ABSTRACT

Major software vendors now offer software maintenance as service. Bug detection and removal are major tasks in the maintenance phase of the software life cycle. Vendors often rely on in-house built analysis tools for software fault localization. Programming language grammar is an absolute requirement for building such tools, especially for platforms which are built by third parties. However, the grammar may not be publicly available. Grammars obtained from language reference manuals are often not complete due to language evolution. Manual fixes of such a grammar is a cumbersome task, and requires special expertise.

Grammatical inference techniques have been traditionally used to infer grammar from the strings of a language, in the context of computational linguistic, but have not been successful in learning industry standard complex programming language grammars. In this paper we present a programming language grammar inference system, called Gramin, which is used to infer grammar from sample programs. Gramin employs various optimizations to make grammar inference practical in the domain of programming languages. We demonstrate the effectiveness of the Gramin system by inferring a programming language grammar used in industry, and not available in public domain.

Categories and Subject Descriptors

D.3.4 [Processors]: Compilers, Parsing

General Terms

Design, Languages, Theory

Keywords

Grammar Inference, Programming Language Grammar Inference, Grammar Induction, Grammar Learning, Incremental Learning

1. INTRODUCTION

A grammar defines the syntactic structure of all strings in a given language, and is essential for the formal definition of the language. However, in many cases, such a formal grammar does not exist publicly, or if it does exist, it may not be complete due to evolution of the language. Manually fixing the grammar is a tedious and error-prone task. Thus, there is a need for automated grammatical inference (or grammatical induction). To put it succinctly, the problem is to learn a formal grammar from a set of strings in the language.

Several variations of the problem exist, depending on the input to the induction system. We term positive samples as strings that are part of a language and negative samples as strings that are not part of the language. Common variations of the problem include availability of only positive samples, positive and negative samples, and existence of an oracle which tells whether a given string is part of the language or not part of it. The problem is naturally tackled by the machine learning techniques, and applies to the domain of Computational Linguistics, Text mining, Speech Recognition, Computational Biology, Web Intelligence, and Robotics.

The problem of grammatical inference has long been studied in the context of learning grammars for natural languages, and in the theoretical domain to study different classes of learnable grammars. Only in recent times, its importance has been felt in the area of software engineering ([17]). Due to the large size of software, large volume of software development, and incompleteness of software testing to find all bugs, software products are often released with bugs. Removal of such bugs, becomes the major task in the maintenance stage of the software life cycle. Major software vendors, therefore, rely on software bug detection tools. Such tools are often employed not only in the development phase of the life cycle, but also in maintenance phase, as software maintenance is also offered as a service. Grammar is essential in building such software debugging tools. Service providers often work on platforms which are domain specific and built by a third-party. In such cases, the grammar may be proprietary and hence not publicly available. Most of the language documentation is available in the reference manual. However, this tends to become incomplete ([16]) as the language evolves in various versions of the software. Thus grammar inference remains the only way of automatically completing the grammar for programs.

While there is a considerable body of theoretical results on grammar inference of context-free languages([19, 15]), which typically all programming language grammars are, there is little experience with applying grammar inference to fix complex programming language grammars. The main differences between learning context free grammar in programming language domains and other domains are the bigger size of the samples, and the complexities of the programming language grammar e.g. presence of recursion, nesting, looping which do not exist typically in natural language grammar. The known efforts to infer grammars for use in programming-
and future work in Section 6.

other grammatical inference work in programming language do-
of the work is presented in Section 2. The Focus algorithm, the
search, and top-to-bottom rule evaluation order to encode prefer-
have developed the Gramin system in XSB Prolog ([28]). We use
to those nonterminals where the change can occur. The algorithm
called Focus to reduce the scope of search of rules in Gramin-one
				\[ d(A, i, j) \]
\[ A \rightarrow a, j = i + 1 \]
\[ d(B, i, k), \quad d(C, k, j) \]
\[ A \rightarrow BC \]

Figure 2: CYK Parsing

goodness criteria to prefer a set of rules over another. The criteria
are based on (i) size of the generated rules, (ii) common syntac-
tic properties of programming language grammar, and (iii) various
other heuristics which works better in practice. We believe that
these optimizations are necessary towards grammatical inference
to work in practice in PL domain. We develop a new algorithm
called Focus to reduce the scope of search of rules in Gramin-one
to those nonterminals where the change can occur. The algorithm
is based on the observation that only few tokens are responsible for
unsuccessful parsing of a string.

We demonstrate that such heuristics is essential for making pro-
gramming language grammar inference practical and effective.
We have developed the Gramin system in XSB Prolog ([28]). We use
Prolog\'s natural backtracking ability to encode our backtracking
search, and top-to-bottom rule evaluation order to encode prefer-
ence rules.

The paper is organized as follows. The background material of
the work is presented in Section 2. The Focus algorithm, the
Gramin-one algorithm, and the overall search procedure are pre-
sented in Section 3. The effectiveness of Gramin system is pre-
sented in Section 4. The comparison of our work with various
other grammatical inference work in programming language do-
main in presented in Section 5. We conclude with some discussion
and future work in Section 6.

2. BACKGROUND

Context Free Grammar. A context free grammar is a 4-tuple \( G = (N, T, P, s) \), where \( N \) is a finite set of non-terminals, \( T \) is a finite set of terminal symbols, \( P \) is a set of production rules of the form \( p \rightarrow u, p \in N, u \in (N \cup T)^+ \), and \( s \) is a starting symbol. We write \( w \Rightarrow_{G^*} x \) for \( w, x \in (N \cup T)^+ \), if there is a rule \( p \rightarrow u \in P \)

and string \( z_1, z_2 \in (N \cup T)^* \) such that \( w = z_1pz_2 \) and \( x = z_1uz_2 \).

The language of \( G \) is the set \( L(G) = \{ w \in T^* | \Rightarrow_{G^*} w \} \), where the relation \( \Rightarrow_{G^*} \) is the reflexive transitive closure of the \( \Rightarrow_G \). We say a nonterminal \( A \) can parse/accept a string \( S \), if \( A \Rightarrow_{G^*} S \). We also say, a nonterminal \( A \) can include a string \( S_1 \), if \( A \Rightarrow_{G^*} S \land S_1 \) is a substring of \( S \).

Chomsky normal form (CNF) rules are of the forms \( A \rightarrow a \) and
\( A \rightarrow Q R \). Where \( A, Q, R \) are nonterminals and \( a \) is a terminal
symbol. Our generated rules are in the form \( A \rightarrow \beta \) and \( A \rightarrow \beta \gamma \),
where \( (\beta, \gamma) \in (N \cup T)^* \). This form is known as extended CNF [21].
The feature of extended CNF is that the grammars of this form is
simpler than those of CNF. We generate a grammar dependency
graph by creating nodes for each nonterminal and terminal
symbols. An edge is added from node \( A \) to \( \beta \) if there exists a rule
\( A \rightarrow \beta \), and two edges added between \( A \) to \( \beta \) and \( A \) to \( \gamma \) if there
exists a rule \( A \rightarrow \beta \gamma \). Note that we do not distinguish an edge
\( \beta \) added due to \( A \rightarrow \beta \) or \( A \rightarrow \beta \gamma \), as our aim is to capture
any form of dependency between nonterminals. The format of the
SORT Statement in ABAP language as available online is shown
below:
\[ \text{SORT <itab> \{<order>\} \[BY \langle f1\rangle \[<order>\]... \[<fn> \[<order>\]\].} \]

This is encoded in the CNF format shown in Figure 1.

CYK Parser. In CYK parsing, a table (upper triangular matrix) is
filled up gradually based on certain rules, given in Figure 2. Here,
the relation \( \text{token(Index, TokClass)} \) represents the tokens ob-
tained by lexical analysis, and denotes that TokClass is a class of
token obtained at index Index, starting from zero. For examples,
given a statement \( \text{sort tl by fl} \), the corresponding token relation is:
\[ \text{token(0,SORT). \quad token(1,IDENTIFIER).} \]
\[ \text{token(2,BY). \quad token(3,IDENTIFIER).} \]
\[ \text{token(4,DOT).} \]

We use \( \text{tokens[i...j]} \) to denote string from index \( i \) to \( j - 1 \). The
CYK rules, are to be read as, if the antecedents above the bar are
true, along with the side condition, then the consequences (below
the bar) are derivable/deducible.

The predicate \( d(Symbol, StartIndex, endIndex) \) denotes that
cell \( [StartIndex, EndIndex] \) of the CYK table has a value \( Symbol \).
Here, the symbol is a nonterminal symbol, and signifies that the
string of tokens from index \( StartIndex \) to \( EndIndex \) – 1 can be
deduced from rules generated from the \( Symbol \).

The main advantage of CYK parsing is that it generates the pars-
ing table in a completely bottom-up fashion, irrespective of any
context. This is particularly useful in grammar inference where the
context is not available when parsing of sample string goes to an
error state.

3. ALGORITHM

In this section, we present the Gramin algorithm to infer gram-
mar rules satisfying all input programs. Gramin takes an initial set
of grammar rules and a set of programs that need to be parsed by the generated grammar.

### 3.1 Overview

The Gramin framework is shown in Figure 3. The Gramin system starts with all the input programs at hand, breaking them into its constituent statements, using the lexer. All the statements are tokenized by the lexer, removing duplicate statement (but keeping the frequency) and put to individual files. This makes our technique semi-automatic, as we rely on knowing the end markers of basic and compound statements, however, this is useful, as mentioned in [5], especially when the grammars have many types of statements to learn. For each type of statement, the statement rule is considered for enhancement. All samples of a statement type go through the parsing procedure. Statements that are not parsed constitute the positive sample set.

For each type of statement, Gramin employs the learning algorithm on statements from its positive sample set, in order of decreasing frequency. We notice that the statements which are used most frequently constitute the basic form of statement, and statements with lesser frequency are variations to this basic form. Thus, we generate the grammar for statements in order of increasing frequency. In case, when frequency information is not available we can order the statements based on number of distinct tokens and their containment relation.

On each statement of the positive sample set, Gramin employs an algorithm called Focus to determine a set of nonterminal and substring pairs (called focus nonterminal and focus string respectively, and collectively called FocusSet) such that the reachable (reflexive and transitive) nonterminals of focus nonterminal in the grammar dependency graph can undergo addition of rules to parse the substring. The parsing of the substring will result into parsing of the entire string. The focus nonterminals are ordered in terms of the length of the corresponding substring. We describe the Focus procedure in Subsection 3.2.

The Gramin-one procedure is called with a focus nonterminal-substring pair, and an initial set of rules. The bridging procedure generates sets of rules, each set along with the initial set of rules, parses the given sample. In practice, many rule sets can parse the given sample string. However, the rule generation procedure (bridging) itself gives more priority and prunes some form of rules.

Each generated set of rules goes through a pruning procedure where either the rule set is selected or not selected based on certain criteria. Ranking procedure is applied to the pruned rule sets. The highest ranked rule set is chosen as the output of Gramin-one, and the rest of the ranked rule sets are stored as options, required in backtracking. Subsection 3.3 discusses many pruning and ranking criteria.

The generated rules are merged to the original set of rules, and the procedure is continued with another statement from the sample set. The procedure is complete when every sample for the type of statement is exhausted. In case when Gramin-one fails to generate a set of rules, it first backtracks to the remaining focus pairs. If no more focus pairs are left it backtracks to the last statement and tries to select another pruned result in order of ranking. In case such result set does not exist, backtracking is continued. Note that the FocusStack and GStack are used in Figure 3 to illustrate backtracking. The search procedure is described in Subsection 3.4.

### 3.2 Focus

For each positive sample, Gramin calls the Gramin-one procedure which can augment the existing set of rules with a new set of rules that can parse the unparsed statement. As there can be many possible sets of rules that can parse a statement, the search space for Gramin-one procedure is large. The overall goal of the Focus algorithm is to reduce the search space for Gramin-one. The Focus algorithm is based on the observation that, when a statement is not parsed by the corresponding statement rule, only certain parts of the statement cause the unsuccessful parsing. Focus procedure associates a nonterminal with a substring such that the nonterminal includes the parts. The reduction in search space is achieved based on three factors: i) in practice, the number and length of such parts causing the unsuccessful parsing is relatively small, ii) association of an entire part with a nonterminal, iii) choosing appropriate nonterminals to parse. None of these optimizations introduces the possibility of non-parsing of the sentence; they only give preferences to certain nonterminals for addition of rules over others, as the result rules are more PL grammar like.

The Focus algorithm is presented in Figure 4. The algorithm is called with the nonterminal for the statement type for which we are trying to complete the rules. For example, to complete the rule for the sort statement we call the Focus algorithm as Focus(sort_statement, 0, L), where zero is the index of the first token of the unparsed statement, and L is the number of tokens generated from the unparsed statement.

We consider the case 1 in the Figure 4. If A already accepts a part of the input string, making two disjoint parts which still need to be included, Focus selects the nonterminal A to accept the string, and does not try to create other non-terminals to accept the disjoint strings. If case 1 is not satisfied, we consider the case 2.

In case 2, we consider the rules of the form $A \rightarrow BC$. In subcase 2.1, the nonterminal B and C parses two parts of the input string. The remaining part (the string from index J to J+P, say $S_p$) of the string is not parsed by either B or C. In this case, we derive that either the rules of B or the rules of C can be augmented to include rules such that the unparsed part of the string can be
parsed. If either of them includes the $S_p$, $A$ can parse the entire string. Note that here we are eliminating the possibilities that i) the rules of $A$ can be changed (like $A \rightarrow B \text{D, D, R}_a$), ii) $B$ and $C$ together includes $S_p$. The first elimination is based on the fact that this elimination will only cause in the reduction in readability of semantic association of names of nonterminals, as new rules can be added to the rules of $B$ to parse $R_a$. The second elimination is based on the observation that the $R_a$ is typically another clause or option associated with either $B$ or $C$, but not both.

The Focus algorithm fails to focus (returns with the called set an answer set) for the terminals.

In sub-case 2.II, the nonterminal $B$ accepts the left part of the string, and $C$ does not accept any suffix. In that case, we prefer that $C$ accepts the remaining part as it is supposed to accept a string on right side of $B$, eliminating the possibility of including the remaining part by $B$. However, when $C$ is optional (expressed using condition: if $\exists d(A, I, J)$, then the both $B$ and $C$ can accept the remaining part. The Case III, is similar to case II.

Example. Consider an example unparsed string for the sort statements rule presented in Section 2.

```
SORT t1[]. BY f1. which is tokenized as below:

token(0,SORT). token(1,IDENTIFIER).
token(2,LBRACKET). token(3,RBRACKET).
token(4,DOT).
```

Note that $\text{tab}$ rule says that $\text{tab}$ (internal table) can be only an identifier, whereas in this example, the $\text{tab}$ is represented by three tokens representing string $t1[]$. The illustration for the example is shown in Figure 5. As a result of the Focus procedure, we obtain three results, $(\text{tab}, 1, 4)$, $(C, 2, 6)$, and $(\text{sort}$ $\text{option}, 2, 4)$.

We now present a modification of the Focus procedure based on the following observations,

- If all the non-empty strings accepted by a nonterminal can start (end) with only one keyword, then it is highly unlikely that the rule will be changed to accept a string which do not start (end) with the keyword.

Many examples of this observation exist, e.g. for, while statements start with for and while keywords, respectively. As for the

Figure 3: Gramin: An Iterative Backtracking Algorithm

Figure 4: Focus Algorithm
The Focus algorithm can be seen as a novel error recovery technique available in existing compilers [23]. In Section 4 we demonstrate the effectiveness of having Focus algorithm in grammar inference.

3.3 Gramin-one

In this section we present the algorithm for generating CNF rules which along with the input rules will parse the sample string (refer to Function BridgingPruning in Figure 3). We first present the bridging rules ([10]) which can generate the CNF rules to parse an unparsed string.

The bridging rules are presented in Figure 7. Each rule has three parts, the consequent (below the line) can be derived from the antecedents (above the line) and side condition (right side of the line). The expression \( i \rightarrow k \) defines a relation called bridge, and denotes that the substring \( \text{token}[i \ldots k] \) can be parsed using the rules of Nonterminal \( A \) (or in other words string generated from nonterminal \( A \) will contain the string \( \text{token}[i \ldots k] \)). Note that the predicate \( d/3 \) denotes the CYK table, which is derived using only the initial set of rules, whereas derivation of bridge uses newly derived rules. The relation \( d/3 \) is computed before calling the Gramin-one procedure and is not changed during the Gramin-one. Thus, precondition of the rule on relation \( d/3 \) occurs as side condition, whereas rules and bridge relation occur as antecedents of the rules, as both the relations are changing.

The first rule generates a new rule \( A \rightarrow \beta \gamma \), and bridges \( \text{tokens}[i \ldots k] \), if \( \beta \) and \( \gamma \) have individually parsed the left and right parts of the string. The rule evaluation itself can be done either as bottom-up or top-down. Gramin-one employs a top-down strategy for evaluating these rules, in that case, \( A \) can be bound in the generated rule if its bound in the call bridge/3. The second rule resembles with the second rule of CYK parsing in Figure 2. The 3rd, 4th, and 7th rules are very similar, if there exist a rule \( R \rightarrow QR \) which along with the input rules will parse the sample string, can be exceedingly large. However, the number of possible rule sets derived from the bridging rules, where each rule set along with the initial rules can derive the sample string, can be exceedingly large. In practise, only few of those possible rule sets can be considered as good rules. Only those rules will sustain through the Gramin-one procedure and will not be discarded by pruning.

Pruning Bridge Relation. Some strategies are implemented as conditions to each bridging rule, if the condition is not satisfied, then the consequent is not included in the bridge relation and new rules are not generated (whenever applicable).

Using Grammar Dependency Graph. Gramin uses grammar dependency graph to restrict the domain of unbound nonterminals in bridging rules. For example, if \( A \rightarrow QR \) is generated using the second bridging rule, where \( Q \) recognizes an identifier. In typical programming language grammar many nonterminals will parse an identifier. The domain of \( Q \) is therefore large, and so is the domain of \( QR \) which is the cross product of domain of \( Q \) and \( R \). The
use of this rule restricts the domain of $Q$ to all the nonterminals which is either reachable from the nonterminal $A$ in the grammar dependency structure or a freshly generated nonterminal. Typical PL grammar are not expressed with minimum number of nonterminals. In practice, nonterminals which accept same string are given different names in the grammar to represent different semantic entities. Using this pruning strategy, we maintain the semantic dependencies by preserving the same dependency structure in original grammar.

Example. For input string ‘SORT t1 BY f1.’, some of the rule sets generated by bridging rule are given below:

\[
\begin{align*}
\frac{A \rightarrow Q, i \rightarrow j}{i \rightarrow A, k} & \quad d(Q, i, j) \quad (1) \\
\frac{A \rightarrow Q, i \rightarrow j}{i \rightarrow k} & \quad d(Q, i, j) \quad (2) \\
\frac{A \rightarrow Q, i \rightarrow j}{i \rightarrow k} & \quad d(R, i, j) \quad (3) \\
\frac{A \rightarrow Q, i \rightarrow j}{i \rightarrow k} & \quad d(R, i, j) \quad (4)
\end{align*}
\]

![Figure 7: Bridging rules](image)

1In general, it is possible to infer the nonterminals with same type based on an analysis of dependency graph. This is part of the future work.

cally equivalent to terminals. For example, the nonterminals `identifier, end_of_statement` typically do not undergo changes. We therefore omit these nonterminals from the domain of LHS of a newly generated rule.

Example. We consider the input 'SORT t1 BY f1 STABLE.' to the Gramin-one algorithm. The Focus algorithm limits the scope of search to $(A, 0, 5)$. Bridging rule generates the answer $E \rightarrow STABLE$ along with other answers. The answer is valid as $E$ will accept the string f1 STABLE. However, as $E$ serves the purpose of expressing $sort_{by\_item+}$, introducing $E \rightarrow STABLE$ will accept the string $STABLE+ \rightarrow E$, which is likely to be incorrect. Removing $E$ from LHS of the generated rule will prune other rules generated by bridging rules. Also the bridging rule generates the answers where `identifier` is in the LHS of the rule, e.g. $identifier \rightarrow STABLE$, which are also ruled out by Gramin-one.

**Pruning/Ranking Solutions.** The following strategies are used to either prune or rank rule sets (refer to functions $Prune(G_1)$ and $Rank(G_2)$ in Figure 3) generated by the bridging rules.

**Matching Parenthesis.** The programming language grammar maintains an invariant that every open structure should have its corresponding closed structure maintaining proper nesting. Following this, the newly introduced rule along with the initial rules should generate matching parenthesis. The similar rules are applicable for braces and brackets.

**Single Form of Recursion.** The bridging rules can generate different sets of solution which are semantically equivalent. Three different forms of recursion left, right, double can be used to express the repetition. Gramin-one system only allows right recursive rules. The other two forms of recursion are pruned out when the right recursive form is present.

**Number of Productions Rule/Nonterminals.** In order to get a small grammar, we order the sets of rules based on the cardinality of each set. The set which has less production rules or nonterminals are preferred over others. Nakamura et al. ([10]) uses the criteria of number of production rules, so that the resultant grammar contain minimum number of production rules.
Subset Rules. Bridging rules can present a set of rules which may be the superset of an already produced set. Gramin only considers rules whose subset rule set has not yet been generated by Gramin-one. For our example, consider these two rule sets generated by bridging rules. Here the second rule set is pruned out.

Input: \text{SORT t1 by f1 STABLE.}
\[
\begin{align*}
\{ A &\rightarrow A \text{ STABLE} \\
A &\rightarrow A \text{ STABLE, sort.clause } \rightarrow X \text{ stab, } X \rightarrow \text{IDENTIFIER by} \}
\end{align*}
\]

Repetition of entities. If there are more than one similar structures appear consecutively, then generate a rule representing the repetition. Also, avoid introducing recursion of non-similar entities, and recursion having no base case to terminate.

Programming syntax is repetitive in nature, and use of recursion is very common in PL grammars. Gramin uses this observation to generate new rules that are recursive in nature, rather than rules having finite number of consecutive structures. For example, instead of generating a rule

\[A \rightarrow \text{identifier identifier},\]
Gramin-one gives preference to

\[A \rightarrow X, A \rightarrow X A, X \rightarrow \text{identifier} \]. 

Infact, the grammar obtained by repetition/recursion rule is preferred over other grammars. The use of delimiters like comma is very common in denoting such repetitive syntax. We include the following rule in the set of bridging rules.

\[
A \rightarrow \beta \rightarrow d(A, i, j), \ d(\beta, i, j) \ d(\beta, j, k)
\]

Example. We present an example where Gramin-one avoids generating unintended recursion rules. For the input ‘SORT t1 BY t1 STABLE.’ the bridging rules are going to generate following rule set: \{A \rightarrow A \text{ STABLE}\} to include the keyword \text{STABLE} at the end of the string accepted by \text{A}. However, this introduces an unintended recursion, which is pruned.

3.4 Search

Bridging rules can be used to generate rules which along with initial input samples accept one positive sample statement. Typically many such rule sets can be generated to accept one sample statement. Gramin-one system uses heuristics based strategy to prune rule sets and order one rule set over another. However, generating all possible rule sets and subsequently applying all heuristics to get the best possible solution is not always feasible, as all possible rule sets may not fit into memory. Note that the bridging rule do not always produce rule set in order of preference, even though certain preference is imposed by ordering the set of bridging rules.

In this scenario, Gramin employs a search strategy where it uses Gramin-one rules to obtain a predefined number of rule sets. Then it employs the goodness criteria to find the best solution among those rule sets, and keeps the other solutions in store. The chosen solution is then added to the set of rules, and the resultant set of rules is used as input to generate rules for next sample statement. When this process fails to generate rule set for a statement, the process backtracks to the last statement, generates a single solution using bridging rules (if possible), adds the solution to the already existing solution set in store. Finally, it computes the best of the solution set and goes forward with the best solution set. The complexity is not described in Figure 3. The pseudo code of the search procedure is available at [26].

4. EXPERIMENTAL RESULTS

In this section we present the results of some of the experiments performed using Gramin. Parts of the Gramin system are implemented in Java and XSB Prolog [28]. XSB uses a top-down goal directed resolution strategy, which explores only the required portion of the search space, required to solve the query. We use XSB version 3.1 with batch mode, where for each query it computes answers one by one (instead of computing the entire relation in default mode). Due to the goal directed nature of XSB’s resolution strategy, Gramin does not compute the entire bridge relation. Instead it only generates the result sets particular to the query obtained by focus algorithm.

An Industrial Case Study. As a case study, we consider a complex industry standard language called ABAP. It is not a general purpose programming language, as such its grammar is not readily available. The language has many syntax variations for each type of statement, which makes it particularly harder for manual fixing. This makes it a right candidate for grammar inference. In our experiments, different types of database related statements are extracted from these programs, kept in separate folders. Each statement is represented as token/2 predicate as Prolog facts and stored in a file. The statements are ordered in decreasing order of frequency. We only show our output for 10 statements with higher frequencies, whereas in practice there are 687 lexical variations.

We start with a very basic grammar which can parse only the first statement, and describe how the grammar changes with every sample. The input sample and the changes are presented below.

<table>
<thead>
<tr>
<th>No.</th>
<th>Freq</th>
<th>Smg</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2080</td>
<td>SORT ID BY ID DOT</td>
</tr>
<tr>
<td>2</td>
<td>993</td>
<td>SORT ID DOT</td>
</tr>
<tr>
<td>3</td>
<td>855</td>
<td>SORT ID BY ID DOT</td>
</tr>
<tr>
<td>4</td>
<td>318</td>
<td>SORT ID BY ID ID DOT</td>
</tr>
<tr>
<td>5</td>
<td>254</td>
<td>SORT ID BY ID DESCENDING DOT</td>
</tr>
<tr>
<td>6</td>
<td>165</td>
<td>SORT ID BY ID ASCENDING DOT</td>
</tr>
<tr>
<td>7</td>
<td>124</td>
<td>SORT ID BY ID ID ID DOT</td>
</tr>
<tr>
<td>8</td>
<td>120</td>
<td>SORT ID BY NAME DOT</td>
</tr>
<tr>
<td>9</td>
<td>63</td>
<td>SORT ID BY ID DESCENDING ID DESCENDING DOT</td>
</tr>
<tr>
<td>10</td>
<td>58</td>
<td>SORT ID BY ID ID ID ID DOT</td>
</tr>
<tr>
<td>11</td>
<td>34</td>
<td>SORT ID LBRACKET RBRACKET BY ID DOT</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{Initial Rules} \\
\text{strmt }\rightarrow \text{sort.stmt} \\
\text{sort.stmt }\rightarrow \text{sort.stmt}\ D\ T\ O\ T \\
\text{byclause }\rightarrow \text{BY id name} \\
\text{fieldname }\rightarrow \text{id} \\
\text{sort.stmt1 }\rightarrow \text{table byclause}
\end{align*}
\]

<table>
<thead>
<tr>
<th>No.</th>
<th>Focus</th>
<th>Rules Added</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>parsed</td>
</tr>
<tr>
<td>2</td>
<td>sort_stmt1, 1, 2</td>
<td>sort_stmt1 $\rightarrow$ ID</td>
</tr>
<tr>
<td>3</td>
<td>fieldname, 3, 5</td>
<td>fieldname $\rightarrow$ n1.fieldname</td>
</tr>
<tr>
<td></td>
<td>n1 $\rightarrow$ ID</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>parsed</td>
</tr>
<tr>
<td>5</td>
<td>n1, 3, 3</td>
<td>n1 $\rightarrow$ ID DESCENDING</td>
</tr>
<tr>
<td>6</td>
<td>n1, 3, 3, n1</td>
<td>n1 $\rightarrow$ ID ASCENDING</td>
</tr>
<tr>
<td>7</td>
<td>-</td>
<td>parsed</td>
</tr>
<tr>
<td>8</td>
<td>fieldname, 5, 4</td>
<td>byclause $\rightarrow$ BY NAME</td>
</tr>
<tr>
<td>9</td>
<td>-</td>
<td>parsed</td>
</tr>
<tr>
<td>10</td>
<td>-</td>
<td>parsed</td>
</tr>
<tr>
<td>11</td>
<td>table, 1, 4</td>
<td>table $\rightarrow$ identifier n4</td>
</tr>
<tr>
<td></td>
<td>n4 $\rightarrow$ LBRACKET RBRACKET</td>
<td></td>
</tr>
</tbody>
</table>

The first statement is parsed by the initial set of rules.\(^2\) For the second statement, the focus non-terminal is sort_stmt1 and the

\(^2\)Gramin employs a heuristic to generate the initial set of rules for ABAP language, making the procedure truly incremental for ABAP. We do not discuss the heuristic here.
string is ID (token at position 1). For statement 3, the bridging rule for recursion is applied to obtain a recursion rule. Note that one of the possible recursion rule set is fieldname → ID|fieldname and fieldname → ID which is also generated using bridging rules. However, Gramin prefers the rules with a nonterminal such that any option to the nonterminal can be considered, which is very common in PL, and as can be seen from the statement 5. For statement 5, the initial focus nonterminal obtained is fieldname with string DESCENDING. This generates the rules which are pruned by Or-ing with similar condition type. Backtracking, the focus string obtained is ID DESCENDING (index 3 and 4). This generates the rule n1 → ID DESCENDING, n3. Similar backtracking is also seen for statement 8, where generated rules for fieldname are pruned using Or-ing with similar type condition. The total set of rules matches with the hand-written rules for the given sample set.

We present the effectiveness of various optimizations described in the paper by comparing with the basic case where no strategies has been implemented. The comparison is done on the number of results that bridging rules would have generated with or without the optimization. We first demonstrate the optimization based on the Focus algorithm. The result, shown in Figure 8, demonstrates many fold reduction in the number of results generated by the Gramin for a single statement. The results clearly demonstrate decrease in search space. Note that for some cases our system goes out of memory (after producing 5 million grammars) when we tried to obtain this result without any pruning strategy in place.

Next, we present the effectiveness of different pruning conditions, which are implemented as conditions in the bridging rules (Figure 9). The Gram., LHS, and Or columns denote the optimizations based on restriction of RHS of a rule using grammar dependency, restriction of domain of LHS Nonterminals, and Or-ing similar types respectively. The numbers in these three columns denote the number of results obtained only if that particular optimization is turned on. The last column presents the results when all optimizations are turned on. The optimizations by restricting the domain to the reachable grammar dependency and restriction to LHS nonterminals have been very effective in pruning the results. Note that the result zero in the last column denotes that none of the grammars survived the pruning stage, which results in backtracking.

After the bridging rules generate result sets, Gramin employs global pruning strategies like parenthesis matching, superset removal, single form of recursion, and inter rule grammar dependency checking.

To show the effect of parenthesis removal, we consider the 2 sample statements having parenthesis. The result is shown in Figure 10. For the first example, among the 25 results obtained for sort10 example, without any pruning, 18 results have proper parenthesis matching. Note that pruning parenthesis is only effective for samples which have brackets or parenthesis. The next result (Figure 11) is due to recursion removal and grammar dependency checking.

After the pruning strategies are applied, several ranking strategies are used to select the appropriate rule set. Figure 11 presents the results that bridging rules would have generated with or without the optimization.

<table>
<thead>
<tr>
<th>Statement</th>
<th>Without Focus</th>
<th>With Focus</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Size</td>
<td>Space</td>
</tr>
<tr>
<td>sort1</td>
<td>3</td>
<td>484</td>
</tr>
<tr>
<td>sort2</td>
<td>6</td>
<td>249287</td>
</tr>
<tr>
<td>sort3</td>
<td>7</td>
<td>2856</td>
</tr>
<tr>
<td>sort4</td>
<td>6</td>
<td>86044</td>
</tr>
<tr>
<td>sort5</td>
<td>6</td>
<td>86044</td>
</tr>
<tr>
<td>sort6</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>sort7</td>
<td>5</td>
<td>3301</td>
</tr>
<tr>
<td>sort8</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>sort9</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

Figure 8: Effectiveness of Focus optimization (- means > 5M)

<table>
<thead>
<tr>
<th>Statement</th>
<th>No Pruning</th>
<th>Number of Grammars</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Size</td>
<td>Space</td>
</tr>
<tr>
<td>sort1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>sort2</td>
<td>139</td>
<td>95</td>
</tr>
<tr>
<td>sort3</td>
<td>187</td>
<td>552</td>
</tr>
<tr>
<td>sort4</td>
<td>54</td>
<td>39</td>
</tr>
<tr>
<td>sort5</td>
<td>54</td>
<td>39</td>
</tr>
<tr>
<td>sort6</td>
<td>58892</td>
<td>34456</td>
</tr>
<tr>
<td>sort7</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>sort8</td>
<td>10446</td>
<td>6696</td>
</tr>
<tr>
<td>sort9</td>
<td>1214540</td>
<td>655646</td>
</tr>
</tbody>
</table>

Figure 9: Effectiveness of Pruning Conditions

Figure 10: Parenthesis Matching

<table>
<thead>
<tr>
<th>Statement</th>
<th>Pruning Rule Sets</th>
<th>Ranking Rule Sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Init.</td>
<td>Rec.</td>
</tr>
<tr>
<td></td>
<td>Gram.</td>
<td></td>
</tr>
<tr>
<td>sort1</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>sort2</td>
<td>17</td>
<td>12</td>
</tr>
<tr>
<td>sort3</td>
<td>184</td>
<td>136</td>
</tr>
<tr>
<td>sort4</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>sort5</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>sort6</td>
<td>1776</td>
<td>1312</td>
</tr>
<tr>
<td>sort7</td>
<td>540</td>
<td>2856</td>
</tr>
<tr>
<td>sort8</td>
<td>240</td>
<td>1992</td>
</tr>
<tr>
<td>sort9</td>
<td>17424</td>
<td>13968</td>
</tr>
</tbody>
</table>

Figure 11: Effect of Removal of Recursion (Rec.) and rule set pruning due of grammar dependency (Gram.), and Frequency Distribution (Value, Frequency) after pruning

Figure 12: Effectiveness of Focus and Pruning on ABAP examples (- means > 5M)
5. RELATED WORK

The problem of grammatical inference has been extensively researched, especially in the context of learning grammars for natural languages, and in theoretical domain to study different classes of learnability of grammars. Excellent account of this research is presented at [4, 18]. One of the important results presented by Gold states that no grammar in the Chomsky hierarchy can be obtained only using positive examples [9]. The result by Angluin [1] states that the problem of grammar learning asking a polynomial number of queries to an oracle is NP-Hard, which was later proved to be NP-Complete. We observe that in most practical cases, especially in the domain of programming languages, it is not possible to obtain negative examples. We also do not assume existence of a compiler. Although, in some cases, compiler of the target language does exist, which can act as an oracle. In recent past, only few techniques have dealt with grammar inference in the context of programming languages. In this section, we compare our approach with such techniques.

Nakamura et. al. have described a system called Synapse, which essentially employs two techniques based on bottom-up CYK algorithm to obtain a rule set for one positive sample string, along with various kinds of search techniques to arrive at minimal or semi-minimal set of rules for all positive samples. In [22] they have built a backtracking based inductive CYK algorithm which learns CFG rules from positive and negative examples. At each step, it takes an unprocessed string, and generates a minimal set of production rules to derive the string, followed by the check that it does not derive any given negative sample. The inductive CYK algorithm relates to rule 6 in Figure 7. In recent work [21, 10] they have described a Prolog version of the Synapse system which is based on the idea of bridging rule set. The method is essentially a top down search which generates production rules that bridge or make up, any lacking parts of an incomplete derivation tree of the result of the parsing positive string. However, their bridging rule based algorithm generates exceeding large number of grammars in PL domain, and their size based ranking criteria is insufficient to produce good PL grammars. Though our initial set of rules is motivated from the bridging rule set presented in [21, 10], we extended their bridging rule set to include certain rules considering common patterns in programming language grammar. Moreover, we have used Focus algorithm to reduce the grammar search space, making it feasible for learning statement level PL grammar. No such reduction is used in Synapse. Finally, we have devised several pruning and ranking strategy to derive solution set which is more suitable towards being programming language grammar.

Jalote et. al. have presented several techniques ([7, 8]) for specifically programming language grammar inference. The main motivation of their work is the availability of a variant of a well known programming language for which the grammar does not exist in public domain. Their approach has four phases, LHS gathering, RHS gathering, rule building, and rule checking. LHS gathering phase employs LR parsing to arrive at an error state, doing possible reduction (oblivious of the next symbol in the input), and inspects item sets on top of the stack to obtain the next non-terminal symbol which could be reduced. The RHS gathering phase employs CYK algorithm to obtain possible RHSs starting from the last occurrence of the newly introduced symbol, for a specified length. The Focus algorithm has similar in aim as LHS gathering phase, but the procedure totally differs. The RHS gathering phase is also very different from the bridging rules. Their technique is catered towards finding an unknown keyword symbol, whereas our technique assumes the existence of all keywords, but not the variations in the rules.

In [6] the author has established the importance of goodness criteria for programming language grammar rules for rule selection in learning process. Grammar based metrics are either size based which are defined in terms of number of terminals, nonterminals, productions of the grammar or structure based which are defined in terms of relationships between the nonterminals. Existing grammar metrics [3, 24] are not sufficient for the purpose of grammar rule selection. Dubey et. al. ([6]) describes two more criteria. A rule is good if it is smaller in size and covers/derives larger substring. Another criteria presented in this paper is based on patterns that grammar writers follow: the list pattern, operator precedence. Note that we also use this criterion for rule selection. Along with these, we contributed new heuristics to obtain ‘good’ rules.

Mernik et. al. ([20, 27]) have proposed a genetic programming based approach which is based on grammar specific heuristic operators. Their approach extracts grammars of small domain specific languages. None of their techniques attempts to infer grammar of real programming languages. Similar approach has been considered by Javed et. al. in [12, 13]. In [14] Javed et. al. have considered the beam search, and minimum description length [25] heuristic (as also in [6]) to direct the search towards simpler grammar.
6. CONCLUSION AND FUTURE WORK
In this paper we have presented the Gramin system which infers the programming language grammar from sample programs using grammar inference techniques. While grammar inference techniques have traditionally been used in the NLP and bio-informatics domains, these techniques are now being applied to the programming language domain too. Existing techniques in grammar inference, which are applied in the NLP domain, do not consider regularities and complexities of programming language (PL) grammars. Instead of all purpose grammar inference system, we needed techniques to make grammar inference system particularly suitable towards inferring PL grammars. Gramin is a practical system suitable for inferring PL statement grammar.

In future, we would also like to develop PL grammar inference algorithm which uses the compiler present for the target language. The compiler could be used for controlling the over-generalization existed in generated rules. We would also like to build a system that exploits the availability of all positive samples at hand. If all statements are available, then it is possible to infer common patterns existed in many statements and use them to generation process to reduce backtracking. We are also mining several online grammars to capture common patterns existed in them.

ACKNOWLEDGEMENT. We thank Prof. K. Nakamura for sharing the system Synapse version 4. We Sincerely thank Mangala Gorwi Nanda and Vibha Sinha for giving important feedback to this report, Julian Dolby for sharing ABAP grammar, Pradipta Paul and Sourin Ghosh for providing ABAP sample programs.

7. REFERENCES