# Grammar Generation with Genetic Programming

## Evolutionary Grammar Generation

**Masterarbeit**
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Sandro De Zanet

Bern im Juli 2009

Leiter der Arbeit

Prof. Dr. Oscar Nierstrasz

Toon Verwaest

Institut für Informatik und angewandte Mathematik

Sandro De Zanet
dezanets@gmail.com

Software Composition Group
University of Bern
Institute of Computer Science and Applied Mathematics
Neubrückstrasse 10
CH-3012 Bern
http://scg.unibe.ch/

# Abstract

External domain specific languages are ubiquitous in computer science. Getting ahold of definitions of these languages and being able to analyze them is difficult. The code has to be parsed and transformed to a model before we can even start to retrieve meaningful information. Often a parser is not openly available or is written in an other language. Hence a developer analyzing the code has to manually figure out the grammar and write his own parser. This thesis will address the problem by automating the grammar and parser retrieval process. The approach uses a combination of *Parsing Expression Grammars* and *Genetic Programming*.

# Acknowledgements

I would like to thank the following persons who have made the completion of this Master Thesis possible: Prof. Oscar Nierstrasz, Toon Verwaest and the whole scg staff and fellow master students.

# Contents

# Chapter 1

# Introduction

As more and more software projects are developed, more and more legacy code is generated. Over time they grow big and it gets more difficult to keep a good overview. This leads to unmaintainable and error prone code. Developers who join a team need a lot of time to understand the legacy. Changes to such code are expensive in terms of time as well as money. With a fast and detailed understanding of software, cost can be minimized and the number of errors kept low. This is why it is important to be able to understand big software projects in less time.

To understand large software we need to analyze it. Normally this is done manually by skimming through the code and understanding more and more of the code. Although a developer will see more details of the project by doing it manually, it is difficult to keep track of the general overview of the project while trailing off into specifics of the implementation.

We need a way of automatically getting general data out of code (Lines of code, Number of methods, etc.). By processing this data, we can tell the programmer if there is something wrong with the source code and which parts have to be refactored. They show the most important parts of the projects and how they work together. There are a lot of different metrics tools to address the various problems and hot spots of software. Unfortunately for every language there has to be an export and import facility, which parses and analyzes the code. So for every language one needs to write a parser or at least write some sort of plugin for existing compilers, if available. This is tedious work and needs a lot of familiarization with the topic of parsing. There are more general tools like Moose [9], which use a language independent metamodel (FAMIX [3]) to define data and structures of object oriented languages. In this way metric analysis tools only have to be written once and can be reused. One still has to write a parser for every new language, though. With the ever increasing number of scripting languages and external Domain Specific Language (DSL), as well as new languages, it is a lot of work to write new exporters every time. In this work we will present a possible simplification and automation of the process of grammar extraction from source code.

In many cases neither the compiler nor the grammar of a language is freely available. If a grammar is available, sometimes it's not defined in a standard format like Backus Naur Form (BNF), from which a parser could be generated (semi-) automatically. Therefore if we want to analyze code, we need to write our own parser. While for simple DSLs the parsing process is usually straightforward and doesn't need a big experience, for more complex languages writing a parser is cumbersome work and often very time intensive. Programmers usually don't have the knowledge for parsing and therefore must acquire it. A lot of time spent on correct definitions of BNFs, grammar ambiguities and precedence rules. Mostly, programmers don't care about the inner workings of a parser and are simply interested in the meta data they can gain. Thus, the parsing overhead seems even more amiss.

## 1.1 Related work

There have been attempts to simplify and automate the process of extracting software meta-models from source code. Markus Kobel investigated the idea of direct mapping of source code to a formal representation of the considered source code [10]. In that case it was the aforementioned FAMIX metamodel. They let the user select meaningful subtexts from the source code. By building the corresponding metamodel object in FAMIX the program can map source code to the FAMIX model. For instance one can map a method signature in the code with the metamodel of method signature in FAMIX. Their prototype, CodeSnooper, tries to generate a local parser for the specified example. Naturally, the more examples you provide the better the parser gets (with the danger of overgeneralization). With their prototype they could extract metrics for elements of Java code. They had problems with Ruby, although this was more of a technical problem than one of the general approach.

A more general approach has been followed by Marjan Mernik, Goran Gerlič, Viljem Žumer, namely the generation of grammars for arbitrary languages [2]. They used Genetic Programming, an evolution based search algorithm, to find grammars for legacy source code written in a relatively simple programming language. In contrast to the work of Markus Kobel [10], their goal was to find a grammar to which the input source complies instead of a direct mapping to meta-models. Although with the grammar we don't directly extract data this approach is still very useful. A parser can be easily generated from a grammar. With the parser we can build an abstract syntax tree. The final mapping of an AST to a meta-models is then almost straightforward.

In their approach using Genetic Programming, the grammars were evolved directly rather than transforming them into special evolvable models. With this quite simple setup they achieved useful and compact grammars for small DSLs. For instance they evolved a grammar for simple arithmetic expressions as well as a robot steering language. To avoid the problem of the token definitions, users had to define them by hand. They are not part of the evolution process. It might not be that big an issue, since tokens are very similar from one language to the next.

## 1.2 Solution in a nutshell

In this thesis we will show a solution for the first step of software metamodel extraction. We will address the automatic generation of a grammar for an arbitrary language and thus its parser. We want to be as automatic as possible: we only want to rely on sample code. The process should automatically find a grammar which describes a programming language.

As we saw, there are different ways to approach this goal. A solution has to deal with the problem of finding patterns in an almost arbitrary stream of characters. The canonical way of analyzing such a stream is applying a formal grammar. Our goal is to find this grammar.

To find the grammar we could apply pattern recognition on the characters to find balanced parentheses, recurring syntactical patterns or keywords. There are some problems with this approach: keywords, for instance, are very difficult to detect through statistics, since names (of variables, methods, etc.) can recur as often as keywords. So it is difficult to identify patterns statistically. Instead of relying on statistics, we could define certain recurring patterns manually beforehand. This, however, limits the number of patterns that can be found. It will potentially miss syntactic features. Having to define these patterns beforehand involves a lot of work from the user as well, which breaks our premise of high automation.

For this thesis we decided to take another strategy than statistical analysis to find a suitable grammar. We however kept in mind that the user effort should be minimized. One can match patterns on the characters, but it's quite cumbersome. While some patterns match parts of the source, they don't fully describe the whole language. Only the part of the language which has been presented to the search algorithm is recognized. The missing parts have to be filled in with more general statements, which might not be restrictive enough. It is also difficult to know beforehand which patterns will occur in the source files. Programming languages use different kinds of white-spaces used for scoping, variables, etc.

Since there are a lot of equivalent grammars which fully describes a certain source code and since we don't have any additional constraints of the grammar beforehand, we need a searching algorithm which is able to find patterns in a seemingly random data set. The solution we use in this thesis is Genetic Programming, which was also used in *"Can a Parser be Generated From Examples?"* [2]. This special type of evolutionary algorithm uses the principles of biological evolution to search and optimize results for a given problem. This algorithm can improve very simple and bad solutions iteratively to get a more sophisticated result. Beginning with randomly generated grammars we can breed them to a sufficiently useful grammar. Like in nature, in Genetic Programming the grammars nearest to the aimed goal are selected to pass on to the next generation. The crucial part of the algorithm is precisely the function which selects a new generation of grammars, the so-called fitness function. It determines what the goal of the search is and thus directs the algorithm towards the solution.

The equivalent of the fitness function in nature would be for instance the survival probability of a creature's offspring.

Since the fitness function is so important, we will need to take a closer look at how to define it. It defines what constitutes a "good" grammar, what kind of grammars we want to achieve. On the one hand, we don't want too general grammars, while on the other hand we shouldn't let the grammars be too specialized towards the source code samples.

An additional issue is the definition of the data types, e.g. how the envisioned grammars should be encoded. We need an easily modifiable structure so we can evolve it with Genetic Programming. For the evolutionary approach, we have to be able to *gradually* change grammars. It is also important that we are able to test the fitness of the grammars in a simple and efficient way. Since in our case the fitness test involves a lot of parsing, the transformation between the definition of grammars and the corresponding parsers should be fast and easily achievable.

Usually grammars for computer languages are defined with Context Free Grammars using the (Extended) Backus Naur Form. As they are well known, the essential algorithms for parser generation and optimized parsing are available (Lexer, YACC, etc.). They usually compile grammars to *C* code, which represent the code for the parser corresponding to the grammar. This source code has to be compiled again to executed. Ambiguities resulting from the grammar have to be solved manually, which is difficult an error prone. This is a disadvantage for the envisioned automation of the process, since we want to hide the generation of parsers from the user. Unfortunately, ambiguous grammars cannot be discarded automatically. In fact this problem is undecidable.

Our approach for the grammar is to use Parsing Expression Grammars (PEG) [7]. A PEG is basically a composable grammar, recursively descent parsers. After it is defined (normally written as internal domain specific language) it can be used right away, without need of further transformation or generation of supplementary files. No additional parsing is needed. Furthermore, due to their greedy nature (unlike CFGs) they have no inherent ambiguities. This facilitates the automatic generation of grammars, since the user does not have to intervene. Due to their uniform nature, they are easily implemented and extended to suit our needs.

While naively implemented PEGs have potentially exponential time complexity they can be optimized to perform in linear time using Memoization.

## 1.3 Overview

In this thesis we will first look at the basics of the parsers and what search algorithm we will use. In chapter 2 we will have a look at the details of Parsing Expression Grammars and in chapter 3 at Genetic Programming and other evolutionary algorithms. We will combine both techniques in chapter 4. Furthermore, we will address optimization

problems as well as parameter calibration. The more interesting implementation details are depicted in chapter 2.4.1 along with their consequences for the results. Finally, we will discuss the outcome of the algorithm in chapter 5, where different real DSLs are taken to the test. Here we will see the limitations of the approach. To conclude we will summarize the results and present ideas for future work. This will be shown in chapter 6.

# Chapter 2

# Parsing Expression Grammars

Our goal is to find grammars for source code of certain programming languages. In this chapter we will first discuss what kind of grammars are available and which one suits most our need. Subsequently we will look at the details of the selected grammar.

The most commonly used type of grammar is the Context Free Grammar (CFG) introduced by Noam Chomsky. It owes its prevalence to the fact that it can describe most of the languages used nowadays. Since it is known for a long time, there are broadly available tools for the generation of parsers. Predominantly the *LR (k) parser*. Most performance and validity problems are solved and well researched. While parsers can be generated from CFGs (for instance with YACC), it sometimes needs human interaction to straighten out ambiguities.

Since we want the process to be as automated as possible, we directly generate grammars as Parsing Expression Grammars [8].

PEGs are commonly used as internal domain specific languages - they can be used directly in the host language. There is no need for an additional scripting language and no subsequent parsing and compiling of the scripting language has to be performed. Once a PEG has been built, it can immediately be used as a parser. So the difference between the runtime structure and the source code is smaller which makes it more understandable and dynamic.

In contrast to LR (k) parsers, PEGs are greedy and unambiguous by design. This is a requirement for an automated approach where no human interaction is possible. PEGs are composed of interchangeable and connectable parsers, forming a graph of parsers. They are not able to express left recursive grammars as CFGs do. Fortunately (meaningful) left recursions can be rewritten.

In terms of speed PEGs can be optimized in theory to be as fast as LR (k) parsers with Memoization with a additional cost in space complexity. Since we are rather interested in the speed part this does not hinder us in using PEGs.

Thus, Parsing Expression Grammars give us a big advantage and we will use them for our approach.

## 2.1 Operators

PEGs are parser objects normally composed of other PEG parsers. Every PEG has a root node. The root is the entry point for the recursively descent parser. There are different types of parsers which act differently on an stream of characters and possibly delegate processing to their children parsers.

All parsers can succeed or fail. If a parser fails, the algorithm will backtrack to the last successful parsing location. If possible it will continue with other parsers. If the root parser fails, then the input string is not defined by the grammar, i.e. the PEG does not match the string.

There are several different basic PEGs that can be combined to build complex parsers. We need PEGs, which can parse single characters and substrings. The following list presents the typical basic parsers used in PEGs:

| Notation | Description |
|---|---|
| 'x' | Single character x |
| [x-y] | Character class from character x to character y |
| . | Any character |

Since these are the most primitive PEGs, they constitute the base for our parsers. They are the only ones that consume one or more (literal string) characters. If the character at the current position of the input string does not match the character related to the parser, then it will obviously fail and induce backtracking, if possible.

An existing graph of PEGs *e* can be extended by using the following unary postfix operators:

| Notation | Description |
|---|---|
| e? | Optional |
| e* | Zero or more |
| e+ | One or more |

Unary suffix operators determine how many times the parser *e* will be sequentially applied on the input string. It generates a new parser with the semantics of the operator which delegates the child. For instance the option operator tries to apply its child parser *e* on the input string. If *e* succeeds, the parsing will be applied, otherwise (if *e* fails) *e* will be omitted. This makes sure *e* will be applied at most once.

The the zero-or-more operator tries to apply *e* as long as *e* doesn't fail. Similar to the option parser, if *e* fails, nothing happens.

The one-or-more parser differs slightly from the zero-or-more. It has to consume at least one *e*, otherwise it will fail.

Similarly there are prefix operators that act on a parser *e*:

| Notation | Description |
|---|---|
| & e | And predicate |
| !e | Not predicate |

Prefix operators change the way of how the parser backtracks. The And-predicate parses the string but does not consume it and will unconditionally backtrack. The information about its success or failure is preserved though and passed up to its parent parser. Similarly the Not-predicate does not consume the string and will unconditionally backtrack. However in this case the success/failure information of the child *e* is inverted. For example the grammar *('a' !'b' .*)* will parse everything beginning with an *a* that is not followed by a *b*.

Parsers can be concatenated by these binary operators:

| Notation | Description |
|---|---|
| $e_1\ e_2$ | Sequence |
| $e_1$ / $e_2$ | Prioritized choice |

Sequences impose an order on the application of the parsers on the input string. They have to consume the string subsequently and if either parser fails, the sequence will fail as well. In the prioritized choice the first parser will attempt to parse. On failure the operator falls back to the second. If both fail the choice operator fails as well.

Not all PEG combinations are valid though. Infinite recursion can occur if there are loops in the graph, i.e. if there is a left recursion in the grammar definition. Looping parsers - parsers which repetetively apply their children on the string -, like the zero or more parser, can raise a problem as well. If their underlying parser succeeds on parsing, but does not consume any character of the input string, then the zero or more parser will endlessly loop as well. These issues will be addressed later on.

## 2.2 Representation and parsing

To illustrate how a PEG works in action, we will look at a simple grammar parsing a short string. In figure 2.1 we have an example of parsing ":a:b:" with the grammar

**Listing 2.1: Simple grammar**

```
0 -> {(':' [a-z] 0) | ':'}
```

which defines all strings with lowercase characters that are delimited by colons. Note that $\{\ldots\}$ demark choice and $(\ldots)$ demark sequences. We used two different parentheses to be able to detect the type if there is only one element.

We will look at the parsing step by step:

1. The input string is fed to the root choice node which delegates it to its first child.
2. The sequence passes the string to its first child.
3. The character parser can parse the first character and consumes it.
4. The range parser can parse 'a' and consumes it as well
5. The choice is recursively called and works again like in the previous four steps, consuming ':' and 'b' this time.
6. Choice is expanded again and the string passes down till to the colon parser which can still parse the remaining column in the input string.
7. The following character parser fails on the empty string which leads to a backtracking to the choice parser.
8. The choice will try with the second child, the single colon parser which succeeds. Every parser then succeeds up to the to the root. Since the root succeeds the grammar accepts the input string.

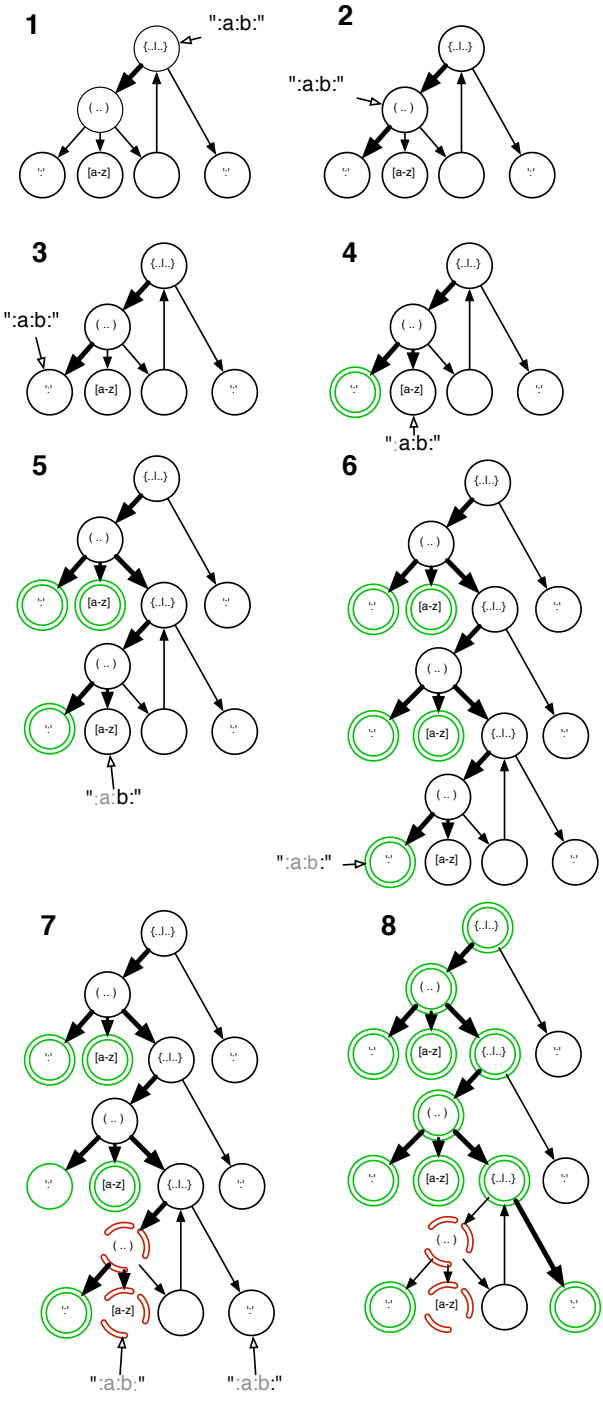


Figure 2.1: Example of parsing with a PEG

## 2.3 Bound behavior

PEGs can be straightforwardly expanded to generate abstract syntax trees. By adding behavior to specific nodes of the parser graph an AST can be constructed while parsing. We will demonstrate this using the example of the previous section.

**Listing 2.2: Simple grammar**

```
0 -> {(':' [a-z] 0) | ':'}
```

In order to generate an AST we just need to define behavior on exactly two nodes of the grammar. On the range parser (*[a-z]*) we define a function that takes the parsed character and wraps it into an AST node. On the choice parser (which is also the root) we define a function that takes all the results of the children and adds them to a new AST node. If we run this enhanced parser on the same string as before (":a:b:") we get a simple tree. The tree consists of an AST node 'a' with a child AST node 'b'. Notice that parsing results are passed up to the respective parent parsers if there is no behavior defined on the current parser. So, in our case, the sequence parser just passes all the results to its parent, which is the choice parser.

An interesting application is to use PEGs directly as simple interpreters. As long as the defined language is functional, it can be executed directly while parsing. For instance a calculator language can be implemented by defining the functions (additions, multiplications, etc.) on the PEG nodes representing the latter. This makes PEGs very powerful for quickly testing various constructs of a language.

## 2.4 Memoization

Parsers for context free grammars have been optimized to perform in linear time. However, parsing with a PEG as described in the previous section is not performant, that is it performs in worse than linear time. Our main performance issue is parsing. It is heavily used for the calculation of the fitness, so we need to optimize the parsing process. We will first look at the complexity of the aforementioned, naive algorithm, to present a linear version using Memoization (in combination with PEGs called Packrat Parsing [4]).

As variables we define $p$, the examined parser, $|p| = m$ the number of nodes in the parser $p$, $s$ the string we want to parse and $n = |s|$, the number of characters in $s$. In the worst case the algorithm takes every possible path through the graphs. Therefore the complexity of the algorithm is the number of permutations the nodes can do on the length of the string (with multiple occurrences) which leads to a complexity of $O(m^n)$.

We want to know the complexity depending on the length of the string, so $n$ is variable. The number of nodes $m$, however, is constant in this context, since the grammar

doesn't change while parsing. Therefore $m = c$ is constant and hence the complexity is $O(c^n) = O(e^n)$. An exponential complexity is something we want avoid at all cost, so we need some way of optimizing the parse process.

Luckily, there is a way to get the parser to be as fast as $O(n)$. This is the same performance as a canonical parser for a CFG. By introducing a cache we can prevent a PEG from reapplying sub parsers to a certain substring multiple times. For every start character of a substring a list is generated that holds all the parsers that already attempted to parse it as well as their specific result. The results can either be a fail or a success. If there is a success, additional information is given for what has been parsed and the optional result of the behavior functions of that sub parser. Thus, while parsing a string we can check the current character position in the cache. If the position has already been computed with the same parser the result can be immediately returned without parsing it again. Otherwise the result will be computed and stored for later reuse.

With this caching we assure a linear parsing complexity. For every character, every parser node will attempt to parse only once. Hence, in the worst case, the list of every character holds all $m$ parsers (every parser attempted once) which results in a complexity of $O(n \cdot m)$. As before $m = c$ is constant and the complexity $O(c \cdot n) = O(n)$.

In their paper Ralph Becket and Zoltan Somogyi point out that Memoization does not increase performance in PEGs for most languages [11]. However, we deal with random rather than handcrafted grammars. Thus Memoization is still beneficial, since we mostly don't generate grammars which behave like handcrafted ones. Even if the goal of the search is a good (in the sense of readable) grammar, we still generate a lot of bad grammars. The latter have to be tested. This is why we used Memoization nonetheless.

### 2.4.1 Example

As an example we will look at the implementation of the parser which generates PEG objects from a string. We implemented the parser for the PEG language and used it for tests and for the deserialization of PEGs when sent over the net (explained in chapter 4).

The definition of the PEG grammar is listed below:

**Listing 2.3: PEG grammar**

```
0 -> (([0-9])+ ' ' '-' '>' ' '
      1 -> {
      #epsilon
      'e' |
      #zero or more
      2 -> ('(' 1 ')' '*') |
      #one or more
```

```
        3 -> ('(' 1 ')' '+') |
        #new link
        0 |
        #prioritized choice
        4 -> ('{' 5 -> {6 -> (1 ' ' '|' ' ' 5) | 1} '}') |
        #sequence
        7 -> ('(' 8 -> {9 -> (1 ' ' 8) | 1} ')') |
        #any
        '.' |
        #number
        10 -> ('[' '0' '-' '9' ']') |
        #lowercase
        11 -> ('[' 'a' '-' 'z' ']') |
        #uppercase
        12 -> ('[' 'A' '-' 'Z' ']') |
        #character
        13 -> (''' . ''') |
        #backlink
        14 -> {([0-9])+}
})
```

Numbers with an arrow denote new rules, beginning with the root rule ♯0. Rules are used by writing the number without an arrow. Characters are denoted with two single quotes around it. Sequences use normal parentheses and choice is written with curly braces.

In order to build the PEG we defined behavior on every rule. For instance rule ♯2 takes the child result and wraps a zero-or-more parser around it. Similarly rule ♯4 stand for the choice and adds all the children results to a choice parser.

The trickiest part of it were the rule backlinks. Rules had to be stored in a global registry so they could be retrieved from every parser. If the rule hasn't been parsed yet but was already used as a backlink, it would be added to the registry with an empty value. Once the rule has been parsed the backlink is automatically resolved.

## 2.5 Conclusion

For the purpose of automatic generation of grammars PEGs are suited for our approach. PEGs are an equally powerful subset of CFGs with the added important features of being easy to manipulate and combine. Unlike CFGs, they are unambiguous by construction. They don't have to be modified to be unambiguous and can be used directly after definition. The concern of not being performant is not an issue, since PEGs can be optimized to perform with the time complexity of canonical parsers used for CFGs using Memoization. The increased space complexity does not hinder us since we are interested in speed rather than space efficiency in order to check a lot of grammars in a shorter time.

Furthermore, as we will see in later chapters, the graph-structure with its interchangeable nodes simplifies some operations which are needed by Genetic Programming algorithms to modify grammars. PEGs do not have to be compiled to intermediate code and then compiled again. Rather they can be used as internal DSLs. As such, the task of generating grammars gets a lot easier, since it can be done on the fly in the host language.

# Chapter 3

# Genetic Programming

In this chapter we will discuss our second prerequisite for the automatic generation of grammars for DSLs. As we saw in the previous chapter we use PEGs to define grammars. As mentioned in the introduction, we will use Genetic Programming as our search algorithm.

Genetic Programming belongs to the class of Evolutionary Algorithms, which are based on the principles of biological evolution. This class of algorithms relies on the basics of evolution to find optimal solutions in a search space. We will look at evolutionary algorithms in general and subsequently go more into detail about Genetic Programming.

## 3.1 Evolutionary algorithms

Evolutionary Algorithms (EA) are used to search for solutions in a big problem space with the aid of the mechanics of evolution. Biological evolution is based on three main characteristics : reproduction, mutation and natural selection. While reproduction conserves the species from extinction, mutation allows for change. The real promoter of evolution, though, is natural selection which serves as a filter for more adapted variations of individuals in a species.

These principles can be applied to computer science in a formal way. Individuals are represented by potential solutions to a given problem. The equivalent to natural selection is defined as the so-called *fitness function*. It determines the degree of adaption or fitness of an individual. The nearer the individual approaches the ideal solution, the better the fitness. Thus the fitness function computes numerical rating which represents how well an individual solves the given problem. Artificial selection then discards (kills) the individuals with the worst fitness, thereby preventing them from reproducing.

Since biological evolution starts from an existing population of species, we need to bootstrap an initial population before we can begin evolving it. This initial population is generally a number of random individuals. These initial individuals usually don't perform well, although some will already be a tad better than others. That is exactly what we need to get evolution going.

The final part is reproduction, i.e. to generate a new generation from the surviving previous generation. For that purpose an evolutionary algorithm usually uses two types of genetic operators: point mutation and crossover (We will refer to point mutations as mutations, although crossover is technically also a mutation). Mutations change an individual in a random location to alter it slightly, thus generating new information. Crossover[1] however, takes at least two individuals and cuts out part of one of them, to put it in the other individual(s). By only moving around information, Crossover does not introduce new information. Be aware that every modification of an individual has to result in a new individual that is valid. Validity is very dependent on the search space - it generally means that fitness function as well as the genetic operators should be applicable to a valid individual. A schematic view is shown in fig. 3.1.

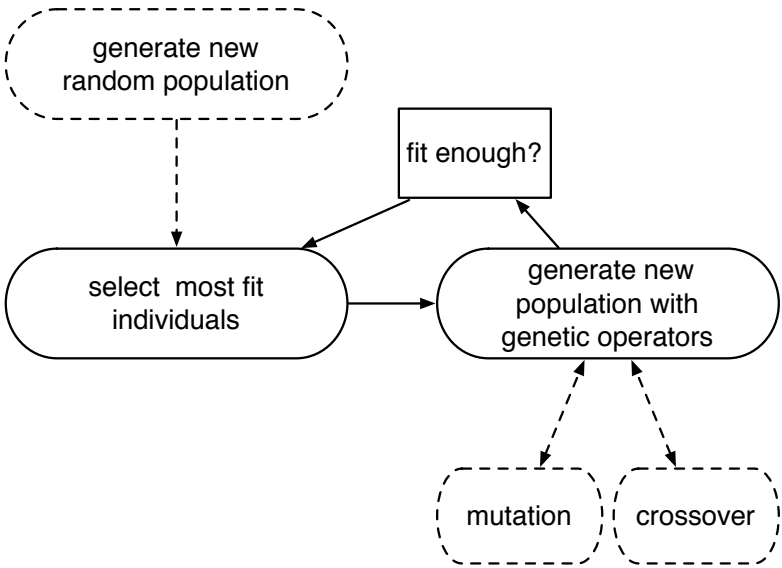


Figure 3.1: Principles of an Evolutionary Algorithm

There are alternatives to rejecting a certain number of badly performing individuals per generation. To compute the new generation, one can generate new individuals from all individuals of the old generation. This would not result in an improvement since the selection is completely random. Hence the parent individuals are selected

[1] Crossover in biology is the process of two parental chromosomes exchanging parts of genes in the meiosis (cell division for reproduction cells)

by a certain probability. Here the probability is correlated with the fitness. Compared to the automatic preservation of good individuals, this probabilistic approach has the disadvantage of possibly losing very well performing PEGs - information can get lost. On the other hand, it helps to get out of local maxima by amplifying the choice of possible search routes.

As in biology, the hope is that it is possible to improve with more or less small but steady steps to get to the envisioned solution. Being a stochastic algorithm, there is no guarantee of a success. The success is mainly dependent on the size of the improvement steps the algorithm has to take. Bigger improvements steps have a lower probabilty of happening, while small steps have a higher probabilty but can't improve solutions by much. If we look at the overall probability of the search (the probability to reach the solution), we find that the probability decreases even more so. The size of the steps depends on one hand heavily on the problem space but also on the offered modifications to change individuals. Finally it depends on the granularity of the fitness function.

## 3.2 Example of Evolutionary Algorithms - Antenna

In many areas evolutionary algorithms have been successfully applied. In a NASA project, for instance, antennas have been *evolved* to consume the least amount of energy for the most optimal ratio of power use versus broadcasting distance (see 3.2). While it could have been calculated manually, it is a very complex and laborious task.

With the evolutionary approach a large search space can be covered and evaluated [5]. Using a fast and accurate physics simulation to determine the fitness, an antenna could be evolved which performed very well. The result is quite counterintuitive and would have been difficult to calculate (or think of) from scratch.

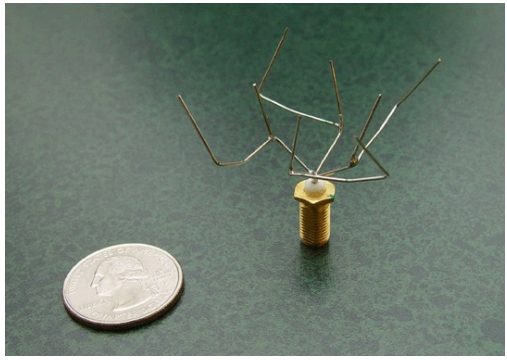

Figure 3.2: The evolved space antenna

## 3.3 Types of EA

There are a few different types of EA, which mainly differ in their encoding of the individuals.

**Genetic algorithm** Encoding as an array which represent the individual by parametrization.

**Genetic programming** The individual takes the form of a computer program.

**Evolutionary programming** Individual is a computer program, only some numerical parts can evolve.

An interesting analogy is found in the difference between GAs and GPs, where the first is only an encoding for a solution, while the second is the solution itself. In biology the difference between the encoding and the actual emerged individual (solution) is called the genotype/phenotype-difference. In Genetic Programming this difference is not present and genotype and phenotype describe the very same thing.

An advantage of Genetic Programming is that modifications are directly performed on the solution rather than on a external number representing it (like in GA). Therefore the change is more local which helps to get smaller probability steps (as mentioned before).

The disadvantage of mixed phenotype and genotype is the increased difficulty of operations on the structure of the individual. Since individuals typically are trees or graphs in GP, modifications on them are also more computationally intensive than for Genetic Algorithms, where mutations are only performed on numbers in arrays.

## 3.4 Restrictions to individuals

The algorithm described above imposes some distinctive restrictions on the individuals that have to be evolved. These are:

**Heritability** Exact copies of individuals have to have the same ability as the originals. Additionally, they slightly change in the process of copying, they should only slightly change their ability as well. Heritability is especially important, since it maintains the information from one generation to the next. This prevents the loss of already achieved successes.

**Variability** It must be possible to change individuals into other valid individuals. There must be a high variability on individuals with vast potential of small changes. Variability is needed to introduce new information into the system in significantly small steps. Without this feature the individuals can't change effectively, which results in stagnation.

**Selection** There must exist a partial ordering of the individuals which is given by the fitness function. The fitness function should rate individuals nearer to the envisioned goal higher. Selection (survival of the fittest) is the actual promoter of the evolution. It favors good individuals over bad ones. Over a big history of populations and equally big variation the quality of the individuals can steadily increase.

Every system on which we want to be evolvable, has to actually fulfill these criteria.

## 3.5 Example from nature

As an example from nature, one can look at the well-known evolution of the peppered moth [1]. This originally white moth lived on birch trees, on which it was very well concealed on the equally white bark. While there were always some mutations, which made some moths slightly darker, they stood out in contrast to the white moths. Hence they were more likely to be eaten by birds.

Then, in the beginning of the 19th century, the bark of the trees became dark because of the pollution from the industrialization. To be white in this case, was a very bad feature. Being gray on the other hand was a huge advantage. The feature for "grayness" spread over the population and replaced the feature for white color. But at the same time some even darker moths appeared, which replaced the lighter gray moths. Eventually all moths ended up having the color of the black bark.

Later on, when the industry stopped using charcoal, a lot less carbon was emitted. This made the birch trees white again, which led to the inverse evolution from black to white moths.

## 3.6 Conclusion

In the previous chapter we decided to use Parsing Expression Grammars to represent our solutions. While we could still write them as strings (or in some binary tree structure) to be mapped onto a value list in order to fit the encoding GA. It is a lot simpler to use Genetic Programming. Since PEGs can already be executed as parsers and are themselves the definition of a grammar, they fit the encoding of Genetic Programming. They are unambiguous by construction which minimizes the required user interaction.

With an Evolutionary Strategy an unmodifiable parsing graph would be another option. We didn't adopt that approach, since it would confine the search space of our grammars too much to bear useful results.

# Chapter 4

# Combination of PEGs and Genetic Programming

In the two previous chapters, we explained the two principal techniques which we will use to automatically generate grammars for existing source code. We will use Parsing Expression Grammars to characterize the grammars and Genetic Programming as the search algorithm.

In this chapter we will look at the details of the manipulation of PEGs. We will search for a good fitness function to find better solutions. Finally we will look at the list of parameters which we determined with comparative evaluations.

## 4.1 Tuning PEGs for Genetic Programming

As described in the previous chapter, we need to meet three conditions for PEGs to build an evolvable system : heritability, variability and selection.

Heredity is easily implemented by deep copying the graph. The restriction of small steps from chapter 3 will be addressed in the different ways of modifications of PEGs.

With PEGs an infinite number of grammars are possible. There are a lot of syntactically different although semantically identical grammars. We will have to make sure that every modification of a grammar leads to another valid grammar.

The last condition, *Selection*, is defined with the fitness function. It has to be able to generate a partial order of the grammars. The fitness function will try to parse the existing code with the grammar and rate quality of the parsing.

To transform PEGs to evolvable structures, some adaptions have to be made. In the definition by Bryan Ford [7] sequence and choice operators only have two child parsers. Longer sequences (or choose statements) can be built by nesting them. To make profitable changes more probable and to keep the number of nodes down (which also increases the execution speed as well) they have been generalized to an arbitrary length.

First we will look at the modifications that are possible for grammars and are commonly used in evolutionary algorithms: mutation and crossover.

### 4.1.1 Mutation

A mutation of a grammar is the modification of one of its randomly chosen nodes. Every parser node is different and therefore it has to change itself in a different way according to its type. So for instance the character parser will mutate the character it parser. Similarly a range parser alters its range.

There are though some more common mutations that affect the structure of the grammar and are not dependent on the node. They will only affect not primitive and not unary parser nodes:

**add child** A new randomly generated parser will be inserted in the list of children (fig. 4.1)

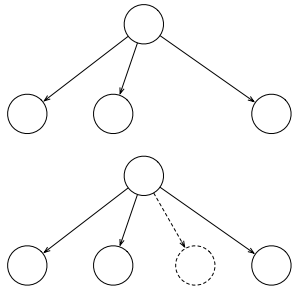

Figure 4.1: Add a node

**add link** Works similarly to adding a child. Although in this case the new parser is not randomly generated but selected from one of the nodes of the already existing PEG. This results in a link to this parser (like a CFG rule, fig. 4.2)

**remove child** A randomly selected child will be removed. No effect, if there is only one child. Remark that we don't allow composite parsers with no children, since they don't constitute a valid grammar (4.3)

The mutation has no effect on unary or primitive parsers like the character parser.

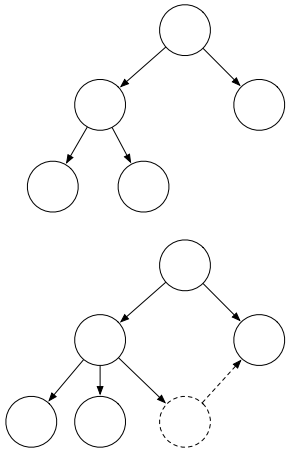

Figure 4.2: Add back link node

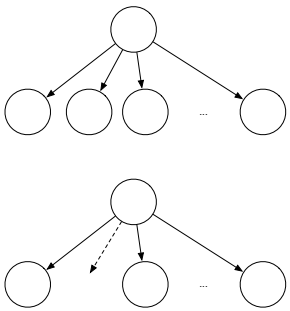

Figure 4.3: Delete a node

To ensure the evolvability of more complex parsers we need more complex mutations. After the initial population got sorted mostly only single character parsers were left and couldn't mutate to parsers with more nodes.

The following mutations add the possibility to insert nodes between the current parser and the root parser:

**deletion** The selected parser first moves all its children to the parent parser, thus replacing itself by its children (fig. 4.4)

**insertion** The selected parser is replaced by a composite parser (sequence or choice). The selected parser is then added to the new parser. This results in the insertion of a new parser in between the selected parser and its parser. If the selected parser is the root, the new parser becomes the new root. (fig. 4.5)

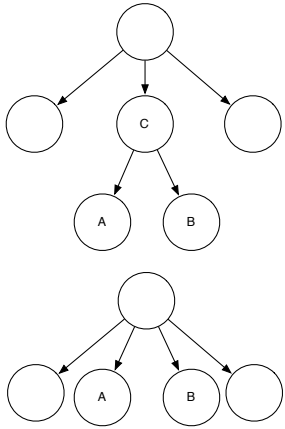


Figure 4.4: Push a node up

Insertion ensures diversity in graph depth. Graph depth is important for the emergence of more complex structures. The deletion, on the other hand, puts a counterweight to it. This ensures that grammars do not grow too big and that is possible to simplify structures.

### 4.1.2 Crossover

Crossover is the second, less often used modification. The term is borrowed from the biological chromosomal crossover, where it describes the exchange of genes of two paired up chromosomes (one from the mother and one from the father) in the process of the Meiosis [1].

Applied to grammars, this results in copying a subgraph of one grammar into another graph. If one has two grammars $p_1$ and $p_2$, a new grammar is created by selecting a random node of $p_1$ and adding it to random node of $p_2$. Note that the parsers will not be directly linked. Rather the subgraph of $p_1$ is copied independently from its parent.

The motivation for crossover is to preserve useful, more complex structures that have already evolved. By combining two modestly performing parsers one can generate a new one that combines the useful parts of the parents into a parser that performs well in the two places of the parents.

[1] Process of generating reproductive cells like sperms and eggs

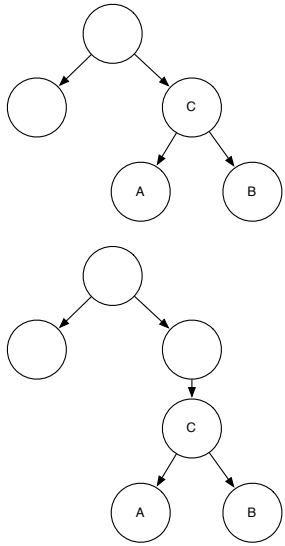


Figure 4.5: Insert a node

### 4.1.3 Fitness function

The fitness function is the most important part of the evolutionary algorithm by indirectly defining the envisioned goal. It determines the quality of PEGs which is a representation of the distance to the solution. Without an elaborate fitness function, the search will head in the wrong direction. There is also the risk of finding non-intended solutions or to be trapped in local extrema.

Normally the fitness function is defined as a number which stands for a better rating, the higher it is. We decided to define the fitness function the other way around: worse PEG have a higher fitness number; the fitness of 0 defines the best achievable solution. This makes sense because we use the size metric of a PEG which is worse the bigger a PEG gets. Furthermore, we have found a possible solution if all the characters of the source code have been parsed. Hence we measure the number of characters to parse which is better the smaller it is.

For our problem we first implemented the most straightforward solution. We let each PEG parse every source code file. The number of characters that it cannot parse is added to the final fitness. There is an additional penalty for not being able to parse a file at all (parser fails on the first character). The reason behind this is that we wanted a stronger differentiation between the grammars that could at least parse one character and the completely useless grammars. In this way a better grammar is a grammar with a *lower* fitness, a grammar with fitness zero being one that can fully parse every source code. We don't allow negative fitness (the reason is explained later).

The fitness function for a PEG $p$ therefore looks as follows:

$$\text{fitness}_1(p) = \sum_{s}^{sources} \text{unparsed}(p, s)$$

The *unparsed* function takes a parser $p$ and a string $s$ and computes the string part that could not be parsed and returns a constant value for unparsable strings (unparsable regarding the parser $p$).

However, if we apply the previously defined fitness function directly we notice that the grammars become very long with a lot of junk nodes in it. For instance epsilon parsers accumulate in sequences. Similarly "parasitic" parser nodes hide in dead (never executed) branches of choose parsers. This is unwanted since the user can hardly understand it. Furthermore the computational time increases with every additional node, hence the search slows down for every generation. The fitness function can be enhanced by by adding the number of nodes of the parser to the fitness.

$$\text{fitness}_{2a}(p) = \text{fitness}_1(p) + |p|$$

where $|p|$ counts all the nodes in p.

The new fitness function improves the resulting parsers. The length of parsers should lie in an interval though. The linear approach though does not address it, thus an exponential function is more suited. It grows slowly for a certain range of small number and ever more for bigger numbers. A parser can evolve in the small length interval and can't go beyond (unless it is lot better in parsing).

$$\text{fitness}_{2b}(p) = \text{fitness}_1(p) + c^{|p|}$$

where c is a constant so that $c > 1$

To refine the interval we have to add a lower limit as well. Normally the best parsers in the first few generations are very short and simple parsers. The step from a very short parser to a longer better one is very improbable with the one-sided exponential approach. A penalty for very short parser lengths can be achieved with a negative exponential function (see graph 4.6):

$$\text{fitness}_{2c}(p) = \text{fitness}_1(p) + c_1^{|p|} + c_2^{-|p|}$$

This fixes solve the length problem and the junk is discarded without inhibiting the evolution process.

While we can tackle the issues of length, it is not enough to generate useful parsers. After running a couple of tests a pattern emerged. The surviving parsers add more and more characters to a choice parser till it has used all of the characters of the sources. This parser is equivalent to the *Any* parser and can parse any character source string. An additional repetition parser as the parent of the choose gives it the power to parse all the sources without an error:

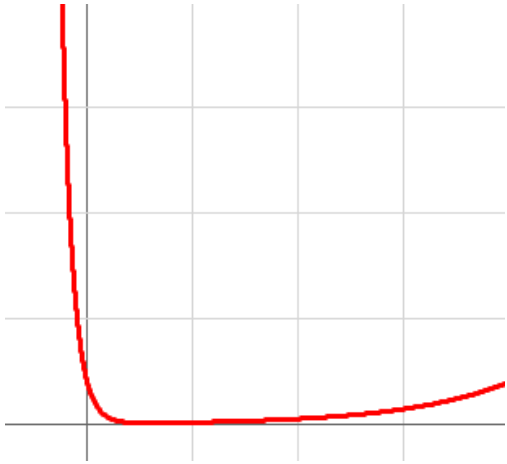

Figure 4.6: Fitness graph designating the fitness (vertical) depending on the length of the parser (horizontal)

**Listing 4.1: Cheating Parser**

```
(('a' | 'b' | 'c' | ... | 'z' | ...))+
```

The reasons for this to evolve is on hand that it can evolve easily step by step, by adding a character at a time. On the other hand the first functional ancestor that has the same structure consists only of three parsers (character, choose, repetition). However such a parser brings no additional layer of abstraction to the code.

A way of encountering this kind of problem is to add penalties to certain - strongly generalizing - parsers like the choose and the repetition parsers, while not giving any to the sequence and character parsers.

We could also redefine the penalty as a bonus for parsers like sequence and character parsers, giving them negative penalties. While these penalties favor restricting parser nodes, it adds a lot of junk to the parsers in the long run. Since boni would be added regardless of the function of a parser node, they can be deliberately added to dead branches without changing the ability to parse strings. Dead nodes with boni enhance the fitness value of the parser by giving no further improvement to the capabilities. Hence although they will survive, no improvement is gained over the former generation and the evolution stagnates. Therefore we need to define the penalty on the parsers we don't want rather than boni to the ones we want, to preserve the link between a good fitness and good parsers.

$$\text{fitness}_3(p) = \text{fitness}_{2c}(p) + \sum_{\text{node}} (\text{penalty}(\text{node})) \qquad \text{penalty}(p) \geq 0$$

The added penalty leads to an improvement of the result, although in most cases it only increases the number of generations needed to find the same solutions as the ones found without penalty. One approach would be to look at the dynamic information of the parser. By analyzing the performance of the different nodes while parsing we could penalize dead branches. Alternatively there could be a bonus for the richness of structure, although the "richness" would be difficult to define. We didn't consider these options in this thesis.

The biggest problem is overgeneralization. We can't eliminate it with tweaks in the fitness function. By adding negative examples that the parser should reject and not be able to parse, all too general parsers can be discarded easily. The number of parsed characters of the negatives would then be added to the fitness as an additional penalty. The new fitness function then looks as follows:

$$\text{fitness}_4(p) = \text{fitness}_3(p) + \sum_{\text{wrong}} (\text{len}(\text{wrong}) - \text{unparsed}(p, \text{wrong}))$$

This last step substantially increases the quality of the evolved parser. Overly generic parsers are sorted out rapidly, since they can parse wrong files as well. This gives them a worse fitness.

As we want to have an automated algorithm with as little input as possible, we face the problem that the user has to give negative files. We tried to solve this problem by generating negatives by cutting the available source code into pieces, shuffling them and gluing them together again. Like that files are created that have a lot in common with the correct code (same character set, similar constructions) but hopefully with broken syntax structures.

Obviously a big code base of another language (for instance the Linux kernel source) can serve as database of negative examples as well. It is not clear if the parsers will find subtle language constructs though, since it is easier to reject a completely different language. On the other hand scrambled code of the language to be found can still be valid syntax and would be a false negative, thus breaking the validity of the fitness function.

## 4.2 Left recursion and endless loops

Genetic Programming only allows valid individuals, in our case parsers. However, syntactically valid programs might be semantically bogus because of left recursions. Left recursions occur when a PEG points to itself (even indirectly), without a character consuming PEG in between. This leads to endless loops during parsing, since the pointer on the string to parse does not change. Similar endless loops can occur for instance in zero or more parsers, which iterates as long as it's child succeeds. If the child does succeed, but does not consume anything (like the epsilon parser), the system hangs.

Since we generate parsers randomly, we quickly produce invalid grammars and run into the mentioned problem. There are different ways of avoiding left recursion. Alessandro Warth, James R. Douglass, and Todd Millstein enhanced PEGs to be resistant to left recursions as well as endless loops [12]. In their work they modified Memoization, the optimization cache, which stores parser results on substrings, which have been computed already. Memoization can be expanded to just fail or at least omit parsing, if at the parse position a parser has already been applied. While this method is useful to avoid a hanging system, we don't want to generate grammars which contain left recursions. They break in canonical implementations of PEGs and are more difficult to understand.

Instead of using the PEGs robust to left recursion, we check if our randomly generated parsers are left recursive. If they are, we reject them without testing the fitness. Calculationg the fitness is done through parsing of samples and could thus already end in infinite recursion. We will have to test every PEG which has been generated through the evolutionary process (mutation and crossover) again.

To check the validity of a PEG we need three functions which rely on each other. By recursively checking the PEG graph we will be able determine whether a grammar is valid or not. The functions check a certain property of a PEG: validity, consumption type and consumption loop.

*Validity* is the function we have to call to determine, if the PEG has any kind of structure that can lead to an infinite loop. It relies on the consumption type function. Consumption type determines all the kinds of consumptions the PEG could possibly perform (for details see below). The consumption loop function searches for direct loops of non-consumption.

The consumption loop check identifies all the possible types of consumption the parser can do. It produces a combination of the three types of consumption: failure (F), success with consumption (SC) and success without consumption (SNC). So for instance the character parser can fail (while trying to parse a wrong character) as well as succeed with consumption (when the character is correct). But it can't succeed without consumption. The following table depicts all the outcomes of the various PEG parser types:

| | |
|---|---|
| 'x' | (SC, F) |
| "xy...z" | (SC, F) |
| [x-y] | (SC, F) |
| . | (SC, F) |
| (e) | consumptiontype(e) |
| e? | consumptiontype(e) ∪ (SNC) |
| e* | consumptiontype(e) - {F} |
| e+ | consumptiontype(e) |
| $e_1$ $e_2$ ... $e_n$ | $\begin{cases} \bigcup consumptiontype(\text{e}_i) & ,\forall e_i \quad \{SNC\} \in e_i \\ \bigcup consumptiontype(\text{e}_i) \text{ - } \{\text{SNC}\} & ,else \end{cases}$ |
| $\{e_1 \mid e_2 \mid \ldots \mid e_n\}$ | $\bigcup$ consumptiontype($e_i$). |

In a second step we will define the well-formedness of a parser recursively. This is the place where on one hand the left recursion is tested with the recursion check and on the other hand loops are detected.

| | |
|---|---|
| 'x' | True |
| "xy...z" | True |
| [x-y] | True |
| . | True |
| (e) | wellformed(e) |
| e? | wellformed(e) |
| e* | $\begin{cases} False & ,\{SNC\}\in \text{consumptiontype(e)} \\ wellformed(e) & ,else \end{cases}$ |
| e+ | $\begin{cases} False & ,\{SNC\}\in \text{consumptiontype(e)} \\ wellformed(e) & ,else \end{cases}$ |
| $e_1 \quad e_2 \quad \dots \quad e_n$ | $\bigwedge ncl(e_1)$ and $\bigwedge wellformed(e_i)$ |
| $\{e_1 \mid e_2 \mid \dots \mid e_n\}$ | $\bigwedge ncl(e_i)$ and $\bigwedge wellformed(e_i)$ |

Finally the "no consumption loop"-check (ncl) as used above is defined. It checks if there is a potential loop in the grammar graph where there is nothing consumed of the string. $V$ defines a list of already visited nodes and $p$ is the currently checked parser.

| | |
|---|---|
| 'x' | False |
| "xy...z" | False |
| [x-y] | False |
| . | False |
| (e) | ncl(e) |
| e? | ncl(e) or ($e \in V \wedge \{SNC\} \in consumptiontype(e)$) ( |
| e* | ncl(e) or ($e \in V \wedge \{SNC\} \in consumptiontype(e)$) |
| e+ | ncl(e) or ($e \in V \wedge \{SNC\} \in consumptiontype(e)$) |
| $e_1 \quad e_2 \quad \dots \quad e_n$ | Holds true if for one of the $e_i$ ncl($e_i$) holds true or ($e \in V \wedge \{SNC\} \in consumptiontype(e)$) |
| $\{e_1 \mid e_2 \mid \dots \mid e_n\}$ | Holds true if for one of the $e_i$ ncl($e_i$) holds true or ($e \in V \wedge \{SNC\} \in consumptiontype(e)$) |

The well-formedness filters all non-left recursive and endless loop containing grammars. Since they will end in erratic behavior they're not wanted by the user anyway, so they can be dropped.

## 4.3 Examples

To illustrate the solution we will analyze the algorithm performing on some example grammar. First we look at a simple left recursion:

**Listing 4.2: Parser with an endless recursion**

```
0 -> (0 'a')
```

When checking well-formedness from this grammar we have to look at sequence definition which tells us to perform "no-consumption-loops" on every child. Here already we get a "no-consumption-loop" which is a direct recursion. So this grammar is not valid (well-formed). An example for an endless loop is the following grammar:

**Listing 4.3: Parser with an endless loop**

```
0 -> (e)*
```

where "e" is the epsilon parser. If this PEG is fed a string it will endlessly consume epsilon. To detect this loop we will first apply a well-formedness check on the zero-or-more parser which tells us to check if the consumption type of its child is not SNC which means that it can success without consuming any characters. Since epsilon is typically among this type of consumers the well-formedness test will fail in the epsilon parser and thus the loop is detected.

## 4.4 Optimization of Genetic Programming

We looked at the optimizations of the fitness function in the last section. There are more optimizations for GP which speed up the search. The optimization include strategies to avoid local extrema.

After a couple of generations the populations consist of many copies of only a few distinct grammars. The diversity falls to a minimum. Therefore the very first action is to disallow duplicates in the population for a new generation, thus increasing the amount of information available for each generation that can be shuffled together.

A solution to this problem is to add more different types of mutations in order to decrease the probability of certain more complex transformations, like adding nodes, collapsing nodes or putting a node in between two others. While that speeds up the search, it is still not satisfactory enough.

A second, more difficult to detect scenario where duplicates can emerge poses an extra problem. We discard all but the $k$ best grammars from the list. So if a grammar $g$ is generated for generation $i$ and is then discarded because it was not fit enough, it can be regenerated for the generation $i + 1$. Hence the fitness function is applied on $g$ one

more time. Since the minimal fitness over all the surviving grammars cannot decrease from one generation to the next, and the fitness for a given grammar is static, $g$ will be discarded again. That leads to two problems: increased computation time for the evaluation of the fitness for a useless grammar as well as less variability.

A straightforward solution for the problem is to hold a list of already discarded grammars and to check if the newly generated grammars are in the list. Grammars had to be completely identical to be equal. More sophisticated algorithms could have been used but on the one hand they would have decreased performance drastically and on the other hand the diversity of grammars would have suffered. Even with this simple equality check the number of generations needed to get to a certain fitness decreased remarkably.

After a while such cached results become obsolete as the top grammars grow further away from the discarded parsers. This allows us to remove them from the cache, in order to free up memory.

## 4.5 Parallelization

To improve performance on one side and results on the other, we distribute the computation into different processes. This is easily done by the distribution of the fitness calculation of the grammars. This calculation is computationally the most intensive. For the calculation of the fitness, a certain grammar has to parse all the input files. Since neither the grammar nor the input files change after parsing, this process can be easily distributed on different processors. To diminish the overhead, the computation should be divided by grammar and not for every source file.

We decided to take a slightly different approach to improve the results as well. We often noticed that the algorithm got stuck for quite long times in local minima. One way of preventing this problem, is to run the algorithm on the different processors independently and from time to time to merge the results. This idea is once again taken from nature: distinct species have evolved mostly thanks to isolation of one previously uniform specimen. First this leads to different races (like in dogs) and later to species of their own that cannot interbreed.

The reason why the algorithm gets stuck at local extrema is mainly that after some time the grammar gets bigger and the number of possibilities of grammar definitions increases. For instance, the parsers 'a' and ('a' e) are able to parse exactly the same set of strings (which consists only of the letter $a$). Hence after a while the best grammars are all very similar to each other, although exact duplicates are eliminated. In biology that phenomenon is called a low biodiversity. With a low biodiversity though is smaller probability to get out of a local extrema situation, since the ground on which novel parsers can grow does not allow for much variation.

One idea to prevent this would be to increase the percentage of the grammars taken to the next generation. The problem in this case is that the selection is not strong enough

anymore and the process takes a lot more time.

To prevent thinning out the diversity, we added a server/client structure to our algorithm. The server is the main evolution going on and the clients simulate an isolated environment to increase the variability of grammars. It takes the idea again from real evolution where species differentiate from one common ancestor being seperated (for instance via island formations). Similarly this separate evolving system tries again to ”revive” worse grammars, increasing the paths of mutations tried, to find a better solution. Splitting GP into islands is known as *Multipopulation Genetic Programming*[6].

On startup the server generates grammars randomly as before and computes one generation. Before it reiterates though it pushes some of the rejected grammars to a ”waste basket”. Clients can then connect to the server and take grammars out of the waste basket to perform evolution on them.

After a predefined number of generations the client pushes back its best grammars back to server. These new grammars are then handled like any other grammars generated on the server.

## 4.6 Preliminary data

To further decrease the number of generations needed, and to improve the quality of the resulting grammar, we need preliminary information to steer the algorithm in the right direction. The main reason for this is due to wrongly set parameters of the Genetic Programming. This issue will be covered in the next chapter.

Some languages do not use all characters of the available character set, so the first thing we did was to scan the code files and find all actually used characters. However, the inherent grammar could be defined using more characters than actually used in the available code samples. We still preserved the ability to introduce any character, but characters unused in the source would have a lower probability.

Parsing scoped code with this setup is still a problem. So we define matching characters beforehand and introducing them as new kind of mutation. Though this helped for instance for brainfuck code, it wasn't as successful as we hoped (see 5.3). The problem was again that we only looked at matching characters, while there are languages where whole words are matched.

## 4.7 Tweaking the parameters

A lot of the (implicit) parameters in the mentioned processes cannot be computed beforehand and have to be adjusted with heuristics. On one hand we have parameters for PEGs. While normal PEGs have no parameters, their evolvable counterparts have

some parameters that have to be found heuristically. Certain mutations, like the mutations of a character, rely on probabilities on which character to choose (see last section). The mutation that inserts a new random child node to a parser will choose the index with a uniformly distributed probability. A new random parser is created by randomly adding parsers to other parsers, until a certain tree depth has been reached. Here the only parameter is the maximal tree depth. It has to be chosen to fit the population size, otherwise the number of possible grammars might be too small. This would decrease the diversity and annihilate the advantages of a bigger population. Finally we have the probability of choosing a node in a parser to be mutated. Again we used an uniformly distributed probability over all the nodes of the parser.

Genetic Programming itself on the other hand, has a lot more unclear parameters that have to be determined stochastically due to its own stochastic nature. There are the parameters for selection and population size, which we called $k$ and $n$ respectively. $n$ here is the total number of individuals per generation a $k$ the number of individuals that are passed on to the next generation unchanged (and $n - k$ the number of individuals that are discarded).

Additionally we investigated the differences between mutations and crossover. We determined which one is better at improving the fitness, or rather which ratio of combination is most beneficial. First we will take a look at $n$ and $k$. To determine the best values for $n$ and $k$, we computed test runs for combinations of the two parameters. To visualize the results we plotted them on a grid as shown in figure 4.7.

The values of $n$ are represented on the horizontal axis, while the values of $k$ are represented on the vertical axis. The position of a pixel therefore indicates a combination of an $n$ and $k$. Gray values show the attained fitness of that test run: the worst are white going over bright and dark gray to black, which indicates a very good fitness.

In the case of this image, the test run was performed beginning with an $n = 3$ to $n = 450$. Accordingly all the $k$ from $1 \dots n - 1$ were run so that we got a triangle. By ignoring $k = 0$ and $k = n$ we avoided the impossible or completely random (no selection) cases respectively. Since this process is very time intensive we only computed 20 generations per $n$,$k$-pair.

What this image shows is that the algorithm performs best for bigger $n$ and smaller $k$. However there is a problem with very small $k$. While it improves the speed (and decreases the needed number of generations), it often leads to dead ends, i.e. local extrema. With a small $k$ the diversity of the individuals is radically lowered, which decreases the probability to improve after a local minimum is reached. Thus the best choice is a $k$ which is still in the fully dark zone but not too low at the bottom.

On the other hand, the fitness gets better as $n$ increases. This is expected, since a higher $n$ means that more new individuals are generated; the coverage of all possible mutations is higher.

Unfortunately, as $n$ grows, computational time increases. A small $k$, which we already showed to be beneficial, increases the computational time as well. For every generation

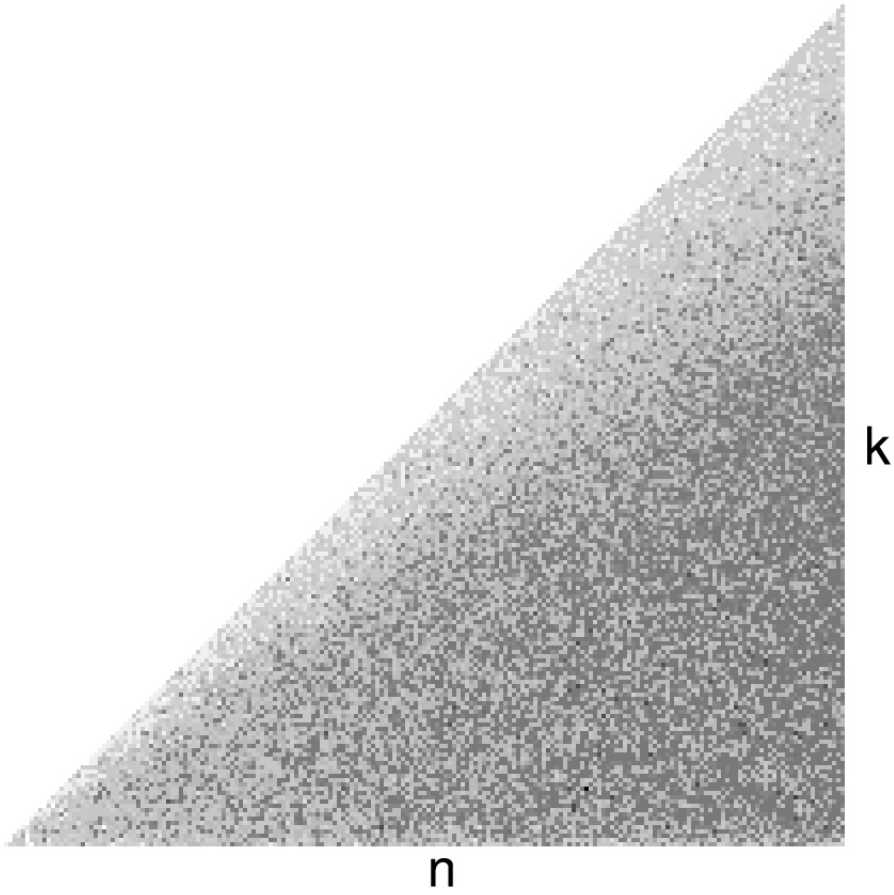


Figure 4.7: Heat map of the fitness

a lot of new individuals have to be generated and checked for duplicates. Additionally every new individuals need to be rated with the fitness function.

Similarly we test for the best mutation/crossover ratio. According to the tests with $n$ and $k$ we choose a relatively big $n = 300$ and a small $k = 20$. Very low values for this ratio (0 = all crossover, 1 = all mutation) take a long time to compute. The reason for this is that crossover does not introduce novel structures. It is only a reshuffling of the already present information. Since we want every individual to be different from the other in the new generation, it takes more time to find distinct ones by only shuffling nodes around. So we only calculate fitness above the 0.5% mutation.

The following image depicts the results of the evaluation. We run 20 tests for every ration between 50% and 100% mutation over crossover. The colors are used as in the previous analysis. Dark stands for a better fitness down to bright grey and white for

a worse fitness, like in the $n/k$-picture. The x-axis has no meaning in this context. As one can see, there is no significant difference between the results. While there are

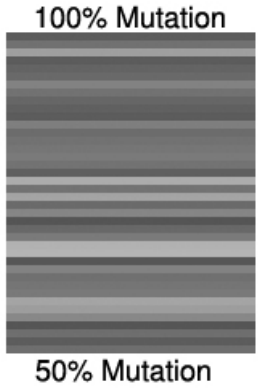


Figure 4.8: Heat map of mutation over crossover ratio

fluctuations, they are not consistent for any value of the mutation/crossover ratio. It almost looks like crossover has no impact on the quality of the results. The only effect measurable, was an increased computational time. Finally we dropped the crossover completely.

Sean Luke and Lee Spector [13] compared crossover and mutation in classical Genetic Programming problems. With four different (*hard* and *easy*) problems they tried to figure out the merits of both modification methods. They tried to correlate population size to the success of mutation or crossover as well. There were some minor advantages for the mutation when they tried to solve a hard problem with a small population size. Overall crossover performed a little better. They used a lot bigger population sizes and generations as we did in the test above. However, they had to conclude that there is no statistically significant difference between mutation and crossover. Crossover never performed very well, if it was used without mutation.

Similarly to our tests the size of the population improved the fitness result. They didn't measure the performance of the $k$ though. In this case they used the traditional value of ten percent.

## 4.8 Conclusion

As we saw in this chapter, PEGs can be easily adapted to fit to the evolutionary requirements of Genetic Programming. Different types of mutations were added to be able to alter PEGs. More mutations had to be added as a reaction to excessively low probabilities of certain structural changes.

We addressed the problem of left recursion and endless loops in grammars, which emerge constantly due to the random nature of generation and mutation of PEGs in the process of evolution.

The fitness plays one of the more important roles in the formation of useful grammars. A set of steps are needed to get an suitable fitness function. It has to take into account all the the fundamental problems while generating PEGs. We experience the well-known phenomenon of over-generalization which forced us to introduce negative examples. Finally we isolated the most important parameters of GP: mutation over crossover rates and the reproduction ratio. Although our setting comprehends a variety of different techniques with their own parameters which have nothing to do with GP, we found the same results as a much larger analysis of the GP evaluations of Sean Luke and Lee Spector for these parameters.

# Chapter 5

# Case studies

We applied our approach to different types of grammars. We gradually made the languages more difficult to be able to tell when we got to the point where our approach was not sophisticated enough to find any useful solutions anymore. We looked at the grammars of canonical identifiers, a simple configuration file and a programming language, namely brainfuck.

## 5.1 Identifier

Since the identifier is used in virtually every language, we needed to know that our algorithm was able to find a solution to this problem. Although the definition in language differs in the details, they mostly look like this:

**Listing 5.1: Envisioned parser**

```
([a-z] (`_` | [0-9] | [a-z])*)
```

An identifier is completely lowercase and has to start with a lowercase character. It is followed by alphanumerical characters including the underscore character. We provided the character with a small set of correct identifiers and an equal number of negative examples. After approximately 200-300 generations the algorithm consistently found the following or a similar grammar:

**Listing 5.2: Found grammar**

```
(([a-z] ({`\n` | `_` | [0-9]})*))*
```

This is not exactly the grammar we searched for. It is able to parse the same string as the envisioned parser though. The additional line break is due to the fact that the input

strings are stored in files which have a line break at the end of the file. It is interesting to see that the algorithm finds a way to reuse the range parser [a-z]. As we designed the envisioned grammar we had an intention of what the different parsers mean (”first character has to be a lowercase”). The algorithm can’t know this intended meaning, so the result is satisfactory for our purpose.

In image 5.1 we see a run of the program. It shows the behavior of evolution with stable plateaus for several generations on one hand and rapidly changing and optimizing individuals in between plateaus on the other hand. The rapidly changing parts indicate a new successful mutation being optimized until it reaches a new plateau.

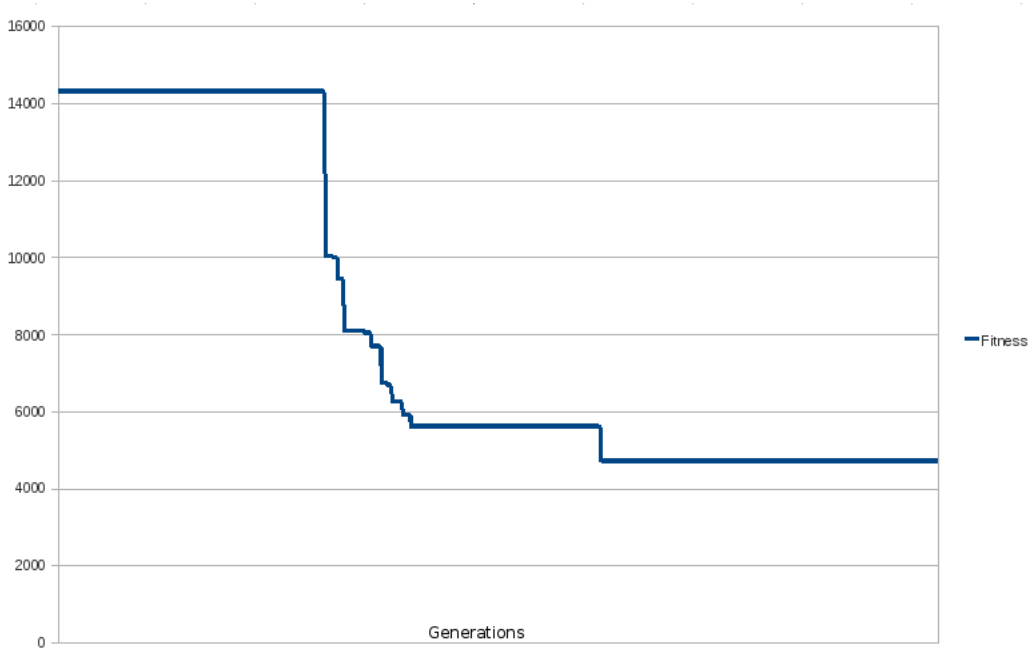


Figure 5.1: Best fitness per generation

## 5.2 Keyword

Our next quest was to determine if the algorithm is able to figure out keywords. For this purpose we invented a simple configuration file language:

**Listing 5.3: Envisioned parser**

```
0 -> (`c` `a` `t` `:` ` ` ([a-z])+ 1 -> {2 -> (`\n` 0) | e})
```

We want to find the keyword ”cat:” followed by an arbitrary number of lowercase characters. The language can parse multiple lines with this same pattern. This is a slightly more difficult problem. On one hand it enforces the grammars to use the exact characters which is more difficult than just using a range parser. On the other hand the pattern is repeated an arbitrary number of times, so the algorithm has to abstract a line as an entity.

After 1000-2000 generations the algorithm found grammars similar to the following one:

**Listing 5.4: Found parser**

```
(0 -> (`c` `a` `t` `:` ` ` 2 -> (([a-z])+ `\n`)))+
```

It gradually improves by starting with the simple character parser ‘c‘ and subsequently the whole keyword ”cat:” and the rest of the line. Unfortunately the algorithm is not able to parse files with more than one line. When trying with a higher population (from 100/10 to 300/30) we achieved a much better results similar to the following:

**Listing 5.5: Found parser**

```
0 -> (`c` `a` `t` 2 -> (`:` ` ` e 3 ->
  ([a-z] (4 -> {5 -> (0) | `\n` | 6 -> ([a-z])})*)))
```

With this configuration we could find grammars which could parse whole files complying to our simple language. This particular example of a solution uses recursion to accomplish the repetition of the lines. A solution with a repetition did not evolve. Note that the grammar is not as restrictive as our hand crafted one. This is due to the set of examples which did not include negative examples for instance for consecutive newlines.

## 5.3 Real language

To test parenthese matching we used the very simple but Turing complete esoteric programming language *brainfuck*. The language consists of eight characters, each one denoting an operation on a tape. Two of them are square brackets which stand for a while loop. They can be nested and have to be matched. With this simple language we have a prototype for matched parentheses. Having so few grammatical structures it is ideal to perform a lot of small tests with different parameters. Keywords in this case are just characters, further speeding up the search by making it easier to focus on the structure. We already showed that keywords can be found with the previous example.

As before we have a grammar in mind which should be found. The following is the envisioned grammar:

**Listing 5.6: Envisioned brainfuck grammar**

```
0 -> ('+' | '-' | '<' | '>' | ',' | '.' | 1 ->
  ('[' 2 -> (0)* ']'))
```

It consists of all the keywords which designate tape operations and the loop marked with square brackets. It contains a backlink to the root of the grammar so they can be nested.

Trying many different combinations of $n$ and $k$ (see 4.7) we were not able to reproduce a similar PEG with our algorithm. It didn't manage to find the matched parenthesis in the stream of characters and almost always produced a result like the following:

**Listing 5.7: Found brainfuck grammar**

```
(0 -> {'<' | ']' | '.' | ',' | '>' | '-' | '[' | '+'})*
```

The situation occured that the algorithm just found the easiest way of matching the source code. It just listed all the existing characters in a list without adding structure. Early generations produce simple character parsers or sequences with a few characters. These parsers are better than the rest since they can parse the beginning of source code. The easiest way to expand this simple parsers is by adding a choice and just add the rest of (in this case) the small set of possible characters.

The reason why this happens is mainly due to the fitness function which can't properly give a bonus to the structure of a grammar. An idea would be to add boni for the maximal depth of a grammar. But this solution mainly just increases the size of grammars and encourages additions of meaningless sequences and choices.

## 5.4 Conclusion

In this chapter we evaluated the performance of the algorithm. It operated well on simple grammars and could detect identifiers and keywords. It struggled to find matched parentheses. This is mainly due to the fact that sequences (which are needed for matched parentheses) are more brittle than choice operators in an evolutionary sense. Parsers can be added to choices more easily without breaking the parser and therefore they are evolutionary more successful. Solutions to these problems are outlined in Future work 6.1.

# Chapter 6

# Conclusion

As software projects grow, we want to be able to maintain an overview, on the one hand to ensure a good design and on the other hand to enable new team members to quickly understand a project. To facilitate the understanding of big projects analysis can be automated. For this purpose we need a parser, and thus a grammar of a language. Since we don't want to implement our own parsers and just quickly extract useful metrics, we need a way to help the user to generate grammars.

The goal of this work is to evolve formal grammars for a given set of sources written in a programming language. This greatly helps in mapping source code to metamodels, since we can generate an AST with a parser. The mapping between an AST and metamodels is different depending on the language and the kind of data that we want to extract. So the mapping should be done by hand, but the grammar extraction can be automated.

We used Genetic Programming as an evolutionary search algorithm and Parsing Expression Grammars as a grammar representation. The evolutionary strategy has been chosen due to its generic approach to problem solving. It can build solutions gradually by slightly improving solutions over time. Since the problem at hand is complex, we couldn't generate grammars in one step, further encouraging this approach. This work also looked at how all the crucial parameters had to be selected to get the best result, focusing on the Genetic Programming parameters.

We saw that indeed PEGs and Genetic Programming combine nicely. PEGs can be transformed and mutated in an easy way due to their modular nature. It is a prerequisite of Genetic Programming. Genetic Programming is powerful tool to find solutions to a given problem. Due to its abundance of free parameters it is difficult to handle though. Finding an appropriate fitness function has proven to be challenging. Over several steps we improved the fitness function and thus the resulting grammars. It involved finding a pratical definition of a useful grammar. We combined size and the ability to parse into the final fitness function.

In Genetic Programming the goal lies in finding a solution as quickly and easily as possible, which can lead to overly simple solutions. So it's no coincidence that we only found very simple grammars and had to improve the fitness function.

We also dealt with overgeneralization. Due to the complex task, solutions tend to be compliant grammars with small restrictions to the syntax of a grammar. We solved this problem with negative (counter) examples to give a penalty to too general grammars.

To get hold of one of the the many parameters we had to set, we ran tests on the parameters of Genetic Programming. We found the same results as in literature. On the one hand a high population size combined with a strong selection. On the other hand, still according to literature, we found that crossover didn't have a great impact on the results compaired to simple mutation. Crossover just decreased the performance which led us to drop it completely.

Good results were achieved with small token languages. We run into problems for more complex languages for which no grammar has been found with our algorithm. The main reason for this was that we used parsers which were very simple and not domain specific. This approach helps keep the steps that can be taken from one generation to next small, Nevertheless sometimes (too) many mutations are needed to generate higher abstractions. This often occurs when the algorithm is stuck in local minima. Although we took measures to avoid local minima in the fitness function, by having lengths limits (lower and higher), negative examples, etc., we couldn't get rid of them.

As the search process is very CPU expensive we implemented a distributed system with one main search program which communicates with several smaller search programs. Besides being now able to distribute the computating we gained another important advantage. By exchanging results between isolated search spaces we improved the grammar diversity and thus increased the chance of finding a good grammar.

## 6.1 Future work

We found some promising results but a lot of work has still to be done. We will look at some ways how the algorithm could be improved in the this section.

One idea would be to allow predefined grammars. They would be defined by the user for some parts of the final grammar that are already known. For instance identifiers, parentheses matching or keywords could be introduced as prefabricated PEG trees. This would accelerate the search process and lead it into the correct direction.

Smaller grammars are easier to find. Thus the source code could be cut into pieces. The search would then be applied to each part separately. For instance in an object oriented language method declarations, class declaration etc. could be separated into groups of with the same grammar. This approach would need a manual concatenation of the

different generated grammars later on. This approach is similar to the approach of the *Example-Driven Reconstruction of Software Models* [10]. Our approach would replace the part where a sub grammar is generated. In order to enlarge the search space, equivalent grammars could be rejected. If two grammars are equivalent, one of both should be preferred. Either based on their size or complexity. This would lead to more different grammars and enrich the diversity.

Improvements can also be achieved by analyzing parser runtime behavior of the PEGs. For instance one can look at activity of the different nodes of a PEG. Small and active subgraphs of PEGs . So on the one hand, nodes which are never or seldomly used, should get a penalty. Highly active nodes, on the other hand, should be reused and have a higher probability of crossover. PEGs are well suited for runtime behavior analysis. We can define functions on all the nodes and easily track their behavior. Particular attention has to be given that we only count the successes on a node to determine it's fitness.

Genetic Programming allows for improvement as well. The biggest problem for this is to determine the parameters correctly. It can be done more or less manually, like in this thesis. One could also think of evolving the parameters themselves. To process many GP runs at the same time is computationally expensive though.

A similar approach would be to adapt the parameters over a GP run. As we saw in chapter 4, the conditions in the beginning of a run are different from the subsequent part. In the beginning grammars tend to be simple and small, while with increasing time the complexity grows. If the parameters would be adapted to the various situations, the solution might be improved.

# Chapter 7

# Appendix

## 7.1 Quickstart

In this section we will give a quickstart for our implementation of the evolutionary search for grammars. It can be executed either as single application with only one population running or it can be run as a server/client application (see 4).

### 7.1.1 Prerequisites

In order to use the program you will need an installation of *Python*. The source can be checked out via *svn* from `https://svn.origo.ethz.ch/geneticgrammarextraction/GGEPython/src`. All the source to be analyzed has to be put into a single folder, along with the negative examples.

### 7.1.2 Single Evolution

With single evolution grammars are searched with one single population. The *Python* file is found in *src/singleevolution.py*. Execute the search with

**Listing 7.1: Single Evolution**

```
python singleevolution.py
  <population size> <surviving population size> <mutation/crossover ratio>
       <generations> <folder> <positive> <negative>
```

### 7.1.3 Master/Slave Evolution

The fully fledged program is used as a client/server-application. For this purpose the server is started with

**Listing 7.2: Single Evolution**

```
python serverevolution.py
```

and subsequently a small number of clients with

**Listing 7.3: Single Evolution**

```
python clientevolution.py
```

Server and client communicate with each other. The result is printed from the server program.

# List of Figures

# Listings

# Bibliography


[1] Grant, Bruce and Rory J. Howlett. *Background selection by the peppered moth (Biston betularia Linn.): individual differences*. Biological Journal of the Linnean Society 33: 217-232, 1988.

[2] Marjan Mernik, Goran Gerlič, Viljem Žumer, Barret R. Bryant. *Can a Parser be Generated From Examples?*. University of Maribor, 2000

[3] Sander Tichelaar, *Modeling Object-Oriented Software for Reverse Engineering and Refactoring* Ph.D. thesis, University of Bern, December 2001

[4] Bryan Ford. *Packrat parsing:: simple, powerful, lazy, linear time, functional pearl*. ICFP '02, October 2002

[5] Jason D. Lohn, Derek S. Linden, Gregory S. Hornby, William F. Kraus, Adaan Rodriguez-Arroyo. *Evolutionary Design of an X-Band Antenna for NASA's Space Technology 5 Mission* Proceedings of the 2003 NASA/DoD Conference on Evolvable Hardware, 2003

[6] Marco Tomassini, Leonardo Vanneschi, Francisco Fernández and Germán Galeano. *Genetic and Evolutionary Computation GECCO 2003*. Springer Berlin / Heidelberg

[7] Bryan Ford. Parsing Expression Grammars: A Recognition-Based Syntactic Foundation. Massachusetts Institute of Technology, January 2004

[8] Bryan Ford. *Parsing expression grammars: a recognition-based syntactic foundation*. POPL '04, January 2004

[9] Oscar Nierstrasz, Stéphane Ducasse, Tudor Gĭrba. *The story of moose: an agile reengineering environment*. Universität Bern, September 2005

[10] Oscar Nierstrasz, Markus Kobel, Tudor Gîrba, Michele Lanza, Horst Bunke. *Example-Driven Reconstruction of Software Models*. Universität Bern, March 2007

[11] Ralph Becket and Zoltan Somogyi. *DCGs + Memoing = Packrat Parsing - But is it worth it?*. University of Melbourne, January 2008

[12] Alessandro Warth, James R. Douglass, Todd Millstein. *Packrat Parsers Can Support Left Recursion*. University of California, Los Angeles and Viewpoints Research Institute, The Boeing Company

[13] Sean Luke, Lee Spector. *A Comparison of Crossover and Mutation in Genetic Programming*. University of Maryland, Hampshire College