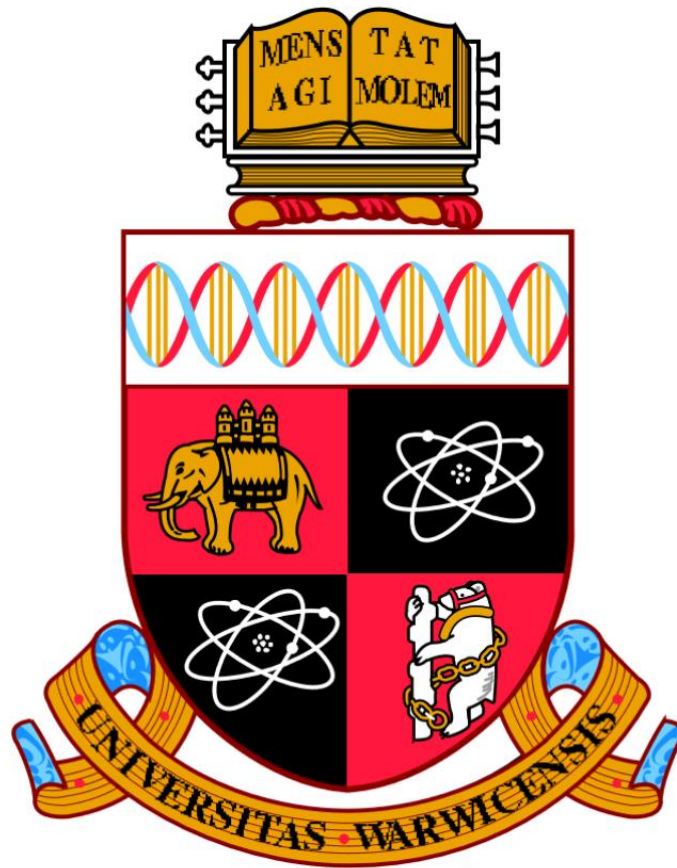


# Algorithms for Compressed Strings: Theory and Practice



Author: Callum Onyenaobiya

Supervisor: Dmitry Chistikov

Year of Study: 3<sup>rd</sup> year (CS310)

## Abstract

With the ever-growing demands of information storage and data transfer, the need for data compression is becoming increasingly important. Compression has previously been used to save storage and communication costs at the cost of requiring a decompression step when wanting to use this data; this can take excessive computational resources that we would rather avoid. Encouragingly, advances in compressed string processing algorithms give a new perspective in which compression can be regarded as a form of pre-processing which allows the efficient processing of strings as well as to reduce storage. The project explores many operations on compressed strings which do not have publicly available implementations including LZ77 encoding to straight-line programs, compressed equality checking and querying compressed strings. Hence, our aims were to research, design and implement a variety of operations performed on compressed representations, particularly straight-line programs. We integrate our implementations into a single piece of software designed to run such operations on text files for educational and experimental purposes. All algorithms discussed in detail were implemented and integrated successfully, with only an alternate, deterministic solution to compressed equality checking being explored but lacking implementation.

### Keywords List

- Data
- Compression
- Lempel-Ziv
- Grammars
- LZ77
- Equality
- Trees
- Operations

# Table of Contents

<b>1</b>	<b><i>Background</i></b>	<b>1</b>
1.1	Importance and brief History of Data Compression	1
1.2	Objectives	1
1.2.1	Adding to Existing Software	1
1.3	Types of Data	2
1.4	Types Of Data Compression	2
1.4.1	Lossless Compression	2
1.4.2	Lossy Compression	2
1.5	Types of Compression Techniques	3
1.6	Evaluating Compression Performance	4
1.7	Theoretical Limits of Data Compression	4
1.7.1	Quantifying Information	4
1.7.2	Shannon Entropy	5
1.7.3	Shannon Theorem	5
1.7.4	Kolmogorov Complexity	6
1.8	Explored Lossless Compression Algorithms	6
1.8.1	Run-length algorithms	6
1.8.1.1	Hardware data compression	7
1.8.1.1.1	Compression	7
1.8.1.1.2	Decompression	8
1.8.2	Dictionary-Based Compression	8
1.8.2.1	LZ77	9
1.8.2.1.1	Compression	10
1.8.2.1.2	Decompression	11
1.8.2.2	LZ78	12
1.8.2.2.1	Compression	12
1.8.2.2.2	Decompression	13
1.9	Context Free Grammars	13
1.9.1	Chomsky Normal Form	13
1.9.2	Straight Line Programs	13
1.10	Explored Operations on Compressed Representations	14
<b>2</b>	<b><i>Methodology</i></b>	<b>16</b>
2.1	Approach to Unseen Algorithms	16
2.2	Research methodology	16
2.2.1	Research dependencies	16
2.2.2	Agile Research	20
2.2.3	Resolving Research Issues	20
2.3	Development methodology	21
2.3.1	Agile Development	21

2.3.2	Agile Testing.....	21
2.4	Software Used.....	21
2.5	Legal, Social, Ethical, Professional Issues .....	22
2.6	Project Timetable .....	22
3	<b>Research.....</b>	<b>23</b>
3.1	<b>LZ77 Encoder .....</b>	<b>23</b>
3.1.1	Alternatives Considered .....	23
3.2	<b>LZ77 Decoder .....</b>	<b>23</b>
3.3	<b>Grammar Parser .....</b>	<b>24</b>
3.3.1	Defining syntax .....	24
3.3.2	Parsing.....	24
3.3.3	Only Allowing SLPs .....	24
3.3.4	Only Allowing CNF.....	25
3.3.4.1	Checking for CNF.....	25
3.3.4.2	Converting to CNF (Optional).....	26
3.3.4.3	Querying a Grammar.....	26
3.4	<b>LZ77 to SLP Conversion.....</b>	<b>27</b>
3.4.1	G-Factorization .....	27
3.4.2	AVL-Grammars.....	28
3.4.2.1	Concatenation algorithm.....	29
3.4.3	Construction of Small Grammar-Based Compression From LZ77 .....	30
3.5	<b>Compressed Equality Checker .....</b>	<b>32</b>
3.5.1	Overview .....	32
3.5.1.1	Randomised Solution.....	33
3.5.1.2	Deterministic Solution (optional) .....	33
3.5.2	Subsequence Decomposition .....	33
3.5.2.1	Randomised Decomposition .....	33
3.5.2.1.1	Assigning Priorities .....	33
3.5.2.1.2	Decomposition Rule.....	33
3.5.2.2	Deterministic Decomposition (optional) .....	33
3.5.2.2.1	Six Colouring Algorithm .....	34
3.5.2.2.2	Three Colouring Algorithm .....	34
3.5.2.2.3	Decomposition Rule.....	35
3.5.3	Representation of Sequences .....	35
3.5.4	Storing Sequences .....	36
3.5.5	Operations .....	36
3.5.5.1	Equality.....	36
3.5.5.2	Making Sequences.....	37
3.5.5.3	Concatenate .....	37
3.5.5.3.1	Randomised Concatenation .....	37
3.5.5.3.2	Deterministic Concatenation (optional).....	37
3.5.5.4	Split.....	38
4	<b>Design.....</b>	<b>38</b>
4.1	<b>LZ77 Encoder .....</b>	<b>38</b>
4.1.1	System Requirements.....	38
4.1.2	Software Design.....	38
4.2	<b>LZ77 Decoder .....</b>	<b>39</b>
4.2.1	System Requirements.....	40

4.2.2	Software Design .....	40
<b>4.3</b>	<b>Grammar Parser .....</b>	<b>40</b>
4.3.1	System Requirements .....	41
4.3.2	Software Design .....	41
<b>4.4</b>	<b>LZ77 to SLP Conversion .....</b>	<b>42</b>
4.4.1	System Requirements .....	42
4.4.2	Software Design .....	42
<b>4.5</b>	<b>Compressed Equality Checking .....</b>	<b>44</b>
4.5.1	System Requirements .....	44
4.5.2	Software Design .....	45
<b>4.6</b>	<b>Tree Printer .....</b>	<b>47</b>
4.6.1	System Requirements .....	47
4.6.2	Software Design .....	47
<b>4.7</b>	<b>Main Program .....</b>	<b>48</b>
4.7.1	System Requirements .....	48
4.7.2	Software Design .....	49
<b>5</b>	<b><i>Implementation</i> .....</b>	<b>50</b>
<b>5.1</b>	<b>Main Menu .....</b>	<b>50</b>
<b>5.2</b>	<b>LZ77 Operations .....</b>	<b>50</b>
5.2.1	Compression .....	50
5.2.2	Decompression .....	51
5.2.3	LZ77 to SLP .....	51
<b>5.3</b>	<b>Grammar Operations .....</b>	<b>52</b>
5.3.1	Parsing a Grammar .....	52
5.3.2	Viewing Parse Tree .....	52
5.3.3	Balance an Inputted Grammar .....	53
5.3.4	Compute G-factors .....	53
5.3.5	Querying a Grammar .....	54
<b>5.4</b>	<b>Signature Data Structure Operations .....</b>	<b>54</b>
5.4.1	Creating a New Signature .....	54
5.4.2	Concatenating Sequences .....	55
5.4.3	Splitting a Sequence .....	56
5.4.4	Equality Checking .....	56
5.4.5	Saving a Store For Later Use. ....	57
<b>5.5</b>	<b>Additional Observations Using Software .....</b>	<b>58</b>
5.5.1	LZ77 Compression Software .....	58
5.5.2	Optimised Querying .....	59
5.5.3	Conservation of Signature Structures .....	59
5.5.4	Randomised Nature of Software .....	61
<b>5.6</b>	<b>Documentation of Software .....</b>	<b>62</b>
<b>6</b>	<b><i>Testing</i> .....</b>	<b>62</b>
<b>6.1</b>	<b>LZ77 Operations .....</b>	<b>62</b>
<b>6.2</b>	<b>Grammar Operations .....</b>	<b>63</b>
<b>6.3</b>	<b>Signature Store Operations .....</b>	<b>64</b>
<b>7</b>	<b><i>Evaluation and Lessons Learned</i> .....</b>	<b>64</b>

<b>8</b>	<b><i>Conclusions and Future Work</i></b>	<b>65</b>
8.1	<b>Conclusions</b>	<b>65</b>
8.1.1	Project Overview	65
8.1.2	Evaluation Against System Requirements	66
8.2	<b>Future Work</b>	<b>68</b>
<b>9</b>	<b><i>Author's Assessment of Project</i></b>	<b>68</b>
<b>10</b>	<b><i>References</i></b>	<b>69</b>





# 1 Background

## 1.1 Importance and brief History of Data Compression

With the ever-growing demands of information storage and data transfer, the need for data compression is becoming increasingly important. Data compression is a technique which is used to decrease the size of data [1]; without it, simple tasks such as image, audio and video uploads to a website would not be practical. Mobile phones would not be able to provide communication clearly without data compression [1]. Compression has previously been used to save storage and communication costs at the cost of requiring a decompression step when wanting to use this data; this can take excessive computational resources that we would rather avoid [2]. With data compression techniques, we can reduce the consumption of resources, such as hard disk space or transmission bandwidth [1] making a vast variety of technologies possible. The list below gives a brief survey of the historical milestones in data compression [3]:

- 1000BC - Shorthand
- 1829 - Braille code
- 1842 - Morse code
- 1930 - Analog Compression
- 1950 - Huffman codes
- 1975 - Arithmetic coding
- 1977 - Dictionary-based compression
- The 1980s
  - o Early - FAX
  - o Mid - Video conferencing, still images (JPEG), improved FAX standard (JBIG)
  - o Late - Onward Motion video compression (MPEG)
- The 1990s
  - o Early - Disk compression (stacker)
  - o Mid - Satellite TV
  - o Late - Digital TV (HDTV), DVD, MP3

## 1.2 Objectives

### 1.2.1 Adding to Existing Software

The operations we have explored throughout the project and briefly discussed in 1.10 have current research papers dedicated to them, as made evident by our citations. Despite this, implementations of such algorithms are not publicly available, and hence the principal aim of our project was not only to explore and study such algorithms but also provide publicly

available software which implemented them. Such software would prove useful to computer scientists exploring these papers, who want to strengthen their understanding of such algorithms through analysis of their implementations.

As discussed in 1.9.2, straight-line programs give a proper representation for the algorithmic problems we explore for compressed strings. However, such a representation is not practical for real-world application, and hence our original software will stem from a variant of the LZ77 compression scheme, as unlike LZ78, LZ77 has not been patented and is the reason why successors based on LZ77 are so widely used [4]. From this, we can utilise algorithms discussed in 1.10 to obtain our required straight-line program representation and build operations on it. Details of the types of operations explored can be found later in this introduction (see 1.10) with the discussion of implementation details of the algorithms chosen throughout 5.

## 1.3 Types of Data

The word *data* includes any digital information that can be processed by a computer [3]; this includes text, voice, video, still images and audio. The data before any compression process is called the *source* data. The three most common types of source data in computer science are [3]:

- **Text:** Data usually represented by ASCII code (or EBCDIC).
- **Image:** Data often represented by a two-dimensional array of *pixels* in which it's colour code associates each pixel.
- **Sound:** Data represented by a periodic function.

In the application world, *multimedia* usually refers to the compressed data whose source originating from a mixture of text, image and sound.

## 1.4 Types Of Data Compression

Data compression is merely a means for efficient digital representation of a source of data; such data includes text, image and audio [3]. The primary goals of data compression are to represent a source in digital form with as few bits as possible while meeting a minimum requirement of reconstruction [3]. We achieve this goal by removing any redundancy presented in the original source. There are two types of data compression algorithms, *lossless* and *lossy*, each varying in the ability to reconstruct the original source.

### 1.4.1 Lossless Compression

In lossless data compression, one can reconstruct the original message exactly from the compressed message without any loss of information [1]. Lossless compression techniques are mainly applied to symbolic data; such data includes character text, numerical data, computer source code and executable graphics and icons [3].

Lossless compression techniques are used when the original source of data is so important that we cannot afford to lose any details. For example, medical images, text and images preserved for legal reasons [3].

### 1.4.2 Lossy Compression

In contrast, a compression method is lossy only if it is not possible to reconstruct the original source precisely from the compressed representation [3], this is because we only reconstruct an

approximation of the original message [1]; there are some insignificant details that may get lost during the process of compression [3]. Approximate reconstruction may perform well in terms of compression-ratio (see 1.6) but usually requires a trade-off between visual quality and the computation complexity [3].

We typically use lossy algorithms for image, sound and video compression since the reductions in quality observed by such compression techniques are minimal.

## 1.5 Types of Compression Techniques

We often refer to data compression as *coding* because it aims to find a shorter way of representing data [3]. Compression and decompression can be referred to as *encoding* and *decoding* respectively. Some major compression algorithms are as follows [3]:

- ***Run-length coding***

The main idea is to replace *consecutively* repeated symbols in a source with a code pair consisting of either the repeating symbol and the number of its occurrences or a sequence of non-repeating symbols.

*Example: We can represent the string ABBCCCDDDD using (A,1)(B,2)(C,3)(D,4)*

- ***Quantisation***

The main idea is to apply a particular computation to a set of data in order to achieve an approximation in a more straightforward form [3].

*Example: Consider the set of integers (7, 223, 15, 28, 64, 37, 145); we can represent each integer by eight binary bits. However, if we use a common divider to apply to each integer and round its value to the nearest integer, say 16, the above set becomes (0,14,1,2,4,2,9). Now, we can store each integer using only four bits.*

- ***Statistical-based coding***

In statistical-based coding, we use statistical information to replace a fixed-size code of symbols by a shorter variable-sized code.

*Example: We can code the more frequently occurring symbols with fewer bits. We obtain this statistical information by merely counting the frequency of each character in a file. Alternatively, we use the probability of each character.*

- ***Dictionary-based coding***

The main idea is to find the frequently occurring sequences of symbols (FOSSs) and build a dictionary of them. We then associate each sequence with an index and replace the FOSS occurrences with the indices.

- ***Transform-based coding***

The transform-based approach models data by mathematical functions [3] and applies mathematical rules to primarily diffuse data. We aim to change a mathematical quantity (i.e. a sequence of numbers) to another form with useful features. We mainly use it in *lossy* data compression algorithms involving [3]:

- Analysing a signal such as sound and image
- Decomposing it into frequency components
- Making use of the limitations of human perception.

- ***Motion prediction***

Motion prediction is another *lossy* compression technique used for sound and *moving* images. We replace objects (i.e. a 16x16 block of pixels) in frames with references of the same object at a slightly different position in the previous frame.

## 1.6 Evaluating Compression Performance

We can use various criteria when measuring the performance of a compression algorithm; it depends on what our priority concern is [3]. The simplest of these criteria is the effect that compression makes to the difference in the size of the input file before the compression versus the size of the output after the compression. It can be challenging to measure the performance of a compression algorithm in general because its compression behaviour depends much on whether the data contains the correct patterns that a particular algorithm looks for [3].

The most straight forward way to measure the effect of compression is to use the *compression ratio*. The aim is to measure the effect of compression by the shrinkage of the size of the source in comparison with the size of the compressed version.

There are several ways of measuring the compression effect, such as:

- **Compression ratio:** The ratio of the size after compression to size before compression.

$$\text{Compression ratio} = \frac{\text{size after compression}}{\text{size before compression}}$$

- **Compression factor:** The inverse of compression ratio.

$$\text{Compression factor} = \frac{\text{size before compression}}{\text{size after compression}}$$

- **Saving percentage:** displaying shrinkage as a percentage

$$\text{Saving percentage} = \frac{\text{size before compression} - \text{size after compression}}{\text{size before compression}}$$

*Example: A source image file with 512x512 pixels with 262,144 bytes is compressed into a file with 65,536 bytes. The compression ratio is 0.25, and the compression factor is 4. The saving percentage is 75%.*

## 1.7 Theoretical Limits of Data Compression

### 1.7.1 Quantifying Information

The notion of a *bit* has been made famous by computers. It is a unit of information that takes two values, 0 or 1. The information size of a set  $A$  is the number of bits that is necessary to encode each element of  $A$  separately [5], i.e.

$$H_0(A) = \log_2 |A|.$$

The unit is the *bit*. If we have two sets  $A$  and  $B$ , then

$$H_0(A \times B) = H_0(A) + H_0(B).$$

The information size is not necessarily an integer. If we need to encode the elements of  $A$ , the number of necessary bits is  $\lceil H_0(A) \rceil$ .

Now, let  $\mathbf{A} = (A, p)$  be a discrete probability space. Meaning,  $A = \{a_1, \dots, a_n\}$  is a finite set and each element has a probability associated with it,  $p_i$ . The information gain  $G(B|A)$

measures the gain obtained by the fact that the outcomes belongs to the set  $B \subset A$ . We denote  $p(B) = \sum_{i \in B} p_i$  [5].

The information gain is:

$$G(B|A) = \log_2 \frac{1}{p(B)}$$

The unit for information gain is the bit. One bit is gained if  $p(B) = \frac{1}{2}$  [5].

### 1.7.2 Shannon Entropy

In information theory, entropy provides an absolute limit on the shortest possible average length of a lossless compression encoding of the data produced by a source.

*Shannon entropy* was named after Claude Shannon, although its origins date back to Pauli and Von Neumann [5].

The Shannon entropy of  $\mathbf{A}$  is [5]:

$$H(\mathbf{A}) = - \sum_{i=1}^n p_i \log_2 p_i$$

### 1.7.3 Shannon Theorem

A significant problem in information theory is dealing with large quantities of information that need encoding. Starting with a finite set,  $A$  that denote the 26 letters from the Latin alphabet or a larger set of words. We consider a file that contains  $N|A|$  symbols (where  $N$  is large). The number of bits required so that the file can be encoded without loss of information is simply the information size,  $H_0(A^N) = NH_0(A)$ . If we allow an error  $\delta$  and seek to encode only files that fall in a set  $B \subset A$  such that  $p(B) \geq 1 - \delta$ , the information size is given by  $H_\delta(A)$ , where

$$H_\delta(A) = \max_{\substack{B \subset A \\ p(B) \geq 1-\delta}} \log_2 |B|$$

Regarding probability, in the real world, information is selected for a purpose, and its content is well chosen [5]. We model the problem of data transmission using probabilities: A string of  $N$  characters are selected at random using some probability. We want to encode this string to transmit and decode it; we assume no error during such operations. This model is behind all compression algorithms [5].

To find  $H_\delta(A^N)$  (the probability of  $(a_1, \dots, a_N) \in A^N$  is  $\prod p(a_i)$ ) we notice that  $H_\delta(A) \leq H_0(A)$  and that  $H(A) \leq H_0(A)$ . We have  $H_0(A^N) = NH_0(A)$ , but  $H_\delta(A^N)$  is smaller than  $NH_\delta(A)$  in general.

We then arrive at the *Shannon source coding theorem*, for any  $\delta > 0$ :

$$\lim_{N \rightarrow \infty} \frac{1}{N} H_\delta(A^N) = H(\mathbf{A}).$$

The theorem means that when allowing for a small error and our message is large, the number of required bits is approximately  $NH(\mathbf{A})$ . The limit in the theorem is a true limit and thus, Shannon entropy gives the optimal compression rate, that can be approached but not improved [5].

## 1.7.4 Kolmogorov Complexity

Kolmogorov Complexity is a way of evaluating the quantity of information in a piece of data [6]. Intuitively, the Kolmogorov Complexity of a piece of data is the shortest program that could reproduce a given piece of data [6]. For example, consider the string composed of the letter b repeated 1 million times. If we were to print the string out, it would take a considerable amount of space. Intuitively, the string is simple, and its length is not an indication of the complexity of the information it contains. We can also represent the same string can also by the simple phrase “repeat b 1 million times”; with such direction, one could perfectly reproduce the string. By contrast, consider a truly random sequence of 1 million characters. To reproduce such a specific sequence would require the entire sequence itself. There would not be a shorter set of directions that could be used to produce the sequence [6]. There is not a shorter set of directions that could be used to produce the sequence. Therefore, despite the two strings being of equal length, the sequence of b’s is more straightforward than the random sequence. We capture this difference with the notion of Kolmogorov Complexity.

Fix  $\Sigma = \{0,1\}$ . Let  $f : \Sigma^* \rightarrow \Sigma^*$ . Then (relative to  $f$ ), a description of string  $\sigma$  is simply some  $\tau$  with  $f(\tau) = \sigma$ . We restrict  $f$  by requiring that it be *computable*.

We are now able to define Kolmogorov Complexity  $C_f$  [7]:

$$C_f(x) = \begin{cases} \min\{|p| : f(p) = x\} & \text{if } x \in \text{ran } f \\ \infty & \text{otherwise} \end{cases}$$

It is better to have a notion of complexity that does not vary according to which  $f$  we choose. We achieve a somewhat independence if we use a Universal Turing Machine. A Turing Machine  $U$  exists such that for all partial computable  $f$ , there exists a program  $p$  such that  $\forall y, U(p, y) = f(y)$ . We define a partial computable function  $g$  by letting  $g0^{|p|}1py = U(p, y)$ . The following fact removes the dependence on  $f$ , thus allowing us to talk about the Kolmogorov complexity [7]:

*For all partial computable  $f$ , there exists a constant  $c$  such that for all  $x$ ,  $C_g(x) \leq C_f(x) + c$*

We now define  $C(x) = C_g(x)$  and can move to randomness. The idea is that a string is random if it cannot be compressed. Meaning, if it has no short description. Using  $C(x)$ , we can formalize this idea via the following [7]:

*For all  $n$ , there exists some  $x$  with  $|x| = n$  such that  $C(x) \geq n$ . Such  $x$  are called (Kolmogorov) random.*

## 1.8 Explored Lossless Compression Algorithms

The project chooses to explore *lossless* data compression in great detail. Before exploring the variety of operations performed on compressed strings, research was conducted to understand what lossless data compression algorithms currently exist and how they differ.

### 1.8.1 Run-length algorithms

As briefly discussed in 1.5, run-length algorithms assign codewords to consecutive recurrent symbols; we refer to such recurrent symbols as *runs*. The main idea is to replace consecutive repeating symbols by a short codeword unit containing [3]:

- A single symbol
- A run-length count
- Interpreting indicator

*Example: We can replace the string **AAAAAAAAA**, containing 9 consecutive repeating As, with a short unit **r9A** consisting of the symbol r, 9 and A, where r represents ‘repeating symbol’, 9 means ‘occurring 9 times’ and A indicates that this should be interpreted as ‘symbol A’ [3].*

Run-length algorithms are most effective when the data source contains many runs of consecutive symbols [3]. The symbols can be:

- Characters in a text file
- Binary digits in a binary file
- Black-and-white pixels in an image.

### **1.8.1.1 Hardware data compression**

Despite being simple, run-length algorithms have their practical uses. The Hardware Data Compression (HDC) algorithm, is used by tape drives connected to IBM computer systems and in the IBM SNA, which uses a similar algorithm, as a standard data communication [3].

The coder replaces sequences of consecutive identical symbols in the standard way already discussed; it contains three elements [3]:

- A single symbol
- A run-length count
- An indicator signifying how we interpret the symbol and count.

#### **1.8.1.1.1 Compression**

In the purest version of the HDC algorithm, we only use ASCII codes for [3]:

- The single symbols
- A total of 123 control characters with a run-length count, including:
  - o Repeating control characters:  $r_1, r_2, \dots, r_{63}$
  - o Non-repeating control characters:  $n_1, n_2, \dots, n_{63}$

Every repeating character,  $r_i$  ( $2 \leq i \leq 63$ ), is followed by either a control character or a symbol. If the following symbol is another control character,  $r_i$  (alone) signifies  $i$  repeating blanks. Otherwise,  $r_i$  signifies that the symbol after it repeats  $i$  times [3]. Each non-repeating character,  $n_i$  ( $1 \leq i \leq 63$ ), is followed by a sequence of  $i$  non-repeating symbols.

With these rules, we can, therefore, apply the following encoding run-length algorithm:

- 1) If a string of  $i$  ( $2 \leq i \leq 63$ ) consecutive spaces is found, output a single repeating control character  $r_i$ .
- 2) If a string  $i$  ( $3 \leq i \leq 63$ ) consecutive symbols (not spaces) is found, output two characters: a repeating control character  $r_i$  followed by the symbol.

- 3) Otherwise, identify the longest string  $i$  ( $1 \leq i \leq 63$ ) non-repeating symbols that do not satisfy the above conditions, and output the non-repeating control character  $n_i$  followed by the string.

*Example: AAA\_ \_ \_ BCDEFGH\_ \_ 22IJKLMNAAAAAAAAA can be compressed to  $r_3Ar_4n_8BCDEFGHr_2r_22n_6IJKLMNr_7A$*

#### 1.8.1.1.2 Decompression

The decoding process follows a very similar procedure to encoding:

- 1) If a repeating character  $r_i$  is found:
  - a. Output  $i$  spaces if no symbol follows.
  - b. Otherwise, output  $i$  symbols.
- 2) Otherwise, output the next  $i$  non-repeating symbols.

### 1.8.2 Dictionary-Based Compression

Again, as briefly discussed earlier, dictionary-based compression is a technique that incorporates the structure in the data in order to achieve better compression; we achieve this by eliminating the redundancy of storing repetitive strings for words and phrases repeated within the text stream [3]. We keep a record of the most common words or phrases in a document and use their pointers in the dictionary as output tokens. In an ideal scenario, the tokens are much shorter in comparison with the words or phrases themselves and the words and phrases are frequently repeated in the text stream [3].

There are three types of dictionary encoders [8]:

- **Static dictionary**  
The purest form of dictionary coding comes in the form of a static dictionary. Such a dictionary may contain frequently occurring phrases of arbitrary length, or n-grams (n-letter combinations). This kind of dictionary can easily be built upon existing coding such as ASCII by using previously unused codewords or extending the length of codewords to accommodate the dictionary entries [8]. A static dictionary achieves little compression for most data sources.
- **Semi-adaptive dictionary**  
We improve on static dictionaries by using a semi-adaptive encoder. We create a dictionary custom-tailored for the message to be compressed [8]; this makes it necessary to store the dictionary together with the data. It also requires two passes over the data, one to build the dictionary and another one to compress the data [8].
- **Adaptive dictionary**  
In adaptive dictionary encoding, the dictionary is being built in a single pass, while at the same time also encoding the data [8]. The Lempel-Ziv algorithms belong to this category of dictionary coders.



The main dictionary-based compression algorithm explored was the Lempel-Ziv Algorithm. It is not a single algorithm, but a whole family of algorithms [8]; the Lempel-Ziv algorithm is an ‘offshoot’ of the two algorithms proposed by Jacob Ziv and Abraham Lempel in their landmark papers in 1977 and 1978 which are LZ77 and LZ78 respectively [9]. The Lempel-Ziv algorithms are widely used in compression utilities such as GZIP and GIF image compression; Figure 1 gives a brief depiction of the Lempel-Ziv family.

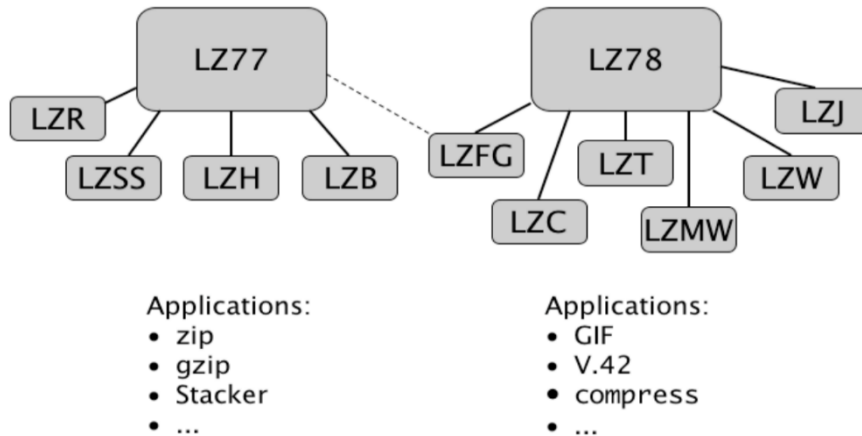


Figure 1: The Lempel-Ziv Algorithm Family [5].

### 1.8.2.1 LZ77

Jacob Ziv and Abraham Lempel have presented their dictionary-based scheme in 1977 for lossless data compression [9]. LZ77, like all dictionary-based encoders, exploit the fact that words and phrases within a text file are likely to be repeated. When we observe repetition, we can encode it as a pointer to an earlier occurrence, with the pointer followed by the number of characters that were matched [9].

In the LZ77 approach, the dictionary to use is a *portion* of the previously seen input file. A sliding window is used to define the dictionary and scan the input sequence of symbols [3].

The sliding window consists of two parts:

- **Search window:** contains the set of characters of the recently seen symbol sequence.
- **Lookahead buffer:** contains the next set of characters of the sequence to be encoded.

The size of each buffer is fixed in advance [3].

Figure 2 shows an example of such a sliding window: let our search window be a size of 16 bytes (1 byte per symbol) and lookahead buffer a size of 12, where the sequence of symbols in the search window has been seen and compressed, but we are yet to compress the sequence in the lookahead buffer.

(or, <-- text moving)

window moving -->

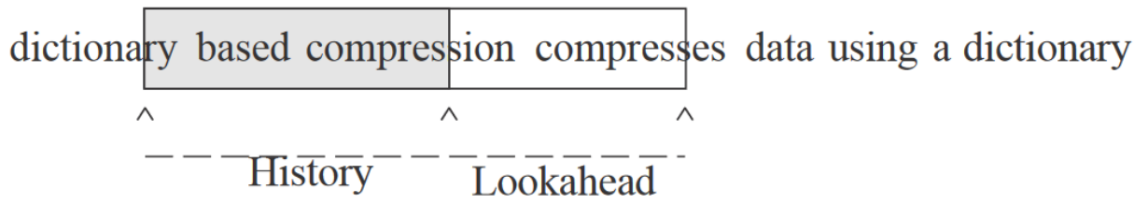


Figure 2: Search window and Lookahead buffer [3].

In a practical implementation, the search window is some thousands of bytes long, and the lookahead buffer is only tens of bytes [9].

#### 1.8.2.1.1 Compression

A typical compression step is as follows: the algorithm searches the sliding window for the longest match with the beginning of the lookahead buffer and outputs a pointer to that match, or until we reach the end of the lookahead buffer. In LZ77, we encode a sequence in the form of a tuple <offset, string length, character> defined as follows [9]:

- Offset: The location of the beginning of the match in the search window relative to the current location
- String length: The length of the match.
- Character: The first non-matching character in the lookahead window following the characters that match.

In the case that the algorithm fails to find a match, we generate a null pointer as the pointer (both the offset and the string length equal 0), and we use the first symbol in the lookahead buffer, i.e. (0,0 'character').

Figure 3 gives an example string *abracadabrad* being compressed using the LZ77 algorithm where our output tuples are represented as (offset, string length, character) as discussed previously. In this example, we use a search window of size 7, and a lookahead buffer size of 5.

7	6	5	4	3	2	1											output	
							a	b	r	a	c		ada...	(0,0,a)				
						a	b	r	a	c	a		dab...	(0,0,b)				
					a	b	r	a	c	a	d		abr...	(0,0,r)				
				a	b	r	a	c	a	d	a		bra...	(3,1,c)				
		a	b	r	a	c	a	d	a	b	r		ad	(2,1,d)				
a	b	r	a	c	a	d	a	b	r	a	d		(7,4,d)					
a	d	a	b	r	a	d												
Search buffer							Look-ahead buffer											

Figure 3: Example of LZ77 compression [6].

We observe that the LZ77 encoder compresses the 12-character input into six tuples. Assuming the maximum number of bits required for each tuple =

$$\lceil \log_2(\text{search window}) \rceil + \lceil \log_2(\text{search window} + \text{lookahead}) \rceil + \lceil \log_2(|\text{alphabet}|) \rceil$$

Our tuple size =

$$\lceil \log_2(5) \rceil + \lceil \log_2(12) \rceil + \lceil \log_2(5) \rceil = 10$$

We can, therefore, evaluate the performance of this compression using the techniques discussed in 1.6:

$$\text{Compression ratio} = \frac{(12 * 8)}{6 * (10)} = 1.6$$

$$\text{Compression factor} = \frac{6 * (10)}{(12 * 8)} = 0.625$$

$$\text{Saving percentage} = \frac{(12 * 8) - 6 * (10)}{(12 * 8)} = 37.5\%$$

#### 1.8.2.1.2 Decompression

When it comes to decompression, the decoder also maintains a search window and lookahead buffer of the same size as the encoder. However, it is far simpler to decode rather than encode because there are no matching problems. The decoder reads a token and decides whether the token represents a match [3]. If we find a match, the decoder then uses the offset and string length in the token to reconstruct the match. Otherwise, it directly outputs the character in the token [3].

Using our previously computed tuples, we can pass them into our decoder and output the expected *abracadabrad*, as shown in Figure 4 [4].

input		7	6	5	4	3	2	1
(0,0,a)								a
(0,0,b)							a	b
(0,0,r)						a	b	r
(3,1,c)				a	b	r	a	c
(2,1,d)		a	b	r	a	c	a	d
(7,4,d)	abrac	a	d	a	b	r	a	d

Figure 4: Example of LZ77 decompression [7].

### 1.8.2.2 LZ78

Unlike LZ77, LZ78 maintains an explicit dictionary [9]. The encoded output consists of two elements: an index referring to the longest matching dictionary entry and the first non-matching symbol [9], the algorithm then adds the index and symbol pair to the dictionary. When we encounter a symbol not yet in our dictionary, the codeword has the index value 0, and we add this to the dictionary. With this method, the algorithm gradually builds up a dictionary [9].

#### 1.8.2.2.1 Compression

We see the most straightforward encoding algorithm for encoding in LZ78 in Figure 5.

```

w := NIL;
while (there is input){
    K := next symbol from input;
    if (wK exists in the dictionary) {
        w := wK;
    } else {
        output (index(w), K);
        add wK to the dictionary;
        w := NIL;
    }
}

```

Figure 5: LZ88 encoding algorithm [8].

This version of the algorithm does not prevent the dictionary from growing forever [8]. There are various solutions to limit dictionary size; the easiest way is to stop adding entries and continue as a *static dictionary* or to bin the existing dictionary in its entirety and start from scratch after a certain number of entries has been reached [8].

Figure 6 gives an example string **ABBCBCABABCAABCAAB** being compressed using the LZ78 algorithm [9].

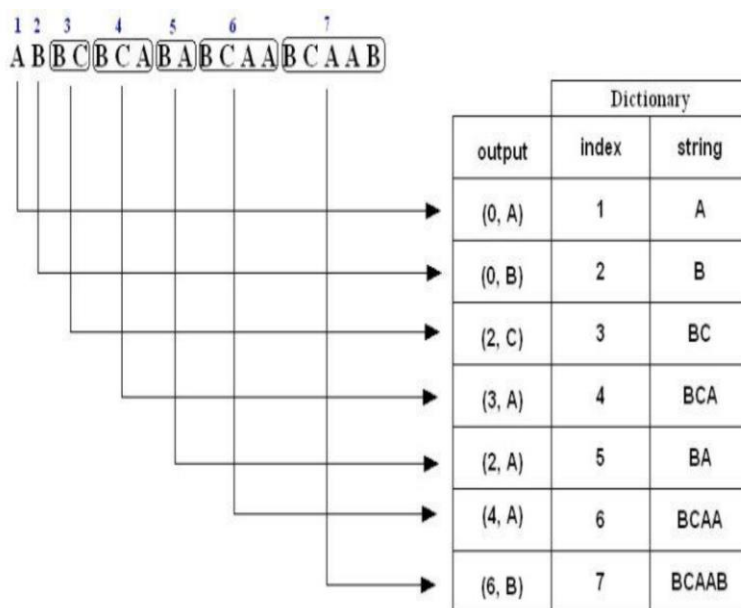


Figure 6: Example of the LZ78 algorithm [9].

The output of such a compression gives: (0,A)(0,B)(2,C)(3,A)(2,A)(4,A)(6,B).

We can view LZ77 as a restricted version of LZ78 since the factors referring the strings are restricted to start and end in substrings that correspond to the expansion of a single proceeding factor [10].

#### 1.8.2.2.2 Decompression

Since LZ78 maintains an explicit dictionary, decompression is natural. For each tuple, we directly obtain the string stored in the given index (if non-zero), and append that and the tuple's character to our final output.

Like LZ77, LZ78 is slow in compression but benefits from a much faster decompression due to its explicit dictionary. Despite this, it does not always achieve as high compression ratios as LZ77 due to the extra overheads required. The most significant advantage LZ78 has over the LZ77 algorithm is the reduced number of string comparisons required after each encoding step [9].

## 1.9 Context Free Grammars

We have made many advances in compressed string processing algorithms, giving us an interesting new perspective on data compression, not only to be regarded as a form of pre-processing which reduces space requirements for storage but allowing the efficient processing of strings to beat a straightforward “decompress-and-check” strategy [11]. To appreciate such algorithms, the understanding of ‘Context-Free Grammars’ is essential for the reader, knowledge which I first acquired from my Formal Languages class (CS259) [12]. A context-free grammar is a set of productions used to generate patterns of strings and consists of the following components:

- A set of *terminal symbols*, which are characters in the alphabet
- A set of *nonterminal symbols*: placeholders for sequences of terminal symbols generated by the given nonterminal.
- A set of *productions*: rules for replacing nonterminal symbols in a string with other non-terminal/terminal symbols.
- A *start symbol*: a particular nonterminal that appears in the initial string generated by the grammar.

### 1.9.1 Chomsky Normal Form

For a context-free grammar to be in *Chomsky Normal Form*, the following conditions must hold:

*A context-free grammar  $A = (V, \Gamma, S, P)$  ( $V$  is the set of nonterminals,  $\Gamma$  is the set of terminals,  $S \in V$  is the initial nonterminal, and  $P \subseteq V \times (V \cup \Gamma)^*$  is the set of productions) is in Chomsky Normal Form if every production has one of the following forms [11]:*

- $(A, a)$  with  $a \in \Gamma$
- $(A, BC)$  with  $B, C \in V$

Every context-free grammar can be transformed into an equivalent grammar in Chomsky normal form in polynomial time [11].

### 1.9.2 Straight Line Programs

When discussing algorithms on compressed strings, we have to make precise the compressed representation we want to use [11]. Such a representation should have two properties [11]:

- It should cover many compression schemes from practice.
- It should be mathematically easy to handle.

Straight-line programs have both properties. A straight-line program is a context-free grammar that generates exactly one word. More formally:

*A straight-line program over the terminal alphabet  $\Gamma$  is a context-free grammar  $A = (V, \Gamma, S, P)$  ( $V$  is the set of nonterminals,  $\Gamma$  is the set of terminals,  $S \in V$  is the initial nonterminal, and  $P \subseteq V \times (V \cup \Gamma)^*$  is the set of productions) such that the following conditions hold [11]:*

- *For every  $A \in V$ , there exists exactly one production of the form,  $(A, a) \in P$  for,  $a \in (V \cup \Gamma)^*$ .*
- *The relation  $\{(A, B) \mid (A, a) \in P, B \in \text{alph}(a)\}$  is acyclic.*

Rules are unambiguous and non-recursive [10]. Every production  $(A, a)$  can also be written as  $A \rightarrow a$ . The word generated by the given straight-line program is denoted by  $\text{eval}(A)$ .

The derivation tree of the straight-line program  $A = (V, \Gamma, S, P)$  is a finite rooted ordered tree, where every node is labelled with a symbol from  $V \cup \Gamma$  [11]. The root is labelled with the initial nonterminal  $S$  and every node that is labelled with a symbol from  $\Gamma$  is a leaf of the tree. A node that is labelled with a nonterminal  $A$  such that  $(A \rightarrow a_1 \dots a_n) \in P$  (where  $(a_1 \dots a_n) \in V \cup \Gamma$ ) has  $n$  children that are labelled from left to right with  $a_1 \dots a_n$  [11].

The size of the straight-line program  $A = (V, \Gamma, S, P)$  is  $|A| = \sum_{(A,a) \in P} |a|$ . Every straight-line program can be transformed in linear time into an equivalent straight-line program in Chomsky normal form, as discussed in 1.9.1.

*Example [11]: Consider the straight line program  $A$  over the terminal alphabet  $\{a, b\}$  that consists of the following productions:  $A_1 \rightarrow b, A_2 \rightarrow a$  and  $A_i \rightarrow A_{i-1}A_{i-2}$  for  $3 \leq i \leq 7$ . The start nonterminal is  $A_7$ . Then  $\text{eval}(A) = abaababaabaab$ , which is the 7<sup>th</sup> Fibonacci word. The straight-line program  $A$  is in Chomsky normal form and  $|A| = 12$ .*

It is easy to see that with straight-line programs consisting of  $n$  productions, repeated doubling generates a string of length  $2^n$ . Given this, we can indeed see a straight-line program as a compressed representation of the string it generates [11].

## 1.10 Explored Operations on Compressed Representations

Many of the algorithms explored in this report stem from a survey conducted in [11], which discuss algorithmic problems on strings given in a compressed form via straight-line programs (SLP), such algorithms include:

- **Restructuring to and from compressed texts (Straight-line programs, LZ77, LZ78) without explicit decompression.**

The computation of small straight-line programs for a given input is known as the grammar-based compression problem [11]. For a string  $w$ , let  $\text{opt}(w)$  be the size of

the minimal SLP for  $w$ . Thus, there exists an SLP  $A$  such that  $eval(A) = w$ ,  $|A| = opt(w)$  and for every SLP  $B$  with  $eval(B) = w$ ,  $|B| \geq |A|$  holds.

The following theorem provides a lower bound for grammar-based compression:

*Unless  $P = NP$  there is no polynomial time algorithm with the following specification:*

- *The input consists of a string  $w$  over some alphabet  $\Sigma$*
- *The output is a grammar  $A$  such that  $eval(A) = w$  and  $|A| \leq \frac{8569}{8569} opt(w)$*

The best known upper bound for grammar-based compression is provided by Rytter [13], who explores the relationship between SLPs and dictionary-based compression.

Many dictionary-based compressed representations can be converted efficiently into straight-line programs; [13] illustrates that from the LZ77-encoding of a string, one can compute in polynomial time a straight-line program for the string and vice versa. Unlike LZ77, LZ78 can be seen as a grammar, and can convert to a straight-line program with less overhead [10]:

*For every factor  $f_k = (f_i, a)$  we create a node  $v_k$  with  $v_i$  as its left child and the terminal node representing  $a$  as its right child. We then build a binary tree with the sequence  $v_1, \dots, v_z$  as leaves.*

From this, we can see the close relationship straight-line programs have with the two intermediate Lempel-Ziv compressed representations, as well as the close relationship both algorithms have between each other.

#### - **Compressed Equality Checking.**

Given two straight-line programs  $A$  and  $B$ , does  $eval(A) = eval(B)$  hold? A simple decompress and compare algorithm is very inefficient, taking exponential time to compute  $eval(A)$  and  $eval(B)$ . Polynomial time algorithms exist, as discussed in papers such as [14], with the idea of computing for each string a signature, which is a small number, and that allows to do the equality test in constant time by simply comparing signatures.

#### - **Querying compressed strings**

One of the most straightforward questions one can ask about a given string is whether a position in the string carries a specific symbol. We define the *compressed querying problem* as follows:

Input: An SLP  $A$  over an alphabet  $\Gamma$ , a number  $1 \leq i \leq |eval(A)|$ , and a symbol  $a \in \Gamma$ .

Question: Does  $eval(A)[i] = a$  hold?

The *compressed querying problem* is P-complete, as discussed in [15].

# 2 Methodology

## 2.1 Approach to Unseen Algorithms

For every algorithm explored, we enforced the following simple three-step procedure:

- **Study:** In order to develop a successful portfolio of operations on compressed strings, understanding thoroughly how the algorithms work has been essential. Understanding has been achieved by thoroughly reading through various academic papers discussing such operations. Many papers were sourced initially from [11]. We have referenced any addition papers where appropriate. The selection process for finding relevant papers is discussed further in 2.
- **Implement:** Once we established a complete understanding of the given algorithm, it was then possible to begin implementation, using the methods discussed in 2.3.
- **Analyse:** In Theoretical Computer Science, we can analyse algorithms in a variety of ways such as time and space complexity. In the studying phase of each algorithm, both time and space complexity, and quality of understanding were assessed and were a factor in choosing which algorithms to implement. This project takes an interest in exploring the algorithms in a practical sense by pre-computing arbitrary test data, both compressed and uncompressed, varying in length and using this data as input for our implemented algorithms. We will do this by using our implementation of the LZ77 compression scheme (discussed in 3.1) to compress files of varying sizes containing structured texts, i.e. a book. We can, therefore, analyse the performance of an algorithm based on different sized blocks of data as well as how they compare to their uncompressed counterparts.

Given this approach, it was necessary to have two separate methodologies, both for research and development.

## 2.2 Research methodology

### 2.2.1 Research dependencies

Many of the explored algorithms throughout the project stemmed from an original survey conducted on compressed string algorithms in [11]. The explored operations were selected as fundamental to the project because:

- 1) They are fundamental operations in real-world applications.
- 2) They are technically challenging.
- 3) Can be realistically completed within the given time frame.

We have already discussed the operations in 1.10 with further discussion of their algorithmic details taking place in their respective research phase (see 3). Once we established such operations, it was necessary to gather papers discussing them in order to gain a stronger understanding of their requirements. After a given operation's study phase, it is possible that we explored a wide variety of solutions and, in those cases, we only implement selected algorithm/s



due to project time constraints; criteria when selecting such algorithm/s to be implemented were as follows:

- 1) Algorithmic performance.
- 2) Understanding of the algorithm.
- 3) Essential tools required for implementation.

Although we discuss the design of our software in detail in 4, it was necessary to outline a basic skeleton to establish any research dependencies (i.e. whether the study/implementation of a given algorithm requires prior work) that may be present and prioritise them accordingly. This task was made possible by reading our collected paper's introductory sections, which gave a basic overview of the requirements of each algorithm. The principal goals of the skeleton were to establish an initial research timetable that best suited our project.

Figure 7 outlines this basic skeleton, where  $(A \rightarrow B)$  indicates that **B** depends on the completion of **A**. It is important to note that dependencies imply that the software produced at any given box would be production ready and user facing (i.e. an application that the user can interact with). Despite this, the skeleton has been intentionally designed such that all boxes can be completed independently of others provided a suitable application interface (API) is provided for each; such details are outlined during implementation. Boxes outlined in a solid line required no further reading beyond the background reading completed in 1.

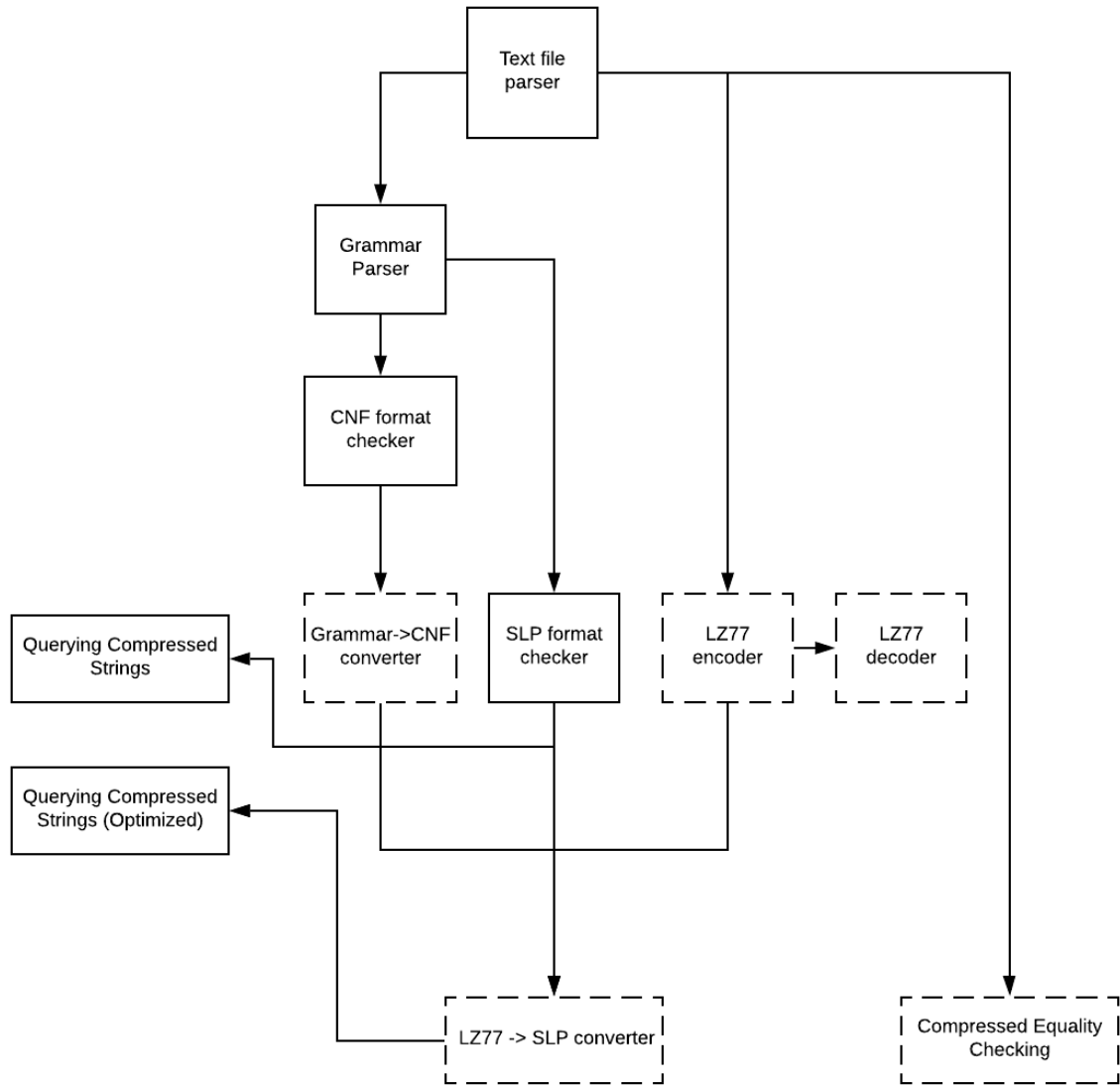


Figure 7: Research dependency diagram.

- **Text file parser**

Since our software stemmed off the dictionary based LZ77 compression scheme, it was necessary to ensure our software could take a text file as input. Such a task did not require the reading of research papers as all common programming languages, including our language of choice, make this trivial.

- **LZ77 encoder**

As already discussed, our LZ77 encoder only required the ability to read a given text file; this required further reading of existing literature, discussed in 3.1.

- **LZ77 decoder**

Our decoder required, as input, a set of LZ77 encoded tuples; we, therefore, required the completion of the LZ77 encoder; this again required further reading of existing literature, discussed in 3.2.

- **Grammar Parser**

Our grammar parser required, as input, a text file containing the grammar; we therefore also required our text file parser in this case. The notation required as input for

grammars is a design choice; both the chosen design choices and methods of implementation are discussed further in 3 and 4.

- **CNF format checker**

Many algorithms, particularly relating to SLP/LZ77 conversion, require the given SLP to be in Chomsky normal form. It was, therefore, necessary to implement software which checked this; this required that we complete the implementation of our grammar parser. We have already discussed the necessary background required for implementation (see 1.9.1).

- **SLP format checker**

It is also necessary to implement an SLP format checker since as discussed previously, all our explored algorithms operate on SLPs. The grammar parser needed to be implemented by this stage in order for this to be possible. We have already discussed the necessary background required for implementation (see 1.9.2).

- **Grammar -> CNF converter**

The only way to confirm if our parsed grammar required a CNF conversion was to have the CNF format checker already implemented. We required further research to understand the algorithm necessary for such a conversion, as discussed during implementation (see 3.3.4.2).

- **Querying compressed strings**

The unoptimised version of the querying compressed strings problem is trivial, as explored in 3.3.4.3. We therefore only needed our grammar parser available to parse the desired SLP to be queried.

- **LZ77 -> SLP converter**

To convert an LZ77 representation to one of an SLP one required both an LZ77 encoded representation and a CNF formatted SLP to be available (we discuss reasons for CNF, in this case in 3.4). Hence, both the LZ77 encoder and an SLP format checker are required. We required further research to understand the algorithm necessary for such a conversion, as discussed during the research section (see 3.4). The converter can also be used to balance SLPs (as discussed in 3.4.1) and hence all SLP related software should be complete.

- **Querying compressed strings (optimised)**

We obtain a significant optimisation for the querying compressed strings problem directly through our implementation of the SLP/LZ77 converter; this is discussed further during the research phase of SLP/LZ77 (see 3.4). We required no further reading beyond the unoptimised solution to the problem, as this is merely an optimisation to the queried SLP as opposed to the querying algorithm itself.

From our research dependency diagram, we were able to construct a priority ordering of which algorithms were going to be studied and built first on our software; excluding any operations that required no further study, the order was as follows:

- 1) LZ77 encoding
- 2) LZ77 decoding
- 3) Grammar -> CFG conversion
- 4) LZ77->SLP conversion
- 5) Compressed Equality Checking

## **2.2.2 Agile Research**

For every new operation, a variety of papers discussing algorithms implementing it were studied. We opted to take an agile approach to manage research progress, similar to 2.3.1. We gave each operation a sprint cycle, typically lasting 1-2 weeks long; within each cycle, we explored a variety of algorithms implementing the operation with the minimum requirement of having a complete understanding of at least one algorithm (the algorithm we would implement). With complete understanding, we build a set of system designs for the researched component, (see 4). The designs consist of UML class diagrams of the given component; UML class diagrams map out the structure of a system by modelling its classes, attributes, operations and relationships between objects [16]. A set of system requirements were also compiled, utilising the MoSCoW method; a prioritisation method used to decide which requirements to complete first, which must come later and which to exclude if necessary [17].

## **2.2.3 Resolving Research Issues**

It was possible that issues during each study or development phase could potentially arise. Issues could stem from multiple sources, such as:

- **Lack of understanding**

Many of the concepts explored were technically challenging and hence, it was possible that understanding aspects of specific algorithms could prove difficult. We made plans before research that ensured the project could continue if this occurred:

- *Discuss issues with supervisor:* by maintaining secure communication with the supervisor of this project (i.e. weekly meetings), we were able to address any issues relating to technical understanding immediately. By selecting a supervisor who specialises in theoretical computer science, we were able to ensure such meetings were productive.
- *Begin work on another operation:* by designing our research dependency diagram in 2.2.1, we were able to begin work on another operation that already had its dependencies satisfied. Besides, the diagram was designed such that we could continue work on any box regardless of its dependencies; this would merely mean that the given operation was not yet production ready and user-facing. We could then address the original issue later, maximising productivity.
- *Explore alternative algorithm:* Many of the operations explored have a variety of solutions. We, therefore, could explore alternative solutions to our operations if necessary. If exploring alternative solutions was necessary, we could seamlessly restructure our research dependency diagram as needed.

- **Additional research requirements (i.e. papers referencing additional technologies and algorithms)**

It was possible that the introductory sections of the papers read did not give a full overview of the knowledge required to implement an algorithm successfully. If this occurred, the following measures helped to combat this:

- *Restructure dependency diagram:* Our research dependency diagram was easily modifiable. If any additional research was required, we could add dependencies anywhere on the diagram without affecting previous work; we could then adjust our priorities and timetable accordingly.

## 2.3 Development methodology

Once we had selected the algorithm of choice for a given operation's implementation, the appropriate measures were taken to ensure the development phase was productive.

### 2.3.1 Agile Development

Due to the nature of the project, the software development aspect needed to be fast and incremental, an agile methodology provided just that [18]. More specifically, we adopted the *Scrum* methodology for its 'inspect and adapt' feedback loops [19] which coped with the potentially fast changing specification throughout the development process.

For each operation's implementation phase, we divided time into short work cadences, known as sprints, typically lasting two weeks long. Sprints were used to integrate the next operation into the current software with the desire for the software to be kept in a potentially shippable (adequately integrated and tested) state at all times [19]. At the end of each sprint, we conducted a short review of the potentially shippable product increment and arranged a meeting with the supervisor to plan the next sprint cycle. The next sprint cycle could be one of the following:

- *Next research cycle (2.2.2)*: begin the study of the next operation, dictated by the research dependency diagram in 2.2.1.
- *Alternative research cycle*: in the case that the cycle failed due to reasons discussed in 2.2.3, an alternative algorithm will be explored, and a new research cycle will begin.
- *Repeat cycle*: If the current cycle was unsuccessful, we repeat the same cycle with identical goals.

### 2.3.2 Agile Testing

Due to the flexible nature of our development cycles, we wrote tests in advance of each development sprint; after each research cycle, once the established inputs/outputs were understood, tests would be written before development began. Thus, after each development sprint, tests could be run immediately and evaluated ahead of the next sprint cycle. Our tests were written using JUnit. JUnit is a Regression Testing Framework used by developers to implement unit testing in Java and, therefore, accelerate programming speed and increase code quality [20].

## 2.4 Software Used

We now outline all software and languages used throughout the project.

- **GitHub**  
A code-hosting platform for collaboration and version control [21].
- **Trello**  
Due to the agile/scrum nature of our project, storyboarding was necessary. Trello provides a series of boards and lists to organise and prioritise projects in a flexible way [22].
- **Eclipse**  
A platform designed for building integrated web and application development tooling [23]. Also used to build JavaDoc.

- **Java**  
The main language used throughout the project; chosen due to the author's experience in the language.

## **2.5 Legal, Social, Ethical, Professional Issues**

- When studying and implementing researched algorithms, it was essential to make sure one has the access rights to any papers used. Also, it is essential to cite these papers wherever used.
- When using software libraries, it is essential to check copyright licensing rules. The only software library used is within 'Apache Commons'. One can find their licensing rules at [24]; it states that we can use such libraries can for non-profit use provided we append the correct declarations to the top of the project.
- There were no social or ethical issues within the project.

## **2.6 Project Timetable**

Throughout terms one and two, a timetable was built and split into blocks based on our research dependency diagram (see 2.2.1). Blocks could be put on hold (due to reasons in 2.2.3) without affecting other blocks.

Term 1:

Task	Start Date	End Date
Read and understand papers relating to SLPs and LZ77	08/10/2018	15/10/2018
Discuss variants of the LZ77 compression scheme, implementing a variant	15/10/2018	20/10/2018
Test implemented LZ77 compression scheme	20/10/2018	22/10/2018
Implement and test a Context Free Grammar parser, with functionality to determine whether it is a valid SLP in CNF.	22/10/2018	05/11/2018
Read papers discussing from LZ77 to SLP	05/11/2018	12/11/2018
Implement an algorithm for restructuring from LZ77 to SLP	12/11/2018	19/11/2018
Test implemented algorithm for restructuring LZ77 to SLP	19/11/2018	26/11/2018
Complete progress report write up	08/10/2018	26/11/2018

Term 2:

Task	Start Date	End Date
Study papers on algorithms for Compressed Equality Checking	07/01/2019	14/01/2019
Implement the chosen algorithm for Compressed Equality Checking	14/01/2019	28/01/2019
Test implemented algorithm for Compressed Equality Checking	28/01/2019	04/02/2019

Complete presentation first draft	21/01/2019	28/01/2019
Complete final presentation	04/02/2019	18/02/2019
Complete project report write up	18/02/2019	04/03/2019

## 3 Research

To build a complete design of our software, we had to conduct all the necessary research as per our research dependency diagram (see 2.2.1). From this, we were able to map out an entire diagram illustrating each component's functionality and requirements; this was an organic process meaning the software designs grew as we conducted further research on new operations. Informal boxes on the research dependency diagram (2.2.1) transformed into formal descriptions of each component (i.e. class diagram objects), illustrating their attributes, operations and relationships amongst other completed components.

### 3.1 LZ77 Encoder

We have previously discussed the LZ77 encoding algorithm in some detail in 1.8.2.1.1; Figure 8 gives basic pseudocode outlining the LZ77 algorithm, as taken from the original LZ77 paper.

```
while (lookAheadBuffer not empty) {
    get a reference (position, length) to longest match;
    if (length > 0) {
        output (position, length, next symbol);
        shift the window length+1 positions along;
    } else {
        output (0, 0, first symbol in the lookahead buffer);
        shift the window 1 character along;
    }
}
```

Figure 8: Original LZ77 algorithm [8].

#### 3.1.1 Alternatives Considered

The LZ77 encoding algorithm we used is not self-referential. In the self-referential version of LZ77, a given factor  $f_i$  can refer to a substring that is a concatenation of a suffix of  $f_1 \dots f_{i-1}$  and a prefix of itself [10]. The decision to use an LZ77 algorithm without self-referencing was taken to accommodate the chosen SLP/LZ77 conversion algorithm, as discussed further in 3.4.

### 3.2 LZ77 Decoder

As discussed in 1.8.2.1.2, the decoder reads a *Reference* and decides whether the *Reference* represents a match [3]. If we find a match, the decoder then uses the *offset* and *stringLength* in the *Reference* to reconstruct the match. Otherwise, it directly outputs the character in the token

[3]. We therefore only had to make minor additions to our *LZ77encoder*; adding an *LZ77decoder* and supplying it with a *decompress* method which outputs the original string.

## 3.3 Grammar Parser

### 3.3.1 Defining syntax

When designing our grammar parser, a specific syntax had to be defined for a user to follow when inputting a grammar. Defining the syntax was entirely our choice provided the syntax obeyed standard grammar rules. They were as follows:

- A *terminal symbol*, a character in the alphabet:
  - o Represented as a single character, i.e. a or A or 2.
- A *nonterminal symbol*, a placeholder for a sequence of terminal symbols generated by the given nonterminal.
  - o Any set of characters surrounded by brackets, i.e. (A) or (ABC) or ('2D).
- A set of *productions*: rules for replacing nonterminal symbols in a string with other non-terminal/terminal symbols.
  - o Productions represented using an arrow (->), a single nonterminal on the left, any number of nonterminals/terminals on the right, i.e. (A)->(B)(C)d.
- A *start symbol*: a particular nonterminal that appears in the initial string generated by the grammar.
  - o Compulsory (S).

The file should contain a set of productions, where each new line represents a new function. Figure 9 represents an example of grammar which should be accepted by our parser.

```
(A) -> (B)(W)
(B) -> c
(W) -> w
(S) -> (A)(B)c
```

Figure 9: Example accepted grammar.

Evaluating this grammar, which would be evaluated from (S), gives *cwcc*

### 3.3.2 Parsing

To read a file, we read each line one at a time, attempting to split the line at the expected '->'. We then attempt to parse the left-hand side as an expected nonterminal, and the right-hand side as a mixture of both terminals and nonterminals. Nonterminals and terminals are identified using the syntax rules stated in 3.3.1. We read each line one character at a time, storing previously seen characters in a buffer until one of the rules is satisfied; we then add either a nonterminal or terminal to the grammar accordingly. Upon violation of a rule, or no rule is satisfied on a given line, the software throws an error.

### 3.3.3 Only Allowing SLPs



As discussed in 2.2.1, the operations we implemented require the user's grammar to be a straight-line program, hence implementing such a check was necessary. Reviewing our definition from 1.9.2:

*A straight-line program over the terminal alphabet  $\Gamma$  is a context-free grammar  $A = (V, \Gamma, S, P)$  ( $V$  is the set of nonterminals,  $\Gamma$  is the set of terminals,  $S \in V$  is the initial nonterminal, and  $P \subseteq V \times (V \cup \Gamma)^*$  is the set of productions) such that the following conditions hold [11]:*

- *For every  $A \in V$ , there exists exactly one production of the form,  $(A, a) \in P$  for,  $a \in (V \cup \Gamma)^*$ .*

*The relation  $\{(A, B) | (A, a) \in P, B \in \text{alph}(a)\}$  is acyclic.*

We, therefore, needed to implement two methods to confirm a given grammar is an SLP:

- **Exactly one production per *NonTerminal***  
Due to our system designs, checking that each *NonTerminal* has exactly one production is unnecessary. Due to our parser, any new productions for a *NonTerminal* overwrite any previous production; this is inherited from the use of a *map*, the structure used to hold all *NonTerminals* of a given *Grammar*.
- **Grammar cycle checker:**  
Cycle checking was implemented using a simple *HashSet*: from the starting *NonTerminal*, (S), we initialise an empty *HashSet*, H, and begin processing the *right* side; upon seeing another *NonTerminal*, we check whether it already exists within H. If the *NonTerminal* does exist in H, then this implies a cycle and therefore this grammar is an invalid SLP.

### 3.3.4 Only Allowing CNF

#### 3.3.4.1 Checking for CNF

As also discussed in 2.2.1, the operations we implemented require, as input, a grammar in Chomsky normal form. A grammar in CNF can be obtained either directly from user input or dealt with by our software; we created designs able to handle both cases. Reviewing our definition from 1.9.1:

*A context-free grammar  $A = (V, \Gamma, S, P)$  ( $V$  is the set of nonterminals,  $\Gamma$  is the set of terminals,  $S \in V$  is the initial nonterminal, and  $P \subseteq V \times (V \cup \Gamma)^*$  is the set of productions) is in Chomsky Normal Form if every production has one of the following forms [11]:*

- $(A, a)$  with  $a \in \Gamma$
- $(A, BC)$  with  $B, C \in V$

Our software should only allow grammars of the form  $(A) \rightarrow (B)(C)$  or  $(A) \rightarrow a$ ; we easily implemented this by counting the number of *NonTerminals* and *Terminals (characters)* on the right-hand side of each production. The following results are accepted:

- Nonterminals = 2 and Terminals = 0
- NonTerminals = 0 and Terminals = 1

For a given *Grammar*,  $G$ , we, therefore, only need to check that every *NonTerminal*'s production in the *map* satisfies one of these conditions. If any *NonTerminal* violates both conditions, the grammar is not in CNF.

### 3.3.4.2 Converting to CNF (Optional)

If the inputted grammar does not satisfy the CNF conditions, we then have the option to convert the grammar. The CNF conversion process is as follows [25]:

- 1) Eliminate start symbol from the right-hand side: If the start symbol ( $S$ ) is on the right-hand side of any production in the grammar, create a new production as  $(S0) \rightarrow (S)$
- 2) Eliminate null, unit and useless productions: If the grammar contains null, unit or useless production rules, we eliminate them.

*In a grammar, a nonterminal symbol ( $A$ ) is a nullable variable if there is a production  $(A) \rightarrow (\text{epsilon})$  or there is a derivation starting at  $A$  that ends up with  $(A) \rightarrow \dots \rightarrow (\text{epsilon})$ .*

*Algorithm for removing null productions:*

- 1) Find all nullable nonterminal
- 2) For each production  $(A) \rightarrow a$ , construct all productions  $(A) \rightarrow x$  where we obtain  $x$  from 'a' by removing one or multiple nonterminals from Step 1.
- 3) Combine the original productions with the result of step 2 and remove all (epsilon)-productions.

*Algorithm for removing unit productions:*

- 1) To remove  $(A) \rightarrow (B)$ , add production  $(A) \rightarrow x$  to the grammar rule whenever  $(B) \rightarrow x$  occurs in the grammar
- 2) Remove  $(A) \rightarrow (B)$  from the grammar.
- 3) Repeat until we remove all unit productions.
- 3) Eliminate terminals from the right-hand side if they exist with other terminals or nonterminals.  
For example,  $(X) \rightarrow x(Y)$  can be decomposed as:  
 $(X) \rightarrow (Z)(Y)$   
 $Z \rightarrow x$
- 4) Eliminate the right-hand side with more than two nonterminals.  
For example,  $(X) \rightarrow (X)(Y)(Z)$  can be decomposed as:  
 $(X) \rightarrow (P)(Z)$   
 $(P) \rightarrow (X)(Y)$

The process of implementing a grammar  $\rightarrow$  CNF converter is optional as such a tool is not strictly required for our overall software to function; the user can merely instead be prompted to give their grammar in the correct form and have the CNF checker (3.3.4.1) confirm their input.

### 3.3.4.3 Querying a Grammar

Assuming the given grammar is an SLP in Chomsky Normal Form, we can interpret this as a binary tree rooted at its start symbol. Modelling our grammar as a binary tree, Figure 10 gives an algorithm for locating the  $i^{\text{th}}$  character of the given grammar.

```

get(i:position,s:startnode) {
    if (i = 1) {
        return s.value
    }
    If (s.left.size <= position)
        return get(i, s.left)
    return right.get(i – s.left.size, s.right);
}

```

Figure 10: Algorithm for retrieving  $i^{\text{th}}$  character of a grammar.

The querying problem is optimised if the given SLP is balanced (making the algorithm  $O(\log n)$  instead of  $O(n)$ ); balancing of SLPs is made possible in 3.4.1.

## 3.4 LZ77 to SLP Conversion

The main algorithm we chose to explore proposes a new grammar-based compression algorithm based on Lempel-Ziv factorisation. As discussed briefly in 1.10, the problem of finding the smallest size SLP generating a given text is NP-complete. The paper, therefore, considers the following problem (an approximation of grammar-based compression) [13]:

*Instance:* given a text  $w$  or length  $n$ ,

*Question:* construct in polynomial time a grammar  $G$  such that  $eval(G) = w$  and the ratio between  $|G|$  and the size of the minimal grammar for  $w$  is *small*.

### 3.4.1 G-Factorization

For simplicity, we assume all grammars are in Chomsky normal form. For a grammar  $G$  generating  $w$  we define the parse-tree  $Tree(G)$  of  $w$  as a derivation tree of  $w$ , in this tree we identify terminal symbols with their parents, in this way every internal node has exactly two (see Figure 11) [13].

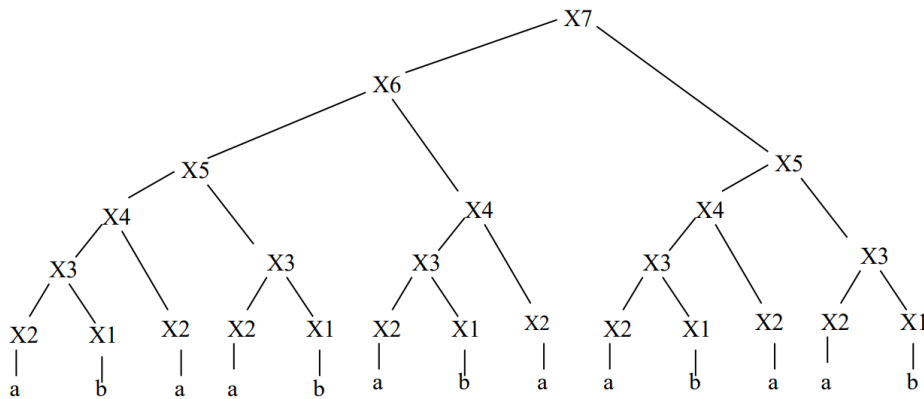


Figure 11: Parse tree: abaabababababab.

We then define a G-factorization as follows: We scan  $w$  from left to right, each time taking as the next G-factor the longest unscanned prefix which is generated by a single nonterminal which has already occurred to the left or a single letter if there is no such nonterminal. The

factors of our LZ77-factorisations and G-factorisations are LZ-factors and G-factors respectively.

Figure 12 gives an example G-factorisation of *abaababaabaab*, where:

LZ77-factorisation =  $f_1 f_2 f_3 f_4 f_5 f_6 = a, b, a, aba, baaba, ab$

G-factorisation =  $g_1 g_2 g_3 g_4 g_5 g_6 = a, b, a, ab, aba, abaab$

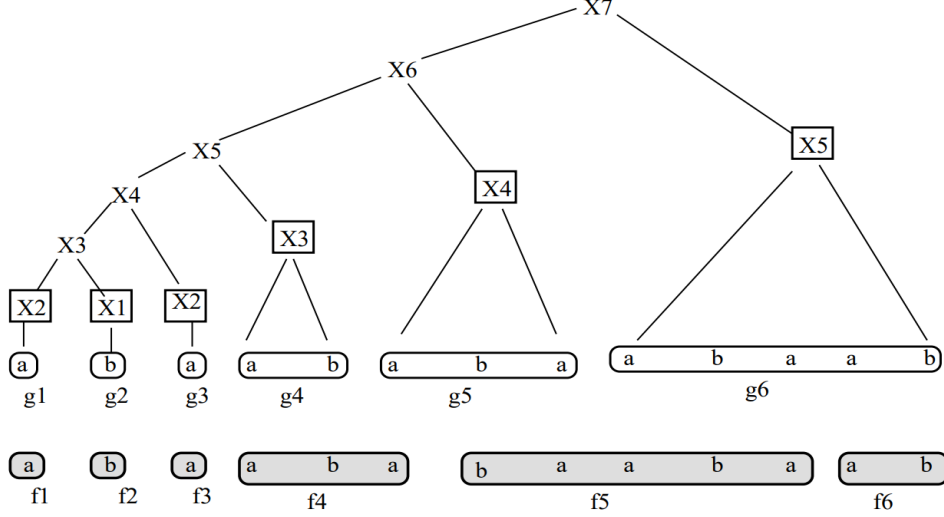


Figure 12: LZ-factorization (shaded) is shown below G-factorization

Proven in [13], we observe that the number of LZ-factors is not greater than the number of G-factors. Additionally, for each string  $w$  and its grammar-based compression  $G$ ,  $|LZ(w)| \leq |G|$ .

The importance of *balanced grammars* is discussed in 3.3.4.3. We can take any grammar  $G$  generating a single text, produce the G-factorization and then transform it into a balanced grammar the same way we do for LZ-factorisation, discussed in 3.4.

### 3.4.2 AVL-Grammars

To appreciate *balanced grammars*, [13] introduces a new type of grammar: AVL-grammars. They correspond naturally to AVL-trees. We usually use AVL-trees in the context of binary search trees; here they are used in the context of storing in the leaves the consecutive symbols of the input string  $w$  [13].

We use the standard AVL-trees:

- For each node  $v$ , the balance of  $v$ , denoted by  $bal(v)$  is the difference in height between the left and right subtrees rooted at  $v$ .
- A tree  $T$  is *AVL-balanced* if and only if  $|bal(v)| \leq 1$  for each node  $v$ .
- We say a grammar  $G$  is *AVL-balanced* if  $Tree(G)$  is *AVL-balanced*.
- Denote  $height(G)$  the height of  $Tree(G)$  and  $height(A)$  the height of the parse tree rooted at nonterminal  $A$ .
- If the grammar  $G$  is *AVL-balanced*, then  $height(G) = O(\log(n))$  [13].

When adopting AVL-trees for the *AVL-balanced grammar* case, for each nonterminal  $A$ , additional information about the balance of  $A$  is kept.  $bal(A)$  is the balance of the node corresponding to  $A$  in the tree  $Tree(G)$ . We do not define the balance of terminal symbol nodes, they are identified with their fathers (nonterminals generating single symbols). Each nonterminal  $B$  is a leaf of  $Tree(G)$  and have  $bal(B) = 0$  [13].

The essential operation is the *concatenation* of sequences of leaves of two trees.

### 3.4.2.1 Concatenation algorithm

The following lemma is crucial to the construction of SLPs from an LZ77 factorisation:

*Lemma: Assume  $A, B$  are two nonterminals of AVL-balanced grammars. We therefore can construct in  $O(|\text{height}(A) - \text{height}(B)|)$  time an AVL-balanced grammar  $G = \text{Concat}(A, B)$  where  $\text{eval}(G) = \text{eval}(A) \cdot \text{eval}(B)$ , by adding only in  $O(|\text{height}(A) - \text{height}(B)|)$  nonterminals.*

We now look at the concatenation algorithm for two AVL-balanced trees  $X, Y$  with roots in  $A$  and  $B$ . The algorithm was first explored by Knuth [26] when exploring the concatenation of linear lists. The AVL-trees contain symbols only in the leaves, so to concatenate two trees we do not need to delete the root of one of them, as this implies a costly restructuring.

The algorithm is as follows: Assume that  $\text{height}(X) \geq \text{height}(Y)$ , the other case is symmetrical. We follow the rightmost branch of  $X$ , the heights of nodes decrease each time at most by 2 (all trees are balanced from previous concatenations). We stop at a node  $v$  such that  $0 \leq \text{height}(v) - \text{height}(y) \leq 1$ . We create a new node,  $v'$ , such that its father is the father of  $v$  and its sons are  $v, B$ , where  $B$  is the root of  $Y$ . Figure 13 illustrates an example concatenation.

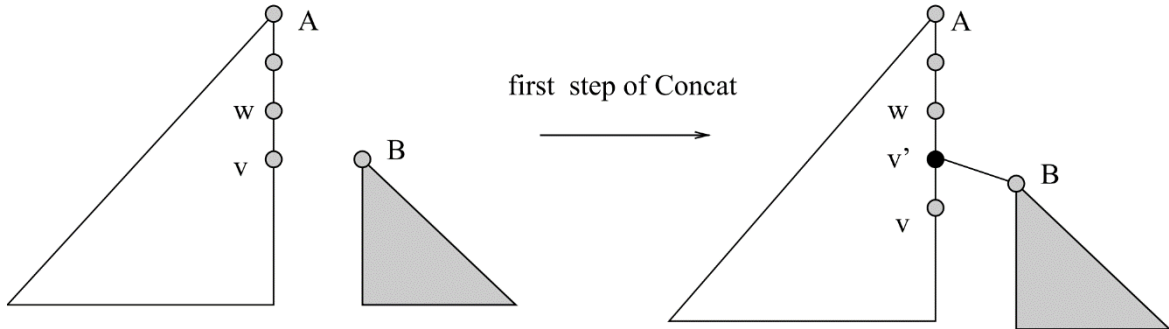


Figure 13:  $\text{concatenate}(A, B)$ . The edge  $(w, v)$  is split into  $(w, v')$  and  $(v', v)$  where  $0 \leq \text{height}(v) - \text{height}(B) \leq 1$ . The node  $v'$  is a new node. The corresponding grammar productions are added [11].

When the above example of concatenation takes place, the resulting tree can be unbalanced by at most 2. [13] provides suitable rotations to solve this problem, as seen in Figure 14.

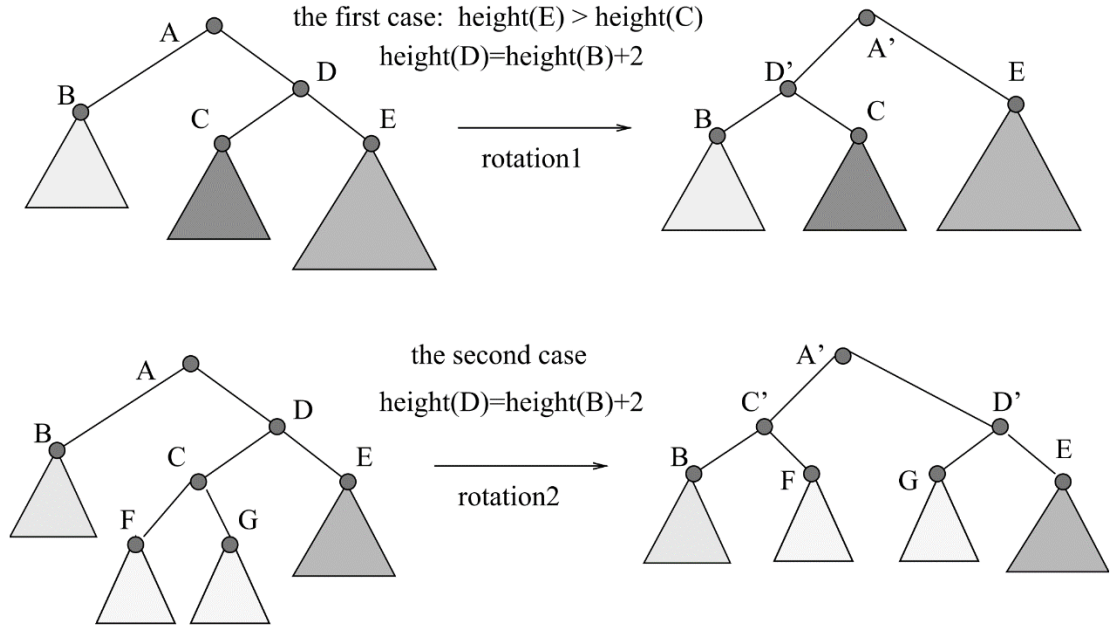


Figure 14: Concatenation cases requiring rotation [11].

In the above example, all nodes but root  $A$  are well balanced, which is overbalanced to the right. A single rotation in  $Tree(G)$  corresponds to a local change of constant number productions and creation of new nonterminals. The root now becomes balanced, but its father/some node above can still be unbalanced, and hence we continue processing upwards to the root of  $Tree(G)$  [13].

The concatenation algorithm for AVL-trees can be applied to the parse trees and automatically extended to AVL-grammars.

### 3.4.3 Construction of Small Grammar-Based Compression From LZ77

Assuming we already have an LZ77-factorisation of  $f_1 f_2 \dots f_k$ , we can convert it into a straight-line program (a grammar) whose size increases by a logarithmic factor [13]. The construction is possible if we assume we have an LZ77-factorisation  $w = f_1 f_2 \dots f_k$  and have already constructed an *AVL-balanced* grammar of size  $O(i(\log(n)))$  for the prefix  $f_1 f_2 \dots f_{i-1}$ . If  $f_i$  is a terminal generated by a nonterminal  $A$  then we set  $G = \text{concatenate}(G, A)$ . Otherwise, we locate the *grammar decomposition* of the factor  $f_i$ .

A *grammar decomposition* of the segment  $f_i$  is the segment corresponding to  $f_i$  in the prefix  $f_1 f_2 \dots f_{i-1}$ . Since  $G$  is balanced, we can find a logarithmic number of nonterminals  $S_1 S_2 \dots S_{t(i)}$  such that  $f_i = \text{eval}(S_1). \text{eval}(S_2). \dots \text{eval}(S_{t(i)})$  [13]. Figure 15 gives an example.

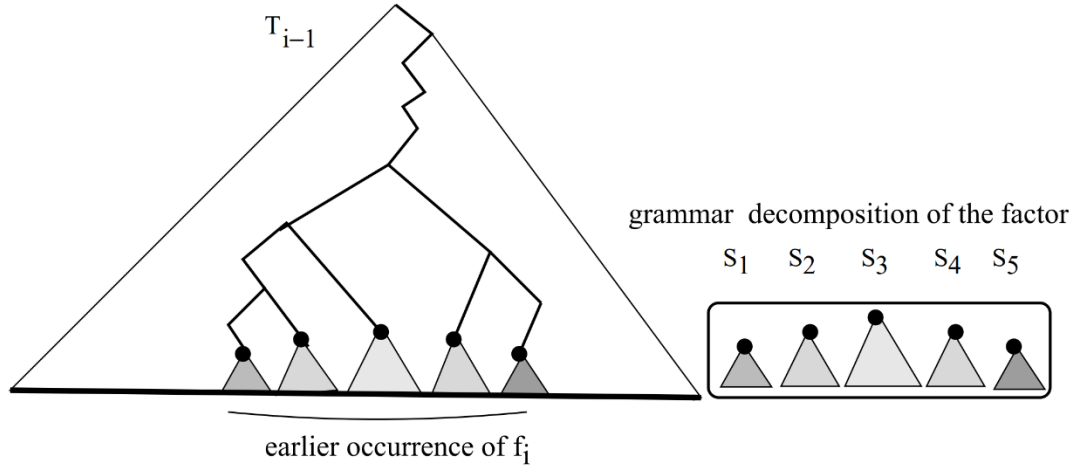


Figure 15: Grammar decomposition of  $f_i$ .

We concatenate the parts of the grammar corresponding this factor with  $G$  using the concatenation algorithm discussed in 3.4.2.1. We assume the first  $|\Sigma|$  nonterminals correspond to letters in our alphabet and hence exist at the beginning. We initialize  $G$  to the grammar generating the first symbol of  $w$  and containing all nonterminals for terminal symbols. The algorithm begins with the computation of the LZ77-factorization, using the methods discussed in 3.1. Figure 16 outlines the entire algorithm.

```

Construct-Grammar( $w$ );  $\{|w| = n\}$ 
  compute LZ77-factorization  $f_1 f_2 f_3 \dots f_k$ 
  if  $k > n/\log(n)$  then return trivial  $O(n)$  size grammar
  else
    for  $1 \leq i \leq k$ :
      (1) Let  $S_1 S_2 \dots S_{t(i)}$  be grammar decomposition of  $f_i$ ;
      (2)  $H = \text{concatenate}(S_1 S_2 \dots S_{t(i)})$ ;
      (3)  $G = \text{concatenate}(G; H)$ ;
  return  $G$ ;

```

Figure 16: LZ77-factorisation to grammar algorithm.

Due to AVL-grammar characteristics (see 3.4.2), we have  $t(i) = O(\log(n))$ . Therefore, the number of two-argument concatenations needed to implement a single step (2) is  $O(\log(n))$ , each of them adding  $O(\log(n))$  nonterminals [13]. Steps (1) and (3) can also be done in  $O(\log(n))$  time as the height of an AVL-grammar is logarithmic.

*Example:*

Consider the input: **abaaab**

The LZ77-factorisation of this string is:  $a, b, aa, ab$

Taking each LZ77-factor in order:

- Loop 1 (a): Since all symbols in the alphabet already have nonterminals existing, call it nonterminal 1, we merely concatenate() the nonterminal to our empty grammar  $G$ .
- Loop 2 (b): Again, the single symbol 'b' already has a nonterminal producing it (call it nonterminal 2), and hence we concatenate( $G, 2$ ), creating nonterminal 3.

- *Loop 3 (aa): The grammar decomposition for 'aa' can be found by concatenating to two existing nonterminals producing 'a', creating nonterminal  $4 = \text{concatenate}(1,1)$ . We then concatenate this new nonterminal to our existing grammar  $G$ , creating nonterminal  $5 = \text{concatenate}(G,4)$ .*
- *Loop 4 (ab): The grammar decomposition for 'ab' can be found through an existing nonterminal in  $G$  already producing it, 3. We duplicate this nonterminal and concatenate() it to  $G$ , creating nonterminal  $6 = \text{concatenate}(G,3)$ .*

Figure 17 gives the final output of  $\text{Construct-Grammar}(\text{abaab})$ .

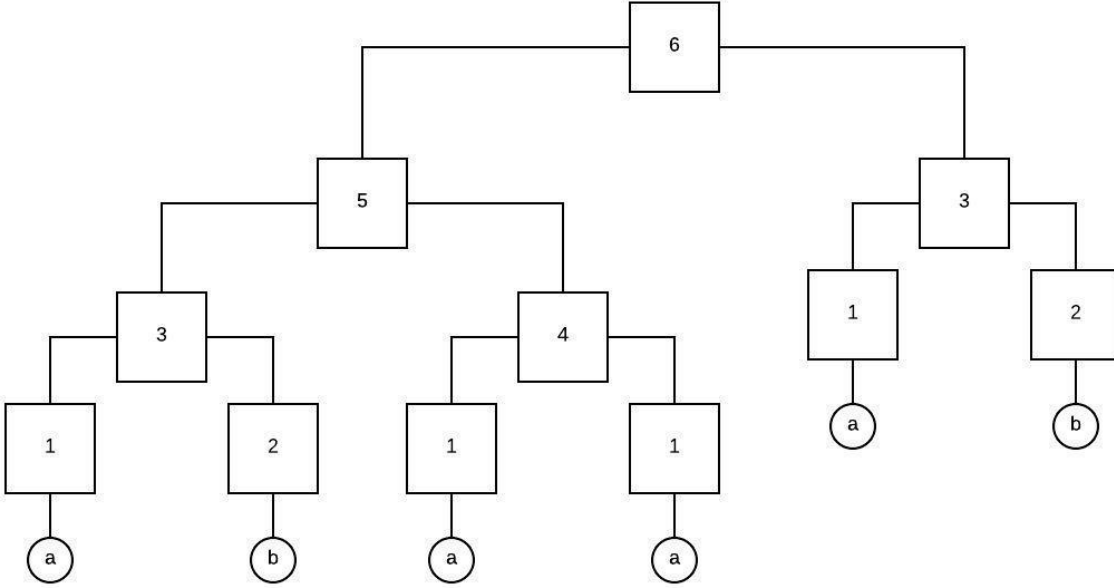


Figure 17: Output of  $\text{Construct-Grammar}(\text{abaab})$ .

## 3.5 Compressed Equality Checker

From our previously established understanding of grammars, compressed equality checking can be done using a simple decompress-and-compare strategy; such a solution is very inefficient and, as discussed in 1.10, takes exponential time to compute  $\text{eval}(A) = \text{eval}(B)$ .

In [14], although not explicitly stated, we obtain a polynomial time algorithm for compressed equality checking through an efficient data structure  $D$  which supports the following operations:

- Add a string consisting of the symbol  $a$  to  $D$ .
- For two strings  $x$  and  $y$  in  $D$ , add the string  $xy$  to  $D$ . The strings  $x$  and  $y$  remain in  $D$ .
- Check whether two strings  $x$  and  $y$  in  $D$  are identical.

The main idea is to compute for each variable a signature, which is a small number, that allows us to run an equality test in constant time by merely comparing two signatures. A signature corresponds to a nonterminal of an SLP. The paper proposes two solutions, both a randomised and deterministic. We now give a formal overview of both solutions.

### 3.5.1 Overview



The data structure maintains, dynamically, a family  $F$  of sequences over a universe  $U$ . Let  $s_1, s_2$  be sequences,  $a_j \in U$  for  $j = 1, \dots, n$ , and let  $i$  be an integer. The data structure supports the following operations on an initially empty family of sequences:

- **Makesequence**( $s, a_1$ ): Creates the sequence  $s_1 = a_1$ .
- **Equal**( $s_1, s_2$ ): Returns true if  $s_1 = s_2$
- **Concatenate**( $s_1, s_2, s_3$ ): Creates the sequence  $s_3 = s_1 s_2$ , without destroying  $s_1$  and  $s_2$ .
- **Split**( $s_1, s_2, s_3, i$ ): Creates two new sequences  $s_2 = a_1 \dots a_i$  and  $s_3 = a_{i+1} \dots a_n$  without destroying  $s_1 = a_1 \dots a_n$ .

### 3.5.1.1 Randomised Solution

In the randomised solution, we compute for each sequence  $s$  a unique signature  $sig(s)$ , an integer. This signature is used to perform equality checking. The signature of a sequence  $s = a_1 a_2 \dots a_n$  with  $a_i \neq a_{i+1}$  for all  $i, 1 \leq i < n$ , is computed as follows:

The sequence  $s$  is first broken into subsequences (blocks) of length at least 2 and expected length  $\leq \log(n)$ . Then, each subsequence, say  $b = a_i \dots a_j$ , is replaced by a single integer which is computed by means of a pairing function  $p$ , i.e.,  $b$  is replaced by  $p(a_i, p(a_{i+1}, \dots, p(a_{j-1}, a_j) \dots))$ . Afterward, we apply the same rules again to the shrunk sequence until the sequence has length 1 (our signature). Randomization is used to break a sequence into subsequences. For each element of the sequence, a real number is chosen and subsequences begin at local minima. Therefore, subsequences have length at least 2 (except maybe the first).

### 3.5.1.2 Deterministic Solution (optional)

In the deterministic solution, we replace the randomised approach to breaking a sequence into subsequences by a deterministic one. It is based on an algorithm for three-colouring rooted trees. We generate subsequences of length at most 4 and decide for each index  $i$  whether  $a_i$  starts a new subsequence by looking at  $O(\log^*(m))$  neighbours of  $a_i$ .

## 3.5.2 Subsequence Decomposition

### 3.5.2.1 Randomised Decomposition

#### 3.5.2.1.1 Assigning Priorities

Let  $U$  be a universe and  $s = a_1 \dots a_n$  with  $a_i \in U$  and  $a_i \neq a_{i+1}$  for all  $i, 1 \leq i < n$ . Each element  $a \in U$  is assigned a random priority  $priority(a) \in [0,1]$ .

#### 3.5.2.1.2 Decomposition Rule

An element  $a_i$  of  $s$  is a local minimum of  $s$  if it has a successor in  $s$  and its priority is a local minimum in the sequence  $priority(a_1) \dots priority(a_n)$  of priorities corresponding to  $s$ . Every local minimum is marked, then, at every marked position (and the first) a block starts. It ends just before the next marked position (with the last block ending at position  $n$ ).

### 3.5.2.2 Deterministic Decomposition (optional)

As stated previously, the deterministic decomposition algorithm is essentially a sequential version of the *three-colouring technique for rooted trees* [27], and we consider it as a constructive proof of the following [14]:

*For every integer  $N$  there is a function  $f: [-1 \dots N - 1]^{\log^*(N+11)} \rightarrow \{0,1\}$  such that for every sequence  $a_1 \dots a_n \in [0 \dots N - 1]^*$  with  $a_i \neq a_{i+1}$  for all  $i$  with  $1 \leq i < n$  the sequence  $d_1 \dots d_n \in \{0,1\}^*$  defined by  $d_i = f(\tilde{a}_{i-\log^*(N-6)}, \dots, \tilde{a}_{i+4})$ , where  $\tilde{a}_j = a_j$  for all  $j$  with  $1 \leq j \leq n$  and  $\tilde{a}_j = -1$  otherwise, satisfies:*

- $d_i + d_{i+1} \leq 1$  for all  $1 \leq i < n$ ,
- $d_i + d_{i+1} + d_{i+2} + d_{i+3} \geq 1$  for all  $1 \leq i \leq n - 3$

The sequence  $d_1 \dots d_n$  is used to decompose the sequence  $a_1 \dots a_n$  by starting a new subsequence at index  $i = 1$  and at every index  $i$  with  $d_i = 1$ ; by doing this, no subsequence has length  $> 4$  and that all but the first and last subsequence have length at  $\geq 2$ .

Let  $s = a_1 \dots a_n$  with  $a_i \in [0 \dots N - 1]$  and  $a_i \neq a_{i+1}$ . A  $k$ -colouring of a list is an assignment  $C: \{a_1, \dots, a_n\} \rightarrow \{0, \dots, k - 1\}$ . A valid colouring is a colouring such that no two adjacent elements have the same colour.

Informally, we first compute a valid  $\lceil \log N \rceil$  colouring. We then replace the elements in the list by their colours and consider this set to be the new universe; we then iterate the colouring procedure. After  $O(\log^* N)$  steps we get a valid six-colouring and then further reduce by a difference procedure to obtain a three-colouring.

#### 3.5.2.2.1 Six Colouring Algorithm

Identify each  $a_i$  and its colour with its binary representation (containing  $\lceil \log N \rceil$  bits). The bits are indexed from 0 and the  $j^{th}$  bit of the representation of a colour of the element  $a_i$  is given by  $C_i(j)$ . Figure 18 takes as input the sequence  $s = a_1 \dots a_n$  and computes a six-colouring for  $s$ . We use  $C_i$  to denote the colour of  $a_i$  and  $N_c$  to denote the number of used colours.

**Six-Colours**( $a_1 \dots a_n : s$ ):

```

 $N_c = N;$ 
for all  $i \in \{1, \dots, n\}$  do
     $C_i = a_i$ 
while  $N_c > 6$  do
     $C_1 = C_1(0)$ 
    for all  $i \in \{2, \dots, n\}$  do
         $j_i = \min\{j \mid C_i(j) \neq C_{i-1}(j)\}$ 
         $b_j = C_i(j_i)$ 
         $C'_i = 2j_i + b_i$ 
     $N_c = \max\{C_i \mid i \in \{1, \dots, n\}\} + 1$ 

```

Figure 18: Six-colours produces a valid six-colouring of a list  $a_1 \dots a_n$  where  $a_i \in \{0, \dots, N - 1\}$  for all  $i, 1 \leq i \leq n$  in  $O(n \log^* N)$  time [14].

#### 3.5.2.2.2 Three Colouring Algorithm

Computing a valid three-colouring from a given six-colouring is simple. We replace each colour  $C_i \in \{3, 4, 5\}$  of an element  $a_i$  by the smallest colour in  $\{0, 1, 2\}$  which is not assigned to one of its neighbours. Figure 19 gives the details.

```

Three-Colours( $a_1 \dots a_n : s$ )
  Six-Colours ( $a_1 \dots a_n$ )
   $c_0 = \infty$ 
   $c_{n+1} = \infty$ 
  for  $c = 3$  to 5 do
    forall  $i \in \{1, \dots, n\}$  do
      if  $C_i = c$  then
         $C_i = \min(\{0, 1, 2\} - \{C_{i-1}, C_{i+1}\})$ 

```

Figure 19: Three-colours produces a valid three-colouring of a list  $a_1 \dots a_n$  where  $a_i \in \{0, \dots, N-1\}$  for all  $i, 1 \leq i \leq n$  in  $O(n \log^* N)$  time [14].

### 3.5.2.2.3 Decomposition Rule

For any sequence  $a_1 \dots a_n$  we define the sequence  $d_1 \dots d_n$  by computing a valid three-colouring using the methods discussed in 3.5.2.2.2. We then set  $d_i = 1$  iff the colour of  $a_i$  (which is now  $\in \{0, 1, 2\}$ ) is a local maximum in the sequence of colours,  $d_i = 0$  otherwise.

## 3.5.3 Representation of Sequences

In this section, we explain how to assign unique signatures to sequences and how we represent sequences. A signature is a small integer, after  $m$  operations, there is no signature exceeding  $m^3$  [14]. Each sequence  $s$  can be uniquely written as  $a_1^{l_1} \dots a_n^{l_n}$  with  $a_i \neq a_{i+1}$  for all  $i, 1 \leq i < n$  and for all  $l_i$ , a positive integer, where  $a_i^{l_i}$  denotes a subsequence of  $l_i$  repetitions of  $a_i$ ; this representation is very similar to run-length encoding, as discussed briefly in 1.8.1.

Informally, we assign the sequence  $s = a_1^{l_1} \dots a_n^{l_n}$  a signature in the following way: each element  $a_i \in U$  gets a signature (performed by  $\overline{sig}(s)$ , later). To eliminate repetitions, each power  $a_i^{l_i}$  ( $1 \leq i \leq n$ ) also has a signature assigned. We then compute a subsequence decomposition of the sequence  $sig(a_1^{l_1}) \dots sig(a_n^{l_n})$  using the methods discussed in 3.5.2. For each subsequence, we then compute a signature by repeated application of a pairing function i.e. pairs of signatures are encoded by a new signature. The resulting sequence is denoted by  $shrink(s)$ . We then apply  $shrink(s)$  to the whole procedure, the sequence of subsequence encodings; this is repeated until the original sequence is reduced to a single integer (its signature).

We now formally give details, denote the current set of signatures to be  $S = [0..max\_sig]$ .  $S$  is the disjoint union  $S_U \cup S_P \cup S_R$  and each element in  $S$  encodes one of the following [14]:

- An element in  $U: S_U \rightarrow U$ .
- A pair in  $S \times S: S_P \rightarrow \{(a, b) \mid a, b \in S \text{ and } a \neq b\}$
- A power in  $S \times \mathbb{N}_{\geq 2}: S_R \rightarrow \{(a, i) \mid a \in S \text{ and } i \in \mathbb{N}, i \geq 2\}$ .

The inverse functions  $u$ ,  $p$ , and  $r$  are maintained as dictionaries. In the randomised scheme (see 3.5.2.1), every element  $s \in S$  that encodes a power is also given a random priority  $priority(s) \in [0, 1]$ .

We define the signature  $sig(s)$  [14] of a sequence  $s = a_1^{l_1} \dots a_n^{l_n}$  with  $a_i \neq a_{i+1}$  for all  $i, 1 \leq i < n$  and  $n \geq 1$  below.  $shrink(s)$  and  $\overline{sig}(s)$  are defined afterwards [14].

$$\text{sig}(s) = \begin{cases} \overline{\text{sig}}(a_1) & \text{if } n = 1 \text{ and } l_1 = 1, \\ r^{-1}((a_1, l_1)) & \text{if } n = 1, l_1 > 1, \text{ and } (a_1, l_1) \in \text{range}(r), \\ \text{maxsig} + 1 & \text{if } n = 1, l_1 > 1, \text{ and } (a_1, l_1) \notin \text{range}(r), \\ \text{sig}(\text{shrink}(s)) & \text{if } n > 1. \end{cases}$$

Before we define  $\text{shrink}(s)$ , . Let  $n > 1$ , then function for eliminating powers,  $\text{elpow}(s)$ , is defined by [14]:

$$\text{elpow}(s) = \text{sig}(a_1^{l_1}) \dots \text{sig}(a_n^{l_n})$$

In  $\text{elpow}(s)$ , every power is replaced by its signature (using  $\text{sig}(s)$ , defined above). Denote  $\text{elpow}(s)$  by  $g_1 \dots g_n$  where  $g_i = \text{sig}(a_i^{l_i})$  for all  $i$ ,  $1 \leq i \leq n$ . We then decompose this sequence into subsequences (see 3.5.2). Let  $b_1 \dots b_k$  denote the subsequence decomposition of  $\text{elpow}(s)$ . We then define  $\text{shrink}(s)$  by [14]:

$$\text{shrink}(s) = \overline{\text{sig}}(b_1) \dots \overline{\text{sig}}(b_k)$$

We now define  $\overline{\text{sig}}$  [14], it is defined for all sequences  $s = a_1 \dots a_n$  with  $a_i \neq a_{i+1}$  for all  $i$ ,  $1 \leq i < n$ ;  $\text{sig}(s)$  is now completely defined.

$$\overline{\text{sig}} = \begin{cases} a_1 & \text{if } n = 1 \text{ and } a_1 \in S, \\ u^{-1}(a_1) & \text{if } n = 1, a_1 \in U, \text{ and } a_1 \in \text{range}(u), \\ \text{maxsig} + 1 & \text{if } n = 1, a_1 \in U, \text{ and } a_1 \notin \text{range}(u), \\ p^{-1}((\overline{\text{sig}}(a_1), \overline{\text{sig}}(a_2))) & \text{if } n = 2 \text{ and } (\overline{\text{sig}}(a_1), \overline{\text{sig}}(a_2)) \in \text{range}(p), \\ \text{maxsig} + 1 & \text{if } n = 2 \text{ and } (\overline{\text{sig}}(a_1), \overline{\text{sig}}(a_2)) \notin \text{range}(p), \\ \overline{\text{sig}}(a_1, \overline{\text{sig}}(a_2, \dots, \overline{\text{sig}}(a_{n-1}, a_n) \dots)) & \text{if } n > 2. \end{cases}$$

### 3.5.4 Storing Sequences

Let  $s = a_1^{l_1} \dots a_n^{l_n}$ , let  $\text{elpow}(s) = \text{sig}(a_1^{l_1}) \dots \text{sig}(a_n^{l_n})$ , let  $b_1 \dots b_k$  be the sequence of subsequences of  $\text{elpow}(s)$  and let  $\text{shrink}(s) = \overline{\text{sig}}(b_1) \dots \overline{\text{sig}}(b_k)$ .

We represent a sequence  $s$  by a list of sequences  $\bar{s} = (\tau_0 \dots \tau_{2t})$  where  $\tau_0 = s$  and for all  $i$ ,  $1 \leq i \leq t$ ,  $\tau_{2t-1} = \text{elpow}(\tau_{2t-2})$  and  $\tau_{2t} = \text{shrink}(\tau_{2t-2})$ . In order the support our desired operations, we store each  $\tau_j$  as a balanced binary tree  $T_j$  in such a way that an in-order traversal of  $T_j$  returns  $\tau_j$  [14]. Each node  $v$  contains:

- An element  $a$  of  $\tau_j$ ,
- The size of the subtree rooted at  $v$
- The mark of the element  $a$  if  $j$  is odd.

$\bar{s}$  is maintained as a linked list of the roots of the trees  $T_j$ .  $F$  is maintained as a linked list of the heads of these lists.

### 3.5.5 Operations

#### 3.5.5.1 Equality

Let  $s_1$  and  $s_2$  be sequences.  $\text{Equal}(s_1, s_2)$  can be implemented by checking whether  $\text{sig}(s_1) = \text{sig}(s_2)$ , returning *true* if this is the case. This can be done in constant time.

### 3.5.5.2 Making Sequences

For  $a \in U$ ,  $\text{Makesequence}(s, a)$  creates a single node binary tree which represents  $s = \text{sig}(a)$ . It therefore retrieves  $\text{sig}(a)$  by evaluating  $u(a)$  if  $a \in \text{range}(u)$ . Otherwise, it assigns a new signature and  $u$  is extended.

### 3.5.5.3 Concatenate

#### 3.5.5.3.1 Randomised Concatenation

The input is the hierarchical representations of sequences  $s_1$  and  $s_2$  (see 3.5.3). We need to compute the hierarchical representation of  $s_3 = s_1 s_2$ . *Lemma 6* of [14] shows that for each tree of  $\bar{s}_3$ , only a small middle part has to be recomputed.

Figure 20 lays out the concatenation algorithm in the randomised case.

**RanConcatenate( $s_1, s_2, s_3: s$ )**

    Compute  $T_{s_3}$  by joining  $T_{s_1}$  and  $T_{s_2}$

    Compute  $T_{\text{elpow}(s_3)}$  by joining  $T_{\text{elpow}(s_1)}$  and  $T_{\text{elpow}(s_2)}$  (in the case that  $a_l = a_{l+1}$ , recompute the corresponding element of  $\text{elpow}(s_3)$ ).

    Let  $s = s_3$ , let  $z$  be such that  $g$  encodes the subsequence of  $s_3$  containing  $a_l$  and let  $\bar{s}_3$  be an empty list.

    while  $|s| > 1$  do

        Append  $s$  and  $\text{elpow}(s)$  at the end of the list  $\bar{s}_3$

        Choose/retrieve the priorities of the subsequence of  $g(s)$  and compute the marks of in  $T_{\text{elpow}(s)}$  accordingly.

        Assign  $\text{shrink}(s)$  to  $s$  as discussed above.

        Compute  $T_s$ . If  $|s| > 1$ , then compute  $T_{\text{elpow}(s)}$  and update  $z$ .

    Append  $s$  to the end of  $\bar{s}_3$ .

Figure 20: Randomised concatenation algorithm [14].

#### 3.5.5.3.2 Deterministic Concatenation (optional)

We mainly implement the deterministic operations in the same way as the randomised operations; the main difference is the computation of the block decomposition (see 3.5.2.2) [14].

**DetConcatenate**( $s_1, s_2, s_3:s$ )

Compute  $T_{s_3}$  by joining  $T_{s_1}$  and  $T_{s_2}$

Compute  $T_{elpow(s_3)}$  by joining  $T_{elpow(s_1)}$  and  $T_{elpow(s_2)}$  (in the case that  $a_l = a_{l+1}$ , recompute the corresponding element of  $elpow(s_3)$ ).

Let  $s = s_3$ , let  $z$  be such that  $g$  encodes the subsequence of  $s_3$  containing  $a_l$  and let  $\bar{s}_3$  be an empty list.

while  $|s| > 1$  do

Append  $s$  and  $elpow(s)$  at the end of the list  $\bar{s}_3$

Run *three-colours*( $g$ ) and change the marks accordingly.

Assign *shrink*( $s$ ) to  $s$  as discussed above.

Compute  $T_s$ . If  $|s| > 1$ , then compute  $T_{elpow(s)}$  and update  $z$ .

Append  $s$  to the end of  $\bar{s}_3$ .

Figure 21: Deterministic concatenation algorithm [14].

### 3.5.5.4 Split

In our rendition of split, we simply compute signatures for sequences  $s_2 = a_1 \dots a_i$  and  $s_3 = a_{i+1} \dots a_n$  without destroying  $s_1 = a_1 \dots a_n$ . Optimisations for the split function can be found in [14]; they were not explored due to time constraints.

## 4 Design

### 4.1 LZ77 Encoder

#### 4.1.1 System Requirements

The following system requirements have been established for the LZ77 encoder:

Requirement Type	Requirement	MoSCoW
Functional	The user can load a desired text file into the program.	M
Functional	The user's inputted text file can be compressed to a list of compressed tuples	M
Functional	The compressed tuples can be written to the desired output file.	S

#### 4.1.2 Software Design

Figure 22 gives a basic UML class diagram representing our LZ77 encoder.

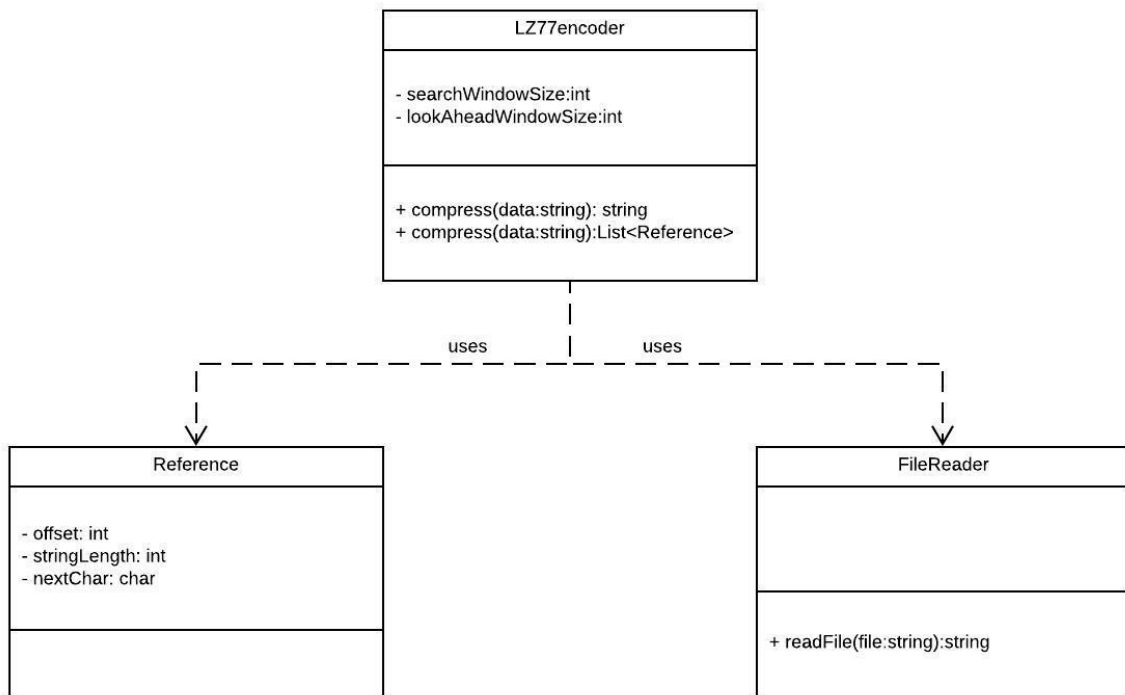


Figure 22: UML class diagram for LZ77 encoder.

#### - **FileReader**

Our LZ77 encoder requires, as input, a file to be read as specified by the user. Its *readFile* method is a utility method used to read data from a given file, outputting its contents as a string.

#### - **LZ77 encoder**

As discussed previously, our LZ77 encoder has two key attributes: a *searchWindowSize* and a *lookAheadWindowSize*, both attributes are user assigned, with getters/setters omitted from all diagrams to remove unnecessary verbosity.

We give two *compress* methods which both utilise the encoding algorithm discussed in 3.1. The first *compress* method outputs the contents of the compressed file when a user only wants to use the tool for educational purposes. The second *compress* method will output the expected encoded data; a list of *References* which is utilised internally by other operations.

#### - **Reference**

As discussed in 1.8.2.1.1, our encoded output will be a sequence of tuples consisting of:

- *offset*: The location of the beginning of the match in the search window relative to the current location
- *stringLength*: The length of the match.
- *nextChar*: The first non-matching character in the lookahead window following the characters that match.

The *LZ77encoder* utilises our *Reference* object, as depicted by our UML diagram.

## 4.2 LZ77 Decoder

## 4.2.1 System Requirements

The following system requirements have been established for the LZ77 encoder:

Requirement Type	Requirement	MoSCoW
Functional	The user can load a desired LZ77 compressed file into the program.	M
Functional	The user's inputted LZ77 compressed file can be decompressed into the original message.	M
Functional	A decompressed message can be written to the desired output file.	S

## 4.2.2 Software Design

Figure 23 gives the modified UML class diagram.

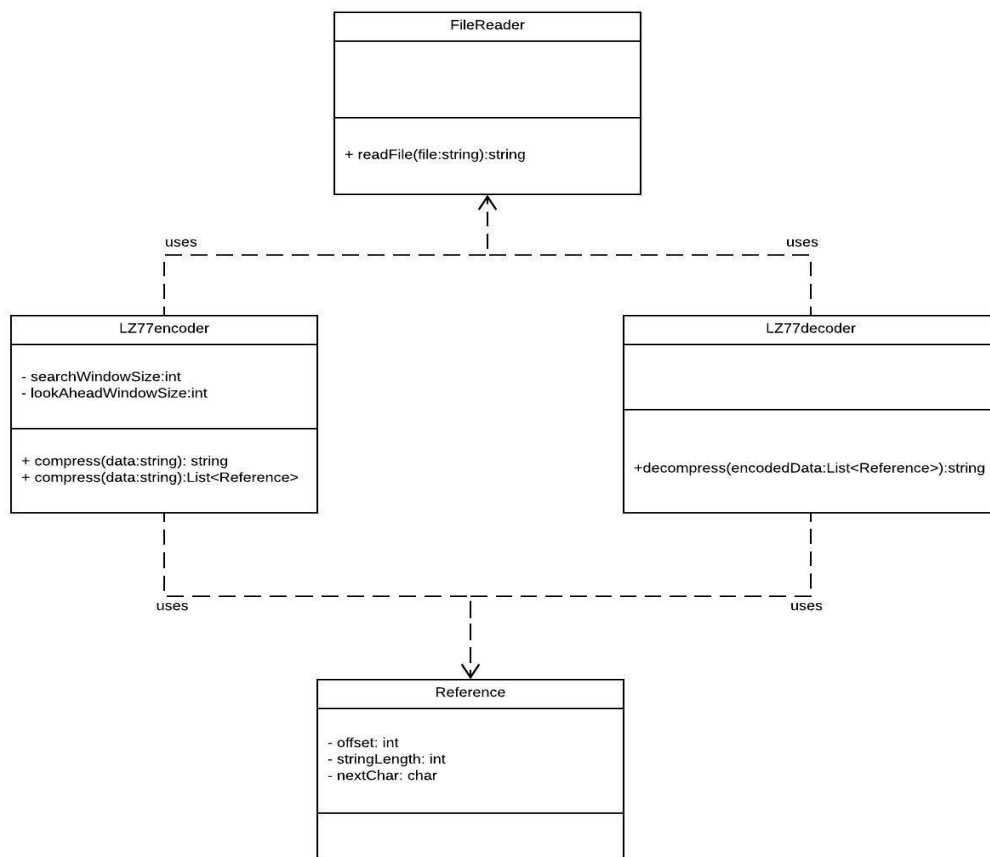


Figure 23: UML class diagram for LZ77 encoder/decoder.

## 4.3 Grammar Parser



### 4.3.1 System Requirements

The following system requirements have been established for the LZ77 encoder:

Requirement Type	Requirement	MoSCoW
Functional	The user load a desired text file into the program.	M
Functional	The program can correctly parse a user loaded grammar.	M
Functional	The program can correctly identify a grammar in Chomsky Normal Form.	M
Functional	The program can correctly identify a straight-line program.	M
Functional	The program can convert a grammar into Chomsky Normal Form.	S

### 4.3.2 Software Design

Figure 24 shows a UML class diagram for our *GrammarParser*, utilising the same *FileReader* class as made previously.

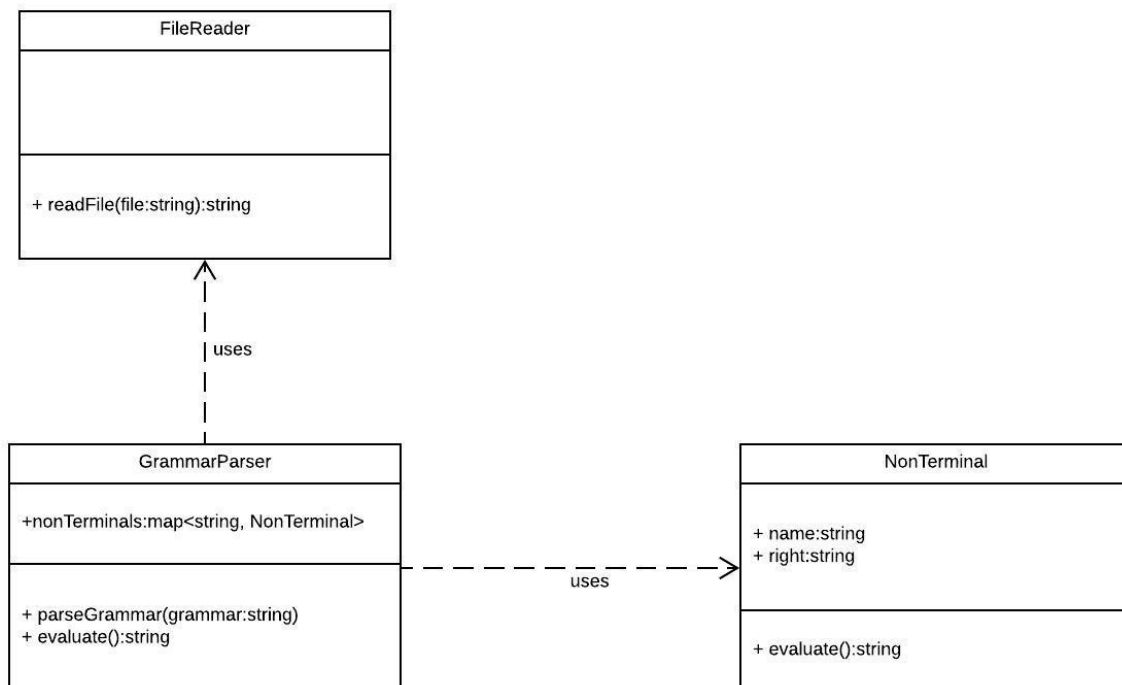


Figure 24: UML class diagram of grammar parser.

- **GrammarParser**

An object used to represent a parsed, user-defined, grammar. Holds the *parseGrammar* method which utilises the *readFile* method from *FileReader* to obtain a user-defined grammar; parses using the methods discussed in 3.3.2 and outputs loads the *nonTerminals* map. We store *NonTerminal*s in a map where its key is the *NonTerminal* name, i.e. (A) will be stored as <"A", *NonTerminal* object>. The *evaluate()* method attempts to find the nonterminal "S" from the map, and evaluating from its *NonTerminal* object; this is only possible if the *isSLP()* method returns true as the function would otherwise fall into an infinite loop. The *isCNF()* method checks that all *NonTerminal* objects in the *map* satisfy the discussed CNF conditions. We create *NonTerminal* objects through the use of helper functions processing the left and right side of each production separately.

- **NonTerminal**

Stores the *name* of the *NonTerminal* (the left side of the production), and the unprocessed *right* side of the production. On calling the *evaluate()* function, the right-hand side is processed, building the output string recursively when we find other *NonTerminal*s that exist in the *nonTerminals* map. The internal *isCNF()* function in any *NonTerminal* checks the discussed CNF conditions for the given production (see 3.3.4.1).

## 4.4 LZ77 to SLP Conversion

### 4.4.1 System Requirements

The following system requirements have been established for the LZ77 encoder:

Requirement Type	Requirement	MoSCoW
Functional	The program can take a list of LZ77-compressed tuples and output the resulting grammar.	M
Functional	The program can take a set of G-factors and output the resulting grammar.	M

### 4.4.2 Software Design

Figure 25 gives the complete UML class diagram of our LZ77/SLP converter designs. The designs include our *LZ77encoder* (see 4.1.2) and our *GrammarParser* (see 4.3.2) class with additional functionality.



The *Node* interface specifies the required behaviours of the following classes:

- The *Branch* class represents a production  $(A) \rightarrow (B)(C)$ . In this example,  $(A)$  represents the *name*,  $(B)$  represents *left*, and  $(C)$  represents *right*. The required functions work recursively

- *Terminal*

The *Terminal* class represents a production  $(A) \rightarrow a$ , as discussed previously, all terminal characters are simply stored in the leaves, hence storing the nonTerminal's name is unnecessary.

- *evaluate()*: returns the *character* itself.
- *getHeight()*: returns 0, as required.
- *getBalance()*: returns 0, as required.
- *get(i: int)*: returns the value of this character regardless if input, bounds checking should be completed before *get(...)* call.

#### - Converter

The *Converter* class utilises our *LZ77encoder* to perform the LZ77/SLP conversion algorithm discussed in 3.4.3. We utilise an *index* variable to name new *Nodes* and a *map* to store existing *Nodes*. The *Converter* can take a file location as input, or a set of factors to immediately begin concatenation. After conversion, we output a *Grammar*.

#### - Grammar

The *Grammar* class is used to represent the *startNode* of a grammar in Chomsky normal form (see 1.9.1). *evaluate()* will construct, recursively from the *startNode*, the output of the grammar. *get(...)* will recursively locate the  $i^{\text{th}}$  character of the tree rooted from the *startNode*. The *Grammar* class is also where we compute a grammar's *gFactors()*, using the description discussed in 3.4.1 we utilise a *set* to store previously seen prefixes, we then run a *pre-order traversal* of our grammar, storing all prefixes in the set as we traverse. Upon finding a prefix already existing in our set, we add this to our *gFactors* list. Using the *gFactors*, we can utilise our *Converter* to produce a *balanced grammar*, using the same algorithm as 3.4.3.

#### - GrammarParser

The same *GrammarParser* class used in 4.3.2. A *toCnfGrammar()* method has been added to convert a user inputted grammar (obtained via the *GrammarParser*, see 4.3.2) to the required CNF *Grammar* used by our *Converter*. We can only perform this operation provided *isCNF()*, and *isSLP()* are satisfied.

## 4.5 Compressed Equality Checking

### 4.5.1 System Requirements

The following system requirements have been established for the LZ77 encoder:

Requirement Type	Requirement	MoSCoW
Functional	The user can enter a desired sequence and retrieve a unique signature (randomized solution).	M
Functional	The user can concatenate two sequences and retrieve a unique signature (randomized solution).	M
Functional	The user can split a sequence at a given position, and retrieve two unique	M

	signatures (randomised solution).	
Functional	The user can enter the desired sequence and retrieve a unique signature (deterministic solution).	S
Functional	The user can concatenate two sequences and retrieve a unique signature (deterministic solution).	S
Functional	The user can split a sequence at a given position, and retrieve two unique signatures (deterministic solution).	S

## 4.5.2 Software Design

Figure 26 gives a detailed UML class diagram representing the data structure described in 3.5. We implement the randomised solution to the data structure, intentionally designed such that a simple replacement of the function *blockRandomised(...)* with a *blockDeterministic(...)* implementation would function as expected.

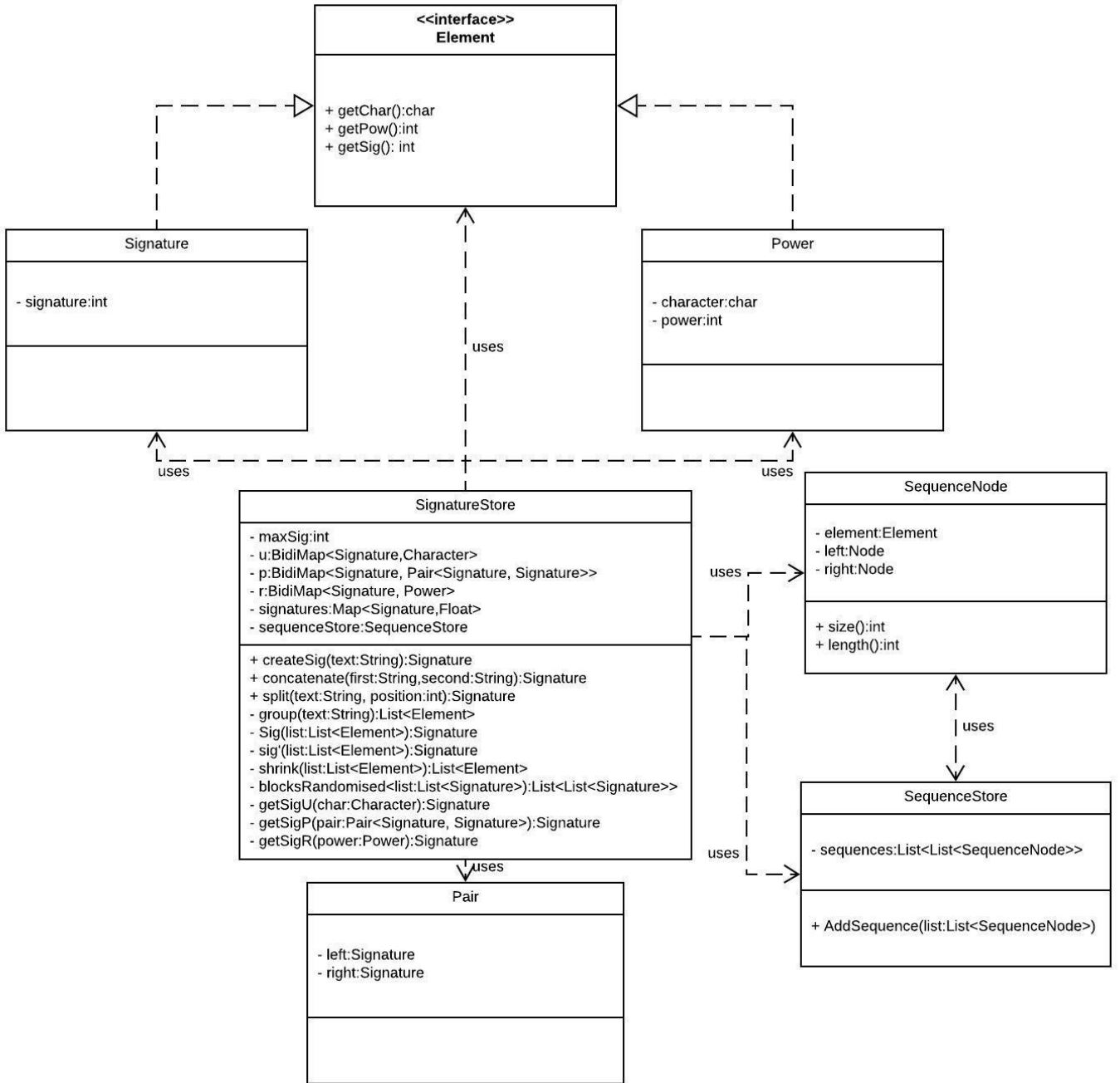


Figure 26: UML class diagram for storing sequences to check equality.

- **Element <<interface>>**

Due to the polymorphic nature of our  $\overline{sig}()$  and  $sig()$  functions (see 3.5.3), an *Element* interface was required to represent the two types of input the function would expect:

- **Signature**

A series of *Signatures* could be given as input to our  $\overline{sig}()$  function and a pairing function used to reduce to a single signature (as described in  $\overline{sig}()$  3.5.3.  $getSig()$  is implemented for this class but the *Element* methods simply return null, as they are never used.

- **Power**

The algorithm initially calls *sig()* on sequences in the form  $a_1^{l_1} \dots a_n^{l_n}$  with  $a_i \neq a_{i+1}$  for all  $i$ ,  $1 \leq i < n$  and for all  $l_i$ , a positive integer, where  $a_i^{l_i}$  denotes a subsequence of  $l_i$  repetitions of  $a_i$  (see 3.5.3). We therefore require a *Power* class to hold such a representation. *getPow()* and *getChar()* are implemented for this class but the other *Element* methods simply return null, as they are never used.

- **SignatureStore**

Where all sequences are stored and signatures computed. *group()* is used to convert user input into the required sequence of *Powers*. Holds three bidirectional hashmaps (*BidiMaps*, from the apache commons collections library, one can find all licensing in 2.5) which are used to hold the disjoint union of signatures described in 3.5.3.

*getSigU(...)*, *getSigR(...)* and *getSigP(...)* are used to access their respective *BidiMaps* and utilise *MaxSig*, to increment the *Signature* number and extend the *BidiMap* if we fail to find a match. We store the random priority values for *Signatures* in a *map*. We store sequences as a list of rooted binary trees (as described in 3.5.4) using our *SequenceStore*. We define the remaining methods in 3.5.3.

- **SequenceStore**

Stores all sequences using methods discussed in 3.5.4, trees are represented using *SequenceNodes*.

- **SequenceNode**

The class used to store sequences as described in 3.5.4.

- **Pair**

The class used to represent a pair of signatures stored in our *p BidiMap*.

## 4.6 Tree Printer

As our software is designed primarily for educational purposes, implementing visualisations of produced outputs was important. In order to achieve this, we designed a *TreePrinter* class to interface with existing tree structures.

### 4.6.1 System Requirements

The following system requirements have been established for the LZ77 encoder:

Requirement Type	Requirement	MoSCoW
Functional	The program can output a visual representation of any tree structure.	S

### 4.6.2 Software Design

Figure 27 gives a UML class diagram illustrating our *TreePrinter* designs.

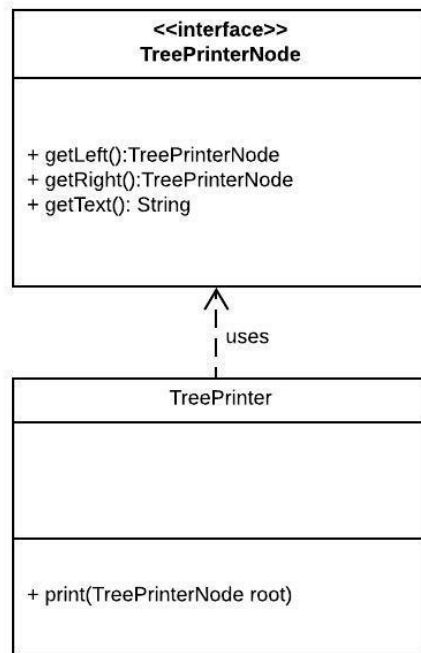


Figure 27: UML class diagram for visualising trees.

- **<<interface>> TreePrinterNode**  
All tree-like structures, i.e. our *SequenceNode* and *Node* classes will be required to implement the *TreePrinterNode* interface as well. *getText()* is used to output the necessary visualisations to the output tree.
- **TreePrinter**  
Has a single method, *print(...)*, which traverses a tree from its root, level by level, outputting the results of *getText()* in a tree-like fashion.

## 4.7 Main Program

Our main software will encompass all our work through a series of menu options, providing the functionality to all previously designed operations.

The following system requirements have been established for the LZ77 encoder:

### 4.7.1 System Requirements

Requirement Type	Requirement	MoSCoW
Functional	The program provides access to all LZ77 related operations (4.1,4.2).	M
Functional	The program provides access to all grammar related operations (3.3).	M
Functional	The program provides access to all signature store related operations (3.5).	M



### 4.7.2 Software Design

Figure 28 gives a basic UML class diagram illustrating how our main software will utilise our previously designed classes. Any additional class dependencies have been omitted from this diagram as they have been illustrated previously in their respective design sections (throughout 4).

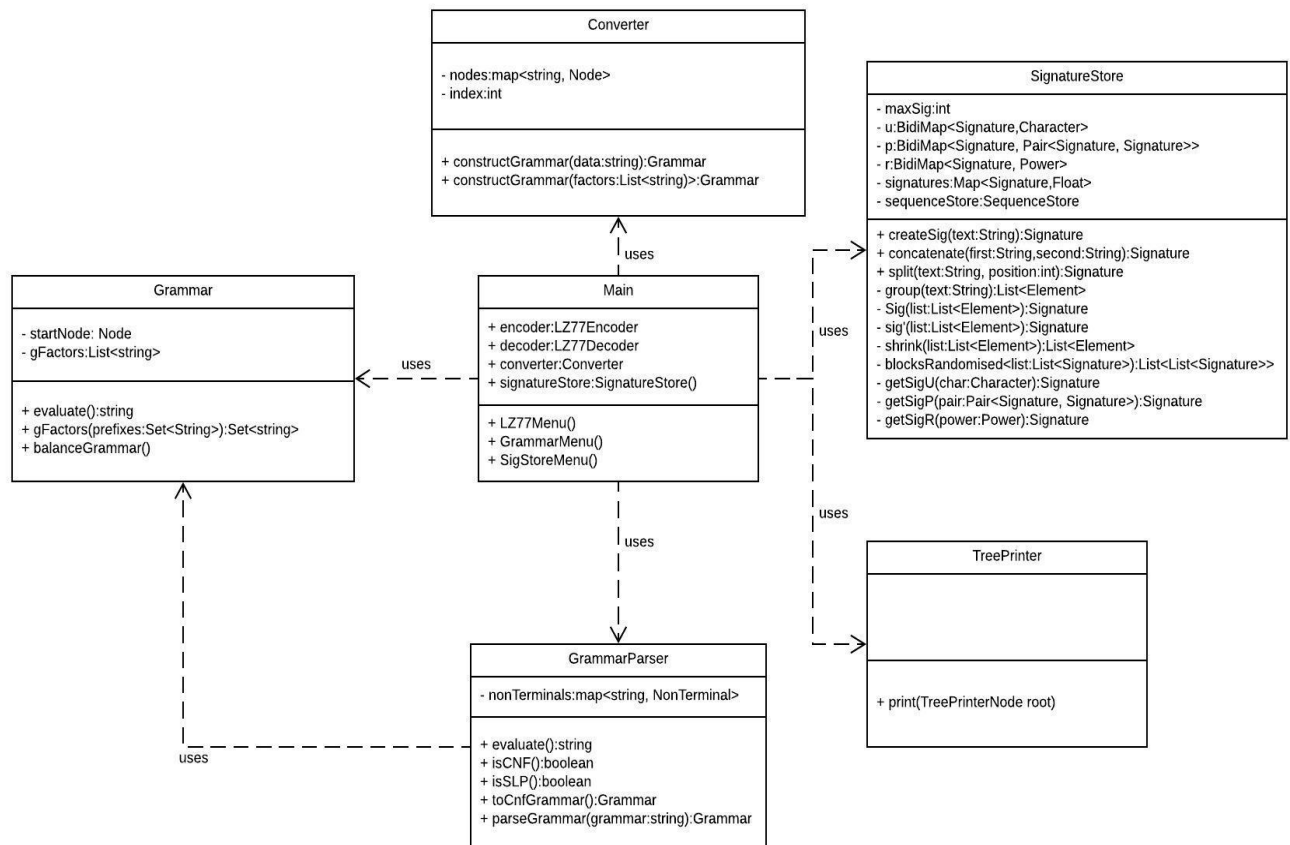


Figure 28: UML class diagram illustrating Main class encompassing all designs.

- Main

The main class where we initialise our program. Utilises previously designed software, allowing the end user to access their functionality via a series of menus:

- **LZ77Menu()**  
Contains functionality to compress (3.1), decompress (3.2) and convert LZ77 compressed files into balanced *Grammars* (3.4).
- **GrammarMenu()**  
Contains functionality to parse a file containing a Context Free Grammar in Chomsky Normal Form (3.3), balance a given grammar (3.4.1) and view its parse tree.
- **SigStoreMenu()**  
Contains functionality to compress and assign a signature to a given sequence (3.5.5.2), concatenate two sequences (3.5.5.3), split a sequence using the same

logic we use to make sequences (3.5.5.4), and compare equality of two sequences (3.5.5.1).

## 5 Implementation

Throughout the implementation section, we give a thorough walkthrough of the final piece of software and all its functionality through a series of examples. On all available menus, input “q” will bring the user back to the previous menu.

### 5.1 Main Menu

On launching the application, the program greets the user with a series of options for LZ77 operations, grammar operations and our Signature store data structure. Figure 29 presents the menu.

```
Please select an option

l - LZ77 operations
g - Grammar operations
s - Compressed signature data structure operations
d - View current directory
q - quit
```

*Figure 29: Program main menu.*

### 5.2 LZ77 Operations

On input ‘l’, the user is given a series of options related to LZ77 operations. Figure 30 displays the menu.

```
Select an option:
c - compress a file
d - decompress a file
s - convert LZ77 file to SLP
q - back
```

*Figure 30: LZ77 operations menu.*

#### 5.2.1 Compression

On input ‘c’, the user is required to enter a text file to be compressed. Once the software completes the compression phase, the user will see the outputted LZ77 compressed tuples. Also, an output file named *original\_file\_name.lz77* will be saved to the working directory; the output file is then available to be decompressed. Figure 31 shows a file, *exampleInput.txt*, containing the string “abaababaabaab” being compressed into a set of seven tuples, shown in Figure 32.

```
File to compress:
exampleInput.txt
Search window size:
32500
Lookahead window size:
250
```

Figure 31: Example LZ77 compression input.

```
Offset: 0, stringLen :0, nextChar: a
Offset: 0, stringLen :0, nextChar: b
Offset: 2, stringLen :1, nextChar: a
Offset: 3, stringLen :2, nextChar: b
Offset: 5, stringLen :4, nextChar: a
Offset: 0, stringLen :0, nextChar: b
File successfully compressed as exampleInput.txt.lz77
```

Figure 32: Result of compressing file containing the string "abaababaabaab".

The example compression outputs the file *exampleInput.txt.lz77* into the user's directory.

### 5.2.2 Decompression

On input 'd', the user is required to enter a *.lz77* compressed file and a desired output file name. Once the software completes the decompression phase, the user will have the desired output file in their working directory containing the original message contained by the original *.lz77* compressed file. Figure 33 shows our previous *.lz77* file, *exampleInput.txt.lz77*, being decompressed into its original message in *exampleOutputFile.txt*.

```
File to decompress:
exampleInput.txt.lz77
Output file name:
exampleOutputFile.txt
File decompressed successfully.
```

Figure 33: Example LZ77 decompression input.

### 5.2.3 LZ77 to SLP

On input 's', the user is again required to enter a *.lz77* file and a desired output file name. Once the software completes the conversion, the user will have a set of productions printed to the desired output file in the working directory which defines the balanced SLP representation of the LZ77 compressed file. Also, the program will display the parse tree of the balanced SLP.

Figure 34 shows our previous *exampleInput.txt.lz77* file being converted into a balanced SLP, with productions printed to *exampleOutputGrammar.txt*, shown by Figure 35.

```
File to convert:
exampleInput.txt.lz77
Output file name:
exampleOutputGrammar.txt
```

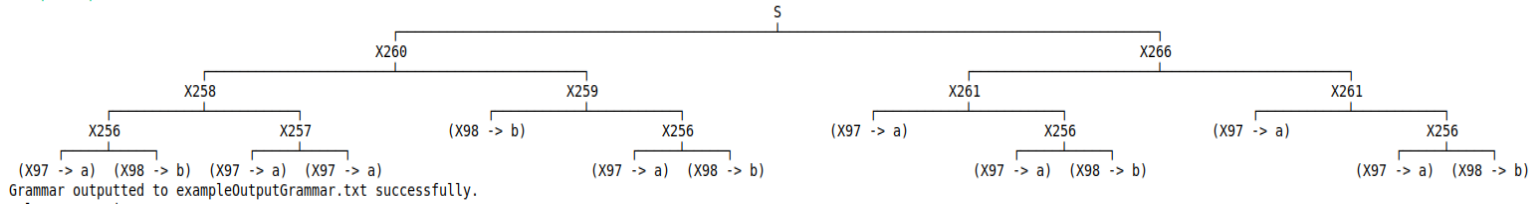


Figure 34: Converting our *exampleInput.txt.lz77* file into a balanced SLP.

```
1 (X261) -> (X97) (X256)
2 (X256) -> (X97) (X98)
3 (S) -> (X260) (X266)
4 (X98) -> b
5 (X259) -> (X98) (X256)
6 (X266) -> (X261) (X261)
7 (X260) -> (X258) (X259)
8 (X257) -> (X97) (X97)
9 (X258) -> (X256) (X257)
10 (X97) -> a
```

Figure 35: Balanced grammar outputted to *exampleOutputGrammar.txt*.

## 5.3 Grammar Operations

### 5.3.1 Parsing a Grammar

On input “g”, the user is immediately required to enter a text file containing a set of productions defining an SLP in Chomsky Normal Form. Once the program has completed the parsing phase (i.e. a correctly defined grammar in Chomsky Normal Form including a start symbol (S)) the user’s grammar will be successfully loaded into the program and can run the available operations. Figure 36 shows our input file, *exampleGrammar.txt*, being successfully parsed and loaded into the program.

```
g
Enter grammar file to parse:
exampleGrammar.txt
Evaluation: hello
Grammar successfully parsed.
Select an option:
t - view parse tree
b - balance grammar
g - compute gFactors
get - retrieve character at given position
q - back
```

Figure 36: Grammar parsing example input with the options menu.

### 5.3.2 Viewing Parse Tree

On input “t” after successfully loading a grammar file, the user can view the parse tree of the loaded grammar. Figure 37 displays the parse tree of the loaded grammar from *exampleGrammar.txt*.

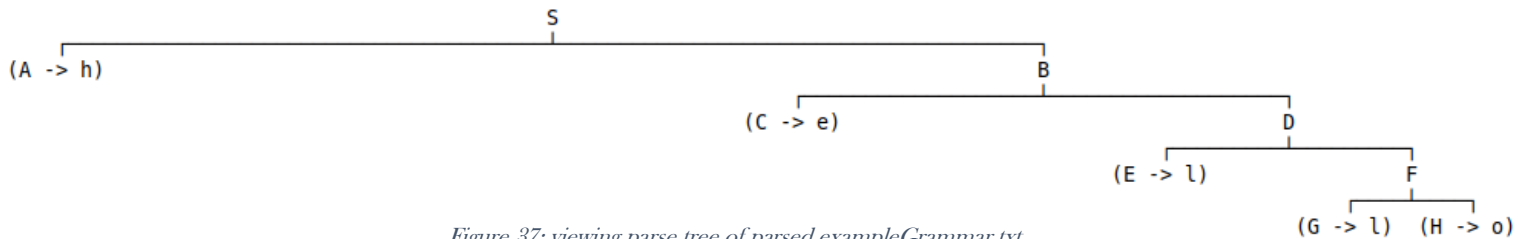


Figure 37: viewing parse tree of parsed *exampleGrammar.txt*.

### 5.3.3 Balance an Inputted Grammar

On input “b” after successfully loading a grammar file, the user is required to enter a desired output file for the newly balanced grammar. After a grammar is successfully balanced, the user will have a set of productions printed to the desired output file in the working directory which defines a newly balanced grammar which produces the same word as the original grammar; the newly balanced grammar is then loaded into the program and is given the same menu options. Figure 38 to Figure 40 shows our original grammar, from *exampleGrammar.txt*, being balanced successfully and outputted to *exampleBalancedGrammar.txt*.

```

Enter output file name:
exampleBalancedGrammar.txt
Grammar outputted to exampleBalancedGrammar.txt successfully.

```

Figure 38: Example balancing grammar operation input.

```

1 (X258) -> (X108) (X108)
2 (X101) -> e
3 (X108) -> l
4 (X260) -> (X258) (X111)
5 (X111) -> o
6 (S) -> (X256) (X260)
7 (X104) -> h
8 (X256) -> (X104) (X101)

```

Figure 39: *exampleBalancedGrammar.txt* output file.

t

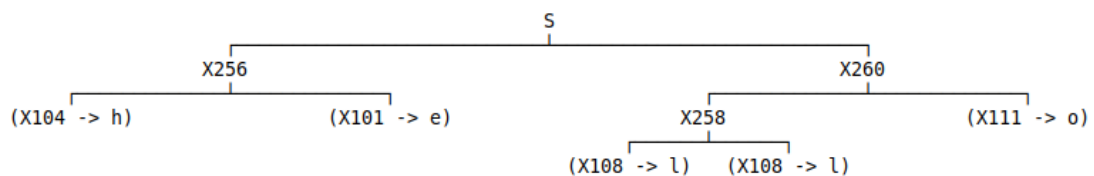


Figure 40: Viewing the newly balanced parse tree.

### 5.3.4 Compute G-factors

On input “g” after successfully loading a grammar file, the user can view a list of the G-factors of the inputted grammar. Figure 41 displays the G-factors for *exampleGrammar.txt*.

```
.  
g  
Grammar factors of hello:  
h  
e  
l  
l  
o
```

Figure 41: grammar factors for *exampleGrammar.txt*.

### 5.3.5 Querying a Grammar

On input “get” after successfully loading a grammar file, the user is required to enter the desired position. The program will then output the character located at the given position, provided the position inputted is at most the size of the grammar. Figure 42 gives an example.

```
get  
Enter position:  
3  
The character at position 3 is: l
```

Figure 42: Retrieving position 3 of *exampleGrammar.txt*.

## 5.4 Signature Data Structure Operations

On input “s”, the program gives the user a series of options asking whether they want to create a new signature store, load an existing signature store, save the current signature store or view the signature store’s available operations. Figure 43 gives the primary menu, and Figure 44 gives the menu on user input “o”.

```
s  
Select an option:  
l - Load existing data structure  
n - Create new data structure  
s - save current signature store  
o - run operations
```

Figure 43: Signature store menu.

```
o  
Select an option:  
c - create new signature  
j - join/concatenate two sequences  
s - split a sequence  
comp - compare two sequences by their signature
```

Figure 44: Signature store operations menu.

### 5.4.1 Creating a New Signature

On input “c” of the operations menu, the user is required to enter a sequence they wish to compress to a signature. Once the program has successfully compressed the sequence, it

displays a parse tree of signatures representing the original sequence, as well as the unique signature which defines that sequence; the sequence, as well as its signature, is stored into the current signature store. Figure 45 shows the program creating a signature for the sequence “testing”.

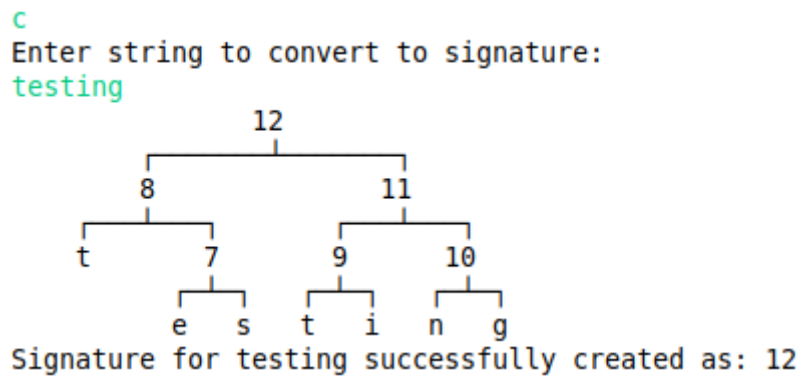


Figure 45: signature created for the sequence “testing”.

## 5.4.2 Concatenating Sequences

On input “j” of the operations menu, the user is required to enter two sequences they wish to concatenate. The program will attempt to create a signature for both sequences, returning an existing signature if one exists for that sequence. Once the program has successfully concatenated the sequences, the program displays a parse tree of each original sequence, followed by the parse tree of the newly concatenated sequence, as well as the unique signature which defines that sequence; the sequence, as well as its signature, is stored into the current signature store. Figure 46 shows the concatenation of sequences “test” and “ing”, resulting in the same signature as Figure 45.

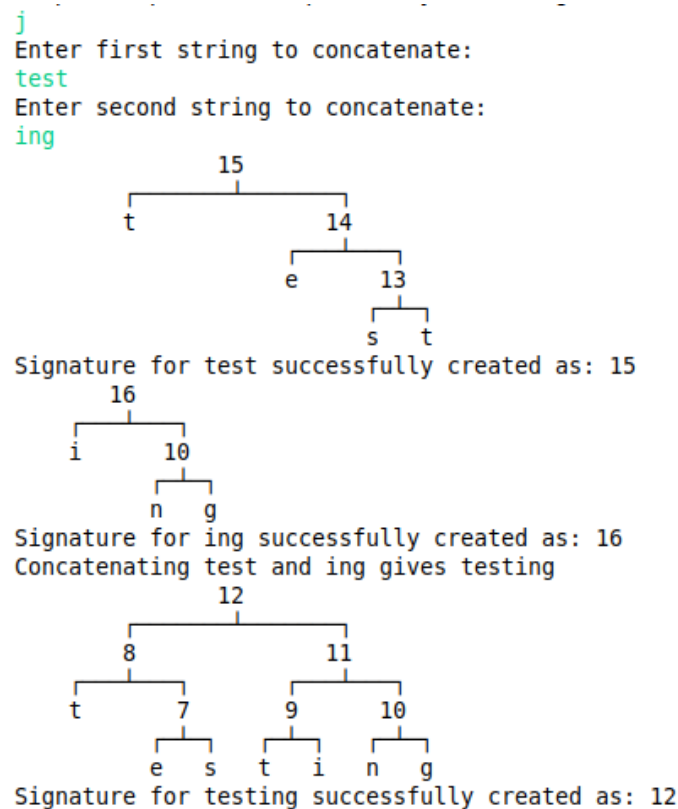


Figure 46: concatenating sequences “test” and “ing”, resulting in the same signature as previously.

### 5.4.3 Splitting a Sequence

On input “s” of the operations menu, the user is required to enter a sequence as well as an integral position,  $i$ , they wish to split the sequence at. The program will attempt to create a signature for both sequences (split at position  $i$ ), returning an existing signature if one exists for each sequence. Once the program has successfully split the sequence, the program displays a parse tree for the original sequence and each sequence (split at position  $i$ ). Both newly created sequences are then stored in the signature store. Figure 47 shows the splitting of sequence “testing” at position 4, resulting in the same signatures for sequences “test” and “ing” as created in Figure 46.

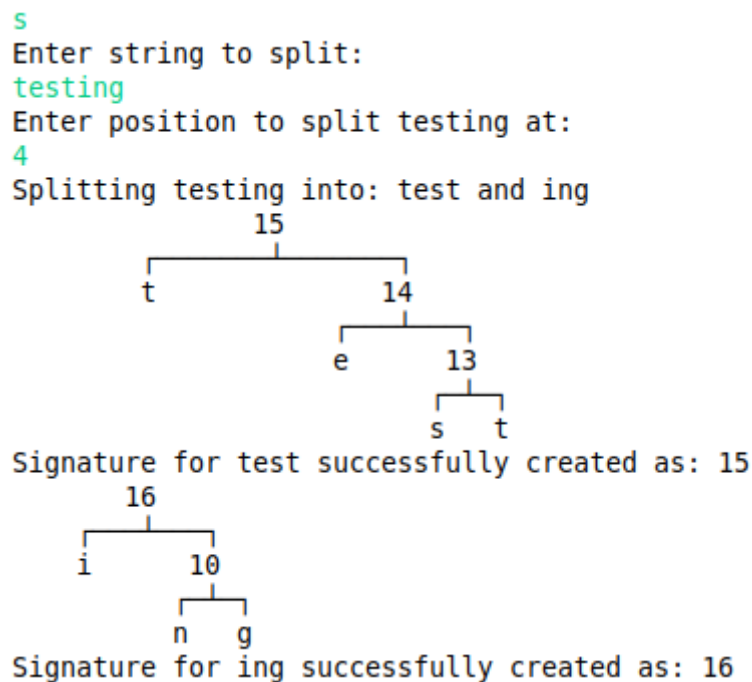


Figure 47: splitting the sequence “testing” at position 4, resulting in the same signatures as previously created.

### 5.4.4 Equality Checking

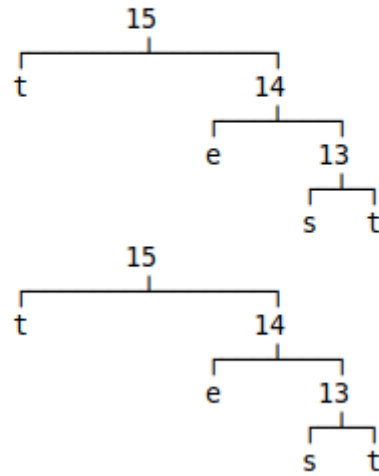
On input “comp” of the operations menu, the user is required to enter two sequences they wish to compare under the current signature store. The program will attempt to create a signature for both sequences, returning an existing signature if one exists for each sequence. Once the program has successfully obtained a signature value for each sequence, the program displays a parse tree for both sequences, and a simple equality comparison is conducted between both integral values to confirm whether they are equal. Figure 48 and Figure 49 show two comparisons of identical and differing sequences respectively.



```

comp
Enter first string to compare:
test
Enter second string to compare:
test

```



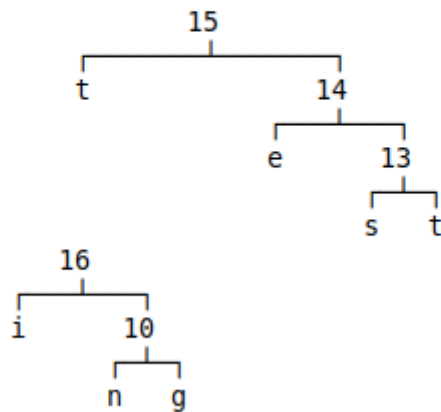
15 = 15. Therefore same sequence.

Figure 48: comparing identical sequences.

```

comp
Enter first string to compare:
test
Enter second string to compare:
ing

```



15 != 16. Therefore sequences differ.

Figure 49: comparing different sequences.

### 5.4.5 Saving a Store For Later Use.

On input “s” of the original signature store menu, the user is required to enter a desired output file name. The program then saves the current signature store in the desired output file. The user can reload the output file on input “l”; the loaded signature store will contain all previously created signatures. Figure 50-Figure 52 shows the saving and reloading of the created signature store, showing that previously created signatures are not lost.

```

File name:
exampleStore
Signature store successfully saved as exampleStore

```

Figure 50: saving the current signature store in exampleStore file.

```
Signature store file to load:
exampleStore
Signature loaded successfully.
```

Figure 51: reloading exampleStore from working directory.

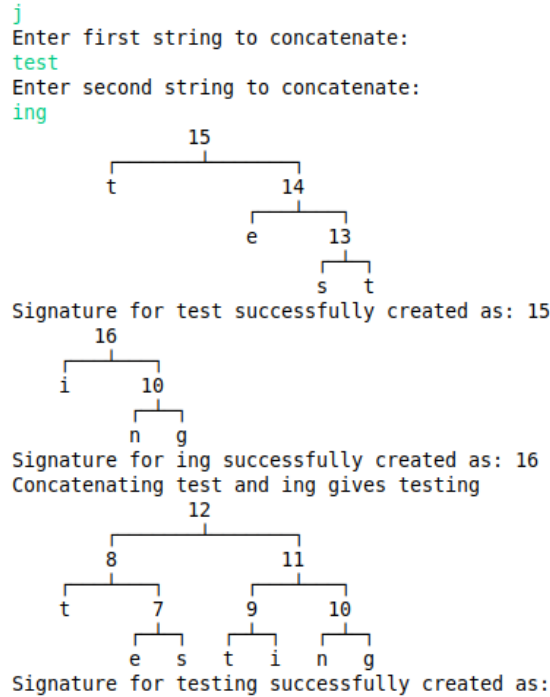


Figure 52: operations produce the same signatures as previously created.

## 5.5 Additional Observations Using Software

### 5.5.1 LZ77 Compression Software

The software has been tested using various book text files of varying lengths, confirming that the size after compression has decreased and that one can retrieve the original text can through decompression. In the following example, text sourced from [28] (Base) was used and multiplied different times to test and observe effects on compression using a search window size of 32000, and a lookahead window of 258:

<u>Text File</u>	<u>Raw data size</u> <u>(X, bits)</u>	<u>Execution time</u> <u>(milliseconds)</u>	<u>Compressed tuple</u> <u>size (Y, bits)</u>	<u>Compression</u> <u>factor (X/Y)</u>
Base	163384	55	116064	1.408
Base*5	816888	188	126176	6.474
Base*10	1797144	382	141344	12.715

Random strings of varying length were also tested to observe whether compression was possible.

<u>Text File</u>	<u>Raw data size</u> <u>(bits)</u>	<u>Execution time</u> <u>(milliseconds)</u>	<u>Compressed</u> <u>tuple size (bits)</u>	<u>Compression</u> <u>factor</u>
Random string small	992992	375	1294240	0.767
Random string large	8008000	3157	10501376	0.763

It was interesting observing the improvement in compression factor as we repeated the given input, as well as the poor performance displayed when merely attempting to compress

completely random strings; this observation fits what we would expect from our understanding of entropy in data compression (see 1.7).

## 5.5.2 Optimised Querying

Adding a timing feature to our querying function, we can observe the improved querying times once we successfully balance a grammar. Figure 53 shows our *unbalancedGrammar.txt* file, with a balance of -21, being queried for position 20. After rebalancing, Figure 54 shows the improved querying times.

```
unbalancedGrammar.txt
Evaluation: ACEGIKMOQSUY[]_acegikl
Grammar successfully parsed.
Balanced = -21

get
Enter position:
20
The character at position 20 is: i, in time: 139311 nanoseconds.
```

Figure 53: Locating position 20 in *unbalancedGrammar.txt*.

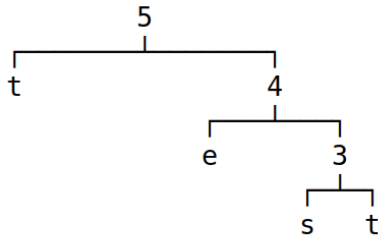
```
get
Enter position:
20
The character at position 20 is: i, in time: 10154 nanoseconds.
```

Figure 54: Locating position 20 in our newly balanced version of *unbalancedGrammar.txt*.

## 5.5.3 Conservation of Signature Structures

Figure 55 to Figure 57 shows the creation of sequences “test” and “Sequence” which are concatenated to produce the new sequence “testSequence”.

```
Enter string to convert to signature:
test
```

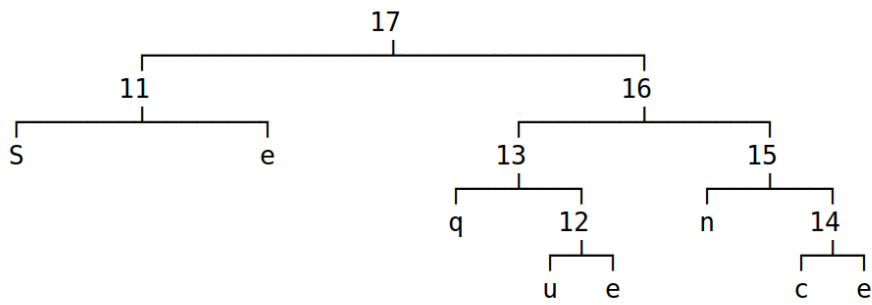


```
Signature for test successfully created as: 5
```

Figure 55: Creating a signature for sequence “test”.

Enter string to convert to signature:

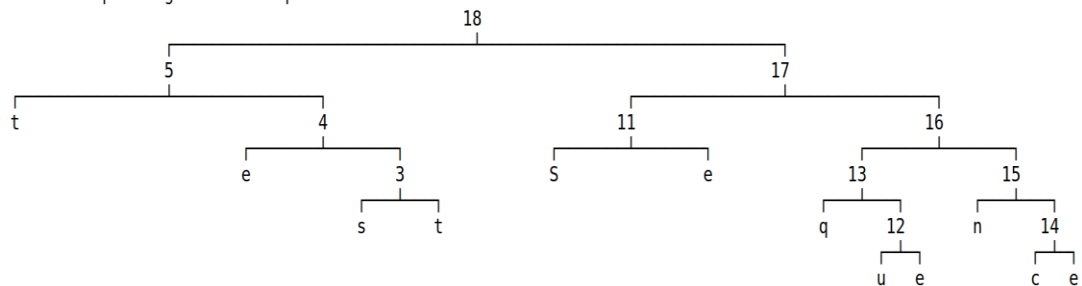
Sequence



Signature for Sequence successfully created as: 17

Figure 56: Creating a signature for sequence "Sequence".

Concatenating test and Sequence gives testSequence



Signature for testSequence successfully created as: 18

Figure 57: Concatenation of sequences "test" and "Sequence", producing signature for "testSequence".

We can observe in Figure 58 that despite constructing the sequence "testSequence" via a different pair of sequences i.e. "testSe" and "quence", we still compute an identical signature to Figure 57.

Enter first string to concatenate:

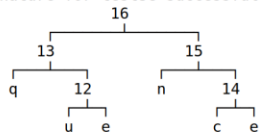
testSe

Enter second string to concatenate:

quence

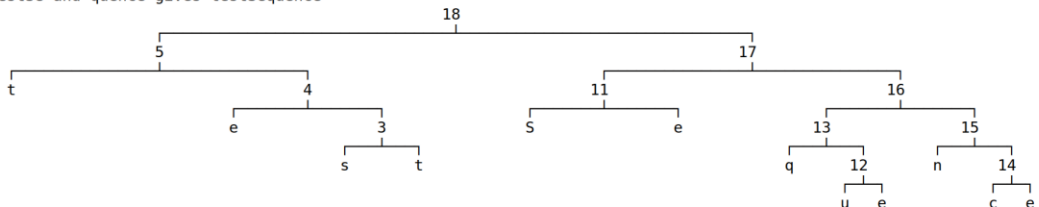


Signature for testSe successfully created as: 20



Signature for quence successfully created as: 16

Concatenating testSe and quence gives testSequence



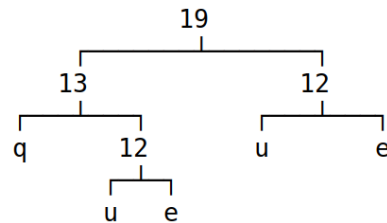
Signature for testSequence successfully created as: 18

Figure 58: Concatenating sequences "testSe" and "quence", producing the same signature for "testSequence" as previously constructed.

Additionally, in Figure 59, creating a signature for a sequence containing a subset of letters from “testSequence” i.e. “queue” utilises the same signatures for those subsets (“que” = 13, “ue”=12).

Enter string to convert to signature:

queue



Signature for queue successfully created as: 19

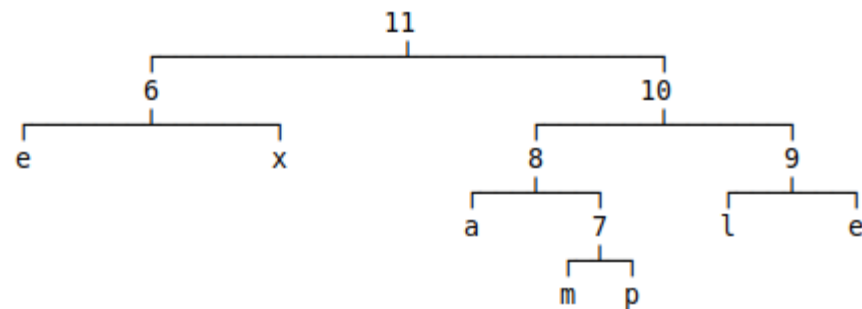
Figure 59: creating a signature for the sequence "queue".

### 5.5.4 Randomised Nature of Software

Due to the randomised nature of our solution, the priority values assigned to signatures will vary throughout instances of the signature store (see 3.5.2.1.1); this, therefore, means that block sizes will vary and create differing structures for a given sequence in different instances. Figure 60 shows an instance of our signature store creating a signature for the sequence “example”.

Enter string to convert to signature:

example



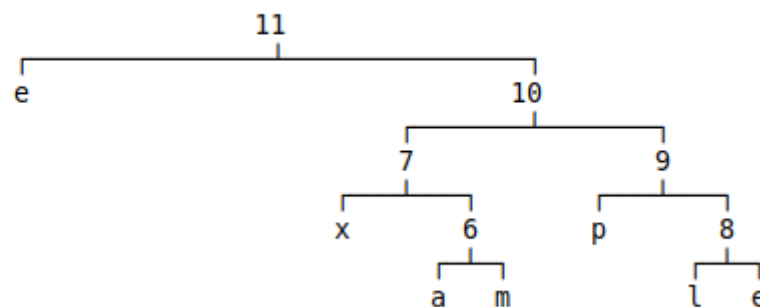
Signature for example successfully created as: 11

Figure 60: Creating a signature for the sequence "example".

On loading a new instance of our signature store and creating the same sequence, the program randomly assigns new priorities to our signatures (see 3.5.2.1.1) and hence a differing structure to our original store is given. Figure 61 shows the new structure.

Enter string to convert to signature:

example



Signature for example successfully created as: 11

Figure 61: Creating a signature for the same sequence, "example", but in a separate signature store with differing priority values.

Such scenarios would not occur in the deterministic version of the signature store; due to time constraints, this is a task for future work (see 8.2)

## 5.6 Documentation of Software

The code is publicly available on Github (see <https://github.com/u1604913/CS310-Project>) with an extensive README.md file to aid users in using the software. We have written detailed documentation to aid users in understanding the inner workings of the software. In addition, an extensive Javadoc has been written and is available within the Github repository; a preview with a link to the Javadoc is shown in Figure 62.

Interface Summary	
Interface	Description
Element	Element interface for SignatureStore data structure.
Node	Node interface used to represent grammars.
TreePrinterNode	TreePrinterNode object used to represent a node printable by our TreePrinter class.
Class Summary	
Class	Description
Branch	Represents a branch of a CnfGrammar i.e.
CFG	CFG object used to represent a parsed user grammar.
CnfGrammar	Object representing a grammar in CNF.
Converter	Converter methods used to convert LZ77 compressed tuples into a balanced SLP.
grammarTests	Tests for all grammar related operations.
LZ77	Functions required to compress a string into LZ77-compressed tuples.
LZ77Tests	Test for LZ77 related operations.
Main	Initialises CLI program.
Pair	Pair object used to represent a pair of signatures for SignatureStore data structure.
Power	Power object used to represent compressed run of characters in SignatureStore data structure.
Reference	Reference object used to represent an LZ77-Compressed tuple.
SequenceNode	SequenceNode object used to represent a compressed sequence in our SignatureStore data structure.
SequenceStore	SequenceStore used to store compressed sequences of our SignatureStore data structure.
Signature	Signature object used to represent a signature in our SignatureStore data structure.
SignatureStore	SignatureStore data structure used to convert sequences into unique signatures.
signatureStoreTests	Tests relating to all signature store data structure operations.
Terminal	Terminal object represents a terminal production in CnfGrammar i.e.
TreePrinter	

Figure 62: Javadoc for software, [link](#).

## 6 Testing

As discussed in 2.3.2, JUnit test cases were written before implementation and were readily available upon completion of each component. We will now discuss the tests conducted for each set of operations.

### 6.1 LZ77 Operations

The following table outlines the tests conducted for all LZ77 operations; JUnit tests are located in *LZ77Tests.java*.

Test name	Input	Expected Output
compressionTest1	The string “aabbaaab”	A list of tuples: (0,0,”a”),(1,1,”b”),(1,1,”a”),(5,2,”b”)
compressionTest2	The string “aaaaaaaaaa”	A list of tuples: (0,0,”a”),(1,1,”a”),(3,3,”a”),(7,3,”a”)
compressionTest3	The string “thisisatestthisisatest”	A list of tuples:

		(0,0,"t"),(0,0,"h"),(0,0,"i"),(0,0,"s"),(2,2,"a"),(7,1,"e"),(6,1,"t"),(11,10,"t")
decompressionTest1	A list of tuples: (0,0,"a"),(1,1,"b"),(1,1,"a"),(5,2,"b")	The string "aabbaaab"
decompressionTest2	A list of tuples: (0,0,"a"),(1,1,"a"),(3,3,"a"),(7,3,"a")	The string "aaaaaaaaaa"
decompressionTest3	A list of tuples: (0,0,"t"),(0,0,"h"),(0,0,"i"),(0,0,"s"),(2,2,"a"),(7,1,"e"),(6,1,"t"),(11,10,"t")	The string "thisisatestthisisatest"

The code was not pushed to production until all test cases were satisfied.

## 6.2 Grammar Operations

The following table outlines the tests conducted for all grammar operations; JUnit tests are located in *grammarTests.java*.

Test name	Input	Expected Output
parseGrammarTest1	The file "grammarTest.txt", shown in Figure 63.	A Grammar(S) consisting of the following nodes: Terminal("G", 'g'), Terminal("F", 'f'), Branch("D", F, G), Terminal("C", 'c'), Branch("B", D, E), Branch("S", B, C).
slpTest1	The file "grammarTest.txt", shown in Figure 63.	Correctly identifies the input to be an SLP.
slpTest2	The file "grammarTest2.txt", shown in Figure 64.	Correctly identifies the input not to be an SLP.
cnfTest1	The file "grammarTest.txt", shown in Figure 63.	Correctly identifies the input to be in CNF.
cnfTest2	The file "grammarTest2.txt", shown in Figure 64.	Correctly identifies the input not to be in CNF.
balanceTest1	A correctly balanced grammar.	Correctly identifies that the SLP is balanced.
balanceTest2	An unbalanced grammar.	Correctly identifies that the SLP is not balanced.
rebalanceTest	The unbalanced grammar from balanceTest2	Produces a balanced SLP.

The code was not pushed to production until all test cases were satisfied.

(S) -> (B)(C)  
 (B) -> (D)(E)  
 (C) -> c  
 (D) -> (F)(G)  
 (E) -> e  
 (F) -> f  
 (G) -> g

Figure 63: *grammarTest.txt*

(S) -> (A)(C)  
 (A) -> a  
 (B) -> b  
 (C) -> (S)

Figure 64: grammarTest2.txt

## 6.3 Signature Store Operations

The following table outlines the tests conducted for all Signature store operations; JUnit tests are located in *signatureStoreTests.java*.

Test name	Input	Expected Output
equalityTest1	Two sequences, “test” and “test”.	Identical signatures established for both sequences and said to be equal. i.e. createSig(“test”) = createSig(“test”).
equalityTest2	Two sequences, “test1” and “ing”.	Differing signatures established for both sequences and said not to be equal. i.e. createSig(“test1”) != createSig(“ing”).
concatEqualityTest1	Three sequences, “test”, “ing”, and “testing”.	Concatenating the first two sequences should give the same signature as producing a signature directly from the third. i.e. concat(“test”, “ing”) = createSig(“testing”).
concatEqualityTest2	Three sequences, “test”, “ing”, and “testing different”.	Concatenating the first two sequences should give a different signature to producing a signature directly from the third. i.e. concat(“test”, “ing”) != createSig(“testing different”).
concatEqualityTest3	Four sequences, “test”, “ing”, “aaa”, and “testingaaa”.	Concatenating the third sequence to the concatenation of the first two sequences should produce the same signature as producing a signature directly from the fourth. i.e. concat(concat(“test”, “ing”), “aaa”) = createSig(“testingaaa”).
concatEqualityTest5	Four sequences, “test”, “ing”, “testi”, “ng”.	Concatenating the first and second sequence should produce the same signature as concatenating the third and fourth sequence. i.e. concat(“test”, “ing”) = concat(“testi”, “ng”).

The code was not pushed to production until all test cases were satisfied.

## 7 Evaluation and Lessons Learned

We conducted the project in a systematic, professional manner. Through our agile research and development phases (see 2), we were able to build a sophisticated piece of software which is fully functional.

Throughout the project’s research phases, differing notation was experienced between papers and hence slowed down understanding of specific operations. If a research phase was taking longer than expected, the procedures, planned ahead of time in 2.2.3, were reviewed



and allowed progress to continue; this resulted in completing all the necessary research on time. An example of where our dependencies proved necessary was during the research phase of compressed equality checking (see 3.5). Sufficient understanding was taking far longer than expected, and hence project progress slowed; to combat this, we temporarily moved on to building the main program (see 4.7) that did not require the completion of all components. Project progress could therefore resume, and we established an understanding of our compressed equality checking algorithm at a later date.

Development phases were conducted using standard software development practices including version control and agile development. The use of sprints was certainly appreciated; keeping operations in self-contained cycles, as well as having pre-written tests, meant that a production ready piece of software was readily available at the end of each cycle.

The project taught the author the importance of organisation and timetabling. If we had not designed our research dependency diagram in 2.2.1, and hence our timetable in 2.6, the project would have likely taken far longer than expected; we would have required unexpected research gathering due to a lack planning. Regular meetings with the supervisor proved a useful way of consolidating weekly project progress as well as outlining future research and development cycles.

## 8 Conclusions and Future Work

### 8.1 Conclusions

#### 8.1.1 Project Overview

To summarise, the project reached a satisfying conclusion. Through our well-planned research and design phases, we were able to produce a fully functional piece of software implementing and integrating all the non-optional operations discussed in 5. Such operations included:

- **An LZ77 encoder**  
Encodes a user inputted text file into a list of LZ77 compressed tuples (see 5.2.1).
- **An LZ77 decoder**  
Decodes a user inputted .lz77 file (from our LZ77 encoder), outputting the original messages (see 5.2.2).
- **A grammar parser**  
Checks whether a given grammar is in Chomsky Normal Form and is a valid straight-line program (see 5.3.1).
- **LZ77 -> Straight-line program converter**  
Converts an LZ77-compressed file (using our LZ77 encoder) to a balanced straight-line program (see 5.2.3), the system can also be used to balanced grammars obtained from our grammar parser (see 5.3.3).
- **Optimised querying of compressed strings**  
As a byproduct of our LZ77->SLP converter, we can query straight-line programs in logarithmic time provided they are balanced.
- **Compressed Equality Checker (Signature store data structure, randomised solution)**

Compresses a sequence into a single signature (an integer) used to check equality of sequences in constant time. The concatenation and splitting of sequences have also been implemented successfully (see 5.4).

The final, fully integrated software is now publicly available on GitHub, with relevant documentation included. The repository is available at <https://github.com/u1604913/CS310-Project>

## 8.1.2 Evaluation Against System Requirements

We now look back at our original system requirements:

Requirement Type	Requirement	MoSCoW	Completed? (Yes/No)
<b><u>LZ77 encoder requirements</u></b>			
Functional	The user can load a desired text file into the program.	M	Yes
Functional	The user's inputted text file can be compressed to a list of compressed tuples	M	Yes
Functional	The compressed tuples can be written to the desired output file.	S	Yes
<b><u>LZ77 decoder requirements</u></b>			
Functional	The user can load a desired LZ77 compressed file into the program.	M	Yes
Functional	The user's inputted LZ77 compressed file can be decompressed into the original message.	M	Yes
Functional	A decompressed message can be written to the desired output file.	S	Yes
<b><u>Grammar parser requirements</u></b>			
Functional	The user loads a desired text file into the program.	M	Yes
Functional	The program can correctly parse a user loaded grammar.	M	Yes
Functional	The program can correctly identify a	M	Yes

	grammar in Chomsky Normal Form.		
Functional	The program can correctly identify a straight-line program.	M	Yes
Functional	The program can convert a grammar into Chomsky Normal Form.	S	No
<b><u>LZ77 to SLP requirements</u></b>			
Functional	The program can take a list of LZ77-compressed tuples and output the resulting grammar.	M	Yes
Functional	The program can take a set of G-factors and output the resulting grammar.	M	Yes
<b><u>Compressed equality checking requirements</u></b>			
Functional	The user can enter a desired sequence and retrieve a unique signature (randomized solution).	M	Yes
Functional	The user can concatenate two sequences and retrieve a unique signature (randomized solution).	M	Yes
Functional	The user can split a sequence at a given position and retrieve two unique signatures (randomised solution).	M	Yes
Functional	The user can enter the desired sequence and retrieve a unique signature (deterministic solution).	S	No
Functional	The user can concatenate two sequences and retrieve a unique signature (deterministic solution).	S	No
Functional	The user can split a sequence at a given position, and retrieve	S	No

	two unique signatures (deterministic solution).		
<b>Tree printer requirements</b>			
Functional	The program can output a visual representation of any tree structure.	S	Yes
<b>Main program requirements</b>			
Functional	The program provides access to all LZ77 related operations (4.1,4.2).	M	Yes
Functional	The program provides access to all grammar related operations (3.3).	M	Yes
Functional	The program provides access to all signature store related operations (3.5).	M	Yes

Figure 65: All system requirements and whether they have been completed.

All ‘must-have’ requirements were completed in their entirety; any non-crucial (‘should have’) requirements have not been completed due to time constraints and hence assigned to future work.

## 8.2 Future Work

Despite the project’s success, some explored operations were not implemented due to time constraints and were not fundamental to the final software piece (see 2.2.1), such operations include an ‘optional’ tag in their title and are listed below:

- **Grammar to Chomsky Normal Form (3.3.4.2)**  
Converting a user inputted grammar was not fundamental to our project (as we could simply require the user to input one, see 3.3.4.1) but would be a useful feature, we have already discussed its algorithmic details, and hence only implementing is required.
- **Compressed equality checking signature data structure (deterministic solution, 3.5.1.2)**  
Although we successfully implemented the Signature data structure using a randomised solution (see 5.4), a deterministic one does exist. The critical difference is in the algorithm’s block decomposition step (3.5.2); we have already discussed algorithmic details, and hence only implementing is required.

# 9 Author’s Assessment of Project

A wide variety of algorithms on compressed strings were not yet implemented before the completion of this project. The field of Computer Science can now benefit from having publicly available implementations of complex algorithms for experimental and educational

use. The project required a variety of skills in the fields of research, software development and project management. A wide array of challenging papers discussing algorithms unseen to the author were explored and required extensive understanding to implement; because of this, proficient research skills were required, displaying strong abilities in extracting the necessary information to build intricate, sophisticated designs.

## 10 References

- [1] M. Hosseini, "A Survey of Data Compression Algorithms and their Applications," in *Applications of Advanced Algorithms - Simon Fraser University*, Burnaby, 2012.
- [2] Keisuke Goto, Shirou Maruyama, Shunsuke Inenaga, Hideo Bannai, Hiroshi Sakamoto, Masayuki Takeda, "Restructuring Compressed Texts without Explicit Decompression," *CoRR*, vol. abs/1107.2729, 2011.
- [3] I. Pu, "Data compression," University of London, London, 2004.
- [4] J. Müller, "Data Compression LZ77," 25 11 2008. [Online]. Available: <http://jens.jm-s.de/comp/LZ77-JensMueller.pdf>. [Accessed 31 03 2019].
- [5] D. Ueltschi, "Shannon entropy," [Online]. Available: <http://www.ueltschi.org/teaching/chapShannon.pdf>. [Accessed 10 04 2019].
- [6] Travis Bauer, Thomas Brounstein , "Using Data Compression to Detect Inflection Points," Sandia National Labs.
- [7] L. Fortnow, "Kolmogorov Complexity," 01 2000. [Online]. Available: <https://people.cs.uchicago.edu/~fortnow/papers/kaikoura.pdf>. [Accessed 05 03 2019].
- [8] C. Zeeh, "The Lempel Ziv Algorithm," 16 01 2003. [Online]. Available: [https://fenix.tecnico.ulisboa.pt/downloadFile/3779571247713/LempelZiv\\_Zeeh.pdf](https://fenix.tecnico.ulisboa.pt/downloadFile/3779571247713/LempelZiv_Zeeh.pdf). [Accessed 2019 03 30].
- [9] Suman M. Choudhary, Anjali S. Patel, Sonal J. Parmar, "Study of LZ77 and LZ78 Data Compression," *International Journal of Engineering Science and Innovative Technology (IJESIT)*, vol. 4, no. 3, pp. 45-49, 2015.
- [10] P. H. Cording, "Notes on Compression Schemes," [Online]. Available: <http://www2.compute.dtu.dk/courses/02282/2015/compression/compression-notes.pdf>. [Accessed 03 27 2019].
- [11] M. Lohrey, "Algorithmics on SLP-compressed strings: A survey," *Groups Complexity Cryptology*, vol. 4, no. 2, pp. 241-299, 2012.
- [12] R. M. Sridharan, "CS259 Formal Languages," 27 03 2019. [Online]. Available: <https://warwick.ac.uk/fac/sci/dcs/teaching/modules/cs259/>. [Accessed 27 03 2019].

- [13 W. Rytter, "Application of Lempel-Ziv Factorization to the Approximation of Grammar-Based Compression," *Theoretical Computer Science*, vol. 302, no. 1-3, pp. 211-22, 2003.
- [14 Kurt Mehlhorn, R. Sundar, Christian Uhrig, "Maintaining Dynamic Sequences under Equality Tests in Polylogarithmic Time," *Algorithmica*, vol. 17, no. 2, pp. 183-198, 1997.
- [15 Yury Lifshits, Markus Lohrey, "Querying and Embedding Compressed Texts," in *Mathematical Foundations of Computer Science*, Lesn, Springer, 2006, pp. 681-692.
- [16 Lucidchart, "UML Class Diagram Tutorial," [Online]. Available:  
| <https://www.lucidchart.com/pages/uml-class-diagram>. [Accessed 05 04 2019].
- [17 D. Haughey, "MOSCOW Method," [Online]. Available:  
| <https://www.projectsmart.co.uk/moscow-method.php>. [Accessed 24 04 2019].
- [18 Stackify, "What is Agile Methodology? How It Works, Best Practices, Tools," 17 09 2017.  
| [Online]. Available: <https://stackify.com/agile-methodology/>. [Accessed 03 04 2019].
- [19 M. James, "An Empirical Framework For Learning (Not a Methodology)," [Online].  
| Available: <http://scrummethodology.com/>. [Accessed 03 04 2019].
- [20 tutorialsPoint, "JUnit - Test Framework," 24 04 2019. [Online]. Available:  
| [https://www.tutorialspoint.com/junit/junit\\_test\\_framework.htm](https://www.tutorialspoint.com/junit/junit_test_framework.htm).
- [21 Github, "About Github," Github, [Online]. Available: <https://github.com/about>. [Accessed  
| 24 04 2019].
- [22 Trello, "About Trello," Atlassian, [Online]. Available: <https://trello.com/en-GB>. [Accessed  
| 24 04 2019].
- [23 Eclipse, "What is Eclipse?," [Online]. Available:  
| [https://help.eclipse.org/mars/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Fguide%2Fint\\_eclipse.htm](https://help.eclipse.org/mars/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Fguide%2Fint_eclipse.htm). [Accessed 04 24 2019].
- [24 Apache, "<http://www.apache.org/licenses/LICENSE-2.0>," January 2004. [Online].  
| [Accessed 09 0 2018].
- [25 Tutorialspoint, "Chomsky Normal Form," [Online]. Available:  
| [https://www.tutorialspoint.com/automata\\_theory/chomsky\\_normal\\_form.htm](https://www.tutorialspoint.com/automata_theory/chomsky_normal_form.htm). [Accessed  
2019 03 03].
- [26 D. Knuth, "Balanced Trees," in *The Art of Computing*, 2nd ed., vol. 3, Reading, Addison-  
| Wesley, 1998, p. 474.
- [27 A. V. Goldberg, S. A. Plotkin, G. E., "Shannon Parallel symmetry-breaking in sparse  
| graphs," in *Discrete Math.*, SIAM J, 1988, pp. 434-446.
- [28 J. Prentice, "100 west by 53 north," [Online]. Available:  
| <http://textfiles.com/stories/100west.txt>. [Accessed 27 11 2018].

- [30] A. Commons, "org.apache.commons.compress.compressors.lz77support," [Online].  
Available: <https://commons.apache.org/proper/commons-compress/apidocs/org/apache/commons/compress/compressors/lz77support/LZ77Compressor.html>. [Accessed 12 10 2018].