Online Input Grammar Synthesis Aided Symbolic Execution

KE MA[*‡], National University of Defense Technology, China
YUNLAI LUO[*‡], National University of Defense Technology, China
ZHENBANG CHEN[†‡], National University of Defense Technology, China
WEIJIANG HONG[‡], National University of Defense Technology, China
YUFENG ZHANG, Hunan University, China
JI WANG[‡], National University of Defense Technology, China

Symbolic execution faces the challenge of generating valid inputs when analyzing the program with complex input formats. Token-based symbolic execution can partially tackle this challenge but is still doomed by the difficulty of passing input checking and failing to analyze the code after input checking. We propose Lase, an online input grammar synthesis aided symbolic execution method, to generate valid inputs for improving the effectiveness of symbolic execution. Inside Lase, we propose an input grammar-oriented search strategy and a token-level grammar synthesis method. The search strategy selects the paths to cover more syntax rules in priority. The token-level grammar synthesis improves the synthesized grammar's precision and completeness while ensuring efficiency. The experimental results on real-world parsing programs with complex input grammars demonstrate that Lase can improve the coverage of parsing code and generate more valid inputs to improve the coverage of functionality code significantly. Furthermore, compared with the state-of-the-art grammar synthesis methods, the grammars learned by Lase have better precision and recall on most benchmark programs.

CCS Concepts: • **Software and its engineering → Software verification and validation**.

Additional Key Words and Phrases: symbolic execution, grammar synthesis, parsing

ACM Reference Format:
Ke Ma, Yunlai Luo, Zhenbang Chen, Weijiang Hong, Yufeng Zhang, and Ji Wang. 2026. Online Input Grammar Synthesis Aided Symbolic Execution. *Proc. ACM Program. Lang.* 10, OOPSLA1, Article 135 (April 2026), 28 pages. https://doi.org/10.1145/3798243

1 Introduction

Symbolic execution [Baldoni et al. 2018; King 1976] is a widely used technique for thoroughly exploring a program's path space, and has wide applications in many areas, such as automatic

[*]The first two authors contributed equally to this research and are co-first authors.
[†]Zhenbang Chen is the corresponding author.
[‡]Also with the affiliation: State Key Laboratory of Complex & Critical Software Environment, National University of Defense Technology, Changsha, China.

Authors' Contact Information: Ke Ma, College of Computer Science and Technology, National University of Defense Technology, Changsha, China, ke@nudt.edu.cn; Yunlai Luo, College of Computer Science and Technology, National University of Defense Technology, Changsha, China, luoyl@nudt.edu.cn; Zhenbang Chen, College of Computer Science and Technology, National University of Defense Technology, Changsha, China, zbchen@nudt.edu.cn; Weijiang Hong, College of Computer Science and Technology, National University of Defense Technology, Changsha, China, hongweijiang17@nudt.edu.cn; Yufeng Zhang, College of Computer Science and Electronic Engineering, Hunan University, Changsha, China, yufengzhang@hnu.edu.cn; Ji Wang, College of Computer Science and Technology, National University of Defense Technology, Changsha, China, wj@nudt.edu.cn.

ACM 2475-1421/2026/4-ART135
https://doi.org/10.1145/3798243

```
1  enum Token{T_EOF=0,T_NUM,T_OP};
2  int token;
3  int getNextToken(){
4    char c = getchar();
5    //eat space
6    while (isspace(c))
7      c = getchar();
8    if (c == EOF) return T_EOF;
9    if (c >= '0' && c <= '9'){
10     return T_NUM;
11   } else if (c == '+' || c == '-'){
12     return T_OP;
13   }
14   throw "Token Exception!";
15 }
```

```
16 void parseTerm(){
17   token=getNextToken();
18   switch(token){
19     case T_NUM: break;
20     default: throw "Parsing Exception!";}
21   parseOp();
22 }
23 void parseOp(){
24   token=getNextToken();
25   switch(token){
26     case T_OP: parseTerm();break;
27     case T_EOF: break;
28     default: throw "Parsing Exception!";}
29 }
30 void main() { parseTerm(); execute(); }
```

Fig. 1. An example to illustrate the challenge for DSE to analyze programs with complex input grammar.

software testing [Cadar et al. 2008], bug detection [Yi et al. 2024], and program repair [Shariffdeen et al. 2021], to name a few. Until now, symbolic execution still faces two key challenges [Baldoni et al. 2018], *i.e.*, *path explosion* and *constraint solving*, which hinder its development and more extensive applications. Dynamic symbolic execution [Godefroid et al. 2005; Sen et al. 2005] (DSE), also known as concolic testing or whitebox fuzzing, mitigates the two challenges by combining symbolic execution and concrete execution to reduce the times and complexity of constraint solving and enable a deeper analysis of the program.

DSE analyzes a program $\mathcal{P}$ by executing $\mathcal{P}$ both concretely and symbolically. Given an initial input $\mathcal{I}$, DSE executes $\mathcal{P}$ concretely on $\mathcal{I}$. Along with the concrete execution, DSE also records the input constraint satisfying which the input can steer $\mathcal{P}$ to execute the same path as with $\mathcal{I}$. These constraints are computed symbolically based on the variables' symbolic values. For example, when executing an assignment statement `y := x + 1` where the variable `x` has the concrete value of 1 and symbolic value of x_0, DSE obtains the concrete value of `y` to be 2 and its symbolic value as $x_0 + 1$. When DSE encounters a branch statement, it selects a branch according to the branch condition's concrete value and collects the corresponding symbolic condition. Following the above example, if DSE encounters a branch statement $\mathcal{S}_b$ with the branch condition `y > 0` right after the assignment statement, DSE will execute the *true* branch because y's concrete value is 2, and the true branch's symbolic condition, *i.e.*, $x_0 + 1 > 0$, will be collected. When the concrete execution with the input $\mathcal{I}$ terminates, the conjunction of the collected symbolic conditions forms the input constraint, called *path condition* (*PC*), which should be satisfied for the inputs resulting in the same program path as $\mathcal{I}$. This procedure of executing $\mathcal{P}$ concretely and symbolically with an input at the same time is called *concolic execution*. Next, DSE will generate a new *PC* based on the *PCs* of the terminated concolic executions, and the constraint solving of the new *PC* will derive a new input with which the $\mathcal{P}$'s execution will cover a new branch of $\mathcal{P}$. For instance, if DSE wants to explore $\mathcal{S}_b$'s *false* branch, it generates $C_{prefix} \wedge x_0 + 1 \leq 0$ where C_{prefix} is the conjunction of the symbolic conditions collected before executing $\mathcal{S}_b$. If there are no symbolic conditions, C_{prefix} is $true$. Then, the inputs generated by solving $C_{prefix} \wedge x_0 + 1 \leq 0$ will make $\mathcal{P}$ execute $\mathcal{S}_b$'s *false* branch. In this way, DSE can efficiently and systematically explore $\mathcal{P}$'s path space by many concolic executions.

DSE struggles when analyzing programs with complex input formats, which widely exist, *e.g.*, document processing applications and compilers. These programs need to *parse* the input and then process the information after the parsing succeeds. Usually, we can express complex input formats in a formal grammar [Hopcroft et al. 2007; Linz 2006], and the parsing procedure with respect to the grammar often contains lexical and syntactic analysis stages. Lexical analysis (called *tokenization*) checks whether the input characters can be grouped into valid tokens with respect to the lexical rules and generates a token sequence. Then, the syntactic analysis checks whether the token sequence conforms to syntactic rules. When DSE analyzes the programs with complex

input formats, DSE *frequently produces invalid inputs* that violate either lexical or syntactic rules, making it difficult to reach the program parts after parsing. Figure 1 shows a simplified calculator program to illustrate the problem, where *numbers are restricted to a single digit and only addition and subtraction are supported.* In this example, the `getNextToken` function performs tokenization, while `parseTerm` performs syntactic analysis, and both functions filter out invalid inputs by raising exceptions. When analyzing this program, DSE generates many invalid inputs, such as `"x+1"` (lexical error due to the invalid token `"x"`) and `"1++"` (syntactic error due to the `"+"`). When the initial input is `"1+1+1+1+1"` and every character of the input has a symbolic value, DSE finds 16 valid inputs and 169 invalid ones. When we extend the program to enable parentheses-related syntax (*e.g.*, `"(1)+1"` and `"(1+1)"`), DSE produces 197 valid inputs and 5210 invalid ones with the same initial input, and the ratio of invalid inputs further increases. The proportion of valid inputs decreases from 8.65% to 3.64%. As the input formats become more complex, DSE tends to struggle with passing the parsing part of the program, rarely reaching the functional code of the program, *e.g.*, the `execute` function in Figure 1. For example, *in our evaluation, DSE fails to find any new valid input for the parser program,* `Java`*, in one hour.* This problem dooms DSE's effectiveness and efficiency for analyzing programs with complex input formats.

There are two categories of existing work addressing the problem. One category utilizes a formal grammar of input format to improve the effectiveness and efficiency of DSE [Godefroid et al. 2008; Majumdar and Xu 2007]. However, the formal input grammar is usually not available [Bastani et al. 2017]. Input grammars often reside the minds of software developers or described in natural language, which are seldom in a machine-readable format. Manually creating the formal input grammar is also labor-intensive [Godefroid et al. 2008]. For example, the format document of Java *class* file has 63 pages [Oracle 2011], and the document of TCP/IP network protocol has 37 pages on package format [RFC 1989]. Furthermore, grammars may be incomplete or closed-source. On the other hand, there also exists work [Pan et al. 2021] that employs input abstraction during DSE *without* the need for a formal input grammar. With input grammar agnostic, token symbolization-based symbolic execution [Pan et al. 2021] (Gadse) extracts token summaries to produce only valid tokens, which ensures the passing of lexical analysis, thereby improving the efficiency of DSE. However, Gadse still struggles in passing syntactic analysis, especially for programs with complex syntactic rules of input formats. For example, for the parser `Java` mentioned above, Gadse still fails to find any new valid inputs within 1 hour. Overall, without given grammars, improving DSE's efficiency for programs with complex input requirements remains challenging.

Our key insight is that the effectiveness and efficiency of symbolic execution can be further enhanced if it incorporates more detailed grammatical structure information (*e.g.*, syntax rules) during analysis. Recent work on grammar synthesis [Arefin et al. 2024; Bastani et al. 2017] can automatically learn grammars by generalizing valid inputs. Hence, we can leverage grammar synthesis to automatically learn more detailed input grammars during symbolic execution. Inspired by this idea, we propose an iterative framework, called Lase, which synergizes symbolic execution and grammar synthesis in a feedback loop. Lase synthesizes an input grammar from the valid inputs generated during symbolic execution, and then derives new valid inputs from the learned grammar to improve symbolic execution, increasing the likelihood of passing the parsing phase. Meanwhile, as symbolic execution proceeds, it discovers new valid inputs, which in turn trigger another round of grammar synthesis and input generation. Hence, the input grammar can be incrementally expanded and refined, leading to further improvements in symbolic execution.

Inside Lase, the main technical challenge is synthesizing better (*i.e.*, more precise and complete) grammar for symbolic execution with an acceptable synthesis cost, since a better grammar is more likely to generate higher-quality inputs to improve symbolic execution. As a token sequence comprises many character-level inputs, performing grammar synthesis at the token level can reduce

the search space during generalization, thereby lowering the cost of grammar synthesis. Therefore, we propose to use a token-level symbolic execution [Pan et al. 2021] (Gadse) to improve the input representation by generating token sequences instead of character-level inputs. Besides, Gadse itself can also improve the effectiveness of symbolic execution. For the token-level inputs generated by Gadse, we propose a token-level grammar synthesis method and a new generalization operation to improve the quality of the synthesized input grammar. Additionally, to enhance the efficiency of symbolic execution in generating representative inputs, we propose a new search strategy for symbolic execution that covers more grammar rules more quickly.

We have implemented Lase for Java programs on a JPF-based dynamic symbolic execution engine [Anand et al. 2007] and conducted the experiments on 15 open-source Java programs with 10 different input grammars. Lase outperforms the baseline regarding valid inputs and code coverage. The experimental results demonstrate Lase's effectiveness and efficiency.

In summary, the main contributions of this paper are as follows.

- We propose Lase, a framework that synthesizes grammars online to aid symbolic execution. To improve the effectiveness of the framework, Lase performs symbolic execution and grammar synthesis at the token level.
- We propose a token-level grammar synthesis method for the token sequence inputs generated by the token-level symbolic execution. Our synthesis method improves the quality of learned grammars at an acceptable cost.
- We propose a grammar-oriented search strategy for token-based symbolic execution to increase code coverage by covering more grammar rules.
- We have conducted extensive experiments on 15 open-source Java parser programs. The experimental results indicate that Lase improves the effectiveness of symbolic execution in terms of code coverage. Compared to token-based symbolic execution, Lase increases statement coverage of parsing code by 7.66% and 7.23% under DFS and BFS, respectively. Lase achieves a statement coverage increase of 88.17% and 99.22% for functionality code compared with BFS and DFS strategies, respectively. Furthermore, Lase can learn better grammars for 10 out of 14 programs.

2 Illustration

We use a motivating example abstracted from a real-world Java parser to illustrate Lase. Figure 2 shows the abbreviated parsing code implementing the Java grammar in Figure 3, using recursive descent parsing [Aho et al. 1986]. During parsing, the function `getNextToken` examines input characters and returns different token values. Then, based on tokens, `parseProgram` checks the token sequence with respect to the syntax rules in a recursive descent manner.

2.1 Token-based Symbolic Execution

As stated in Section 1, we adopt grammar-agnostic token-based symbolic execution (Gadse) [Pan et al. 2021] here for analyzing the program $\mathcal{P}$. Gadse first extracts a summary of $\mathcal{P}$'s tokenization code. The summary is a map associating each token value with a character-level path constraint that inputs must satisfy to produce that token. Then, based on the map, Gadse performs DSE on $\mathcal{P}$ by symbolizing token values instead of the input characters, while collecting token-level path constraints. The newly constructed token-level path constraints are solved to generate token sequences. With the help of the previously collected summary, a token sequence can be converted to a character-level input that can drive $\mathcal{P}$ along a different path.

Given the initial input `"class A1{int A1(){}}"` (denoted by $\mathcal{I}_i$) for the program (denoted by $\mathcal{P}$) in Figure 2, we demonstrate Gadse's procedure of analyzing the program. In the first stage, Gadse obtains the tokenization summary of the program $\mathcal{P}$ in a tree style, which is partially displayed in Figure 3. For example, the constraint $\mathcal{TC}_{\texttt{ID}}$ for the identifier token (`T_ID`) is represented as a

```
enum Token{T_EOF=0, T_ID, T_INT /* int */,
  T_LP /* ( */, T_RP /* ) */, T_LBR /* { */,
  T_RBR /* } */, T_SC /* ; */, T_CLS /* class */,
  T_ITF /* interface */, T_ENUM /* enum */ ...};

// Lexical Fragment
Token token;
Token getNextToken() { ... }

// Syntactic Fragment
void parseBlock() throws Exception{
  consume(T_LBR);
  label2:
  while(true){
    token = getNextToken();
    switch(token){
      case RBR: break label2; case ...
      default: throw "Parsing Exception!";}
    parseStmt();
  }
  consume(T_RBR);
}
void parseMethod() throws Exception{
  parseType();consume(T_ID);consume(T_LP);
  parseFormats();consume(T_RP);
  token = getNextToken();
  switch(token){
    case T_LBR: parseBlock();break;
    case T_SC: consume(T_SC);break;
    default: throw "Parsing Exception!";}
}
```

```
void parseClass() throws Exception{
  consume(T_CLS);comsume(T_ID);consume(T_LBR);
  //consume throws error if next token mismatches
  label1:
  while(true){
    token = getNextToken();
    switch(token){
      case T_INT:...:parseMethod();break;
      case T_LBR: parseBlock();break; ...
      default: break label1;}
  }
  consume(T_RBR);
}

void parseProgram() throws Exception{
  label0:
  while(true){
    token = getNextToken();
    switch(token){
      case T_CLS: parseClass();break;
      case T_ITF: parseInterface();break;
      case T_ENUM: parseEnum();break; ...
      case T_SC: parseEmptyTypeDecl();break;
      case T_EOF: break label0;
      default: throw "Parentheses Error!";}
  }
}

int main() throws Exception{
  parseProgram(); return 0;
}
```

Fig. 2. The motivating example's program.

$$
\begin{array}{rcl}
\langle \text{Prog} \rangle & ::= & (\langle \text{Class} \rangle \mid \langle \text{Interface} \rangle \mid \langle \text{Enum} \rangle \mid \text{';'} \mid \ldots)^* \\
\langle \text{Class} \rangle & ::= & \text{'class'}\ \langle \text{ID} \rangle\ \text{'\{'}\ (\langle \text{Method} \rangle \mid \langle \text{Block} \rangle \mid \ldots)^*\ \text{'\}'} \\
\langle \text{Method} \rangle & ::= & \langle \text{Type} \rangle?\ \langle \text{ID} \rangle\ \text{'('}\langle \text{Paras} \rangle?\ \text{')'}\ (\langle \text{Block} \rangle \mid \text{';'}) \\
\langle \text{Block} \rangle & ::= & \text{'\{'}\ \langle \text{Stmt} \rangle^*\ \text{'\}'} \\
\langle \text{Stmt} \rangle & ::= & \langle \text{WhileS} \rangle \mid \langle \text{IfS} \rangle \mid \langle \text{ReturnS} \rangle \mid \langle \text{Block} \rangle \mid \text{';'} \mid \ldots \\
\langle \text{WhileS} \rangle & ::= & \text{'while'}\ \text{'('}\ \langle \text{Expr} \rangle\ \text{')'}\ \langle \text{Stmt} \rangle \\
\langle \text{IfS} \rangle & ::= & \text{'if'}\ \text{'('}\ (\langle \text{Expr} \rangle\ \text{')'}\langle \text{Stmt} \rangle \\
\langle \text{ReturnS} \rangle & ::= & \text{'return'}\ \langle \text{Expr} \rangle?\ \text{';'} \\
\langle \text{Expr} \rangle & ::= & \ldots \\
\langle \text{Type} \rangle & ::= & \text{'int'} \mid \text{'char'} \mid \langle \text{ID} \rangle \mid \ldots \\
\langle \text{ID} \rangle & ::= & \langle \text{Letter} \rangle\langle \text{PartLetter} \rangle^* \\
\langle \text{Letter} \rangle & ::= & \text{'a'} \mid \ldots \mid \text{'z'} \mid \text{'A'} \mid \ldots \mid \text{'Z'} \mid \text{'\_'} \\
\langle \text{PartLetter} \rangle & ::= & \text{'0'} \mid \ldots \mid \text{'9'} \mid \langle \text{Letter} \rangle
\end{array}
$$

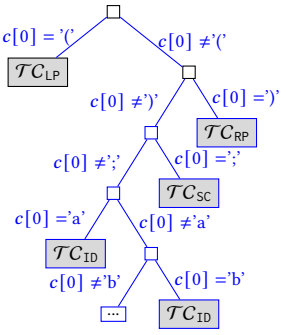


Fig. 3. The motivating example's partial grammar (left) and partial constraint tree for the lexical code (right).

disjunction of the constraints corresponding to all generated inputs of this token, shown as follows.

$$(c[0] \neq \text{'('} \wedge \ldots \wedge c[0] = \text{'a'}) \vee (\ldots \wedge c[0] = \text{'b'}) \vee \ldots \vee (\ldots \wedge c[0] = \text{'a'} \wedge \ldots \wedge c[1] = \text{'a'}) \vee \ldots \quad (1)$$

Then, in the second stage, GADSE symbolizes the initial input $\mathcal{I}_i$ at the token level and collects the corresponding path constraint. Executing $\mathcal{P}$ on $\mathcal{I}_i$ yields the token sequence $\langle$T_CLS, T_ID, T_LBR, T_INT, T_ID, T_LP, T_RP, T_LBR, T_RBR, T_RBR$\rangle$, leading to the following path constraint, where $\mathbf{T}[i]$ denotes the ith symbolic token value. The complete constraint is omitted for brevity and only the critical part is presented here.

$$\begin{aligned} &\ldots \wedge \mathbf{T}[0] = \text{T\_CLS} \wedge \mathbf{T}[1] = \text{T\_ID} \wedge \mathbf{T}[2] = \text{T\_LBR} \wedge \mathbf{T}[3] = \text{T\_INT} \wedge \mathbf{T}[4] = \text{T\_ID} \\ &\wedge \mathbf{T}[5] = \text{T\_LP} \wedge \mathbf{T}[6] = \text{T\_RP} \wedge \mathbf{T}[7] = \text{T\_LBR} \wedge \mathbf{T}[8] = \text{T\_RBR} \wedge \mathbf{T}[9] = \text{T\_RBR} \end{aligned} \quad (2)$$

Suppose that GADSE employs the depth-first search (DFS) strategy. The new token-level path constraint after negating the last condition is as follows.

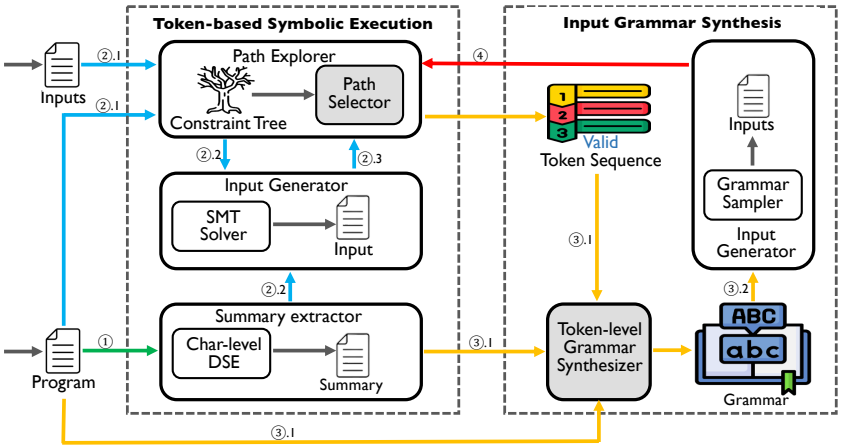


Fig. 4. The basic framework of Lase, where the labels of arrow lines indicate the related phases or sub-phases.

$$\ldots \wedge \mathbf{T}[0] = \texttt{T\_CLS} \wedge \mathbf{T}[1] = \texttt{T\_ID} \wedge \mathbf{T}[2] = \texttt{T\_LBR} \wedge \mathbf{T}[3] = \texttt{T\_INT} \wedge \mathbf{T}[4] = \texttt{T\_ID} \\ \wedge \mathbf{T}[5] = \texttt{T\_LP} \wedge \mathbf{T}[6] = \texttt{T\_RP} \wedge \mathbf{T}[7] = \texttt{T\_LBR} \wedge \mathbf{T}[8] = \texttt{T\_RBR} \wedge \mathbf{T}[9] \neq \texttt{T\_RBR} \tag{3}$$

Then, Gadse solves the new path constraint to generate a token sequence. Suppose that the following token sequence is generated.

$$\langle \texttt{T\_CLS}, \texttt{T\_ID}, \texttt{T\_LBR}, \texttt{T\_INT}, \texttt{T\_ID}, \texttt{T\_LP}, \texttt{T\_RP}, \texttt{T\_LBR}, \texttt{T\_RBR}, \texttt{T\_LBR} \rangle \tag{4}$$

This token sequence can be converted to a new character-level input based on the summary. If the new input `"class A1{int A1(){}{"` is given, it cannot be parsed by the program. During the second iteration of the while loop in `parseClass`, `parseBlock` throws an exception upon encountering `T_EOF` after the last `'{'`.

Gadse then continues to construct new token-level path constraints for generating new inputs. In this way, the inputs generated by Gadse always pass lexical analysis (*i.e.*, tokenization), since they are derived solely from legal tokens. It excludes inputs such as `"alass A1{int A1(){}}"`. Hence, compared with pure symbolic execution, Gadse improves efficiency. However, when the input grammar is complex and the number of syntax rules increases, Gadse still struggles to generate valid inputs because the token sequences may not be syntactically valid, such as the previously generated input `"class A1{int A1(){}{"`. *In our evaluation, Gadse fails to find any new valid input for the* Java *parser of Figure 2's example program in one hour, which makes it challenging to analyze the program's functional aspects and ultimately reducing efficiency.*

2.2 Grammar Synthesis Aided SE

2.2.1 Basic Procedure. Figure 4 shows Lase's basic framework, which takes a program and initial inputs as input. The whole procedure can be divided into 4 phases, token summary extraction (①), token-level symbolic execution (②), grammar synthesis (③), and valid input-guided symbolic execution (④). We first extract summaries from the tokenization code (①), which will be used to instantiate token sequences (②.2) and compute each character's bounds for grammar synthesis (③.1). Then, we symbolize the initial inputs at the token level to do the symbolic execution of the program (②.1). Inside the path explorer, we maintain a constraint tree on which the path selector picks up a new branch and constructs a new constraint (②.2) to generate a new input (②.3). When the path explorer finds a valid input, we feed it to the grammar synthesizer to generate an input grammar online (③.1). The synthesis is carried out at the token level, and takes the program and

its summary as input for checking the grammar candidates during generalization. Whenever a new grammar is synthesized, a set of "valid" inputs with respect to it will be generated (③.2) and returned to the path explorer to guide symbolic execution (④). Since these new inputs tend to pass the grammar checking phase and help analyze the functional code, symbolic execution prioritizes these inputs. After exploring the inputs generated by the synthesized grammar, the path explorer continues with the unexplored paths under our new search strategy. Once the path explorer discovers a new valid input, it collaborates with grammar synthesis to find an opportunity to explore more diverse paths based on the newly generated valid inputs. In this iterative way (②, ③ and ④), the symbolic execution and the online grammar synthesis can be carried out to benefit each other. Online synthesized grammar improves symbolic execution's effectiveness and efficiency. Symbolic execution also gradually improves the synthesized grammar's completeness.

2.2.2 Demonstration. In Lase, for the example program $\mathcal{P}$ and $\mathcal{I}_i$: `"class A1{int A1(){}}"`, the token sequence ⟨T_CLS, T_ID, T_LBR, T_INT, T_ID, T_LP, T_RP, T_LBR, T_RBR, T_RBR⟩ and the token summaries will be fed to the token-level grammar synthesizer to learn an input grammar.

Grammar synthesis. Glade [Bastani et al. 2017] is a SOTA *black-box* character-level method and assumes that the program can be executed to judge whether an input is valid. Glade generalizes the valid inputs in two steps: 1) recursively divides input strings at the **character** level and leverages generalization operations to expand them to regular expressions; 2) enlarges the regular expressions to a context-free grammar (CFG) by merging the non-terminals in the regular expression grammar.

The generalization process in the first step is a sequence of regular expressions $\langle L_0, ..., L_n \rangle$, where L_0 only contains valid input strings, and $L_i \subseteq L_{i+1}$ for each $0 \leq i \leq n-1$. Each generalization step tries to enlarge L_i and avoids over-generalization simultaneously. There are two generalization operations: Repetition and Alternation, represented as $[]_{\text{rep}}$ and $[]_{\text{alt}}$, respectively. Repetition $\alpha_1([\alpha_2]_{\text{alt}})^*[\alpha_3]_{\text{rep}}$ tries to partition L_i into three parts and repeats the second part, where $L_i = \alpha_1\alpha_2\alpha_3$, and α_2 is surrounded by $[]_{\text{alt}}$, indicating α_2 can be generalized with Alternation later, similar to α_3. Alternation $[\alpha_1]_{\text{rep}} + [\alpha_2]_{\text{alt}}$ attempts to partition L_i into two interchangeable parts, where $L_i = \alpha_1\alpha_2$ and we can recursively apply Repetition and Alternation to the first and second parts, respectively. As multiple divisions may exist and some are illegal, Glade checks whether the generalization step is correct by deriving a few strings from the new regular expression and checking whether the program can accept them. For Repetition, $\alpha_1\alpha_3$ and $\alpha_1\alpha_2\alpha_2\alpha_3$ are verified, while α_1 and α_2 are evaluated for Alternation. If the strings generated from the candidate grammar are accepted by the program, Glade selects the candidate as L_{i+1}.

In the second stage, Glade learns the recursive attribute in the final regular expressions L_n and enlarges L_n to a context-free grammar (CFG). Glade acquires the recursive property by merging those non-terminals in the grammar. For instance, if in S ::=`'('` S_1 `')'`, S and S_1 are mergeable, we can obtain S ::=`'('` S `')'` where S can recursively derive S.

This character-based approach may increase the time required to verify candidates' validity. In the first stage of regular expression generalization, all the partitions are manipulated at the character level, which means the characters from different tokens can be divided into one group. For example, the initial input `"class A1{int A1(){}}"` can derive many candidates using a character-level synthesis method, such as $(\texttt{[class A1\{int A1()\{\}\}]}_{\text{alt}})^*$ and $\texttt{cla}(\texttt{[ss A1]}_{\text{alt}})^*\texttt{[\{int A1()\{\}\}]}_{\text{rep}}$. However, most divisions inside tokens are invalid. Candidate $\texttt{cla}(\texttt{[ss A1]}_{\text{alt}})^*\texttt{[\{int A1()\{\}\}]}_{\text{rep}}$ can yield samples such as `"class A1ss A1{int A1(){}}"` and `"cla{int A1(){}}"`, which are invalid for the program.

Token-level grammar synthesis. Our token-level synthesis method avoids partitioning strings within tokens, thereby reducing invalid candidates. Furthermore, since Glade only checks a subset of strings in the candidate grammar, this process is imprecise and may lead to over-generalization.

Character-level partitioning can exacerbate this problem, as more incorrect partitions need to be checked. *Hence, synthesis at the character level may introduce both **efficiency** and **precision** issues.*

Token-level grammar synthesis generalizes valid inputs in three steps: 1) the synthesizer learns **token-level** regular expressions from the valid token sequences; 2) the regular expressions will then be enlarged to a **token-level** context-free grammar (denoted by $\mathcal{G}_t$); 3) Finally, based on the **token summaries**, $\mathcal{G}_t$ is generalized to a character-level CFG $\mathcal{G}_c$.

Considering the input "`class A1{int A1(){}}`", we derive the valid token sequence from GADSE, *i.e.*, ⟨T_CLS, T_ID, T_LBR, T_INT, T_ID, T_LP, T_RP, T_LBR, T_RBR, T_RBR⟩. By applying the repetition operation $[]_{\text{rep}}$ to this 10-token sequence, we obtain 66 candidates, rather than the 231 for the 20-character input string. Then, we check each candidate by sampling. For example, if we check the partition: $\{\alpha_1 = \varepsilon, \alpha_2 = \langle$T_CLS, T_ID, T_LBR, T_INT, T_ID, T_LP, T_RP, T_LBR, T_RBR, T_RBR$\rangle, \alpha_3 = \varepsilon\}$, we derive two token sequences, *i.e.*, ⟨⟩ and ⟨T_CLS, T_ID, T_LBR, T_INT, T_ID, T_LP, T_RP, T_LBR, T_RBR, T_RBR, T_CLS, T_ID, T_LBR, T_INT, T_ID, T_LP, T_RP, T_LBR, T_RBR, T_RBR⟩, repeating α_2 zero and two times. Then, we convert them to character-level inputs with the token value constraints and constants (collected in GADSE) to verify the validity. These sequences are transformed into "" and "`class A1{int A1(){}}class A1{int A1(){}}`", respectively. Both of them can be accepted, so we consider this partition as a legal candidate.

For each iteration of generalization in the first stage, we select the first legal candidate to be further generalized. The whole procedure of regular expression generalization is as follows.

$$\begin{aligned}
& [\texttt{CLS ID LBR INT ID LP RP LBR RBR RBR}]_{\text{rep}} \\
\Rightarrow\ & ([\texttt{CLS ID LBR INT ID LP RP LBR RBR RBR}]_{\text{alt}})^* \\
\Rightarrow\ & ([\texttt{CLS ID LBR INT ID LP RP LBR RBR RBR}]_{\text{rep}})^* \\
\Rightarrow\ & \ldots \Rightarrow (\texttt{CLS ID LBR } ([\texttt{INT ID LP RP LBR RBR}]_{\text{rep}})^* \texttt{ RBR})^* \\
\Rightarrow\ & (\texttt{CLS ID LBR } (([\texttt{INT ID LP RP}]_{\text{alt}})?[\texttt{LBR RBR}]_{\text{rep}})^* \texttt{ RBR})^* \\
\Rightarrow\ & \ldots \Rightarrow (\texttt{CLS ID LBR } ((\texttt{INT}?\texttt{ ID LP RP})?\texttt{ LBR RBR})^* \texttt{ RBR})^*
\end{aligned} \tag{5}$$

Besides token-level synthesis, we also introduce the Exchange ($[]_{\text{exh}}$) and question mark (?) operators to enhance the first stage, which will be clarified in the next section.

In the second stage, we transform the final regular expression into an equivalent context-free grammar (CFG) given below, where the "?" indicates the sequence occurs zero or one times.

$$\begin{array}{llllllllllll}
S & ::= & S_1^* & S_1 & ::= & S_2\ S_3^*\ S_4 & S_3 & ::= & S_5?\ S_6 & S_5 & ::= & S_7?\ S_8 \\
S_2 & ::= & \texttt{CLS ID LBR} & S_4 & ::= & \texttt{RBR} & S_6 & ::= & \texttt{LBR RBR} & S_7 & ::= & \texttt{INT} \\
S_8 & ::= & \texttt{ID LP RP} & & & & & & & & &
\end{array} \tag{6}$$

Next, we merge the non-terminals to further generalize the grammar and identify recursive properties. We determine whether two non-terminals are mergeable by replacing one with the other in the original strings. For example, when we merge S_1 with S_3, we generate a sample set, {"`int A1(){}`", "`class A1{class A1{int A1(){}}}`"}. The former string is illegal, so these two non-terminals cannot be merged. In this case, none of the non-terminals can be merged, and thus the grammar obtained in the second stage remains identical to that of the first stage.

In the third stage, we apply regular expression synthesis to each token t and generalize its potential character values. Suppose that the grammar of token "`ID`" synthesized from "`A1`" is `ID ::= A(1)`*. We verify the potential character values by putting them into the inputs where t appears and checking the validity of newly constructed inputs. In this step, we employ SMT optimization theory [Li et al. 2014] to calculate each character's upper and lower bounds and then generalize the token's characters to range between these bounds. For example, from the token value "`ID`"'s constraint, we derive that its first character lies within the range from 65 ('`A`') to 122 ('`z`'), while the remaining characters range from 48 ('`0`') to 122. When testing

⟨EXPR⟩ ::= ⟨AEXPR⟩ | ⟨MEXPR⟩ | ⟨RSEXPR⟩ | ⟨BOEXPR⟩ | ⟨BAEXPR⟩
⟨AEXPR⟩ ::= '+' ⟨TERM⟩ ⟨TERM⟩
⟨MEXPR⟩ ::= '*' ⟨TERM⟩ ⟨TERM⟩
⟨RSEXPR⟩ ::= '>>' ⟨TERM⟩ ⟨TERM⟩
⟨TERM⟩ ::= ⟨NUM⟩ | ⟨ID⟩

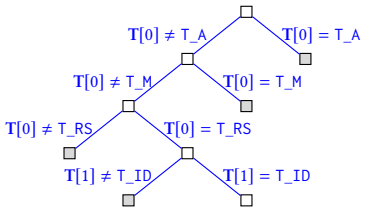


Fig. 5. The example for demonstrating search strategy.

whether `'a'` can replace `'A'`, we generate `"class a1{int a1(){}}"`. Note that only characters within the computed bounds need to be considered, which improves the precision of the grammar. Consequently, the grammar for `"ID"` can be expressed as

$$\texttt{ID} := (\texttt{'a'}|...|\texttt{'z'}|\texttt{'A'}|...|\texttt{'Z'}|\texttt{'\_'})(\texttt{'0'}|...|\texttt{'9'}|\texttt{'a'}|...|\texttt{'z'}|\texttt{'A'}|...|\texttt{'Z'}|\texttt{'\_'})^* \tag{7}$$

After merging the grammar of tokens into that in 6, we obtain the final context-free grammar $\mathcal{G}_c$.

Compared with character-level synthesis, our token-level grammar synthesis method needs *fewer steps* to synthesize the grammar, which improves the efficiency and precision of the synthesis.

Grammar Synthesis Aided Symbolic Execution. Lase starts with token-level symbolic execution. When a valid input is generated through DSE, it proceeds to grammar synthesis. From the initial input `"class A1{int A1(){}}"`, we synthesize a grammar $\mathcal{G}_c$ and sample additional valid inputs from it. Specifically, we generate 3 token sequences and translate them into strings, including `"class A1{}class A1{{}}"`, `"class A1{{}}class A1{}"`, and `"class A1{{}{}{}}"`. All of them are fed back to Gadse for symbolic execution. We prioritize them because they are more likely to be accepted by the parser. Afterward, symbolic execution proceeds along the unexplored paths. Whenever symbolic execution discovers a new valid input, grammar synthesis is invoked again to extend the grammar. Suppose that after several iterations, Gadse produces `"; class A1{A1(){}}"`. From this input, we can infer an improved grammar that includes the empty type declaration rule (`';'`). This synthesis allows us to derive additional valid inputs (*e.g.*, `";;;;;;class A1{}"`), which are then returned to Gadse while the old inputs are discarded. In this manner, symbolic execution prioritizes valid inputs while continuing to explore new ones, thereby progressively completing the synthesized grammar. *On the benchmark* `Java`*, Lase can generate more than 280 valid inputs and synthesize grammars 3 times within 1 hour, which indicates the effectiveness of our framework.*

The advantages of the synergy between symbolic execution and grammar synthesis are two-fold. First, inputs from the synthesized grammar are likely valid, reducing constraint solving. Besides, the paths corresponding to these inputs are prioritized, guiding symbolic execution to explore deeper parts of the program beyond grammar checking. Second, Gadse separates inputs into token sequences, enabling token-level synthesis and avoiding redundant partitioning and candidate checks. In addition, Gadse can provide more well-distributed inputs to grammar synthesis owing to token-level symbolization. The synergy between symbolic execution and grammar synthesis is the key to enabling these advantages.

2.3 Grammar-Oriented Search Strategy

During symbolic execution, we observed that certain grammar rules are difficult to cover, which may reduce the efficiency of symbolic execution. In the motivating example (Figure 3), the production ⟨Prog⟩ := `';'`* is particularly hard to trigger within a limited time. To clarify this issue, we present a detailed analysis using the expression grammar shown in Figure 5. The expression grammar is typically defined within the Java grammar, and here we modify it to adopt a prefix expression format. The right side of the figure shows the symbolic execution tree obtained after executing

Algorithm 1: Framework of Online Grammar Synthesis Aided Token-based SE

```
   LASE(P, I_0)
   Data: P is a program, and I_0 is the set of initial inputs.
1  begin
2      Done, I, I_g ← false, I_0, ∅
3      TC, TS ← ∅, ∅                          ▷ Token constraints and character values
4      (Σ, TC, TS) ← GenTokenSummary(P, M_t)   ▷ Get the token summary
5      while ¬Done do
6          I_g ← I_g ∪ I                       ▷ Record the global input set
7          (α, TS) ← TokenDSE(P, Σ, TC, TS, I) ▷ α is a valid token sequence
8          G_t ← TGrammarSynthesis(P, α, Σ, TC, TS)   ▷ Synthesize a CFG from α
9          G_c ←TokenGeneralization(P, G_t, α, TC, TS) ▷ Generalize the tokens in G_t
10         G ← G ⊎ G_c                         ▷ Enlarge the synthesized CFG G
11         I ← GenerateInput(G) \ I_g          ▷ Generate inputs from the CFG G
12     end
13 end
```

the initial input `">> a 1"`, where the gray leaves denote unexplored branches. For brevity, only the portion related to the first two tokens is shown. With the DFS strategy, we will first flip the branch $\mathbf{T}[2]$ = `T_NUM` for the third token since it is the deepest unexplored branch. According to the constraint tree, we explore $\mathbf{T}[0] \neq$ `T_RS` until we have finished the ⟨RSEXPR⟩. DFS strategy tends to explore inputs sharing the same prefix for a long time. Furthermore, exploring ⟨AEXPR⟩ is hard because it has the shallowest depth. For the BFS strategy, the branches under $\mathbf{T}[0]$ = `T_A` or $\mathbf{T}[0] \neq$ `T_A` are selected in a similar probability. ⟨AEXPR⟩ will be explored with a higher probability than ⟨BAEXPR⟩. We can easily observe that DFS and BFS strategies exhibit biases towards different grammar rules, leading to unbalanced grammar exploration.

To address the above issue, our insight is that *the early tokens often determine the syntax rules used for parsing*. If we can generate the token sequences with more different token prefixes, the probability of covering more syntax rules is higher. Hence, we select the unexplored branches with smaller token indices to construct constraints in priority, making it easier to generate different token prefixes. For the example, we can traverse all grammar rules efficiently if we prioritize branches with a smaller index, *i.e.*, $\mathbf{T}[0]$. So, we first choose $\mathbf{T}[0]$ = `T_A`, followed by $\mathbf{T}[0]$ = `T_M` and then $\mathbf{T}[0] \neq$ `T_RS`. After exploring all the branches of the first token, we will choose branches of the second one and then the third. This way, we can quickly cover all the grammar rules. For the above example, with the initial input `">> a 1"`, we need 19 constraint solving attempts to cover all grammar rules under the BFS strategy, and the number is 29 under the DFS strategy. *With the help of the proposed search strategy, LASE needs only 5 solver calls.*

3 Method

Algorithm 1 shows our synergy framework, which takes a program $\mathcal{P}$ and a set of initial inputs as input. Our framework first extracts the summary of the tokenization code in $\mathcal{P}$ (Line 4), where M_t is the tokenization function. The summary contains token values (Σ), character-level constraints of each token value ($\mathcal{TC}$), and their corresponding string values ($\mathcal{TS}$). The summary will be used later in token-based symbolic execution and token-level grammar synthesis.

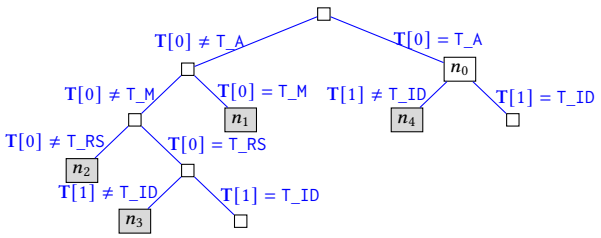
Then, in the while loop, we first perform token-level symbolic execution (TokenDSE at Line 7) with the initial inputs and the token summary. When the symbolic execution finds a valid input,

the token sequence α of the input will be fed immediately to the grammar synthesis component to learn a CFG $\mathcal{G}_t$ (TGrammarSynthesis at Line 8). Then, the grammar $\mathcal{G}_t$ is generalized and refined by the token summary to get the complete grammar $\mathcal{G}_c$ (TokenGeneralization at Line 9). Additionally, we maintain a global variable $\mathcal{G}$ to store the currently acquired input grammar, which expands in each synthesis iteration(Line 10). Symbolic execution leverages $\mathcal{G}$ to determine whether online grammar synthesis is necessary. Specifically, it checks whether the newly discovered valid input can be accepted by $\mathcal{G}$. If the input is not accepted, this indicates that $\mathcal{G}$ should be enlarged by learning and merging the new grammar. From $\mathcal{G}$, we can quickly generate a large number of "valid" program inputs, thereby enhancing both the efficiency and effectiveness of symbolic execution. This iterative process continues until the resource limit is reached, which we omit here for brevity.

3.1 Grammar-Oriented Search Strategy

We integrate our new search strategy into token-level symbolic execution (Line 7). In this process, symbolic execution follows a path using concrete input while collecting **token-level** constraints along the way with token-based symbolic input. These constraints are stored in the form of a binary constraint tree, as depicted in Figures 4 and 5. In the constraint tree, each explored branch is inserted as a child node of the previously covered branch, while a sibling node is created by negating the corresponding constraint to preserve the unexplored branch. In the BFS strategy, shallower unexplored nodes are given higher priority, while in the DFS strategy, deeper nodes take precedence over shallower ones. Symbolic execution constructs a new path constraint by selecting the unexplored branch with the highest priority, tracing back to the root node, and combining the path constraints using conjunction. Finally, symbolic execution invokes an SMT solver to generate a new input with respect to the constraint and uses it to advance to a different path.

In symbolic execution, the search strategy plays a crucial role in path selection. We propose a grammar-oriented search strategy for TokenDSE (Line 7), which *prioritizes unexplored branches with smaller token indices*. Specifically, we select the unexplored branch with the *smallest token index*, where the index represents the position of the token in the sequence. If multiple branches share the same index, we place the shallowest one at the front. The key insight behind this approach is that earlier tokens often determine the syntax rules used during parsing, so prioritizing tokens with smaller indices allows us to cover more grammar productions more efficiently. Experimental results in Section 4 demonstrate the effectiveness of our method.



Example. Based on the example in Figure 5 with the initial input ">> a 1", our method sorts unexplored branches as per their priority: $\{n_0 : \mathbf{T}[0] = \texttt{T\_A}, n_1 : \mathbf{T}[0] = \texttt{T\_M}, n_2 : \mathbf{T}[0] \neq \texttt{T\_RS}, n_3 : \mathbf{T}[1] \neq \texttt{T\_ID}\}$, where we show the nodes corresponding to the first two tokens for brevity. We first choose the unexplored node n_0 and construct the path constraint $\mathbf{T}[0] = \texttt{T\_A}$. Assuming that the solution obtained is "+ a 1", we then collect the constraint $\mathbf{T}[0] = \texttt{T\_A} \wedge \mathbf{T}[1] = \texttt{T\_ID} \wedge \mathbf{T}[2] \neq \texttt{T\_ID} \wedge \mathbf{T}[2] = \texttt{T\_NUM}$ and update the constraint tree as shown in the left figure. Here, the node n_4 is the false branch of the second condition in the above constraint and is inserted as a child of n_0. Note that node n_4 is at the second level of the constraint tree but has a lower priority than n_2 at the third level, as we prioritize constraints with smaller token indices. After covering all of the unexplored branches of $\mathbf{T}[0]$, the path selector proceeds to choose n_3 to construct a new path constraint for solving.

Algorithm 2: Token Based Grammar Synthesis

```
TGrammarSynthesis(P, α_t, Σ, TC, TS)
Data: P is the program, α_t is the token sequence, Σ is the alphabet of token set, TC is the
      token constraint map, TS is the character-level seed map of each token.
1  begin
2      L_cur ← {[α_t]_rep}
3      while ¬done do
4          M ← ConstructCandidates(L_cur)        ▷ Construct the candidates for generalization
5          for L̃ ∈ M do
6              S_t ← SampleTokenSeqs(L_cur, L̃)   ▷ Construct L̃'s token-level words for checking
7              S ← SolveAndInstantiate(S_t, TC, TS)   ▷ Token-level words to input strings
8              if CheckCandidate(S, P) then
9                  L_cur ← L̃
10                 break
11             end
12         end
13     end
14     return L_cur
15 end
```

3.2 Token Based Grammar Synthesis

Algorithm 2 shows the details of the token-based grammar learning procedure. The algorithm consists of *two phases*: the regular expression learning phase and the context-free grammar learning phase. *Algorithm 2 does not differentiate these two phases because both can be regarded as a generalize-and-check loop.* The function ConstructCandidates (Line 4) generalizes the learned grammar, and CheckCandidate(Line 8) checks whether the generalization is valid in the order of candidates' priority. In each while loop, the learned grammar contains the one learned in the last loop. Note that all the grammars learned in Algorithm 2 are based on tokens. In other words, the grammar alphabet consists of tokens rather than characters.

Regular Expression Generalization. This phase generalizes token sequences to regular expressions. Initially, the learned grammar $\mathcal{L}_{cur}$ is set as the regular expression equivalent to the given input token sequence $[\alpha_t]_{\text{rep}}$, where []rep indicates it can be generalized with the Repetition Operation. In each while loop, the function ConstructCandidates (Line 4) generates candidates using the following operations.

- **Repetition**. Repetition aims to identify the largest possible segment of the input that can occur multiple times, which is a common pattern in regular expressions. In expression $P[\alpha_t]_{\text{rep}}Q$, for *each* decomposition $\alpha_t = \alpha_1\alpha_2\alpha_3$ such that $\alpha_2 \neq \varepsilon$, generate candidates:

$$P\alpha_1([\alpha_2]_{\text{alt}})^*[\alpha_3]_{\text{rep}}Q \qquad P\alpha_1([\alpha_2]_{\text{exh}})[\alpha_3]_{\text{rep}}Q$$

- **Alternation**. Alternation detects whether two segments are interchangeable. In expression $P[\alpha_t]_{\text{alt}}Q$, for each decomposition $\alpha_t = \alpha_1\alpha_2$ such that $\alpha_i \neq \varepsilon$, generate the following candidate. Here, the + operator represents a choice between two strings rather than their concatenation to distinguish between string alternation and rule alternation within the meta-grammar.

$$P([\alpha_1]_{\text{rep}} + [\alpha_2]_{\text{alt}})Q$$

- **Exchange**. Exchange attempts to discover two segments surrounding a delimiter that can be interchanged. In expression $P[\alpha_t]_{\text{exh}}Q$, for each decomposition $\alpha_t = \alpha_1\alpha_2\alpha_3$ such that each $\alpha_i \neq \varepsilon$, generate candidate

$$P((([\alpha_1]_{\text{rep}} + [\alpha_3]_{\text{rep}})[\alpha_2]_{\text{rep}})^*([\alpha_1]_{\text{rep}} + [\alpha_3]_{\text{rep}}))Q$$

Compared to Glade, we introduce a new operation, **Exchange**, which generalizes the expression by merging the first and third parts in case they are exchangeable. Our motivation arises from the fact that, in many complex grammar definitions, there exists a production of the form $\langle \mathsf{T}_1 \rangle := \langle \mathsf{T}_2 \rangle(\langle \mathsf{T}_3 \rangle\langle \mathsf{T}_2 \rangle)^*$, where $\langle \mathsf{T}_2 \rangle$ may take two alternative forms. For instance, in `"a+1"`, $\langle \mathsf{T}_2 \rangle$ can be either an identifier or a number. More specifically, **Exchange** is designed for cases where the elements before and after a delimiter can be interchanged. This pattern is prevalent in complex grammars, such as `"int a , bool b"` in C++ parameter declarations and `' "name":"Tom" ; "Age":7 '` in JSON. **Repetition** and **Alternation** impose limitations on generalization, as they only account for continuous sub-expressions, making them insufficient for capturing the full flexibility of such structures. **Exchange** improves Glade's generalization ability on a single input while ensuring that the generalized expression is still regular. In addition to the **Exchange**, we further enhance the generalization capability for a single input by introducing a question mark operator (?). This operator is used to describe a situation where a subsequence may either appear once or disappear, when neither the **Repetition** nor the **Exchange** is applicable, with the specific case elided for brevity.

We follow Glade to prioritize shorter α_1 and longer α_2 for **Repetition** and shorter α_1 for **Alternation**. And we prefer longer α_1 and α_3 in terms of Exchange. In addition, for **Repetition** with a specific partition, we prioritize $P\alpha_1([\alpha_2]_{\text{alt}})^*[\alpha_3]_{\text{rep}}Q$ because **Exchange** Operation will lose the opportunity for alternation between $\alpha_1'\alpha_2'$ and α_3', where $\alpha_2 = \alpha_1'\alpha_2'\alpha_3'$. Hence, in each step of a regular expression generalization (an iteration of the while loop, in Algorithm 2), we rank the candidates of different partition and generalization operators according to their priority and store them in M.

Next, we need to check the validity of those ranked candidates in M and choose the first legal one, $\tilde{L}$, as the next $\mathcal{L}_{cur}$ to be further generalized. The key idea of the function SampleTokenSeqs (Line 6 in Algorithm 2) is to generate representative members of $\tilde{L} \setminus \mathcal{L}_{cur}$ (denoted by S_t) that can reflect the generalization. For example, if $\mathcal{L}_{cur}$ is $\alpha_1\alpha_2\alpha_3$ and $\tilde{L}$ is $\alpha_1\alpha_2^*\alpha_3$, SampleTokenSeqs derives $\{\alpha_1\alpha_3, \alpha_1\alpha_2\alpha_2\alpha_3\}$. This sample set reflects the direction of generalization, capturing potential variations of the input. The algorithm checks the validity of $\tilde{L}$ by checking whether the token sequences in S_t are valid with respect to $\mathcal{P}$. For **Repetition**, the sequences $\alpha_1\alpha_3$ and $\alpha_1\alpha_2\alpha_2\alpha_3$ are subject to verification, whereas for **Alternation**, α_1 and α_2 are evaluated. Furthermore, in the context of the **Exchange** operation, the input set $\{\alpha_1, \alpha_3, \alpha_1\alpha_2\alpha_1\alpha_2\alpha_3, \alpha_1\alpha_2\alpha_3\alpha_2\alpha_3\}$ is employed as the validation set. Since grammar synthesis operates on token sequences while programs take strings as input, we need to convert the sample token sequences into strings. To achieve this, SolveAndInstantiate (Line 7) instantiates each token using concrete strings observed in the input and denotes the sampled inputs as S. We do not introduce unseen strings, as different token instances exercise the same grammar rule during token-level regular expression synthesis. CheckCandidate(Line 8) submits these inputs to program $\mathcal{P}$ to verify whether $\tilde{L}$ is a valid generalization. The first valid candidate is extended as much as possible while avoiding excessive generalization and is chosen as the next expression to be further enlarged. It is worth noting that there may exist many valid candidates, and selecting the first valid one may overlook other promising directions, potentially hindering further generalization. However, due to the vastness of the search space, we prefer to trade completeness for efficiency. This limitation is mitigated by the token-level approach, as grouping characters into

tokens reduces the search space and increases the likelihood of identifying the most generalized "valid" grammar.

$$T_{\text{rep}} ::= \beta \mid T^*_{\text{alt}} \mid \beta T^*_{\text{alt}} \mid T^*_{\text{alt}} T_{\text{rep}} \mid \beta T^*_{\text{alt}} T_{\text{rep}} \mid T_{\text{exh}} \mid \beta T_{\text{exh}} \mid T_{\text{exh}} T_{\text{rep}} \mid \beta T_{\text{exh}} T_{\text{rep}} \quad \text{(Repetition)}$$

$$T_{\text{alt}} ::= T_{\text{rep}} \mid T_{\text{rep}} + T_{\text{alt}} \quad \text{(Alternation)}$$

$$T_{\text{exh}} ::= (T_{\text{meg}} T_{\text{rep}})^* T_{\text{meg}} \qquad T_{\text{meg}} ::= T_{\text{rep}} + T_{\text{rep}} \quad \text{(Exchange)}$$

The entire process of regular expression generalization follows the above meta-grammar, where $\beta \in \Sigma^* \setminus \{\varepsilon\}$ ranges over all non-empty substrings of the input token sequence α_t. Specifically, an element attached with the Kleene star (*) can appear zero or multiple times, while two elements surrounding the + operator can be interchanged.

We enclose the initial valid token sequence α_t with $[]_{\text{rep}}$, designating it as the start non-terminal T_{rep}, which indicates that it can be generalized using the **Repetition** operation in the next step. Each production rule in the meta-grammar corresponds to a step in the regular expression generalization process (an iteration of the while loop in Algorithm 2). This process continues until the expression consists solely of the meta-grammar's terminal symbols for $\mathcal{L}_{cur}$, meaning that no further regular expression generalization is possible. At this point, the final regular expression becomes eligible for context-free grammar generalization.

Example. Consider the input `"a+1"`. According to the grammar rules ⟨Term⟩ ::= ⟨ID⟩ | ⟨NUM⟩ and ⟨AddExpr⟩ ::= ⟨AddExpr⟩⟨OP⟩⟨Term⟩ | ⟨Term⟩, where ⟨OP⟩ can be either `'+'` or `'-'`, the input is tokenized into the sequence ⟨ID, OP, NUM⟩, abbreviated as ION. We first apply the T_{rep} derivation rule ($[\text{ION}]_{\text{rep}}$), transforming it into $[\text{ION}]_{\text{exh}}$. We sample input set for the candidate as {I, N, IONON, IOION}. It then expands into $(([\text{I}]_{\text{rep}} + [\text{N}]_{\text{rep}})\ [\text{O}]_{\text{rep}})^*([\text{I}]_{\text{rep}} + [\text{N}]_{\text{rep}})$ through the Exchange operation, governed by the rule of T_{exh}, since the sampled set is valid. Each bracketed component ($[\text{I}]_{\text{rep}}$, $[\text{O}]_{\text{rep}}$, $[\text{N}]_{\text{rep}}$) retains the rep non-terminal marker, enabling recursive generalization through subsequent meta-grammar rule applications.

It is worth noting that performing generalization at the character level presents a drawback. For each iteration of the while loop in Algorithm 2 (a generalization step), numerous candidate decompositions of $\mathcal{L}_{cur}$ may need to be considered, leading to increased computational complexity. Our token-based synthesis approach alleviates this issue by operating at the token level, where token sequences are generally shorter than character-level inputs, as a single token can represent multiple characters. Consequently, the number of possible decompositions is significantly reduced, improving synthesis efficiency. Furthermore, since candidate sampling and evaluation are inherently approximate and prone to inaccuracies, reducing the number of candidate checks helps improve the precision of the synthesized grammar.

Context-Free Grammar generalization. The second phase extends the regular expression from the first stage into a context-free grammar through a two-step process. In the first step, the regular expression is converted into an equivalent context-free grammar. Since the generalization in the first stage aligns with the production rules of the meta-grammar, we can directly derive a context-free grammar from these productions. For instance, the transformation $[\text{ION}]_{\text{rep}} \Rightarrow [\text{ION}]_{\text{exh}}$ can yield the production rule $S := S_1$. Similarly, from $[\text{ION}]_{\text{exh}} \Rightarrow (([\text{I}]_{\text{rep}} + [\text{N}]_{\text{rep}})\ [\text{O}]_{\text{rep}})^*([\text{I}]_{\text{rep}} + [\text{N}]_{\text{rep}})$, we derive $S_1 := ((S_2|S_3)S_4)^*(S_2|S_3)$. In this process, we introduce a non-terminal for each subsequence surrounded by an operation marker that can be further generalized.

Next, we generalize the context-free grammar by iteratively merging non-terminals. This generalization process introduces recursion, and the merging operations in this stage monotonically expand the language defined by the grammar. The function ConstructCandidates (Line 4) is responsible for merging different non-terminals in $\mathcal{L}_{cur}$. Next, the function SampleTokenSeqs (Line 6) constructs new token sequences by swapping occurrences of these non-terminals. Subsequently,

SolveAndInstantiate (Line 7) and CheckCandidate (Line 8) transform the token sequences in the sample set and verify whether a given candidate constitutes a valid generalization, respectively. The key idea of this phase is to interchange substrings generated by the two merged non-terminals within their respective contexts and then assess whether the modified string is accepted by program $\mathcal{P}$. *Notably, we introduce a non-terminal for each token, and token-associated non-terminals can also be merged at this stage*, which highlights an advantage of synthesizing grammar at the token level.

Example. For the CFG synthesis stage, we aim to generalize the following grammar.

$$S ::= S_1 \quad S_1 ::= (S_2\ \texttt{0})^*\ S_2 \quad S_2 ::= S_3 \mid S_4 \quad S_3 ::= \texttt{I} \quad S_4 ::= \texttt{N}$$

At the beginning, we attempt to merge S_1 with S_2. By replacing S_2 with S_1, we obtain a candidate:

$$S ::= S_1 \quad S_1 ::= (S_1\ \texttt{0})^*\ S_1 \quad S_1 ::= S_3 \mid S_4 \quad S_3 ::= \texttt{I} \quad S_4 ::= \texttt{N}$$

The function SampleTokenSeqs generates new token sequences by swapping occurrences of S_1 and S_2 within the initial valid token sequence ION. Inside the initial input, S_1 is ION, while S_2 can derive values I and N. As a result, we obtain {I, N, IOION, IONON} as the sample set. The entire procedure is displayed as follows, where each line represents an iteration of the while loop (Algorithm 2).

Merge	Candidate	Sample Set
$S_1\ S_2$	$S ::= S_1 \quad S_1 ::= (S_1\ \texttt{0})^*\ S_1 \quad S_1 ::= S_3 \mid S_4 \quad S_3 ::= \texttt{I} \quad S_4 ::= \texttt{N}$	{I, N, IOION, IONON}
$S_3\ S_4$	$S ::= S_1 \quad S_1 ::= (S_1\ \texttt{0})^*\ S_1 \quad S_1 := S_3 \quad S_3 := \texttt{I} \mid \texttt{N}$	{IOI, NON}
$S_1\ S_3$	$S ::= S_1 \quad S_1 ::= (S_1\ \texttt{0})^*\ S_1 \mid \texttt{I} \mid \texttt{N}$	{I, N, IOI, NON, IOION, IONON}
$S\ \ S_1$	$S ::= (S\ \texttt{0})^*\ S \mid \texttt{I} \mid \texttt{N}$	{I, N, ION, IOION, IONON}

Token Generalization. Since our grammar synthesis operates at the token level, the resulting context-free grammar is inherently token-based. To complete the grammar, it is necessary to generate the character-level values for each token. This is accomplished by synthesizing a grammar for each token independently. Given that regular expressions are well-suited to represent the possible values of a token, we apply the first stage of our synthesis method (regular expression generalization) to the seed values of each token. Furthermore, for each individual character value of a token, we leverage the constraints gathered by Gadse to refine the search space for token generalization.

Algorithm 3 depicts the details of token generalization. For each token T, we first apply SMT optimization [de Moura and Bjørner 2008] on constraints $\mathcal{TC}$[T], which are gathered through symbolic execution, in order to compute the range $\mathcal{B}$[i] (Line 6, SMTOptimize). For each character-level seed s of token T (*i.e.*, elements in $\mathcal{TS}$[T] collected during the first-stage symbolic execution), we utilize the regular expression synthesis method (Line 9) to generate a regular expression R_t for s. After this step, each token is represented as a regular expression composed by the characters of the seed inputs. The function SearchCheck is responsible for identifying all possible character values within a regular expression. Rather than iterating through the entire ASCII chart, SearchCheck (Line 11) *only* considers characters within the range $\mathcal{B}$[i] and returns the set of possible values for each character c in s. To determine whether a character t_i can be incorporated, SearchCheck examines all token sequences in which T appears, replacing c with t_i in each sequence to check if the resulting string is accepted by $\mathcal{P}$. For t_i to be valid, it must be accepted by all token sequences in the seed set. Once the valid characters are identified, a production rule for the character generalization process is generated (Line 12). Subsequently, each occurrence of s_i in R_t is replaced with the corresponding generalized character C_i, and the generalization rule is added for each seed value of T. This token generalization process enhances the precision of the synthesized grammar through SMT optimization-based refinement.

Algorithm 3: Token Generalization

TokenGeneralization($\mathcal{P}$, L_t, α_t, $\mathcal{TC}$, $\mathcal{TS}$)

Data: $\mathcal{P}$ is the program, L_t is the synthesized token-level context-free grammar, α_t is an input token sequence, $\mathcal{TC}$ is the token constraint map, $\mathcal{TS}$ is the character-level seed map of each token.

1 **begin**
2 $L \leftarrow L_t$
3 **for** *each* $\mathtt{T} \hat{\in} \mathcal{TC}$ **do**
4 $PC_t \leftarrow \mathcal{TC}[\mathtt{T}]$
5 **for** *each* $1 \leq i \leq \max_{s \in \mathcal{TS}[\mathtt{T}]} len(s)$ **do**
6 $\mathcal{B}[i] \leftarrow \mathsf{SMTOptimize}(PC_t, i)$ ▷ Get a character's upper and lower bounds
7 **end**
8 **for** *each* $s \in \mathcal{TS}[\mathtt{T}]$ **do**
9 $R_t \leftarrow \mathsf{RegExpSynthesis}(\mathcal{P}, s)$ ▷ Synthesize a regular expression
10 **for** *each* $1 \leq i \leq len(s)$ **do**
11 $\{t_0, ..., t_n\} \leftarrow \mathsf{SearchCheck}(\mathcal{P}, \mathcal{B}[i], \alpha_t, \mathcal{TS})$ ▷ Search the possible characters
12 $L \leftarrow L \cup \{C_i := t_0 \mid ... \mid t_n\}$ ▷ Generalize each character in the token
13 $R_t \leftarrow R_t[C_i/s_i]$ ▷ Replace each s_i with C_i
14 **end**
15 $L \leftarrow L \cup \{\mathtt{T} := R_t\}$
16 **end**
17 **end**
18 **return** L
19 **end**

Example. For the grammar S ::= $(S\ \mathtt{0})^*S \mid \mathtt{I} \mid \mathtt{N}$ obtained in the previous stage, we now need to determine the character values of `I`, `0`, and `N`. Suppose the constraint of token `T_ID` is $(c[0] \geq \texttt{'a'} \wedge c[0] \leq \texttt{'z'}) \wedge (c[1] \geq \texttt{'a'} \wedge c[1] \leq \texttt{'z'}) \wedge \ldots$, and $\mathcal{TC}[\mathtt{ID}] = \{\mathsf{ab}, \mathsf{a}\}$. From this constraint, we derive that each character of `T_ID` has its lower and upper bounds, namely `'a'` and `'z'` (Line 6, SMTOptimize). Applying regular-expression generalization (Line 9, RegExpSynthesis), we obtain `I` ::= ab^* for `"ab"`, since the sampled set {`"a+1"`, `"abb+1"`} is valid. Next, we search for possible values between `'a'` and `'z'` by substituting each character for `'a'` in `"ab+1"` and likewise for `'b'` (Line 11, SearchCheck). As all newly constructed strings are valid, we finally derive `I` ::= $(\texttt{'a'} \mid \texttt{'b'} \mid \cdots \mid \texttt{'z'})(\texttt{'a'} \mid \texttt{'b'} \mid \cdots \mid \texttt{'z'})^*$ (Line 12).

3.3 Discussion

Intuitively, our newly proposed search strategy enhances DSE's efficiency by enabling broader coverage of grammar rules within limited time. It tends to be effective when early program regions contain a large width of conditionals, as different prefixes can steer DSE toward diverse basic blocks. However, this approach does not inherently guarantee superior performance over alternative search strategies in terms of valid input discovery, as it does not directly enhance the capacity for generating syntactically valid inputs. Furthermore, it does not necessarily improve parser code coverage, mainly for two reasons: 1) With sufficient time, all strategies can cover most production rules of low-complexity grammars. 2) Certain production rules can only be exercised when deep conditional branches are successfully traversed.

Our online grammar synthesis-aided method enhances token-based symbolic execution in two ways: 1) We employ the online synthesized grammar to generate valid inputs, reducing the frequency of constraint solving. The use of grammar synthesis allows for the execution of more inputs. 2) The grammar synthesis approach enables the generation of more valid inputs with different structures. These inputs help to exercise deeper functional logic beyond the initial parsing stage.

In addition, we apply token-based DSE to partition inputs into token sequences, enabling an effective token-level grammar synthesis. We gradually improve the synthesized grammar by incorporating the valid inputs newly generated by token-based DSE. This approach enables learning a more complete input grammar using only a few valid or even invalid inputs. Besides, the new search strategy can improve the quality of inputs used in grammar synthesis.

Nevertheless, some factors pose threats to the effectiveness of Lase. Due to time limits in both stages of Gadse, we may obtain incomplete token sets and few, poorly distributed valid inputs, resulting in an incomplete grammar. Moreover, Grammar-derived inputs may share similar structures, and invalid inputs can also trigger exception-handling paths, limiting coverage gains in parsers. In principle, we believe the idea of Lase can be applied to large-scale programs. However, its scalability is constrained by path explosion and the implementation of the underlying tools.

4 Evaluation

We have implemented Lase for Java programs based on Gadse [Pan et al. 2021] and Glade [Bastani et al. 2017]. Gadse is a grammar-agnostic concolic execution engine based on JPF [Anand et al. 2007]. The underlying SMT solver of Gadse is Z3 [de Moura and Bjørner 2008]. We have conducted extensive experiments to answer the following two research questions.

- **RQ1:** How effective are online grammar synthesis and our new search heuristic in improving token-based symbolic execution? Here, effectiveness is quantified as code coverage.
- **RQ2:** How effective and efficient is our token-level grammar synthesis method compared with the *state-of-the-art* methods using token-level inputs or character-level inputs? Here, effectiveness means higher precision and recall, and efficiency is measured by the synthesis time.

4.1 Experimental Setup

Benchmarks. We use 15 open-source parser programs written in Java as our benchmarks, listed in Table 1. There are ten kinds of languages among these benchmarks. These benchmarks are built based on JavaCC [Kodaganallur 2004] or Antlr [Parr and Quong 1995]. Note that we do not use the same benchmarks as Gadse since we target the programs with complex input grammar and thus exclude those with simple grammars from Gadse's benchmarks. In addition, we also employ 5 compiler or interpreter programs to estimate the effect of valid inputs produced by analyzing these 15 Java parser programs. Most of the benchmarks have complex input grammars, making it time-consuming and laborious to generate valid inputs to test them. Here, we only count the lines of code in the parsers, excluding those in the libraries on which the parser depends.

Table 1. The benchmark Java programs.

Subject	#SLOC	Brief Description
Csharp2	4145	A C# Parser
Jsijcc	9970	A JavaScript Compiler
Gap	4924	A Gap Parser
Java	22095	A Java Parser
C6	3900	A Parser for C
Feel	3979	A Feel Parser
R	6574	A R Parser
php	8311	A Php Parser
Java6	4597	A Java Parser
Oajava	15640	A Java Parser
Sixpath	5925	A Xpath Parser
Selfdef7	5035	A Self-defined Language's Parser
Java1	8766	A Java Parser
Antlrjava	6614	An antlr-based Java Parser
C15	3705	A Parser for C
Total	**114180**	15 open source Java programs

Evaluation metrics. We use statement and branch coverages as the metrics for evaluating the effectiveness and efficiency of symbolic execution. Besides, to further evaluate the effectiveness, we

collected valid inputs and executed them on compilers. We collected the coverage of the compiler's code after parsing, indicating the ability to test functionality code. We use JaCoCo [Hoffmann et al. 2014] to collect code coverage statistics while excluding the library code. For grammar synthesis, we use precision and recall rates as the metrics. Given the oracle grammar $\mathcal{G}_o$ of a program $\mathcal{P}$, we adopt the following three metrics for evaluating a synthesized grammar's effectiveness, where $s \hat{\in} \mathcal{G}$ denotes that s can be recognized by grammar $\mathcal{G}$, and the synthesized grammar is $\mathcal{G}_s$.

- **Precision**. To compute the precision, we use the method from Treevada [Arefin et al. 2024] to sample synthesized grammars, and $S(\mathcal{G}_s)$ denotes $\mathcal{G}_s$'s sampling input set.

$$\frac{\#\{s \mid s \in S(\mathcal{G}_s) \wedge s \hat{\in} \mathcal{G}_o\}}{\#S(\mathcal{G}_s)} \tag{8}$$

- **Recall**. We evaluate the recall by sampling the oracle grammar $\mathcal{G}_o$ and calculating the acceptance rate of the sampling set *w.r.t.* the synthesized grammar. We use Tribble [Havrikov 2019] to generate the inputs for oracle grammars. Besides, we use Lark [lark parser 2021] to generate the Earley parsers [Earley 1970] for the synthesized grammars. The oracle grammars are manually defined, and the measured recall serves as a relative indicator of generalization capability.

$$\frac{\#\{s \mid s \in S(\mathcal{G}_o) \wedge s \hat{\in} \mathcal{G}_s\}}{\#S(\mathcal{G}_o)} \tag{9}$$

- $\mathbf{F_1}$. We also use F_1 score to evaluate the global effectiveness.

$$\frac{2 \times Precision \times Recall}{Precision + Recall} \tag{10}$$

Note that the synthesized grammars are used as input generators for symbolic execution, not for human reading. Therefore, we do not consider grammar readability as an evaluation metric.

Baselines. To answer the two questions, we compare Lase with the baselines.

- **RQ1:** We compare code coverage against the baselines (Gadse under BFS or DFS strategy).
- **RQ2:** To evaluate the synthesis method, we compare it with the state-of-the-art black-box grammar synthesizers (Glade [Bastani et al. 2017], Arvada [Kulkarni et al. 2021], and Treevada [Arefin et al. 2024]). We add two additional baselines, *i.e.*, token-level Glade and character-level Lase, to answer the research question better.

Configuration. We create a driver for each parsing program as the entry of symbolic execution. The driver invokes the main parsing interface of the program with a string input. For **RQ1**, the time limit for all methods is one hour, and the first stage of Gadse is set to 20 minutes. In Lase, DSE and grammar synthesis alternate during the remaining 40 minutes. Once token-based DSE identifies one valid input rejected by the previous grammar, we learn a new one. We also sample 100 inputs from our newly learned grammar to feed to token-based DSE. All the experiments are carried out on a server equipped with an Intel Xeon Gold 6258R CPU @2.70 GHz, 112 logical cores, and 1.5 TiB of RAM, and the OS is Ubuntu Linux 22.04. With the same inputs, Glade-related methods and Treevada consistently generate the same grammar in each of their respective runs, so we conducted three runs to obtain a reliable average execution time. We run Arvada ten times to reduce the influence of the uncertainty, using the average values and collecting the average variances.

4.2 Experimental Results

4.2.1 Results of RQ1. **Increase in parsers' coverage.** Table 2 shows the code coverage results of **Lase** and the baselines (Gadse under **DFS** and **BFS**). Moreover, we show the results of Gadse equipped with our new search strategy (**NS**) and grammar synthesis module under DFS and BFS

strategies (**GS#D** and **GS#B**) to inspect the influence of each part. We use dark gray to highlight the best results and light gray to mark the second-best results, indicating which settings are the primary influencing factors. As indicated in the table, Lase covers more branches and statements in 14 out of 15 programs compared to **BFS**. Specifically, we obtain an average growth of 7.64% (-2.21%~22.21%) in branch coverage, and an average relative increase of statement coverage is 7.23% (-0.14%~21.88%). In addition, compared to Gadse under DFS, Lase improves 12 parsing programs. On average, Lase covers 8.16% (-9.69%~28.13%) more branches and 7.66% (-4.76%~26.65%) more statements, respectively. Moreover, Lase can explore more paths than Gadse under BFS and DFS strategies. These results indicate that Lase can improve symbolic execution's efficiency and effectiveness in analyzing complex parsing code.

Table 2. Coverage results of parsers. The metrics B and S represent branch and statement coverage, respectively.

Program		DFS	BFS	Lase	NS	GS#D	GS#B
Csharp2	B	588.3	470.0	557.0	558.0	572.0	467.3
	S	1161.3	967.7	1110.7	1113.3	1134.3	961.7
Jsijcc	B	1584.0	1582.3	1689.3	1639.7	1651.0	1663.0
	S	2740.0	2730.0	2830.7	2805.0	2756.3	2792.3
Gap	B	1030.3	973.7	1052.7	1056.3	941.3	973.3
	S	1557.7	1489.0	1602.0	1607.3	1454.3	1489.7
Java	B	1535.7	1523.3	1861.7	1856.0	1478.7	1658.7
	S	2588.7	2507.3	3056.0	3035.0	2597.0	2796.0
C6	B	588.7	595.0	663.0	598.0	620.7	624.3
	S	1231.3	1257.0	1300.0	1264.0	1274.7	1297.7
Feel	B	907.3	915.0	969.7	961.7	805.7	932.3
	S	1382.3	1405.0	1464.0	1456.0	1238.0	1418.3
R	B	1097.0	1088.7	1139.7	1101.0	1148.3	1120.3
	S	2144.0	2140.7	2257.3	2163.0	2263.3	2219.3
Php	B	869.3	878.0	898.0	896.3	874.3	890.7
	S	1969.0	1990.0	2101.7	2082.0	1984.7	2071.0
Java6	B	681.0	710.7	741.3	737.0	659.3	722.0
	S	1428.0	1502.3	1590.7	1583.0	1383.7	1570.7
OaJava	B	1487.3	1655.7	1905.7	1951.0	1490.0	1740.0
	S	2912.0	3170.3	3688.0	3772.3	2974.3	3355.0
Sixpath	B	1177.0	1087.0	1063.0	1051.0	1070.0	1057.7
	S	1813.0	1726.0	1726.7	1710.0	1742.3	1726.3
Selfdef7	B	873.0	975.7	982.0	981.0	930.0	977.0
	S	1539.0	1663.0	1667.0	1664.0	1598.0	1665.0
Java1	B	1310.7	1420.0	1597.7	1606.0	1290.3	1519.7
	S	1873.7	1959.0	2202.7	2195.7	1855.0	2101.7
Antlrjava	B	729.0	772.7	803.3	824.7	755.0	783.3
	S	1868.0	1970.7	2110.7	2182.3	1928.0	2015.7
C15	B	579.7	573.0	574.7	574.0	571.7	576.0
	S	1264.7	1259.0	1257.3	1257.0	1224.0	1262.0

Gadse equipped with our new search strategy outperforms the ones with grammar synthesis module on most of these benchmarks because the new search strategy guides Gadse to cover grammar rules efficiently. Grammar synthesis mainly contributes to generating valid inputs which may not improve the coverage of the parsing code since these valid inputs may cover a few structures which often represent a small subset of all grammatical structures. Nevertheless, grammar synthesis can help to cover more code after parsing, which will be demonstrated later. Besides, the limited parser coverage improvement with Lase is mainly due to invalid inputs generated by Gadse, which still exercise exception handling code.

Increase in coverage of compiler and interpreter programs. To further evaluate Lase's effectiveness, we collect the *valid* inputs generated by different methods when analyzing the parsing programs and feed these inputs to the compilers or interpreters to inspect the code coverage of the functionality code after parsing, *e.g.*, optimization and code generation. We use `Janino` [jan 2001], `CLoli` [clo 2015], `SimpleCsharp` [sim 2023], `Jphp` [jph 2020], and `Jscript` [jsc 2024] as the compilers or interpreters for Java, C, Csharp, PHP, and JavaScript programs, respectively. Note that the remaining languages in the benchmarks are either not commonly used or we have not found suitable interpreters or compilers for them, so they are not evaluated. Figure 6 shows the comparison of coverage increases for **Lase**, **NS**, and **GS#D** relative to **DFS** on the functionality code of different compilers, while Figure 7 displays the results of **Lase**, **NS**, and **GS#B** relative to **BFS**. _B means branch coverage and _S means statement coverage. The X-axis shows benchmarks, and the Y-axis displays the relative coverage increase calculated as follows, where GR is $(C_t - C_b)/C_b$, and C_t and C_b represent the coverage results of the target method and the baseline, respectively.

$$Y = \begin{cases} \log_{10}(GR * 100 + 1) & GR \geq 0 \\ -\log_{10}(-GR * 100 + 1) & GR < 0 \end{cases} \tag{11}$$

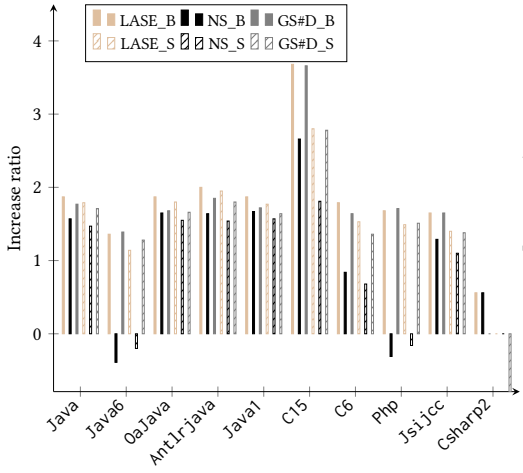


Fig. 6. Relative increase to GADSE under DFS

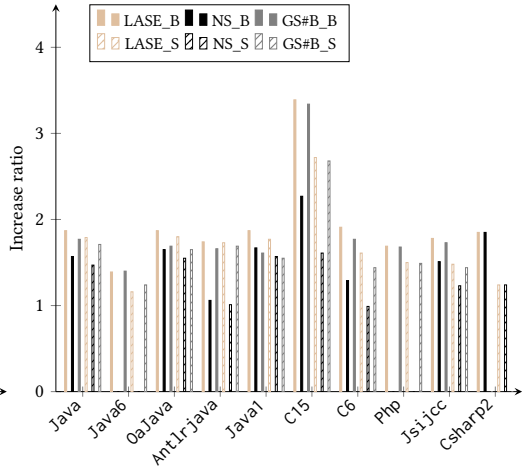


Fig. 7. Relative increase to GADSE under BFS

Compared to GADSE under the BFS strategy, LASE obtains an average growth of 299.85% (23.76% ~2442.9%) in branch coverage, and an average relative increase of statement coverage is 88.17% (13.6%~519.03%). In addition, LASE obtains 532.25% (2.63%~4826.88%) more branch coverage and 99.22% (0.0%~624.15%) more statement coverage than GADSE under the DFS strategy, on average. Moreover, LASE outperforms **NS** on all benchmarks and performs better on most programs than **GS** under both DFS and BFS strategies. Both **NS** and **GS** can improve the code coverage after parsing in most benchmarks because **NS** improves the grammar production coverage and **GS** generates more valid inputs with different structures. **NS** reduces the coverage on `Java6` and `Php` slightly compared to **DFS**, by less than 2%, as GADSE under DFS generates more valid inputs, which indicates that our new search strategy may not be universally superior. For `Csharp2`, LASE, **NS**, and **GS#D** generate more valid inputs compared to **DFS** but the valid inputs found by these approaches cover similar grammar productions, thus providing no additional benefit for coverage. Besides, both **GS#B** and **BFS** can only find one valid input for `Csharp2`. Overall, LASE combines the advantages of these two methods and can generate more structurally diverse valid inputs to cover more functionality code.

Statistics of Valid Inputs. LASE can generate a larger number of valid inputs by directly sampling from the grammar without constraint solving. The average number of valid inputs for each benchmark generated by different methods is shown in Table 3. As indicated by the table, most of the time, the methods equipped with grammar synthesis (**GS#D**, **GS#B**, and **LASE**) can generate more valid inputs. **LASE** discovers the most valid inputs on 8 out of 15 programs.

Table 3. Average number of valid inputs.

programs	DFS	BFS	GS#D	GS#B	NS	LASE
Csharp2	14.0	1.0	14.33	1.0	24.0	24.0
Gap	1.0	16.33	98.33	111.0	41.0	133.67
Java	1.0	1.0	100.67	100.67	5.0	286.33
C6	2.0	10.0	168.0	111.0	99.0	1699.67
Fee1	408.0	96.67	823.67	136.67	3254.67	3126.67
R	80.33	1.0	44.33	1.0	137.67	135.0
Php	1.67	1.0	491.67	343.33	1.0	330.67
Java6	3.33	1.0	98.33	98.0	1.0	98.67
OaJava	1.0	1.0	10.33	9.0	29.0	51.33
Sixpath	1190.0	1073.33	907.0	1100.33	1073.67	1098.0
Selfdef7	5.0	219.0	191.0	460.33	115.0	397.0
Java1	1.0	1.0	48.33	51.33	29.0	100.0
Antlrjava	2.0	3.0	129.67	178.0	9.0	251.0
C15	24.0	8.0	365.33	134.67	340.0	2262.67

In addition, among all programs, LASE can discover multiple valid inputs through symbolic execution for grammar synthesis on 8 benchmarks. Out of these 8 programs, LASE achieves a better grammar (*i.e.*, than the first synthesized one, with a higher F_1 score) for 5 programs, which indicates that the synergy between symbolic execution and grammar synthesis improves the synthesized input grammars. Moreover, compared to both **GS#D** and **GS#B**, LASE can find more valid inputs for grammar synthesis in 7 benchmarks,

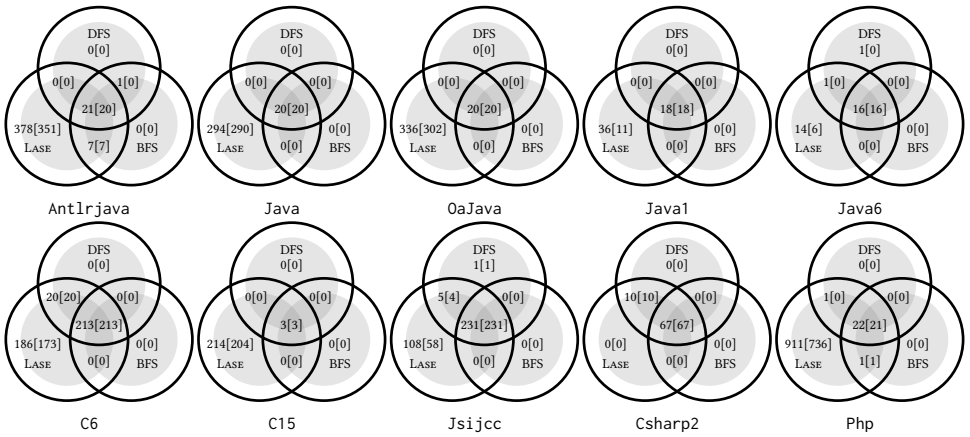


Fig. 8. Results of Bug Detection. The outer circles represent the union of mutants killed across three runs for each method, while the shaded inner regions indicate the intersection of mutants killed in all three runs. The numbers outside and inside of the brackets are the numbers of unioned and intersected killed mutants, respectively. For example, in `Antlrjava`, 378 mutants are exclusively killed by Lase's three runs, among which 351 are consistently killed in Lase's every run; 21 mutants are killed by all the three methods, among which 20 are consistently killed by each method's every run. The differences among runs arise from minor perturbations in each program execution and the inherent randomness in the input sampling of Lase.

and Lase obtains the highest recalls of the synthesized grammars for 5 benchmarks and the same recalls for the remaining ones, which indicates that the new search strategy helps to find more valid inputs with diverse structures. On some benchmarks, *e.g.*, `Java6`, Lase faces difficulties due to the large search space and a small proportion of valid inputs. Nevertheless, even when Lase fails to identify additional valid inputs through symbolic execution, it can still produce further valid inputs from the synthesized grammar.

Results of 12 hours. We conduct a 12-hour experiment to further evaluate the effectiveness of Lase on parsers' coverage and compilers' coverage. Besides, we set the first stage of Gadse to 1 hour. For parser coverage, compared to Gadse under the BFS strategy, Lase obtains an average growth of 6.06% (-7.65%∼23.78%) in branch coverage, and an average relative increase in statement coverage is 5.82% (-4.11%∼26.33%). In addition, compared to Gadse under DFS, Lase covers 9.1% (-9.99%∼28.65%) more branches and 8.03% (-4.62%∼20.26%) more statements, respectively. On compiler programs, compared to Gadse under the BFS strategy, Lase obtains an average growth of 415.39% (7.32% ∼3351.61%) in branch coverage, and an average relative increase of statement coverage is 128.06% (4.58%∼666.13%). In addition, Lase obtains 748.76% (7.32%∼6587.5%) more branch coverage and 149.34% (4.58%∼796.23%) more statement coverage than Gadse under the DFS strategy, on average. The detailed results are given in the Appendix A of the supplementary document [Ma et al. 2026].

Bug Detection. To evaluate Lase's effectiveness in detecting bugs, we employ mutation testing [Andrews et al. 2005], which is a well-established technique for assessing the fault-detection capability of software testing methods, for evaluation. We mutate the compiler programs and test the mutated programs by the inputs generated by Lase and the baselines. A mutant is killed when it causes the compiler to produce a different compilation result from the original version. The number of the killed mutants reflects the bug-detection capability of each method. We employ Major [Just 2014], a Java mutation testing tool, to mutate the functional code of the five compiler programs in Figures 6&7. Figure 8 displays the number of killed mutants obtained by different approaches. As indicated by the figure, Lase outperforms baselines on all benchmarks in terms of the capability of

bug-finding. Moreover, Lase can cover all bugs found by baselines for functionality code on 8 out of 10 benchmarks.

Scalability. To evaluate the scalability of Lase, we conducted experiments on large-scale programs with a size of approximately 100 KLOC. Since we cannot find the parser code generated by tools such as JavaCC [Kodaganallur 2004] and Antlr [Parr and Quong 1995] that could reach this scale, we constructed a synthetic grammar based on `Antlrjava`, and the parser of the synthetic grammar contains 115,692 lines of code, including 41,343 lines from the `Antlr` library. The main idea of constructing the synthetic grammar is to increase the number of the syntactic production rules in the grammar. Hence, we copy the existing grammar rules into different versions by adding different prefixes to the non-terminals. Then, we union the start non-terminals of the different versions in sequence to get a larger grammar. On this benchmark, Lase outperforms all six comparison methods. In the 12-hour experiment, Lase achieves an improvement of 83.67% in branch coverage and 102.32% in statement coverage compared to **BFS**, respectively. Relative to **DFS**, Lase attains the improvements of 114.97% in branch coverage and 131.42% in statement coverage, respectively. Note that Lase performs better on this benchmark than others because the grammar rules in the synthetic grammar have a higher degree of similarity. More detailed results are given in the Appendix B of the supplementary document.

In summary, Lase outperforms the baselines when the execution time is extended. Even with extended time budgets, the baselines still struggle to generate valid inputs effectively. Lase can enhance DSE by covering more grammar rules and generating more valid inputs more quickly. Moreover, Lase can be applied to the large-scale programs with more grammar rules.

Answer to RQ1: Lase can explore more paths, branches, and statements of the parsing code. More importantly, it generates more valid inputs covering more functionality code.

We collect the valid inputs found by Gadse under the BFS strategy for grammar learning. Table 4 presents the results of our grammar synthesis's effectiveness. The first column lists the programs. Note that `Jsijcc` fails to terminate for some inputs, so we remove it from our benchmarks for RQ2. The second column lists the metrics, where **P** and **R** represent Precision and Recall, respectively. Then, the rest columns display the results of six configurations, *i.e.*, **Lase**, **Glade-t** (Glade at the token level), **Lase-c** (Glade with **Exchange** operation), **Glade-c**, **Arvada-c** and **Treevada-c**. Lase and Glade-t conduct synthesis at the token-level (-t) while Lase-c, Glade-c, Arvada-c and Treevada-c operate at the character-level (-c). We use dark and light gray to mark the best and second-best results, respectively. Arvada-c's results consist of two numbers, where the first one is the average result, and the second one is the variance.

Compared with other methods, Lase gains the best score on 10 out of 14 programs for F_1, 6 for precision, and 8 for recall. Lase-c performs best in 9 benchmarks for recall. However, Lase-c learns the grammars that accept nearly any string, and the precision is close to 0, which indicates a tendency toward over-generalization. Note that for `Sixpath`, all the methods learn an over-generalized grammar. Treevada-c achieves better precision in 5 programs than all other methods. Nevertheless, most of its recall is 0, indicating limited generalization capabilities. These results indicate Lase's effectiveness on grammar synthesis. It is worth noting that for some benchmarks, all synthesis approaches fail to obtain high recall because we can only explore a small part of the parsing program in a limited time budget, and the valid inputs found by symbolic execution can cover a few grammar rules. Nevertheless, Lase can gradually improve the completeness of the synthesized grammar by the synergy between symbolic execution and grammar synthesis.

The effect of token-level synthesis. We compare two different groups of configuration: (i) Lase *vs.* Lase-c, (ii) Glade-t *vs.* Glade-c. For each group, the difference is that the former adopts

Table 4. Grammar synthesis results. Here, precision (**P**), recall (**R**) and F1 score (**F1**) are employed to assess the quality of the grammars.

Programs		Lase	Glade-t	Lase-c	Glade-c	Arvada-c	Treevada-c
Csharp2	**P**	0.899	0.835	0.003	0.366	0.649 ± 0.29	0.128
	R	0.420	0.420	0.000	0.000	0.000 ± 0.00	0.000
	F1	0.573	0.559	0.000	0.000	0.000 ± 0.00	0.000
Gap	**P**	0.912	0.705	0.252	0.521	0.646 ± 0.06	0.894
	R	0.380	0.200	1.000	0.200	0.375 ± 0.25	0.000
	F1	0.536	0.312	0.402	0.289	0.409 ± 0.22	0.000
Java	**P**	0.609	0.662	0.121	0.714	0.440 ± 0.28	0.407
	R	0.070	0.070	0.850	0.070	0.000 ± 0.00	0.000
	F1	0.126	0.127	0.211	0.128	0.000 ± 0.00	0.000
C6	**P**	0.833	0.693	0.213	0.613	0.613 ± 0.22	0.444
	R	0.940	0.600	0.000	0.000	0.117 ± 0.04	0.000
	F1	0.883	0.643	0.000	0.000	0.190 ± 0.07	0.000
Feel	**P**	0.776	0.570	0.720	0.398	0.615 ± 0.06	0.751
	R	1.000	0.980	1.000	1.000	0.800 ± 0.13	0.770
	F1	0.874	0.721	0.837	0.569	0.690 ± 0.06	0.760
R	**P**	0.565	0.519	0.476	0.449	0.695 ± 0.16	0.709
	R	0.350	0.230	0.250	0.230	0.000 ± 0.00	0.000
	F1	0.432	0.319	0.328	0.304	0.000 ± 0.00	0.000
Php	**P**	0.208	0.466	0.180	0.230	0.517 ± 0.15	1.000
	R	1.000	0.000	1.000	1.000	0.000 ± 0.00	0.000
	F1	0.344	0.000	0.306	0.373	0.000 ± 0.00	0.000
Java6	**P**	0.693	0.550	0.086	0.652	0.573 ± 0.16	1.000
	R	0.260	0.260	0.090	0.000	0.000 ± 0.00	0.000
	F1	0.378	0.353	0.088	0.000	0.000 ± 0.00	0.000
OaJava	**P**	0.536	0.467	0.037	0.251	0.439 ± 0.23	0.361
	R	0.660	0.130	0.850	0.130	0.000 ± 0.00	0.000
	F1	0.592	0.203	0.071	0.171	0.000 ± 0.00	0.000
Sixpath	**P**	0.000	0.000	0.000	0.000	0.000 ± 0.00	fail
	R	0.980	0.980	0.980	0.930	0.000 ± 0.00	fail
	F1	0.000	0.000	0.000	0.000	0.000 ± 0.00	fail
Selfdef7	**P**	0.848	0.949	0.098	0.499	0.506 ± 0.17	0.773
	R	0.240	0.000	1.000	0.000	0.573 ± 0.46	0.000
	F1	0.374	0.000	0.179	0.000	0.393 ± 0.31	0.000
Java1	**P**	0.294	0.351	0.043	0.524	0.630 ± 0.13	0.513
	R	0.130	0.130	0.850	0.130	0.000 ± 0.00	0.000
	F1	0.180	0.190	0.082	0.208	0.000 ± 0.00	0.000
Antlrjava	**P**	0.618	0.196	0.010	0.097	0.506 ± 0.27	1.000
	R	0.540	0.540	0.850	0.280	0.008 ± 0.01	0.000
	F1	0.577	0.288	0.019	0.144	0.015 ± 0.02	0.000
C15	**P**	0.419	0.361	0.041	0.511	0.804 ± 0.16	1.000
	R	0.200	0.000	0.000	0.000	0.000 ± 0.00	0.000
	F1	0.271	0.000	0.000	0.000	0.000 ± 0.00	0.000

the token-level synthesis while the latter adopts the character-level synthesis. We have (i) Lase achieves higher F_1 on 12 programs and lower F_1 on 1 program than Lase-c; (ii) Compared to Glade-c, Glade-t obtains higher F_1 on 8 programs and lower F_1 on 3 program. Overall, the token-level synthesis achieves better precision on most benchmarks than the character-level one. The results indicate that token-level synthesis can reduce wrong divisions in tokens and improve the precision of synthesis.

The effect of Exchange operation. We compare two different groups of configuration: (i) Lase *vs.* Glade-t, (ii) Lase-c *vs.* Glade-c. For each group, the difference is that the former incorporates **Exchange** operation, while the latter merely supports Repetition and Alternation operations. Lase-c gains higher F_1 (or remains the same) on 10 benchmarks than Glade-c but it loses precision on 11 programs. Moreover, compared with Glade-t, Lase can get higher recall with negligible impact on precision, and it outperforms Glade-c on most metrics for most benchmarks. These results demonstrate that our new generalization operations performed at the token level can generalize the inputs as much as possible while avoiding over-generalization.

Efficiency of Synthesis. We compare the synthesis time of different synthesis methods. Table 5 shows the detailed results. The last five columns in the first row show the abbreviated names of the baseline methods. Compared to Glade-related methods, Avarda-related methods consume a larger order of magnitude of time, which is prohibitively slow for online grammar synthesis. Lase is slower than the character-level methods, *i.e.*, Lase-c and Glade-c, on most benchmarks since we introduced token generalization. Compared with Glade-t, Lase is supposed to need more time for the new generalization operation and candidate checking. However, sometimes **Exchange** can reduce the number of partitions required, improving efficiency. Overall, Glade-related methods complete synthesis within 10 seconds for the majority of benchmarks.

Table 5. Grammar synthesis time results (s).

Programs	Lase	GT	LC	GC	AC	TC
Csharp2	8.68	8.59	6.60	3.33	1115.47	655.04
Gap	10.22	9.43	16.35	8.34	8743.91	8161.50
Java	8.24	8.71	5.29	2.27	955.26	291.46
C6	7.72	7.63	7.91	5.79	4237.18	3067.33
Feel	2.47	3.75	67.40	5.38	45492.89	24888.73
R	9.81	11.07	6.11	2.78	139.23	108.07
Php	7.07	8.37	3.02	2.03	334.02	113.10
Java6	6.59	8.20	3.87	2.20	1282.58	577.87
OaJava	8.38	8.83	4.54	2.24	833.18	240.68
Sixpath	11.70	11.43	28.60	9.95	1889.13	fail
Selfdef7	2.98	1.36	7.41	2.08	8056.39	7021.74
Java1	7.93	8.85	6.50	1.87	815.85	362.74
Antlrjava	6.20	7.02	6.46	3.61	1530.64	566.76
C15	7.93	8.62	7.17	3.44	1467.18	629.90
Aver	7.57	7.99	12.66	3.95	5492.35	3591.15

Answer to RQ2: Token-level grammar synthesis improves the grammar's precision and recall with an acceptable time overhead.

5 Related Work

Symbolic execution. Many existing approaches improve symbolic execution from different aspects. There exists work of pruning redundant paths [Cui et al. 2013; Kuznetsov et al. 2012; Yu et al. 2018] to improve efficiency. Similarly, Lase prioritizes valid inputs, consequently pruning those less likely to reach functionality code. Other work employs compositional symbolic execution [Anand et al. 2008; Godefroid 2007; Kim et al. 2019; Santos et al. 2020]. For instance, SMART [Godefroid 2007] summarizes low-level functions by extracting input-output relations, reusing these summaries when the low-level functions are invoked. Anand *et al.* [Anand et al. 2008] performed compositional symbolic execution in a demand-driven manner, which is more scalable for large programs. FOCAL [Kim et al. 2019] leverages function summary refinement to build symbolic path formulas targeting unit failures. Gillian [Santos et al. 2020] unifies symbolic execution, compositional program reasoning, and bi-abduction [Calcagno et al. 2009] to support whole-program analysis. Pan *et al.* [Pan et al. 2021] proposed token-based symbolic execution (Gadse), symbolizing inputs at the token level. Gadse prominently improves the efficiency of symbolic execution but still suffers from syntax checking. Moreover, Lase synthesizes grammar online for passing the parsing code more easily. Furthermore, search strategy is also a key factor for symbolic execution. Many search strategies are proposed to improve code coverage [Cadar et al. 2008; Xie et al. 2009] and exercise paths satisfying some properties [Zhang et al. 2015]. Our search strategy is designed to generate well-distributed inputs for programs with complex input grammars.

Model and grammar learning. Our work is related to model learning [Ali et al. 2019]. kTails [Biermann and Feldman 1972] provides a general state-merging framework for learning Finite State Machines (FSMs) from trace-like behaviors. kTails has been extended and applied in many scenarios, such as invariant learning [Ernst et al. 2007] and regular grammar learning [Grönfors and Juhola 1992]. Angluin proposes L^* [Angluin 1987] to learn FSMs with the assumption of having a *teacher* and an *oracle*. L^* has also been extended and applied to many problems, such as test sequence generation [Hagerer et al. 2002] and conformance testing [Groce et al. 2002]. The model learning algorithms based on kTails and L^* are general. Glade [Bastani et al. 2017] provides an input

grammar synthesis method. Glade's assumption is between kTails and L^*, Glade only requires a *Teacher*, which is more practical. For learning input grammar, Höschele and Zeller [Höschele and Zeller 2016] propose AUTOGRAM that uses dynamic taint analysis to trace the data flow of sample inputs to synthesize the input grammar. *Mimid* [Gopinath et al. 2020] tracks control flows to recover parse trees, based on which *Mimid* leverages the idea of parsing tree-based on grammar synthesis [Jim and Mandelbaum 2010; Lin and Zhang 2008] to infer input grammar. Arvada [Kulkarni et al. 2021] is a black-box grammar synthesis method that seeks the merging opportunities of the input samples for generalization. Treevada [Arefin et al. 2024] makes the generalization of Arvada deterministic. These learning methods inspire our learning algorithm. There exists an approach named REINAM [Wu et al. 2019] that utilizes symbolic execution to generate the valid inputs for grammar learning. However, it is a one-way process, whereas our method can gradually improve the grammar with fewer inputs. REINAM employs reinforcement learning to guide the grammar synthesis procedure. Unlike REINAM, we use token-level inputs for grammar synthesis to shrink the search space. Besides, our approach is orthogonal to REINAM.

Many black-box grammar synthesis approaches either segment inputs at the **character** level [Bastani et al. 2017] or preprocess them by grouping characters heuristically [Arefin et al. 2024; Kulkarni et al. 2021], to further generalize different components. These two kinds of methods cannot accurately partition inputs. However, Lase directly divides inputs into token sequences, thereby improving precision and efficiency.

Grammar-aided program analysis. Lase is relevant to those leveraging input grammar to aid symbolic execution. Godefroid *et al.* [Godefroid et al. 2008] leverage an input grammar to improve the concolic testing [Godefroid et al. 2005] of the programs with complex input grammars. CESE [Majumdar and Xu 2007] employs the input grammar to generate the valid inputs as the seed inputs for concolic testing. Grammar-based fuzzing [Havrikov and Zeller 2019; Wang et al. 2019] improves fuzzing's effectiveness and efficiency with the help of input grammar. Havrikov *et al.* [Havrikov and Zeller 2019] proposed a k-path algorithm for generating inputs that can achieve high grammar coverage and adequately test programs. Superion [Wang et al. 2019] equips grey-box fuzzing with a given grammar, guiding the mutation in a grammar-aware way. All these approaches require the existence of an input grammar. In addition, there are also some approaches [Mathis et al. 2020; Pan et al. 2021] towards grammar-agnostic analysis. LFuzzer [Mathis et al. 2020] provides a token-based fuzzing framework that records the token information to improve fuzzing efficiency. Compared with these approaches, our approach synergizes the grammar synthesis with token-level symbolic execution to improve the effectiveness and efficiency further. Besides, combining the token-based fuzzing framework [Mathis et al. 2020] with our synthesis algorithm is interesting.

6 Conclusion

Symbolic execution of programs with complex input grammars faces the challenge of generating valid inputs. We propose Lase, which synergizes grammar synthesis with token-based symbolic execution. Lase generates more valid inputs that guide symbolic execution to pass the parsing code while progressively improving the synthesized grammar. Inside Lase, a token-level grammar synthesis method is proposed to improve the learned grammar's completeness and precision. Furthermore, we propose a search strategy that enables Lase to explore more grammar rules within limited time. Experimental results demonstrate the effectiveness and efficiency of Lase. There are two aspects for the next step: (1) the evaluation on more extensive benchmarks; (2) the synergy of our synthesis method with other input generation techniques, *e.g.*, fuzzing.

Data Availability. Our artifact is available at the following URL: https://github.com/zbchen/Lase, with a permanent archive deposited on Zenodo [Ma et al. 2026].

Acknowledgments

This research was supported by National Key R&D Program of China (No. 2024YFF0908003) and the NSFC Program (No. 62172429, 62032024, and 62372162). We also thank the anonymous reviewers for their valuable feedback.

References

2001. Java compiler. http://janino-compiler.github.io/janino

2015. C compiler. https://github.com/FTRobbin/LoliCCompiler.git

2020. PHP compiler. https://github.com/jphp-group/jphp.git

2023. Csharp compiler. https://github.com/Fireball19/simple-csharp-compiler.git

2024. JavaScript interpreter. https://github.com/TopchetoEU/jscript.git

Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley. https://www.worldcat.org/oclc/12285707

Shahbaz Ali, Hailong Sun, and Yongwang Zhao. 2019. Model Learning: A Survey on Foundation, Tools and Applications. *CoRR'19* abs/1901.01910 (2019). arXiv:1901.01910 http://arxiv.org/abs/1901.01910

Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. 2008. Demand-driven compositional symbolic execution. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems.* 367–381.

Saswat Anand, Corina S. Pasareanu, and Willem Visser. 2007. JPF-SE: A Symbolic Execution Extension to Java PathFinder. In *TACAS'07*. Springer, 134–138. doi:10.1007/978-3-540-71209-1_12

J. H. Andrews, L. C. Briand, and Y. Labiche. 2005. Is mutation an appropriate tool for testing experiments?. In *ICSE'05 (ICSE '05)*. Association for Computing Machinery, 402–411. doi:10.1145/1062455.1062530

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

Mohammad Rifat Arefin, Suraj Shetiya, Zili Wang, and Christoph Csallner. 2024. Fast Deterministic Black-box Context-free Grammar Inference. ICSE 2024.

Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *ACM Comput. Surv.* 51, 3 (2018), 50:1–50:39. doi:10.1145/3182657

Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2017. Synthesizing program input grammars. In *PLDI '17*. ACM, 95–110. doi:10.1145/3062341.3062349

Alan W. Biermann and Jerome A. Feldman. 1972. On the Synthesis of Finite-State Machines from Samples of Their Behavior. *IEEE Trans. Computers* 21, 6 (1972), 592–597. doi:10.1109/TC.1972.5009015

Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX'08*. 209–224.

Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. 2009. Compositional shape analysis by means of bi-abduction. In *POPL 2009*. 289–300.

Heming Cui, Gang Hu, Jingyue Wu, and Junfeng Yang. 2013. Verifying systems rules using rule-directed symbolic execution. In *ASPLOS 2013*. ACM, 329–342. doi:10.1145/2451116.2451152

Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS'08*. 337–340. doi:10.1007/978-3-540-78800-3_24

Jay Earley. 1970. An Efficient Context-Free Parsing Algorithm. *Commun. ACM* 13, 2 (1970), 94–102. doi:10.1145/362007.362035

Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.* 1-3 (2007), 35–45. doi:10.1016/j.scico.2007.01.015

Patrice Godefroid. 2007. Compositional dynamic test generation. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007*. ACM, 47–54. doi:10.1145/1190216.1190226

Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. 2008. Grammar-based whitebox fuzzing. In *PLDI '08*. ACM, 206–215. doi:10.1145/1375581.1375607

Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing. In *PLDI 2005*. ACM, 213–223. doi:10.1145/1065010.1065036

Rahul Gopinath, Björn Mathis, and Andreas Zeller. 2020. Mining input grammars from dynamic control flow. In *ESEC/FSE '20*. ACM, 172–183. doi:10.1145/3368089.3409679

Alex Groce, Doron A. Peled, and Mihalis Yannakakis. 2002. Adaptive Model Checking. In *TACAS'02*. Springer, 357–370. doi:10.1007/3-540-46002-0_25

Tapio Grönfors and Martti Juhola. 1992. Experiments and comparison of inference methods of regular grammars. *IEEE Trans. Syst. Man Cybern.* 22, 4 (1992), 821–826. doi:10.1109/21.156594

Andreas Hagerer, Hardi Hungar, Oliver Niese, and Bernhard Steffen. 2002. Model Generation by Moderated Regular Extrapolation. In *FASE'02*. Springer, 80–95. doi:10.1007/3-540-45923-5_6

Nikolas Havrikov. 2019. tribble 1.0.0. https://github.com/havrikov/tribble

Nikolas Havrikov and Andreas Zeller. 2019. Systematically Covering Input Structure. In *ASE 2019*. IEEE, 189–199. doi:10.1109/ASE.2019.00027

Marc R Hoffmann, E Mandrikov, and M Friedenhagen. 2014. JaCoCo Java code coverage library.

John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. 2007. *Introduction to automata theory, languages, and computation, 3rd Edition*. Addison-Wesley.

Matthias Höschele and Andreas Zeller. 2016. Mining input grammars from dynamic taints. In *ASE '16*. ACM, 720–725. doi:10.1145/2970276.2970321

Trevor Jim and Yitzhak Mandelbaum. 2010. Efficient Earley Parsing with Regular Right-hand Sides. *Electron. Notes Theor. Comput. Sci.* 253, 7 (2010), 135–148. doi:10.1016/j.entcs.2010.08.037

René Just. 2014. The major mutation framework: efficient and scalable mutation analysis for Java *(ISSTA 2014)*. Association for Computing Machinery, 433–436. doi:10.1145/2610384.2628053

Yunho Kim, Shin Hong, and Moonzoo Kim. 2019. Target-driven compositional concolic testing with function summary refinement for effective bug detection. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019*. ACM, 16–26. doi:10.1145/3338906.3338934

James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (1976), 385–394. doi:10.1145/360248.360252

Viswanathan Kodaganallur. 2004. Incorporating Language Processing into Java Applications: A JavaCC Tutorial. *IEEE Softw.* 21, 4 (2004), 70–77. doi:10.1109/MS.2004.16

Neil Kulkarni, Caroline Lemieux, and Koushik Sen. 2021. Learning Highly Recursive Input Grammars. In *ASE 2021*. IEEE, 456–467. https://doi.org/10.1109/ASE51524.2021.9678879

Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. 2012. Efficient state merging in symbolic execution. In *PLDI '12*. ACM, 193–204. doi:10.1145/2254064.2254088

lark parser. 2021. Lark - a parsing toolkit for Python. https://github.com/lark-parser/lark

Yi Li, Aws Albarghouthi, Zachary Kincaid, Arie Gurfinkel, and Marsha Chechik. 2014. Symbolic optimization with SMT solvers. In *POPL '14*. ACM, 607–618. doi:10.1145/2535838.2535857

Zhiqiang Lin and Xiangyu Zhang. 2008. Deriving input syntactic structure from execution. In *FSE '08*. ACM, 83–93. doi:10.1145/1453101.1453114

Peter Linz. 2006. *An introduction to formal languages and automata, 4th Edition*. Jones and Bartlett Publishers.

Ke Ma, Yunlai Luo, Zhenbang Chen, Weijiang Hong, Yufeng Zhang, and Ji Wang. 2026. Artifact for: Online Input Grammar Synthesis Aided Symbolic Execution. doi:10.5281/zenodo.18811809

Rupak Majumdar and Ru-Gang Xu. 2007. Directed test generation using symbolic grammars. In *ASE '07*. ACM, 134–143. doi:10.1145/1321631.1321653

Björn Mathis, Rahul Gopinath, and Andreas Zeller. 2020. Learning input tokens for effective fuzzing. In *ISSTA '20*. ACM, 27–37. doi:10.1145/3395363.3397348

Oracle. 2011. Java class format. https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.10.1.9

Weiyu Pan, Zhenbang Chen, Guofeng Zhang, Yunlai Luo, Yufeng Zhang, and Ji Wang. 2021. Grammar-agnostic symbolic execution by token symbolization. In *ISSTA '21*. ACM, 374–387. doi:10.1145/3460319.3464845

Terence John Parr and Russell W. Quong. 1995. ANTLR: A Predicated- *LL(k)* Parser Generator. *Softw. Pract. Exp.* 25, 7 (1995), 789–810. doi:10.1002/spe.4380250705

RFC. 1989. TCP/IP package format. https://www.rfc-editor.org/rfc/inline-errata/rfc793.html

José Fragoso Santos, Petar Maksimovic, Sacha-Élie Ayoun, and Philippa Gardner. 2020. Gillian: Compositional Symbolic Execution for All. *CoRR* abs/2001.05059 (2020). arXiv:2001.05059 https://arxiv.org/abs/2001.05059

Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: a concolic unit testing engine for C. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005*. ACM, 263–272. doi:10.1145/1081706.1081750

Ridwan Shariffdeen, Yannic Noller, Lars Grunske, and Abhik Roychoudhury. 2021. Concolic program repair. In *PLDI 2021* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 390–405. doi:10.1145/3453483.3454051

Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: grammar-aware greybox fuzzing. In *ICSE '19*. IEEE / ACM, 724–735. doi:10.1109/ICSE.2019.00081

Zhengkai Wu, Evan Johnson, Wei Yang, Osbert Bastani, Dawn Song, Jian Peng, and Tao Xie. 2019. REINAM: reinforcement learning for input-grammar inference. In *ESEC/FSE '19*. ACM, 488–498. doi:10.1145/3338906.3338958

Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. 2009. Fitness-guided path exploration in dynamic symbolic execution. In *DSN 2009*. 359–368. doi:10.1109/DSN.2009.5270315

Qiuping Yi, Yifan Yu, and Guowei Yang. 2024. Compatible Branch Coverage Driven Symbolic Execution for Efficient Bug Finding. 8, PLDI, Article 213 (June 2024), 23 pages. doi:10.1145/3656443

Hengbiao Yu, Zhenbang Chen, Ji Wang, Zhendong Su, and Wei Dong. 2018. Symbolic verification of regular properties. In *ICSE 2018*. ACM, 871–881. doi:10.1145/3180155.3180227

Yufeng Zhang, Zhenbang Chen, Ji Wang, Wei Dong, and Zhiming Liu. 2015. Regular Property Guided Dynamic Symbolic Execution. In *ICSE 2015, Volume 1*. 643–653. doi:10.1109/ICSE.2015.80

Received 2025-10-10; accepted 2026-02-17