# Can a Parser be Generated from Examples?

Marjan Mernik, Goran Gerlič, Viljem Žumer

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

{marjan.mernik, goran.gerlic, zumer}@uni-mb.si

Barrett R. Bryant

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

bryant@cis.uab.edu

## Keywords

Grammar induction, evolutionary algorithm, parser generation.

## Abstract

One of the open problems in the area of domain-specific languages is how to make domain-specific language development easier for domain experts not versed in a programming language design. Possible approaches are to build a domain-specific language from parameterized building blocks or by language (grammar) induction. This paper uses an evolutionary approach to grammar induction. Grammar-specific genetic operators for crossover and mutation are proposed to achieve this task. Suitability of the approach is shown by small experiments where underlying grammars are successfully genetically obtained and parsers are than automatically generated.

## 1. Introduction

Genetic programming [6] is a successful technique for getting computers to automatically solve problems. It has been successfully used in a wide variety of problems [1] such as data mining, text and image classification, robotic control, etc. In general, genetic programming works well for problems where solutions can be expressed with a modestly short program. For example, methods working on typical data structures such as stacks, queues and lists have been successfully evolved using genetic programming in [8]. However, we could not expect that large and complex programs (e.g. parsers/compilers) can be genetically evolved using current genetic programming techniques. Therefore, our goal to genetically generate a compiler for a domain-specific language seems to be too hard a problem using just the genetic programming approach. However, it is a well known fact that not just a complete compiler/interpreter but also other language-based tools such as language-knowledgeable editors, type checkers, animators, etc. can be automatically generated from formal language specifications [4]. Such formal language specifications for general-purpose programming languages are still too large. On the other hand, specifications for domain-specific languages, also called little languages, are small enough that we can expect that a successful solution can be found using genetic programming.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SAC 2003, Melbourne, Florida, USA

A programming language is formally specified with lexical, syntax and semantic specifications. For lexical and syntax specifications almost standard formalisms exist: regular definitions and BNF (Backus-Naur Form). No such standard method exists for semantic specifications. The most convenient formalisms are attribute grammars, operational semantics, denotational semantics, and algebraic specifications. Nowadays many sophisticated compiler generators exist, which automatically generate a compiler/interpreter and other language-based tools from formal specifications. However, writing language specifications is not an easy task. Since domain (application) experts are not also language design experts one of the possible solutions is that such specifications can be evolved using genetic programming. Combining genetic programming and specification languages is a novel approach in genetic programming as stated in [1, page 83]: "Would it be possible to evolve programs on the level of an algebraic specification that only later on would be translated into code? To our knowledge, nobody has tried GP on this level yet, but it might be a fruitful area to work on." In presented research work we are using LISA specifications [10, 11], which can be seen as a domain-specific language for expressing the syntax and semantics of the underlying language.

## 2. Related Work

The impact of different representations of grammars was explored in [14] where experimental results shows that an evolutionary algorithm using standard context-free grammars (BNF) outperforms those using Greibach Normal Form (GNF), Chomsky Normal Form (CNF) or bit-string representation [9]. The reason is that GNF and CNF tend to have more rules and more symbols than the equivalent grammar in standard form and hence the grammar search space is larger. Moreover, standard form (BNF) is likely to possess better building blocks on which the evolutionary algorithm may search the problem space than GNF or CNF. The experimental assessment in [14] was very limited due to the large processing time (processing of one generation had taken several hours; using our system processing of one generation takes just few seconds). This was due to use of the chart parser, which is used commonly in a natural language parsing and can accept also ambiguous grammars. With this approach a grammar was successfully inferred for the language of correctly balanced and nested brackets.

In [2] a genetic approach was used for inferring grammars of regular languages only and compared with the RPNI (Regular Positive and Negative Inference) algorithm [12] which can identify any regular language in the limit. Experiments show that the genetic approach is comparable to other grammatical inference approaches. Grammatical inference (grammar induction or

© 2003 ACM 1-58113-625-0/03/03...$5.00.

language learning) is a process of learning of a grammar from training data. Researchers have developed various algorithms, which all have in common that the largest class of languages, which can be efficiently learned by provably converging algorithms are regular languages. In the case of learning context-free grammars more information is required than a set of positive and negative samples, e.g. a set of skeleton parse trees [13]. Therefore, learning context-free grammars is still a real challenge in grammatical inference [5].

Machine learning of grammars finds many applications in syntactic pattern recognition, computational biology, natural language acquisition, etc. In this paper a new application of grammatical inference is suggested. Development of domain-specific languages is a hard problem for domain experts not versed in a programming language design. We believe that syntax of a small domain-specific language can be inferred from positive and negative programs provided by domain experts. Our work is also related to renovation and legacy systems where renovation tools can be rapidly build once a grammar is available. However, current grammar inference techniques are not able to infer grammars of general-purpose programming languages (e.g. Cobol) [7].

## 3. Genetically Generated Grammars

Genetic programming (GP) is the latest evolutionary approach in which an evolving population consists of computer programs. Each member of the population, a chromosome, represents a possible solution in the search space of all possible programs. Since the search space, all possible programs written in a chosen programming language, is too large it is restricted by the user-defined function set $\mathcal{F}$ and the terminal set $\mathcal{T}$. The set $\mathcal{T}$ contains variables and constants and the set $\mathcal{F}$ functions that are a priori believed to be useful for the problem domain [6]. The selection of sets $\mathcal{T}$ and $\mathcal{F}$ is one of the most important tasks in genetic programming since functions, variables and constants should be powerful enough to be able to represent a solution to the problem. In our case the population consists of LISA language specifications from which a parser is automatically generated. Language specifications consist of lexical, syntax and semantic specifications. To obtain quick preliminary results and to show that our approach is feasible we start our work on grammar induction first. The other reason is that semantics can be defined only when syntax is already correctly inferred. Hence, in the discussion below only the grammar induction and genetically generated grammars are discussed.

For effective use of an evolutionary algorithm we have to choose a suitable representation of the problem, suitable genetic operators and parameters, and the evaluation function to determine the fitness of chromosomes. All of these are discussed in a sequel. Our GP system has the following parameters:

- standard GP parameters
  - G: maximum number of generations
  - pop_size : size of the population
  - $p_c$: crossover probability
  - $p_m$: mutation probability
- parameters that prevent grammars from becoming too large
  - max_prod_size: maximum number of productions of one grammar
  - max_RHS_size: maximum number of right-hand symbols of one production
- $p_h$: heuristic operators probability (explained below)
- $p_d$: delete operator probability (explained below)

In addition to the above control parameters the input to every GP system are the set of terminals $\mathcal{T}$ and the set of functions $\mathcal{F}$. In our case, the set $\mathcal{T}$ consists of terminal symbols defined with regular expressions and the set $\mathcal{F}$ consists of nonterminal symbols. For example, if a grammar for arithmetic expressions with the only operator + is to be inferred, the following sets $\mathcal{T}$ and $\mathcal{F}$ might be appropriate:

```
𝒯 = {int, operator}
ℱ = {E, T}
```

where terminal symbols are defined with following regular expressions:

```
int       [0-9]+
operator \+
```

Our GP system starts with a population of randomly generated grammars. This initialization phase is done by the following algorithm:

```
for i = 1 to pop_size
  grammar_size[i] = randomNumber(1, max_prod_size)
  for j = 1 to grammar_size[i]
    LHS[i][j] = randomly select element from the set ℱ
    RHS_size[i][j] = randomNumber(0, max_RHS_size)
    for k = 1 to RHS_size[i][j]
      V[i][j][k] = randomly select element from the set 𝒯 ∪ ℱ
    endfor
    if RHS_size[i][j] = 0 then
      add production LHS[i][j] → ε
    else
      add production LHS[i][j]→V[i][j][1] ... V[i][j][RHS_size[i][j]]
  endfor
  create grammar[i]
endfor
```

## 3.1 Representation

For the encoding of a grammar into a chromosome we used a direct encoding as a list of production rules as suggested in [14] since this encoding outperforms bit-string representations. Among different context-free grammar formalisms such as Greibach Normal Form (GNF) and Chomsky Normal Form (CNF) the standard and the most general notation has been used (BNF).

## 3.2 Genetic Operators

The 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 with probability $p_c$. They are cut at the same random position and the second halves are swapped between two grammars. To ensure that after crossover the two offspring were both legal grammars the breakpoint position could not appear in the middle of the production rule. Furthermore, the breakpoint position is chosen randomly from the smallest of two grammars selected for crossover. Hence, grammars are able to change the length through crossover. An example of the crossover is presented in Fig. 1.

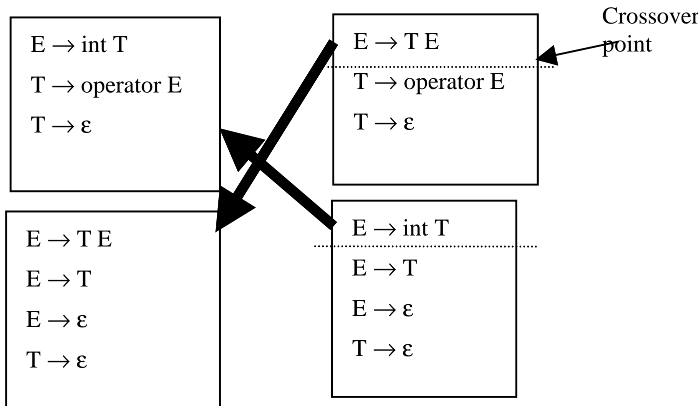


Fig. 1: The crossover operator

After crossover, all grammars undergo mutation. The chromosome `grammar[i]` is chosen randomly with probability $p_m$, next the production rule `j` (`j=1..grammar_size[i]`) and symbol `V[i][j][k]` (`k=0..RHS_size[i][j]`) are chosen randomly. In the case that the left hand nonterminal LHS is selected (`k=0`) then this nonterminal is changed to another nonterminal symbol. Otherwise the symbol is randomly replaced with any terminal or nonterminal symbol. An example of the mutation operator is presented in Fig. 2.

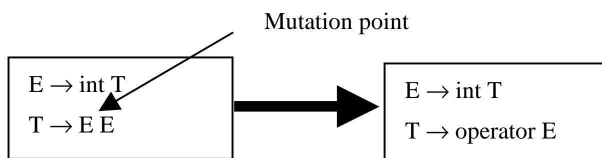


Fig. 2: The mutation operator

To ensure that after crossover and mutation (also in the initialization phase when chromosomes are randomly generated) a chromosome represents a legal grammar a special procedure is performed where non-reachable or superfluous nonterminal symbols are detected and eliminated.

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

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

which exploit the knowledge of grammars, namely extended BNF (EBNF), where often grammar symbols appear optionally or iteratively. Heuristic operators work in a similar manner as the mutation operator. The chromosome `grammar[i]` is chosen randomly with probability $p_h$, next the production rule `j` (`j=1..grammar_size[i]`) and symbol `V[i][j][k]` (`k=1..RHS_size[i][j]`) are chosen randomly. The symbol `V[i][j][k]` can then appear optionally or iteratively. The following transformation on a grammar is performed under the option operator:

```
LHS[i][j] → V[i][j][1]
         ... V[i][j][k] ...
             V[i][j][RHS_size[i][j]]
=> (option operator)
LHS[i][j] → V[i][j][1]
         ... W[i][j][k] ...
             V[i][j][RHS_size[i] [j]]
W[i][j][k] → V[i][j][k]
W[i][j][k] → ε
```

where the symbol `W[i][j][k]` is a fresh nonterminal symbol which does not appear before in `grammar[i]`. An example of the option operator is presented in Fig. 3.

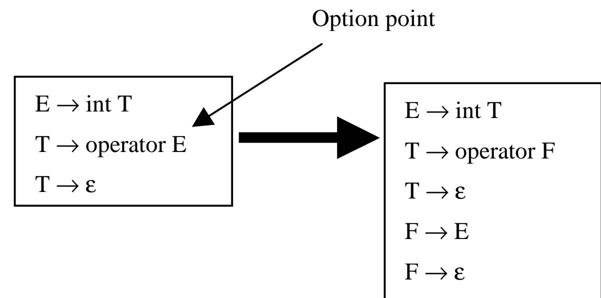


Fig. 3: The option operator

Similar transformations on grammars are performed under iteration* and iteration+ operators. In addition to heuristic operators the following delete operator is also useful. The job of the delete operator is to randomly remove a production from a grammar. The chromosome `grammar[i]` is chosen randomly with probability $p_d$, next the production rule `j` (`j=1..grammar_size[i]`) is chosen randomly and is deleted from the `grammar[i]`.

## 3.3 Evaluation of chromosomes

Chromosomes were evaluated at the end of each generation by testing each grammar on a sample of positive and negative statements. For each grammar in the population an LR(1) parser was automatically generated using the compiler generator tool LISA [11]. The generated parser was then run on fitness cases

(Fig. 4). A grammar's fitness value is proportional to the length of the correctly parsed positive sample (better grammars recognized a bigger portion of the positive sample). Further, we are using penalty functions for those grammars which recognize also negative samples. The fitness value of each grammar is between 0 and 1, where interval 0 .. 0.5 denotes that grammar did not recognized all positive samples and interval 0.5 .. 1 denotes that grammar recognized all positive samples and did not reject all negative samples. A grammar with fitness value 1 means that the generated LR(1) parser successfully parsed all positive samples and none negative samples. For the given `grammar[i]` its fitness $f_j$`(grammar[i])` on the j-fitness case is defined as:

$f_j(\texttt{grammar[i]}) = \texttt{s/(length(program}_j\texttt{)*2)}$

where

$\texttt{s = length(successfully parsed program}_j\texttt{)}$

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

$$f(\texttt{grammar[i]}) = (\textstyle\sum^{N}_{k=1} f_k(\texttt{grammar[i]}))/N$$

If a grammar correctly recognized all positive samples than it is tested also on negative samples. Its fitness value is defined as:

`f(grammar[i]) = 1.0 - (m/M*2)`

where

`m = number of recognized negative samples`

`M = number of all negative samples`

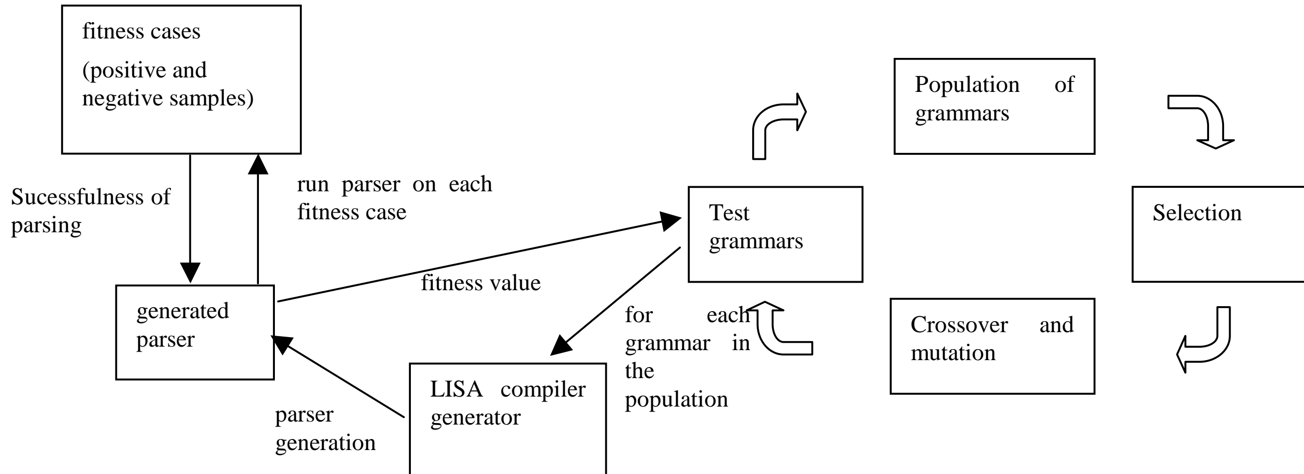


Fig. 4: The evaluation of chromosomes

## 4. Preliminary Results

Although we have already achieved some interesting experimental results our research work is still in a preliminary stage. Descriptions of our experiments follow.

Experiment No. 1: The evolution of a grammar for simple arithmetic expressions

G = 30, pop_size = 300, $p_c$ = 0.65, $p_m$ = 0.20,
max_prod_size = 4, max_RHS_size = 4, $p_h$ = 0, $p_d$ = 0
$\mathcal{T}$ = {int, operator} with regular definitions:
int [0-9]+
operator \+
$\mathcal{F}$ = {E, T, F}
pos_num_cases = 1
$prog_1$ = 5 + 20 +  4 + 12

Best solution was found (fitness value: 1) in generation no. 32:

F → #operator F
F → ε
F → #int F

This experiment clearly shows the well-known problem of over-generalization [3]. The generated grammar successfully parsed the given program; on the other hand it successfully parsed also many illegal arithmetic expressions such as: + 5 + or 5 4 3. Hence, fitness cases have to contain also negative samples. When negative samples (+ 2;  1 + 2 +;  + +) and heuristic operators have been included the following grammar has been obtained in generation no. 14:

F → #int T
T → #operator #int T
T → ε

Experiment No. 2: The evolution of a grammar for robot movement with reserved words

G = 50, pop_size = 300, $p_c$ = 0.5, $p_m$ = 0.1,
max_prod_size = 5, max_RHS_size = 4, $p_h$ = 0.3. $p_d$ = 0
$\mathcal{T}$ = {Command, Begin, End} with regular definitions:
Command left | right
Begin begin
End end
$\mathcal{F}$ = {S, E, T, F, R}
pos_num_cases = 3
$prog_1$ = begin left  right end
$prog_2$ = begin left  right left left end
$prog_3$ = begin left left right right right left left left end
neg_num_cases = 3
$prog_1$ = left right end
$prog_2$ = begin begin left right end
$prog_3$ = begin left left right

Best solution was found (fitness value: 1) in generation no. 10:

F → #Begin T #End
T → #Command T
T → epsilon

Experiment No. 3: The evolution of nesting block grammar

G = 50, pop_size = 300, $p_c = 0.55$, $p_m = 0.35$, $p_h = 0.3$,
max_prod_size = 6, max_RHS_size = 4
$\mathcal{T}$ = {Begin, End} with regular definitions:
Begin begin
End end
$\mathcal{F}$ = {S, E, T, F, R}
pos_num_cases = 3
$prog_1$ = begin end
$prog_2$ = begin begin end end
$prog_3$ = begin
begin end
begin begin end end
end
neg_num_cases = 4
$prog_1$ = begin
$prog_2$ = end
$prog_3$ = begin begin end
$prog_4$ = begin end end

Best solution was found (fitness value: 1) in generation no. 4:

E $\rightarrow$ #Begin T #End
T $\rightarrow$ E T
T $\rightarrow$ epsilon

Experiments show that heuristic operators greatly improve the search and induced grammars are better.

## 5. Conclusions and Future Work

Many programming language related problems such as renovation problems [7] and development of domain-specific languages could be solved by grammatical inference. However, learning context-free grammars is still a real challenge in grammatical inference [5]. They were some attempts of learning context-free grammars using evolutionary algorithms [9, 14] with little success on real examples. We extend these works by grammar-specific heuristic operators, which greatly improve the search and induce better grammars. We would like to pursue further research on grammar inference in a manner that not only syntax but also semantics can be inferred from a given set of data. This can be achieved with another genetic programming system applied on attribute grammar when a context-free grammar is already successfully induced. We suggest that this is not done simultaneously but rather as a two-stage process. To our knowledge nobody has tried this yet. Our future work is to automatically generate a complete compiler from examples, not just a parser.

## 6. References


[1] W. Banzhaf, P. Nordin, R.E. Keller, F.D. Francone. Genetic Programming: An Introduction. Morgan Kaufmann Publisher, 1998.

[2] P. Dupont. Regular Grammatical Inference from Positive and Negative Samples by Genetic Search: The GIG method. Lecture Notes in Artificial Intelligence, Vol. 862, Proceedings of the 2nd International Colloquium on Grammatical Inference and Applications, ICGI'94, pp. 236 -245, 1994.

[3] M.E. Gold. Language Identification in the Limit. Information and Control Vol. 10, pp. 447 – 474, 1967.

[4] P. Henriques, M. Varanda Pereira, M. Mernik, M. Lenič, E. Avdičaušević, V. Žumer. Automatic Generation of Language-based Tools. Electronic Notes in Theoretical Computer Science, Vol, 65, No. 3, Eds: Mark van den Brand and Ralf Laemmel, Elsevier Science Publisher, 2002.

[5] C. de la Higuera. Current Trends in Grammatical Inference. Advances in Pattern Recognition, Joint IAPR International Workshops SSPR+SPR'2000, Lecture Notes in Computer Science, Vol. 1876, pp. 28-31, 2001.

[6] J.R. Koza. Genetic Programming: On the Programming of Computers by Natural Selection. MIT Press, 1992.

[7] R. Laemmel, C. Verhoef. Cracking the 500-Language Problem. IEEE Software, Vol. 18, No. 6, pp. 78 - 88, 2001.

[8] W.B. Langdon. Genetic Programming and Data Structures: Genetic Programming + Data Structures = Automatic Programming!. Kluwer Academic Publishers, 1998.

[9] S. Lukas. Structuring Chromosomes for Context-Free Grammar Evolution. 1st International Conference on Evolutionary Computing, pp. 130 – 135, 1994.

[10] M. Mernik, M. Lenič, E. Avdičaušević, V. Žumer. Multiple attribute grammar inheritance. Informatica, vol. 24, no. 3, pp. 319-328, 2000.

[11] M. Mernik, M. Lenič, E. Avdičaušević, V. Žumer. LISA: An Interactive Environment for Programming Language Development. 11th International Conference on Compiler Construction, CC'2002, Ed: Nigel Horspool, Lecture Notes in Computer Science, Vol. 2304, pp. 1 – 4, 2002.

[12] J. Oncina, P. Garcia. Inferring regular Languages in Polynomial Update Time. Series in Machine Perception and Artificial Intelligence, Vol. 1, pp. 49 – 61, World Scientific, 1992.

[13] Y. Sakakibara. Efficient Learning of Context-Free Grammars from Positive Structural Examples, Information and Computation 97, pp. 23 –60, 1992

[14] P. Wyard. Representational Issues for Context Free Grammar Induction Using Genetic Algorithm. Lecture Notes in Artificial Intelligence, Vol. 862, Proceedings of the 2nd International Colloquium on Grammatical Inference and Applications, pp. 222 -235, 1994.