Available online at www.sciencedirect.com
SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science 141 (2005) 99–116

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Inferring Context-Free Grammars for Domain-Specific Languages

Matej Črepinšek, Marjan Mernik [1],[2]

University of Maribor,
Faculty of Electrical Engineering and Computer Science,
Smetanova 17, 2000 Maribor, Slovenia

Barrett R. Bryant, Faizan Javed, Alan Sprague[1],[3]

The University of Alabama at Birmingham,
Department of Computer and Information Sciences,
Birmingham, AL 35294-1170, U.S.A.

Abstract

In the area of programming languages, context-free grammars (CFGs) are of special importance since almost all programming languages employ CFG's in their design. Recent approaches to CFG induction are not able to infer context-free grammars for general-purpose programming languages. In this paper it is shown that syntax of a small domain-specific language can be inferred from positive and negative programs provided by domain experts. In our work we are using the genetic programming approach in grammatical inference. Grammar-specific heuristic operators and non-random construction of the initial population are proposed to achieve this task. Suitability of the approach is shown by examples where underlying context-free grammars are successfully inferred.

Keywords: Grammar induction, Grammar inference, Learning from positive and negative examples, Genetic programming, Exhaustive search

[1] This work is sponsored by bilateral project BI-US/03-04/5 "GenParse: Generating a Parser from Examples" between Slovenia and USA.

[2] Email: {`matej.crepinsek, marjan.mernik`}`@uni-mb.si`

[3] Email: {`bryant, javedf, sprague`}`@cis.uab.edu`

doi:10.1016/j.entcs.2005.02.055

1 Introduction

Machine learning of grammars finds many applications in software engineering, syntactic pattern recognition, computational biology, computational linguistic, speech recognition, natural language acquisition, etc. For example, software engineers usually want to recover grammar from legacy systems in order to automatically generate various software analysis and modification tools. Usually in this case the grammar can be semi-automatically recovered from compilers and language manuals [12]. In application areas outside software engineering, grammars are mainly used as an efficient representation of artifacts that are inherently structural and/or recursive (e.g. neural networks, structured data and patterns) [19]. Here compilers and manuals do not exist and semi-automatic grammar recovery as suggested in [12] is not possible. The grammar needs to be extracted solely from artifacts represented as sentences/programs written in some unknown language.

The grammar induction problem also provides interesting theoretical undertones. Grammatical inference (grammar induction or language learning), a subfield of machine learning, is the process of learning of grammar from training data. One of the most important theoretical results in this field is Gold's theorem [7] which states that it is impossible to identify any of the four classes of languages in the Chomsky hierarchy in the limit using only positive samples. Using both negative and positive samples, the Chomsky hierarchy languages can be identified in the limit. Intuitively, Gold's theorem can be explained by recognizing the fact that the final generalization of positive samples would be an automaton that accept all strings. Using only positive samples results in an uncertainty as to when the generalization steps should be stopped. This implies the need for some restrictions or background knowledge on the generalization process. Despite the dispiriting results, research has continued in this area. Learning algorithms have been developed that exploit knowledge of negative samples, structural information, or restrict grammars to some subclasses (e.g. even linear grammars, k-bounded grammars, structurally reversible languages, terminal distinguishable context-free languages, etc.) [15] where identification in the limit is possible only from positive samples.

Our work is also related to renovation and legacy systems where renovation tools can be rapidly built once a grammar is available. However, current grammar inference techniques are not able to infer grammars of general-purpose programming languages (e.g. Cobol) [11]. By using the approach presented in this paper it is possible to infer grammars for small domain-specific languages.

The paper is organized as follows: Section 2 gives a short overview of the seminal results in the grammar inference literature. Details of the genetic approach to grammar inference and comparisons with the brute force approach are presented in Section 3. Section 4 shows a few experimental results of the genetic approach. The concluding comments and future work are mentioned in Section 5.

2 Related Work

So far, grammar inference has been mainly successful in inferring regular languages. Researchers have developed various algorithms (e.g., RPNI - Regular Positive and Negative Inference algorithm [22]) which can learn regular languages from positive and negative samples. Let S^+ and S^- denote the set of positive and negative samples from the language generated by the finite state automaton A. Automaton A is consistent with a sample $S = (S^+, S^-)$ if it accepts all positive samples and rejects all negative samples. A set S^+ is *structurally complete* with respect to automaton A if S^+ covers each transition of A and uses all final states of A as an accepting state [6]. Positive samples need to be structurally complete in order for the inference process to be successful; it is not possible to infer a transition in an automaton if the positive samples do not evidence it. In [5] a genetic approach was used for inferring grammars of regular languages only and compared with the RPNI which can identify any regular language in the limit. Experiments show that the genetic approach is comparable to other grammatical inference approaches.

Learning context-free grammars $G = (V, T, P, S)$ is more difficult than learning regular grammars. Using *representative* positive samples (that is, positive samples which exercise every production rule in the grammar) along with negative samples did not result in the same level of success as with regular grammar inference. Hence, some researchers resorted to using additional knowledge to assist in the induction process. Sakakibara [25] used a set of skeleton derivation trees (unlabelled derivation trees), where the input to the learning process are *complete structured sentences*, that is sentences with parentheses inserted to indicate the shape of the derivation tree of the grammar. It was shown in [25] that in this case, learning CFG's was possible from positive samples only. Since the grammatical structure (topology) was known, the problem of learning CFG's could be reduced to the problem of labeling non-terminals (similar to the partitioning problem of non-terminals). Recently an enhancement to this algorithm was proposed [26], where learning CFG's was possible from partially structured sentences (some pairs of left and right parentheses missing). For example, a completely structured sample for the language $L = \{a^m b^m c^n | m, n \geq 1\}$ is $((a(ab))b)(c(cc))$, while some partially structure samples are $(a(ab)b)(c(cc))$, $(a(ab)b)(ccc)$, and $(aabb)(ccc)$.

However, in the area of programming languages it is impractical to assume that completely or partially structured samples exist.

Despite the fact that many researchers have looked into the problem of CFG induction [13] [14] [20] [21] there has been no one convincing solution to the problem as of now. In all the work cited so far, experiments were performed on theoretical sample languages such as $L = \{ww|w \in \{a,b\}^+\}$, $L = \{w = w^R|w \in \{a,b\}^+\}$, $L = \{\#_a(w) = \#_b(w)|w \in \{a,b\}^+ \wedge \#_x(w)$ is a number of $x's$ in $w\}$, and $L = \{ab^icb^ia|i \geq 1\}$, instead of on real or even toy programming languages. Therefore, learning CFG's is still a real challenge in grammatical inference [8].

MARS [9] is a semi-automatic inference system in the area of Domain-specific modeling (DSM). DSM is an example of model-driven software engineering where domain experts can use high-level specifications to describe the solution of a problem in their domain using domain concepts. The motivation of the MARS project was to address the issue of metamodel drift, which occurs when instance models in a repository are separated from their defining metamodel. Making use of already existing tools along with new grammar inference algorithms, the MARS system recovers metamodels that correctly define the mined instance models.

3 Genetically Generated Grammars

To infer context-free grammars for domain-specific languages, the genetic programming approach was adopted. In genetic programming, a program is constructed from terminal set $\mathcal{T}$ and user-defined function set $\mathcal{F}$. The set $\mathcal{T}$ contains variables and constants and the set $\mathcal{F}$ contains functions that are a priori believed to be useful for the problem domain. In our case, the set $\mathcal{T}$ consists of terminal symbols defined with regular expressions and the set $\mathcal{F}$ consists of nonterminal symbols. From these two sets appropriate grammars (chromosomes) can be evolved, which can be seen as a domain-specific language for expressing the syntax. For effective use of an evolutionary algorithm we have to choose

- a suitable representation of the problem,
- suitable genetic operators
- control parameters, and
- the evaluation function to determine the fitness of chromosomes.

3.1 Representation

For the encoding of a grammar into a chromosome we used a direct encoding as a list of BNF production rules as suggested in [28] since this encoding outperforms bit-string representations.

3.2 Genetic Operators

Specific one-point crossover, mutation and heuristic operators have been proposed as genetic operators. The one-point crossover is performed in the following manner: two grammars are chosen randomly and are cut at the same random position; the second halves are then swapped between the two grammars. To ensure that after crossover the two offsprings are both legal grammars, the breakpoint position cannot appear in the middle of the production rule. The breakpoint position is chosen randomly from the smaller of two grammars selected for crossover. An example of the crossover operation is presented in Figure 1. After crossover, grammars undergo mutation, where a symbol in a randomly chosen production is mutated. An example of the mutation operator is presented in Figure 2.

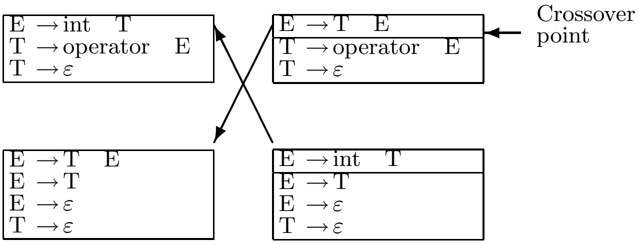


Fig. 1. The crossover operator

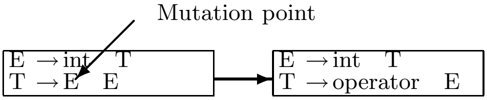


Fig. 2. The mutation operator

To enhance the search, the following heuristic operators have been proposed:

- option operator,
- iteration* operator and
- iteration+ operator

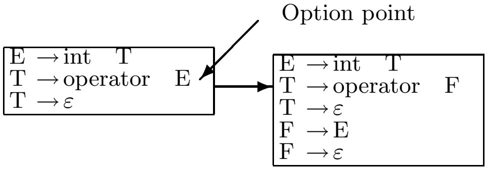


Fig. 3. The option operator

which exploit the knowledge of grammars, namely extended BNF (EBNF), where grammar symbols often appear optionally or iteratively. Heuristic operators work in a similar manner as the mutation operator. A symbol in a randomly chosen production can appear optionally or iteratively. An example of the option operator is presented in Figure 3.

Similar transformations on grammars are performed under iteration* and iteration+ operators. To ensure that after genetic operators a chromosome represents a legal grammar a special procedure is performed where non-reachable or superfluous nonterminal symbols are detected and eliminated.

3.3 Control parameters

In addition to standard parameters which control an evolutionary algorithm such as population size, number of generations, crossover and mutation probability, the following additional control parameters that prevent grammars from becoming too large have been introduced:

- max_prod_size: maximum number of productions of one grammar,
- max_RHS_size: maximum number of right-hand symbols of one production.

3.4 Fitness Function

Chromosomes were evaluated at the end of each generation by testing each grammar on a sample of positive and negative samples. For each grammar in the population an LR(1) parser was automatically generated using the compiler generator tool LISA [18]. The generated parser was then run on fitness cases. Figure 4 illustrates this process. A grammar's fitness value is proportional to the length of the correctly parsed positive samples. It is desirable to have a grammar which accepts all the positive samples and rejects all the negative samples.

Many grammars can be concocted which reject the negative samples. However, our search converges to the desired grammar better when we obtain grammars which accept the positive samples. Hence, it is a natural move to search in the space of all grammars which accept the positive samples, only. Negative samples are only taken into account when a grammar is capable of

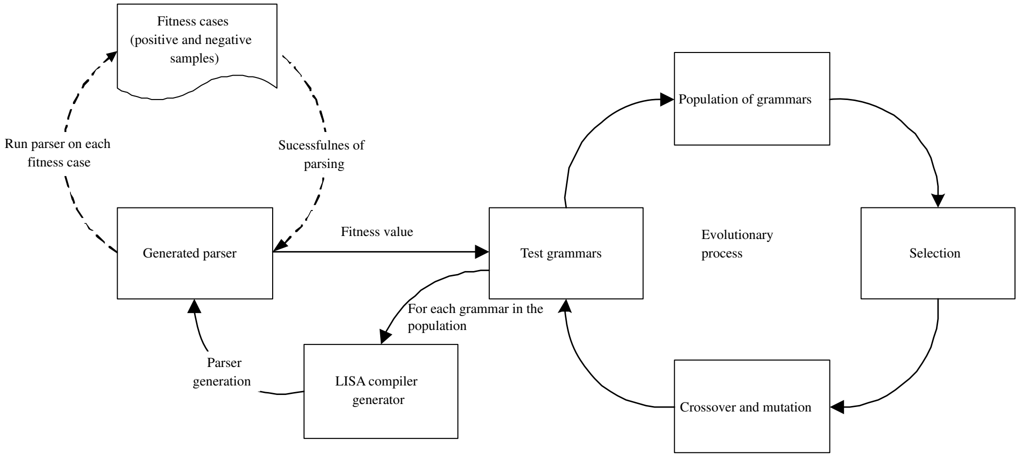


Fig. 4. The evaluation of chromosomes

accepting all the positive samples. Another reason is that negative samples are needed mainly to prevent overgeneralization of grammars [7]. Keeping these facts in view, the fitness value of each grammar is defined to be between 0 and 1, where interval 0 .. 0.5 denotes that the grammar did not recognize all positive samples and interval 0.5 .. 1 denotes that the grammar recognized all positive samples and did not reject all negative samples. A grammar with fitness value of 1 signifies that the generated LR(1) parser successfully parsed all positive samples and rejected all negative samples. For the given $grammar[i]$ its fitness $f_j(grammar[i])$ on the j-fitness case is defined as:

$$f_j(grammar[i]) = \frac{s}{length(program_j) * 2}$$

where $s = length(successfully\ parsed\ program_j)$ and $length(program)$ is a number of tokens in a given program.

The total fitness $f(grammar[i])$ is defined as:

$$f(grammar[i]) = \frac{\sum_{k=1}^{N} f_k(grammar[i])}{N}$$

where N = number of all positive samples

A grammar is tested on the negative samples set only if it successfully recognizes all positive samples. Here, the portion of successfully parsed negative sample is not important. Therefore, its fitness value is defined as:

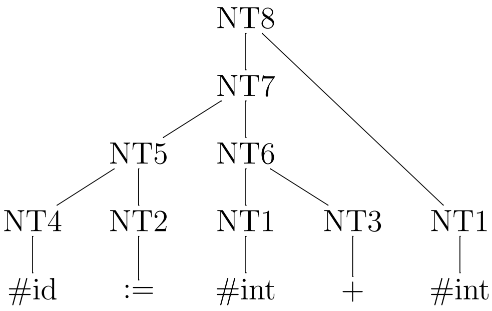
$$f(grammar[i]) = 1.0 - \frac{m}{M * 2}$$

where m = number of recognized negative samples
M = number of all negative samples

3.5 New ideas and approaches

In preliminary experiments [16] we observed that the heuristic operators considerably improved the search process, resulting in fitter induced grammars. However, we still were not able to induce bigger grammars despite the fact that the respective sub-grammars were previously induced in a separate process. For example, we were successful in finding a context-free grammar for simple expressions and and a grammar for simple assignment statements where the right hand expression can only be a numeric value. But, when both sub-languages were combined, our earlier approach failed to find a solution. Upon closer analysis of our results, it became clear that the randomly generated initial population was an impediment for the induction process. The search space of all possible grammars is expansive, and in order to increase the likelihood that the initial population contains well-situated individuals, the initial population should exploit knowledge from the positive samples by generating a few valid derivation trees by simple composition of consecutive symbols. For example, in fig. 5 one of the possible derivation trees and appropriate context-free grammars for positive sample $a := 9 + 2$ of the aforementioned example is presented. Of course, such a grammar will still not recognize all positive samples. But it is much closer to a suitable grammar than a randomly generated grammar.

Yet, in some other cases inference of the underlying context-free grammar was still not successful simply because composition of consecutive symbols is not always correct. What we need is to identify sub-languages and construct derivation trees for sub-programs first. But this is as hard as the original problem. Since using completely structured [25] or partially structured samples [26] are impractical we are using an approximation: frequent sequences. A string of symbols is called a *frequent sequence* if it appears at least θ times, where θ is some preset threshold. Our basic idea is to construct an initial derivation tree in which frequent sequences are recognized by a single nonterminal (see Table 2 where such a nonterminal starts with FR). Since a frequent sequence doesn't contain many symbols its derivation tree can be constructed using brute-force approach. In a brute-force approach all possible derivation trees are generated and checked for suitability. If we limit ourselves to binary trees, then all possible derivations trees can be calculated in a following manner. The number of external nodes (l) in a non-empty, full binary tree is one more than the number of internal nodes (n).



NT8 → NT7 NT1

NT7 → NT5 NT6

NT6 → NT1 NT3

NT5 → NT4 NT2

NT4 → #id

NT3 → +

NT2 → :=

NT1 → #int

Fig. 5. One possible derivation tree for positive sample a := 9 + 2

$$l = n + 1$$

In our case, internal nodes represent non-terminals and external nodes represent terminals in the derivation tree. Given a program the number of external nodes (l) is therefore known and consequently also the number of internal nodes (n). The number of all possible full binary trees with n internal nodes is given by the n-th Catalan number (C_n)[2].

$$C_n = \frac{(2n)!}{(n+1)!n!}$$

For example, there are 14 different full binary trees when $l = 5$ ($n = 4$), as shown in Figure 6 and Table 1).

Even for small n, the number of all possible full binary trees increases exponentially (Figure 7). For full binary trees to be valid derivation trees, the interior nodes need to be labeled with non-terminals. If n is the number of interior nodes, then there are n^n different ways to label the interior nodes. But we can do it even better. When $n = 3$ instead of $27(3^3)$ different labelling of nonterminals only 5 of them are distinct, while others can be obtained by consistent renaming of nonterminals. Actually, we are interested only in distinct labelling of nonterminals. Its enumeration is defined by Bell numbers [1]. For example $n = 2$, we get 4 $(2 * 2)$ derivation trees, as shown in Figure 8 and Table 1).

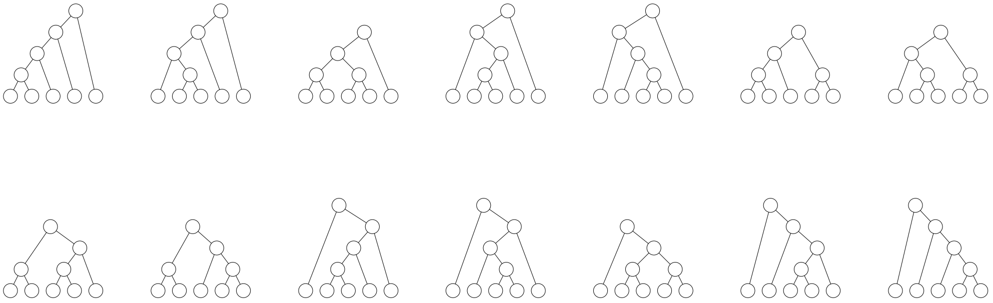


Fig. 6. All full binary trees for $l = 5$ ($n = 4$)

Table 1
Search space with labeling of nonterminals

n	Catalan number	Labeling of nonterminals (Catalan num. * Bell num.)	Search space (Catalan numbers * n^n)
1	1	1	1
2	2	4	8
3	5	25	135
4	14	210	3584
5	42	2184	131250
6	132	26796	6158592
7	429	376233	3.53299947 E8
8	1430	5920200	2.399141888 E10
9	4862	1.02816714 E8	1.883638417518 E12
10	16796	1.9479161 E9	1.6796 E14
11	58786	3.989041602 E10	1.6772331868538246 E16
12	208012	8.76478739164 E11	1.85465588644262707 E18
13	742900	2.05370522473 E13	2.2500591668738474 E20
14	2674440	5.1054878272968 E14	2.9718395534545382 E22

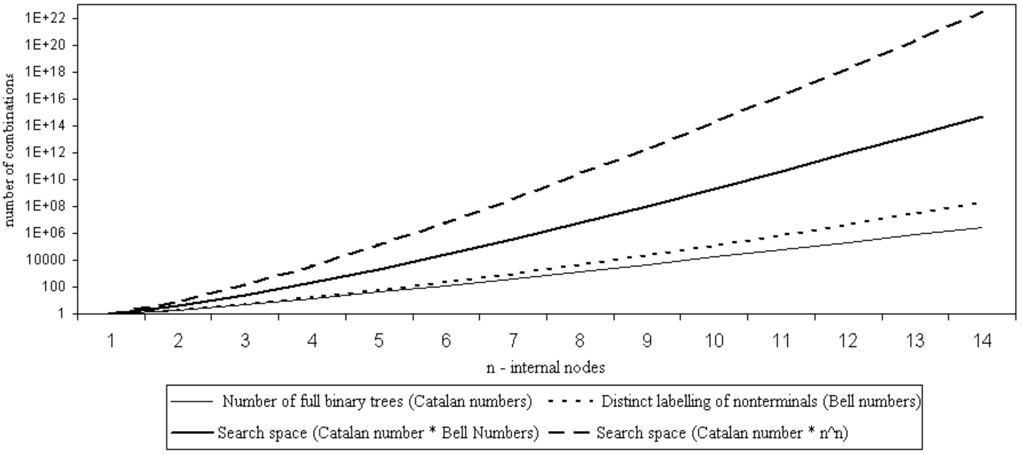


Fig. 7. Reduced search space in brute force approach

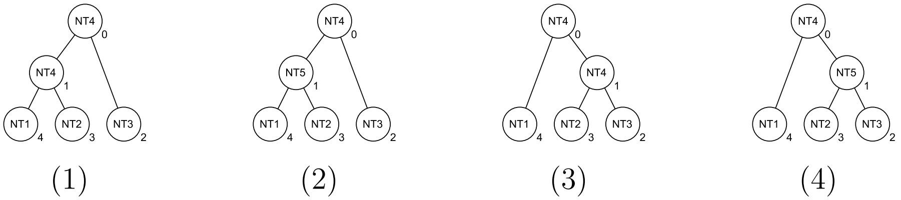
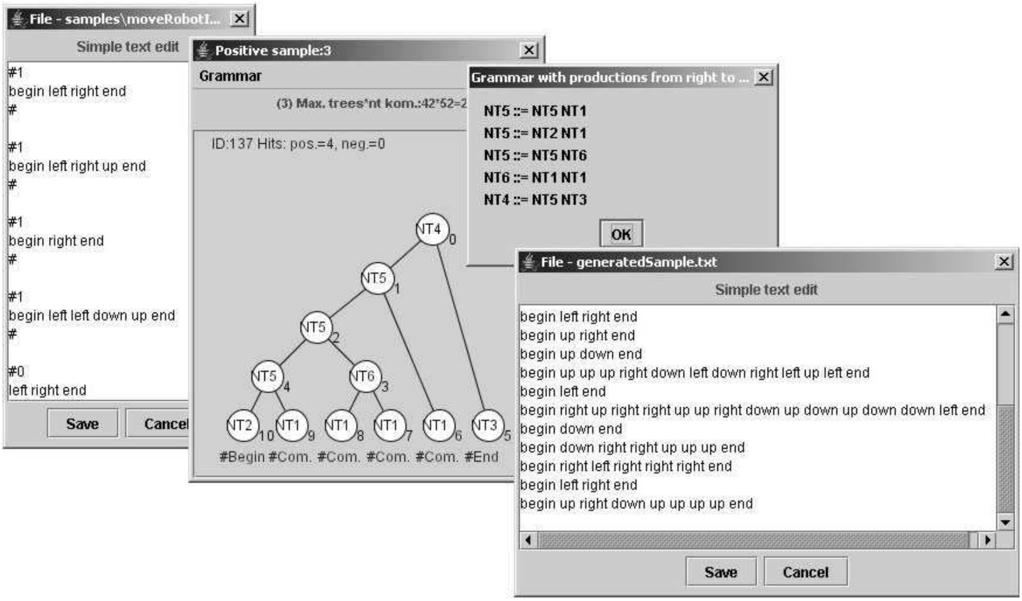


Fig. 8. All distinct labelings of nonterminals when $l = 3$ $(n = 2)$

Based on these observations a grammar inference tool called GIE-BF (Figure 9) has been implemented. The GIE-BF employs the brute force approach



```
#1
begin left right end
#

#1
begin left right up end
#

#1
begin right end
#

#1
begin left left down up end
#

#0
left right end
```

(3) Max. trees*nt kom.:42*52=2

ID:137 Hits: pos.=4, neg.=0

```
NT5 ::= NT5 NT1
NT5 ::= NT2 NT1
NT5 ::= NT5 NT6
NT6 ::= NT1 NT1
NT4 ::= NT5 NT3
```

```
begin left right end
begin up right end
begin up down end
begin up up up right down left down right left up left end
begin left end
begin right up right right up up right down up down up down down left end
begin down end
begin down right right up up up end
begin right left right right right end
begin left right end
begin up right down up up up up end
```

Fig. 9. GIE-BF tool

for CFG learning. From a given set of positive and negative samples, a valid derivation tree is systematically searched for. The search is stopped when either the derivation tree found produces a context-free grammar in CNF that accepts all positive samples and reject all negative samples, or when the search space is exhausted. The induced CFG is then used to generate samples (programs), the analysis of which can indicate if more negative samples are needed in the starter set; in the event that more negative samples are needed, we deduce that the inferred CFG is over-generalized. Using GIE-BF we were able to find CFG's for small sub-languages or frequent sequences.

4 Experimental Results

Using an evolutionary approach enhanced by grammar-specific heuristic operators and by better construction of the initial population we were able to infer grammars for small domain-specific languages [17] such as video store [24], stock and sales [24], simple desk calculation language - DESK [23], and a simplified version of feature description language (FDL) [4]. Inferred grammars as well as some other parameters (G - number of generations when solution is found, pop_size – population size, N – number of positive samples, M – number of negative samples) are shown in Table 2.

Due to space limitation not all positive and negative samples as well as control parameters are shown in Table 2. This is done below only for the DESK example (Table 3).

All inferred grammars in our experiment (Table 2) accept all positive samples and reject all negative samples, hence achieving our goal. But actually how good are the inferred grammars? Our measure of how "good" an inferred grammar is defined as the number of extraneous non-terminals and productions in it, while describing the same language as the original grammar. Table 4 shows the original and inferred grammars, while Table 5 compares between the original and inferred grammars. It can be noticed that inferred grammars have a few more productions and non-terminal symbols than the original grammars. However, in many cases direct relations can be found (Table 4) among productions and terminals (e.g., $ITEM = FR2$, $SALES = NT3$, $SALE = FR0$ are nonterminals that match in the first example; see also Figure 10). Therefore, we can conclude that inferred grammars are of good quality and that the correct structures are captured. Notice that inferred grammars can not be ambiguous since such grammars are rejected in the evaluation process. Since an LR(1) parser can not be generated for such grammars, they are assigned a fitness value of 0.

Table 2
Inferred grammars for some DSLs

DSL	G	pop_size	N	M	Inferred grammar	An example of positive sample
stock and sales	1	300	10	3	NT6 → NT5 NT3 NT5 → NT4 #sales NT4 → #stock NT2 NT2 → FR2 NT2 NT2 → ε FR2 → #item #price #qty NT3 → FR0 NT3 NT3 → ε FR0 → #item #price	stock description twix 0.70 10 mars 0.65 12 bar 1.09 5 sales description mars 0.65 twix 0.70 mars 0.65
DESK	1	300	5	3	NT7 → NT6 NT3 NT6 → NT5 #where NT5 → NT4 NT2 NT4 → #print #id NT3 → FR4 NT3 NT3 → ε NT2 → FR3 NT2 NT2 → ε FR4 → #id #assign #int FR3 → #plus #id	print a + b where a = 10 b = 20
video store	28	300	8	8	NT15 → NT11 NT7 NT15 NT15 → ε NT11 → NT10 NT6 NT10 → NT5 NT10 NT10 → ε NT7 → NT5 NT7 NT7 → ε NT6 → #name #days NT5 → #title #type	jurassicpark child roadtrip reg ring new andy 3 jurassicpark child roadtrip reg ann 2 ring new
FDL	127	300	8	4	NT7 → NT2 NT7 NT7 → ε NT2 → NT1 FR8 NT1 → #feature #: FR8 → #op #(NT11 #, NT11 #) NT11 → #feature NT11 → FR8	c : all(c1, more-of(f4, f5)) c1 : one-of(f1, c2) c2 : all(f4, f5)

Table 3
Positive/negative samples and some control parameters for DESK example

<table>
<tr><th colspan="2">Input samples</th></tr>
<tr><th>positive</th><th>negative</th></tr>
<tr><td>print b where b = 20 c=1 d=22
print b + b + b + b where b = 20
print a + b where a = 10 b = 20
print a + b where a = 10 b = 20 c=11
print a + b + b + a where a = 10 b = 20</td><td>print where a = 10
a + b where a = 10 b = 20
print a + b a = 10 b = 20</td></tr>
<tr><th colspan="2">Control parameters</th></tr>
<tr><td colspan="2">max_RHS_size = 4
max_prod_size 15
pc = 0.1
pm = 0.2
pheuristic = 0.8
minLengthFrequency = 2
maxLengthFrequency = 3
thresholdFrequency = 8</td></tr>
</table>

All grammars in our experiment have been inferred using a relatively small numbers of positive and negative samples (see Table 2, columns 4 and 5). In our opinion, this is due to a small number of alternative productions in the original grammars. All possible alternative productions were covered by a small number of positive samples. To infer larger grammars, a much bigger set of positive and negative samples would be required. More than the number of positive samples, it is more important how the positive samples are constructed. Positive samples have to be representative, i.e., exercise every production rule in the grammar. Therefore, the number of positive samples is in correlation with the number of alternative productions in original grammar. However, this is not known in advance. It is important that positive samples exhibit the variety of sentences that can be generated by the grammar. Figuring out the number of negative samples required is a more complicated issue. We didn't experiment with how large the set of positive and negative samples should be in order to be able to infer the correct grammar. We theorize that this can be done by simply removing/adding positive and negative samples from the starting set.

Another issue is if it is reasonable to expect that the positive and negative samples are always available. In the domain of programming languages these two sets are easily obtainable. To support this process the GIE-BF tool enable us to automatically generate samples from the inferred grammar (see right most window in Fig. 9). If in such a set a sample is generated that should not intended to be positive, we classify this as case of grammar overgeneralization and the specific sample is included in the set of negative samples. As a consequence, the whole grammar inference process has to be restarted with the new set of samples. Our current research focus is on investigating how to incrementally adapt the inferred grammars to the violating sample without re-starting the whole inference process. However, negative samples might be hard to obtain in other domains such as legacy systems and domain-specific modeling [9]. Since negative samples are crucial in inference of regular and context-free grammars [7], in these other domains alternative knowledge needs to be exploited in order to control grammar overgeneralization.

Last but not least, it is important that the grammar inference process be expeditious. In our experiments the population usually contains around 300 grammars; for each grammar an LR(1) parser has to be generated and tested on positive and negative samples (in our experiments each generated parser is run on average on 12 samples). Evaluation of one generation takes approximately 6 seconds or 10 generations in 1 minute on a typical PC (Pentium III, 1.2 GHz and 1 Gb RAM). For example, the "video store" grammar was inferred in 3 minutes and the FDL grammar in about 13 minutes.

So far, we have been able to infer grammars for small domain-specific languages which are bigger in size and more pragmatic than in other research efforts. We are convinced that this approach, when enhanced with other data mining techniques and heuristics, is scalable and feasible to infer grammars of realistically sized languages belonging to different programming paradigms [27], such as procedural, functional, object-oriented, logic, and process functional [10]. Although the current results are promising, we intend to investigate other techniques to extend this work.

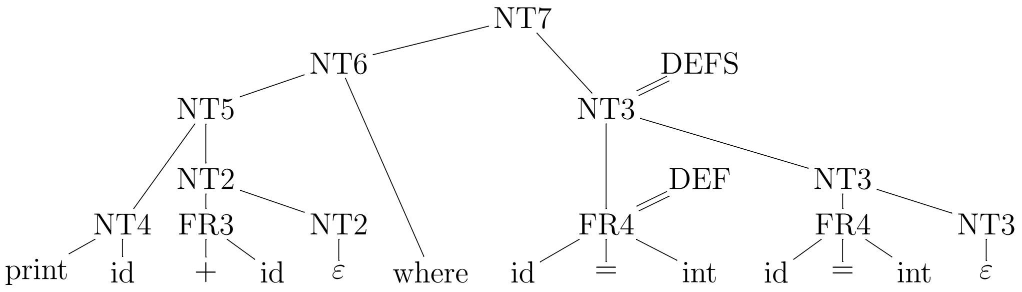


Fig. 10. Derivation tree for a program written in DESK language

Table 4
Inferred and original grammars from Table 2

Original grammar	Inferred grammar	Equivalence

VendingMachine →
#stock STOCK #sales SALES
STOCK → STOCK ITEM | ε
ITEM → #item #price #qty
SALES → SALES SALE | ε
SALE → #item #price
	NT6 → NT5 NT3
	NT5 → NT4 #sales
	NT4 → #stock NT2
	NT2 → FR2 NT2
	NT2 → ε
	FR2 → #item #price #qty
	NT3 → FR0 NT3
	NT3 → ε
	FR0 → #item #price
		FR2 = ITEM
		NT3 = SALES
		FR0 = SALE

DESK → #print EXPR CONST
EXPR → EXPR #plus #id | #id
CONST → #where DEFS
DEFS → DEFS DEF | ε
DEF → #id #assign #int
	NT7 → NT6 NT3
	NT6 → NT5 #where
	NT5 → NT4 NT2
	NT4 → #print #id
	NT3 → FR4 NT3
	NT3 → ε
	NT2 → FR3 NT2
	NT2 → ε
	FR4 → #id #assign #int
	FR3 → #plus #id
		NT3 = DEFS
		FR4 = DEF

Videostore → MOVIES USERS
MOVIES → MOVIES MOVIE | ε
MOVIE → #title #type
USERS → USERS USER | ε
USER → #name #days RENTALS
RENTALS →
RENTALS RENTAL | ε
RENTAL → MOVIE
	NT15 → NT11 NT7 NT15
	NT15 → ε
	NT11 → NT10 NT6
	NT10 → NT5 NT10
	NT10 → ε
	NT7 → NT5 NT7
	NT7 → ε
	NT6 → #name #days
	NT5 → #title #type
		NT10 = MOVIES
		NT5 = MOVIE
		NT7 = RENTALS

FDL → FDEF FDL | ε
FDEF → #feature #: FEXP
FEXP → #op #(FLIST #)
FLIST →
FEXP #, FEXP | #feature
	NT7 → NT2 NT7
	NT7 → ε
	NT2 → NT1 FR8
	NT1 → #feature #:
	FR8 → #op #(NT11 #,
	NT11 #)
	NT11 → #feature
	NT11 → FR8
		NT2 = FDEF
		FR8 = FEXP

Table 5
Comparison of inferred and original grammars from Table 4

DSL	Number of Nonterminals		Number of Productions	
	Original grammar	Inferred grammar	Original grammar	Inferred grammar
stock and sales	5	7	7	9
DESK	5	8	7	10
Video store	7	6	10	9
FDL	4	5	6	7

5 Conclusions and Future Work

Previous attempts at learning context-free grammars resulted in ineffectual success on real examples. We extended this work by introducing grammar-specific heuristic operators and facilitating better construction of the initial population where knowledge from positive samples has been exploited. Our future work involves exploring the use of data mining techniques in grammar inference, augmenting the brute force approach with heuristics, and investigating the Support Vector Machine (SVM) classification technique for context-free grammar inference [3].

References

[1] E.T. Bell. Exponential numbers. *Amer. Math*, 41:4111–419, 1934.

[2] Jonathan M. Borwein and David H. Bailey. *Mathematics by Experiment: Plausible Reasoning in the 21st Century.* A. K. Peters, Ltd., Wellesley, MA, USA, 2004.

[3] Nello Cristianini and John Shawe-Taylor. *An Introduction to Support Vector Machines and other Kernel-Based Learning Methods.* Cambridge University Press, 2000.

[4] A. van Deursen and P. Klint. Domain-specific language design requires feature descriptions. *Journal of Computing and Information Technology*, 10(1):1–18, March 2002.

[5] P. Dupont. Regular grammatical inference from positive and negative samples by genetic search: the GIG method. In Rafael C. Carrasco and Jose Oncina, editors, *Proceedings of the Second International ICGI Colloqium on Grammatical Inference and Applications*, volume 862 of *LNAI*, pages 236–245, Berlin, September 1994. Springer Verlag.

[6] P. Dupont, L. Miclet, and E. Vidal. What is the search space of the regular inference? In *Grammatical Inference and Applications, Second International Colloquium, ICGI-94 Alicante, Spain, September 21 - 23, 1994; Proceedings*, volume 862 of *Lecture Notes in Computer Science*, pages 25–37, 1994.

[7] E. M. Gold. Language identification in the limit. *Information and Control*, 10:447–474, 1967.

[8] C. de la Higuera. Current trends in grammatical inference. In F. J. Ferri et al., editors, *Advances in Pattern Recognition, Joint IAPR International Workshops SSPR+SPR'2000*, volume 1876 of *LNCS*, pages 28–31. Springer, 2000.

[9] Faizan Javed, Marjan Mernik, Jing Zhang, Jeff Gray, and Barrett Bryant. MARS: A metamodel recovery system using grammar inference. Technical report, University of Alabama at Birmingham, http://www.cis.uab.edu/softcom/GenParse/mars.htm, 2004.

[10] Ján Kollár. Unified approach to environments in a process functional programming language. *Computers and Informatics*, 22(6):439–456, 2003.

[11] Ralf Lämmel and Chris Verhoef. Cracking the 500-language problem. *IEEE Software*, 18(6):78–88, November/December 2001.

[12] Ralf Lämmel and Chris Verhoef. Semi-automatic grammar recovery. *Software—Practice & Experience*, 31(15):1395–1438, December 2001.

[13] Pat Langley and Sean Stromsten. Learning context-free grammars with a simplicity bias. In *Machine Learning: ECML 2000, 11th European Conference on Machine Learning, Barcelona, Catalonia, Spain, May 31 - June 2, 2000, Proceedings*, volume 1810 of *Lecture Notes in Artificial Intelligence*, pages 220–228. Springer, 2000.

[14] J. A. Laxminarayana and G. Nagaraja. Inference of a subclass of context free grammars using positive samples. In *ECML/PKDD 2003 Workshop on Learning Context-Free Grammars*, 2003.

[15] L. Lee. Learning of context-free languages: a survey of the literature. Technical Report TR-12-96, Center for Research in Computing Technology, Harvard University, Cambridge, MA, 1996.

[16] Marjan Mernik, Goran Gerlič, Viljem Žumer, and Barrett Bryant. Can a parser be generated from examples? In *Proceedings of the ACM Symposium on Applied Computing, Melbourne*, pages 1063–1067, 2003.

[17] Marjan Mernik, Jan Heering, and Anthony Sloane. When and how to develop domain-specific languages. Technical report, CWI Report SEN-E0309, http://ftp.cwi.nl/CWIreports/SEN/SEN-E0309.pdf, 2003.

[18] Marjan Mernik, Mitja Lenič, Enis Avdičaušević, and Viljem Žumer. LISA: An Interactive Environment for Programming Language Development. In Nigel Horspool, editor, *11th International Conference on Compiler Construction*, volume 2304, pages 1–4. Lecture Notes in Computer Science, Springer-Verlag, 2002.

[19] Marjan Mernik, Matej Črepinšek, Tomaž Kosar, Damijan Rebernak, and Viljem Žumer. Grammar-based systems: Definition and examples. *Informatica*, 28(3):245–255, November 2004.

[20] Katsuhiko Nakamura and Takashi Ishiwata. Synthesizing context free grammars from sample strings based on inductive CYK algorithm. In *Grammatical Inference: Algorithms and Applications, 5th International Colloquium, ICGI 2000, Lisbon, Portugal, September 11 - 13, 2000; Proceedings*, volume 1891 of *Lecture Notes in Artificial Intelligence*, pages 186–195. Springer, 2000.

[21] T. Oates, T. Armstrong, J. Harris, and M. Nejman. Leveraging lexical semantics to infer context-free grammars. In *ECML/PKDD 2003 Workshop on Learning Context-Free Grammars*, 2003.

[22] J. Oncina and P. Garcia. Inferring regular languages in polynomial update time. In N. Pérez de la Blanca, A. Sanfeliu, and E. Vidal, editors, *Pattern Recognition and Image Analysis*, volume 1 of *Series in Machine Perception and Artificial Intelligence*, pages 49–61. World Scientific, Singapore, 1992.

[23] Jukka Paakki. Attribute grammar paradigms — A high-level methodology in language implementation. *ACM Computing Surveys*, 27(2):196–255, June 1995.

[24] Maria Varanda Pereira, Marjan Mernik, Tomaž Kosar, Pedro Henriques, and Viljem Žumer. Object-oriented attribute grammar based grammatical approach to problem specification. Technical report, University of Braga, Department of Computer Science, http://marvin.uni-mb.si/technical-report/oomadg2004.pdf, 2002.

[25] Yasubumi Sakakibara. Efficient learning of context-free grammars from positive structural examples. *Information and Computation*, 97(1):23–60, March 1992.

[26] Yasubumi Sakakibara and Hidenori Muramatsu. Learning context-free grammars from partially structured examples. In *Grammatical Inference: Algorithms and Applications, 5th International Colloquium, ICGI 2000, Lisbon, Portugal, September 11 - 13, 2000; Proceedings*, volume 1891 of *Lecture Notes in Artificial Intelligence*, pages 229–240. Springer, 2000.

[27] D. A. Watt. *Programming Language Concepts and Paradigms.* Prentice-Hall, 1991.

[28] P. Wyard. Representational issues for context free grammar induction using genetic algorithms. *Lecture Notes in Computer Science*, 862:222–235, 1994.