f**) state.

$$A; \quad A \leftarrow B / a; \quad B \leftarrow a \ c$$

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

$$\begin{aligned}
& (\blacktriangleright S, \overset{S}{|} | | abc, p) \quad \rightsquigarrow \quad (\blacktriangleright A_L, \overset{S A_L}{|} | | | abc, p) \quad \rightsquigarrow \quad (\blacktriangleright B_L, \overset{S A_L \ B_L}{|} | | | | abc, p) \\
& \rightsquigarrow (B_L \blacktriangleleft, \overset{S A_L \ B_L}{|} | | | | a | | bc, p) \rightsquigarrow (\blacktriangleright B_R, \overset{S A_L \ B_R}{|} | | | | a | | bc, p) \rightsquigarrow (B_R \blacktriangleleft, \overset{S A_L \ B_R}{|} | | | | ab | | c, f) \\
& \rightsquigarrow (A_L \blacktriangleleft, \overset{S A_L}{|} | | | ab | | c, f) \quad \rightsquigarrow \quad (\blacktriangleright A_R, \overset{S A_R}{|} | | | ab | c, p) \quad \rightsquigarrow \quad (A_R \blacktriangleleft, \overset{S A_R}{|} | | | a | b | c, p) \\
& \rightsquigarrow (S \blacktriangleleft, \overset{S}{|} | | | a | b | c, p)
\end{aligned}$$

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

Fig. 4: An example of the parsing process in the abstract machine.

Example Consider the PEG in Figure 4a and its parsing process on the input string abc . It first attempts the left branch of the choice in A , namely $B \leftarrow ac$, 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 4b, which is shown below:

$$\langle abc, (B_R \blacktriangleleft, 0, \perp, (\blacktriangleright A_L, 0, 0, (\blacktriangleright S, 0, \perp, \perp))), 2, 2, f \rangle \quad \equiv \quad (B_R \blacktriangleleft, \overset{S A_L \ B_R}{|} | | | | ab | | c, f)$$

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 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 p :

$$\langle abc, (\blacktriangleright A_R, 0, \perp, (\blacktriangleright S, 0, \perp, \perp)), 0, 2, p \rangle \quad \equiv \quad (\blacktriangleright A_R, \overset{S A_R}{|} | | | ab | c, p)$$

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

$$\langle abc, (S \blacktriangleleft, 0, \perp, \perp), 1, 2, p \rangle \quad \equiv \quad (S \blacktriangleleft, \overset{S}{|} | | | a | b | c, p)$$

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.

$$\begin{aligned}
\llbracket A_L \blacktriangleleft \rrbracket_c(\sigma) &= \begin{cases} \sigma.\text{next} & \text{if } R(A) = _ \cdot _ \wedge \phi = \mathbf{p} \\ \sigma.\text{exit} & \text{if } R(A) = _ \cdot _ \wedge \phi = \mathbf{f} \\ \sigma.\text{remove.exit} & \text{if } R(A) = _ / _ \wedge \phi = \mathbf{p} \\ \sigma.\text{revert.flip.next} & \text{if } R(A) = _ / _ \wedge \phi = \mathbf{f} \\ \sigma.\text{remove.entry} & \text{if } R(A) = _ * _ \wedge \phi = \mathbf{p} \\ \sigma.\text{revert.flip.exit} & \text{if } R(A) = _ * _ \wedge \phi = \mathbf{f} \\ \sigma.\text{remove.exit} & \text{if } R(A) = _ ? _ \wedge \phi = \mathbf{p} \\ \sigma.\text{revert.flip.exit} & \text{if } R(A) = _ ? _ \wedge \phi = \mathbf{f} \\ \sigma.\text{revert.flip.exit} & \text{if } R(A) = ! _ \\ \sigma.\text{revert.exit} & \text{if } R(A) = \& _ \end{cases} \\
\llbracket \blacktriangleright A_L \rrbracket_c(\sigma) &= \begin{cases} \llbracket \beta \rrbracket_\beta(\sigma) & \text{if } R(A) = \beta \cdot _ \\ \llbracket \beta \rrbracket_\beta(\sigma.\text{mark}) & \text{if } R(A) = \beta / _ \\ \llbracket \beta \rrbracket_\beta(\sigma.\text{mark}) & \text{if } R(A) = \ominus \beta \end{cases} \quad \llbracket \varepsilon \rrbracket_\beta(\sigma) = \sigma.\text{next} \\
\llbracket \blacktriangleright S \rrbracket_c(\sigma) &= \llbracket \beta_{\text{start}} \rrbracket_c(\sigma) \quad \llbracket a \rrbracket_\beta(\sigma) = \begin{cases} \sigma'.\text{next} & \text{if } a = b \\ \sigma'.\text{flip.next} & \text{otherwise} \end{cases} \\
&\quad \text{where } \sigma.\text{read} = (\sigma', b) \\
\llbracket \blacktriangleright A_R \rrbracket_c(\sigma) &= \begin{cases} \llbracket \beta \rrbracket_\beta(\sigma) & \text{if } R(A) = _ \oplus \beta \\ \sigma.\text{next} & \text{if } R(A) = \ominus _ \end{cases} \quad \llbracket \cdot \rrbracket_\beta(\sigma) = \begin{cases} \sigma'.\text{next} & \text{if } \perp \neq b \\ \sigma'.\text{flip.next} & \text{otherwise} \end{cases} \\
&\quad \text{where } \sigma.\text{read} = (\sigma', b) \\
\llbracket A_R \blacktriangleleft \rrbracket_c(\sigma) &= \sigma.\text{return} \quad \llbracket A \rrbracket_\beta(\sigma) = \sigma.\text{call}(A)
\end{aligned}$$

Fig. 5: The semantics $\llbracket - \rrbracket_c$ of control points $c \in \mathbb{C}$ and $\llbracket - \rrbracket_\beta$ of base expressions $\beta \in \mathbb{B}$ for a PEG $P = (V_N, V_T, R, \beta_{\text{start}})$ where $\sigma = \langle x, \kappa, i^c, i^s, \phi \rangle$ and $\kappa = (c, i, i^m, \kappa')$.

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

$$\sigma \rightsquigarrow \sigma' \iff \sigma = \langle _, (c, _, _, _), _, _, _ \rangle \wedge \sigma' = \llbracket c \rrbracket_c(\sigma)$$

where the transition logic is defined by the semantics of control points $\llbracket - \rrbracket_c$ and base expressions $\llbracket - \rrbracket_\beta$, as detailed in Figure 5.

The helper functions in Figure 6 manipulate specific components of a state. For brevity, we use the notation $X[Y = Z]$ to denote updating the component Y in X with the new value Z . The **read** operation reads a symbol from the input string at the current consumed index, updating both the consumed and seen indices accordingly. Note that when inspecting the end-of-string (EOS) for the first time, only the seen index increases, while the consumed index remains unchanged. The operations **entry**, **next**, and **exit** describe how the continuation advances when moving to the entry, next, or exit points of the start expression (**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.

$$\begin{aligned}
\sigma.\text{read} &= \begin{cases} (\sigma[i^c = i^c + 1][i^s = i_0^s], x[i^c]) & \text{if } i^c < \|x\| \\ (\sigma[i^c = i^c][i^s = i_0^s], \perp) & \text{otherwise} \end{cases} \quad \text{where } i_0^s = \begin{cases} i^s + 1 & \text{if } i^c = i^s \\ i^s & \text{otherwise} \end{cases} \\
\sigma.\text{entry} &= \sigma[c = \blacktriangleright A_L] \text{ where } c = A_L \blacktriangleleft \\
\sigma.\text{next} &= \begin{cases} \sigma[c = \mathbf{S} \blacktriangleleft] & \text{if } c = \blacktriangleright \mathbf{S} \\ \sigma[c = A_L \blacktriangleleft] & \text{if } c = \blacktriangleright A_L \\ \sigma[c = \blacktriangleright A_R] & \text{if } c = A_L \blacktriangleleft \\ \sigma[c = A_R \blacktriangleleft] & \text{if } c = \blacktriangleright A_R \end{cases} \\
\sigma.\text{exit} &= \sigma[c = A_R \blacktriangleleft] \text{ where } c = A_L \blacktriangleleft \\
\sigma.\text{mark} &= \sigma[\kappa = \kappa[i^m = i^c]] \\
\sigma.\text{remove} &= \sigma[\kappa = \kappa[i^m = \perp]] \\
\sigma.\text{revert} &= \sigma[i^c = i^m].\text{remove} \\
\sigma.\text{call}(A) &= \sigma[\kappa = (\blacktriangleright A_L, i^c, \perp, \kappa)] \\
\sigma.\text{return} &= \sigma[\kappa = \kappa'].\text{next} \\
\sigma.\text{flip} &= \sigma[\phi = \neg\phi]
\end{aligned}$$

Fig. 6: Helper functions for states Σ of the abstract machine where $\sigma = \langle x, \kappa, i^c, i^s, \phi \rangle$ and $\kappa = (c, i, i^m, \kappa')$.

Theorem 3 (Correctness of the Abstract Machine). *For a partial PEG $P = (V_N, V_T, R, \beta_{\text{start}})$ 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 §1.2:*

$$\begin{aligned}
\langle x, (\blacktriangleright \mathbf{S}, 0, \perp, \perp), 0, 0, \mathbf{p} \rangle &\rightsquigarrow^* \langle x, (\mathbf{S} \blacktriangleleft, 0, \perp, \perp), i^c, i^s, \mathbf{p} \rangle \iff (\beta_{\text{start}}, x) \Downarrow x[i^c :] \\
\langle x, (\blacktriangleright \mathbf{S}, 0, \perp, \perp), 0, 0, \mathbf{p} \rangle &\rightsquigarrow^* \langle x, (\mathbf{S} \blacktriangleleft, 0, \perp, \perp), i^c, i^s, \mathbf{f} \rangle \iff (\beta_{\text{start}}, x) \Downarrow \mathbf{f}
\end{aligned}$$

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

Proof. The abstract machine mimics the parsing process described in the evaluation relation of PEGs defined in §1.2. The normal form makes all non-base expressions to be defined as separate rules and the abstract machine captures the state of the machine when a nonterminal is called. In addition, the seen index i^s simply captures the furthest index reached during the parsing process according to the definition of the evaluation relation. Thus, the proof can be done by applying structural induction on the evaluation relation and case analysis on the structure of the normalized PEG P .

4.2 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 $\text{halt}(P, x)$ to denote this halting state of the abstract machine when parsing an input string x with a PEG P :

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

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.

$$\begin{aligned}
\text{propagate}(P, \kappa, \hat{o}) &= \begin{cases} \hat{o} & \text{if } \kappa = \perp \\ \text{propagate}(P, \kappa', \hat{o}') & \text{otherwise} \end{cases} \\
\hat{o}' &= \begin{cases} \hat{o} \sqcup ([i^m, +\infty], \top) & \text{if } c = \blacktriangleright A_L \wedge R(A) = _ / _ \wedge \hat{\mathbf{f}} = \top \wedge i^m \neq \perp \\ (\hat{i} \uparrow \sqcup [i^m, i^m], \perp) & \text{if } c = \blacktriangleright A_L \wedge R(A) = _ * _ \wedge \hat{\mathbf{f}} = \top \wedge i^m \neq \perp \\ (\hat{i} \uparrow, \perp) & \text{if } c = \blacktriangleright A_L \wedge R(A) = _ * _ \wedge \hat{\mathbf{f}} = \perp \wedge i^m \neq \perp \\ (\hat{i} \sqcup [i^m, i^m], \perp) & \text{if } c = \blacktriangleright A_L \wedge R(A) = _ ? _ \wedge \hat{\mathbf{f}} = \top \wedge i^m \neq \perp \\ (\hat{i}, \perp) & \text{if } c = \blacktriangleright A_L \wedge R(A) = _ ? _ \wedge \hat{\mathbf{f}} = \perp \wedge i^m \neq \perp \\ ([i^m, i^m], \perp) & \text{if } c = \blacktriangleright A_L \wedge R(A) = _ ! _ \wedge \hat{\mathbf{f}} = \top \wedge i^m \neq \perp \wedge \hat{i} = \perp \\ ([i^m, i^m], \top) & \text{if } c = \blacktriangleright A_L \wedge R(A) = _ ! _ \wedge \hat{\mathbf{f}} = \top \wedge i^m \neq \perp \wedge \hat{i} \neq \perp \\ (\perp, \perp) & \text{if } c = \blacktriangleright A_L \wedge R(A) = _ ! _ \wedge \hat{\mathbf{f}} = \perp \wedge i^m \neq \perp \wedge \hat{i} = \perp \\ (\perp, \top) & \text{if } c = \blacktriangleright A_L \wedge R(A) = _ ! _ \wedge \hat{\mathbf{f}} = \perp \wedge i^m \neq \perp \wedge \hat{i} \neq \perp \\ (\perp, \hat{\mathbf{f}}) & \text{if } c = \blacktriangleright A_L \wedge R(A) = _ \& _ \wedge i^m \neq \perp \wedge \hat{i} = \perp \\ ([i^m, i^m], \hat{\mathbf{f}}) & \text{if } c = \blacktriangleright A_L \wedge R(A) = _ \& _ \wedge i^m \neq \perp \wedge \hat{i} \neq \perp \\ (\hat{i} \uparrow, \top) & \text{if } c = \blacktriangleright A_L \wedge R(A) = _ \cdot _ \\ \hat{o} & \text{otherwise} \end{cases}
\end{aligned}$$

Fig. 7: The **propagate** function, which propagates the abstract output $\hat{o} = (\hat{i}, \hat{\mathbf{f}})$ backward through the continuation $\kappa = (c, _, i^m, \kappa')$.

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:

$$\begin{aligned}
\text{Abstract Outputs} & \quad \hat{o} \in \hat{\mathbb{O}} = \hat{\mathbb{I}} \times \hat{\mathbb{F}} \\
\text{Abstract Index Ranges} & \quad \hat{i} \in \hat{\mathbb{I}} = \{\perp\} \uplus \mathbb{I} \times (\mathbb{I} \uplus \{+\infty\}) \\
\text{Abstract Fails} & \quad \hat{\mathbf{f}} \in \hat{\mathbb{F}} = \{\perp, \top\}
\end{aligned}$$

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

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

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

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

For example, $\text{output}(\mathbf{a}\square, \mathbf{aab}) = ([1, +\infty], \top)$ indicates that any instantiation either fails or succeeds while consuming at least 1 symbol from **aab**. Similarly, $\text{output}(\mathbf{a}(\mathbf{b} \& \square)?, \mathbf{abab}) = ([1, 2], \perp)$ 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 $\sigma = (x, \kappa, i^c, _, _) \in \Sigma$ where the continuation $\kappa = (_, _, i^m, \kappa')$, the **reachable string** $\text{reach}(\sigma)$ is defined as:

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

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:

$$A; \quad A \leftarrow \mathbf{a} \ B; \quad B \leftarrow C / \square; \quad C \leftarrow \mathbf{b} \ \square \quad \left(\triangleright C_R, \begin{array}{c} \text{SAR} \ B_L \ C_R \\ | \ | \ \mathbf{a} \ | \ | \ | \ \mathbf{b} \ | \ | \ \mathbf{cd}, \mathbf{f} \\ \text{m} \quad \text{c}_s \end{array} \right)$$

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:

$$\kappa_1 \approx_c \kappa_2 \iff (\kappa'_1 = \kappa'_2 = \perp) \vee (c_1 = c_2 \wedge \kappa'_1 \approx_c \kappa'_2)$$

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:

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

4.3 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, the partial PEG P_3 always produces the output **b** for the input **ab** without reaching the hole. This contradicts the expected result **f**:

$$P_3 : A \leftarrow \mathbf{a} / \square \quad (\mathbf{ab}, \mathbf{f}) \in \mathcal{E} \quad \text{output}(P_3, \mathbf{ab}) = ([1, 1], \perp)$$

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**:

$$P'_3 : A \leftarrow \mathbf{a} \square \quad (\mathbf{aaab}, \mathbf{aaab}) \in \mathcal{E} \quad \text{output}(P'_3, \mathbf{aaab}) = ([1, +\infty], \top)$$

As indicated by the abstract output $([1, +\infty], \top)$, the set of possible concrete outputs of P'_3 for this input is $\{\mathbf{aab}, \mathbf{ab}, \mathbf{b}, \epsilon, \mathbf{f}\}$. 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 $\text{prune}_{\text{ex}}(P)$ holds, then every instantiation of P violates at least one example:*

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

Proof. Suppose a partial PEG P is invalid with respect to an example $(x, o) \in \mathcal{E}$. By Theorem 4 for validity, the abstract machine produces a parsing result $o' \neq o$ when parsing x . By Theorem 3, every instantiation of P yields the same result o' , which also differs from o . Consequently, no instantiation of P can satisfy the example set $\mathcal{E}$.

4.4 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 below. For inputs **ab** and **bb**, it halts at the hole with the following states:

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

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 $\blacktriangleright A_R$). 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 $\text{prune}_{\text{inc}}(P)$ holds, then every instantiation of P produces inconsistent parsing results for at least one pair of examples in $\mathcal{E}$:*

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

Proof. Suppose a partial PEG P yields inconsistent parsing results for two examples $(x_1, o_1), (x_2, o_2) \in \mathcal{E}$ such that $o_1 \neq o_2$. By Theorem 5 for consistency, the abstract machine reaches identical *remaining controls* and *reachable strings* for both examples. By the definition of the abstract machine, any instantiation of P produces the same parsing result whenever these two components coincide. Therefore, all instantiations of P must yield the same output for (x_1, x_2) , which contradicts the expected outputs $o_1 \neq o_2$. Hence, no instantiation of P can satisfy the example set $\mathcal{E}$.

5 Evaluation

This section reports an additional experiment that extends the evaluation of **RQ1** in §8 of the main paper, which asks whether SynPEG can synthesize PEGs for our benchmarks and compares it against LLMs. While the main paper compares SynPEG against a no-reasoning LLM baseline, here we further compare it against a reasoning LLM to examine whether stronger reasoning ability is sufficient to solve our benchmarks.

5.1 Comparison with a Reasoning LLM

We evaluate a reasoning LLM (GPT-5.4 with the highest reasoning effort) under the same experimental settings as the main paper, including the prompts, five independent runs per benchmark, and a 5-minute timeout. Table 1 reports the per-benchmark results.

The reasoning model solves 13 of 15 benchmarks, a substantial improvement over the no-reasoning baseline, but it still fails on the two hardest ones (Benchmark 5 and Benchmark 14). On Benchmark 5 ($\mathbf{a}^{2^n}$), all five runs in the LLM_{Ex} setting reach 99.9% accuracy yet do not actually solve it, as they overfit the examples and cannot express $\mathbf{a}^{2^n}$ for all n . The remaining configurations, Benchmark 5 in the $\text{LLM}_{\text{Ex+Desc}}$ setting and Benchmark 14 in both settings, time out in all runs while searching for a generalized solution.

Table 1: Performance of a reasoning LLM (GPT-5.4 with the highest reasoning effort) under the LLM_{Ex} and $\text{LLM}_{\text{Ex+Desc}}$ settings, compared with SynPEG, with accuracy (**acc**), execution time (**t**), and generated PEG size ($\|P\|$). The $\pm$ values indicate the standard deviation across multiple runs.

No.	LLM_{Ex}			$\text{LLM}_{\text{Ex+Desc}}$			SynPEG		
	acc (%)	t (s)	$\ P\ $	acc (%)	t (s)	$\ P\ $	acc (%)	t (s)	$\ P\ $
1	100.0 $\pm$ 0.0	28.83 $\pm$ 8.55	4.2 $\pm$ 0.4	100.0 $\pm$ 0.0	19.34 $\pm$ 11.24	4.0 $\pm$ 0.0	100.0	0.44	5.0
2	100.0 $\pm$ 0.0	36.00 $\pm$ 16.57	5.0 $\pm$ 0.0	100.0 $\pm$ 0.0	44.93 $\pm$ 41.41	5.2 $\pm$ 0.4	100.0	0.02	4.0
3	100.0 $\pm$ 0.0	39.77 $\pm$ 44.24	4.0 $\pm$ 0.0	100.0 $\pm$ 0.0	30.00 $\pm$ 26.00	3.8 $\pm$ 0.4	100.0	0.01	3.0
4	100.0 $\pm$ 0.0	152.28 $\pm$ 34.39	5.6 $\pm$ 0.5	100.0 $\pm$ 0.0	170.10 $\pm$ 44.47	5.3 $\pm$ 0.5	100.0	0.62	5.0
5	99.9 $\pm$ 0.0	256.66 $\pm$ 35.18	21.0 $\pm$ 17.0	TO	TO	TO	100.0	1.08	5.0
6	100.0 $\pm$ 0.0	145.83 $\pm$ 49.61	6.0 $\pm$ 0.0	100.0 $\pm$ 0.0	147.11 $\pm$ 38.16	5.8 $\pm$ 0.4	100.0	0.08	4.0
7	100.0 $\pm$ 0.0	27.96 $\pm$ 22.40	9.0 $\pm$ 2.7	100.0 $\pm$ 0.0	12.57 $\pm$ 7.47	10.0 $\pm$ 2.2	100.0	0.03	5.0
8	100.0 $\pm$ 0.0	26.86 $\pm$ 15.21	10.0 $\pm$ 0.0	100.0 $\pm$ 0.0	28.42 $\pm$ 8.59	10.0 $\pm$ 0.0	100.0	0.13	6.0
9	100.0 $\pm$ 0.0	46.87 $\pm$ 34.05	7.6 $\pm$ 1.9	100.0 $\pm$ 0.0	36.09 $\pm$ 17.33	7.0 $\pm$ 1.2	100.0	0.75	6.0
10	100.0 $\pm$ 0.0	35.95 $\pm$ 11.72	6.4 $\pm$ 0.9	100.0 $\pm$ 0.0	52.93 $\pm$ 20.93	6.0 $\pm$ 0.0	100.0	0.58	5.0
11	100.0 $\pm$ 0.0	74.44 $\pm$ 46.65	9.6 $\pm$ 0.9	100.0 $\pm$ 0.0	90.70 $\pm$ 42.77	9.0 $\pm$ 1.4	100.0	0.40	5.0
12	100.0 $\pm$ 0.0	59.75 $\pm$ 32.27	4.0 $\pm$ 0.0	100.0 $\pm$ 0.0	39.76 $\pm$ 21.40	4.0 $\pm$ 0.0	100.0	0.06	4.0
13	100.0 $\pm$ 0.0	38.17 $\pm$ 6.32	4.0 $\pm$ 0.0	100.0 $\pm$ 0.0	39.18 $\pm$ 20.33	4.0 $\pm$ 0.0	100.0	0.57	6.0
14	TO	TO	TO	TO	TO	TO	100.0	0.83	5.0
15	100.0 $\pm$ 0.0	29.78 $\pm$ 9.80	5.0 $\pm$ 0.0	100.0 $\pm$ 0.0	107.69 $\pm$ 22.83	8.0 $\pm$ 0.0	100.0	0.90	5.0
Avg.	93.3	66.61	7.2	81.2	54.59	6.3	100.0	0.43	4.9
Sol.		13.00/15			13.00/15			15.00/15	

Moreover, the reasoning model is slow and cost-ineffective even when it succeeds, taking about one minute and 2.1K output tokens (roughly \$0.36) per task on average. In contrast, SynPEG is deterministic and synthesizes correct PEGs for all benchmarks in under one second on average.

In conclusion, even a reasoning LLM cannot reliably solve our benchmarks, as it fails on the hardest tasks and remains slow and cost-ineffective, which further supports the answer to **RQ1** that example-based PEG synthesis is a non-trivial task unsuitable for general-purpose models.

References

1. 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>
2. 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>
3. 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>