Shift Gray Codes

by

Aaron Michael Williams

B.Math., Combinatorics & Optimization, University of Waterloo, 2001
B.Math., Computer Science, University of Waterloo, 2001
M.Math., Combinatorics & Optimization, University of Waterloo, 2004

A Dissertation Submitted in Partial Fulfillment of the Requirements for the Degree of
Doctor Of Philosophy
in the Department of Computer Science

Shift Gray Codes

by

Aaron Michael Williams

B.Math., Combinatorics & Optimization, University of Waterloo, 2001
B.Math., Computer Science, University of Waterloo, 2001
M.Math., Combinatorics & Optimization, University of Waterloo, 2004

Supervisory Committee

Dr. Frank Ruskey
Co-Supervisor
(Department of Computer Science)

Dr. Wendy Myrvold
Co-Supervisor
(Department of Computer Science)

Dr. Peter Dukes
Outside Member
(Department of Mathematics & Statistics)

Dr. Ulrike Stege
Departmental Member
(Department of Computer Science)

Supervisory Committee

Dr. Frank Ruskey
Co-Supervisor
(Department of Computer Science)

Dr. Wendy Myrvold
Co-Supervisor
(Department of Computer Science)

Dr. Peter Dukes
Outside Member
(Department of Mathematics & Statistics)

Dr. Ulrike Stege
Departmental Member
(Department of Computer Science)

Abstract

Combinatorial objects can be represented by strings, such as 21534 for the permutation (1 2) (3 5 4), or 110100 for the binary tree corresponding to the balanced parentheses (()()). Given a string $\boldsymbol{s} = s_1 s_2 \cdots s_n$, the *right-shift operation* $\overrightarrow{\mathsf{shift}}(\boldsymbol{s}, i, j)$ replaces the substring $s_i s_{i+1} \cdots s_j$ by $s_{i+1} \cdots s_j s_i$. In other words, s_i is right-shifted into position j by applying the permutation $(j \;\; j-1 \;\; \cdots \;\; i)$ to the indices of $\boldsymbol{s}$. Right-shifts include *prefix-shifts* ($i = 1$) and *adjacent-transpositions* ($j = i + 1$). A *fixed-content language* is a set of strings that contain the same multiset of symbols. Given a fixed-content language, a *shift Gray code* is a list of its strings where consecutive strings differ by a shift. This thesis asks if shift Gray codes exist for a variety of combinatorial objects. This abstract question leads to a number of practical answers.

The first prefix-shift Gray code for multiset permutations is discovered, and it provides the first algorithm for generating multiset permutations in O(1)-time while using O(1) additional variables. Applications of these results include more efficient exhaustive solutions to stacker-crane problems, which are natural NP-complete traveling salesman variants. This thesis also produces the fastest algorithm for generating balanced parenthesis in an array, and the first minimal-change order for fixed-content necklaces and Lyndon words.

These results are consequences of the following theorem: Every bubble language has a right-shift Gray code. Bubble languages are fixed-content languages that are closed under certain adjacent-transpositions. These languages generalize classic combinatorial objects — k-ary trees, ordered trees with fixed branching sequences, unit interval graphs, restricted Schröder and Motzkin paths, linear-extensions of B-posets — and their unions, intersections, and quotients. Each Gray code is circular and is obtained from a new variation of lexicographic order known as cool-lex order.

Gray codes using only $\overrightarrow{\mathsf{shift}}(\boldsymbol{s}, 1, n)$ and $\overrightarrow{\mathsf{shift}}(\boldsymbol{s}, 1, n-1)$ are also found for multiset

permutations. A universal cycle that omits the last (redundant) symbol from each permutation is obtained by recording the first symbol of each permutation in this Gray code. As a special case, these shorthand universal cycles provide a new fixed-density analogue to de Bruijn cycles, and the first universal cycle for the "middle levels" (binary strings of length $2k+1$ with sum k or $k+1$).

Contents

List of Tables

List of Figures

List of Algorithms

Acknowledgements

The topic of combinatorial generation was introduced to me in a course taught by Frank Ruskey. The course followed new portions of Don Knuth's *The Art of Computer Programming*, and although it was never mentioned in class, the students were impressed to find citations to Professor Ruskey's work in each of Knuth's six sections on tuples, permutations, combinations, partitions, trees, and history. Quite literally he has written the book on *Combinatorial Generation* [62], and his knowledge and contributions to the area were invaluable during the preparation of this thesis. I am also indebted to Professor Ruskey for his insight in posing the following open research problem to the class

Can the binary strings with s zeros and t ones be ordered by prefix-shifts?

The answer to this question provided the starting point for the results contained in this thesis. It also provided me with a topic for the *Generalizations of de Bruijn Cycles and Gray Codes* conference that was influential to my research. The trip to Banff gave me an opportunity to meet other luminaries in the field, and was just one of many research trips that I was able to take from the University of Victoria, I'd like to thank both Frank Ruskey and Wendy Myrvold for their generous financial support and for allowing me to expand my horizons on these ventures.

I'd also like to acknowledge both of my supervisors for their understanding and patience. Although the results for this thesis were completed and sketched by the end of 2007, it took a significant amount of time to refine their formal proofs. Several important concepts — including the frozen prefix of a string ($\mathsf{frozen}(\boldsymbol{s})$), the non-structural definition of bubble languages, and the utility of having both left-shift and right-shift cool-lex operations ($\overleftarrow{\mathsf{cool}}(\boldsymbol{s})$ and $\overrightarrow{\mathsf{cool}}(\boldsymbol{s})$) — were only identified during the past year. Their faith that "almost done" would eventually become "really almost done" should be credited whenever the reader stumbles across clarity within these pages. Of course, both professors also recognized when enough was enough, and additional results and future research are summarized in the final chapter.

Finally, I'd like to acknowledge the members of my thesis committee. Peter Dukes found several errors that had gone unnoticed, and Ulrike Stege convinced me to write the final chapter that I had been avoiding. Ron Graham's trip to Victoria made the day of my defense a special occasion, and I was fortunate to share the event with a number of my closest friends.

This thesis is dedicated to Margaret J. Williams, who found the energy to work full-time as a teacher, earn a university degree in the evenings, and raise me as a single parent. She also provided me with formative tools — including a `Vic 20` *computer and the* `Rami` *binary toy by* `Quercetti` *— even though her own interests were not in these areas. Her support and sacrifices continued throughout my entire education, despite my preoccupation with endless courses and research. When I suffered a painful and immobilizing back injury in 2007, she flew from Ontario to British Columbia and stayed with me for two months while I recovered. Her presence directly contributed to the contents of this thesis, as it allowed me to focus my attention away from the agonizing experience at hand, and onto the shorthand universal cycle results found in the latter half of Chapter 4. From the snoopy calculator to shift Gray codes, thank you!*

dn

Chapter 1

Combinatorial Generation

"It is really quite simple. We have been compiling a list which shall contain all the possible names of God."
"I beg your pardon?"
"We have reason to believe," continued the Lama imperturbably, "that all such names can be written with not more than nine letters in an alphabet we have devised."
"And you have been doing this for three centuries?"
"Yes. We expected it would take us about 15,000 years to complete the task."
"Oh." Dr. Wagner looked a little dazed. "Now I see why you wanted to hire one of our machines."

- The Lama and Dr. Wagner in *The Nine Billion Names of God*

Within Arthur C. Clarke's classic 1953 short story [10], the Lama is confronted with a monumental task. In his belief system, the possible names of God can be written using at most nine letters from a special alphabet. His goal is to write out each of the nine billion possible names, thereby bringing a satisfactory end to existence. With this sole purpose, and three centuries of work, the inhabitants of his lamasery in Tibet have written only 2% of the possibilities. To fast-track the process, the Lama has traveled to New York to enlist the services of Dr. Wagner, and the *Mark V* automatic sequence computer. By turning over his faith to this machine — capable of thousands of calculations per second — the Lama estimates that his task can be completed within one hundred days.

Despite its fictional nature, Arthur C. Clarke's short story accurately predicted the research area that provides the topic of this thesis. *Combinatorial generation* uses discrete mathematics and theoretical computer science to achieve its goal of "efficiently creating all possibilities". To illustrate the type of result contained in this thesis, consider the following operation.

Reordering a binary string using right-shifts
Shift the first bit to the right until it passes over a 01 or the last bit.

(i)

For example, operation (i) transforms 100100100 into 001100100 since the first bit is shifted to the right until its passes over the first 01. This transformation is illustrated by $\overrightarrow{100100}100 = 001100100$. More generally, *right-shifts* are illustrated using right-arrows (i.e., $\overrightarrow{abcde}f = acdebf$). As another example, operation (i) produces $\overrightarrow{110000} = 100001$ since the first bit of 110000 is shifted past the last bit without every passing over a 01. Despite its simplicity, operation (i) has the property that it eventually reorders the bits of any binary string in all possible ways. For example, the binary strings with three 0s and three 1s are shown below, and each string is obtained by applying the operation to the previous string

$$\overrightarrow{111000}, \overrightarrow{110001}, \overrightarrow{100011}, \overrightarrow{000111}, \overrightarrow{001011}, \overrightarrow{010011}, \overrightarrow{100101}, \overrightarrow{001101}, \overrightarrow{010101}, \overrightarrow{101001},$$
$$\overrightarrow{011001}, \overrightarrow{110010}, \overrightarrow{100110}, \overrightarrow{001110}, \overrightarrow{010110}, \overrightarrow{101010}, \overrightarrow{011010}, \overrightarrow{110100}, \overrightarrow{101100}, \overrightarrow{011100}. \quad (1.1)$$

The order of strings in (1.1) is known as a *right-shift Gray code* since successive strings differ by a right-shift. Furthermore, the Gray code is *circular* since one additional application of operation (i) transforms the last string into the first ($\overrightarrow{011100} = 111000$). The strings in (1.1) are known as the $(3, 3)$-combinations, and in general the binary strings with s 0s and t 1s are known as the (s, t)-*combinations*. The fact that (s, t)-combinations are generated by operation (i) is the most basic corollary of a general theory on shift Gray codes that is developed in this thesis.

While operation (i) has mathematical intrigue, results in combinatorial generation must also be seen through the screen of a computer scientist. To continue our illustration, it is helpful to state the operation that is inverse to operation (i).

Reordering a binary string using left-shifts
Shift the bit following the first 01 (or the last bit) into the first position.

(ii)

In general, *left-shifts* are illustrated using left-arrows (i.e., $\overleftarrow{abcde}f = aebcdf$). Operations (i) and (ii) are also known as *prefix-shifts*. This is because operation (i) right-shifts the first symbol in a string, whereas (ii) left-shifts a symbol into the first position of a string. Notice that the prefix-shift performed by operation (ii) has the effect of simply "undoing" the prefix-shift performed by operation (i). One advantage of operation (ii) is that no scanning is required to perform successive applications. This is because the operation depends only on the position of the first 01, and because the operation changes this position in a predictable pattern. To see this pattern

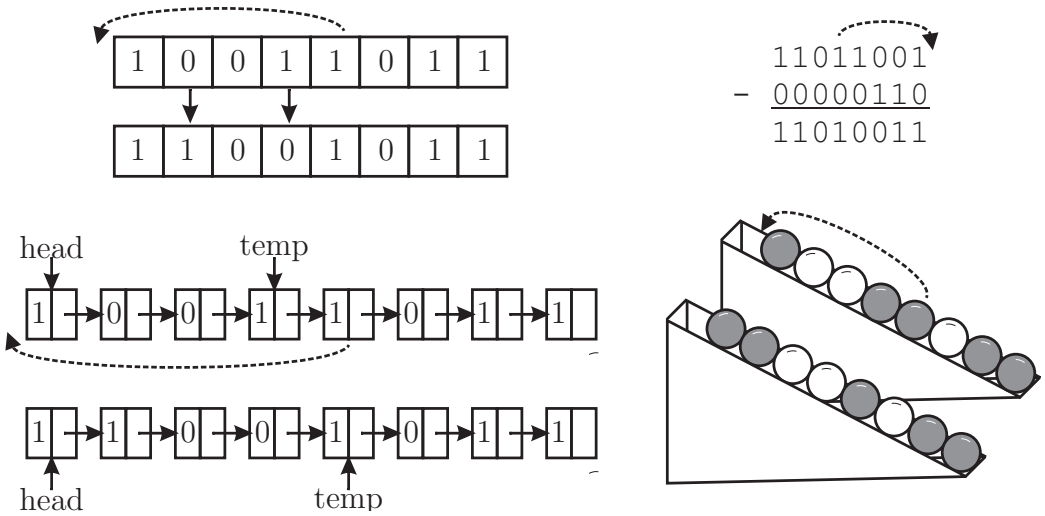


Figure 1.1: Clockwise from bottom-left: Operation (ii) can be implemented quickly in a linked-list (by pointer manipulations), or in an array (by complementing at most four values), or in a computer word (by shifts, masks, and arithmetic), or by hand. Each illustration shows $\overleftarrow{10011011}$ = 11001011. The computer word illustration is on the reverse of the string to match one of the steps in Algorithms 4 and 5.

more clearly, consider the order of $(3,3)$-combinations that is generated by operation (ii). For each string, the position of the first 01 is underlined. Notice that in each subsequent string, the underlined substring either moves one position to the right, or is reset to the beginning of the string if this string starts with 01.

$$\overleftarrow{0\underline{11}100}, \overleftarrow{1\underline{01}100}, \overleftarrow{11\underline{01}00}, \overleftarrow{0\underline{11}010}, \overleftarrow{1\underline{01}010}, \overleftarrow{0\underline{10}110}, \overleftarrow{0\underline{01}110}, \overleftarrow{1\underline{00}110}, \overleftarrow{11\underline{00}10}, \overleftarrow{0\underline{11}001},$$
$$\overleftarrow{1\underline{01}001}, \overleftarrow{0\underline{10}101}, \overleftarrow{0\underline{01}101}, \overleftarrow{1\underline{00}101}, \overleftarrow{0\underline{10}011}, \overleftarrow{0\underline{01}011}, \overleftarrow{0\underline{00}111}, \overleftarrow{1\underline{00}011}, \overleftarrow{11\underline{00}01}.\overleftarrow{111000}. \quad (1.2)$$

For this reason, successive applications of operation (ii) can be applied extremely quickly. Another advantage of (ii) is its versatility. In particular, operation (ii) can be efficiently implemented using a wide variety of standard data types including linked lists, arrays, and computer words. Figure 1.1 illustrates this fact, and Chapter 4 includes implementations of the associated algorithms using these data types. Figure 1.1 also shows how operation (ii) can be implemented by hand. The benefit of this manual interpretation is discussed later in this chapter in the context of *The Nine Billion Names of God*.

Despite its fundamental nature, operation (ii) and its algorithmic consequences were only discovered recently (see Ruskey-Williams [70, 73]). The associated *cool-lex order* of (s,t)-combinations turns out to be a subtle variation of lexicographic order,

as seen by Figure 1.2. Furthermore, cool-lex order hides additional properties that may hold interest for discrete mathematicians and theoretical computer scientists. To conclude our initial introduction to the results contained in this thesis, consider the following string of length $\binom{6}{3} = 20$

$$11001010110100111000. \tag{1.3}$$

This string is known as a *shorthand universal cycle* for $(3,3)$-combinations. This is because it contains every $(3,3)$-combination exactly once as a circular substring, with the proviso that the last (redundant) symbol of is omitted. For example, the first five symbols of (1.3) are 11001, and this substring is *shorthand* for the $(3,3)$-combination 110010. Alternatively, (1.3) encodes the following ordering of $(3,3)$-combinations, where each string differs from the previous by shifting the first symbol into the last or second-last position

$$\begin{aligned}&\overrightarrow{110010}, \overrightarrow{100101}, \overrightarrow{001011}, \overrightarrow{010101}, \overrightarrow{101010}, \overrightarrow{010110}, \overrightarrow{101100}, \overrightarrow{011010}, \overrightarrow{110100}, \overrightarrow{101001},\\ &\overrightarrow{010011}, \overrightarrow{100110}, \overrightarrow{001110}, \overrightarrow{011100}, \overrightarrow{111000}, \overrightarrow{110001}, \overrightarrow{100011}, \overrightarrow{000111}, \overrightarrow{001101}, \overrightarrow{011001}.\end{aligned} \tag{1.4}$$

To interpret this pattern correctly, notice that (1.3) contains the first bit of each successive strings in (1.4). Similarly, the second, third, fourth, and fifth bits in (1.4) are simply rotations of (1.3). This fascinating pattern can be derived quite prestidigitally from the cool-lex order found in (1.1). The string in (1.3) is also known as a universal cycle for the middle levels. For any given value of k, the *middle levels* are the binary strings of length $2k+1$ containing k or $k+1$ copies of 1. Prior to this thesis, there was no explicit construction known for these universal cycles.

For thorough coverage of combinatorial generation, the reader is directed to the upcoming volume of *The Art of Computer Programming* by Don Knuth. Named by *American Scientist* as one of the best twelve physical-science monographs of the 20th century, along with other notables such as Albert Einstein's *The Meaning of Relativity* and Richard Feynman's *QED*, *The Art of Computer Programming* is considered by many as the preeminent textbook in computer science. Several fascicles of this new volume have been printed [45, 46, 47] and include over 400 pages on the subject of combinatorial generation. In particular, the *Generating all Combinations* fascicle includes cool-lex order for $(4,4)$-combinations under the name "suffix-rotated" as well as Knuth's own MMIX computer word implementation that is "incredibly efficient". Another resource that will greatly expand the research area upon its release is the

aptly named *Combinatorial Generation* textbook by Ruskey [62].

The remainder of this chapter is organized into three sections. Section 1.1 describes four historical foundations of modern combinatorial generation. Section 1.2 then discusses contemporary results in combinatorial generation. Section 1.3 completes the chapter by outlining the new results contained in this thesis. To help motivate the reader, each section relates its material to *The Nine Billion Names of God*.

1.1 Historical Foundations

Judging from the title of Clarke's story, it is likely that the special alphabet used by the Lama contains 13 distinct symbols.[1] However, one question that is left unanswered is the *order* in which the names are generated by the Mark V. Two engineers, George and Chuck, are sent to Tibet with the task of assembling and then programming the Mark V. The simplest and most obvious choice — but unlikely the most efficient — would have been to program the computer to output the names in lexicographic order. Three other possibilities are also outlined in the upcoming subsections, and collectively they form a historical foundation for combinatorial generation.

1.1.1 Lexicographic Order

"I see. You've been starting at AAAAAAAAA and working up to ZZZZZZZZZ."
"Exactly — though we use a special alphabet of our own."
- The Lama in response to Dr. Wagner in *The Nine Billion Names of God*

Humanity's most established order is lexicographic order. In lexicographic order, the letters in an alphabet are given a relative order. Then, the relative order of any two words is determined by the relative order of their leftmost differing letters. For example, *aardvark* comes before *aardwolf* in the English dictionary because when comparing the leftmost differing symbols v comes before w in the English alphabet. Likewise, *sloth*_ comes before *sloths* because the absence of a letter is assumed to have the lowest order. In a similar manner, lexicographic order is used to compare the relative value of numbers in the Hindu-Arabic number system, except that in this case leading zeros need to be prefixed to ensure that the numbers have the same length. For example, 13 has a lower value than 14. Likewise, 0999 has lower value than

[1] There are 10.6-billion tridenary (base-13) strings of length nine, but not all require consideration according to the Lama's belief system.

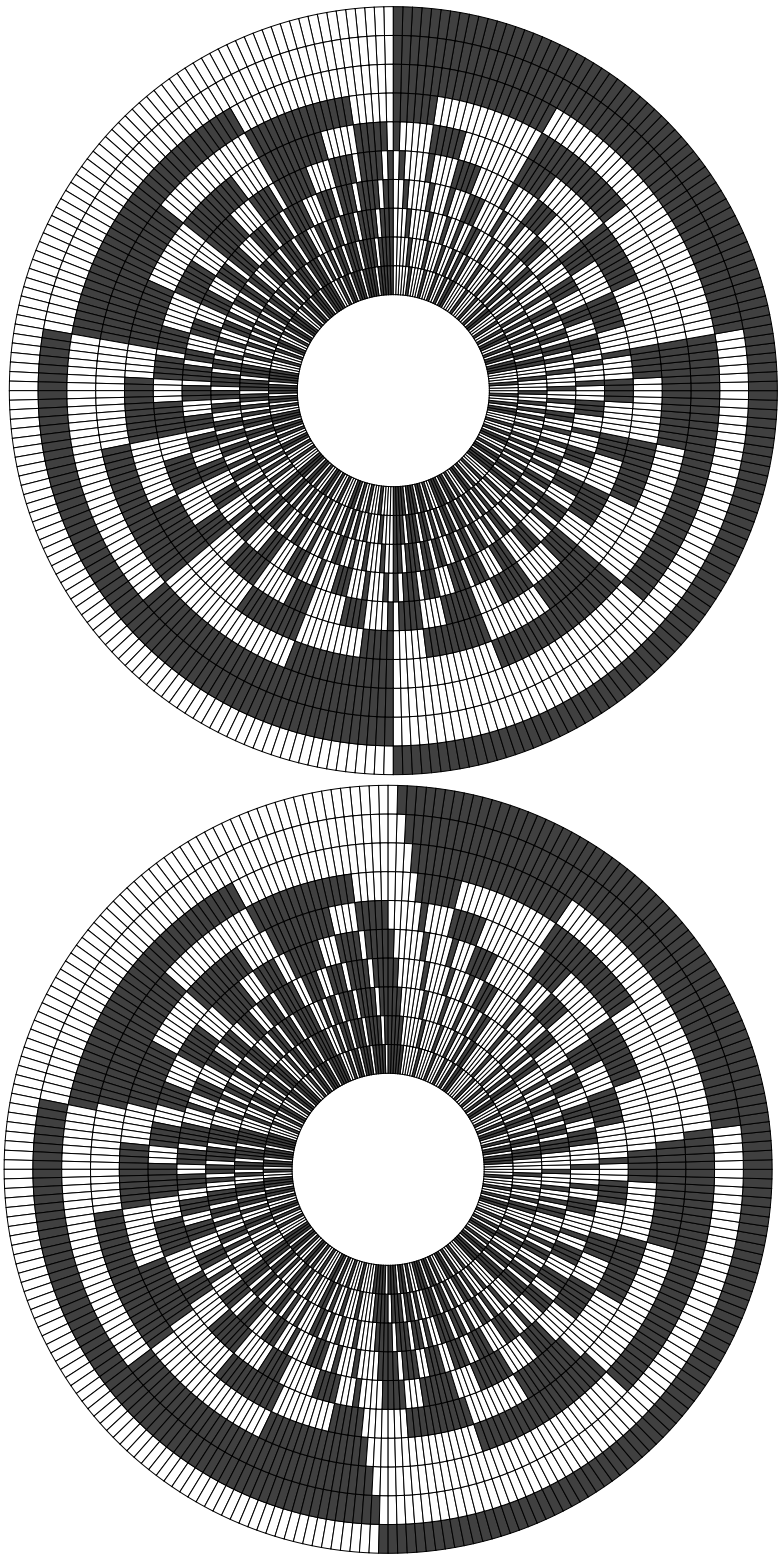


Figure 1.2: An artistic representation of co-lexicographic (above) and cool-lex (below) order for $(5, 5)$-combinations (see [6]). White and black regions represent 0 and 1 respectively. Individual strings are read along a line segment originating from the center, and the first and last strings are at either side of 12 o'clock. Cool-lex proceeds leftwards (counterclockwise) and involves left-shifts, while reverse cool-lex proceeds rightwards (clockwise) and involves right-shifts. Co-lexicographic order proceeds counterclockwise, while reverse co-lexicographic order proceeds clockwise.

1000. (Alternatively, one can consider numbers to be right-justified with the absence of a symbol having the lowest order, and so _999 is less than 1000.) Furthermore, the same principles have been applied to other objects for thousands of years. Historical examples include Shao Yung's ordering of binary hexagrams in *I Ching*:

and Yang Hsuing's ordering of ternary tetragrams:

Notice that the two philosophers had differing opinions on whether the flat "yang" should have the highest or lowest order, and whether the symbols should be read from top-to-bottom or bottom-to-top. These historical examples were brought to the author's attention by [62] and [47].

Although lexicographic order is a natural choice for man, it is not necessarily a natural choice for machine. The primary difficulty comes from the problem of *roll-over*. For example, driving one additional kilometer in a vehicle will cause its odometer to roll-over from 299999 to 300000, thereby changing the value of every digit in the process. Similarly, using nine letter words over the Lama's 13-letter language there would be roll-over from AMMMMMMMM to BAAAAAAAA. This *worst-case* behavior can be the limiting factor in many situations. For example, the Mark series of computers were known for their peculiar timing idiosyncracies, where certain operations could only be performed during certain clock cycles (see Wikipedia [98]). Accordingly, if programmed in lexicographic order, then the Mark V may have been forced to wait for the longest possible update time before outputting each successive name. Depending on its parallelization capabilities, updating each letter could have taken nine times as long as updating a single letter. Furthermore, the frequency of roll-overs could also lead to undue electrical consumption and mechanical wear on the Mark V's electromatic typewriter[2] (both of these considerations are significant when running a multi-month generator-powered project on a remote Tibetan mountaintop). For this reason, the Manhattan-based engineers may have asked their colleagues at Bell Laboratories for suggestions before embarking on their long voyage to the lamasery.

[2]The Harvard Mark IV was used by the U.S. Air Force in 1952, but the Mark V was never officially built. Otherwise, its output could have been on an early daisy-wheel printer, where each letter change requires a daisy-wheel to spin.

1.1.2 The Binary Reflected Gray Code

"Once it has been programmed properly it will permute each letter in turn and print the result. What would have taken us fifteen thousand years it will be able to do in a hundred days."

- The Lama in *The Nine Billion Names of God*

While working with Bell Laboratories, Frank Gray filed U.S. patent 2,632,058 in 1947 based on the *binary reflected code* [29]. Gray's result shows how binary strings of length n can be ordered so that successive strings differ in exactly one bit. For example, Gray's ordered list for the binary strings of length $n = 2$ is denoted $\mathcal{G}_2$ and appears below. Notice that it contains every possible binary string of length two exactly once, and that a single overlined bit is changed to obtain each successive string. For example, $0\overline{0}$ is succeeded by 01.

$$\mathcal{G}_2 = 0\overline{0}, \overline{0}1, 1\overline{1}, 10 \qquad \mathsf{reverse}(\mathcal{G}_2) = 1\overline{0}, \overline{1}1, 0\overline{1}, 00$$

$$\mathcal{G}_3 = \underbrace{00\overline{0}, 0\overline{0}1, 01\overline{1}, \overline{0}10}_{0 \cdot \mathcal{G}_2}, \underbrace{11\overline{0}, 1\overline{1}1, 10\overline{1}, 100}_{1 \cdot \mathsf{reverse}(\mathcal{G}_2)}$$

The term *reflected* in *binary reflected code* comes from the operation of reversing the order that each string appears in a list. For example, $\mathsf{reverse}(\mathcal{G}_2)$ starts with the last string in $\mathcal{G}_2$, namely 10, and ends with the first string in $\mathcal{G}_2$, namely 00. Since each successive string in $\mathsf{reverse}(\mathcal{G}_2)$ must also differ in the value of a single bit, this simple operation allows Gray to extend the pattern to binary strings with one more bit. In particular, the ordered list $\mathcal{G}_{n+1}$ is constructed by prefixing 0 to every string in $\mathcal{G}_n$, followed by prefixing 1 to every string in $\mathsf{reverse}(\mathcal{G}_n)$. This construction is illustrated above for $n = 2$. In general, $\mathcal{G}_0 = \epsilon$ (the empty string) and then for $n > 0$,

$$\mathcal{G}_{n+1} = 0 \cdot \mathcal{G}_n,\ 1 \cdot \mathsf{reverse}(\mathcal{G}_n).$$

This type of construction is *recursive* since it describes the overall structure of the ordered list in terms of smaller ordered lists of the same type. Dr. Wagner may have also been aware that the binary reflected code can be described by a highly efficient *iterative operation* which describes how to change any string in the list into the next string in the list (see Bitner-Ehrlich-Reingold [4]). Iteration is often more desirable than recursion due to its low overhead; this is especially true in this case since the available memory is limited to a small number of ferrite magnetic registers. Before discussing how the binary reflected code relates to the nine billion names of God, it is

useful to point out that if the second sublist in the above expression is not reflected, then the result is a recursive construction for binary strings in lexicographic order. In other words, the binary reflected Gray code is a subtle variation of lexicographic order.

If George and Chuck were aware of the binary reflected code, then they may have wondered if the same principle could be extended to n-tuples in general. Given an alphabet, the term *n-tuple* refers to the set of all possible strings of length n over that alphabet. For this application, George and Chuck would have been primarily interested in 9-tuples over a 13-letter alphabet. With some back-of-the-envelope reckoning, they may have realized that reversing every second sublist, that is,

$$\mathcal{G}_{n+1} = \text{A} \cdot \mathcal{G}_n,\ \text{B} \cdot \mathsf{reverse}(\mathcal{G}_n),\ \text{C} \cdot \mathcal{G}_n,\ \text{D} \cdot \mathsf{reverse}(\mathcal{G}_n),\ \ldots,\ \text{M} \cdot \mathcal{G}_n,$$

ensures that successive strings differ by the increment, or decrement, of a single letter. For example, AMMMMMMMM is followed by BMMMMMMMM in this *tridenary reflected Gray code*, and in general all roll-overs are avoided. Furthermore, a similarly efficient iterative operation also exists for this generalized notion of a *reflected Gray code* [46]. The two Americans could have used this iterative operation as a basis for programming the Mark V, and this approach would lead to significantly faster generation of each successive name (and a faster return home).

Despite its simplicity, or perhaps because of it, the binary reflected code, or *binary reflected Gray code* as it is now known, has become an extremely versatile piece of mathematics. Within information and communication technology its uses are wide and varied, with applications including analog-to-digital conversion, error correction, and decreased power consumption in hand-held devices (see Wikipedia [97]). Furthermore, the same order was used in telegraphy by Émile Baudot as early as 1878. It has also been used for other diverse purposes, including the CODACON spectrometer, and appears in research titles ranging from measurement and instrumentation to quantum chemistry (see Betta-Pietrosanto-Scaglione [3] and Sawae-Sakata-Tei-Takarabe-Manmoto [78]). To honor Gray's popularization of this ubiquitous pattern, the term *Gray code* is now synonymous with the concept of *minimal-change order*. In other words, a Gray code is an ordering of objects such that successive objects differ in some small prescribed manner.

1.1.3 de Bruijn Cycles

"The second matter is so trivial that I hesitate to mention it - but it's surprising how often the obvious gets overlooked."

- Dr. Wagner in *The Nine Billion Names of God*

If Dr. Wagner's PhD was in mathematics, then he may have been aware of *de Bruijn cycles*. Published just one year before Frank Gray's patent, Nicolaas Govert de Bruijn proved that the binary strings of length n can be packed into one string of length 2^n (see de Bruijn [13]). For example,

$$0000100110101111 \tag{1.5}$$

is a string of length $2^4 = 16$, and it contains each of the 16 binary strings of length $n = 4$ exactly once as a substring. For the sake of clarification, the substrings appear in the following order

$$0000, 0001, 0010, 0100, 1001, 0011, 0110, 1101,$$
$$1010, 0101, 1011, 0111, 1111, 1110, 1100, 1000. \tag{1.6}$$

Notice that each substring overlaps the previous substring in three bits, and the final three substrings wrap-around from the end of (1.5) to the beginning. Alternatively, each successive binary string in (1.6) is obtained by removing the leftmost bit and inserting a new rightmost bit. Therefore, de Bruijn cycles also produce a type of minimal-change order. Similar de Bruijn cycles also exist for n-tuples over any alphabet, including 9-tuples over a 13-letter alphabet. Depending upon the exact details of the Lama's belief system, the overlapping of potential names may have been acceptable. For example,

AAAAAAAAABA···

may have been an acceptable encoding of the names AAAAAAAAA, AAAAAAAAB, AAAAAAAABA, and so on. Allowing this overlap could have drastically reduced the time taken to complete the project. This is especially true considering that the bottleneck in creating lists is often the output of strings, as opposed to the creation of the strings within the computer's memory. By using a de Bruijn cycle, the engineers would have reduced the amount of output by a factor of nine, not to mention the reduction in downtime required for replacing worn-out parts of the electromatic typewriter.

"All we need to do is to find something that wants replacing during one of the overhaul

> periods — something that will hold up the works for a couple of days. We'll fix it, of course, but not too quickly."
>
> - Chuck in *The Nine Billion Names of God*

If George and Chuck wanted to use this approach, then they would have needed to be able to efficiently create a suitable de Bruijn cycle. Although it was not part of de Bruijn's original research, it is possible to induce a de Bruijn cycle for the n-tuples over an alphabet by a reduction from lexicographic order of the same words. For the sake of illustration, consider the lexicographic order of binary strings of length four

$$0000, \underline{0001}, 0010, \underline{0011}, 0100, 0\underline{101}, 0110, \underline{0111},$$
$$1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111. \quad (1.7)$$

The de Bruijn cycle in (1.5) is obtained by concatenating the underlined portions of the strings in (1.7). The underlined portions represent the non-repeating or *aperiodic prefix* of the corresponding string. For example, the 01 is underlined in 0101 since 0101 can be formed by repeating 01 twice. Likewise, 000$\underline{0}$ since 0000 can be formed by repeating 0 four times. Furthermore, a string only has an underlined portion if it is the *lexicographically smallest* in its rotation set. The *rotation set* of a string $\boldsymbol{s}$ is denoted by $\circlearrowleft(\boldsymbol{s})$ and is a set containing every rotation of the string. For example,

$$\circlearrowleft(0010) = \{0010, 0100, 1000, 0001\}.$$

Within this set, 0001 is the lexicographically smallest. Thus, 0010 does not have an underlined portion in (1.7), but 0001 does. Similarly, 0101 and 1111 also have underlined portions. A very interesting theoretical result discussed in [46] that dates back to the 1930s (see Martin [55]) is the fact that this technique always creates the lexicographically smallest possible de Bruijn cycle. More recent research has shown that this de Bruijn cycle can be generated efficiently (see Fredericksen-Maiorana [21], Fredericksen-Kessler [20], and Ruskey-Savage-Wang [66]). This construction is known as the *FKM algorithm*.

> "A rather more interesting problem is that of devising suitable circuits to eliminate ridiculous combinations. For example, no letter must occur more than three times in succession."
>
> - The Lama in *The Nine Billion Names of God*

Roughly speaking, *backtracking* involves the avoidance of unnecessary work. As mentioned by the Lama in his initial meeting with Dr. Wagner, certain tuples need

not be generated, including those that contain three identical letters in succession. Although fewer than a half-billion possibilities have three identical letters in succession, George and Chuck surely would have preferred to spend an extra half-week back in Manhattan than waiting for the Mark V to output these "ridiculous combinations". While backtracking to avoid these possibilities would be relatively easy using lexicographic order or the tridenary reflected Gray code, it would be significantly more challenging using de Bruijn cycles.

Similar to the binary reflected code, de Bruijn cycles have also proven to have a wide variety of applications, and have been the subject of a number of generalizations referred to as *universal cycles* as initiated by Chung-Diaconis-Graham [9].

1.1.4 Johnson-Trotter-Steinhaus Order

> This, thought George, was the craziest thing that had ever happened to him. *Project Shangri-La*, some wit back at the labs had christened it.
>
> \- in *The Nine Billion Names of God*

The *Shangri-La* reference is to a 1937 movie entitled *Lost Horizon* in which a group of westerners find themselves "trapped" by a High Lama (played by Sam Jaffe) at an idyllic Himalayan Shangri-La. As is the case in the movie, the westerners in this story have concerns about staying too long in the mountaintop paradise. With the Mark V's job quickly coming to an end, and their transport not arriving for another week, Chuck is worried about the monks' reaction when the last name is printed and the world does not end. On the other hand, George fears that the monks will re-examine their centuries-old calculations and conclude that the search for His names is not complete.

> Just what obscure calculations had convinced the monks that they needn't bother to go on to words of ten, twenty, or a hundred letters, George didn't know. One of his recurring nightmares was that there would be some change of plan and that the High Lama (whom they'd naturally called Sam Jaffe, though he didn't look a bit like him) would suddenly announce that the project would be extended to approximately A.D. 2060.
>
> \- George in *The Nine Billion Names of God*

Although the prospect of generating longer words was daunting to George, the good news was that the Mark V was already programmed to output n-tuples. Thus, assuming he could teach the junior monks to run the scheduled maintenance and repairs, he could then, in theory, make a small change to the program and return home. However, this would not be true if he needed to reprogram the automatic

sequence computer to generate a different type of list. Judging from the Lama's instructions on avoiding consecutive identical letters, George could have wondered if the next task would involve writing out the six billion permutations of the special 13-letter alphabet. The *permutations* of a set, or alphabet, include every rearrangement of its symbols. For example, the permutations of $\{1, 2, 3, 4\}$ are

$$1234, 1243, 1324, 1342, 1423, 1432, 2134, 2143, 2314, 2341, 2413, 2431,$$
$$3124, 3142, 3214, 3241, 3412, 3421, 4123, 4132, 4213, 4231, 4312, 4321.$$

Of course, the Mark V could be programmed to output the permutations in lexicographic order, as illustrated above. However, similar timing issues would still exist, with AMLKJIHGFEDCB rolling-over to BACDEFGHIJKLM in this alternate list of names. Adding to his concern would be the realization that the minimal-change operation used in the reflected Gray code simply cannot work for permutations. For instance, given the permutation ABCDEFGHIJKLM, consider what happens when a single letter, say B, is changed to another letter, say L. The resulting string, ALCDEFGHIJKLM is not a permutation since it is missing the letter B and contains two copies of L. However, if the original copy of L is changed to B, then the resulting string, ALCDEFGHIJKBM is again a permutation. The net result is that the B and L have been transposed. In general, *transpositions* are illustrated using line-segments (i.e., $\underbrace{abcde}f = aecdbf$). Ten years after Clarke's short story was published, the Johnson-Trotter-Steinhaus minimal-change order for permutation was published [42, 90, 87]. Within this order, each successive permutation differs by an *adjacent-transposition*, meaning that the transposed symbols are next to each other. For example, the order for $n = 4$ appears below

$$12\underbrace{34}, 1\underbrace{24}3, \underbrace{14}23, 4\underbrace{12}3, \underbrace{41}32, 1\underbrace{43}2, 1\underbrace{34}2, \underbrace{13}24, 3\underbrace{12}4, 3\underbrace{14}2, \underbrace{34}12, 4\underbrace{31}2,$$
$$\underbrace{43}21, 3\underbrace{42}1, 32\underbrace{41}, \underbrace{32}14, 23\underbrace{14}, 2\underbrace{34}1, \underbrace{24}31, 42\underbrace{31}, \underbrace{42}13, 2\underbrace{41}3, \underbrace{21}43, 2134.$$

Within the above list[3], notice that the largest symbol, 4, sweeps back and forth, with a single pause when it reaches the extreme left and right positions. This idea drives the entire order since there are n permutations of $\{1, 2, \ldots, n\}$ that can be obtained by inserting the symbol n into a fixed permutation of $\{1, 2, \ldots, n-1\}$. The sweeping motion exhausts all n of these possibilities for a permutation of $\{1, 2, \ldots, n-1\}$, and is fol-

[3]This list was obtained from Frank Ruskey's *The Combinatorial Object Server* [81] where one can find many orders including others from this thesis.

lowed by the single transposition that creates the next permutation of $\{1, 2, \ldots, n-1\}$. Historically, the Johnson-Trotter-Steinhaus order was influential to the development of combinatorial generation in academia, and by the 1970s there were several survey papers discussing the merits of various methods for generating permutations (see Ord-Smith [56, 57], Sedgewick [79], and Roy [61]). Generating permutations by transpositions was also important to the private sector by this time, as evidenced by an internal memorandum by Goldstein-Graham [27] from Bell Laboratories in 1964.

Although George could not look into the future for help, he could ask for divine intervention. Such a request may have reminded George of Sunday morning, and the wonderfully intricate patterns of music played in many churches. The art, and science, of *method ringing* was developed in English church towers during the 17th century (see Wikipedia [100]). The oversized tuned bells in these towers were not particularly adept at creating melodies. For this reason, the ringers would focus on sounding all of the bells in some order, one after another. Such orders are called *rounds*, and are essentially permutations of $\{1, 2, \ldots, n\}$, where n is the total number of bells and ringers. If the ringers wished to play two rounds in succession, then the second round would have to be played in a similar order to the first. For example, the succession of the following two rounds would not be feasible

$$1\ 2\ 3\ 4\ 5\ 6\ 7\ 8$$
$$1\ 6\ 3\ 4\ 5\ 2\ 7\ 8.$$

This is due to the fact that in the second round, 6 would still be completing its ringing cycle from the first round. For this reason, the study of *change ringing* focuses on playing successive rounds such that a small number of adjacent-transpositions occur. (The adjacent-transposition operation also ensures that each ringer maintains a relatively consistent cadence.) For example, the succession of the following two rounds would be possible in change ringing.

$$1\ 2\ 3\ 4\ 5\ 6\ 7\ 8$$
$$1\ 3\ 2\ 4\ 5\ 7\ 6\ 8$$

One of the biggest goals in change ringing is to avoid repeated rounds, and method ringing is focused on using mathematical patterns for this purpose. If every possible round is played exactly once, then the performance is called an *extent*. Extents with $n = 7$ are called *peals*, and the $7! = 5040$ distinct rounds are played thousands of times per year. For this reason, George's prayers could have been answered one Sunday morning in New York at a holy "bell lab". In particular, the Johnson-Trotter-Steinhaus order may have been played as a send-off for George's Tibetan adventure.

A verbose description of *plain change* order and its variations appear in the 1668 text *Tintinnalogia* by Duckworth-Stedman [15] subtitled *Wherein is laid down plain and easie Rules for Ringing all sorts of Plain Changes.*

1.2 Contemporary Results

The historical foundations of combinatorial generation were surveyed in the previous section. To motivate these foundations, *The Nine Billion Names of God* was used to demonstrate the utility of the reflected Gray code and de Bruijn cycles for n-tuples, as well as the Johnson-Trotter-Steinhaus order for permutations. Implicit in the discussions of these minimal-change orders was a particular sequence of three steps designed to achieve the goal of "efficiently creating all possibilities". This sequence of steps includes:

1. Model the possibilities by a suitable combinatorial object.
2. Find a minimal-change order for the instances of the combinatorial object.
3. Implement an efficient algorithm that generates the minimal-change order.

(Within the last two steps, a universal cycle is included in the possible minima-change orders.) The primary purpose of this section is to survey contemporary results in combinatorial generation, and it does so by following the three steps listed above. Section 1.2.1 introduces several additional combinatorial objects including necklaces and trees, as well as the concept of a fixed-content language. Sections 1.2.2 and 1.2.3 present contemporary transposition Gray codes and shift Gray codes for fixed-content languages, as well as the new concept of a shorthand universal cycles for fixed-content languages. Section 1.2.4 then completes the sequence of steps by discussing known algorithms, and the measures of efficiency that are used in this thesis.

The secondary purpose of this section is to illustrate that optimization problems can also be solved by following the same sequence of three steps. In an *optimization problem*, each instance of a combinatorial object has an associated *value*, and the goal is to find an instance with the greatest value. One way to solve an optimization problem is to generate each possibility and *evaluate* it. This approach is often (derisively) referred to as a *brute force solution* within computer science. Despite this fact, there are many real-world situations when this approach is either necessary or desirable. For example, brute force is often employed when the underlying problem is extremely difficult. Section 1.2.5 returns to the lamasery to introduce a notoriously difficult problem known as the stacker-crane problem. Discussions from Sections 1.2.1-1.2.4

contribute to an understanding of how to implement a highly-optimized brute force solution to the stacker-crane problem. In particular, this optimized brute force solution provides a concrete real-world application of the shift Gray codes and efficient algorithms developed in this thesis.

1.2.1 Combinatorial Objects

"What source of electrical energy have you?"
"A diesel generator providing 50 kilowatts at 110 volts. It was installed about five years ago and is quite reliable. It's made life at the lamasery much more comfortable, but of course it was really installed to provide power for the motors driving the prayer wheels."
- Dr. Wagner and the Lama in *The Nine Billion Names of God*

To model the possible solutions of a given problem, it is helpful to first have an understanding of several basic combinatorial objects. Thus far, the reader has been introduced to n-tuples and permutations. This section expands this list of combinatorial objects by introducing necklaces and trees. This section also discusses the value of a combinatorial object based on its pairs of adjacent symbols. Before beginning this discussion, it is useful to note that the term *combinatorial object* is often used to refer to a specific *instance* of a combinatorial object (i.e., a specific permutation of $\{1, 2, \ldots, n\}$) or to the set of all such instances (i.e., the set of all permutations of $\{1, 2, \ldots, n\}$). In other words, "combinatorial object" is a figure of speech known as a synecdoche. More precisely, s *synecdoche* occurs whenever a specific term is used to refer to a more general term, or a general term is used to refer to a more specific term (see `dictionary.com` [36]).

In his initial meeting with Dr. Wagner, the Lama mentions that a diesel generator was installed at the lamasery to provide power for the motors that automatically turn their prayer wheels. Although prayer wheels are most often adorned with the mantra *Om Mani Padme Hum*, it is also common to inscribe their outer surface with the *Eight Auspicious Signs* (see Wikipedia [101]). Although the precise order can differ around the world, in Tibet the *Ashtamangala* typically refers to the following order of these symbols: endless knot, lotus flower, victory banner, wheel of Dharma, treasure vase, golden fish pair, parasol, and conch shell. To provide additional karma in advance of the Lama's trip to New York, it would not have been surprising if the lamasery dedicated a prayer wheel to each of the possible circular orderings of

the Eight Auspicious Signs.[4] More abstractly, this section describes these possible inscriptions on the lamasery's prayer wheels as the necklaces containing eight beads of different colours.

To formalize the definition, recall the notion of string rotations from Section 1.1.3. A *necklace* is an equivalence class of strings under rotation. By convention, the symbols in a necklace are known as *coloured beads*. For example, the necklaces containing two black beads, two grey beads, and two white beads are illustrated in Figure 1.3. Each necklace in Figure 1.3 can be represented by a string of symbols by mapping

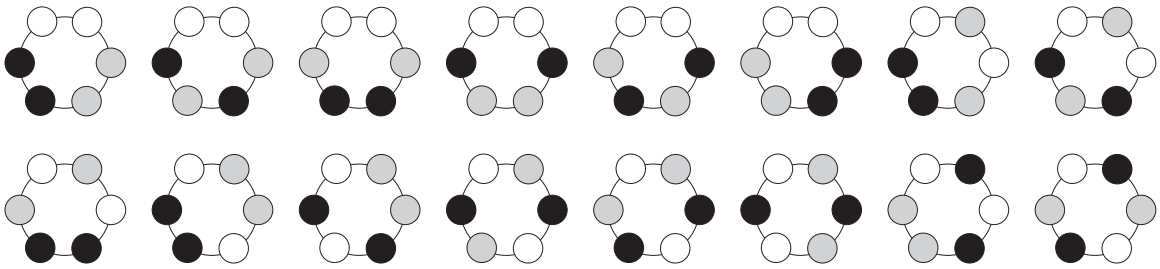


Figure 1.3: Necklaces containing two black, two grey, and two white beads.

● ↔ 1, ◍ ↔ 2, ○ ↔ 3, and then by recording one of its clockwise rotations. In particular, it is customary to use the lexicographically largest or smallest clockwise rotation. For example, the lexicographically largest representation of the necklace in the top-left corner of Figure 1.3 is 332211.

While necklaces can be used to encapsulate circular orders, it can also be desirable to encapsulate hierarchical orders. For example, the High Lama would have tutored several students at the lamasery over the years, and then these students would have themselves tutored additional students, and so on. The resulting student-teacher relationship would form a type of family tree. This thesis will focus on ordered trees with a fixed-branching sequence in Chapter 2. For the sake of illustration, Figure 1.4 depicts the ordered trees containing a single node with three children, two nodes with a single child, and three leaves (in black). Each tree in Figure 1.4 can be represented by a string of symbols by mapping each node to its number of children, and then by recording its pre-order traversal. For example, the leftmost tree in Figure 1.4 can be represented by 311000.

Although the combinatorial objects in these two figures are quite different, they are similar in at least one respect. To formalize this connection, say that a *multiset* is a

[4]There are 5040 such possibilities since the endless knot can be followed by any of the 7! permutations of the remaining symbols.

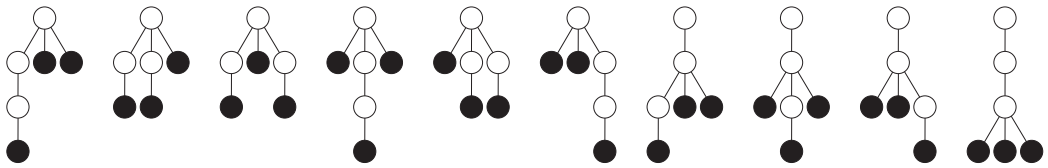


Figure 1.4: Ordered trees with branching sequence $\{0, 0, 0, 1, 1, 3\}$.

set that allows repetition, and a *multiset permutation* is a permutation of a multiset. Notice that each necklace in Figure 1.3 can be encoded as a permutation of the multiset $\{1, 1, 2, 2, 3, 3\}$, and each tree in Figure 1.4 can be encoded as a permutation of the multiset $\{0, 0, 0, 1, 1, 3\}$. As a third example, the set of all $\frac{4!}{1!\cdot 2!\cdot 1!} = 12$ multiset permutations over $\{1, 2, 2, 3\}$ appears below

$$\{1223, 1232, 1322, 2123, 2132, 2213, 2231, 2312, 2321, 3122, 3212, 3221\}. \tag{1.8}$$

Multiset permutations generalize both permutations and (s, t)-combinations. In particular, permutations are the multiset permutations of the set $\{1, 2, \ldots, n\}$, and (s, t)-combinations are the permutations of the multiset containing s copies of 0 and t copies of 1. More generally, a *fixed-content language* is any set of strings that contain the same multiset of symbols. In particular, fixed-content languages can be used to represent the necklaces in Figure 1.3, the trees in Figure 1.4, and the set of multiset permutations found in (1.8). On the other hand, $\{122, 112\}$ is a simple example if a language that does not have fixed-content.

A fixed-content language whose strings contain only 0s and 1s is known as a *fixed-density language* with *density* referring to the number of 1s. In other words, fixed-density languages are subsets of (s, t)-combinations.

As previously mentioned, each instance of a combinatorial object can have an associated value. (This is not the case in *The Nine Billion Names of God* since each n-tuple has equal importance as a possible name of God.) For example, if a combinatorial object is represented by a string, then its associated value could depend on its pairs of adjacent symbols. Given a string $s_1 s_2 \cdots s_n$, the *ordered pairs of adjacent symbols* and *unordered pairs of adjacent symbols* of $\boldsymbol{s}$ are respectively

$$\underbrace{s_1 s_2,\ s_2 s_3,\ \ldots,\ s_{n-1} s_n}_{\text{ordered pairs of adjacent symbols}} \quad \text{and} \quad \underbrace{\{s_1, s_2\},\ \{s_2, s_3\},\ \ldots,\ \{s_{n-1}, s_n\}}_{\text{ordered pairs of adjacent symbols}}. \tag{1.9}$$

For simplicity, these terms are henceforth abbreviated to *ordered pairs* and *unordered pairs*.

Another important consideration is the representations of a combinatorial object. A *representation* of a combinatorial object is a data type that can be used to store the object, together with a convention for how to store the object in this data type. For example, this section section discussed how certain necklaces and trees can be represented by fixed-content languages using the convention of a lexicographically largest rotation and a pre-order traversal, respectively. As another example, Figure 1.1 showed that there are simple conventions for representing (s, t)-combinations in an array, linked list, or computer word. In some situations, the representation is fixed by the application at hand.

Before concluding this section, an important point involving fixed-content languages must be raised. Many combinatorial objects are naturally represented by sets of strings that do not have fixed-content. On the other hand, many of these combinatorial objects have subsets that are naturally represented by fixed-content languages. At first it may seem to be more useful to have a minimal-change order for the unrestricted language that does not have fixed-content. However, minimal-change orders for the restricted fixed-content languages are often more useful. This is due to the fact that minimal-change orders for the fixed-content subsets can often be combined into minimal-change orders for the superset. Conversely, it is much less likely that a minimal-change order for the superset can be divided into minimal-change orders for the fixed-content subsets. This point is not discussed again until Chapter 5, but it should be considered a primary motivation for exploring the existence of minimal-change orders for fixed-content languages.

1.2.2 Minimal-Change Orders

> "Your Mark V computer can carry out any routine mathematical operation involving up to ten digits. However, for our work we are interested in letters, not numbers. As we wish you to modify the output circuits, the machine will be printing words, not columns of figures."
>
> - The Lama speaking to Dr. Wagner in *The Nine Billion Names of God*

This thesis focuses on combinatorial objects that can be represented by fixed-content languages. Minimal-change orders for fixed-content languages are most commonly based on transpositions or shifts. Besides their mathematical simplicity, these two operations are of practical importance to computer scientists. In particular, transpositions are basic operations in arrays, and shifts are basic operations in linked lists and binary computer words. This section compares these two operations with respect to their effect on adjacent symbols, and then provides a technical discussion

on specific transposition Gray codes and shift Gray codes that are known to exist.

No discussion of minimal-change orders is complete without mention of a general result by Sekanina [80]. The result states that the cube of any connected graph has a Hamiltonian path. (A *Hamiltonian path* is a path that travels through each vertex of a graph exactly once, and the *cube* of a graph is obtained by adding an arc from node u to node v so long as the shortest path from u to v is at most three.) Therefore, if σ is some operation on strings, and if $\mathbf{L}$ is a set of strings in which any string can be transformed into any other string by repeated applications of σ, then there exists an ordering of the strings in $\mathbf{L}$ such that at most three applications of σ are necessary to transform any string into the next string. In general, a minimal-change order using at most k applications of σ between successive objects is known as a $k-\sigma$ *Gray code* for $\mathbf{L}$. (When $k = 1$ the term is shortened to a σ *Gray code.*) The aforementioned $3-\sigma$ Gray code due to Sekanina can be obtained by a prepostorder traversal of any spanning tree in the initial graph (see [47] for further details).

Operations

Before he could finish the sentence, the Lama had produced a small slip of paper.
"This is my certified credit balance at the Asiatic Bank."
"Thank you. It appears to be–ah–adequate."
- The Lama and Dr. Wagner in *The Nine Billion Names of God*

The most important measures of a minimal-change order are the type of operation it uses, and its number of applications required to create successive instances of the underlying combinatorial object within the minimal-change order. In general, no single operation is more useful than another. The reason for this fact has to do with the different representations of a combinatorial object. For example, if an application forces the combinatorial object to be stored in an array, then a minimal-change order involving transpositions will likely result in a faster algorithm than a minimal-change order involving shifts. On the other hand, an algorithm based on linked-lists may be more efficient if it is based on a minimal-change order involving shifts. In general, the relative expense of an operation is dependent on the representation of the underlying combinatorial object. On the other hand, it is almost always desirable to reduce the number of application required to create successive instances.

A second expense for an operation arises when the instances of a combinatorial object have an associated value. To illustrate this point, let us compare the expense of transpositions and shifts with respect to the ordered and unordered pairs described in (1.9). To aid in the comparison, another natural operation on strings is briefly

considered. A *substring-reversal*, or simply *reversal*, replaces a substring with its reversal. This operation is illustrated using bi-directional arrows (i.e., $a\overleftrightarrow{bcde}f = aedcbf$). Notice that an adjacent-transposition is a special case of a left-shift, right-shift, and substring-reversal, as illustrated below

$$123\underbrace{45}6 = 12\overleftarrow{345}6 = 12\overrightarrow{345}6 = 12\overleftrightarrow{345}6 = 124356.$$

(*Substring-reversal Gray codes* are not covered in detail in this thesis, although it is mentioned that combinatorial generation using this operation has applications to computational biology [83], and an elegant result for generating permutations by substring-reversals was discovered by Zaks [104].)

When applying transpositions and shifts to a string, only a small number of ordered and unordered pairs can change. For example, an adjacent-transposition $\cdots\underbrace{s_i s_{i+1}}\cdots$ will change at most three ordered pairs, and at most two unordered pairs. (In particular, the ordered pairs $s_{i-1}s_i$, $s_i s_{i+1}$, and $s_{i+1}s_{i+2}$ are replaced by the ordered pairs $s_{i-1}s_{i+1}$, $s_{i+1}s_i$, and $s_i s_{i+2}$.) Similarly, a prefix-shift $\overleftarrow{s_1\cdots s_i}\cdots$ will change at most two pairs, regardless of whether the pairs are ordered or unordered. (In particular, the ordered pairs $s_{i-1}s_i$ and $s_i s_{i+1}$ are replaced by the ordered pairs $s_i s_1$ and $s_{i-1}s_{i+1}$.) A substring-reversal $\cdots\overleftrightarrow{s_i\cdots s_j}\cdots$ can change at most two unordered pairs. (In particular, the unordered pairs $\{s_{i-1}, s_i\}$ and $\{s_j, s_{j+1}\}$ are replaced by the unordered pairs $\{s_{i-1}, s_j\}$ and $\{s_i, s_{j+1}\}$.) On the other hand, the number of ordered pairs that are changed by a substring-reversal is bounded only by the length of the string. (In particular, $\overleftrightarrow{s_1\cdots s_n}$ changes all $n-1$ ordered pairs.) These bounds are presented in Table 1.1, along with similar bounds for general shifts and transpositions. Table 1.1 is discussed again in Section 1.2.4. In general, this type of analysis provides an important motivation for developing different types of Gray codes for the same combinatorial object.

	adjacent-transposition	transposition	prefix-shift	shift	reversal
unordered	2	4	2	3	2
ordered	3	4	2	3	n-1

Table 1.1: The maximum number of ordered and unordered pairs that are changed after applying various operations to a string of length n.

Before concluding this section it is mentioned that many of the shift Gray codes presented in this thesis can also be viewed as transposition Gray codes. For example, the left-shift Gray code of $(3,3)$-combinations given in (1.2) is expressed below using

transpositions

$$\underbrace{0}\underbrace{11100}, \underbrace{1}\underbrace{01100}, \underbrace{11}\underbrace{0}\underbrace{0}, \underbrace{0}\underbrace{11010}, \underbrace{1}\underbrace{0}\underbrace{1}\underbrace{0}, \underbrace{0}\underbrace{1}\underbrace{0}\underbrace{110}, \underbrace{0}\underbrace{0}\underbrace{1110}, \underbrace{1}\underbrace{00110}, \underbrace{11}\underbrace{0}\underbrace{0}\underbrace{1}\underbrace{0}, \underbrace{0}\underbrace{11001},$$
$$\underbrace{1}\underbrace{0}\underbrace{1}\underbrace{001}, \underbrace{0}\underbrace{1}\underbrace{0}\underbrace{101}, \underbrace{0}\underbrace{0}\underbrace{1101}, \underbrace{1}\underbrace{00}\underbrace{1}\underbrace{01}, \underbrace{0}\underbrace{1}\underbrace{0}\underbrace{011}, \underbrace{00}\underbrace{1}\underbrace{0}\underbrace{11}, \underbrace{0}\underbrace{00111}, \underbrace{1}\underbrace{00011}, \underbrace{11}\underbrace{0}\underbrace{0}\underbrace{01}, \underbrace{111}\underbrace{000}.$$

Notice that the above strings are in a 2-transposition Gray code. For this reason, (1.2) is both a left-shift Gray code and a 2-transposition Gray code. More generally, operations (i) and (ii) can always be described by transposing one or two pairs of bits. Furthermore, this is true for the general result on shift Gray codes for fixed-density languages developed in this thesis.

Transposition Gray Codes

Given the Johnson-Trotter-Steinhaus order, it becomes natural to ask which other fixed-content languages can be generated by single adjacent-transpositions. There are some very basic obstructions to this goal, so the question was later expanded to single transpositions, and then to a constant number of transpositions. For example, it is not possible to generate (2,2)-combinations using single adjacent-transpositions. As with many results in combinatorial generation, it is helpful to frame the argument in terms of graph theory. Consider the *adjacent-transposition graph* in Figure 1.5 which contains a vertex for every (2,2)-combination and an edge for every pair of vertices whose strings differ by an adjacent-transposition. Since there is no Hamilton path in the graph there is no adjacent-transposition Gray code for (2,2)-combinations.

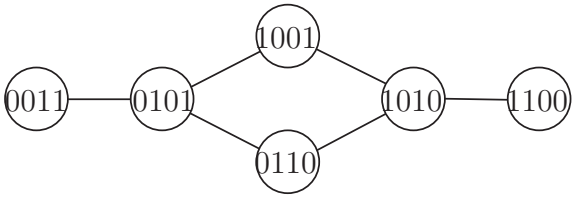


Figure 1.5: Adjacent-transposition graph for (2,2)-combinations.

More generally, a parity argument by Ruskey [64] shows that (s,t)-combinations have an adjacent-transposition Gray code if and only if s and t are both odd. On the other hand, it is possible to relax the adjacent-transposition operation and create a transposition Gray code for combinations. For example, the relative order of the (s,t)-combinations within the binary reflected code on $n = s + t$ bits is a transposition Gray code (see Liu-Tang [54]) that can be generated by an efficient algorithm (see Ehrlich [4]). This order is illustrated below for $s = 2$ and $t = 2$, where strings are

crossed out if they are not (2,2)-combinations

$$\mathcal{G}_4 = \not{0}\not{0}\not{0}\not{0}, \ \not{1}\not{0}\not{0}\not{0}, \ 1100, \ \not{0}\not{1}\not{0}\not{0}, \ 0110, \ \not{1}\not{1}\not{1}\not{0}, \ 1010, \ \not{0}\not{0}\not{1}\not{0},$$
$$0011, \ \not{1}\not{0}\not{0}\not{1}, \ \not{1}\not{1}\not{1}\not{1}, \ \not{0}\not{1}\not{1}\not{1}, \ 0101, \ \not{1}\not{1}\not{0}\not{1}, \ 1001, \ \not{0}\not{0}\not{0}\not{1}.$$

In general, the transposed symbols in this Gray code are arbitrarily far apart. However, it is possible to create a *two-close* transposition Gray code for (s,t)-combinations, in which each transposition involves symbols that are separated by at most one other symbol (see Ruskey [63] and Chase [8]). More generally, efficient algorithms and two-close transposition Gray codes exist for k-ary Dyck words (see Vajnovszki-Walsh [94]). Further results along this line are contained in the *Generating All Trees* fascicle of *The Art of Computer Programming* [47].

Transposition Gray codes also exist for multiset permutations. These Gray codes are the basis of efficient algorithms (see Ko-Ruskey [48], Takaoka [89], Vajnovszki [92], and Korsh-LaFollette [52]). Multiset permutations and balanced parentheses can be simultaneously generalized to linear-extensions of partially ordered sets, which are discussed in Section 2.3. In this case transposition Gray codes exist in some cases (see Pruesse-Ruskey [58], [64], and Stachowiak [85]), but not all cases (see Pruesse-Ruskey [60]). In all cases, Canfield-Williamson [7] showed that a constant number of transpositions can be used to create efficient algorithms for generating them. Further generalizations have been considered including *basic words of anti-matroids* in which 2-transposition Gray codes (using one or two transpositions between successive objects) are known to exist by Pruesse-Ruskey [59].

Although there are no known Gray codes for multiset necklaces, an efficient lexicographic algorithm is known (see Sawada [75]). Furthermore, when the multiset is restricted to contain only 0s and 1s then transposition Gray codes (see Wang-Savage [96] and Ueda [91]) and efficient generating algorithms (see Ruskey-Sawada [67, 68]) are known to exist, although the representatives used in this case are not always lexicographically smallest or largest. (Multiset necklace languages are also known as *fixed-content necklace languages*, and *fixed-density necklace languages* refer to the binary case.) Efficient algorithms for generating *unlabeled necklaces* (which allow the symbols to be permuted) exist (see Ruskey-Sawada [69]), *binary necklaces* (which have no density restriction) do not have a Gray code changing a single bit [96] but do have a Gray code changing at most two bits (see Vajnovszki [93]), and Gray codes changing at most three symbols also exist for unrestricted necklaces over arbitrary bases (see Weston-Vajnovszki [95]).

Shift Gray Codes

In general, shift Gray codes have received much less attention from the academic community. Excluding the aforementioned results involving adjacent-transpositions, shift Gray codes were previously known for multiset permutations (see Korsh-Lipschutz [53]) and linear extensions of partially ordered sets (see Korsh-LaFollette [51]). These Gray codes have efficient algorithms, although the implementations span several pages with multiple instructions on each line. Prefix-shift Gray codes for permutations were also known to exist (see Langdon [25, 26] and Corbett [12] and [40]).

1.2.3 Universal Cycles

To understand the relationship between fixed-content languages and universal cycles, consider the problem of constructing a universal cycle for the six permutations of $\{1, 2, 3\}$. Since the universal cycle must contain 321, then it is safe to assume that the alleged universal cycle is of the form $321xyz$ where $x, y, z \in \{1, 2, 3\}$. Within this alleged universal cycle, $21x$ is a substring, and therefore $21x$ must be a permutation of $\{1, 2, 3\}$. Thus, $x = 3$. Similarly, $1xy = 13y$ is a substring of the universal cycle and so $y = 2$. Likewise, $z = 1$. However, 321321 is certainly not a universal cycle for the permutations of $\{1, 2, 3\}$ since it contains two copies of 321 and no copies of 123. In general, universal cycles for fixed-content languages rarely exist. (More precisely, they exist if and only if the language is comprised of every rotation of a single string.)

To get around this limitation, one can use an alternate representation for each permutation. For example, the six-digit string below on the left is an *order-isomorphic universal cycle* for the permutations of $\{1, 2, 3\}$. A string is order-isomorphic to a permutation of $\{1, 2, \ldots, n\}$ if the string contains n distinct integers whose relative orders are the same as in the permutation.

$$\underbrace{321341}_{\text{order-isomorphic univeral cycle}} \qquad \underbrace{321, 213, 134, 341, 413, 132}_{\text{substrings}} \quad \underbrace{321, 213, 123, 231, 312, 132}_{\text{permutations}}$$

In particular, the substrings of the order-isomorphic universal cycle appear above in the middle, and the corresponding permutations appear above on the right. In the above example, one additional symbol was required since the symbols in $\{1, 2, 3, 4\}$ were used for the permutations of $\{1, 2, 3\}$. Johnson [41] recently confirmed a long-standing and difficult conjecture [9] by showing that one additional symbol is sufficient for making order-isomorphic universal cycles for the permutations of $\{1, 2, \ldots, n\}$.

One drawback of using order-isomorphism is that the resulting permutations can vary significantly from one to the next.

This thesis introduces the idea of using *shorthand isomorphism*. The shorthand representation of a permutation is simply the permutation with its last symbol removed. For example, the shorthand representation of 12345 is 1234. An example of a *shorthand universal cycle* for the permutations of $\{1, 2, 3\}$ appears below, along with its substrings of length two and the permutations that result from suffixing the last "missing" symbol

$$\underbrace{321312}_{\text{shorthand universal cycle}} \qquad \underbrace{32, 21, 13, 31, 12, 23}_{\text{substrings}} \qquad \underbrace{321, 213, 132, 312, 123, 231}_{\text{permutations}}.$$

Notice that the resulting permutations differ from one another in a very predictable fashion. In particular, each successive permutation differs by shifting the first symbol to the right past all of the other symbols, or past all of the other symbols except one. For example, if 12345 is a substring of a shorthand universal cycle for the permutations of $\{1, 2, 3, 4, 6\}$ then the next symbol must either be 1 or 6. These two possibilities, along with their substrings of length five and the result permutations appear below

$$\underbrace{123456\ldots}_{\text{shorthand universal cycle}} \qquad \underbrace{12345, 23456, \ldots}_{\text{substrings}} \qquad \underbrace{\overrightarrow{123456}, 234561, \ldots}_{\text{permutations}}$$

$$\underbrace{123451\ldots}_{\text{shorthand universal cycle}} \qquad \underbrace{12345, 23451, \ldots}_{\text{substrings}} \qquad \underbrace{\overrightarrow{12345}6, 234516, \ldots.}_{\text{permutations}}$$

Notice that 123456 is followed either by 234561 or 234516. In general, shorthand universal cycles always provide prefix-shift Gray codes that shift the first symbol into the last or second-last position. The former case provides a slight improvement over general prefix-shifts with respect to the number of ordered and unordered pairs that change. In particular, the prefix-shift $\overrightarrow{s_1 \cdots s_n}$ replaces the ordered pair $s_1 s_2$ by the ordered pair $s_n s_1$. For this reason, shorthand universal cycles provide a slight advantage, on average, to prefix-shift Gray codes with respect to the total number of ordered and unordered pairs that change.

Shorthand isomorphism can be applied to any fixed-content language, including multiset permutations. Furthermore, the same observations involving prefix-shifts still hold. Despite these advantages, the idea of shorthand isomorphism is somewhat new to the academic community. Under a different name, Jackson [39] proved that

shorthand universal cycles exist for the permutations of $\{1, 2, \ldots, n\}$. (In particular, he proved that a universal cycle for the *k-permutations* of $\{1, 2, \ldots, n\}$ exist for all n and $1 \leq k \leq n - 1$, and the $k = n - 1$ case is equivalent to a shorthand universal cycle for permutations.) However, like de Bruijn's pioneering work, the proof relies on graph theoretic arguments and does not provide a reasonable construction method for large values of n. Knuth asked for an efficient construction within the *Generating all Permutations* fascicle [46] of the new volume of *The Art of Computer Programming*. An answer to this question was provided Ruskey-Williams [71] and is due to appear in the final printing of the volume. An alternative answer was known to the bell-ringing community and is discussed in Section 4.2.1.

1.2.4 Efficient Algorithms

> Patiently, inexorably, the computer had been rearranging letters in all their possible combinations, exhausting each class before going on to the next. As the sheets had emerged from the electromagnetic typewriters, the monks had carefully cut them up and pasted them into enormous books.
>
> \- in *The Nine Billion Names of God*

Section 1.1 used *The Nine Billion Names of God* to discuss the advantages of minimal-change orders over lexicographic order. In particular, minimal-change orders were of interest to George and Chuck since they could reduce the amount of time and effort expended by the Mark V printer. On the other hand, the engineers may have tried to convince the Lama that no amount of printing was necessary. To justify this possibility, suppose the Lama's belief system did not explicitly state that the possible names of God had to be written, but instead could be spoken or thought. If this were the case, then the engineers may have suggested that the Mark V simply generate each possible name within its internal memory. In this scenario, the bottleneck in completing the project would no longer be printing each name once. Instead the bottleneck would be representing each name once within the contents of the Mark V's memory.

In fact, the aforementioned scenario arises quite frequently in modern-day computer science. More specifically, it is often desirable to create each combinatorial object once in computer memory and to bypass storing or printing each object. Knuth makes note of this fact on the first page of his *Generating all Combinations* fascicle [45] when stating that the goal "is to study methods for running through all possibilities". In particular, this situation arises when solving optimization problems by brute force. To measure the efficiency of these types of algorithms it is necessary

to introduce several technical terms. When "running through all possibilities" the possibilities are best generated *in-place* using a *single shared object*. This means that a single combinatorial object is stored in the computer memory, and then it is repeatedly modified to create every possibility. The speed of a combinatorial generation algorithm can then be measured in terms of how quickly it can make each successive modification. In particular, the time required for each modification can be lower than the size of the possibility. This is especially true if the combinatorial objects are generated in a minimal-change order. For example, the reflected Gray code for n-tuples and the Johnson-Trotter-Steinhaus order for the permutations of $\{1, 2, \ldots, n\}$ can be generated by algorithms that require a constant amount of time to create each successive possibility, irrespective of the value of n [46]. More generally, the term *loopless* is due to Ehrlich [18] and refers to an algorithm that creates each successive possibility in worst-case O(1)-time, where the hidden constant does not depend on the size of the possibilities being created. A number of the efficient algorithms discussed in Section 1.2.2 are loopless algorithms including [4], [94], [89], [92], [52], [7], [53], and [51].

In practice, many of the fastest algorithms are not loopless, but run in *constant amortized time (CAT)*. This means that individual modifications may take more than O(1)-time, but the modifications take a constant amount of time on average. Lexicographic orders are frequently used to create CAT algorithms, although clever optimization and analysis is often required to prove the result. See [62] for the best resource on CAT algorithms using lexicographic order. CAT algorithms mentioned in Section 1.2.2 include [48] and [75]. Also see [66] for analysis on the construction of the lexicographically smallest de Bruijn cycle.

In terms of memory consumption, an in-place combinatorial generation algorithm can be measured by how much *additional memory* it uses. Additional memory does not include the storage used for the single shared object, and in some cases may be lower than this quantity. In particular, algorithms with this property may use a *constant number of additional variables*, where each additional variable is a single integer or pointer requiring $O(\log n)$ bits. In the case of optimization problems, additional memory also does not include memory necessary for specifying the associated values. (The table in Figure 1.6 provides an example of this uncounted expense.)

Before concluding this discussion on efficient algorithms, it is important to recall that many problems associate a value with each instance of a combinatorial object. In these cases, the bottleneck of solving the problem may involve computing the value of each object as opposed to generating each object. On the other hand, in some situations it is possible to update the associated value of successive objects in

the same way that the single shared object is modified. For example, recall that Table 1.1 summarizes the number of ordered and unordered pairs that are changed by applying various operations. In particular, at most two pairs are changed when applying the prefix-shift operation. Therefore, if a prefix-shift Gray code is used in an application whose value depends solely on these pairs, then it will be possible to update the associated value of each successive object in worst-case O(1)-time. In other words, the evaluation loopless. For optimization problems, simultaneous worst-case O(1)-time generation and worst-case O(1)-time evaluation is the ultimate goal with respect to combinatorial generation.

1.2.5 Stacker-Crane Problem

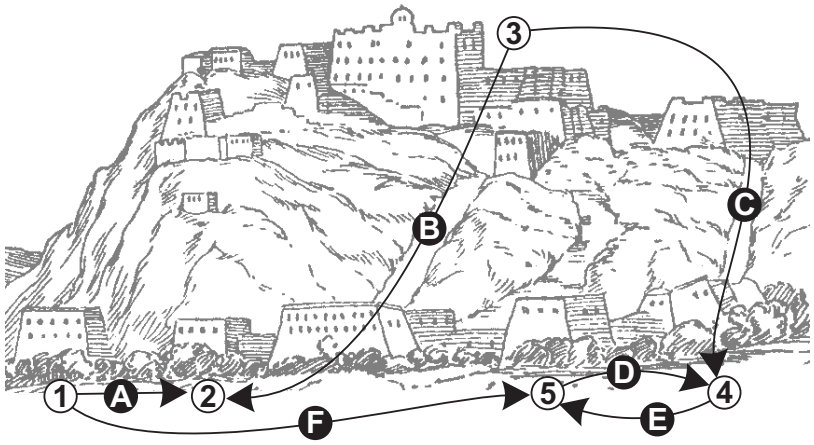
> The High Lama and his assistants would be sitting in their silk robes, inspecting the sheets as the junior monks carried them away from the typewriters and pasted them into the great volumes.
>
> \- in *The Nine Billion Names of God*

For three centuries, the inhabitants of the Tibetan lamasery had been filling massive volumes with the potential names of God. To ensure that none of the names were overlooked, the Lamas must have had rigorous checks and balances built into their daily routine. Once a volume was completed, it would have to be inspected by several other Lamas before being deposited into the library. Empty books would then make their way to the Lamas who were ready to start writing anew, and from time-to-time it would have been necessary to remove books from the library to do additional "bookkeeping" on the progress of the overall list.

Great care also would have been taken to ensure that no book was lost or misplaced during its transportation between the myriad of rooms and buildings in the lamasery. For this reason, the High Lama of a previous generation may have decreed that a single *courier* be in charge of all book deliveries on a given day. To avoid confusion (and limit back strain) the courier would be limited to carrying at most one giant volume at a time. Furthermore, the High Lama may have forbidden the courier from leaving books at an intermediate location. Finally, to provide a consistent environment free of disruptions, the courier may have been instructed to pick up and deliver all of the books while the remaining Lamas were joined in morning prayer. For this reason, the courier's goal is to minimize the amount of elapsed time between the beginning of his first pick up and the end of his last delivery.

Such a system may have been in place by 1661 when Austrian explorer and professor of mathematics Johann Grueber traveled overland from Peking to Lhasa. Grue-

ber's sketch[5] of the Potala palace is reproduced in Figure 1.6, and could have resembled the lamasery due to Clarke's description of its adequate balance at the Asiatic Bank. Figure 1.6 also formulates a sample problem for the courier at the lamasery involving six books and five rooms.



	①	②	③	④	⑤
①	0	1	8	6	5
②	1	0	7	5	5
③	3	2	0	3	4
④	6	5	8	0	1
⑤	5	5	9	1	0

Figure 1.6: An instance of the courier's stacker-crane problem at the lamasery. There are five rooms, ①-⑤ and six books, Ⓐ-Ⓕ. The books need to be delivered between the rooms along their specified route. The table provides minimum room-to-room distances measured in walking minutes.

Now let us consider the possible solutions to the courier's problem outlined in Figure 1.6.[6] Due to the aforementioned constraints and goals, the courier must repeatedly pick up a book, deliver the book to its destination along the quickest route, and then move to the room where the next book is located. Therefore, the courier's entire route can be modeled by the order of the books that he delivers. More formally, his *delivery order* in this case is simply a permutation of

$$\{Ⓐ, Ⓑ, Ⓒ, Ⓓ, Ⓔ, Ⓕ\}. \tag{1.10}$$

Now consider the amount of time required for the courier to complete the following delivery order

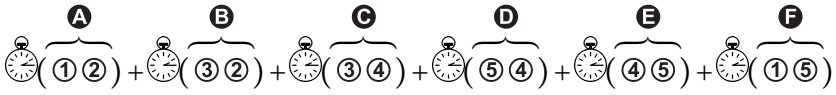
$$ⒶⒷⒸⒹⒺⒻ. \tag{1.11}$$

The total duration can be divided into two partial durations depending on whether the courier is carrying a book or not. More precisely, the first duration includes the times required to deliver each book once it has been picked up. Since each of the

[5]The complete sketch appears in Athanasius Kircher's *China Illustrata* from 1667 [43].

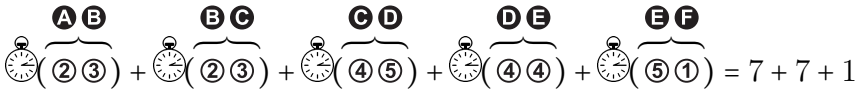
[6]The mountainous terrain explains the lack of symmetry in room-to-room distances in Figure 1.6.

books is delivered once, then this first duration is



(1.12)

The second duration includes the intermediate times consumed between deliveries, when the courier is walking from one room to the next without carrying a book. For example, given the delivery order in (1.11) then this second duration is



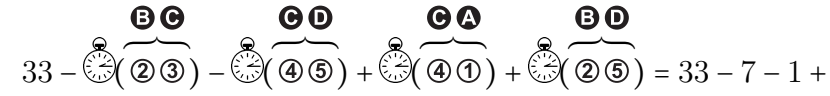
(1.13)

Therefore, the courier's total duration for the delivery order in (1.11) is 13 + 20 = 33, by (1.12) and (1.13).

Given this description, it is clear that the first duration does not depend on the delivery order, and the second duration depends solely on the ordered pairs in the delivery order. For this reason, Table 1.1 implies that the duration of a delivery order can be updated in O(1)-time whenever the delivery order is modified by a shift or a transposition. In particular, only two additions and two subtractions from the table in Figure 1.6 need to be applied to update the duration of a delivery order once if it is modified by a prefix-shift. For example, given the following delivery order

$$\overleftarrow{\text{ⒶⒷⒸⒹⒺⒻ}} = \text{ⒸⒶⒷⒹⒺⒻ},$$

its duration can be computed from (1.12) and (1.13) as follows



(1.14)

Now that the basic elements of the courier's problem are understood, it is important to point out that there is a simplifying assumption made in the sample problem given in Figure 1.6. In particular, there is at most one book that needs to be transported between any ordered pair of rooms. For example, book Ⓕ is the single book that needs to be transported from room ① to room ⑤. Now suppose that the courier's problem is modified to include the transport of a second book from room ① to room ⑤. Since the restrictions of the problem forbid the courier from carrying two books at once, the solutions can still be specified by the order of the books that are delivered.

In particular, the solutions to this problem could be modeled by the permutations of

$$\{\text{Ⓐ}, \text{Ⓑ}, \text{Ⓒ}, \text{Ⓓ}, \text{Ⓔ}, \text{Ⓕ}, \text{Ⓖ}\}$$

where Ⓖ represents the additional book. However, this model has the disadvantage that it contains twice as many possibilities as distinct solutions. In particular, transposing Ⓕ and Ⓖ in any permutation will not actually change the courier's route. The courier can avoid this redundancy by modeling the solutions on the permutations of the following multiset

$$\{\text{Ⓐ}, \text{Ⓑ}, \text{Ⓒ}, \text{Ⓓ}, \text{Ⓔ}, \text{Ⓕ}, \text{Ⓕ}\}.$$

In general, the courier is faced with an optimization problem on multiset permutations, where the associated value depends solely on the ordered pairs. The courier's problem is known in theoretical computer science and combinatorial optimization as a *stacker-crane problem* (SCP). In general, the stacker-crane problem arises when objects need to be quickly transported directly between locations by a single vehicle that can carry at most one object at a time. SCP is extremely difficult to solve in practice, and Frederickson-Hecht-Kim [23] have shown that the associated decision problem is NP-complete. (For an introduction to NP-completeness see Garey-Johnson [24].) In particular, SCP generalizes one of the most important NP-complete problems known as the *traveling salesman problem* (TSP). Informally, TSP is the special case of SCP where the objects need to be picked up but not delivered. For thorough coverage of TSP and its variations see Applegate-Bixby-Chvátal-Cook [1] and Gutin-Punnen [30].

The results of this thesis are applicable to SCP for several reasons. Chapter 3 provides the first prefix-shift Gray code for multiset permutations. (The operation that generates this Gray code is similar to operation (ii), and appears in a more formal setting in (4.3a) on page 149.) Prefix-shifts provide the best results with respect to changing ordered pairs in Table 1.1. Furthermore, this advantage is even more pronounced given the fact that adjacent-transposition Gray codes do not always exist for multiset permutations, as illustrated by Figure 1.5. Chapter 4 provides a loopless algorithm for generating this prefix-shift Gray code. In particular, Algorithm 7 is the first loopless algorithm for generating multiset permutations that uses a constant number of additional variables. (The algorithm has the surprising property that it does not store any information regarding the contents of the multiset.) Besides being more efficient, Algorithm 7 is also considerably simpler than any previously known algorithms for generating shift Gray codes. Collectively, these results provide a simple brute force solution to SCP featuring worst-case O(1)-time generation, worst-

case O(1)-time evaluation, and the use of O(1) additional variables.[7]

Interestingly, there is also a sense in which the courier could generate and evaluate his possible routes by a loopless manual algorithm. Recall the bottom-right illustration in Figure 1.1. By using marbles of various shades, a Lama could easily implement Algorithm 7 using one hand. (In particular, the Lama needs only to keep his eye on the leftmost pair of lighter-darker marbles in order to apply successive applications of (4.3a).) With his free hand, the same Lama could use an abacus to update the duration of each successive possibility using the type of calculations shown in (1.14). Alternatively, given the 300 year history of the project at the lamasery, it is also possible that the Lamas would have computed shorthand universal cycles for the permutations of all small multisets. By using the resulting prefix-shift Gray codes, a Lama could reduce his total amount of abacus work. (In particular, recall the observation from Section 1.2.3 that each prefix-shift of the form $\overrightarrow{s_1 \cdots s_n}$ changes only one ordered pair.)

Finally, the results of this thesis are also robust enough to handle changes in the underlying problem. Suppose the courier was falling behind on his name-writing assignments. For this reason, he wished to minimize not only the delivery time, but also the amount of time he spent walking from his room to pick up the first book, and the amount of time he spent returning to his room after completing the last delivery. With some additional provisions, the courier could model his route based on multiset necklaces instead of multiset permutations. Unfortunately, prior to this thesis there were no known Gray code for multiset necklaces. However, the same general theory that produced operations (i) and (ii) also produces similar rules for generating a shift Gray code for these objects.

Before concluding the discussion of this problem, it is mentioned that [23] provides a $\frac{9}{5}$-approximation algorithm for SCP and proves that SCP is NP-complete by reducing TSP to SCP. In their version of the problem there is a specified *initial vertex* $v_0 \in \mathbb{V}$. Their reduction also proves the NP-completeness of SCP without an initial vertex by replacing TSP by the minimum weight Hamiltonian path problem. Stacker-crane problems also remain NP-hard when the underlying mixed-graph is a tree, although stronger approximation algorithms exist by Frederickson-Guan [22] that almost always provide exact solutions as shown by Coja-Oghlan-Krumke-Nierhoff [11]. Generalizations including k cranes have also been considered [23] and also have real-world applications (see Burkard-Fruhwirth-Rote [5]). Stacker-crane problems can also be described as *single vehicle pickup and delivery problems* (or *single vehicle dial-a-ride*

[7] This brute force solution could be described as using *cute force*.

problems) that forbid multiple pickups between deliveries.

1.3 New Results

The purpose of the previous two sections was to survey historical and contemporary results in combinatorial generation, and to motivate the theoretical and practical value of studying shift Gray codes. The remainder of this thesis focuses on the abstract goals of constructing shift Gray codes and shorthand universal cycles, and for developing efficient algorithms for generating them. This section outlines the basic results.

Chapter 2 builds a framework that generalizes the fixed-content languages mentioned in this section together with fixed-content languages that represent balanced parentheses, k-ary Dyck words, unit interval graphs, Schroöder and Motzkin paths, linear-extensions of posets, and additional variations of necklaces including Lyndon words (see Table 2.1 on page 38). The term *bubble* was chosen since the definition is related to bubble sort (see Knuth [44]). In particular, any string in a bubble language can be sorted into non-increasing order by a series of left-shifts that "bubble" larger symbols to the left.

Chapter 3 takes advantage of this new framework by proving that every bubble language has a simple left-shift Gray code. All of these Gray codes can be expressed using the cool-lex variation of lexicographic order. *Co-lexicographic order* is lexicographic order applied to strings read from right-to-left instead of left-to-right. This name is often abbreviated to *co-lex*, and this explains the *cool-lex* moniker. Table 1.2 compares the recursive structure of co-lex and cool-lex for the permutations of $\{1, 2, 3, 4\}$. Co-lex order is typically defined by the value of the rightmost symbol. This last symbol of each string is underlined and rewritten on the right side of the first column. Notice that each suffix in question appears contiguously and in increasing order: 1, 2, 3, 4. Furthermore, the same pattern holds recursively.

The second column again contains co-lex order on the left side, however in this case a different suffix is underlined and copied on the right side of the column. To describe the suffixes, notice that the first string, 4321, has its symbols in non-increasing order. Furthermore, it does not have an underlined suffix. This is because the underlined suffixes are the shortest suffixes that are not also suffixes of this special string 4321. These suffixes are known as *scuts*, which is an English word meaning "short stubby tail" (see `dictionary.com` [34]). For example, the scut of 4231 is 31 because it is not a suffix of 4321 and the next shortest suffix, 1, is a suffix of 4321. Notice that

Co-lex	last	Co-lex	scut	Cool-lex	scut
4321	1	4321		1432	2
3421	1	3421	421	4132	2
4231	1	4231	31	3412	2
2431	1	2431	31	1342	2
3241	1	3241	41	3142	2
2341	1	2341	41	4312	2
4312	2	4312	2	2431	31
3412	2	3412	2	4231	31
4132	2	4132	2	1423	3
1432	2	1432	2	4123	3
3142	2	3142	2	2413	3
1342	2	1342	2	1243	3
4213	3	4213	3	2143	3
2413	3	2413	3	4213	3
4123	3	4123	3	3421	421
1423	3	1423	3	2341	41
2143	3	2143	3	3241	41
1243	3	1243	3	1324	4
3214	4	3214	4	3124	4
2314	4	2314	4	2314	4
3124	4	3124	4	1234	4
1324	4	1324	4	2134	4
2134	4	2134	4	3214	4
1234	4	1234	4	4321	

Table 1.2: Two recursive views of co-lex order, together with the recursive view of cool-lex order for permutations of $\{1, 2, 3, 4\}$.

each scut appears contiguously and in the following order: 421, 31, 41, 2, 3, 4. Put another way, the scuts are ordered by decreasing length and then by increasing first symbol. Furthermore, the same pattern holds recursively.

With this second recursive interpretation of co-lex in mind, cool-lex order can be summarized succinctly as a reordering of the scuts and the string in non-increasing order. In particular, the non-increasing string 4321 appears last (instead of first) and the scuts are ordered by increasing first symbol and then by decreasing length (instead of by decreasing length and then by increasing first symbol). For example, the left side of the third column contains the cool-lex order for the permutations of $\{1, 2, 3, 4\}$. The scuts are again underlined and copied on the right side of the column. This time the scuts appear contiguously and in the following order: 2, 31, 3, 4, 41, 421. Furthermore, the same pattern holds recursively. Figure 1.7 illustrates co-lex and cool-lex order for the permutations of $\{1, 2, 3, 4, 5, 6, 7\}$. When comparing pages 6

and 36, notice that Figure 1.2 emphasizes the reordering of the non-increasing string, while Figure 1.7 emphasizes the reordering of the scuts.

Cool-lex order for bubble languages has at least two important applications. The first application involves efficient algorithms. In general, any bubble language can be generated in cool-lex order by a simple algorithm found on page 128. Furthermore, this generic algorithm can be highly optimized for specific bubble languages as seen in the first half of Chapter 4. The second application discussed in this thesis involves shorthand universal cycles. Interestingly, reverse cool-lex can be used to create shorthand universal cycles for multiset permutations in much the same way that de Bruijn cycles can be created from lexicographic order. In other words, cool-lex order provides a multiset permutation analogue to lexicographic order with respect to the FKM algorithm. For an illustration, consider again the right-shift Gray code for (3,3)-combinations given in (1.1), except cross off the strings that are not lexicographically largest in their rotation set

$$\not{1}1\not{0}\not{0}\not{0}1, 1\not{0}\not{0}\not{0}1\not{1}, \not{0}\not{0}\not{0}1\not{1}\not{1}, \not{0}\not{0}\not{1}\not{0}\not{1}\not{1}, \not{0}\not{1}\not{0}\not{0}\not{1}\not{1}, 1\not{0}\not{0}\not{1}\not{0}\not{1}, \not{0}\not{0}\not{1}\not{1}\not{0}\not{1}, \not{0}\not{1}\not{0}\not{1}\not{0}\not{1}, 1\not{0}\not{1}\not{0}\not{0}\not{1}, \not{0}\not{1}\not{1}\not{0}\not{0}\not{1},$$
$$\underline{110010}, \not{1}\not{0}\not{0}\not{1}\not{1}\not{0}, \not{0}\not{0}\not{1}\not{1}\not{1}\not{0}, \not{0}\not{1}\not{0}\not{1}\not{1}\not{0}, \underline{10}1010, \not{0}\not{1}\not{1}\not{0}\not{1}\not{0}, \underline{110100}, \not{1}\not{0}\not{1}\not{1}\not{0}\not{0}, \not{0}\not{1}\not{1}\not{1}\not{0}\not{0}, \underline{111000}.$$

For example, 101100 is crossed off because 110010 is a lexicographically larger rotation. Appending the underlined aperiodic prefix of each remaining string yields

$$11001010110100111000,$$

which is the shorthand universal cycle previously seen in (1.3). These results are investigated in the second half of Chapter 4 and rely on an interesting shorthand rotation property that is hidden in the cool-lex order of every bubble language.

1.3.1 Summary

This thesis introduces the concept of a bubble language in Chapter 2 and a new variation of lexicographic order called cool-lex order in Chapter 3. Whenever bubble languages are expressed in cool-lex order the result is a left-shift Gray code. Furthermore, these left-shift Gray codes can be used to create the efficient generation algorithms and shorthand universal cycles found in Chapter 4. Specific results include:

- the first prefix-shift Gray code for multiset permutations,

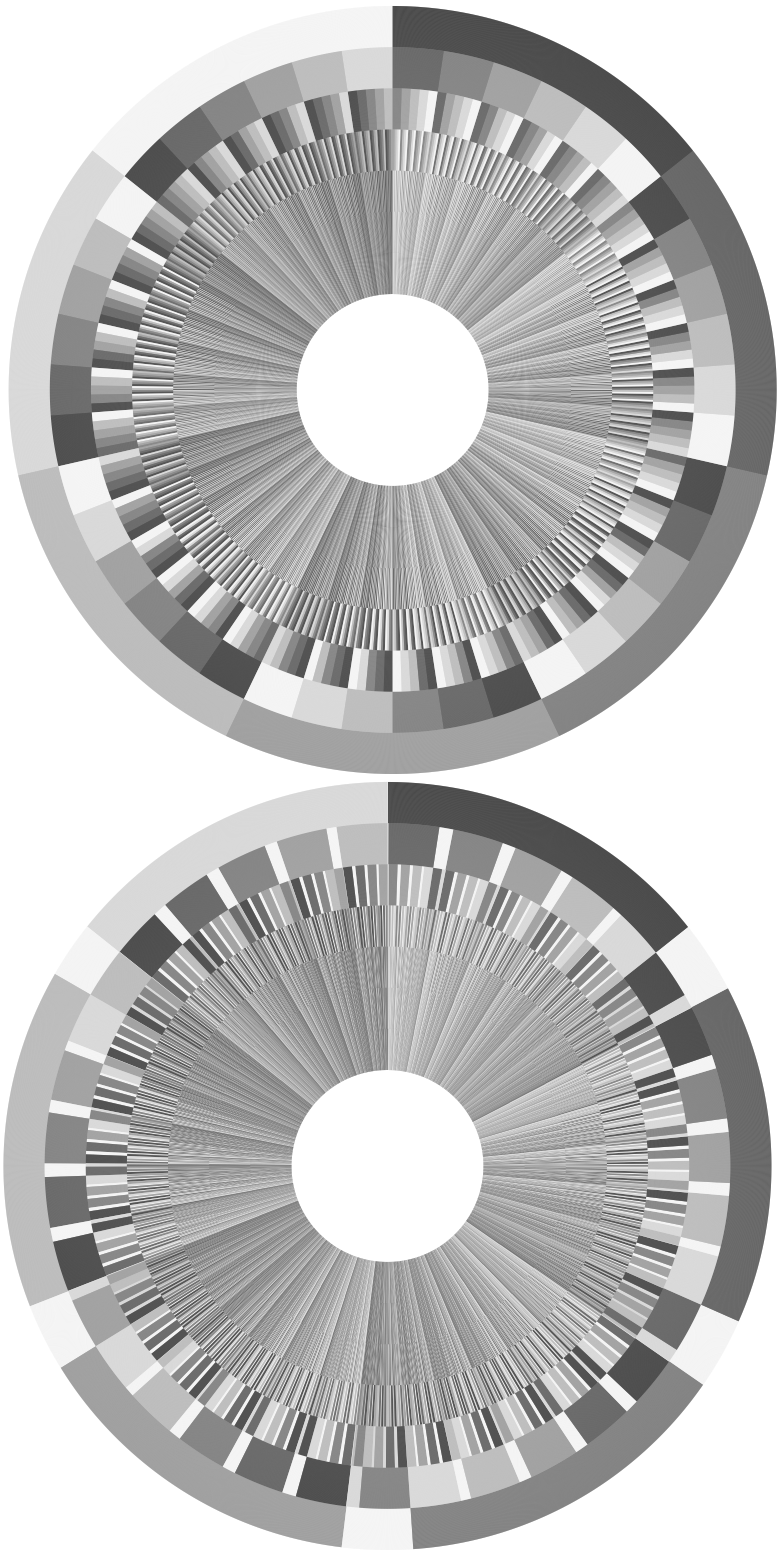


Figure 1.7: An artistic representation of co-lexicographic (above) and cool-lex (below) order for permutations of $\{1, 2, 3, 4, 5, 6, 7\}$. The lightest and darkest regions represent 1 and 7 respectively. Individual strings are read along a line segment originating from the center, and the first and last strings are at either side of 12 o'clock. Cool-lex proceeds leftwards (counterclockwise) and involves left-shifts, while reverse cool-lex proceeds rightwards (clockwise) and involves right-shifts. Co-lexicographic order proceeds counterclockwise, while reverse co-lexicographic order proceeds clockwise.

- the first shift Gray code for linear-extensions of B-posets, ordered trees with fixed branching sequence, restricted Schröder and Motzkin paths,
- the first minimal-change order for multiset necklaces, pre-necklaces, and Lyndon words,
- the first loopless algorithm using a constant number of additional variables for generating multiset permutations,
- the most efficient array-based algorithm for balanced parentheses in practice (see Arndt [2]),
- the first universal cycle for the middle levels (fixed-density de Bruijn cycle), and
- the first shorthand universal cycles of multiset permutations.

The interested reader will also find additional results and open problems in Chapter 5.

Chapter 2

Bubble Languages

"Mathematics is the art of giving the same name to different things."
- Jules Henri Poincaré

This chapter introduces the concept of a bubble language. Bubble languages provide a new abstraction that encapsulates a number of previously studied combinatorial objects. In particular, this chapter proves that bubble languages can be used to represent all of the combinatorial objects listed in Table 2.1. This general framework allows interesting results for all of these objects to be proven simultaneously in Chapter 3. Prior to bubble languages, the objects listed in Table 2.1 were known to be interrelated in several isolated ways. For example, binary trees, Catalan paths, and balanced parentheses can all be represented by the same language, while a subset of this language can be used to represent connected unit interval graphs. Similarly, linear-extensions of B-posets include k-ary Dyck words and balanced parentheses as special cases.

Permutations	*Trees*	*Grid Paths*	*Linear-Extensions*	*Necklaces*
multiset permutations	ordered trees with branching sequence	restricted Motzkin and Schröder paths	linear-extensions of B-posets	multiset necklaces
permutations	k-ary trees	Dyck paths	k-ary Dyck words	multiset pre-necklaces
(s,t)-combinations	binary trees	Catalan paths	balanced parentheses	multiset Lyndon words
connected unit interval graphs				
Graphs				

Table 2.1: Bubble languages include representations of many combinatorial objects.

It is important to note that a small concession is made in the representation of several combinatorial objects found in Table 2.1. This concession is rooted in differing historical conventions used for representing differing combinatorial objects. For example, 0004 is the standard representation for the only necklace over $\{0, 0, 0, 4\}$. On the other hand, 4000 is the standard representation for the only ordered tree with a root and four leaves. This thesis chooses to follow conventions that are more closely related to ordered trees instead of necklaces. When discussing individual combinatorial objects it is a relatively simple matter to convert between these two conventions[1]. In particular, concessions are made on the representations of linear-extensions, and necklace variations including Lyndon words and pre-necklaces.

Preliminary concepts and conventions necessary to define bubble languages are given in Section 2.1. Bubble languages are then formally defined in Section 2.2, and this section also includes a simplified definition for fixed-density bubble languages. Section 2.3 discusses the majority of entries found in Table 2.1. Several important properties of bubble languages are provided in Section 2.4 including closure under various operations, shift identities, and a structural characterization.

2.1 Preliminaries

This section introduces concepts and conventions that are used throughout the thesis. These concepts include fixed-content languages and the special case of fixed-density languages, the quotient operation on languages, non-increasing strings, the frozen prefix of a string within a language, and shift operations including bubble left-shifts. The reader is advised that Appendix A on page 186 provides a quick reference and index for all notation and conventions used in this thesis. This appendix also include a basic interrelated example.

2.1.1 Fixed-Content Languages

This section defines a fixed-content language as a set of strings containing the same multiset of symbols. Towards this definition it is necessary to introduce sets and multisets, strings, and languages. Although each concept is simple, collectively they can lead to cumbersome and repetitive notation. For this reason, a number of default

[1]To follow the historical convention for necklaces, the reader would need to change comparisons involving individual symbols. For example, $s_h \leq s_{h+1}$ would replace $s_h \geq s_{h+1}$ in Definition 2.1.4 to convert *non-increasing prefixes* into *non-decreasing prefixes*. As a second example, $s_k > s_{k+2}$ would replace $s_k < s_{k+2}$ in Definition 3.1.3.

conventions will be followed throughout this thesis, as summarized at the end of this section. Typography is also used to aid in the presentation of this thesis, with symbols, sets and multisets, strings, and languages respectively given in regular, sans-serif, bold, and blackboard typeface.

A *set* is a collection of *distinct symbols*. By convention, the set Σ contains m distinct symbols and

$$\Sigma = \{d_1, d_2, \ldots, d_m\}.$$

The symbols in a set are assumed to be totally ordered by $<$. By convention, the symbols in Σ are ordered as follows

$$d_1 < d_2 < \cdots < d_m.$$

The examples in this thesis involve sets whose symbols are single digit non-negative integers, and the order $<$ is given by numerical order. For example, if $\Sigma = \{0, 1, 3, 8\}$ then $m = 4$ and

$$d_1, d_2, \ldots, d_m = 0, 1, 3, 8.$$

(Due to the nature of the results in this thesis, the simplifying assumption $d_i = i$ cannot be used.) By similar convention, the set Σ' contains the m' distinct symbols satisfying $d'_1 < d'_2 < \cdots < d'_{m'}$.

A *multiset* is a set together with a function that maps each symbol in the set to a positive integer. The value of this function denotes the number of copies or *multiplicity* of each symbol within the multiset. In other words, a multiset generalizes a set by allowing its symbols to be repeated any positive number of times. The *underlying set* of a multiset $\mathbb{M}$ is denoted by $\mathsf{set}(\mathbb{M})$. By convention, $\mathsf{set}(\mathbb{M}) = \Sigma$. Furthermore, the multiplicity of d_i within multiset $\mathbb{M}$ is n_i. It is also assumed that $\mathbb{M}$ contains n symbols in total. Thus,

$$n = n_1 + n_2 + \cdots + n_m.$$

In particular, the n symbols in $\mathbb{M}$ are denoted by e_i where

$$e_1 \leq e_2 \leq \cdots \leq e_n.$$

Therefore, by convention the multiset $\mathbb{M}$ can be expressed in the following ways

$$\mathbb{M} = \{e_1, e_2, \ldots, e_n\} = \{\underbrace{d_1, \ldots, d_1}_{n_1 \text{ copies}}, \underbrace{d_2, \ldots, d_2}_{n_2 \text{ copies}}, \ldots, \underbrace{d_m, \ldots, d_m}_{n_m \text{ copies}}\}.$$

The *sum* of $\mathbb{M}$ is denoted by $\Sigma(\mathbb{M})$ and is the sum of its symbols. To illustrate these conventions, suppose $\mathbb{M} = \{0, 1, 1, 1, 3, 8, 8\}$. Then, $n = 7$, $m = 4$, $\Sigma = \{0, 1, 3, 8\}$ and $\Sigma(\mathbb{M}) = 22$. Furthermore,

$$e_1, e_2, \ldots, e_n = 0, 1, 1, 1, 3, 8, 8$$
$$d_1, d_2, \ldots, d_m = 0, 1, 3, 8.$$

Since $\mathbb{M}$ contains one copy of $d_1 = 0$, three copies of $d_2 = 1$, one copy of $d_3 = 3$, and two copies of $d_4 = 8$ the multiplicities are as follows

$$n_1, n_2, \ldots, n_m = 1, 3, 1, 2.$$

Besides the individual multiplicities given above, it is also useful to be able to refer to *cumulative multiplicities* within $\mathbb{M}$. Let $\underline{n}_i$ represent the number of symbols less than or equal to d_i in $\mathbb{M}$. That is,

$$\underline{n}_i = n_1 + n_2 + \cdots + n_i.$$

Similarly, let $\overline{n}_i$ represent the number of symbols greater than or equal to d_i in $\mathbb{M}$. That is,

$$\overline{n}_i = n_i + n_{i+1} + \cdots + n_m.$$

For example, given $\mathbb{M} = \{0, 1, 1, 1, 3, 8, 8\}$, then $\underline{n}_2 = 4$ since the symbols $0, 1, 1, 1$ are less than or equal to $d_2 = 1$. Similarly, $\overline{n}_3 = 3$ since the symbols $3, 8, 8$ are greater than or equal to $d_2 = 3$. By similar convention, n', m', e'_i, d'_i, n'_i, $\underline{n}'_i$, and $\overline{n}'_i$ refer to the corresponding values in multiset $\mathbb{M}'$ instead of $\mathbb{M}$.

Multiset $\mathbb{M}'$ is a *subset* of multiset $\mathbb{M}$, and this is written $\mathbb{M}' \subseteq \mathbb{M}$ if each distinct symbol appears at least as many times in $\mathbb{M}$ as in $\mathbb{M}'$. That is, $\mathbb{M}' \subseteq \mathbb{M}$ if and only if $\Sigma' \subseteq \Sigma$ and there do not exist i and i' such that $d'_{i'} = d_i$ and $n'_{i'} > n_i$. The multisets $\mathbb{M}$ and $\mathbb{M}'$ are equal if $\mathbb{M}' \subseteq \mathbb{M}$ and $\mathbb{M} \subseteq \mathbb{M}'$, and this is denoted by $\mathbb{M} = \mathbb{M}'$. If $\mathbb{M}' \subseteq \mathbb{M}$ then $\mathbb{M} \backslash \mathbb{M}'$ represents the multiset that remains when each symbol in $\mathbb{M}'$ is removed

from $\mathbb{M}$. For example, if $\mathbb{M}' = \{0, 1, 8, 8\}$ and $\mathbb{M} = \{0, 1, 1, 1, 3, 8, 8\}$ then $\mathbb{M}' \subseteq \mathbb{M}$ and

$$\mathbb{M} \backslash \mathbb{M}' = \{0, 1, 1, 1, 3, 8, 8\} \backslash \{0, 1, 8, 8\} = \{1, 1, 3\}.$$

A *string* is a finite sequence of symbols. The *sum* of a string $\boldsymbol{s}$ is denoted by $\Sigma(\boldsymbol{s})$ and is the sum of its symbols. The *length* of a string $\boldsymbol{s}$ is the number of symbols in the sequence and is denoted $|\boldsymbol{s}|$. The *empty string* is denoted ϵ and is the unique string satisfying $|\epsilon| = 0$. The *content* of $\boldsymbol{s}$ is denoted $\mathsf{content}(\boldsymbol{s})$ and is the multiset of symbols it contains. The symbols within a string are separated by $\cdot$. By convention, the individual symbols within a string can be referred to using subscripts beginning with 1. That is, if $\boldsymbol{s}$ is a string, then its ith symbol is denoted by s_i. To illustrate these definitions, if $\boldsymbol{s} = 7 \cdot 2 \cdot 6 \cdot 9 \cdot 9 \cdot 5 \cdot 4$ then

$$\Sigma(\boldsymbol{s}) = 38 \qquad |\boldsymbol{s}| = 7 \qquad \mathsf{content}(\boldsymbol{s}) = \{2, 4, 5, 6, 7, 9, 9\} \qquad s_4 = s_5 = 9.$$

The *concatenation* of strings $\boldsymbol{s}$ and $\boldsymbol{t}$ is denoted by $\boldsymbol{s} \cdot \boldsymbol{t}$ and is the sequence of symbols in $\boldsymbol{s}$ followed by the sequence of symbols in $\boldsymbol{t}$. When a string or symbol is concatenated to the end of a string then it is also known as *suffixing*. Repeated concatenations of the same string or symbol are represented by exponentiation. For example, if $\boldsymbol{a} = 1 \cdot 3$ and $\boldsymbol{b} = 3 \cdot 1$ and $\boldsymbol{c} = \epsilon$ then

$$\boldsymbol{a} \cdot \boldsymbol{b} \cdot \boldsymbol{c} = 1 \cdot 3 \cdot 3 \cdot 1 \qquad \boldsymbol{a}^2 \cdot 1^2 \cdot 3^{2\cdot} = 1 \cdot 3 \cdot 1 \cdot 3 \cdot 1 \cdot 1 \cdot 3 \cdot 3.$$

Throughout this thesis, $\cdot$ will only be used to highlight certain concatenations. That is, $\boldsymbol{a} \cdot \boldsymbol{b} = \boldsymbol{ab}$ and $s_1 \cdot s_2 \cdot s_3 = s_1 s_2 s_3$. This convention does not cause ambiguity within the examples since each symbol is assumed to be a single digit non-negative integer. Thus, 13 will always be interpreted as $1 \cdot 3$ within the examples.

If $\boldsymbol{s} = \boldsymbol{abc}$ then $\boldsymbol{a}$ is a *prefix* of $\boldsymbol{s}$, $\boldsymbol{b}$ is a *substring* of $\boldsymbol{s}$, and $\boldsymbol{c}$ is a *suffix* of $\boldsymbol{s}$. Furthermore, $\boldsymbol{a}$ is a *strict prefix* if $\boldsymbol{bc} \neq \epsilon$, $\boldsymbol{b}$ is a *strict substring* if $\boldsymbol{ac} \neq \epsilon$, $\boldsymbol{c}$ is a *strict suffix* if $\boldsymbol{ab} \neq \epsilon$,

A *language* is a set of strings. The *size* of a language $\mathbf{L}$ is denoted by $|\mathbf{L}|$ and equals the number of strings it contains. The *empty language* is denoted by $\varnothing$ and is the unique language satisfying $|\varnothing| = 0$. A language $\mathbf{L}$ is *trivial* if $|\mathbf{L}| = 1$, and otherwise is *non-trivial*. Since languages are sets, $\boldsymbol{s} \in \mathbf{L}$ means that the string $\boldsymbol{s}$ is in the language $\mathbf{L}$. Now the main definition in this section can be given.

Definition 2.1.1 (Fixed-Content Language)**.** *A* fixed-content *language* $\mathbf{L}$ *is a lan-*

guage in which $\boldsymbol{s}, \boldsymbol{t} \in \mathbf{L}$ *implies*

$$\mathsf{content}(\boldsymbol{s}) = \mathsf{content}(\boldsymbol{t}).$$

In other words, a fixed-content language is a set of strings where every string contains the same multiset of symbols.

The *content* of a fixed-content language $\mathbf{L}$ is denoted $\mathsf{content}(\mathbf{L})$ and is the multiset of symbols contained in any one of its strings. That is, if $\mathbf{L}$ is a fixed-content language with $\boldsymbol{s} \in \mathbf{L}$ then

$$\mathsf{content}(\mathbf{L}) = \mathsf{content}(\boldsymbol{s}).$$

By convention, $\mathbf{L}$ is assumed to be a fixed-content language with content $\mathbb{M}$. For example, if $\mathbf{L} = \{2110, 2101, 2011, 1201, 1210\}$ then $|\mathbf{L}| = 5$, and

$$\mathsf{content}(\mathbf{L}) = \mathbb{M} = \{0, 1, 1, 2\}.$$

By similar convention, $\mathbf{L}'$ is assumed to be a fixed-content language with content $\mathbb{M}'$. To further strengthen these conventions, strings such as $\boldsymbol{r}'$, $\boldsymbol{s}'$, and $\boldsymbol{t}'$ will only be discussed in conjunction with $\mathbf{L}'$.

The results in this thesis are simplified when considering *fixed-density languages*, which are defined below.

Definition 2.1.2 (Fixed-Density Language)**.** *A* fixed-density language $\mathbf{L}$ *is a fixed-content language with*

$$\mathsf{content}(\mathbf{L}) = \{0, 0, \ldots, 0, 1, 1, \ldots, 1\}.$$

In other words, a fixed-density language is a set of strings where every string contains the same number of 0*s, the same number of* 1*s, and no other symbols.*

Before concluding this section, the default conventions are collected into the following list.

- The language $\mathbf{L}$ is a fixed-content language.
- The content of $\mathbf{L}$ is the multiset of symbols $\mathbb{M}$. That is,

$$\mathsf{content}(\mathbf{L}) = \mathbb{M}.$$

- The multiset $\mathbb{M}$ contains n total symbols. In particular,

$$\mathbb{M} = \{e_1, e_2, \ldots, e_n\}.$$

- The symbols within $\mathbb{M}$ are ordered as follows

$$e_1 \leq e_2 \leq \cdots \leq e_n.$$

- The underlying set of symbols within $\mathbb{M}$ is Σ.
- The set Σ contains m symbols. In particular,

$$\Sigma = \{d_1, d_2, \ldots, d_m\}.$$

- The symbols within Σ are ordered as follows

$$d_1 < d_2 < \cdots < d_m.$$

- Within $\mathbb{M}$, the multiplicity of d_i is n_i.
- Within $\mathbb{M}$, the cumulative multiplicity of the symbols less than or equal to d_i is $\underline{n}_i$.
- Within $\mathbb{M}$, the cumulative multiplicity of the symbols greater than or equal to d_i is $\overline{n}_i$.
- The ith symbol in string $\boldsymbol{s}$ is denoted s_i. That is, $\boldsymbol{s} = s_1 s_2 \cdots$.

Analogous conventions hold for $\mathbf{L}'$. For example, the content of $\mathbf{L}'$ is $\mathbb{M}'$, and Σ' contains m' symbols that are ordered by $d'_1 < d'_2 < \ldots < d'_m$.

Collectively, the above conventions allow certain values to be expressed succinctly. For example, d_i is a function of $\mathbb{M}$, but its value can also be inferred from a single string. For example, if $\boldsymbol{s} \in \mathbf{L}$ and $\boldsymbol{s} = 545130321$ then by convention $\mathbf{L}$ is a fixed-content language with content $\mathbb{M} = \{0, 1, 1, 2, 3, 3, 4, 5, 5\}$. Therefore, $d_3 = 2$ since 2 is the 3rd-smallest symbol in $\mathbb{M}$. Furthermore, $\overline{n}_3 = 6$ since $2, 3, 3, 4, 5, 5$ are the six symbols that are greater than or equal to 2 in $\mathbb{M}$.

2.1.2 Quotient Operation

An important operation in this thesis is the quotient operation.

Definition 2.1.3 (Quotient (/))**.** *Given a language* $\mathbf{L}$ *and a string* $\boldsymbol{z}$*, the* quotient *operation results in the following language*

$$\mathbf{L}/\boldsymbol{z} = \{\boldsymbol{p} \mid \boldsymbol{p} \cdot \boldsymbol{z} \in \mathbf{L}\}.$$

In other words, $\mathbf{L}/\boldsymbol{z}$ *is the language that results from removing the suffix* $\boldsymbol{z}$ *from every string in* $\mathbf{L}$ *that has* $\boldsymbol{z}$ *as a suffix, and from discarding every string in* $\mathbf{L}$ *that does not have* $\boldsymbol{z}$ *as a suffix.*

To illustrate this operation, suppose

$$\mathbf{L} = \{332211, 332121, 332112, 331221, 323211, 323121, 322311, 322131, 321321, 321312, 321231\}.$$

Then,

$$\mathbf{L}/2 = \{33211, 32131\} \qquad \mathbf{L}/312 = \{321\} \qquad \mathbf{L}/3 = \varnothing.$$

Every language of the form $\mathbf{L}/\boldsymbol{z}$ is known as a *quotient* of $\mathbf{L}$. Notice that $\mathbf{L}/\boldsymbol{z}$ will always be a fixed-content language since $\mathbf{L}$ is assumed to be a fixed-content language. In particular,

$$\mathsf{content}(\mathbf{L}/\boldsymbol{z}) = \mathbb{M} \backslash \mathsf{content}(\boldsymbol{z}).$$

If $\mathsf{content}(\boldsymbol{z})$ is not a subset of $\mathbb{M}$ then $\mathbf{L}/\boldsymbol{z}$ is empty. Finally,

$$\mathbf{L}/\epsilon = \mathbf{L}$$

since every string has the empty string ϵ as a suffix. In particular, this fact is used as the base case for the recursive definition of cool-lex order in Definition 3.2.2.

2.1.3 Non-Increasing Strings

An *increase* in string $\boldsymbol{s}$ is a pair of consecutive symbols $s_i \cdot s_{i+1}$ such that $s_i < s_{i+1}$. A string is *non-increasing* if it does not contain an increase. This section defines the non-increasing prefix of a string, as well as the non-increasing string that arises from a multiset of symbols. This section also describes a helpful notational convention with respect to these related concepts.

A prefix that contains no increases is known as a non-increasing prefix, and the longest such prefix of a string is known as its *non-increasing prefix*. This concept is formally defined below, and is followed by several simple examples.

Definition 2.1.4 (Non-Increasing Prefix)**.** *The non-increasing prefix of* $\boldsymbol{s}$ *is*

$$\lnot(\boldsymbol{s}) = s_1 s_2 \dots s_k$$

where k *is the maximum value such that* $s_h \geq s_{h+1}$ *for all* h *within* $1 \leq h < k$*. In other words, the non-increasing prefix is the longest prefix that does not contain an increasing pair of consecutive symbols.*

The notation in Definition 2.1.4 reflects the notion that $s_1 s_2 \cdots s_k$ is proceeds like a downward "staircase". This concept is illustrated below on several strings with content $\mathbb{M} = \{1, 2, 2, 3, 4\}$

$$\lnot(43221) = 43221 \qquad \lnot(32214) = 3221 \qquad \lnot(12234) = 1 \qquad \lnot(22341) = 22.$$

Given a multiset of symbols $\mathbb{M}$, there is a unique non-increasing string $\boldsymbol{s}$ with $\mathsf{content}(\boldsymbol{s}) = \mathbb{M}$. This string is denoted by $\lnot(\mathbb{M})$ and can be referred to as the *non-increasing string with content* $\mathbb{M}$. For example,

$$\lnot(\{0, 0, 2, 3, 3, 4, \}) = 433200 \text{ and } \lnot(\mathbb{M}) = e_n \cdot e_{n-1} \cdots e_1$$

where the second equality follows from the conventions on $\mathbb{M}$. Before proceeding, it is worth clarifying that $\lnot(\boldsymbol{s})$ denotes the non-increasing prefix of a string $\boldsymbol{s}$, while $\lnot(\mathbb{M})$ denotes the non-increasing string with content $\mathbb{M}$. To reinforce the distinction, notice that $\lnot(\boldsymbol{s}) = \lnot(\mathsf{content}(\boldsymbol{s}))$ if and only if $\boldsymbol{s}$ is non-increasing.

In this thesis, we will often discuss strings that have content $\mathbb{M}$ that also have a specific suffix such as $\boldsymbol{z}$. In particular, it will often be the case that the remaining symbols are arranged in non-increasing order. Formally, this string can be specified by

$$\lnot(\mathbb{M} \backslash \mathsf{content}(\boldsymbol{s})) \cdot \boldsymbol{z}.$$

For notational convenience, the above string can also be expressed as follows

$$\lnot \cdot \boldsymbol{z}.$$

In other words, when $\lnot$ appears without any arguments, then it will denote the prefix that is non-increasing and allows the resulting string to have content $\mathbb{M}$. That is,

$$\lnot \cdot \boldsymbol{z} = \boldsymbol{p} \cdot \boldsymbol{z} \text{ where } \mathsf{content}(\boldsymbol{p} \cdot \boldsymbol{z}) = \mathbb{M} \text{ and } \lnot(\boldsymbol{p}) = \boldsymbol{p}.$$

For example, if $\mathbb{M} = \{1, 2, 2, 3, 3, 4\}$ then

$$\urcorner\llcorner = 433221 \qquad \urcorner\llcorner \cdot 21 = 4332 \cdot 21 \qquad \urcorner\llcorner \cdot 321 = 432 \cdot 321 \qquad \urcorner\llcorner \cdot 34 \cdot 221 = 334221.$$

Notice that $\urcorner\llcorner \cdot \boldsymbol{z}$ is only valid when $\mathsf{content}(\boldsymbol{z}) \subset \mathbb{M}$. Thus, using the above example, $\urcorner\llcorner \cdot 11$ would be undefined.

2.1.4 Frozen Prefixes

While the non-increasing prefix of string depends only on the string itself, the frozen prefix of a string also depends on a language which contains the string.

Definition 2.1.5 (Frozen Prefix)**.** *The* frozen prefix *of* $\boldsymbol{s} \in \mathbf{L}$ *is*

$$\text{🌡}_{\mathbf{L}}(\boldsymbol{s}) = s_1 s_2 \cdots s_f$$

where f *is the maximum value such that*

$$\mathbf{L}/s_{f+1} s_{f+2} \cdots s_n = \{s_1 s_2 \cdots s_f\}.$$

In other words, the frozen prefix of $\boldsymbol{s}$ *is the longest prefix that cannot be rearranged to create another string within* $\mathbf{L}$*.*

The symbol in Definition 2.1.5 reflects the notion that $\mathbf{L}$ "freezes" the prefix $s_1 s_2 \cdots s_f$ within $\boldsymbol{s}$. Within the definition, notice that the value of f is well-defined. In particular,

$$\mathbf{L}/s_2 s_3 \cdots s_n = \{s_1\}$$

and so $f \geq 1$. (In particular, $\boldsymbol{s} \in \mathbf{L}$, and there cannot be two strings with suffix $s_2 s_3 \cdots s_n$ since $\mathbf{L}$ has fixed-content.) By convention,

$$\text{🌡}(\boldsymbol{s}) = \text{🌡}_{\mathbf{L}}(\boldsymbol{s}).$$

To illustrate Definition 2.1.5, suppose

$$\mathbf{L} = \{1234, 1243, 1324, 1342, 1423, 2134, 2314, 2341, 2413, 3124, 3412, 4123\}.$$

Notice that $\mathbf{L}$ is the language with content $\{1, 2, 3, 4\}$ and the property that each string has at least two increases. For example, $1342 \in \mathbf{L}$ since it has increases 13 and 34. On the other hand, $3142 \notin \mathbf{L}$ since its only increase is 14. The frozen prefixes for

several strings in $\mathbf{L}$ are given below

$$🌡(2341) = 234 \qquad 🌡(1243) = 12 \qquad 🌡(1234) = 1.$$

For example, $|🌡(1243)| \geq 2$ because rearranging its first two symbols results in 2143, and $2143 \notin \mathbf{L}$ since its only increase is 14. On the other hand, $|🌡(1243)| \leq 3$ because its first three symbols can be rearranged to create 1243, and $1243 \in \mathbf{L}$ since it has the increases 12 and 24. Therefore, $|🌡(1243)| = 2$, and so $🌡(1243) = 12$ as given above. When discussing an individual string $\boldsymbol{s} \in \mathbf{L}$, the ith position in $\boldsymbol{s}$ is said to be *frozen* if $i \leq |🌡_{\mathbf{L}}(\boldsymbol{s})|$ and is otherwise *unfrozen*. For example, the second and third positions of 1243 are respectively frozen and unfrozen since $|🌡(1243)| = 2$, as shown in the middle column above. Informally, it is often convenient to use this nomenclature when referring to a symbol instead of a position. For example, there is no confusion in stating that the 2 is frozen within 1243

To conclude this section it is useful to consider what happens to frozen prefixes when a string is added to a language. For the sake of illustration, consider again the language $\mathbf{L}$ given above. If $\mathbf{L}' = \mathbf{L} \cup \{3241\}$ then notice that

$$|🌡_{\mathbf{L}'}(2341)| = |2| < |234| = |🌡_{\mathbf{L}}(2341)|.$$

Notice that the length of the frozen prefix of 2341 has decreased. This is due to the fact that the added string, 3241, allows a shorter prefix of 2341 to be rearranged to create another string in the resulting language. In general, the following remark holds and is used in Section 2.4.1.

Remark 2.1.6 (Frozen prefixes in language subsets)**.** *Suppose* $\mathbf{L} \subseteq \mathbf{L}'$*. Then, for any* $\boldsymbol{s} \in \mathbf{L}$*,*

$$|🌡_{\mathbf{L}'}(\boldsymbol{s})| \leq |🌡_{\mathbf{L}}(\boldsymbol{s})|.$$

In other words, adding strings to a language can only decrease the length of a frozen prefix.

2.1.5 Shifts

This section defines several types of shifts including left-shifts, right-shifts, and bubble left-shifts. In the case of the first two definitions, the notation $\mathsf{shift}(\boldsymbol{s}, i, j)$ will refer to shifting the ith symbol into the jth position, with $\overleftarrow{\mathsf{shift}}(\boldsymbol{s}, i, j)$ being used when $i \geq j$ (since the ith symbol is shifted to the left into the jth position), and $\overrightarrow{\mathsf{shift}}(\boldsymbol{s}, i, j)$ being

used when $i \leq j$ (since the ith symbol is shifted to the right into the jth position).

Definition 2.1.7 (Left-shift)**.** *The* left-shift *of the ith symbol in $\boldsymbol{s} = s_1 s_2 \cdots s_n$ into the jth position with $i \geq j$ is*

$$\overleftarrow{\mathsf{shift}}(\boldsymbol{s}, i, j) = s_1 s_2 \cdots s_{i-1} s_j s_i s_{i+1} \cdots s_{j-1} s_{j+1} s_{j+2} \cdots s_n.$$

In other words, $\overleftarrow{\mathsf{shift}}(\boldsymbol{s}, i, j)$ is the string that results from moving s_i to the left into position j, and moving the intermediate symbols, $s_i s_{i+1} \cdots s_{j-1}$, one position to the right. That is, the substring $s_j s_{j+1} \cdots s_i$ is replaced by $s_i s_j s_{j+1} \cdots s_{i+1}$.

Definition 2.1.8 (Right-shift)**.** *The* right-shift *of the ith symbol in $\boldsymbol{s} = s_1 s_2 \cdots s_n$ into the jth position with $i \leq j$ is*

$$\overrightarrow{\mathsf{shift}}(\boldsymbol{s}, i, j) = s_1 s_2 \cdots s_{i-1} s_{i+1} s_{i+2} \cdots s_j s_i s_{j+1} s_{j+2} \cdots s_n.$$

In other words, $\overrightarrow{\mathsf{shift}}(\boldsymbol{s}, i, j)$ is the string that results from moving s_i to the right into position j, and moving the intermediate symbols, $s_{i+1} s_{i+2} \cdots s_j$, one position to the left. That is, the substring $s_i s_{i+1} \cdots s_j$ is replaced by $s_{i+1} s_{i+2} \cdots s_j s_i$.

The left-shift $\overleftarrow{\mathsf{shift}}(\boldsymbol{s}, i, j)$ is equivalent to applying the permutation $(i\ i+1 \cdots j)$ to the positions of $\boldsymbol{s}$. Similarly, the right-shift $\overrightarrow{\mathsf{shift}}(\boldsymbol{s}, i, j)$ is equivalent to applying the permutation $(j\ j+1 \cdots i)$ to the positions of $\boldsymbol{s}$. If $\overleftarrow{\mathsf{shift}}(\boldsymbol{s}, i, j)$ is written then by assumption $i \geq j$, and if $\overrightarrow{\mathsf{shift}}(\boldsymbol{s}, i, j)$ is written then by assumption $i \leq j$. A shift is *trivial* if the resulting string is the same as the original, and otherwise the shift is *non-trivial*. (Notice that $i = j$ is sufficient, but not necessary, for ensuring that a shift is trivial.) The following three examples illustrate these shifts, with the shift in the middle being trivial

$= \overleftarrow{\mathsf{shift}}(3858487, 5, 3)$	$\overleftarrow{\mathsf{shift}}(7269954, 5, 4)$	$= \overrightarrow{\mathsf{shift}}(5330971, 1, 7)$
$= 38\overleftarrow{585}87$	$= 726\overleftarrow{99}54$	$= \overrightarrow{5330971}$
$= 3845887$	$= 7269954$	$= 3309715.$

A bubble left-shift is a left-shift that moves a symbol past exactly one symbol that is different than the symbol being shifted.

Definition 2.1.9 (Bubble Left-shift)**.** *The* bubble left-shift *of the ith symbol in $\boldsymbol{s} = s_1 s_2 \cdots s_n$ is*

$$\overleftarrow{\mathsf{bubble}}(\boldsymbol{s}, i) = \overleftarrow{\mathsf{shift}}(\boldsymbol{s}, i, h)$$

where h is the maximum value such that $s_h \neq s_i$ and $h < i$. In other words, a bubble left-shift moves a symbol to the left past one differing symbol.

Notice that $\overleftarrow{\mathsf{bubble}}(\boldsymbol{s}, i)$ is undefined when the first i symbols of $\boldsymbol{s}$ contain only one symbol. That is, $\overleftarrow{\mathsf{bubble}}(\boldsymbol{s}, i)$ is undefined when $s_1 s_2 \cdots s_i = s_1^i$. The *bubble right-shift* $\overrightarrow{\mathsf{bubble}}(\boldsymbol{s}, i)$ is analogously defined. These definitions are illustrated below

$\overleftarrow{\mathsf{bubble}}(332211, 6)$	$\overleftarrow{\mathsf{bubble}}(3309715, 2)$	$\overrightarrow{\mathsf{bubble}}(7269954, 4)$
$= 332\overleftarrow{211}$	$= \mathsf{undefined}$	$= 726\overrightarrow{9954}$
$= 332121$		$= 7269594.$

In the leftmost example, notice that the non-increasing string 332211 is the lexicographically largest permutation of $\{1, 1, 2, 2, 3, 3\}$ and 332121 is the lexicographically second-largest permutation of $\{1, 1, 2, 2, 3, 3\}$. (The definition of lexicographic order is formalized in Section 2.3.7.) This simple observation is generalized by the following remark, which provides insight to the definition of bubble languages found in the next section.

Remark 2.1.10 (Lexicographically largest and second-largest strings)**.** *The non-increasing string $\lceil\!\lfloor$ is the lexicographically largest string with content $\mathbb{M}$. The lexicographically second-largest string with content $\mathbb{M}$ is*

$$\overleftarrow{\mathsf{bubble}}(\lceil\!\lfloor, n).$$

In other words, the lexicographically second-largest string is obtained by bubble left-shifting the last symbol in the lexicographically largest string.

2.2 Bubble Language Definition

Given the preliminary results in Chapter 2.1, the concept of a bubble language can now be discussed. Section 2.2.1 gives the formal definition of a bubble language for fixed-content languages, and then Section 2.2.2 specializes the definition to fixed-density languages. Several basic properties of bubble languages are also discussed in Section 2.2.1.

2.2.1 Fixed-Content

The definition of a bubble language appears below. This is followed by a discussion of several key properties and an illustration of the definition.

Definition 2.2.1 (Bubble language (fixed-content))**.** *A fixed-content language* $\mathbf{L}$ *is a* bubble language *if for every* $\boldsymbol{s} \in \mathbf{L}$ *with* $\boldsymbol{s} \neq \llcorner$

$$\overleftarrow{\mathsf{bubble}}(\boldsymbol{s}, |\llcorner(\boldsymbol{s})| + 1) \in \mathbf{L}, \tag{2.1}$$

and for every $\boldsymbol{s} \in \mathbf{L}$

$$\overleftarrow{\mathsf{bubble}}(\boldsymbol{s}, i) \in \mathbf{L} \text{ for all } i \text{ within } |\text{🌡}(\boldsymbol{s})| < i \leq |\llcorner(\boldsymbol{s})|. \tag{2.2}$$

In other words, bubble languages are closed under bubble left-shifting every unfrozen symbol in the non-increasing prefix, as well as the symbol following the non-increasing prefix (if it exists).

To illustrate Definition 2.2.1, suppose that $\mathbf{L}$ is a bubble language and that

$$4332114213 \in \mathbf{L}. \tag{2.3}$$

The two conditions in Definition 2.2.1 will be discussed in turn. First, condition (2.1) implies that the symbol following the non-increasing prefix can always be bubble left-shifted to create an additional string in the language. This bubble left-shiftable symbol is illustrated below in bold

$$\underset{\llcorner}{4332114}213.$$

Thus, (2.1) implies that $433211\overleftarrow{4}213 = 4332141213 \in \mathbf{L}$. In general, (2.1) implies that the frozen prefix of any string is non-increasing. That is, if $\mathbf{L}$ is a bubble language and $\boldsymbol{s} \in \mathbf{L}$ then

$$|\text{🌡}(\boldsymbol{s})| \leq |\llcorner(\boldsymbol{s})|. \tag{2.4}$$

This important fact justifies the inequality given in (2.2) and will be used implicitly throughout this thesis. By repeated applications of condition (2.1), the symbol following the non-increasing prefix can be bubble left-shifted until it merges into the non-increasing prefix. This is illustrated below

$$433211\overleftarrow{4}213, 43321\overleftarrow{4}1213, 4332\overleftarrow{4}11213, 433\overleftarrow{4}211213, 43\overleftarrow{4}3211213, 4433211213 \in \mathbf{L}$$

where the arrows represent the shift that moves the bolded 4 toward the non-increasing prefix. Once the symbol following the non-increasing prefix is left-shifted into the non-increasing prefix then the process can be repeated on the resulting string, so long as it is not non-increasing. This fact is illustrated below by continuing from the rightmost string above

$$44332\overleftarrow{1\mathbf{2}}13, 44332\mathbf{2}113 \in \mathbf{L}.$$

The process can again be repeated to give the following

$$4433221\overleftarrow{1\mathbf{3}}, 443322\overleftarrow{1\mathbf{3}}1, 44332\overleftarrow{2\mathbf{3}}11, 4433\overleftarrow{2\mathbf{3}}211, 4433\mathbf{3}2211 \in \mathbf{L}.$$

Now the process cannot be repeated since the resulting string, 443332211, is itself non-increasing. In general, given a string in a bubble language, the symbols in any prefix can rearranged into non-increasing order and the resulting string is guaranteed to be in the language. This simple non-increasing prefix property is a direct consequence of condition (2.1) and so it is a necessary condition for being a bubble language. (On the other hand, $\mathbf{L} = \{321, 213\}$ shows that it is possible to satisfy the non-increasing prefix property without satisfying (2.1); in particular, (2.1) would imply that $\overleftarrow{\mathsf{bubble}}(213, 3) = 231 \in \mathbf{L}$.)

Remark 2.2.2 (Non-increasing prefix property in bubble languages)**.** *Suppose* $\mathbf{L}$ *is a bubble language. Then,*

$$\boldsymbol{p} \cdot \boldsymbol{z} \in \mathbf{L} \text{ implies } \ulcorner \cdot \boldsymbol{z} \in \mathbf{L}. \tag{2.5}$$

In other words, any prefix of any string in a bubble language can be rearranged into non-increasing order and the result is also in the language.

In particular, Remark 2.2.2 implies that every non-empty bubble language contains the non-increasing string $\ulcorner$. In other words, every non-empty bubble language contains the lexicographically largest string with the language's content. The following lemma shows that non-trivial bubble languages contain the lexicographically second-largest string. (The bubble language

$$\{332211, 332121, 323211\}$$

shows that bubble languages can contain three strings without containing the lexicographically third-largest string.) Lemma 2.2.3 is especially useful when combined with the closure of bubble languages under quotients found in Lemma 2.4.1.

Lemma 2.2.3 (Non-Empty and Non-Trivial Bubble Languages)**.** *Suppose* $\mathbf{L}$ *is a bubble language. Then,*

$$|\mathbf{L}| \geq 1 \text{ implies } \urcorner\llcorner \in \mathbf{L}. \tag{2.6}$$

Furthermore,

$$|\mathbf{L}| \geq 2 \text{ implies } \overleftarrow{\mathsf{bubble}}(\urcorner\llcorner, n) \in \mathbf{L}. \tag{2.7}$$

In other words, non-empty bubble languages contain the lexicographically largest string, and non-trivial bubble languages contain the lexicographically second-largest string.

Proof. The proof of (2.6) follows from Remark 2.2.2 by $\boldsymbol{z} = \epsilon$. To prove (2.7) suppose that $\mathbf{L}$ is a bubble language and $|\mathbf{L}| \geq 2$. Therefore, (2.6) implies that $\urcorner\llcorner \in \mathbf{L}$. Furthermore, $\mathsf{bubble}(\urcorner\llcorner) \neq \urcorner\llcorner$ since there is some other string in $\mathbf{L}$. Therefore, from (2.2) it must be that $\overleftarrow{\mathsf{bubble}}(\urcorner\llcorner, n) \in \mathbf{L}$. □

It is noted that condition (2.1) is not sufficient for ensuring a shift Gray code. For example,

$$\{654321, 564321, 653421, 654312\}$$

satisfies (2.1) because

$$654321 = \overleftarrow{56}4321 = 65\overleftarrow{34}21 = 6543\overleftarrow{12}.$$

However, it does not have a shift Gray code since the corresponding directed graph in Figure 2.1 does not have a directed Hamiltonian path.

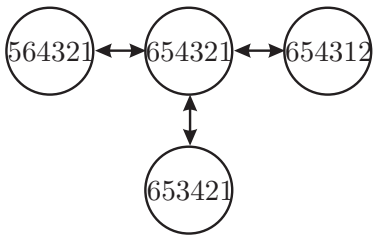
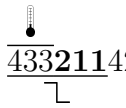


Figure 2.1: Left-shift adjacency graph for $\mathbf{L} = \{654321, 564321, 653421, 654312\}$.

Now let us consider the previously discussed string in (2.3) with respect to the second condition (2.2). Recall that the frozen prefix must be non-increasing by (2.4), but its exact length depends on specific bubble language under consideration. For the sake of argument, suppose that the length of the frozen prefix in (2.3) is 3. This is illustrated below

$$\underset{\urcorner\llcorner}{\overline{433}}2114213.$$

Condition (2.2) then states that remaining symbols in the non-increasing prefix can be bubble left-shifted to create additional strings in the language. These bubble left-shiftable symbols are illustrated below in bold



Thus, (2.2) implies $43\overleftarrow{32}114213 = 4323114213 \in \mathbf{L}$ and $433\overleftarrow{21}14213 = 43\overleftarrow{3211}4213 = 4331214213 \in \mathbf{L}$. Before concluding this section, it is noted that condition (2.2) is also not sufficient for ensuring a shift Gray code. For example,

$$\mathbf{L} = \{43215, 43521, 53214, 53421\}$$

vacuously satisfies (2.2) since every non-increasing prefix is frozen. On the other hand, $\mathbf{L}$ does not have a shift Gray code since no shift transforms 43215 or 43521 into 53214 or 53421. In other words, the underlying shift adjacency graph is not connected.

2.2.2 Fixed-Density

"There are 10 types of people in the world — those who understand binary and those who don't."

- Anonymous

This section focuses on bubble languages that have fixed-density. When a language has fixed-density then the two conditions found in Definition 2.2.1 can be replaced by a single condition. To illustrate this unifying condition, suppose that $\mathbf{L}$ is a fixed-density bubble language and that

$$11110001100 \in \mathbf{L}.$$

From condition (2.1) in Definition 2.2.1, another string in $\mathbf{L}$ can be created by bubble left-shifting the symbol following the non-increasing prefix. That is,

$$11110010100 \in \mathbf{L}.$$

Notice that this second string in $\mathbf{L}$ can be obtained from the first string in $\mathbf{L}$ by swapping the symbols in its leftmost 01 substring. Furthermore, this is always true since the only increasing pair of symbols in a fixed-density language is 01. In other words, condition (2.1) is equivalent to the following condition: For any binary string

$\boldsymbol{z}$ and all $i, j \geq 0$

$$1^i0^j \cdot 01 \cdot \boldsymbol{z} \in \mathbf{L} \text{ implies } 1^i0^j \cdot 10 \cdot \boldsymbol{z} \in \mathbf{L}.$$

Notice that the symbols in the leftmost 01 have been swapped within the above implicant. Since the above condition is identical to (2.1) in Definition 2.2.1, then it is a necessary condition for a fixed-density language to be a bubble language. On the other hand, a small argument also shows that this new condition is also sufficient. In other words, a fixed-density language is a bubble language if and only if it satisfies the above condition.

Theorem 2.2.4 (Bubble Language (Fixed-Density))**.** *A fixed-density language* $\mathbf{L}$ *is a bubble language if and only if for any binary string* $\boldsymbol{z}$ *and all* $i, j \geq 0$

$$1^i0^j \cdot 01 \cdot \boldsymbol{z} \in \mathbf{L} \textit{ implies } 1^i0^j \cdot 10 \cdot \boldsymbol{z} \in \mathbf{L}. \tag{2.8}$$

In other words, a fixed-density language is a bubble language if the language is closed under the operation of swapping the symbols in the leftmost 01 *substring, which is equivalent to bubble left-shifting the symbol following the non-increasing prefix.*

As previously discussed, (2.8) is equivalent to (2.1) within the definition of a bubble language. Therefore, if a language violates (2.8) then it is not a bubble language. Similarly, if a language satisfies (2.8) then it satisfies (2.1). Therefore, to complete the proof of Theorem 2.2.4, it remains only to prove that fixed-density languages satisfying (2.8) also satisfy condition (2.2) within the definition of a bubble language.

Proof. Suppose that $\mathbf{L}$ is a fixed-density language that satisfies (2.8). It will be proven that $\mathbf{L}$ satisfies (2.2). In order to do this, suppose that $\boldsymbol{s} = 1^x0^y \cdot \boldsymbol{z} \in \mathbf{L}$ and $\urcorner\llcorner(\boldsymbol{s}) = 1^x0^y$. If $\text{🌡}(\boldsymbol{s}) = \urcorner\llcorner(\boldsymbol{s})$ then (2.2) is (vacuously) true. Otherwise, the non-increasing prefix of $\boldsymbol{s}$ is not frozen. Therefore, the non-increasing prefix must contain both 1s and 0s, and only the 1s are frozen. That is, $\text{🌡}(\boldsymbol{s}) = 1^x$ and $x, y > 0$. Therefore, in order to prove that (2.2) holds, it must be shown that

$$\overleftarrow{\mathsf{bubble}}(\boldsymbol{s}, x+1) = \overleftarrow{\mathsf{bubble}}(1^x0^y \cdot \boldsymbol{z}, x+1) = 1^{x-1}010^{y-1}\boldsymbol{z} \in \mathbf{L}.$$

Since the non-increasing prefix of $\boldsymbol{s}$ is not frozen, then it can be rearranged to create another string in $\mathbf{L}$. That is, there exists $\boldsymbol{p} \cdot \boldsymbol{z} \in \mathbf{L}$ with $\boldsymbol{p} \neq 1^x0^y$. Since $\boldsymbol{p} \neq 1^x0^y$ then it contains a rightmost 01. Therefore, $\boldsymbol{p}$ can be expressed as

$$\boldsymbol{p} = \boldsymbol{q} \cdot 0^a \cdot 01 \cdot 0^b$$

where $\boldsymbol{q}$ contains $x-1$ copies of 1, and $y-a-b-1$ copies of 0. By Remark 2.2.2, $\boldsymbol{q}$ can be rearranged into non-increasing order to produce another string in $\mathbf{L}$. (Remark 2.2.2 can be applied because $\mathbf{L}$ satisfies (2.8), which is equivalent to (2.1), and $\boldsymbol{s} =\in \mathbf{L}$.) That is,

$$1^{x-1} \cdot 0^{y-a-b-1} \cdot 0^a \cdot 01 \cdot 0^b \boldsymbol{z} = 1^{x-1} \cdot 0^{y-b} \cdot 1 \cdot 0^b \boldsymbol{z} \in \mathbf{L}.$$

Therefore, by $y-b-1$ applications of (2.8) it is also true that

$$1^{x-1}010^{y-1}\boldsymbol{z} \in \mathbf{L},$$

which completes the proof. □

2.3 Examples

> "There are lists of mathematical objects that have a historical interest, or maybe I should say that there are lists of historical objects that have a mathematical interest.
>
> - Frank Ruskey

This section proves that a number of specific languages are bubble languages. The specific languages are important because they can be used to represent familiar and well-studied combinatorial objects such as multiset permutations (Section 2.3.1), balanced parentheses and k-ary Dyck words (Section 2.3.2), linear-extensions of B-posets (Section 2.3.3), unit interval graphs (Section 2.3.4), ordered trees with fixed branching sequence (Section 2.3.5), and multiset necklaces and multiset Lyndon words (Section 2.3.7). (Additional bubble languages are discussed in Section 2.4.1.) Although these results help to motivate the definition of bubble languages, they are not explicitly used elsewhere in this thesis. For this reason, the specific languages are formally defined, but the combinatorial objects they represent are discussed informally. This section also provides several languages that are not bubble languages. In particular, these languages include natural representations for linear-extensions of posets (Section 2.3.3) and multiset bracelets (Section 2.3.7).

To prove a language is a bubble language, it is necessary to verify that it satisfies the two shift conditions given in Definition 2.2.1. On the other hand, to prove a language is not a bubble language, a single counterexample suffices. Due to the nature of the conditions in (2.2) and (2.1), such counterexamples simply involve a string inside of the given language and a string outside of the given language. For this reason, it can be said that non-bubble languages have small *certificates* that verify their non-membership in the bubble language hierarchy.

2.3.1 Multiset Permutations

The language containing all strings over $\mathbb{M}$ is referred to as the language of *multiset permutations* of $\mathbb{M}$. This definition is formalized below, together with the remark that all fixed-content languages are subsets of some language of multiset permutations.

Definition 2.3.1 (Multiset permutations)**.** $\Pi(\mathbb{M})$ *denotes the language of* multiset permutations *of* $\mathbb{M}$ *defined by*

$$\Pi(\mathbb{M}) = \Big\{ \boldsymbol{s} \mid \mathsf{content}(\boldsymbol{s}) = \mathbb{M} \Big\}$$

In other words, $\Pi(\mathbb{M})$ *contains every permutation of the symbols within the multiset* $\mathbb{M}$*.*

Notice that every language with fixed-content $\mathbb{M}$ is a subset of the multiset permutations over $\mathbb{M}$. Similarly, all fixed-density languages are a subset of some *combination* language.

Definition 2.3.2 (Combinations)**.** $\mathbf{C}(j,k)$ *denotes the language of* combinations *defined by*

$$\mathbf{C}(j,k) = \Big\{ s_1 s_2 \cdots s_n \mid \{s_1, s_2, \cdots, s_n\} = \{\overbrace{0,0,\ldots,0}^{j \text{ copies}}, \overbrace{1,1,\ldots,1}^{k \text{ copies}}\} \Big\}$$

In other words, $\mathbf{C}(j,k)$ *contains every string with* j *copies of* 0 *and* k *copies of* 1*.*

Examples of these languages are given below

$\Pi(\{1,2,3\}) = \{123, 213, 132, 312, 231, 321\}$ and $\mathbf{C}(2,1) = \Pi(\{0,0,1\}) = \{001, 010, 100\}$.

Since every possible string over $\mathbb{M}$ is contained within $\Pi(\mathbb{M})$ then both conditions in Definition 2.2.1 are satisfied. Therefore, multiset permutations languages (and the special case of combinations) are bubble languages.

2.3.2 Balanced Parentheses and k-ary Dyck Words

One language that is particularly important to computer scientists and discrete mathematicians is the language of *balanced parentheses* with fixed length. A string is a *balanced parentheses string* if it contains an equal number of open parentheses as closed parentheses, and the property that every prefix contains at least as many open parentheses as closed parentheses. For example, (()()) is a balanced parentheses string, while ())(() is not a balanced parentheses string due to its prefix of length

three. The number of balanced parentheses strings of length $2k$ is equal to the kth Catalan number

$$\frac{1}{k+1}\binom{2k}{k}.$$

This number counts a many additional combinatorial objects, in fact a 27-page textbook excerpt [86] includes 66 combinatorial interpretations. In particular, the balanced parentheses strings of length $2i$ are in bijective-correspondence with binary trees that contain i internal nodes. This fact is illustrated for $i = 3$ in Figure 2.2.

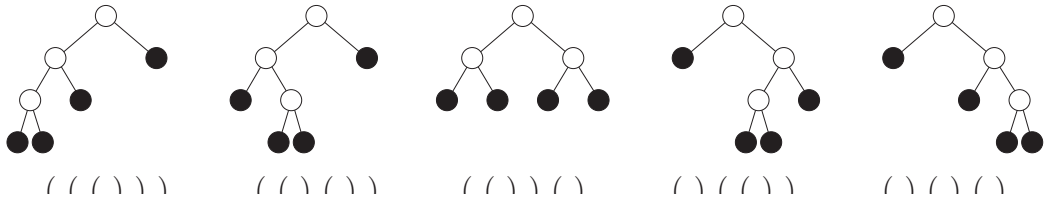


Figure 2.2: Binary trees with 3 internal nodes are in one-to-one correspondence with balanced parentheses strings of length 6.

Balanced parentheses strings are often represented with binary strings, where 1 and 0 represent an open and closed parenthesis, respectively.

Definition 2.3.3 (Balanced parentheses)**.** $\mathbf{P}(i)$ *denotes the language of* balanced parentheses *of length* $2i$ *defined by*

$$\mathbf{P}(i) = \{\boldsymbol{s} \in \mathbf{C}(i,i) \mid \text{ if } \boldsymbol{s} = \boldsymbol{pz} \text{ then } 2 \cdot \Sigma(\boldsymbol{p}) \geq |\boldsymbol{p}|\}.$$

In other words, $\mathbf{P}(i)$ *is the fixed-density language representing the balanced parentheses strings of length* $2i$*, where* 1 *represents '(' and* 0 *represents ')'.*

Examples of this language are given below

$\mathbf{P}(1) = \{10\}$

$\mathbf{P}(2) = \{1100, 1010\}$

$\mathbf{P}(3) = \{111000, 110100, 110010, 101100, 101010\}$

$\mathbf{P}(4) = \{11110000, 11101000, 11100100, 11100010, 11011000, 11010100, 11010010,$
$11001100, 11001010, 10111000, 10110100, 10110010, 10101100, 10101010\}.$

More generally, *k-ary Dyck words* ensure that every prefix contains at least $k-1$ times as many 0s as 1s. Similarly, k-ary Dyck words of length ki are in bijective correspondence with k-ary trees containing i internal nodes [86].

Definition 2.3.4 (k-ary Dyck Words)**.** $\mathbf{D}(k,i)$ *denotes the language of* k-ary Dyck words *of length* ki *defined by*

$$\mathbf{D}(k,i) = \{\boldsymbol{s} \in \mathbf{C}((k-1)\cdot i, i) \mid \text{ if } \boldsymbol{s} = \boldsymbol{pz} \text{ then } k\cdot\Sigma(\boldsymbol{p}) \geq |\boldsymbol{p}|\}.$$

In other words, $\mathbf{D}(k,i)$ *is the fixed-density language representing* k*-ary Dyck words.*

For example,

$\mathbf{D}(3,1) = \{100\}$

$\mathbf{D}(3,2) = \{110000, 101000, 100100\}$

$\mathbf{D}(3,3) = \{111000000, 110100000, 110010000, 110001000, 110000100, 101100000,$
$101010000, 101001000, 101000100, 100110000, 100101000, 100100100\}.$

Balanced parentheses and k-ary Dyck words are special cases of the bubble languages discussed in Section 2.3.3. Balanced parentheses can also be viewed as walks in a grid, and variations of these walks are discussed in Section 2.4.1. Chapter 5 discusses a further restriction of balanced parentheses.

2.3.3 Linear-Extensions

Linear-extensions of B-posets generalize combinations, balanced parentheses, and k-ary Dyck words. Linear-extensions of B-posets are a specialization of linear-extensions of posets, which are discussed informally at the end of this section along with Hasse diagrams. To begin the discussion of these languages, notice that strings within $\mathbf{P}(k)$ (balanced parentheses) must satisfy the condition that the ith-leftmost 1 must occur within the first $2i-1$ symbols. For example,

$$11000\underline{1}10 \notin \mathbf{P}(4)$$

because the (underlined) 3rd-leftmost 1 is not contained within the first five symbols. More generally, a subset of $\mathbf{C}(j,k)$ can be defined by the values

$$1 \leq b_1 < b_2 < b_3 < \ldots < b_k \leq j+k$$

and the property that the ith 1 must occur within the first b_i symbols. In other words, the density of the first b_i symbols must be at least i. In this thesis, languages of this type are referred to as *linear extensions of B-posets*. These languages are formally

defined below, followed by examples, remarks, and a general discussion of the terms *poset* and *linear-extensions*.

Definition 2.3.5 (Linear-Extensions of B-posets ($\mathbf{B}$))**.** *Suppose that* $1 \leq b_1 < b_2 < b_3 < \ldots < b_h \leq n$ *and* $n \geq h$*. Then, the language of* linear extensions of B-posets *with respect to these values is*

$$\mathbf{B}_n(b_1, b_2, \ldots, b_h) = \{\boldsymbol{s} \in \mathbf{C}(n-h, h) \mid \Sigma(s_1 s_2 \ldots s_{b_g}) \geq g \text{ for all } 1 \leq g \leq h\}.$$

In other words, the g*th* 1 *must occur within the first* b_g *symbols. Equivalently, the density of the first* b_g *symbols must be at least* g*.*

For example,

$$\mathbf{B}_6(3,5) = \{110000, 101000, 011000, 1001000, 0101000, 001100, 100010, 010010, 001010\}$$

since the above strings are exactly the binary strings of length six where the leftmost 1 is in the first three positions, and the second-leftmost 1 is in the first five positions. The next three remarks note that linear-extensions of B-posets can be used to represent combinations, balanced parentheses, and k-ary Dyck words.

Remark 2.3.6 (Combinations and linear-extensions of B-posets)**.** *Let* $n = j + k$*. Then,*

$$\mathbf{C}(j,k) = \mathbf{B}_n(j+1, j+2, \ldots, n).$$

In other words, combinations are a special case of linear-extensions of B-posets.

Remark 2.3.7 (Balanced parentheses and linear-extensions of B-posets)**.** *Let* $n = 2i$*. Then,*

$$\mathbf{P}(i) = \mathbf{B}_n(1, 3, 5, \ldots, n-1).$$

In other words, balanced parentheses are a special case of linear-extensions of B-posets.

Remark 2.3.8 (k-ary Dyck words and linear-extensions of B-posets)**.** *Let* $n = k \cdot i$*. Then,*

$$\mathbf{D}(k,i) = \mathbf{B}_n(1, k+1, 2k+1, \ldots, (i-1)\cdot k + 1).$$

In other words, k-ary Dyck words are a special case of linear-extensions of B-posets.

Now it is proven that the linear-extensions of B-posets are a bubble language.

Theorem 2.3.9 (Linear-extensions of B-posets are bubble languages)**.** *Suppose* $1 \leq b_1 < b_2 < b_3 < \ldots < b_h \leq n$ *and* $n \geq h$*. Then,* $\mathbf{B}_n(b_1, b_2, \ldots, b_h)$ *is a bubble language*

Proof. Let $\mathbf{L} = \mathbf{B}_n(b_1, b_2, \ldots, b_h)$. By Theorem 2.2.4, it needs only to be shown that

$$1^i0^j \cdot 01 \cdot \boldsymbol{z} \in \mathbf{L} \text{ implies } 1^i0^j \cdot 10 \cdot \boldsymbol{z} \in \mathbf{L}.$$

However, this follows directly from Definition 2.3.5 since the sum of every prefix in $1^i0^j \cdot 10 \cdot \boldsymbol{z}$ is at least as large as the sum of every prefix in $1^i0^j \cdot 01 \cdot \boldsymbol{z}$. □

B-posets were introduced by Pruesse-Ruskey [60] and are special cases of partially ordered sets. A *partially ordered set* $\mathcal{P}$ is a reflexive, transitive, and anti-symmetric relation $R(\mathcal{P})$ on a set $S(\mathcal{P})$. If (a, b) is in the relation $R(\mathcal{P})$ then $a \succeq b$ is written. (The reader is reminded that partially ordered sets are usually defined with $\preceq$.) If $a \succeq b$ and $a \neq b$, then $a \succ b$ is written. For example, the following defines a relation $R(\mathcal{P})$ over $S(\mathcal{P}) = \{1, 2, 3, 4, 5, 6\}$:

$$6 \succ 5 \quad 5 \succ 4 \quad 6 \succ 4 \quad 3 \succ 2 \quad 2 \succ 1 \quad 3 \succ 1 \quad 6 \succ 3 \quad 6 \succ 2 \quad 6 \succ 1 \quad 5 \succ 2 \quad 5 \succ 1 \quad 4 \succ 1. \tag{2.9}$$

This relation is visualized in Figure 2.3.

A *linear-extension* of $\mathcal{P}$ is a permutation of $S(\mathcal{P})$ that satisfies the property that if $a \succeq b$ then a appears to the left of b. For example, 632541 is not a linear-extension of the poset specified in (2.9) because $5 \succ 2$ and 5 appears to the right of 2. The linear-extensions of the poset specified in (2.9) are given below

$$\{654321, 635421, 653421, 635241, 653241\}. \tag{2.10}$$

Notice that this is not a bubble language. In particular, $\boldsymbol{s} = 654321 \in \mathbf{L}$ and $\bullet(\boldsymbol{s}) = 654$. However,

$$\overleftarrow{\mathsf{bubble}}(654321, 5) = 654231 \notin \mathbf{L}$$

and so the above language does not satisfy condition (2.2). In general, the linear-extensions of partially ordered sets do not form bubble languages.

Within the poset specified in (2.9), notice that $6 \succ 5 \succ 4$. That is, $6 \succ 5$ and $5 \succ 4$. In general, $x_1 \succ x_2 \succ \cdots \succ x_a$ is known as a *chain*. Two chains $x_1 \succ x_2 \succ \cdots \succ x_a$ and $y_1 \succ y_2 \succ \cdots \succ y_b$ are *disjoint* unless $x_i = y_j$ for some $1 \leq i \leq a$ and $1 \leq j \leq b$. A set of chains is disjoint if they are pairwise disjoint. A set of chains *cover* $\mathcal{P}$ if they collectively contain all of the symbols in $S(\mathcal{P})$. (B-posets were originally defined as posets that have two disjoint chains $x_1 \succ x_2 \succ \cdots \succ x_a$ and $y_1 \succ y_2 \succ \cdots \succ y_b$ in which $y_j \not\succ x_i$ for all $1 \leq i \leq a$ and $1 \leq j \leq b$ [60].)

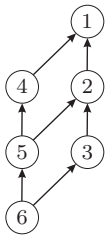


Figure 2.3: Hasse diagram for the poset in (2.9).

Given a linear-extension of $\mathcal{P}$ that has chain $x_1 \succ x_2 \succ \cdots \succ x_a$, notice that x_i must appear to the left of x_j for all $i < j$. For example, the poset specified in (2.9) has the chain $6 \succ 5 \succ 4$, and so the relative order of these symbols is fixed within each of the linear-extensions found in (2.10). Therefore, if these symbols are replaced by a symbol that doesn't otherwise appear within $S(\mathcal{P})$, then the original symbols can be recovered from any linear-extension based on their order within the chain. Such a substitution is called a *chain substitution*. For example, by substituting the symbols in the disjoint and covering chains $6 \succ 5 \succ 4$ and $3 \succ 2 \succ 1$ with '(' and ')' respectively, then the linear-extensions in (2.10) become

$$\{(\ (\ (\ )\ )\ ),\ (\ )\ (\ (\ )\ ),\ (\ (\ )\ (\ )\ ),\ (\ )\ (\ )\ (\ ),\ (\ (\ )\ )\ (\ )\ \}.$$

Notice that above language contains exactly the balanced parentheses of length six and so it is a bubble language. Figure 2.4 shows how similar chain substitutions can be used to create bubble languages for k-ary Dyck words, restricted Motzkin paths, and multiset permutations.

Before concluding this section it is mentioned that posets can be visualized by directed acyclic graphs known as *Hasse Diagrams*. Hasse Diagrams contain one node for each symbol and every non-transitive arc. More precisely, if the partially-ordered set $\mathcal{P}$ is defined using $\preceq$, then the Hasse Diagram contains an arc from the node for $x \in S(\mathcal{P})$ to the node for $y \in S(\mathcal{P})$ if and only if $x \prec y$ and there does not exist $z \in S(\mathcal{P})$ such that $x \prec z \prec y$. Furthermore, it is customary for the nodes to be arranged so that every arc is directed upwards. Since the partially ordered sets in this thesis are defined using $\succeq$ then these conventions are followed with the exception that larger symbols appear below smaller symbols. For example, Figure 2.3 contains the Hasse diagram for partially ordered set defined in (2.9) with linear-extensions given in (2.10). Within a Hasse diagram, a chain substitution corresponds to the nodes on a directed path being given the same label. Figure 2.4 contains examples of these modified Hasse diagrams for balanced parentheses, k-ary Dyck words, restricted

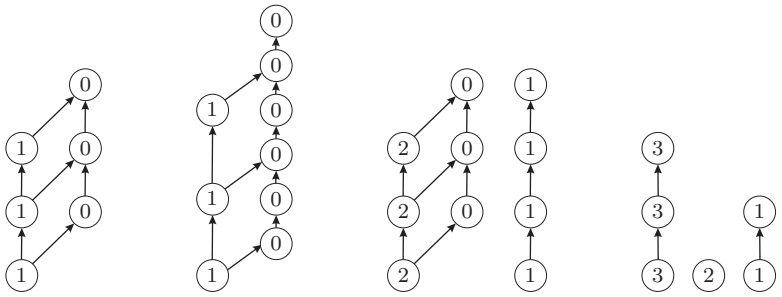


Figure 2.4: Hasse diagrams for $\mathbf{P}(3)$ (balanced parentheses), $\mathbf{D}(3,3)$ (k-ary Dyck words), $\mathbf{M}_4(5)$ (restricted Motzkin paths), and $\Pi(\{1,1,2,3,3,3\})$ (multiset permutations) using chain substitutions.

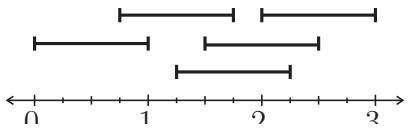
Motzkin paths, and multiset permutations. Finally, it is mentioned that the linear-extensions of the partially-ordered set $\mathcal{P}$ are in one-to-one correspondence with the *topological orderings* of its Hasse diagram [86].

2.3.4 Unit Interval Graphs

The *interval* $[a,b]$ is the set of real numbers between a and b inclusively. That is,

$$[a,b] = \{x \in \mathbb{R} \mid a \leq x \leq b\}.$$

An interval $[a,b]$ is a *unit interval* if $b = a{+}1$. Suppose $\mathbb{I} = \{[a_1,b_1],[a_2,b_2],\ldots,[a_n,b_n]\}$ is a set of intervals. Then $\mathbb{I}$ is a set of *unit intervals* if each of its intervals is a unit interval, and is *proper* unless $[a_j,b_j] = [a_k,b_k]$ for some $1 \leq j < k \leq n$. $\mathbb{I}$ can be visualized by n parallel line segments. For example,



represents the following set of connected proper unit intervals

$$\mathbb{I} = \{[0,1],\ [0.75,1.75],\ [1.25,2.25],\ [1.5,2.5],\ [2,3]\}. \tag{2.11}$$

Note that $[a_j,b_j]$ and $[a_k,b_k]$ have non-empty intersection if $a_j \leq a_k \leq b_j$ or $a_j \leq b_k \leq b_j$. In this case the intervals are said to *intersect*. $\mathbb{I}$ is *connected* unless $\mathbb{I}$ can be partitioned into non-empty $\mathbb{I}_1$ and $\mathbb{I}_2$ such that $[a_j,b_j]$ and $[a_k,b_k]$ do not intersect for all $[a_j,b_j] \in \mathbb{I}_1$ and all $[a_k,b_k] \in \mathbb{I}_2$.

The *parenthetical representation* of a set of proper unit intervals $\mathbb{I}$ is denoted by $\wr(\mathbb{I})$ and is obtained by sweeping across the real-line from left to right and recording

1 at the beginning of each interval, and 0 at the end of each interval. For example, the parenthetical representation of $\mathbb{I}$ in (2.11) is

$$\text{ⓘ}\ (\mathbb{I}) = 1101101000. \tag{2.12}$$

Notice 1101101000 is a balanced parentheses string. In general, if $\mathbb{I}$ is a set of n proper unit intervals then $\text{ⓘ}\ (\mathbb{I}) \in \mathbf{P}(n)$ because each of the n intervals begins before it ends. Similarly, if $\boldsymbol{s} \in \mathbf{P}(n)$ then there exists a set of n proper unit intervals $\mathbb{I}$ such that $\text{ⓘ}\ (\mathbb{I}) = \boldsymbol{s}$. Furthermore, $\mathbb{I}$ is connected if and only if no strict non-empty prefix of $\text{ⓘ}\ (\mathbb{I})$ contains the same number of 1s and 0s. Equivalently, $\mathbb{I}$ is connected if and only if $\text{ⓘ}\ (\mathbb{I}) = 1 \cdot \boldsymbol{t} \cdot 0$ for some $\boldsymbol{t} \in \mathbf{P}(n-1)$.

Parenthetical representations encapsulate which intervals intersect. In particular, suppose that $\mathbb{I} = \{[a_1, b_1], [a_2, b_2], \ldots, [a_n, b_n]\}$ is a set of proper unit intervals with $a_j < a_{j+1}$ for all $1 \le j < n$. Then, for all $1 \le j < k \le n$, $[a_j, b_j]$ intersects $[a_k, b_k]$ if and only if the jth 0 appears to the right of the kth 1 within $\text{ⓘ}\ (\mathbb{I})$. For example, the second 0 appears to the right of the fourth 1 in (2.12), and so $[0.75, 1.75]$ intersects $[1.5, 2.5]$ within (2.11).

Intersecting intervals can also be represented using a graph. Formally, a *graph* $\mathsf{G} = (\mathbb{V}, \mathbb{E})$ is a set of *vertices* $\mathbb{V} = \{v_1, v_2, \ldots, v_n\}$ together with a set of *edges* $\mathbb{E} = \{e_1, e_2, \ldots, e_m\}$, where each edge is an unordered subset of two distinct vertices. An edge is represented by the two vertices it contains. That is, if $e = \{v_j, v_k\}$ then $e = v_j v_k$ (or equivalently, $e = v_k v_j$) is written. A graph $\mathsf{G} = (\mathbb{V}, \mathbb{E})$ is an *interval graph* if it has an *interval representation*, which is a set of intervals $\mathbb{I} = \{[a_1, b_1], [a_2, b_2], \ldots, [a_n, b_n]\}$ such that for all $1 \le j \le n$ and $1 \le k \le n$

$$v_j v_k \in \mathbb{E} \iff [a_j, b_j] \text{ intersects } [a_k, b_k].$$

Interval graphs have a number of applications including assembling contiguous subsequences in DNA mapping and in resource allocation problems (see Wikipedia [99]). Basic results on interval graphs can be found in Golumbic [28].) A graph is a *unit interval graph* if it has a unit interval representation. Graphs can be visualized by drawing a circle for each vertex and a line between every pair of vertices that comprise an edge. For example, the graph $\mathsf{G} = (\mathbb{V}, \mathbb{E})$ with

$$\mathbb{V} = \{v_1, v_2, v_3, v_4, v_5\} \text{ and } \mathbb{E} = \{v_1v_2, v_2v_3, v_2v_4, v_3v_4, v_3v_5, v_4v_5\} \tag{2.13}$$

is visualized in Figure 2.5. Furthermore, the graph specified in (2.13) is a unit interval

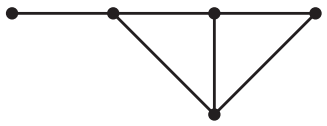


Figure 2.5: A unit interval graph for the unit intervals in (2.11) (or (2.14)).

graph since (2.11) provides a suitable unit interval representation. For example, the interval $[0, 1]$ corresponds to the leftmost vertex in Figure 2.5.

Notice that the graph in Figure 2.5 has been drawn without any vertex or edge labels. Two graphs $\mathsf{G} = (\mathbb{V}, \mathbb{E})$ and $\mathsf{G}' = (\mathbb{V}', \mathbb{E}')$ are *isomorphic* if there exists a bijection $\alpha : \mathbb{V} \to \mathbb{V}'$ such that

$$v_j v_k \in \mathbb{E} \iff \alpha(v_j)\alpha(v_k) \in \mathbb{E}'.$$

A graph $\mathsf{G} = (\mathbb{V}, \mathbb{E})$ is *connected* unless $\mathbb{V}$ can be partitioned into non-empty $\mathbb{V}_1$ and $\mathbb{V}_2$ such that $v_j v_k \notin \mathbb{E}$ for all $v_j \in \mathbb{V}_1$ and $v_k \in \mathbb{V}_2$. An interval graph is connected if and only if it has a connected interval representation. Figure 2.6 contains the ten connected unit interval graphs with five vertices.

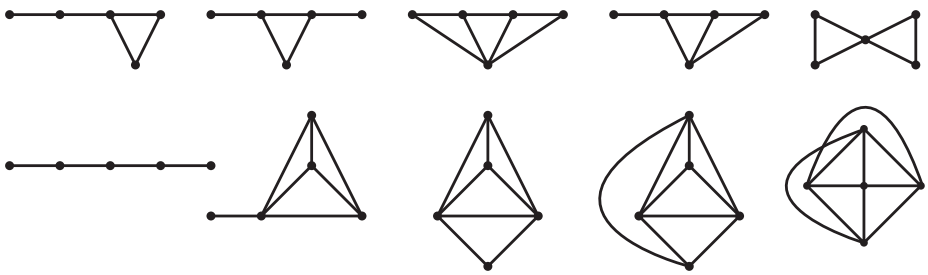
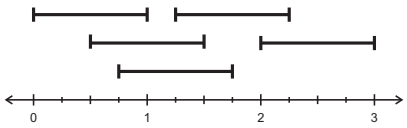


Figure 2.6: Connected unit interval graphs with five vertices.

Two sets of connected proper unit intervals can have isomorphic interval graphs, even if they have different parenthetical representations. For example, consider the following line segments



that represent the following set of proper unit intervals

$$\mathbb{I}' = \{[0, 1],\ [0.50, 1.50],\ [0.75, 1.75],\ [1.25, 2.25],\ [2, 3]\}. \tag{2.14}$$

The intervals in $\mathbb{I}'$ from (2.14) are a reflection of the intervals in $\mathbb{I}$ from (2.11). More precisely, if $[a,b] \in \mathbb{I}$ then $[3-b, 3-a] \in \mathbb{I}'$. Since this reflection does not change the pairs of intervals that intersect, then $\mathbb{I}$ and $\mathbb{I}'$ are both interval representation for the graph in Figure 2.5. On the other hand, $\text{◊}(\mathbb{I}) \neq \text{◊}(\mathbb{I}')$ since

$$\text{◊}(\mathbb{I}') = 1110100100. \tag{2.15}$$

Since the reflection causes the leftmost interval to become the rightmost interval, and causes the beginning and ending of each interval to change, then $\text{◊}(\mathbb{I}')$ in (2.15) can be obtained by reflecting $\text{◊}(\mathbb{I})$ in (2.12) and interchanging the 0s and 1s. To formalize this operation, suppose $\boldsymbol{s}$ is a binary string of length n. Then, $\overline{\boldsymbol{s}}$ is defined to be

$$\overline{\boldsymbol{s}} = \overline{s_n} \cdot \overline{s_{n-1}} \cdot \cdots \cdot \overline{s_1}.$$

The previous discussion has shown that if $\mathbb{I}$ and $\mathbb{I}'$ contain n proper unit intervals, then they will have isomorphic interval graphs whenever

$$\text{◊}(\mathbb{I}) \in \{\text{◊}(\mathbb{I}'), \overline{\text{◊}(\mathbb{I}')}\}. \tag{2.16}$$

On the other hand, it is also true that if (2.16) does not hold then $\mathbb{I}$ and $\mathbb{I}'$ will have non-isomorphic interval graphs. This leads to the following language definition. Within the definition, the *canonical* parenthetical representation of each $\mathbb{I}$ is chosen to be lexicographically larger of $\text{◊}(\mathbb{I})$ and $\overline{\text{◊}(\mathbb{I})}$.

Definition 2.3.10 (Unit Interval Graphs)**.**

$$\mathbf{I}(n) = \{1\boldsymbol{t}0 \mid \boldsymbol{t} \in \mathbf{P}(n-1) \text{ and } \boldsymbol{t} \geq \overline{\boldsymbol{t}}\}.$$

In other words, the above language contains the canonical parenthetical representation for every connected unit interval graph on n vertices.

Now it is proven that this language is a bubble language.

Theorem 2.3.11 (Bubble Language for Proper Interval Graphs)**.** *The language $\mathbf{I}(n)$ is a bubble language.*

Proof. Let $\mathbf{L} = \mathbf{I}(\frac{n}{2})$ for even n. Suppose $\boldsymbol{s} \in \mathbf{L}$ and $k = |\llcorner(\boldsymbol{s})| < n$. Let $\boldsymbol{r} = \overleftarrow{\mathsf{shift}}(\boldsymbol{s}, k+1, k)$. By Theorem 2.2.4, the proof of this theorem is completed by showing that $\boldsymbol{r} \in \mathbf{L}$. Since $\mathbf{P}(\frac{n}{2})$ is a bubble language by Theorem 2.3.9 and Remark 2.3.7, then $\boldsymbol{r}$ is also a balanced parentheses string. That is, $\boldsymbol{r} \in \mathbf{P}(\frac{n}{2})$. Therefore, from Definition

2.3.10 the proof of this theorem is completed by showing that $\boldsymbol{r} \geq \overline{\boldsymbol{r}}$. There are two cases to consider, depending on whether there are two or more 10 substrings within $\boldsymbol{s}$.

In the first case, $\boldsymbol{s}$ contains two 10 substrings. Therefore, $\boldsymbol{s} = 1^a0^b1^c0^d$ with $a, b, c, d > 0$. Then,

$$\boldsymbol{r} = 1^a0^{b-1}101^{c-1}0^d.$$

Since $\boldsymbol{s} \in \mathbf{L}$ then $\boldsymbol{s} \geq \overline{\boldsymbol{s}}$, and so $a \geq d$. There are four subcases to consider depending on the relationship between a and d, and the value of b. In the first subcase $a = d$. Therefore, $b = c$. (This is because $a + c = b + d = n$.) In this subcase

$$\begin{aligned}\boldsymbol{r} &= 1^a0^{b-1}101^{b-1}0^a \\ &= \overline{\boldsymbol{r}}\end{aligned}$$

and so $\boldsymbol{r} \geq \overline{\boldsymbol{r}}$ as desired. In the second and third subcases $a = d+1$. Therefore, $b = c+1$. In the third subcase $b = 2$. In this subcase

$$\begin{aligned}\boldsymbol{r} &= 1^a0^{b-1}101^{b-2}0^{a-1} \\ &= 1^a010^a \\ &= \overline{\boldsymbol{r}}\end{aligned}$$

and so $\boldsymbol{r} \geq \overline{\boldsymbol{r}}$ as desired. In the third subcase $b > 2$. In this subcase

$$\begin{aligned}\boldsymbol{r} &= 1^a0^{b-1}101^{b-2}0^{a-1} \\ &> 1^{a-1}0^{b-2}101^{b-1}0^a \\ &= \overline{\boldsymbol{r}}\end{aligned}$$

and so $\boldsymbol{r} \geq \overline{\boldsymbol{r}}$ as desired. In the fourth subcase $a \geq d + 2$. In this subcase

$$\begin{aligned}\boldsymbol{r} &= 1^a0^{b-1}101^{c-1}0^d \\ &> 1^d0^{c-1}101^{b-1}0^a \\ &= \overline{\boldsymbol{r}}\end{aligned}$$

and so $\boldsymbol{r} \geq \overline{\boldsymbol{r}}$ as desired.

In the second case, $\boldsymbol{s}$ contains at least three 10 substrings. Therefore, $\boldsymbol{s} =$

$1^a0^b1\boldsymbol{z}01^c0^d$ with $a, b, c, d > 0$ and $\boldsymbol{z}$ being a binary string. Then,

$$\begin{aligned}\boldsymbol{r} &= 1^a0^{b-1}10\boldsymbol{z}01^c0^d \\ &> 1^d0^c1\overline{\boldsymbol{z}}101^{b-1}0^a \\ &= \overline{\boldsymbol{r}}\end{aligned}$$

where the underlined prefix and suffix imply the inequality, and so $\boldsymbol{r} \geq \overline{\boldsymbol{r}}$ as desired. (To explain the inequality, notice that since $\boldsymbol{s} \in \mathbf{L}$ then $\boldsymbol{s} \geq \overline{\boldsymbol{s}}$. Therefore, $a \geq d$, and if $a = d$ then $b \leq c$.) □

2.3.5 Ordered Trees with Fixed Branching Sequence

A *tree* is an acyclic graph. *Rooted trees* are trees in which one of the nodes is specified as the *root*. In every pair of adjacent nodes in a rooted tree, the *parent* is the node closest to the root, and the *child* is the node furthest from the root. Every node has a unique parent except for the root which has no parent. The *out-degree* of a node is its number of children. An *ordered tree* is a rooted tree in which the children of each node are ordered. When drawing rooted trees the root appears at the top and the parents are drawn above their children. When drawing ordered trees the children of each node appear from left to right based on their order. Typically, the *branching sequence* of an ordered tree is defined as a non-increasing sequence containing the out-degree of every node. In this thesis the branching sequence is viewed as the multiset of out-degrees of every node.

A fixed-content language can be formed by considering all possible trees that have a given branching sequence. In particular, each tree is differentiated by its unique *pre-order branching traversal*, which is a string containing the out-degree of every node as encountered within a pre-order traversal, with the caveat that the last out-degree is omitted. (Since the last visited node in a pre-order traversal of a tree must be a leaf, the omitted out-degree is always 0.) For example,

$$\{31100, 31010, 31001, 30110, 30101, 30011, 13100, 13010, 13001, 11300\} \tag{2.17}$$

contains every pre-order branching traversal over the multiset $\{0, 0, 1, 1, 3\}$. From the previous discussion, the strings in this language are in one-to-one correspondence with the ordered trees containing one node with out-degree 3, two nodes with out-degree 1, and three leaf nodes with out-degree 0. These ordered trees were given in Figure 1.4 on page 18.

In general, a string $\boldsymbol{s}$ of non-negative integers is the pre-order branching traversal of an ordered tree if and only if the following condition holds (see Ruskey [62]):

$$\Sigma(\boldsymbol{s}) = |\boldsymbol{s}| \text{ and } \boldsymbol{s} = \boldsymbol{p} \cdot \boldsymbol{z} \text{ implies } \Sigma(\boldsymbol{p}) \geq |\boldsymbol{p}|.$$

Essentially this condition ensures that the tree has the correct number of leaves. (More generally, Havel [32], Erdös and Gallai [19], and Hakimi [31] characterized the existence of graphs with a given *degree sequence*. Within a rooted tree, the degree of the root equals its number of children and the degree of every other node is one less than its number of children. Alternate representations of ordered trees have also been considered. For example, Korsh-LaFollette [50] omit the 0s from the pre-order branching traversal and then use an auxiliary zero sequence to differentiate the ordered trees. Wu-Chang-Wang [103] also omit the 0s from the pre-order branching traversal, and additionally fix the order of the non-zero entries. Thus, the trees represented in (2.17) would be split into three different sets depending on the relative order of 1,1,3. An auxiliary right-distance sequence is then used to differentiate the ordered trees in this finer set.)

Definition 2.3.12 (Ordered trees with fixed branching sequence)**.** *The language of* ordered trees with branching sequence $\mathbb{M}$ *with* $\Sigma(\mathbb{M}) = m$ *is*

$$\mathbf{T}(\mathbb{M}) = \{\boldsymbol{s} \in \Pi(\mathbb{M}) \mid \text{ if } \boldsymbol{s} = \boldsymbol{p} \cdot \boldsymbol{z} \text{ then } \Sigma(\boldsymbol{p}) \geq |\boldsymbol{p}|\}.$$

In other words, $\mathbf{T}(\mathbb{M})$ *contains every permutation of* $\mathbb{M}$ *in which no prefix has a smaller sum than length. Equivalently,* $\mathbf{T}(\mathbb{M})$ *contains every pre-order branching traversal over* $\mathbb{M}$*.*

Theorem 2.3.13 (Ordered trees with fixed branching sequences are a bubble language)**.** *Suppose* $\mathbb{M}$ *is a multiset of non-negative integers with* $\Sigma(\mathbb{M}) = n$*. Then,* $\mathbf{T}(\mathbb{M})$ *is a bubble language.*

Proof. Suppose $\boldsymbol{s} \in \mathbf{T}(\mathbb{M})$ and $\boldsymbol{s} \neq \lrcorner(\boldsymbol{s})$. Let

$$\boldsymbol{r} = \overleftarrow{\mathsf{bubble}}(\boldsymbol{s}, |\lrcorner(\boldsymbol{s})| + 1).$$

Notice that $\boldsymbol{r}$ is obtained from $\boldsymbol{s}$ by shifting a symbol to the left of a single smaller symbol. Therefore, the sum of every prefix of $\boldsymbol{r}$ is at least as large as the sum of the same length prefix of $\boldsymbol{s}$. Since the sum of every prefix of $\boldsymbol{s}$ is at least as large as its length, then the same property holds for $\boldsymbol{r}$. These two facts explain the following

inequalities for all j within $1 \le j \le |\boldsymbol{s}|$:

$$\Sigma(r_1 r_2 \cdots r_j) \ge \Sigma(s_1 s_2 \cdots s_j) \ge j.$$

Therefore, $\boldsymbol{r} \in \mathbf{T}(\mathbb{M})$, proving (2.1).

To prove the other condition within Definition 2.2.1 suppose $\boldsymbol{s} \in \mathbf{T}(\mathbb{M})$ and i is within

$$|\text{🌡}(\boldsymbol{s})| < i \le |\text{⌐L}(\boldsymbol{s})|.$$

Let

$$\boldsymbol{r} = \overleftarrow{\mathsf{bubble}}(\boldsymbol{s}, i) = \overleftarrow{\mathsf{shift}}(\boldsymbol{s}, i, h). \tag{2.18}$$

This time $\boldsymbol{r}$ is formed from $\boldsymbol{s}$ by shifting a symbol to the left of a single larger symbol. In particular, the sum of every prefix of $\boldsymbol{r}$ is equal to the sum of the same length prefix of $\boldsymbol{s}$, except for the prefix of length h. Therefore, for all j within $1 \le j < h$ and $h + 1 \le j \le |\boldsymbol{s}|$

$$\Sigma(r_1 r_2 \cdots r_j) = \Sigma(s_1 s_2 \cdots s_j) \ge j.$$

Therefore, in order to show that $\boldsymbol{r} \in \mathbf{T}(\mathbb{M})$ it remains only to show that

$$\Sigma(r_1 r_2 \cdots r_h) \ge h. \tag{2.19}$$

Before proving this result it is useful to point out that (2.18) implies

$$r_1 r_2 \cdots r_h = s_1 s_2 \cdots s_{h-1} s_i$$

and in particular $r_h = s_i$.

In order to prove (2.19) it is necessary to collect a number of small facts:

- $r_1 r_2 \cdots r_h$ is non-increasing,
- if $r_h = 0$ then $h \ge 2$,
- if $r_h = 0$ then $r_1 \ge 2$, and
- if $h \ge 2$ then $r_{h-1} > r_h$.

To see how these facts prove (2.19), first notice that if $r_1 r_2 \cdots r_h$ is non-increasing and $r_h > 0$ then $r_1 r_2 \cdots r_h$ contains only positive integers and so (2.19) must be true. On

the other hand, if $r_h = 0$ then the above facts imply that

$$\begin{aligned}\Sigma(r_1 r_2 \cdots r_h) &= r_1 + \Sigma(r_2 r_3 \cdots r_{h-1}) + r_h \\ &\geq 2 + \Sigma(r_2 r_3 \cdots r_{h-1}) + 0 \\ &\geq 2 + h - 2 + 0 \\ &= h\end{aligned}$$

which again proves (2.19). Therefore, in order to complete the proof it remains only to prove the above facts. First, since $i \leq |\lceil\!\llcorner(\boldsymbol{s})|$ then $s_1 s_2 \cdots s_i$ is non-increasing. Therefore, $r_1 r_2 \cdots r_h$ is also non-increasing.

Second, it cannot be the case that $r_h = 0$ and $h = 1$. To see why, notice that $s_1 > 0$ since $\boldsymbol{s} \in \mathbf{T}(\mathbb{M})$. Therefore, if $r_h = 0$ and $h = 1$ then it would have to be the case that $i = 2$. In other words, $\boldsymbol{r}$ would have been formed from $\boldsymbol{s}$ by bubble left-shifting its second symbol (equal to 0) past its first symbol (not equal to 0). That is,

$$s_1 \cdot s_2 = s_1 \cdot s_i = s_1 \cdot 0.$$

However, this is not possible since $i > |\rceil\!\bullet(\boldsymbol{s})|$ and certainly the above prefix is frozen.

Third, it cannot be the case that $r_h = 0$ and $r_1 \leq 1$. Otherwise $\boldsymbol{r}$ would have a non-increasing prefix containing only 1s and at least one 0. Due to (2.19) this would imply that $\boldsymbol{s}$ also has a non-increasing prefix containing only 1s and at least one 0. However, this is not possible since $\boldsymbol{s} \in \mathbf{T}(\mathbb{M})$ and such a prefix would have a smaller sum than length.

Fourth, it cannot be the case that $h \geq 2$ and $r_{h-1} \leq r_h$. This is due to the fact that $r_1 r_2 \cdots r_h$ is non-increasing (which implies $r_{h-1} \geq r_h$) and the fact that r_h was bubble left-shifted within a non-increasing prefix (which implies $r_{h-1} \neq r_h$). □

Further extensions of $\mathbf{T}(\mathbb{M})$ are discussed in Chapter 5.

2.3.6 Schröder and Motzkin Paths

Balanced parentheses can also be viewed as paths inside an h by h grid. If '(' is considered to be *up* and ')' is considered to be *right*, then the balanced parentheses of length $2h$ are precisely the *Catalan paths* that travel from co-ordinate $(0, 0)$ to co-ordinate (h, h) with the property that the path never goes below the diagonal line between these two co-ordinates. Figure 2.7 contains the Catalan paths for $h = 3$. Two natural variations of Catalan paths include Schröder paths and Motzkin paths. Each

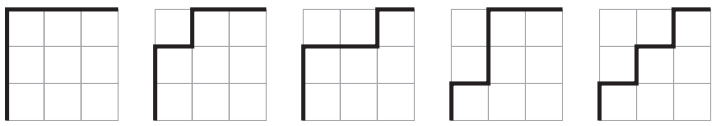


Figure 2.7: Catalan paths from $(0,0)$ to $(3,3)$.

type of path allows for a particular *non-Catalan move*. The difference between the two is that the non-Catalan move in Schröder paths covers twice as much distance as the non-Catalan move in Motzkin paths.

Formally, a *Schröder path* is a Catalan path that also allows for *diagonal* moves that proceed up one position and to the right one position. Schröder paths have at least 19 different combinatorial interpretations [86]. There are 22 Schröder paths for $h = 3$ (including the five Catalan paths in Figure 2.7) and Figure 2.8 contains those with at least one diagonal move. By instead considering '(' to be *north-east* and ')'

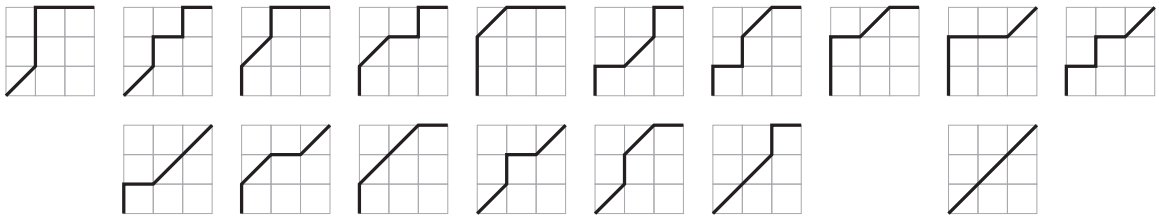


Figure 2.8: Schröder paths from $(0,0)$ to $(3,3)$ with at least one diagonal move.

to be *south-east* then balanced parentheses can be viewed as *rotated Catalan paths* from $(0,0)$ to $(2h,0)$ with the property that the path never goes below the horizontal line between these two co-ordinates. Figure 2.9 contains the rotated paths for $h = 3$. A *Motzkin path* is a rotated Catalan path that also allows for an even number of *east*

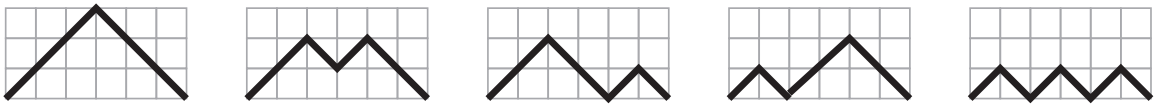


Figure 2.9: Rotated Catalan paths from $(0,0)$ to $(6,0)$.

moves. Motzkin paths have at least 13 different combinatorial interpretations [86][2]. There are 9 Motzkin paths for $h = 4$ as seen in Figure 2.10. The paths in Figures 2.8 and 2.10 are grouped by their number of non-Catalan moves. In general, Schröder

[2] *Coincidentally*, 'S' and 'M' are respectively the 22nd and 13th letter in English.

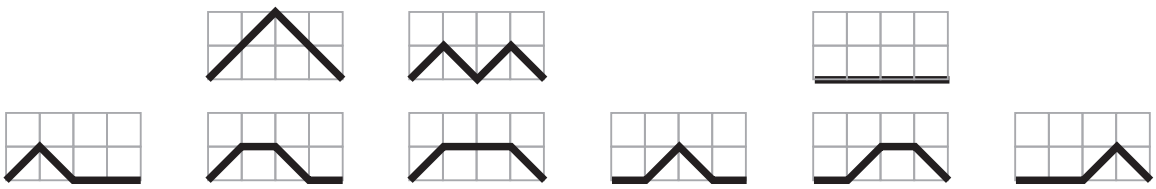


Figure 2.10: Motzkin paths from $(0,0)$ to $(4,0)$.

paths and Motzkin paths with a fixed length and a fixed number of non-Catalan moves are respectively *restricted Schröder paths* and *restricted Motzkin paths*.

To represent Schroöder and Motzkin paths as strings, the symbol 2 is used for the Catalan move up (northeast), the symbol 1 is used for the non-Catalan move diagonal (east), and the symbol 0 is used for the Catalan right (southeast). For example, the Schröder paths in Figure 2.8 are respectively

$$\begin{aligned}\mathbf{S}(3) = \{&222000, 220200, 220020, 202200, 202020, 12200, 12020, 21200, 21020, 22100,\\ &20120, 20210, 22010, 22001, 20201, 2011, 2101, 2110, 1201, 1210, 1120, 111\}.\end{aligned}$$

Similarly, the Motzkin paths in Figure 2.8 are respectively

$$\mathbf{M}(2) = \{2200, 2020, 2011, 2101, 2110, 1201, 1210, 1120, 1111\}.$$

In both cases, notice that the 1 symbols are free to move anywhere within a given string (see Figure 2.4). To formalize the definition of these languages, let $\#_i(\boldsymbol{s})$ be the number of copies of symbol i within string $\boldsymbol{s}$.

Definition 2.3.14 (Schröder and Motzkin paths)**.** *The language of* Schröder paths *from* $(0,0)$ *to* (h,h) *is*

$$\mathbf{S}(h) = \{\boldsymbol{s} \mid \boldsymbol{s} \in \Pi(\mathsf{content}(2^i 1^k 0^i)) \text{ where } 2i{+}k = h \text{ and if } \boldsymbol{s} = \boldsymbol{pz} \text{ then } \#_2(\boldsymbol{p}) \geq \#_0(\boldsymbol{p})\}.$$

The language of Motizkin paths *from* $(0,0)$ *to* $(2h,0)$ *is*

$$\mathbf{M}(h) = \{\boldsymbol{s} \mid \boldsymbol{s} \in \Pi(\mathsf{content}(2^i 1^{2k} 0^i)) \text{ where } 2i{+}2k = h \text{ and if } \boldsymbol{s} = \boldsymbol{pz} \text{ then } \#_2(\boldsymbol{p}) \geq \#_0(\boldsymbol{p})\}.$$

The *restricted* forms of these languages are obtained by fixing the number of non-Catalan moves. In particular, let $\mathbf{S}_k(h)$ be the set of strings representing the *restricted Schroöder paths* from $(0,0)$ to (h,h) using precisely k diagonal moves. That is, $\mathbf{S}_k(h)$ is the fixed-content subset of $\mathbf{S}_k(h)$ containing strings with precisely

k copies of 1. Similarly, let $\mathbf{M}_{2k}(h)$ be the set of strings representing the *restricted Motzkin paths* from $(0,0)$ to $(2h,0)$ using precisely $2k$ east moves. That is, $\mathbf{M}_{2k}(h)$ is the fixed-content subset of $\mathbf{M}_{2k}(h)$ containing strings with precisely $2k$ copies of 1. Interestingly, these restricted languages are essentially the same. For example, the expressions earlier in this section show that

$$\mathbf{S}_2(3) = \mathbf{M}_2(2) = \{2011, 2101, 2110, 1201, 1210, 1120\}.$$

(The only difference between the restricted languages is that $\mathbf{S}_k(h)$ allows for an odd number of 1s.) Furthermore, the above language is identical to the language $\mathbf{T}(\{0,1,1,2\})$ (ordered trees). In general, the following equalities follow directly from Definition 2.3.12 and the restricted forms of Definition 2.3.14

$$\mathbf{S}_k(h) = \mathbf{T}(\mathsf{content}(2^{h-k}1^k0^{h-k})) \qquad \mathbf{M}_{2k}(h) = \mathbf{T}(\mathsf{content}(2^{h-k}1^{2k}0^{h-k}))$$
$$\mathbf{S}(h) = \mathbf{S}_0(h) \cup \mathbf{S}_1(h) \cup \cdots \cup \mathbf{S}_h(h) \qquad \mathbf{M}(h) = \mathbf{M}_0(h) \cup \mathbf{M}_2(h) \cup \cdots \cup \mathbf{M}_{2h}(h).$$

The top lines respectively prove that restricted Schröder and Motzkin paths are bubble languages, as stated by the following theorem. (Furthermore, $\mathbf{T}(\mathsf{content}(2^h0^h))$ is identical to $\mathbf{P}(h)$ after interchanging 2s and 1s. Therefore, restricted Schroöder and Motzkin paths generalize balanced parentheses by setting $k = 0$.)

Theorem 2.3.15 (Restricted Schröder and Motzkin paths are a bubble languages)**.** *$\mathbf{S}_k(h)$ is a bubble language whenever $0 \le k \le h$. Similarly, $\mathbf{M}_k(h)$ is a bubble language whenever $0 \le 2k \le h$.*

2.3.7 Necklaces, Lyndon Words, and Bracelets

It is often useful to consider strings in equivalence classes based on their rotations. This section discusses several variations on this idea including necklaces, Lyndon words, and bracelets. A number of basic string concepts will be introduced including lexicographic comparisons and rotations. After necklaces and Lyndon words are defined, two basic properties will be discussed. Then, it will then be shown that necklaces and Lyndon words are bubble languages. To prove these two results a simple necessary condition for necklaces is given in Remark 2.3.16, and a simple sufficient condition for Lyndon words is given in Remark 2.3.17. These necessary and sufficient conditions involve partitioning strings into substrings known as peaks. Bracelets in general are not bubble languages, and this fact is proven by counterexample. Section

2.4.1 discusses pre-necklaces, and closure properties of bubble languages are used to prove that pre-necklaces are bubble languages.

Given two strings, $\boldsymbol{s} = s_1 s_2 \cdots s_n$ and $\boldsymbol{t} = t_1 t_2 \cdots t_h$, of possibly unequal length, say that $\boldsymbol{s}$ is *lexicographically larger* than $\boldsymbol{t}$ and denote this by $\boldsymbol{s} > \boldsymbol{t}$ if one of following mutually exclusive conditions hold

1. there exists j such that $s_j > t_j$ and $s_i = t_i$ for all $1 \leq i < j$
2. $n < h$ and $s_i = t_i$ for all $1 \leq i \leq n$.

Within the first condition, s_j and t_j are known as the *leftmost differing symbols* between $\boldsymbol{s}$ and $\boldsymbol{t}$, and $\boldsymbol{s} > \boldsymbol{t}$ because the leftmost differing symbol is larger in $\boldsymbol{s}$. On the other hand, the second condition states that $\boldsymbol{s} > \boldsymbol{t}$ when $\boldsymbol{s}$ is a strict prefix of $\boldsymbol{t}$. The second condition is somewhat non-standard, but its merits will become apparent later in this section when strings are partitioned into substrings known as peaks. If $\boldsymbol{s}$ is lexicographically larger than $\boldsymbol{t}$, or $\boldsymbol{s}$ is equal to $\boldsymbol{t}$, then $\boldsymbol{s} \geq \boldsymbol{t}$ is written.

Given a string $\boldsymbol{s} = s_1 s_2 \cdots s_n$, the *rotation of* $\boldsymbol{s}$ *beginning at its* i*th symbol* is denoted by $\circlearrowleft_i(\boldsymbol{s})$. That is,

$$\circlearrowleft_i(\boldsymbol{s}) = s_i s_{i+1} \cdots s_n s_1 s_2 \cdots s_{i-1}.$$

It is also helpful to adopt the convention that $\circlearrowleft_0(\boldsymbol{s}) = \circlearrowleft_n(\boldsymbol{s})$ when $|\boldsymbol{s}| = n$, since this simplifies the results in Section 4.2.3. The *rotation set* of a string $\boldsymbol{s}$ is denoted by $\circlearrowleft(\boldsymbol{s})$ and is a set containing every rotation of the string. For example, if $\boldsymbol{s} = 221313$ and $\boldsymbol{t} = 213213$ then

$$\circlearrowleft(\boldsymbol{s}) = \{223131, 231312, 313122, 131223, 312231, 122313\} \tag{2.20}$$

$$\circlearrowleft(\boldsymbol{t}) = \{213213, 132132, 321321\}. \tag{2.21}$$

Any $\boldsymbol{s} \in \circlearrowleft(\boldsymbol{s})$ is known as a *representative* of $\circlearrowleft(\boldsymbol{s})$. (Notice that rotation set membership is an equivalence relation because $\boldsymbol{s} \in \circlearrowleft(\boldsymbol{t}) \implies \boldsymbol{t} \in \circlearrowleft(\boldsymbol{s})$, and $(\boldsymbol{r} \in \circlearrowleft(\boldsymbol{s})$ and $\boldsymbol{s} \in \circlearrowleft(\boldsymbol{t})) \implies \boldsymbol{r} \in \circlearrowleft(\boldsymbol{t}))$.) In particular, a *necklace* is a string that is lexicographically largest within its rotation set. With regards to (2.20) and (2.21), only 313122 and 321321 are necklaces. More thoroughly,

$$332211, 332121, 332112, 331221, 331212, 331122, 323211, 323121,$$
$$323112, 322311, 322131, 321321, 321312, 321231, 313122, 312312$$

are the necklaces over $\{1, 1, 2, 2, 3, 3\}$. With respect to Figure 1.3 on page 17, the above strings are lexicographically largest with respect to clockwise ordering and the

mapping ● ↔ 1, ◍ ↔ 2, ○ ↔ 3. (The literature most often examines necklaces and their variations using lexicographically smallest strings.)

A string $\boldsymbol{s}$ is *aperiodic* if $|\circlearrowleft(\boldsymbol{s})| = |\boldsymbol{s}|$. (Notice that $|\circlearrowleft(\boldsymbol{s})| \leq |\boldsymbol{s}|$ always holds, and that $|\circlearrowleft(\boldsymbol{s})| < |\boldsymbol{s}|$ only when $\boldsymbol{s}$ is periodic in the sense that it can be expressed as $\boldsymbol{s} = \boldsymbol{r}^i$ for some $i > 1$.) A *Lyndon word* is an aperiodic necklace. With regards to (2.20) and (2.21), only 313122 is a Lyndon word. More thoroughly, 312312 and 321321 are the only necklaces over $\{1, 1, 2, 2, 3, 3\}$ that are not Lyndon words.

The *reverse* of $\boldsymbol{s}$ is denoted $\mathsf{reverse}(\boldsymbol{s})$. That is, $\mathsf{reverse}(s_1 s_2 \cdots s_n) = s_n s_{n-1} \cdots s_1$. A *bracelet* is a string $\boldsymbol{s}$ that is largest in $\circlearrowleft(\boldsymbol{s}) \cup \circlearrowleft(\mathsf{reverse}(\boldsymbol{s}))$. With regards to (2.20) and (2.21), only 321321 is a bracelet. More thoroughly,

$$332211, 332121, 332112, 331221, 323211, 323121, 322311, 322131, 321321, 321312, 321231$$

are the bracelets over $\{1, 1, 2, 2, 3, 3\}$ arising from Figure 1.3.

Fixed-content languages can be formed using these three concepts by considering all possible strings containing a given multiset of symbols $\mathbb{M}$. In particular, the *necklace language* is denoted by $\mathbf{N}(\mathbb{M})$, the *Lyndon language* is denoted by $\mathbf{N}^-(\mathbb{M})$, and the *bracelet language* is denoted by $\mathbf{R}(\mathbb{M})$. For example, when $\mathbb{M} = \{1, 1, 1, 0, 0, 0\}$ then

$$\begin{aligned}\mathbf{N}(\mathbb{M}) &= \{111000, 110100, 110010, 101010\}\\ \mathbf{N}^-(\mathbb{M}) &= \{111000, 110100, 110010\}\\ \mathbf{R}(\mathbb{M}) &= \{111000, 110100, 101010\}.\end{aligned}$$

Notice that the above bracelet language is not a bubble language. In particular, $1010 \cdot 10$ is a bracelet but $1100 \cdot 10$ is not. Therefore, bracelets in general do not satisfy the increasing prefix property, which is necessary for being a bubble language. To further illustrate the necklace and Lyndon languages, when $\mathbb{M} = \{3, 3, 2, 2, 1, 1\}$, note that

$$\begin{aligned}\mathbf{N}(\mathbb{M}) = \{&312312, 313122, 321231, 321312, 321321, 322131, 322311, 323112,\\ &323121, 323211, 331122, 331212, 331221, 332112, 332121, 332211\},\ \text{and}\\ \mathbf{N}^-(\mathbb{M}) = \mathbf{N}(\mathbb{M}) &\backslash \{312312, 321321\}.\end{aligned}$$

Notice that in both of these last two examples $\mathbf{N}^-(\mathbb{M})$ is a strict subset of $\mathbf{N}(\mathbb{M})$. This is due to the fact that the symbol multiplicities in $\{3, 3, 2, 2, 1, 1\}$ have a common

factor larger than one. When there is no such common factor then $\mathbf{N}^{-}(\mathbb{M})$ and $\mathbf{N}(\mathbb{M})$ are identical.

When determining whether or not a string is a necklace or Lyndon word, a candidate string $\boldsymbol{s}$ must have its largest symbol in its first position. That is, $s_1 = d_m$. Assuming this fact holds, then $\boldsymbol{s}$ can be partitioned uniquely into peaks. A *peak* is a substring beginning with consecutive symbols equal to d_m, and terminating immediately prior to the next symbol equal to d_m. More formally, a peak of $\boldsymbol{s}$ is a substring $s_i s_{i+1} \cdots s_j$ that satisfies the following conditions

- $s_i = d_m$,
- $i = 1$ or $s_{i-1} < d_m$,
- $j = |\boldsymbol{s}|$ or $s_{j+1} = d_m$, and
- if $s_h = d_m$ and $i < h \leq j$ then $s_{h-1} = d_m$.

The following example shows how a string is partitioned into its peaks

$$\boldsymbol{s} = \underbrace{442231}_{\blacktriangle(\boldsymbol{s},1)} \cdot \underbrace{44311}_{\blacktriangle(\boldsymbol{s},2)} \cdot \underbrace{4422}_{\blacktriangle(\boldsymbol{s},3)} \cdot \underbrace{413}_{\blacktriangle(\boldsymbol{s},4)} . \tag{2.22}$$

As above, the ith peak from the left in $\boldsymbol{s}$ is denoted by $\blacktriangle(\boldsymbol{s}, i)$ and is referred to as the *ith peak of $\boldsymbol{s}$*.

By using peaks and $>$, a simple necessary condition for a string to be a necklace can be stated: If $\boldsymbol{s}$ is a necklace, then no peak can be lexicographically larger than its first peak. To illustrate this condition, notice that the string in (2.22) is not a necklace because of its rotation beginning at position 7

$$\circlearrowleft_7(\boldsymbol{s}) = 443114422413442231 > 442231443114422413 = \boldsymbol{s}.$$

This inequality is due to the fact that $\circlearrowleft_7(\boldsymbol{s})$ begins with the second peak of $\boldsymbol{s}$, and the fact that the second peak of $\boldsymbol{s}$ is lexicographically larger than the first peak in $\boldsymbol{s}$. That is,

$$\blacktriangle(\boldsymbol{s}, 2) = 44311 > 442231 = \blacktriangle(\boldsymbol{s}, 1).$$

Notice that the above inequality follows from the first condition given for $>$. In particular, $3 > 2$ and these two symbols are the leftmost differing symbol between 44$\underline{3}$11 and 44$\underline{2}$231 (as underlined). Therefore, $\circlearrowleft_7(\boldsymbol{s}) > \boldsymbol{s}$ because their leftmost

differing symbols equal the leftmost differing symbols in $\blacktriangle(\boldsymbol{s}, 2)$ and $\blacktriangle(\boldsymbol{s}, 1)$. The string in (2.22) is also not a necklace because of its rotation beginning at position 12

$$\circlearrowleft_{12}(\boldsymbol{s}) = 442241344223144311 > 442231443114422413 = \boldsymbol{s}.$$

This inequality is due to the fact that $\circlearrowleft_{12}(\boldsymbol{s})$ begins with the third peak of $\boldsymbol{s}$, and the fact that the third peak of $\boldsymbol{s}$ is lexicographically larger than the first peak in $\boldsymbol{s}$. That is,

$$\blacktriangle(\boldsymbol{s}, 3) = 4422 > 442231 = \blacktriangle(\boldsymbol{s}, 1).$$

Notice that the above inequality follows from the second condition given for $>$. In particular, 4422 is a strict prefix of 442231. Therefore, $\circlearrowleft_{12}(\boldsymbol{s}) > \boldsymbol{s}$ because the leftmost differing symbol between these two strings is guaranteed to be larger within $\circlearrowleft_{12}(\boldsymbol{s})$. (In particular, the symbol following any peak must equal the largest symbol d_m.) This necessary condition for necklaces is stated in the following remark.

Remark 2.3.16 (Necessary condition for necklaces)**.** *Suppose $\boldsymbol{s} \in \mathbf{N}(\mathbb{M})$. Then, for all i*

$$\blacktriangle(\boldsymbol{s}, 1) \geq \blacktriangle(\boldsymbol{s}, i).$$

In other words, if $\boldsymbol{s}$ is a necklace then none of its peaks are lexicographically larger than its first peak.

Notice that the condition in Remark 2.3.16 is not sufficient for a string to be a necklace. For example, no peak in 11010011010 is lexicographically larger than its first, but it is not a necklace due to its rotation starting at position 7.

By using peaks and $>$, a simple sufficient condition for a string to be a Lyndon word can be stated: If the first peak in $\boldsymbol{s}$ is lexicographically larger than all of its other peaks, then $\boldsymbol{s}$ is a Lyndon word. To illustrate this condition, consider the following Lyndon word and its partition into peaks

$$\boldsymbol{s} = \underbrace{442231}_{\blacktriangle(\boldsymbol{s},1)} \cdot \underbrace{44113}_{\blacktriangle(\boldsymbol{s},2)} \cdot \underbrace{4422313}_{\blacktriangle(\boldsymbol{s},3)}. \tag{2.23}$$

In particular, $\boldsymbol{s}$ is lexicographically larger than its rotation beginning at position 7 due to the first condition given for $>$. Similarly, $\boldsymbol{s}$ is lexicographically larger than its rotation beginning at position 12 due to the second condition given for $>$. This sufficient condition for Lyndon words is stated in the following remark.

Remark 2.3.17 (Sufficient condition for Lyndon words)**.** *Suppose that for all* i

$$\blacktriangle(\boldsymbol{s}, 1) > \blacktriangle(\boldsymbol{s}, i).$$

Then, $\boldsymbol{s} \in \mathbf{N}^{-}(\mathbb{M})$.

Notice that the condition in Remark 2.3.17 is not necessary for a string to be a Lyndon word. For example, the first peak in 11011010 is not lexicographically larger than its second peak, but it is a Lyndon word.

By using Remarks 2.3.16 and 2.3.17 it is possible to prove that $\mathbf{N}(\mathbb{M})$ and $\mathbf{N}^{-}(\mathbb{M})$ are bubble languages by way of two lemmas. Since these results involve proving that necklaces and Lyndon words are closed under certain bubble left-shifts, it is instructive to point out that necklaces and Lyndon words are not closed under arbitrary bubble left-shifts. For example, notice that 210202011 is a Lyndon word but

$$\overleftarrow{\mathsf{bubble}}(210202011, 6) = 210\overleftarrow{202}011 = 210220011$$

is not a necklace since it fails Remark 2.3.16 due to its second peak, 220011, being lexicographically larger than its first peak, 210. Therefore, necklaces and Lyndon words are not closed under bubble left-shifts, even if the bubble left-shift moves a larger symbol past a smaller symbol.

Lemma 2.3.18 will prove that bubble left-shifting the symbol following the non-increasing prefix of a necklace always results in a Lyndon word. (Since $\mathbf{N}^{-}(\mathbb{M}) \subseteq \mathbf{N}(\mathbb{M})$ this proves that necklaces and Lyndon words satisfy the closure property stated in (2.1).) The proof involves five simple cases which are listed below using $k = |\llcorner(\boldsymbol{s})|$. Each case is accompanied by an example where $\boldsymbol{s} \in \mathbf{N}(\mathbb{M})$ and $\boldsymbol{r} = \overleftarrow{\mathsf{bubble}}(\boldsymbol{s}, k+1) \in \mathbf{N}^{-}(\mathbb{M})$ with $\mathbb{M} = \{1,1,1,1,2,2,2,2,3,3,3,3,4,4,4,4\}$. Within $\boldsymbol{s}$ and $\boldsymbol{r}$ the peaks are separated by $\cdot$ to emphasize how peaks are modified by the bubble left-shift.

Case One: $s_{k+1} < d_m$. For example,

$$\boldsymbol{s} = 44311322{\cdot}44311232$$
$$\boldsymbol{r} = 44313122{\cdot}44311232.$$

Case Two: $s_{k+1} = d_m$ and $s_{k+2} = d_m$ and $s_{k-1} < d_m$. For example,

$$\boldsymbol{s} = 44332211{\cdot}44332211$$
$$\boldsymbol{r} = 4433221{\cdot}41{\cdot}4332211.$$

Case Three: $s_{k+1} = d_m$ and $s_{k+2} = d_m$ and $s_{k-1} = d_m$. For example,

$$\boldsymbol{s} = 442{\cdot}4423333221111$$
$$\boldsymbol{r} = 4442{\cdot}423333221111.$$

Case Four: $s_{k+1} = d_m$ and $s_{k+2} < d_m$ and $s_{k-1} < d_m$. For example,

$$\boldsymbol{s} = 422{\cdot}4213{\cdot}413{\cdot}421331$$
$$\boldsymbol{r} = 42{\cdot}42213{\cdot}413{\cdot}421331.$$

Case Five: $s_{k+1} = d_m$ and $s_{k+2} < d_m$ and $s_{k-1} = d_m$. For example,

$$\boldsymbol{s} = 42{\cdot}4233332{\cdot}42{\cdot}41111$$
$$\boldsymbol{r} = 442233332{\cdot}42{\cdot}41111.$$

(Notice that $k = 1$ is not possible within a necklace since the first symbol cannot be smaller than the second.)

Lemma 2.3.18 (Necklaces and Lyndon words satisfy (2.1))**.** *Suppose* $\boldsymbol{s} \in \mathbf{N}(\mathbb{M})$ *with* $\boldsymbol{s} \neq \boldsymbol{\urcorner\!\llcorner}$ *and* $k = |\boldsymbol{\urcorner\!\llcorner}(\boldsymbol{s})|$*. Then,*

$$\overleftarrow{\mathsf{bubble}}(\boldsymbol{s}, k+1) \in \mathbf{N}^{-}(\mathbb{M}).$$

In other words, a Lyndon word results from bubble left-shifting the symbol following the non-increasing prefix of a necklace.

Proof. Let $\boldsymbol{r} = \overleftarrow{\mathsf{bubble}}(\boldsymbol{s}, k+1)$. For the first case, suppose that $s_{k+1} < d_m$. Then,

$$\blacktriangle(\boldsymbol{r}, 1) > \blacktriangle(\boldsymbol{s}, 1)$$

and $\blacktriangle(\boldsymbol{r}, h) = \blacktriangle(\boldsymbol{s}, h)$ for all $h \geq 2$. Therefore, Remark 2.3.16 implies that $\blacktriangle(\boldsymbol{r}, 1)$ is lexicographically larger than each of its peaks. Therefore, $\boldsymbol{r}$ is a Lyndon word by Remark 2.3.17. Therefore, for the remainder of the proof it can be assumed that $s_{k+1} = d_m$. This implies that $\blacktriangle(\boldsymbol{s}, 1)$ is non-increasing. Furthermore, $k + 2 \leq n$ since $\blacktriangle(\boldsymbol{s}, 2)$ must contain at least two symbols. It is also true that $s_k < d_m$. The proof is now divided into four remaining cases depending on the values of s_{k+2} and s_{k-1}.

Case Two: Suppose $s_{k+2} = d_m$ and $s_{k-1} < d_m$. Then,

$$\blacktriangle(\boldsymbol{r}, 1) \cdot s_k = \blacktriangle(\boldsymbol{s}, 1), \qquad \blacktriangle(\boldsymbol{r}, 2) = d_m s_k, \qquad d_m \cdot \blacktriangle(\boldsymbol{r}, 3) = \blacktriangle(\boldsymbol{s}, 2)$$

and $\blacktriangle(\boldsymbol{r}, h+1) = \blacktriangle(\boldsymbol{s}, h)$ for all $h \geq 3$. Therefore, Remark 2.3.16 implies that $\blacktriangle(\boldsymbol{r}, 1)$ is lexicographically larger than each of its peaks, except possibly its second peak $\blacktriangle(\boldsymbol{r}, 2) = d_m s_k$. However, $\blacktriangle(\boldsymbol{r}, 1)$ must begin with at least two copies of d_m, which implies $\blacktriangle(\boldsymbol{r}, 1) > d_m s_k$. (In particular, $s_{k+2} = d_m$ and Remark 2.3.16 imply that $\blacktriangle(\boldsymbol{s}, 1)$ begins with at least two copies of d_m, and $s_{k-1} < d_m$ implies that $\blacktriangle(\boldsymbol{s}, 1)$ ends with at least two symbols that are less than d_m. Thus, $\blacktriangle(\boldsymbol{r}, 1)$ begins with at least two copies of d_m.) Therefore, $\boldsymbol{r}$ is a Lyndon word by Remark 2.3.17.

Case Three: Suppose $s_{k+2} = d_m$ and $s_{k-1} = d_m$. Then,

$$\blacktriangle(\boldsymbol{r}, 1) = d_m \cdot \blacktriangle(\boldsymbol{s}, 1), \qquad d_m \cdot \blacktriangle(\boldsymbol{r}, 2) = \blacktriangle(\boldsymbol{s}, 1)$$

and $\blacktriangle(\boldsymbol{r}, h) = \blacktriangle(\boldsymbol{s}, h)$ for all $h \geq 3$. Therefore, Remark 2.3.16 implies that $\blacktriangle(\boldsymbol{r}, 1)$ is lexicographically larger than each of its peaks. Therefore, $\boldsymbol{r}$ is a Lyndon word by Remark 2.3.17.

Case Four: Suppose $s_{k+2} < d_m$ and $s_{k-1} < d_m$. Let $\blacktriangle(\boldsymbol{s}, 2) = d_m \cdot \boldsymbol{z}$ and $\blacktriangle(\boldsymbol{r}, 2) = d_m \cdot \boldsymbol{y}$. Then,

$$\blacktriangle(\boldsymbol{r}, 1) \cdot s_k = \blacktriangle(\boldsymbol{s}, 1), \qquad d_m \cdot s_k \cdot \boldsymbol{y} = d_m \cdot \boldsymbol{z}$$

and $\blacktriangle(\boldsymbol{r}, h) = \blacktriangle(\boldsymbol{s}, h)$ for all $h \geq 3$. Therefore, Remark 2.3.16 implies that $\blacktriangle(\boldsymbol{r}, 1)$ is lexicographically larger than each of its peaks, except possibly its second peak $\blacktriangle(\boldsymbol{r}, 2)$. Note that $\blacktriangle(\boldsymbol{r}, 1)$ begins $d_m s_2$ and $\blacktriangle(\boldsymbol{r}, 2)$ begins $d_m \cdot s_k s_{k+2}$. Since $\blacktriangle(\boldsymbol{s}, 1)$ is non-increasing then $s_2 \geq s_k$. If $s_2 > s_k$ then $\blacktriangle(\boldsymbol{r}, 1) > \blacktriangle(\boldsymbol{r}, 2)$ as desired. Otherwise, $s_2 = s_k$. Since $\blacktriangle(\boldsymbol{s}, 1)$ is non-increasing and $s_{k-1} < d_m$, then it must be that $\blacktriangle(\boldsymbol{s}, 1) = d_m \cdot s_k^h$ for some $h \geq 2$. Since $\boldsymbol{s}$ is a necklace then Remark 2.3.16 implies that $\blacktriangle(\boldsymbol{s}, 1) \geq \blacktriangle(\boldsymbol{s}, 2)$. Therefore, $s_k^h \geq \boldsymbol{z}$. Therefore, $s_k^{h-1} > s_k \cdot \boldsymbol{z}$. Therefore, $\blacktriangle(\boldsymbol{r}, 1) = d_m \cdot s_k^{h-1} > d_m \cdot s_k \cdot \boldsymbol{z} = \blacktriangle(\boldsymbol{r}, 2)$. Therefore, $\boldsymbol{r}$ is a Lyndon word by Remark 2.3.17.

Case Five: Suppose $s_{k+2} < d_m$ and $s_{k-1} = d_m$. Then,

$$\blacktriangle(\boldsymbol{r}, 1) = \overleftarrow{\mathsf{shift}}(\blacktriangle(\boldsymbol{s}, 1) \cdot \blacktriangle(\boldsymbol{s}, 2), k+1, k)$$

and $\blacktriangle(\boldsymbol{r}, h+1) = \blacktriangle(\boldsymbol{s}, h)$ for all $h \geq 1$. Therefore, Remark 2.3.16 implies that $\blacktriangle(\boldsymbol{r}, 1)$ is lexicographically larger than each of its peaks. Therefore, $\boldsymbol{r}$ is a Lyndon word by

Remark 2.3.17. □

Now it must be shown that necklaces and Lyndon words are closed under bubble left-shifting unfrozen symbols within the non-increasing prefix. To illustrate this result, consider the following string

$$\boldsymbol{s} = 44321 \cdot 43 \cdot 421.$$

When $\boldsymbol{s}$ is treated as a necklace $\bullet(\boldsymbol{s}) = 44$ since its first three symbols can be rearranged to give a necklace $4342143421 = (43421)^2$ but the first two symbols cannot. When $\boldsymbol{s}$ is treated as a Lyndon word $\bullet(\boldsymbol{s}) = 443$ since the first four symbols can be rearranged to give a Lyndon word 4423143421 but the first three symbols cannot. Since $|\curlywedge(\boldsymbol{s})| = 5$ this means that bubble left-shifting the ith symbol of $\boldsymbol{s}$ should create a new necklace for $i = 3, 4, 5$ and should create a new Lyndon word for $i = 4, 5$ according to (2.2). This can be verified below, where the result of each bubble left-shift is partitioned into peaks

$$\overleftarrow{\mathsf{bubble}}(\boldsymbol{s}, 3) = 43 \cdot 421 \cdot 43 \cdot 421, \quad \overleftarrow{\mathsf{bubble}}(\boldsymbol{s}, 4) = 44231 \cdot 43 \cdot 421, \quad \overleftarrow{\mathsf{bubble}}(\boldsymbol{s}, 5) = 44312 \cdot 43 \cdot 421.$$

Notice that the first bubble left-shift divides the first peak of $\boldsymbol{s}$ into two peaks, whereas the remaining bubble left-shifts simply rearrange the symbols in the first peak. These two cases illustrate the two cases found within the formal proof of the following lemma.

Lemma 2.3.19 (Necklaces and Lyndon words satisfy (2.2))**.** *Suppose* $\mathbf{L} = \mathbf{N}(\mathbb{M})$ *or* $\mathbf{L} = \mathbf{N}^{-}(\mathbb{M})$*, and* $\boldsymbol{s} \in \mathbf{L}$ *and* i *is within* $|\bullet(\boldsymbol{s})| < i \leq |\curlywedge(\boldsymbol{s})|$*. Then,*

$$\overleftarrow{\mathsf{bubble}}(\boldsymbol{s}, i) \in \mathbf{L}.$$

In other words, necklaces and Lyndon words are closed under bubble left-shifting any unfrozen symbol in the non-increasing prefix.

Proof. Since $i \leq |\curlywedge(\boldsymbol{s})|$ then $s_1 s_2 \cdots s_i$ is non-increasing. Let $\mathbb{M}' = \{s_1, s_2, \ldots, s_i\}$. There are two cases depending on the value of m', which denotes the number of distinct symbols in $\mathbb{M}'$. Since $i > |\bullet(\boldsymbol{s})|$, then $s_1 s_2 \cdots s_i$ can be rearranged within $\boldsymbol{s}$ to create $\boldsymbol{t} \in \mathbf{L}$ with $\boldsymbol{t} \neq \boldsymbol{s}$. Notice that the existence of $\boldsymbol{t} \in \mathbf{L}$ implies that there are at least two distinct symbols in $\mathbb{M}'$. That is, $m' \geq 2$. The proof now divides into two cases depending on whether or not $m' = 2$. Each case is completed by proving $\boldsymbol{r} = \overleftarrow{\mathsf{bubble}}(\boldsymbol{s}, i) \in \mathbf{L}$. Thus, it is assumed that $\boldsymbol{t} \neq \boldsymbol{r}$ since $\boldsymbol{t} \in \mathbf{L}$.

Suppose $m' = 2$. Therefore, $\mathbb{M}'$ contains only the distinct symbols $d'_2 > d'_1$. To simplify the following discussion, let $x = d'_1$ and $y = d'_2$ for the remainder of this case. Notice that $\boldsymbol{s}$ and $\boldsymbol{r}$ can be expressed as follows

$$\boldsymbol{s} = y^{h-1}yxx^{i-h-1}\boldsymbol{z}$$
$$\boldsymbol{r} = y^{h-1}xyx^{i-h-1}\boldsymbol{z}$$

where $\boldsymbol{z} = s_{i+1}s_{i+2}\cdots s_n$ and h satisfies $1 < h < i$. Notice that

$$\blacktriangle(\boldsymbol{r}, 1) = y^{h-1}x$$

since $y = d_m$ follows from the fact that necklaces and Lyndon words begin with the largest possible symbol. The proof of this case depends on the following

$$\blacktriangle(\boldsymbol{r}, 1) > \blacktriangle(\boldsymbol{t}, 1).$$

To see why this is true, notice that $\blacktriangle(\boldsymbol{t}, 1) \neq \blacktriangle(\boldsymbol{s}, 1)$ since equality would imply $\boldsymbol{t} = \boldsymbol{s}$. Also, $\blacktriangle(\boldsymbol{t}, 1) \neq \blacktriangle(\boldsymbol{r}, 1)$ since equality would imply $\boldsymbol{t} = \boldsymbol{r}$. Therefore, $\blacktriangle(\boldsymbol{t}, 1)$ must either contain fewer than $h - 1$ copies of y or more than one copy of x. Therefore, $\blacktriangle(\boldsymbol{r}, 1) > \blacktriangle(\boldsymbol{t}, 1)$ as claimed. Since $\boldsymbol{t} \in \mathbf{L}$ and $\blacktriangle(\boldsymbol{r}, 1) > \blacktriangle(\boldsymbol{t}, 1)$, then Remarks 2.3.16 and 2.3.17 imply that $\boldsymbol{r} \in \mathbf{L}$.

Suppose $m' > 2$. Since $s_1s_2\cdots s_i$ is non-increasing, Remark 2.1.10 implies that $s_1s_2\cdots s_i$ and $r_1r_2\cdots r_i$ are the two lexicographically largest strings over $\mathbb{M}'$. Therefore,

$$r_1r_2\cdots r_i > t_1t_2\cdots t_i.$$

To see why this is true, notice that $t_1t_2\cdots t_i \neq r_1r_2\cdots r_i$ since equality would imply $\boldsymbol{t} = \boldsymbol{r}$. Also, $t_1t_2\cdots t_i \neq s_1s_2\cdots s_i$ since equality would imply $\boldsymbol{t} = \boldsymbol{s}$. Therefore, $t_1t_2\cdots t_i$ is a string over $\mathbb{M}'$ but it is not equal to either of the two lexicographically largest strings over $\mathbb{M}'$. Therefore,

$$\blacktriangle(\boldsymbol{r}, 1) > \blacktriangle(\boldsymbol{t}, 1)$$

since $m' > 2$ implies that $\blacktriangle(\boldsymbol{r}, 1)$ begins with $r_1r_2\cdots r_i$. Since $\boldsymbol{t} \in \mathbf{L}$ and $\blacktriangle(\boldsymbol{r}, 1) > \blacktriangle(\boldsymbol{t}, 1)$, then Remarks 2.3.16 and 2.3.17 imply that $\boldsymbol{r} \in \mathbf{L}$. □

The previous two lemmas prove the following theorem.

Theorem 2.3.20 (Necklaces and Lyndon words are bubble languages)**.** *The languages* $\mathbf{N}(\mathbb{M})$ *and* $\mathbf{N}^{-}(\mathbb{M})$ *are bubble languages.*

2.4 Properties of Bubble Languages

This section discusses several properties of bubble languages. Section 2.4.1 shows that bubble languages are closed under the operations of union, intersection, and quotients. The first two of these operations are useful for considering the overall breadth of languages contained in the bubble language hierarchy, but the results are otherwise unused throughout the thesis. The third operation is used in several places in the remainder of the thesis, including Section 2.4.3 in this Chapter. Furthermore, this operation is used to prove that pre-necklaces can be represented by bubble languages.

Section 2.4.2 informally discusses two different notions of a maximal left-shift, and shows that these two concepts coincide when shifting certain symbols within bubble languages. This discussion prepares the reader for the formal definition of a greedy left-shift found in Section 3.1.1, and also gives intuition behind the existence of left-shift Gray codes for bubble languages. The main result in Section 2.4.2 also aids in the formal proof of the reverse cool-lex order found in Section 3.4.1 that is required for the results on shorthand universal cycles in Section 4.2.

Section 2.4.3 defines the concepts of scuts and tails, and proves that these concepts have special properties in bubble languages. In particular, these properties are essential to the left-shift Gray code for bubble languages presented in Chapter 3. Section 2.4.4 then uses these properties, along with the non-increasing prefix property and closure under quotients, to provide an alternate characterization of bubble languages.

2.4.1 Closure

This section gives a sample of operations that can be used to create new bubble languages from previously existing bubble languages.

Union and Intersection

The following theorem proves that bubble languages are closed under union and intersection. The theorem assumes that the multisets of symbols used in the constituent languages are identical since otherwise the union would not be a fixed-content language, and the intersection would be empty.

Theorem 2.4.1 (Bubble languages are closed under union and intersection)**.** *Suppose* $\mathbf{L}$ *and* $\mathbf{L}'$ *are bubble languages, and* $\mathbb{M} = \mathbb{M}'$*. Then,* $\mathbf{L} \cup \mathbf{L}'$ *and* $\mathbf{L} \cap \mathbf{L}'$ *are also bubble languages.*

Proof. Consider $\mathbf{L} \cup \mathbf{L}'$. Notice that $\mathbf{L} \cup \mathbf{L}'$ is a fixed-content language with $\mathsf{content}(\mathbf{L} \cup \mathbf{L}') = \mathbb{M}$ since $\mathbb{M} = \mathbb{M}'$. For the sake of contradiction suppose that $\mathbf{L} \cup \mathbf{L}'$ is not a bubble language since it violates (2.1). (The case where $\mathbf{L} \cup \mathbf{L}'$ violates (2.2) is considered in the next paragraph.) Therefore, there exists $\boldsymbol{s} \in \mathbf{L} \cup \mathbf{L}'$ such that $k = |\overleftarrow{\boldsymbol{s}}(\boldsymbol{s})| < n$ and $\overleftarrow{\mathsf{bubble}}(\boldsymbol{s}, k+1) \notin \mathbf{L} \cup \mathbf{L}'$. Since $\boldsymbol{s} \in \mathbf{L} \cup \mathbf{L}'$ then without loss of generality $\boldsymbol{s} \in \mathbf{L}$. Furthermore, since $\overleftarrow{\mathsf{bubble}}(\boldsymbol{s}, k+1) \notin \mathbf{L} \cup \mathbf{L}'$ then it must also be that $\overleftarrow{\mathsf{bubble}}(\boldsymbol{s}, k+1) \notin \mathbf{L}$. Therefore, $\mathbf{L}$ violates (2.1) and this contradicts the assumption that $\mathbf{L}$ is a bubble language.

For the sake of contradiction suppose that $\mathbf{L} \cup \mathbf{L}'$ is not a bubble language since it violates (2.2). Therefore, there exists $\boldsymbol{s} \in \mathbf{L} \cup \mathbf{L}'$ and an i within $|\overset{\bullet}{\boldsymbol{s}}_{\mathbf{L} \cup \mathbf{L}'}(\boldsymbol{s})| < i \leq |\overleftarrow{\boldsymbol{s}}(\boldsymbol{s})|$ such that $\overleftarrow{\mathsf{bubble}}(\boldsymbol{s}, i) \notin \mathbf{L} \cup \mathbf{L}'$. Due to the bound on i there must exist some rearrangement of the first i symbols of $\boldsymbol{s}$ that results in another string in $\mathbf{L} \cup \mathbf{L}'$. Call this string $\boldsymbol{r}$. Without loss of generality $\boldsymbol{r} \in \mathbf{L}$. Since $\mathbf{L}$ is a bubble language, then by the non-increasing prefix property in Remark 2.2.2 the first i symbols of $\boldsymbol{r}$ can be rearranged back into non-increasing order to result in another string in $\mathbf{L}$. In other words, $\boldsymbol{s} \in \mathbf{L}$ as well. Since $\boldsymbol{r} \in \mathbf{L}$ and $\boldsymbol{s} \in \mathbf{L}$ then the following bound holds $|\overset{\bullet}{\boldsymbol{s}}(\boldsymbol{s})| < i \leq |\overleftarrow{\boldsymbol{s}}(\boldsymbol{s})|$. Furthermore, since $\overleftarrow{\mathsf{bubble}}(\boldsymbol{s}, i) \notin \mathbf{L} \cup \mathbf{L}'$ then $\overleftarrow{\mathsf{bubble}}(\boldsymbol{s}, i) \notin \mathbf{L}$. Therefore, $\mathbf{L}$ violates (2.2) and this contradicts the assumption that $\mathbf{L}$ is a bubble language. Since $\mathbf{L} \cup \mathbf{L}'$ has fixed-content and does not fail (2.1) or (2.2) then it is a bubble language.

Consider $\mathbf{L} \cap \mathbf{L}'$. Notice that $\mathbf{L} \cap \mathbf{L}'$ is a fixed-content language with $\mathsf{content}(\mathbf{L} \cap \mathbf{L}') = \mathbb{M}$ since $\mathbb{M} = \mathbb{M}'$. For the sake of contradiction suppose that $\mathbf{L} \cap \mathbf{L}'$ is not a bubble language since it violates (2.1). (The case where $\mathbf{L} \cap \mathbf{L}'$ violates (2.2) is considered in the next paragraph.) Therefore, there exists $\boldsymbol{s} \in \mathbf{L} \cap \mathbf{L}'$ such that $k = |\overleftarrow{\boldsymbol{s}}(\boldsymbol{s})| < n$ and $\overleftarrow{\mathsf{bubble}}(\boldsymbol{s}, k+1) \notin \mathbf{L} \cap \mathbf{L}'$. Since $\boldsymbol{s} \in \mathbf{L} \cap \mathbf{L}'$ then $\boldsymbol{s} \in \mathbf{L}$ and $\boldsymbol{s} \in \mathbf{L}'$. Furthermore, since $\overleftarrow{\mathsf{bubble}}(\boldsymbol{s}, k+1) \notin \mathbf{L} \cap \mathbf{L}'$ then without loss of generality $\overleftarrow{\mathsf{bubble}}(\boldsymbol{s}, k+1) \notin \mathbf{L}$. Therefore, $\mathbf{L}$ violates (2.1) and this contradicts the assumption that $\mathbf{L}$ is a bubble language.

For the sake of contradiction suppose that $\mathbf{L} \cap \mathbf{L}'$ is not a bubble language since it violates (2.2). Therefore, there exists $\boldsymbol{s} \in \mathbf{L} \cap \mathbf{L}'$ and an i within $|\overset{\bullet}{\boldsymbol{s}}_{\mathbf{L} \cap \mathbf{L}'}(\boldsymbol{s})| < i \leq |\overleftarrow{\boldsymbol{s}}(\boldsymbol{s})|$ such that $\overleftarrow{\mathsf{bubble}}(\boldsymbol{s}, i) \notin \mathbf{L} \cap \mathbf{L}'$. Since $\boldsymbol{s} \in \mathbf{L} \cap \mathbf{L}'$ then $\boldsymbol{s} \in \mathbf{L}$ and $\boldsymbol{s} \in \mathbf{L}'$. Furthermore, since $\overleftarrow{\mathsf{bubble}}(\boldsymbol{s}, i) \notin \mathbf{L} \cap \mathbf{L}'$ then without loss of generality $\overleftarrow{\mathsf{bubble}}(\boldsymbol{s}, i) \notin \mathbf{L}$. Finally, Remark 2.1.6 implies that i is within the bound $|\overset{\bullet}{\boldsymbol{s}}(\boldsymbol{s})| < i \leq |\overleftarrow{\boldsymbol{s}}(\boldsymbol{s})|$ since $\mathbf{L} \cap \mathbf{L}' \subseteq \mathbf{L}$ and so $|\overset{\bullet}{\boldsymbol{s}}(\boldsymbol{s})| \leq |\overset{\bullet}{\boldsymbol{s}}_{\mathbf{L} \cap \mathbf{L}'}(\boldsymbol{s})|$. Therefore, $\mathbf{L}$ violates (2.2) and this contradicts the assumption that $\mathbf{L}$ is a bubble language. Since $\mathbf{L} \cap \mathbf{L}'$ has fixed-content and does not fail (2.1) or (2.2) then it is a bubble language. □

Quotients

This section proves that bubble languages are closed under the quotient operation, and uses this result to prove that pre-necklaces are represented by a bubble language.

Theorem 2.4.2 (Bubble languages are closed under quotients)**.** *If* $\mathbf{L}$ *is a bubble language and* $\boldsymbol{z}$ *is any string, then the language* $\mathbf{L}/\boldsymbol{z}$ *is a bubble language.*

Proof. Let $\mathbf{L}' = \mathbf{L}/\boldsymbol{z}$ where $\mathbf{L}$ is a bubble language. For the sake of contradiction, suppose that $\mathbf{L}'$ is not a bubble language since it violates (2.1). (The case where $\mathbf{L}'$ violates (2.2) is considered in the next paragraph.) Therefore, there exists $\boldsymbol{s}' \in \mathbf{L}'$ such that $k = |\urcorner\llcorner(\boldsymbol{s}')| < n'$ and $\overleftarrow{\mathsf{bubble}}(\boldsymbol{s}', k) \notin \mathbf{L}'$. Let $\boldsymbol{s} = \boldsymbol{s}' \cdot \boldsymbol{z}$. Since $\boldsymbol{s}'$ is not non-increasing then $\urcorner\llcorner(\boldsymbol{s}) = \urcorner\llcorner(\boldsymbol{s}')$ and so $|\urcorner\llcorner(\boldsymbol{s})| = k$. Furthermore, $\boldsymbol{s} \in \mathbf{L}$ since $\boldsymbol{s}' \in \mathbf{L}$, and $\overleftarrow{\mathsf{bubble}}(\boldsymbol{s}, k) \notin \mathbf{L}$ since $\overleftarrow{\mathsf{bubble}}(\boldsymbol{s}', k) \notin \mathbf{L}'$. Therefore, $\mathbf{L}$ violates (2.1) which contradicts the assumption that $\mathbf{L}$ is a bubble language.

Similarly, for the sake of contradiction suppose that $\mathbf{L}'$ is not a bubble language since it violates (2.2). Therefore, there exists $\boldsymbol{s}' \in \mathbf{L}'$ and i within $|\text{🌡}_{\mathbf{L}'}(\boldsymbol{s}')| < i \leq |\urcorner\llcorner(\boldsymbol{s}')|$ such that $\overleftarrow{\mathsf{bubble}}(\boldsymbol{s}', i) \notin \mathbf{L}'$. Let $\boldsymbol{s} = \boldsymbol{s}' \cdot \boldsymbol{z}$. By these choices, $\boldsymbol{s} \in \mathbf{L}$ and $\overleftarrow{\mathsf{bubble}}(\boldsymbol{s}, i) \notin \mathbf{L}$. Therefore, it remains only to show that i is within the range $|\text{🌡}_{\mathbf{L}}(\boldsymbol{s})| < i \leq |\urcorner\llcorner(\boldsymbol{s})|$. To prove this fact, it will be shown that $|\urcorner\llcorner(\boldsymbol{s}')| \leq |\urcorner\llcorner(\boldsymbol{s})|$ and $|\text{🌡}_{\mathbf{L}}(\boldsymbol{s})| < i$. The first of these two inequalities is certainly true. To see why the second is true, notice that since $|\text{🌡}_{\mathbf{L}'}(\boldsymbol{s}')| < i$ then the first i symbols of $\boldsymbol{s}'$ can be rearranged to create another string in $\mathbf{L}'$. Call this string $\boldsymbol{r}'$. Therefore, $\boldsymbol{r}' \cdot \boldsymbol{z} \in \mathbf{L}$. Since the first i symbols of $\boldsymbol{s}$ can be rearranged to create $\boldsymbol{r}' \cdot \boldsymbol{z} \in \mathbf{L}$ then $|\text{🌡}_{\mathbf{L}}(\boldsymbol{s})| < i$ as claimed. Therefore, $\mathbf{L}$ violates (2.2) which contradicts the assumption that $\mathbf{L}$ is a bubble language. Since $\mathbf{L}'$ has fixed-content and does not fail (2.1) or (2.2) then it is a bubble language. □

A *pre-necklace* is a string $\boldsymbol{p}$ in which $p \ \cdot \boldsymbol{z}$ is a necklace for some $\boldsymbol{z}$. Given a multiset of symbols $\mathbb{M}$, the *pre-necklace language* is denoted by $\mathbf{N}^+(\mathbb{M})$ and contains every pre-necklace with content $\mathbb{M}$. For example, when $\mathbb{M} = \{1, 1, 1, 0, 0, 0\}$ then

$$\mathbf{N}^+(\mathbb{M}) = \{111000, 110100, 110010, 101010, 110001, 101001\}.$$

In particular, notice that the last two strings, 110001 and 101001, are not necklaces. On the other hand, necklaces can be obtained by suffixing 000 to the end of each string. That is, $110001 \cdot 000$ and $101001 \cdot 000$. The following remark generalizes this observation.

Remark 2.4.3 (Pre-necklaces and Necklaces)**.** *Suppose* $\mathsf{content}(\boldsymbol{p}) = \mathbb{M}$, $\boldsymbol{z} = d_1^{n_1}$, *and* $\mathbb{M}' \backslash \mathsf{content}(\boldsymbol{z}) = \mathbb{M}$. *Then,*

$$\boldsymbol{p} \in \mathbf{N}^+(\mathbb{M}) \iff \boldsymbol{pz} \in \mathbf{N}(\mathbb{M}')$$

In other words, pre-necklaces over $\mathbb{M}$ *are in one-to-one correspondence with necklaces over* $\mathbb{M} \cup \{\underbrace{d_1, \ldots, d_1}_{n_1 \text{ copies}}\}$ *that end in* $d_1^{n_1}$.

Theorem 2.4.4 (Pre-necklaces are bubble languages)**.** *The language* $\mathbf{N}^+(\mathbb{M})$ *is a bubble language.*

Proof. Let $\boldsymbol{z} = d_1^{n_1}$ and $\mathbb{M}' \backslash \mathsf{content}(\boldsymbol{z}) = \mathbb{M}$. Also, let $\mathbf{L}' = \mathbf{N}(\mathbb{M}')$ and $\mathbf{L} = \mathbf{N}^+(\mathbb{M})$. By Theorem 2.3.20, $\mathbf{L}'$ is a bubble language. Therefore, Theorem 2.4.2 implies that $\mathbf{L}'/\boldsymbol{z}$ is also a bubble language. However, Remark 2.4.3 implies that $\mathbf{L} = \mathbf{L}'/\boldsymbol{z}$. Therefore, $\mathbf{L}$ is also a bubble language. □

2.4.2 Maximum and Maximal Shifts

The results in Chapter 3 involve shifting symbols "as far to the left as possible" within strings inside of bubble languages. This section discusses two interpretations of this concept, and shows that the two interpretations are often equivalent within bubble languages.

Given a string $\boldsymbol{s} \in \mathbf{L}$ and a position i in the string, the *maximum left-shift* of the ith symbol in $\boldsymbol{s}$ is $\mathsf{shift}(\boldsymbol{s}, i, j)$ where j is the minimum value such that $\mathsf{shift}(\boldsymbol{s}, i, j) \in \mathbf{L}$. On the other hand, the *maximal left-shift* of the ith symbol in $\boldsymbol{s}$ adds the condition that each intermediate shift must also produce a string in the language. That is, the result is $\mathsf{shift}(\boldsymbol{s}, i, j)$ where j is the minimum value such that $\mathsf{shift}(\boldsymbol{s}, i, h) \in \mathbf{L}$ for all h satisfying $j \leq h \leq i$. These concepts are not equivalent as illustrated below for $43124213 \in \mathbf{N}(\{1, 1, 2, 2, 3, 3, 4, 4\})$ (multiset necklaces) and the index $i = 8$

$$\overleftarrow{43124213} = 43312421 \text{ versus } 4312421\overleftarrow{3} = 43124231.$$

The left-shift on the left is maximum since 43312421 is a necklace and since shifting the symbol one more position would produce the non-necklace $\overleftarrow{43124213} = 34312421$. On the other hand, the left-shift on the right is maximal since a further left-shift would produce the non-necklace $4312\overleftarrow{4213} = 43124321$. On the other hand, these two concepts are often identical in bubble languages. For example. if the index is changed

to $i = 4$ in the above example, then both a maximum and maximal left-shift produce the following

$$4\overleftarrow{3124}213 = 42314213.$$

To demonstrate the result, it will be shown that certain maximal left-shifts continue until the shifted symbol joins the frozen prefix of the resulting string. Hence, the maximal left-shift is also a maximum left-shift. For example, $\bullet(42314213) = 42$, and so the shifted symbol has joined the frozen prefix in the above example. The necessary condition presented in Lemma 2.4.5 is the ability to maximally left-shift the symbol until it becomes part of the non-decreasing prefix. In the above example, $43\overleftarrow{124}213 = 43214213$ is a necklace, and so the shifted symbol can be maximally left-shifted until it becomes part of the resultant non-increasing prefix. (On the other hand, in the previous example the shifted symbol did not reach the non-increasing prefix and so the equivalence was not guaranteed by Lemma 2.4.5.)

Lemma 2.4.5 (Maximal and Maximum Left-Shifts)**.** *Suppose* $\mathbf{L}$ *is a bubble language,* $\boldsymbol{s} \in \mathbf{L}$*, and* $\boldsymbol{r} = \overleftarrow{\mathsf{shift}}(\boldsymbol{s}, i, j)$ *where* j *is the smallest value such that* $\overleftarrow{\mathsf{shift}}(\boldsymbol{s}, i, h) \in \mathbf{L}$ *for all* $j \le h \le i$*. Furthermore, suppose that* $|\lrcorner(\boldsymbol{r})| \ge j$ *and* $\boldsymbol{s} \ne \boldsymbol{r}$ *and* $f = |\bullet(\boldsymbol{r})|$*. Then,*

$$\overleftarrow{\mathsf{shift}}(\boldsymbol{s}, i, j) = \overleftarrow{\mathsf{shift}}(\boldsymbol{s}, i, f).$$

In other words, if a symbol can be maximally and non-trivially left-shifted to join the non-increasing prefix of a string in a bubble language, then the left-shift can be reexpressed as a left-shift ending at the last symbol in the frozen prefix of the resulting string. Thus, maximal and maximum left-shifts are equivalent in these situations.

Proof. Since $\boldsymbol{r} = \overleftarrow{\mathsf{shift}}(\boldsymbol{s}, i, j)$ then the shifted symbol is $s_i = r_j$. Notice that r_j is part of the non-increasing prefix of $\boldsymbol{r}$ by the supposition $|\lrcorner(\boldsymbol{r})| \ge j$. Therefore, if $f < j$ then (2.2) would imply that $\overleftarrow{\mathsf{bubble}}(\boldsymbol{r}, j) \in \mathbf{L}$. However, this would contradict the choice of j. Thus, $|\bullet(\boldsymbol{r})| \ge j$ and so the frozen prefix of $\boldsymbol{r}$ includes $r_1 r_2 \cdots r_j$. Furthermore, since $\boldsymbol{s} \ne \boldsymbol{r}$ then it must be that $\overrightarrow{\mathsf{bubble}}(\boldsymbol{r}, j) \in \mathbf{L}$. Therefore,

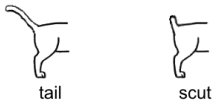
$$r_j = r_{j+1} = \cdots = r_f$$

since otherwise $\overrightarrow{\mathsf{bubble}}(\boldsymbol{r}, j) \in \mathbf{L}$ would contradict the fact that $r_1 r_2 \cdots r_f$ is frozen. Therefore, $\overleftarrow{\mathsf{shift}}(\boldsymbol{s}, i, j) = \overleftarrow{\mathsf{shift}}(\boldsymbol{s}, i, f)$ as claimed. □

In Section 3.1.1 these left-shifts will be formally defined as greedy left-shifts, and are used in the left-shift Gray code for all bubble languages.

2.4.3 Scuts and Tails

In English, a *scut* is a short and wide tail found on animals such as deer or rabbits. The difference between a tail and a scut is illustrated below.



In this thesis, the scut is a special type of suffix that is present in every string that is not non-increasing. On the other hand, a tail is a string that is comprised of a non-increasing prefix followed by a scut. This section shows that the scuts and tails in bubble languages must obey several properties, and these properties are the keys to proving the shift Gray code found in Chapter 3. Scuts are discussed in Section 2.4.3, tails are discussed in Section 2.4.3, and their relationship to bubble languages is presented in Section 2.4.3.

Scuts

This section discusses scuts and their relationship to strings, multisets, and languages. The *scut* of a string is its shortest suffix that is not also a suffix of the non-increasing arrangement of its symbols. To illustrate this notion, consider the following examples of scuts for various strings over $\mathbb{M} = \{1, 1, 1, 2, 2, 3\}$

$$\mathsf{scut}(121213) = 3, \quad \mathsf{scut}(211231) = 31, \quad \mathsf{scut}(212311) = 311, \quad \mathsf{scut}(223111) = 3111,$$
$$\mathsf{scut}(232111) = 32111, \quad \mathsf{scut}(312112) = 2, \quad \mathsf{scut}(231121) = 21, \quad \mathsf{scut}(312211) = 211.$$

(Within the examples, the non-increasing arrangement of $\mathbb{M}$ is 322111 and so the scuts are shortest suffixes that are not also suffixes of 322111.) Definition 2.4.6 formalizes this concept using the conventions for multiset $\mathbb{M}$ discussed in Section 2.1.1.

Definition 2.4.6 (Scut of a string (scut))**.** *Suppose* $\boldsymbol{s} = s_1 s_2 \cdots s_n \in \mathbf{L}$ *and* $\boldsymbol{t} = \mathrm{\lrcorner}$ *and* $\mathrm{\lrcorner}(\boldsymbol{s}) \neq \boldsymbol{s}$*. Then,*

$$\mathsf{scut}(\boldsymbol{s}) = s_j s_{j+1} \cdots s_n$$

where j *is the minimum value such that* $s_{j+1} s_{j+2} \cdots s_n = t_{j+1} t_{j+2} \cdots t_n$*. In other words,* scut *gives the shortest suffix of a string that is not also a suffix of* $\mathrm{\lrcorner}(\boldsymbol{s})$*.*

Several examples of scuts over $\mathbb{M} = \{1, 1, 1, 2, 2, 3\}$ were given prior to Definition 2.4.6. In fact, these examples include every possible scut when $\mathbb{M} = \{1, 1, 1, 2, 2, 3\}$. In particular, given this choice of $\mathbb{M}$, the first symbol of a scut cannot be 1 since it

is simply not possible for a scut to begin with the smallest symbol. Furthermore, the scuts beginning with 2 must follow with a strict suffix of 111 (the symbols smaller than 2), and the scuts beginning with 3 must follow with a strict suffix of 22111 (the symbols smaller than 3). In general, when working with a given multiset of symbols, the possible scuts can be specified by their first symbol and their length. Equivalently, the possible scuts can be specified by their first symbol and the number of smaller symbols that are excluded from the scut. For technical reasons, this second method of specification proves to be more useful throughout this thesis, and so it will be followed. (In particular, Remark 2.4.13 nicely relates shifts, scuts, and the non-increasing string using this specification, while scuts excluding a single symbol are of the utmost importance by Lemma 2.4.19.) Informally, $\mathsf{scut}_{\mathbb{M}}(j,i)$ refers to the scut beginning with the jth smallest distinct symbol in $\mathbb{M}$, and which does not contain i smaller symbols. As was the case with languages, multisets that appear as subscripts in this thesis are optional and the default value is $\mathbb{M}$. For example, $\mathsf{scut}(j,i) = \mathsf{scut}_{\mathbb{M}}(j,i)$. The scuts from above are now expressed using this notation

$$\mathsf{scut}(3,5) = 3, \quad \mathsf{scut}(3,4) = 31, \quad \mathsf{scut}(3,3) = 311, \quad \mathsf{scut}(3,2) = 3111,$$
$$\mathsf{scut}(3,1) = 32111, \quad \mathsf{scut}(2,3) = 2, \quad \mathsf{scut}(2,2) = 21, \quad \mathsf{scut}(2,1) = 211.$$

Notice that $\mathsf{scut}(j,i)$ will be well-defined if and only if j is within $2 \le j \le m$ and i is within $1 \le i \le \underline{n}_{j-1}$. (This is because every scut must begin with a symbol larger than the smallest, and must exclude at least one smaller symbol and no more than the total number of symbols smaller than its first symbol.) This leads to the following formal definition of scut. For the remainder of the thesis, whenever $\mathsf{scut}(j,i)$ is written it will be assumed that the bounds given for i and j in Definition 2.4.7 are satisfied.

Definition 2.4.7 (Scut creation (scut))**.** *Suppose j is within $2 \le j \le m$, and i is within $1 \le i \le \underline{n}_{j-1}$. Then,*

$$\mathsf{scut}_{\mathbb{M}}(j,i) = d_j e_{\underline{n}_{j-1}-i} e_{\underline{n}_{j-1}-i-1} \dots e_1.$$

In other words, $\mathsf{scut}_{\mathbb{M}}(j,i)$ is the scut that has d_j as its first symbol and excludes the largest i symbols that are smaller than d_j.

Given a language $\mathbf{L}$, the set of all scuts is represented by $\mathsf{scuts}(\mathbf{L})$. After this concept is formalized and illustrated, two important values involving this set will be introduced.

Definition 2.4.8 (Scuts of a language (scuts))**.** *Suppose* $\mathbf{L}$ *is a language. Then,*

$$\mathsf{scuts}(\mathbf{L}) = \{\mathsf{scut}(\boldsymbol{s}) \mid \boldsymbol{s} \in \mathbf{L}\}.$$

In other words, $\mathsf{scuts}(\mathbf{L})$ *is the set of all scuts that appear within some string in* $\mathbf{L}$*.*

For example, if

$$\mathbf{L} = \{43211, 43121, 43112, 42311, 42131, 41321\}$$

then

$$\mathsf{scuts}(\mathbf{L}) = \{21, 2, 311, 31\} = \{\mathsf{scut}(2,1), \mathsf{scut}(2,2), \mathsf{scut}(3,1), \mathsf{scut}(3,2)\}. \tag{2.24}$$

In particular, $3 \notin \mathsf{scuts}(\mathbf{L})$ since there is no string within $\mathbf{L}$ that has 3 as a suffix. Given $\mathsf{scuts}(\mathbf{L})$, there are two types of maximum values that we consider. The first is the maximum value of j for which some $\mathsf{scut}(j,i)$ is in the set. This value will be denoted by $☘_{\mathbf{L}}$. Given a particular value of j, the second is the maximum value of i for which $\mathsf{scut}(j,i)$ is in the set. This value will be denoted by $☘_{\mathbf{L}}(j)$. (The radioactive symbol is chosen to represent the fact that any higher values come with the danger of having no associated string in the given language.) These values are illustrated below for the previous example

$$☘ = 3 \qquad ☘(2) = 2 \qquad ☘(3) = 2$$

where the three values respectively follow from $\mathsf{scut}(3,1)$ (or $\mathsf{scut}(3,2)$), $\mathsf{scut}(2,2)$, and $\mathsf{scut}(3,2)$. These maximum values are now formally defined, and are featured in Section 2.4.3 where they help characterize the possible values contained in $\mathsf{scuts}(\mathbf{L})$ for bubble languages.

Definition 2.4.9 (Maximum excluded from a scut ($☘$))**.** *Within fixed-content language* $\mathbf{L}$*,* maximum number of symbols that can be excluded from a scut beginning with the jth smallest symbol *is*

$$☘_{\mathbf{L}}(j) = \max_i\{\mathsf{scut}(j,i) \in \mathsf{scuts}(\mathbf{L})\}.$$

In other words, $\mathsf{scut}(j, ☘_{\mathbf{L}}(j))$ *is the shortest scut that begins with* d_j *for some string in* $\mathbf{L}$*. If* $\mathsf{scut}(j,i) \notin \mathsf{scuts}(\mathbf{L})$ *for all* i *then define* $☘_{\mathbf{L}}(j) = 0$*.*

Definition 2.4.10 (Maximum first symbol in scut ($\clubsuit$))**.** *Within fixed-content language* $\mathbf{L}$*, the* maximum first symbol of a scut *is*

$$\clubsuit_{\mathbf{L}} = \max_{j}\{\exists\ i \text{ such that } \mathsf{scut}(j,i) \in \mathsf{scuts}(\mathbf{L})\}.$$

If $\mathsf{scuts}(\mathbf{L}) = \varnothing$ *then define* $\clubsuit_{\mathbf{L}} = 1$.

Tails

In bubble languages, it is a simple task to determine whether or not a given scut appears in the language. In particular, Remark 2.2.2 implies that a scut will appear in $\mathsf{scuts}(\mathbf{L})$ if and only if the string comprised of a non-increasing prefix followed by the scut is in the language. That is, if $\mathbf{L}$ is a bubble language then

$$\mathsf{scut}(j,i) \in \mathsf{scuts}(\mathbf{L}) \iff \urcorner\llcorner \cdot \mathsf{scut}(j,i) \in \mathbf{L}. \tag{2.25}$$

For example, if $\mathbf{L}$ is a bubble language over $\mathbb{M} = \{1,1,2,2,3,3\}$ then determining if $31 \in \mathsf{scuts}(\mathbf{L})$ is equivalent to determining if $3221 \cdot 31 \in \mathbf{L}$. When scuts are elongated by non-increasing prefixes in this way, then the resulting string will be known as a *tail*. (This terminology is also helpful in Chapter 3 where tails are the last strings within various sublists of cool-lex order.) A formal definition of tails appears below, followed by a restatement of (2.25).

Definition 2.4.11 (Tails)**.** *The* tail *with scut* $\mathsf{scut}(j,i)$ *is*

$$\mathsf{tail}_{\mathbb{M}}(j,i) = \urcorner\llcorner \cdot \mathsf{scut}(j,i).$$

In other words, tails are strings comprised of a non-increasing prefix followed by a scut.

Remark 2.4.12 (Tail and scut inclusion in bubble languages)**.** *Suppose* $\mathbf{L}$ *is a bubble language. Then,*

$$\mathsf{scut}(j,i) \in \mathsf{scuts}(\mathbf{L}) \iff \mathsf{tail}(j,i) \in \mathbf{L}.$$

In other words, a given scut appears within a bubble language if and only if its tail string is in the language.

One interpretation of $\mathsf{tail}(j,i)$ is that it is the lexicographically largest string over $\mathbb{M}$ with a copy of d_j in position $\overline{n}_j + i$. Equivalently, $\mathsf{tail}(j,i)$ can be interpreted in terms of its relationship to the non-increasing string. In particular, $\mathsf{tail}(j,i)$ is

the string that is obtained from $\urcorner\llcorner$ by shifting the rightmost copy of d_j a total of i positions to the right. This second interpretation proves to be extremely useful within this thesis. To illustrate this relationship, consider the following two examples over $\mathbb{M} = \{1, 1, 1, 2, 2, 3, \}$

$$\begin{aligned} \overrightarrow{322111} &= 221131 \\ &= 2211 \cdot 31 \\ &= \urcorner\llcorner \cdot \mathsf{scut}(3,4) \\ &= \mathsf{tail}(3,4) \end{aligned} \qquad \begin{aligned} 322\overrightarrow{111} &= 321112 \\ &= 32111 \cdot 2 \\ &= \urcorner\llcorner \cdot \mathsf{scut}(2,3) \\ &= \mathsf{tail}(2,3). \end{aligned}$$

In both cases the non-increasing string 322111 is modified by a right-shift and the result is a tail. For instance, in the second column the rightmost copy of $d_2 = 2$ is right-shifted past 3 smaller symbols to create $\mathsf{tail}(2,3)$. This relationship is formalized by the following remark.

Remark 2.4.13 (Tails and non-increasing strings)**.** *Suppose $h = \overline{n}_j$ with $j \geq 2$ and i within $1 \leq i \leq \underline{n}_{j-1}$. Then,*

$$\overrightarrow{\mathsf{shift}}(\urcorner\llcorner, h, h+i) = \mathsf{tail}(j,i).$$

In other words, $\mathsf{tail}(j,i)$ is obtained from $\urcorner\llcorner$ by right-shifting the rightmost copy of d_j a total of i positions.

The relationship between tails and the non-increasing string allows for a number of simple observations to be made. First of all, the non-increasing prefix of $\mathsf{tail}(j,i)$ has length $\overline{n}_j - 1 + i$. (Whenever $\mathsf{tail}(j,i)$ is written, it is assumed that the bounds on i and j given in Definition 2.4.7 are satisfied.) For example, if $\mathbb{M} = \{1,1,2,2,3,3,4,4,5,5\}$ then

$$|\urcorner\llcorner(\mathsf{tail}(4,5))| = |\urcorner\llcorner(5543322141)| = |55433221| = |554| + |33221| = 3 + 5 = 8$$

since the non-increasing prefix is comprised of the $\overline{n}_4 - 1 = 3$ symbols which are at least as large as 4 together with the 5 excluded smaller symbols. The reader should also note that $\mathsf{tail}(j,i)$ can be partitioned into its non-increasing prefix and its suffix $\mathsf{scut}(j,i)$, (In other words, the non-increasing prefix of $\mathsf{tail}(j,i)$ ends immediately before its scut.) Second, if $\boldsymbol{s} = \mathsf{tail}(j,i)$ and $k = |\urcorner\llcorner(\boldsymbol{s})|$ then several inequalities involving the symbols in the substring $s_k \cdot s_{k+1} \cdot s_{k+2}$ can be noted. First, $s_k < s_{k+1}$ since the last symbol in the non-increasing prefix is always smaller than the next

symbol. Furthermore, $s_k \geq s_{k+2}$ since $s_k \cdot s_{k+2}$ is a substring of the non-increasing string. To illustrate these inequalities, suppose that $\mathbb{M}$ is unchanged from above and $\boldsymbol{s} = \mathsf{tail}(4,4) = 5543322411$. Then, $s_k \cdot s_{k+1} \cdot s_{k+2} = 241$ and inequality $s_{k+1} > s_k \geq s_{k+2}$ is easily verified. (The $\mathsf{tail}(4,5)$ example above provides a situation where the second inequality is not strict.) One consequence of these inequalities is that s_{k+1} is strictly greater than both the symbol to its left and its right. For this reason, bubble shifts amongst these symbols can be expressed as shifts of length one. These simple facts are recorded by the following two remarks.

Remark 2.4.14 (Non-increasing prefix length in tails)**.**

$$|\llcorner(\mathsf{tail}(j,i))| = \overline{n}_j - 1 + i$$

Remark 2.4.15 (Symbol inequalities in tails)**.** *Suppose* $\boldsymbol{s} = \mathsf{tail}(j,i)$ *and* $k = |\llcorner(\boldsymbol{s})|$. *Then,*

$$s_{k+1} > s_k \geq s_{k+2}.$$

The relationship between tails and the non-increasing string given in Remark 2.4.13 also allows for a number of simple equalities involving shifts to be stated. For example, if the first symbol in the scut of a tail is shifted one position to the left or right, then this will increase or decrease the number of excluded symbols respectively. Furthermore, if a symbol in the non-increasing prefix is bubble right-shifted then this will create a tail whose scut begins with this bubbled symbol and which excludes one smaller symbol. To illustrate these ideas in more detail, consider the following examples over $\mathbb{M} = \{1,1,2,2,3,3,4,4,5,5\}$. The initial strings in the first two columns are identical and $k = 6$ is the length of its non-increasing prefix. Finally, each equality is represented using left-shifts. (In order for this to be possible, notice that right-shifting one symbol past another can also be accomplished by a left-shift. That is, $\overrightarrow{\mathsf{shift}}(\boldsymbol{s}, x, x+1) = \overleftarrow{\mathsf{shift}}(\boldsymbol{s}, x+1, x)$.)

$\overleftarrow{\mathsf{bubble}}(\mathsf{tail}(4,3), k+1)$	$\overleftarrow{\mathsf{bubble}}(\mathsf{tail}(4,3), k+2)$	$\overleftarrow{\mathsf{bubble}}(\llcorner, 7)$
$= \overleftarrow{\mathsf{bubble}}(5543324211, k+1)$	$= \overleftarrow{\mathsf{bubble}}(5543324211, k+2)$	$= \overleftarrow{\mathsf{bubble}}(5544332211, 7)$
$= 5543\overleftarrow{324}211$	$= 5543\overleftarrow{324}211$	$= 5544\overleftarrow{332}211$
$= 5543342211$	$= 5543322411$	$= 5544323211$
$= 55433 \cdot 42211$	$= 5543322 \cdot 411$	$= 554432 \cdot 3211$
$= \mathsf{tail}(4,2)$	$= \mathsf{tail}(4,4)$	$= \mathsf{tail}(3,1)$

In the first column, $s_k \neq s_{k+1}$ (by Remark 2.4.15) so the bubble left-shift amounts to a shift of one position. This left-shift causes the shifted 4 to exclude one fewer symbol from its scut, thereby transforming $\mathsf{tail}(4,3)$ into $\mathsf{tail}(4,2)$. In general, bubble left-shifting the $(k+1)$st symbol will exclude one fewer symbol from the tail's scut (so long as there is another smaller symbol in the non-increasing prefix to exclude). In the second column, the second symbol in the scut is bubble left-shifted. This bubble left-shift is equivalent to bubble right-shifting the first symbol in the scut by Remark 2.4.15 since $s_{k+1} \neq s_{k+2}$. This left-shift causes the shifted 2 to move from the scut into the non-increasing prefix, thereby increasing the number of excluded symbols in the scut and transforming $\mathsf{tail}(4,3)$ into $\mathsf{tail}(4,4)$. In general, bubble left-shifting the $(k+2)$nd symbol will exclude an additional symbol from the tail's scut (so long as there is another smaller symbol in the scut to exclude). In the third column a symbol in the non-increasing prefix is bubble left-shifted. This left-shift causes the shifted 2 to pass over a single 3, thereby creating a non-increasing prefix followed by a scut that begins with 3 and excludes one smaller symbol. That is, $\urcorner\llcorner$ is transformed into $\mathsf{tail}(3,1)$. In general, bubble left-shifting d_j in $\urcorner\llcorner$ will create $\mathsf{tail}(d_{j+1},1)$ (so long as d_{j+1} is a symbol). The following lemmas formalize these simple results.

Lemma 2.4.16 (Bubble left-shift $(k+1)$st symbol in tail)**.** *Suppose* $k = |\urcorner\llcorner(\mathsf{tail}(j,i))|$ *and* $i > 1$*. Then,*

$$\overleftarrow{\mathsf{bubble}}(\mathsf{tail}(j,i), k+1) = \mathsf{tail}(j, i-1).$$

In other words, bubble left-shifting the first symbol in the scut of $\mathsf{tail}(j,i)$ *results in* $\mathsf{tail}(j,i-1)$*, so long as* $\mathsf{tail}(j,i-1)$ *is well-defined.*

Proof. Let $h = \overline{n}_j$ (in the non-increasing string h is the index of the rightmost copy of d_j). The proof is a result of the following derivation:

$$\begin{aligned}
&\overleftarrow{\mathsf{bubble}}(\mathsf{tail}(j,i), k+1) \\
&= \overleftarrow{\mathsf{shift}}(\mathsf{tail}(j,i), k+1, k) && \text{by Remark 2.4.15} \\
&= \overleftarrow{\mathsf{shift}}(\mathsf{tail}(j,i), h+i, h+i-1) && \text{by Remark 2.4.14} \\
&= \overleftarrow{\mathsf{shift}}(\overrightarrow{\mathsf{shift}}(\urcorner\llcorner, h, h+i), h+i, h+i-1) && \text{by Remark 2.4.13} \\
&= \overrightarrow{\mathsf{shift}}(\urcorner\llcorner, h, h+i-1) \\
&= \mathsf{tail}(j, i-1) && \text{by Remark 2.4.13.}
\end{aligned}$$

□

Lemma 2.4.17 (Bubble left-shift $(k+2)$nd symbol in tail)**.** *Suppose* $k = |\urcorner\llcorner(\mathsf{tail}(j,i))|$ *and* $i < \underline{n}_{j-1}$*. Then,*

$$\overleftarrow{\mathsf{bubble}}(\mathsf{tail}(j,i), k+2) = \mathsf{tail}(j, i+1)$$

In other words, bubble left-shifting the second symbol in the scut of $\mathsf{tail}(j,i)$ *results in* $\mathsf{tail}(j,i+1)$*, so long as* $\mathsf{tail}(j,i+1)$ *is well-defined.*

Proof. Let $h = \overline{n}_j$ (in the non-increasing string h is the index of the rightmost copy of d_j). The proof is a result of the following derivation:

$$\begin{aligned}
&\overleftarrow{\mathsf{bubble}}(\mathsf{tail}(j,i), k+2) \\
&= \overleftarrow{\mathsf{shift}}(\mathsf{tail}(j,i), k+2, k+1) && \text{by Remark 2.4.15} \\
&= \overleftarrow{\mathsf{shift}}(\mathsf{tail}(j,i), h+i+1, h+i) && \text{by Remark 2.4.14} \\
&= \overleftarrow{\mathsf{shift}}(\overrightarrow{\mathsf{shift}}(\urcorner\llcorner, h, h+i), h+i+1, h+i) && \text{by Remark 2.4.13} \\
&= \overrightarrow{\mathsf{shift}}(\urcorner\llcorner, h, h+i+1) \\
&= \mathsf{tail}(j, i+1) && \text{by Remark 2.4.13.}
\end{aligned}$$

□

Lemma 2.4.18 (Bubble left-shift in non-increasing string)**.** *Suppose* h *is within* $\overline{n}_{j+1} + 1 \le h \le \overline{n}_j$ *and* $j < m$*. Then,*

$$\overleftarrow{\mathsf{bubble}}(\urcorner\llcorner, h) = \mathsf{tail}(j+1, 1).$$

In other words, bubble left-shifting a copy of d_j *in the non-increasing string results in* $\mathsf{tail}(j+1,1)$ *so long as* d_j *is not the maximum symbol.*

Proof. Before starting the proof notice that within $\urcorner\llcorner$ all of the symbols between positions $\overline{n}_{j+1}+1$ and $\overline{n}_j$ are equal to d_j, and the symbol in position $\overline{n}_{j+1}$ is equal to d_{j+1}. The proof is a result of the following derivation

$$\begin{aligned}
&\overleftarrow{\mathsf{bubble}}(\urcorner\llcorner, h) \\
&= \overleftarrow{\mathsf{bubble}}(\urcorner\llcorner, \overline{n}_{j+1}+1) && \text{from above} \\
&= \overleftarrow{\mathsf{shift}}(\urcorner\llcorner, \overline{n}_{j+1}+1, \overline{n}_{j+1}) && \text{from above} \\
&= \overrightarrow{\mathsf{shift}}(\urcorner\llcorner, \overline{n}_{j+1}, \overline{n}_{j+1}+1) \\
&= \mathsf{tail}(j+1, 1) && \text{by Remark 2.4.13.}
\end{aligned}$$

□

Scuts in Bubble Languages

While Remark 2.4.12 showed that it is easy to determine whether or not an individual scut is contained in a bubble language, the two lemmas in this section show that the set of all scuts is well-organized within bubble languages. Since bubble languages satisfy the non-increasing prefix property given in Remark 2.2.2, then these two lemmas imply that tails are also well-organized within bubble languages.

The first lemma shows that if $\mathsf{scut}(j,i)$ is the suffix of some string in a bubble language, then $\mathsf{scut}(j,i-1)$ will also be the suffix of some string in the language (so long as $i>1$). In other words, if $\mathbf{L}$ is a bubble language then $\mathsf{scut}(j,i) \in \mathsf{scuts}(\mathbf{L})$ for every i within $1 \le i \le \clubsuit(j)$ (note that this range might be empty). This result follows from the fact that if a scut is present in a bubble language then its associated tail must be within the language, and the fact that these tails differ by bubble left-shifts that produce additional strings within bubble languages.

Lemma 2.4.19 (Longer scuts in bubble languages)**.** *Suppose* $\mathbf{L}$ *is a bubble language. Then,*

$$\mathsf{scut}(j,i) \in \mathsf{scuts}(\mathbf{L}) \text{ implies } \mathsf{scut}(j,i-1) \in \mathsf{scuts}(\mathbf{L}) \text{ whenever } i>1. \tag{2.26}$$

In other words, if a scut appears in a bubble language then so do scuts with the same first symbol that are longer (i.e. exclude fewer symbols).

Proof. Let $k = |\llcorner(\mathsf{tail}(j,i))|$. The proof is a result of the following series of implications

$$\begin{aligned}
&\mathsf{scut}(j,i) \in \mathsf{scuts}(\mathbf{L}) \\
&\implies \mathsf{tail}(j,i) \in \mathbf{L} && \text{by Remark 2.4.12} \\
&\implies \overleftarrow{\mathsf{bubble}}(\mathsf{tail}(j,i),k+1) \in \mathbf{L} && \text{by (2.1) in Definition 2.2.1} \\
&\implies \mathsf{tail}(j,i-1) \in \mathbf{L} && \text{by Lemma 2.4.16} \\
&\implies \mathsf{scut}(j,i-1) \in \mathsf{scuts}(\mathbf{L}).
\end{aligned}$$

□

The second lemma shows that if $\mathsf{scut}(j,1)$ is the suffix of some string in a bubble language, then $\mathsf{scut}(j-1,1)$ will also be the suffix of some string in the language (so

long as $j > 2$). In other words, if $\mathbf{L}$ is a bubble language then $\mathsf{scut}(j,1) \in \mathsf{scuts}(\mathbf{L})$ for every j within $2 \leq j \leq \clubsuit$ (note that this range might be empty). This result follows from the fact that if such a scut is present in a bubble language then its associated tail must be within the language, and the fact that these tails differ from the non-increasing string by a bubble left-shift. This limits the possible length of the frozen prefix within the non-increasing string, and allows for other bubble left-shifts to give strings in the language, and these strings are the tails of the desired scuts.

Lemma 2.4.20 (Smaller first symbol scuts in bubble languages)**.** *Suppose* $\mathbf{L}$ *is a bubble language.*

$$\mathsf{scut}(j,1) \in \mathsf{scuts}(\mathbf{L}) \text{ implies } \mathsf{scut}(j-1,1) \in \mathsf{scuts}(\mathbf{L}) \text{ whenever } j > 2. \tag{2.27}$$

In other words, if a scut is the suffix of some string in a bubble language then there is a string in the language whose scut has a smaller first symbol.

Proof. Let $h = \overline{n}_j$ and $g = \overline{n}_{j-1}$. The proof is now a result of the following series of implications

$$\begin{aligned}
&\mathsf{scut}(j,1) \in \mathsf{scuts}(\mathbf{L}) \\
&\implies \mathsf{tail}(j,1) \in \mathbf{L} && \text{by Remark 2.4.12} \\
&\implies \overleftarrow{\mathsf{bubble}}(\urcorner\llcorner, h+1) \in \mathbf{L} && \text{by Lemma 2.4.18} \\
&\implies |\thermometer(\urcorner\llcorner)| \leq h \\
&\implies \overleftarrow{\mathsf{bubble}}(\urcorner\llcorner, g+1) \in \mathbf{L} && \text{by (2.2) in Definition 2.2.1} \\
&\implies \mathsf{tail}(j-1,1) \in \mathbf{L} && \text{by Lemma 2.4.18} \\
&\implies \mathsf{scut}(j-1,1) \in \mathsf{scuts}(\mathbf{L}).
\end{aligned}$$

□

Before concluding this section, it is mentioned that the previous two lemmas also hold for certain non-bubble languages. For example, if $\mathbf{L} = \mathbf{R}(\mathbb{M})$ (multiset bracelets) with $\mathbb{M} = \{1,1,2,2,3,3,4,4\}$ then

$$\mathsf{scuts}(\mathbf{L}) = \{21, 2, 3211, 311, 31, 3, 432211, 42211, 4211, 411, 41\}.$$

However, even if these lemmas hold for a particular non-bubble language, it cannot be guaranteed that the same lemmas will hold after the quotient operation is applied

to the language. For example, if $\mathbf{L}' = \mathbf{L}/3$ then

$$\mathsf{scuts}(\mathbf{L}') = \{2, 21, 4211, 411, 41\}.$$

In this case notice that there is no scut of the form $\mathsf{scut}_{\mathbb{M}'}(3,1) = 3211$, which violates Lemma 2.4.20 due to the presence of $\mathsf{scut}_{\mathbb{M}'}(4,1)$. Furthermore, the scut $\mathsf{scut}_{\mathbb{M}'}(4,2)$ is present but $\mathsf{scut}_{\mathbb{M}'}(4,1)$ is not, and this violates Lemma 2.4.19. On the other hand, bubble languages are closed under quotients, as proven by Theorem 2.4.2. Therefore, Lemmas 2.4.19 and 2.4.20 apply to every bubble language and every language formed from the quotient operation of the language. As a counterpoint, the next section proves that non-bubble languages always have a quotient that fails Lemma 2.4.19 or Lemma 2.4.20 or Remark 2.2.2.

2.4.4 Structure

In this chapter we have shown that non-empty bubble languages have the non-increasing prefix property, and that their scuts are well-organized. It has also been shown that the quotients of bubble languages are also bubble languages. This section proves that these derived properties fully characterize bubble languages. That is, if every non-empty quotient of $\mathbf{L}$ contains the non-increasing string and has well-organized scuts, then $\mathbf{L}$ is a bubble language. This result is formalized in Theorem 2.4.21 and completes our investigation of bubble languages. Although the result provides a structural definition for bubble languages, it is otherwise unused in this thesis.

The proof of the theorem is relatively straight-forward. If $\mathbf{L}$ is not a bubble language then there exists a string $\boldsymbol{s}$ and an index h that prove this fact. More specifically, there exists $\boldsymbol{s} \in \mathbf{L}$ and h satisfying $|\text{🌡}(\boldsymbol{s})| < h \leq |\llcorner(\boldsymbol{s})| + 1$ such that $\overleftarrow{\mathsf{bubble}}(\boldsymbol{s}, h) \notin \mathbf{L}$. Given these choices of $\boldsymbol{s}$ and h, the quotient $\mathbf{L}' = \mathbf{L}/s_{h+1}s_{h+2}\cdots s_n$ can be shown to fail one of the derived properties. That is, when $\overleftarrow{\mathsf{bubble}}(\boldsymbol{s}, h)$ proves that $\mathbf{L}$ is not a bubble language, then the quotient obtained by removing the symbols following this shift will either fail the non-increasing prefix property, or its scuts will not be well-organized. To illustrate three situations that can arise within this argument, consider the following fixed-content language that is not a bubble language

$$\mathbf{L} = \{44332211, 43234112, 44332121, 44323121, 43324121, 44321213, 43421213\}.$$

The language violates (2.1) because $|\urcorner\llcorner(43234112)| + 1 = 4$ but

$$\overleftarrow{\mathsf{bubble}}(43234112, 4) = 43\overleftarrow{234}112 = 43324112 \notin \mathbf{L}.$$

Alternatively, the symbols following the above shift are 4112 and $\mathbf{L}' = \mathbf{L}/4112 = \{4323\}$ fails Remark 2.2.2 since $\urcorner\llcorner(\mathbb{M}') = 4332 \notin \mathbf{L}'$. Similarly, $\mathbf{L}$ also violates (2.1) because $|\urcorner\llcorner(43324121)| + 1 = 5$ but

$$\overleftarrow{\mathsf{bubble}}(43324121, 5) = 433\overleftarrow{24}121 = 43342121 \notin \mathbf{L}.$$

Alternatively, the symbols following the above shift are 121 and $\mathbf{L}' = \mathbf{L}/121 = \{44332, 44323, 43324\}$ fails Lemma 2.4.19 because $\mathsf{scut}_{\mathbb{M}'}(3,3) = 4 \in \mathsf{scuts}(\mathbf{L}')$ and $\mathsf{scut}_{\mathbb{M}'}(3,2) = 42 \notin \mathsf{scuts}(\mathbf{L}')$. Finally, $\mathbf{L}$ violates (2.2) because $|🌡(44321213)| < 4 \leq |\urcorner\llcorner(44321213)|$ but

$$\overleftarrow{\mathsf{bubble}}(44321213, 4) = 44\overleftarrow{321}213 = 44231213 \notin \mathbf{L}.$$

Alternatively, the symbols following the above shift are 1213 and $\mathbf{L}' = \mathbf{L}/1213 = \{4432, 4342\}$ fails Lemma 2.4.19 because $\mathsf{scut}_{\mathbb{M}'}(3,1) = 42 \in \mathsf{scuts}(\mathbf{L}')$ and $\mathsf{scut}_{\mathbb{M}'}(2,1) = 3 \notin \mathsf{scuts}(\mathbf{L}')$.

Theorem 2.4.21 (Structure)**.** *A fixed-content language* $\mathbf{L}$ *is a bubble language if and only if the following three conditions hold for every string* $\boldsymbol{z}$ *such that* $\mathbf{L}' = \mathbf{L}/\boldsymbol{z}$ *is non-empty:*

$$\urcorner\llcorner(\mathbb{M}') \in \mathbf{L}' \tag{2.28}$$

and

$$\mathsf{scut}(j,i) \in \mathsf{scuts}(\mathbf{L}') \text{ implies } \mathsf{scut}(j,i-1) \in \mathsf{scuts}(\mathbf{L}') \text{ whenever } i > 1 \tag{2.29}$$

and

$$\mathsf{scut}(j,1) \in \mathsf{scuts}(\mathbf{L}') \text{ implies } \mathsf{scut}(j-1,1) \in \mathsf{scuts}(\mathbf{L}') \text{ whenever } j > 2. \tag{2.30}$$

In other words, bubble languages can be characterized as fixed-content languages whose quotients include the non-increasing string (Remark 2.2.2) and whose scuts satisfy the properties in Lemmas 2.4.19 and 2.4.20.

Proof. If $\mathbf{L}$ is a bubble language then $\mathbf{L}'$ is also a bubble language by Theorem 2.4.2. Therefore, $\mathbf{L}'$ satisfies the given conditions by Remark 2.2.2 and Lemmas 2.4.19 and 2.4.20. The contrapositive of the remaining direction is now proven. Suppose that $\mathbf{L}$

is not a bubble language. Therefore, there exists $\boldsymbol{s} \in \mathbf{L}$ that violates (2.1) or (2.2). These two cases are handled separately, and in both cases let $k = |\urcorner\llcorner(\boldsymbol{s})|$.

In the first case suppose that $\boldsymbol{s}$ does not satisfy (2.1). Thus, $k < n$ and $\boldsymbol{t} = \overleftarrow{\mathsf{bubble}}(\boldsymbol{s}, k+1) \notin \mathbf{L}$. Let $\mathbf{L}' = \mathbf{L}/s_{k+2}s_{k+3}\cdots s_n$ (and so $n' = k+1$), $\boldsymbol{s}' = s_1 s_2 \cdots s_{k+1}$, and $\boldsymbol{t}' = t_1 t_2 \cdots t_{k+1}$. Since $\boldsymbol{s} \in \mathbf{L}$ and $\boldsymbol{t} \notin \mathbf{L}$, then $\boldsymbol{s}' \in \mathbf{L}'$ and $\boldsymbol{t}' \notin \mathbf{L}'$. Notice that the first k symbols of $\boldsymbol{s}'$ are non-increasing, and that $\boldsymbol{s}'$ and $\boldsymbol{t}'$ differ by an adjacent-transposition of their last two symbols. There are two cases to consider, depending on whether or not $\boldsymbol{t}' = \urcorner\llcorner(\mathbb{M}')$. If $\boldsymbol{t}' = \urcorner\llcorner(\mathbb{M}')$ then $\mathbf{L}'$ does not satisfy (2.28) since $\mathbf{L}'$ is non-empty. Otherwise, $\boldsymbol{s}'$ and $\boldsymbol{t}'$ can be expressed as follows

$$\boldsymbol{s}' = \mathsf{tail}_{\mathbb{M}'}(j, i) \text{ and } \boldsymbol{t}' = \overleftarrow{\mathsf{bubble}}(\boldsymbol{s}', n') = \mathsf{tail}_{\mathbb{M}'}(j, i-1)$$

for some $j \geq 2$ and $i = \underline{n}_{j-1} \geq 2$, where the last equality above follows from Lemma 2.4.16. Since $\boldsymbol{s}' \in \mathbf{L}'$ then $\mathsf{scut}_{\mathbb{M}'}(j, i) \in \mathsf{scuts}(\mathbf{L}')$. Since $\boldsymbol{t}' \notin \mathbf{L}'$ then either $\mathsf{scut}_{\mathbb{M}'}(j, i-1) \notin \mathsf{scuts}(\mathbf{L}')$ or $\mathbf{L}'/\mathsf{scut}_{\mathbb{M}'}(j, i-1)$ does not satisfy (2.28). Therefore, the first case concludes by noting that if $\boldsymbol{s}$ does not satisfy (2.1) then a quotient of $\mathbf{L}$ violates (2.28) or (2.29).

In the second case suppose that $\boldsymbol{s} \in \mathbf{L}$ does not satisfy (2.2). Therefore, there exists an h satisfying $|\bullet(\boldsymbol{s})| < h \leq |\urcorner\llcorner(\boldsymbol{s})|$ such that $\boldsymbol{t} = \overleftarrow{\mathsf{bubble}}(\boldsymbol{s}, h) \notin \mathbf{L}$. Let $\mathbf{L}' = \mathbf{L}/s_{h+1}s_{h+2}\cdots s_n$ (and so $n' = h$), $\boldsymbol{s}' = s_1 s_2 \cdots s_h$, and $\boldsymbol{t}' = t_1 t_2 \cdots t_h$. Notice that

$$\boldsymbol{s}' = \urcorner\llcorner(\mathbb{M}') \text{ and } \boldsymbol{t}' = \overleftarrow{\mathsf{bubble}}(\boldsymbol{s}', n') = \mathsf{tail}_{\mathbb{M}'}(2, 1)$$

where the last equality above follows from Lemma 2.4.18. Since $\boldsymbol{t} \notin \mathbf{L}$, then $\boldsymbol{t}' \notin \mathbf{L}'$. Therefore, if $\mathbf{L}'/\mathsf{scut}_{\mathbb{M}'}(2, 1)$ is assumed to satisfy (2.28), then $\mathsf{scut}_{\mathbb{M}'}(2, 1) \notin \mathsf{scuts}(\mathbf{L}')$. However, $|\mathbf{L}'| \geq 2$ due to the bound on h and so $\mathsf{scuts}(\mathbf{L}') \neq \varnothing$. Therefore, $\mathbf{L}'$ must violate either (2.29) or (2.30) since otherwise the existence of any scut in $\mathsf{scuts}(\mathbf{L}')$ would imply $\mathsf{scut}_{\mathbb{M}'}(2, 1) \notin \mathsf{scuts}(\mathbf{L}')$. Therefore, the second case is concluded by noting that if $\boldsymbol{s}$ does not satisfy (2.2) then a quotient of $\mathbf{L}$ violates (2.28) or (2.29) or (2.30). □

Chapter 3

Cool-lex Order

"A list is only as strong as its weakest link."

- Don Knuth

This chapter introduces a new variation of lexicographic order named cool-lex order. Recursively, cool-lex order is similar to co-lex order with one key difference. Instead of ordering strings based on their last symbol, cool-lex instead orders strings based on their suffix known as a scut. This change in focus allows greater flexibility in arranging the strings in a given language. When a bubble language is expressed in cool-lex order then the resulting list is a left-shift Gray code. Furthermore, these left-shift Gray codes can be described one string at a time by a simple iterative operation known as a cool left-shift. Besides being relatively easy to generate, these left-shift Gray codes also have interesting properties related to universal cycles.

Section 3.1 begins this chapter by focusing on the cool left-shift operation. This operation is based on the concept of a greedy left-shift. Section 3.2 then provides the recursive definition of cool-lex order. In Section 3.3 it is proven that the cool-left shift operation circularly generates the cool-lex order of any bubble language. In other words, Sections 3.1 and 3.2 use different approaches to describe the same left-shift Gray code for bubble languages. Section 3.4 discusses properties of these left-shift Gray codes, including the reverse cool-lex order for bubble languages, and its surprising shorthand rotation property. The reader is reminded that Appendix A provides a reference for notation.

3.1 Iterative Order

While bubble languages are defined in terms of left-shifting a symbol past one differing symbol, the Gray code given in this chapter relies on greedily left-shifting a symbol

past as many symbols as possible. *Greedy left-shifts* are defined in Section 3.1.1 and *cool left-shifts* are defined in Section 3.1.2. Section 3.1.3 then provides invariants for these two types of shifts. These invariants are useful for motivating the recursive definition found in Section 3.2 and for proving the equivalence found in Section 3.3.

3.1.1 Greedy Left-Shifts

While a bubble left-shift was defined solely using a string, the greedy left-shift is defined on a string that is within a given language. This definition appears below and is followed by several examples. The section concludes with a remark involving tails.

Definition 3.1.1 (Greedy left-shift ($\overleftarrow{\mathsf{greedy}}$))**.** *Given $\boldsymbol{s} \in \mathbf{L}$ and an index j, the* greedy left-shift *of the jth symbol in $\boldsymbol{s}$ is*

$$\overleftarrow{\mathsf{greedy}}_{\mathbf{L}}(\boldsymbol{s}, j) = \overleftarrow{\mathsf{shift}}(\boldsymbol{s}, j, i)$$

where i satisfies

- $\overleftarrow{\mathsf{shift}}(\boldsymbol{s}, j, h) \in \mathbf{L}$ *for all h within $i \leq h \leq j$*
- $i = 1$ *or* $\overleftarrow{\mathsf{shift}}(\boldsymbol{s}, j, i-1) \notin \mathbf{L}$.

In other words, $\overleftarrow{\mathsf{greedy}}$ *left-shifts a symbol in a string as far as possible while maintaining the property that every intermediate shift is in the given language. By convention* $\overleftarrow{\mathsf{greedy}}(\boldsymbol{s}, j) = \overleftarrow{\mathsf{greedy}}_{\mathbf{L}}(\boldsymbol{s}, j)$ *so the language is assumed to be* $\mathbf{L}$ *unless otherwise stated.*

The definition is now illustrated by three simple examples using the language $\mathbf{L} = \mathbf{T}(\mathbb{M})$ (ordered trees with fixed branching sequence) with $\mathbb{M} = \{0, 0, 0, 0, 1, 2, 2, 3\}$, the string $\boldsymbol{s} = 32100020$, and three different indices

$\overleftarrow{\mathsf{greedy}}(32100020, 6)$	$\overleftarrow{\mathsf{greedy}}(32100020, 7)$	$\overleftarrow{\mathsf{greedy}}(32100020, 8)$
$= 3\overleftarrow{2100020}$	$= \overleftarrow{32100020}$	$= 3210002\,\overleftarrow{0}$
$= 30210020$	$= 23210000$	$= 32100020.$

In the first example, no further left-shift of the 0 is possible since 0 is an invalid prefix in the language due to the fact that its sum is less than its length. In the second example, the 2 can be greedily left-shifted all the way into the first position. In the third example, the 0 cannot be greedily left-shifted beyond its current location

because $3210002\overleftarrow{0} = 32100002 \notin \mathbf{L}$ due to the fact that the prefix 3210000 has a smaller sum than length.

As a counterpoint to the previous examples, the next three examples consider the same string $\boldsymbol{s} = 412321411232$, the same index 6, and three different languages. From left-to-right the languages are multiset necklaces ($\mathbf{L} = \mathbf{N}(\mathbb{M})$), Lyndon words ($\mathbf{L} = \mathbf{N}^{-}(\mathbb{M})$), and permutations ($\Pi(\mathbb{M})$), all where $\mathbb{M} = \{0, 0, 0, 0, 1, 2, 2, 3\}$

$\overleftarrow{\mathsf{greedy}}(412321411232, 6)$	$\overleftarrow{\mathsf{greedy}}(412321411232, 6)$	$\overleftarrow{\mathsf{greedy}}(412321411232, 6)$
$= 41\overleftarrow{2321}411232$	$= 412\overleftarrow{321}411232$	$= \overleftarrow{412321}411232$
$= 411232411232$	$= 412132411232$	$= 141232411232.$

In the first example, no further left-shift of the 1 is possible since necklaces must begin with the largest symbol. In the second example, no further left-shift of the 1 is possible since $412\overleftarrow{321}1411232 = 411232411232$ is not aperiodic. In the third example, the 1 can be greedily left-shifted into the first position since multiset permutations contain all arrangements of the given multiset of symbols. The difference between the above results help explain how the cool left-shift given in Section 3.1.2 can generate Gray codes for many different languages.

This section concludes with a small observation about tails. As previously discussed in Chapter 2, the symbol following the non-increasing prefix of a string in a bubble language can be left-shifted until it joins the non-increasing prefix. Not only is the resulting string in the language, but so is the result of each intermediate shift. For example, when $\mathbf{L}$ is a bubble language and $\mathbb{M} = \{1, 1, 2, 2, 3, 3, 4, 4, 5, 5\}$ then

$$\begin{aligned}\overleftarrow{\mathsf{greedy}}(\mathsf{tail}(3,3), k+1) &= \overleftarrow{\mathsf{greedy}}(5544322131, 9)\\ &= \overleftarrow{\mathsf{greedy}}(5544\overleftarrow{322131}, 5)\\ &= \overleftarrow{\mathsf{greedy}}(5544332211, 4)\end{aligned}$$

where k represents the length of the non-increasing prefix in the initial string $\mathsf{tail}(3, 3)$. Notice that the greedy left-shift can be reexpressed to as a greedy left-shift within the non-increasing string, and this fact is independent of the particular bubble language being considered. This fact is formalized by the following remark.

Remark 3.1.2 (Greedy left-shifting the $(k+1)$st symbol in tails)**.** *Suppose* $\mathbf{L}$ *is a bubble language,* $\mathsf{tail}(j, i) \in \mathbf{L}$*,* $k = |\mathsf{L}(\mathsf{tail}(j, i))|$*, and* $h = \overline{n}_{j+1}$*. Then,*

$$\overleftarrow{\mathsf{greedy}}(\mathsf{tail}(j, i), k+1) = \overleftarrow{\mathsf{greedy}}(\mathsf{L}, h+1)$$

In other words, a greedy left-shift of the first symbol in the scut of a tail can be reexpressed as a greedy left-shift within the non-increasing string.

3.1.2 Cool Left-Shifts

The cool left-shift is defined below for any string within a bubble language. Essentially it is a greedy left-shift applied to one of the two symbols that follow the non-increasing prefix of the given string. Determining which symbol is shifted depends only on the relative values of the last symbol in the non-increasing prefix and the second symbol following the non-increasing prefix. (Special cases arise when the non-increasing prefix includes the entire string, or the entire string except the last symbol.) Remarkably, Theorem 3.3.5 on page 126 will prove that this simple operation generates any bubble language. The reader is also pointed to the pseudocode on page 128 to see how the operation can be used to create a simple program to generate any bubble language.

Definition 3.1.3 (Cool Left-Shift)**.** *Given* $\boldsymbol{s} \in \mathbf{L}$ *where* $\mathbf{L}$ *is a bubble language and* $k = |\mathcal{L}(\boldsymbol{s})|$*, the* cool-left shift *denoted* $\overleftarrow{\mathsf{cool}}_{\mathbf{L}}(\boldsymbol{s})$ *is*

$$= \begin{cases} \overleftarrow{\mathsf{greedy}}(\boldsymbol{s}, n) & \text{if } k = n \text{ or } k = n-1 \quad (3.1a) \\ \overleftarrow{\mathsf{greedy}}(\boldsymbol{s}, k+1) & \text{if } k \le n-2 \text{ and } (s_k < s_{k+2} \text{ or } \overleftarrow{\mathsf{greedy}}(\boldsymbol{s}, k+2) = \boldsymbol{s}) \quad (3.1b) \\ \overleftarrow{\mathsf{greedy}}(\boldsymbol{s}, k+2) & \text{otherwise } (\text{(3.1a) and (3.1b) do not hold}). \quad (3.1c) \end{cases}$$

In other words, $\overleftarrow{\mathsf{cool}}$ *greedily left-shifts the second symbol in the scut unless that symbol doesn't exist* ($k = n$ *or* $k = n-1$) *or it is larger than some symbol to its left* ($s_k < s_{k+2}$) *or the shift would be trivial* ($\overleftarrow{\mathsf{greedy}}(\boldsymbol{s}, k+2) = \boldsymbol{s}$)*. Otherwise,* $\overleftarrow{\mathsf{cool}}$ *greedy left-shifts the first symbol in the scut or the last symbol if the string is entirely non-increasing* ($k = n$)*.*

The definition is now illustrated by way of five examples using the language $\mathbf{L} = \mathbf{N}(\mathbb{M})$ (multiset necklaces) with $\mathbb{M} = \{1, 1, 2, 2, 3, 3, 4, 4\}$. As per the definition, the value of k will be used to represent $|\mathcal{L}(\boldsymbol{s})|$, and this value changes depending upon the example being discussed. The first two examples illustrate special cases when the non-increasing prefix covers the entire string or every symbol in the string except the last. That is, s_n is shifted when $k = n$ or $k = n-1$. In both situations a greedy

left-shift of the last symbol in the string is performed

$$\begin{aligned}
&\overleftarrow{\mathsf{cool}}(44332211) && \text{note } k = n \\
&= \overleftarrow{\mathsf{greedy}}(44332211, 8) && \text{by (3.1a)} \\
&= 44\overleftarrow{3321}1 \\
&= 44133221
\end{aligned}
\qquad
\begin{aligned}
&\overleftarrow{\mathsf{cool}}(44322113) && \text{note } k = n-1 \\
&= \overleftarrow{\mathsf{greedy}}(44322113, 8) && \text{by (3.1b)} \\
&= 4\overleftarrow{4322113} \\
&= 43432211.
\end{aligned}$$

The next example illustrates when the symbol following the non-increasing prefix is greedily left-shifted due to the last symbol in the non-increasing prefix being less than the second symbol following the non-increasing prefix. That is, s_{k+1} is shifted when $s_k < s_{k+2}$. (Since s_k is the smallest symbol within the first $k+1$ symbols of $\boldsymbol{s}$, this case is equivalent to stating that there is some smaller symbol to the left of s_{k+2} within $\boldsymbol{s}$.) In particular, the example below has $s_k = 1$ and $s_{k+2} = 2$ since the first $k+2$ are 4432132

$$\begin{aligned}
&\overleftarrow{\mathsf{cool}}(44321321) && \text{note } s_k < s_{k+2} \\
&= \overleftarrow{\mathsf{greedy}}(44321321, 6) && \text{by (3.1b)} \\
&= 4\overleftarrow{4321}321 \\
&= 43432121.
\end{aligned}$$

The symbol following the non-increasing prefix is also greedily left-shifted when the second symbol following the non-increasing prefix cannot be greedily left-shifted to create a new string in the language. That is, s_{k+1} is shifted when $\overleftarrow{\mathsf{greedy}}(\boldsymbol{s}, k+2) = \boldsymbol{s}$. (This case is equivalent to stating that $\overleftarrow{\mathsf{bubble}}(\boldsymbol{s}, k) \notin \mathbf{L}$.)

$$\begin{aligned}
&\overleftarrow{\mathsf{cool}}(42314231) && \text{note } \overleftarrow{\mathsf{greedy}}(\boldsymbol{s}, k+2) = \boldsymbol{s} \\
&= \overleftarrow{\mathsf{greedy}}(42314231, 3) && \text{by (3.1b)} \\
&= 4\overleftarrow{23}14231 \\
&= 43214231.
\end{aligned}$$

In the example above, the first $k+2$ symbols are 4231 and so $s_{k+2} = 1$. The symbol cannot be greedily left-shifted to create another string in the language because the shortest such shift produces $42\overleftarrow{31}4231 = 42134231 \notin \mathbf{L}$.

The final example illustrates when the second symbol following the non-increasing prefix is greedily left-shifted. In order for this to happen, the symbol must exist, there

must be no smaller symbol to its left within the string, and greedily left-shifting the symbol must produce a different string in the language. That is, s_{k+2} is shifted when $k+2 \le n$, $s_k \ge s_{k+2}$, and $\overleftarrow{\mathsf{greedy}}(\boldsymbol{s}, k+2) \neq \boldsymbol{s}$

$$\begin{aligned}
&\overleftarrow{\mathsf{cool}}(43214123) && \text{note } k+2 \le n,\ s_k \ge s_{k+2}, \text{ and } \overleftarrow{\mathsf{greedy}}(\boldsymbol{s}, k+2) \neq \boldsymbol{s} \\
&= \overleftarrow{\mathsf{greedy}}(43214123, 3) && \text{by (3.1c)} \\
&= 43\overleftarrow{2141}23 \\
&= 43121423.
\end{aligned}$$

When $\overleftarrow{\mathsf{cool}}$ is applied to specific languages it can often be simplified. The same is also true when $\overleftarrow{\mathsf{cool}}$ is applied to bubble languages that have fixed-density. These simplifications are necessary when trying to optimize the efficiency of the associated generation algorithms.

To conclude this section, the following lists illustrate that the strings in $\mathbf{N}(\mathbb{M})$ (multiset necklaces) and $\mathbf{N}^{-}(\mathbb{M})$ (Lyndon words) with $\mathbb{M} = \{1,1,2,2,3,3\}$ are circularly generated by the $\overleftarrow{\mathsf{cool}}$ operation. In each table, the left column contains a string $\boldsymbol{s} \in \mathbf{L}$, the middle column contains the condition that $\boldsymbol{s}$ satisfies with respect to Definition 3.1.3 and the given language $\mathbf{L}$, and the right column contains the corresponding equation number from Definition 3.1.3. The string in each row is transformed into the string in the row below it by applying $\overleftarrow{\mathsf{cool}}$, with the arrow representing the actual left-shift. The string in the last row is transformed into the string in the first row in the same manner. Of particular interest are the strings 323121 and 323112, which produce different successors due to the difference between the necklaces and Lyndon words.

3.1.3 Invariants

This section provides an *invariant* for greedy left-shifts, and then extends it to an invariant for cool left-shifts applied to bubble languages. To illustrate the idea, consider the following two greedy left-shifts involving the language $\mathbf{L} = \mathbf{N}(\mathbb{M})$ (multiset necklaces) with $\mathbb{M} = \{0,0,1,1,2,2,3,3\}$ and $\mathbf{L}' = \mathbf{L}/\boldsymbol{z}$ with $\boldsymbol{z} = 301$.

$$\begin{aligned}
\overleftarrow{\mathsf{greedy}}_{\mathbf{L}}(31220301, 5) &= 31\overleftarrow{220}301 \\
&= 31022301
\end{aligned}
\qquad
\begin{aligned}
\overleftarrow{\mathsf{greedy}}_{\mathbf{L}'}(31220, 5) \cdot 301 &= 31\overleftarrow{220} \cdot 301 \\
&= 31022301.
\end{aligned}$$

$\mathbf{L} = \mathbf{N}(\{1,1,2,2,3,3\})$				$\mathbf{L} = \mathbf{N}^{-}(\{1,1,2,2,3,3\})$		
$\boldsymbol{s}$	condition	$\overleftarrow{\mathsf{cool}}$		$\boldsymbol{s}$	condition	$\overleftarrow{\mathsf{cool}}$
$3\overleftarrow{31221}$	$s_k < s_{k+2}$	(3.1b)		$3\overleftarrow{31221}$	$s_k < s_{k+2}$	(3.1b)
$3\overleftarrow{23121}$		(3.1c)		$3\overleftarrow{23121}$	$\overleftarrow{\mathsf{greedy}}(\boldsymbol{s}, k+2) = \boldsymbol{s}$	(3.1b)
$\overleftarrow{321321}$	$s_k < s_{k+2}$	(3.1b)				
$33\overleftarrow{2121}$		(3.1c)		$33\overleftarrow{2121}$		(3.1c)
$3\overleftarrow{31212}$		(3.1c)		$3\overleftarrow{31212}$		(3.1c)
$\overleftarrow{313122}$	$\overleftarrow{\mathsf{greedy}}(\boldsymbol{s}, k+2) = \boldsymbol{s}$	(3.1b)		$\overleftarrow{313122}$	$\overleftarrow{\mathsf{greedy}}(\boldsymbol{s}, k+2) = \boldsymbol{s}$	(3.1b)
$3\overleftarrow{31122}$	$s_k < s_{k+2}$	(3.1b)		$3\overleftarrow{31122}$	$s_k < s_{k+2}$	(3.1b)
$3\overleftarrow{23112}$		(3.1c)		$3\overleftarrow{23112}$		(3.1c)
$\overleftarrow{312312}$	$s_k < s_{k+2}$	(3.1b)				
$\overleftarrow{321312}$	$\overleftarrow{\mathsf{greedy}}(\boldsymbol{s}, k+2) = \boldsymbol{s}$	(3.1b)		$\overleftarrow{321312}$	$\overleftarrow{\mathsf{greedy}}(\boldsymbol{s}, k+2) = \boldsymbol{s}$	(3.1b)
$33\overleftarrow{2112}$	$k = n-1$	(3.1a)		$33\overleftarrow{2112}$	$k = n-1$	(3.1a)
$32\overleftarrow{3211}$		(3.1c)		$32\overleftarrow{3211}$		(3.1c)
$32\overleftarrow{2311}$		(3.1c)		$32\overleftarrow{2311}$		(3.1c)
$3\overleftarrow{21231}$	$s_k < s_{k+2}$	(3.1b)		$3\overleftarrow{21231}$	$s_k < s_{k+2}$	(3.1b)
$\overleftarrow{322131}$	$\overleftarrow{\mathsf{greedy}}(\boldsymbol{s}, k+2) = \boldsymbol{s}$	(3.1b)		$\overleftarrow{322131}$	$\overleftarrow{\mathsf{greedy}}(\boldsymbol{s}, k+2) = \boldsymbol{s}$	(3.1b)
$33\overleftarrow{2211}$	$k = n$	(3.1a)		$33\overleftarrow{2211}$	$k = n$	(3.1a)

Table 3.1: Necklaces (left) and Lyndon words (right) over $\{1,1,2,2,3,3\}$ in cool-lex order.

In the first case, the greedy left-shift is applied to $31220301 \in \mathbf{L}$. In the second case, the greedy left-shift is applied to the portion of that string that is within $\mathbf{L}'$, and then $\boldsymbol{z}$ is suffixed to the result. In the two cases the resulting strings are equal. In general, this will always be true when greedily left-shifting a symbol that is to the left of the suffix $\boldsymbol{z}$. This is because any such left-shift will preserve the suffix $\boldsymbol{z}$, and since $\boldsymbol{s}' \cdot \boldsymbol{z} \in \mathbf{L}$ if and only if $\boldsymbol{s}' \in \mathbf{L}/\boldsymbol{z}$. The following remark states the precise result.

Remark 3.1.4 (Greedy left-shift invariant)**.** *Suppose* $\mathbf{L}' = \mathbf{L}/\boldsymbol{z}$*,* $\boldsymbol{s}' \in \mathbf{L}'$*, and* j *is within* $1 \leq j \leq n'$*. Then,*

$$\overleftarrow{\mathsf{greedy}}_{\mathbf{L}}(\boldsymbol{s}' \cdot \boldsymbol{z}, j) = \overleftarrow{\mathsf{greedy}}_{\mathbf{L}'}(\boldsymbol{s}', j) \cdot \boldsymbol{z}.$$

In other words, greedy left-shifts can be applied within language quotients.

Given the result in Remark 3.1.4 it is now possible to develop a similar invariant for the cool left-shift. To illustrate the invariant, consider the following cool left-shifts applied to the bubble language $\mathbf{L} = \mathbf{T}(\mathbb{M})$ (ordered trees with fixed branching sequence) with $\mathbb{M} = \{0, 0, 0, 0, 1, 1, 2, 2, 3\}$, and $\mathbf{L}' = \mathbf{L}/\boldsymbol{z}$ with $\boldsymbol{z} = \mathsf{scut}(3, 4) = 200$

$\overleftarrow{\mathsf{cool}}_{\mathbf{L}}(311200200)$	$\overleftarrow{\mathsf{cool}}_{\mathbf{L}}(321001200)$	$\overleftarrow{\mathsf{cool}}_{\mathbf{L}}(321100200)$
$= 3\overleftarrow{11200200}$	$= \overleftarrow{321001200}$	$= 3\overleftarrow{21100200}$
$= 301120200$	$= 132100200$	$= 302110020$
$\overleftarrow{\mathsf{cool}}_{\mathbf{L}'}(311200) \cdot 200$	$\overleftarrow{\mathsf{cool}}_{\mathbf{L}'}(321001) \cdot 200$	$\overleftarrow{\mathsf{cool}}_{\mathbf{L}'}(321100) \cdot 200$
$= 3\overleftarrow{11200} \cdot 200$	$= \overleftarrow{321001} \cdot 200$	$= 3\overleftarrow{21100} \cdot 200$
$= 301120200$	$= 132100200$	$= 302110200.$

In each column $\overleftarrow{\mathsf{cool}}$ is applied to a string in $\mathbf{L}$, and then to the corresponding string in $\mathbf{L}'$ with the suffix $\boldsymbol{z}$ appended to the result. Notice that in the first two columns the resulting strings are identical. In the last column, the string is of the form $\mathsf{tail}(3, 4)$ and the resulting strings are not equal. In general, the following lemma shows that non-increasing prefixes are the only obstacle to the invariant. That is, adding or removing a scut does not alter the behaviour of $\overleftarrow{\mathsf{cool}}$, except when the portion without the scut is non-increasing. The proof relies on Remark 3.1.4 and the fact that special circumstances must apply when $\overleftarrow{\mathsf{cool}}$ shifts a symbol that is two symbols beyond the non-increasing prefix. Since bubble languages are closed under taking language quotients, the proven invariant is actually much stronger.

Lemma 3.1.5 (Cool left-shift invariant)**.** *Suppose* $\mathbf{L}$ *is a bubble language,* $\mathbf{L}' = \mathbf{L}/\mathsf{scut}(j,i)$*,* $s' \in \mathbf{L}'$*, and* $\boldsymbol{s}' \neq \llcorner(\mathbb{M}')$*. Then,*

$$\overleftarrow{\mathsf{cool}}_{\mathbf{L}}(\boldsymbol{s}' \cdot \mathsf{scut}(j,i)) = \overleftarrow{\mathsf{cool}}_{\mathbf{L}'}(\boldsymbol{s}') \cdot \mathsf{scut}(j,i).$$

In other words, the cool left-shift is invariant upon suffixing a scut unless the remaining symbols before the scut are arranged in non-increasing order.

Proof. To simplify the proof, and the left-side of the claimed equality, let $\boldsymbol{s} = \boldsymbol{s}' \cdot \mathsf{scut}(j,i)$. Notice that $\boldsymbol{s} \in \mathbf{L}$ because $\boldsymbol{s}' \in \mathbf{L}'$ and $\mathbf{L}' = \mathbf{L}/\mathsf{scut}(j,i)$. Let $k' = |\llcorner(\boldsymbol{s}')|$ and $k = |\llcorner(\boldsymbol{s})|$. Since $\boldsymbol{s}'$ is not non-increasing then $k = k'$, and in particular $k < n'$. There are two cases to consider.

Case One: $k \leq n' - 2$. In this case, the relevant symbols are within $\boldsymbol{s}'$. That is, $s'_k = s_k$ and $s'_{k+2} = s_{k+2}$. Therefore, the relevant greedy left-shifts are identical,. That is, $\overleftarrow{\mathsf{greedy}}_{\mathbf{L}'}(\boldsymbol{s}', h) \cdot \mathsf{scut}(j,i) = \overleftarrow{\mathsf{greedy}}_{\mathbf{L}}(\boldsymbol{s}, h)$ for $h \in \{k+1, k+2\}$ by Remark 3.1.4. In particular, both of these greedy left-shifts will be trivial or neither will be trivial. Therefore, the conditions used to determine which symbol is left-shifted by $\overleftarrow{\mathsf{cool}}$ will also be identical. More precisely, if $s_k < s_{k+2}$ or $\overleftarrow{\mathsf{greedy}}_{\mathbf{L}}(\boldsymbol{s}, k+2) = \boldsymbol{s}$ then

$$\overleftarrow{\mathsf{cool}}_{\mathbf{L}'}(\boldsymbol{s}') \cdot \mathsf{scut}(j,i) = \overleftarrow{\mathsf{greedy}}_{\mathbf{L}'}(\boldsymbol{s}', k+1) \cdot \mathsf{scut}(j,i) = \overleftarrow{\mathsf{greedy}}_{\mathbf{L}}(\boldsymbol{s}, k+1) = \overleftarrow{\mathsf{cool}}_{\mathbf{L}}(\boldsymbol{s})$$

and otherwise

$$\overleftarrow{\mathsf{cool}}_{\mathbf{L}'}(\boldsymbol{s}') \cdot \mathsf{scut}(j,i) = \overleftarrow{\mathsf{greedy}}_{\mathbf{L}'}(\boldsymbol{s}', k+2) \cdot \mathsf{scut}(j,i) = \overleftarrow{\mathsf{greedy}}_{\mathbf{L}}(\boldsymbol{s}, k+2) = \overleftarrow{\mathsf{cool}}_{\mathbf{L}}(\boldsymbol{s}).$$

Case Two: $k' = n' - 1$. Since s_{k+2} is not contained in $\boldsymbol{s}'$ then the previous argument does not work. In this case, the proof relies on the following claim

$$s_k < s_{k+2}. \tag{3.2}$$

This inequality follows from two facts. First, $k' = n' - 1$ implies that $\boldsymbol{s}'$ is non-increasing until its second-last symbol. Therefore, $s'_k = s_k$ is the smallest symbol within $\boldsymbol{s}'$. Second, $s_{k+2} = d_j$ because it is the first symbol within $\mathsf{scut}(j,i)$. Since $\mathsf{scut}(j,i)$ must exclude at least one smaller symbol then it must be that $s_k < s_{k+2}$ since s_k is the smallest symbol within $\boldsymbol{s}'$. The proof is now a result of the following

derivation

$$
\begin{aligned}
&\overleftarrow{\mathsf{cool}}_{\mathbf{L}'}(\boldsymbol{s}') \cdot \mathsf{scut}(j,i) \\
&= \overleftarrow{\mathsf{greedy}}_{\mathbf{L}'}(\boldsymbol{s}', n') \cdot \mathsf{scut}(j,i) && \text{by } k = n' - 1 \text{ and (3.1a)} \\
&= \overleftarrow{\mathsf{greedy}}_{\mathbf{L}}(\boldsymbol{s}' \cdot \mathsf{scut}(j,i), n') && \text{by Remark 3.1.4} \\
&= \overleftarrow{\mathsf{greedy}}_{\mathbf{L}}(\boldsymbol{s}, n') && \text{by } \boldsymbol{s} = \boldsymbol{s}' \cdot \mathsf{scut}(j,i) \\
&= \overleftarrow{\mathsf{greedy}}_{\mathbf{L}}(\boldsymbol{s}, k'+1) && \text{by } k' = n' - 1 \\
&= \overleftarrow{\mathsf{greedy}}_{\mathbf{L}}(\boldsymbol{s}, k+1) && \text{by } k = k' \\
&= \overleftarrow{\mathsf{cool}}_{\mathbf{L}}(\boldsymbol{s}) && \text{by } s_k < s_{k+2} \text{ and (3.1b).}
\end{aligned}
$$

□

3.2 Recursive Order

This section recursively defines the *cool-lex order* for bubble languages. The cool-lex order is created by specifying a list of scuts in Section 3.2.1, and then extending this to a list of strings in Section 3.2.2. Section 3.2.3 characterizes the first and last strings in these lists. Section 3.2.4 extends the results on first and last strings to sublists. The first two subsections also offer asides that explain how cool-lex order differs from *co-lexicographic order*.

As a preliminary step, the terms *order* and *list* should be clarified. In this thesis a *list* is a comma-separated sequence of distinct strings, and an *order* or *list* of a language is a list containing each string of the language. For example, if $\mathbf{L} = \{1110, 1101, 1011\}$ then the following list is an order of $\mathbf{L}$

$$\mathcal{T} = 1011, 1110, 1101.$$

As above, lists are represented using calligraphic typeface. Two strings $\boldsymbol{r}$ and $\boldsymbol{s}$ are *consecutive* in $\mathcal{T}$ if $\boldsymbol{r}$ is followed by $\boldsymbol{s}$, or it $\boldsymbol{r}$ is the last string in $\mathcal{T}$ and $\boldsymbol{s}$ is the first string in $\mathcal{T}$. While this thesis is focused on lists of fixed-content languages, the lists of scuts given in Section 3.2.1 involve sets of strings that do not have fixed-content.

3.2.1 Scut Order

The *cool-lex scut order* of a language is an order of the language's scuts. More specifically, $\mathcal{Z}(\mathbf{L})$ is an order of $\mathsf{scuts}(\mathbf{L})$ in which the first symbol of each scut appears in increasing order, and subject to this constraint the number of excluded symbols of each scut appears in increasing order. For example, the cool-lex scut order for $\mathbf{L} = \Pi(\mathbb{M})$ (multiset permutations) with $\mathbb{M} = \{1, 1, 2, 2, 3, 3, 4, 4\}$ appears below

$$\mathcal{Z}(\mathbf{L}) = 21, 2, 3211, 311, 31, 3, 432211, 42211, 4211, 411, 41, 4 \tag{3.3}$$

and can be reexpressed as follows

$$\mathcal{Z}(\mathbf{L}) = \mathsf{scut}(2,1), \mathsf{scut}(2,2), \mathsf{scut}(3,1), \mathsf{scut}(3,2), \mathsf{scut}(3,3), \mathsf{scut}(3,4),$$
$$\mathsf{scut}(4,1), \mathsf{scut}(4,2), \mathsf{scut}(4,3), \mathsf{scut}(4,4), \mathsf{scut}(4,5), \mathsf{scut}(4,6).$$

Within this order notice that the $\mathsf{scut}(j, i)$ values appear with increasing values of j, and then for each particular value of j the values of i appear in increasing order.

As a brief aside, it is interesting to point out that the cool-lex scut order is a slight variation of the scut order found in *co-lexicographic order* (or simply *co-lex order*). In co-lex order, strings appear in increasing lexicographic order when they are read from right-to-left. For example, the following list contains co-lex order for $\mathbf{P}(4)$ (balanced parentheses)

$$11110000, 11101000, 11011000, 10111000, 11100100, 11010100, 10110100,$$
$$11001100, 10101100, 11100010, 11010010, 10110010, 11001010, 10101010.$$

Notice that each string is in $\mathbf{P}(4)$ and that the strings are ordered lexicographically when read from right-to-left. (Also notice that consecutive strings can vary quite substantially such as 10111000 followed by 11100100.) As another example, the first string in the order for $\mathbf{L} = \Pi(\mathbb{M})$ (multiset permutations) with $\mathbb{M} = \{1, 1, 2, 2, 3, 3, 4, 4\}$ is 44332211, while the last string is 11223344. In co-lex order, the scuts are ordered by decreasing length followed by increasing first symbol. For example, the scuts in $\Pi(\mathbb{M})$ appear as follows in co-lex order as compared to the cool-lex order in (3.3)

$$432211, 42211, 3211, 4211, 311, 411, 21, 31, 41, 2, 3, 4.$$

One reason that cool-lex order has not been previously discovered is that co-lex order

is more commonly understood as an order based on the rightmost symbol instead of the scut. For example, in the co-lex order of the previously mentioned language $\mathbf{L}$ the rightmost symbols appear in the order $1, 2, 3, 4$.

Now the cool-lex scut order is formalized. In the definition, the $\bigcirc\!\!\!,$ is used to create a list from individual strings, and is illustrated after the definition.

Definition 3.2.1 (Cool-lex scut order ($\mathcal{Z}$))**.** *The* cool-lex scut order *for the scuts of a bubble language* $\mathbf{L}$ *is*

$$\mathcal{Z}(\mathbf{L}) = \bigcirc\!\!\!,_{j=2,3,\ldots,\clubsuit} \; \bigcirc\!\!\!,_{i=1,2,\ldots,\clubsuit(j)} \mathsf{scut}(j,i).$$

In other words, the cool-lex scut order lists a bubble language's scuts by increasing first symbol and then by increasing number of excluded smaller symbols.

To illustrate the use of $\bigcirc\!\!\!,$ within the definition, the previous list of $\mathcal{Z}(\mathbf{L})$ in (3.3) is reexpressed as follows

$$\begin{aligned}
\mathcal{Z}(\mathbf{L}) &= \bigcirc\!\!\!,_{j=2,3,\ldots,\clubsuit} \; \bigcirc\!\!\!,_{i=1,2,\ldots,\clubsuit(j)} \mathsf{scut}(j,i) \\
&= \bigcirc\!\!\!,_{i=1,2} \mathsf{scut}(2,i), \; \bigcirc\!\!\!,_{i=1,2,3,4} \mathsf{scut}(3,i), \; \bigcirc\!\!\!,_{i=1,2,3,4,5,6} \mathsf{scut}(4,i) \\
&= \mathsf{scut}(2,1),\ \mathsf{scut}(2,2),\ \mathsf{scut}(3,1),\ \mathsf{scut}(3,2),\ \mathsf{scut}(3,3),\ \mathsf{scut}(3,4), \\
&\qquad \mathsf{scut}(4,1),\ \mathsf{scut}(4,2),\ \mathsf{scut}(4,3),\ \mathsf{scut}(4,4),\ \mathsf{scut}(4,5),\ \mathsf{scut}(4,6).
\end{aligned}$$

In particular, the first scut in $\mathcal{Z}(\mathbf{L})$ will always be $\mathsf{scut}(2,1)$ and the last scut within $\mathcal{Z}(\mathbf{L})$ will always be $\mathsf{scut}(j,i)$ with $j = \clubsuit$ and $i = \clubsuit(j)$.

Before extending the cool-lex order from scuts to strings in Section 3.2.2, one aspect of Definition 3.2.1 should be clarified. Within the definition it is claimed that $\mathcal{Z}(\mathbf{L})$ is a list of $\mathsf{scuts}(\mathbf{L})$ when $\mathbf{L}$ is a bubble language. From the definitions of $\clubsuit$ and $\clubsuit(j)$ it is clear that $\mathcal{Z}(\mathbf{L})$ contains every scut within $\mathsf{scuts}(\mathbf{L})$. Furthermore, when $\mathbf{L}$ is a bubble language then Lemmas 2.4.19 and 2.4.20 imply that for every value of i and j within the given ranges there is a string in $\mathbf{L}$ that has scut $\mathsf{scut}(j,i)$. Therefore, when $\mathbf{L}$ is a bubble language then $\mathcal{Z}(\mathbf{L})$ is in fact a list of $\mathsf{scuts}(\mathbf{L})$ as claimed.

3.2.2 String Order

In this section the scut order in Section 3.2.1 is extended to strings. This is done recursively by ordering the strings based on the order of their scuts within the cool-lex

scut order. The unique string that does not have a scut, namely $\urcorner\llcorner$, is chosen to be the last string in cool-lex order. To formalize cool-lex notion, two pieces of notation need to be generalized. First, the $\cdot$ operation is generalized so that it can concatenate a given string to the end of every string in a list. This operation is also known as *suffixing*. For example, suffixing 12 to the end of every string in $\mathcal{T} = 1110, 1101, 1011$ is accomplished by the following

$$\begin{aligned}\mathcal{T} \cdot 12 &= 1110 \cdot 12, 1101 \cdot 12, 1011 \cdot 12 \\ &= 111012, 110112, 101112.\end{aligned}$$

Second, when a list appears underneath $\bigcirc\!\!\!\!,$ then the strings in the list will be considered in sequence. This convention is illustrated after the definition of cool-lex order below.

Definition 3.2.2 (Cool-lex order ($\overleftarrow{\mathcal{C}}$))**.** *The* cool-lex order *for non-empty bubble language* $\mathbf{L}$ *is*

$$\overleftarrow{\mathcal{C}}(\mathbf{L}) = \bigcirc\!\!\!\!, \underset{z \in \mathcal{Z}(\mathbf{L})}{} \overleftarrow{\mathcal{C}}(\mathbf{L}/z) \cdot z,\ \urcorner\llcorner.$$

In other words, cool-lex order has strings with the same scut ordered contiguously, the order of the scuts is by cool-lex scut order, and the non-increasing string is last.

For example, when $\mathbf{L} = \Pi(\mathbb{M})$ (multiset permutations) with $\mathbb{M} = \{1, 1, 2, 2, 3, 3, 4, 4\}$ then

$$\begin{aligned}\overleftarrow{\mathcal{C}}(\mathbf{L}) &= \bigcirc\!\!\!\!, \underset{z \in \mathcal{Z}(\mathbf{L})}{} \overleftarrow{\mathcal{C}}(\mathbf{L}/z) \cdot z,\ \urcorner\llcorner \\ &= \overleftarrow{\mathcal{C}}(\mathbf{L}/21) \cdot 21,\ \overleftarrow{\mathcal{C}}(\mathbf{L}/2) \cdot 2,\ \overleftarrow{\mathcal{C}}(\mathbf{L}/3211) \cdot 3211,\ \overleftarrow{\mathcal{C}}(\mathbf{L}/311) \cdot 311,\ \overleftarrow{\mathcal{C}}(\mathbf{L}/31) \cdot 31, \\ &\quad \overleftarrow{\mathcal{C}}(\mathbf{L}/3) \cdot 3,\ \overleftarrow{\mathcal{C}}(\mathbf{L}/432211) \cdot 432211,\ \overleftarrow{\mathcal{C}}(\mathbf{L}/42211) \cdot 42211,\ \overleftarrow{\mathcal{C}}(\mathbf{L}/4211) \cdot 4211, \\ &\quad \overleftarrow{\mathcal{C}}(\mathbf{L}/411) \cdot 411,\ \overleftarrow{\mathcal{C}}(\mathbf{L}/41) \cdot 41,\ \overleftarrow{\mathcal{C}}(\mathbf{L}/4) \cdot 4,\ 44332211.\end{aligned}$$

since

$$\mathcal{Z}(\mathbf{L}) = 21, 2, 3211, 311, 31, 3, 432211, 42211, 4211, 411, 41, 4.$$

Notice that the strings are grouped according to their scut, and within each of these groups the strings are recursively ordered in cool-lex order. The base case of this recursion occurs when $|\mathbf{L}| = 1$; in this case the non-increasing string is the only string in the language and the only string in the cool-lex list. (Recursively, co-lex order and cool-lex order are identical except for the order of the scuts, and the fact that

the non-increasing string appears first within co-lex and last within cool-lex.) This section concludes with the observation that $\overleftarrow{\mathcal{C}}(\mathbf{L})$ is in fact a list of $\mathbf{L}$ whenever $\mathbf{L}$ is a bubble language[1] This follows inductively from the fact that $\mathcal{Z}(\mathbf{L})$ contains every possible scut in $\mathsf{scuts}(\mathbf{L})$ and the fact that $\urcorner\llcorner \in \mathbf{L}$ by Remark 2.2.2.

3.2.3 First and Last

This section focuses on the first and last strings within each cool-lex list. Given a non-empty list $\mathcal{T}$, let $\mathsf{first}(\mathcal{T})$ and $\mathsf{last}(\mathcal{T})$ respectively represent the first and last strings in $\mathcal{T}$. The last string in cool-lex order is always the non-increasing string due to the placement of $\urcorner\llcorner$ within Definition 3.2.2. To understand the nature of the first string in cool-lex order, it is useful to consider an example language such as $\mathbf{L} = \mathbf{N}(\mathbb{M})$ (multiset necklaces) with $\mathbb{M} = \{1, 1, 1, 2, 3, 3, 3\}$. Within Definition 3.2.2 notice that there are two options for the first string. Either the first string has the scut $\mathsf{scut}(2, 1)$ or the language contains only one string and so the first string is equal to the last string which is non-increasing. In the current example, $\mathbf{L}$ contains more than one string, and so

$$\begin{aligned}\mathsf{first}(\overleftarrow{\mathcal{C}}(\mathbf{L})) &= \mathsf{first}(\overleftarrow{\mathcal{C}}(\mathbf{L}')) \cdot \mathsf{scut}_{\mathbb{M}}(2, 1) \\ &= \mathsf{first}(\overleftarrow{\mathcal{C}}(\mathbf{L}')) \cdot 211\end{aligned}$$

where $\mathbf{L}' = \mathbf{L}/\mathsf{scut}_{\mathbb{M}}(2, 1)$. Again, $\mathbf{L}'$ contains more than one string and so the same argument can be repeated

$$\begin{aligned}\mathsf{first}(\overleftarrow{\mathcal{C}}(\mathbf{L}')) &= \mathsf{first}(\overleftarrow{\mathcal{C}}(\mathbf{L}'')) \cdot \mathsf{scut}_{\mathbb{M}'}(2, 1) \\ &= \mathsf{first}(\overleftarrow{\mathcal{C}}(\mathbf{L}'')) \cdot 3\end{aligned}$$

where $\mathbf{L}'' = \mathbf{L}'/\mathsf{scut}_{\mathbb{M}'}(2, 1)$. This process would be continued again except for the fact that $\mathbf{L}''$ contains only the non-increasing string 331. Therefore,

$$\mathsf{first}(\overleftarrow{\mathcal{C}}(\mathbf{L}'')) = 331.$$

[1] Chapter 5 discusses the cool-lex list of any fixed-content language.

By using the previous steps, the conclusion is that

$$\begin{aligned}\mathsf{first}(\overleftarrow{\mathcal{C}}(\mathbf{L})) &= \mathsf{first}(\overleftarrow{\mathcal{C}}(\mathbf{L}'')) \cdot \mathsf{scut}_{\mathbb{M}'}(2,1) \cdot \mathsf{scut}_{\mathbb{M}}(2,1) \\ &= 331 \cdot 3 \cdot 211 \\ &= 3313211.\end{aligned}$$

One way to interpret the above string is that it was obtained by greedily left-shifting the smallest symbol in the non-increasing string. That is,

$$\begin{aligned}\mathsf{first}(\overleftarrow{\mathcal{C}}(\mathbf{L})) &= \overleftarrow{\mathsf{greedy}}(\urcorner\llcorner, n) \\ &= \overleftarrow{\mathsf{greedy}}(332211, 6) \\ &= 33\overleftarrow{32111} \\ &= 3313211.\end{aligned}$$

To prove this fact in general, it is useful to introduce the following lemma.

Lemma 3.2.3 (Greedy left-shift)**.** *Suppose* $\mathbf{L}$ *is a bubble language,* $|\mathbf{L}| > 1$*, and* $\mathbf{L}' = \mathbf{L}/\mathsf{scut}(2,1)$*. Then,*

$$\overleftarrow{\mathsf{greedy}}(\urcorner\llcorner, n) = \overleftarrow{\mathsf{greedy}}(\mathsf{tail}(2,1), n').$$

In other words, when a bubble language contains more than one string, then a greedy left-shift of the last symbol in the non-increasing prefix of the tali with scut $\mathsf{scut}(2,1)$*.*

Proof. Since $\mathbf{L}$ is a bubble language and $|\mathbf{L}| > 0$ then $\urcorner\llcorner \in \mathbf{L}$ by Remark 2.2.2. Furthermore, since $|\mathbf{L}| > 1$ then $|\text{🌡}(\urcorner\llcorner)| < n$ since otherwise no rearrangements of the symbols could produce another string in the language. Therefore, by (2.2) in Definition 2.2.1 it must be that

$$\overleftarrow{\mathsf{bubble}}(\urcorner\llcorner, n) \in \mathbf{L}.$$

However, $n = \overline{n}_1$ so the above bubble-left shift produces $\mathsf{tail}(2,1)$ by Lemma 2.4.18. Furthermore, the shifted symbol is moved into position n' which is the last position of the non-increasing prefix of this tail. Therefore,

$$\overleftarrow{\mathsf{greedy}}(\urcorner\llcorner, n) = \overleftarrow{\mathsf{greedy}}(\mathsf{tail}(2,1), n').$$

□

Now the main result of this section is proven.

Lemma 3.2.4 (First and last strings)**.** *Suppose* $\mathbf{L}$ *is a non-empty bubble language. Then,*

$$\mathsf{last}(\overleftarrow{\mathcal{C}}(\mathbf{L})) = \curlywedge \qquad\qquad \mathsf{first}(\overleftarrow{\mathcal{C}}(\mathbf{L})) = \overleftarrow{\mathsf{greedy}}(\curlywedge, n).$$

In other words, in the cool-lex order for any bubble language, the last string is non-increasing, and the first string is obtained by greedily left-shifting the last symbol in the non-increasing string.

Proof. Let $\mathcal{T} = \overleftarrow{\mathcal{C}}(\mathbf{L})$. The first equality comes directly from Definition 3.2.2. The second equality is proven by induction on $|\mathbf{L}|$. If $|\mathbf{L}| = 1$ then Definition 3.2.2 implies that

$$\mathsf{first}(\mathcal{T}) = \curlywedge.$$

Furthermore, $\overleftarrow{\mathsf{greedy}}(\curlywedge, =)\curlywedge$ in this case since $\curlywedge$ is the only string in the language by Remark 2.2.2. Therefore, the base case of $|\mathbf{L}| = 1$ is true. Therefore, assume the result is true when $|\mathbf{L}| \leq h$. If $|\mathbf{L}| = h + 1$ then let $\mathbf{L}' = \mathbf{L}/\mathsf{scut}(2,1)$ and $\mathcal{T}' = \overleftarrow{\mathcal{C}}(\mathbf{L})$. This language is well-defined since if $\mathbf{L}$ contains more than one string then there must be at least two distinct symbols within $\mathbb{M}$. Furthermore, $\mathbf{L}'$ must be non-empty since if $\mathbf{L}$ contains more than one string, then it must contain a string with a scut, and by Lemmas 2.4.19 and Lemmas 2.4.20 it must that $\mathsf{scut}(2,1) \in \mathsf{scuts}(\mathbf{L})$. Finally, $|\mathbf{L}'| < |\mathbf{L}|$ since $\curlywedge \in \mathbf{L}$ by Remark 2.2.2 and $\curlywedge$ does not have $\mathsf{scut}(2,1)$ as a suffix. Therefore, the inductive assumption can be applied to $\mathcal{T}'$ to give the following result

$$\mathsf{first}(\mathcal{T}') = \overleftarrow{\mathsf{greedy}}(\curlywedge(\mathbb{M}'), n')$$

Therefore, by Definition 3.2.2

$$\mathsf{first}(\mathcal{T}) = \overleftarrow{\mathsf{greedy}}(\curlywedge(\mathbb{M}'), n') \cdot \mathsf{scut}(2,1) = \overleftarrow{\mathsf{greedy}}(\mathsf{tail}(2,1), n') = \overleftarrow{\mathsf{greedy}}(\curlywedge, n)$$

where the last equality follows from Lemma 3.2.3. □

Chapter 5 points out that the similarity between the first and last strings allows circular cool-lex Gray codes to be combined in various ways to create Gray codes for certain non-fixed-content languages.

3.2.4 Heads and Tails

The previous subsection formalized the first and last strings in cool-lex lists for bubble languages, and this section focuses on the first and last strings in each cool-lex sublist. The term *sublist* will be used to refer to lists of the form $\overleftarrow{\mathcal{C}}(\mathbf{L}/\mathsf{scut}(j,i))\cdot\mathsf{scut}(j,i)$ (as seen in Definition 3.2.2) for bubble languages $\mathbf{L}$. Since bubble languages are closed under quotients by Theorem 2.4.2, the first and last strings in these sublists follow immediately from Lemma 3.2.4. The first and last strings in these sublists will be called heads and tails respectively. Tails were first seen in Section 2.4.3 and are simply non-increasing prefixes followed by scuts, while heads are introduced here. The only result requiring proof in this section is Lemma 3.2.8, which formalizes which greedy left-shift is used when applying the cool-left shift to a tail.

Definition 3.2.5 (Heads)**.** *The* head *with scut* $\mathsf{scut}(j,i)$ *is*

$$\mathsf{head}_{\mathbf{L}}(j,i) = \overleftarrow{\mathsf{greedy}}(\mathsf{tail}(j,i),k)$$

where $k = |\llcorner(\mathsf{tail}(j,i))|$*. In other words, the head with scut* $\mathsf{scut}(j,i)$ *is obtained from the tail with scut* $\mathsf{scut}(j,i)$ *by greedily left-shifting the last symbol in its non-increasing prefix.*

Remark 3.2.6 (First in sublists are heads)**.** *Suppose* $\mathbf{L}$ *is a bubble language and* $\mathbf{L}' = \mathbf{L}/\mathsf{scut}(j,i)$*. Then,*

$$\mathsf{first}(\overleftarrow{\mathcal{C}}(\mathbf{L}')\cdot\mathsf{scut}(j,i)) = \mathsf{head}(j,i).$$

In other words, when $\mathbf{L}$ *is a bubble language then the last string with scut* $\mathsf{scut}(j,i)$ *in its cool-lex order is* $\mathsf{head}(j,i)$*.*

Remark 3.2.7 (Last in sublists are tails)**.** *Suppose* $\mathbf{L}$ *is a bubble language and* $\mathbf{L}' = \mathbf{L}/\mathsf{scut}(j,i)$*. Then,*

$$\mathsf{last}(\overleftarrow{\mathcal{C}}(\mathbf{L}')\cdot\mathsf{scut}(j,i)) = \mathsf{tail}(j,i).$$

In other words, when $\mathbf{L}$ *is a bubble language then the last string with scut* $\mathsf{scut}(j,i)$ *in its cool-lex order is* $\mathsf{tail}(j,i)$*.*

The following lemma specializes the cool left-shift to tails. To illustrate the lemma,

consider the language $\mathbf{L} = \mathbf{N}(\mathbb{M})$ (multiset necklaces) with $\mathbb{M} = \{1,1,2,2,3,3,4,4\}$

$\overleftarrow{\mathsf{cool}}(\mathsf{tail}(3, \clubsuit(3)))$	$\overleftarrow{\mathsf{cool}}(\mathsf{tail}(4, \clubsuit(4)))$	$\overleftarrow{\mathsf{cool}}(\mathsf{tail}(4, \clubsuit(4)-1))$
$= \overleftarrow{\mathsf{cool}}(\mathsf{tail}(3,4))$	$\overleftarrow{\mathsf{cool}}(\mathsf{tail}(4,5))$	$\overleftarrow{\mathsf{cool}}(\mathsf{tail}(4,4))$
$= \overleftarrow{\mathsf{cool}}(44322113)$	$= \overleftarrow{\mathsf{cool}}(43322141)$	$= \overleftarrow{\mathsf{cool}}(43322411)$
$= \overleftarrow{\mathsf{greedy}}(44322113, 8)$	$= \overleftarrow{\mathsf{greedy}}(43322141, 7)$	$= \overleftarrow{\mathsf{greedy}}(43322411, 7)$
$= \overleftarrow{\mathsf{greedy}}(44322113, k+1)$	$= \overleftarrow{\mathsf{greedy}}(43322141, k+1)$	$= \overleftarrow{\mathsf{greedy}}(43322411, k+2)$

In each of the columns, the value of k represents the length of the non-increasing prefix in the given example. (For instance $k = |\curlywedge(44322113)| = 7$ in the first column.) The first two columns illustrate cases when the $(k+1)$st symbol is greedily left-shifted by the cool left-shift. In the first column, $k = n-1$ and so the $(k+1)$st symbol is shifted due to (3.1a) in Definition 3.1.3. In the second column, the $(k+2)$nd symbol cannot be shifted to the left to result in another string in the language. That is, $\overleftarrow{\mathsf{greedy}}(43322141, k+2)$ is a trivial shift. Therefore, the $(k+1)$st symbol is shifted due to (3.1b) in Definition 3.1.3. These two examples represent the two situations in which the cool left-shift will greedily left-shift the first symbol in the scut when applied to a tail. (The remaining condition in (3.1b) is not possible to satisfy within a tail.) In both examples, notice that the string is equal to $\mathsf{tail}(j,i)$ where $i = \clubsuit(j)$. In the third column the string is equal to $\mathsf{tail}(j,i)$ where $i < \clubsuit(j)$ and the $(k+2)$nd symbol is greedily left-shifted by the cool left-shift. This result is now formalized by the following lemma.

Lemma 3.2.8 (Cool left-shift on tails)**.** *Suppose* $\mathbf{L}$ *is a bubble language,* $\mathsf{tail}(j,i) \in \mathbf{L}$*, and* $k = |\curlywedge(\mathsf{tail}(j,i))|$*. Then,*

$$\overleftarrow{\mathsf{cool}}(\mathsf{tail}(j,i)) = \begin{cases} \overleftarrow{\mathsf{greedy}}(\mathsf{tail}(j,i), k+1) & \text{if } i = \clubsuit(j) \\ \overleftarrow{\mathsf{greedy}}(\mathsf{tail}(j,i), k+2) & \text{otherwise } (i < \clubsuit(j)). \end{cases}$$

In other words, when the cool left-shift is applied to $\mathsf{tail}(j,i)$*, it greedily left-shifts the symbol after the non-increasing prefix if* $i = \clubsuit(j)$ *and otherwise greedily left-shifts the second symbol in the scut if* $i < \clubsuit(j)$*.*

Proof. Let $\boldsymbol{s} = \mathsf{tail}(j,i)$ and $k = |\curlywedge(\boldsymbol{s})|$. When $i = \clubsuit(j)$ then either $k = n-1$ or $\overleftarrow{\mathsf{shift}}(\boldsymbol{s}, k+2, k+1) \notin \mathbf{L}$. (Otherwise, $\mathsf{tail}(j, i+1) \in \mathbf{L}$ by Lemma 2.4.17 which would contradict the fact that $i = \clubsuit(j)$.) Therefore, $\overleftarrow{\mathsf{cool}}(\boldsymbol{s}) = \overleftarrow{\mathsf{greedy}}(\boldsymbol{s}, k+1)$ by (3.1b) in

Definition 3.1.3. On the other hand, when $i < \clubsuit(j)$ then $\overleftarrow{\mathsf{shift}}(\boldsymbol{s}, k+2, k+1) = \mathsf{tail}(j, i+1) \in \mathbf{L}$ and $s_k \geq s_{k+2}$ (since removing s_{k+1} from $\boldsymbol{s}$ would result in a non-increasing string). Therefore, $\overleftarrow{\mathsf{cool}}(\boldsymbol{s}) = \overleftarrow{\mathsf{greedy}}(\boldsymbol{s}, k+1)$ by (3.1c) in Definition 3.1.3. □

3.3 Gray code

This section proves that the cool left-shift *circularly generates* the cool-lex order for any bubble language. This means that if $\mathbf{L}$ is a bubble language and $\boldsymbol{s} \in \mathbf{L}$, then $\overleftarrow{\mathsf{cool}}(\boldsymbol{s})$ is the string after $\boldsymbol{s}$ in $\overleftarrow{\mathcal{C}}(\mathbf{L})$. Furthermore, if $\boldsymbol{s}$ is the last string in $\overleftarrow{\mathcal{C}}(\mathbf{L})$ then $\overleftarrow{\mathsf{cool}}(\boldsymbol{s})$ is the first string in $\overleftarrow{\mathcal{C}}(\mathbf{L})$. Thus,

$$\overleftarrow{\mathcal{C}}(\mathbf{L}) = \overleftarrow{\mathsf{cool}}(\boldsymbol{s}), \overleftarrow{\mathsf{cool}}_2(\boldsymbol{s}), \overleftarrow{\mathsf{cool}}_3(\boldsymbol{s}), \cdots, \overleftarrow{\mathsf{cool}}_{|\mathbf{L}|-1}(\boldsymbol{s}), \boldsymbol{s} \tag{3.4}$$

where $\boldsymbol{s} = \urcorner\llcorner$ and $\overleftarrow{\mathsf{cool}}_{|\mathbf{L}|}(\boldsymbol{s}) = \boldsymbol{s}$. This result is stated as Theorem 3.3.5 in Section 3.3.2. Since $\overleftarrow{\mathsf{cool}}$ always performs a left-shift, and since $\overleftarrow{\mathcal{C}}(\mathbf{L})$ contains every string in $\mathbf{L}$, then Theorem 3.3.5 proves that $\overleftarrow{\mathcal{C}}(\mathbf{L})$ is a *circular left-shift Gray code* that is generated by the $\overleftarrow{\mathsf{cool}}$ operation.

To give an overview of the proof, consider the following expression for $\overleftarrow{\mathcal{C}}(\mathbf{L})$ with $\mathbf{L} = \Pi(\mathbb{M})$ (multiset permutations) and $\mathbb{M} = \{1,1,2,2,3,3\}$

$$\overleftarrow{\mathcal{C}}(\mathbf{L}/21)\cdot 21,\ \overbrace{\overleftarrow{\mathcal{C}}(\mathbf{L}/2)\cdot 2,\ \overleftarrow{\mathcal{C}}(\mathbf{L}/3211)\cdot 3211,}^{\text{transition type } j}$$
$$\underbrace{\overleftarrow{\mathcal{C}}(\mathbf{L}/311)\cdot 311,\ \overleftarrow{\mathcal{C}}(\mathbf{L}/31)\cdot 31,}_{\text{transition type } i}\ \underbrace{\overleftarrow{\mathcal{C}}(\mathbf{L}/3)\cdot 3,\ 332211}_{\text{transition to } \urcorner\llcorner}.$$

Within this list there are essentially four distinct types of transitions, three of which are illustrated above. Transitions of type i are from a sublist with scut $\mathsf{scut}(j,i)$ to a sublist with scut $\mathsf{scut}(j,i+1)$. Transitions of type j are from a sublist with scut $\mathsf{scut}(j,i)$ to a sublist with scut $\mathsf{scut}(j+1,1)$. Finally there are two individual transitions involving the non-increasing string. The transition to $\urcorner\llcorner$ results in the last string in the list, while the from $\urcorner\llcorner$ results in the first string in the list. In each case it must be proven that the cool left-shift makes the proper modification to bridge these transitions. These results are contained in Section 3.3.1. It must also be shown that the cool left-shift makes the proper modifications within each sublist. This result follows from the cool left-shift invariant, and is discussed in the final proof of Section 3.3.2.

3.3.1 Transitions

This section shows how greedy left-shifts can transform certain tails into certain heads, certain tails into the non-increasing prefix, and the non-increasing prefix into a certain head. The results are then extended to the cool left-shift. Each of the four types of transitions are handled in a similar manner with the following conventions. Within the examples, the value of k will represent the length of the non-increasing prefix for each column. For example, if 3221000200 appears in the first column of an example then $k = 7$ within the first column. Also, within each transition the choices of i and j will imply that $\mathsf{tail}(j,i) \in \mathbf{L}$ due to Remark 2.4.12, and Lemmas 2.4.19 and 2.4.20. Furthermore, the fact that $\urcorner\llcorner \in \mathbf{L}$ follows from Remark 2.2.2 and is used in the last two transitions. These simple observations are necessary when discussing greedy left-shifts and cool left-shifts, and are not repeated for each separate case.

Transition Type i

The first type of transition takes $\mathsf{tail}(j,i)$ to $\mathsf{head}(j,i+1)$. This transition involves greedily left-shifting the second symbol in the scut, and requires the additional condition that this shift is non-trivial. To illustrate the transition consider the following examples with the language $\mathbf{L} = \mathbf{T}(\mathbb{M})$ (ordered trees with fixed branching sequence) with $\mathbb{M} = \{0,0,0,0,0,1,2,2,2,3\}$

$$
\begin{aligned}
&\overleftarrow{\mathsf{greedy}}(\mathsf{tail}(3,4),k+2)\\
&= \overleftarrow{\mathsf{greedy}}(3221000200,9)\\
&= \overleftarrow{\mathsf{greedy}}(322100020\overleftarrow{0},8)\\
&= \overleftarrow{\mathsf{greedy}}(3221000020,8)\\
&= \overleftarrow{\mathsf{greedy}}(\mathsf{tail}(3,5),8)\\
&= \mathsf{head}(3,5)
\end{aligned}
\qquad\qquad
\begin{aligned}
&\overleftarrow{\mathsf{greedy}}(\mathsf{tail}(4,3),k+2)\\
&= \overleftarrow{\mathsf{greedy}}(2223100000,5)\\
&= \overleftarrow{\mathsf{greedy}}(222\overleftarrow{3}100000,4)\\
&= \overleftarrow{\mathsf{greedy}}(2221300000,4)\\
&= \overleftarrow{\mathsf{greedy}}(\mathsf{tail}(4,4),4)\\
&= \mathsf{head}(4,4)
\end{aligned}
$$

In both cases notice that the symbol being greedily left-shifted is the second symbol in the scut. Since the greedy left-shift is non-trivial the symbol can be left-shifted at least one position to create an additional string in the language. (In particular, $3221000020 \in \mathbf{L}$ and $2221300000 \in \mathbf{L}$.) Therefore, the original greedy left-shift can be reexpressed as a second greedy left-shift. The initial left-shift also has two important effects. The first effect is that an additional symbol is excluded from the scut, which explains the change from i to $i+1$ in the transition. The second effect is that the

shifted symbol becomes the last symbol in the non-increasing prefix, which makes the reexpressed greedy left-shift equal to a head. When translating this result to the cool left-shift, recall from Lemma 3.2.8 that the $(k+2)$nd symbol is greedily left-shifted in $\mathsf{tail}(j,i)$ whenever $i < \clubsuit(j)$ as above. Therefore, the above greedy left-shifts are examples of cool left-shifts.

Lemma 3.3.1 (Transition i)**.** *Suppose* $\mathbf{L}$ *is a bubble language,* $j \leq \clubsuit$*,* $i < \clubsuit(j)$*. Then,*

$$\overleftarrow{\mathsf{cool}}(\mathsf{tail}(j,i)) = \mathsf{head}(j,i+1).$$

In other words, the cool left-shift transforms $\mathsf{tail}(j,i)$ *into* $\mathsf{head}(j,i+1)$ *as long as* $i < \clubsuit(j)$*.*

Proof. Let $k = |\llcorner(\mathsf{tail}(j,i))|$. First notice the following equality which follows from Remark 2.4.14

$$k+1 = |\llcorner(\mathsf{tail}(j,i+1))|. \tag{3.5}$$

The proof is now a result of the following derivation

$$\begin{aligned}
&\overleftarrow{\mathsf{cool}}(\mathsf{tail}(j,i)) \\
&= \overleftarrow{\mathsf{greedy}}(\mathsf{tail}(j,i), k+2) && \text{by Lemma 3.2.8} \\
&= \overleftarrow{\mathsf{greedy}}(\overleftarrow{\mathsf{shift}}(\mathsf{tail}(j,i), k+2, k+1), k+1) && \text{by non-trivial shift and Remark 2.4.15} \\
&= \overleftarrow{\mathsf{greedy}}(\mathsf{tail}(j,i+1), k+1) && \text{by Lemma 2.4.17} \\
&= \mathsf{head}(j,i+1) && \text{by (3.5).}
\end{aligned}$$

□

Transition Type j

The second type of transition takes $\mathsf{tail}(j,i)$ to $\mathsf{head}(j+1,1)$. This transition involves greedily left-shifting the first symbol in the scut, and requires the additional condition that $j < \clubsuit$. To illustrate the transition consider the following examples with the

language $\mathbf{L} = \mathbf{N}^{-}(\mathbb{M})$ (multiset Lyndon words) with $\mathbb{M} = \{1,1,2,2,3,3,4,4,5,5\}$

$$\begin{aligned}
&\overleftarrow{\mathsf{greedy}}(\mathsf{tail}(3,4), k+1) \\
&= \overleftarrow{\mathsf{greedy}}(5544322113, 10) \\
&= \overleftarrow{\mathsf{greedy}}(554\overleftarrow{4322113}, 5) \\
&= \overleftarrow{\mathsf{greedy}}(554\overleftarrow{4}332211, 4) \\
&= \overleftarrow{\mathsf{greedy}}(5543432211, 4) \\
&= \overleftarrow{\mathsf{greedy}}(\mathsf{tail}(4,1), 4) \\
&= \mathsf{head}(4,1)
\end{aligned}
\qquad
\begin{aligned}
&\overleftarrow{\mathsf{greedy}}(\mathsf{tail}(4,6), k+1) \\
&= \overleftarrow{\mathsf{greedy}}(5543322114, 10) \\
&= \overleftarrow{\mathsf{greedy}}(554\overleftarrow{3322114}, 3) \\
&= \overleftarrow{\mathsf{greedy}}(55\overleftarrow{4}4332211, 2) \\
&= \overleftarrow{\mathsf{greedy}}(5454332211, 2) \\
&= \overleftarrow{\mathsf{greedy}}(\mathsf{tail}(5,1), 2) \\
&= \mathsf{head}(5,1)
\end{aligned}$$

Notice that the symbol being greedily left-shifted is the first symbol in the scut and that the symbol is less than $\clubsuit$. (In particular, $\clubsuit = 5$ since $d_5 = 5$ is the largest symbol that is the first symbol of a scut in this language.) Since the symbol is the first symbol in the scut then it can be greedily left-shifted past every symbol that is less than or equal to itself. After this first intermediate shift the resulting string is entirely non-increasing. However, since the shifted symbol is less than $\clubsuit$ then it can additionally be left-shifted past another symbol. This second intermediate shift produces the stated tail, and the shifted symbol is the smallest symbol in its non-increasing prefix. Therefore, the reexpressed greedy left-shift is equal to a head. When translating this result to the cool left-shift, recall from Lemma 3.2.8 that the $(k+1)$st symbol is greedily left-shifted in $\mathsf{tail}(j,i)$ when $i = \clubsuit(j)$ as above. (This transition assumes that $j < \clubsuit$; the case when $j = \clubsuit$ is handled by the next transition.) Therefore, the above greedy left-shifts are examples of cool left-shifts.

Lemma 3.3.2 (Transition j)**.** *Suppose* $\mathbf{L}$ *is a bubble language,* $j < \clubsuit$*, and* $i = \clubsuit(j)$*. Then,*

$$\overleftarrow{\mathsf{cool}}(\mathsf{tail}(j,i)) = \mathsf{head}(j+1,1).$$

In other words, the cool left-shift transforms $\mathsf{tail}(j, \clubsuit(j))$ *into* $\mathsf{head}(j+1,1)$ *as long as* $j < \clubsuit$*.*

Proof. Let $k = |\mathsf{L}(\mathsf{tail}(j,i))|$ and $h = \overline{n}_{j+1}$. First notice the following equality, which follows from Remark 2.4.14

$$|\mathsf{L}(\mathsf{tail}(j+1,1))| = h - 1 + 1 = h. \tag{3.6}$$

The proof is now a result of the following derivation

$$\begin{aligned}
&\overleftarrow{\mathsf{cool}}(\mathsf{tail}(j,i)) \\
&= \overleftarrow{\mathsf{greedy}}(\mathsf{tail}(j,i),k+1) && \text{by Lemma 3.2.8} \\
&= \overleftarrow{\mathsf{greedy}}(\urcorner\llcorner,h+1) && \text{by Remark 3.1.2} \\
&= \overleftarrow{\mathsf{greedy}}(\overleftarrow{\mathsf{shift}}(\urcorner\llcorner,h+1,h),h) && \text{by } j < \clubsuit \text{ and (2.2)} \\
&= \overleftarrow{\mathsf{greedy}}(\mathsf{tail}(j+1,1),h) && \text{by (3.6) and Lemma 2.4.18} \\
&= \mathsf{head}(j+1,1).
\end{aligned}$$

□

Transition to non-increasing

The third type of transition takes $\mathsf{tail}(j,i)$ to $\urcorner\llcorner$. This transition involves greedily left-shifting the first symbol in the scut, and requires the additional condition that $j = \clubsuit$. To illustrate the transition consider the following examples with the language $\mathbf{L} = \mathbf{N}(\mathbb{M})$ (multiset necklaces) with $\mathbb{M} = \{1,1,1,1,2,2,2,3,3,4\}$

$$\begin{aligned}
&\overleftarrow{\mathsf{greedy}}(\mathsf{tail}(3,7),k+1) \\
&= \overleftarrow{\mathsf{greedy}}(4322211113,10) \\
&= \overleftarrow{\mathsf{greedy}}(\overleftarrow{4322211113},2) \\
&= \overleftarrow{\mathsf{greedy}}(4332221111,2) \\
&= 4332221111 \\
&= \urcorner\llcorner.
\end{aligned}$$

Notice that the symbol being greedily left-shifted is the first symbol in the scut and that the symbol is equal to $\clubsuit$. (In particular, $\clubsuit = 3$ since $d_3 = 3$ is the largest symbol that is the first symbol of a scut in this language.) Since the symbol is the first symbol in the scut then it can be greedily left-shifted past every symbol that is less than or equal to itself. After this intermediate shift the resulting string is entirely non-increasing. Any further shift is not possible due to the relationship between $\clubsuit$ and the frozen prefix of the non-increasing string found in Lemma 2.4.18. Therefore, the resulting string is the non-increasing string. The transition is now formalized below. When translating this result to the cool left-shift, recall from Lemma 3.2.8 that the $(k+1)$st symbol is greedily left-shifted in $\mathsf{tail}(j,i)$ when $i = \clubsuit(j)$ as above.

(This transition assumes that $j = \clubsuit$; the case when $j < \clubsuit$ was handled by the previous transition.) Therefore, the above greedy left-shift is an example of a cool left-shifts.

Lemma 3.3.3 (Transition to $\urcorner\llcorner$)**.** *Suppose* $\mathbf{L}$ *is a bubble language,* $j = \clubsuit$ *and* $i = \clubsuit(j)$*. Then,*

$$\overleftarrow{\mathsf{cool}}(\mathsf{tail}(j,i))k+1 = \urcorner\llcorner.$$

In other words, the cool left-shift transforms $\mathsf{tail}(j,i)$ *into the non-increasing string when* $j = \clubsuit$ *and* $i = \clubsuit(j)$*.*

Proof. Let $k = |\urcorner\llcorner(\mathsf{tail}(j,i))|$ and $h = \overline{n}_{j+1}$. Then,

$$\begin{aligned}
&\overleftarrow{\mathsf{cool}}(\mathsf{tail}(j,i)) \\
&= \overleftarrow{\mathsf{greedy}}(\mathsf{tail}(j,i), k+1) && \text{by Lemma 3.2.8} \\
&= \overleftarrow{\mathsf{greedy}}(\urcorner\llcorner, h+1) && \text{by Remark 3.1.2} \\
&= \urcorner\llcorner && \text{by } j = \clubsuit \text{ and Lemma 2.4.18.}
\end{aligned}$$

□

Transition from non-increasing

The fourth type of transition takes $\urcorner\llcorner$ to $\mathsf{head}(2,1)$. This transition involves greedily left-shifting the nth symbol in the non-increasing string, and requires the additional condition that there are at least two strings in the language. To illustrate the transition consider the following example with $\mathbf{L} = \Pi(\mathbb{M})$ (multiset permutations) with $\mathbb{M}\backslash\{1,1,2,2,3,3,4,4,5,5\}$

$$\begin{aligned}
&\overleftarrow{\mathsf{greedy}}(\urcorner\llcorner, n) \\
&= \overleftarrow{\mathsf{greedy}}(5544332211, 10) \\
&= \overleftarrow{\mathsf{greedy}}(554433221\overleftarrow{1}, 8) \\
&= \overleftarrow{\mathsf{greedy}}(5544332121, 8) \\
&= \overleftarrow{\mathsf{greedy}}(\mathsf{tail}(2,1), 8) \\
&= \mathsf{head}(2,1).
\end{aligned}$$

Notice that the symbol being greedily left-shifted is the last symbol. Since there are at least two strings in the language then $\urcorner\llcorner \in \mathbf{L}$ and it cannot be entirely frozen. Therefore, its last symbol can be bubble left-shifted to create another string in the

language. The string that is created is $\mathsf{tail}(2,1)$. Moreover, this intermediate shift causes the shifted symbol becomes the last symbol in the non-increasing prefix of this string. Therefore, the reexpressed greedy left-shift is equal to $\mathsf{head}(2,1)$. When translating this result to the cool left-shift, recall from Definition 3.1.3 that the nth symbol is greedily left-shifted in the non-increasing string as above. Therefore, the above greedy left-shift is an example of a cool left-shift.

Lemma 3.3.4 (Transition from $\lrcorner$)**.** *Suppose* $\mathbf{L}$ *is a non-trivial bubble language. Then,*

$$\overleftarrow{\mathsf{cool}}(\lrcorner) = \mathsf{head}(2,1).$$

In other words, the cool left-shift transforms the non-increasing string into $\mathsf{head}(2,1)$.

Proof. Since $\mathbf{L}$ is non-trivial

$$|\lrcorner(\lrcorner)| < n \tag{3.7}$$

since otherwise $\mathbf{L}$ could contain at most one string. Let $k = |\lrcorner(\mathsf{tail}(2,1))|$ and notice that $k = \overline{n}_2$ by Remark 2.4.14. Therefore,

$$\overleftarrow{\mathsf{bubble}}(\lrcorner, n) = \overleftarrow{\mathsf{shift}}(\lrcorner, n, k). \tag{3.8}$$

The proof is now a result of the following derivation

$$\begin{aligned}
&\overleftarrow{\mathsf{cool}}(\lrcorner) \\
&= \overleftarrow{\mathsf{greedy}}(\lrcorner, n) && \text{by (3.1a) in Definition 3.1.3} \\
&= \overleftarrow{\mathsf{greedy}}(\overleftarrow{\mathsf{shift}}(\lrcorner, n, k), k) && \text{by (3.7) and (2.2), and (3.8)} \\
&= \overleftarrow{\mathsf{greedy}}(\mathsf{tail}(2,1), k) && \text{by (3.8) and Lemma 2.4.18} \\
&= \mathsf{head}(2,1).
\end{aligned}$$

□

3.3.2 Proof of Generation

This section proves the main result of this chapter.

Theorem 3.3.5 (Left-Shift Gray code for bubble languages)**.** *The cool left-shift circularly generates the cool-lex order for any bubble language. In other words,* $\overleftarrow{\mathcal{C}}(\mathbf{L})$ *is a circular left-shift Gray code for any bubble language* $\mathbf{L}$*, and if* $\boldsymbol{s} \in \mathbf{L}$ *then the next string in the Gray code is always equal to* $\overleftarrow{\mathsf{cool}}(\boldsymbol{s})$.

Proof. The proof is by induction on $|\mathbf{L}|$. When $|\mathbf{L}| = 1$ then $\mathbf{L} = \{\urcorner\llcorner\}$ and $\overleftarrow{\mathsf{cool}}(\urcorner\llcorner) = \urcorner\llcorner$. Therefore the base case holds when $|\mathbf{L}| = 1$. Now suppose that $|\mathbf{L}| > 1$ and the theorem holds for every bubble language that has fewer strings than $\mathbf{L}$. Let $\mathbf{L}' = \mathbf{L}/\mathsf{scut}(j,i)$ for some $\mathsf{scut}(j,i) \in \mathsf{scuts}(\mathbf{L})$. Since bubble languages are closed under quotients by Theorem 2.4.2, then $\mathbf{L}'$ is a bubble language. Furthermore, $|\mathbf{L}'| < |\mathbf{L}|$. (In particular, $\urcorner\llcorner \in \mathbf{L}$ by Remark 2.2.2 but $\urcorner\llcorner$ does not have a scut, and so it does not contribute a string to $\mathbf{L}'$.) Therefore, the inductive assumption can be applied to $\mathbf{L}'$ and so $\overleftarrow{\mathsf{cool}}$ circularly generates $\overleftarrow{\mathcal{C}}(\mathbf{L}')$. The cool left-shift invariant in Lemma 3.1.5 then implies that $\overleftarrow{\mathsf{cool}}$ generates $\overleftarrow{\mathcal{C}}(\mathbf{L}') \cdot \mathsf{scut}(j,i)$. (In particular, the last string in $\overleftarrow{\mathcal{C}}(\mathbf{L}') \cdot \mathsf{scut}(j,i)$ is $\mathsf{tail}(j,i)$ by Remark 3.2.7, and the cool left-shift invariant applies to all strings that are not tails.) Therefore, when considering whether or not $\overleftarrow{\mathsf{cool}}$ circularly generates $\overleftarrow{\mathcal{C}}(\mathbf{L})$ only the transitions between the various sublists and the non-increasing strings needs to be considered. In particular, Definition 3.2.2 there are four types of transitions to consider.

The first type of transition is between the last string in $\overleftarrow{\mathcal{C}}(\mathbf{L}/\mathsf{scut}(j,i)) \cdot \mathsf{scut}(j,i)$ and the first string in $\overleftarrow{\mathcal{C}}(\mathbf{L}/\mathsf{scut}(j,i{+}1)) \cdot \mathsf{scut}(j,i{+}1)$. In this case Lemma 3.3.1 implies that $\overleftarrow{\mathsf{cool}}$ makes the correct transformation. (Recall from Remark 3.2.7 and 3.2.6 that the last string in each sublist is a tail, and that the first string in each sublist is a head.) The second type of transition is between the last string in $\overleftarrow{\mathcal{C}}(\mathbf{L}/\mathsf{scut}(j,i)) \cdot \mathsf{scut}(j,i)$ and the first string in $\overleftarrow{\mathcal{C}}(\mathbf{L}/\mathsf{scut}(j+1,1)) \cdot \mathsf{scut}(j+1,1)$. In this case Lemma 3.3.2 implies that $\overleftarrow{\mathsf{cool}}$ makes the correct transformation. The third transition is between the last string in $\overleftarrow{\mathcal{C}}(\mathbf{L}/\mathsf{scut}(j,i)) \cdot \mathsf{scut}(j,i)$ and $\urcorner\llcorner$. In this case Lemma 3.3.3 implies that $\overleftarrow{\mathsf{cool}}$ makes the correct transformation. The fourth transition is between $\urcorner\llcorner$ and the first string in $\overleftarrow{\mathcal{C}}(\mathbf{L}/\mathsf{scut}(2,1)) \cdot \mathsf{scut}(2,1)$. In this case Lemma 3.3.4 implies that $\overleftarrow{\mathsf{cool}}$ makes the correct transformation. Therefore, $\overleftarrow{\mathsf{cool}}$ circularly generates $\overleftarrow{\mathcal{C}}(\mathbf{L})$ and so the theorem is true by induction. □

A simple corollary of Theorem 3.3.5 is that cool-lex order provides an $(m+2)$-*assignment Gray code* for any bubble language $\mathbf{L}$. (Recall that m is the number of distinct symbols in the strings within $\mathbf{L}$.) To illustrate why this is true, consider the difference between the following two strings

$$\boldsymbol{s} = 44433322211145$$
$$\boldsymbol{t} = 54443332221114.$$

Notice that $\boldsymbol{t}$ can be obtained from $\boldsymbol{s}$ by left-shifting the last symbol into the first position. Alternatively, $\boldsymbol{t}$ can be obtained from $\boldsymbol{s}$ by six *assignments*. (In particular,

$t_i \neq s_i$ for $i \in \{1, 4, 7, 10, 13, 14\}$.) In general, each cool left-shift involves moving a symbol past at most $k+1$ differing symbols, where k is the number of distinct symbols in the non-increasing prefix. Since the number of distinct symbols in the non-increasing prefix is at most m, then bubble languages have $(m+2)$-assignment Gray codes. When $m = 2$ the result is a 4-assignment Gray code. However, when $m = 2$ then every pair of assignments can be accomplished by a single transposition, and so the 4-assignment Gray code is also a 2-transposition Gray code. These results are summarized by Corollary 3.3.6.

Corollary 3.3.6 (Assignment and Transposition Gray code for bubble languages)**.** *Every bubble language has a circular $(m+2)$-assignment Gray code when expressed in cool-lex order, where m is the number of distinct symbols in the language's content. When $m = 2$ this result is also a 2-transposition Gray code.*

Another corollary of Theorem 3.3.5 is that there is a simple algorithm for generating the strings in any bubble language. The following pseudocode (and Corollary 3.3.6) provide a starting point for the efficient algorithms developed in Chapter 4.

Require: $\mathbf{L}$ is a bubble language with strings of length n
Require: $\boldsymbol{s}$ is initially the non-increasing string $\lrcorner$
  $k \leftarrow n-1$
  while $k < n$ **do**
    if $k = n-1$
      $\boldsymbol{s} \leftarrow \overleftarrow{\mathsf{greedy}}(\boldsymbol{s}, n)$
    else
      if $s_k \geq s_{k+2}$ and $\overleftarrow{\mathsf{shift}}(\boldsymbol{s}, k+2, k+1) \in \mathbf{L}$
        $\boldsymbol{s} \leftarrow \overleftarrow{\mathsf{greedy}}(\boldsymbol{s}, k+2)$
      else
        $\boldsymbol{s} \leftarrow \overleftarrow{\mathsf{greedy}}(\boldsymbol{s}, k+1)$
      end
    end
    visit$(\boldsymbol{s})$
    $k \leftarrow |\lrcorner(\boldsymbol{s})|$
  end

Algorithm 1: Iterative algorithm for generating the strings of any bubble language $\mathbf{L}$ using $\overleftarrow{\mathsf{shift}}$ and $\overleftarrow{\mathsf{greedy}}$.

3.4 Properties

This section presents several properties of cool-lex order. Section 3.4.2 proves that concatenating strings in cool-lex order produces a circular string containing the first $n-1$ symbols of every rotation of the strings in the original bubble language. This result will be the primary result needed to prove the shorthand universal cycle result in Section 4.2. Central to the results in Section 3.4.2 is the cool right-shift, which generates cool-lex order for bubble languages in reverse order. This operation is presented in Section 3.4.1.

3.4.1 Reverse Order

This section formalizes *reverse cool-lex order* and the operation that generates this order. The reverse cool-lex order for a non-empty bubble language $\mathbf{L}$ is

$$\overrightarrow{\mathcal{C}}(\mathbf{L}) = \boldsymbol{s}, \overrightarrow{\mathsf{cool}}(\boldsymbol{s}), \overrightarrow{\mathsf{cool}}_2(\boldsymbol{s}), \overrightarrow{\mathsf{cool}}_3(\boldsymbol{s}), \ldots, \overrightarrow{\mathsf{cool}}_{|\mathbf{L}|-1}(\boldsymbol{s}) \tag{3.9}$$

where $\boldsymbol{s} = \curlywedge$, $\overrightarrow{\mathsf{cool}}$ is the *cool right-shift* operation that is defined in this section and then will be proven to be inverse to $\overleftarrow{\mathsf{cool}}$ (and so $\overrightarrow{\mathsf{cool}}_{|\mathbf{L}|}(\boldsymbol{s}) = \boldsymbol{s}$). Notice that (3.9) provides the reverse order given in (3.4). The definition of $\overrightarrow{\mathsf{cool}}$ is based on a slightly modified notion of the non-increasing prefix given by the following definition.

Definition 3.4.1 (Weakly Non-Increasing Prefix)**.** *The* weakly non-increasing prefix *of* $\boldsymbol{s}$ *with* $|\boldsymbol{s}| = n$ *and* $|\curlywedge(\boldsymbol{s})| = k$ *is*

$$\curlywedge(\boldsymbol{s}) = s_1 s_2 \ldots s_h$$

where $h = k$ *if* $k = n$ *or* $s_{k-1} < s_{k+1}$*, and otherwise* $h = |\curlywedge(s_1 s_2 \cdots s_{k-1} s_{k+1} s_{k+2} \cdots s_n)| + 1$*. In other words, the weakly non-increasing prefix is the longest non-increasing prefix with the proviso that the value of the last symbol in the non-increasing prefix is ignored.*

The following examples illustrate the weakly non-increasing prefix

$$\curlywedge(43211234) = 43211 \qquad\qquad \curlywedge(43132214) = 4313221.$$

In the first case above notice that "ignoring" the last symbol in the non-increasing prefix does not result in a longer non-increasing prefix. That is, the non-increasing prefix of 43211234 is the same as the non-increasing prefix of 43211/234 except for the

ignored symbol. Therefore, in this case the non-increasing prefix and the weakly non-increasing prefix are identical. On the other hand, in the second case above notice that ignoring the last symbol in the non-increasing prefix does result in a longer non-increasing prefix. That is, the non-increasing prefix of 43132214 is not the same as the non-increasing prefix of 43132214. Therefore, in this case the non-increasing prefix and the weakly non-increasing prefix are not identical.

The other concept required in definition of $\overrightarrow{\mathsf{cool}}$ is that of a *bounded right-shift*. This shift is the same as a maximal left-shift except that a) the shift is to the right, and b) the distance of the shift is bounded by a maximum position. The concept is formalized and then illustrated below.

Definition 3.4.2 (Bounded Right-Shift)**.** *Given* $\boldsymbol{s} \in \mathbf{L}$ *and indices* i *and* j *with* $1 \leq i \leq j \leq n$*, the* bounded right-shift *in* $\boldsymbol{s}$ *is*

$$\overrightarrow{\mathsf{max}}_{\mathbf{L}}(\boldsymbol{s}, i, j) = \overrightarrow{\mathsf{shift}}(\boldsymbol{s}, i, h)$$

where h *satisfies*

- $\overrightarrow{\mathsf{shift}}(\boldsymbol{s}, i, g) \in \mathbf{L}$ *for all* g *within* $i \leq g \leq h$
- $h = j$ *or* $\overrightarrow{\mathsf{shift}}(\boldsymbol{s}, i, h+1) \notin \mathbf{L}$.

In other words, $\overrightarrow{\mathsf{max}}$ *right-shifts a symbol in a string as far as possible while maintaining the property that every intermediate shift is in the given language and that the shift does not pass the maximum specified position.*

By convention, $\overrightarrow{\mathsf{max}}(\boldsymbol{s}, i, j) = \overrightarrow{\mathsf{max}}_{\mathbf{L}}(\boldsymbol{s}, i, j)$ so the language is assumed to be $\mathbf{L}$ unless otherwise stated. To illustrate the definition, consider the following examples for $\mathbf{L} = \mathbf{P}(6)6$ (balanced parentheses)

$$\overrightarrow{\mathsf{max}}(110110100010, 2, 10) = 1\overrightarrow{10110100}010 = 101101001010$$

$$\overrightarrow{\mathsf{max}}(110110100010, 3, 10) = 11\overrightarrow{0110100}010 = 111101000010.$$

In the first case above, the right-shift is limited by the fact that any further shift would produce a string that is outside of the language. On the other hand, in the second case the right-shift is limited by the provided bound. Now the cool right-shift can be formally defined.

Definition 3.4.3 (Cool Right-Shift)**.** *Given* $\boldsymbol{r} \in \mathbf{L}$ *where* $\mathbf{L}$ *is a non-trivial bubble language and* $f = |\mathsf{1}(\boldsymbol{r})|$ *and* $h = |\mathsf{L}(\boldsymbol{r})|$ *and* $g = |\mathsf{LL}(\boldsymbol{r})|$*, the* cool right shift *is*

$$\overrightarrow{\mathsf{cool}}_{\mathbf{L}}(\boldsymbol{r}) = \begin{cases} \overrightarrow{\mathsf{max}}(\boldsymbol{r}, f, h) & \text{if } f < h & \text{(3.10a)} \\ \overrightarrow{\mathsf{max}}(\boldsymbol{r}, f, g) & \text{if } f = h,\ h < g,\ \text{and } r_h > r_g & \text{(3.10b)} \\ \overrightarrow{\mathsf{shift}}(\boldsymbol{r}, f, g+1) & \text{if } f = h,\ (h = g \text{ or } r_h \leq r_g),\ \text{and } g < n & \text{(3.10c)} \\ \overrightarrow{\mathsf{shift}}(\boldsymbol{r}, f, n) & \text{otherwise (if } f = h \text{ and } g = n\text{)} & \text{(3.10d)} \end{cases}$$

The following theorem proves that $\overrightarrow{\mathsf{cool}}$ and $\overleftarrow{\mathsf{cool}}$ are inverse operations.

Theorem 3.4.4 (Inverse)**.** *Suppose* $\mathbf{L}$ *is a non-trivial bubble language and* $\boldsymbol{s} \in \mathbf{L}$*. Then,*

$$\overrightarrow{\mathsf{cool}}(\overleftarrow{\mathsf{cool}}(\boldsymbol{s})) = \boldsymbol{s}.$$

In other words, $\overrightarrow{\mathsf{cool}}$ *is the inverse of* $\overleftarrow{\mathsf{cool}}$ *when applied to strings in bubble languages.*

Proof. Let $\boldsymbol{r} = \overleftarrow{\mathsf{cool}}(\boldsymbol{s})$. It needs to be proven that $\overrightarrow{\mathsf{cool}}(\boldsymbol{r}) = \boldsymbol{s}$. Towards this goal, let $f = |\mathsf{1}(\boldsymbol{r})|$, $h = |\mathsf{L}(\boldsymbol{r})$, $g = |\mathsf{LL}(\boldsymbol{r})|$, and $k = |\mathsf{L}(\boldsymbol{s})|$. By Definition 3.1.3 and Lemma 2.4.5,

$$\boldsymbol{r} = \overleftarrow{\mathsf{shift}}(\boldsymbol{s}, x, f) \tag{3.11}$$

for some $x \in \{k+1, k+2, n\}$. The proof now divides into several cases. In each case, $\boldsymbol{s} \neq \boldsymbol{r}$ holds due to Theorem 3.3.5 and the fact that $\mathbf{L}$ is a non-trivial bubble language.

In the first case suppose $k = n$. That is, $\boldsymbol{s}$ is the non-increasing string. Therefore, by (3.11) and (3.1a)

$$\boldsymbol{r} = \overleftarrow{\mathsf{shift}}(\boldsymbol{s}, n, f).$$

Therefore, $f = h$ and $g = n$. Therefore, by (3.10d)

$$\overrightarrow{\mathsf{cool}}(\boldsymbol{r}) = \overrightarrow{\mathsf{shift}}(\boldsymbol{r}, f, n).$$

Therefore, the result is true in this case.

In the second case suppose $k = n - 1$. Therefore, by (3.11) and (3.1b)

$$\boldsymbol{r} = \overleftarrow{\mathsf{shift}}(\boldsymbol{s}, n, f).$$

There are two subcases to consider. In the first subcase, $\mathsf{L}(\boldsymbol{r}) = \boldsymbol{r}$. In this subcase $f < h$ and $h = n$. Therefore, by (3.10a)

$$\overrightarrow{\mathsf{cool}}(\boldsymbol{r}) = \overrightarrow{\mathsf{max}}(\boldsymbol{r}, f, h).$$

However, $\overrightarrow{\mathsf{max}}(\boldsymbol{r}, f, h) = \overrightarrow{\mathsf{shift}}(\boldsymbol{r}, f, n)$ because $\overleftarrow{\mathsf{cool}}(\boldsymbol{s}) = \overleftarrow{\mathsf{shift}}(\boldsymbol{s}, n, f)$. Therefore, the result is true in this subcase. In the second subcase, $\llcorner(\boldsymbol{r}) \neq \boldsymbol{r}$. In this subcase $f = h$ and $g = n$. Therefore,

$$\overrightarrow{\mathsf{cool}}(\boldsymbol{r}) = \overrightarrow{\mathsf{shift}}(\boldsymbol{r}, f, n).$$

Therefore, the result is true in this subcase. Therefore, the result is true in this case and for the remaining cases it is implicitly assumed that $k \leq n - 2$.

In the third case suppose that $s_k < s_{k+2}$. Therefore, by (3.11) and (3.1b)

$$\boldsymbol{r} = \overleftarrow{\mathsf{shift}}(\boldsymbol{s}, k + 1, f).$$

There are two subcases to consider. In the first subcase $h = k + 1$. In this subcase $f < h$. Therefore, by (3.10a)

$$\overrightarrow{\mathsf{cool}}(\boldsymbol{r}) = \overrightarrow{\mathsf{max}}(\boldsymbol{r}, f, h).$$

However, $\overrightarrow{\mathsf{max}}(\boldsymbol{r}, f, h) = \overrightarrow{\mathsf{shift}}(\boldsymbol{r}, f, k{+}1)$ because $h = k{+}1$ and $\overleftarrow{\mathsf{cool}}(\boldsymbol{s}) = \overleftarrow{\mathsf{shift}}(\boldsymbol{s}, k{+}1, f)$. Therefore, the result is true in this subcase. In the second subcase $h < k + 1$. In this subcase $g = k + 1$. Furthermore, $f = h$, $h < g$, and $r_h > r_g$. Therefore, by (3.10b)

$$\overrightarrow{\mathsf{cool}}(\boldsymbol{r}) = \overrightarrow{\mathsf{max}}(\boldsymbol{r}, f, g).$$

However, $\overrightarrow{\mathsf{max}}(\boldsymbol{r}, f, g) = \overrightarrow{\mathsf{shift}}(\boldsymbol{r}, f, k{+}1)$ because $g = k{+}1$ and $\overleftarrow{\mathsf{cool}}(\boldsymbol{s}) = \overleftarrow{\mathsf{shift}}(\boldsymbol{s}, k{+}1, f)$. Therefore, the result is true in this subcase. Therefore, the result is true in this case.

In the fourth case suppose that $s_k \geq s_{k+2}$ and $\overleftarrow{\mathsf{greedy}}(\boldsymbol{s}, k + 2) = \boldsymbol{s}$. Therefore, by (3.1b)

$$\boldsymbol{r} = \overleftarrow{\mathsf{shift}}(\boldsymbol{s}, k + 1, f).$$

Also notice that $\overleftarrow{\mathsf{greedy}}(\boldsymbol{s}, k + 2) = \boldsymbol{s}$ implies that $\boldsymbol{w} = \overrightarrow{\mathsf{shift}}(\boldsymbol{s}, k + 1, k + 2) \notin \mathbf{L}$. (In particular, $s_{k+1} \neq s_{k+2}$ since otherwise $\overleftarrow{\mathsf{bubble}}(\boldsymbol{s}, k + 2) = \overleftarrow{\mathsf{bubble}}(\boldsymbol{s}, k + 1) \in \mathbf{L}$ by (2.1).) There are two subcases to consider. In the first subcase $h \geq k + 2$. In this subcase $f < h$. Therefore, by (3.10a)

$$\overrightarrow{\mathsf{cool}}(\boldsymbol{r}) = \overrightarrow{\mathsf{max}}(\boldsymbol{r}, f, h).$$

However, $\overrightarrow{\mathsf{max}}(\boldsymbol{r}, f, h) = \overrightarrow{\mathsf{shift}}(\boldsymbol{r}, f, k{+}1)$ because $h \geq k{+}2$ and $\overleftarrow{\mathsf{cool}}(\boldsymbol{s}) = \overleftarrow{\mathsf{shift}}(\boldsymbol{s}, k{+}1, f)$ and $\boldsymbol{w} \notin \mathbf{L}$. Therefore, the result is true in this subcase. In the second subcase $h < k{+}1$. In this subcase $g \geq k + 2$. Furthermore, $f = h$ and $r_h > r_g$. (The last inequality follows

from $r_h = s_{k+1} > s_k \geq s_{k+2} \geq r_g$.) Therefore, by (3.10b)

$$\overrightarrow{\mathsf{cool}}(\boldsymbol{r}) = \overrightarrow{\mathsf{max}}(\boldsymbol{r}, f, g).$$

However, $\overrightarrow{\mathsf{max}}(\boldsymbol{r}, f, g) = \overrightarrow{\mathsf{shift}}(\boldsymbol{r}, f, k+1)$ because $g \geq k+2$ and $\overleftarrow{\mathsf{cool}}(\boldsymbol{s}) = \overleftarrow{\mathsf{shift}}(\boldsymbol{s}, k+1, f)$ and $\boldsymbol{w} \notin \mathbf{L}$. Therefore, the result is true in this subcase. Therefore, the result is true in this case.

In the fifth case suppose that $s_k \geq s_{k+2}$ and $\overleftarrow{\mathsf{greedy}}(\boldsymbol{s}, k+2) \neq \boldsymbol{s}$. Therefore, by (3.1c)

$$\boldsymbol{r} = \overleftarrow{\mathsf{shift}}(\boldsymbol{s}, k+2, f).$$

There are two subcases to consider. In the first subcase $h = k+1$. In this subcase $f = h$ and $h = g$. Therefore, by (3.10c)

$$\overrightarrow{\mathsf{cool}}(\boldsymbol{r}) = \overrightarrow{\mathsf{shift}}(\boldsymbol{r}, f, g+1).$$

However, $\overrightarrow{\mathsf{shift}}(\boldsymbol{r}, f, g+1) = \overrightarrow{\mathsf{shift}}(\boldsymbol{r}, f, k+2)$ since $g = h = k+1$. Therefore, the result is true in this subcase. In the second subcase $h < k+1$. In this subcase $g = k+1$. Furthermore, $f = h$ and $r_h \leq r_g$. Therefore, by (3.10c)

$$\overrightarrow{\mathsf{cool}}(\boldsymbol{r}) = \overrightarrow{\mathsf{shift}}(\boldsymbol{r}, f, g+1)$$

However, $\overrightarrow{\mathsf{shift}}(\boldsymbol{r}, f, g+1) = \overrightarrow{\mathsf{shift}}(\boldsymbol{r}, f, k+2)$ since $g = k+1$. Therefore, the result is true in this subcase. Therefore, the result is true in this case. □

3.4.2 Shorthand Rotations

shorthand [noun]
- a simplified or makeshift manner or system of communication.[35]

-

This section proves an interesting property involving shorthand rotations within the reverse cool-lex order of any bubble language. Recall from Section 2.3.7 that $\circlearrowleft_i(\boldsymbol{s})$ is the rotation of $\boldsymbol{s}$ starting at position i, and that $\circlearrowleft(\boldsymbol{s})$ is the set of all rotations of $\boldsymbol{s}$. Also recall from Section 1.2.3 that the shorthand representation of a string simply removes its last symbol, as formalized below.

Definition 3.4.5 (Shorthand)**.** *The* shorthand *of a string* $\boldsymbol{s} = s_1 s_2 \cdots s_n$ *is*

$$\mathsf{short}(\boldsymbol{s}) = s_1 s_2 \cdots s_{n-1}.$$

Moreover, given a set of strings $\mathbf{L}$*, let* $\mathsf{short}(\mathbf{L})$ *be the set of strings containing the shorthand of each string in* $\mathbf{L}$*.*

The *shorthand rotation property* holds for a list $\mathcal{T}$ if for every $\boldsymbol{s}$ in $\mathcal{T}$ and i satisfying $1 \leq i \leq |\boldsymbol{s}|$, there exist consecutive strings $\boldsymbol{r}$ and $\boldsymbol{t}$ in $\mathcal{T}$ such that $\mathsf{short}(\circlearrowleft_i(\boldsymbol{s}))$ is a substring of $\boldsymbol{r} \cdot \boldsymbol{t}$. In other words, if the strings in $\mathcal{T}$ are concatenated together, then its substrings include the set $\mathsf{short}(\circlearrowleft(\boldsymbol{s}))$ for each string $\boldsymbol{s}$ in $\mathcal{T}$. Although this property may at first seem esoteric, it turns out to be the key towards the interesting results found in Section 4.2. This section proves that the shorthand rotation property holds for the reverse cool-lex order of any bubble language. Before discussing the proof, the property is illustrated for two different strings within $\overrightarrow{\mathcal{C}}(\mathbf{L})$ of $\mathbf{L} = \mathbf{T}(\{3,3,1,1,0,0,0,0\})$ (ordered trees with fixed branching sequence). First consider one of its strings

$$\boldsymbol{s} = 30030101.$$

The shorthand rotations of this string are listed below

$$\mathsf{short}(\circlearrowleft(\boldsymbol{s})) = \{3003010, 0030101, 0301013, 3010130, \\ 0101300, 1013003, 0130030, 1300301\}.$$

In this case, these desired substrings simply appear within $\overleftarrow{\mathsf{cool}}(\boldsymbol{s}) \cdot \boldsymbol{s}$ and $\boldsymbol{s} \cdot \overrightarrow{\mathsf{cool}}(\boldsymbol{s})$ as illustrated below

$$\begin{aligned} \overleftarrow{\mathsf{cool}}(\boldsymbol{s}) \cdot \boldsymbol{s} \cdot \overrightarrow{\mathsf{cool}}(\boldsymbol{s}) &= \overleftarrow{30030101} \cdot 30030101 \cdot \overrightarrow{30030101} \\ &= 33000101 \cdot 30030101 \cdot 30300101 \\ &= 3300\underline{0101300301013010130}0300101. \end{aligned}$$

In particular, the non-circular substrings of length $n - 1 = 7$ in the above underlined substring are

$$0101300, 1013003, 0130030, 1300301, 3003010, 0030101, 0301013, 3010130.$$

Since the above strings are identical to those in $\mathsf{short}(\circlearrowleft(\boldsymbol{s}))$, then the shorthand rotation property has been demonstrated for $\boldsymbol{s} = 30030101$ in $\overrightarrow{\mathcal{C}}(\mathbf{L})$. Before continuing with a second example, the reader is reminded that $\mathsf{last}(\mathcal{T})$ and $\mathsf{first}(\mathcal{T})$ are considered consecutive within $\mathcal{T}$ (see page 111) and so $\overleftarrow{\mathsf{cool}}(\boldsymbol{s})$ and $\boldsymbol{s}$ are consecutive in $\overrightarrow{\mathcal{C}}(\mathbf{L})$, as are $\boldsymbol{s}$ and $\overrightarrow{\mathsf{cool}}(\boldsymbol{s})$ by Theorems 3.3.5 and 3.4.4.

In other cases, it is more difficult to find all of the shorthand rotations for a given

string in reverse cool-lex order, and the desired substrings may be spread across the concatenation of up to m additional pairs of consecutive strings in the list. For example, consider

$$\boldsymbol{s} = 31103000$$

as another string in the same language of ordered trees with fixed branching sequence. In this case the concatenation of $\overleftarrow{\mathsf{cool}}(\boldsymbol{s}) \cdot \boldsymbol{s}$ and $\boldsymbol{s} \cdot \overrightarrow{\mathsf{cool}}(\boldsymbol{s})$ does not contain all of the shorthand rotations as illustrated below

$$\begin{aligned}\overleftarrow{\mathsf{cool}}(\boldsymbol{s}) \cdot \boldsymbol{s} \cdot \overrightarrow{\mathsf{cool}}(\boldsymbol{s}) &= 3\overleftarrow{1103000} \cdot 31103000 \cdot \overrightarrow{3110}3000 \\ &= 30110300 \cdot 31103000 \cdot 11303000 \\ &= 3011030\underline{0031103000}11303000.\end{aligned}$$

Notice that the underlined substring contains only four shorthand rotations of $\boldsymbol{s}$. To find the remaining four shorthand rotations, consider the concatenations beginning at strings obtained by greedily left-shifting the unfrozen symbols in the non-increasing prefix of $\boldsymbol{s}$, as well as the symbol following the non-increasing prefix of $\boldsymbol{s}$. These strings are given below

$$\begin{aligned}\overleftarrow{\mathsf{greedy}}(\boldsymbol{s}, 2) &= \overleftarrow{31}1030000 \\ &= 13103000\end{aligned} \qquad \begin{aligned}\overleftarrow{\mathsf{greedy}}(\boldsymbol{s}, 4) &= 3\overleftarrow{110}3000 \\ &= 30113000\end{aligned} \qquad \begin{aligned}\overleftarrow{\mathsf{greedy}}(\boldsymbol{s}, 5) &= \overleftarrow{3110}3000 \\ &= 33110000.\end{aligned}$$

Concatenating these strings with their successors in reverse cool-lex gives

$$\begin{aligned}&13103000 \cdot \overrightarrow{\mathsf{cool}}(13103000) \\ &= 13103000 \cdot \overrightarrow{1310}3000 \\ &= 13103000 \cdot 31013000 \\ &= 13\underline{10300031}013000\end{aligned} \qquad \begin{aligned}&30113000 \cdot \overrightarrow{\mathsf{cool}}(30113000) \\ &= 30113000 \cdot \overrightarrow{3011}3000 \\ &= 30113000 \cdot 31130000 \\ &= 3011\underline{300031}130000\end{aligned} \qquad \begin{aligned}&33110000 \cdot \overrightarrow{\mathsf{cool}}(33110000) \\ &= 33110000 \cdot \overrightarrow{3311}0000 \\ &= 33110000 \cdot 31100300 \\ &= 3311\underline{000311}00300.\end{aligned}$$

Collectively, the underlined substrings contain the remaining four shorthand rotations of $\boldsymbol{s}$. To understand why this is the case, consider the middle column above. Notice that the two strings involved in the concatenation can be reexpressed as left- and right-shifts of the fourth symbol within $\boldsymbol{s} = 31103000$. That is,

$$30113000 \cdot 31130000 = 3\overleftarrow{110}3000 \cdot 311\overrightarrow{03}000.$$

In other words, a single copy of 0 is swept back and forth with respect to $\boldsymbol{s}$ in the

above concatenation. This ensures that a suffix of $\boldsymbol{s}$ and a prefix of $\boldsymbol{s}$ are respectively undisturbed within the concatenated strings, thereby allowing for the inclusion of the shorthand rotation of $\boldsymbol{s}$ that omits this copy of 0, namely $\mathsf{short}(\circlearrowleft_5(\boldsymbol{s})) = \underline{3000 \cdot 311}$.

Formally, these arguments require the results of Section 2.4.5. Towards the results of Section 4.2, it is also worth noting that the shorthand rotation $\mathsf{short}(\circlearrowleft_i(\boldsymbol{s}))$ will always appear at the ith position of two concatenated strings in reverse cool-lex order. For example, in the middle column above, $\mathsf{short}(\circlearrowleft_5(\boldsymbol{s})) = 3000311$ begins at the fifth position of $30113000 \cdot 31130000$. This point holds in general for all of the shorthand rotations. The proof of the shorthand rotation property is divided into four lemmas. Within the lemmas, the expression $\bigodot \mathcal{T}$ refers to the concatenation of all of the strings in list $\mathcal{T}$. For example, the following expression shows how this operation is applied to the reverse cool-lex order of $\mathbf{T}(\{3,2,0,0,0\})$ (ordered trees with fixed branching sequence).

$$\begin{aligned}\bigodot \overrightarrow{\mathcal{C}}(\mathbf{T}(\{3,2,0,0,0\})) &= 32000, 20300, 23000, 30020, 30200\\ &= 32000 \cdot 20300 \cdot 23000 \cdot 30020 \cdot 30200\\ &= 32000203002300030020302 00.\end{aligned}$$

Using this notation, the shorthand rotation property holds for a list $\mathcal{T}$ if the substrings of $\bigodot \mathcal{T}$ include $\mathsf{short}(\circlearrowleft(\boldsymbol{s}))$ for all $\boldsymbol{s}$ in $\mathcal{T}$.

Lemma 3.4.6 (Rotations 1 to $|🌡(\boldsymbol{s})|+1$)**.** *Given a bubble language* $\mathbf{L}$*, and* $\boldsymbol{s} \in \mathbf{L}$ *with* $f = |🌡(\boldsymbol{s})|$*, then* $\mathsf{short}(\circlearrowleft_i(\boldsymbol{s}))$ *is a substring of* $\bigodot \overrightarrow{\mathcal{C}}(\mathbf{L})$ *for all* $1 \le i \le f+1$*. In particular,* $\mathsf{short}(\circlearrowleft_i(\boldsymbol{s}))$ *begins at the* i*th position of* $\boldsymbol{s} \cdot \overrightarrow{\mathsf{cool}}(\boldsymbol{s})$*.*

Proof. Let $\boldsymbol{t} = \overrightarrow{\mathsf{cool}}(\boldsymbol{s})$. By Definition 3.4.3,

$$\boldsymbol{t} = \overrightarrow{\mathsf{shift}}(\boldsymbol{s}, f, j)$$

for some value of $j \ge f$. Therefore,

$$t_1 t_2 \cdots t_{f-1} = s_1 s_2 \cdots s_{f-1}.$$

Therefore, $\mathsf{short}(\circlearrowleft_i(\boldsymbol{s}))$ begins at the ith position of $\boldsymbol{s} \cdot \boldsymbol{t}$ for all i within $1 \le i \le f+1$. □

Lemma 3.4.7 (Rotations $|🌡(\boldsymbol{s})|+2$ to $|┐└(\boldsymbol{s})|+1$)**.** *Given a bubble language* $\mathbf{L}$*, and* $\boldsymbol{s} \in \mathbf{L}$ *with* $k = |┐└(\boldsymbol{s})|$*, then* $\mathsf{short}(\circlearrowleft_i(\boldsymbol{s}))$ *is a substring of* $\bigodot \overrightarrow{\mathcal{C}}(\mathbf{L})$ *for all* $|🌡(\boldsymbol{s})|+2 \le$

$i \le k+1$. *In particular,* $\mathsf{short}(\circlearrowleft_i(\boldsymbol{s}))$ *begins at the* i*th position of* $\boldsymbol{r}\cdot\overrightarrow{\mathsf{cool}}(\boldsymbol{r})$ *where* $\boldsymbol{r} = \overleftarrow{\mathsf{greedy}}(\boldsymbol{s}, i-1)$.

Proof. Let $\boldsymbol{t} = \overrightarrow{\mathsf{cool}}(\boldsymbol{r})$. Also, let $f = |\text{🌡}(\boldsymbol{r})|$ and $h = |\lceil\!\llcorner(\boldsymbol{r})|$ and $g = |\lceil\!\llcorner(\boldsymbol{r})|$, to align with Definition 3.4.3. Since $\boldsymbol{r} = \overleftarrow{\mathsf{greedy}}(\boldsymbol{s}, i-1)$ then

$$r_i r_{i+1}\cdots r_n = s_i s_{i+1}\cdots s_n.$$

Furthermore, since s_{i-1} is in the non-increasing prefix of $\boldsymbol{s}$ (by $i \le k+1$)), and since $\boldsymbol{r} neq \boldsymbol{s}$ (by $|\text{🌡}(\boldsymbol{s})| + 2 \le i \le k+1$ and (2.2)) then Lemma 2.4.5 implies that

$$\boldsymbol{r} = \overleftarrow{\mathsf{shift}}(\boldsymbol{s}, i-1, f).$$

Moreover, the same reasoning implies that $f = h$ and $h < g$ and $g = k$. Since $f = h$, then Definition 3.4.3 implies that

$$\boldsymbol{t} \in \{\overrightarrow{\mathsf{max}}(\boldsymbol{r}, f, g), \overrightarrow{\mathsf{shift}}(\boldsymbol{r}, f, g+1), \overrightarrow{\mathsf{shift}}(\boldsymbol{r}, f, n)\}.$$

Therefore,

$$\boldsymbol{t} = \overrightarrow{\mathsf{shift}}(\boldsymbol{r}, f, j)$$

for $j \ge i-1$. (The bound on j follows from the facts that $i-1 \le n$, $i-1 \le g$ (by $i \le k+1$ and $g = k$), and $\overleftarrow{\mathsf{greedy}}(\boldsymbol{s}, i-1) = \overleftarrow{\mathsf{shift}}(\boldsymbol{s}, i-1, f)$.) Since $\boldsymbol{r} = \overleftarrow{\mathsf{shift}}(\boldsymbol{s}, i-1, f)$ and $\boldsymbol{t} = \overrightarrow{\mathsf{shift}}(\boldsymbol{r}, f, j)$ for $j \ge i-1$, then $\boldsymbol{t} = \overrightarrow{\mathsf{shift}}(\boldsymbol{s}, i-1, j)$. Therefore,

$$t_1 t_2\cdots t_{i-2} = s_1 s_2\cdots s_{i-2}.$$

Therefore, $\mathsf{short}(\circlearrowleft_i(\boldsymbol{s}))$ begins at the ith position of $\boldsymbol{r}\cdot\boldsymbol{t}$ for all i within $|\text{🌡}(\boldsymbol{s})| + 2 \le i \le k+1$. □

Lemma 3.4.8 (Rotation $k+2$)**.** *Given a bubble language* $\mathbf{L}$*, and* $\boldsymbol{s} \in \mathbf{L}$ *with* $k = |\lceil\!\llcorner(\boldsymbol{s})|$*, then* $\mathsf{short}(\circlearrowleft_i(\boldsymbol{s}))$ *is a substring of* $\odot\,\overrightarrow{\mathcal{C}}(\mathbf{L})$ *for* $i = k+2$*. In particular,* $\mathsf{short}(\circlearrowleft_i(\boldsymbol{s}))$ *begins at the* i*th position of* $\boldsymbol{r}\cdot\overrightarrow{\mathsf{cool}}(\boldsymbol{r})$ *where* $\boldsymbol{r} = \overleftarrow{\mathsf{greedy}}(\boldsymbol{s}, k+1)$.

Proof. Let $\boldsymbol{t} = \overrightarrow{\mathsf{cool}}(\boldsymbol{r})$. Also, let $f = |\text{🌡}(\boldsymbol{r})|$ and $h = |\lceil\!\llcorner(\boldsymbol{r})|$ and $g = |\lceil\!\llcorner(\boldsymbol{r})|$, to align with Definition 3.4.3. By (2.1) and Lemma 2.4.5,

$$\boldsymbol{r} = \overleftarrow{\mathsf{shift}}(\boldsymbol{s}, k+1, f).$$

Therefore,

$$r_{k+2} r_{k+3}\cdots r_n = s_{k+2} s_{k+3}\cdots s_n.$$

There are now two cases to consider depending on how far s_{k+1} is left-shifted when creating $\boldsymbol{r}$. In the first case, s_{k+1} is left-shifted past all smaller symbols but not past any symbols larger than s_{k+1}. In this case, $f < k$ and $k < h$. Therefore, $f < h$ and so by Definition 3.4.3,

$$\boldsymbol{t} = \overrightarrow{\mathsf{max}}(\boldsymbol{r}, f, h).$$

Since $h \geq k+1$ and $\overleftarrow{\mathsf{greedy}}(\boldsymbol{s}, k+1) = \overleftarrow{\mathsf{shift}}(\boldsymbol{s}, k+1, f)$, then

$$\boldsymbol{t} = \overrightarrow{\mathsf{shift}}(f, j,$$

) for $j \geq k+1$. Since $\boldsymbol{r} = \overleftarrow{\mathsf{shift}}(\boldsymbol{s}, k+1, f)$ and $\boldsymbol{t} = \overrightarrow{\mathsf{shift}}(f, j,)$ for $j \geq k+1$, then $\boldsymbol{t} = \overrightarrow{\mathsf{shift}}(\boldsymbol{s}, k+1, j)$. Therefore,

$$t_1 t_2 \cdots t_k = s_1 s_1 \cdots s_k.$$

Therefore, $\mathsf{short}(\circlearrowleft_{k+2}(\boldsymbol{s}))$ begins at the $(k+2)$nd position of $\boldsymbol{r} \cdot \boldsymbol{t}$.

In the second case, s_{k+1} is left-shifted past all smaller symbols and past at least one symbol larger than s_{k+1}. In this case, $f = h$ and $h < g$ and $r_h > r_g$. Therefore, by Definition 3.4.3,

$$\boldsymbol{t} = \overrightarrow{\mathsf{max}}(\boldsymbol{r}, f, g).$$

However, it is also true that $g \geq k+1$. Since $\boldsymbol{r} = \overleftarrow{\mathsf{shift}}(\boldsymbol{s}, k+1, f)$ and $\boldsymbol{t} = \overrightarrow{\mathsf{shift}}(f, g,)$ for $g \geq k+1$, then $\boldsymbol{t} = \overrightarrow{\mathsf{shift}}(\boldsymbol{s}, k+1, g)$. Therefore,

$$t_1 t_2 \cdots t_k = s_1 s_1 \cdots s_k.$$

Therefore, $\mathsf{short}(\circlearrowleft_{k+2}(\boldsymbol{s}))$ begins at the $(k+2)$nd position of $\boldsymbol{r} \cdot \boldsymbol{t}$. □

Lemma 3.4.9 (Rotations $|\llcorner(\boldsymbol{s})| + 3$ to n)**.** *Given a bubble language* $\mathbf{L}$*, and* $\boldsymbol{s} \in \mathbf{L}$ *with* $k = |\llcorner(\boldsymbol{s})|$*, then* $\mathsf{short}(\circlearrowleft_i(\boldsymbol{s}))$ *is a substring of* $\bigodot \overrightarrow{\mathcal{C}}(\mathbf{L})$ *for all* $k+3 \leq i \leq n$*. In particular,* $\mathsf{short}(\circlearrowleft_i(\boldsymbol{s}))$ *begins at the* i*th position of* $\boldsymbol{r} \cdot \overrightarrow{\mathsf{cool}}(\boldsymbol{r})$ *where* $\boldsymbol{r} = \overleftarrow{\mathsf{cool}}(\boldsymbol{s})$*.*

Proof. By $\boldsymbol{r} = \overleftarrow{\mathsf{cool}}(\boldsymbol{s})$ and Definition 3.1.3,

$$r_{k+3} r_{k+4} \cdots r_n = s_{k+3} s_{k+4} \cdots s_n.$$

Since $\overrightarrow{\mathsf{cool}}(\boldsymbol{r}) = \overrightarrow{\mathsf{cool}}(\overleftarrow{\mathsf{cool}}(\boldsymbol{s})) = \boldsymbol{s}$ by Theorem 3.4.4, then $\mathsf{short}(\circlearrowleft_i(\boldsymbol{s}))$ begins at the ith position of $\boldsymbol{r} \cdot \overrightarrow{\mathsf{cool}}(\boldsymbol{r})$ for all i within $k+3 \leq i \leq n$. □

Theorem 3.4.10 (Shorthand Rotation Property)**.** *If* $\mathbf{L}$ *is a bubble language and* $\boldsymbol{s} \in \mathbf{L}$*, then there exists* $\boldsymbol{t} \in \mathbf{L}$ *such that the substring of length* $n-1$ *in* $\boldsymbol{t} \cdot \overrightarrow{\mathsf{cool}}(\boldsymbol{t})$ *equals*

$\mathsf{short}(\circlearrowleft)\,(\boldsymbol{s})i$. *In other words,* $\bigodot \vec{\mathcal{C}}(\mathbf{L})$ *contains every shorthand rotation of every string in* $\mathbf{L}$ *as a substring.*

Proof. The proof follows immediately from Lemmas 3.4.6-3.4.9. □

Chapter 4

Applications

"There is a deeply satisfying feeling that one obtains by watching a well ordered list of combinatorial objects marching down the computer screen."

- Frank Ruskey

This chapter investigates two types of applications for the results in this thesis. The first application is in creating efficient algorithms for generating combinatorial objects. The second application involves the creation of shorthand universal cycles.

4.1 Algorithms

"Algorithms existed for at least five thousand years, but people did not know that they were algorithmizing."

- Doron Zeilberger

By Theorem 3.3.5 any bubble language can be generated by the cool left-shift operation. Furthermore, the pseudocode on page 128 provides a simple algorithm that works for all bubble languages. This pseudocode can be optimized in different ways depending on the specific bubble language. (The only optimization made in the pseudocode involves an erroneous value for k on the first iteration.) This section will discuss three generation algorithms that have already appeared in print: combinations [73] in Section 4.1.1, balanced parentheses [72] in Section 4.1.2, and multiset permutations [102] in Section 4.1.3. The efficiency of these *cool-lex generation algorithms* is briefly discussed before presenting the individual algorithms.

Since cool left-shifts are greedy left-shifts involving the first or second symbol following the non-increasing prefix of each string, the efficiency of these generation algorithms depends upon the efficiency of performing these greedy left-shifts. In the

case of combinations and multiset permutations the greedy left-shifts are greatly simplified since every shift produces a string in the language. That is, the corresponding algorithms always shift symbols into the first position. The greedy left-shifts are also simplified in the case of generating balanced parentheses. In particular, left-shifting the last symbol in a prefix of the form 1^i0^j10 is not possible when $i = j$. When $i > j$ then the last symbol in the prefix 1^i0^j10 can be greedily left-shifted until it reaches the second position, thereby replacing the prefix 1^i0^j10 by $101^{i-1}0^j1$. On the other hand, a greedy left-shift of the last symbol in a prefix of the form 1^i0^j1 will always produce $1^{i+1}0^j$.

The above discussion on balanced parentheses also points out an important simplification involving the generation of fixed-density bubble languages. Notice that a greedy left-shift of the last symbol in a prefix of the form 1^i0^j1 will always cause the prefix to be replaced by some $1^i0^h10^{j-h}$. Although this thesis describes the operation as a left-shift, the change between 1^i0^j1 and $1^i0^h10^{j-h}$ can also be described as a single *transposition*. In particular, the two prefixes differ by a transposition of the symbols in position $i + j + 1$ and $i + h + 1$. Similarly, a greedy left-shift of the last symbol in a prefix of the form 1^i0^j10 can always be described using either one or two transpositions. Therefore, the left-shift Gray codes presented in this thesis for fixed-density bubble languages are also 2-*transposition Gray codes*, meaning that each pair of successive strings differ by either one or two transpositions. Algorithmically, this means that the fixed-density bubble languages can be generated using operations that are basic to arrays instead of linked lists. For this reason, the algorithms presented in Sections 4.1.1 and 4.1.2 utilizes an array, whereas the algorithm presented in Section 4.1.3 utilizes a linked list. The formal definition of a transposition appears below. (In general, the cool-lex left-shift Gray codes presented in this thesis are also $(m + 2)$-transposition Gray codes, where m is the number of distinct symbols in the multiset of symbols. When $m = 2$ only half of these $m + 2 = 4$ transpositions are required.)

Definition 4.1.1 (Transpositions)**.** *The* transposition *of the ith and jth symbols within $\boldsymbol{s} = s_1s_2\cdots s_n$ where $1 \le i \le j \le n$ is*

$$\mathsf{transpose}(\boldsymbol{s}, i, j) = s_1s_2\cdots s_{i-1}s_js_{i+1}s_{i+2}\cdots s_{j-1}s_is_{j+1}s_{j+2}\cdots s_n.$$

In other words, the transposition puts the value s_j into position i and the value s_i into position j.

Each of the algorithms presented in this chapter is loopless and uses a constant

number of additional variables. As mentioned in Section 1.2, the algorithms generate the strings *in-place* using a *single shared object*. This means that a single array or linked list stores the current string in cool-lex order, and then the string is modified (using transpositions or shifts) to create the next string in the cool-lex order. After each modification the visit informs the user of the algorithm that the next string is ready for consideration. In this context the term *loopless* means that each successive visit is called in worst-case O(1)-time, where the hidden constant does not depend on the parameters of the particular language being considered. Similarly, the term *constant number of additional variables* means that the algorithms can only use O(1) simple variables, not including the single shared object storing the current string. (In this thesis the simple variables include only individual pointers and single integers that can range in value from 1 up to n.) In the presented implementations the visit call passes the array or linked list to the user of the algorithm. However, in some applications it may provide greater efficiency to instead pass the modification that produced the new string. For example, if each string has an associated value, then passing the modification may allow the user of the algorithm to update the value of each successive string in a manner that is also loopless and uses a constant number of additional variables. Chapter 5 discusses loopless algorithms for additional combinatorial objects.

4.1.1 Combinations

"There are many ways to make a combination, it isn't such a very sticky chore."
- **Jiminy Cricket** singing on *Adding Combinations*[16]

In this section it will be assumed that $\mathbf{L} = \mathbf{C}(s,t)$. That is, the language being generated is the set of all binary strings containing s copies of 0 and t copies of 1. As previously mentioned, languages of this type are also known as (s,t)-combinations (or simply combinations) since each string in $\mathbf{C}(s,t)$ can be used to represent a different choice of t elements from a set of size $s+t$. Within the presented algorithm y represents the position of the leftmost 0 in the current string, and x represents the position of the leftmost 1 in the current string such that $x > y$. The following remarks reexpress the greedy left-shifts found in Definition 3.1.3 given these assumptions.

Remark 4.1.2 ($\overleftarrow{\mathsf{greedy}}(\boldsymbol{s}, k+1)$ for combinations)**.** *Suppose* $\boldsymbol{s} = 1^{y-1}0^{x-y}11\boldsymbol{z} \in \mathbf{C}(s,t)$ *with* $0 < y < x$ *and* $k = |\llcorner(\boldsymbol{s})| = x - 1$. *Then,*

$$\overleftarrow{\mathsf{greedy}}(\boldsymbol{s}, k+1) = \overleftarrow{\mathsf{shift}}(\boldsymbol{s}, x, 1) = 1^y0^{x-y}1\boldsymbol{z} = \mathsf{transpose}(\boldsymbol{s}, y, x).$$

In other words, greedy left-shifting the symbol following the non-increasing prefix is equivalent to transposing the xth and yth symbol.

Remark 4.1.3 ($\overleftarrow{\mathsf{greedy}}(\boldsymbol{s}, k+2)$ for combinations)**.** *Suppose* $\boldsymbol{s} = 1^{y-1}0^{x-y}10\boldsymbol{z} \in \mathbf{C}(s,t)$ *with* $0 < y < x$ *and* $k = |\mathsf{L}(\boldsymbol{s})| = x - 1$*. Then,*

$$\overleftarrow{\mathsf{greedy}}(\boldsymbol{s}, k+2) = \overleftarrow{\mathsf{shift}}(\boldsymbol{s}, x+1, 1) = 01^{y-1}0^{x-y}1\boldsymbol{z} = \mathsf{transpose}(\mathsf{transpose}(\boldsymbol{s}, y, x), 1, x+1).$$

In other words, greedy left-shifting 0 *when it is the second symbol following the non-increasing prefix is equivalent to transposing the xth and yth symbol followed by transposing the first and $(x+1)$st symbols in the result.*

To further simplify the cool left-shift operation in the case of combinations, consider the condition

$$s_k < s_{k+2} \text{ or } \overleftarrow{\mathsf{greedy}}(\boldsymbol{s}, k+2) = \boldsymbol{s}$$

found in (3.1b) of Definition 3.1.3. Notice that $\overleftarrow{\mathsf{greedy}}(\boldsymbol{s}, k+2) \neq \boldsymbol{s}$ whenever $k = |\mathsf{L}(\boldsymbol{s})|$ and $k+2 \leq n$. Furthermore, the non-increasing prefix of every string in a non-trivial fixed-density bubble languages must end in 0. Therefore, $s_k = 0$. For these reasons the above condition is simplified to $s_{k+2} = 1$. This observation together with $x = k+1$ and the previous two remarks allows the cool left-shift operation for combinations to be simplified as follows

$$\overleftarrow{\mathsf{cool}}(\boldsymbol{s}) = \begin{cases} \mathsf{transpose}(\boldsymbol{s}, y, x) & \text{if } s_{x+1} = 1 \quad (4.1a) \\ \mathsf{transpose}(\mathsf{transpose}(\boldsymbol{s}, y, x), 1, x+1) & \text{otherwise.} \quad (4.1b) \end{cases}$$

(The above expression assumes that the non-increasing prefix ends at least two symbols before the end of $\boldsymbol{s}$. The two strings that do not satisfy this assumption are discussed after the algorithms are presented.)

In both of the following algorithms — **CoolCombo**(s,t) and **BranchlessCombo**(s,t) — the current string is stored in array b and $b[i]$ refers to the ith symbol in b. Both algorithms produce the same output, but **BranchlessCombo**(s,t) is a *branchless algorithm* since it uses standard techniques to remove the two if-statements found in **CoolCombo**(s,t). (Within this algorithm it is also necessary to always perform two transpositions, even when one is sufficient.)

The algorithm **CoolCombo**(s,t) is now explained in more detail. Remarks 4.1.2 and 4.1.3 show how each successive string can be efficiently modified into the next string by using the values of x and y. In particular, lines 6 and 7 in **CoolCombo**(s,t) perform the transposition of the xth and yth symbols, while lines 11 and 12 perform

the transposition of the first and $(x+1)$st symbol whenever the latter is 0. (Notice that the $x+1 = k+2$ and that the value of x is incremented between the two previously mentioned pairs of lines.) Remarks 4.1.2 and 4.1.3 also show how the values of x and y can be efficiently updated to reflect these modifications. When the $(x+1)$st symbol is 1 then Remark 4.1.2 shows that the updates simply involve incrementing the two values as performed on lines 8 and 9. On the other hand, when the $(x+1)$st symbol is 0 then Remark 4.1.3 shows that the updates have two cases depending on whether or not $1^{y-1} = \epsilon$ in the resulting string $01^{y-1}0^{x-y}1\boldsymbol{z}$. The algorithm tests if the $(x+1)$st symbol is 0 on line 10. Within the if-block the value of y is set to 1 on line 16 to reflect the fact that the resulting string begins with 0. The value of x either remains incremented or is reset to 2 depending on whether or not $1^{y-1} = \epsilon$. (Again notice that the value of y is incremented before reaching the if-statement on line 13.)

```
Require: t > 0
 1: b ← array(1^t 0^s)
 2: x ← t
 3: y ← t
 4: visit(b)
 5: while x < s + t do
 6:    b[x] ← 0
 7:    b[y] ← 1
 8:    x ← x + 1
 9:    y ← y + 1
10:    if b[x] = 0
11:       b[x] ← 1
12:       b[1] ← 0
13:       if y > 2
14:          x ← 2
15:       end
16:       y ← 1
17:    end
18:    visit(b)
19: end
```

Algorithm 2: **CoolCombo**(s,t) is a loopless algorithm that generates (s,t)-combinations in array b using the additional variables x and y.

In Algorithms 2 and 3 the last string in cool-lex order

$$\mathsf{last}(\overleftarrow{\mathcal{C}}(\mathbf{L})) = \llcorner = 1^t0^s$$

is generated first instead of last for consistency with the published versions [70, 73].

Require: $t > 0$
1: $b \leftarrow \text{array}(1^t 0^s)$
2: $x \leftarrow t$
3: $y \leftarrow t$
4: visit(b)
5: **while** $x < s + t$ **do**
6: $\quad b[x] \leftarrow 0$
7: $\quad b[y] \leftarrow 1$
8: $\quad b[1] \leftarrow b[x+1]$
9: $\quad b[x+1] \leftarrow 1$
10: $\quad x \leftarrow x + 1 - (x-1) \cdot b[2] \cdot (1 - b[1])$
11: $\quad y \leftarrow b[1] \cdot y + 1$
12: $\quad$ visit(b)
13: **end**

Algorithm 3: BranchlessCombo(s,t) is a loopless and branchless algorithm that generates (s,t)-combinations in array b using the additional variables x and y.

The initializations on lines 2 and 3 allow this string to be changed into the first string in cool-lex order

$$\mathsf{first}(\overleftarrow{\mathcal{C}}(\mathbf{L})) = \overleftarrow{\mathsf{greedy}}(1^t 0^s, s+t) = 01^t 0^{s-1}$$

after the first iteration. The last string generated by the algorithms is then

$$\mathsf{last}(\overleftarrow{\mathcal{C}}(\mathbf{L}/1)) \cdot 1 = 1^{t-1} 0^s 1$$

which is the unique string where $x = s + t$ and explains the terminating condition on line 5. (Expressions for the first and last strings in cool-lex order can be found in Lemma 3.2.4.)

Before concluding this section it is mentioned that Knuth [45] has created a loopless and branchless implementation of **CoolCombo**(s,t) for the MMIX architecture. This implementation is presented below with a C implementation to its right. The $\ominus$ on line 9 refers to the special *saturating substraction operation* (also known as *monus* or *dot minus*) found in MMIX and is used to avoid the simple if-statement found in the C implementation.

4.1.2 Balanced Parentheses

In this section it will be assumed that $\mathbf{L} = \mathbf{P}(t)$. That is, the language being generated is the set of all balanced parentheses containing t copies of 1 and t copies of 0, where each 1 represents an open parenthesis and each 0 represents a closed parenthesis.

Require: $t > 0$
1: $R_2 \leftarrow (1 \ll s + t)$
2: $R_3 \leftarrow (1 \ll t) - 1$
3: **while** $R_3 \wedge R_2 = 0$ **do**
4: visit(R_3)
5: $R_0 \leftarrow R_3 \wedge (R_3 + 1)$
6: $R_1 \leftarrow R_0 \oplus (R_0 - 1)$
7: $R_0 \leftarrow R_1 + 1$
8: $R_1 \leftarrow R_1 \wedge R_3$
9: $R_0 \leftarrow (R_0 \wedge R_3) \ominus 1$
10: $R_3 \leftarrow R_3 + R_1 - R_0$
11: **end**

Algorithm 4: WordCombo(s, t) is a loopless and branchless algorithm for the MMIX architecture that generates computer words with s zeros and t ones stored in register R_3 using the additional registers R_0, R_1, and R_2.

```
void WordCombo(int s, int t) {
   unsigned int R0, R1, R2, R3;
   R2 = (1 << s+t);
   R3 = (1 << t) - 1;
   while ((R3 & R2) == 0) {
      visit(R3);
      R0 = R3 & (R3+1);
      R1 = R0 ^ (R0-1);
      R0 = R1 + 1;
      R1 = R1 & R3;
      R0 = ((R0 & R3) >= 1) ? (R0 & R3) - 1 : 0;
      R3 = R3 + R1 - R0;
   }}
```

Algorithm 5: C implementation of Algorithm 4.

The presented **CoolCat**(t) algorithm is remarkably similar to **CoolCombo**(t,t) and uses the same conventions for the values of x and y and b. (The term **CoolCat** pays homage to the fact that $|\mathbf{P}(t)|$ is the tth Catalan number.) The similarity between the algorithms is largely due to the following remark that shows the operation of greedy left-shifting the symbol following the non-increasing prefix is identical in combinations and balanced parentheses.

Remark 4.1.4 ($\overleftarrow{\mathsf{greedy}}(\boldsymbol{s}, k+1)$ for balanced parentheses)**.** *Suppose* $\boldsymbol{s} = 1^{y-1}0^{x-y}11\boldsymbol{z} \in \mathbf{P}(t)$ *with* $1 < y < x$ *and* $k = |\overline{\mathsf{L}}(\boldsymbol{s})| = x - 1$*. Then,*

$$\overleftarrow{\mathsf{greedy}}(\boldsymbol{s}, k+1) = \overleftarrow{\mathsf{shift}}(\boldsymbol{s}, x, 1) = 1^y0^{x-y}1\boldsymbol{z} = \mathsf{transpose}(\boldsymbol{s}, y, x).$$

In other words, greedy left-shifting the symbol following the non-increasing prefix is equivalent to transposing the x*th and* y*th symbol.*

On the other hand, there are two differences between combinations and balanced parentheses when greedy left-shifting a 0 that is the second symbol following the non-increasing prefix. First, the greedy left-shift is trivial if the non-increasing prefix is itself balanced (contains the same number of 0s and 1s). That is, the greedy left-shift does not change the string if and only if $x - 1 = 2(y - 1)$. (Recall that y represents the position of the first 0 in the current string and x represents the position of the first 1 in the current string with $x > y$.) On the other hand, when the non-increasing prefix contains more 0s than 1s, then the greedy left-shift can shift the 0 as far as the second position. These results are formalized in the following remark.

Remark 4.1.5 ($\overleftarrow{\mathsf{greedy}}(\boldsymbol{s}, k+2)$ for balanced parentheses)**.** *Suppose* $\boldsymbol{s} = 1^{y-1}0^{x-y}10\boldsymbol{z} \in \mathbf{P}(t)$ *with* $1 < y < x$ *and* $k = |\overline{\mathsf{L}}(\boldsymbol{s})| = x-1$*. Then,* $\overleftarrow{\mathsf{greedy}}(\boldsymbol{s}, k+2) = \boldsymbol{s}$ *if* $x-1 = 2(y-1)$ *and otherwise*

$$\overleftarrow{\mathsf{greedy}}(\boldsymbol{s}, k+2) = \overleftarrow{\mathsf{shift}}(\boldsymbol{s}, x+1, 2) = 101^{y-2}0^{x-y}1\boldsymbol{z} = \mathsf{transpose}(\mathsf{transpose}(\boldsymbol{s}, y, x), 2, x+1).$$

In other words, greedy left-shifting a 0 *when it is the second symbol following the non-increasing prefix is trivial when the non-increasing prefix contains the same number of* 0*s and* 1*s, and is otherwise equivalent to transposing the* x*th and* y*th symbol followed by transposing the second and* $(x+1)$*st symbols in the result.*

These remarks allow the cool left-shift operation ($\overleftarrow{\mathsf{cool}}(\boldsymbol{s})$) for balanced parenthe-

ses to be simplified as follows

$$= \begin{cases} \mathsf{transpose}(\boldsymbol{s}, y, x) & \text{if } s_{x+1} = 1 \text{ or } x - 1 = 2(y-1) \quad (4.2a) \\ \mathsf{transpose}(\mathsf{transpose}(\boldsymbol{s}, y, x), 2, x+1) & \text{otherwise.} \quad (4.2b) \end{cases}$$

(The above expression assumes that the non-increasing prefix ends at least two symbols before the end of $\boldsymbol{s}$. The only string that does not satisfy this assumption is the non-increasing string, since $1^{t-1}0^t1 \notin \mathbf{P}(t)$, and this string is discussed after the algorithm is presented.)

```
Require: t > 0
 1: b ← array(1^t 0^t)
 2: x ← t
 3: y ← t
 4: visit(b)
 5: while x < 2t − 1 do
 6:    b[x] ← 0
 7:    b[y] ← 1
 8:    x ← x + 1
 9:    y ← y + 1
10:    if b[x] = 0
11:       if x = 2y − 2
12:          x ← x + 1
13:       else
14:          b[x] ← 1
15:          b[2] ← 0
16:          x ← 3
17:          y ← 2
18:       end
19:    end
20:    visit(b)
21: end
```

Algorithm 6: **CoolCat**(t) generates balanced parentheses of length $2t$ ($\mathbf{P}(t)$) in array b using the additional variables x and y

When comparing **CoolCat**(t) and **CoolCombo**(s,t), notice that the lines between 14 and 17 account for the difference in non-trivial shifts found in Remark 4.1.3 and 4.1.5. Again the **CoolCat**(t) algorithm begins by visiting 1^t0^t for consistency with the published version [72], although this time the termination condition is $x < 2t - 1$ instead of $x < 2t$ due to the fact that $1^{t-1}0^t1 \in \mathbf{C}(t,t)$ and $1^{t-1}0^t1 \in \mathbf{P}(t)$. Finally, the additional increment of x in line 12 is explained by the following observation: If $1^i0^i10\boldsymbol{z} \in \mathbf{P}(t)$ then the first symbol in $\boldsymbol{z}$ is equal to 1. This allows the new

value of x to be computed without scanning the resulting string.

It is mentioned that the algorithm presented in [72] is slightly more general than **CoolCat**(t) since it generates any $\mathbf{P}(t)/0^{t-s}$. See [2] for an implementation and practical run-time analysis. Chapter 5 discusses how cool-lex order can be used to generate binary trees directly.

4.1.3 Multiset Permutations

In this section it will be assumed that $\mathbf{L} = \Pi(\mathbb{M})$. That is, the language being generated is the set of all permutations of the multiset $\mathbb{M}$. The presented algorithm is remarkable for the fact that it stores no information about this multiset. In particular, the algorithm does not store its number of total symbols, its number of distinct symbols, or the relative frequencies of any of its symbols. The cool left-shift is simplified below. In the case of multiset permutations, the greedy left-shifts are non-trivial and result in the symbol being shifted into the first position. Again, k is the length of the non-increasing prefix of string $\boldsymbol{s}$ with length n. That is, $k = |\gamma(\boldsymbol{s})|$ and $n = |\boldsymbol{s}|$.

$$\overleftarrow{\mathsf{cool}}(\boldsymbol{s}) = \begin{cases} \overleftarrow{\mathsf{shift}}(\boldsymbol{s}, n, 1) & \text{if } k = n \text{ or } k = n-1 \quad (4.3a) \\ \overleftarrow{\mathsf{shift}}(\boldsymbol{s}, k+1, 1) & \text{if } k \le n-2 \text{ and } s_k < s_{k+2} \quad (4.3b) \\ \overleftarrow{\mathsf{shift}}(\boldsymbol{s}, k+2, 1) & \text{otherwise.} \quad (4.3c) \end{cases}$$

The algorithm stores the current string in a singly-linked list pointed to by its head pointer `h`. Each node in the linked list has a *value field* named *val* (storing the symbol) and a *next field* named *next*. The omitted `init`$(\mathbb{M})$ call creates a singly-linked list storing the symbols of $\mathbb{M}$ in non-increasing order with `h`, `i`, and `j` pointing to its first, second-last, and last nodes, respectively. The `i` and `j` pointers are used for keeping track of the location of the non-increasing prefix and are again discussed after the algorithm. The two other pointers used in the algorithm, `s` and `t`, are used for performing each shift. (The algorithm can also be implemented using two pointers instead of four, although the resulting program is slightly more complicated.) It is assumed that $\mathbb{M}$ contains at least two distinct symbols. The first three iterations of **MultiCool**$(\{1,1,2,4\})$ are illustrated in Figure 4.1.

The last string in cool-lex order, the non-increasing string γ, is again visited first instead of last. The algorithm then continues until this string is encountered again. For this reason, the first iteration is a special case since `init`$(\mathbb{E})$ initializes `i` to be "off-by-one". After the first iteration, `i` points to the last node in the multiset permutation's non-increasing prefix and `j` points to the next node.

```
[h, i, j] ← init(𝕄)
visit(h)
while j.next ≠ φ or j.val < h.val do
  if j.next ≠ φ and i.val ≥ j.next.val
    s ← j
  else
    s ← i
  end
  t ← s.next
  s.next ← t.next
  t.next ← h
  if t.val < h.val
    i ← t
  end
  j ← i.next
  h ← t
  visit(h)
end
```

Algorithm 7: MultiCool($\mathbb{M}$) is a loopless algorithm that generates the permutations of multiset $\mathbb{M}$ in singly linked list with head `h` using additional pointers `i`, `j`, `s`, and `t`.

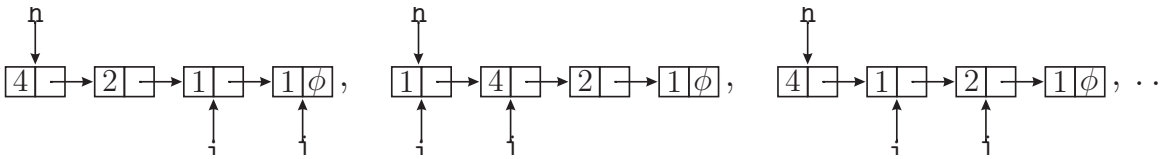


Figure 4.1: The first three `visit` calls in **MultiCool**($\{1, 1, 2, 4\}$) will produce the configurations given above, where left and right boxes of each node refer to its value (*.val*) and next (*.next*) fields.

Chapter 5 discusses the efficiency of implementing **MultiCool**($\mathbb{M}$) using an array instead of a linked list.

4.2 Shorthand Universal Cycles

"Only the old people write me in longhand!"
- Brett Somers

"I am told that I talk in shorthand and then smudge it."
- J. R. R. Tolkien

The second discussed type of application involves the construction of shorthand universal cycles. These combinatorial objects are intimately related to shift Gray

codes, and this section begins by outlining several of these connections. Recall from Section 1.2.3 that *universal cycles* generalize the concept of de Bruijn cycles by packing every string of a language exactly once into a single circular string. Also recall that universal cycles for fixed-content languages require an alternate representation of each string (except in trivial cases). For instance, the shorthand representation removes the last symbol from a string (see Definition 3.4.5) and a *shorthand universal cycle* is a universal cycle containing the shorthand of every string in a language. For example, the following string

$$321131123121 \tag{4.4}$$

is a shorthand universal cycle for $\mathbf{L} = \Pi(\{1,1,2,3\})$ (multiset permutations) since its substrings of length $n-1=3$ are

$$321, 211, 113, 131, 311, 112, 123, 231, 312, 121, 213, 132 \tag{4.5}$$

and comprise the shorthand representation of every string in $\mathbf{L}$. Shorthand representation is particularly useful for fixed-content languages since the removed last symbol is redundant and is uniquely determined by the remaining symbols. In the above case, given any three symbols in $\{1,1,2,3\}$ then the fourth symbol is uniquely determined. Thus, the substrings in (4.5) can be extended to strings in $\mathbf{L}$ by simply suffixing the missing symbol

$$3211, 2113, 1132, 1312, 3112, 1123, 1231, 2311, 3121, 1213, 2131, 1321. \tag{4.6}$$

As mentioned in Section 1.2.3 shorthand universal cycles for fixed-content languages are also interesting due to their associated circular right-shift Gray codes in which each $\boldsymbol{s}$ is followed by either $\overrightarrow{\mathsf{shift}}(\boldsymbol{s}, 1, n)$ or $\overrightarrow{\mathsf{shift}}(\boldsymbol{s}, 1, n-1)$. Since there are two choices for each transition, then the Gray code can also be represented by a binary string of length $|\mathbf{L}|$. For example, the permutations in (4.6) can be generated by the following binary string

$$001100010010 \tag{4.7}$$

where the ith bit is 0 or 1 if the ith permutation can be transformed into the $(i+1)$st permutation by a right-shift of length n or $n-1$, respectively. That is, the permutations begin $\overrightarrow{3211}, \overrightarrow{2113}, \overrightarrow{1132}, \ldots$ and so the binary string begins $001\cdots$. (The seventh and twelfth bits in (4.7) could be 0 or 1 since $\overrightarrow{1231} = \overrightarrow{1231}$ and $\overrightarrow{1321} = \overrightarrow{1321}$.) Given a cyclic right-shift Gray code for $\mathbf{L}$ using only $\overrightarrow{\mathsf{shift}}(\boldsymbol{s}, 1, n)$ and $\overrightarrow{\mathsf{shift}}(\boldsymbol{s}, 1, n-1)$, then an associated shorthand universal cycle can be obtained by concatenating

the first symbol of each string. For example, the shorthand universal cycle in (4.4) simply contains the first symbol of each permutation in (4.6). These observations are summarized below.

Remark 4.2.1 (Shorthand Universal Cycles and Shift Gray Codes)**.** *If* $\mathbf{L}$ *is a fixed-content language, then* $\mathbf{L}$ *has a shorthand universal cycle if and only if it has a cyclic Gray code where successive strings are obtained by right-shifting the first symbol into the last or second-last position.*

Since right-shifts are transformed into left-shifts by reversing the order of strings (or by reversing each individual string) then shorthand universal cycles also provide special cases of left-shift Gray codes. Shorthand universal cycles are also related to shift Gray codes in another important way. Suppose $\boldsymbol{s}$ and $\boldsymbol{t}$ are consecutive non-overlapping substrings of length n in a shorthand universal for a fixed-content language $\mathbf{L}$

$$\cdots s_1 s_2 \cdots s_n t_1 t_2 \cdots t_n \cdots.$$

To ensure that the substrings in the universal cycle have the correct content, notice that the ith occurrence of symbol j within $\boldsymbol{t}$ cannot occur more than one position to the left of the ith occurrence of symbol j within $\boldsymbol{s}$. To make this observation more concrete, consider a shorthand universal cycle for the permutations of $\{1, 1, 2, 3\}$ containing the following substring

$$\cdots 112\underline{31}\underline{3}21\cdots.$$

Notice that the first occurrence of 3 in $\boldsymbol{s} = 1123$ appears at position 4, while the first occurrence of 3 in $\boldsymbol{t} = 1321$ appears in position 2. Therefore, the underlined substring above contains too many copies of 3. One way to ensure that shorthand universal cycles have substrings with the correct content is to ensure that successive substrings of length n form a right-shift Gray code. This observation suggests that reverse cool-lex order could be of use in the construction of shorthand universal cycles. For example, the shorthand universal cycle given in (4.4) is simply the concatenation of $\mathbf{N}(\{1, 1, 2, 3\})$ (multiset necklaces) in reverse cool-lex order

$$\overrightarrow{\mathcal{C}}(\mathbf{N}(\{1, 1, 2, 3\})) = 3\overrightarrow{211}, 31\overrightarrow{12}, \overrightarrow{3121}.$$

On the other hand, arbitrary right-shift Gray codes of necklaces do not have this property. For example, the concatenation of the following circular right-shift Gray

for $\mathbf{N}(\{0,0,0,0,1,1,1\})$ is not a shorthand universal cycle

$$000\overrightarrow{011}1, 00\overrightarrow{011}\underline{101}, \underline{0010}1\overrightarrow{\underline{10}}1, \underline{00\overrightarrow{10}}011, 000\overrightarrow{10}11$$

since the underlined substring 010010 appears twice.

This section proves that shorthand universal cycles exist for multiset permutations. Furthermore, the proof is constructive and uses reverse cool-lex order in much the same way as lexicographic order is used to construct de Bruijn cycles in the FKM algorithm. The main theorem is contained in Section 4.2.3. Sections 4.2.1 and 4.2.2 discuss the interesting special cases of permutations and binary strings, and introduce notation and intermediate results. The end of Section 4.2.3 also briefly discusses the efficient generation of shorthand universal cycles and the creation of shorthand universal cycles that avoid periodic strings.

4.2.1 Permutations

Prior to this thesis, the only explicitly constructed shorthand universal cycles were for permutations. Historically, the first construction arose from the bell-ringing community [33], while the first published construction appeared quite recently [71]. Although their associated algorithms are not discussed in this thesis, both constructions allow the shorthand universal cycles to be generated in O(1)-time per symbol. A similar algorithmic result may also hold for the construction provided in this thesis (see the note at the end of this chapter). Shorthand universal cycles for permutations are also of particular interest due to their one-to-one correspondence with Hamiltonian cycles in directed Cayley graphs on the symmetric group generated by σ_n and σ_{n-1} where σ_k is the rotation $(1\ 2\ \cdots\ k)$ (see [71] for more details). The three constructions are now discussed in more detail.

Bell-Ringer's Construction

The bell-ringer's construction can be succinctly described in terms of the associated Gray code for permutations, and relies on two values. Given $p_1p_2\cdots p_n \in \Pi(\{1,2,\ldots,n\})$ let j represent the maximum value of p_1 and p_n, and let k represent the maximum value such that $n\ n-1\ \cdots k$ appears in the permutation as a circular substring. For example, if the permutation is 431265 then $j = 5$ (since 5 is the maximum of 4 and 5) and $k = 3$ (since 6543 appears as a circular substring). Given these values, the next

permutation is obtained by a prefix shift of length $n-1$ if

$$k-1 \leq j \leq n-1 \tag{4.8}$$

and otherwise the next permutation is obtained by a prefix shift of length n. For example, given the aforementioned permutation 431265 then the next permutation is 312645 since (4.8) is satisfied and so a prefix shift of length $n-1$ is performed. As a more complete example, the permutations of $\{1,2,3,4\}$ appear below along with each permutation's associated value of k and j (above as (k,j)) and the prefix shift used to obtain the next permutation (below)

$$\begin{array}{cccccccccccc}
\overset{(1,4)}{4321}_{0} & \overset{(1,4)}{3214}_{0} & \overset{(1,3)}{2143}_{1} & \overset{(4,3)}{1423}_{1} & \overset{(4,4)}{4213}_{0} & \overset{(4,4)}{2134}_{0} & \overset{(4,2)}{1342}_{0} & \overset{(4,3)}{3421}_{1} & \overset{(4,4)}{4231}_{0} & \overset{(4,4)}{2314}_{0} & \overset{(4,3)}{3142}_{1} & \overset{(1,2)}{1432}_{1} \\
\overset{(3,4)}{4312}_{0} & \overset{(3,4)}{3124}_{0} & \overset{(4,3)}{1243}_{1} & \overset{(4,3)}{2413}_{1} & \overset{(4,4)}{4123}_{0} & \overset{(4,4)}{1234}_{0} & \overset{(4,2)}{2341}_{0} & \overset{(4,3)}{3412}_{1} & \overset{(4,4)}{4132}_{0} & \overset{(4,4)}{1324}_{0} & \overset{(4,3)}{3241}_{1} & \overset{(3,2)}{2431}_{1}.
\end{array}$$

This order has been described as a variation of the Johnson-Trotter-Steinhaus order for permutations [33], which is known to bell-ringers as *plain changes* and dates back at least to 1668 [15]. The associated shorthand universal cycle is

$$432142134231431241234132$$

while the associated binary string is restated below

$$001100010011001100010011. \tag{4.9}$$

Contemporary Construction

The recently published construction can be succinctly formalized using the binary string interpretation. The base case occurs with 00 giving the permutations $21, 12$ when $n = 2$. If $x_1x_2\cdots x_{n!}$ is the binary string representation for generating $\Pi(n)$ by prefix-shifts then

$$001^{n-2}\ \overline{x}_1\ 001^{n-2}\ \overline{x}_2\ \cdots\ 001^{n-2}\ \overline{x}_{n!} \tag{4.10}$$

is the binary string representation for generating $\Pi(n+1)$ by prefix-shifts, where $\overline{x_i} = 1 - x_i$ and recall that 1^{n-2} represents $n-2$ copies of 1. For example, the binary string for $n = 3$ is

$$00\ \bar{0}\ 00\ \bar{0}\ = 001001. \tag{4.11}$$

Thus, the associated circular right-shift Gray code for $n = 3$ is

$$\overrightarrow{321}, \overrightarrow{213}, \overrightarrow{132}, \overrightarrow{312}, \overrightarrow{123}, \overrightarrow{231} \tag{4.12}$$

and the associated shorthand universal cycle is

$$321312.$$

Continuing the pattern given by (4.10), the binary string for $n = 4$ is obtained from (4.11) as follows

$$001\ \bar{0}\ 001\ \bar{0}\ 001\ \bar{1}\ 001\ \bar{0}\ 001\ \bar{0}\ 001\ \bar{1} = 001100110010001100110010. \tag{4.13}$$

The associated circular right-shift Gray code for $n = 4$ is

$$4321, 3214, 2143, 1423, 4213, 2134, 1342, 3412, 4132, 1324, 3241, 2431,$$
$$4312, 3124, 1243, 2413, 4123, 1234, 2341, 3421, 4231, 2314, 3142, 1432$$

and the associated shorthand universal cycle is

$$432142134132431241234231. \tag{4.14}$$

Interestingly, the universal cycle for $n = 4$ given in (4.14) can be obtained by prefixing 4 to the front of each permutation in the right-shift Gray code for $n = 3$ given in (4.12). That is,

$$432142134132431241234231 = \bigodot 4321, 4213, 4132, 4312, 4123, 4231.$$

This relationship holds in general, and provides an alternate method for constructing these shorthand universal cycles.

Cool-lex Construction

The shorthand universal cycle for $\Pi(\{1, 2, 3, 4\})$ arising from this thesis is

$$432142134123423143124132 \tag{4.15}$$

and the associated permutations of $\{1,2,3,4\}$ are

$$4321, 3214, 2143, 1423, 4213, 2134, 1342, 3412, 4123, 1234, 2341, 3421,$$
$$4231, 2314, 3142, 1432, 4312, 3124, 1243, 2413, 4132, 1324, 3241, 2431.$$

To illustrate the origins of the shorthand universal cycle in (4.15), consider the cool-lex order for multiset necklaces over $\{1,2,3,4\}$

$$\overleftarrow{\mathcal{C}}(\mathbf{N}(\{1,2,3,4\})) = 4132, 4312, 4231, 4123, 4213, 4321. \tag{4.16}$$

Notice that a shorthand universal cycle is not created by concatenating the strings in (4.16) since 242 would be an invalid substring within $431\underline{2}\cdot\underline{42}31$. On the other hand, reverse cool-lex order gives

$$\overrightarrow{\mathcal{C}}(\mathbf{N}(\{1,2,3,4\})) = 4321, 4213, 4123, 4231, 4312, 4132 \tag{4.17}$$

and the shorthand universal cycle in (4.15) is the concatenation of these strings. The following definition provides notation for succinctly expressing concatenations of this type.

Definition 4.2.2 (Cool-lex Shorthand Universal Cycle with Periodic Permutation Repetitions)**.** *Given a multiset of symbols* $\mathbb{M}$*, the* cool-lex shorthand universal cycle with periodic permutation repetitions *is*

$$\boldsymbol{U}_+(\mathbb{M}) = \bigodot \overrightarrow{\mathcal{C}}(\mathbf{N}(\mathbb{M})).$$

In other words, $\boldsymbol{U}_+(\mathbb{M})$ *is the concatenation of multiset necklaces in reverse cool-lex order.*

For example,

$$\boldsymbol{U}_+(\{1,2,3,4\}) = 4321\cdot 4213\cdot 4123\cdot 4231\cdot 4312\cdot 4132$$

is the concatenation of the strings in (4.17), and provides the shorthand universal cycle in (4.15). The name and symbology used in Definition 4.2.2 is due to the fact that $\boldsymbol{U}_+(\mathbb{M})$ can contain duplicated substrings, as illustrated in Section 4.2.2. On the other hand, $\boldsymbol{U}_+(\mathbb{M})$ is a shorthand universal cycle whenever $\mathbb{M}$ is a set. (In fact, Theorem 4.2.4 implies that $\boldsymbol{U}_+(\mathbb{M})$ is a shorthand universal cycle for $\Pi(\mathbb{M})$ if and only if the greatest common divisor of the multiplicities of the symbols in $\mathbb{M}$ is 1.)

The binary string associated with the cool-lex shorthand universal cycle in (4.15) is

$$001100100011001100100011. \tag{4.18}$$

While the binary strings in (4.9), (4.13), and (4.18) are identical up to rotation and reflection, the three constructions produce different shorthand universal cycles for larger values of n. Chapter 5 discusses the goal of minimizing and maximizing the sum of these binary strings.

4.2.2 Fixed-Density de Bruijn Cycles

While shorthand universal cycles for permutations have previously been constructed, shorthand universal cycles for fixed-density binary strings are also interesting. In particular, a shorthand universal cycle for $\mathbf{C}(k,k)$ is a *universal cycle for the middle levels*. That is,

$$\mathsf{short}(\mathbf{C}(k,k)) = \mathbf{C}(k-1,k) \cup \mathbf{C}(k,k-1)$$

where the *middle levels* are the strings in $\mathbf{C}(k-1,k) \cup \mathbf{C}(k,k-1)$. For example, the following string

$$11100011001010110100 \tag{4.19}$$

is a universal cycle for the middle levels with $k = 3$ since its substrings are those in $\mathbf{C}(2,3) \cup \mathbf{C}(3,2)$

$$11100, 11000, 10001, 00011, 00110, 01100, 11001, 10010, 00101, 01010,$$
$$10101, 01011, 10110, 01101, 11010, 10100, 01001, 10011, 00111, 01110.$$

On the other hand, the strings in $\mathbf{C}(3,3)$ can be obtained from those above by suffixing 0 and 1 to those in $\mathbf{C}(2,3)$ and $\mathbf{C}(3,2)$ respectively

$$111000, 110001, 100011, 000111, 001101, 011001, 110010, 100101, 001011, 010101,$$
$$101010, 010110, 101100, 011010, 110100, 101001, 010011, 100110, 001110, 011100.$$

For this reason, the string in (4.19) can also be considered a *fixed-density de Bruijn cycles*. In particular, the above strings are ordered by prefix right-shifts of length five and six that can easily be determined from (4.19). Due to the number of applications for binary de Bruijn cycles, it is also possible that fixed-density de Bruijn cycles could have significant value in the real-world. Hurlbert and Isaac [37] note that it is possible

to prove that fixed-density de Bruijn cycles exist by modifying de Bruijn's original arguments [13]. On the other hand, this proof does not lead to an efficient construction. Furthermore, the construction using reverse cool-lex order in Section 4.2.1 only works when the number of 0s and 1s are relatively prime (see Theorem 4.2.4). To illustrate the problem that occurs when the number of 0s and 1s share a common factor, consider the reverse cool-lex order for multiset necklaces over $\{0,0,0,1,1,1\}$

$$\overrightarrow{\mathcal{C}}(\mathbf{N}(\{0,0,0,1,1,1\})) = 111000, 110010, 101010, 110100. \tag{4.20}$$

In this case, the cool-lex shorthand universal cycle with periodic permutation repetitions is

$$\boldsymbol{U}_{+}(\{0,0,0,1,1,1\}) = 11100011\underline{0010101010}10100. \tag{4.21}$$

However, this is not a shorthand universal cycle since the underlined substring contains three copies of 01010 and three copies of 10101. Fortunately, the problem of repeated substrings can be fixed by including only the non-repeating or aperiodic prefix of each string in the concatenation.

Definition 4.2.3 (Aperiodic prefix)**.** *The* aperiodic prefix *of* $\boldsymbol{s} = s_1 s_2 \cdots s_n$ *is*

$$\mathsf{aperiodic}(\boldsymbol{s}) = s_1 s_2 \cdots s_p$$

where p *is the smallest divisor of* n *such that*

$$\boldsymbol{s} = \overbrace{s_1 s_2 \cdots s_p \cdot s_1 s_2 \cdots s_p \cdot \cdots \cdot s_1 s_2 \cdots s_p}^{n/p \text{ copies of } s_1 s_2 \cdots s_p}. \tag{4.22}$$

In other words, $\mathsf{aperiodic}(\boldsymbol{s})$ *is the shortest prefix of* $\boldsymbol{s}$ *that can be concatenated an integral number of times to produce* $\boldsymbol{s}$*. Moreover, given a list* $\mathcal{T}$*, let* $\mathsf{aperiodic}(\mathcal{T})$ *be the list containing the aperiodic prefix of each string in* $\mathcal{T}$ *and in the same relative order.*

A string is *periodic* if $\mathsf{aperiodic}(\boldsymbol{s}) \neq \boldsymbol{s}$ and is otherwise *aperiodic*. Within (4.20) the only periodic necklace is 101010 and $\mathsf{aperiodic}(101010) = 10$. Therefore, the modified concatenation using aperiodic prefixes is

$$111000 \cdot 110010 \cdot 10 \cdot 110100 \tag{4.23}$$

which is the shorthand universal cycle previously seen in (4.19). A consequence of the results in Section 4.2.3 is that same general construction always produces fixed-

density de Bruijn cycles. In this upcoming section, the modified $\boldsymbol{U}_+(\{0,0,0,1,1,1\})$ will be denoted by $\boldsymbol{U}(\{0,0,0,1,1,1\})$. Figure 4.2 shows the strings in $\mathbf{C}(4,5)\cup\mathbf{C}(5,4)$ and in $\mathbf{C}(5,5)$ that result from $\boldsymbol{U}(\{0,0,0,0,0,1,1,1,1,1\})$. More generally, the next section proves that $\boldsymbol{U}(\mathbb{M})$ produces a shorthand universal cycle for the permutations of any multiset $\mathbb{M}$.

This section concludes with an intermediate theorem involving $\boldsymbol{U}_+(\mathbb{M})$. The result formalizes the repeated shorthand rotations within $\boldsymbol{U}_+(\mathbb{M})$, as well as the number of times they are repeated. To illustrate the result, notice that the periodic strings in $\mathbf{C}(3,3)$ are 010101 and 101010, and the shorthand versions of these strings — 01010 and 10101 — are precisely the substrings that were repeated within (4.21). Furthermore, they each appeared $\frac{|\boldsymbol{s}|}{|\mathsf{aperiodic}(\boldsymbol{s})|} = \frac{6}{2} = 3$ times.

Theorem 4.2.4 (Cool-lex Shorthand Universal Cycles for Multiset Permutations with Periodic Permutation Repetitions)**.** *$\boldsymbol{U}_+(\mathbb{M})$ includes $\frac{|\boldsymbol{s}|}{|\mathsf{aperiodic}(\boldsymbol{s})|}$ copies of $\mathsf{short}(\boldsymbol{s})$ for each $\boldsymbol{s} \in \Pi(\mathbb{M})$. In other words, $\boldsymbol{U}_+(\mathbb{M})$ is a shorthand universal cycle for the permutations of $\mathbb{M}$ except that the shorthand of periodic permutations are repeated.*

Proof. Consider an individual permutation $\boldsymbol{s} \in \Pi(\mathbb{M})$. Let $g = |\mathsf{aperiodic}(\boldsymbol{s})|$ and $j = \frac{n}{g}$. Let $\boldsymbol{r}$ be the necklace representation (i.e., lexicographically largest rotation) of $\boldsymbol{s}$. Notice that $g = |\mathsf{aperiodic}(\boldsymbol{r})|$ since aperiodic prefix lengths are closed under rotations. Finally, let h be the minimum positive integer such that $\boldsymbol{s} = \circlearrowleft_h(\boldsymbol{r})$. Since $\boldsymbol{r}$ repeats every g symbols then

$$\boldsymbol{s} = \circlearrowleft_h(\boldsymbol{r}) = \circlearrowleft_{h+g}(\boldsymbol{r}) = \circlearrowleft_{h+2\cdot g}(\boldsymbol{r}) = \cdots = \circlearrowleft_{h+(j-1)\cdot g}(\boldsymbol{r}).$$

Thus, there are j rotations of $\boldsymbol{r}$ that equal $\boldsymbol{s}$. (Notice $h + (j-1)\cdot g \le n$ since $h \le g$.) Therefore, Lemmas 3.4.6-3.4.9 imply that $\mathsf{short}(\boldsymbol{s})$ appears as a substring of $\boldsymbol{U}_+(\mathbb{M})$ at least j times. □

4.2.3 Multiset Permutations

This section proves that reverse cool-lex order can be used to create shorthand universal cycles for the permutations of any multiset. In particular, the shorthand universal cycle is denoted $\boldsymbol{U}(\mathbb{M})$ and is defined below. The proof involves several small steps. The section finishes with brief discussions of efficient generation and shorthand universal cycles that avoid periodic multiset permutations.

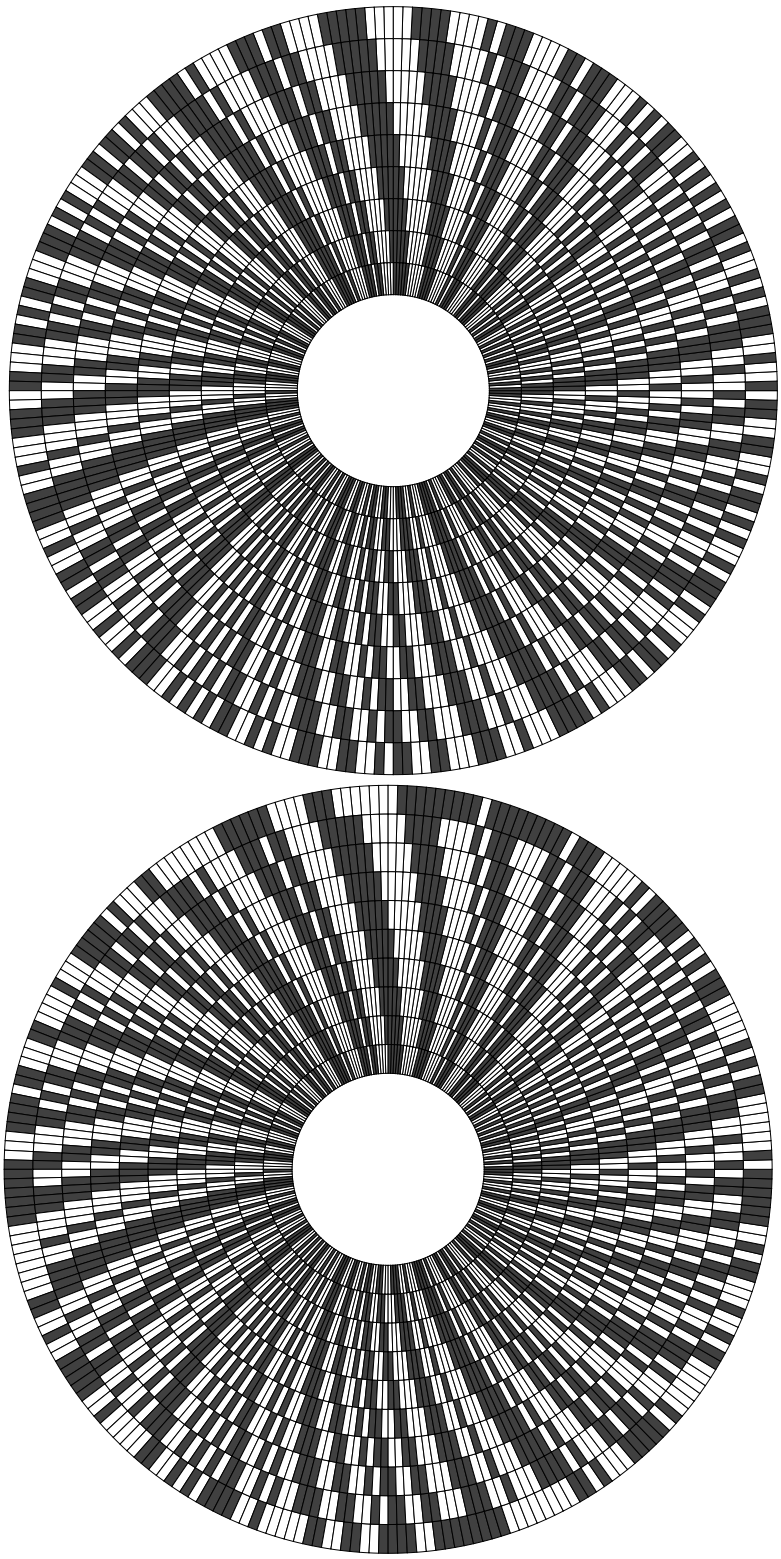


Figure 4.2: An artistic representation of the cool-lex shorthand universal cycle for $(5,5)$-combinations. Each ring is identical up to rotation (except for the outermost bottom ring) and can either be interpreted as a universal cycle for the middle levels or a fixed-density de Bruijn cycle. White and black regions represent 0 and 1 respectively. Individual strings are read along a line segment originating from the center, and the first and last strings are at either side of 12 o'clock. The top figure includes the strings comprising the middle levels: $(4,5)$- and $(5,4)$-combinations. The bottom figure suffixes the last (redundant) bits to create $(5,5)$-combinations.

Definition 4.2.5 (Cool-lex Shorthand Universal Cycle)**.** *Given a multiset of symbols* $\mathbb{M}$*, the* cool-lex universal cycle with periodic repetitions *is*

$$\boldsymbol{U}(\mathbb{M}) = \bigodot \mathsf{aperiodic}(\overrightarrow{\mathcal{C}}(\mathbf{N}(\mathbb{M}))).$$

In other words, $\boldsymbol{U}(\mathbb{M})$ *is the concatenation of the aperiodic prefixes of multiset necklaces in reverse cool-lex order.*

For example, $\boldsymbol{U}(\{1,1,2,2,3,3\})$ can be constructed with the aid of Table 3.1 as follows

$$\begin{aligned}\boldsymbol{U}(\{1,1,2,2,3,3\}) = 332211 \cdot 322131 \cdot 321231 \cdot 322311 \cdot 323211 \cdot 332112 \cdot 321312 \cdot 312 \cdot \\ 323112 \cdot 331122 \cdot 313122 \cdot 331212 \cdot 332121 \cdot 321 \cdot 323121 \cdot 331221.\end{aligned}$$

In order to prove that $\boldsymbol{U}(\mathbb{M})$ is a shorthand universal cycle for the permutations of $\mathbb{M}$ it is necessary to prove several small results. The first lemma involves the length of $\boldsymbol{U}(\mathbb{M})$.

Lemma 4.2.6 (Cool-lex Shorthand Universal Cycle Length)**.** *Given any multiset* $\mathbb{M}$*,*

$$|\boldsymbol{U}(\mathbb{M})| = |\Pi(\mathbb{M})|.$$

In other words, cool-lex shorthand universal cycles have the correct length.

Proof. Notice that the rotation set $\circlearrowleft(\boldsymbol{s})$ contributes $|\mathsf{aperiodic}(\boldsymbol{s})|$ to both sides of the stated equality for each necklace $\boldsymbol{s} \in \mathbf{N}(\mathbb{M})$. □

The second lemma examines the role of periodic necklaces in cool-lex order. To illustrate the results of the lemma, consider the strings before and after the periodic necklace 312312 in cool-lex order

$$\begin{aligned}\overrightarrow{\mathsf{cool}}(312312) &= 3\overrightarrow{12}312 & \overleftarrow{\mathsf{cool}}(312312) &= 3\overleftarrow{12}312 \\ &= 323112 & &= 321312.\end{aligned}$$

In particular, the strings before and after 312312 are aperiodic, and adjacent symbols (s_2 and s_3) in $\boldsymbol{s}$ are respectively right-shifted and left-shifted when creating $\overrightarrow{\mathsf{cool}}(312312)$ and $\overleftarrow{\mathsf{cool}}(312312)$.

Lemma 4.2.7 (Periodic Necklaces)**.** *Suppose* $\mathbf{L} = \mathbf{N}(\mathbb{M})$ *and* $\boldsymbol{s} \in \mathbf{L}$ *and* $|\mathbf{L}| \geq 2$ *and* $\mathsf{aperiodic}(\boldsymbol{s}) \neq \boldsymbol{s}$ *and let* $|\mathsf{pre}(\boldsymbol{s})| = k$*. Then, there exist values* $x > k$ *and* $y \leq k$ *such that*

$$\mathsf{aperiodic}(\overrightarrow{\mathsf{cool}}(\boldsymbol{s})) = \overrightarrow{\mathsf{cool}}(\boldsymbol{s}) \text{ and } \overrightarrow{\mathsf{cool}}(\boldsymbol{s}) = \overrightarrow{\mathsf{shift}}(\boldsymbol{s}, k, x)$$

and

$$\mathsf{aperiodic}(\overleftarrow{\mathsf{cool}}(\boldsymbol{s})) = \overleftarrow{\mathsf{cool}}(\boldsymbol{s}) \text{ and } \overleftarrow{\mathsf{cool}}(\boldsymbol{s}) = \overleftarrow{\mathsf{shift}}(\boldsymbol{s}, k+1, y)$$

In other words, two periodic necklaces cannot be consecutive in cool-lex order. Furthermore, adjacent symbols (s_k *and* $s_k + 1$*) are right-shifted and left-shifted, respectively, when applying* $\overrightarrow{\mathsf{cool}}$ *and* $\overleftarrow{\mathsf{cool}}$ *to a periodic necklace.*

Proof. Before starting the main part of the proof, it is useful to point out that $|\mathsf{aperiodic}(\boldsymbol{s})| \geq 2$ and $\mathsf{pre}(\boldsymbol{s})$ is a prefix of $\mathsf{aperiodic}(\boldsymbol{s})$. Therefore, $k \leq n-2$.

Let $\boldsymbol{t} = \overrightarrow{\mathsf{cool}}(\boldsymbol{s})$. It is claimed that $|\mathsf{fun}(\boldsymbol{s})| = k$. This is because $|\mathsf{fun}(\boldsymbol{s})| < k$ would imply $\overleftarrow{\mathsf{bubble}}(\boldsymbol{s}, k) \in \mathbf{L}$ by (2.2). However, this is not possible since the first k symbols of $\overleftarrow{\mathsf{bubble}}(\boldsymbol{s}, k)$ are lexicographically smaller than $\mathsf{pre}(\boldsymbol{s})$, and since $\overleftarrow{\mathsf{bubble}}(\boldsymbol{s}, k)$ must contain at least one copy of $\mathsf{aperiodic}(\boldsymbol{s})$ and hence at least one copy of $\mathsf{pre}(\boldsymbol{s})$. Therefore, by $\boldsymbol{t} \neq \boldsymbol{s}$ and Definition 3.4.3, there exists $x > k$ such that $\boldsymbol{t} = \overrightarrow{\mathsf{shift}}(\boldsymbol{s}, k, x)$. To complete the discussion of $\boldsymbol{t}$, it must be shown that $\mathsf{aperiodic}(\boldsymbol{t}) = \boldsymbol{t}$. This follows from the fact that the first k symbols of $\boldsymbol{t}$ are lexicographically larger than any other k consecutive symbols in $\boldsymbol{t}$.

Let $\boldsymbol{r} = \overleftarrow{\mathsf{cool}}(\boldsymbol{s})$. It is claimed that either $s_k < s_{k+2}$ or $\overleftarrow{\mathsf{bubble}}(\boldsymbol{s}, k+2) \notin \mathbf{L}$. This is because $s_k \geq s_{k+2}$ would imply that the first $k+1$ symbols of $\overleftarrow{\mathsf{bubble}}(\boldsymbol{s}, k+2)$ would be lexicographically smaller than another substring of length $k+1$ in $\overleftarrow{\mathsf{bubble}}(\boldsymbol{s}, k+2)$. (Precisely, if $\boldsymbol{s} = \mathsf{pre}(\boldsymbol{s})^2$ then the lexicographically larger substring of length $k+1$ in $\overleftarrow{\mathsf{bubble}}(\boldsymbol{s}, k+2)$ would begin at position $k+2$. Otherwise, $\overleftarrow{\mathsf{bubble}}(\boldsymbol{s}, k+2)$ would necessarily contain a substring equal to $\mathsf{aperiodic}(\boldsymbol{s})$, and the substring of length $k+1$ beginning at this substring would be lexicographically larger.) Therefore, by $\boldsymbol{r} \neq \boldsymbol{s}$ and Definition 3.1.3, there exists $y \leq k$ such that $\boldsymbol{r} = \overleftarrow{\mathsf{shift}}(\boldsymbol{s}, k+1, y)$. To complete the discussion of $\boldsymbol{r}$, it must be shown that $\mathsf{aperiodic}(\boldsymbol{r}) = \boldsymbol{r}$. This follows from the fact that the first k symbols of $\boldsymbol{r}$ are lexicographically larger than any other k consecutive symbols in $\boldsymbol{r}$. □

To understand the consequences of Lemma 4.2.7, consider the cool-lex shorthand universal cycle with periodic permutation repetitions for $\{1,1,2,2,3,3\}$ given below. Since the periodic necklaces over $\{1,1,2,2,3,3\}$ are 312312 and 321321, then the periodic permutations over $\{1,1,2,2,3,3\}$ are contained in $\circlearrowleft(312312) \cup \circlearrowleft(321321)$,

and Theorem 4.2.4 implies that the shorthand of each of these permutations will be repeated twice within $\boldsymbol{U}_+(\{1,1,2,2,3,3\})$. One consequence of Lemma 4.2.7 is that all of these repeated substrings will appear in close proximity to their corresponding periodic necklace

$$\boldsymbol{U}_+(\{1,1,2,2,3,3\}) = 332211\cdot322131\cdot321231\cdot322311\cdot323211\cdot332112\cdot321\underline{312}\cdot\underline{312312}\cdot \\ \underline{3}23112\cdot331122\cdot313122\cdot331212\cdot332\underline{12}1\cdot\underline{321321}\cdot\underline{32}3121\cdot331221.$$

The two periodic necklaces — 312312 and 321321 — are underlined together with the suffixes and prefixes they share their respective predecessor and successor. For the sake of illustration, let us focus on 312312, and let $n = 6$ and $k = |\curlywedge(312312)| = 2$. Lemma 4.2.7 implies that 312312 shares a suffix of length $n-(k+1) = 3$ with 321312, and a prefix of length $k-1 = 1$ with 323112. Because of these facts, the underlined concatenation

$$\underline{312}\cdot\underline{312312}\cdot\underline{3}$$

has length $2n-2 = 10$, and therefore contains a total of $n = 6$ non-circular substrings of length $n-1 = 5$. Furthermore, all of these substrings will be shorthand rotations of 312312. (In other words, 312312 represents one of the easy cases of Theorem 3.4.10, in which all of the shorthand rotations of $\boldsymbol{s}$ are contained within the $\overleftarrow{\mathsf{cool}}(\boldsymbol{s})\cdot\boldsymbol{s}\cdot\overrightarrow{\mathsf{cool}}(\boldsymbol{s})$.) Moreover, if 312312 is replaced by its aperiodic prefix 312 then the concatenation

$$\underline{312}\cdot\underline{312}\cdot\underline{3}$$

has length $n-2+|\mathsf{aperiodic}(312312)| = 7$, and so a total of $|\mathsf{aperiodic}(312312)| = 3$ (non-redundant) shorthand rotations remain. Moreover, the substrings using a prefix or suffix of this modified concatenation are unchanged. The same trick can be applied independently to the remaining periodic necklace 321321 since Lemma 4.2.7 implies that the two are not consecutive in cool-lex order. Now the intuition behind the main theorem of this section is complete. The formal proof is slightly generalized in order to accommodate the discussion following the theorem.

Theorem 4.2.8 (Cool-lex Shorthand Universal Cycles for Multiset Permutations)**.** *If $\mathbb{M}$ is a multiset, then $\boldsymbol{U}(\mathbb{M})$ is a shorthand universal cycle for the permutations of $\mathbb{M}$. In other words, concatenating the aperiodic prefix of each necklace in reverse cool-lex order creates a shorthand universal cycle for the permutations of the multiset.*

Proof. By Lemma 4.2.6, $|\boldsymbol{U}(\mathbb{M})| = |\Pi(\mathbb{M})|$. Therefore, the theorem can be proven by

showing that $\mathsf{short}(\boldsymbol{s})$ is a substring of $\boldsymbol{U}(\mathbb{M})$ for each $\boldsymbol{s} \in \Pi(\mathbb{M})$. From Theorem 4.2.4, $\boldsymbol{s}$ is a substring of $\boldsymbol{U}_+(\mathbb{M})$. Therefore, $\mathsf{short}(\boldsymbol{s})$ is a substring of consecutive necklaces $\boldsymbol{r} \cdot \boldsymbol{t}$ with $\boldsymbol{r}, \boldsymbol{t} \in \mathbf{L} = \mathbf{N}(\mathbb{M})$ and $\boldsymbol{t} = \overrightarrow{\mathsf{cool}}(\boldsymbol{r})$. In particular, suppose that

$$\mathsf{short}(\boldsymbol{s}) = r_x r_{x+1} \cdots r_n t_1 t_2 \cdots t_{x-2}$$

for x satisfying $1 \leq x \leq n$. (When $x = 1$, $\mathsf{short}(\boldsymbol{s}) = r_1 r_2 \cdots r_{n-1}$.) Thus,

$$s_1 s_2 \cdots s_{n-x+1} = r_x r_{x+1} \cdots r_n \tag{4.24}$$

and

$$s_{n-x+2} s_{n-x+3} \cdots s_{n-1} = t_1 t_2 \cdots t_{x-2}. \tag{4.25}$$

The proof is divided into cases depending on which of $\boldsymbol{r}$, $\boldsymbol{s}$, and $\boldsymbol{t}$ are periodic. By Lemma 4.2.7 it cannot be that both $\boldsymbol{r}$ and $\boldsymbol{t}$ are periodic.

In the first case suppose that $\boldsymbol{r}$ and $\boldsymbol{s}$ and $\boldsymbol{t}$ are aperiodic. Therefore, $\boldsymbol{r} \cdot \boldsymbol{t}$ is a substring of $\boldsymbol{U}(\mathbb{M})$. Therefore, $\mathsf{short}(\boldsymbol{s})$ is a substring $\boldsymbol{U}(\mathbb{M})$ by (4.24) and (4.25).

In the second case suppose that $\boldsymbol{r}$ and $\boldsymbol{s}$ are aperiodic, and $\boldsymbol{t}$ is periodic. Let $k = |\llcorner(\boldsymbol{t})|$. From Lemma 4.2.7, $r_{k+2} r_{k+3} \cdots r_n = t_{k+2} t_{k+3} \cdots t_n$. Therefore, $x \leq k + 1$. (Otherwise, $\mathsf{short}(\boldsymbol{s}) = \mathsf{short}(\circlearrowleft_x (\boldsymbol{t}))$, which contradicts that $\boldsymbol{s}$ is aperiodic.) Notice that $\mathsf{short}(\boldsymbol{s})$ is a non-circular substring of

$$\boldsymbol{r} \cdot \mathsf{aperiodic}(\boldsymbol{t})^h \cdot \overrightarrow{\mathsf{cool}}(\boldsymbol{t})$$

beginning at position x for all values of $h \geq 0$. This is due to (4.24), as well as (4.25) and because the first $x - 2 \leq k - 1$ symbols of $\mathsf{aperiodic}(\boldsymbol{t})^h \cdot \overrightarrow{\mathsf{cool}}(\boldsymbol{t})$ are $t_1 t_2 \cdots t_{k-1}$ by Lemma 4.2.7. This case is completed by noting that $r \cdot \mathsf{aperiodic}(\boldsymbol{t})^h \cdot \overrightarrow{\mathsf{cool}}(\boldsymbol{t})$ is a substring of $\boldsymbol{U}(\mathbb{M})$ for $h = 1$.

In the third case suppose that $\boldsymbol{r}$ is periodic, and $\boldsymbol{s}$ and $\boldsymbol{t}$ are aperiodic. Let $k = |\llcorner(\boldsymbol{r})|$. From Lemma 4.2.7, $t_1 t_2 \cdots t_{k-1} = r_1 r_2 \cdots r_{k-1}$. Therefore, $x \geq k + 2$. (Otherwise, $\mathsf{short}(\boldsymbol{s}) = \mathsf{short}(\circlearrowleft_x (\boldsymbol{r}))$, which contradicts that $\boldsymbol{s}$ is aperiodic.) Notice that $\mathsf{short}(\boldsymbol{s})$ is a non-circular substring of

$$\overleftarrow{\mathsf{cool}}(\boldsymbol{r}) \cdot \mathsf{aperiodic}(\boldsymbol{r})^h \cdot \boldsymbol{t}$$

beginning at position $x + |\mathsf{aperiodic}(\boldsymbol{r})^h|$ for all values of $h \geq 0$. This is due to (4.24) and because the last $n - x + 1 \leq n - (k + 1)$ symbols of $\overleftarrow{\mathsf{cool}}(\boldsymbol{r}) \cdot \mathsf{aperiodic}(\boldsymbol{r})^h$ are $r_x r_{x+1} \cdots r_n$ by Lemma 4.2.7, as well as (4.25). This case is completed by noting that

$\overleftarrow{\mathsf{cool}}(\boldsymbol{r}) \cdot \mathsf{aperiodic}(\boldsymbol{r})^h \cdot \boldsymbol{t}$ is a substring of $\boldsymbol{U}(\mathbb{M})$ for $h = 1$.

In the fourth case suppose that $\boldsymbol{s}$ is periodic. Let $\boldsymbol{u}$ be the necklace representation (i.e., lexicographically largest rotation) of $\boldsymbol{s}$. Notice that $\boldsymbol{u}$ must also be periodic since $\boldsymbol{u} \in \circlearrowleft(\boldsymbol{s})$ and all rotations of $\boldsymbol{s}$ are periodic. Moreover, $|\mathsf{aperiodic}(\boldsymbol{s})| = |\mathsf{aperiodic}(\boldsymbol{u})|$. Let $k = |\curlywedge(\boldsymbol{u})|$. By Lemma 4.2.7, $\overleftarrow{\mathsf{cool}}(\boldsymbol{u})$ and $\boldsymbol{u}$ share a suffix of length $n - (k+1)$, and $\overrightarrow{\mathsf{cool}}(\boldsymbol{u})$ and $\boldsymbol{u}$ share a prefix of length $k - 1$. Let $\boldsymbol{p}$ and $\boldsymbol{z}$ respectively denote these suffixes and prefixes. Notice that every shorthand rotation of $\boldsymbol{u}$ appears as a non-circular substring of

$$\boldsymbol{z} \cdot \mathsf{aperiodic}(\boldsymbol{u})^h \cdot \boldsymbol{p}$$

exactly h times for all values of $h \geq 0$. This is because the concatenation has length $n - 2 + h \cdot |\mathsf{aperiodic}(\boldsymbol{u})|$, and because $\boldsymbol{z}$ and $\boldsymbol{p}$ are respectively suffixes and prefixes of repeated concatenations of $\mathsf{aperiodic}(\boldsymbol{u})$. This case is completed by noting that $\boldsymbol{z} \cdot \mathsf{aperiodic}(\boldsymbol{u})^h \cdot \boldsymbol{p}$ is a substring of $\boldsymbol{U}(\mathbb{M})$ for $h = 1$, and that $\mathsf{short}(\boldsymbol{s})$ is a shorthand rotation of $\boldsymbol{u}$. □

Within the proof of Theorem 4.2.8, the value of $h = 1$ denotes the number of times that $\mathsf{aperiodic}(\boldsymbol{w})$ is repeated within $\boldsymbol{U}(\mathbb{M})$ whenever $\boldsymbol{w}$ is a periodic necklace . Changing the value of h for $\mathsf{aperiodic}(\boldsymbol{w})$ simply changes the number of times that each shorthand rotation of $\boldsymbol{w}$ appears within the universal cycle. (In particular, $\mathsf{short}(\boldsymbol{s})$ appears once as a substring regardless of the value of h in the first three cases, while $\mathsf{short}(\boldsymbol{s})$ appears h times as a substring in the fourth case.) Thus, if $h = 1$ is replaced by $h = 0$ then the shorthand rotations of $\boldsymbol{w}$ will be removed from the universal cycle. Since periodic permutations are rotations of periodic necklaces, and since aperiodic necklaces are Lyndon words, then this discussion leads to the following definition.

Definition 4.2.9 (Cool-lex Shorthand Universal Cycle avoiding Periodic Permutations)**.** *Given a multiset of symbols* $\mathbb{M}$*, the* cool-lex universal cycle avoiding periodic permutations *is*

$$\boldsymbol{U}_{-}(\mathbb{M}) = \bigodot \overrightarrow{\mathcal{C}}(\mathbf{N}^{-}(\mathbb{M})).$$

In other words, $\boldsymbol{U}_{-}(\mathbb{M})$ *is the concatenation of Lyndon words in reverse cool-lex order.*

For example,

$$\boldsymbol{U}_{-}(\{1,1,2,2,3,3\}) = 332211 \cdot 322131 \cdot 321231 \cdot 322311 \cdot 323211 \cdot 332112 \cdot 321312 \cdot 323112 \cdot 331122 \cdot 313122 \cdot 331212 \cdot 332121 \cdot 323121 \cdot 331221.$$

The reader may notice that this shorthand universal cycle *avoids* the substrings 31231, 23123, 12312, 32132, 21321, and 13213, which are shorthand for the periodic permutations over $\{1, 1, 2, 2, 3, 3\}$. The idea of considering universal cycles that avoid periodic strings was suggested by Shallit [82]. Chapter 5 discusses $\boldsymbol{U}_{\_}(\mathbb{M})$, as well as an analogous modification for the the FKM algorithm on n-tuples.

Chapter 5

Conclusions and Final Thoughts

"Every solution breeds new problems."
- Arthur Bloch

"A conclusion is the place where you got tired of thinking."
- Arthur Bloch

This thesis was motivated by the following question:

Do shift Gray codes exist for a variety of combinatorial objects?

At the conclusion of this thesis, the answer is clearly 'yes'. In addition, there are shift Gray codes that can be generated by simple and efficient algorithms that operate on data structures including arrays, computer words, and linked lists. Furthermore, shift Gray codes exist for multiset permutations when the allowable set of shifts is heavily restricted. Applications for these restrictive shift Gray codes include efficient exhaustive solutions to stacker-crane problems, and the explicit construction of shorthand universal cycles for multiset permutations. All of these results are based on the introduction of a new concept known as a bubble language, and a new variation of lexicographic order known as cool-lex order.

This chapter briefly summarizes the main results found within this thesis. Furthermore, this chapter includes a number of interesting observations that were not necessary for achieving the main results. Finally, a handful of open problems are provided for the interested reader. These topics are discussed in turn for bubble languages in Section 5.1, cool-lex order in Section 5.2, algorithms in Section 5.3, and shorthand universal cycles in Section 5.4.

5.1 Bubble Languages

Chapter 2 introduced the concept of a bubble language. Section 2.2 defines a bubble language as a set of fixed-content strings $\mathbf{L}$ that is closed under adjacent-transpositions between the frozen prefix $\bullet(\boldsymbol{s})$ and the non-decreasing prefix $\urcorner\llcorner(\boldsymbol{s})$ of each string $\boldsymbol{s} \in \mathbf{L}$. A number of previously studied combinatorial objects are shown to have natural representations as bubble languages in Section 2.3 (see Table 2.1 on page 38 for a summary). Properties involving bubble languages are investigated in Section 2.4. In particular, Section 2.4.1 shows that bubble languages are closed under union, intersection, and quotients. Furthermore, the equivalence of maximal and maximum left-shifts seen Section 2.4.2 is important for understanding the results of Chapters 3 and 4. Finally, Section 2.4.4 provides a structural characterization of bubble languages involving suffixes of strings known as scuts.

5.1.1 Additional Results for Bubble Languages

This section discusses additional methods for creating new bubble languages, and the interesting question of determining the total number of fixed-density bubble languages. One previously undiscussed technique for creating bubble languages is to base the inclusion of strings on the value of a well-chosen function. To illustrate this idea, say that an *inversion* in string $\boldsymbol{s} = s_1 s_2 \cdots s_n$ is any pair of indices (i, j) satisfying $1 \le i < j \le n$ and $s_i < s_j$. (It is more customary to define an inversion with $s_i > s_j$; this concession is analogous to those discussed on page 39.) Table 5.1 provides the number of inversions for each permutation of $\{1, 1, 2, 2, 3, 3\}$. The problem of efficiently generating permutations with a fixed number of inversions was solved by Effler-Ruskey [17].

The language of *multiset permutations with at most i inversions* over $\mathbb{M}$ is

$$\mathbf{V}(\mathbb{M}, i) = \{\boldsymbol{s} \in \Pi(\mathbb{M}) \mid \boldsymbol{s} \text{ has at most } i \text{ inversions}\}.$$

Although a formal proof is left as an exercise for the reader, it is relatively easy to see that $\mathbf{V}(\mathbb{M}, i)$ is a bubble language since adjacent-transpositions can change the number of inversions by at most ± 1. Furthermore, the fact that $\mathbf{V}(\mathbb{M}, i)$ is a bubble language has additional consequences due to the closure properties of bubble languages. In particular, if $\mathbf{L}$ is a bubble language with content $\mathbb{M}$ then $\mathbf{L} \cap \mathbf{V}(\mathbb{M}, i)$ is a bubble language containing the strings in $\mathbf{L}$ with at most i inversions.

The previous discussion showed that bubble language can be refined based on

$i=0$	$i=1$	$i=2$	$i=3$	$i=4$	$i=5$	$i=6$	$i=7$	$i=8$	$i=9$	$i=10$	$i=11$	$i=12$
332211	323211	233211	232311	133221	132321	123321	123231	113322	113232	112332	112323	112233
	332121	322311	233121	223311	133212	132231	123312	122331	121332	113223	121233	
		323121	313221	231321	213321	132312	131322	123132	122313	121323		
		331221	321321	232131	223131	133122	132132	123213	123123	122133		
		332112	322131	233112	231231	213231	132213	131232	131223	211233		
			323112	312321	231312	213312	212331	132123	211323			
			331212	313212	232113	221331	213132	211332	212133			
				321231	312231	223113	213213	212313				
				321312	312312	231132	221313	213123				
				322113	313122	231213	231123	221133				
				331122	321132	311322	311232	311223				
					321213	312132	312123					
						312213						
						321123						

Table 5.1: Multiset permutations of $\{1,1,2,2,3,3\}$ with i inversions. Bubble languages are obtained by providing an upper-bound on the number of inversions.

the number of inversions. To further illustrate this type of approach, say that a prefix is *balanced* if it contains the same number of 1s and 0s. Table 5.2 provides the number of balanced prefixes for the balanced parentheses of length ten. In general, it is not possible to refine a fixed-density bubble language using an upper-bound on the number of balanced prefixes since this value can increase when replacing the leftmost 01 with 10. More specifically, (2.8) leads to an increase in the number of balanced prefixes precisely when a prefix of the form 1^i0^i01 with $i > 0$ is replaced by 1^i0^i10. However, prefixes of the form 1^i0^i01 with $i > 0$ are not present in balanced parentheses. Therefore, bubble languages include *balanced parentheses of length* $2i$ *with at most* b *balanced prefixes*

$$\mathbf{P}_b(i) = \{\boldsymbol{s} \in \mathbf{P}(i) \mid \boldsymbol{s} \text{ has at most } b \text{ balanced parentheses}\}.$$

The cardinality of these languages is described by A009766 in Sloane's Online Encyclopedia of Integer Sequences [84].

Another way to expand the collection of known bubble languages is to introduce additional closure properties. Towards this goal, the structural characterization of bubble languages in Theorem 2.4.21 can be used. This theorem immediately implies that bubble languages are closed under removing every string with certain scuts. In particular, every string with suffix $\mathsf{scut}(j,i)$ can be removed whenever it is the shortest but not only scut with a given first symbol d_j ($i = \clubsuit(j)$ and $i > 1$) or when it is the only scut with the largest possible first symbol ($j = \clubsuit$ and $i = \clubsuit(j) = 1$).

$b = 1$	$b = 2$	$b = 3$	$b = 4$	$b = 5$
1101010100	1011010100	1010110100	1010101100	1010101010
1101011000	1011011000	1010111000	1010110010	
1101100100	1011100100	1011001100	1011001010	
1101101000	1011101000	1011010010	1100101010	
1101110000	1011110000	1011100010		
1110010100	1100110100	1100101100		
1110011000	1100111000	1100110010		
1110100100	1101001100	1101001010		
1110101000	1101010010	1110001010		
1110110000	1101100010			
1111000100	1110001100			
1111001000	1110010010			
1111010000	1110100010			
1111100000	1111000010			

Table 5.2: Balanced parentheses of length ten with b balanced prefixes. Bubble languages are obtained by providing an upper-bound on the number of balanced prefixes.

Another closure operation to consider is the left-quotient operation. The *left-quotient* operation is analogous to the quotient operation of Section 2.1.2 except that a prefix is used instead of a suffix. Formally,

$$\mathbf{L}\backslash\boldsymbol{p} = \{\boldsymbol{z} \mid \boldsymbol{p} \cdot \boldsymbol{z} \in \mathbf{L}\}.$$

In general, bubble languages are not closed under left-quotients. For example, the following left-quotient of fixed-density necklaces provides a simple counter-example

$$\begin{aligned}\mathbf{N}(\{1,1,1,0,0,0\})\backslash 10 &= \{111000, 110100, 110010, 101010\}\backslash 10 \\ &= \{1010\}.\end{aligned}$$

On the other hand, if $\mathbf{L}$ is a non-empty bubble language then $\mathbf{L}\backslash d_m$ is also a bubble language. In other words, bubble languages are closed under left-quotients using a single copy of the largest symbol. A formal proof can be obtained by using Definition 2.2.1 and is left as an exercise for the reader. The importance of this small result is enhanced by additional results discussed in Section 5.2.1 and 5.4.1.

Given the various methods of constructing bubble languages, one may wonder if it is possible to count the total number of bubble languages over the multiset $\mathbb{M}$. In the case of fixed-density bubble languages, there is an elegant solution due to Ruskey

[65]. Recall that (2.8) in Theorem 2.2.4 states that if a string is in a fixed-density bubble language $\mathbf{L}$, then replacing its leftmost 01 with 10 results in another string in $\mathbf{L}$. This implication can be modeled using a partially ordered set, where $\boldsymbol{s} \prec \boldsymbol{t}$ if $\boldsymbol{s}$ is the result of replacing the leftmost 01 in $\boldsymbol{t}$ by 10. Figure 5.1 provides the Hasse diagram for this *fixed-density bubble language poset* for $\mathbf{C}(3,3)$ (the binary strings with three 0s and three 1s) using the standard convention that $\boldsymbol{s}$ appears below $\boldsymbol{t}$ when $\boldsymbol{s} \prec \boldsymbol{t}$.

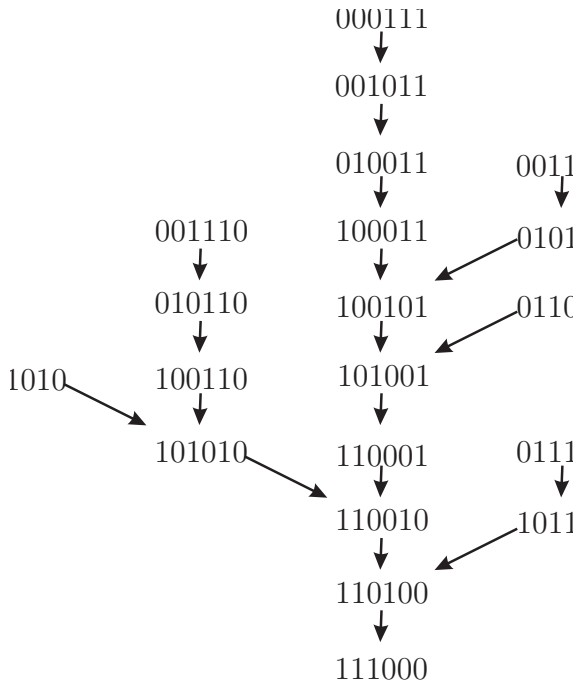


Figure 5.1: The fixed-density bubble language poset for $\mathbf{C}(3,3)$.

Using this framework suggested by Sawada [76], a fixed-density bubble language is simply an ideal of its bubble language poset. An *ideal* of a poset $\mathcal{P}$ is a subset of $S(\mathcal{P})$ with the property that a is in the subset whenever b is in the subset and $a \prec b$. In particular, the non-empty ideals in Figure 5.1 correspond to the non-empty bubble languages containing three copies of 0 and three copies of 1. In general, the number of non-empty ideals for the bubble poset over $\mathbf{C}(j,k)$ is counted by the following recurrence

$$R(j,k) = R(j-1,k) + R(0,k-1) \cdot R(1,k-1) \cdot \cdots \cdot R(j,k-1).$$

In particular, $R(0,0) = 0$ and otherwise $R(j,0) = 1$ and $R(0,k) = 1$ since there is respectively one non-empty bubble languages of the form $\{0^j\}$ and $\{1^k\}$. Table 5.3

gives the number of non-empty fixed-density bubble languages for small values of j and k.

R(j,k)	0	1	2	3	4	5
0	0	1	1	1	1	1
1	1	2	3	4	5	6
2	1	3	9	33	153	873
3	1	4	31	922	137245	119147224
4	1	5	129	114457	15691060817	18695327164179654 57
5	1	6	651	73825416	11538887748955842 1	$> 2.1656456 \times 10^{36}$

Table 5.3: The number of fixed-density bubble languages over $\mathbf{C}(j,k)$ for $0 \leq j,k \leq 5$.

Since a bubble language poset is a tree, its ideals can be generated in Gray code order by a loopless algorithm. In other words, the bubble languages with a fixed number of 1s and a fixed number of 0s can be generated in worst-case O(1)-time, where successive bubble languages differ by the addition or deletion of a single string. This result is due to Koda-Ruskey [49] and applies to any poset whose Hasse diagram is a forest.

5.1.2 Open Problems for Bubble Languages

Theorem 2.3.13 proves that $\mathbf{T}(\mathbb{M})$ (ordered trees with fixed branching sequence) is a bubble language. However, it appears that the result could be generalized. The interested reader may wish to find necessary and sufficient conditions for $\mathbb{M}$ that make the following language a bubble language

$$\{\boldsymbol{s} \in \Pi(\mathbb{M}) \mid \text{ if } \boldsymbol{s} = \boldsymbol{p} \cdot \boldsymbol{z} \text{ then } \Sigma(\boldsymbol{p}) \geq |\boldsymbol{p}|\}.$$

It may also be useful to explore alternate representations for combinatorial objects that do not otherwise yield bubble bubble languages. Specific examples worth investigating include bracelets, and linear-extensions of additional posets.

Besides finding additional examples of bubble languages, it may also be useful to consider additional Gray codes for bubble languages. Since the defining closure properties of bubble languages involve adjacent-transpositions, it would not be unreasonable for bubble languages to have natural Gray codes involving transpositions. It may also be fruitful to consider modifying (2.1) and (2.2) to encapsulate a different set of fixed-content languages. Furthermore, generalizing the concept of a bubble language to include languages that do not have fixed-content should also be investigated. This possibility is further discussed in Section 5.2.2.

During the writing of this thesis, the concept of a bubble language arose from the desire to classify the fixed-content languages that appear in shift Gray code order when expressed in cool-lex order. This methodology could certainly be applied to other well-known Gray code orders. For example, is there a general class of languages whose successive strings differ by one (or two) bit changes when expressed in the binary reflected Gray code seen in Section 1.1.2? Similarly, is there a general class of languages whose successive strings differ by (adjacent-)transpositions in the Johnson-Trotter-Steinhaus order seen in Section 1.1.4? More generally, one could ask when order x has a Gray code using operation y.

In retrospect, (2.1) and (2.2) in Definition 2.2.1 are natural conditions for creating Gray codes. Although this iterative definition of bubble languages was discovered after the recursive characterization presented in Theorem 2.4.21, this investigation could have proceeded in reverse. In other words, interesting new languages could be obtained by starting with a small set of closure properties that seem related to the creation of Gray codes. This approach differs from the prevailing methodology of starting with a previously studied language and attempting to create a Gray code for it.

Additional open problems include counting the number of fixed-content bubble languages, and the selection of a random string in a bubble language.

5.2 Cool-lex Order

Cool-lex order was introduced in Chapter 3 and is a new variation of lexicographic order. Section 3.2 describes how this variation is based on an alternate view of co-lexicographic order that partitions strings by their scut as opposed to their rightmost symbol. When a bubble language is expressed in cool-lex order, then its strings appear in a left-shift Gray code as proven in Section 3.3. Furthermore, there is a simple iterative rule that generates these cool-lex orders one string at a time. The shift is known as the cool left-shift, and is denoted by $\overleftarrow{\mathsf{cool}}$ starting in Section 3.1. In particular, this iterative rule uses the greedy left-shift $\overleftarrow{\mathsf{greedy}}$, which is a language-dependent operation introduced in Section 3.1.1. In the case of multiset permutations, the result is the first prefix-shift Gray code. In the case of fixed-content necklaces and Lyndon words, the result is the first minimal-change order. Besides these Gray code results, Section 2.4 points out that cool-lex orders of bubble languages have interesting properties. In particular, the rotation properties given in Section 3.4.2 are central to the shorthand universal cycle results given in Chapter 4.

5.2.1 Additional Results for Cool-lex Order

A basic property of cool-lex order is that subsets of strings appear as sublists. That is,

$$\mathbf{L}' \subseteq \mathbf{L} \text{ implies } \overleftarrow{\mathcal{C}}(\mathbf{L}') \sqsubset \overleftarrow{\mathcal{C}}(\mathbf{L})$$

where $\overleftarrow{\mathcal{C}}(\mathbf{L}') \sqsubset \overleftarrow{\mathcal{C}}(\mathbf{L})$ denotes that $\overleftarrow{\mathcal{C}}(\mathbf{L}')$ is a *sublist* of $\overleftarrow{\mathcal{C}}(\mathbf{L})$, meaning that any two strings in $\overleftarrow{\mathcal{C}}(\mathbf{L}')$ appears in the same relative order that they appear in $\overleftarrow{\mathcal{C}}(\mathbf{L})$. This basic property follows from the definition of cool-lex order, and allows any fixed-content language to be listed in cool-lex order by using the appropriate sublist of $\overleftarrow{\mathcal{C}}(\Pi(\mathbb{M}))$. Interestingly, this property was not (explicitly) required anywhere in the thesis. Another area for additional results involves list-quotients. A *list-quotient* is a list operation that is analogous to the quotient operation on languages found in Section 2.1.2, except that the resulting strings are listed in the same relative order they appeared in the original list. Again, / and \ are respectively used for list-quotients involving suffixes and prefixes. For example, Table 5.4 shows the four single symbol quotients for a cool-lex listing of combinations. Within the table, notice that both of the quotients involving suffixes produce the cool-lex order of the resulting strings. In general, the identity

$$\overleftarrow{\mathcal{C}}(\mathbf{L}/d_i) = \overleftarrow{\mathcal{C}}(\mathbf{L})/d_i$$

follows immediately from Definition 3.2.2 for all $1 \leq i \leq m$. More interesting are the quotients involving prefixes. Notice that the list-quotient involving a single copy of the largest symbol in Table 5.4 also produces the cool-lex list of the remaining strings. In general, the identity

$$\overleftarrow{\mathcal{C}}(\mathbf{L}\backslash d_m) = \overleftarrow{\mathcal{C}}(\mathbf{L})\backslash d_m \tag{5.1}$$

appears to hold. Finally, the list-quotient of a prefix containing a single copy of the smallest symbol in Table 5.4 produces co-lexicographic order of the resulting strings. Although this result does not hold for general fixed-content bubble languages, the interested reader can explore whether it holds for fixed-density bubble languages.

While the results of this thesis are focused on fixed-content languages, it may also be useful to examine how $\overleftarrow{\mathsf{cool}}$ and $\overrightarrow{\mathsf{cool}}$ alter the underlying combinatorial object itself. For example, in [72] it is shown that cool-lex order can also be used to create a loopless algorithm for generating binary trees in their standard representation using left-child and right-child pointers. Figure 5.2 contains an example of the binary trees with three internal nodes in cool-lex order. More generally, it would be interesting to study how $\overleftarrow{\mathsf{cool}}$ and $\overrightarrow{\mathsf{cool}}$ alter k-ary trees and ordered trees with fixed branching

$\overleftarrow{\mathcal{C}}(\mathbf{C}(3,3))\backslash 1$	$\overleftarrow{\mathcal{C}}(\mathbf{C}(3,3))\backslash 0$	$\overleftarrow{\mathcal{C}}(\mathbf{C}(3,3))$	$\overleftarrow{\mathcal{C}}(\mathbf{C}(3,3))/0$	$\overleftarrow{\mathcal{C}}(\mathbf{C}(3,3))/1$
	11100	011100	01110	
01100		101100	10110	
10100		110100	11010	
	11010	011010	01101	
01010		101010	10101	
	10110	010110	01011	
	01110	001110	00111	
00110		100110	10011	
10010		110010	11001	
	11001	011001		01100
01001		101001		10100
	10101	010101		01010
	01101	001101		00110
00101		100101		10010
	10011	010011		01001
	01011	001011		00101
	00111	000111		00011
00011		100011		10001
10001		110001		11000
11000		111000	11100	
cool-lex	co-lex		cool-lex	cool-lex

Table 5.4: List-quotients involving prefixes (leftmost two columns) and suffixes (rightmost two columns) for the cool-lex order of combinations with three 0s and three 1s.

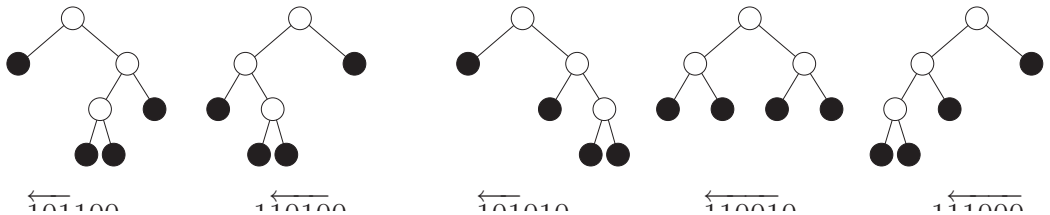


Figure 5.2: Binary trees differ by O(1) pointer manipulations in cool-lex order [72].

sequence.

In terms of strings, it is also possible that there are natural generalizations of cool-lex order for languages that do not have fixed-content. For example, it is possible to generate all binary strings of length n by slightly modifying the iterative right-shift rule for generating combinations first seen on page 2. In particular, the following rule simply modifies the resulting string when the first bit does not pass over 01.

Generating binary strings of length n	(iii)
Shift the first bit to the right until it passes over 01, and if it passes over the last bit without passing over 01 then change its value.	

For example, the above rule changes 0111000 into 1110001. This is because the first bit, 0, is shifted to the right but never passes over 01. Therefore, it's value is changed to 1 when it passes over the last bit. This transformation is denoted by $\overrightarrow{\underline{0}111000} = 1110000$, where the underline represents that the shifted bit's value is changed. This rule is illustrated below for the binary strings of length 4

$$\overrightarrow{\underline{1}111}, \overrightarrow{\underline{1}110}, \overrightarrow{\underline{1}100}, \overrightarrow{\underline{1}000}, \overrightarrow{\underline{0}000}, \overrightarrow{\underline{0}001}, \overrightarrow{0010}, \overrightarrow{\underline{0}100},$$
$$\overrightarrow{1001}, \overrightarrow{0011}, \overrightarrow{0101}, \overrightarrow{1010}, \overrightarrow{\underline{0}110}, \overrightarrow{1101}, \overrightarrow{1011}, \overrightarrow{\underline{0}111}. \quad (5.2)$$

To see why the rule generates all binary strings of length n, notice that the binary strings of the form $1^{n-j}0^j$ will be generated by the rule in sequence for $j = 0, 1, 2, \ldots, n$. The rule then generates the reverse cool-lex order of $\mathbf{C}(n-j, j)\backslash\{1^{n-j}0^j\}$ for $j = 1, 2, \ldots, n-1$. This is due to the fact the rule does not change the value of any bit until 01^j0^{n-j-1} (the last string in $\overrightarrow{\mathcal{C}}(\mathbf{C}(n-j, j))$), and since the rule transforms this string into $1^j0^{n-j-1}1$ (the second string in $\overrightarrow{\mathcal{C}}(\mathbf{C}(n-j-1, j+1))$). More generally, the rule can be generalized to generate the binary strings of length n with any continuous range of densities. These results were noticed during a collaboration with Stevens [88].

Given these rules for generating binary strings, one may ask if there is a similar rule for generating n-tuples over an m-letter alphabet. More specifically, one may ask if

such a rule can be derived from the cool-lex rule for generating multiset permutations by modifying the resulting string when the first symbol does not pass over consecutive $d_i d_j$ with $i < j$. A potential answer to this question appears in Section 5.3, although this potential answer also raises the question of whether it provides the "correct" generalization of cool-lex order.

5.2.2 Open Problems for Cool-lex Order

Besides formally proving (5.1), the interested reader may also wish to examine additional properties of cool-lex order for bubble languages. One avenue for research centers on the total distance that symbols must be shifted when circularly generating cool-lex order. In particular, the following list provides the cool-lex order of permutations over $\{1, 2, 3, 4\}$

$$\overleftarrow{1432}, \overleftarrow{4132}, \overleftarrow{3412}, \overleftarrow{1342}, \overleftarrow{3142}, \overleftarrow{4312}, \overleftarrow{2431}, \overleftarrow{4231}, \overleftarrow{1423}, \overleftarrow{4123}, \overleftarrow{2413}, \overleftarrow{1243},$$
$$\overleftarrow{2143}, \overleftarrow{4213}, \overleftarrow{3421}, \overleftarrow{2341}, \overleftarrow{3241}, \overleftarrow{1324}, \overleftarrow{3124}, \overleftarrow{2314}, \overleftarrow{1234}, \overleftarrow{2134}, \overleftarrow{3214}, \overleftarrow{4321}.$$

Within this order, the total distance that the shifted symbols travel is

$$1+2+2+1+2+3+1+3+1+2+2+1+2+3+2+1+3+1+2+2+1+2+3+3 = 46.$$

In [102] it is shown that the average distance of shifted symbols is slightly less than 2 when generating the permutations of a set in cool-lex order. Is this the minimum distance across all prefix-shift Gray codes for permutations? Furthermore, can similar results be proven for the permutations of a multiset? The application of the aforementioned result in [102] is that Algorithm 7 can be modified to create a CAT algorithm for generating permutations of a set in an array. In particular, only three array assignments are required to generate the next permutation in the average case.

Another area of open problems includes further extensions of $\overleftarrow{\mathsf{cool}}$. While the previously mentioned extensions allow the value of the shifted symbol to be changed, it may also be beneficial to think of every shift as the deletion of a symbol followed by the insertion of a symbol. By using these two operations individually, it may be possible to further generalize the results of this thesis to languages whose strings do not necessarily have the same length.

Another way to create Gray codes for languages that do not have fixed-content is to simply *layer* the constituent cool-lex lists. For example, an ordering of the binary necklaces of length n can be obtained by layering the fixed-density necklace languages

in cool-lex order by decreasing density. This is illustrated below for $n = 5$ where $\mathbb{M}_i$ is used to denote $\mathsf{content}(1^i0^{5-i})$ and $\overline{1} = 0$

$$\begin{aligned}&\overrightarrow{\mathcal{C}}(\mathbb{M}_5), \overrightarrow{\mathcal{C}}(\mathbb{M}_4), \overrightarrow{\mathcal{C}}(\mathbb{M}_3), \overrightarrow{\mathcal{C}}(\mathbb{M}_2), \overrightarrow{\mathcal{C}}(\mathbb{M}_1), \overrightarrow{\mathcal{C}}(\mathbb{M}_0)\\ &= 1111\overline{1}, 111\overline{1}0, \overrightarrow{11\overline{1}00}, 110\overline{1}0, \overrightarrow{11\overline{0}00}, 10\overline{1}00, \overline{1}0000, 00000.\end{aligned}$$

To prove that this always produces a type of Gray code, the interested reader needs only to prove that the last string in one sublist is similar to the first string in the next sublist. Furthermore, this task is quite simple since Lemma 3.2.4 on page 116 provides exact expressions for these boundary strings. For a more involved example, the following list reverses successive cool-lex sublists for restricted Schröder paths to obtain a list of all unrestricted Schröder paths of a given length

$$\begin{aligned}&\overrightarrow{\mathcal{C}}(\mathbf{S}_0(3)), \overleftarrow{\mathcal{C}}(\mathbf{S}_1(3)), \overrightarrow{\mathcal{C}}(\mathbf{S}_2(3)), \overleftarrow{\mathcal{C}}(\mathbf{S}_3(3))\\ &= 22\overrightarrow{2000}, 2\overrightarrow{20}020, 20\overrightarrow{2020}, 2\overrightarrow{20}200, 202\mathbf{2}\mathbf{0}0, \overleftarrow{202}\mathbf{1}0, 2\overleftarrow{2010}, \overleftarrow{20}201, \overleftarrow{2}2001, \overleftarrow{1}2200, 2\overleftarrow{1200},\\ &\quad \overleftarrow{2012}0, \overleftarrow{1}2020, \overleftarrow{2102}0, 2\mathbf{2}\mathbf{1}\mathbf{0}0, \overrightarrow{2\mathbf{1}}\overrightarrow{1}0, 1\overrightarrow{12}0, \overrightarrow{1210}, \overrightarrow{21}01, \overrightarrow{120\overline{1}}, \mathbf{20}11, \mathbf{1}11\end{aligned}$$

This approach ensures that successive Schröder paths with the same content differ by a shift, while successive Schröder paths with different content differ by the deletion of a 2 and a 0 and the insertion of a 1 (in bold). (In particular, successive strings of different content will either be $2^i1^j0^i$ followed by $2^{i-1}1^{j+1}0^{i-1}$, or $202^{i-1}1^j0^{i-1}$ followed by $202^{i-2}1^{j+1}0^{i-2}$.) In general, the interested reader may wish to examine different ways of layering cool-lex lists for various languages that do not have fixed-content.

Finally, the recursive definition of cool-lex order shows that useful variations of lexicographic order have not yet been exhausted. In particular, the cool-lex variation is based on a somewhat non-standard view of co-lexicographic order for fixed-content languages. For this reason, it may be useful to consider additional ways of reexpressing standard lexicographic orders. More specifically, it may be possible to to create interesting lists by simply reordering the scuts.

5.3 Algorithms

By using $\overleftarrow{\mathsf{cool}}$ it is possible to develop efficient iterative algorithms for generating certain bubble languages. Specific loopless algorithms are presented for combinations, balanced parentheses, and multiset permutations in Sections 4.1.1, 4.1.2, and 4.1.3, respectively. The first two of these algorithms store the current string in an array,

while the third algorithm stores the current string in a linked list. In all three cases the provided algorithms use a constant number of additional variables. In particular, the multiset permutation algorithm has the surprising property that it is both loopless and does not explicitly store information on the underlying multiset of symbols that it operates upon. Section 4.1.1 also includes an efficient branchless algorithm, and an implementation using computer words.

5.3.1 Additional Results for Algorithms

One may wish to create loopless algorithms for additional fixed-density bubble languages by modifying the existing algorithms for combinations and balanced parentheses. However, a complication can arise when attempting to update the value of x in O(1)-time. To illustrate the difficulty, consider the following transition that occurs when generating k-ary Dyck words

$$\overleftarrow{10^{k-1}10}\boldsymbol{z} = 110^{k}\boldsymbol{z}.$$

Notice that $\urcorner\llcorner(110^{k}\boldsymbol{z}) \in \{110^{k}, 110^{k+1}, \cdots, 110^{2k-2}\}$ by Definition 2.3.4. In algorithmic terms, this means that there are $k-1$ possibilities for the next value of x. (In the algorithm for balanced parentheses there is only one possibility since balanced parentheses are 2-ary Dyck words.) A simple solution to this problem is to keep track of each index where consecutive array entries are 01. By following this approach it is relatively simple to create loopless algorithms for k-ary Dyck words and linear-extensions of B-posets. Furthermore, by introducing a data structure that allows successive comparisons between $\boldsymbol{s}$ and $\overline{\boldsymbol{s}}$ in O(1)-time, it is also possible to generate connected proper interval graphs with a loopless algorithm.

While the aforementioned results follow naturally from the general framework developed in this thesis, Algorithm 8 is more substantial since it is conjectured to generate the m^n n-tuples of $\{1, 2, \cdots, m\}$. Although a formal proof of correctness is beyond the scope of this section, its correctness can be easily verified for small values,

including the following list of 4-tuples over $\{1, 2, 3, 4\}$ that it generates in reverse order

$\overrightarrow{3333}, \overrightarrow{3330}, \overrightarrow{3300}, \overrightarrow{3000}, \overrightarrow{0002}, \overrightarrow{0020}, \overrightarrow{0200}, \overrightarrow{2003}, \overrightarrow{0023}, \overrightarrow{0203}, \overrightarrow{2030}, \overrightarrow{0230}, \overrightarrow{2300}, \overrightarrow{3002}, \overrightarrow{0032}, \overrightarrow{0302},$
$\overrightarrow{3020}, \overrightarrow{0320}, \overrightarrow{3203}, \overrightarrow{2033}, \overrightarrow{0233}, \overrightarrow{2303}, \overrightarrow{3023}, \overrightarrow{0323}, \overrightarrow{3230}, \overrightarrow{2330}, \overrightarrow{3302}, \overrightarrow{3032}, \overrightarrow{0332}, \overrightarrow{3323}, \overrightarrow{3233}, \overrightarrow{2333},$
$\overrightarrow{3332}, \overrightarrow{3320}, \overrightarrow{3200}, \overrightarrow{2002}, \overrightarrow{0022}, \overrightarrow{0202}, \overrightarrow{2020}, \overrightarrow{0220}, \overrightarrow{2203}, \overrightarrow{2023}, \overrightarrow{0223}, \overrightarrow{2230}, \overrightarrow{2320}, \overrightarrow{3202}, \overrightarrow{2032}, \overrightarrow{0232},$
$\overrightarrow{2302}, \overrightarrow{3022}, \overrightarrow{0322}, \overrightarrow{3223}, \overrightarrow{2233}, \overrightarrow{2323}, \overrightarrow{3232}, \overrightarrow{2332}, \overrightarrow{3322}, \overrightarrow{3220}, \overrightarrow{2202}, \overrightarrow{2022}, \overrightarrow{0222}, \overrightarrow{2223}, \overrightarrow{2232}, \overrightarrow{2322},$
$\overrightarrow{3222}, \overrightarrow{2222}, \overrightarrow{2220}, \overrightarrow{2200}, \overrightarrow{2000}, \overrightarrow{0001}, \overrightarrow{0010}, \overrightarrow{0100}, \overrightarrow{1003}, \overrightarrow{0013}, \overrightarrow{0103}, \overrightarrow{1030}, \overrightarrow{0130}, \overrightarrow{1300}, \overrightarrow{3001}, \overrightarrow{0031},$
$\overrightarrow{0301}, \overrightarrow{3010}, \overrightarrow{0310}, \overrightarrow{3103}, \overrightarrow{1033}, \overrightarrow{0133}, \overrightarrow{1303}, \overrightarrow{3013}, \overrightarrow{0313}, \overrightarrow{3130}, \overrightarrow{1330}, \overrightarrow{3301}, \overrightarrow{3031}, \overrightarrow{0331}, \overrightarrow{3313}, \overrightarrow{3133},$
$\overrightarrow{1333}, \overrightarrow{3331}, \overrightarrow{3310}, \overrightarrow{3100}, \overrightarrow{1002}, \overrightarrow{0012}, \overrightarrow{0102}, \overrightarrow{1020}, \overrightarrow{0120}, \overrightarrow{1200}, \overrightarrow{2001}, \overrightarrow{0021}, \overrightarrow{0201}, \overrightarrow{2010}, \overrightarrow{0210}, \overrightarrow{2103},$
$\overrightarrow{1023}, \overrightarrow{0123}, \overrightarrow{1203}, \overrightarrow{2013}, \overrightarrow{0213}, \overrightarrow{2130}, \overrightarrow{1230}, \overrightarrow{2310}, \overrightarrow{3102}, \overrightarrow{1032}, \overrightarrow{0132}, \overrightarrow{1302}, \overrightarrow{3012}, \overrightarrow{0312}, \overrightarrow{3120}, \overrightarrow{1320},$
$\overrightarrow{3201}, \overrightarrow{2031}, \overrightarrow{0231}, \overrightarrow{2301}, \overrightarrow{3021}, \overrightarrow{0321}, \overrightarrow{3213}, \overrightarrow{2133}, \overrightarrow{1233}, \overrightarrow{2313}, \overrightarrow{3123}, \overrightarrow{1323}, \overrightarrow{3231}, \overrightarrow{2331}, \overrightarrow{3312}, \overrightarrow{3132},$
$\overrightarrow{1332}, \overrightarrow{3321}, \overrightarrow{3210}, \overrightarrow{2102}, \overrightarrow{1022}, \overrightarrow{0122}, \overrightarrow{1202}, \overrightarrow{2012}, \overrightarrow{0212}, \overrightarrow{2120}, \overrightarrow{1220}, \overrightarrow{2201}, \overrightarrow{2021}, \overrightarrow{0221}, \overrightarrow{2213}, \overrightarrow{2123},$
$\overrightarrow{1223}, \overrightarrow{2231}, \overrightarrow{2321}, \overrightarrow{3212}, \overrightarrow{2132}, \overrightarrow{1232}, \overrightarrow{2312}, \overrightarrow{3122}, \overrightarrow{1322}, \overrightarrow{3221}, \overrightarrow{2212}, \overrightarrow{2122}, \overrightarrow{1222}, \overrightarrow{2221}, \overrightarrow{2210}, \overrightarrow{2100},$
$\overrightarrow{1001}, \overrightarrow{0011}, \overrightarrow{0101}, \overrightarrow{1010}, \overrightarrow{0110}, \overrightarrow{1103}, \overrightarrow{1013}, \overrightarrow{0113}, \overrightarrow{1130}, \overrightarrow{1310}, \overrightarrow{3101}, \overrightarrow{1031}, \overrightarrow{0131}, \overrightarrow{1301}, \overrightarrow{3011}, \overrightarrow{0311},$
$\overrightarrow{3113}, \overrightarrow{1133}, \overrightarrow{1313}, \overrightarrow{3131}, \overrightarrow{1331}, \overrightarrow{3311}, \overrightarrow{3110}, \overrightarrow{1102}, \overrightarrow{1012}, \overrightarrow{0112}, \overrightarrow{1120}, \overrightarrow{1210}, \overrightarrow{2101}, \overrightarrow{1021}, \overrightarrow{0121}, \overrightarrow{1201},$
$\overrightarrow{2011}, \overrightarrow{0211}, \overrightarrow{2113}, \overrightarrow{1123}, \overrightarrow{1213}, \overrightarrow{2131}, \overrightarrow{1231}, \overrightarrow{2311}, \overrightarrow{3112}, \overrightarrow{1132}, \overrightarrow{1312}, \overrightarrow{3121}, \overrightarrow{1321}, \overrightarrow{3211}, \overrightarrow{2112}, \overrightarrow{1122},$
$\overrightarrow{1212}, \overrightarrow{2121}, \overrightarrow{1221}, \overrightarrow{2211}, \overrightarrow{2110}, \overrightarrow{1101}, \overrightarrow{1011}, \overrightarrow{0111}, \overrightarrow{1113}, \overrightarrow{1131}, \overrightarrow{1311}, \overrightarrow{3111}, \overrightarrow{1112}, \overrightarrow{1121}, \overrightarrow{1211}, \overrightarrow{2111},$
$\overrightarrow{1111}, \overrightarrow{1110}, \overrightarrow{1100}, \overrightarrow{1000}, \overrightarrow{0000}, \overrightarrow{0003}, \overrightarrow{0030}, \overrightarrow{0300}, \overrightarrow{3003}, \overrightarrow{0033}, \overrightarrow{0303}, \overrightarrow{3030}, \overrightarrow{0330}, \overrightarrow{3303}, \overrightarrow{3033}, \overrightarrow{0333}.$

Recursively, the above list is similar to the list of binary strings in (5.2) since it can be partitioned into sublists of the form $\mathtt{L}(\mathbb{M})$ and $\overrightarrow{\mathcal{C}}(\Pi(\mathtt{L}(\mathbb{M})))\backslash\{\mathtt{L}(\mathbb{M})\}$. Iteratively, the list is generated by shifting the first symbol to the right, and allowing the value of this shifted symbol to change if it does not pass over an increasing pair of symbols $d_i d_j$ with $i < j$. The algorithm stores the n-tuple in a doubly-linked list, and uses only two additional pointers. The call to `init`(n, m) returns a doubly-linked list containing one node with value 1 followed by $n-1$ nodes with value m. During each iteration, the pointers `h`, `k`, and `t` respectively point to the head of the tuple, the last node in the non-increasing prefix of the tuple, and the node that is left-shifted into the first position of the tuple.

While recursive algorithms were not explicitly discussed in this thesis, significant work has been done by Sawada-Tsang to develop recursive CAT algorithms for fixed-density necklaces and Lyndon words using cool-lex order [77].

```
h ← init(n, m)
visit(h)
k ← h
while k.next ≠ φ or k.val < m do
  if k.next = φ
    t ← k
    k ← k.prev
  else if k.next.next = φ or k.val < k.next.next.val
    t ← k.next
  else
    t ← k.next.next
  end
  if k.next = φ or k.next.next = φ
    if t.val = m
      t.val ← 1
    else if t.val ≥ h.val
      t.val ← t.val + 1
    else if t.val = 1
      t.val ← h.val
    end
  end
  t.prev.next ← t.next
  if t.next ≠ φ
    t.next.prev ← t.prev
  end
  h.prev ← t
  t.next ← h
  t.prev ← φ
  h ← t
  if h.val > h.next.val
    k ← h
  end
  visit(h)
end
```

Algorithm 8: CoolTuple(n, m) is a loopless algorithm that is conjectured to generate the n-tuples of $\{1, 2, \ldots, m\}$ in a doubly linked list with head `h` using additional pointers `k` and `t`.

5.3.2 Open Problems for Algorithms

The interested reader may wish to pursue loopless algorithms for additional fixed-content bubble languages by modifying the provided algorithm for multiset permutations. Natural candidates include ordered trees with fixed branching sequence, and multiset permutations with a maximum number of inversions. At first glace, the former problem appears to be substantially simpler than the latter. Even more challenging would be the creation of loopless algorithms that generate fixed-density or fixed-content necklaces and Lyndon words.

Ranking is an important topic in combinatorial generation that was not considered in this thesis. Given a string $\boldsymbol{s}$ in list $\mathcal{T}$, the *rank of* $\boldsymbol{s}$ *in* $\mathcal{T}$ is the position that $\boldsymbol{s}$ appears in $\mathcal{T}$ and is this value is denoted $\mathsf{rank}_{\mathcal{T}}(\boldsymbol{s})$. (It is customary for the first string in the list to be assigned rank 0.) The rank of combinations and balanced parentheses in cool-lex order can be computed using $\mathrm{O}(n)$ arithmetic operations [73, 72]. Besides these two results, the amount of research into ranking cool-lex order has not been commensurate with its potential. In particular, cool-lex order has a fairly simple recursive structure, and so it may be possible to develop useful general results. Applications of ranking and unranking can include efficient parallelization of generation algorithms, efficient random selection of strings in the underlying language, and alternate proof techniques for previously discovered results.

Of equal importance to developing new algorithms is the realization of new applications for existing algorithms. The stacker-crane problem in Section 1.2.5 provides a real-world application for the loopless algorithm for generating multiset permutations in cool-lex order. An interesting open problem is to determine additional applications for the algorithms in this thesis. Towards the goal of increasing the applicability of these algorithms, it would be useful to further analyze the amount of work performed when cool-lex order is generated using arrays (see the note at the end of Section 4.1.3).

5.4 Shorthand Universal Cycles

Universal cycles are circular strings whose substrings include exactly one copy of each string in a given language. This thesis recounts the simple observation that universal cycles do not exist for fixed-content languages (except in trivial cases). On the other hand, the last symbol of each string in a fixed-content language is redundant. For this reason, this thesis suggests the investigation of shorthand universal cycles.

A shorthand universal cycle for language $\mathbf{L}$ has the property that its substrings include exactly one copy of the first $n-1$ symbols of each string in $\mathbf{L}$. Due to the aforementioned redundancy, this single omitted symbol does significantly impact the difficulty of obtaining the strings in $\mathbf{L}$ from a shorthand universal cycle. Furthermore, in Section 4.2 it is shown that shorthand universal cycles are intimately related to the main topic of this thesis. In particular, if $\mathbf{L}$ is a fixed-content language then it has a shorthand universal cycle if and only if it has a circular right-shift Gray code in which the first symbol is always shifted into the last or second-last position.

Section 4.2.1 points out that shorthand universal cycles for permutation were previously shown to exist under another name, and that their efficient generation dates back to an anonymous bell-ringer. Section 4.2.2 points out that shorthand universal cycles for combinations include universal cycles for the middle levels. Section 4.2.3 goes on to prove that shorthand universal cycles for multiset permutations can be explicitly constructed using reverse cool-lex order. Interestingly this construction is a direct analogue of the FKM algorithm for constructing universal cycles for n-tuples using lexicographic order. Finally, a modification of the construction is discussed for shorthand universal cycles of aperiodic multiset permutations.

5.4.1 Additional Results for Shorthand Universal Cycle

While efficient generation of shorthand universal cycles is not discussed in the thesis, there do appear to be at least two cases when they can be efficiently generated for the permutations of a multiset. The first case occurs when $n_m = 1$. That is, the underlying multiset $\mathbb{M}$ contains a single copy of its largest symbol. In this case the necklaces over $\mathbb{M}$ are aperiodic and consist of d_m prefixed to every permutations of $\mathbb{M}$ with d_m removed. From (5.1) in Section 5.2.1, the (reverse) cool-lex list of the necklaces can be obtained by prefixing d_m to the (reverse) cool-lex list of the permutations. Therefore, the construction reduces to prefixing d_m to every permutation of $\mathbb{M}\backslash\{d_m$ in reverse cool-lex order. To make this discussion more concrete, notice that

$$\boldsymbol{U}(\{1,1,2,2,3\}) = 32211 \cdot 32112 \cdot 31122 \cdot 31212 \cdot 32121 \cdot 31221$$

and

$$\overrightarrow{\mathcal{C}}\,(\Pi(\{1,1,2,2\})) = 2211, 2112, 1122, 1212, 2121, 1221.$$

Thus, the shorthand universal cycle can be constructed from the loopless algorithm for generating multiset permutations (although the order or the individual strings

need to be reversed). The second case occurs when $m = 2$. That is, the underlying multiset contains only two distinct symbols. In this case the CAT algorithms developed by Sawada-Tsang [77] also allow the shorthand universal cycle to be constructed efficiently.

5.4.2 Open Problems for Shorthand Universal Cycles

In general, the study of shorthand universal cycles is completely open. As previously mentioned, one important question is to determine efficient generation algorithms for the shorthand universal cycles that are known to exist. Another important question involves the existence of shorthand universal cycles for additional combinatorial objects. In particular, the end of Section 4.2.3 mentions the $\boldsymbol{U}_{\_}(\mathbb{M})$ modification for avoiding periodic multiset permutations. Formally proving that $\boldsymbol{U}_{\_}(\mathbb{M})$ is always a shorthand universal cycle for the aperiodic permutations over $\mathbb{M}$ would be an ideal exercise. The reader may also wish to consider an analogous modification of the FKM algorithm for n-tuples in which only the Lyndon words of length n are included in the concatenation. For example, the following concatenation results from removing the "short" Lyndon words (0, 01, and 1) from the underlined portions of (1.7)

$$000100110111.$$

Notice that this modified de Bruijn cycle avoids 0000, 1010, 0101, and 1111.

Given the existence of a (shorthand) universal cycle for a fixed-content language $\mathbf{L}$, one may wish to know the number of (shorthand) universal cycles that exist for $\mathbf{L}$. For example, it is known that there are $2^{2^{n-1}-n}$ de Bruijn cycles for the binary strings of length n [13] (also see de Bruijn [14] for an acknowledgement of priority on this result dating back to Flye Sainte-Marie in 1894 [74]).

Besides counting the number of shorthand universal cycles, one may instead wish to further refine their properties. For example, recall that successive strings in $\mathbf{L}$ obtained from a shorthand universal cycle will differ by right-shifting the first symbol into the last or second-last position. Given this observation, it may be natural to attempt to maximize or minimize the number of times the first symbol is shifted into the last position. In terms of the discussion in Section 4.2.1, these questions are analogous to minimizing or maximizing the sum of the associated binary string. This pursuit does have potential applications, since in the stacker-crane problem of Section 1.2.5, the operation $\mathsf{shift}(\boldsymbol{s}, 1, n)$ requires the addition and subtraction of only one shortest path length. This concern was addressed for the shorthand universal cycle

for permutations given in [71]. Another question addressed in [71] is whether strings can be efficiently ranked and unranked within a given shorthand universal cycle. It may also be possible to generalize shorthand universal cycles for certain $\mathbf{L}$ to higher dimensions (see Hurlbert-Isaac [38] for a discussion of de Bruijn tori).

While the discussion of universal cycles in this thesis has focused on shorthand universal cycle, the examples constructed in this thesis can also be considered as universal cycles on different sets of strings. In particular, the shorthand universal cycles for $\mathbf{C}(j,k)$ are simply universal cycles for $\mathbf{C}(j-1,k) \cup \mathbf{C}(j,k-1)$. Given this interpretation it becomes natural to ask for constructions of density-range de Bruijn cycles. A *density-range de Bruijn cycle* is a universal cycle for the binary strings of length n subject to a minimum and maximum number of 1s. For example, the following universal cycle contains the 26 binary strings of length 5 with at least 0 and at most 3 copies of 1

$$111000 \cdot 100000 \cdot 110010 \cdot 10 \cdot 110100 = 11100010000011001010110100.$$

Existence of these density-range de Bruijn cycles is not difficult to verify [37], however their construction and efficient generation is open.

Appendix A

Notation

"Any inaccuracies in this index may be explained by the fact that it has been sorted with the help of a computer."

- Don Knuth

The tables in this appendix summarize the notation in this thesis. Within the tables there are a number of examples. Unless otherwise specified the examples refer to the following values

$$\mathbf{L} = \{5440, 5404, 5044, 4540, 4450\}$$
$$\mathcal{T} = 5440, 4450, 4540, 5404, 5044.$$

The language $\mathbf{L}$ is an example of a bubble language, while $\mathcal{T}$ is the reverse cool-lex order of the strings in $\mathbf{L}$. For simplicity, several conventions are specified in Chapter 2. These conventions set additional values depending on the chosen fixed-content language $\mathbf{L}$. In particular, the content of the strings in $\mathbf{L}$ is denoted by $\mathbb{M}$. Given the above choice of $\mathbf{L}$, this value is

$$\mathbb{M} = \{0, 4, 4, 5\}.$$

Similarly, n, m, n_i, $\underline{n}_1$, and $\overline{n}_1$ denote symbol multiplicities in $\mathbb{M}$ according to Table A.13. Unless otherwise specified, these values will also be used in examples contained in the tables.

Object	Description	Typography	Example	Page
symbol	indivisible unit	regular	3	40
set	set of symbols	uppercase blackboard	Σ	40
multiset	multiset of symbols	uppercase blackboard	$\mathbb{M}$	40
string	sequence of symbols	lowercase bold	$\boldsymbol{s}$	42
language	set of strings	uppercase bold	$\mathbf{L}$	42
list	sequence of strings	uppercase calligraphic	$\mathcal{T}$	111

Table A.1: Typography for symbols, sets, multisets, strings, languages, and lists.

Language	Description	Page	
	combinations	$\mathbf{C}(2,1) = \{001, 010, 100\}$	57
$\mathbf{P}(h)$	balanced parentheses	$\mathbf{P}(2) = \{1100, 1010\}$	58
$\mathbf{P}_b(h)$	$\mathbf{P}(h)$ with $\leq b$ balanced prefixes	$\mathbf{P}_1(2) = \{1100\}$	169
$\mathbf{D}(k,i)$	k-ary Dyck words	$\mathbf{D}(3,2) = \{110000, 101000, 100100\}$	58
$\mathbf{B}_n(b_1,\ldots)$	linear-extensions of B-posets	$\mathbf{B}_6(1,2,5) = \{111000, 110100, 110010\}$	60
$\mathbf{I}(h)$	connected unit interval graphs	$\mathbf{I}(3) = \{111000, 110100\}$	66

Table A.2: Combinatorial objects represented by fixed-density languages.

Language	Description	Page	
$\Pi(\mathbb{M})$	multiset permutations	$\Pi(\mathbb{M}) = \{0445, 0454, 0544, 4045, \ldots\}$	57
$\mathbf{T}(\mathbb{M})$	ordered trees w/ branching seq.	$\mathbf{T}(\{0,0,1,3\}) = \{3100, 3010, 3001, 1300\}$	69
$\mathbf{M}_w(h)$	restricted Motzkin paths	$\mathbf{M}_1(2) = \{12200, 21200, 22100, 22010, 22001, \ldots$	74
$\mathbf{S}_w(h)$	restricted Schröder paths	$\ldots, 12020, 21020, 20120, 20210, 20201\} = \mathbf{S}_1(2)$	73
$\circlearrowleft(\boldsymbol{s})$	rotation set	$\circlearrowleft(1123) = \{1123, 1231, 2311, 3112\}$	75
$\mathbf{N}(\mathbb{M})$	necklaces	$\mathbf{N}(\{0,0,2,2\}) = \{2200, 2020\}$	76
$\mathbf{N}^-(\mathbb{M})$	Lyndon words	$\mathbf{N}^-(\{0,0,2,2\}) = \{2020\}$	76
$\mathbf{N}^+(\mathbb{M})$	pre-necklaces	$\mathbf{N}^+(\{0,0,2,2\}) = \{2200, 2020, 2002\}$	86
$\mathbf{R}(\mathbb{M})$	bracelets	$\mathbf{R}(\{0,0,2,2\}) = \{2200, 2020\}$	76
$\mathbf{V}(\mathbb{M}, i)$	$\Pi(\mathbb{M})$ with $\leq i$ inversions	$\mathbf{V}(\mathbb{M}, 2) = \{5440, 5404, 5044, 4540, 4504, 4450\}$	168

Table A.3: Combinatorial objects represented by fixed-content languages.

Shift	Assumption	Description	Example	Page
$\cdots\overleftarrow{s_i \cdots s_j}\cdots$	$1 \leq i \leq j \leq \lvert\boldsymbol{s}\rvert$	left-shift s_i into the jth index	$\overleftarrow{5440} = 5044$	2
$\cdots\overrightarrow{s_i \cdots s_j}\cdots$	$1 \leq i \leq j \leq \lvert\boldsymbol{s}\rvert$	right-shift s_i into the jth index	$5\overrightarrow{044} = 5440$	2
$\overleftarrow{\mathsf{shift}}(\boldsymbol{s}, i, j)$	$1 \leq j \leq i \leq \lvert\boldsymbol{s}\rvert$	left-shift s_i into the jth index	$\overleftarrow{\mathsf{shift}}(4450, 3, 1) = \overleftarrow{4450}$	48
$\overrightarrow{\mathsf{shift}}(\boldsymbol{s}, i, j)$	$1 \leq i \leq j \leq \lvert\boldsymbol{s}\rvert$	right-shift s_i into the jth index	$\overrightarrow{\mathsf{shift}}(5440, 1, 3) = \overrightarrow{5440}$	48
$\overleftarrow{\mathsf{bubble}}(\boldsymbol{s}, i)$	$\exists j < i,\ s_i \neq s_j$	left-shift s_i past one $\neq s_i$	$\overleftarrow{\mathsf{bubble}}(5440, 3) = \overleftarrow{5440}$	49
$\overrightarrow{\mathsf{bubble}}(\boldsymbol{s}, i)$	$\exists j > i,\ s_i \neq s_j$	right-shift s_i past one $\neq s_i$	$\overrightarrow{\mathsf{bubble}}(5440, 2) = 5\overrightarrow{440}$	50
$\overleftarrow{\mathsf{greedy}}_{\mathbf{L}}(\boldsymbol{s}, i)$	$\boldsymbol{s} \in \mathbf{L}$	left-shift s_i while in $\mathbf{L}$	$\overleftarrow{\mathsf{greedy}}_{\mathbf{L}}(5440, 4) = \overleftarrow{5440}$	103
$\overrightarrow{\mathsf{max}}_{\mathbf{L}}(\boldsymbol{s}, i, j)$	$\boldsymbol{s} \in \mathbf{L}$ and $i \leq j$	... while in $s_i \cdots s_j$ and $\mathbf{L}$	$\overrightarrow{\mathsf{max}}_{\mathbf{L}}(4450, 1, 4) = \overrightarrow{4450}$	130
$\overleftarrow{\mathsf{cool}}_{\mathbf{L}}(\boldsymbol{s})$	$\boldsymbol{s} \in \mathbf{L}$	cool-lex left-shift	$\overleftarrow{\mathsf{cool}}_{\mathbf{L}}(4540) = \overleftarrow{4540}$	105
$\overrightarrow{\mathsf{cool}}_{\mathbf{L}}(\boldsymbol{s})$	$\boldsymbol{s} \in \mathbf{L}$	cool-lex right-shift	$\overrightarrow{\mathsf{cool}}_{\mathbf{L}}(4540) = 5\overrightarrow{404}$	129

Table A.4: Shifts.

Operation	Assumption	Description	Example	Page
$\cdots s_i \cdots s_j \cdots$	$1 \le i \le j \le \lvert \boldsymbol{s} \rvert$	transpose s_i and s_j	$\overbrace{5044}$ = 4045	13
$\mathsf{transpose}(\boldsymbol{s}, i, j)$	$1 \le i \le j \le \lvert \boldsymbol{s} \rvert$	transpose s_i and s_j	$\mathsf{transpose}(5044, 1, 4) = 4045$	141
$\cdots \overleftrightarrow{s_i \cdots s_j} \cdots$	$1 \le i \le j \le \lvert \boldsymbol{s} \rvert$	reverse $s_i \cdots s_j$	$\overleftrightarrow{5044}$ = 4405	21

Table A.5: Transpositions and substring reversals.

Operation	Description	Examples	Page
$\boldsymbol{r} \cdot \boldsymbol{t}$	concatenate strings $\boldsymbol{r}$ and $\boldsymbol{t}$	$\boldsymbol{r} = 4$ and $\boldsymbol{t} = 450$ implies $\boldsymbol{r} \cdot \boldsymbol{t} = 4450$	42
$\boldsymbol{s}^i$	i concatenations of $\boldsymbol{s}$	$\boldsymbol{s}^3 = \boldsymbol{s} \cdot \boldsymbol{s} \cdot \boldsymbol{s}$	42
$\mathcal{T} \cdot \boldsymbol{z}$	concatenate $\boldsymbol{z}$ to list $\mathcal{T}$	$\mathcal{T} \cdot 5 = 54405, 44505, 45405, 54045, 50445$	114
$\bigodot \mathcal{T}$	concatenate list $\mathcal{T}$	$\bigodot \mathcal{T} = 54404504540540450 44$	136

Table A.6: Concatenations.

Operation	Description	Examples	Page
$\mathsf{L}(\boldsymbol{s})$	non-increasing prefix of $\boldsymbol{s}$	$\mathsf{L}(4405) = 440$	45
$\mathsf{LL}(\boldsymbol{s})$	weakly non-increasing prefix of $\boldsymbol{s}$	$\mathsf{LL}(3212133121313) = 32121$	129
$\mathsf{L}_{\mathbf{L}}(\boldsymbol{s})$	frozen prefix of $\boldsymbol{s}$ in $\mathbf{L}$	$\mathsf{L}_{\mathbf{L}}(5404) = 54$	47
$\mathsf{short}(\boldsymbol{s})$	shorthand of $\boldsymbol{s}$	$\mathsf{short}(5404) = 540$	133
$\mathsf{aperiodic}(\boldsymbol{s})$	aperiodic prefix of $\boldsymbol{s}$	$\mathsf{aperiodic}(1101011010) = 11010$	158

Table A.7: Prefixes.

Operation	Description	Examples	Page
$\mathsf{content}(\boldsymbol{s})$	content of string $\boldsymbol{s}$	$\mathsf{content}(4540) = \{0, 4, 4, 5\}$	42
$\#_i(\boldsymbol{s})$	occurrences of i in $\boldsymbol{s}$	$\#_4(4540) = 2$	73
$\Sigma(\boldsymbol{s})$	sum of symbols in $\boldsymbol{s}$	$\Sigma(4405) = 13$	41
$\circlearrowleft_i(\boldsymbol{s})$	rotation of $\boldsymbol{s}$ starting at s_i	$\circlearrowleft_3(4405) = 0544$	75
$\blacktriangle(\boldsymbol{s}, i)$	ith peak of $\boldsymbol{s}$	$\blacktriangle(3212133121313, 2) = 33121$	77
⏱	room-to-room times	⏱(①②) = 1	??

Table A.8: Miscellaneous string operations.

Operation	Description	Examples	Page
$\mathsf{content}(\mathbf{L})$	content of fixed-content $\mathbf{L}$	$\mathsf{content}(\mathbf{L}) = \{0,4,4,5\}$	43
$\mathsf{reverse}(\mathcal{T})$	reverse list $\mathcal{T}$	$\mathsf{reverse}(\mathcal{T}) = 5044, 5404, 4540, 4450, 5440$	8
$\mathsf{first}(\mathcal{T})$	first string in list $\mathcal{T}$	$\mathsf{first}(\mathcal{T}) = 5440$	115
$\mathsf{last}(\mathcal{T})$	last string in list $\mathcal{T}$	$\mathsf{last}(\mathcal{T}) = 5044$	115
$\bigcirc_{,}$	creates list (integer index)	$\bigcirc_{j=1,2,3,\,,} 0^j1^j = 01, 0011, 000111$	113
$\bigcirc_{,}$	creates list (list index)	$\bigcirc_{\boldsymbol{z}\in 123,213,321,\,,} z_1 \cdot \boldsymbol{z} = 1123, 2213, 3321$	114
$\mathbf{L}/\boldsymbol{z}$	quotient of $\mathbf{L}$ using $\boldsymbol{z}$	$\mathbf{L}/40 = \{54, 45\}$	44
$\mathbf{L}\backslash\boldsymbol{p}$	left-quotient of $\mathbf{L}$ using $\boldsymbol{p}$	$\mathbf{L}/5 = \{440, 404, 044\}$	170
$\mathsf{short}(\mathbf{L})$	shorthand of $\mathbf{L}$	$\mathsf{short}(\mathbf{L}) = \{544, 540, 504, 454, 445\}$	133
$\mathsf{aperiodic}(\mathcal{T})$	aperiodic prefix of $\mathcal{T}$	$\mathsf{aperiodic}(1010, 1100) = 10, 1100$	158
$\mathsf{rank}_{\mathcal{T}}(\boldsymbol{s})$	rank of $\boldsymbol{s}$ in $\mathcal{T}$	$\mathsf{rank}_{\mathcal{T}}(4540) = 2$	182

Table A.9: Language and list operations.

Operation	Description	Examples	Page
$\urcorner\llcorner(\mathbb{M})$	the non-increasing string over $\mathbb{M}$	$\urcorner\llcorner(\{0,4,4,5\}) = 5440$	46
$\mathsf{set}(\mathbb{M})$	underlying set of symbols in multiset $\mathbb{M}$	$\mathsf{set}(\mathbb{M}) = \{0,4,5\}$	40
$\subseteq$	multiset subset	$\{4,4,5\} \subseteq \{0,4,4,5\}$	41
$=$	multiset equality	$\{0,4,4,5\} \neq \{0,4,5\}$	41
$\backslash$	multiset difference	$\{0,4,4,5\}\backslash\{0,4,5\} = \{4\}$	41

Table A.10: Multiset operations and relations.

Scut / Tail	Assumption	Description	Example	Page
$\mathsf{scut}(\boldsymbol{s})$	$\boldsymbol{s} \neq \urcorner\llcorner$	scut of $\boldsymbol{s}$	$\mathsf{scut}(4450) = 50$	89
$\mathsf{scuts}(\mathbf{L})$		set of $\mathsf{scut}(s)$ for $\boldsymbol{s} \in \mathbf{L}$	$\mathsf{scuts}(\mathbf{L}) = \{540, 50, 4\}$	90
$\mathsf{scut}_{\mathbb{M}}(j,i)$	$2 \le j \le m$	scut $d_j\cdots$ excluding i symbols	$\mathsf{scut}_{\mathbb{M}}(3,2) = 50$	90
$\mathsf{tail}_{\mathbb{M}}(j,i)$	and	non-increasing $\cdot\ \mathsf{scut}_{\mathbb{M}}(j,i)$	$\mathsf{tail}_{\mathbb{M}}(2,1) = 5404$	92
$\mathsf{head}_{\mathbb{M}}(j,i)$	$1 \le i \le \underline{n}_j$	greedy left-shift in $\mathsf{tail}_{\mathbb{M}}(j,i)$	$\mathsf{head}_{\mathbb{M}}(2,1) = \overleftarrow{5404}$	118
$\clubsuit_{\mathbf{L}}$		max j with $\mathsf{scut}_{\mathbf{L}}(j,i) \in \mathsf{scuts}(\mathbf{L})$	$\clubsuit_{\mathbf{L}} = 3$	91
$\clubsuit_{\mathbf{L}}(j)$		max i with $\mathsf{scut}_{\mathbf{L}}(j,i) \in \mathsf{scuts}(\mathbf{L})$	$\clubsuit_{\mathbf{L}}(3) = 2$	91

Table A.11: Scuts, heads, and tails.

List	Description	Example	Page
$\mathcal{Z}(\mathbf{L})$	cool-lex order of $\mathsf{scuts}(\mathbf{L})$	$\mathcal{Z}(\mathbb{M}) = 4, 50, 540$	112
$\overleftarrow{\mathcal{C}}(\mathbf{L})$	cool-lex order of $\mathbf{L}$	$\overleftarrow{\mathcal{C}}(\mathbf{L}) = 5\overleftarrow{044}, \overleftarrow{5404}, \overleftarrow{4540}, \overleftarrow{4450}, 5\overleftarrow{440}$	113
$\overrightarrow{\mathcal{C}}(\mathbf{L})$	reverse cool-lex order of $\mathbf{L}$	$\overrightarrow{\mathcal{C}}(\mathbf{L}) = \overrightarrow{5440}, 44\overrightarrow{50}, \overrightarrow{4540}, 5\overrightarrow{404}, 5\overrightarrow{044}$	129
$\boldsymbol{U}_{+}(\mathbb{M})$	ucycle for $\Pi(\mathbb{M})$ with repetitions	$\boldsymbol{U}_{+}(\{0,0,1,1\}) = 11001010$	156
$\boldsymbol{U}(\mathbb{M})$	ucycle for $\Pi(\mathbb{M})$	$\boldsymbol{U}(\{0,0,1,1\}) = 110010$	159
$\boldsymbol{U}_{-}(\mathbb{M})$	ucycle for $\Pi(\mathbb{M})$ for aperiodic	$\boldsymbol{U}_{-}(\{0,0,1,1\}) = 1100$	165

Table A.12: Cool-lex orders and shorthand universal cycles (ucycles).

Convention	Description	Examples	Page
$\mathsf{content}(\mathbf{L}) = \mathbb{M}$	content of $\mathbf{L}$ is $\mathbb{M}$	$\mathbb{M} = \{0, 4, 4, 5\}$	43
$\mathsf{set}(\mathbb{M}) = \Sigma$	underlying set of $\mathbb{M}$ is Σ	$\Sigma = \{0, 4, 5\}$	40
$\Sigma = \{d_1, d_2, \cdots, d_m\}$	set Σ contains m symbols	$m = 3$	40
$\mathbb{M} = \{e_1, e_2, \cdots, e_n\}$	multiset $\mathbb{M}$ contains n symbols	$n = 4$	40
$d_1 < d_2 < \cdots < d_m$	strictly increasing order of d_i	$d_1 = 0$, $d_2 = 4$, $d_3 = 5$	40
$e_1 \le e_2 \le \cdots \le e_m$	non-decreasing order of e_i	$e_1 = 0$, $e_2 = e_3 = 4$, $e_4 = 5$	40
n_i	multiplicity of d_i in $\mathbb{M}$	$n_1 = 1$, $n_2 = 2$, $n_3 = 1$	40
$\underline{n}_i = n_1 + n_2 + \cdots + n_i$	number of symbols $\le d_i$ in $\mathbb{M}$	$\underline{n}_1 = 1$, $\underline{n}_2 = 3$, $\underline{n}_3 = 4$	41
$\overline{n}_i = n_m + n_{m-1.5} + \cdots + n_i$	number of symbols $\ge d_i$ in $\mathbb{M}$	$\overline{n}_1 = 4$, $\overline{n}_2 = 3$, $\overline{n}_3 = 1$	41
$\boldsymbol{s} = s_1 s_2 \cdots$	ith symbol in a string	$\boldsymbol{s} = 4045$ implies $s_2 = 0$	42
$\lrcorner = e_n e_{n-1.5} \cdots e_1$	short for $\lrcorner(\mathbb{M})$	$\lrcorner = 5440$	46
$\lrcorner \cdot \boldsymbol{z}$	short for $\lrcorner(\mathbb{M} \backslash \mathsf{content}(\boldsymbol{z})) \cdot \boldsymbol{z}$	$\lrcorner \cdot 04 = 54 \cdot 04$	46

Table A.13: Conventions.

Bibliography

[1] D.L. Applegate, R.E. Bixby, V. Chvátal, and W.J. Cook. *The Traveling Salesman Problem: A Computational Study*. Princeton University Press, 2007. [Cited on page 31.]

[2] J. Arndt. *Matters computational*. 2008. [Cited on pages 37 and 149.]

[3] G. Betta, A. Pietrosanto, and A. Scaglione. A Gray-code-based fiber optic liquid level transducer. *IEEE Transactions on Instrumentation and Measurement*, 47(1):174–178, February 1998. [Cited on page 9.]

[4] J. R. Bitner, G. Ehrlich, and E. M. Reingold. Efficient generation of the binary reflected Gray code and its applications. *Communications of the ACM*, 16(9):517–521, September 1976. [Cited on pages 8, 22, and 27.]

[5] R. Burkard, B. Fruhwirth, and G. Rote. Vehicle routing in an automated warehouse: Analysis and optimization. *Annals of Operations Research*, 57(1):29–44, 1995. [Cited on page 32.]

[6] C. Campbell and S. Moire et. al. *The Feast: A Collection of Artwork in Black and White*. 2006. ISBN 0-978066-30-98. [Cited on page 6.]

[7] E. R. Canfield and S. G. Williamson. A loop-free algorithm for generating the linear extensions of a poset. *Order*, 12(1):57–75, 1995. [Cited on pages 23 and 27.]

[8] P.J. Chase. Combination generation and graylex ordering. *Congressus Numerantium*, 69(19):215–242, 1989. [Cited on page 23.]

[9] F. Chung, P. Diaconis, and R.L. Graham. Universal cycles for combinatorial structures. *Discrete Mathematics*, 110:43–59, 1992. [Cited on pages 12 and 24.]

[10] A.C. Clarke. *The Collected Stories of Arthur C. Clarke*. Gollancz, 2001. ISBN 0-575-07065-X. [Cited on page 1.]

[11] A. Coja-Oghlan, S.O. Krumke, and T. Nierhoff. A heuristic for the stacker crane problem on trees which is almost surely exact. *Journal of Algorithms*, 61:1–19, September 2006. [Cited on page 32.]

[12] P. F. Corbett. Rotator graphs: An efficient topology for point-to-point multiprocessor networks. *IEEE Transactions on Parallel and Distributed Systems*, 3:622–626, 1992. [Cited on page 24.]

[13] N.G. de Bruijn. A combinatorial problem. *Koninkl. Nederl. Acad. Wetensch. Proc. Ser A*, 49:758–764, 1946. [Cited on pages 10, 158, and 184.]

[14] N.G. de Bruijn. Acknowledgement of priority to c. flye sainte-marie on the counting of circular arrangements of 2n zeros and ones that show each n-letter word exactly once. *T.H. Report 75-WSK-06, Technological University Eindhoven*, page 13 pages, 1975. [Cited on page 184.]

[15] R. Duckworth and Fabian Stedman. *Tintinnalogia*. 1668. [Cited on pages 15 and 154.]

[16] Jiminy Cricket (Cliff "Ukelele Ike" Edwards) and Rica Moore. Adding combinations. *Addition & Subtraction* Disneyland LP Record ST-1922, 1963. [Cited on page 142.]

[17] S. Effler and F. Ruskey. A cat algorithm for generating permutations with a fixed number of inversions. *Information Processing Letters*, 86(2):107–112, April 2003. [Cited on page 168.]

[18] G. Ehrlich. Loopless algorithms for generating permutations, combinations and other combinatorial configurations. *Journal of the ACM*, 20(3):500–513, 1973. [Cited on page 27.]

[19] P. Erdös and T. Gallai. Graphs with prescribed degrees of vertices (hungarian). *Mat. Lapok*, 11:264–274, 1960. [Cited on page 69.]

[20] H. Fredericksen and I. J. Kessler. An algorithm for generating necklaces of beads in two colors. *Discrete Mathematics*, 61:181–188, 1986. [Cited on page 11.]

[21] H. Fredericksen and J. Maiorana. Necklaces of beads in k colors and $k$ary de Bruijn sequences. *Discrete Mathematics*, 23(3):207–210, 1978. [Cited on page 11.]

[22] G.N. Frederickson and D.J. Guan. Nonpreemptive ensemble motion planning on a tree. *Journal of Algorithms*, 15:29–60, July 1993. [Cited on page 32.]

[23] G.N. Frederickson, M.S. Hecht, and C.E. Kim. Approximation algorithms for some routing problems. *SIAM Journal on Computing*, 7(2):178–193, May 1978. [Cited on pages 31 and 32.]

[24] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, New York, 1979. [Cited on page 31.]

[25] Jr. G.G. Langdon. An algorithm for generating permutations. *Communications of the ACM*, 10(5):298–299, May 1967. [Cited on page 24.]

[26] Jr. G.G. Langdon. Letters to the editor: Generating permutations by nested cycling. *Communications of the ACM*, 11(6):392, June 1968. [Cited on page 24.]

[27] A.J. Goldstein and R.L. Graham. Sequential generation by transposition of all the arrangements of n symbols. *Bell Telephone Laboratories (Internal Memorandum), Murray Hill, NJ*, 1964. [Cited on page 14.]

[28] M. C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. Academic Press, 1980. [Cited on page 64.]

[29] F. Gray. Pulse code communication. *U.S. Patent 2,632,058*, 1947. [Cited on page 8.]

[30] G. Gutin and A.P. Punnen. *The traveling salesman problem and its variations (Combinatorial Optimization)*, volume 12. Springer, 2007. [Cited on page 31.]

[31] S. Hakimi. On the realizability of a set of integers as degrees of the vertices of a graph. *SIAM J. Applied Math.*, 10:496–506, 1962. [Cited on page 69.]

[32] V. Havel. A remark on the existence of finite graphs (hungarian). *Časopis Pěst. Mat.*, 80:477–480, 1955. [Cited on page 69.]

[33] A. Holroyd. Personal communication. 2009. [Cited on pages 153 and 154.]

[34] Dictionary.com (Unabridged) Random House. "scut". `http://dictionary.reference.com/browse/scut`. [Cited on page 33.]

[35] Dictionary.com (Unabridged) Random House. "shorthand". `http://dictionary.reference.com/browse/shorthand`. [Cited on page 133.]

[36] Dictionary.com (Unabridged) Random House. "synecdoche". `http://dictionary.reference.com/browse/synecdoche`. [Cited on page 16.]

[37] G. Hurlbert and G. Isaac. Personal communication. *Canadian Conference on Discrete and Algorithmic Mathematics (CanaDAM)*, 2009. [Cited on pages 157 and 185.]

[38] G. Hurlbert and G. Isaak. On the de bruijn torus problem. *J. Comb. Theory A*, 61(1):50–62, 1995. [Cited on page 185.]

[39] B. Jackson. Universal cycles of k-subsets and k-permutations. *Discrete Mathematics*, 149:123–129, 1996. [Cited on page 25.]

[40] M. Jiang and F. Ruskey. Determining the Hamilton-connectedness of certain vertex-transitive graphs. *Discrete Mathematics*, 133:159–170, 1994. [Cited on page 24.]

[41] J. R. Johnson. Universal cycles for permutations. *Discrete Mathematics*, (in press), 2008. [Cited on page 24.]

[42] S. M. Johnson. Generation of permutations by adjacent transpositions. *Mathematics of Computation*, 17:282–285, 1963. [Cited on page 13.]

[43] A. Kircher. *China illustrata*, volume 3: Sorting and Searching. Johannes Jansson van Waesberg, Amsterdam, 1667. [Cited on page 29.]

[44] D. E. Knuth. *The Art of Computer Programming*, volume 3: Sorting and Searching. Addison-Wesley, third edition edition, 1997. [Cited on page 33.]

[45] D. E. Knuth. *The Art of Computer Programming*, volume 4 fasicle 3: Generating All Combinations and Partitions. Addison-Wesley, errata (updated 10/02/2008) edition, 2005. ISBN 0-201-85394-9. [Cited on pages 4, 26, and 145.]

[46] D. E. Knuth. *The Art of Computer Programming*, volume 4 fasicle 2: Generating All Tuples and Permutations. Addison-Wesley, errata (updated 10/02/2008) edition, 2005. ISBN 0-201-85393-0. [Cited on pages 4, 9, 11, 26, and 27.]

[47] D. E. Knuth. *The Art of Computer Programming*, volume 4 fasicle 4: Generating All Trees, History of Combinatorial Generation. Addison-Wesley, errata (updated 10/02/2008) edition, 2006. ISBN 0-321-33570-8. [Cited on pages 4, 7, 20, and 23.]

[48] C. W. Ko and F. Ruskey. Generating permutations of a bag by interchanges. *Information Processing Letters*, 41:263–269, 1992. [Cited on pages 23 and 27.]

[49] Y. Koda and F. Ruskey. A Gray code for the ideals of a forest poset. *Journal of Algorithms*, 15:324–340, 1993. [Cited on page 172.]

[50] J. F. Korsh and P. S. LaFollette. Multiset permutations and loopless generation of ordered trees with specified degree sequence. *Order*, 34(2):309–336, 2000. [Cited on page 69.]

[51] J. F. Korsh and P. S. LaFollette. Loopless generation of linear extensions of a poset. *Order*, 18(2):115–126, 2002. [Cited on pages 24 and 27.]

[52] J. F. Korsh and P. S. LaFollette. Loopless array generation of multiset permutations. *The Computer Journal*, 47(5):612–621, 2004. [Cited on pages 23 and 27.]

[53] J. F. Korsh and S. Lipschutz. Generating multiset permutations in constant time. *Journal of Algorithms*, 25:321–335, 1997. [Cited on pages 24 and 27.]

[54] C. N. Liu and D. T. Tang. Algorithm 452: enumerating combinations of m out of n objects. *Comm. ACM*, 16(8):485, August 1973. [Cited on page 22.]

[55] M. H. Martin. A problem in arrangements. *Bull. Amer. Math. Soc.*, 40:859–864, 1934. [Cited on page 11.]

[56] R. J. Ord-Smith. Generation of permutation sequences: part 1. *The Computer Journal*, 13(2):152–155, 1970. [Cited on page 14.]

[57] R. J. Ord-Smith. Generation of permutation sequences: part 2. *The Computer Journal*, 14(2):136–139, 1971. [Cited on page 14.]

[58] G. Pruesse and F. Ruskey. Generating the linear extensions of certain posets by transpositions. *SIAM J. Discrete Mathematics*, 4:413–422, 1991. [Cited on page 23.]

[59] G. Pruesse and F. Ruskey. Gray codes from antimatroids. *Order*, 10:239–252, 1993. [Cited on page 23.]

[60] G. Pruesse and F. Ruskey. Generating linear extensions fast. *SIAM Journal on Computing*, 23(2):373–386, April 1994. [Cited on pages 23 and 61.]

[61] M.K. Roy. Evaluation of permutation algorithms. *The Computer Journal*, 21:296–301, 1978. [Cited on page 14.]

[62] F. Ruskey. *Combinatorial Generation*. (in preparation). [Cited on pages xi, 5, 7, 27, and 69.]

[63] F. Ruskey. Adjacent interchange generation of combinations. *Journal of Algorithms*, 9(2):162–180, June 1988. [Cited on page 23.]

[64] F. Ruskey. Generating linear extensions of posets by transpositions. *Journal of Combinatorial Theory (B)*, 54:77–101, 1992. [Cited on pages 22 and 23.]

[65] F. Ruskey. Personal communication. 2009. [Cited on page 171.]

[66] F. Ruskey, C. Savage, and T.M.Y. Wang. Generating necklaces. *J. Algorithms*, 13:414–430, 1992. [Cited on pages 11 and 27.]

[67] F. Ruskey and J. Sawada. An efficient algorithm for generating necklaces with fixed density. In *SODA '99: Proceedings of the tenth annual ACM-SIAM symposium on discrete algorithms*, pages 729–758, Baltimore, Maryland, United States, 1999. Society for Industrial and Applied Mathematics. [Cited on page 23.]

[68] F. Ruskey and J. Sawada. An efficient algorithm for generating necklaces with fixed density. *SIAM Journal of Computing*, 29(2):671–684, 1999. [Cited on page 23.]

[69] F. Ruskey and J. Sawada. A fast algorithm to generate unlabeled necklaces. In *SODA '00: Proceedings of the eleventh annual ACM-SIAM symposium on discrete algorithms*, pages 256–262, San Francisco, California, United States, 2000. Society for Industrial and Applied Mathematics. [Cited on page 23.]

[70] F. Ruskey and A. Williams. Generating combinations by prefix shifts. In *COCOON '05: Computing and Combinatorics, 11th Annual International Conference*, volume 3595 of *Lecture Notes in Computer Science*, pages 570–576, Kunming, China, 2005. Springer-Verlag. [Cited on pages 3 and 144.]

[71] F. Ruskey and A. Williams. An explicit universal cycle for the $(n-1)$-permutations of an n-set. *ACM Transactions on Algorithms*, (accepted), 2008. [Cited on pages 26, 153, and 185.]

[72] F. Ruskey and A. Williams. Generating balanced parentheses and binary trees by prefix shifts. In *CATS '08: Fourteenth Computing: The Australasian Theory Symposium*, volume 77 of *CRPIT*, Wollongong, Australia, 2008. ACS. [Cited on pages 140, 148, 149, 174, 176, and 182.]

[73] F. Ruskey and A. Williams. The coolest way to generate combinations. *Discrete Mathematics*, 309(17):5305–5320, September 2009. [Cited on pages 3, 140, 144, and 182.]

[74] C.Flye Sainte-Marie. Solution to question nr. 48. *L'intermédiaire des Mathématiciens*, 1:107–110, 1894. [Cited on page 184.]

[75] J. Sawada. A fast algorithm to generate necklaces with fixed-content. *Theoretical Computer Science*, 1-3(301):477–489, 2003. [Cited on pages 23 and 27.]

[76] J. Sawada. Personal communication. 2009. [Cited on page 171.]

[77] J. Sawada and J. Tsang. Personal communication. 2009. [Cited on pages 180 and 184.]

[78] R. Sawae, T. Sakata, M. Tei, K. Takarabe, and Y. Manmoto. Gray code and the initialization problem of NMR quantum computers. *International Journal of Quantum Chemistry*, 95:558–560, 2003. [Cited on page 9.]

[79] R. Sedgewick. Permutations generation methods. *ACM Comput. Surv.*, 9(2):137–164, 1977. [Cited on page 14.]

[80] M. Sekanina. On an ordering of the set of vertices of a connected graph. *Publ. Fac. Sci. Univ. Brno*, 413(2):137–142, 1960. [Cited on page 20.]

[81] Combinatorial Object Server. `http://www.theory.csc.uvic.ca/~cos`. [Cited on page 13.]

[82] J. Shallit. Personal communication. *Canadian Conference on Discrete and Algorithmic Mathematics (CanaDAM)*, 2009. [Cited on page 166.]

[83] A.C. Siepel. An algorithm to enumerate sorting reversals for signed permutations. *Journal of Computational Biology*, 10(3–4):575–597, 2003. [Cited on page 21.]

[84] N. J. A. Sloane. The on-line encyclopedia of integer sequences. `http://www.research.att.com/~njas/sequences/A009766`. [Cited on page 169.]

[85] G. Stachowiak. Hamilton paths in graphs of linear extensions for unions of posets. *SIAM J. Discrete Math.*, 5:199–206, 1992. [Cited on page 23.]

[86] R. Stanley. *Enumerative Combinatorics*. Cambridge University Press, 1997. [Cited on pages 58, 63, and 72.]

[87] H. Steinhaus. *One Hundred Problems in Elementary Mathematics*. Pergamon Press, 1963. Reprinted by Dover Publications in 1979. [Cited on page 13.]

[88] B. Stevens. Personal communication. 2007. [Cited on page 176.]

[89] T. Takaoka. An O(1) time algorithm for generating multiset permutations. In *ISAAC '99: Algorithms and Computation, 10th International Symposium*, volume 1741 of *Lecture Notes in Computer Science*, pages 237–246, Chennai, India, 1999. Springer. [Cited on pages 23 and 27.]

[90] H. Trotter. Algorithm 115: Perm. *Comm. ACM*, 5(8):434–435, August 1962. [Cited on page 13.]

[91] T. Ueda. Gray codes for necklaces. *Discrete Mathematics*, 219(1-3):235–248, 2000. [Cited on page 23.]

[92] V. Vajnovszki. A loopless algorithm for generating the permutations of a multiset. *Theoretical Computer Science*, 2(307):415–431, 2003. [Cited on pages 23 and 27.]

[93] V. Vajnovszki. More restrictive Gray codes for necklaces and Lyndon words. *Information Processing Letters*, 106(3):96–99, 2008. [Cited on page 23.]

[94] V. Vajnovszki and T. Walsh. A loop-free two-close Gray-code algorithm for listing k-ary Dyck words. *Journal of Discrete Algorithms*, 4(4):633–648, 2006. [Cited on pages 23 and 27.]

[95] V. Vajnovszki and M. Weston. Gray codes for necklaces and Lyndon words of arbitrary base. *Pure Mathematics and Applications/Algebra and Theoretical Computer Science*, 17(1-2):175–182, 2006. [Cited on page 23.]

[96] T.M.Y. Wang and C. Savage. A Gray code for necklaces of fixed density. *SIAM Journal on Discrete Mathematics*, 9(4):654–673, 1996. [Cited on page 23.]

[97] Wikipedia. Gray code. http://en.wikipedia.org/wiki/Gray_code. [Cited on page 9.]

[98] Wikipedia. Harvard Mark II. http://en.wikipedia.org/wiki/Harvard_Mark_II. [Cited on page 7.]

[99] Wikipedia. Interval graphs. http://en.wikipedia.org/wiki/Interval_graph. [Cited on page 64.]

[100] Wikipedia. Method ringing. http://en.wikipedia.org/wiki/Method_ringing. [Cited on page 14.]

[101] Wikipedia. Prayer wheel. http://en.wikipedia.org/wiki/Prayer_wheel. [Cited on page 16.]

[102] A. Williams. Loopless generation of multiset permutations using a constant number of variables by prefix shifts. In *SODA '09: The Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, New York, New York, USA, 2009. [Cited on pages 140 and 177.]

[103] R. Wu, J. Chang, and Y. Wang. Loopless generation of non-regular trees with a prescribed branching sequence. *The Computer Journal*, 2009. [Cited on page 69.]

[104] S. Zaks. A new algorithm for generation of permutations. *BIT Numerical Mathematics*, 24(2):196–204, 1984. [Cited on page 21.]