University of North Carolina at Chapel Hill

Honors Thesis

Some Context-free Languages and Their Associated Dynamical Systems

Author:
Landon James

Supervisor:
Dr. Karl Petersen

Approved:

April 10, 2014

1. Introduction

This paper focuses on certain context-free dynamical systems within the framework of symbolic dynamics and formal language theory. Our main results include using a block counting method to calculate the entropy of the Dyck languages, applying the Chomsky-Schützenberger theorem to the Łukasiewicz language, discovering the structure of winning strategies for a combinatorial game involving the Dyck languages, and showing how to construct positive entropy minimal subshifts whose winning strategies are worth studying. These main results are supplemented with an overview of some features of formal languages and symbolic dynamics.

We begin by introducing some notation that will be used throughout the paper and then review some properties of formal languages, specifically focusing on context-free languages and their canonical example, the Dyck languages. This is followed by a historical look at Donald Knuth's presentation of parenthesis languages [11], an early precursor of the Dyck languages. Then two context-free languages, the Dyck and Łukasiewicz languages, are defined and shown to be, in a sense, equivalent. We state the Chomsky-Schützenberger theorem, which shows that there is a strong connection between the Dyck languages and all other context-free languages, and show how this works for the Łukasiewicz example, following the proof by Dexter Kozen [12]. Several combinatorial properties of the Dyck languages are reviewed and their entropy is calculated, an endeavor motivated by a paper of Tom Meyerovitch [16]. We finish by looking at combinatorial games that can be played with subshifts as defined by Salo and Törmä [19]. We show that the winning shift for any Dyck language is the golden mean shift and raise the question of finding the winning strategies for positive entropy minimal subshifts. As a first step in this problem, we construct explicitly a positive entropy minimal subshift, following the exposition by Downarowicz [2].

2. Background and Notation

An *alphabet* is a set of symbols and generally denoted by Σ. The *Kleene closure* of an alphabet or language is the set of all finite strings over the alphabet, including the empty string ϵ, and is denoted Σ^*. A *language* over an alphabet is a subset of the Kleene closure of the alphabet, $L \subset \Sigma^*$. For a string w we will denote its length by $|w|$. The *entropy* of a language L is defined as $h(L) = \lim_{n\to\infty} \log(|L_n|)/n$, where L_n is the set of words of length n in the language. This measures the exponential growth rate of the number of words in the language. For a set of strings D and string w we define wD (or Dw) as the set of all elements in D with w left (or right) concatenated. Similarly if W is a set of strings, then WD (or DW) is the set of all elements of D with all elements of W left (or right) concatenated.

The Chomsky Hierarchy organizes languages by the computing ability of the formal machines required to generate a language or recognize its members. Every level in the hierarchy contains all levels below it. The levels in the hierarchy are

the following:

$$\text{regular} \subset \text{context-free} \subset \text{context-sensitive} \subset \text{recursively enumerable}.$$

In this paper we will be concerned mostly with context-free languages, but regular languages will also appear, so we give a short overview of related notation.

Regular languages are defined by regular expressions. If E is a regular expression, then it generates a language $L(E)$. Regular expressions and the languages that they generate can be defined recursively using concatenation, Kleene closure, and the operator + as follows [6].

Basis:

1) ϵ and $\varnothing$ are regular expressions, with $L(\epsilon) = \{\epsilon\}$ and $L(\varnothing) = \varnothing$
2) Any symbol a in the alphabet in question is a regular expression with $L(a) = \{a\}$
3) A variable E representing a regular language is a regular expression.

Induction: Assume that E and F are regular expressions. Then

1) EF is a regular expression, with $L(EF) = L(E)L(F)$
2) E^* is a regular expression, with $L(E^*) = (L(E))^*$
3) $E + F$ is a regular expression, with $L(E + F) = L(E) \cup L(F)$.

An expression for any regular language can be constructed from these building blocks.

Context-free languages are produced by *context-free grammars*, which are 4-tuples of the form $G = (V, \Sigma, P, S)$. In this formulation

- V is a set of *non-terminal symbols* (or *variables*)
- Σ is a set of *terminal symbols* known as the *alphabet*
- P is a finite set of relations $r : V \to (V \cup \Sigma)^*$ know as *production rules*
- $S \in V$ is the *start symbol* that begins all productions.

For brevity, we will sometimes lump together several productions with the same non-terminal on the left-hand side by separating the right-hand sides by bars. Thus the two productions $P \to Q$ and $P \to R$ become $P \to Q \mid R$. For $u, v \in (V \cup \Sigma)^*$, we say that u *yields* v or $u \to^* v$ if we can arrive at v from u by applying a finite number of production rules. We define the *language generated by the grammar* to be $L(G) = \{v \in \Sigma^* : S \to^* v\}$. Two grammars G and G' are said to be *equivalent* if $L(G) = L(G')$.

Sometimes it will be useful to require that our productions be of a certain form. If all productions p are of the form $p = A \to BC$ or $p = A \to a$ for some non-terminals A, B, C and terminals a, then we say that it is in *Chomsky normal form* (often abbreviated CNF). Every context-free grammar can be transformed to Chomsky normal form in a deterministic way while maintaing the same language [6]. Another useful form is called *Greibach Normal Form*. In this all productions are of the form $p = A \to aA_1A_2 \cdots A_n$ or $p = S \to \epsilon$ for non-terminals

$A, A_1, \cdots A_n$, terminals a, and start symbol S. Again every context-free grammar can be transformed into a grammar in Greibach Normal Form with the same associated language [5].

The set of context-free languages is closed under several set-theoretic operations. These operations are union, concatenation, Kleene closure, and intersection with a regular language [6]. The set of context-free languages is not, in general, closed under intersection and set complementation. Some of these closure properties are quite easy to see. Take two distinct context-free gramars $G_1 = (V_1, \Sigma_1, P_1, S_1)$ and $G_2 = (V_2, \Sigma_2, P_2, S_2)$. For union we create a new grammar $G_3 = (V_1 \cup V_2, \Sigma_1 \cup \Sigma_2, P_3, S_3)$, where $P_3 = P_1 \cup P_2 \cup \{S_3 \rightarrow S_1 \mid S_2\}$. Then $L(G_3) = L(G_1) \cup L(G_2)$. For concatenation we perform a similar construction, but with the new production rule $S_3 \rightarrow S_1 S_2$, giving $L(G_3) = L(G_1)L(G_2)$. To construct a grammar for the Kleene closure of $L(G_1)$ we change the grammar only by adding the new start symbol S_3 and the production $S_3 \rightarrow S_3 S_1$. The intersection of context-free and regular languages will be mentioned later.

For a given context-free grammar G consider a string $x \in L(G)$. Define P_x^* as the set of sequences of productions in G that lead from the start symbol S to x. If $\text{card}(P_x^*) = 1$ for all $x \in L(G)$, then we say that the grammar G is *unambiguous*. If for some x it is the case that $\text{card}(P_x^*) > 1$, then the grammar G is said to be *ambiguous*. If no grammar for a specific context-free language is unambiguous, we call that language *inherently ambiguous*. There do exist inherently ambiguous languages, such as the union of $\{a^n b^m c^m d^n \mid n, m > 0\}$ and $\{a^n b^n c^m d^m \mid n, m > 0\}$. It is proved in [6] that no word in the subset $\{a^n b^n c^n d^n \mid n > 0\}$ can have a unique sequence of productions leading to it from S.

It is interesting to note that the problem of determining whether a context-free language is ambiguous is algorithmically undecidable. This means that there are no algorithms which always lead to a correct yes-no answer for all instances of this problem. Given two context-free grammars G_1 and G_2 over the same alphabet, the problems of determining whether $L(G_1) = L(G_2)$, $L(G_1) \subset L(G_2)$, or $L(G_1) \cap L(G_2) = \varnothing$ are all also undecidable.

Each class of languages in the Chomsky Hierarchy has an associated class of abstract machines that produce or recognize its members. The machines related to regular languages are known as *finite-state automata* (sometimes simply called finite automata). A finite-state automaton is a finite set of states with a transition function. When presented at the designated start state with a string, the automaton will, based on the string and the transition function, tell you whether the string is in the language. More formally, a finite-state automaton is a 5-tuple $M = (Q, \Sigma, \delta, S, F)$ in which

- Q is a finite set of *machine states*
- Σ is the alphabet of the input strings
- δ is a mapping from $Q \times \Sigma \rightarrow Q$ known as the *transition function*
- $S \in Q$ is the *start state*

- $F \subseteq Q$ is the set of states in which the machine accepts.

Given a string $s = s_1 s_2 \dots s_n$, the automaton begins in state S and then transitions to the state $\delta(S, s_1) = S_1$. Repeating this for the rest of the characters, we say that a string has been *accepted* by the machine if $\delta(S_{n-1}, s_n) \in F$. That is, if the machine finishes in the set of accepting states then input string s is in the language defined by the automaton.

Machines in the class associated with context-free languages are known as *pushdown automata*. A pushdown automaton is a finite-state automaton that has access to a stack onto which it can push symbols that can be read later. The automaton transitions between states through a transition function that takes in the current state, the current input, and the symbol on top of the stack, then returns the new state and possibly manipulates the stack. More formally, a pushdown automaton can be defined as a 7-tuple $M = (Q, \Sigma, \Gamma, \delta, S, Z, F)$ in which

- Q is a finite set of *machine states*
- Σ is the alphabet of the input strings
- Γ is the set of symbols that can be placed on the stack
- δ is a mapping from $Q \times \Sigma \times \Gamma \to Q \times \Gamma^*$ known as the *transition function*
- $S \in Q$ is the *start state*
- $Z \in \Gamma$ is the symbol initially on the stack
- $F \subseteq Q$ is the set of states in which the machine accepts.

Given a string $s = s_1 s_2 \dots s_n$, the automaton begins in state S with symbol Z on the stack and then, if $\delta(S, s_1, Z) = (S_1, Z_1)$, the machine transitions to state S_1 and pushes symbol Z_1 onto the stack. Repeating this for the rest of the characters, we say that a string has been *accepted* by the machine if $\delta(S_{n-1}, s_n, Z_{n-1}) = (S_n, Z_n)$, and $S_n \in F$. That is, if the machine finishes in the set of accepting states then the input string s is in the language defined by the automaton. These pushdown automata represent an intermediate level of computational power above simple finite-state automata and below the well-known Turing Machines.

When confronted with a language we will sometimes want to test whether or not it is context-free or regular. Some useful tools for doing this are known as pumping lemmas.

Theorem 2.1 (**The Pumping Lemma for Regular Languages**). *Let L be a regular language. Then there exists an integer $p \geq 1$, known as the* pumping length*, such that every string $s \in L$ with $|s| \geq p$ may be written as $s = xyz$, with x, y, z obeying the following properties:*

1) $|y| \geq 1$
2) $|xy| \leq p$
3) $xy^n z \in L$ *for all* $n \geq 0$.

Proof. The proof is based on the structure of a finite automaton F that accepts the language. Set p equal to the number of states in this machine. Let $s \in L$

with $|s| \geq p$. Let q_0 be the start state of the machine. Let $q_1, \ldots q_p$ be the next p states visited as the string s is followed. Since F has only p states, at least one of these must be repeated. Call this state r. Let $q_i = q_j = r$. The substring of s corresponding to the path $q_{i+1}, \ldots, q_j$ will be called y. The sections of s preceding and following y will be called x and z, respectively. Since we can transition from state q_i directly to q_{j+1}, the string xy^0z is in the language. We could also repeat the cycle $q_i, \ldots, q_j$ as many times as we wish, so xy^nz is in the language for all n. If $i = 0$ (implying $x = \epsilon$), or $j = |s| - 1$ (implying $z = \epsilon$), then the justification is similar, but we begin or end with the cycle. □

Theorem 2.2 (**The Pumping Lemma for Context-Free Languages**). *Let L be a context-free language. Then there exists an integer $p \geq 1$ such that every string $s \in L$ with $|s| \geq p$ may be written as $s = uvxyz$, with u, v, x, y, z obeying the following properties:*

1) $|vxy| \leq p$
2) $|vy| \geq p$
3) $uv^nxy^nz \in L$ *for all* $n \geq 0$.

The use of the pumping lemmas is limited by giving only a necessary, but not sufficient, condition for a language being regular or context-free. For instance, we can use the pumping lemma for context-free languages to prove that a language is not context-free, but some non-context-free languages also meet the requirements. A further result, known as the *Myhill-Nerode Theorem*, fully characterizes the regular languages, while a result known as *Ogden's Lemma* strengths the context-free pumping lemma but still doesn't give a full characterization.

Given a language L and an alphabet Σ, the *extractive extension* $E(L) = \{w \in \Sigma^* : \exists z = xwy \in L$ such that $x, y \in \Sigma^*\}$ is the original language together with all substrings of words in the language. This property is useful when studying the subshifts associated with these languages, since we often want the language to be closed under a shift operation. Using the work from Kim Johnson's thesis [9], for a context-free language L, a grammar defining $E(L)$ can be constructed as follows, showing that the extractive extension of a context-free language is also context-free. Given a context-free grammar $G = (V, \Sigma, P, S)$ in CNF, define $E(G) = (V', \Sigma, P', S)$ with $V' = V \cup \{A_l, A_r, A' \mid A \in V\}$; and if P contains the productions

$A \to BC$ and $D \to t$, then P' contains the following:

$$\begin{aligned} A &\to B \mid C \mid B_r C_l \\ A_l &\to B_l \mid B' C_l \\ A_r &\to C_r \mid B_r C' \\ A' &\to B' C' \\ D &\to t \\ D_r &\to t \\ D_l &\to t \\ D' &\to t. \end{aligned}$$

3. Parenthesis Grammars

We begin our exploration of context-free languages by looking at the subset of the context-free grammars known as parenthesis grammars, focusing on results by Knuth [11]. Since the practical purpose of parentheses is to contain things, we consider grammars and languages containing parentheses and also other terminal symbols. The other terminal symbols can be thought of as the "real" symbols meant to be contained by the sets of parentheses. A *parenthesis grammar* is a context-free grammar of the form $G = (V, \Sigma, P, S)$ where Σ contains at least one pair of parentheses denoted α and β, and all productions are of the form $A \to \alpha\theta\beta$ where $\theta \in (V \setminus \{\alpha, \beta\})^*$ [11]. Knuth notes that grammars containing multiple pairs of parentheses can be dealt with in a similar fashion but omits the details. A language generated by one of these grammars is called a *parenthesis language*. The additional requirements on the structure of these languages allow us to overcome some of the difficulties faced when working with other context-free languages. Specifically, we will show that one can algorithmically test whether a given grammar produces a parenthesis language, and, if it does, we can generate a parenthesis grammar for that language.

We first introduce some helpful terminology and notation borrowed from Donald Knuth [11]. The functions *content* $c(\theta)$ and *deficiency* $d(\theta)$ are defined for any $\theta \in \Sigma^*$. They are defined first for symbols by:

$$c(x) = \begin{cases} 1, \text{ if } x = \alpha \\ 0, \text{ if } x \in \Sigma \setminus \{\alpha, \beta\} \\ -1, \text{ if } x = \beta \end{cases} \qquad d(x) = \begin{cases} 0, \text{ if } x = \alpha \\ 0, \text{ if } x \in \Sigma \setminus \{\alpha, \beta\} \\ 1, \text{ if } x = \beta \end{cases}$$

with $c(\epsilon) = d(\epsilon) = 0$, and then extended to strings by $c(\theta x) = c(\theta) + c(x)$, and $d(\theta x) = \max(d(\theta), d(x) - c(\theta))$. We say that a string θ is *balanced* if $c(\theta) = 0$.

The symbols x, y in the string $\alpha x \theta y \omega$ are called *associates* if θy is balanced. A language L is called *balanced* if all of its strings are balanced, and it is said to have *bounded associates* if there exists a constant m such that $\theta \in L$ and $x = \theta_i$ implies that x has no more than m associates. We say that a grammar is *completely qualified* if, for all non-terminals A, there exist numbers $c(A)$ and $d(A)$ such that

if $A \rightarrow^* \theta$ then $c(\theta) = c(A)$ and $d(\theta) = d(A)$. A completely qualified grammar in which $c(A) = d(A) = 0$ for all non-terminals A is called a *balanced grammar*.

To prove the main result of Knuth's paper [11], we will need several preliminary theorems and lemmas. These will be provided without proof, as they involve long, complicated constructions that are not needed to understand the main result.

Theorem 3.1 ([11], Th. 1)**.** *If $G = (V, \Sigma, P, S)$ is a context-free grammar, then there is an algorithm which determines whether or not $L(G)$ is balanced.*

Theorem 3.2 ([11], Th. 3)**.** *Let $G = (V, \Sigma, P, S)$ be a context-free grammar for which $L(G)$ is balanced and has bounded associates. Then it is possible to construct algorithmically an equivalent balanced grammar from G.*

Lemma 3.3 ([11], Lem. 3)**.** *Let $G = (V, \Sigma, P, S)$ be a parenthesis grammar. Then $L(G)$ is balanced and has bounded associates.*

Lemma 3.4 ([11], Lem. 5)**.** *Let $G = (V, \Sigma, P, S)$ be a context-free grammar for which $L(G)$ is balanced. It is possible to algorithmically construct a completely qualified grammar $G' = (V', \Sigma, P', S')$ such that $L(G') = L(G)$.*

Lemma 3.5 ([11], Lem. 6)**.** *Let $G = (V, \Sigma, P, S)$ be a balanced grammar for which $L(G)$ is balanced and has bounded associates. Then it is possible to construct algorithmically an equivalent parenthesis grammar G' from G.*

With these results in mind we will prove the main result of [11], Theorem 3.6.

Theorem 3.6 ([11], Th. 4)**.** *A context-free language is a parenthesis language if and only if it is balanced and has bounded associates. If $G = (V, \Sigma, P, S)$ is a context-free grammar, there is an algorithm which determines whether or not $L(G)$ is a parenthesis language; and if $L(G)$ is a parenthesis language, an equivalent parenthesis grammar $G' = (V', \Sigma, P', S)$ can be constructed effectively from G.*

Proof. We prove the if and only if statement first. Lemma 3.3 tells us that, for a parenthesis grammar G, $L(G)$ is balanced and has bounded associates. To prove the converse, note that if we are given a balanced language with bounded associates then we can use Theorem 3.2 and Lemma 3.5 to construct a parenthesis grammar that generates the same language as G.

Now we consider the decision problem of algorithmically determining whether or not $L(G)$ is a parenthesis language. Theorem 3.1 tells us that we can determine whether $L(G)$ is balanced. If it is, we may then continue and use Lemma 3.4 to construct a completely qualified grammar, G' such that $L(G') = L(G)$. At this point we still do not know whether $L(G)$ has bounded associates. We need to apply Theorem 3.2, which is valid only for languages that are balanced and have bounded associates. Luckily, the algorithm used in the proof of Theorem 3.2 fails if the language does not have bounded associates. So we apply Theorem 3.2 to G' and receive a new balanced grammar G'' or a failure. Applying Lemma 3.5 to G'' gives us a another grammar G''' which is a parenthesis grammar. Thus there

exists an algorithm that begins with a context-free grammar G and either returns an equivalent parenthesis grammar or fails and tells us that $L(G)$ is not a parenthesis language. □

We will now mention a theorem (without proof) and several corollaries about the set-theoretic properties of parenthesis languages. The operator Δ is defined as $A\Delta B = (A \setminus B) \cup (B \setminus A)$ and commonly known as the *exclusive or* operation.

Theorem 3.7 ([11], Th. 5)**.** *If L_1 and L_2 are parenthesis languages, then so is $L_1 \setminus L_2$, and it is possible to algorithmically construct a parenthesis grammar G such that $L(G) = L_1 \setminus L_2$.*

Corollary 3.8 ([11], Cor. 1)**.** *The set of parenthesis languages is closed under finite intersections.*

Proof. Recall that the set of context-free languages is closed under finite unions. So we have $L_1 \cap L_2 = L_1 \cup L_2 \setminus (((L_1 \cup L_2) \setminus L_1) \cup ((L_1 \cup L_2) \setminus L_2))$. □

Corollary 3.9 ([11], Cor. 2)**.** *If G_1 and G_2 are parenthesis grammars, then there is an algorithm to decide whether $L(G_1) = L(G_2)$.*

Proof. We can construct a parenthesis grammar for $L(G_1) \setminus L(G_2)$ by Theorem 3.7. It is easy to test algorithmically whether the language produced by this grammar is empty. □

Corollary 3.10 ([11], Cor. 3)**.** *The complement of a parenthesis language is context-free.*

Corollary 3.11 ([11], Cor. 4)**.** *If G_1 is any context-free grammar and G_2 is a parenthesis grammar, then there exists an algorithm to test whether $L(G_1) \subset L(G_2)$.*

Proof. Since $L(G_2)$ is a parenthesis language, by Theorem 3.6, it must be balanced and have bounded associates. Then any subset of $L(G_2)$ must also obey these rules. Again by Theorem 3.6, We can test $L(G_1)$ for these properties. If $L(G_1)$ is balanced and has bounded associates, it must be a parenthesis language. We can then test whether or not $L(G_1) \setminus L(G_2)$ is empty. If it is, then $L(G_1) \subset L(G_2)$. □

4. The Dyck Language and the Łukasiewicz Language

We will now consider another subset of the context-free languages known as the Dyck languages. The Dyck languages are a subset of the parenthesis languages, as will become apparent when they are defined. For $m \geq 1$ define the alphabet as $\Sigma_m = \{\alpha_i : 1 \leq i \leq m\} \cup \{\beta_i : 1 \leq i \leq m\}$ (with the subscript omitted when its value is made obvious by the context). Σ_m can be thought of as a collection of types of parentheses of various *levels*. In this interpretation, α_i's and β_i's, respectively, represent opening and closing parentheses of level i. We want the language we study over these letters to reflect this interpretation. Thus the Dyck language may be thought of as the set of all strings of parentheses that can be used to group a

mathematical expression in a valid way. With this definition in mind we begin our formalization of this language.

The *well-balanced m-Dyck language* is generated by the grammar

$$G = (\{S\}, \Sigma_m, P, S),$$

with the set of productions P defined as follows:

$$S \to SS \mid \alpha_i S \beta_i \mid \epsilon \ (1 \leq i \leq m).$$

This structure is easy to see intuitively, since in each production we either create a pair of parentheses, or place the seeds of two non-nested parenthetical structures beside each other (SS).

The preceding grammar can be converted to one in Chomsky normal form with an equivalent language using an algorithm given by Hopcroft and Ullman [6]. The algorithm gives the following productions, with new start symbol S_0, and adding new non-terminals $\{S_x : x \in \Sigma_m\}$:

$$\begin{aligned}
S_0 &\to S \mid \epsilon \\
S &\to SS \mid A_i S_{\beta_i} \mid A_i B_i \ (1 \leq i \leq m) \\
S_{\beta_i} &\to S B_i \ (1 \leq i \leq m) \\
A_i &\to \alpha_i \ (1 \leq i \leq m) \\
B_i &\to \beta_i \ (1 \leq i \leq m).
\end{aligned}$$

The production $S_0 \to S \mid \epsilon$ is not in standard Chomsky normal form, but this is allowed to keep ϵ in the language. Although this appears much more complicated than the original set of productions, its only significant change is splitting up the production $S \to \alpha_i S \beta_i$. Now, for example, we have the productions $S \to A_i S_{\beta_i}$ and $S_{\beta_i} \to S B_i$, which, after concurrent applications, produce the non-terminal string $A_i S B_i$. This split gives productions fulfilling the requirements of Chomsky normal form.

At certain points it will be helpful to view the Dyck languages in more algebraic terms. The following notation is borrowed from Meyerovitch [16].

A *monoid* is a set Σ with an associative binary operation and an identity element. Defining the associative binary operation $\cdot$ on Σ^* to be concatenation makes Σ^* into a monoid. In the following, taking something modulo the monoid means repeatedly scanning the string from left to right, and applying the given rules until there are no longer any rules that may be applied. For an initial string s and monoid M, the resulting string is denoted $s \pmod M$. Let M be the monoid over the set $\Sigma_m^* \cup \{0, 1\}$ generated by the following rules:

1) $\alpha_i \cdot \beta_i = \epsilon = 1 \pmod M, i = 1, \ldots, m$
2) $\alpha_i \cdot \beta_j = 0 \pmod M, i \neq j$
3) $1 \cdot 1 = 1 \pmod M$
4) $x \cdot 0 = 0 \pmod M$ for all $x \in M$.

The *m-Dyck language* is defined as $D_m = \{l \in \Sigma^* \mid \ l \not\equiv 0 \pmod{M}\}$. The *well-balanced m-Dyck language* is $D_m^w = \{l \in \Sigma^* \mid \ l \equiv 1 \pmod{M}\}$. We will speak of D_m more often, since it is the extractive extension of the well-balaned m-Dyck language and is therefore more suitable for study in dynamical systems. To see this note that any string in D_m can be extended, by concatenating parentheses on the left and right sides, to a well-balanced string. So every element of D_m is a substring of an element of the well balanced Dyck language D_m^w.

We can now define the one and two-sided subshifts associated with the Dyck language:

$$X_m^+ = \{y \in \Sigma^{\mathbb{N}} : y_r, \dots y_l \in D_m \text{ for all } 0 \le r \le l < \infty\}$$
$$X_m = \{x \in \Sigma^{\mathbb{Z}} : x_r, \dots x_l \in D_m \text{ for all } -\infty < r \le l < \infty\}.$$

We will also study a context-free language over the alphabet $\{a_0, a_1\}$ called the *Łukasiewicz language*, generally denoted $Ł = L((\{S\}, \{a_0, a_1\}, P, S))$, with the set of productions P as follows:

$$S \to a_0 S S \mid a_1.$$

This grammar can be converted to an equivalent one in Chomsky normal form, which has the following productions:

$$S \to A_0 U \mid a_1$$
$$U \to S S$$
$$A_0 \to a_0.$$

5. The Łukasiewicz Language is Equivalent to $D_1^w a_1$

Recall that the Łukasiewicz language Ł, over the alphabet $\{a_0, a_1\}$ and with start symbol S is defined by the productions $S \to a_0 S S \mid a_1$. The Dyck language D_1^w with one type of parentheses can be formed from the productions $S \to a_0 S a_1 S \mid \epsilon$, interpreting a_0 and a_1 as left and right parentheses, respectively. We will show that $Ł = D_1^w a_1$ by first giving inductive definitions of the languages and then showing that Ł and $D_1^w a_1$ satisfy the same inductive structure. These inductive definitions are really just ways of rewriting context-free grammars in set notation. Note that the results about parenthesis grammars from Knuth [11] can be applied here, namely since Ł and D_1^w are parenthesis grammars this equality could be established algorithmically.

To illustrate the idea of inductively defining a set we use the example of the Łukasiewicz language. The set associated with it can be defined as $Ł = a_0 ŁŁ \cup a_1$. The elements in this set are a_1 and anything that can be made by replacing the Ł's in $a_0 ŁŁ$ with elements already known to be in the set. The first element (besides a_1) that can be made in this way is $a_0 a_1 a_1$. This is followed by the three elements $a_0 a_0 a_1 a_1 a_0 a_1 a_1$, $a_0 a_0 a_1 a_1 a_1$, and $a_0 a_1 a_0 a_1 a_1$. These are made by replacing both

Ł's with $a_0a_1a_1$, replacing the first with $a_0a_1a_1$ and the second with a_1, and replacing the first with a_1 and the second with $a_0a_1a_1$ respectively. The set Ł consists of all words that can be constructed following this procedure. More precisely, one may define $Ł_1 = \{a_1\}$ and, for $n > 1$, $Ł_n = a_0Ł_{n-1}Ł_{n-1} \cup Ł_{n-1}$. Then $Ł = \cup_{n=1}^{\infty} Ł_n$.

Theorem 5.1. *Let D be defined by the recursive equation $D = a_0Da_1D \cup \{\epsilon\}$. Then a string is in D if and only if it can be formed by the productions $S \to a_0Sa_1S \mid \epsilon$, and therefore $D = D_1^w$, and $D_1^w = a_0D_1^wa_1D_1^w \cup \epsilon$.*

Proof. First the backwards direction. If d is a string formed from the starting symbol S, then it is either ϵ or of the form a_0xa_1y, where x and y are strings also formed from S. This means that d is a string of the same form as the strings in D and thus must be in D itself.

In the other direction, if $d \in D$, then we argue inductively on the length of d. If d is of length 0, then $d = \epsilon$ and it can be formed from S. Assuming that this is true for all strings of length less than or equal to $n-1$, we show that it is true for d with length n. $d \in D$ implies that $d = a_0xa_1y$ for $x, y \in D$. This implies that x and y have length less than n, and by our inductive hypothesis they can be formed from S. So d can be formed from S. □

We can give a similar inductive definition for the Łukasiewicz language.

Theorem 5.2. *Let L be the set defined by the recursive equation $L = a_0LL \cup a_1$. A string is in L if and only if it can be formed by the productions $S \to a_0SS \mid a_1$, and therefore $L = Ł$, and $Ł = a_0ŁŁ \cup a_1$.*

Proof. Similar to the proof for D and D_1^w. □

Theorem 5.3. $Ł = D_1^wa_1$.

Proof. Note that

$$\begin{aligned} D_1^w &= a_0D_1^wa_1D_1^w \cup \{\epsilon\} \text{ and} \\ D_1^wa_1 &= (a_0D_1^wa_1D_1^w \cup \{\epsilon\})\, a_1 \\ &= a_0D_1^wa_1D_1^wa_1 \cup \{a_1\}. \end{aligned}$$

This shows that Ł and $D_1^wa_1$ satisfy the same inductive definitions, and thus they contain the same words. This means that $Ł = D_1^wa_1$, because languages are identical exactly when they contain the same words. □

It is interesting to note that although Ł and D_1^w have similar languages, they differ in one important property. D_1^w is closed under concatenation; but since each element of Ł has an extra a_1 appended to the end, it is not closed under concatenation.

6. The Chomsky-Schützenberger Theorem Applied to the Łukasiewicz Language

The Chomsky-Schützenberger theorem shows that context-free languages have an important relationship with the Dyck languages. In fact, every context-free language is a sort of coding of the intersection of a Dyck language and a regular language. Recall that a Dyck language is over an alphabet Σ_m of symbols interpreted as opening and closing parentheses of various levels. The following definition is important in the discussion of this result.

Definition: Let L_1 and L_2 be languages. A *homomorphism* from L_1 to L_2 is an onto mapping $h : L_1 \to L_2$ such that $h(fg) = h(f)h(g)$ for all f and g in L_1.

Theorem 6.1 (**Chomsky-Schützenberger Theorem** [12])**.** *A language A over an alphabet Δ is context-free if and only if there exist $m \geq 0$, a regular language R, and a homomorphism $h : \Sigma_m^* \to \Delta^*$ such that $A = h(D_m \cap R)$.*

Note that we cannot in general say that the mapping h is an isomorphism. If a grammar G that generates a context-free language is ambiguous (contains multiple sequences of productions that can lead to the same final string), then there will be multiple strings in $D_n \cap R$ that map to the same string in $L(G)$. This will be made explicit in the construction that follows.

We illustrate the theorem with an example based on the Łukasiewicz language, Ł, over the alphabet $\Delta = \{a_0, a_1\}$. In the example we will find m, and construct R and h, so that Ł $= h(D_m \cap R)$. The following construction is based on a proof of the Chomsky-Schützenberger theorem by Dexter Kozen [12].

Let $G = (\{S, U, A_0\}, \Delta, P, S)$ be the above CNF-grammar generating Ł so that P consists of the following productions:

$$\begin{aligned} S &\to A_0 U \mid a_1 \\ U &\to SS \\ A_0 &\to a_0. \end{aligned}$$

Note that this grammar is indeed in Chomsky normal form, since every production is of the form $A \to BC$ or $A \to a$. We now define the sets $\Gamma = \{\alpha_{1p}, \beta_{1p}, \alpha_{2p}, \beta_{2p} : p \in P\}$, and $P' = \{p' : p \in P\}$ containing the productions

$$p' = \begin{cases} A \to \alpha_{1p} B \beta_{1p} \alpha_{2p} C \beta_{2p} \text{ if } p = A \to BC \\ A \to \alpha_{1p} \beta_{1p} \alpha_{2p} \beta_{2p} \text{ if } p = A \to a \end{cases}$$

Since there are two types of parentheses introduced for each production and $\text{card}(P) = 4$ for the Łukasiewicz language, Γ contains eight types of parentheses and $D_8 \subset \Gamma^*$. Let $G' = (\{S, U, A_0\}, \Gamma, P', S)$.

This construction is designed to allow us to encode sequences of productions from G. Our new language $L(G')$ does this by creating two pairs of parentheses for each production in G. Since the productions in G' introduce these pairs as terminal symbols at every step, the sequence of productions used to reach the final string is recorded in the string itself. Every production in G' is associated with a production

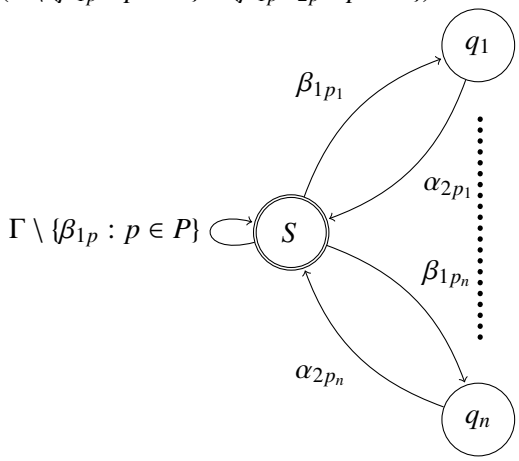
in G, so every string in $L(G')$ encodes a sequence of productions from G. Suppose that applying our homomorphism, $h : L(G') \to L(G)$, to an element $g' \in L(G')$ gives $h(g') = g$. If g' is reached by following the sequence of productions $s' \in P'^*$, then g is the string in $L(G)$ reached by following the sequence of productions $s \in P^*$ such that $s'_i = p'$ implies $s_i = p$.

The rules in P' make it obvious that $L(G') \subset D_8$, so, to apply the theorem in the case of this example we need to find a regular language that characterizes the properties of $L(G')$ that are not shared by all members of D_8. The following observations hold for all strings in $L(G')$:

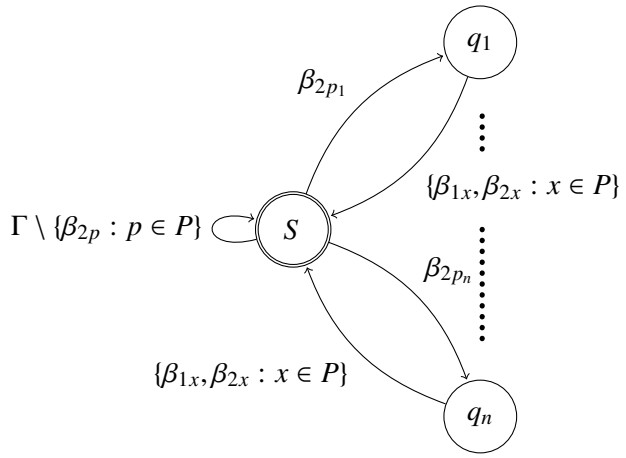
1) Every β_{1p} is immediately followed by a α_{2p}
2) No β_{2p} is immediately followed by a α_1 or α_2 of any type
3) If $p = A \to BC$, then every α_{1p} is immediately followed by α_{1q} for some $q \in P$ with left-hand side B, and every α_{2p} is followed by α_{1r} for some $r \in P$ with left-hand side C
4) If $p = A \to a$, then every α_{1p} is followed by a β_{1p} and every α_{2p} by a β_{2p}.

We will show that each of $1) - 4)$ can be described by a regular expression, and hence that $R = \{x \in \Gamma^* : \text{x satisfies } 1) - 4)\}$ is a regular language. We will give a regular expression and a deterministic finite automaton to formalize each of the above observations, $1) - 4)$. The intersection of these subsets will give the subset of Γ^* that obeys rules $1) - 4)$. Since the set of regular languages is closed under intersection [6], the language R will also be regular.

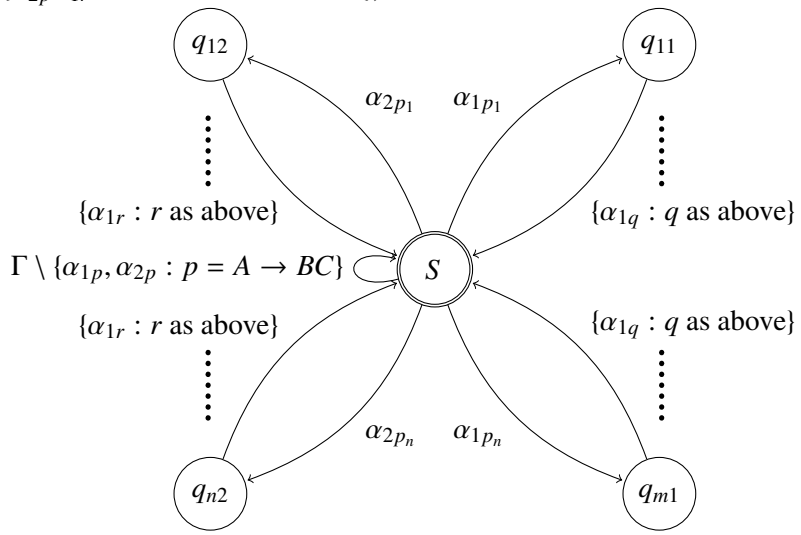
$R1)$ $(\Gamma \setminus \{\beta_{1p} : p \in P\} + \{\beta_{1p}\alpha_{2p} : p \in P\})^*$



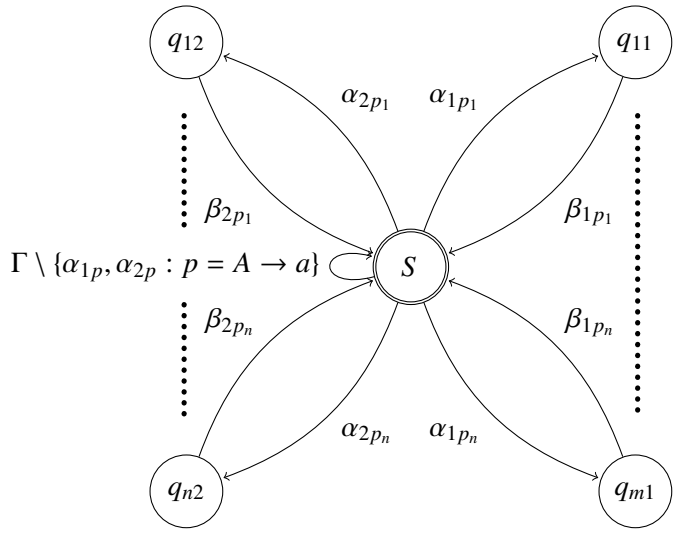
$R2)$ $(\Gamma \setminus \{\beta_{2p} : p \in P\} + \{\beta_{2p}\beta_{1x}, \beta_{2p}\beta_{2x} : x \in P\})^*$



*R*3) $(\Gamma \setminus \{\alpha_{1p}, \alpha_{2p} : p \text{ is of the form } p = A \to BC\} + \{\alpha_{1p}\alpha_{1q} : q \text{ has left hand side } B\} + \{\alpha_{2p}\alpha_{1r} : r \text{ has left hand side } C\})^*$



*R*4) $(\Gamma \setminus \{\alpha_{1p}, \alpha_{2p} : p \text{ is of the form } p = A \to a\} + \{\alpha_{1p}\beta_{1p}, \alpha_{2p}\beta_{2p} : p \text{ is of the form } p = A \to a\})^*$



We will analyze the regular expression $R1$ and its associated automaton to provide some insight into how the expressions 1) – 4) were constructed. The other automata can be explained similarly. Recall that $R1 = (\Gamma \setminus \{\beta_{1p} : p \in P\} + \{\beta_{1p}\alpha_{2p} : p \in P\})^*$. Thus $R1$ is the Kleene closure of the union of two languages, namely $\Gamma \setminus \{\beta_{1p} : p \in P\}$ and $\{\beta_{1p}\alpha_{2p} : p \in P\}$. The first set, $\Gamma \setminus \{\beta_{1p} : p \in P\}$, is everything in Γ less all symbols of the form β_{1p}. This is because observation 1) places restrictions only on what directly follows β_{1p}'s. The second set, $\{\beta_{1p}\alpha_{2p} : p \in P\}$, contains concatenated symbols of the form $\beta_{1p}\alpha_{2p}$. Since observation 1) states that every β_{1p} must be followed by a α_{2p}, this is allowed. Taken together the two sets $\Gamma \setminus \{\beta_{1p} : p \in P\}$ and $\{\beta_{1p}\alpha_{2p} : p \in P\}$ provide all available 1-blocks and all 2-blocks beginning with β_{1p} that obey rule 1). Taking the Kleene closure of the union of these sets gives all strings obeying observation 1).

The automaton for $R1$ is based on the structure of the regular expression. The automaton has the beginning and accepting state S. For each β_{1p_i} (of which we assume there are n) there is a corresponding state q_i. Beginning in the start state and on the first symbol in a given string, we follow the arrows in the graph based upon the next symbol encountered in our string. If we ever encounter a symbol for which there is no corresponding arrow from the current node, we know the string is not in our language. The same is true if our string is finite and we terminate in a non-accepting state, in this case any state other than S. Beginning at S, upon encountering any symbol in $\Gamma \setminus \{\beta_{1p} : p \in P\}$ we stay in S. When a β_{1p_i} is encountered, we leave S and move to q_i. From here we must either see a α_{2p_i} and return to S, or terminate.

Let $R = R1 \cap R2 \cap R3 \cap R4$ be the regular language defined by the intersection of the regular languages generated by rules 1)–4). Then $L(G') = D_8 \cap R$, since, as was previously mentioned, $L(G')$ is the subset of D_8 obeying the rules that define R. We can find an automaton for this intersection by means of the following construction. Let the automata for each of $R1 - R4$ be of the form $A_i = (Q_i, \Gamma, \delta_i, S_i, F_i)$ $(1 \leq i \leq$

4), with Q_i the set of states, Γ the alphabet, δ_i the transition function, S_i the start state, and F_i the set of accepting states. Then the automaton obeying all of these rules will be

$$A = (\prod_{i=1}^{4} Q_i, \Gamma, \delta = (\delta_1, \delta_2, \delta_3, \delta_4), (S_1, S_2, S_3, S_4), \prod_{i=1}^{4} F_i)$$

The states are the elements of the Cartesian product of the previous automata, the alphabet is unchanged, the transition function is a 4-tuple of the other transition functions, the start state is the 4-tuple of the previous start states, and the accepting states are the elements of the Cartesian product of the previous sets of accepting states. Thus this machine models running all of the other machines at once and will accept only when all of the other machines would have accepted.

We now need only to construct a homomorphism $h : \Gamma^* \to \Delta^*$ such that Ł = $h(D_8 \cap R)$. This function must take a word in $L(G')$, constructed by the string of productions $p'_1, p'_2, \cdots, p'_n$, and recover from it the word in $L(G)$ that would be produced by $p_1, p_2, \cdots, p_n$. Since only productions of the form $p = A \to a$ insert a terminal symbol into a string, only parentheses indexed by the corresponding p' should be mapped to symbols. Since the production p inserts one symbol and p' produces 4 parentheses, we need to map only one of these to the symbol a associated with p. We arbitrarily choose to map α_{1p} to a for every production of the form $p = A \to a$. Every other symbol (for productions of both forms) must be mapped to ϵ, the empty string. This gives us the following function:

$$h = \begin{cases} h(\alpha_{1p}) = h(\beta_{1p}) = h(\alpha_{2p}) = h(\beta_{2p}) = \epsilon \text{ if } p = A \to BC \\ h(\alpha_{1p}) = a, h(\beta_{1p}) = h(\alpha_{2p}) = h(\beta_{2p}) = \epsilon \text{ if } p = A \to a. \end{cases}$$

h is extended to full strings by defining that, for a string $s = s_1 \dots s_n$, $h(s) = h(s_1) \dots h(s_n)$.

The preceding construction shows that Ł, the Łukasiewicz language, is equal to $h(L(G')) = h(D_8 \cap R)$, the homomorphic image of the intersection of a Dyck language and a regular language.

The Chomsky-Schützenberger theorem can be used to provide crude upper bounds on the entropy of some languages. Since each language is the homomorphic image of a the intersection of a Dyck language and a regular language, the entropy of the Dyck language provides and upper bound.

7. Counting Words in Dyck Languages

In this section we investigate combinatorial properties of the Dyck languages. Specifically, for each n we want to know how many words of length n are in the language. This will allow us to compute the entropy and, through the Chomsky-Schützenberger theorem, will give us bounds on the entropies of other context-free languages. Krieger [13] and Krieger-Matsumoto [14] studied some regular languages related to the Dyck languages, Inoue and Krieger [7] found the entropy of certain subshifts of the Dyck languages, and Meyerovitch [16] found the entropy

for the well-balanced Dyck languages D_m^w, and we will show that the extractive extension D_m has the same entropy.

Fix $m \geq 1$ and consider the the m-Dyck language and the associated subshifts $X_m \subset \Sigma_m^{\mathbb{Z}}$ and $X_m^+ \subset \Sigma_m^{\mathbb{N}}$. We will count the number of strings of each length for several subsets of the full language and use them to calculate the entropy in the next section.

Balanced Strings: Balanced strings must be of even length since all parentheses must occur in pairs. We claim that there are

$$\binom{2k}{k}\frac{m^k}{k+1}$$

balanced strings of length $2k$. To see this, note that $C(2k,k)/(k+1)$ is the kth Catalan number and counts the total number of possible balanced strings of length $2k$ for one type of parentheses [1]. To move from a balanced string on one type of parentheses to a balanced string on m types of parentheses, note that for each pair of parentheses in a balanced string with one type of parentheses we can choose a type from among the m available in our language. If the string is of length $2k$, there will be k pairs of parentheses and thus m^k possible ways to assign the different types.

No Balanced Substrings: Let $NB_m(n)$ be the number of m-Dyck strings of length n with no balanced substrings. In a string with no balanced substrings all β's must precede all α's, since if any α comes before a β it is either a balanced substring or violates the monoid rule. So for a string of length n there will be m^n possible strings containing only β's, m^n possible strings with $(n-1)\beta$'s followed by one α, $\cdots$, m^n possible strings of all α's. Thus we can see that $NB_m(n) = (n+1)m^n$. We will leave out the subscript m when the context makes it obvious we are working with the m-Dyck Language.

One Balanced Substring: We count now the number of strings in D_m that have exactly one balanced substring. A balanced substring must be of even length, since all parentheses must occur in pairs. Let s be our string, ℓ be the length of our full string, and $n = 2k$ be the length of the substring we are considering. We have the following total number of possibilities with one balanced substring of length $2 \leq n \leq \ell$.

$$(\ell - n + 1)NB(\ell - n)\binom{2k}{k}\frac{m^k}{k+1}$$

We will explain each multiplicative term separately.

The first factor $(\ell - n + 1)$, denotes the number of substrings of length n in a string of length ℓ. To see this, consider the case for $n = 3$. If $s = s_1 s_2 \cdots s_\ell$, we can begin with the substring $s_1 s_2 s_3$. Continuing this pattern we get $s_2 s_3 s_4$, $s_3 s_4 s_5$, and so on until $s_{\ell-2} s_{\ell-1} s_\ell$. There are $\ell - 2$ values for the index of the first character, and this equals $\ell - 3 + 1$.

The second factor $NB(\ell - n)$ is the total number of strings of length $\ell - n$ with no balanced substrings. Lets say that our balanced substring occupies the spaces $s_m \cdots s_{m+n-1}$. The other two (possibly empty) sections of s, $s_1 \cdots s_{m-1}$ and $s_{m+n} \cdots s_\ell$, must form a string with no balanced substrings when concatenated. If either section contained a balanced string of its own then our assumption that our string only has one balanced substring would be violated. If they form a balanced substring when concatenated, then s_{m-1} and s_{m+n} must be a matching pair of parentheses. But this would violate our assumption that our balanced substring is of length n, since these would increase it to length $n + 2$.

The third factor is simply the number of possible balanced strings of length $n = 2k$, which we derived previously.

Continuing counting by restricting the number of balanced strings quickly leads to an increase in difficulty. Explicitly counting these could be an interesting combinatorial problem. But to find the entropy, detailed knowledge of these cardinalities is not necessary.

8. Entropies of Dyck Languages

Recall that the entropy of a language L is defined as $h(L) = \lim_{n\to\infty} \log |L_n|/n$, where L_n is the set of words of length n in the language. We will now calculate the entropies of the Dyck languages by making use of the counting arguments above.

Balanced Strings: As shown in the section on counting, there are

$$\binom{2k}{k}\frac{m^k}{k+1}$$

balanced strings of length $n = 2k$. The exponential growth rate of the number of balanced strings on m types of parentheses is

$$\lim_{k\to\infty} \frac{1}{2k} \log(\binom{2k}{k}\frac{m^k}{k+1}) = \frac{1}{2}\log m + \log 2.$$

This can be seen using Stirling's approximation in the form $\log(n!) = n\log(n) - n + O(\log n)$.

Extractive Extension: We claim that the extractive extension of a language has the same entropy as the original language. The only new words in the extractive extension are substrings of words already in the language. For an alphabet of size m, there are $N(n) \leq m^n$ words of length n in the language. Each of these words has $n(n+1)/2$ nonempty substrings, n of length one, $n-1$ of length two, on to one substring of length n. So for each n there are less than or equal to $N(n)n(n+1)/2$ distinct new strings in the extractive extension. So passing from the number, $N(n)$, of words of length n in the language to the number of words of length n in the extractive extension at most multiplies $N(n)$ by a polynomial in n, and this does not affect the exponential growth rate.

The language D_m with which we are primarily concerned is the extractive extension of the well-balanced Dyck language D_m^w. Taking the results in this section together tells us that

$$h(D_m) = \frac{1}{2}\log m + \log 2.$$

9. The Dyck Languages and Subshift Games

Consider a game for two players A and B. Given a finite alphabet Σ, a set of allowed words $X \subset \Sigma^*$, and some ordering of moves $a \in \{A, B\}^*$, A and B each place elements from the alphabet left to right, taking turns in the order designated. A attempts to keep the created word within the set of allowed words, while B tries to force the word out of the allowed set. We say that A has a *winning strategy for* a if, no matter what B plays in his turns, A can keep all of the words generated in X.

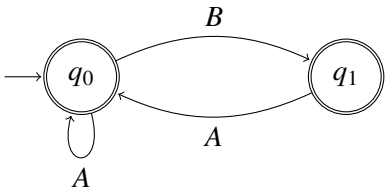
To formalize this we use the notation from Salo and Törmä [19] to talk about these games. A *word game* is a tuple (Σ, n, X), where Σ is the alphabet, $n \in \mathbb{N} \cup \{\mathbb{N}\}$, and $X \subset \Sigma^n$ is a set of allowed words. An *ordered word game* is a tuple (Σ, n, X, a), where (Σ, n, X) is a word game and $a \in \{A, B\}^n$. If X is a subshift and $n = \mathbb{N}$ we call our games *subshift games* and *ordered subshift games* respectively.

In an ordered game $G = (\Sigma, n, X, a)$, the players create a word $w \in S^n$ by having player a_i place one element of Σ at coordinate i of the new word. Player A wins if $w \in X$, otherwise player B wins. We define the *winning set* of X as

$$W(X) = \{a \in \{A, B\}^n : A \text{ has a winning strategy for } (\Sigma, n, X, a)\}.$$

These winning sets are *downwards closed*, meaning that for any string $w \in W(X)$ we can replace any B's in w with A's and the resulting string will still be in $W(X)$. There is, interestingly, no natural bijection between a language or subshift and its winning set. These sets don't even necessarily have the same cardinality. There are examples where the language of a subshift is uncountable, but its winning set is countable [19].

For the Dyck language we will consider the game $G = (\Sigma, \mathbb{N}, Y, a)$, where $\Sigma = \{\alpha_i \mid 1 \le i \le m\} \cup \{\beta_i \mid 1 \le i \le m\}$ and $X_m^+ = \{y \in \Sigma^{\mathbb{N}} \mid y_r, \dots y_l \in D_m \text{ for all } 0 \le r \le l < \infty\}$, as previously defined. We assert that the winning set of the Dyck language is the well-known Golden Mean shift with A identified with 0 and B identified with 1. This shift allows all two-blocks except BB to appear and can be represented by the regular expression $GM = (A + BA)^*(B + \epsilon)$, or by the finite automaton below.



Theorem 9.1. $W(X_m^+) = GM$.

Proof. First we show that $W(X_m^+) \subset GM$. If two B's appear consecutively in our playing order, then player B can place α_p followed by β_q for $p \neq q$ and $1 \leq p, q \leq m$. This removes the string from D_m and causes B to win the game. So our winning set must be a subset of GM, since two B's may not appear consecutively.

Now we show that $GM \subset W(X_m^+)$. To do this we provide a winning strategy for A given any set of moves $a \in GM$. Player A adds symbols in such a way that there are never any unmatched α's, meaning that reducing the so-far accumulated string by the monoid rules leaves no α's. To see that this strategy always results in a win, let turn $i \geq 1$ be player A's last turn before B plays, and let w be the string we are constructing. We place restrictions on $w_1 \dots w_i$, and thus player A's actions before B's turn. When $w_1 \dots w_i$ is reduced by our monoid rule there must be no α's (left parentheses) remaining. If there were any they would have to occur to the right of any β's (right parentheses), otherwise they would have already been removed by the monoid rule; or one would be immediately to the left of a β with which it did not match, putting the string out of the language. The rightmost of these α's, we will say it is an α_p, must occur at position w_i. So player B taking turn $i+1$ can place β_q $(p \neq q)$ at position w_{i+1} and force the string out of D_m. If there are no open α's, then player B can place an α or β of any level without violating our monoid rule, since only pairs of the type $\alpha_p \cdot \beta_q$ with $q \neq p$ are reduced to 0 modulo the monoid. So player A's strategy is to leave player B with a string containing no unpaired α's on his turn. □

We will now review some results about the relation between shifts and their associated winning shifts. These results are all taken from [19] with slight modifications.

Theorem 9.2 ([19], Prop. 3.4)**.** *Let Σ be an alphabet, and let $X \subset \Sigma^{\mathbb{N}}$ be a subshift. Then $W(X)$ is also a subshift, and $L(W(X)) = W(L(X))$.*

Theorem 9.3 ([19], Prop. 4.1)**.** *Let Σ be an alphabet and, let $X \subset \Sigma^{\mathbb{Y}}$ be a subshift. If $\mathbb{Y} = \mathbb{Z}$, then $W(X) = A^{\mathbb{Z}}$ if and only if X is the orbit closure of a periodic element of $\Sigma^{\mathbb{Z}}$. If $\mathbb{Y} = \mathbb{N}$, then $W(X) = A^{\mathbb{N}}$ if and only if* $\mathrm{card}(L_1(X)) = 1$*, i.e. X consists of a single constant sequence.*

Proof. First the case in which $\mathbb{Y} = \mathbb{Z}$. If X is periodic, then for every element $x \in X$ we know that x_i is uniquely determined by $x_{(-\infty, i-1]}$. So player B could win the game with a single move in any position. Thus we must have $W(X) = A^{\mathbb{Z}}$. Conversely, we assume that $W(X) = A^{\mathbb{Z}}$. In this case only player A is allowed to make moves, meaning that player B could win if allowed to make a single move. This means that for every $x \in X$, and every $i \in \mathbb{Z}$, x_i is uniquely determined by $x_{(-\infty, i-1]}$, which is equivalent to saying that X is periodic.

For the case in which $\mathbb{Y} = \mathbb{N}$, simply note that if $BA^{\mathbb{N}} \notin W(X)$, then only one character can appear at the first position of any word in X. Since X is shift invariant every character appearing in a word in X must appear as the first letter of some word in X. This means that X must contain a single constant sequence. □

In the following proposition the function $L^{-1}(Y)$ is defined as the subshift uniquely determined by the language Y, and we say that a subshift X is *Sturmian* if and only $\text{card}(L_n(X)) = n + 1$ for all $n \in \mathbb{N}$.

Theorem 9.4 ([19], Pr. 4.2)**.** *Let Σ be an alphabet and, let $X \subset \Sigma^{\mathbb{N}}$ be a subshift. Then $W(X) = L^{-1}(A^*BA^*) \subset \{A, B\}^{\mathbb{N}}$ if and only if X is Sturmian.*

Proof. In the backwards direction we prove the contrapositive. Note that the the condition $\text{card}(L_n(X)) = n + 1$ for all n is equivalent to saying that for all n there is exactly one $w \in L_n(X)$ with for which there are b, b' with $b \neq b'$ and $wb, wb' \in L_{n+1}(X)$. If $BA^kB \in L(W(X))$ for some k then, because player B can place any character they please, there exist $u, v \in L_k(X)$ such that $aub, aub', a'vb, a'vb' \in L_{k+2}(X)$ for some $a, a', b, b' \in \Sigma$. So we have that $\text{card}(L_n(X)) > n + 1$.

In the forward direction we also prove the contrapositive. If $w, w' \in L_k(X)$ $(w \neq w')$ with $wb, wb', w'b, w'b' \in L_{k+1}(X)$, then we can factor these as $w = ucv, w' = u'c'v$ (for $c, c' \in \Sigma$), implying that $cvb, cvb', c'vb, c'vb' \in L(X)$. Since the c, c' and b, b' are not equal and can be placed at either side of v, this means that player B could be allowed to make a move at either poshtion. Thus we have that

$$BA^{|v|}B \in W(L(X)) = L(W(X)).$$

□

Theorem 9.5 ([19], Prop. 5.4)**.** *For all $n \in \mathbb{N}$ and $L \subset \{0, 1\}^n$, we have* $\text{card}(L) = \text{card}(W(L))$.

Proof. The proof is by induction on n. The initial case with $n = 1$ is trivial. If $\text{card}(L) = 1$ then $W(L) = \{A\}$, and if $\text{card}(L) = 2$ then $W(L) = \{A, B\}$.

Now suppose that $n > 1$. For each $c \in \{0, 1\}$ define $L_c = \{w \in \{0, 1\}^{n-1} : cw \in L\}$. Then $L = 0L_0 \cup 1L_1$, and by our induction hypothesis we know that $\text{card}(W(L_c)) = \text{card}(L_c)$ holds for all $c \in \{0, 1\}$. We will now break the proof into two cases.

First, let $a \in W(L)$, and assume that $a_0 = A$. If $a_{[1,n-1]} \in W(L_c)$, then A has a winning strategy beginning with c that follows the strategy $a_{[1,n-1]}$. Conversely, if A has a strategy beginning with $c \in \{0, 1\}$, then $a_{[1,n-1]} \in W(L_c)$. Therefore

$$\begin{aligned} &\text{card}(\{a \in W(L) : a_0 = A\}) \\ &= \text{card}(W(L_0)) + \text{card}(W(L_1)) - \text{card}(W(L_0) \cap W(L_1)). \end{aligned}$$

Our second case assumes that $a_0 = B$. So $a_{[a_1,n-1]} \in W(L_0) \cap W(L_1)$ since player B could place any character in the first position. Thus we have

$$\text{card}(\{a \in W(L) : a_0 = B\}) = \text{card}(W(L_0) \cap W(L_1)).$$

Putting these results together with our induction hypothesis, we find that

$$\begin{aligned} &\operatorname{card}(W(L)) \\ &= \operatorname{card}(W(L_0)) + \operatorname{card}(W(L_1)) \\ &= \operatorname{card}(L_0) + \operatorname{card}(L_1) \\ &= \operatorname{card}(L). \end{aligned}$$

□

Corollary 9.6 ([19], Cor. 5.5)**.** *If X is a binary subshift, then $h(X) = h(W(X))$.*

Therefore every positive entropy subshift on a binary alphabet has a winning subshift with the same positive entropy. In the next section we describe a method for constructing highly repetitive subshifts with positive entropy. Because of the complicated nature of the sequences in such subshifts, we are unable to define the winning subshifts explicitly, but we know that they exist and have positive entropy.

10. Constructing Minimal Subshifts With Positive Entropy

The work in this section improves upon Example 4.9 from the Salo and Törmä paper, in which the authors construct a minimal subshift X for which $W(X)$ is uncountable. We construct a minimal subshift X for which $W(X)$ is not only uncountable but has positive entropy. Using a technique described by Downarowicz [2] and Susan Williams [22] that expands on work originally done by Jacobs-Keane [8] and Oxtoby [17], we construct a sequence w such that

1) every block in w appears in w infinitely many times and with bounded gaps between appearances
2) $\lim_{n\to\infty} \log N_n(w)/n > 0$.

Sequences with the first property are called *minimal* [18]. The orbit closure of any such sequence is a minimal topological dynamical system in the sense that it has no proper closed invariant sets (or equivalently every orbit is dense). Since X is binary and has positive entropy its winning shift will also have positive entropy [19, Cor. 5.5].

For this construction we use an auxiliary positive entropy subshift $Y \subset \{0,1\}^{\mathbb{N}}$ and require sequences of numbers $s = (s_i)$, $s' = (s'_i)$ and $q_i = s_i/s_{i-1}, q'_i = s'_i/s'_{i-1}$ satisfying $q'_i < q_i$ and $\lim_{i\to\infty} s'_i/s_i > 0$. We also introduce the numbers $r_i = q_i - q'_i$ and $t_i = s'_{i-1}r_i$ (with the convention that $s'_0 = 1$). The construction requires a sequence of blocks B_i such that $|B_i| = t_i$ and every word in $L(Y)$ appears as a prefix of some B_i. Sequences constructed in this way are known as *Toeplitz sequences*, and are *regularly almost periodic*, meaning that for each block, the set of places in the sequence at which the block appears contains an arithmetic progression. We construct a sequence of infinite strings $w^{(i)}$ on the alphabet $\{0, 1, \blacklozenge\}$ which converges to a string w on the alphabet $\{0, 1\}$. The construction begins with the string $w^{(0)} = \{\blacklozenge\}^{\mathbb{N}}$, then proceeds in steps.

Step i ($1 \le i \le \infty$)**:**

1) Divide $w^{(i-1)}$ into sections of length s_i, known as *basic blocks*.
2) Fill in the first t_i lozenges in each section with block B_i. The result is $w^{(i)} \in \{0, 1, \blacklozenge\}^{\mathbb{N}}$.

Given n, for sufficiently large i the initial n-block of $w^{(i)}$ contains no $\blacklozenge$'s, i.e. $w^{(i)}_{[1,n]} \in \{0,1\}^*$, and then $w^{(j)}_{[1,n]} = w^{(i)}_{[1,n]}$ for all $j \ge i$. Let $w = \lim_{i\to\infty} w^{(i)}$ and $X = \overline{O(w)}$.

We now describe a specific example in which $Y = \{0,1\}^{\mathbb{N}}$. The blocks B_i are the length t_i prefixes of the $(i-1)$st shift of the binary Champernowne sequence. This sequence, denoted C, is formed by concatenating all strings of each length in lexicographic order. The sequence C is known to be normal, and an initial segment of the sequence is $01000110110000010100 11\ldots$. In a more compact notation, $B_i = (\sigma^{i-1}(C))_{[0,t_{i-1}-1]}$. To see that the B_i's obey the requirement that every block of $L(Y)$ appears as the prefix of some B_i, consider any block $x \in L(Y)$. Every block, including x, appears infinitely often in C. Since B_i begins at position i of C, and the B_i are of increasing length, there must be one that begins at the same place as some appearance of x and is long enough to contain x.

The rest of our important sequences are defined as follows:

$$\begin{aligned} s_i &= \prod_{n=1}^{i}(n^2+1) \\ s'_i &= \prod_{n=1}^{i}(n^2) = (i!)^2 \\ r_i &= q_i - q'_i = (i^2+1) - i^2 = 1 \\ t_i &= s'_{i-1} r_i = s'_{i-1}. \end{aligned}$$

We see that these sequences obey the requirements, since $q_i = q'_i + 1$, $\prod_{i=1}^{\infty} i^2/(i^2+1)$ converges [21, Th. 1.41]. With Mathematica, we can estimate the value of the limit as $\lim_{i\to\infty} s'_i/s_i \approx 0.27 > 0$.

The first few values of each sequence used in the construction of the $w^{(i)}$'s are represented here for easy reference during the construction. $(s_i) = 2, 10, 100, \ldots$ $(s'_i) = 1, 4, 36, \ldots$ $(t_i) = 1, 1, 4, 36, \ldots$ $(B_i) = 0, 1, 0001, 00110..., \ldots$ The initial sequences of $w^{(i)}$ for $i = 0, \ldots, 4$ are shown below.

♦	♦	♦	♦	♦	♦	♦	♦	♦	♦	♦	♦	♦	♦	♦	♦	♦	♦	♦	♦	♦	♦	♦	♦
0	♦	0	♦	0	♦	0	♦	0	♦	0	♦	0	♦	0	♦	0	♦	0	♦	0	♦	0	♦	0
0	1	0	♦	0	♦	0	♦	0	♦	0	1	0	♦	0	♦	0	♦	0	♦	0	1	0	♦	0
0	1	0	0	0	0	0	0	0	1	0	1	0	♦	0	♦	0	♦	0	♦	0	1	0	♦	0
0	1	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0

This construction can be seen to lead to a minimal subshift due to its recurrent nature. If a block first appears in step i of the construction, then it reappears with

a gap no larger than s_i. That the subshift has positive topological entropy follows from the following proposition from Downarowicz [2]. For this proposition we will need the concept of the *density*, $d_i = (1 - s'_i/s_i)$, of the basic blocks in X. The density approximates the percentage of lozenges left in each i basic block after step i in the construction.

Theorem 10.1 ([2], Cor. 14.5)**.** *For the subshift X, constructed as above using the subshift Y, we have*

$$h_{\text{top}}(X, \sigma) = (\lim_{i\to\infty} d_i) h_{\text{top}}(Y, \sigma).$$

Proof. First note that since we fill in $t_i = s'_{i-1} r_i$ lozenges in each s_i-block at step i, s'_{i-1}/s_i is approximately the percentage of lozenges filled in each basic block after step i. So the density at each step, $d_i = (1 - s'_i/s_i)$ (really $(1 - s'_{i-1}/s_i)$ would be more appropriate, but this indexing is easier to work with and does not change the final outcome), is the approximate percentage of unfilled lozenges in each s_i-length basic block. The limit of d_i exists, since $d_i = (1 - s'_i/s_i)$, and we know that the limit of s'_i/s_i exists and is positive.

Immediately after step i, each basic block of length s_i is filled with characters and lozenges in exactly the same way. This leaves approximately $d_i s_i = (1 - s'_i/s_i)s_i = s_i - s'_i$ lozenges in each. Throughout the rest of the construction there are always basic blocks of length s_i with unfilled lozenges. If $N_{d_i s_i}(Y)$ is the number of blocks in $L(Y)$ of length $d_i s_i$ then there are $N_{d_i s_i}(Y)$ ways to fill the lozenges with these blocks. To see this, note that each $d_i s_i$ length block in Y appears in some B_m, since all blocks in Y appear as prefixes of some B_m. The B_m blocks fill in the remaining lozenges in the basic blocks, and so eventually place each of the $d_i s_i$ length blocks of Y into the lozenges of a basic block of length s_i.

The preceding implies that $N_{s_i}(X) = N_{d_i s_i}(Y)$, since after step i the lozenges in the basic blocks of length s_i are filled in $N_{d_i s_i}(Y)$ ways. Recall that we can find the entropy by looking at the number of blocks of each length along a subsequence of lengths, since $\lim_{n\to\infty} \log N_n(X)/n$ is known to exist (by Fekete's Lemma on subadditivity). Thus the topological entropy of X is calculated as follows:

$$\begin{aligned}
h_{\text{top}}(X, \sigma) &= \lim_{i\to\infty} (1/s_i) \log(N_{s_i}(X)) \\
&= \lim_{i\to\infty} (1/s_i) \log(N_{d_i s_i}(Y)) \\
&= \lim_{i\to\infty} (d_i/d_i)(1/s_i) \log(N_{d_i s_i}(Y)) \\
&= \lim_{i\to\infty} d_i \lim_{i\to\infty} (1/d_i s_i) \log(N_{d_i s_i}(Y)) \\
&= (\lim_{i\to\infty} d_i)\, h_{\text{top}}(Y, \sigma).
\end{aligned}$$

□

Here is a brief discussion of why we think the winning shift $W(X)$ of the subshift X constructed above is worth studying in detail. Recall that Salo and Törmä proved that small or minimal subshifts have small winning shifts: $W(X) = A^{\mathbb{Z}}$ if and only

if X is the orbit closure of a periodic element of $\Sigma^{\mathbb{Z}}$, and $W(X) = L^{-1}(A^*BA^*) \subset \{A, B\}^{\mathbb{N}}$ if and only if X is Sturmian. A minimal subshift that is the orbit closure of a Toeplitz sequence is, in a sense, the next step in this hierarchy. It allows somewhat more freedom than a Sturmian shift, but not nearly as much freedom as a shift like $\{0, 1\}^{\mathbb{N}}$, since each block appears with bounded gap and in fact along an arithmetic progression. Because of this we expect the winning shifts of such minimal binary subshifts to have fewer constraints than periodic and Sturmian subshifts, but more than the full shift in X, or shifts of finite type. For an X such as the one constructed above, the B's in elements of $W(X)$ would have to be fairly sparse due to the almost periodic nature of the sequences.

11. Conclusion

The work above has left some open questions. More work could be done on calculating the number of words in various subsets of the Dyck languages. Another interesting avenue of research would be investigating higher-dimensional analogues of context-free languages: so instead of languages consisting of one-dimensional arrays of characters, one considers n-dimensional arrays. Finding an analogue of the Dyck languages in this higher-dimensional setting is not trivial, since even in the two-dimensional case some non-arbitrary choices must be made, for example whether to add upwards and downwards facing parentheses and whether to require diagonals to follow the Dyck rules. Little is known about the relation between a subshift and its winning shift. For instance, is there some useful way to define a mapping between X and $W(X)$ for all X? Another interesting project would be to describe explicitly the winning subshifts associated with the minimal positive entropy subshifts constructed in the final section. This would require detailed understanding of the predictability or freedom of choice as words are formed in languages that are simultaneously highly structured and of positive entropy.

References

[1] Richard A. Brualdi, *Introductory Combinatorics*, 5th ed., Pearson Prentice Hall, Upper Saddle River, NJ, 2010. MR2655770 (2012a:05001)

[2] Tomasz Downarowicz, *Survey of odometers and Toeplitz flows*, Algebraic and Topological dynamics, Contemp. Math., vol. 385, Amer. Math. Soc., Providence, RI, 2005, pp. 7–37, DOI 10.1090/conm/385/07188, (to appear in print). MR2180227 (2006f:37009)

[3] Roland Fischer, *Sofic systems and graphs*, Monatsh. Math. **80** (1975), no. 3, 179–186. MR0407235 (53 #11018)

[4] Harry Furstenberg, *Disjointness in ergodic theory, minimal sets, and a problem in Diophantine approximation*, Math. Systems Theory **1** (1967), 1–49. MR0213508 (35 #4369)

[5] Sheila A. Greibach, *A new normal-form theorem for context-free phrase structure grammars.*, J. ACM **12**, no. 1, 42-52.

[6] John E. Hopcroft and Jeffrey D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley Publishing Co., Reading, Mass., 1979. Addison-Wesley Series in Computer Science. MR645539 (83j:68002)

[7] Kokoro Inoue and Wolfgang Krieger, *Excluding words from Dyck shifts*, ArXiv e-prints (2013), available at `1305.4720`.

[8] Konrad Jacobs and Michael Keane, $0-1$-*sequences of Toeplitz type*, Z. Wahrscheinlichkeitstheorie und Verw. Gebiete **13** (1969), 123–131. MR0255766 (41 #426)

[9] Kim Johnson, *Beta-shift Dynamical Systems and Their Associated Languages*, Ph.D. Thesis, University of North Carolina at Chapel Hill, 1999.

[10] J. Justesen, *Information rate and source coding of context-free languages*, Topics in information theory (Second Colloq., Keszthely, 1975), North-Holland, Amsterdam, 1977, pp. 357–368. Colloq. Math. Soc. János Bolyai, Vol. 16. MR0464723 (57 #4647)

[11] Donald Knuth, *A characterization of parenthesis languages*, Information and Control **11** (1967), 269–289.

[12] Dexter C. Kozen, *Automata and Computability*, Undergraduate Texts in Computer Science, Springer-Verlag, New York, 1997. MR1633052 (99j:68001)

[13] Wolfgang Krieger, *On the uniqueness of the equilibrium state*, Math. Systems Theory **8** (1974/75), no. 2, 97–104. MR0399412 (53 #3256)

[14] Wolfgang Krieger and Kengo Matsumoto, *Zeta functions and topological entropy of the Markov-Dyck shifts*, Münster J. Math. **4** (2011), 171–183. MR2869260 (2012j:37025)

[15] Werner Kuich, *On the entropy of context-free languages*, Information and Control **16** (1970), 173–200. MR0269447 (42 #4343)

[16] Tom Meyerovitch, *Tail invariant measures of the Dyck shift*, Israel J. Math. **163** (2008), 61–83, DOI 10.1007/s11856-008-0004-7. MR2391124 (2009a:37016)

[17] John C. Oxtoby, *Transitive points in a family of minimal sets*, Measure theory and its applications (Sherbrooke, Que., 1982), Lecture Notes in Math., vol. 1033, Springer, Berlin, 1983, pp. 258–262, DOI 10.1007/BFb0099862, (to appear in print). MR729540 (85h:28018)

[18] Karl Petersen, *Ergodic Theory*, Cambridge Studies in Advanced Mathematics, vol. 2, Cambridge University Press, Cambridge, 1989. Corrected reprint of the 1983 original. MR1073173 (92c:28010)

[19] Ville Salo and Ilkka Törmä, *Playing with subshifts*, Fundamenta Informaticae (2013).

[20] Ludwig Staiger, *The entropy of Łukasiewicz languages*, Theor. Inform. Appl. **39** (2005), no. 4, 621–639, DOI 10.1051/ita:2005032. MR2172142 (2006h:68100)

[21] E. C. Titchmarsh, *Theory of Functions*, Science Press, Peking, 1964 (English). MR0197687 (33 #5850)

[22] Susan Williams, *Toeplitz minimal flows which are not uniquely ergodic*, Z. Wahrsch. Verw. Gebiete **67** (1984), no. 1, 95–107, DOI 10.1007/BF00534085. MR756807 (86k:54062)