BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER SYSTEMS
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

AUTOMATED FILE EDITING USING GENETIC PROGRAMMING

AUTOMATIZOVANÁ ÚPRAVA SOUBORŮ POMOCÍ GENETICKÉHO PROGRAMOVÁNÍ

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. MAREK SEDLÁČEK**
AUTOR PRÁCE

SUPERVISOR **prof. Ing. LUKÁŠ SEKANINA, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2022

Brno University of Technology
Faculty of Information Technology

Department of Computer Systems (DCSY) Academic year 2021/2022

Master's Thesis Specification

24981

Student:	**Sedláček Marek, Bc.**
Programme:	Information Technology and Artificial Intelligence
Specialization:	Bioinformatics and Biocomputing
Title:	**Automated File Editing Using Genetic Programming**
Category:	Artificial Intelligence

Assignment:

1. Familiarize yourself with principles of genetic programming and genetic improvement of software.
2. Design a software tool that allows the user to specify an input-output file transformation by examples and enables learning this transformation using genetic programming.
3. Propose a suitable internal representation of the transformation code and an appropriate compiler.
4. Propose a suitable problem encoding, genetic operators, search method, and fitness function for the genetic programming method.
5. Implement the tool in the chosen programming language.
6. Perform detailed statistical and performance evaluation of the implemented tool using selected benchmarks.
7. Discuss the obtained results and their possible improvements.

Recommended literature:

- According to the instructions of the supervisor.

Requirements for the semestral defence:

- Items 1 to 4 of the assignment.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Sekanina Lukáš, prof. Ing., Ph.D.**
Head of Department:	Sekanina Lukáš, prof. Ing., Ph.D.
Beginning of work:	November 1, 2021
Submission deadline:	May 18, 2022
Approval date:	October 29, 2021

Abstract

File editing is an integral part of today's work for many people, but not everyone has programming skills or deep knowledge of editing tools to make their editing efficient and quick. This is exactly what the program presented in this thesis – Ebe – is trying to solve. Ebe takes snippets of file edits done by the user and using genetic programming it finds the correct algorithm to transform the whole file or even multiple files into the desired output. Ebe consist of multiple parts, which had to be designed and implemented to achieve its goals. For this purpose a new programming language was designed to suite file editing and work well with genetic programming, an interpreter for this language was implemented as well as a compiler that uses genetic programming to synthesize the editing algorithm based on given examples. Ebe was then tested with other tools for file editing. These experiment focused on the overall editing speed and Ebe ended up having better editing times than Python 3 and similar editing times as the language AWK in most experiments. These experiments proved, that for many frequent editing tasks Ebe has a potential as an alternative tool for file editing.

Abstrakt

Úprava souborů je nedílnou součástí práce pro mnoho lidí, ale ne každý umí programovat a nebo má dostatečnou znalost editovacích nástrojů, aby byl tento proces efektivní a rychlý. Toto je přesně to, co se snaží program představený v této práci – Ebe – vyřešit. Ebe z úryvků úprav provedených uživatelem pomocí genetického programovaní nalezne algoritmus na požadovanou transformaci souboru, který je pak možné použít na celý soubor nebo i více souborů najednou. Ebe se skládá z více částí, které musely být navrhnuty a implementovány ke splnění stanovených cílů. Za tímto účelem byl navrhnut nový programovací jazyk pro editaci souborů a kompatibilitu s genetickým programováním. Dále byl implementován interpreter pro tento jazyk a také překladač, který z poskytnutých ukázek syntetizuje editační algoritmus. Ebe bylo poté otestován a porovnáno s dalšími nástroji pro úpravu souborů. Tyto experimenty byly zaměřeny na celkovou editační dobu a Ebe ve většině experimentů dosáhlo lepších časů než jazyk Python 3 a podobných editačních časů jako jazyk AWK. Tyto experimenty potvrdily, že pro mnoho často prováděných úprav má Ebe potenciál jako alternativní nástroj pro tyto úlohy.

Keywords

compiler, interpreter, genetic programming, code synthesis, file editing, programming language

Klíčová slova

překladač, interpreter, genetické programování, syntéza kódu, úprava souborů, programovací jazyk

Reference

SEDLÁČEK, Marek. *Automated File Editing Using Genetic Programming.* Brno, 2022. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor prof. Ing. Lukáš Sekanina, Ph.D.

Automated File Editing Using Genetic Programming

Declaration

I hereby declare that this Master's thesis was prepared as an original work by the author under the supervision of prof. Ing. Lukáš Sekanina, Ph.D. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

. . . .
Marek Sedláček
May 9, 2022

Acknowledgements

I would like to thank my supervisor prof. Ing. Lukáš Sekanina, Ph.D. for his help and interest in this thesis. I would also like to thank my family and friends for all the support during my studies and working on this thesis. Lastly many thanks to Bc. Klára Ungrová for the design of Ebe's logo.

Contents

Chapter 1

Introduction

This thesis introduces a program for automated file editing from given examples and its underlying components – a compiler, which does a code synthesis using genetic programming, a new language designed for file editing and to work well with genetic programming and an interpreter, which can interpret programs written in this language. The goal of this thesis is not just to showcase a working program for the aforementioned task, but also to explore the possibilities of using genetic programming to synthesise programs. The evolutionary approach and more specifically genetic programming display potential to help solving tasks, which would be for a pure algorithmic approach very time consuming or in other ways too overwhelming. The given problem of finding an algorithm to transform one given example into another one is just fit for genetic programming, because these two examples can be easily used to calculate the fitness of some candidate solution. On top of this, many editing tasks done in real life are not too complex and do not require complex algorithms to realize them, which simplifies the task for genetic programming and helps with the time required to find a solution.

The proposed program requires the user to only provide two examples and then perform the desired edits over the whole file. This means that the user does not have to have any programming knowledge and only needs to know how to use the program. For many non-programmers doing editing tasks on large files might mean spending many hours in an editor and doing all these repetitive edits by hand. The proposed tool could help cut down this time immensely.

The goal for this program is to allow non-programmers, but generally anyone who uses it, to achieve editing speeds similar or even better to those knowledgeable in programming and algorithmic thinking, and thus, provide a tool, that could compete with other currently used tools. On top of promising low editing times, the tool also promises minimum required knowledge of it and no need for external interaction once the examples are provided. On the other hand it also promises to allow for customization and tweaks from the outside, if the users wishes to do so.

This program can also prove beneficial to programmers, to whom it might offer an alternative tool, which would solve the problem at hand without any need to think about the algorithm needed, debugging the code and checking output correctness.

In the next chapter the overall problem with editing files and tools used for this are described. Since file parsing is a big part of all these tools and of file editing in general, this is also mentioned here with emphasis on text file editing. Comparison methods for text similarity are mentioned here as well. This chapter also generally describes code compilation and interpretation and mentions some problems in these fields. Bytecode is discussed as it is a big part of interpretation. Chapter 3 deals with evolutionary methods and discusses more in detail genetic programming and its use for code synthesis. The main parts of genetic programming are described with examples and some comparisons of different approaches are given. This final section in this chapter also outlines the problems with exact expression generation and introduces symbolic regression method. Chapter 4 introduces and in detail describes the implemented program Ebe. All its parts including the language (Ebel) designed for it are described with many examples and argumentation for the choices made. Furthermore some experiments and comparisons are presented to support these choices. The end of this chapter is designated to help understand the tool's usage and some advanced features it offers. Chapter 5 then in more detail describes Ebe's implementation and all the supporting tools and the structure around it to support its potential as a full-fledged and usable tool. Examples of interesting problems and their solutions are also mentioned. The final chapter contains 4 experiments that compare Ebe with other tools used for the similar task. This gives some rough approximation of how well Ebe can compare to these tools and if it even can compete with them.

Chapter 2

Compilation methods for file editing

A file editing program can be thought of as a special case of a compiler or rather transpiler. Transpiler is a program, that converts one source code program into another [49]. Such example can be one of the very first transpilers XLT86, which converted 8080 assembly code into 8086 code and nowadays transpilers are quite popular for converting code into web languages such as Java Script [49]. In a case of a file editing program, this "transcompilation" is in many cases from one format to the same, but only changing some values or doing other small edits, which are far away from what usual transpilers do. But the reason for this comparison is to showcase, that many techniques used in compilers, interpreters and transpilers can be also used for automated file editing.

2.1 Current methods for file editing

Nowadays there are many methods and tools for specialized file editing and file transformation. These tools are not only needed in the IT sector, but in many other sectors. File types and their formats vary a lot, and thus, only one tool cannot handle all these types, but rather specializes on certain file type and format. But there still exist tools, which can handle almost all possible formats, but those tools then require the user to put in a lot of effort to make them correctly work for all the possible inputs and cases. Such example can be the programming language AWK, which was designed in the late 1980s by Alfred Aho, Peter Weinberger and Brian Kernighan for the sole purpose of file editing [3]. The whole problem with file editing in computers is very well described in the preface of their book:

> *Computer users spend a lot of time doing simple, mechanical data manipulation – changing the format of data, checking its validity, finding items with some property, adding up numbers, printing reports, and the like. All of these jobs ought to be mechanized, but it's a real nuisance to have to write a special purpose program in a standard language like C or Pascal each time such a task comes up.*
>
> *Awk is a programming language that makes it possible to handle such tasks with very short programs, often only one or two lines long [3].*

This short quote very well conveys, that even in the infancy of modern computing, there was a big need for effective and fast file editing and transformation. And since AWK is used till this day and is a built-in tool for many operating systems [12] only suggests that this need has not gone away, but in fact may have even gone up, with the rise of high computing and large storage capabilities for computers.

But even though tool such as AWK offers file editing in a few lines, as the authors say, writing these lines requires knowledge of the tool and in this case some programming skills. Furthermore the created program is often designed for exactly one file format.

2.1.1 Manual file editing

One of the simplest file editing methods is file editing done manually by hand in a file editor. This approach is fine for small one time edits, but can get very impractical and take a lot of time, when done on a large file or even multiple files and may very easily lead to errors.

The biggest upside to this approach is that it can be done by someone, who has no programming skills and only very little knowledge regarding the use of a file editor.

There are also editors like Vim (*Vi iMproved*), which allow for more advanced editing using special shortcuts, editing modes and methods. Examples of this are replacing text based on regular expression, editing multiple lines at once using vertical selection or even doing calculations with selected numeric values [48]. On the other hand such editors are usual not very intuitive and require the user to learn the tool more in depth.

2.1.2 Automated file editing

Automated file editing in this sense means a program that takes an input file, transforms it and outputs this newly edited file. For example this can be a script written in any language that does this job, such as a script in the AWK programming language, as mentioned in the chapter intro. Another example of automated file editing can be a specialized tools, such as converters from one file format to another or in some sense even a compiler, which converts input program into a program in a different language.

Unlike manual approach, use of an automated file editing brings in the option to write the script once and then execute it over any number of files that have the same format. On the other hand it requires someone skilled in programming to write such a script. This approach can also bring in errors like manual file editing, since the written script might not account for all the cases or even contain errors itself.

```
awk 'BEGIN{RS=""; FS="\n"} {split($1,a,","); host[a[42]] = 0; for (i=1; i<=
    NF; i++) if (match($i,"closed") != 0) host[a[42]]++} END{for (each in
    host) print each ": " host[each]}' in.nmap
```

Listing 2.1: AWK example of file editing, which might even be illegible to those who are programmers, but don't use AWK.

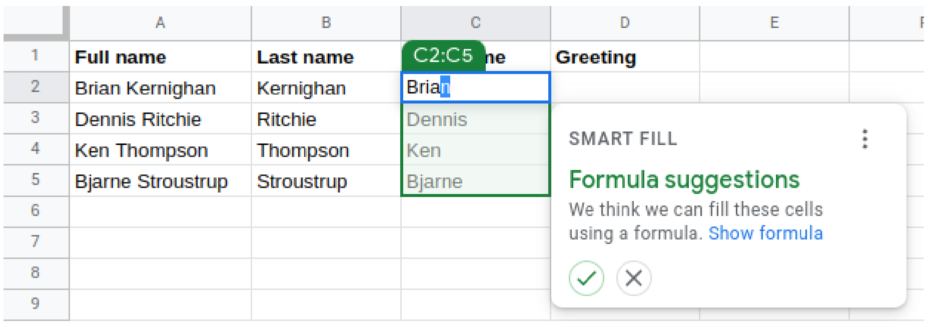
```
import csv
import sys

if len(sys.argv) < 2:
    exit(1)
out_text = []
with open(sys.argv[1], "r") as gtf_file:
    reader = csv.reader(gtf_file, delimiter='\t')
    for line in reader:
        line[3] = str(int(line[3]) - 153350000)
        line[4] = str(int(line[4]) - 153350000)
        out_text.append("\t".join(line))
with open(sys.argv[1]+".out", "w") as out_file:
    out_file.write("\n".join(out_text))
```

Listing 2.2: Python 3 example of file editing, which might be more legible, than AWK, but require more lines of code.

2.1.3 File editing from examples

For special cases, such as working with tables, there are tools, which allow for automated file, or rather cell, editing with detection of the editing pattern. Example of this is the Google Sheets Smart Fill feature or Microsoft Excel Flash Fill feature, which detects simple patterns used and applies these to other similar rows (this can be seen in Figure 2.1), unfortunately this algorithm cannot find all combinations and might fail even with quite a simple one (see Figure 2.2).



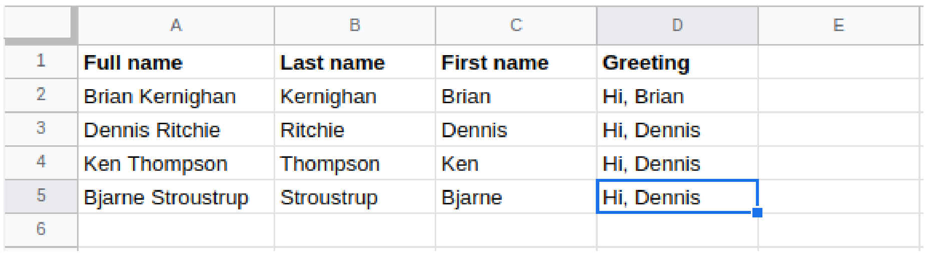
	A	B	C	D	E
1	**Full name**	**Last name**	C2:C5 **ne**	**Greeting**	
2	Brian Kernighan	Kernighan	Brian		
3	Dennis Ritchie	Ritchie	Dennis		
4	Ken Thompson	Thompson	Ken		
5	Bjarne Stroustrup	Stroustrup	Bjarne		
6					
7					
8					
9					

SMART FILL

Formula suggestions

We think we can fill these cells using a formula. Show formula

Figure 2.1: Google Sheets Smart Fill feature.

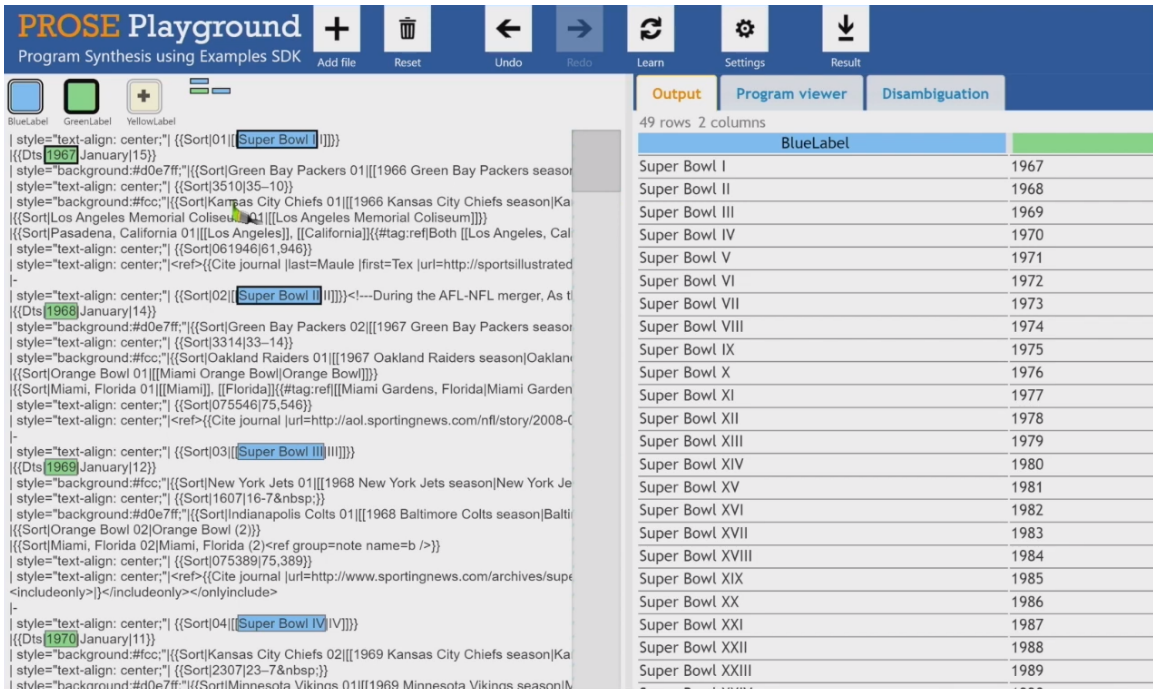


	A	B	C	D	E
1	**Full name**	**Last name**	**First name**	**Greeting**	
2	Brian Kernighan	Kernighan	Brian	Hi, Brian	
3	Dennis Ritchie	Ritchie	Dennis	Hi, Dennis	
4	Ken Thompson	Thompson	Ken	Hi, Dennis	
5	Bjarne Stroustrup	Stroustrup	Bjarne	Hi, Dennis	
6					

Figure 2.2: Example, where Google Sheets Smart Fill feature cannot find the correct pattern.

Microsoft also offers framework PROSE, which can from given examples provide a program in domain-specific language, which adheres to given examples [51] [2]. This framework consists of multiple other projects, that do programming by example, such as FlashExtract [37] or FlashRelate [6]. Part of this framework is also the aforementioned Flash Fill.

PROSE offers also an interactive interface (see Figure 2.3), where the user can select values for extraction or other operations and have the generated program be executed over the input as well.



```
| style="text-align: center;"| {{Sort|01|[[Super Bowl I|I]]}}
|{{Dts|1967|January|15}}
| style="background:#d0e7ff;"|{{Sort|Green Bay Packers 01|[[1966 Green Bay Packers seaso
| style="text-align: center;"| {{Sort|3510|35–10}}
| style="background:#fcc;"|{{Sort|Kansas City Chiefs 01|[[1966 Kansas City Chiefs season|Ka
|{{Sort|Los Angeles Memorial Coliseum 01|[[Los Angeles Memorial Coliseum]]}}
|{{Sort|Pasadena, California 01|[[Los Angeles]], [[California]]{{#tag:ref|Both [[Los Angeles, Cal
| style="text-align: center;"| {{Sort|061946|61,946}}
| style="text-align: center;"|<ref>{{Cite journal |last=Maule |first=Tex |url=http://sportsillustrated
|-
| style="text-align: center;"| {{Sort|02|[[Super Bowl II|II]]}}<!---During the AFL-NFL merger, As t
|{{Dts|1968|January|14}}
| style="background:#d0e7ff;"|{{Sort|Green Bay Packers 02|[[1967 Green Bay Packers seaso
| style="text-align: center;"| {{Sort|3314|33–14}}
| style="background:#fcc;"|{{Sort|Oakland Raiders 01|[[1967 Oakland Raiders season|Oaklan
|{{Sort|Orange Bowl 01|[[Miami Orange Bowl|Orange Bowl]]}}
|{{Sort|Miami, Florida 01|[[Miami]], [[Florida]]{{#tag:ref|[[Miami Gardens, Florida|Miami Garden
| style="text-align: center;"| {{Sort|075546|75,546}}
| style="text-align: center;"|<ref>{{Cite journal |url=http://aol.sportingnews.com/nfl/story/2008-
|-
| style="text-align: center;"| {{Sort|03|[[Super Bowl III|III]]}}
|{{Dts|1969|January|12}}
| style="background:#fcc;"|{{Sort|New York Jets 01|[[1968 New York Jets season|New York Je
| style="text-align: center;"| {{Sort|1607|16-7&nbsp;}}
| style="background:#d0e7ff;"|{{Sort|Indianapolis Colts 01|[[1968 Baltimore Colts season|Balti
|{{Sort|Orange Bowl 02|Orange Bowl (2)}}
|{{Sort|Miami, Florida 02|Miami, Florida (2)<ref group=note name=b />}}
| style="text-align: center;"| {{Sort|075389|75,389}}
| style="text-align: center;"|<ref>{{Cite journal |url=http://www.sportingnews.com/archives/supe
<includeonly>|}</includeonly></onlyinclude>
|-
| style="text-align: center;"| {{Sort|04|[[Super Bowl IV|IV]]}}
|{{Dts|1970|January|11}}
| style="background:#fcc;"|{{Sort|Kansas City Chiefs 02|[[1969 Kansas City Chiefs season|Ka
| style="text-align: center;"| {{Sort|2307|23–7&nbsp;}}
| style="background:#d0e7ff;"|{{Sort|Minnesota Vikings 01|[[1969 Minnesota Vikings season|M
```

BlueLabel	
Super Bowl I	1967
Super Bowl II	1968
Super Bowl III	1969
Super Bowl IV	1970
Super Bowl V	1971
Super Bowl VI	1972
Super Bowl VII	1973
Super Bowl VIII	1974
Super Bowl IX	1975
Super Bowl X	1976
Super Bowl XI	1977
Super Bowl XII	1978
Super Bowl XIII	1979
Super Bowl XIV	1980
Super Bowl XV	1981
Super Bowl XVI	1982
Super Bowl XVII	1983
Super Bowl XVIII	1984
Super Bowl XIX	1985
Super Bowl XX	1986
Super Bowl XXI	1987
Super Bowl XXII	1988
Super Bowl XXIII	1989

Figure 2.3: Screenshot of PROSE framework extracting data from the values selected by the user.

2.2 File parsing

Parsing is a process, where a parser derives a syntactic structure for the program, fitting the words into a grammatical model. Before such structure can be created, a scanner needs to parse so-called lexemes from the input source [61]. These lexemes are defined by creator of the scanner and parser. In case of programming languages the lexmes are objects such as keywords, variables, constants. . . .

Parsers and scanners process certain language. If the input does not adhere to this language – meaning a string is not generated by grammar of such language [29] – the parser won't accept this string. In the sense of general file parsing this language can be very strict, such as in a case of a file, which may contain only numbers. This file format could be very easily parsed using regular grammar. Example of such a grammar can be seen in specification 2.1, where a regular grammar RG is as a quadruple $RG = (\Sigma, \Delta, R, S)$, where Σ is a finite alphabet of non-terminal symbols, Δ is a finite alphabet of terminal symbols and $\Sigma \cap \Delta = \varnothing$, S is a starting non-terminal ($S \in \Sigma$) and R is a finite set of rules in a form of $\gamma \to \alpha$, where $\gamma \in \Sigma$ and α is either a non-terminal or a non-terminal and a terminal.

$$G_{numbers} = (\{S\}, \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}, \{S \to SS|0|1|2|3|4|5|6|7|8|9\}, \{S\}) \qquad (2.1)$$

In theory and mainly in practice regular expressions are frequently used tool for defining regular languages and can be therefore used for file parsing. The regular expressions over a finite alphabet Δ and the language that these expressions denote are defined recursively as follows [29]:

1. $\varnothing$ is a regular expression denoting the empty set,
2. ε is a regular expression denoting $\{\varepsilon\}$,
3. a, where $a \in \Delta$, is a regular expression denoting $\{a\}$,
4. if r and s are regular expressions denoting languages R and S, respectively, then
   (a) $(r|s)$ is a regular expression denoting $R \cup S$,
   (b) (rs) is a regular expression denoting RS,
   (c) $(r)^*$ is a regular expression denoting R^*.

Regular expressions are quite popular and many languages offer built-in libraries for regular expressions, but the syntax used to write regular expressions may differ in each language [21].

In other cases to parse a file, a stronger grammar might be needed. For example, a file containing a simple C-based language code with conditional statements and simple expressions could not be parsed using a regular grammar. For this case a context-free grammar can be used, which is defined once again as a quadruple $CFG = (\Sigma, \Delta, R, S)$, where Σ, Δ and S follow previous definitions, but the rules for R are following $\gamma \to \alpha$, where $\gamma \in \Sigma$ and $\alpha \in (\Sigma \cap \Delta)^*$ [29]. Then a context free grammar for this simple language could look the following:

$$
\begin{aligned}
& G_C = (\Sigma, \{if, (,), \{,\}, x, y, +, -, *, /, =, ;, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}, R, \langle STMT\rangle) \\
& \quad \Sigma = \{\langle STMT\rangle, \langle ID\rangle, \langle EXPR\rangle, \langle STMLIST\rangle, \langle NUM\rangle, \langle OP\rangle\} \\
& \quad R = \{ \\
& \qquad \langle STMT\rangle \to \{\langle STMLIST\rangle\} \\
& \qquad \langle STMT\rangle \to \langle ID\rangle = \langle EXPR\rangle; \\
& \qquad \langle STMT\rangle \to if(\langle EXPR\rangle)\langle STMT\rangle \\
& \qquad \langle STMLIST\rangle \to \langle STMT\rangle \\
& \qquad \langle STMLIST\rangle \to \langle STMLIST\rangle\langle STMT\rangle \\
& \qquad \langle EXPR\rangle \to \langle ID\rangle \mid \langle NUM\rangle \mid \langle EXPR\rangle\langle OP\rangle\langle EXPR\rangle \\
& \qquad \langle NUM\rangle \to 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\
& \qquad \langle ID\rangle \to x \mid y \\
& \qquad \langle OP\rangle \to + \mid - \mid / \mid * \\
& \quad \}
\end{aligned}
\tag{2.2}
$$

As showcased in specification 2.2, a grammar can be a very powerful tool for file parsing, that guarantees its correctness based on its mathematical grounds. Its implementation can be done in many ways. There are many tools for implementing these grammars such as YACC and LEX [33]. These tools can save a lot of time hand writing a parser, its debugging and checking for correctness by just writing a grammar.

2.3 Text file parsing

Any file can be parsed including byte files such as images, pdfs or music files, but more often file parsing based on lexemes makes more sense for text files, where one file might be correct representation of multiple file formats. For example s simple C code might represent a valid C program, C++ program and a raw text file.

When parsing a raw text file, meaning a text file without specified format, its content is always valid and a parser should account for this. Automated text editor for raw text files should account for this. In its parser numerous lexemes can be defined (for example numbers, delimiters, mathematical operators...), but at the same time there should be a special case for symbol or string, which cannot be parsed into any other lexeme.

Let's assume a parser and a scanner, which defines the following lexemes and regular expressions for parsing them: number (`[0-9]+`), delimiter (`[,.;!| \/-\n]`), symbol (`[#$&(){}\[\]]`) and text (`[a-zA-Z]`). This parser would accept and parse many simple text files, but would halt (not accept) for example the following file:

```
FACT:
Otakar Borůvka was a Czech mathematician.
```

Listing 2.3: Example of text file, which might require special lexemes for unexpected input.

It is also worth mentioning, that in the case of Listing 2.3 just adding a special lexeme for anything that cannot be parsed by any other lexeme does not solve all the problems, since this approach might bring other problems. This special lexeme might be defined for a single character, which in a case of character `:` would be the correct approach, but then for parsing the name `Borůvka` the parser (scanner) would split this name into 3 different lexemes (text `Bor`, other `ů` and text `vka`). In this case connecting the special symbol to the last and following lexeme would be correct, but then this would create lexeme `FACT:`, which might cause other problems. This problem might get even more complicated when working with UNICODE text, where characters may differ in byte size or the text might containing characters like emojis [14].

Another problem could arise from localization, where a different format might be used for certain lexeme based on the authors culture and local standards. Example could be the number "42.200" and "42,200", where both notations are valid format for real number with decimal expansion and also for an integer [40].

2.4 Text comparison methods

Often when working with text files, there is a need to compare two strings. This comparison might be as simple as to check if the strings fully match, but sometimes more complex comparison might be needed, where the strings are compared for their similarity. Methods for string comparisons are needed in many fields and there is already a lot of research behind these methods. For example, bioinformatics often works with genetic sequences, which are represented as strings of letters and here the difference between two sequences (strings) is often needed to determine correct alignment or deduct some properties [7]. Since these methods are looking at the differences between two strings, they often use the word "distance" between two strings. This distance usually represents how many edits (for example deletions or insertions of single characters) are needed to make the two strings identical [38]. If this distance d is then divided by the overall length of the longer string and multiplied by 100 this results in the similarity s of these strings in percentage (see Equation 2.3).

$$s_{t_1,t_2} = \frac{d}{\max\left(|t_1|, |t_2|\right)} \cdot 100 \tag{2.3}$$

2.4.1 Levenshtein distance

In 1966 a Russian mathematician Vladimir I. Levenshtein published his paper titled in English as “Binary codes capable of correcting deletions, insertions, and reversals” [38], which was based on an earlier paper by R. W. Hamming [27]. In this paper, Levenshtein presents his method for measuring how much two string are different (their edit distance) from each other based on three possible operations done to one of the strings:

- deletion – a character is removed form the string,
- insertion – a new character is inserted into the string,
- reversal – two characters are swapped.

Each operation executed increments the overall distance by one.

```
**Input:** Strings $s$ and $t$.
**Data:** Levenshtein distance between $s$ and $t$.
$m \leftarrow |s|$
$n \leftarrow |t|$
$d$ = matrix$[0..m, 0..n]$ of zeroes
for $i \leftarrow 1$ to $m$ do
    $d[i, 0] \leftarrow i$
end
for $i \leftarrow 1$ to $n$ do
    $d[0, i] \leftarrow i$
end
for $j \leftarrow 1$ to $n$ do
    for $i \leftarrow 1$ to $m$ do
        $c \leftarrow 1$ if $s[i] = t[j]$ then
            $c \leftarrow 0$
        end
        $d[i, j] \leftarrow \min(d[i-1, j] + 1, d[i, j-1] + 1, d[i-1, j-1] + c)$
    end
end
return $d[m, n]$
```

Algorithm 1: Pseudo code for Levenshtein distance algorithm.

As seen in Algorithm 1, Levenshtein distance uses a matrix of size $m \times n$, where m and n are lengths of the 2 compared strings. When comparing 2 strings with same or very similar lengths this results in space complexity of $O(n^2)$. This complexity unfortunately appears in most string comparing methods that measure edit distance, because of the use of dynamic programming approach [26]. It is also unfortunately proven that this algorithm cannot run in strongly subquadratic time ($O(n^{2-\delta})$ for some constant $\delta > 0$) unless the strong exponential time hypothesis is false [5].

2.4.2 Jaro distance and Jaro-Winkler distance

In 1985 Matthew A. Jaro proposed a new method for measuring similarity of two strings [31] (now referred to as the Jaro distance) and in 1990 this method was built on by William E. Winkler [63] (now referred to as the Jaro-Winkler distance). The latter method mostly only adjusts the first method results with additional computations, so it will take longer, but might calculate more accurate similarity. Unlike Levenshtein's method, these methods return the similarity, where 0 means no similarity and 1 means identity [11].

$$\Phi_j(s,t) = \frac{1}{3}\left(\frac{N}{|s|} + \frac{N}{|t|} + \frac{0.5T}{N}\right) \tag{2.4}$$

$$d_m(s,t) = \left\lfloor \frac{\max(|s|,|t|)}{2} \right\rfloor - 1 \tag{2.5}$$

The Equation 2.4 showcases Jaro's approach to calculating the similarity [11] of strings s and t, where N is the number of matching characters of strings s and t. Matching characters are characters, which are the same or are not farther than distance d_m specified in expression 2.5.

Winkler then uses the same formula, but gives higher similarity values for strings, which match from the beginning for a set prefix length, i.e.:

$$\Phi_w(s,t) = \Phi_j(s,t) + \alpha \cdot P(1 - \Phi_j), \tag{2.6}$$

where α is the length of common prefix of s and t up to 4 characters and P is a constant scaling factor, where Winkler used the value of 0.1 in his original paper [63].

2.5 Compilation of source code

Compilation is a process of transforming program in some input language into another language, which in many cases is assembly language or binary code [61]. Compilation into binary code is not always needed and some compilers, such as the Java compiler, compiles into some simpler intermediate language, which is still above the binary code [41]. But to achieve this transformation, there are many steps in between and many other optional ones to optimize or achieve other results. A very general and simplified syntax-directed compilation process could be described as following [61]:

1. The source program is loaded into a lexical analyzer (also called *scanner*), which, once called by a syntax analyzer (*parser*), returns a token (scanners and parsers are also described in Section 2.2). These tokens (also called *lexemes*) the scanner extracts from the code are defined by the language's grammar and there are many methods of extracting them, where one of the popular ones is using regular expressions [39].

2. The parser is continuously requesting tokens from the scanner and constructing from them parse trees. These trees are then analyzed for their semantic correctness in semantic analyzer, which may or may not be its own separate unit. The semantic analyzer creates an abstract syntax tree from a parse tree.

3. The abstract syntax tree then goes into an intermediate code generator, which generates some intermediate representation, which level of abstraction is way lower than

the input program, but still above the binary code. Example of such representation might be a three-address code or even something more complex such as internal LLVM representation [47]. Even though it is possible to generate a binary code right from the input program it would be ill-advised, because nevertheless the chosen representation, lowering the abstraction heavily simplifies and helps with following steps.

4. If one were to choose the path of not generating an intermediate code, the compilation could end there, but one of the main reasons for using an intermediate code is to do optimizations and there are many optimizations that can be done here.

5. Once the code has been optimized it can be finally transformed into the binary representation with a code generator and with this step the compilation had produced the target program.

It is worth mentioning, that the above process is in no way the only approach, quite the opposite. In some cases the the compilation might require some steps to be added, some to be removed, some to be repeated multiple times. And in practice many of the processes might appear in other units, such as optimization, which might appear already in the parser in a form of constant folding for example (see Listing 2.4). It should also be noted that each step might be a very complex unit with many sub-units and problems, which needs to be solved to make a working compiler. For example, even the final part – binary code generation – brings in many problems such as register allocation, communication with operating system or just the general problem of different instruction sets [61].

```
expr  : NUMBER         { $$ = atoi($1); }
      | '(' expr ')'   { $$ = $2;       }
      | expr '%' expr { $$ = $1 % $3;   }
      | expr '*' expr { $$ = $1 * $3;   }
      | expr '/' expr { $$ = $1 / $3;   }
      | expr '-' expr { $$ = $1 - $3;   }
      | expr '+' expr { $$ = $1 + $3;   }
      ;
```

Listing 2.4: A snippet of YACC code to showcase the optimization done already in the parser. The code defines an integer expression `expr` and when working with only constants it already computes arithmetical operations at compile time.

2.6 Interpretation of source code

Interpretation is a process, where the input source code is executed by an interpreter. The main difference from compilation (described in Section 2.5) is that interpreter does not generate binary code or some other output program, but rather executes the code (in a form of a binary code) right away [49]. There can be as usual some exceptions, where such exception is the language Java, which uses both processes – compilation and interpretation – the Java compiler generates a Java Bytecode, which is then run in the Java Virtual Machine, which acts as the interpreter [41]. And quite similar to compilation, the interpretation process can be very complex. Many interpreters, such as CPython [53], also generate intermediate code, which can be optimized and then executed. In this case the interpretation process is the same as the compilation process described in Section 2.5, but the last step is code execution instead of binary code generation.

There are many advantages to interpretation compare to compilation. Nowadays interpreters are becoming more and more popular because of the increasing speed of computers [49]. This increase of course also helps speeding up compilers, but interpreters usually allow for quicker development since the binary code generation might be a very difficult task. This can be simplified by using a framework such as LLVM, which already offers "backends" for binary code generation [47], but interpreters also can be chosen for the other features they offer, such as being able to modify the code during interpretation or just the fact, that most checks are done on runtime, which allows for easy prototyping.

There are also many disadvantages to interpretation compare to compilation. As already mentioned, interpretation in practice should take longer, since the scanning and parsing has to be done during the code interpretation. Compiled code also gets rid of the need to share the full source code, which in many cases might be a needed feature and many other features, which are quite often dependent on the compiler or language. For example, J. Merelo-Guervós et al. [43] ranked different languages for genetic algorithm tasks and showcased, that even many of the interpreted languages have high performance and it's not only about being interpreted or compiled, but about other aspects as well.

It is worth mentioning, that in many cases language definitions do not specify if a compiler or interpreter should be used [32], since this would be equal to specifying which parsing methods should be used, and thus, one language might have multiple compilers as well as interpreters.

2.6.1 Bytecode-like language compilation and interpretation

Bytecode is a low abstraction language, which might represent an intermediate code in an interpreter or a compiler [25] [41]. As already mentioned in previous sections, the bytecode (which might serve as intermediate code) is used for the purpose of simpler optimization and binary code generation (for compilers) or code execution (for interpreters). Bytecode has other positive features, such as its speed, where for example Java, which interprets a bytecode has performance very similar to compiled code [43]. Bytecode also showcased to be one of the best representations for formal code verification [10] and if compiler outputs a bytecode, then the source code does not have to be shared.

```
0  LOAD_GLOBAL              0 (print)
2  LOAD_CONST               1 ('Hello, there!')
4  CALL_FUNCTION            1
6  POP_TOP
8  LOAD_CONST               0 (None)
10 RETURN_VALUE
```

Listing 2.5: Bytecode generated by CPython 3 interpreter for simple method, which prints text "Hello, there!".

```
void sayHello();
Code:
   0: getstatic     #3
   3: ldc           #4
   5: invokevirtual #5
   8: return
```

Listing 2.6: Bytecode generated by Java compiler for simple method, which prints text “Hello, there!”.

```
function sayHello() --line 1 through 3
1    GETTABUP     0 0 0    ; _ENV "print"
2    GETTABUP     1 0 1    ; _ENV "i"
3    GETTABUP     2 0 2    ; _ENV "v"
4    CALL         0 3 1    ; 2 in 0 out
5    RETURN0
end
```

Listing 2.7: Bytecode generated by Lua compiler for simple method, which prints text “Hello, there!”.

Listings 2.5, 2.6 and 2.7 showcase different bytecodes used by different compilers and interpreters. All of the examples do the same task, but very much differ from each other. A good bytecode design can be as important as the input language design, since the bytecode has to be able to perform everything the input code does so that the output code is semantically the same. Not only that the bytecode has to correctly map to the input code, it should also justify its use, by bringing in already discussed features, like having optimizations done over it.

If a language is compiled, but outputs a bytecode, then this bytecode can always be compiled into binary code, if needed be. For Java there exists multiple compilers, that can do this task, such as *Launch4j*[1].

In case of interpreted languages, there can also be optimization in a form of “just-in-time compilation”, which is a process in which a code that is often executed (such as code inside a loop) is compiled right into a binary code for faster execution [24]. Since the process of compiling a bytecode into a binary code during its interpretation requires the compilation itself and then the execution of the binary code, it is not very beneficial to do just-in-time compilation for the whole code, but rather for parts of the code, where the binary code speedup would overcome the just-in-time compilation time. This task of detecting when to do just-in-time compilation is quite difficult, since it might not be possible to determine during compilation, which parts would benefit this. For example Java calls these parts of code “HotSpots™” and has its own HotSpot™ detection system, which at runtime identifies such parts of the code and does just-in-time compilation [44]. With this approach the whole process of HotSpot™ detection increases the interpretation time since it has to run alongside the interpretation. For the detection to be beneficial, it has to bring in more speedups with its just-in-time compilations, then it takes away by being present and doing runtime analysis. Another example can be *Numba*[2], which is a just-in-time compiler for Python and its library Numpy. Unlike Java’s HotSpot™, Numba uses decorators, so the user has to manually specify, which parts should be compiled.

[1]http://launch4j.sourceforge.net/
[2]https://numba.pydata.org/

Chapter 3

Code synthesis using genetic programming

Genetic programming has a large use, from hardware circuit design [22] [45] to many other fields, but this section will focus only on its use regarding code synthesis.

Genetic programming might for people, who are unacquainted with it, seem like only a random process that is very much similar to just a random generation, but this is not the case. Even though randomness plays a big part in genetic programming and evolution-based algorithms, it is not a random process. Randomness is merely here to help find new path for evolution. Genetic programming is in more details described in Section 3.2

3.1 Evolutionary algorithms

For evolution to happen in the nature, there needs to be the following [35]:

- Ability to reproduce.
- There needs to be a population of such reproducing entities.
- There needs to be some variety among the entities in a population.
- Some difference in ability to survive in the environment is associated with the variety.

In nature the variety is manifested in the form of chromosomes and results in differences that either helps the entities survive or the opposite. Evolution is all about survival of the fittest, meaning that the entities with the best fitness are the ones that survive and their valuable attributes (in a form of chromosomes) are passed to future generations [17].

The genetic algorithms are inspired by Darwin's evolutionary process. Evolutionary algorithms transform a set (population) of individual objects into a new population using their fitness and genetic operations [35].

The strength of evolutionary algorithms is that they can effectively search complex spaces with having only a fitness function capable of assigning a fitness score to each candidate solution. Nothing else, such as its derivative, is supposed.

3.2 Genetic programming

Genetic programming (*GP*) is a technique popularized by John Koza in the 1990s as a method for automated programming [34]. The whole process of genetic programming somewhat imitates evolution in nature. It starts with randomly initialized population of candidate solutions, which in the case of evolving algorithms would be simple programs, but in other cases it could be circuits or just anything else. Quite often in genetic programming these phenotypes are represented as trees [36], although this representation is not the only one possible, as can be seen in later chapters. These candidate solutions (phenotypes) are crossed with each other, mutated, scored and then the best ones are selected into future populations based on the scoring [19].

The fitness function guides the evolution, by favoring phenotypes showing a better performance [34]. Without a fitness function, there is no way to guide the evolution.On the other hand, this limits genetic programming to a specific set of problems, where it can be employed.

The need for fitness function, somewhat limits the use for code generation, since often the implementation of fitness function would result in no longer needing the GP. This would be because the fitness function would be the solution itself. An example of this would be when one were to evolve a sorting algorithm using genetic programming. The input would an unsorted list of values and the synthesised program needs to sort these values in ascending order. The problem here would be scoring the phenotypes. To score them, the fitness function would not only need to have the final sorted list, but also some means of telling how close is some partially sorted list to a fully sorted one. If the fitness function had these means, the fitness function could be used for this job instead of the generated program. This approach could find some use, if it was utilized to find a new algorithm or optimize an existing one.

But this limitation does not mean that genetic programming has no use in this field. On top of generating code, where fitness function can be provided (such as algorithms for file editing), it can help optimize programs (their time or even space complexity), by evolving parts of the code into ways a programmer would not think of [15] or it can also help automatically fix bugs [20]. In other fields GP can be used for tasks such as classifier or predictor design [45], generating computer art [62], composing music [42], help with problems in quantum computing [57] and many other uses.

Nowadays there is a big popularity of using neural network and deep learning for solving many problems. These approaches are in some ways also similar to evolutionary approach by using loss functions, which are very similar to fitness functions – a function that is guiding the learning [64]. But as showcased in this paper and many cited papers, there can be found many cases, where genetic programming could take place for neural networks, which could bring in many advantages and simplification.

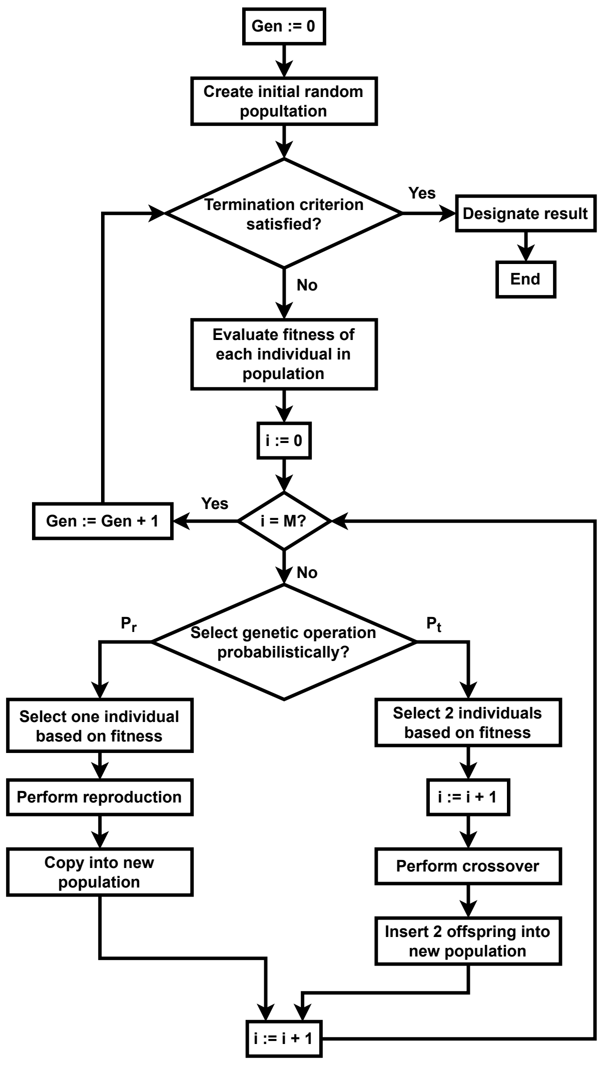
Genetic programming takes the approach of genetic algorithms (described in Section 3.1) and uses crossovers, mutations and selections to find desired solution. Most of the approaches have similar features with some alterations (J. Koza's approach can be seen in Figure 3.1) and here are the features:

1. Initial random population of phenotypes is generated.

2. Every phenotype is scored using fitness function.

3. Main evolutionary cycle starts and ends once a perfectly fitting phenotype is found or other termination condition is met (such as exceeding the evolution time or iterations done).

4. In the cycle selected phenotypes are crossed over, some may be mutated and then new population is created by selection from all these phenotypes.



Gen := 0
Create initial random popultation
Termination criterion satisfied?
Yes
Designate result
End
No
Evaluate fitness of each individual in population
i := 0
Gen := Gen + 1
Yes
i = M?
No
P_r
Select genetic operation probabilistically?
P_t
Select one individual based on fitness
Perform reproduction
Copy into new population
Select 2 individuals based on fitness
i := i + 1
Perform crossover
Insert 2 offspring into new population
i := i + 1

Figure 3.1: Flowchart for the genetic programming paradigm described by J. Koza [35], where M is the size of population and Gen hold the current generation number.

5. New population is once again scored and main cycle repeats.

The general process of genetic programming and its parameters (such as the population size, probability of crossover...) may vary and should vary depending on the problem begin solved. There is not one specific algorithm that would always deliver the best results. The following sections talk more in detail about the specific parts and operations in genetic programming.

3.2.1 Genotype and phenotype

In nature a phenotype is set of characteristics of a living organism, resulting from its combination of genes and the effects of its environment [56]. In genetic programming this is very much similar. Phenotype is the product of a genotype, where genotype is some structure (an abstract syntax tree, a string of bits, a graph...) and phenotype is a candidate solution to some problem [50]. Example of this can be an abstract syntax tree holding some expression, which would represent the genotype and applying it to a problem (converting to an expression) would represent the phenotype (see Figure 3.2).

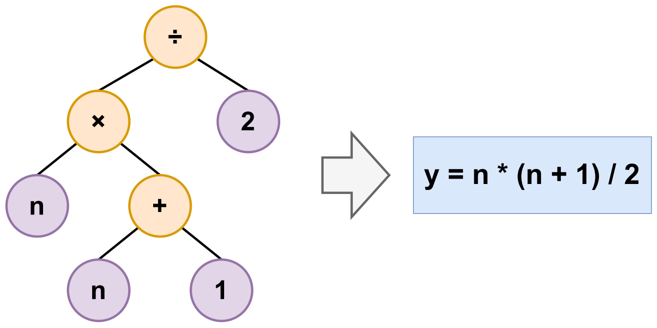


Figure 3.2: Transformation of genotype to a phenotype. An abstract syntax tree (*left*) is the genotype and a generated expression from it (*right*) is the phenotype.

There are many ways to represent a genotype in a computer for the purpose of genetic programming. One of the popular representations is a tree representation, which allows for easy genetic operations [36]. Another example of genotype can be list of integers (or possibly just string of bits) used in Cartesian genetic programming. But a genotype can be represented in any way, which is beneficial to the genetic algorithm and which allows to apply genetic operations.

The tree representation allows for simple operations by working with subtrees or changing values of nodes. In case of using a bit string, the positions of bits play a big role and determine their semantic value [45]. Genetic operations can work with these sections and swap them between phenotypes in case of crossovers or “flip” one bit to apply a mutation. Figure 3.5 showcases how a crossover operation could work for a genotype represented by a graph and Figure 3.7 showcases possible mutation operation on such genotype.

Genotype representation might be in many cases quite an important choice, since the genetic operations done over will happen hundred or even thousands of times, and so, a representation, which is slow will drastically slowdown the evolution process. This fact can be seen in Figure 3.3, which displays results from the same genetic programming algorithm, but using two different genotype representations. In reality this algorithm only simulated Cartesian genetic programming by having a genotype with n triplets of boolean values. Even though this algorithm wasn’t solving any real world problem it did all the operations discussed – crossover, mutation, population evaluation and selection. Both versions evolved 1000 times one population with 50 phenotypes with the size of 12 triplets. The results clearly showcase that the internal representation of genotype matters, when it comes to the GP’s time performance. But this facts is not surprising since this is rather a problem of language abstractions and its overhead. In this case Python 3 language was used, but if a language with lower overhead was used, such as C++ [59], then the difference could be way lower. Nonetheless it is always good to keep in mind that the genetic operations will be executed frequently, and hence, they are a good place for optimizations.

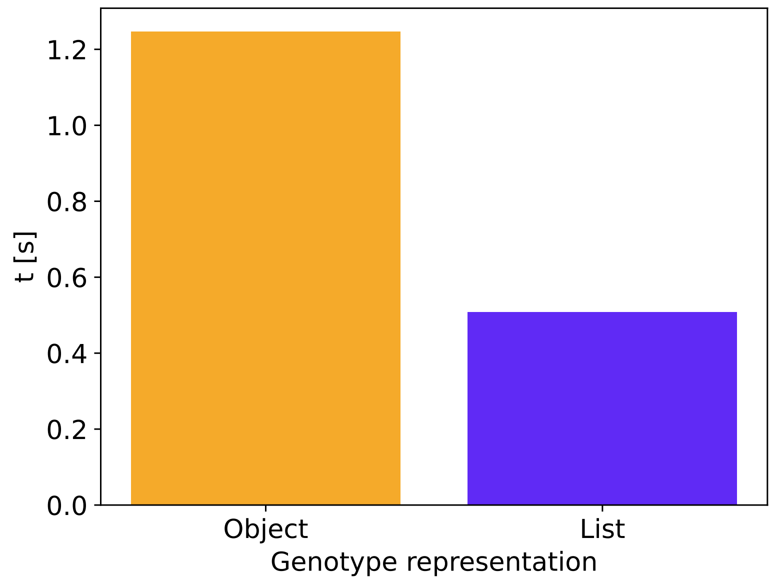


Figure 3.3: Median values from 10 measured runs of the same algorithm, which simulated genetic programming in Python 3. The algorithm used for "Object" value genotype representation in form of an object with public attributes and for "List" it used only a simple list, where the positions represented the attributes.

3.2.2 Population

Population is set of candidate solutions (phenotypes), in which the phenotypes crossover with other selected phenotypes from this population and changes to the population are done with the means of selection [35]. In the GP process described by J. Koza [35], there is only one population, but there are also some approaches utilizing multiple populations – "sub-populations" – and migrate phenotypes between them. This approach even displayed some performance improvements [55]. Sometimes it might be also beneficial to run multiple populations (multiple evolutions) instead of having just one that is being evolved until fit.

3.2.3 Fitness and fitness functions

Fitness is the driving force of Darwinian natural selection and, likewise, for genetic algorithms and genetic programming. In nature fitness is the the probability that the entity survives until reproduction age and reproduces [35]. In genetic programming fitness should also measure the probability of achieving phenotype's goal. Fitness can often be explicit and measured by a fitness function, but it is also possible to use implicit fitness [8].

Fitness can have many representations. In case of symbolic regression, fitness can be the error between phenotype's results on a given data set and expected results [30] (more about symbolic regression can be found in Section 3.3). In case of working with strings, it can be the Levenshtein distance between phenotype's string and desired result and in video games it could be the phenotype's score, where higher score is better. . . . In all these cases, it does not really matter the way we express the fitness as long as we have a way to compare one phenotype with other and tell if it has a better, the same or worse fitness. So the fitness could be a real value, integer, but using something like string would require additional function that could compare these values. This fact is based on fitness' definition, where if two fitness values cannot be compared, then it is impossible to tell, which phenotype has a “better” fitness. Mathematically speaking, this implies that a set of fitness function values have to be totally ordered. It is worth mentioning, that the “better” fitness does not necessary have to be a higher value. In fact this rather depends on the range of values. In case of symbolic regression, where we measure the error, then a lower value represent better fitness and the value 0 represents a perfect fitness. On the other hand, if the fitness function is for instance a score in a video game, then a higher value would be a better fitness and there might not be any best (maximum value) fitness.

3.2.4 Crossover

Crossover is a genetic operation, which allows the creation of a new individual into the population, and thus, new parts of the search space to be tested [35]. J. Koza described in [35] crossover as an operation, where two phenotypes picked based on their fitness create two new individuals, which both contain some genetic material from their parents. As was already mention in the section about genetic programming, this operation can differ, but still accomplishes the task of creating a new individual to explore the search space. In some cases it might be, for the reasons of computation time or genotype representation, more beneficial to create only one offspring. In some cases it might be even possible to modify one of the parents instead of creating a new copy, modifying it and then not putting the parent into next population, but this is rather an implementation detail and the new individual would be the same as if a copy was done. Figure 3.5 displays example of crossover done over a genotype represented by an undirected graph. In this example, only one offspring is created, but it would be very easy to create the second one by selecting the not selected nodes and connecting those.

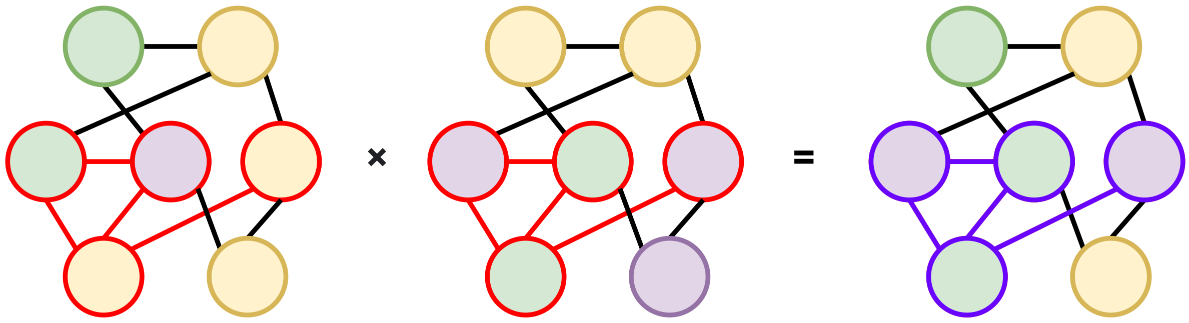


Figure 3.5: Example of a crossover operation on two genotypes represented by an undirected graphs (for the purpose of graph coloring). In this case a random subgraph is selected from the second genotype (red outlined nodes) and an inverse of this is taken from the first genotype. These two subrgaphs are then connected together to create a new genotype.

In genetic programming, in some cases such as Cartesian genetic programming, it is possible to not have crossovers and rely only on mutations [45]. The main thing, which crossovers bring into the evolution is a creation and modification the the genotype structure. For example if some process was working with bit strings as genotype representation, then the structure is always the same (a string of bits) only its length might change, which in

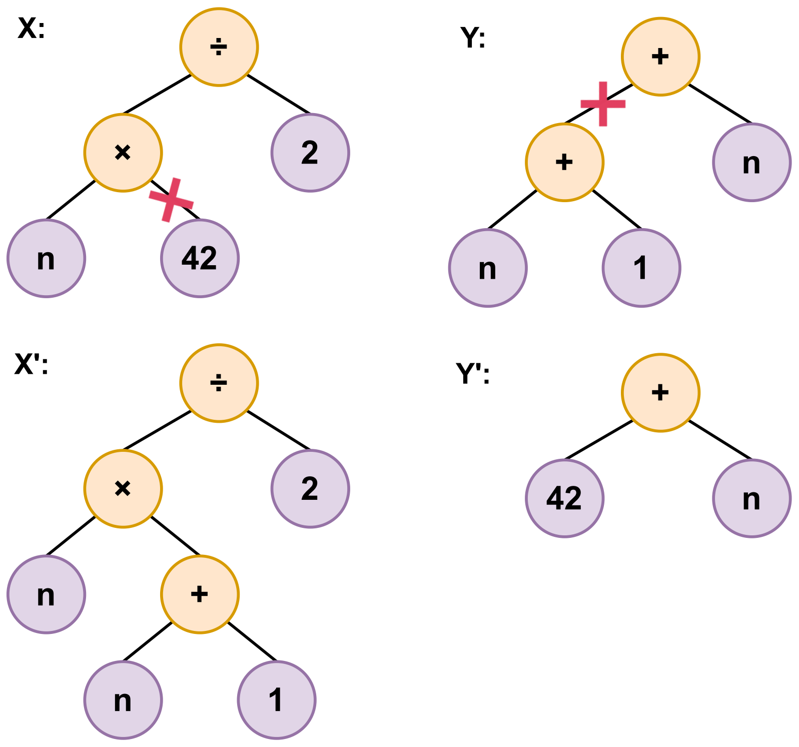


Figure 3.4: Example of a crossover operation on two genotypes represented by an abstract syntax tree. In this case a random subtree is selected for both parent genotypes X and Y and these subtrees are swapped creating two new genotypes X' and Y'.

some cases might not be desirable. Another example could be the undirected graph from Figure 3.5. This graph is the exact representation of the graph, which is being colored by genetic programming and therefore it cannot change its structure. On the other hand, where for example a tree is being evolved (such as in symbolic regression), there it might and probably is desirable to change the structure, which just ordinary mutation cannot do.

3.2.5 Mutation

Mutation is an asexual operation, meaning that it operates on only one individual. It works by altering the value or values in the genotype's structure [35]. In the Cartesian genetic programming over a list of integers it could for example randomly change one integer or, as it is depicted in the Figure 3.7, it could choose a random node in the genotype's graph and change its color to another random color.

Unlike crossover mutation might be necessary in some cases for the population to be converging. This is because the mutation can bring back some VALUE that was lost during evolution. This can be very well described on an example. Let's suppose we are trying to k-color an undirected graph, just as is depicted in Figures 3.5 and 3.7. The initial population gets initialized and even here it might be possible, that some color is missing and using crossovers it will never be possible to find correct coloring, since there is not enough colors. But let us say that the initialization makes sure to include all colors needed by coloring each node in different color. In this case it might be possible, that the crossover gets rid of one or multiple colors by creating new individuals that have the some colors missing (which is unavoidable to achieve minimal graph coloring or k-coloring with $k < |N|$). After a few crossovers it might happen that none of the phenotypes in the population contains some color and consequently it is not possible to correctly k-color the graph in this evolution. For this exact reason it is good to have a mutation operation in place. Mutation usually has quite a low probability chance of it happening [35] and the chance that it generates the missing color also lowers the overall chance of value reintroduction, but this chance is there and makes sure that the population could converge in time.

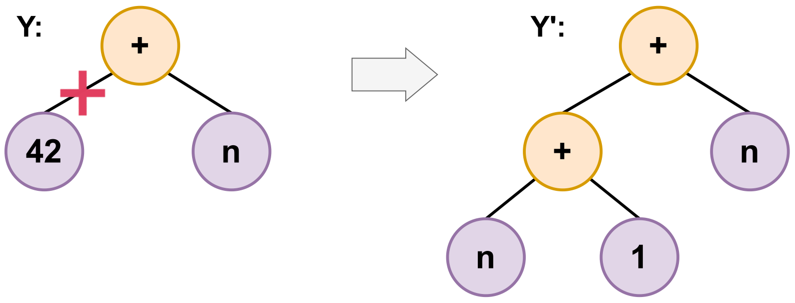


Figure 3.6: Example of a mutation of a genotype represented by an abstract syntax tree. Random subtree is selected and is replaced by new randomly generated tree.

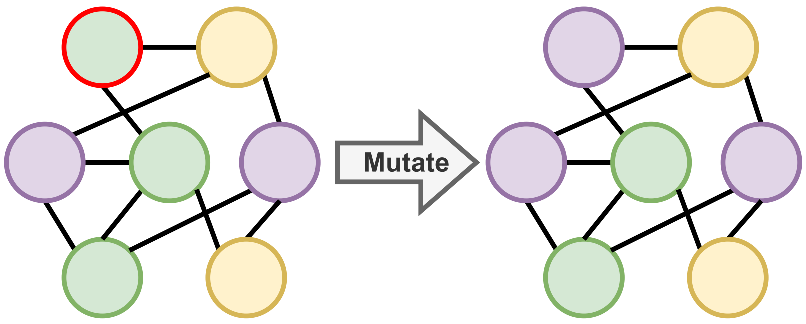


Figure 3.7: Example of a mutation of a genotype represented by an undirected graph (for the purpose of graph coloring). Random node is selected and its color is changed to a random one.

It is also possible to allow mutation do other operations, than just randomly changing an attribute of some chromosome. In case of working with tree genotypes, mutation might often do the operation of removing a selected node and all its child nodes and replacing this with newly generated subtree [35]. It would be also viable for the mutation to only remove the node and reconnect its child nodes to its parent. The behavior should be adjusted to the problem and what brings better results.

3.2.6 Selection

Selection is a process of choosing, which phenotypes will participate in creation of new phenotypes. After the process of crossover and possibly mutation the new phenotypes are without any alteration placed from the old population into the new one and used in next evolution generation [35]. Selection is often based on fitness. One such popular selection method was described by J. Holland in [28]. His selection uses the following equation for the chance $p(s_i)$ that the individual s_i will be chosen into the new population, where $f(s_i(t))$ is the fitness (which should be normalized) of s_i at time t and M is the size of the population:

$$p(s_i) = \frac{f(s_i(t))}{\sum\limits_{j=1}^{M} f(s_j(t))} \tag{3.1}$$

Other alternative selections are "rank selection" and "tournament selection". In rank selection, the selection is based on the rank of the fitness values in the population [23] [35]. In the tournament selection a subset of the population (usually 2 phenotypes) is selected and these phenotypes "battle" with each other, meaning that the one with the best fitness is chosen into the new population [23].

Note that during the selection parents are also part of the selection process and often one phenotype gets to be in multiple populations [35]. This is even more so, if elitism is used. In case of elitism, the phenotype with the best fitness is unaltered and always placed in the new population. This approach makes sure, that the best fitness never worsens [18].

3.3 Symbolic regression

Symbolic regression (*SR*) is one of the examples of real world use for genetic programming. Symbolic regression can be used for many tasks such as econometric modeling and forecasting, empirical discovery of scientific laws, solution of equations yielding a function in symbolic form, programmatic image compression and many others [35]. Symbolic regression searches the space of mathematical expressions trying to find the best model for given data set. It is a set of processes for approximating the relationships between input and output pairs performed by methods, which minimize various error metrics [30] and J. Koza even called symbolic regression an "*error-driven evolution*" [35]. There is not only one, but multiple approaches for symbolic regression [4] [13], but the general principle is the same and it is in the process of genetic programming.

Symbolic regression uses already mentioned phenotype representation as a tree. In this specific case, since SR works with expression, it is an abstract syntax tree (AST). In this representation internal (non-leaf) nodes hold the operation (function, operator...) and leaves hold constants or variables [61] (example AST can be seen in Figure 3.4).

One of the problems with finding an exact expression for given problem using not only symbolic regression, but any approach, is ambiguity. Given a data set it might be possible to find multiple solutions for given inputs and outputs. This can be easily showcased on a very simple example. The task is to find an expression for a given data set:

x	**y**
0	0
4	64
17.4	5268.024
42	74088

Table 3.1: Example data set for symbolic regression.

In almost no time, symbolic regression might find a quite simple solution $y = x^3$, but when run once more the result might this time differ and the found solution could be $y = \sqrt{x^6}$, but these two equations are not the same. If $x = -3$ the first solution outputs $y = -27$, whereas the second solution outputs imaginary result $y = 27i$ or depending on the language might result in an exception. The reason can be easily seen in Figure 3.8. As the graph shows, the two equations are identical on the interval of $\langle 0, \infty)$ and because the given data set covers only this range it is impossible to decide, which was the correct original equation, that produces these outputs.

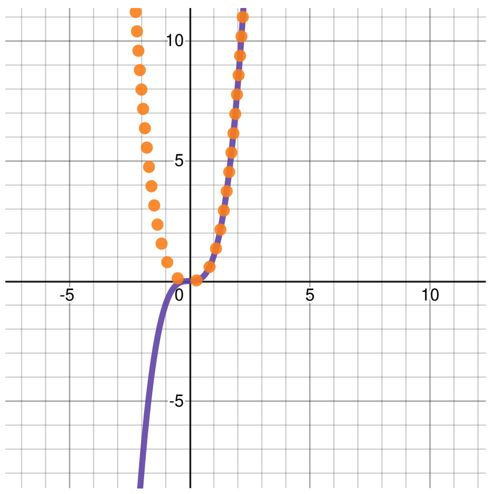


Figure 3.8: Plot of function x^3 (solid line) and $\sqrt{x^6}$ (dotted line).

The problem gets even worse with non-continuous or discrete functions. What if the function, that generated given outputs is a discrete function defined only in the inputs given? Another problem might be function with some constant since the evolution would have to have means of generating a correct constant, but a random generation could take immense time and what if the constant won't even fit in the datatype for a given computational platform? For example if symbolic regression was to find the equation for the Planck-Einstein relation $E = nhv$ [46] it would need to have the means of producing Planck constant $h = 6.62606957 \times 10^{-34}$ [58]. An argument could be given, that the symbolic regression could contain this and other constants that could be placed at the time of generation, which is a valid argument, but this approach works only if it is certain, that other constants won't be present, such as ones, which might be completely random. The same could be said for special functions, such as trigonometric functions or other that might often appear. But in general it is impossible to cover all possible constants and functions for all possible expressions, since there is infinitely many of them.

But more than often symbolic regression is used for the means of predicting the shape of the original function for the real world uses mentioned in the beginning of this section. For this task this approach is well suited and problems described above do not really concern this, since it is expected for the generated function to not be exact, but rather be as similar to the original one as possible.

Generalization also plays an important role for the found expression. The expression can be then tested on a testing data set, so that the expression is not only valid for the data set it was trained on.

Chapter 4

Ebe – Edit by Example

Ebe, which is an abbreviation for *Edit by example*, is a program, that uses genetic programming to edit files based on example patterns defined by the user.

Ebe was made and designed as an alternative approach for editing files, which could be used even by people with no programming skills at all. It should also help save time to those, who know programming and would spend some non-negligible time writing a code to do some file editing. Ebe is quite unique to the other methods for file editing in its use of genetic programming and the input required from the user.

For the user to do their file editing, all they have to do is provide a part (in many cases just one line is sufficient) of the input file and also this part edited by hand. Ebe, more specifically Eben (Ebe engine), then generates Ebel code to find a fitting transformation of the input file into the output file, which can be then used to edit the whole input file or even multiple files with similar structure.

Ebe can be thought of as a compiler, called Ebec (Ebe compiler), and interpreter, called Ebei (Ebe interpreter). This whole compilation and interpretation process can be for instance compared to a Java programming language compiler *javac*. This compiler takes Java source code and compiles it into a Java bytecode, where the bytecode is a lower abstraction, almost instruction-like level language, which can be then interpreted by the Java Virtual Machine [41]. Ebe takes a very similar approach, but instead of accepting some input language, it takes in two files and tries to "compile" (evolve using genetic programming) a code, which transforms the first input file into the second one. The compiled code is very much similar to Java's bytecode and is written in Ebel programming language, which is the language, that the Ebei (Ebe interpreter) understands. Such compiled code can be then interpreted using again Ebe, but this time taking multiple files with same structure as the input file provided during the compilation. The interpretation transforms all input files into new output, with structure similar to the one provided by the user during compilation (example output file). The whole compilation and interpretation process can be seen in Figure 4.1.

Cases, where Ebe should be the most fitting is when one is editing either a large file or multiple files with defined structure and repetition of this structure in those files. Example of such files can be, for instance, comma separated values (*.csv* files), some configuration files (e.g., *.ini* files) or some files defining a structure (e.g., *.gtf*, *.fastq* or *.rdf* files).

On the other hand, Ebe can not be efficiently used for some files, where there is no set structure or periodicity and where is no pattern to the editing done. Although even in some of these cases Ebe might be able to find the correct Ebel to do the editing and overall it is dependant on the specific case and examples provided.

For value transformations, Ebe can work with user defined expressions, since exact expression generation would be otherwise very difficult if not impossible (see Section 3.3). So the user can override the compiler editing decision and force some specific operation (expression) without rewriting the Ebel code by hand.

It is also worth mentioning that even though Ebe might be able to find a fitting program for some quite difficult case, this might take a long time for the evolution, which might not be for many users acceptable. But overall Ebe focuses on short execution time and offers a lot of options for the user the set up the compilation, so that is suits his or hers needs. On top of these user options, Ebe tries to be as user-friendly as possible and accounts for users that have little to no experience with compilers. Because of this the advanced user options for compilations are rather meant for more skilled users and by default the compiler does all the configuration.

Since Ebe is also an interpreter of Ebel language and this language can be also written quite easily by hand, it is possible for users to do file editing by writing out the Ebel code themselves. Although this approach somewhat loses the purpose of avoiding having to write a code to edit a file, it might be useful in cases, where the compilation wouldn't be able to find a fitting program or in cases where editing the expected output file would take as long as writing the Ebel code itself. The advantage to using in such cases Ebel instead of some other programming language is that Ebel was designed exactly for file editing, and thus, it is a fitting choice that might result in lot shorter source code, than in case of some general purpose language.

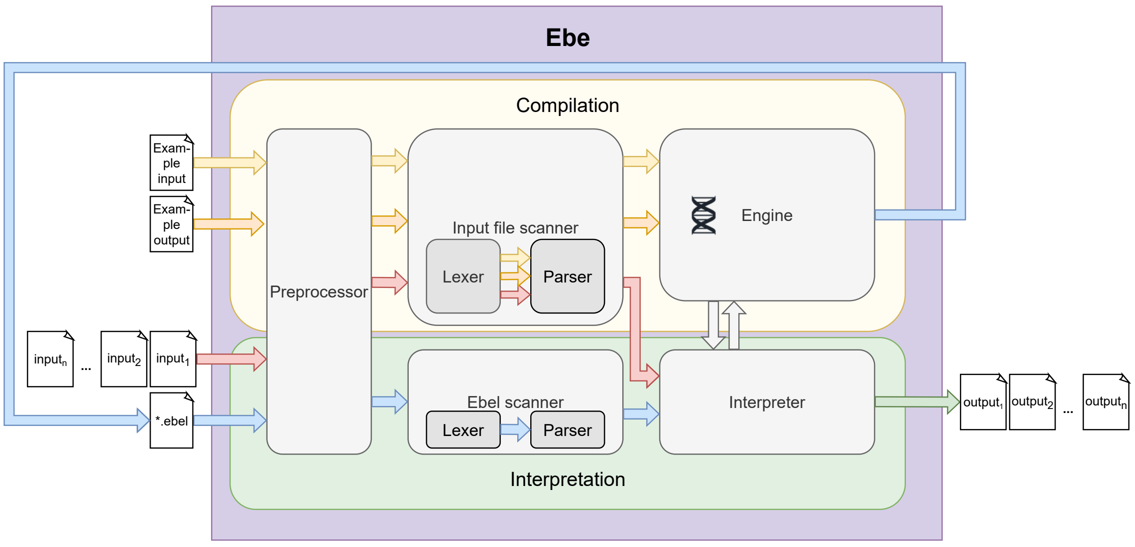


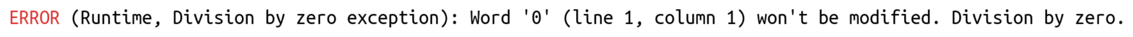
Figure 4.1: Diagram of the Ebe workflow.

4.1 Ebel – Ebe Language

Ebel, which stands for *Ebe language*, is an imperative, case insensitive, dynamically typed, programming language designed for file editing. This language was designed as a lower level target language and is in such a way used by Ebe, more specifically Ebec (Ebe compiler). It is interpreted over a file, where the Ebel code can be thought of as a pipeline of instructions through which the file objects (called *words*) go and get modified by.

Ebel is composed of multiple sections called *passes*. Such pass defines in which way the input file is read (passes are more in detail described in Section 4.1.2). These passes are then composed of instructions (that are described in Section 4.1.4), which work with the pass object (word or a line).

Since Ebel programs are generated by genetic programming, it might be used by non-programmers and there are no constraints on the input format. Ebe's philosophy is to not cause exceptions and errors as long as it is not necessary. Meaning that, when an incorrect instruction or input is encountered, rather than exiting the execution with error, only a warning is printed and the instruction or input is ignored (see Figure 4.2). This approach might be profitable also to the evolution since it might use this behavior to its advantage. This approach also allows for minor differences in the structure of the interpreted files.



```
ERROR (Runtime, Division by zero exception): Word '0' (line 1, column 1) won't be modified. Division by zero.
```

Figure 4.2: Screenshot of Ebe's output, showcasing Ebe's philosophy when encountering an error. Since input led to a division by zero exception, this expression was skipped and no changes were done to this input and a warning was emitted.

```
PASS Words
  NOP
  DEL
  DEL
  PASS number Expression
    SUB $1, $0, 32
    MUL $2, $1, 5
    DIV $0, $2, 9
    RETURN NOP
  PASS derived Expression
    RETURN DEL
PASS Lines
  SWAP 1
  LOOP
```

Listing 4.1: Example of Ebel code.

The Ebel code in Listing 4.1 will do the following:

1. `PASS Words` - file will be interpreted word by word and for each line:
   - 1.1. `NOP` - 1st object will be left as is,
   - 1.2. `DEL` - 2nd object will be deleted,
   - 1.3. `DEL` - 3rd object will be deleted,
   - 1.4. `PASS number Expression` - 4th object, if it is a number will be:
     - 1.4.1. `SUB $1,$0,32` - subtract 32 from its value,
     - 1.4.2. `MUL $2,$1,5` - multiply the new result by 5,

        1.4.3. `DIV $0,$2,9` - divide the result by 9 and save it as the new value for the object,
        1.4.4. `RETURN NOP` - end expression and do not modify the new result.
    1.5. `PASS derived Expression` - if 4th object was not a number, then use the following without regarding its type:
        1.5.1. `RETURN DEL` - delete the object.

2. `PASS Lines` - file will be interpreted line by line and for each line:

    2.1. `SWAP 1` - swap current line with the following one,
    2.2. `LOOP` - repeat until all lines were processed.

4.1.1 Ebel syntax

One of the reasons for Ebel to be syntactically very simple language, similar to a bytecode, is to simplify and speedup the syntactic and semantic analysis of it (the results of this can be seen in Figure 4.4). Ebel's grammar in extended Backus-Naur form can be seen in Listing 4.2.

```
program     : EOF
            | '\n'
            | pragma '\n' code
            | code EOF;

code        : instruction
            | pass
            | code '\n' (instruction|expr_inst|pass)?;

pragma      : '@pragma' PRAGMA
            | pragma '\n' '@pragma' PRAGMA;

instruction : 'concat' INT
            | 'del'
            | 'loop'
            | 'nop'
            | 'swap' INT
            | 'return' instruction;

expr_inst   : 'add'  VAR ',' (VAR|INT|FLOAT) ',' (VAR|INT|FLOAT)
            | 'sub'  VAR ',' (VAR|INT|FLOAT) ',' (VAR|INT|FLOAT)
            | 'mul'  VAR ',' (VAR|INT|FLOAT) ',' (VAR|INT|FLOAT)
            | 'div'  VAR ',' (VAR|INT|FLOAT) ',' (VAR|INT|FLOAT)
            | 'mod'  VAR ',' (VAR|INT|FLOAT) ',' (VAR|INT|FLOAT)
            | 'pow'  VAR ',' (VAR|INT|FLOAT) ',' (VAR|INT|FLOAT)
            | 'move' VAR ',' (VAR|INT|FLOAT) ',' (VAR|INT|FLOAT);

pass        : 'pass' type 'expression[s]?'
            | 'pass' expr_type;

```

```
VAR          : '$[0-9]*';
INT          : '[-]?[0-9]+';
FLOAT        : '([-]?[0-9]+\.[0-9]+[eE][+-]?[0-9]+ | [-]?[0-9]+\.[0-9]+)';
PRAGMA       : 'sym_table_size' [1-9][0-9]*
             | 'requires' '( [0-9]+\.?
                            | [0-9]+\.[0-9]+\.?
                            | [0-9]+\.[0-9]+\.[0-9]+ )'

expr_type    : 'expression[s]?'
             | 'word[s]?'
             | 'line[s]?'
             | 'document[s]?'
             | '".*"';

type         : 'text'
             | 'number'
             | 'float'
             | 'delimiter'
             | 'symbol'
             | 'empty'
             | 'derived';
```
Listing 4.2: EBNF grammar for ebel.

4.1.2 Ebel passes

Pass is the main building block of Ebel code, it can be thought of as a block of code, that does not interact with other passes. Each pass consists of instructions (which are described in Section 4.1.4) and each pass has a type, which defines in what way the input file will be read into it. Pass type can be one of the following:

- **Words** – Reads input word (lexeme) by word. Definition of what is a "word" depends on the input file type and compile time definitions.
- **Lines** – Reads input line by line. Meaning that the object lines pass does operations over is a set of words, where the end of such an object is a new line character.
- **Documents** – Reserved, but not utilized since no good use was yet found for this pass.
- **Expression** – Reads only one specific word from a word pass and does operations with this word. This pass is more often called a "subpass", since expression pass can appear only inside of a words pass. This subpass is needed to realize user defined expressions (see Section 4.1.3).

Words pass and lines pass always read the input from the start and matches each non-control instruction to an exactly one object (word or line respectively). If there are no instructions of input objects to match to the pass code the execution is ended. Only case, where this is not true is, when there are no more instructions and a `LOOP` instruction was used (this is described in Section 4.1.4). Expression subpasses are described in Section 4.1.3.

4.1.3 Expression passes and user defined expressions

Expression pass, or rather subpass, can appear only inside of a words pass (called it's *parent pass*) and its input is a word from the parent words pass at the exact position it is in the pass. It can also contain a word type, which when matched with the input words allows execution of this subpass. Expression type can also not be defined or defined as `DERIVED`, which in both cases allows any word type to be read by it, but this is not recommended since it might cause runtime errors. The expression subpass is ended when `RETURN` instruction is met. `RETURN` instruction might contain as its argument words pass instruction that will be executed on the newly transformed word and after this the input processing of the parent words pass resumes at the position after the expression (input is not read from the beginning as in case of other passes).

```
PASS Words
  PASS NUMBER Expression
    # Number expression instructions
    RETURN SWAP 42
```

Listing 4.3: Example of Ebel expression pass.

The need for such pass is for when a specific word needs to be changed, which in case of, let's say, text file and a `NUMBER` type would be, for example, adding some constant to it or modifying its value in any other way. Such transformations are most likely to be done by the user for a specific value using user defined expressions.

User defined expressions can appear in the output file at position of the transformed input word and to define it, this expression can only appear in between the expression start character sequence (`{!`) and expression end character sequence (`!}`). User defined expressions cannot contain assignment operator nor user defined variables and it is a continuous calculation, which is then assigned to the output word. User might use only one variable, `$`, which refers to the read input. Expression can contain following arithmetic operators:

Precedence	Associativity	Symbol	Description
4	Left to right	()	Parentheses
3	Right to left	^	Exponentiation
2	Left to right	*	Multiplication
		/	Division
		%	Modulo
1	Left to right	+	Addition
		-	Subtraction

Table 4.1: Ebe expression operator precedence and associativity.

User defined expressions are meant rather for more skilled users and because of Ebe's philosophy (mentioned in the introduction of this chapter), to process these expressions, the user has to specify the usage of expressions at compile time using argument option `-expr` or `--expression`. This is partially because using expressions may cause fatal errors to appear (such as undefined operator or keyword) and there would be problems with parsing escape sequence for expression begin (`{!`), which might occasionally appear in the user input file and would require even unskilled users to escape these sequences, which would only cause problems to such users.

```
{! $ ^ ($ + 42) * 3 + ($ - 1) !}
```

Listing 4.4: Example of text file user defined expression for type `NUMBER`.

On the level of Ebel code, as was already mentioned before, a user defined expression will be parsed into an expression pass using operator equivalent instructions (described in Section 4.1.4), where it is now allowed and even needed for more variables to appear. Since these variables are rather meant to be generated by the compiler, these variables have numeric names with `$` symbol prefix, where the variable number is an index to the symbol table for the current expression pass. This allows for the symbol table to be implemented as an array where each index corresponds to a variable with the same name, and so, all the variables can be accessed in the symbol table in constant time $O(1)$. The zeroth index (`$0`) is reserved for the input, which was denoted in the user defined expression as `$` and in Ebel both notations are accepted (meaning that `$0` is equivalent to `$` and vice versa). Ebel expression instructions resemble assembly language instructions and are often using three address code. Meaning that most binary operations have 3 arguments – first is the destination, which is always a variable, second is the first operation source and third is the second operation source. For unary operations it would take only 2 arguments – the destination and the source and for any other arity operation this would be analogical. Example of generated Ebel code for an expression can be seen in Listing 4.5. Notice that variables differ from constants by having `$` symbol prefix. Also note the last `ADD` instruction, which has as its destination `$0` variable, meaning that its output will replace the input word and will appear as the output for this word. In case of generated expressions the compiler uses the last operation as an assignment, to save the amount of instructions used. Although it is possible to use `MOVE` instruction and will be also used by the compiler in case of constant expression (e.g., `{! 42 !}`, which would compile into `MOVE $0, 42`).

```
PASS Words
  PASS number Expression
    ADD $1, $0, 42
    POW $2, $0, $1
    MUL $3, $2, 3
    SUB $4, $0, 1
    ADD $0, $4, $3
    RETURN NOP
  DEL
  NOP
  NOP
  SWAP 42
PASS Lines
  CONCAT 2
  NOP
  LOOP
PASS Words
  NOP
  DEL
  LOOP
```

Listing 4.5: Ebel expression code generated for user defined expression from Listing 4.4.

Ebel is also dynamically typed language, meaning that the variable type might change during code execution and so might the output word type. The amount of variables is bound to fixed number, although this is just an implementation choice, so that the symbol table does not have to change its size dynamically and also there is not much need to have large amount of variables for one expression. But if larger symbol table is needed the `sym_table_size` pragma can be used.

Expression pass also allows for the type to be a string – a so called "match expression". Such expression then works as an if statement for specific text value. If the processed word matches the one in the match expression, then the expression processes it with its actual type. Example of match expression can be seen in Listing 4.6.

```
Pass Words
  Pass "42" Expression
    MUL $0, 3, $0
    RETURN NOP
  Pass "null" Expression
    MOVE $0, 0
    RETURN NOP
  Pass Derived Expression
    RETURN NOP
  LOOP
```

Listing 4.6: Example of match expression, which multiplies all occurrences of the number 42 by 3 and replaces all occurrences of the string *null* with the number 0.

4.1.4 Ebel instructions

Instructions (all listed in Table 4.2 and 4.3) perform the input transformation and based on their proprieties these instructions can be split into:

- **Control instructions** – these instructions don't modify the input object, but change the interpretation environment. Such example can be the `PASS` instruction, which starts a new pass, thus changing the parsing of the input.
- **Process instructions** – these instructions edit the parsed object, such as deleting it (`DEL`), moving it around (`SWAP`) or not modifying it at all (`NOP`).
- **Expression instructions** – these instructions work with word object and can appear only in Expression pass. Table 4.3 describes these instructions.

Additionally other attributes can be, in which pass an instruction can be executed, where expression instructions are special case of this. Expression instructions also have a set of word types they accept, if not met, then expression execution is aborted and warning emitted.

Instruction	Definition	Arguments	Pass	Control
`CONCAT #ofs`	Concatenates current line with a line at specified offset	Positive integer offset	Lines	No
`DEL`	Deletes object		Words, Lines	No
`LOOP`	Loops instructions above this until the end of input is reached. Instructions after this will be executed and after there are no more instruction then the loop will be executed (from the start). `LOOP` nesting is not supported; last `LOOP` will be used.		Words, Lines	Yes
`NOP`	Object is not modified		Words, Lines	No
`PASS name`	Sets input parsing	words/lines/ documents/expression	Expression in Words	Yes
`SWAP #ofs`	Swaps current object with an object at specified offset	Positive integer offset	Words, Lines	No

Table 4.2: Ebel instructions.

Instruction	Definition	src1	src2	Note
`ADD dst, src1, src2`	Addition	number/float	number/float	
`DIV dst, src1, src2`	Division	number/float	number/float	If src2 is 0, exception is raised
`MOD dst, src1, src2`	Modulo	number/float	number/float	If src2 is 0, exception is raised
`MOVE dst, src1, src2`	Assignment	derived		
`MUL dst, src1, src2`	Multiplication	number/float	number/float	
`POW dst, src1, src2`	Exponentiation	number/float	number/float	
`SUB dst, src1, src2`	Subtraction	number/float	number/float	

Table 4.3: Ebel expression instructions.

The destination argument in expression instructions is always the first argument and can be only a variable. Source arguments then can have types based on the instruction (described in Table 4.3), but if not stated otherwise, source arguments have to have a matching type.

4.1.5 Ebel pragmas

Ebel also has a support for special control instructions called pragmas. Pragmas are meant for the interpreter and can change internal settings of it, such as the size of symbol table.

In the current state there is not much need for many pragmas, but addition of a new parser for a new file type could perhaps change this. Currently Ebel recognizes the following pragmas:

- `sym_table_size` *<positive integer>* – this pragma sets the size of symbol table to the given argument. Choosing a lower value will save a few bytes of memory, but might cause a run-time exception.

- `requires` *<version string>* – this pragma causes the interpreter to check the current version of Ebe with the argument of the pragma and if the current version is lower than the argument, then an error is raised and interpretation is aborted. This pragma should be used for code that contains some newer constructs, which would not work with older interpreter versions. The main purpose of this instruction is to help with error identification if such situation were to happen, since the error message would indicate, that a newer version is required and that the code is valid, otherwise the user could be searching for the error in the code. Usage of this pragma should be mostly for scripts that are transferred between systems, which could contain different Ebe versions.

Each pragma starts with the string `@pragma` followed by the pragma's name and then its arguments if needed. Pragmas can appear only at the beginning of the code and have to be on separate lines (for detailed description see Ebel's grammar in Listing 4.1.1 or example 4.7).

```
@pragma requires 0.3
@pragma sym_table_size 5
PASS Words
  PASS float Expression
    ADD $1, $0, 42.0
    POW $2, $0, $1
    MUL $3, $2, 0.2e-3
    SUB $4, $0, -1.4
    ADD $0, $4, $3
    RETURN NOP
  NOP
```

Listing 4.7: Ebel code that uses pragmas.

4.2 Eben – Ebe engine

Ebe engine is a unit that is a part of the compilation process. The main purpose of this unit is to find or rather generate, the Ebel program to transform the user input into the output. Engines are what drives the compilation, finding the best fitting program for set inputs.

Several engines are currently implemented in Ebe. Most of them employ genetic programming, but also a pure random search engine MiRANDa is also available (see Section 5.3.3).

Specific engine can be selected by the user during compilation (using `--engine` option with engine name following it) and additional engine parameters can be set by the user, such as:

- the number of evolutionary runs,
- the number of generations (iterations) in one run,
- size of the population,
- fitness function used,
- evolution timeout (time or minimum phenotype precision).

But generally it is Ebe's job to select the engine that will be used and set its parameters.

More information regarding Eben implementation can be found in Section 5.3.

4.3 Ebec – Ebe compiler

Ebe compiler has multiple tasks to do and is quite different from conventional compilers. It has to load and parse all the input files and then guide engine in Ebel generation.

Ebe compiler differs quite a lot from other compilers mostly by the fact that it has to accept all user inputs, since any input file is always correct. Only constrained parts are user defined expressions, which have to adhere to correct syntax and semantics (see Section 4.1.3). On the other hand, unlike ordinary compiler, Ebec might for correct input not always generate correct output. This is of course caused by the engine not being able to find fitting program using genetic programming (or another heuristic approach). Ebec still outputs the program even when a 100 % fitting program was not found. This is done since there might be cases, where even not a fully correct transformation program might be useful or when the user specified an incorrect expected output. Ebe will notify the user that the compilation was not 100 % successful and the user might then decide to change the compilation parameters and rerun the compilation.

Another difference between Ebec and ordinary compilers is that Ebec is not deterministic. Meaning that for the same input it might generate a different output. Hence, it is also a valid approach to redo any compilation even without changing any parameters to receive a different program, which is mostly wanted in case of compilation not finding a 100 % fitting program.

4.3.1 Lexical, syntactical and semantic analysis of input files

Since all user inputs should be accepted by the lexer and parser for any file format, it has to work with this assumption and in the very least add a special type for unknown lexemes. The reason to allow for formats, which are not fully correct based on their specification is to allow the user to use the tool to fix these incorrect files and also to allow the user to use the parser that is the closest to his or her format to parse the input file and get the best results.

When it comes to semantic analysis it is not really present as the input will be modified and not interpreted. The most important part of the analysis is the lexical analysis and in some formats it might be the only needed unit. The part of Ebe, which adds additional syntax and semantics to the input files are user defined expressions. But not every format has to or even can support these.

4.3.2 Ebel code optimizations

Eben uses genetic programming to create candidate programs to achieve desired file transformation, but even if a 100 % fitting program is found it does not necessarily mean, that the code will be written in the best way possible and it might (and most certainly will) contain redundant instructions or dead code.

One of special cases is a "tail" (uninterrupted sequence at the end of a pass) of `NOP` instructions or other empty instructions. Removing such tail or any instructions, in a way of keeping the same semantics of the code, might result in faster interpretation times. This can be seen in Figure 4.3, where two semantically same programs were interpreted over the same input (the gtf file described in Section 4.4), but one has `NOP` tail (of the length of 26 to match the gtf format) and the other does not have it.

The main reason for this result is that, when these instructions are omitted the interpreter can stop processing the input and even though the `NOP` instruction does not do any operations over the IR, the need to load additional objects slows down the interpretation and results in longer processing time.

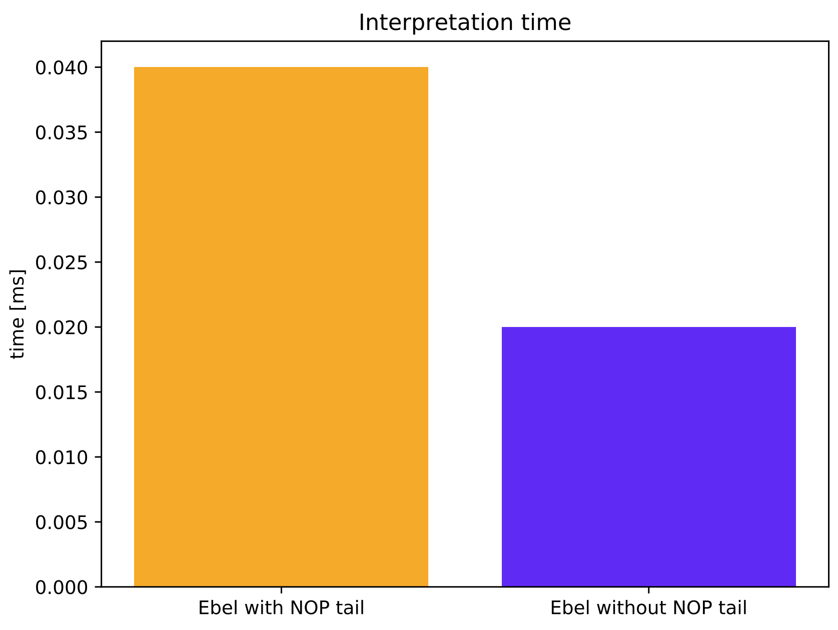


Figure 4.3: Graphs comparing interpretation speed of Ebei 0.2.4 on code with and without NOP tail. The values are median from 10 measured runs with the same input examples.

4.3.3 Fitness functions

To calculate how correct the output program is, Ebec uses fitness functions since the value for each candidate program is already computed during evolution in Eben. There are different fitness function approaches resulting in a different score for the same program, although it is always the same for the same function and input.

The simplest and very naive fitness function is just a *one-to-one* mapping, meaning that the expected output is compared word by word with the generated output at the same positions. If the words are the same a point is added to the total matched sum and the final score is calculated by dividing this sum by the total amount of words in the longer file. Fitness calculation ($f(x, y)$, where x and y are files, where each lexeme can be indexed. In Equation 4.1, a and b are these lexemes) for One-to-one mapping fitness function would be as follows:

$$eq(a, b) = \begin{cases} 1 & \text{for } a = b \\ 0 & \text{otherwise} \end{cases} \tag{4.1}$$

$$f(x, y) = \frac{\sum\limits_{i=1}^{\max(|x|,|y|)} eq(x_i, y_i)}{\max(|x|, |y|)} \tag{4.2}$$

This approach is of course, as already mentioned, very naive and having two identical inputs, where one would be missing the first word might result in 0 % match. The biggest advantage of this function is that its time complexity is linear $\Theta(n)$, where n is length of the shorter input, since unlike in Equation 4.2, in implementation once the shorter string is processed it is sure that all the remaining symbols in the longer string do not match. This function was mainly implemented for its simplicity.

Ebe also contains three other fitness functions that can be used and those are: Levenshtein distance, Jaro distance and Jaro-Winkler distance (all described in Section 2.4)

There is also a special case, which the fitness functions have to take care of, and that is an empty file and empty lines. Such cases have the possibility of causing division by zero in functions which are dividing by string's length. These cases can be easily handled by conditional check. Ebe also has a special type for an empty line in a text file.

4.4 Ebei – Ebe interpreter

Ebe interpreter takes an Ebel source code and one or more files as the input for the interpretation. Interpreter adheres to the Ebe principles mentioned in this chapter's introduction. It tries not to interrupt the interpretation because of simple problems, but rather the interpreter only warns the user about these problems. The main problem, which would cause the interpreter to abort the interpretation would be an incorrect Ebel code, but this should not happen if the user generates this code using Ebe compiler.

Since one of the goals for Ebe is to be fast, the same is also Ebei's goal. The biggest slowdown for interpreter is the need to parse every input file into its internal representation (IR) and even though this slows down the interpretation it is a necessary step to correctly transform the data. On the other hand in comparison to different interpreted languages that can be used for file editing such as Python 3, Ebei works with a lot simpler language, which allows for simpler parsing of this language and for less complicated and therefore lot faster execution. In the case of Python 3 the Python interpreter has to first parse the language into its bytecode and then do the interpretation [53]. Whereas Ebei works with Ebel code, which is already in a form of a bytecode (see Section 4.1.4) and can be right away interpreted. This difference can be seen in Figure 4.4, in which time and CPU usage were measured for the same input problem solved by Ebe and Python 3. For Ebe the code was generated by Ebe itself using user defined expressions (the Ebel code can be seen in Listing 4.8). For Python 3 the code the can be seen in Listing 4.9.

```
PASS Words
  NOP
  NOP
  NOP
  NOP
  NOP
  NOP
  PASS number Expression
    SUB $0, $0, 153350000
    RETURN  NOP
  NOP
  PASS number Expression
    SUB $0, $0, 153350000
    RETURN  NOP
```

Listing 4.8: Ebel code for .gtf file editing.

```python
import csv
import sys

if __name__ == '__main__':
    if len(sys.argv) < 2:
        exit(1)
    out_text = []
    with open(sys.argv[1], "r") as gtf_file:
        reader = csv.reader(gtf_file, delimiter='\t')
        for line in reader:
            line[3] = str(int(line[3]) - 153350000)
            line[4] = str(int(line[4]) - 153350000)
            out_text.append("\t".join(line))
    print("\n".join(out_text))
```

Listing 4.9: Python 3 code for .gtf file editing.

The problem solved by these scripts is taken from a real world situation, where a bigger .gtf (*Gene transfer format*) file was shorten to contain only part of its previous data. But because of this change the start and end coordinates (4th and 5th column) were incorrectly offset by 153350000 and therefore the file had to be modified to subtract this constant for each line (feature).

Data for Figure 4.4 were gained by executing each script 100 times on a single CPU core with maximum priority and measured by `time` utility on Linux. Figure 4.4 displays median calculated from all of these measurements.

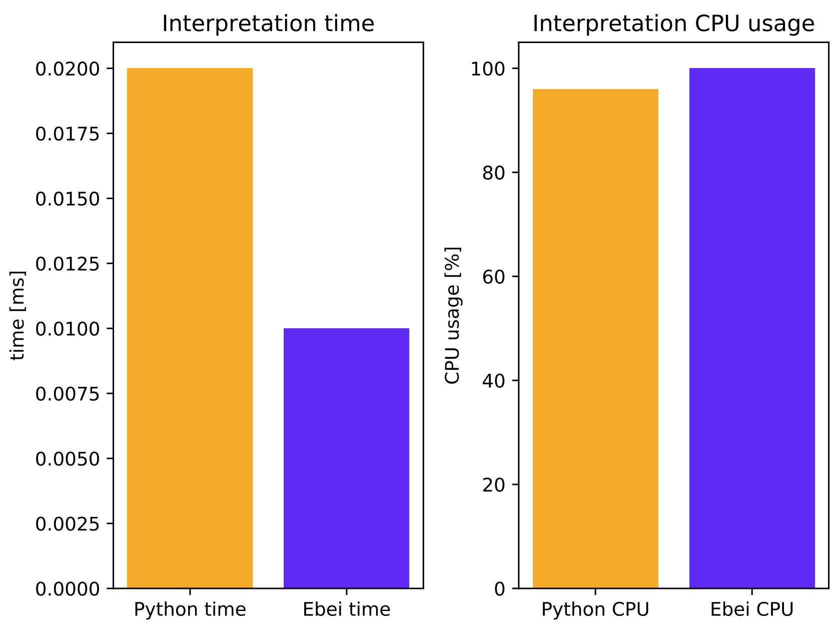


Figure 4.4: Graphs comparing speed and CPU usage of Ebei 0.2.4 and Python 3. Represented are median values from 10 measured runs.

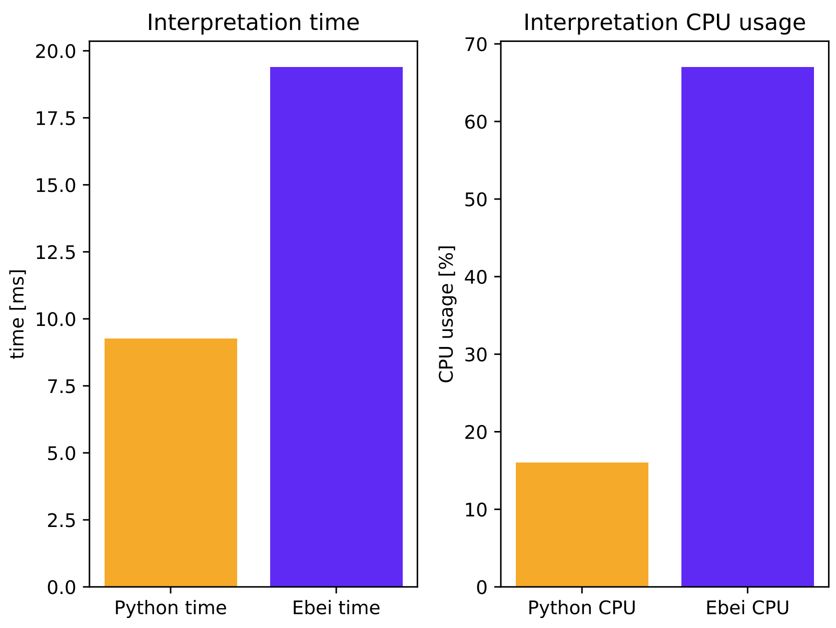


Figure 4.5: Graphs comparing speed and CPU usage of Ebei 0.2.4 and Python 3 using a 123000 lines long input. Represented are median values from 10 measured runs.

Interpreting these scripts over file with only about 400 lines (which is the case of Figure 4.4) results in the input file parsing having not as big of an impact on the interpretation time. But if a lot larger input file is used, then the parsing heavily impacts the performance of Ebei, which can be seen in Figure 4.5.

As already mentioned, the biggest problem with the speed loss in case of a very large file is that each lexeme has to be parsed, whereas in case of Python 3 code only tab symbols need to be found to split the input into columns and then only the start and end columns are parsed as integers.

The example given uses very simple editing, and so, for the very large files the overall speed of Ebei is worse than the speed of Python 3, but this is only in this case. Other cases might require the Python script to parse other lexemes as well, which will have a significant impact on the overall speed of the Python script.

4.4.1 Lexical, syntactical and semantic analysis of Ebel code

On a level of expressions (expression pass), Ebel is very closely tied to the input formats and their parsing. The types declared and parsed by different input formats become Ebel types for its expression instructions. On the other hand the general file editing done by other passes is not dependant on the input file format as long as its own lexer and parser can create valid internal representation. Meaning that any input file format can be edited by Ebe, if there is a lexer and a parser for it. For an input format to work with Ebel expressions it must have its types defined also in Ebel grammar and provide instruction semantics over this type to the interpreter. The other approach would be to use already existing types, but this might not always fit the new input format.

Even though Ebel was not designed as a language to be written by programmers, but rather generated by the compiler, it still offers some “syntactic sugar” for cases when a programmer would write short Ebel parts or edit generated code (perhaps to suite all cases not easily expressible in example input files). This use-case is also the reason why Ebel uses a text format and not binary. Here are some examples of Ebel’s “syntactic sugar”:

- Ebel is case insensitive,
- the return instruction can be omitted and interpreter will return `NOP`,
- the expression type can be omitted and interpreter will use `DERIVED` type,
- variable `$0` can be written as `$`. . . .

All of these features are not difficult nor time consuming for the interpreter to handle and provide more user-friendly environment for users. It is also worth mentioning that all of these features can even be handled during syntactic or semantic analysis, putting even less stress on the interpreter’s runtime duties.

Ebel also supports single line comments (denoted by symbol `#`) and, as already mentioned, somewhat allows even for incorrect semantics by not executing this code and warning the user.

Ebel instruction semantics are also quite simple mostly when it comes to non-expression instructions. These instructions work over internal representation, which is very much needed to modify a file in a fast way. More complexity comes with expression instructions, which work with specific types and have to do additional semantic checks. Such checks are to ensure, that all instruction types for the instruction are correct, to make sure that used variables are defined and that all of the values are in the domain of the instruction. Unfortunately most of these checks have to be done at a runtime, since all of these instructions work with user input.

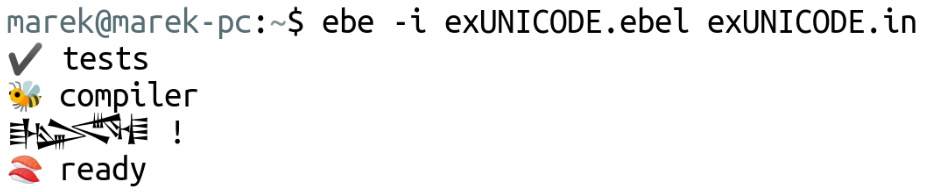
4.5 Ebe's text file parsing

Unlike Ebel file, the example input file and interpreted file can never have an incorrect format. In case of example output file, there might be an error in parsing, if incorrect user defined expression is used and parsed.

Each text file format differs, but this, in many cases, does not really affect the genetic programming solution finding. Ebe can work with plain text file as well as with CSV files or any other format. The tasks Ebe cannot solve are the ones it does not have instructions for, such as generating delimiters, copying words or transforming text. Another problem could be different formats for certain types, such as floating point values, where the comma separates the decimal part or not having floating points at all and splitting such number into 2 integers separated by a dot. These cases are still solvable by changing the text parser or adding a new one.

But the general text file parser contains word types that are quite popular in many formats, these are:

- `TEXT` – A string of alphanumeric symbols including Unicode symbols (see Figure 4.6).
- `NUMBER` – An integer defined as a string of numeric symbols.
- `FLOAT` – Floating point number, which can also be written in scientific notation (for more detailed description see Ebel grammar 4.2, which uses the same `FLOAT` format).
- `DELIMITER` – A symbol that is often used as a delimiter: space, tab, vertical tab, carriage return, form feed, comma, dot, colon, semicolon.
- `SYMBOL` – All other characters, which do not match the previous definitions.



```
marek@marek-pc:~$ ebe -i exUNICODE.ebel exUNICODE.in
✔ tests
🐝 compiler
[illegible] !
🍣 ready
```

Figure 4.6: Screenshot of Ebe interpretation showcasing that Ebe can also work with Unicode symbols.

Note that the `SYMBOL` type is used to match any input that was not accounted for. Meaning, that the lexeme parsing cannot fail.

There are also additional internal types to help with the evolutionary approach:

- `EMPTY` – Denotes an empty line.
- `EXPRESSION` – Denotes a user defined expression.

As can be seen, Ebe's default parser does not use many types, but this helps with making the default text file parser quite general and also allows users to parse a file "by hand", which might be useful when writing Ebel or trying to understand what some Ebel program exactly does.

4.6 Examples of Ebe workflow

Ebe contains simple to use text user interface, where many of the command line options and the program behavior tries to adhere to the ones in other popular compilers. Examples of these commands are:

- **`-h`**, **`--help`** – Prints Ebe usage and simple command line options description.
- **`--version`** – Prints compiler's version.
- **`-o`** – Sets the output path.

On top of having these options, that are familiar to those knowledgeable of compiler use, Ebe also contains alternative versions for most of the command line options, which are very verbose and should be easy to use by non-programmers. Examples of these commands are:

- **`-in`** – Alternative format is **`--example-input`**. Sets the example input file.
- **`-expr`** – Alternative format is **`--expressions`**. Parses user defined expressions.
- **`-e`** – Alternative format is **`--evolutions`**. Sets the amount of evolutions to be done, meaning how many times a new population should be created and evolved.

Ebe contains two main big modules – the compiler and the interpreter and therefore it makes sense to firstly do the compilation, which produces an Ebel program and then this program can be interpreted over a file. But quite often one will do both of these operations right after each other and that is why Ebe also contains "execute" option (invoked by `-x`), which will firstly compile the program and then interpret it over any input files. In this mode it is still possible to use any compilation specific or interpretation specific options as well as even output the Ebel program.

If only the compilation is started without specifying the output Ebel path (`-eo`), then the Ebel path will be the same as example input file's path, the name of the Ebel will be the same as the example input file's name, but the extension will be changed to `.ebel`. This fact will be conveyed to the user by information message after the compilation. The user is also notified about the reached Ebel precision (program's fitness score) and if the program is perfectly fitting (100 % precision) as well as the overall compile time.

If only an interpretation is executed or compilation with interpretation in one run (`-x`) and the output file (or folder in case of multiple input files) is not set, then the edited file content will be printed to the standard output. In case of multiple input files a line denoting each file's name is printed before the output for that file. In case of compilation with interpretation a line denoting the interpreted output is printed before it. In case someone wants to output the file to the standard output without the information messages (for example to use it as an input for other program) a `--no-info-print` command line option can be used. Additionally `--no-warn-print` and `--no-error-print` can be used so that the compiler does not emit any warnings or errors during interpretation and compilation, although these messages are printed to the standard error output and do not effect the standard output.

The following example showcases Ebe's use to edit a CSV file containing the following values in each column: last name, first name, year of birth, chess FIDE rating. Example of the first 3 lines of the file can be seen in the Listing 4.10.

```
Lagrave,Maxime,1990,2758
Navara,David,1985,2693
Carlsen,Magnus,1990,2864
```

Listing 4.10: First 3 lines of the file to be edited (`players.csv`).

The task is to edit this file so that the first column is the first name, followed by the last name, then players age (in the year 2022) and finally their rating. Since this task requires the same changes for each line the example input file can only contain copy of the first line (see Listing 4.11). It is worth noting that alternatively same type lexemes could be used, such as `a,b,1,2`, but using the actual line makes it easier to work with.

```
Lagrave,Maxime,1990,2758
```

Listing 4.11: Contents of the example input file (`ex.in`).

The example input file, can then be copied and edited by hand. But since the birth year has to be recalculated, it requires the use of user defined expression. In this case to convert year of birth into an age for the year 2022, the expression is quite simple, but the user has to know the starting end ending escape sequence (`{!` and `!}`) and also that the year value will be under the variable `$`. The contents of the example output file with correct expression can be seen in Listing 4.12.

```
Maxime,Lagrave,{! 2022 - $ !},2758
```

Listing 4.12: Contents of the example output file (`ex.out`).

With this step the examples are all setup and now it's all left to Ebe using either compilation followed by interpretation (Listing 4.13). But more convenient option using the `-x` option can be seen in the Listing 4.14. These two approaches are equivalent. It needs to be noted, that user defined expressions are not enabled by default (since it is a technique meant for more skilled users and this avoids possible warnings with files containing expression escape sequence). To enable parsing of user defined expression the `-expr` or `--expressions` option has to be used.

```
ebe -in ex.in -out ex.out -eo players.ebel --expressions
ebe -i players.ebel -o edited.csv players.csv
```

Listing 4.13: Ebe editing using compilation followed by interpretation. The Ebel file will be outputted into `players.ebel` and the edited file will be outputted into `edited.csv`.

```
ebe -x -in ex.in -out ex.out -eo players.ebel -o edited.csv --expressions
    players.csv
```

Listing 4.14: Ebe editing using compilation and interpretation in one run. The Ebel file will be outputted into `players.ebel` and the edited file will be outputted into `edited.csv`.

The generated Ebel file, `players.ebel`, will of course vary in each run, because of the genetic programming approach, but the beginning should be always the user defined expression followed by the `SWAP` instruction in a new pass, this might in some generated cases be followed by other passes, but these will have no impact on the input file as long as it has the same structure. Possible contents of the generated Ebel can be seen in Listing 4.14.

```
PASS Words
  NOP
  NOP
  NOP
  NOP
  PASS derived Expression
    SUB $0, 2022, $0
    RETURN  NOP
  NOP
  NOP
PASS Words
  SWAP 2
  NOP
  NOP
```

Listing 4.15: Ebel file `players.ebel` generated by Ebe from execution of command in Listing 4.14.

Finally the edited input file will be placed into `edited.csv`. The first 3 lines of this file can be seen in the Listing 4.16.

```
Maxime,Lagrave,32,2758
David,Navara,37,2693
Magnus,Carlsen,32,2864
```

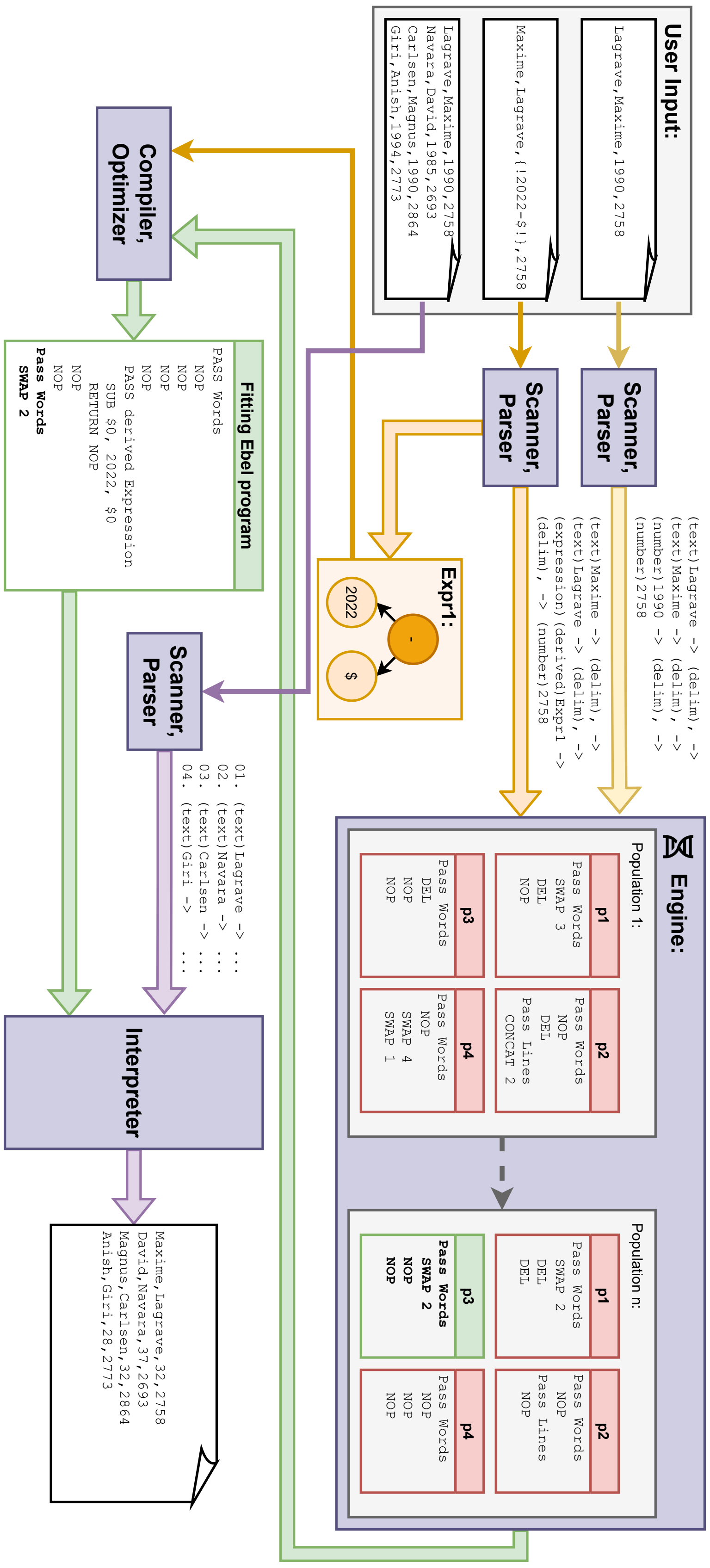
Listing 4.16: First 3 lines of the output file `edited.csv` generated by Ebe from execution of command in Listing 4.14.

This example with simplified description of what Ebe does internally can be seen in diagram form in Figure 4.7.

From such exhaustively described example it might seem like quite a complicated process, but is many cases the whole process consists of 3 simple steps:

1. Create the example input file from the first line (or multiple lines) of the file to be edited.
2. Copy the example input file and edit it by hand into desired form.
3. Execute Ebe passing it the example files created with the file to be edited.

Once the user is more acquainted with the tool, he or she might start setting up compilation parameters to achieve better or faster results or possibly even chain multiple Ebe executions to do more advanced edits or cut down on the compilation time.



```
Lagrave,Maxime,1990,2758
```

```
Maxime,Lagrave,{!2022-$!},2758
```

```
Lagrave,Maxime,1990,2758
Navara,David,1985,2693
Carlsen,Magnus,1990,2864
Giri,Anish,1994,2773
```

```
(text)Lagrave -> (delim), ->
(text)Maxime -> (delim), ->
(number)1990 -> (delim), ->
(number)2758
```

```
(text)Maxime -> (delim), ->
(text)Lagrave -> (delim), ->
(expression)(derived)Expr1 ->
(delim), -> (number)2758
```

Population 1:

```
p1              p2
Pass Words      Pass Words
SWAP 3          NOP
DEL             DEL
NOP             Pass Lines
                CONCAT 2

p3              p4
Pass Words      Pass Words
DEL             NOP
NOP             SWAP 4
NOP             SWAP 1
```

Population n:

```
p1              p2
Pass Words      Pass Words
SWAP 2          NOP
DEL             Pass Lines
DEL             NOP

p3              p4
Pass Words      Pass Words
SWAP 2          NOP
NOP             NOP
NOP             NOP
```

Fitting Ebel program

```
PASS Words
NOP
NOP
NOP
NOP
PASS derived Expression
  SUB $0, 2022, $0
  RETURN NOP
NOP
NOP
Pass Words
SWAP 2
```

```
01. (text)Lagrave -> ...
02. (text)Navara -> ...
03. (text)Carlsen -> ...
04. (text)Giri -> ...
```

```
Maxime,Lagrave,32,2758
David,Navara,37,2693
Magnus,Carlsen,32,2864
Anish,Giri,28,2773
```

Figure 4.7: Ebe's processing pipeline for compilation followed by interpretation.

Chapter 5

Implementation

Ebe is written in C++ (using standard C++17) and this choice was done for the purpose of performance, where speed of the compiler and interpreter was one of the goals for Ebe. It is designed as a free open-source project, that would not only work as a research project, but also as a useful tools. For the purpose of longevity, ease of adding changes, extending and allowing for others to build on Ebe or extend it, Ebe contains the following:

- Documentation comments in the Doxygen[1] format for all its source files. This documentation can then be easily generated using provided script or CMake tool[2]. Generated documentation has the form of structured HTML website.
- Unit testing. These tests use GoogleTest library[3], which allows to easily test Ebe's modules and make sure that no change had undesirable side effects in other parts of the compiler.
- Flex[4] and Bison[5] grammar files. These can be used to alter any already existing parser or scanner and generate it anew.
- Github Wiki page[6]. This page contains additional information about the project, such as detailed description of command line arguments, description of Ebel instructions or patch notes, which inform the users about the differences between Ebe versions.
- All of these resources are in the project's public Github repository[7] and are informally documented in the projects description (`README.md`) file with instructions for installation and some examples on how to use Ebe. Ebe's stable releases are also released in this repository.

Many of the parts mentioned above are not needed to compile the project for general use, but are there to simplify the development. To compile Ebe for general use, it requires only a C++ compiler and preferably CMake tool to allow for simple compilation. For the purpose of quick and easy deployment, the projects contains simple bash script (`install.sh`), which

[1]https://www.doxygen.nl
[2]https://cmake.org
[3]https://github.com/google/googletest
[4]https://github.com/westes/flex
[5]https://www.gnu.org/software/bison
[6]https://github.com/mark-sed/ebe/wiki
[7]https://github.com/mark-sed/ebe

runs the compilation and sets up command `ebe` to use Ebe from anywhere in the computer's terminal.

Since Ebe was developed using Linux operating system, it currently was tested and reliably works only on Linux, this will hopefully change later on.

The code is split into multiple directories (see Appendix A for detailed description of this structure). The whole project is split into three main parts, which virtually contain other directories, but are split for the purpose of project's tidiness. These parts are often used in literature to describe certain parts of compilers or interpreters [61]:

- Front end – Front end is the part that deals with parsing, scanning, AST generation and user input (arguments). Front end's work is the same for compilation and interpretation.
- Mid end – This section in literature might be a part of the back end, but in this projects mid end takes care of the internal representation (IR) and works with expressions. This is used by the compiler and interpreter as well.
- Back end – This section in case of compilation contains also the engine and does the Ebel generation and then its output. In case of interpretation, the instruction execution happens in this part and then the interpretation result outputting.

These sections intersect each other and are here mostly to divide the project into somewhat unified virtual sections to help with the development and to add additional modularity to it.

There are also sections, which either do not belong to any of these sections or belong to all of them. For example the tests are a separate part, but they utilize all of these sections in the testing and then the module with utilities and algorithms is used in all sections, since it is a module with general use.

Ebe also contains detailed logging system. This feature is important in case of an internal error, where it allows to output logging messages of Ebe's internal state and processes that are being done. This feature can be used by users to report any errors and help narrow search space for the bug. This is enabled using command line option and a list of functions can be provided for which the logging should be outputted or even all functions can output their logs. This feature also has a "verbosity level", which changes how detailed the logging messages should be. Example of logging messages can be seen in Figure 5.1.

5.1 Ebe testing and benchmarks

Since Ebe has internal heuristics, which set the evolution parameters, there needed to be a lot of experimenting and benchmarking. For this purpose a special program for Ebe benchmarking was created. This program is written in Python 3 language and offers simple command line interface with a few command line options to setup the benchmarks. The benchmarks are split into two parts. First is benchmarked the compiler and then the interpreter. Ebe compilation, which uses genetic programming, is very non-deterministic and therefore each compilation is run multiple times (the exact number can be set using command line argument) and all of the tests are run on a single core as a process with maximum priority (using Linux's `nice` command) and measured using `time` command. Results of these measurements are then saved into a JSON file with additional information, such as the platform's technical specification, the CPU usage, the compilation precision

```
ebe.cpp::interpret: Ebel preprocessor finished
ebe.cpp::interpret: Ebel scanner started
scanner_ebel.cpp::process: Started parser for a.ebel
ebe.cpp::interpret: Ebel IR:
PASS Words
  DEL
  DEL
  DEL
  NOP
  NOP
  NOP
  NOP

ebe.cpp::interpret: Ebel scanner finished
ebe.cpp::interpret_core: Text preprocessor for a.txt started
ebe.cpp::interpret_core: Text preprocessor finished
ebe.cpp::interpret_core: Text scanner started
ebe.cpp::interpret_core: Text IR:
01. (number)1 -> (delim). -> (delim)\0x20 -> (text)Lili -> (delim)\0x20 -> (text)Caudéran -> (delim):
02. (symbol)| -> (symbol)- -> (delim)\0x20 -> (text)Fleurs -> (delim)\0x20 -> (text)de -> (delim)\0x20
-> (text)sakura
03. (symbol)| -> (symbol)- -> (delim)\0x20 -> (text)près -> (delim)\0x20 -> (text)de -> (delim)\0x20 ->
 (text)la -> (delim)\0x20 -> (text)maison -> (delim)\0x20 -> (text)de -> (delim)\0x20 -> (text)thé
04. (symbol)| -> (symbol)- -> (delim)\0x20 -> (text)glisse -> (delim)\0x20 -> (text)un -> (delim)\0x20
-> (text)kimono

ebe.cpp::interpret_core: Text scanner finished
ebe.cpp::interpret_core: Interpreter started
ir.cpp::process: Words pass processing:
01. (number)1 -> (delim). -> (delim)\0x20 -> (text)Lili -> (delim)\0x20 -> (text)Caudéran -> (delim):
02. (symbol)| -> (symbol)- -> (delim)\0x20 -> (text)Fleurs -> (delim)\0x20 -> (text)de -> (delim)\0x20
-> (text)sakura
03. (symbol)| -> (symbol)- -> (delim)\0x20 -> (text)près -> (delim)\0x20 -> (text)de -> (delim)\0x20 ->
 (text)la -> (delim)\0x20 -> (text)maison -> (delim)\0x20 -> (text)de -> (delim)\0x20 -> (text)thé
04. (symbol)| -> (symbol)- -> (delim)\0x20 -> (text)glisse -> (delim)\0x20 -> (text)un -> (delim)\0x20
-> (text)kimono

ir.cpp::process: Current instruction: DEL; Current word: (number)1
ir.cpp::process: Current instruction: DEL; Current word: (delim).
```

Figure 5.1: Snippet of Ebe logging messages.

achieved, version information and others (see Listing 5.1 for example output). Since the compilation might take a very long time (because code synthesis using genetic programming) in some cases or might even be impossible to compile by Ebe, a timeout is used. This timeout is set using Ebe's timeout option and is by default set for 300 s, but can be changed using command line arguments for benchmarks. Once the benchmark results are outputted, they can be plotted into a graph using a Python 3 plotting script, that uses `matplotlib` library and allows to plot graphs for separate compilation and interpretation or both graphs in one image. The plot can also be a box plot or even a horizontal bar plot. Box plot graphs from this scripts were used throughout this whole thesis (for example see Figure 5.5, 5.4 or 5.7) and the bar plot can be seen in Figure B.1. The bar plot displays median value from all the runs, but some bars might have different format (hatching) to signal that no run had 100 % precision. Whereas in case of box plot it can be easily seen, if there are values only in the 300 s mark.

```
{
  "benchmark": {
    "version": "1.0.0",
    "time:": 1650901857,
    "args": "-f lev"
  },
  "platform": {
    "memory": { "size": 8202158080 },
    "cpu": {
      "model": "Intel(R) Core(TM) i5-4690 CPU",
      "freq_min": 800.0,
      "freq_max": 3900.0,
      "cores": 4
    },
    "os": "Linux-5.13.0-39-generic-x86_64-with-glibc2.29"
  },
  "ebe": { "version": "0.3.1" },
  "results": {
    "ebec": {
      "csv_del_begin": {
        "times": [ 2.66, 303.46, 0.03, 0.03, 0.03 ],
        "cpus": [ 99, 99, 96, 96, 96 ],
        "precisions": [ 100.0, 46.1538, 100.0, 100.0, 100.0 ]
      }
    },
    "ebei": {
      "log_unit_remove": {
        "times": [ 3.91, 3.86, 3.91, 3.88, 3.88 ],
        "cpus": [ 99, 99, 99, 99, 98 ]
      }
    }
  }
}
```

Listing 5.1: Example output of Ebe benchmarks with only 1 test run 5 times.

Anyone can specify their own benchmarks and use the benchmarking scripts. This is because benchmarks only require to follow a set directory structure, which is the following (examples structure can be seen in Listing 5.2 and 5.3):

- Ebec and Ebei tests have to be in their separate folders.
- Each test has to be in its own folder in the top level of the Ebec or Ebei folder.
- Each Ebec test has to have a file with `.in` and `.out` extensions, where these files will be used as the example input file and example output file, respectively. There can also appear a file with `.args` extension, which first line would be passed to Ebe on the command line, meaning that it can be used to specify certain command line arguments for Ebe (mainly `-expr` argument).
- Each Ebei test has to have a file with `.ebel` extension, which will be used as the Ebel code to interpret and then at least one file with `.txt` extension, where all of these files will be used as the input files for interpretation. And just as it was for Ebec, `.args` file can appear.

Any other folders and files other than the mentioned ones are ignored by the script. The test folder name also plays a role, since the name of the folder is then used as the test name in the JSON file with the results and this name can be also used when running only specific set of tests (using `-t` option).

```
ebec-tests/
|- test1/
|   |- test1.in
|   |- test1.out
|   |- test1.args
|- test2/
    |- test2.in
    |- test2.out
```

Listing 5.2: Example of correct benchmark directory structure for Ebec.

```
ebei-tests/
|- test1/
|   |- test1.ebel
|   |- test1.args
|   |- a.txt
|   |- b.txt
|   |- c.txt
|- test2/
    |- test2.ebel
    |- test2.txt
```

Listing 5.3: Example of correct benchmark directory structure for Ebei.

These benchmark problems were designed in order to evaluate Ebe and as well these problems are trying to simulate the usual problems, for which Ebe could be used:

- Ebec benchmarks:
  - **`csv_del_begin`** – Multiple line CSV example with header. Where multiple columns from the beginning are to be removed (only columns 5, 7 and 9 should be kept).
  - **`csv_del_end`** – Similar CSV as `csv_del_begin`, but this time the data to be kept are at the beginning.
  - **`csv_no_header`** – Similar CSV as `csv_del_end`, but there is no header.
  - **`csv_short`** – CSV with header, where only 2 columns are to be removed.
  - **`human_genome_edit`** – Test taken from real world Ebe use case. This tests multiple user defined expressions.
  - **`user_expression1`** – Tests a big user defined expression.
  - **`log_unit_remove`** – Edits are done on Ebe logs (example logs can be seen in Figure 5.1). This contains more difficult structure and most of it is to be removed to extract formatted values.
  - **`phone_numbers`** – Multiple line text file with names followed by phone numbers, where Ebe should attempt to reformat the numbers.
  - **`ws2811_colors`** – Test taken from real world Ebe use case. Task is to extract a column from markdown table.

- Ebei benchmarks:
  - **`filter_calc`** – The Ebel program multiplies all numbers by 0, keeps in delimiters and removes any other word type. The interpreted file is a 10000 lines long text with randomly generated symbols, letters and numbers.
  - **`filter_num_sym`** – The Ebel program removes from the input file everything that is not a number or text. Input file is the same as for `filter_calc`.
  - **`human_genome_edit`** – The Ebel program is the one generated for Ebec test `human_genome_edit`. Input is a 389 lines long .gtf file.
  - **`human_genome_edit_big`** – Same as `human_genome_edit`, but the input file is a 50 MB big .gtf file.
  - **`log_unit_remove`** – The Ebel program is the one fitting for Ebec test `log_unit_remove`. Input file is part of actual Ebe log with 121810 lines (almost 8 MB).
  - **`ws2811_colors`** – The Ebel is the one fitting Ebec `ws2811_colors` test. Input file is 142 lines long markdown table.

Although not every part of Ebe is tested with these tests, it already gives somewhat general results for Ebe's performance, where the main goal is to find differences in runs based on changed parameters.

5.2 Internal representation

Since most file formats use columns and rows for data representation, this is also manifested into the internal representation (*IR*) of the files in Ebe. This means that files are internally represented as list of lines, where each line is a list of words. This structure is contained in a `IR::Node` object, which alongside it hold metadata and methods to simplify its processing. The `IR::Node` object is created for every input file (examples as well as files to edit). Each `IR::Word` object contains the textual value of the parsed lexeme and its type. Graphical visualization of a text IR without metadata can be seen in Figure 5.2.

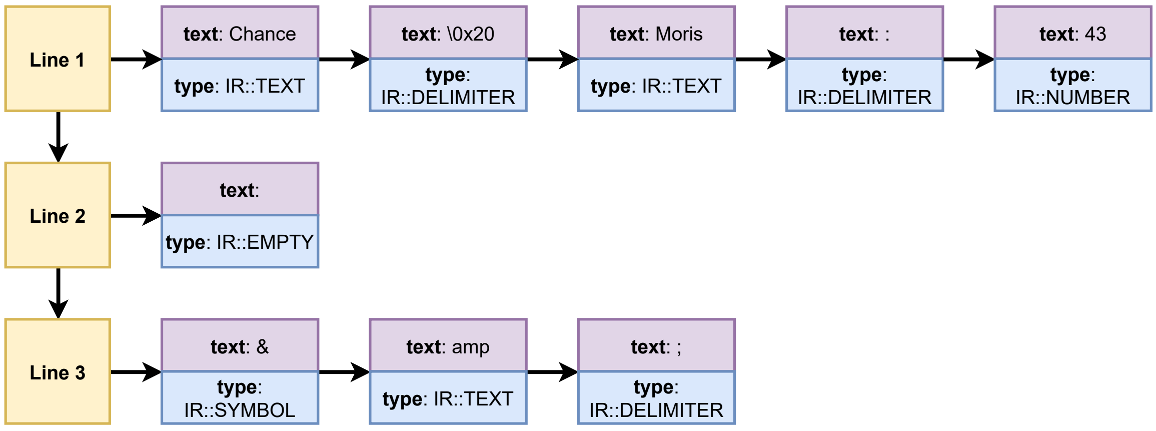


Figure 5.2: Graphical visualization of internal representation for a text file.

Ebel program internal representation is implemented as a list of pass objects (`IR::Pass`). This is accompanied with additional metadata, methods and a `Pragmas` object holding the program pragmas. Each pass in the IR holds a vector of Ebel instructions. The pass object also contains a vector of additional passes, which is utilized by words pass to hold its subpasses (expression passes). This acts somewhat as a method table [49], where each subpass has its index in the vector, which is then used by special `CALL` instruction that executes these subpasses. The parser rewrites all the Ebel code into the version with `CALL` instruction during parsing. Graphical visualization of an example Ebel program can be seen in Figure 5.3. Notice that the return instruction is held by the words pass and not the expression pass. This is because the return instruction processes the value in the context of the whole line.

```
PASS Words
  NOP
  SWAP 42
  Pass NUMBER Expression
    MUL $1, 3, $0
    SUB $0, $1, $0
    RETURN NOP
  Pass DERIVED Expression
    RETURN DEL
PASS Lines
  CONCAT 1
  LOOP
```

Listing 5.4: Example Ebel program.

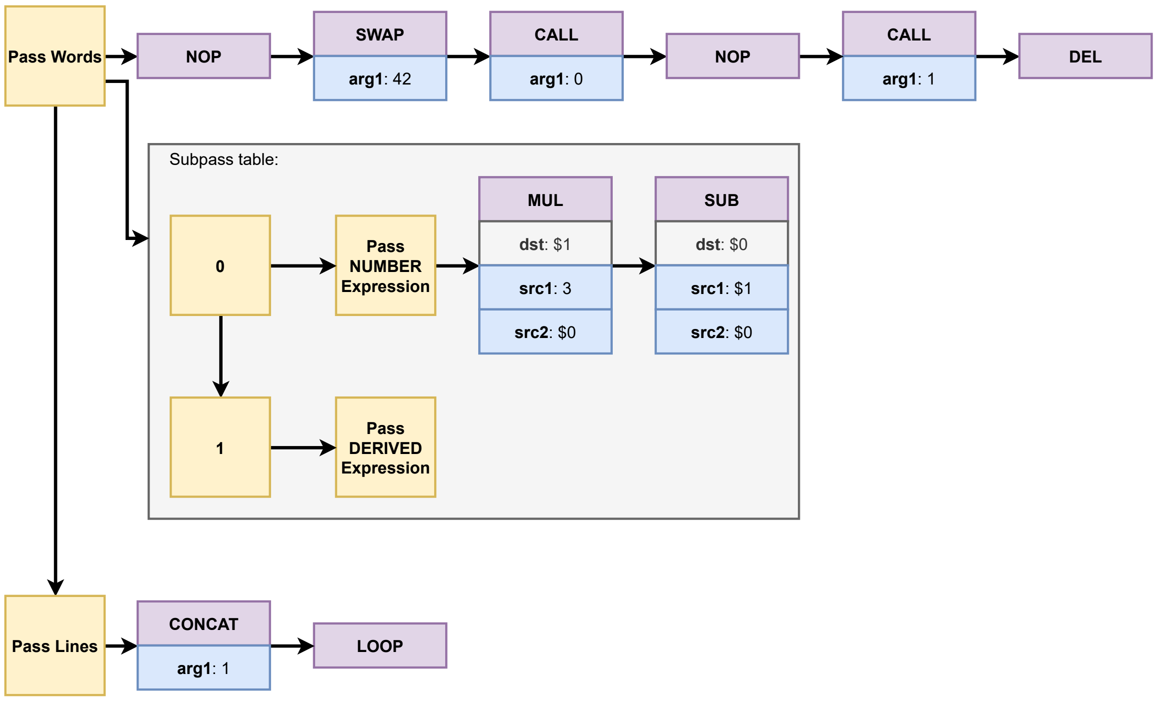


Figure 5.3: Graphical visualization of internal representation for Ebel program from Listing 5.4.

5.3 Implemented engines

As already mentioned, Ebe contains multiple engines and allows for addition of new ones. For all genetic engines it is important to correctly choose evolutionary attributes, such as the mutation chance, population size or initial phenotype size. But because Ebe aims also at those, who do not know anything about genetic programming, it is not possible to acquire these parameters from the user or test out multiple ones during runtime. Rather Ebe relies on its heuristics learned from experiments in combination with a simple input analysis to set these parameters. All genetic engines contain special structure `GPEngineParams`, which holds all these adjustable attributes. This structure is populated at the time of engine's creation, where some of the attributes, that rely on the input, such as initial phenotype size or the number of iterations (generations), are computed by firstly analyzing example input and example output using a fitness function. This comparison returns how similar the examples are. For more similar examples it is more likely, that less edits will be needed, and therefore, the phenotype can start smaller, with less passes and should probably converge with less iterations. The other attributes were hand adjusted during development to achieve the best results. Default values set for all genetic engines can be seen in Table 5.1.

Parameter	**Default value**
Population size	250
Fitness function	One-to-one
Chance of words pass generation	100 % if the input file has 1 line, 80 % if the input file has less than 11 lines, 50 % if the input file has more than 10 lines.
Chance of lines pass generation	100 % − Chance of words pass generation
Mutation chance	15 %
Crossover chance	65 %
Elitism	True

Table 5.1: Some default evolution parameters for engines. Phenotype size, number of passes and pass types are determined based on example input and example output similarity and example input file length.

When it comes to editing file, each input differs, but some edits are more likely to happen than others. For example most of the generated and handwritten Ebel solutions contain many `NOP` instructions and then very often `DEL` instruction. For this reason, the chances for an instruction to be placed into a phenotype of to be mutated into differ. On top of that, some instructions cannot even appear in certain passes (e.g. `CONCAT` instruction in `Pass Words`). The default chances for instruction to appear in a program can be seen in Table 5.2.

Instruction	**Words pass chance**	**Lines pass chance**
CONCAT	0 %	12.5 %
DEL	31.25 %	25 %
LOOP	12.5 %	12.5 %
NOP	43.75 %	37.5 %
SWAP	12.5 %	12.5 %

Table 5.2: Default instruction chances set for any genetic engine. These values can be adjusted by the engine and, for example, engine Taylor sets all, but `NOP` to 0 % and `NOP` to 100 %.

Engine Jenn and Taylor also contain "bloat removing procedure", which is a process, in which instructions, that are not executed during interpretation are removed. This process makes sure that the phenotypes do not become too long, which would slowdown the evolution, because the genetic operators would be applied to parts of the code, which is not even being run.

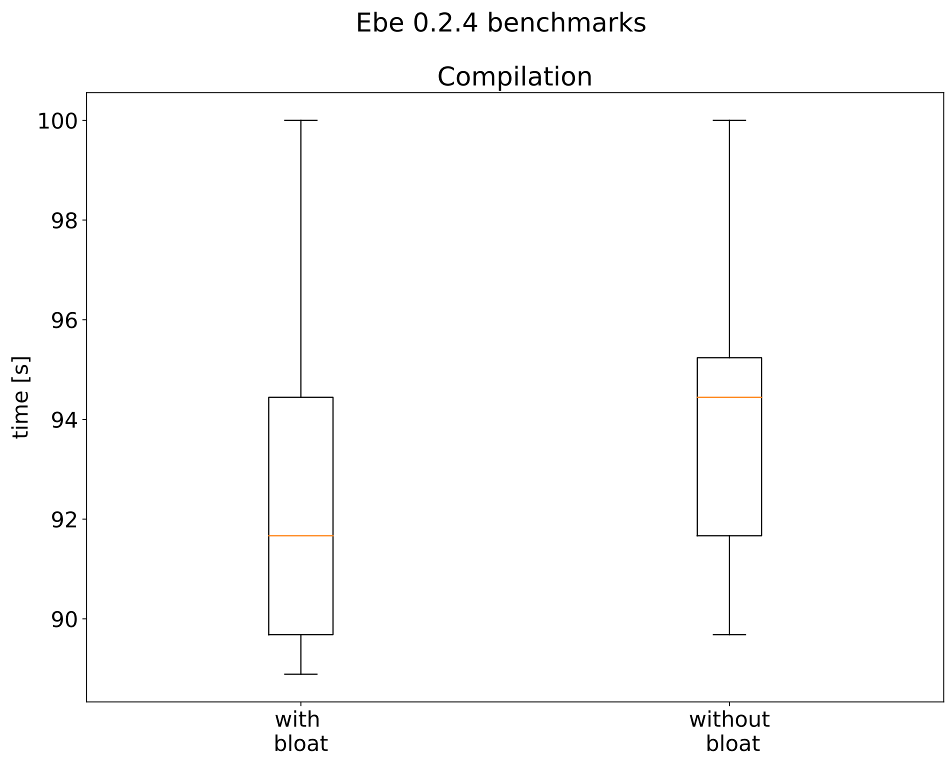


Figure 5.4: Effects of phenotype bloat to its fitness convergence. Represented are values from 30 independent runs on the same example inputs (`csv_no_header`) with the same evolutionary parameters.

Based on benchmarks, which results can be seen in Figure 5.5, engine Jenn was picked as the main engine for Ebe.

Experiments have also proven, that “bloat” makes the population converge faster (see Figure 5.4). Because of this the removal of excess instructions happens only once a set length (which is based on input length) is reached.

Unexpected results were discovered with fitness function comparison, where Ebe benchmarks showcases, that the best time performance is achieved using the most simple *one-to-one* fitness function. Results can be seen in Figure 5.6.

Figure 5.7 showcases very well the results of different population sizes. The lower quartile displays that the smaller population can achieve the fastest time, which is because of its smaller size, and thus, lower initialization time of the population and less operations overall done. On the other hand, the bigger population converges quicker, this is because of the bigger variety in phenotypes, and therefore, the population covers bigger space of the problem. This means, that for a simpler problem a smaller population might converge faster just because of the computational slowdowns, but for a bigger problem, where the evolution itself takes longer time then the initialization, it might be more beneficial to choose a bigger population. On the other hand making the population too big might result in slowdown caused by the computational platform, such as the population taking too much space resulting in problems with caching data.

5.3.1 Genetic engine Jenn

Engine Jenn is a genetic engine, which tries to adhere to the most general approach of genetic programming for code generation and does not specializes on any file format. But Jenn still offers lots of parameter customization to aid the code evolution.

For the purpose of speed optimization, crossovers are done by modifying one of the parents instead of creating a copy of it, which would take a longer time.

```
**Input:** An example input file $f_i$, an example output file $f_o$
**Data:** A best Ebel program $p$ and its fitness
$population \leftarrow$ generate random population of $n$ programs
for $i \leftarrow 0$ to $iterations$ do
    foreach $p \in population$ do
        $p_f \leftarrow$ fitness($p$, $f_i$, $f_o$)
        if $p_f = 100\%$ then
            return $p$, $p_f$
        end
    end
    $population \leftarrow$ sort($population$)
    foreach $p \in population$ do
        if $elitism \land$ first($population$) $= p$ then
            continue
        end
        if random_chance() < mutation chance then
            mutate($p$)
        end
        if random_chance() < crossover chance then
            $p_2 \leftarrow$ random($\{a \mid a \in population \land a \neq p\}$)
            $p \leftarrow$ crossover($p$, $p_2$)
        end
    end
end
foreach $p \in population$ do
    $p_f \leftarrow$ fitness($p$, $f_i$, $f_o$)
end
$p_{best} \leftarrow p \in population \land p$ has highest fitness
return $p_{best}$, $p_{best_f}$
```

Algorithm 2: Jenn engine algorithm.

5.3.2 Mutation engine Taylor

Engine Taylor is a genetic engine, that uses only mutation in its evolution process. This engine is mostly experimental, but might bring some speedup in certain cases, mostly there, where are a lot of `NOP` operations, since Taylor uses "NOP tails" (hence the name). The use of NOP tails here, is that at the beginning of the evolution all phenotypes contain only `NOP` operations, which are during evolution mutated into different instructions. The use of only mutations can help with a speedup, since no crossover is done and so the overall evolution has a simpler computation.

```
Input: An example input file $f_i$, an example output file $f_o$
Data: A best Ebel program $p$ and its fitness
$population \leftarrow$ generate random population of $n$ programs containing only NOP
  instructions in passes
for $i \leftarrow 0$ to $iterations$ do
    foreach $p \in population$ do
        $p_f \leftarrow$ fitness($p$, $f_i$, $f_o$)
        if $p_f = 100\%$ then
            return $p$, $p_f$
        end
    end
    $population \leftarrow$ sort($population$)
    foreach $p \in population$ do
        if $elitism \land$ first($population$) $= p$ then
            continue
        end
        if random_chance() < mutation chance then
            mutate($p$)
        end
    end
end
foreach $p \in population$ do
    $p_f \leftarrow$ fitness($p$, $f_i$, $f_o$)
end
$p_{best} \leftarrow p \in population \land p$ has highest fitness
return $p_{best}$, $p_{best_f}$
```

Algorithm 3: Taylor engine algorithm.

5.3.3 Random engine MiRANDa

Engine MiRANDa uses pure random approach (see Algorithm 4) and was designed just for the purpose of engine comparisons and to showcase, that the evolution approach in genetic engines is far more advanced and is not only about randomness, which does play a big role in it (this can be seen in Figure 5.8).

```
**Input:** An example input file $f_i$, an example output file $f_o$
**Data:** A best Ebel program $p$ and its fitness
$p_{best} \leftarrow$ NIL
$p_{best_f} \leftarrow -\infty$
**for** $i \leftarrow 0$ **to** *iterations* **do**
    $p \leftarrow$ random_program()
    $p_f \leftarrow$ fitness($p$, $f_i$, $f_o$)
    **if** $p_f > p_{best_f}$ **then**
        $p_{best} \leftarrow p$
        $p_{best_f} \leftarrow p_f$
    **end**
**end**
**return** $p_{best}, p_{best_f}$
```

Algorithm 4: MiRANDa engine algorithm.

Figure 5.8 showcases the best candidate fitness of 5 independent runs, in which the same input (markdown table column extraction from experiment 1 in Section 6.1) was used to evolve a program using genetic algorithm (Engine Jenn) and random approach (Engine MiRANDa). Not only that the genetic approach could find fitting program in all of the 5 independent runs within 100 iterations, but it also showcased, that with more time the population performance (best fitness) only improves, unlike in random approach. The reason for only growing fitness is the use of elitism, which makes sure that the current best candidate is not modified and stays in the population. It is also worth mentioning, that the given problem has quite a simple solution using Ebel. Given long enough time the random approach can find this program with its brute-force approach.

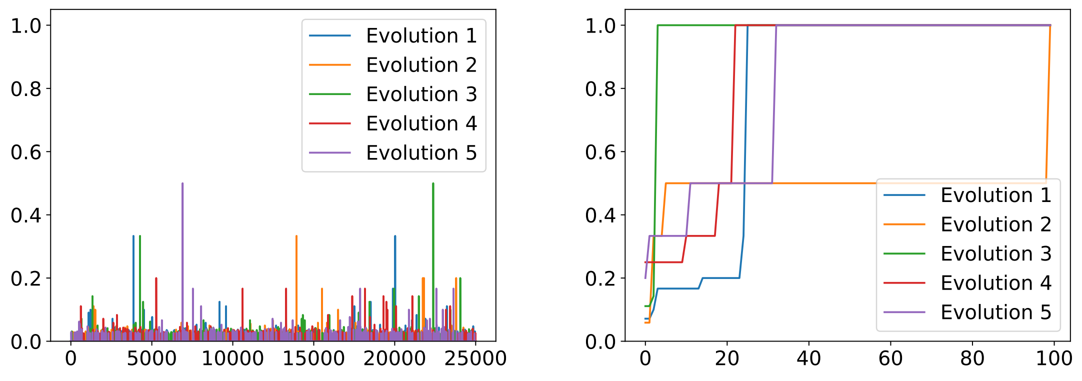


Figure 5.8: Comparison of 5 program generation evaluations using MiRANDa engine and Jenn engine. To have both engines process the same amount of evolutionary cycles, MiRANDa uses 25000 iterations and Jenn uses 100 iterations with 250 phenotypes.

5.4 Implemented optimizations

Ebel generation is done evolutionary, and therefore, most optimizations are good to apply once a fitting program is found. But Ebel already has a quite low abstraction and only variables are in expression passes, which are generated from user defined expressions. Here can be also very little optimized, but the code generation takes advantage of Ebel expression instructions (having separate destination argument) and does not use any `MOVE` instruction, unless `MOVE` would be the only instruction.

Ebe's implementation also plays a big part of the overall compilation and interpretation speed. That is why C++ language was chosen, which is quite widespread language with many years of development offering multiple compilers with great optimizations [60]. There are also garbage collector libraries for C++ [9], but a garbage collector was not used to not slowdown the compiler and interpreter any further, but mainly to allow for better portability since many of the libraries are platform dependant.

5.4.1 Dead code elimination

Compiler, more specifically engine, during evolution generates Ebel code. However, since it has access to the user example inputs, it knows the exact amount of instructions needed, where this amount is equal to at most the length of the input example file. Dead code elimination cannot be done just by cutting off part of the code exceeding the input file length, since some instruction don't "consume" the input word and also words and lines pass start the input file processing anew, so this optimization has to be done for each pass.

This can be approached in two ways. The first option is to do static analysis of the code and input and calculate what part of the code will be executed and what part won't be executed for each pass. The second option is to run this code in the interpreter and see what part of the code is not used. Even though the second option might sounds as more time consuming, the opposite is the case, since this interpretation is always done in order to get fitness value of this program (phenotype) for the evolution, and so, this approach adds only a few more checks.

5.4.2 Redundant code elimination

Ebe engines might create redundant instructions, which are mostly control instructions, such as `LOOP` instruction. In Ebe only one `LOOP` instruction in a pass will be executed and that is the last one. Because of this, all other previous `LOOP` instructions can be safely removed as they do not read in any input and do not change the control flow.

On top of this, if `LOOP` is the first instruction in a pass it can be removed as well, since there are no instructions before it to loop over.

```
interpreter.cpp::eliminate_redundat_code: Eliminated LOOP from:
PASS Words
  DEL
  LOOP
  SWAP 4
  LOOP
  NOP
  LOOP
  LOOP
  NOP
  LOOP
  LOOP
  DEL
  SWAP 5
  DEL

interpreter.cpp::eliminate_redundat_code: ..to:
PASS Words
  DEL
  SWAP 4
  NOP
  NOP
  LOOP
  DEL
  SWAP 5
  DEL
```

Figure 5.9: Ebe logging showcasing redundant code elimination over evolved programs.

5.4.3 Constant folding

This is achieved by correctly writing EBNF (*Extended Backus-Naur Form*) grammar in Flex, so that constant numeric expressions are evaluated first and folded. Example of this can be seen in Figure 5.10. This method is also described in Section 2.5 and example can be seen in Listing 2.4.

```
[marek@fedora ebe]$ cat e.in
{! 40 + 2 * 1 + $ ^ 2 - 5 * 3 !}
[marek@fedora ebe]$ ./build/ebe -in e.in -out e.output -v ebe.cpp::compile
ebe.cpp::compile: Text IN IR:
01. (expression)$=42+$^2-15
```

Figure 5.10: Constant folding (represented in IR form on the last line) of user defined expression (line 2).

5.5 Ebe's modularity and extendability

Ebe's implementation is done in a way, so that it can be easily extended or build upon. On top of already mentioned use of Flex and Bison for simple scanner and parser addition, Ebe is also split into multiple connected, but quite independent parts. Complete separation of modules was not chosen, since it would slowdown the the compilation and interpretation and speed is quite important for Ebe. The two main modules are the compiler and interpreter, both of them can be used independently. This allows for projects like the Bee language and Bee compiler[8] to compile down to Ebel and then use just the Ebe interpreter.

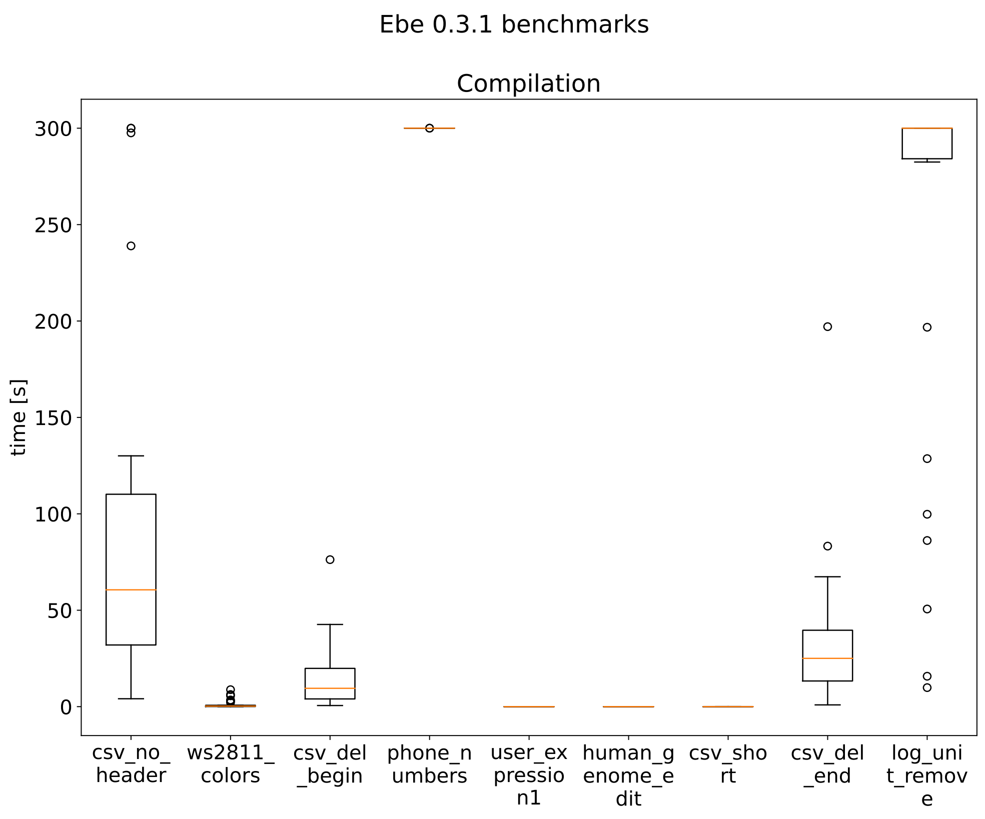
If one was to add a new file format to achieve better compilation results, for possibly even a custom file format, the process would be following:

1. Write a grammar file for Bison and Flex and possibly any additional source files, to handle the input.
2. Write class, which extends `Scanner` class and add a method, which returns a new instance of internal representation of the input file (`IR::Node`).
3. Once implemented, any call to a parser can be replaced with the new parser.

The main reason for multiple parsers and scanners is to optimize the compilation or possibly even allow for edits, which the standard parser does not allow because of the grammar used. This can be easily described on a simple example. Let us assume that one wants to edit a `fasta` format file. This file contains a header and then an uninterrupted string of letters with the possibility to add new lines for better readability [54]. If this input was given to Ebe's default text file parser, then the whole sequence would be parsed as one word with the type `TEXT`, but what would be rather desired is to parse each letter as a separate lexeme, which would allow, for example, to use expression pass with string matching to replace certain letters for different ones.

It would be also possible to add a new parser and scanner for Ebel language, although this does not seem very useful and it would probably be lot easier to change the current grammar used. But if this were to be done, the process would be quite the same to the one described for a new input file parser.

[8]https://github.com/mark-sed/bee-hs



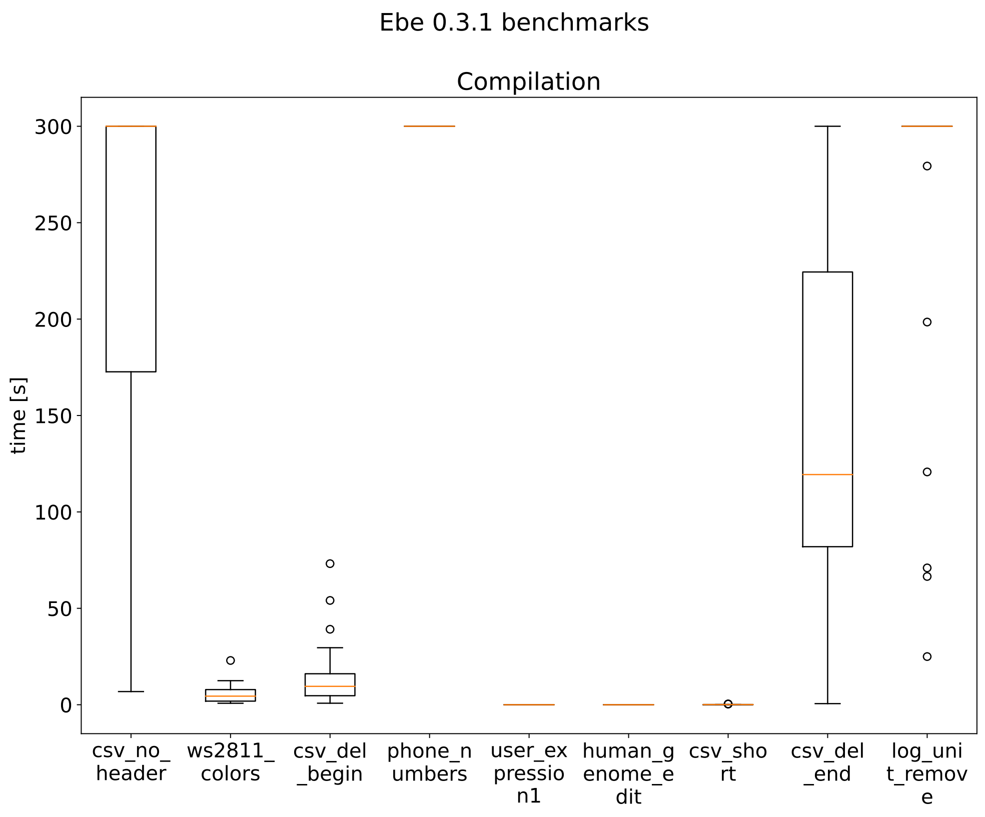


Figure 5.5: Benchmark results for engine Jenn (top) and engine Taylor (bottom).

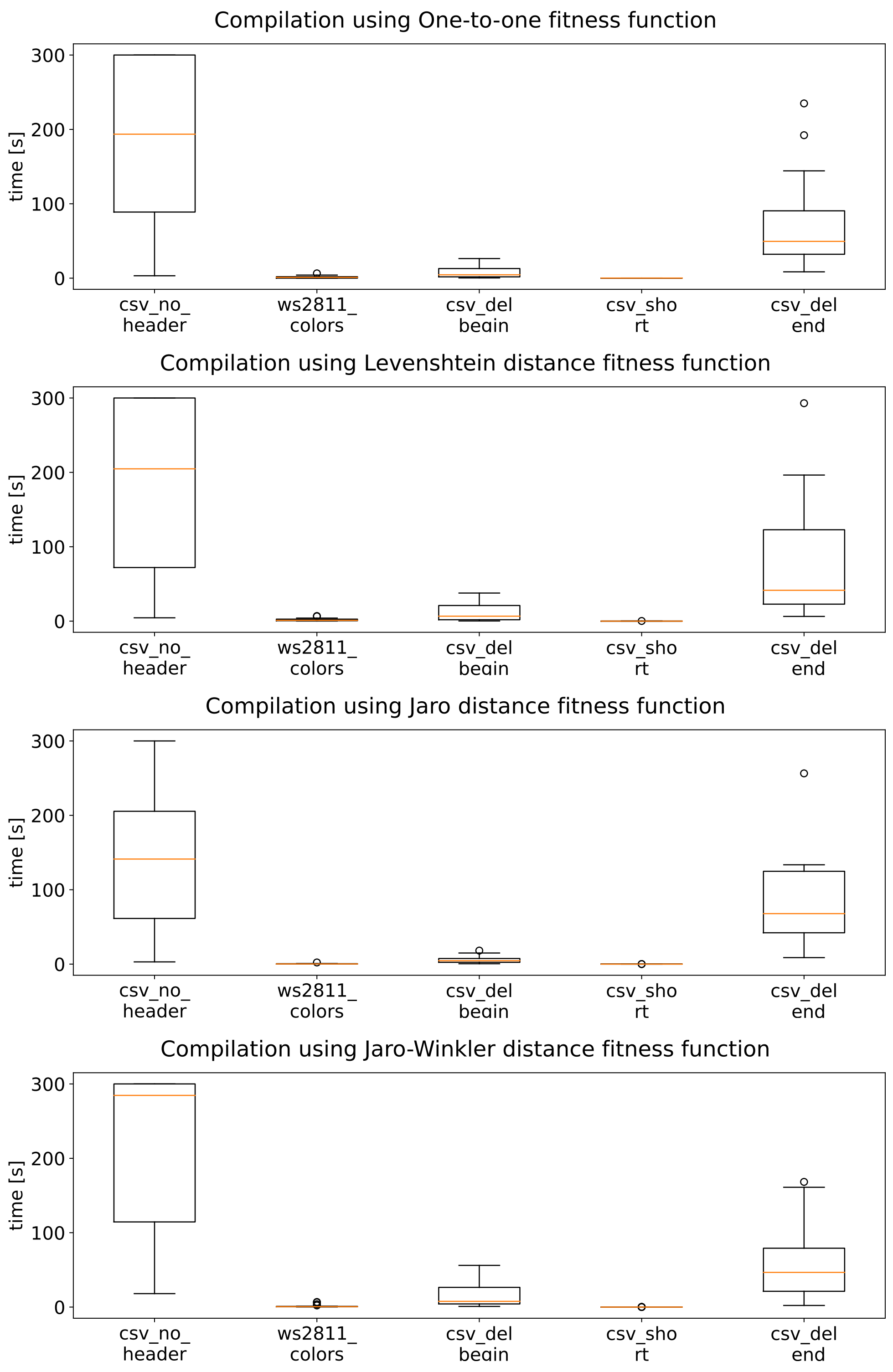


Figure 5.6: Comparisons of compilation times (solution program synthesis) for different Ebe fitness functions. These functions were tested on a subset of benchmark problems (described in Section 5.1) and displayed are values of 20 independent runs of each benchmark problem. Default Ebe parameters were used (see Table 5.1).

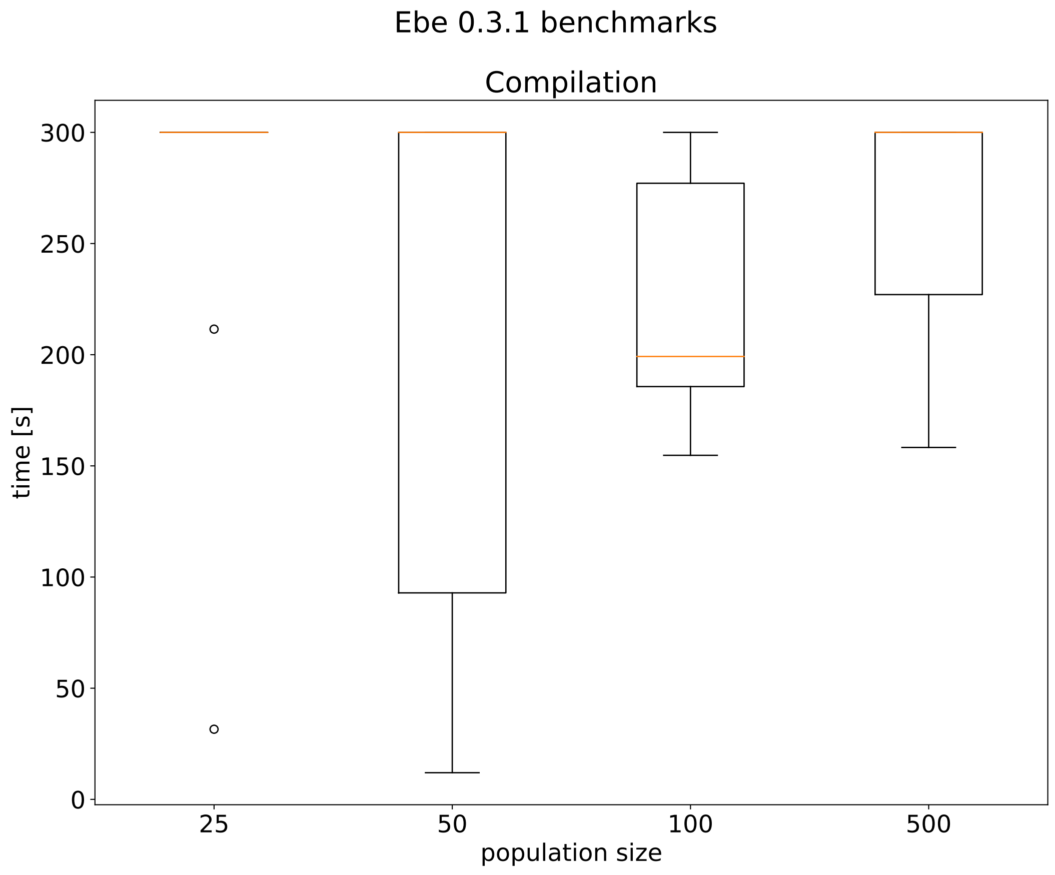


Figure 5.7: Box plots of times of 10 measured runs of Ebe on the same problem (`log_unit_remove` benchmark) with normalized generations (meaning that the amount of generations was proportioned to the population size). The benchmarks used timeout of 300 s, so runs with this time had precision lower than 100 %.

Chapter 6

Experiments and evaluation

Ebe is quite a unique application, and so, it is not very possible to compare its time of editing program synthesis to some other application. But as was mentioned, Ebe promises to be an alternative tool for file editing for non-programmers and also programmers and this is in some way possible to be measured. But here it is not only the time it take some language to process a file, but also about how long it takes to do the whole editing task. Where Ebe excels is, that it does the job of writing the editing script for the user, and thus, saves all of this time. One could also argue, that the compilation time itself should not be measured and only the preparation of examples, since during the compilation Ebe does not require any interaction and the user can do other tasks during this time, but for the sake of as equal grounds as possible the whole process including compilation will be included. Another big difference is the nondeterministic nature of Ebe, which makes each attempt to take different time. For this reason each Ebe edit will be done 10 times and a median value will be used as a result. All results will be in seconds precision.

Another problem with these experiments is the overall knowledge of other measured approaches, since every person has a different knowledge of some language or tool, it is impossible to get some general result. There are also many, even infinite, ways to write a script for one editing task and therefore this can differ based on different libraries or tools used in some language. But at the same time Ebe also offers different approaches and some parameters, which can be adjusted to make it work better in certain cases. All of this is being said to acknowledge, that these results may vary from person to person and the experiments are here mostly to showcase one of the cases to give some vague idea of the main differences.

Even though it would be possible to come up with very complicated editing experiments to limit test all the tools, it seems more useful to test these tools on tasks, that are quite likely to be solved in real life, but still pose some challenge to the selected tools.

All the experiments were done by hand using environment the author was most comfortable with and all was measured from the start of programming or editing to the point, where the tool created expected output.

As was already mentioned, all the experiments with Ebe were done 10 times including the example creation and everything was done by hand. In case of editing the file using text editor, only the first 50 lines were edited, since unlike programming languages or Ebe, this approach does not scale well, but the 50 lines already give a good sample.

Experiments were solved using tools and languages, which are suitable for the task. It would be possible to solve these experiments using languages like C or Haskell, but because of the nature of these languages, they do not seem like suitable tools. Used languages and tools, excluding Ebe, were:

- **Text editor** – This approach is the least technical, but is included, because this is the approach that any non-programmer can use and probably will use. This tool should be also available to any system that works with files, since the editor is also used to view the contents of the file.

- **Linux tools** – This is not just a one program, but rather a set of programs usually available on Linux or Unix systems, where Linux allows to combine these programs into a working pipeline. Examples of such programs are `cat`, `grep`, `sed`, `cut`. . . .

- **AWK** – AWK is also a part of Linux tools, but since it is a program made exactly for editing files it deserves to be tested on its own. AWK contains its own parser and extracting tools, which allow for simple file transformation. On the other hand, the syntax is quite unique and requires some knowledge of it to use the tool properly.

- **Python 3** – Python 3 is a popular interpreted general purpose language with many tools for working with files and even offers some built-in libraries for work with specific formats, such as CSV (*Comma Separated Values*) or JSON (*JavaScript Object Notation*) [52].

- **Handwritten Ebel** – This approach is mostly here to showcase the option to write the Ebel script by hand, although this approach requires deep knowledge of the parser Ebe uses and the Ebel syntax and semantics. Ebel is quite a simple language, but quite unique in its computing model and therefore not as straightforward to use by those, who do not know it well. On the other hand many simple edits can be done by just matching lexemes and simple instructions (`NOP` and `DEL`) in a single pass to construct a working Ebel program.

6.1 Experiment 1 – Markdown table value extraction

A simple real world problem and therefore a valuable experiment. In this experiment the task is to extract hexadecimal values from a markdown table containing all predefined colors in the FastLED library[1]. The table is extracted from the markdown file without the header and contains 142 lines and the table has 3 columns with the desired value in the second column. See Listing 6.1 and 6.2 for the snippet of the file before and after editing, respectively.

```
| CRGB::AliceBlue | 0xF0F8FF | <img src="http://www.colorcombos.com/images/colors/hex/F0F8FF.png"/>
| CRGB::Amethyst | 0x9966CC | <img src="http://www.colorcombos.com/images/colors/hex/9966CC.png"/>
| CRGB::AntiqueWhite | 0xFAEBD7 | <img src="http://www.colorcombos.com/images/colors/hex/FAEBD7.png"/>
```

Listing 6.1: First 3 lines of the markdown file (`supported.md`) before edits.

[1]https://github.com/FastLED/FastLED/wiki/Pixel-reference

```
0xF0F8FF
0x9966CC
0xFAEBD7
```

Listing 6.2: First 3 lines of the markdown file (`edited.md`) after edits.

6.1.1 Ebe solution

For Ebe solution the example input file was created by taking the first line from the markdown file (see Listing 6.3) and this was then copied and edited by hand to include only desired value (see Listing 6.4). The final edited file was then generated with single Ebe execution using the `-x` option, which allows for compilation and interpretation after each other. The command was `ebe -x -in f.in -out f.out -o edited.md supported.md` and output was placed Ebe into `edited.md` file. As can be seen in the Table 6.1 the example generation took the most time, since all compilation runs took way less than a second.

```
| CRGB::AliceBlue | 0xF0F8FF | <img src="http://www.colorcombos.com/images/
   colors/hex/F0F8FF.png"/>
```

Listing 6.3: Contents of the input example file `f.in`.

```
0xF0F8FF
```

Listing 6.4: Contents of the output example file `f.out`.

The editing algorithm found by Ebe in one of the attempts can be seen in Listing 6.5. This solution differs very much from a solution, which would be written by a programmer (see Listing 6.6). This is caused by the nature of how genetic programming works and synthesises its solutions.

```
PASS Words
  DEL
  DEL
  DEL
  NOP
  LOOP
  DEL
PASS Words
  DEL
  LOOP
  NOP
  SWAP 1
PASS Words
  NOP
  DEL
  DEL
  LOOP
  DEL
```

Listing 6.5: Ebel solution generate in one of the experiment 1 attempts by Ebe.

Ebe can find the correct solution in very short time (as can be seen in Table 6.1). The population fitness convergence can be seen in Figure 6.1, where for these 5 runs no more than 12 iterations of evolution were needed.

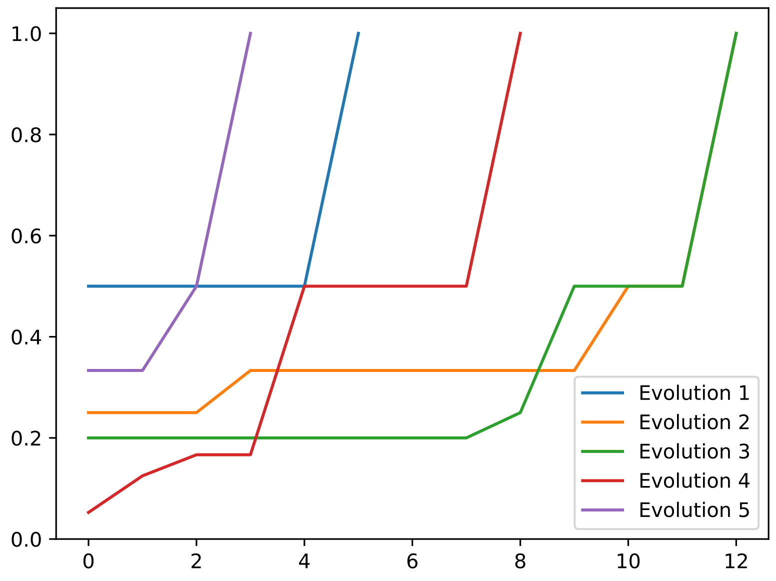


Figure 6.1: Fitness values for 5 independent runs of Ebe for experiment 1. These runs were not part of the 10 attempts done, since the fitness value logging slows down the editing.

Attempt	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Edit time	36 s	36 s	31 s	34 s	34 s	31 s	32 s	31 s	31 s	27 s
Ebe time	0.4 s	0.3 s	0.6 s	0.1 s	0.4 s	<0.1 s	<0.1 s	0.1 s	0.2 s	<0.1 s

Table 6.1: Edit times for all attempts solving experiment 2 with Ebe. Median value is 31.5 s (rounded to 32 s).

6.1.2 Ebel solution

Writing Ebel program requires knowledge of the parser and analyzing the input file. First it is needed to count the number of lexemes before the value, that is to be deleted. Then it is needed to know, if the extracted value is a single lexeme or multiple and finally, if all the words after the extracted one are to be deleted a `LOOP` instruction can be used, otherwise it would be needed to count the lexemes once again.

In this case, there are 9 lexemes before the wanted value, so 9 `DEL` instructions are needed to be placed, then a `LOOP` instruction followed by `NOP`. This takes advantage of the way `LOOP` instruction works (see Section 4.1.4), where the looping commences after the whole pass was processed. This solution can be seen in Listing 6.6. To interpret the Ebel `ebe` command with `-i` can be used – `ebe -i md.ebel -o edited.md supported.md`.

```
Pass Words
  DEL
  DEL
  DEL
  DEL
  DEL
  DEL
  DEL
  DEL
  DEL
  LOOP
  NOP
```

Listing 6.6: Handwritten Ebel solution (`md.ebel`) for experiment 1.

6.1.3 AWK solution

This problem is the perfect task for AWK, since it is a value extraction in a file with clearly defined columns. AWK parses all these columns and these can then be easily indexed and printed. The used AWK code can be seen in Listing 6.7.

```
{print $4}
```

Listing 6.7: AWK solution for experiment 1.

6.1.4 Python 3 solution

Since Python 3 is not a language designed purely for file editing, there needs to be a section for loading the input file and parsing its lines. Once this is done, there are many ways to solve the task. One of which is to use regular expressions. This approach was used and is combined right away with writing the extracted output to a file. The regular expression is quite simple, but requires some basic analysis of the input file. This script can be seen in Listing 6.8.

```
import re

lines = []
with open("supported.md", "r") as f:
    for line in f:
        lines.append(line)
with open("edited.md", "w") as o:
    for l in lines:
        o.write(re.search("(0x[A-F0-9]*)", l).group(1)+"\n")
```

Listing 6.8: Python 3 solution for experiment 1.

6.1.5 Linux tools solution

For this task there is no need for a processing pipeline. Excluding AWK solution, this task is great for tools, that work with regular expressions, such as Grep. It is possible to use the same expression that was used in the Python 3 solution. See the command in Listing 6.9.

```
grep -o "0x[A-F0-9]*" supported.md > edited.md
```

Listing 6.9: Linux tools (grep) solution for experiment 1.

6.1.6 Results and conclusions

Tool	**Editing time**
By hand (first 50 lines)	3 min 40 s
By hand (calculated from previous)	10 min 25 s
Python 3	4 min 52 s
Linux tools (Grep)	33 s
AWK	23 s
Ebel (handwritten)	50 s
Ebe	32 s

Table 6.2: Editing times for experiment 1.

Table 6.2 contains all editing times. Editing the whole file by hand in this case is viable as the number of lines is not too high and the edits for each line are quite simple, but still any other approach is faster and less prone to errors. Using Python 3 takes a long time mostly because the file has to be open, read from and then new file created and written to, these are tasks that all the other tools do for the programmer. Using Grep in this case is a viable option, since all the values adhere to the same format and the values can be easily extracted using a regular expression. AWK has showcased, that this task is the perfect task for it, since the values are well split with spaces and extraction only requires to count the correct column number and then print it. Handwriting an Ebel program is surprisingly fast in this case, but it requires the knowledge of Ebe's parser to correctly match instructions with lexemes.

Finally what Ebe takes the most time is to setup the examples, the overall compilation takes almost no time, because of the simple algorithm needed. Even though this approach took a little bit longer than using AWK and almost the same time as using Grep, it required no thinking about how to solve the editing task, but was all about doing the wanted edits by hand and then letting Ebe take care of the algorithm needed. This would allow someone with no programming skills and only basic knowledge of Ebe's usage to make edits as fast as someone with programming skills.

6.2 Experiment 2 – GTF file value adjustments

This example was already mentioned in this thesis, but since it is a use case from real world problem it seems like a good experiment as well. In this experiment the task is to edit snippet of GTF (Gene Transfer Format) file by subtracting the value 153350000 from 4th and 5th column, which are the feature start and end positions [1]. This is to be done over a file with 389 lines. Example of snippet of the file before and after the edits can be seen in the Listing 6.10 and 6.11. The input file is a snippet of larger file downloaded from the National Library of Medicine and it is a *Homo sapiens (human) genome assembly GRCh38 (hg38) from Genome Reference* made for the Human Genome Project[2].

[2]https://www.ncbi.nlm.nih.gov/assembly/GCF_000001405.26/

```
chr1    hg38_knownGene  exon    153357854   153357881   0.000000    +   .
      gene_id "ENST00000368738.4"; transcript_id "ENST00000368738.4";
chr1    hg38_knownGene  start_codon 153358284   153358286   0.000000    +
    .   gene_id "ENST00000368738.4"; transcript_id "ENST00000368738.4";
chr1    hg38_knownGene  CDS 153358284   153358433   0.000000    +   0
   gene_id "ENST00000368738.4"; transcript_id "ENST00000368738.4";
```

Listing 6.10: First 3 lines of the GTF file (`human.gtf`) before edits.

```
chr1    hg38_knownGene  exon    7854    7881    0.000000    +   .   gene_id
    "ENST00000368738.4"; transcript_id "ENST00000368738.4";
chr1    hg38_knownGene  start_codon 8284    8286    0.000000    +   .
   gene_id "ENST00000368738.4"; transcript_id "ENST00000368738.4";
chr1    hg38_knownGene  CDS 8284    8433    0.000000    +   0   gene_id "
   ENST00000368738.4"; transcript_id "ENST00000368738.4";
```

Listing 6.11: First 3 lines of the GTF file (`edited.gtf`) after edits.

6.2.1 Ebe solution

Solving this task with Ebe requires knowledge of user defined expression and in fact, it is the only part of Ebe compilation needed for this. The overall process is almost the same as in experiment 1. First line of the input file is copied into a new example input file (see Listing 6.12), this is then copied and the 4th and 5th column value is edited to contain expression `{! $ - 153350000 !}` (see Listing 6.13), which tells Ebe to subtract 153350000 from any value in this column. This can be then processed by Ebe using similar command as in experiment 1, but this time `-expr` or `--expression` option has to be used to parse the expression – `ebe -x -in f.in -out f.out -o edited.gtf human.gtf -expr`.

```
chr1    hg38_knownGene  exon    153357854   153357881   0.000000    +   .
      gene_id "ENST00000368738.4"; transcript_id "ENST00000368738.4";
```

Listing 6.12: Example input file (`f.in`) for Ebe solution.

```
chr1    hg38_knownGene  exon    {! $-153350000 !}   {! $-153350000 !}
   0.000000    +   .   gene_id "ENST00000368738.4"; transcript_id "
   ENST00000368738.4";
```

Listing 6.13: Example output file (`f.out`) for Ebe solution.

Attempt	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Edit time	60 s	50 s	48 s	46 s	46 s	40 s	42 s	42 s	41 s	39 s
Ebe time	<0.1 s									

Table 6.3: Edit times for all attempts solving experiment 2 with Ebe. Median value is 44 s.

6.2.2 Ebel solution

The Ebel solution requires the programmer to have knowledge of the Ebel expression pass syntax and semantics, but then it is very straightforward. For each word parsed a `NOP` instruction is written and then at the 4th column (which is not the 4th instruction, since delimiters are not ignored by Ebe parser) and 5th column an expression pass is written with the SUB instruction. The program can be seen in Listing 6.14 and notice that, when handwriting Ebel, $ can be used in place of $0 to make it more similar to the user defined expression.

```
Pass Words
  NOP
  NOP
  NOP
  NOP
  NOP
  NOP
  Pass NUMBER Expression
    SUB $, $, 153350000
    RETURN NOP
  NOP
  Pass NUMBER Expression
    SUB $, $, 153350000
    RETURN NOP
```

Listing 6.14: Handwritten Ebel solution (gtf.ebel) for experiment 2.

If it is not certain, if all the values are of type NUMBER, then it is needed to add “fall-through” expression passes to catch any other types. For example, if the start index could be FLOAT or the string “null”, which it is expected to change into 0 then the solution could look like the one in Listing 6.15. And even if the type was still different, all expression passes have implicit DERIVED expression pass, that will catch different type and execute NOP instruction; meaning, that if the type does not match the type in the expression nothing happens to the word and it is left as is.

```
Pass Words
  NOP
  NOP
  NOP
  NOP
  NOP
  NOP
  Pass NUMBER Expression
    SUB $, $, 153350000
    RETURN NOP
  Pass FLOAT Expression
    SUB $, $, 153350000.0
    RETURN NOP
  PASS "null" Expression
    MOVE $, 0
    RETURN NOP
  NOP
  Pass NUMBER Expression
    SUB $, $, 153350000
    RETURN NOP
  Pass FLOAT Expression
```

```
    SUB $, $, 153350000.0
    RETURN NOP
  PASS "null" Expression
    MOVE $, 0
    RETURN NOP
```

Listing 6.15: Handwritten Ebel solution (`gtf.ebel`) for experiment 2.

6.2.3 AWK solution

AWK allows to modify a column by assigning it a new value and then it's possible to print all the columns at once. Problem is, that unlike Ebe, AWK looses the original separators, which were tabs, but also spaces in some parts. It is possible to change the delimiter using `OFS` argument, but this will change all of the delimiters, and so, the output file differs a bit from the original, but it is still a valid GTF file.

```
awk -v OFS='\t' '{$4 = $4 - 153350000; $5 = $5 - 153350000; print $0}'
    human.gtf > edited.gtf
```

Listing 6.16: AWK solution for experiment 2.

6.2.4 Python 3 solution

The Python 3 solution once again contains mostly file handling procedures, but in this case it is possible to use the `csv` library to parse the GTF file, but using tab symbols instead of comma as the delimiter. There are also spaces used later on in the file, but since this is after the values, that are to be changed it does not affect the algorithm. Once the values are parsed it is needed to parse the string values into type `int`, subtract the value and then back to the `str` type to concatenate it with the rest.

If the programmer did not know how to use the `csv` library, it would be also possible to once again extract the values using regular expressions, edit them and then put the line back together using regular expressions for the parts excluding the edited values, although this would be quite overcomplicated solution. Alternatively it would be also possible to parse the file by hand using the `split` function and then using a similar process used in Listing 6.17.

```
import csv

outf = []
with open("human.gtf", "r") as f:
    reader = csv.reader(f, delimiter='\t')
    for row in reader:
        start = str(int(row[3])-153350000)
        end = str(int(row[4])-153350000)
        outf.append("\t".join(row[0:3]+[start]+[end]+row[5:]))
with open("edited.gtf", "w") as o:
    for l in outf:
        o.write(l+"\n")
```

Listing 6.17: Python 3 solution for experiment 2.

6.2.5 Results and conclusions

Tool	Editing time
By hand (first 50 lines)	13 min 14 s
By hand (calculated from previous)	102 min 57 s
Python 3	8 min 5 s
AWK	53 s
Ebel (handwritten)	1 min 39 s
Ebe	44 s

Table 6.4: Editing times for experiment 2.

In this experiment Linux tools were excluded, since the overall processing pipeline would be very inefficient and probably most people would rather use AWK. Python 3 takes longer time to write, since it is required to select the correct column, parse it, do the calculations and then construct the line back together. Ebel approach is viable since the code is quite straightforward and requires, once again, only to match correctly the words to ignore (use `NOP` instruction). AWK displays once again, that it is a powerful tool for file editing for the skilled users, but this time Ebe even managed to have lower editing time, because of the simple use of user defined expressions.

6.3 Experiment 3 – CSV file transformation

In this experiment a CSV file from P. Corez et al. [16] was used as the file for editing. This file has 1600 entries with wine ratings (quality) and their attributes. The task was to remove all the columns with acidity, meaning column 2, 3 and 4. First few lines from the file before and after edits can be seen in Listing 6.18 and 6.19.

This CSV file was selected since it is a file with many columns of the same or similar type. At the same time this data set contains some values placed in quote marks, which also make it harder for Ebe. Another complication for Ebe is the header, which is left in the file, since tools like Python 3 have means of easily working with it.

```
"","fixed.acidity","volatile.acidity","citric.acid","residual.sugar","chlorides","free.sulfur.dioxide","total.sulfur.dioxide","density","pH","sulphates","alcohol","quality"
"1",7.4,0.7,0,1.9,0.076,11,34,0.9978,3.51,0.56,9.4,5
"2",7.8,0.88,0,2.6,0.098,25,67,0.9968,3.2,0.68,9.8,5
```

Listing 6.18: First 3 lines of the CSV file (`data.csv`) before edits.

```
"","quality","residual.sugar","chlorides","free.sulfur.dioxide","total.sulfur.dioxide","density","pH","sulphates","alcohol"
"1",1.9,0.076,11,34,0.9978,3.51,0.56,9.4,5
"2",2.6,0.098,25,67,0.9968,3.2,0.68,9.8,5
```

Listing 6.19: First 3 lines of the CSV file (`edited.csv`) after edits.

6.3.1 Ebe solution

Solution in Ebe is for this case more complicated, because the input file contains a header, which has different structure then all the other lines. This would not be a problem, if the header had contained only one word tags for each column, but in this cased tags are comprised of multiple words divided by dots. This can be easily solved, by not using the header in the example, but using the second row for example (see Listings 6.20 and 6.21) and then, once the output is generated, the incorrect header has to be edited by hand to fit the data. The editing takes longer time, because of this whole procedure. On top of this, the compilation takes on average a longer time as well, because of the bigger number of columns and lexemes in each line.

```
"1",7.4,0.7,0,1.9,0.076,11,34,0.9978,3.51,0.56,9.4,5
```

Listing 6.20: Input example file for experiment 3.

```
"1",1.9,0.076,11,34,0.9978,3.51,0.56,9.4,5
```

Listing 6.21: Output example file for experiment 3.

Attempt	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Edit time	69 s	134 s	61 s	80 s	57 s	51 s	90 s	126 s	64 s	104 s
Compile time	16 s	87.6 s	12 s	30.6 s	0.3 s	5.2 s	47.6 s	85.5 s	24.4 s	60.4 s

Table 6.5: Edit times for all attempts solving experiment 3 with Ebe. Median value is 74.5 s.

6.3.2 Ebel solution

As with previous experiments, Ebel solution is again the same process of counting lexemes to leave and the ones to delete. The resulting program is quite simple and can be seen in Listing 6.22.

```
Pass Words
  NOP
  NOP
  NOP
  NOP
  DEL
  DEL
  DEL
  DEL
  DEL
  DEL
```

Listing 6.22: Ebel solution for experiment 3

6.3.3 AWK solution

To parse the input CSV file with AWK it is needed to specify the delimiter by using `-F ','` command line option, then AWK can address each column by its index and because there are no commas in the tags in the header, it will be correctly parsed as well. In this case it is possible to print the first column and follow this by a for loop printing all the other columns excluding 2nd, 3rd and 4th (see Listing 6.23).

```
awk -F ',' '{printf $1; for (i=5; i <= NF; i++) printf FS$i; print NL}'
    data.csv > edited.csv
```

Listing 6.23: AWK solution for experiment 3

6.3.4 Python 3 solution

Even though Python 3 offers a library for working with CSV files, in this case it is faster to use the `split` method for strings. This is because the CSV library also parses values and would remove the quotation marks from values that contain them.

The solution consists mostly of reading and writing to a file and the editing is done during writing to the file, where a list range selector is used to select only the 5th and later columns and these values are then connected into a string using the `join` method. The first column is simply prepended using a list selection operator. The program used can be seen in the Listing 6.25.

```
lines = []
with open("data.csv", "r") as f:
    lines = f.readlines()
with open("edited.csv", "w") as o:
    for l in lines:
        cols = l.split(",")
        o.write(cols[0]+","+",".join(cols[4:]))
```

Listing 6.24: Python 3 solution for experiment 3

6.3.5 Linux tools solution

This task can be solved multiple ways using Linux tools, but possibly one of the easiest ones is to use `sed` program, which can match regular expressions and then output them in defined format. The solution requires to create regular expressions for the 1st, 2nd, 3rd and 4th columns and for the rest as well, this can be simply done by using the `.*` pattern followed by comma, which will match the entirety of a column, since commas define each column. The rest is only about outputting just the 1st column followed by a comma and the last regular expression match.

```
sed -r 's/(.*),(.*),(.*),(.*),(.*)/\1,\5/g' data.csv > edited.csv
```

Listing 6.25: `sed` solution for experiment 3

6.3.6 Results and conclusions

Tool	Editing time
By hand (first 50 lines)	3 min 14 s
By hand (calculated from previous)	103 min 28 s
Python 3	1 min 24 s
Linux tools (sed)	45 s
AWK	56 s
Ebel (handwritten)	1 min
Ebe	1 min 15 s

Table 6.6: Editing times for experiment 3.

In this experiment, quite unexpectedly, all tools had almost the same editing time. In case of Ebe, most of the time was spent doing edits by hand, partially also because the file format required small tweaks after the compilation and interpretation. Other tools did not have problems with the file header, which can be associated with the different parsing used in these tools, where a delimiter is used, whereas Ebe bases its edits on position and the parsing creates different number of columns. This could be solved in Ebe, if a special CSV parser was introduced.

6.4 Experiment 4 – JSON file formatting

In this experiment the task is to remove new line symbols from JSON file, meaning that it will be concatenated into one continuous line. The input file is a JSON file with results from Ebe benchmarks with 462 lines. The first 3 line of the file can be seen in Listing 6.26 and the first 67 characters of the expected output can be seen in Listing 6.27.

```
{
"benchmark": {
"version": "1.0.0",
```

Listing 6.26: First 3 lines of the JSON file (`results.json`) before edits.

```
{"benchmark": {"version": "1.0.0","time:": 1644005755,"args": ""},
```

Listing 6.27: First 67 characters of the output JSON file (`edited.json`) after edits.

6.4.1 Ebe solution

This task requires the use of Ebel's `Pass Lines` and `CONCAT` instruction, but this also requires the need to provide long enough example, so that Ebe understands, that the concatenation is to be done over all lines and not just first two or three lines. For this reason, the example used is 5 lines long (see Listings 6.28 and 6.29). In this example there is always the chance of Ebe generating 5 `CONCAT 1` instructions, but the chance is way lower than the chance for generating `CONCAT 1` followed by `LOOP` instruction. Using chances from Table 5.2 and that the instruction argument is picked based on minimum and maximum meaningful value (in this case only arguments 1 to 5 would be executed), it can be calculated, that the chance for Ebe to generate 5 `CONCAT 1` instructions is:

$$p(\text{five concat}) = \left(0.125 \cdot \frac{1}{5}\right)^5 \approx 0.000001\%, \tag{6.1}$$

whereas the chance to generate `CONCAT 1` instruction followed by `LOOP` is:

$$p(\text{concat loop}) = \left(0.125 \cdot \frac{1}{5}\right) \cdot 0.125 = 0.3125\%, \tag{6.2}$$

if the example would contain only 2 lines, then the chance for generating 2 `CONCAT 1` instructions in a pass would be:

$$p(\text{two concat}) = \left(0.125 \cdot \frac{1}{5}\right)^2 = 0.0625\%. \tag{6.3}$$

Equations 6.1, 6.2 and 6.3 are not exactly what happens during the phenotype generation and evolution, since there are also other factors, such as the chance for initial phenotype size, chance for lines pass or even chance for mutation and crossover later during evolution. But these calculations are here to give a rough estimation on the question of which is more likely. All of the programs with the mentioned probabilities are correct based on the given example, but based on the task only one (`CONCAT 1` followed by `LOOP`) is correct and choosing bigger example will make sure, that the chance for the wanted program is more likely, than the other one.

```
{
"benchmark": {
"version": "1.0.0",
"time:": 1644005755,
"args": ""
},
```

Listing 6.28: Input example file for experiment 4.

```
{"benchmark": {"version": "1.0.0","time:": 1644005755,"args": ""},
```

Listing 6.29: Output example file for experiment 4.

Attempt	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Edit time	45 a	36 s	32 s	32 s	29 s	30 s	30 s	30 s	29 s	28 s
Compile time	<0.0 s	0.2 s	0.2 s	0.2 s	0.2 s	0.2 s	0.1 s	0.1 s	0.2 s	<0.0 s

Table 6.7: Edit times for all attempts solving experiment 4 with Ebe. Median value is 30 s.

6.4.2 Ebel solution

Solution in Ebel is very simple, since the task is repetitive concatenation of lines. Unlike in previous experiments, in this experiment it is needed to start with `Pass Lines`, because the implicit pass is `Pass Words`, then the program is just a `CONCAT 1` instruction and `LOOP` instruction (see Listing 6.30) to apply this instruction to all lines.

```
Pass Lines
  CONCAT 1
  LOOP
```

Listing 6.30: Ebel solution for experiment 4.

6.4.3 AWK solution

AWK allows to set the character, by which each line ends. This character is held by the `ORS` variable and it is the new line symbol by default. Changing this variable to an empty string and printing all lines (see Listing 6.31) removes the new line symbol from each line.

```
awk -v ORS="" '{print $0}' results.json > edited.json
```

Listing 6.31: AWK solution for experiment 4.

6.4.4 Python 3 solution

Python 3 contains a method `strip`, which can be used to remove new line symbols from each line, but as was in the previous experiments, it is first needed to read the file and then write to a new one. To apply the `strip` method to all lines a list comprehension can be used (see Listing 6.32).

```
lines = []
with open("results.json", "r") as f:
    lines = f.readlines()
with open("edited.json", "w") as o:
    o.write("".join([x.strip() for x in lines]))
```

Listing 6.32: Python 3 solution for experiment 4.

6.4.5 Linux tools solution

This experiment can be in Linux solved using the `tr` program, which can remove certain characters from a string – in this case the new line character. To load the file into the program `cat` can be used. This process pipeline can be seen in Listing 6.33.

```
cat results.json | tr -d '\n' > edited.json
```

Listing 6.33: `cat` and `tr` solution for experiment 4.

6.4.6 Results and conclusions

Tool	**Editing time**
By hand (first 50 lines)	42 s
By hand (calculated from previous)	6 min 28 s
Python 3	59 s
Linux tools (cat and tr)	15 s
AWK	21 s
Ebel	17 s
Ebe	30 s

Table 6.8: Editing times for experiment 3.

Most tools have proven to be quite efficient and fast at completing the given task and their times are very similar. Python 3 requires additional file operations, which have to be done by the user and this adds some additional time to writing of the script. AWK requires more advanced knowledge of the tool, where the programmer has to be aware of the existence of `ORS` variable. Using `tr` has proven to be the fastest and possibly most straightforward approach, since this task is exactly what the tool is meant for (removing characters or replacing them). Ebel contains instruction just for this task, and thus, the solution is quite straightforward and could be written even by less skilled programmer. Ebe requires the user to know, that giving a bigger example in this case helps with the ambiguity, but even if not provided, if the user was to check the output, he or she could then try running Ebe once again or multiple times until Ebe finds the solution, which the user wants.

Chapter 7

Conclusion

Ebe was also presented at the Excel@FIT 2022 conference, where it received expert committee award and AT&T award. It was also picked by the committee to be one of the 8 projects, that were presented at the conference.

The goals of this thesis laid down were to implement a tool for automated file editing from given user examples, that would use genetic programming to achieve this task. This was accomplished as well as analysis of the tools performance and the evolutionary approach itself. On top of this, the tool has proven to be a viable alternative to competitive editing tools. Ebe, on top of its speed, does not require any programming knowledge or deep understanding of the tool, that many of the competitors do. The implemented program is not a universal tool, but rather a tool for a subset of editing problems, but these are the problems, which are quite often needed to be automatized. The use of genetic programming for the task of file editing has also been quite successful, because of the new language designed specifically for this task. The language itself, with the implemented interpreter, have potential to be a useful resource for other projects that deal with file editing. Even though there are many difficult editing tasks, for which this approach is not suitable, the simpler edits, which are quite common editing tasks, are not a problem and Ebe handles them quickly. Additionally, the tool has been made to be build upon, extended and optimized and many of the tasks, which are not possible or are not optimal in the current state of the tool, could be fixed with more optimizations focusing on these exact problems. Genetic programming and more generally evolutionary approach has proven to be an approach with a lot of potential to tackle many different tasks. There are some underlying constraints to this approach, but this method showcases other great strengths, such as low computing requirements or no previous learning process requirements.

Regarding future of this project, there are many parts of it, which could be extended, tested or changed. Ebe showcases some potential to address even more complex editing problems and the same stands for the language Ebel. Ebel in the current state contains only the limited set of instructions to do the most common editing tasks, but it could be extended to contain even more complex instructions or, for instance, instructions for generating values, rather than just modifying the existing ones. Although such addition would require additional changes and optimization of existing methods, since it could effect editing times for the current tasks it can handle. The Ebel language could be also extended for the use in other file editing projects, such as aforementioned Bee language, and become a full-fledged language for editing files with many specialized instructions.

Bibliography

[1] *GFF/GTF File Format.* [cit. 26. April 2022]. Definition and supported options. Available at: https://www.ensembl.org/info/website/upload/gff.html.

[2] *Prose framework.* Apr 2022. Available at: https://www.microsoft.com/en-us/research/project/prose-framework/.

[3] Aho, A. V., Kernighan, B. W. and Weinberger, P. J. *The AWK Programming Language.* USA: Addison-Wesley Longman Publishing Co., Inc., 1987. ISBN 020107981X.

[4] Arnaldo, I., Krawiec, K. and O'Reilly, U.-M. Multiple Regression Genetic Programming. In: *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation.* New York, NY, USA: Association for Computing Machinery, 2014, p. 879–886. GECCO '14. DOI: 10.1145/2576768.2598291. ISBN 9781450326629. Available at: https://doi.org/10.1145/2576768.2598291.

[5] Backurs, A. and Indyk, P. Edit Distance Cannot Be Computed in Strongly Subquadratic Time (unless SETH is false). *CoRR.* 2014, abs/1412.0348. Available at: http://arxiv.org/abs/1412.0348.

[6] Barowy, D. W., Gulwani, S., Hart, T. and Zorn, B. FlashRelate: Extracting Relational Data from Semi-Structured Spreadsheets Using Examples. *SIGPLAN Not.* New York, NY, USA: Association for Computing Machinery. jun 2015, vol. 50, no. 6, p. 218–228. DOI: 10.1145/2813885.2737952. ISSN 0362-1340. Available at: https://doi.org/10.1145/2813885.2737952.

[7] Berger, B., Waterman, M. S. and Yu, Y. W. Levenshtein Distance, Sequence Comparison and Biological Database Search. *IEEE Transactions on Information Theory.* 2021, vol. 67, no. 6, p. 3287–3294. DOI: 10.1109/TIT.2020.2996543.

[8] Bird, J., Husbands, P., Perris, M., Bigge, B. and Brown, P. Implicit Fitness Functions for Evolving a Drawing Robot. In: *Proceedings of the 2008 Conference on Applications of Evolutionary Computing.* Berlin, Heidelberg: Springer-Verlag, 2008, p. 473–478. Evo'08. ISBN 3540787607.

[9] Boehm, H. j. and Spertus, M. *Transparent Programmer-Directed Garbage Collection for C.* 2007.

[10] Braione, P., Denaro, G., Pezze, M. and Křena, B. Verifying LTL Properties of Bytecode with Symbolic Execution. In: *In proceedings of Third Workshop on Bytecode Semantics, Verification, Analysis and Transformation.* Elsevier Science B. V, 2008. BYTECODE 2008.

[11] Cahyono, S. Comparison of document similarity measurements in scientific writing using Jaro-Winkler Distance method and Paragraph Vector method. *IOP Conference Series: Materials Science and Engineering*. november 2019, vol. 662, p. 052016. DOI: 10.1088/1757-899X/662/5/052016.

[12] Canonical. *Ubuntu Manpage: AWK - pattern scanning and processing language*. [cit. 20. February 2022]. http://manpages.ubuntu.com/manpages/trusty/man1/awk.1posix.html.

[13] Castelli, M., Trujillo, L., Vanneschi, L., Silva, S., Z Flores, E. et al. Geometric Semantic Genetic Programming with Local Search. In: *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*. New York, NY, USA: Association for Computing Machinery, 2015, p. 999–1006. GECCO '15. DOI: 10.1145/2739480.2754795. ISBN 9781450334723. Available at: https://doi.org/10.1145/2739480.2754795.

[14] Consortium, T. U. *The Unicode® Standard Version 12.0 – Core Specification*. 2019. ISBN 978-1-936213-22-1. Available at: http://www.unicode.org/versions/Unicode12.0.0/.

[15] Cooper, K. D., Schielke, P. J. and Subramanian, D. Optimizing for Reduced Code Space Using Genetic Algorithms. In: *Proceedings of the ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems*. New York, NY, USA: Association for Computing Machinery, 1999, p. 1–9. LCTES '99. DOI: 10.1145/314403.314414. ISBN 1581131364. Available at: https://doi.org/10.1145/314403.314414.

[16] Cortez, P., Cerdeira, A., Almeida, F., Matos, T. and Reis, J. Modeling wine preferences by data mining from physicochemical properties. *Decision Support Systems*. 2009, vol. 47, no. 4, p. 547–553. DOI: https://doi.org/10.1016/j.dss.2009.05.016. ISSN 0167-9236. Smart Business Networks: Concepts and Empirical Evidence. Available at: https://www.sciencedirect.com/science/article/pii/S0167923609001377.

[17] Darwin, C. *On the Origin of Species by Means of Natural Selection*. London: Murray, 1859. Or the Preservation of Favored Races in the Struggle for Life.

[18] Dumitrescu, D., Lazzerini, B., Jain, L. C. and Dumitrescu, A. *Evolutionary Computation*. USA: CRC Press, Inc., 2000. ISBN 0849305888.

[19] Ferrel, W. and Alfaro, L. Genetic Programming-Based Code Generation for Arduino. *International Journal of Advanced Computer Science and Applications*. The Science and Information Organization. 2020, vol. 11, no. 11. DOI: 10.14569/IJACSA.2020.0111168. Available at: http://dx.doi.org/10.14569/IJACSA.2020.0111168.

[20] Forrest, S., Nguyen, T., Weimer, W. and Le Goues, C. A Genetic Programming Approach to Automated Software Repair. In: *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*. New York, NY, USA: Association for Computing Machinery, 2009, p. 947–954. GECCO '09. DOI: 10.1145/1569901.1570031. ISBN 9781605583259. Available at: https://doi.org/10.1145/1569901.1570031.

[21] Friedl, J. E. F. *Mastering Regular Expressions.* 3rd ed. O'Reilly, 2006. ISBN 978-0-596-52812-6.

[22] Gajda, Z. and Sekanina, L. Gate-Level Optimization of Polymorphic Circuits Using Cartesian Genetic Programming. In: *Proceedings of the Eleventh Conference on Congress on Evolutionary Computation.* IEEE Press, 2009, p. 1599–1604. CEC'09. ISBN 9781424429585.

[23] Goldberg, D. E. and Deb, K. A Comparative Analysis of Selection Schemes Used in Genetic Algorithms. In: RAWLINS, G. J., ed. Elsevier, 1991, vol. 1, p. 69–93. Foundations of Genetic Algorithms. DOI: https://doi.org/10.1016/B978-0-08-050684-5.50008-2. ISSN 1081-6593. Available at: https://www.sciencedirect.com/science/article/pii/B9780080506845500082.

[24] Gougol, R. *Triggering of Just-In-Time Compilation in the Java Virtual Machine.* San Jose State University, 2009. Master's thesis.

[25] Griswold, R. E. and Griswold, M. T. *The Implementation of the Icon Programming Language.* USA: Princeton University Press, 1986. ISBN 0691084319.

[26] Hadley, G. G. *Nonlinear and dynamic programming / by G. Hadley.* Reading, Mass: Addison-Wesley Pub. Co., 1964. Addison-Wesley series in management science and economics.

[27] Hamming, R. W. Error detecting and error correcting codes. *The Bell System Technical Journal.* 1950, vol. 29, no. 2, p. 147–160. DOI: 10.1002/j.1538-7305.1950.tb00463.x.

[28] Holland, J. H. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence.* Cambridge, MA, USA: MIT Press, 1992. ISBN 0262082136.

[29] Horáček, P., Meduna, A. and Tomko, M. *Handbook of Mathematical Models for Languages and Computation.* The Institution of Engineering and Technology, 2020. 750 p. ISBN 978-1-78561-659-4. Available at: https://www.fit.vut.cz/research/publication/12041.

[30] Iba, H., Feng, J. and Izadi Rad, H. GP-RVM: Genetic Programing-Based Symbolic Regression Using Relevance Vector Machine. In: *2018 IEEE International Conference on Systems, Man, and Cybernetics (SMC).* 2018, p. 255–262. DOI: 10.1109/SMC.2018.00054.

[31] Jaro, M. A. Advances in Record-Linkage Methodology as Applied to Matching the 1985 Census of Tampa, Florida. *Journal of the American Statistical Association.* [American Statistical Association, Taylor & Francis, Ltd.]. 1989, vol. 84, no. 406, p. 414–420. ISSN 01621459. Available at: http://www.jstor.org/stable/2289924.

[32] Joint Technical Committee ISO/IEC JTC 1. *International Standard ISO/IEC 9899.* 2007. Available at: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf.

[33] Kernighan, B. W. and Ritchie, D. M. *The C Programming Language.* 2ndth ed. Prentice Hall Professional Technical Reference, 1988. ISBN 0131103709.

[34] Koza, J. R. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* Cambridge, MA, USA: MIT Press, 1992. ISBN 0-262-11170-5.

[35] Koza, J. R. Genetic programming - on the programming of computers by means of natural selection. In: *Complex adaptive systems.* 1993.

[36] Langdon, W. B. and Banzhaf, W. Repeated Patterns in Tree Genetic Programming. In: *Proceedings of the 8th European Conference on Genetic Programming.* Berlin, Heidelberg: Springer-Verlag, 2005, p. 190–202. EuroGP'05. DOI: 10.1007/978-3-540-31989-4_17. ISBN 3540254366. Available at: https://doi.org/10.1007/978-3-540-31989-4_17.

[37] Le, V. and Gulwani, S. FlashExtract: A Framework for Data Extraction by Examples. *SIGPLAN Not.* New York, NY, USA: Association for Computing Machinery. jun 2014, vol. 49, no. 6, p. 542–553. DOI: 10.1145/2666356.2594333. ISSN 0362-1340. Available at: https://doi.org/10.1145/2666356.2594333.

[38] Levenshtein, V. I. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady.* 1966, vol. 10, no. 8, p. 707–710. Doklady Akademii Nauk SSSR, V163 No4 845-848 1965.

[39] Levine, J. R., Mason, T. and Brown, D. *Lex & Yacc (2nd Ed.).* USA: O'Reilly & Associates, Inc., 1992. ISBN 1565920007.

[40] Libbrecht, P. Notations Around the World: Census and Exploitation. In: *International Conference on Intelligent Computer Mathematics.* April 2010. ISBN 978-3-642-14128-7.

[41] Lindholm, T., Yellin, F., Bracha, G. and Buckley, A. *The Java Virtual Machine Specification, Java SE 8 Edition.* Pearson Education, 2014. ISBN 9780133922721.

[42] Matić, D. A genetic algorithm for composing music. *Yugoslav Journal of Operations Research.* january 2010, vol. 20. DOI: 10.2298/YJOR1001157M.

[43] Merelo Guervós, J., Blancas Álvarez, I., Castillo, P., Romero, G., García Sánchez, P. et al. Ranking the Performance of Compiled and Interpreted Languages in Genetic Algorithms. In: *In Proceedings of the 8th International Joint Conference on Computational Intelligence – ECTA.* January 2016, p. 164–170. DOI: 10.5220/0006048101640170.

[44] Microsystems, S. The Java HotSpot TM Virtual Machine Technical White Paper. In:. 2001.

[45] Miller, J. F. Cartesian Genetic Programming. In: *Natural Computing Series book series (NCS).* June 2003, vol. 43. DOI: 10.1007/978-3-642-17310-3. ISBN 978-3-642-17309-7.

[46] Molina, E. On the analytical demonstration of Planck-Einstein relation. Technological University of Madrid. september 2016. DOI: 10.13140/RG.2.2.35544.39689.

[47] Nacke, K. *Learn LLVM 12: A Beginner's Guide to Learning LLVM Compiler Tools and Core Libraries with C++*. Packt Publishing, 2021. ISBN 9781839213502. Available at: https://books.google.cz/books?id=5atJzgEACAAJ.

[48] Neil, D. *Practical Vim: Edit Text at the Speed of Thought*. 2ndth ed. Pragmatic Bookshelf, 2015. ISBN 1680501275.

[49] Nystrom, R. *Crafting Interpreters*. Genever Benning, 2021. ISBN 9780990582939. Available at: https://books.google.cz/books?id=ySOBzgEACAAJ.

[50] O'Reilly, U.-M. Genetic Programming II: Automatic Discovery of Reusable Programs. *Artificial Life*. 1994, vol. 1, no. 4. DOI: 10.1162/artl.1994.1.4.439.

[51] Polozov, O. *A Framework for Mass-Market Inductive Program Synthesis*. Dissertation.

[52] Ramalho, L. *Fluent Python: Clear, Concise, and Effective Programming*. O'Reilly Media, 2015. ISBN 9781491946251. Available at: https://books.google.co.in/books?id=bIZHCgAAQBAJ.

[53] Shaw, A. *CPython Internals: Your Guide to the Python 3 Interpreter*. Real Python (Realpython.Com), 2021. ISBN 9781775093343. Available at: https://books.google.cz/books?id=fKZwzgEACAAJ.

[54] Shelton, J. M. and Brown, S. J. Fasta-O-Matic: a tool to sanity check and if needed reformat FASTA files. *BioRxiv*. Cold Spring Harbor Laboratory. 2015. DOI: 10.1101/024448. Available at: https://www.biorxiv.org/content/early/2015/08/21/024448.

[55] Shi, X., Long, W., Li, Y. and Deng, D. Multi-population genetic algorithm with ER network for solving flexible job shop scheduling problems. *PLOS ONE*. Public Library of Science (PLoS). may 2020, vol. 15, no. 5, p. e0233759. DOI: 10.1371/journal.pone.0233759. Available at: https://doi.org/10.1371/journal.pone.0233759.

[56] Soanes, C. and Stevenson, A., ed. *Oxford dictionary of English*. London, England: Oxford University Press, 2005.

[57] Spector, L. *Automatic Quantum Computer Programming: A Genetic Programming Approach (Genetic Programming)*. Berlin, Heidelberg: Springer-Verlag, 2006. ISBN 038736496X.

[58] Steiner, R. History and progress on accurate measurements of the Planck constant. *Reports on Progress in Physics*. IOP Publishing. december 2012, vol. 76, no. 1, p. 016101. DOI: 10.1088/0034-4885/76/1/016101. Available at: https://doi.org/10.1088/0034-4885/76/1/016101.

[59] Stroustrup, B. Foundations of C++. In: Seidl, H., ed. *Programming Languages and Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, p. 1–25. ISBN 978-3-642-28869-2.

[60] Stroustrup, B. *The C++ Programming Language*. 4thth ed. Addison-Wesley Professional, 2013. ISBN 0321563840.

[61] Torczon, L. and Cooper, K. *Engineering A Compiler*. 2ndth ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007. ISBN 012088478X.

[62] Wang, Y. and Xie, R. Pixel-Based Approach for Generating Original and Imitating Evolutionary Art. *Electronics*. 2020, vol. 9, no. 8. DOI: 10.3390/electronics9081311. ISSN 2079-9292. Available at: https://www.mdpi.com/2079-9292/9/8/1311.

[63] Winkler, W. String Comparator Metrics and Enhanced Decision Rules in the Fellegi-Sunter Model of Record Linkage. *Proceedings of the Section on Survey Research Methods*. january 1990.

[64] Zhao, H., Gallo, O., Frosio, I. and Kautz, J. *Loss Functions for Neural Networks for Image Processing*. arXiv, 2015. DOI: 10.48550/ARXIV.1511.08861. Available at: https://arxiv.org/abs/1511.08861.

Appendix A

Project file structure

A.1 Ebe file structure

Ebe is organized into multiple directories for easier navigation and working with the project. The project structure is as follows:

- `backend/`
  - `compiler` – base compiler class,
  - `instruction` – Ebel instruction implementation,
  - `interpreter` – Ebei implmenetation,
  - `symbol_table` – symbol table implementation used by Ebei.
- `docs/` – Contains Doxyfile and images used in `readme.md`. Doxygen output is saved into this folder into `html` subfolder.
- `engine/`
  - `engine_jenn` – engine Jenn implementation,
  - `engine_miRANDa` – engine MiRANDa implementation,
  - `engine_taylor` – engine Taylor implementation,
  - `engine` – base engine and genetic engine class.
- `frontend/`
  - `grammars/`
    - `lexer_ebel.ll` – Flex grammar file for Ebel lexer,
    - `lexer_text.ll` – Flex grammar file for text file lexer,
    - `parser_ebel.yy` – Bison grammar file for Ebel parser,
    - `parser_text.yy` – Bison grammar file for text file parser.
  - `FlexLexer.h` – Flex's header file created by The Regents of the University of California (more information regarding the copyright of this file can be find in the file's header comment),
  - `lexel_ebel` – Ebel lexer generated by Flex from `lexer_ebel.ll`,
  - `lexer_text` – Text file lexer generated by Flex from `lexer_text.ll`,

  - `parser_ebel` – Ebel parser generated by Bison from `parser_ebel.yy` (more information regarding the copyright of this file can be find in the file's header comment),
  - `parser_text` – Text file parser generated by Bison from `parser_text.yy` (more information regarding the copyright of this file can be find in the file's header comment),
  - `pragmas` – Ebel pragma parser and pragma class implementation,
  - `preprocessor` – text file and Ebel program preprocessor,
  - `scanner_ebel` – implementation of Ebel code scanner and parser, which creates the internal Ebel representation from input Ebel source code,
  - `scanner_text` – implementation of text file scanner and parser, which creates the internal representation from provided text file,
  - `scanner` – base scanner class,

- `gp/`
  - `fitness` – implementations of fitness functions,
  - `gp` – implementation of certain genetic programming objects (phenotypes and population).

- `midend/`
  - `expression` – structures for working with user defined expressions,
  - `ir` – implementations for internal representations and its accompanying constructs,
  - `tree` – expression tree implementation.

- `tests/`
  - `googletest/` – Googletests library for unit testing with copyright belonging to its rightful owners (see `LICENSE` file inside of this folder),
  - `test_argparse` – unit tests for Ebe argument parsing,
  - `test_interpreter` – unit tests for Ebei,
  - `test_pragmas` – unit tests for Ebel pragmas parsing,
  - `test_scanner` – unit tests for Ebe's scanners,
  - `test_utils` – unit tests for general utility functions.

- `utils/`
  - `arg_parser` – Ebe's argument parsing with custom implementation to provide descriptive information and error messages,
  - `exceptions` – custom C++ exceptions utilized mostly by Ebei,
  - `logging` – Ebe's logging utilities,
  - `rng` – module for working with random number generation and randomization,
  - `utils` – generally useful utility functions.

- `CMakeLists.txt` – CMake compilation script.
- `ebe` – Ebe's entry point. Contains `main` function and executes Ebe's processes.
- `install.sh` – Installation script.
- `LICENCE` – MIT license under which Ebe is distributed.
- `readme.md` – Informal project documentation for Ebe users containing installation information, simple examples and some general Ebe information.

A.2 Ebe tools file structure

Ebe tools (`ebe-tools`) contain scripts for benchmarking, plotting benchmark results and other analytical scripts. The file structure is as follows:

- `analytics` – contains script `plot.py` for plotting Ebe's analytics output (`-a` command line option) as well as bash script to calculate code analytics.
- `benchmarks`
  - `ebe_all/` – benchmarks usable for Ebec and also Ebei,
  - `ebec/` – Ebec benchmark tests,
  - `ebei/` – Ebei benchmark tests,
  - `limit_tests` – additional benchmarks for Ebec (in `ebec/`) and Ebei (in `ebei`), which Ebe cannot 100 % solve, but are here to limit test Ebe,
  - `benchmark.py` – benchmarking script,
  - `measure.sh` – Bash script used by benchmarks to measure Ebe's run time,
  - `one_core.sh` – Bash script used by benchmarks to run a process with maximum priority on a single CPU core (this approach was inspired by an article from Eric Urban[1]),
  - `plot_benchmarks.py` – script to plot benchmark results,
  - `readme.md` – informal information for ebe-tools users.
- `readme.md` – informal information for ebe-tools users.

[1] http://www.hydrogen18.com/blog/howto-give-a-single-process-its-own-cpu-core-in-linux.html

Appendix B

Benchmark results

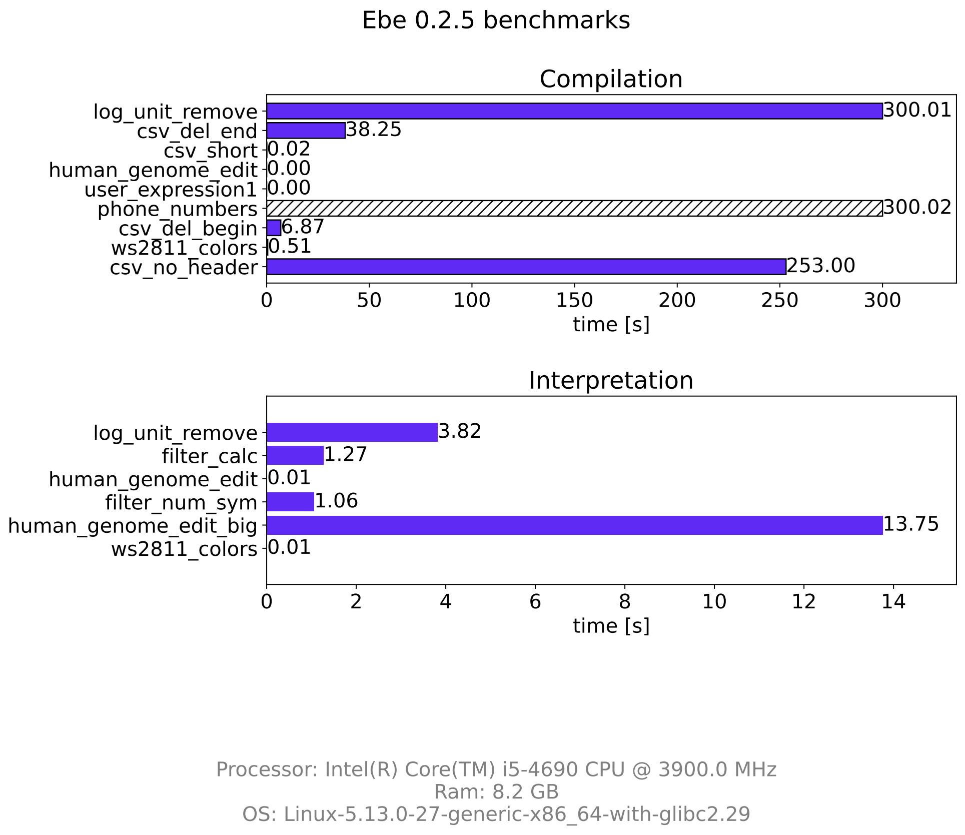
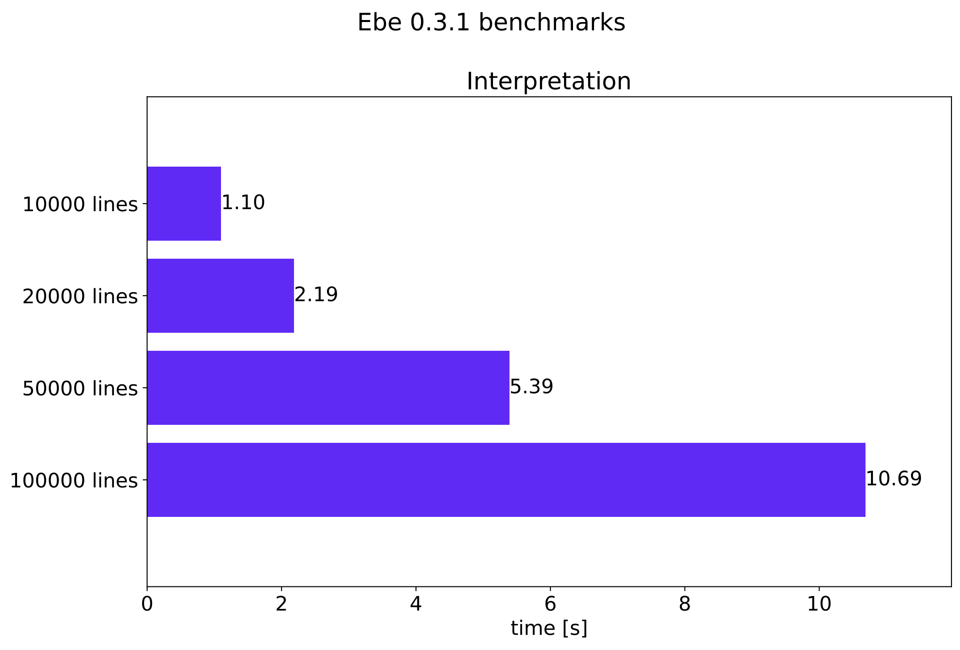


Figure B.1: Benchmarks results plotted using plotting script.



Processor: Intel(R) Core(TM) i5-4690 CPU @ 3900.0 MHz
Ram: 8.2 GB
OS: Linux-5.13.0-39-generic-x86_64-with-glibc2.29

Figure B.2: Benchmarks results for Ebe's interpretation speed using the Ebel program for benchmark `filter_num_sym` (see Section 5.1). The input file is the same as for the original benchmark, but the number of lines changes for each run. This graph showcases Ebe's scalability based on input file size.

Appendix C

Manual

C.1 Ebe installation

Ebe was tested and is known to work on Linux systems and FreeBSD systems. It was successfully compiled with compilers GCC 9.4.0 and higher, Clang 10.0.0 and higher. The compiler has to support C++17, for example GCC before version 8 will not succeed since the library `<filesystem>` does not appear and is placed in `<experimental/filesystem>`. It is also recommended to have `CMake` installed to make use of the already prepared `CMakeLists.txt` and possibly even the installation script. It is also possible to generate Ebe's internal documentation with `Doxygen` using `CMake` or provided `Doxyfile`, the `Doxyfile` is configured to generate graphs using `dot` program and so it is also required. If one was to change the parser of lexer, then `Flex` (tested was version 2.6.4) and `Bison` (minimum required version is 3.7.0).

But to compile Ebe just for the use, only requirement is the C++ compiler (with full support for C++17) and preferably `CMake`. To build Ebe do the following steps:

1. Copy Ebe's folder to your PC, alternatively the newest version can be downloaded from Github: `git clone https://github.com/mark-sed/ebe.git`
2. Change directory into `ebe`: `cd ebe`
3. Run the installation script, if sudo is used, then ebe is placed as a command into `/usr/bin/` to use from anywhere: `sudo bash install.sh`

```
git clone https://github.com/mark-sed/ebe.git  # Optional step to get the
    newest version
cd ebe
sudo bash install.sh
```

Listing C.1: Ebe installation.

After running these command Ebe will be built into `build` folder and if run with root privileges also found under `ebe` command. To test correctness one can run `./build/ebe --help` or `ebe --help`.

```
git clone https://github.com/mark-sed/ebe.git  # Optional step to get the
    newest version
cd ebe
cmake -DCMAKE_BUILD_TYPE=Release -S . -B build
cmake --build build --target ebe
```

Listing C.2: Alternative Ebe installation using only CMake.

C.2 Simple edit example to test Ebe

In this example there are multiple files greeting and saying goodbye to different worlds and the task is to extract only names of these worlds.

1. Setting up files to edit

```
Hello Earth world!
Hello Pluto world!
Hello Moon world!
Hello Kamino world!
Hello Calantha world!
```

Listing C.3: File `hellos.txt`.

```
Goodbye Arda world!
Goodbye Gliese world!
Goodbye Eternium world!
```

Listing C.4: File `goodbyes.txt`.

2. Creating examples for Ebe

```
Hello Earth world!
```

Listing C.5: File `example.in` containing input example – line before the edits (extracted first line from `hellos.txt`).

```
Earth
```

Listing C.6: File `example.out` containing output example – line after the edits.

3. Editing using Ebe

Ebe can be now run passing it the examples. The example input file should be passed using `-in` command line option, the example output file with `-out` and to do also the interpretation in the same run `-x` can be used. Since there are multiple files to edit at once the `-o` command line option should contain path to a folder, where to save the edited files, which names will have the prefix `edited-`. The files to interpret (`hellos.txt` and `goodbyes.txt`) do not require any command line option and can be passes just by path (see listing C.7).

```
ebe -x -in example.in -out example.out hellos.txt goodbyes.txt -o .
```

Listing C.7: File `example.out` containing output example – line after the edits.

After running command from listing C.7 the Ebe should output something similar to the output in listing C.8.

```
Perfectly fitting program found.

Best compiled program has 100% precision (0.0 s).
```

Listing C.8: File `example.out` containing output example – line after the edits.

4. Checking the output

Output from editing will be in current directory, since `-o .` was used, under the same file names with prefix `edited-` (`edited-hellos.txt` and `edited-goodbyes.txt`).

```
Earth
Pluto
Moon
Kamino
Calantha
```

Listing C.9: Expected contents of file `edited-hellos.txt`.

```
Arda
Gliese
Eternium
```

Listing C.10: Expected contents of file `edited-goodbyes.txt`.

C.3 Generating Doxygen documentation

The Doxygen documentation can be easily generated from within the `ebe` directory by running:

```
doxygen docs/Doxyfile
```

Listing C.11: Generating Doxygen documentation for Ebe.

The generated documentation will be placed into `docs/html` folder and can be viewed by opening `docs/html/index.html` file.

C.4 Ebe command line arguments

-	--	Argument	Description	Default value	Mode	Forbidden with	Multiple
-x	--execute		Sets mode to C&I (compiles and interprets in one run)		None		Ignored
-in	--example-input	File path	Example input file		C, C&I		Not yet
-out	--example-output	File path	Example output file		C, C&I		Not yet
-i	--interpret	Ebel file path	Interpret Ebel		I		No
-eo	--ebel-output	Ebel file path	Path to output Ebel code	<-in file base name>.ebel	I, C&I		No
-o	--interpret-output	File path or folder path	Path to the where to save the interpreted output	edited-<-in file name>for multiple	I, C&I		No
		File path	Input file for interpretation	*standard input*	I, C&I		Yes
-expr	--expressions		Parse user expressions		C, C&I		Ignored
-it	--iterations	Integer >0	Number of iterations in each evolution	Dynamically picked by an engine	C, C&I		No
	--population-size	Integer >0	Size of population for GP engines	Dynamically picked by an engine	C, C&I		No
-e	--evolutions	Integer >0	Number of evolutions to be done	Dynamically picked by an engine	C, C&I	-p	No
-E	--engine	Engine name	Engine to use for compilation	Jenn	C, C&I		No
-f	--fitness	Fitness function name	Fitness function for compilation	jaro	C, C&I		No
-p	--precision	Integer <1..100>or none (=100)	Minimum required compilation precision (-t has priority)	C: Ignored; C&I: 100	C, C&I	-e	No
-t	--timeout	Integer >0	After how many seconds at most should compilation stop	Ignored	C, C&I		No
	--version		Ebe's version		None	All	No
-h	–help		Usage information		None	All	No
-v<v/1..5>		CSV list of functions (file.cpp::funtion)	Sets logging level (-vvv is the same as -v3)	Ignored	All		No
-a	--analytics	CSV list of analytic unit names	Enables analytics logs for set units	Ignored	All		No
-aout	--analytics-output	Folder path	Analytics output directory	./	All		No
	--seed	Integer <1..2^32-1>	Seed for RNG	Ignored	C, C&I		No
	--sym-table-size	Positive integer	Sets the size of symbol table	64	I, C&I		No
	--no-warn-print		Turns off warning messages prints		All		Ignored
	--no-error-print		Turns off error message prints		All		Ignored
	--no-info-print		Turns off info messages		All		Ignored

There are 3 modes:

- Compile (C) – Only compile Ebel program.
- Interpret (I) – Only interpret given Ebel program over input files.
- Compile and interpret (C&I) – Compile Ebel program and interpret it over input files.

C.5 Running Ebe benchmarks

Ebe benchmarks reside in `ebe-tools/benchmarks` folder. To run provided default benchmarks, the `benchmark.py` script has to be run from within the `ebe-tools/benchmarks` folder. To run benchmarks with default parameters the following can be used:

```
python3 benchmark.py
```

Listing C.12: Running Ebe benchmarks with default parameters.

Once this command is ran, the script requests sudo privileges, unless run by an administrator user. This is because the `nice` command used to run benchmarks with maximum priority on a single core requires higher user privileges.

Benchmark parameters can be set using command line options. To display message containing all command line options and their short descriptions `python3 benchmark.py -h` can be used. The command line options are following:

- `-ebe` *path* – sets custom path to Ebe. By default benchmarks expect Ebe to be accessible under `ebe` command.
- `-o` *path* – file or directory to which save the benchmark results. By default the output will be placed in current working directory. If this option is not set or it is a directory, then the file with results will be named using current date and time as a prefix, followed by Ebe's version and `_benchmarks.json` suffix.
- `-i` – only Ebei benchmarks will be run.
- `-c` – only Ebec benchmarks will be run.
- `-t` *test_name* – runs only tests names as the argument provided. This option can be specified multiple times to run a set of tests.
- `-ebec` *path* – overrides default Ebec benchmarks path to the one provided.
- `-ebei` *path* – overrides default Ebei benchmarks path to the one provided.
- `-iter` *number* – option's arguments specifies how many times should be each benchmark repeated. This number is 10 by default.
- `-args` *"arguments"* – option's argument will be passed to Ebe for each executed tests. This can be used to set Ebe specific options.
- `-Werror` – all benchmark warnings should be treated as errors – warnings will interrupt benchmarks.

Appendix D

Contents of the included storage media

included storage media contains Ebe's source code, Ebe tools, scripts and input for all experiments from Section 6, LaTeX source code for this thesis and scripts or data sets for all the comparisons and tests mentioned in this thesis.

Folder structure on the included storage media is as follows:

- `ebe` – Ebe project (its structure is described in Section A.1).
- `ebe-tools` – Ebe tools (its structure is described in Section A.2.
- `experiments`
  - `ex1` – experiment 1 source code and input files,
  - `ex2` – experiment 2 source code and input files,
  - `ex3` – experiment 3 source code and input files,
  - `ex4` – experiment 4 source code and input files,
- `latex-source-code` – this thesis LaTeX source code.
- `tests` – comparisons and tests mentioned in this thesis.

Each experiment folder contains original file that was to be edited and additional folders named after the tool that were used. Inside of each folder is then script, which was used or the command used, but saved as `.sh` file. The `ebe` folders contain only the input example file and output example file.