Language Theoretic Properties of Thompson's Group F

An Honors Project for the Department of Mathematics

By Sharif Munther Younes

Bowdoin College, 2013

Contents

1. Introduction

Thompson's group F was defined by Richard Thompson in the 1960's. Since then, group theorists have studied F for its unusual properties. For instance, F has subgroups that are isomorphic to $F \times F$ (imagine if $\mathbb{Z}$ had a subgroup isomorphic to $\mathbb{Z} \times \mathbb{Z}$!); and F was used to construct a finitely presented group with unsolvable word problem. These results suggest that F, though finitely presented, behaves much like an infinitely generated group [3].

Furthermore, F has many manifestations. Thompson's group F can be interpreted as:

(1) the set of associative laws for rearranging parenthesized expressions;
(2) a subset of homeomorphisms $f : I \to I$;
(3) pairs of finite, rooted binary trees; and
(4) a diagram group.

Mathematicians have used these different representations to help resolve questions in many fields, including algebra, logic, and topology. Thus, F has been widely studied both for its intriguing properties and for its connections to many areas of mathematics.

In the 1990's, work by Thurston, Cannon, Gilman, Epstein, and Holt led to the introduction of a class of groups called automatic groups [7]. Automatic groups enjoy many nice algorithmic and computational properties. The initial motivation was to understand the fundamental groups of compact 3-manifolds, since many topological questions about 3-manifolds reduced to computational problems in their fundamental groups. Later, many group theorists wondered if the class of automatic groups included other interesting groups. One of these was Thompson's group F. Thompson's group F possesses many of the properties that automatic groups do: it has quadratic Dehn function and word problem solvable in quadratic time. Moreover, Guba and Sapir showed in [9] that F satisfies one of the three criteria required for automaticity. Decades later, this tantalizing question remains open.

Not long after the introduction of automatic groups, it was shown that there exist compact 3-manifolds whose fundamental groups are not automatic. In addition, many classes of groups with nice algorithmic properties—for example, finitely generated nilpotent groups—fail to be automatic. This led group theorists to conclude that the class of automatic groups was not sufficiently wide. To this end, in [11] Kharlampovich, Khoussainov, and Miasnikov developed a natural generalization of automaticity called graph automaticity. Graph automaticity relaxes the conditions of automaticity by eliminating the dependence on generators. This generalization preserves many of the properties of automatic groups. Further, the class of graph automatic groups includes all automatic groups plus some groups that have a nice representation that does not depend on generators. One group in particular that Kharlampovich, Khoussainov, and Miasnikov hoped to capture was Thompson's group F.

The goal of this project was to investigate whether F is graph automatic since it has a very natural interpretation in terms of binary trees. We showed that this particular representation would not yield a graph automatic structure. So we defined the weakest generalization of graph automaticity that would capture F using this representation. In particular, Theorem 6.1 states that F is graph 2-context-free. Of

course, F might be graph automatic, or even automatic, but our work shows that group theorists will have to look to a less natural representation of F to prove this.

First and foremost, I would like to thank my advisor, Jennifer Taback, for her guidance and patience throughout this project. Even on sabbatical from a few hundred miles away, she has taught me more about math than everyone else I know combined. I would like to thank the Bowdoin Math, Economics, Philosophy, and Computer Science departments for providing the bulk of my education at Bowdoin. The courses I took in these departments greatly developed my analytical abilities and without them this research would have been impossible. I would like to thank Murray Elder, Robert Gilman, and Alexei Miasnikov for reviewing my work and giving me suggestions for further research. Finally, I would like to thank my high school teachers, Steve Weissburg and Severin Drix, for sparking my interest in math many years ago.

This work was supported by a grant from the National Science Foundation.

2. Languages and Automata

2.1. **Background.** In this section we introduce the theory of languages and automata. A formal language is defined with reference to a specific finite set of symbols, called an *alphabet*, which we denote Σ. A *string* w over Σ is a finite sequence of symbols $w = \alpha_1 \cdots \alpha_n$ for $\alpha_i \in \Sigma$. Then a *language* over Σ is any set of strings over Σ. Suppose we fix $\Sigma = \{0, 1\}$. Then consider two very natural languages over Σ:

$$L_1 = \{(01)^n : n \geq 0\}$$
$$L_2 = \{0^n 1^n : n \geq 1\}.$$

We want to know which language is more complex. To answer this question, we turn to automata theory. Roughly speaking, an automaton is a simple computer that reads a string as input and then outputs either "accept" or "reject." Then we can define a language in another way: the set of all strings accepted by a given automaton. So we want to construct the simplest possible automata N_1 and N_2 that accept L_1 and L_2, respectively. Then if N_2 has greater computational power than N_1, L_2 will be a more complex language than L_1.

There are four basic classes of automata. In order of increasing computational power: (1) finite state automata; (2) non-deterministic pushdown automata; (3) linear bounded automata; and (4) Turing machines. In this section, we define each of these objects and explore some of their most important properties.

2.2. **Finite State Automata.** Like languages, finite state automata (FSA) are defined with reference to a specific alphabet Σ. A finite state automaton consists of a finite set of states Q, one of which is the *start state* q_0. The FSA takes a string $w = \alpha_1 \cdots \alpha_n$ for $\alpha_i \in \Sigma$ as input. The FSA is equipped with a transition function $\delta : Q \times \Sigma \to Q$. If $\delta(q_i, \alpha_j) = q_k$, this means that if the FSA is in state q_i and reads the symbol α_j, it must transition to q_k. Finally, the FSA has a set of accept states $F \subseteq Q$ so that the FSA accepts the string w if and only if it is in an accept state after reading α_n. We denote the set of all strings—i.e., language—accepted by an FSA N as $\mathcal{L}(N)$.

Definition 2.1. We formally define an FSA by $N = (Q, \Sigma, \delta, q_0, F)$ where

(1) Q is a finite set of states;
(2) Σ is a finite input alphabet;
(3) δ is a function from $Q \times \Sigma$ to Q;
(4) $q_0 \in Q$ is the start state; and
(5) $F \subseteq Q$ is the set of accept sates.

We can visualize an FSA as a directed, labeled graph where each state is a vertex and there is an edge labeled α_j from $q_i \to q_k$ if and only if $\delta(q_i, \alpha_j) = q_k$. By convention, there is an arrow to the start state labeled "start" and the accept states are marked by a double circle.

Example 2.2. An example is shown in Figure 1. This FSA is defined by

$$N = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_0\})$$

where

$$\delta := \begin{cases} \delta(q_0, 0) &= q_1 \\ \delta(q_0, 1) &= q_2 \\ \delta(q_1, 0) &= q_2 \\ \delta(q_1, 1) &= q_0 \\ \delta(q_2, 0) &= q_2 \\ \delta(q_2, 1) &= q_2. \end{cases}$$

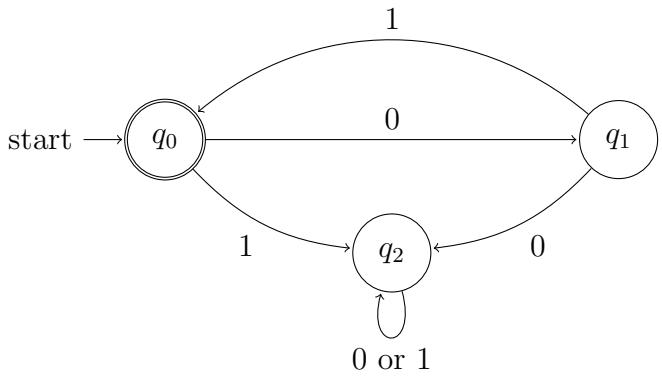

FIGURE 1. Graph representation of the FSA N.

The automaton N accepts the empty string along with all strings of alternating 0's and 1's that begin with a 0 and end with a 1. In fact, $\mathcal{L}(N) = \{(01)^n : n \geq 0\} = L_1$, the first language we defined in the introduction.

Note that if the FSA N in Figure 1 ever reaches state q_2, it will always reject the input string since q_2 is not an accept state and the only edges emanating from q_2 also terminate at q_2. A state from which there is no path to an accept state is called a *dead state*. For convenience, we omit dead states from the graph representation and assume that if there is no edge labeled α_j emanating from state q_i, then reading α_j while the FSA is in state q_i causes the FSA to reject the string. The simplified version of N is shown below in Figure 2.

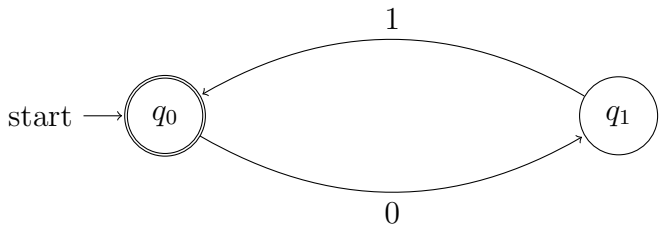

FIGURE 2. Simplified graph representation of the FSA N. Reading 1 in state q_0 or 0 in state q_1 causes N to reject the string automatically.

It is useful to consider another class of automata called *non-deterministic finite state automata* (NFSA). Non-deterministic finite state automata differ from FSA in two important ways. First, NFSA are allowed multiple transitions from a given state on the same input symbol. Second, NFSA are allowed ϵ-transitions where ϵ denotes

the empty string. This means that the NFSA can move between states by reading ϵ—i.e., without reading anything. Because of these two additional features there may be many sequences of transitions compatible with a given input string. An NFSA accepts a string if any one of these sequences leads to an accept state.

Definition 2.3. We formally define an NFSA N by $N = (Q, \Sigma, \delta, q_0, F)$ where

(1) Q is a finite set of states;
(2) Σ is a finite input alphabet;
(3) δ is a function from $Q \times (\Sigma \cup \{\epsilon\})$ to $\mathcal{P}(Q)$ (where $\mathcal{P}(Q)$ is the power set of Q);
(4) $q_0 \in Q$ is the start state; and
(5) $F \subseteq Q$ is the set of accept states.

The difference in the transition function captures the two differences noted above. Since the range of the transition function is $\mathcal{P}(Q)$, it can map a given state-symbol pair to any subset of Q and hence to multiple states. So in the graph of the NFSA, there can be multiple edges from a given state with the same label. Further, the domain of the transition function is $Q \times (\Sigma \cup \{\epsilon\})$. So for a given state q_i, the transition function now has a value for (q_i, ϵ). This means that in the graph of the NFSA, there can be edges labeled ϵ. Hence, the NFSA can transition between states without reading a symbol off the input string.

Example 2.4. Consider the following NFSA over $\Sigma = \{0, 1\}$. This automaton is non-deterministic because there are two edges labeled 1 emanating from state q_0, one that terminates at q_0 and one that terminates at q_1. This NFSA accepts any string that ends with a 1.

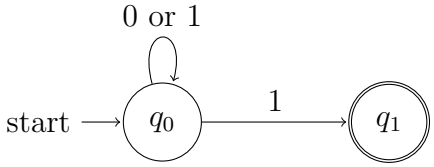

FIGURE 3. Reduced graph for an NFSA that accepts strings ending with a 1.

With any non-deterministic transition, the NFSA has a choice for which state to enter next, so we can imagine that the NFSA is in multiple states at a time. With each non-deterministic transition, more states are added. After the entire string has been read, if any one of these states is an accept state, the NFSA accepts the string. We can see this increase in states schematically in Figure 4 with an accepting run of the string 01.

The NFSA starts in state q_0, reads 0, and stays in state q_0. Then it reads 1. At this point, it can either stay in state q_0 or transition to state q_1. If it stays in q_0, the entire string has been read and the NFSA is not in an accept state. If it transitions to q_1, then the entire string has been read and it is in an accept state. So there is a sequence of transitions compatible with the input string that leads to an accept state and the NFSA accepts the string 01.

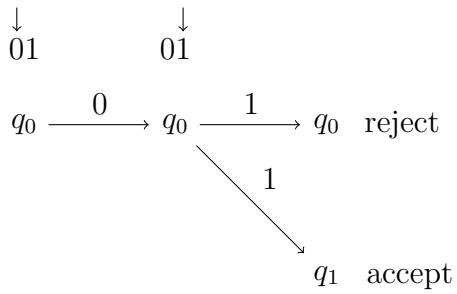

FIGURE 4. Increase in states of an NFSA.

It is clear that an FSA is just a particular case of an NFSA where any given state-symbol pair is mapped to a single state and for any state $q_i \in Q$, $\delta(q_i, \epsilon) = q_i$. Although NFSA appear more powerful than FSA, the following theorem states that they are in fact equivalent. The theorem is standard, so we will only include a sketch of the proof.

Theorem 2.5 (Theorem 2.1 of [10])**.** *Let L be the language accepted by an NFSA M_1. Then there is an FSA M_2 that accepts L.*

Sketch of proof. Let $M_1 = (Q, \Sigma, \delta, q_0, F)$ be an NFSA that accepts L. We want to construct an FSA $M_2 = (Q', \Sigma, \delta', q_0', F')$ that accepts L. The basic idea is that since M_1 can be in multiple states at the same time, M_2 has a state for each subset of states of M_1. So $Q' = \mathcal{P}(Q) - \{\emptyset\}$. Then F' is the set of all states in Q' that contain a state in F. Finally, we let

$$\delta'(\{q_1, \ldots, q_j\}, \alpha) = \{p_1, \ldots, p_k\}$$

if and only if

$$\{\delta(q_1, \alpha), \ldots, \delta(q_j, \alpha)\} = \{p_1, \ldots p_k\}.$$

So δ' is applied to a state $\{q_1, \ldots, q_j\} \in Q'$ by applying δ to each $q_i \in \{q_1, \ldots q_j\}$ individually. □

Example 2.6. For example, the FSA in Figure 5 has been constructed according to the procedure in Theorem 2.5. It accepts the same set of strings as the NFSA in Figure 3: i.e., strings ending with a 1. Note that state $\{q_1\}$ is unreachable and could be removed.

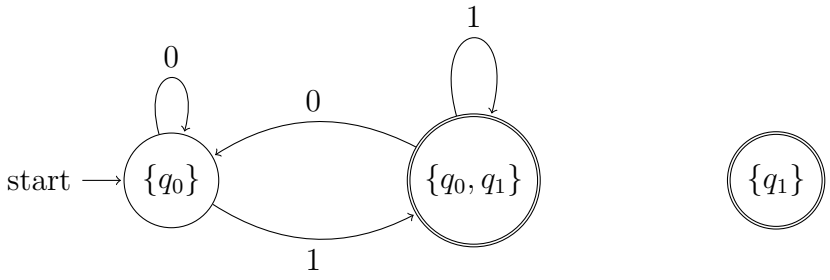

FIGURE 5. FSA that accepts strings ending with a 1.

In this example, the reduced FSA would have as many states as the equivalent NFSA. This is of course not always the case. In general, converting an NFSA to an FSA results in an exponential increase in the number of states since

$$|P(Q) - \{\emptyset\}| = 2^{|Q|} - 1.$$

This is why it is often much more convenient to work with NFSA.

We can now define the simplest class of languages that we will consider—those corresponding to the finite state automata.

Definition 2.7. A language is called *regular* if there is an FSA (or, equivalently, an NFSA) that accepts it.

Remark 2.8. Since L_1 is accepted by the FSA given in Example 2.2, L_1 is a regular language.

Regular languages are the simplest type of language that we will consider. We now introduce some terminology and then prove several properties of regular languages. Given a string $w = \alpha_1 \cdots \alpha_n$, we denote the length of w by $|w| = n$. We let Σ^* denote the set of all strings over Σ. A string that is part of a language is a *word* in the language. If L_1 and L_2 are two languages over Σ, then their *concatenation* L_1L_2 is the set of strings $\{uv : u \in L_1 \text{ and } v \in L_2\}$. Finally, the *Kleene closure* of L, denoted L^*, consists of all strings (including the empty string) formed by concatenating any number of words from L.

The following theorems are standard. For brevity, we will only include a sketch of Theorem 2.9.

Theorem 2.9 (Theorem 3.1 of [10])**.** *The set of regular languages is closed under union, concatenation, and Kleene closure. That is, if L_1 and L_2 are regular languages, so are $L_1 \cup L_2, L_1L_2$, and L_1^*.*

Sketch of proof. Let N_1 and N_2 denote the FSA that accept L_1 and L_2, respectively.

Union. To construct an NFSA that accepts $L_1 \cup L_2$, we simply identify the start states of N_1 and N_2.

Concatenation. To construct an NFSA that accepts L_1L_2, we can just add a copy of N_2 for each accept state of N_1, identifying the accept state of N_1 with the start state of the copy of N_2.

Kleene Closure. To construct an NFSA that accepts L_1^*, we can add an ϵ-transition from each accept state of N_1 to its start state. Further, since the NFSA must accept ϵ, we create an additional accept state q' with an arrow from q_0 to q' labeled ϵ. □

Theorem 2.10 (Theorem 3.2 of [10])**.** *The set of regular languages is closed under complementation. That is, if L is a regular language over Σ, then so is $\Sigma^* - L$.*

Proof. Since L is a regular language, there is some FSA $M = (Q, \Sigma, \delta, q_0, F)$ such that $\mathcal{L}(M) = L$. Then the FSA $N = (Q, \Sigma, \delta, q_0, Q - F)$ accepts every string that M rejects and rejects every string that M accepts. Thus, $\mathcal{L}(N) = \Sigma^* - L$, implying that $\Sigma^* - L$ is regular, as desired. □

Theorem 2.11 (Theorem 3.3 of [10])**.** *The set of regular languages is closed under intersection. That is, if L_1 and L_2 are regular languages over Σ, then so is $L_1 \cap L_2$.*

Proof. Since L_1 and L_2 are regular languages, it follows from Theorems 2.9 and 2.10 that

$$(L_1^c \cup L_2^c)^c = L_1 \cap L_2$$

is also a regular language. □

These theorems provide useful shortcuts for showing that a given language is regular: we can now just show that the language is the Kleene closure or complement of a regular language or the concatenation, union, or intersection of two regular languages.

Example 2.12. The set of all strings that do not have two consecutive zeros is a regular language over $\Sigma = \{0, 1\}$. To show this, we must only show that the set of all strings that do contain two consecutive zeros is a regular language. Well, the NFSA in Figure 6 clearly accepts all and only strings that contain two consecutive zeros.

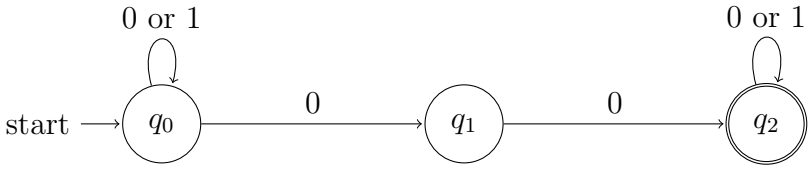

FIGURE 6. NFSA that accepts strings containing two consecutive 0's.

Now suppose that we want to show that a given language is *not* regular. Consider, for instance, $L_2 = \{0^n 1^n : n \geq 1\}$. First we try to construct an NFSA N_2 such that $\mathcal{L}(N_2) = L_2$. Well, N_2 would have to accept 01, so we might start with the graph in Figure 7.

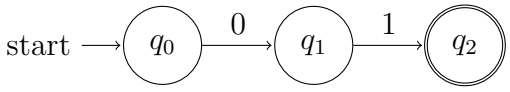

FIGURE 7. First few states in our construction of N_2.

But N_2 also has to accept $0011, 000111$, and so on. So we must add the following states and transitions in our construction of N_2, as in Figure 8.

Immediately we see that this method of constructing N_2 will lead to an infinite number of states, and thus N_2 will not be an NFSA. So the obvious method for constructing an NFSA to accept L_2 does not work. Of course this is not a proof that L_2 is not regular. The following lemma provides a technique to prove that a language is not regular.

Lemma 2.13 (Pumping Lemma, Lemma 3.1 of [10])**.** *For any regular language L, there exists a constant p (called a pumping constant) such that for any word $w \in L$ with $|w| \geq p$, we can write $w = xyz$ so that*

(1) $|xy| \leq p$,
(2) $|y| \geq 1$, and
(3) $xy^n z \in L$ for any $n \geq 0$.

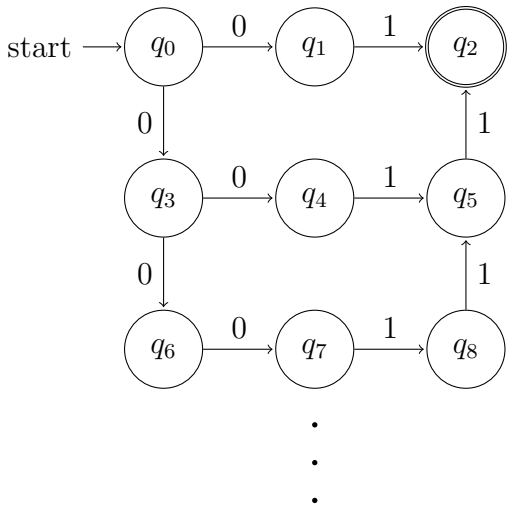

FIGURE 8. More states in our construction of N_2.

Proof. Since L is a regular language, there is an FSA that accepts L. Let N denote the FSA with the fewest number of states that accepts L. Then we let p be the number of states in N. Now consider any word $w \in L$ with $|w| \geq p$. So $w = \alpha_1 \cdots \alpha_n$ for $n \geq p$. Then consider the successive states N visits as it reads w, say, $q_0, q_1, \ldots, q_n$. Since $n \geq p$ and N has only p states, it cannot be the case that all $n+1$ states N visits are distinct. In particular, there must be some i, j with $0 \leq i < j \leq p$ and $q_i = q_j$. So the accepting run of w has the form depicted in Figure 9.

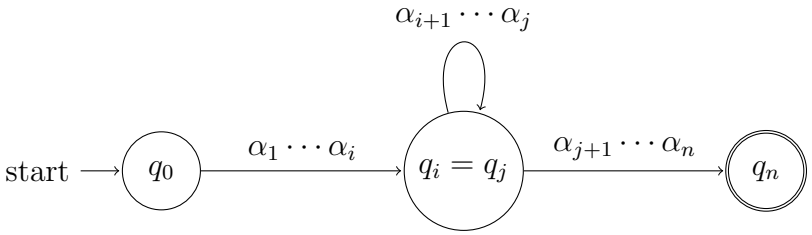

FIGURE 9. Form of accepting run of w. Note that this does not accurately depict the FSA.

Then let $x = \alpha_1 \cdots \alpha_i$, $y = \alpha_{i+1} \cdots \alpha_j$, and $z = \alpha_{j+1} \cdots \alpha_n$. Well, $|xy| \leq p$ since $|xy| = j \leq p$. Further, $|y| \geq 1$ since $|y| = j - i$ and $j > i$. Finally, since $q_i = q_j$, in the graph representing the FSA we can trace a loop in the path beginning and ending at $q_i = q_j$ as many or as few times as we like. Then reading the suffix $\alpha_{j+1} \cdots \alpha_n$ will still lead to the accept state q_n. Hence, $xy^n z$ is a word in L for any $n \geq 0$. This completes the proof. □

Now we have the tools to prove that L_2 is not a regular language, as we had suspected.

Example 2.14. $L_2 = \{0^n 1^n : n \geq 1\}$ is not a regular language

Proof. Suppose for contradiction that L_2 is regular. Then the Pumping Lemma guarantees the existence of a pumping constant p. But consider the word $w = 0^p 1^p \in L$.

Clearly $|w| \geq p$. So by the pumping lemma we can write $w = xyz$ so that $|xy| \leq p$, $|y| \geq 1$, and for all $i \geq 0$, xy^iz is a word in L. Since $|xy| \leq p$, there can be no 1's in xy. So y consists only of 0's. Since $|y| \geq 1$, xy^2z has more 0's than 1's, which contradicts the Pumping Lemma. Hence, as desired, $L_2 = \{0^n1^n : n \geq 1\}$ is not regular. □

In this section we have described two equivalent types of automata, finite state automata and non-deterministic finite state automata, and examined the class of languages that they accept, the regular languages. Of the two languages introduced at the beginning of the discussion, it was shown that L_1 is regular but L_2 is not, implying that L_2 is more complex than L_1. But there is no FSA that accepts L_2, which seems to be a very simple language. This motivates the concept of a pushdown automaton.

2.3. **Non-deterministic Pushdown Automata.** An NFSA cannot "count" or "remember" an arbitrary number of 0's and thus no NFSA can accept L_2. *Non-determinstic pushdown automata* (NPDA), however, can "count" and "remember" arbitrary numbers. NPDA differ from NFSA in that they are further equipped with a *stack* and *stack alphabet* Γ. A stack is a first-in, last-out data structure that stores symbols from Γ. The NPDA's transition function can now manipulate the stack: it reads the top symbol and can add symbols to the top of the stack (*push* symbols) or remove the top symbol (*pop* a symbol). An NPDA can be one of two types, depending on how it accepts strings.

Definition 2.15. An NPDA accepts by *empty stack* if it accepts a string when after reading α_n its stack is empty. An NPDA accepts by *final state* if it accepts a string when after reading α_n it is in an accept state.

It is a standard result that the class of NPDA that accept by empty stack are equivalent to the class of NPDA that accept by final state. For an NPDA that accepts by empty stack, it is customary that the set of accept states F equal the empty set.

Example 2.16. Before formally defining an NPDA, consider the example NPDA N_2 in Figure 10 that accepts $L_2 = \{0^n1^n : n \geq 1\}$. N_2 accepts by final state. Here the stack alphabet is $\Gamma = \{Z, x\}$.

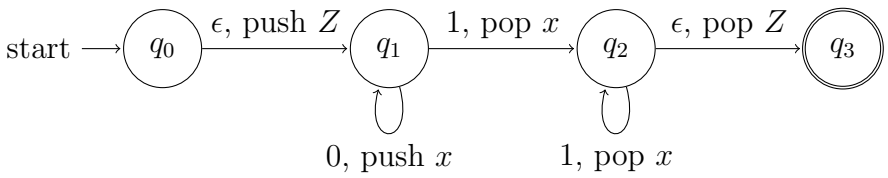

FIGURE 10. N_2, an NPDA that accepts L_2.

Consider some string $w = 0^n1^n$. N_2 starts in state q_0. N_2 must non-determinstically make an ϵ-transition to state q_1, pushing a Z. It then reads n 0's, pushing an x each time. So after reading all the 0's, the stack contains Zx^n. Then N_2 reads a 1, pops an x, and transitions to q_2; it now has stack Zx^{n-1}. It then reads $n-1$ 1's, popping an x each time, and staying in state q_2. After reading each 1, it is in state q_2 and has

stack Z. It can then make an ϵ-transition to state q_3 by popping the Z. Since q_3 is an accept state, N_2 accepts w. So N_2 accepts all strings of the form $w = 0^n1^n$.

Now suppose that N_2 accepts a string w. The only way to end up at accept state q_3 is to make an ϵ-transition from q_2 to q_3 by popping Z. If there were any symbols of w left to read, N_2 would then reject w. So after reading all of w, N_2 must be in state q_2 with stack Z. To get to q_2, N_2 must have read n 1's for $n \geq 1$. But each time it read a 1, it must have popped an x. So there must have been at least n x's on the stack. Further, after reading all the 1's, the stack only contained a Z, so there must have been exactly n x's on the stack. But the only way to add an x to the stack is reading a 0. So N_2 must have read n 0's. Hence, w has the form $w = 0^n1^n$. So N_2 accepts only strings of the form $w = 0^n1^n$.

So N_2 accepts all and only strings of the form $w = 0^n1^n$ and, as desired, $\mathcal{L}(N_2) = L_2$.

Definition 2.17. We formally define an NPDA N by $N = \{Q, \Sigma, \Gamma, \delta, q_0, F)$ where

(1) Q is a finite set of states;
(2) Σ is a finite input alphabet;
(3) Γ is a finite stack alphabet;
(4) δ is a function from $Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma$ to finite subsets of $Q \times \Gamma^*$;
(5) $q_0 \in Q$ is the start state; and
(6) $F \subseteq Q$ is the set of accept states.

The transition function δ maps $Q \times (\Sigma \cup \epsilon) \times \Gamma$ to finite subsets of $Q \times \Gamma^*$. We now discuss the interpretation of

$$\delta(q, \alpha, Z) = \{(p_1, \gamma_1), (p_2, \gamma_2), \ldots, (p_n, \gamma_n)\}$$

where $q, p_i \in Q$, $\alpha \in (\Sigma \cup \{\epsilon\})$, $Z \in \Gamma$ and each $\gamma_i \in \Gamma^*$. This means that if the NPDA is in state q with Z on top of the stack and reads α from the input string, it can, for any i between 1 and n, move to state p_i and replace the symbol Z with the string γ_i. So we can interpret "push x" as replacing Z with Zx (or ϵ with x) and "pop Z" as replacing Z with ϵ. An NPDA cannot make a transition that requires "pop Z" unless Z is on top of the stack. Note that the NPDA depicted in Figure 10 has been reduced significantly by elimination of dead states.

Definition 2.18. A language is called *context-free* if there is an NPDA that accepts it.

Definition 2.19. A *deterministic* PDA is identical to an NPDA except the transition function δ maps $Q \times \Sigma \times \Gamma$ to elements of $Q \times \Gamma^*$ rather than finite subsets of $Q \times \Gamma^*$.

Remark 2.20. Theorem 2.5 states that NFSA and FSA are equivalent. However, NPDA are strictly more powerful than deterministic PDA; that is, deterministic PDA accept only a proper subset of the context-free languages, called the deterministic context-free languages [10].

We now state several properties of the context-free languages. The first is the context-free equivalent of Lemma 2.13, the pumping lemma for regular languages. Then we present some closure properties of the context-free languages.

As with regular languages, it is difficult to prove that a given language is not context-free. For example, consider the language $L = \{0^n 1^n 2^n : n \geq 1\}$. In constructing an NPDA to accept L, we would have the NPDA push a symbol for each 0 it reads, then pop a symbol for each 1 it reads. But then it would not know how many 2's it was supposed to read. This leads us to suspect that L is not context-free. The following lemma will allow us to substantiate these doubts.

Lemma 2.21 (Pumping Lemma for Context-free Languages, Lemma 6.1 of [10])**.** *For any context-free language L, there is a pumping constant p such that for any word $w \in L$ with $|w| \geq p$, we can write $w = qrstu$ where the substrings q, r, s, t, and u satisfy the following:*

(1) $|rst| \leq p$,
(2) $|rt| \geq 1$, and
(3) $qr^n st^n u$ is a word in L for any $n \geq 0$.

Example 2.22. Now we can prove that the language $L = \{0^n 1^n 2^n : n \geq 1\}$ is not context-free.

Proof. Suppose for contradiction that L is context-free. Then the pumping lemma guarantees the existence of a pumping constant p. But consider $w = 0^p 1^p 2^p$. Then we can write $w = qrstu$ where these substrings satisfy the above properties. However, since $|rst| \leq p$, the substring rst can contain at most two of the symbols $0, 1, 2$. Finally, since $|rt| \geq 1$, at least one of r and t contain at least one of the symbols $0, 1, 2$. But then qr^2st^2u does not have an equal number of 0's, 1's, and 2's, a contradiction. □

Theorem 2.23 (Theorem 6.1 of [10])**.** *The set of context-free languages is closed under union, concatenation, and Kleene closure.*

The proof of the above theorem is almost identical to that of Theorem 2.9 and we will not include it here.

The following theorem is in contrast to the case of regular languages.

Theorem 2.24 (Theorem 6.4 of [10])**.** *The set of context-free languages is not closed under intersection.*

In many cases, it is convenient to construct a desired context-free language by intersecting several languages. The following theorem states that we can construct a context-free language by intersecting a (simpler) context-free language with a regular language.

Theorem 2.25 (Theorem 6.5 of [10])**.** *If L is a context-free language and R is a regular language, then $L \cap R$ is a context-free language.*

Proof. In Example 2.22, we showed that the language $L = \{0^n 1^n 2^n : n \geq 1\}$ is not context-free. Since $\{0^n 1^n : n \geq 1\}$ is context-free, it is clear that both $L_1 = \{0^m 1^n 2^n : m, n \geq 1\}$ and $L_2 = \{0^n 1^n 2^m : m, n \geq 1\}$ are also context-free. But $L = L_1 \cap L_2$. If the context-free languages were closed under intersection, L_1 would be context-free, a contradiction. □

Corollary 2.26 (of [10])**.** *The set of context-free languages is not closed under complementation.*

Proof. Since the context-free languages are closed under union by Theorem 2.23, if they were closed under complementation too, then given L_1, L_2 context-free languages,

$$(L_1^c \cup L_2^c)^c = L_1 \cap L_2$$

would also be context-free, which contradicts Theorem 2.24. □

Before discussing the next class of automata, the linear bounded automata, we introduce Turing machines, since a linear bounded automaton is a Turing machine with a simple restriction.

2.4. **Turing Machines and Linear Bounded Automata.** The *Turing machine* (TM) is the most powerful type of automata. It was originally described by Alan Turing in 1936 as a way to formalize our intuitive ideas about algorithms and computation. In fact, the Church-Turing thesis, which is unproven though almost universally accepted, states that if there is an algorithm to perform a given calculation, there is also a TM that can perform the calculation. Surprisingly, while TM are vastly more powerful than NFSA, they only differ from them in three fundamental ways. First, a TM works on an infinite tape, which contains the finite input. Second, a TM is allowed to move left and right along the input tape. Finally, a TM is allowed to write on the tape.

The TM takes as input a tape which has a leftmost cell but is infinite to the right. Each cell holds one symbol. The input string w is copied onto the $|w|$ leftmost cells and each cell to the right of w contains B, a special blank symbol. In a single move, the TM

(1) scans a symbol;
(2) replaces the symbol with any tape symbol (possibly with the same symbol);
(3) moves left or right along the tape; and
(4) changes state.

Note that a TM can work indefinitely. When a TM stops working, we say that it *halts*. A TM halts when an input is accepted, but for a word that is not accepted, it does not necessarily ever halt.

Example 2.27. To illustrate this process, we present an informal design of a simple TM that accepts the language $L_2 = \{0^n 1^n : n \geq 1\}$. The TM repeats the following procedure:

(i) begins by replacing the leftmost 0 with an X,
(ii) moves right until it scans a 1 and replaces it with a Y,
(iii) then moves left until it scans an X, and moves immediately right.

If the TM scans a blank when searching for a 1, it halts and rejects the string. If the TM doesn't find any more 0's in step (i), it checks to see if there are more 1's. If there are, it halts and rejects the string. If there are not, it halts and accepts the string.

Definition 2.28. We formally define a TM N by $N = (Q, \Gamma, B, \Sigma, \delta, q_0, F)$ where

(1) Q is a finite set of states;
(2) Γ is a finite tape alphabet;
(3) $B \in \Gamma$ is the special blank symbol;
(4) $\Sigma \subseteq \Gamma \setminus \{B\}$ is a finite input alphabet;

(5) δ is a function from $Q \times \Gamma$ to $Q \times \Gamma \times \{L, R\}$;
(6) $q_0 \in Q$ is the start state; and
(7) $F \subseteq Q$ is the set of accept states.

We interpret a transition

$$\delta(q, \alpha) = (p, \gamma, L)$$

in the following way. If the TM is in state q and reads α on the tape, it moves to state p, replaces α with γ, and moves left along the type (It would move right if $\delta(q, \alpha) = (p, \gamma, R)$.) A TM accepts a string w if it ever enters an accept state while computing w.

We can now present a formal design of the TM N_2 that accepts L_2. Let N_2 be defined by:

(1) $Q = \{q_0, q_1, q_2, q_3, q_4\}$;
(2) $\Gamma = \{0, 1, X, Y, B\}$;
(3) $\Sigma = \{0, 1\}$;
(4) $F = \{q_4\}$; and
(5) δ given by

$$\begin{aligned}
&\delta(q_0, 0) = (q_1, X, R) && \delta(q_0, Y) = (q_3, Y, R)\\
&\delta(q_1, 0) = (q_1, 0, R) && \delta(q_1, 1) = (q_2, Y, L)\\
&\delta(q_1, Y) = (q_1, Y, R) && \delta(q_2, 0) = (q_2, 0, L)\\
&\delta(q_2, X) = (q_0, X, R) && \delta(q_2, Y) = (q_2, Y, L)\\
&\delta(q_3, Y) = (q_3, Y, R) && \delta(q_3, B) = (q_4, B, L).
\end{aligned}$$

To further illustrate how a TM works, in Figure 11 we give an accepting run on the string 0011.

We can now define the most general class of languages that we will consider—those corresponding to the Turing machines.

Definition 2.29. A language is called *recursively enumerable* if there is a TM that accepts it.

The last type of automata that we will define is called a *linear bounded automaton* (LBA). An LBA is a Turing machine the satisfies two additional conditions:

(1) its input alphabet Σ includes two additional symbols A and Z, the *left* and *right endmarkers*; and
(2) it cannot make moves left of A or right of Z or replace A or Z with another symbol.

Definition 2.30. We formally define an LBA N by $N = (Q, \Gamma, B, \Sigma, \delta, q_0, A, Z, F)$ where

(1) $Q, \Gamma, B, \Sigma, q_0$, and F are defined as for a TM;
(2) δ is defined as for a TM except that it cannot move right of Z or replace A or Z with another symbol; and
(3) $A, Z \in \Sigma$ are the left and right endmarkers.

Example 2.31. The TM N_2 given above is in fact equivalent to an LBA since it never needs to move more than one cell right of the last symbol on the input string and does not have to replace the cell just right of the input string.

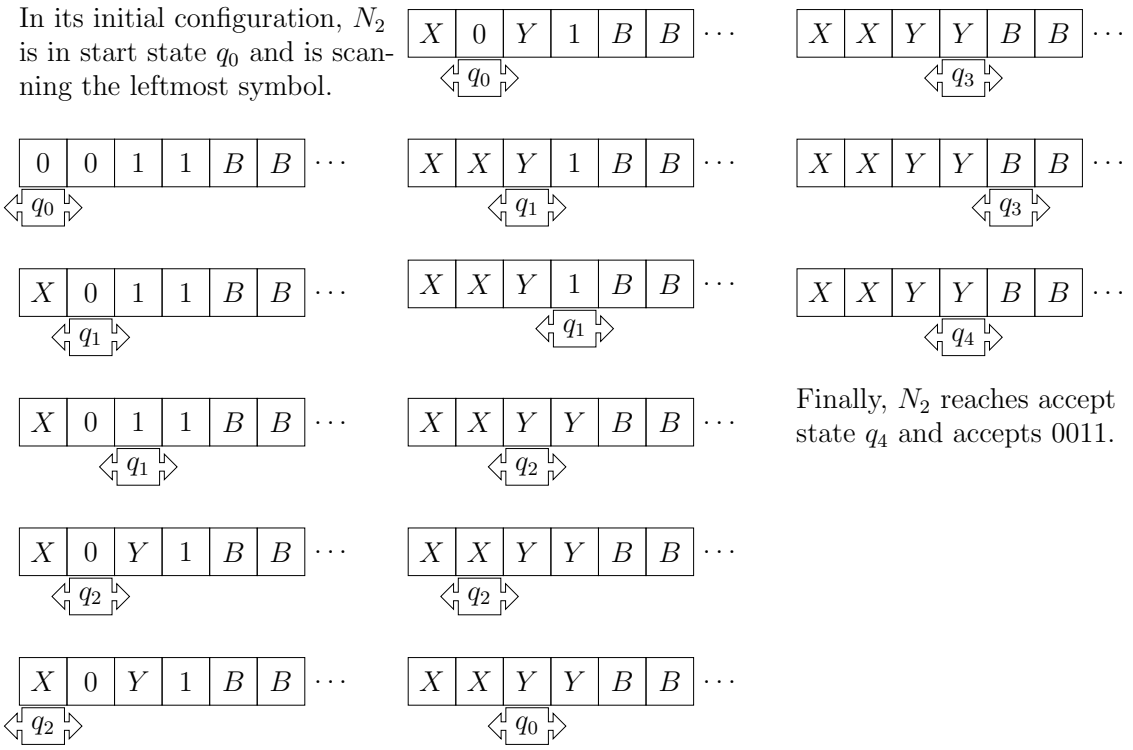

FIGURE 11. Accepting run of N_2 on the input string 0011. This schematic should be read up and down.

Definition 2.32. A language is called *context-sensitive* if there is an LBA that accepts it.

The recursively enumerable and context-sensitive languages are extremely broad categories. The set of context-free languages is properly contained in the set of context-sensitive languages and the set of context-sensitive languages is properly contained in the set of recursively enumerable languages [10]. Very few languages are not context-sensitive and fewer still are not recursively enumerable. Furthermore, both classes have nice closure properties, as seen in the following standard theorem.

Theorem 2.33 ([10])**.** *Both the set of context-sensitive languages and the set of recursively enumerable languages are closed under union, concatenation, Kleene closure, and intersection. The set of context-sensitive languages is closed under complementation, but the set of recursively enumerable languages is not.*

2.5. **The Chomsky Hierarchy.** Since PDA are more powerful than NFSA, for any regular language there is some NPDA that accepts it. Similarly, for any context-free language there is some LBA that accepts it and for any context-sensitive language, there is some Turing machine that accepts it. In fact, the set of regular languages is properly contained in the set of context-free languages, the set of context-free languages is properly contained in the set of context-sensitive languages, and the set of context-sensitive languages is properly contained in the set of recursively enumerable languages. This hierarchy is called the *Chomsky hierarchy* and is shown in Figure 12 below.

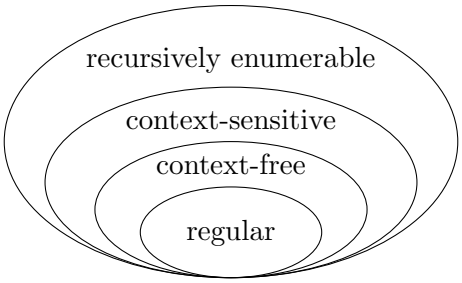

FIGURE 12. Set containment implied by the Chomsky hierarchy.

One of our goals in this paper is to apply the Chomsky hierarchy in the context of group theory. Roughly, we view groups as languages and classify a group according to where its associated language lies in the Chomsky hierarchy.

2.6. **Poly-Context-Free Languages.** The class of context-sensitive languages is very broad compared to the class of context-free languages. In this section, we briefly discuss a fifth set of languages, called the *poly-context-free* (poly-$\mathcal{CF}$) *languages*, which define an infinite hierarchy of languages that lie between the context-free and context-sensitive languages. That is, any context-free language is poly-$\mathcal{CF}$ and any poly-$\mathcal{CF}$ language is context-sensitive. The definitions and results of this section are all from [2].

Definition 2.34. A language is k-context-free (k-$\mathcal{CF}$) if it can be represented as the intersection of k context-free languages. A language is poly-$\mathcal{CF}$ if it is k-$\mathcal{CF}$ for some $k \in \mathbb{N}$.

The poly-$\mathcal{CF}$ languages enjoy some nice closure properties.

Theorem 2.35 (Proposition 2.1 of [2])**.** *The class of poly-$\mathcal{CF}$ languages is closed under intersection. Also, for any k, the class of k-$\mathcal{CF}$ languages is closed under intersection with regular languages.*

Proof. First we want to show that given two poly-$\mathcal{CF}$ languages L_1 and L_2, $L_1 \cap L_2$ is also poly-$\mathcal{CF}$. Well, by definition L_1 is m-$\mathcal{CF}$ and L_2 is n-$\mathcal{CF}$ for $m, n \in \mathbb{N}$. So $L_1 \cap L_2$ is $(m+n)$-$\mathcal{CF}$ and therefore poly-$\mathcal{CF}$.

Next we want to show that the class of k-$\mathcal{CF}$ languages is closed under intersection with regular languages. So choose some k-$\mathcal{CF}$ language L and regular language R. Since L is k-$\mathcal{CF}$,

$$L = L_1 \cap L_2 \cap \ldots \cap L_k$$

for context-free languages L_i. By Theorem 2.25, $L_1 \cap R$ is context-free. Hence,

$$L \cap R = L_1 \cap L_2 \cap \ldots \cap L_k \cap R = (L_1 \cap R) \cap L_2 \cap \ldots \cap L_k$$

is k-$\mathcal{CF}$. □

The following theorem formalizes the idea that the poly-$\mathcal{CF}$ languages form a hierarchy of languages between the context-free languages and the context-sensitive languages.

Theorem 2.36 (Theorem 3.9 of [2]). *For any $k \geq 2$, there is a language that is k-$\mathcal{CF}$ but not $(k-1)$-$\mathcal{CF}$. Thus, the class of k-$\mathcal{CF}$ languages properly contains the class of $(k-1)$-$\mathcal{CF}$ languages.*

This theorem implies the following containment hierarchy.

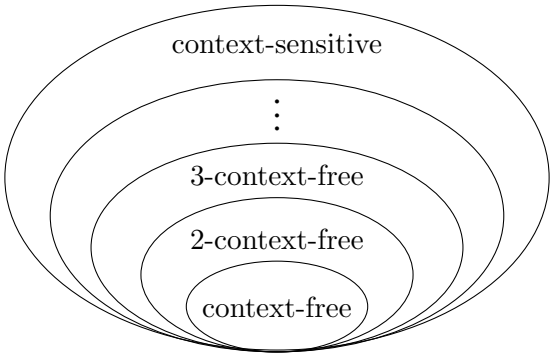

FIGURE 13. Set containment implied by Theorem 2.36.

In this paper, we use the class of 2-$\mathcal{CF}$ languages to analyze a language where symbols in odd-numbered positions form a word in one language and symbols in even-numbered positions form a word in another. To do this, we use the following theorem.

Theorem 2.37. *Given two context-free languages L_1 over Σ_1 and L_2 over Σ_2, the language*

$$L = \{\alpha_1\gamma_1 \cdots \alpha_n\gamma_n : n \in \mathbb{N} \text{ and } w_1 = \alpha_1 \cdots \alpha_n \in L_1, w_2 = \gamma_1 \cdots \gamma_n \in L_2\}$$

is 2-$\mathcal{CF}$.

Proof. First we want to show that the languages

$$L_1' = \{\alpha_1\gamma_1 \cdots \alpha_n\gamma_n : n \in \mathbb{N} \text{ and } w_1 = \alpha_1 \cdots \alpha_n \in L_1, \gamma_i \in \Sigma_2\}$$

and

$$L_2' = \{\alpha_1\gamma_1 \cdots \alpha_n\gamma_n : n \in \mathbb{N} \text{ and } \alpha_i \in \Sigma_1, w_2 = \gamma_1 \cdots \gamma_n \in L_2\}$$

are context-free. To see that L_1' (and similarly L_2') is context-free, consider the NPDA N_1 that accepts L_1. To each state, add a loop for each symbol in Σ_2. Then this new NPDA accepts words from L_1 with symbols from Σ_2 inserted anywhere. We can intersect this language with the regular language

$$A = \{\alpha_1\gamma_1 \cdots \alpha_n\gamma_n : n \in \mathbb{N} \text{ and } \alpha_i \in \Sigma_1, \gamma_i \in \Sigma_2\}$$

to obtain L_1'.

Finally, $L = L_1' \cap L_2'$ and hence L is 2-$\mathcal{CF}$. □

This is all the background in language and automata theory that we need to obtain our main result. In the next two sections, we use this background to study finitely generated groups.

3. Automatic Groups

3.1. **Background.** In the early 1900's, Max Dehn developed a program in group theory that assigned great importance to three algorithmic problems for finitely generated groups. This program still forms the basis for much research in group theory today. One of these problems is called the *word problem*. The word problem for a group G is the algorithmic problem of determining whether or not a word w, written as a string of generators, represents the identity element of G. Before formally defining this idea, we introduce the following definition.

Definition 3.1. Let G be a group with finite generating set S that is closed under inverses. Then we define a homomorphism $\pi : S^* \to G$ in the following way. Given a string $w = s_1 \cdots s_n$ over S, $\pi(w)$ is the group element represented by w, which we denote $\overline{w}$. That is,

$$\pi(w) = \pi(s_1)\pi(s_2)\cdots\pi(s_n) = s_1 \cdots s_n = \overline{w}.$$

Note that since S is a generating set for G, this map is surjective.

Formally, a solution to the word problem is an algorithm which takes as input any string $w \in S^*$ and returns "YES" if and only if $\pi(w) = \overline{w} = e$. An equivalent way to solve the word problem is to present an algorithm that takes as input any two strings $u, v \in S^*$ and returns "YES" if and only if $\overline{u} = \overline{v}$. We can immediately see how this question is connected to automata theory. For some group G generated by S, a solution to the world problem could be an automaton that accepts one of the following languages:

$$L_1 = \{w \in S^* : \overline{w} = e\}$$
$$L_2 = \{(u, v) \in S^* \times S^* : \overline{u} = \overline{v}\}.$$

In [7], Epstein et al., develop the theory of *automatic groups*, a broad class of groups for which the word problem is efficiently solvable. Informally, an automatic group is a group G with a finite generating set S equipped with several FSA, which together form an *automatic structure* for G:

(1) an FSA that recognizes group elements;
(2) an FSA that recognizes when two strings of generators represent the same element;
(3) for each generator $s \in S$, an FSA that recognizes when two elements differ by s.

By introducing automatic groups, Epstein, et al. revolutionized computing with finitely generated groups. Besides an immediate and efficient solution to the word problem, automatic groups enjoy many other nice properties. For automatic groups, there is an algorithm which takes as input a string w of generators and outputs a representative of $\overline{w}$; other algorithms have been developed to construct homomorphisms and isomorphisms between automatic groups; and computations like enumerating group elements are quick in automatic groups.

3.2. **Automatic Groups.** In this section, we formally define the concept of an automatic group and give an example of an automatic structure. Next, we explore some of the properties enjoyed by automatic groups.

Definition 3.2. Fix an alphabet Σ and set of words $\{w_i\}_{i=1}^n$ with each $w_i \in \Sigma^*$. Then the *convolution* of the tuple $(w_1, w_2, \ldots, w_n) \in \left((\Sigma \cup \{\$\})^*\right)^n$, denoted

$$\otimes(w_1, w_2, \ldots, w_n),$$

is defined as follows. The kth symbol of the tuple is $(\alpha_1, \alpha_2, \ldots, \alpha_n)$ where α_i is the kth symbol of w_i if $|w_i| \geq k$ and \$ otherwise.

Intuitively, the convolution of a tuple is obtained by appending the padding symbol \$ to each entry until they are all length $\max_i\{|w_i|\}$ and then group the ith entries of each string together. For example, if we fix $\Sigma = \{a, b\}$ and let $w_1 = ab, w_2 = aabb$, then

$$\otimes(w_1, w_2) = (ab\$\$, aabb) = (a, a), (b, a), (\$, b), (\$, b).$$

Definition 3.3. Let G be a group and S a finite generating set for G that is closed under inverses. Then an *automatic structure* for G consists of the following:

(1) an FSA W over $\Sigma = S$ such that the map $\pi : \mathcal{L}(W) \to G$ is surjective (where π is defined as in Definition 3.1);
(2) an FSA M_ϵ that accepts $\otimes(w_1, w_2)$ with $w_1, w_2 \in \mathcal{L}(W)$ if and only if $\overline{w_1} = \overline{w_2}$; and
(3) for each generator $s \in S$, an FSA M_s that accepts $\otimes(w_1, w_2)$ with $w_1, w_2 \in \mathcal{L}(W)$ if and only if $\overline{w_1}s = \overline{w_2}$.

Then W is called the *word acceptor*, M_ϵ the *equality recognizer*, and $\{M_s\}$ the *multipliers*. A group with an automatic structure is an *automatic group*.

An automatic structure whose word acceptor accepts a unique word for each group element is a particularly good presentation. If the map $\pi : \mathcal{L}(W) \to G$ is bijective, then $\mathcal{L}(W)$ is called a *set of normal forms* for elements of G, as in the following example. In this case, the equality recognizer is trivial: it only needs to recognize if $w_1 = w_2$.

Example 3.4 (Free group)**.** In this example, we present a simple automatic structure for the free group on two generators a and b. So let $S = \{a, b, A, B\}$ where $A = a^{-1}$ and $B = b^{-1}$.

Word acceptor. First we construct the word acceptor: an FSA W that accepts the language of all reduced strings over S, that is, that accepts the language

$$L = \{w \in S^* : a \text{ and } A \text{ are never adjacent and } b \text{ and } B \text{ are never adjacent}\}.$$

Note that L is a set of normal forms for elements of the free group on a and b. The FSA N that accepts L is depicted in Figure 14. Basically, the FSA records which symbol in $\{a, b, A, B\}$ it has just read. If it has just read a, b, A, or B, then reading A, B, a, or b, respectively, will cause it to reject the input since the string would not be reduced. Besides this constraint, the FSA accepts any string of a's, b's, A's, and B's.

Equality recognizer. Since L is a set of normal forms, the equality recognizer is trivial. It just has to check the two strings pairwise to make sure they are identical.

Multipliers. Because L is a set of normal forms, the multipliers are easy to construct. We will only construct M_a since M_b, M_A, and M_B are nearly identical. Generally, given two strings $u, v \in L$, $\overline{u}a = \overline{v}$ if the string u with an a appended to the end equals the string v. The only exception is when u ends with an A. In this case,

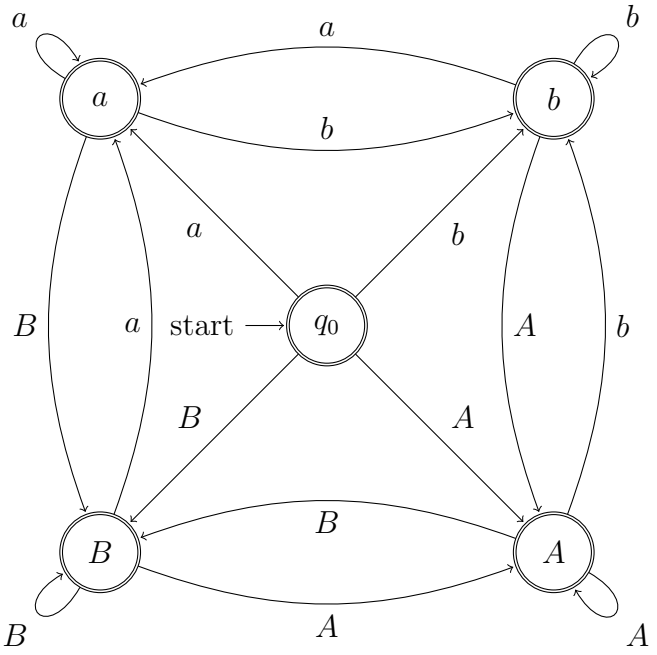

FIGURE 14. The word acceptor N that accepts a set of normal forms for elements of the free group on a and b.

$\overline{u}a = \overline{v}$ if the string v with an A appended to the end equals the string u. We depict the NFSA M_a in Figure 15. Since $u, v \in L$, we can assume that neither contains the sequence aA or the sequence Aa.

The NFSA M_a takes as input the convolution $\otimes(w_1, w_2)$. So M_a is an NFSA over the alphabet $\Sigma = \{a, b, A, B, \$\} \times \{a, b, A, B, \$\}$.

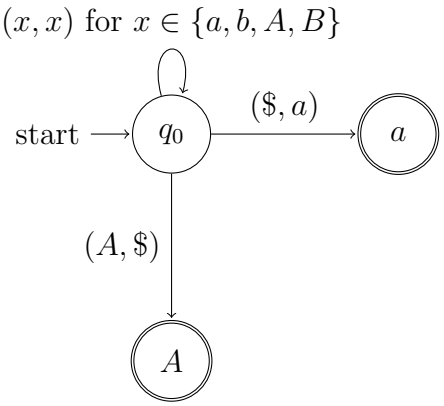

FIGURE 15. The multiplier M_a, which recognizes when $\overline{u}a = \overline{v}$.

Thus, the free group on two generators is an automatic group.

3.3. **Properties of Automatic Groups.** We now discuss some of the properties enjoyed by automatic groups. In many situations, we only need to know that the equality recognizer and multipliers exists, but we do not need to know their exact

structure. So for an automatic group G generated by S that has word acceptor W with $\mathcal{L}(W) = L$, we refer to the automatic structure simply as (S, L).

As stated at the outset, the word problem is efficiently solvable in automatic groups. We now prove this as a corollary to the following theorem. First we state a definition, which allows us to interpret "efficiently solvable."

Definition 3.5. Suppose we have some algorithm A that takes a word from a set $\{w_i\}_{i=1}^n$ and decides if it has a given property. Let $A(k)$ denote the number of steps required to decide if a word of length k has the property. Then we say that A is $O(g(k))$ if there are positive real numbers N and k_0 such that

$$A(k) \leq N \cdot g(k) \text{ for all } k > k_0.$$

By convention, we use the simplest, smallest function $g(k)$. For instance, an algorithm that is $O(k^2)$ is also $O(k^2 + 3k + 7)$ and $O(k^3)$. We would say that it is $O(k^2)$.

Example 3.6. Consider an algorithm A which takes a word and decides if it is a palindrome. The algorithm checks the first and last letters to make sure they are the same; then the second and second-to-last letters; and so on. For a word of length k, this process takes roughly $k/2$ steps; that is, $A(k) = k/2$. We claim that $A(k)$ is $O(k)$. We can use $N = 1$ and $k_0 = 1$. Then clearly

$$A(k) = k/2 \leq k \text{ for all } k > 1,$$

as desired.

Theorem 3.7 (Theorem 2.3.10 of [7])**.** *Let G be an automatic group with structure (S, L). For any word $w_1 \in S^*$, we can find a string $w_2 \in L$ such that $\overline{w_1} = \overline{w_2}$ in time proportional to the square of the length of w_1, that is, in $O(n^2)$ time..*

Sketch of proof. Suppose we are given a word $w_1 = s_1 \cdots s_n$ and want to find a representative $v \in L$ for w_1. The idea is that we first find a representative for s_1, then for $s_1 s_2$, then for $s_1 s_2 s_3$, and so on until we find a representative for w.

To see how this works, suppose we have a representative u for $s_1 \cdots s_{n-1}$. Then we want to find a representative v for $s_1 \cdots s_{n-1} s_n$. Well, the multiplier M_{s_n} must accept $\otimes(u, v)$. So to find v, simply ignore the second element in the labels of M_{s_n}, treating it as an FSA in one variable that accepts u. So we follow u (perhaps with some number of \$'s appended to it) to an accept state and then read off the second elements of the labels we traversed to find v. Note that the number of \$'s appended to u is bounded by C, the number of states in M_{s_n}. If we visited the same state twice reading only \$'s, we could just eliminate the loop.

Finding a representative for $s_1 \cdots s_{n-1} s_n$ given a representative for $s_1 \cdots s_{n-1}$ in this way requires $O(n)$-time since we must read $n + C$ symbols. Because we must repeat this process for each of the n symbols in w_1, the entire procedure requires $O(1 + 2 + \ldots + n) = O(n^2)$-time. □

Corollary 3.8. *Let G be an automatic group with structure (S, L). Then the word problem in G is solvable in quadratic time.*

Proof. Suppose we are given a word w_1 and want to determine if $\overline{w_1} = e$. Then by the above theorem, we can find a representative $w_2 \in L$ such that $\overline{w_1} = \overline{w_2}$ in quadratic time. We then pass (w_2, e) to the equality recognizer, which we can assume is an

FSA. The equality recognizer requires linear time since it visits a single state for each symbol in w_2. Hence, we can determine if $\overline{w_1} = e$ in quadratic time. □

Given a group G with an automatic structure (S, L) it is reasonable to ask how we might obtain an alternative automatic structure for G. The following theorem gives a method for achieving this.

Theorem 3.9 (Corollary 2.3.8 of [7]). *If (S, L) is an automatic structure for a group G and $L' \subseteq L$ is a regular language that maps onto G, then (S, L') is an automatic structure for G.*

Proof. Since L' is regular, we are guaranteed a word acceptor. The equality recognizer and multipliers from the structure (S, L) can function as the equality recognizer and multipliers for (S, L'); they now only have to deal with fewer pairs of input words. □

As with any question in group theory, we want to know whether or not the property we are studying depends on the generating set we use. Does the group possess the property regardless of how we interpret it, or did we happen to pick a good representation? This following theorem, which we state without proof, says that automaticity is invariant under generating set: if a group is automatic with respect to one finite generating set, it is automatic with respect to any finite generating set.

Theorem 3.10 (Theorem 2.4.1 of [7]). *Let G be a group with automatic structure (S, L). If S' is another finite generating set for G, then there is some regular language L' such that (S', L') is also an automatic structure for G.*

The concept of an automatic group was conceived as a way to unify many disparate types of groups. In particular, group theorists hoped the class of automatic groups would include the fundamental groups of compact 3-manifolds, so they could use the fundamental groups' automatic structures to ease computation. Not long after the introduction of automaticity, however, it was shown that certain fundamental groups of compact 3-manifolds, even ones that possessed many properties of automatic groups, were not automatic. So the class of automatic groups is too restricted. This problem motivates a natural generalization of the concept of an automatic group, which we discuss in the following section [11].

4. Generalizations of Automaticity

4.1. **Background.** As observed in the previous section, automatic groups enjoy many nice properties. In particular, as stated in [11], if a group G is automatic,

(1) G is finitely presented,
(2) the Dehn function in G is at most quadratic,
(3) G possesses the Lipschitz property,
(4) the word problem in G is solvable in quadratic time, and
(5) for any word $w \in X^*$, one can find a representative for w in L in quadratic time.

We also noted, however, that the class of automatic groups is not broad enough to include many groups mathematicians hoped it would include. Miasnikov et al. [11] consider properties (1) and (2) unnecessary restrictions on the class of automatic groups. This motivates the concept of graph automaticity, a property with weaker requirements than automaticity that still guarantees properties (4) and (5). Thus, the class of graph automatic groups is much wider than the class of automatic groups, but a graph automatic structure still makes group computations much easier. In the following sections, we define the concept of a graph automatic group and explore some of the properties these groups enjoy. The definitions and results are all from [11]. Finally, we introduce some original generalizations of graph automaticity.

4.2. **Graph Automaticity.** In [11], graph automatic groups are defined as follows. Note that graph automaticity is almost identical to automaticity. The only difference is that graph automaticity eliminates the dependence on generators in the regular language that represents the group. Thus, the class of graph automatic groups includes some groups that have nice representations that do not depend on generators.

Definition 4.1. Let G be a group and S a finite generating set for G. Then a *graph automatic structure* for G consists of the following:

- an FSA W over some alphabet Σ and an onto map $f : \mathcal{L}(W) \to G$ (hereafter we denote $\mathcal{L}(W) = R$);
- an FSA M_ϵ that accepts $\otimes(w_1, w_2)$ with $w_1, w_2 \in R$ if and only if $f(w_1) = f(w_2)$;
- for each generator $s \in S$, an FSA M_s that accepts $\otimes(w_1, w_2)$ with $w_1, w_2 \in R$ if and only if $f(w_1)s = f(w_2)$.

A group with a graph automatic structure is a *graph automatic group*. Since the only difference between the definition of graph automaticity and automaticity is that automaticity requires $\Sigma = S$, we have the following fact.

Proposition 4.2 (Proposition 7.3 of [11])**.** *All automatic groups are also graph automatic.*

As noted in the introduction, graph automatic groups enjoy many of the properties that automatic ones do. Theorem 4.6 states that, as with automatic groups, the word problem in graph automatic groups is solvable in quadratic time. First we state a definition and prove two important lemmas.

Definition 4.3. A function $f : X \to Y$ is *FSA recognizable* if the set

$$\{\otimes(x, y) : f(x) = y\}$$

is regular. A function is deterministic PDA, NPDA, LBA, or Turing machine recognizable if the analogous condition holds, respectively.

Lemma 4.4 (Lemma 8.2 of [11])**.** *Let D be a language over some alphabet Σ. Given a function $\delta : D^n \to D$ that is FSA recognizable and given $(x_1, \ldots, x_n) \in D^n$, we can compute $\delta(x_1, \ldots, x_n)$ in linear time.*

Proof. This proof is similar to the proof of Theorem 3.7. Let N be the FSA that recognizes δ. Then we want to consider all paths of the form

$$(x_1, \ldots, x_n, y') \text{ with } |y'| \leq \max\{|x_i|\}.$$

Denote the set of all such paths by X and let S be the set of all the last states of paths in X. For any given step i, the number of paths is bounded by the number of states in N, since if two paths visit the same state at the same step, we can eliminate one of them. Furthermore, each path is at most length $\max\{|x_i|\}$. Hence, the set S can be computed in time $C \cdot \max\{|x_i|\}$ where C is the number of states in N.

Suppose first that S contains an accept state q. Then there is a path in X from q_0 to q with a label of the form $(x_1, \ldots, x_n, y')$ with $|y'| \leq \max\{|x_i|\}$. So $\delta(x_1, \ldots, x_n) = y'$ and we found y' in linear time.

On the other hand, suppose S does not contain an accept state. Since $\delta(x_1, \ldots, x_n)$ does have a value, there must be a state $s \in S$ and an accept state q such that there is a path from s to q labeled $(\$^k, \ldots, \$^k, y'')$ and $|y''| = k \leq C$, the number of states in N. So $\delta(x_1, \ldots, x_n) = y'y'' = y$. Finding y' takes linear time and finding y'' takes constant time, so the entire process takes linear time. This completes the proof. □

Roughly, the next lemma shows that a representative for an element cannot be much longer than the string of generators corresponding to the element.

Lemma 4.5. *Suppose that a group G is graph automatic with respect to a finite generating set S and alphabet Σ. Then any word $w = s_1 \cdots s_n \in S^*$ has representative $u \in R$ with $f(u) = w$ and $|u|$ is $O(n)$. (Recall that f is the function that maps from the regular language R onto G, as in definition 4.1.)*

The following proof is expanded from the proof of Theorem 8.1 of [11].

Proof (by induction). First enumerate the multipliers in the graph automatic structure for G by $M_{s_1}, \ldots, M_{s_k}$. Then denote

$$C = \max_{i \leq k}\{\text{number of states in } M_{s_i}\}.$$

Next, let γ be the shortest string with $f(\gamma) = e$. Then we claim that any word w has representative u with

$$|u| \leq |\gamma| + C \cdot n.$$

We induct on n.

Base Case. First suppose that $|w| = 0$, i.e., that $w = e$. Then $u = \gamma$ is a representative for w, so the inequality holds.

Induction Step. Suppose that $|w| = n + 1$, i.e., that $w = s_1 \cdots s_{n+1}$ for $s_i \in \Sigma$. Denote by u the shortest representative for w. By induction, we know that there is a representative u_n for $w' = s_1 \cdots s_n$ with

$$|u_n| \leq |\gamma| + C \cdot n.$$

We know that M_{n+1} accepts the convolution $\otimes(w', w)$. So let C' denote the number of states in M_{n+1}. By the argument in the proof of Lemma 4.4, we need to append at most C' padding symbols \$ to w' for $\otimes(w', w)$ to reach an accept state in M_{n+1}. Hence,

$$|u| \leq |u_n| + C' \leq |\gamma| + C \cdot n + C' \leq |\gamma| + C \cdot (n+1),$$

completing the proof. □

Now we can prove the theorem.

Theorem 4.6 (Theorem 8.1 of [11])**.** *Let G be a group with finite generating set S. If G is graph automatic, then the word problem in G is solvable in quadratic time.*

Proof. Suppose we have a word $w = s_1 \cdots s_n \in S^*$. We must decide in quadratic time whether $\pi(w) = e$. First we want to find a representative $u \in R$ such that $f(u) = \pi(w)$. Then we feed (u, γ) to M_ϵ to determine if $\pi(w) = f(u) = f(\gamma) = e$. (Here γ denotes the shortest string with $f(\gamma) = e$.)

Let $\delta_s : R \to R$ denote the function given by

$$\{(w_1, w_2) : w_1, w_2 \in R \text{ and } f(w_1)s = f(w_2)\}.$$

Since G is graph automatic, for any $s \in S$, the function δ_s is FSA recognizable. (The FSA that recognizes this function is the multiplier M_s.) Then $\delta_{s_1}(\epsilon) = u_1$ is a representative in R for s_1. Next we compute $\delta_{s_2}(u_1)$ to obtain a representative $u_2 \in R$ for $s_1 s_2$. Following this procedure, for any i, we can find a string u_i representing $s_1 \cdots s_i$. By Lemma 4.5, $|u_i| \leq C_1 \cdot i$ for some constant C_1. By Lemma 4.4, given u_{i-1}, we can find u_i in time $C_2 \cdot |u_i| \leq C_2 \cdot C_1 \cdot i$ for some constant C_2. Thus, the word $u = u_n \in R$ representing w can be found in time $C_2 \cdot C_1(1 + 2 + \ldots + n) = O(n^2)$.

We then feed $\otimes(u, \epsilon)$ to the equality recognizer M_ϵ. Since $|\epsilon| = 0$ and $|u| \leq C_1 \cdot n$, the equality recognizer only needs $O(n)$-time to decide whether $f(u) = f(\epsilon) = e$.

Hence, it takes $O(n^2)$-time to decide if $\pi(w) = f(u) = e$, completing the proof. □

Remark 4.7. This theorem does not rely on the complexity of the language R—only on the multipliers and equality recognizers.

As with automaticity, graph automaticity is invariant under generating set. To prove this, we use the following lemma.

Lemma 4.8 (Lemma 6.8 of [11])**.** *Let G be a group with finite generating set S. If G is graph automatic with respect to S, then for any word $y \in S^*$, there exists an FSA M_y that given $u, v \in R$, accepts (u, v) if and only if $v = uy$.*

Now we prove the theorem.

Theorem 4.9 (Theorem 6.9 of [11])**.** *If a group G is graph automatic with respect to a finite generating set X, then G is graph automatic with respect to any finite generating set Y of G.*

Proof. Changing generators does not require that we produce a new language R or a new equality recognizer. We only need to show that for each $y \in Y$, there is an FSA M_y that detects multiplication by y. So choose some $y \in Y$. Since G is generated by X, y can be written as some string of generators $y = x_1 \cdots x_n \in X^*$. Then the above lemma guarantees the existence of an FSA M_y that given $u, v \in R$, accepts (u, v) if and only if $v = uy$. This completes the proof. □

So generalizing the notion of automaticity to graph automaticity preserves many nice properties. In the following section, we introduce some generalizations of graph automaticity.

4.3. **Generalizations of Graph Automaticity.**

Definition 4.10. Let G be a group and S a finite generating set for G. Then a *graph context-free structure* for G consists of the following:

- an NPDA W over some alphabet Σ and an onto map $f : \mathcal{L}(W) \to G$;
- an NPDA M_ϵ that accepts $\otimes(w_1, w_2)$ with $w_1, w_2 \in R$ if and only if $f(w_1) = f(w_2)$;
- for each generator $s \in S$, an NPDA M_s that accepts $\otimes(w_1, w_2)$ with $w_1, w_2 \in R$ if and only if $f(w_1)s = f(w_2)$.

We define a generalization of graph automaticity for each language class. For example, a group G is *graph context-sensitive* and *graph recursively enumerable* if the analogous conditions hold with respect to LBA and Turing machines, respectively.

Theorem 4.11. *Let G be a group with finite generating set S. If the following conditions hold with respect to some set of symbols Σ,*

- *there is a recursively enumerable language $R \subset \Sigma^*$ and an onto function $f : R \to G$;*
- *there is an FSA M_ϵ that given two words $w_1, w_2 \in \mathbb{R}$, accepts (w_1, w_2) if and only if $f(w_1) = f(w_2)$; and*
- *for each $s \in S$, there is an FSA M_s that given two words $w_1, w_2 \in R$, accepts (w_1, w_2) if and only if $f(w_1)s = f(w_2)$.*

then the world problem in G is solvable in quadratic time.

Proof. This result follows directly from Remark 4.7. □

In this section, we have introduced several generalizations of automaticity in order to define a hierarchy in the set of finitely generated groups that is analogous to the Chomsky hierarchy for formal languages. This hierarchy is represented in Figure 16.

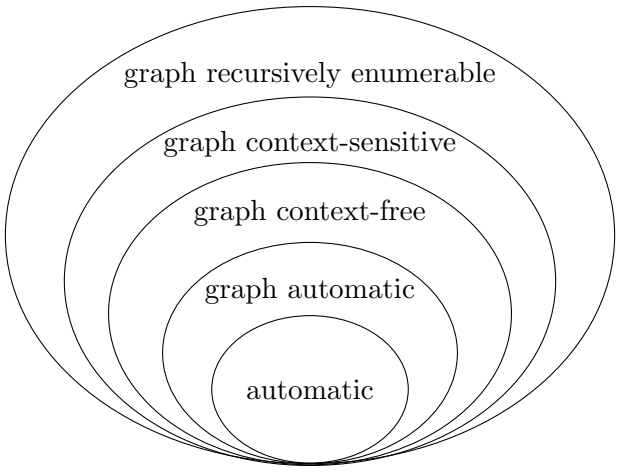

FIGURE 16. The Chomsky hierarchy extended to finitely generated groups.

5. Thompson's Group F

5.1. **Background.** Thompson's group F was introduced by Richard Thompson in the 1960's in connection with questions in logic. Thompson originally defined F to describe associative laws for rearranging parenthesized expressions. For instance, two elements are the rules:

$$x_0 : (ab)c \mapsto a(bc)$$
$$x_1 : a\big((bc)d\big) \mapsto a\big(b(cd)\big)$$

where a, b, c, and d are expressions [1]. Thompson's group F has since found applications in many areas of mathematics including algebra, logic, and topology. In their canonical introduction to Thompson's group F [4], Cannon, Floyd, and Parry define F in a very different way. Many of the following statements are proved in [4], but in some cases, we have supplied our own proofs. We start with a few definitions.

Definition 5.1. A *homeomorphism* $f : A \to B$ is a bijective, continuous function whose inverse $f^{-1} : B \to A$ is also continuous.

Definition 5.2. A *dyadic rational* is a number of the form $a/2^b$ for non-negative integers a and b.

Then Cannon, Floyd, and Parry define F in the following way.

Definition 5.3. Let F be the set of homeomorphisms from the interval $[0, 1]$ to itself satisfying the following four conditions:

(1) Each homeomorphism is a piecewise linear function;
(2) Each function has finitely many points of non-differentiability;
(3) The x-coordinate of each of these points is a dyadic rational number; and
(4) Away from these points, the slopes of the linear pieces are all powers of 2.

Theorem 5.4. *The set F defined above forms a group under function composition. This group is called Thompson's group F.*

Before proving Theorem 5.4 we prove the following lemma.

Lemma 5.5. *Let $f \in F$ and $x \in [0, 1]$ be a dyadic rational. Then $f(x)$ is also a dyadic rational.*

Proof. Let $0 = x_0 < x_1 < x_2 < \cdots < x_n = 1$ be the points of non-differentiability of $f \in F$ where each x_i is a dyadic rational. Since $f(x_0) = 0$, $f(x) = a_1 x$ for $x_0 \leq x \leq x_1$ and a_1 a power of 2. Then since x_1 is a dyadic rational and a_1 is a power of 2, $f(x_1) = b_2$ where b_2 is a dyadic rational. Then it follows that $f(x) = a_2 x + b_2$ for $x_1 \leq x \leq x_2$ and a_2 a power of 2. By induction, then, $f(x) = a_i x + b_i$ for $x_{i-1} \leq x \leq x_i$ where a_i is a power of 2 and b_i is a dyadic rational.

Now choose any dyadic rational $x \in [0, 1]$. Then $f(x) = a_i x + b_i$ for some i. But since a_i is a power of 2, $a_i x$ is a dyadic rational and since b_i is also a dyadic rational, $f(x) = a_i x + b_i$ is a dyadic rational. □

Now we prove Theorem 5.4.

Proof. Associativity. This follows because the operation is function composition.

Closure. First choose $f_1, f_2 \in F$. We want to show that $f_2 \circ f_1 \in F$. Clearly $f_2 \circ f_1$ is a piecewise linear homeomorphism with finitely many points of non-differentiability. Further, since f_1 and f_2 both map dyadic rationals to dyadic rationals, as shown in Lemma 5.5, all points of non-differentiability of $f_2 \circ f_1$ are dyadic rationals. Finally, at any point x where f_1 is differentiable at x and f_2 is differentiable at $f_1(x)$, the slope of $f_2 \circ f_1$ is just the slope of f_1 at x times the slope of f_2 at $f_1(x)$, which must be a power of 2.

Identity. The identity is $f(x) = x$, which is piecewise linear, has no points of non-differentiability, and has slope 2^0.

Closed under inverses. Choose $f \in F$. Clearly f^{-1} is a piecewise linear function. Further, f^{-1} has a point of non-differentiability at (x, y) if and only if f has one at (y, x). So f^{-1} has finitely many points of non-differentiability. Next, since f maps dyadic rationals to dyadic rationals by Lemma 5.5, the points of non-differentiability of f^{-1} also lie at dyadic rationals. Finally, the inverse of a power of 2 is also a power of 2, so every linear piece of f^{-1} has a slope that is a power of 2. □

Example 5.6. We present two important elements, $x_0(x), x_1(x) \in F$, in Figure 17. The piecewise linear functions are defined as follows:

$$x_0(x) = \begin{cases} x/2, & 0 \le x \le 1/2 \\ x, & 1/2 \le x \le 3/4 \\ 2x, & 3/4 \le x \le 1 \end{cases} \quad \text{and} \quad x_1(x) = \begin{cases} x, & 0 \le x \le 1/2 \\ x/2, & 1/2 \le x \le 3/4 \\ x, & 3/4 \le x \le 7/8 \\ 2x, & 7/8 \le x \le 1. \end{cases}$$

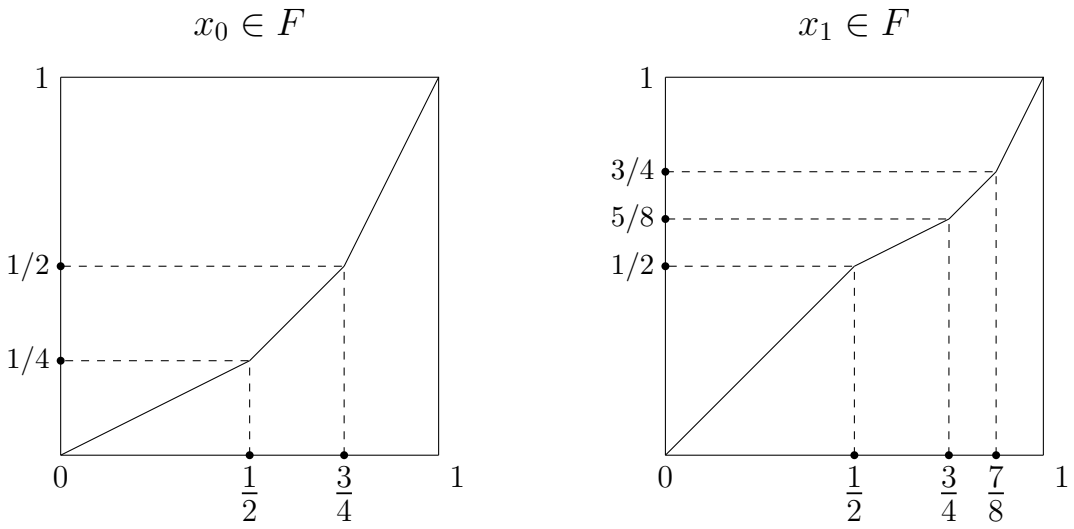

FIGURE 17. The elements $x_0(x)$ and $x_1(x)$ in F.

Theorem 5.7 (Corollary 2.6 of [4])**.** *The set $S = \{x_0(x), x_1(x)\}$ generates Thompson's group F.*

This is the generating set for F that we work with for the remainder of the paper. Since we refer to these homeomorphisms so frequently, we will abbreviate $x_0(x)$ to x_0 and $x_1(x)$ to x_1.

5.2. **Binary Trees.** It can be cumbersome to work with F as a group of piecewise linear homeomorphisms. Instead we will use a useful representation of elements of F: as pairs of finite, rooted binary trees. This representation has been used in proving many of F's properties. For example, Cleary and Taback used binary trees in [5] to show that F is not almost convex and Fordham used binary trees in [8] to construct a word length formula for elements of F.

Definition 5.8. A *tree* is an undirected graph in which any two vertices are connected by exactly one simple path (that is, exactly one path with no repeated vertices).

Definition 5.9. A *finite rooted binary tree* is a tree T such that

(1) T has one vertex v_0 that is designated the root vertex,
(2) if T consists of more than v_0, v_0 has valence 2,
(3) if v is a vertex in T with valence greater than 1, there are exactly two edges, which we denote $e[v, L]$ and $e[v, R]$, that contain v and are not part of the shortest path from v_0 to v, and
(4) T has a finite number of vertices.

Definition 5.10. A vertex v together with the two edges $e[v, L]$ and $e[v, R]$ form a *caret*, as seen in Figure 18. We denote a caret C by the triplet $(v, e[v, L], e[v, R])$.

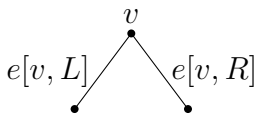

FIGURE 18. A caret.

Definition 5.11. The *right child* of a caret $C = (v, e[v, L], e[v, R])$ is the caret $C_R = (v_R, e[v_R, L],\ e[v_R, R])$ where $e[v, R]$ connects the vertices v and v_R, as seen in Figure 19. The *left child* is defined analogously.

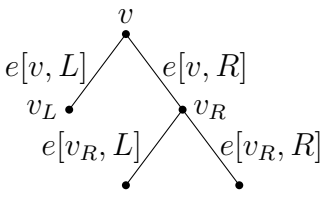

FIGURE 19. A caret C with a right child C_R.

Definition 5.12. A vertex with valence 1 is called a *leaf*, as seen in Figure 20. By convention, leaves are numbered from left to right.

Definition 5.13. A *tree pair diagram* is a pair of finite, rooted binary trees with the same number of leaves. By convention, we place the trees side-by-side. The tree on the left is referred to as the T_- tree; the tree on the right is referred to as the T_+ tree. The reason for this notation will become clear below.

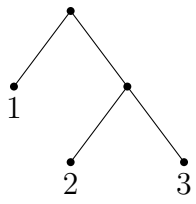

FIGURE 20. Leaves numbered 1, 2, 3.

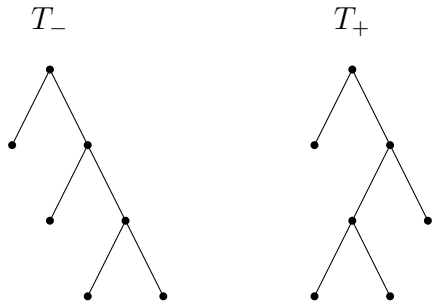

FIGURE 21. A tree pair diagram.

Example 5.14. A tree pair diagram is depicted in Figure 21. In fact, this tree pair diagram corresponds to the element x_1 defined earlier.

Each tree in a tree pair corresponds to a subdivision of the unit interval in the following way. Assign the root vertex the interval $[0, 1]$. Then for any vertex v with valence 2 or 3 that has been assigned $[a, b]$, the vertex that shares $e[v, L]$ is assigned $[a, (b+a)/2]$ and the vertex that shares $e_{v,R}$ is assigned $[(b+a)/2, b]$. Then the leaves of the tree define a subdivision of the interval $[0, 1]$. Further, a pair of trees defines an element of F: we map the interval assigned to leaf 1 of the T_- tree to the interval assigned to leaf 1 of the T_+ tree; and so on. For this reason, the T_- tree is also called the *domain tree* and the T_+ tree the *range tree*.

Example 5.15. In Figure 22 we give an example of how each tree defines a subdivision of $[0, 1]$ and thus how a pair of trees defines a mapping from $[0, 1]$ to itself.

The tree pair in Figure 22 defines the following mapping of subintervals:

$$\begin{aligned}
&[0, 1/2] \to [0, 1/2] \\
&[1/2, 3/4] \to [1/2, 5/8] \\
&[3/4, 7/8] \to [5/8, 3/4] \\
&[7/8, 1] \to [3/4, 1].
\end{aligned}$$

As claimed above, the function defined by this mapping is

$$x_1 = \begin{cases} x, & 0 \le x \le 1/2 \\ x/2, & 1/2 \le x \le 3/4 \\ x, & 3/4 \le x \le 7/8 \\ 2x, & 7/8 \le x \le 1. \end{cases}$$

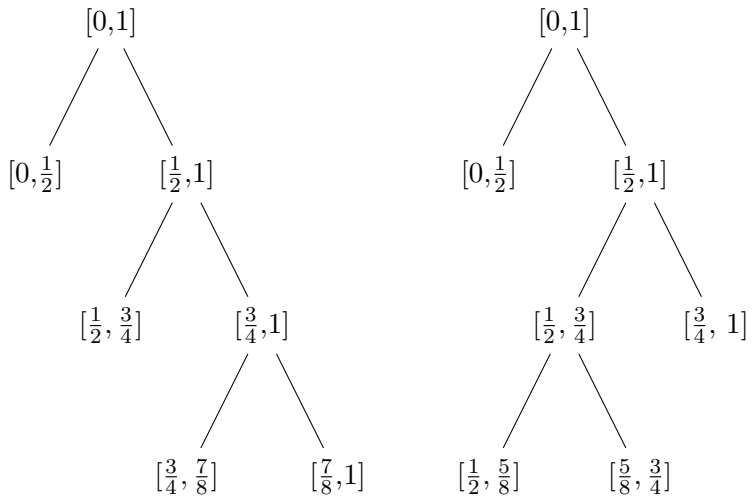

FIGURE 22. A pair of trees with subintervals of $[0, 1]$ assigned to each vertex.

Definition 5.16. Suppose that for a given tree pair both the T_- and T_+ tree contain a caret with leaves numbered n and $n + 1$, as in Figure 23 with $n = 1$. Then these carets are called *redundant* and the tree pair is called *unreduced*. If a tree pair has no redundant carets it is *reduced*.

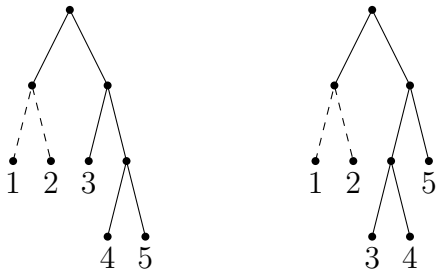

FIGURE 23. An undreduced tree pair diagram with the redundant carets dashed.

Note the effect of these redundant carets. Where the previous tree pair specified the mapping of

$$[0, 1/2] \to [0, 1/2]$$

this unreduced tree pair further specifies

$$[0, 1/4] \to [0, 1/4]$$
$$[1/4, 1/2] \to [1/4, 1/2]$$

which does not change the corresponding piecewise linear homeomorphism. Since we can always add redundant carets, for any given element $f \in F$ there are infinitely many tree pair diagrams that correspond to it. Cannon, Floyd, and Parry show in [4] that in fact there is exactly one reduced tree diagram for each element of F. When we talk about the tree pair diagram associated to $f \in F$, we refer to this unique reduced pair.

5.3. **Multiplication of Binary Trees.** In this section, we show how to multiply pairs of binary trees. The process is analogous to the process of composing piecewise linear functions. To illustrate, take two elements of F, say, f_1 and f_2 depicted in Figure 24. We want to compute $f_1 \cdot f_2$, which by convention means $f_1 \circ f_2$. If we were composing functions, we would have to make sure that f_2 has a break point at (x, y) if and only if f_1 has one at (y, z). So we would add redundant break points to f_1 and f_2, to obtain f_1' and f_2'. Then $(f_1 \circ f_2)'$ would have a break point at any x-value at which f_2' has one. The y-value of the break point would be $(f_1 \circ f_2)(x)$. Finally, we eliminate any redundant break points from $(f_1 \circ f_2)'$ to obtain $f_1 \circ f_2$.

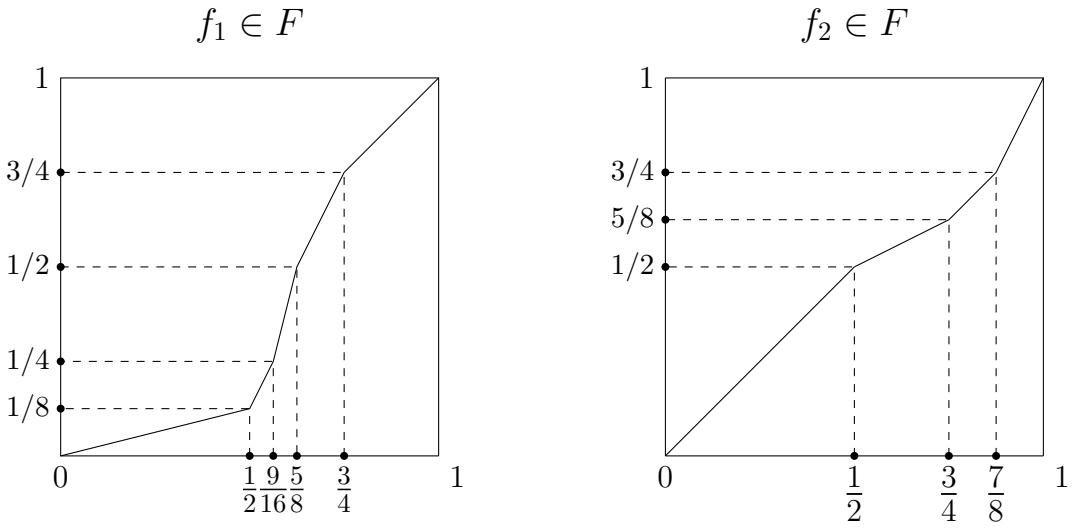

FIGURE 24. The elements $f_1, f_2 \in F$.

Multiplying binary trees is analogous. First we add redundant carets to f_1 and f_2 until the range tree of f_2 is identical to the domain tree of f_1. This step is shown in Figure 26. (In this case, we did not have to add redundant carets to f_1, but this is not always the case.) Then the new domain tree of f_2 and the new range tree of f_1 constitute the tree pair representative for $(f_1 \circ f_2)'$. In the last step, we eliminate redundant carets to obtain the tree pair representative for $f_1 \circ f_2$. This entire process is demonstrated in Figures 25-28.

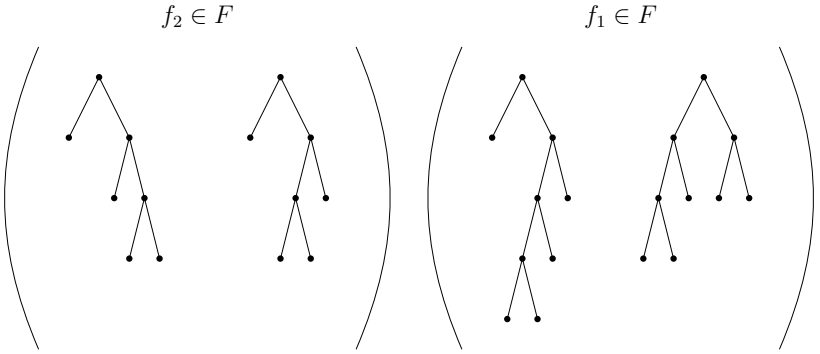

FIGURE 25. Step 1: Line up trees with f_2 on the left.

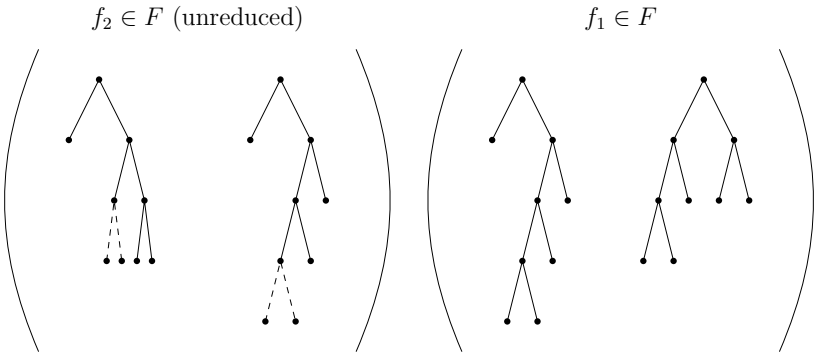

FIGURE 26. Step 2: Add redundant carets.

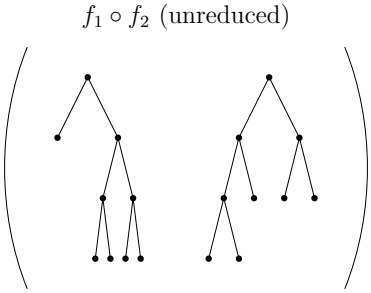

FIGURE 27. Step 3: Ignore new range tree of f_2 and new domain tree of f_1.

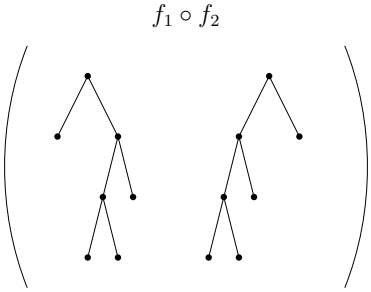

FIGURE 28. Step 4: Eliminate redundant carets.

In Section 6.2, we discuss the effect on a tree pair of multiplication by the generators x_0 and x_1.

5.4. **Labeling Carets in Binary Trees.** In the proofs below, we use two methods for labeling carets in binary trees. The purpose of labeling carets is to associate a string of symbols to each element so that we can construct a set of normal forms for F. Before discussing these methods, we introduce several definitions.

Definition 5.17. The left child of the root caret is a *left exterior caret*. The left child of any left exterior caret is also a left exterior caret. Similarly, the right child of the root caret is a *right exterior caret* and the right child of a right exterior caret is also a right exterior caret.

Definition 5.18. A caret is an *exterior caret* if it is the root caret or else a left or right exterior caret.

Definition 5.19. The right child of a left exterior caret and the left child of a right exterior caret are *interior carets*. The child of an interior caret is an interior caret.

Definition 5.20. An interior caret together with all its descendants constitute an *interior subtree*.

The first method for labeling carets is used by Elder, Fusy, and Rechnitzer in [6] and deals only with interior subtrees. Leaves are labeled "N" or "I" and carets are labeled "n" or "i" as in Figure 29 below:

FIGURE 29. Leaves are labeled "N" or "I" and carets are labeled "n" or "i." By convention, the root caret is labeled "n" and the left-most leaf is not labeled.

First, leaves and carets are labeled. A word over the alphabet {N, I, n, i} encoding an interior subtree consists of all the labels read left to right. The interior subtree in Figure 30 is encoded by the word "nInNiInNnIiNiI."

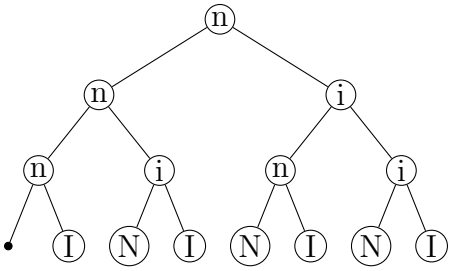

FIGURE 30. Interior subtrees are labeled in [6] as shown.

The second method is our own, based on [8]. In constructing a word length formula, Fordham first numbered the carets in infix order. This means that for any given caret, we first visit its left child, then we visit the caret, then we visit its right child. This numbering system is depicted in Figure 31.

Next, Fordham labeled the carets in the following way, as described in [5].

(1) The left exterior caret with no left child is labeled L_0,
(2) all other left exterior carets are labeled L_L,
(3) a right exterior caret numbered k such that caret $k+1$ is interior is labeled R_I,
(4) a right exterior caret which is not R_I, but for which there is some R_I with a higher infix number, is labeled R_{NI},
(5) a right exterior caret for which there is no interior caret with a higher infix number is labeled R_0,
(6) an interior caret with a right child is labeled I_R, and

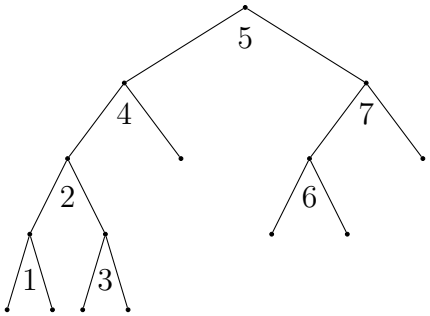

FIGURE 31. A binary tree with its carets numbered in infix order.

(7) an interior caret with no right child is labeled I_0.

By convention, the root caret is considered a left exterior caret and thus is labeled either L_L or L_0. In Figure 32, a tree is labeled according to this classification.

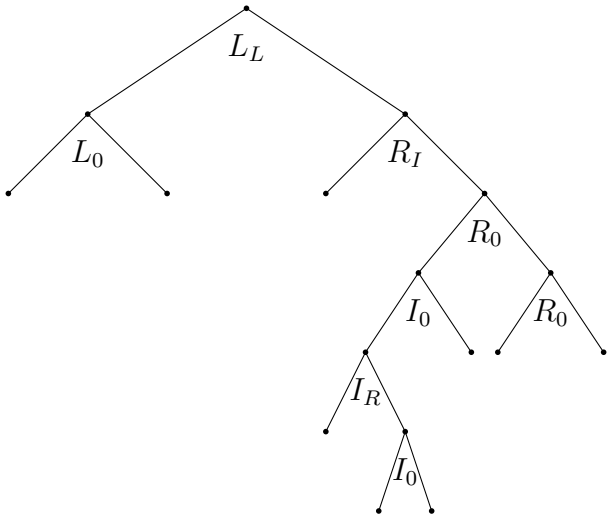

FIGURE 32. A tree labeled according to Fordham's classification.

As Figure 33 shows, however, these caret labels cannot be the basis for a normal form for elements of F since a given word could correspond to multiple elements of F.

We collapsed the five exterior labels into two categories and expanded the two interior labels into four categories as follows:

(1) The root caret is labeled "r,"
(2) all other exterior carets are labeled "e,"
(3) and an interior caret is labeled
    (a) "(" if (i) it is the left child of another interior caret or the child of an exterior caret and (ii) has a right child,
    (b) "a" if (i) it is the left child of another interior caret or the child of an exterior caret and (ii) does not have a right child,
    (c) "b" if (i) it is neither the left child of another interior caret nor the child of an exterior caret and (ii) has a right child, and

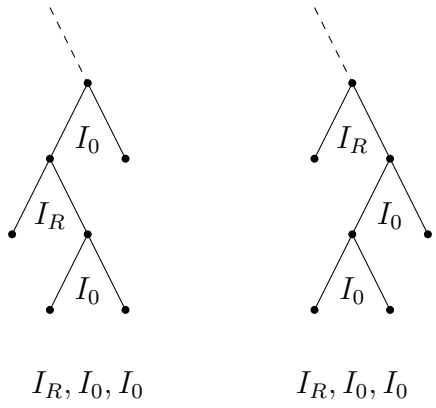

FIGURE 33. Two different interior subtrees that are labeled the same way under Fordham's classification.

(d) ")" if (i) it is neither the left child of another interior caret nor the child of an exterior caret and (ii) does not have a right child.

A word over the alphabet $\Sigma_- = \{r, e, (,), a, b\}$ encoding a binary tree consists of all the caret labels read in infix order. The tree in Figure 34 is encoded by the word "$eea()(ab)raee$".

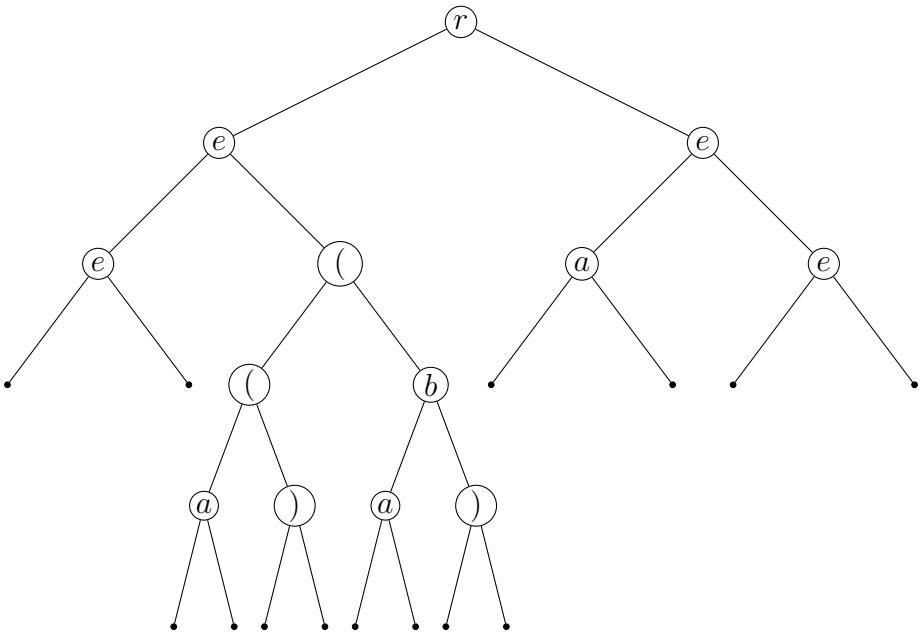

FIGURE 34. Trees are labeled in our notation as shown.

Given a tree pair diagram (T_-, T_+) representing a element of F, we use the alphabet Σ_- to encode the T_- tree and the analogous alphabet $\Sigma_+ = \{r', e', [,], a', b'\}$ to encode the T_+ tree. In the next section, we will construct a language of normal forms for elements of F over the alphabet $\Sigma = \Sigma_- \cup \Sigma_+$.

When considering only interior subtrees, we can translate easily between the notation in [6] and our own. If the caret is labeled "n," the caret is a left child or else

the root caret; if the caret is labeled "i," the caret is a right child. If the leaf directly right of a caret is labeled "N," then the caret has a right child; if the leaf directly right of a caret is labeled "I," then the caret does not. So to translate from the notation in [6] to ours, replace any appearance of "nN" with "(", "iI" with ")", "nI" with "a", and "iN" with "b". Thus, the tree in Figure 30 would be relabeled "$a()(ab)$." This matches the labeling of the first interior subtree in Figure 34.

6. Thompson's Group F is Graph 2-$\mathcal{CF}$

We have now presented the necessary background to prove the following theorem.

Theorem 6.1. *Thompson's group F is graph 2-$\mathcal{CF}$ with respect to the generating set $S = \{x_0, x_1\}$ and the alphabet $\Sigma = \{e, r, e', r', (,), [,], a, b, a', b'\}$.*

To prove this, we construct

- A 2-$\mathcal{CF}$ language $\mathcal{F}$ and a bijective function $\tau : \mathcal{F} \to F$;
- An FSA M_{x_0} that accepts the language $\{(u, v) : \tau(u)x_0 = \tau(v)\}$ and a deterministic PDA M_{x_1} that accepts the language $\{(u, v) : \tau(u)x_1 = \tau(v)\}$.

In fact, since F is isomorphic to the set of reduced tree pair diagrams (which we will denote $\mathcal{T}$) described in Section 5.2, we only need to construct a bijection $\tau : \mathcal{F} \to \mathcal{T}$. We further prove in Lemma 6.8 that $\mathcal{F}$ is not context-free. This means that $\mathcal{F}$ can at best be the basis for a graph 2-$\mathcal{CF}$ structure for Thompson's group F.

6.1. **Word Acceptor.** In this section we define a 2-$\mathcal{CF}$ language $\mathcal{F}$ that is a set of normal forms for Thompson's group F. We begin by constructing a context-free language L'_- of interior subtrees. Next, we construct context-free languages L_- and L_+, which correspond uniquely to binary trees. We combine these languages to define a 2-$\mathcal{CF}$ language L, which corresponds to pairs of binary trees with the same number of carets. Finally, we test that the pair is reduced, yielding $\mathcal{F}$.

6.1.1. *Interior Subtrees.* We first define the language L'_- of strings over the alphabet $\Sigma'_- = \{(,), a, b\}$ that encode interior subtrees. We further show that L'_- is context-free by presenting a NPDA N'_-, shown in Figure 35 such that $\mathcal{L}(N'_-) = L'_-$.

Noting the method for translating between the notation in [6] and our notation described in Section 5.4, we can restate Lemma 6 of [6] in our notation as follows:

Lemma 6.2 (Lemma 6 of [6])**.** *A string β over Σ'_- encodes a non-empty interior subtree tree if and only if*

(1) it begins with "(" or "a",
(2) it ends with ")" or "a",
(3) all of its prefixes α satisfy $|\alpha|_(\geq |\alpha|_)$ (where $|\alpha|_y$ denotes the number of occurrences of "y" in α),
(4) $|\beta|_(= |\beta|_)$.

So L'_- is the set of all strings that satisfy these conditions plus the empty string. Then let N'_- be the following *reduced* NPDA.

(1) $Q = \{q_0, q_1, q_2\}$ is the set of states;
(2) $\Sigma = \{(,), a, b\}$ is the input alphabet;
(3) $\Gamma = \{Z, x\}$ is the stack alphabet;
(4) q_0 is the start state;
(5) $F = \emptyset$ is the set of accept states; and
(6) the transition function δ is defined as follows:
  (i) $\delta(q_0, (, \epsilon) = \{(q_1, Zx)\}$,
  (ii) $\delta(q_0, a, \epsilon) = \{(q_1, Z)\}$,
  (iii) $\delta(q_1, a, Z) = \{(q_1, Z), (q_2, \epsilon)\}$,
  (iv) $\delta(q_1, a, x) = \{(q_1, x)\}$

(v) $\delta(q_1, b, Z) = \{(q_1, Z)\}$,
(vi) $\delta(q_1, b, x) = \{(q_1, x)\}$,
(vii) $\delta(q_1, (, Z) = \{(q_1, Zx)\}$,
(viii) $\delta(q_1, (, x) = \{(q_1, xx)\}$,
(ix) $\delta(q_1,), x) = \{(q_1, \epsilon), (q_2, \epsilon)\}$,
(x) $\delta(q_2, \epsilon, Z) = \{(q_2, \epsilon)\}$.

The NPDA N'_- accepts by empty stack and is depicted as a graph in Figure 35.

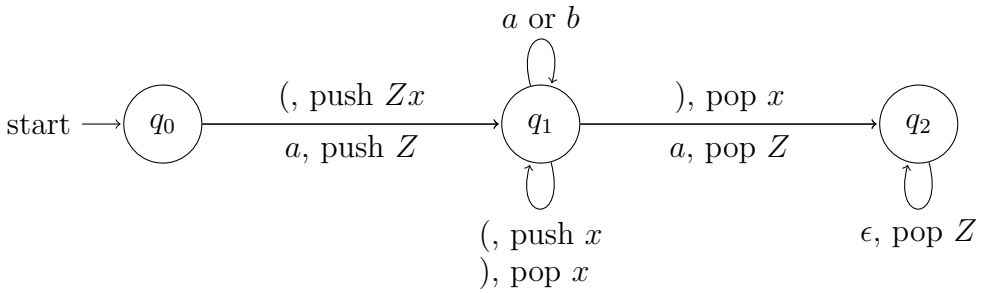

FIGURE 35. The pushdown automaton N'_-, which accepts words over the alphabet Σ'_- corresponding to legitimate interior subtrees. $\mathcal{L}(N'_-) = L'_-$.

We now prove that N'_- indeed accepts L'_-, thus showing that the language of strings that encode interior subtrees is in fact context-free.

Lemma 6.3. *A string β over Σ'_- satisfies the conditions of Lemma 6.2 if and only if $\beta \in \mathcal{L}(N'_-)$. That is to say, $\mathcal{L}(N'_-) = L'_-$.*

Proof. First suppose that $\beta = \gamma_1 \cdots \gamma_n \in \mathcal{L}(N'_-)$. We will show that it satisfies the four conditions of Lemma 6.2.

Condition (1). The only edges emanating from the start state q_0 are labeled "(" and "a". Thus, if β is accepted by N'_-, it must begin with either "(" or "a".

Condition (2). The only edges emanating from the start state q_0 add the symbol Z to the stack. Since N'_- accepts by empty stack, if β is accepted, then at some point N'_- must read a symbol that causes it to remove Z from the stack. There are two ways this can happen. First, if N'_- reads an "a"; in this case, reading any other symbol causes N'_- to reject the string, so β must end in an "a". The second way is by making an ϵ-transition in state q_2. But then N'_- must have arrived at state q_2 by reading a ")". Further, reading any other symbol causes N'_- to reject the string, so β must end in an ")".

Condition (3). Suppose for contradiction that there exists some prefix $\alpha = \gamma_1 \cdots \gamma_j$ of β such that $|\alpha|_(< |\alpha|_)$. Further, suppose without loss of generality that this is not the case for any substring of α. Then γ_j equals ")". Further, if we let α' denote the substring $\gamma_1 \cdots \gamma_{j-1}$, we have that $|\alpha'|_(= |\alpha'|_)$. So after reading γ_{j-1}, the stack can equal either Z or ϵ. But there is no transition of the form $(q_i,), Z)$ or $(q_i,), \epsilon)$, so N'_- must reject β, which is a contradiction.

Condition (4). If β is accepted by N'_-, after the last symbol of β is read the stack must be empty. Since an x is pushed if and only if "(" is read and popped if and only

if ")" is read, then after the last symbol of β is read, N'_- must have read the same number of "(" and ")". Hence, $|\beta|_(= |\beta|_)$.

So we have shown that if $\beta \in \mathcal{L}(N'_-)$, then β satisfies the four conditions of Lemma 6.2. We now show the converse.

Since $\beta = \gamma_1 \cdots \gamma_n$ satisfies conditions (1) and (2), it can have any of four forms.

Case 1: β begins and ends with "a". If β begins with "a", then after reading γ_1, N'_- must be in state q_1 with stack Z. Conditions (3) and (4) ensure that after reading γ_{n-1}, N'_- will still be in state q_1 with stack Z. Then reading $\gamma_n = a$ takes N'_- to state q_2 with an empty stack. Hence, $\beta \in \mathcal{L}(N'_-)$.

Case 2: β begins with "(" and ends with "a". If β beings with "(", then after reading γ_1, N'_- must be in state q_1 with stack Zx. Conditions (3) and (4) ensure that after reading γ_{n-1}, N'_- will be in state q_1 with stack Z. Then reading $\gamma_n = a$ takes N'_- to state q_2 with an empty stack. Hence, $\beta \in \mathcal{L}(N'_-)$.

Case 3: β begins with "a" and ends with ")". If β begins with "a", then after reading γ_1, N'_- must be in state q_1 with stack Z. Conditions (3) and (4) ensure that after reading γ_{n-1}, N'_- will be in state q_1 with stack Zx. Then reading $\gamma_n =)$ takes N'_- to state q_2 with stack Z. Then N'_- can make an ϵ-transition and remove Z from the stack. Hence, $\beta \in \mathcal{L}(N'_-)$.

Case 4: β begins with "(" and ends with ")". If β beings with "(", then after reading γ_1, N'_- must be in state q_1 with stack Zx. Conditions (3) and (4) ensure that after reading γ_{n-1}, N'_- will be in state q_1 with stack Zx. Then reading $\gamma_n =)$ takes N'_- to state q_2 with stack Z. Then N'_- can make an ϵ-transition and remove Z from the stack. Hence, $\beta \in \mathcal{L}(N'_-)$.

So we have shown that if β satisfies the four conditions of Lemma 6.2, then $\beta \in \mathcal{L}(N'_-)$. Hence, a string β satisfies the conditions of Lemma 6.2 if and only if $\beta \in \mathcal{L}(N'_-)$. □

So far, we have defined a mapping

$$\tau'_- : L'_- \to \{\text{interior subtrees}\}.$$

We now show that τ'_- is well-defined. To do this, we need to show that any word $w \in L'_-$ encodes a unique interior subtree. We consider all combinations for pairs of caret labels in $\Sigma'_- = \{a, b, (,)\}$ and show that each combination precisely defines where the corresponding carets must be positioned in the interior subtree. Figure 36 shows every possible combination and the relative placement of the carets they encode.

6.1.2. *Binary Trees.* Now we construct a language L_- over the alphabet

$$\Sigma_- = \{e, r, (,), a, b\}$$

that accepts all and only those words that really correspond to binary trees. First consider a binary tree with no interior subtrees. Such a word will have the form $s_1 r s_2$ where $s_1, s_2 \in \{e\}^*$. A binary tree can have interior subtrees attached to any exterior, non-root caret. So we want to inject words in L'_- into a word of the form $s_1 r s_2$ with the following two conditions: (1) the first symbol must be either e or r and (2) the last symbol must be either e or r. Then the pushdown automaton N_- in Figure 37 accepts all and only such words, which form the context-free language L_-. The state N'_- refers to the pushdown automaton that accepts interior subtrees. The machine cannot make a transition from N'_- unless the substring is a word in L'_-.

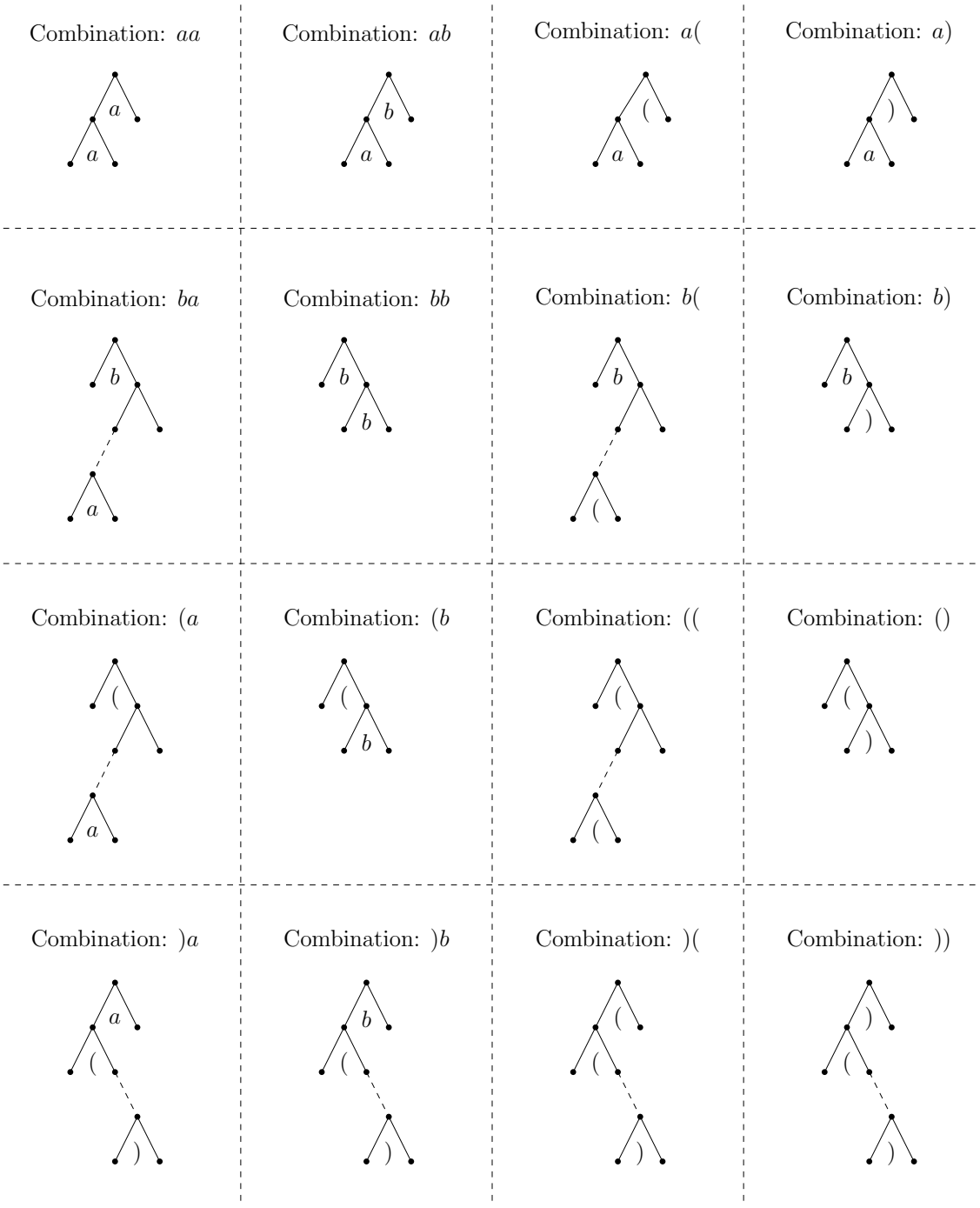

FIGURE 36. This figure shows every possible pair of caret labels in $\Sigma'_{\_}$ and the relative placement of the carets they encode.

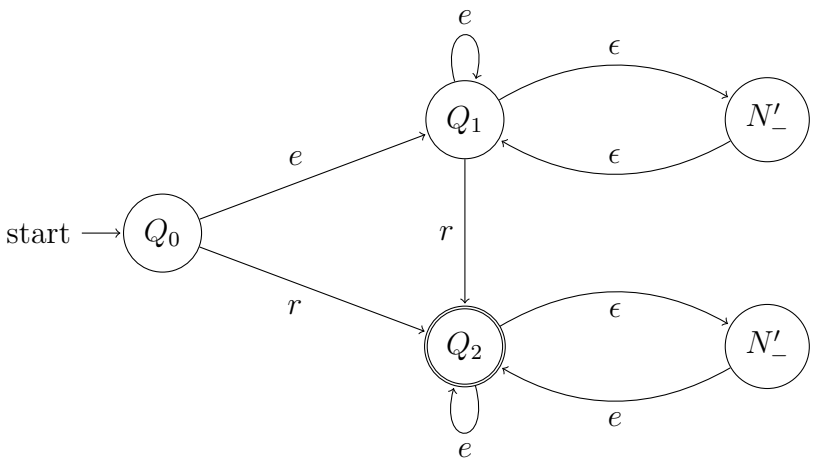

FIGURE 37. The pushdown automaton N_-, which accepts words over the alphabet Σ_- corresponding to legitimate binary trees. $\mathcal{L}(N_-) = L_-$.

In this section, we extended our function τ'_- to a mapping

$$\tau_- : L_- \to \{\text{binary trees}\}.$$

We now argue that τ_- is well-defined. Our argument is similar to that in Section 6.1.1, but now we must consider combinations $\alpha_1\alpha_2$ where

$$\alpha_i \in \{e, r\} \cup L'_-.$$

First consider combinations where $\alpha_1, \alpha_2 \in \{e, r\}$. Then both carets lie along the arch of exterior carets and α_2 is the caret directly to the right of α_1. The case where one α_i encodes an interior subtree is similar. Note that we do not need to consider $\alpha_1\alpha_2$ for $\alpha_1, \alpha_2 \in L'_-$ since L'_- is closed under concatenation, so $\alpha_1\alpha_2 \in L'_-$ also.

6.1.3. *Pairs of Binary Trees.* Let Σ_+ denote the alphabet $\{e', r', [,], a', b'\}$. Then let L_+ be the language over Σ_+ analogous to L_-. By Theorem 2.37, the language

$$L = \{\alpha_1\gamma_1 \cdots \alpha_n\gamma_n : n \in \mathbb{N} \text{ and } \alpha_1 \cdots \alpha_n \in L_-, \gamma_1 \cdots \gamma_n \in L_+\}$$

is 2-$\mathcal{CF}$. Note that L consists of all words where the symbols in even-numbered positions taken alone form a word in L_- and the symbols in odd-numbered positions taken alone form a word in L_+. Further, these two words must be of the same length. Thus each word in L encodes a pair of binary trees with the same number of carets.

6.1.4. *Reduction.* Next we test for reduction in the pair of binary trees. The idea here is to observe the form of a string whose corresponding tree pair is unreduced. For each case, we construct an NFSA that accepts strings of that particular form. Then we intersect L with the complement of the corresponding regular language.

There are three ways that a pair of binary trees can be unreduced: (1) the first exterior caret of each tree is exposed; (2) the last exterior caret of each tree is exposed; (3) each tree has an exposed interior caret of infix number n. Now we observe the form of a string whose corresponding binary tree contains exposed carets.

Remark 6.4. Given a tree pair (T_-, T_+) representing an element of F,

(1) the first exterior caret of the T_- tree is exposed if and only if the word encoding it begins with ee or er;
(2) the last exterior caret of the T_- tree is exposed if and only if the word encoding it ends with re or ee;
(3) an interior caret of the T_- tree is exposed if and only if the word encoding it contains a substring of the form xy for $x \in \{e, r, (, b\}$ and $y \in \{), a\}$.

The first way a pair of trees can be unreduced corresponds to a word beginning with any of the following strings: $ee'xy$ for $x \in \{e, r\}$ and $y \in \{e', r'\}$. The regular language accepted by the FSA X_i in Figure 38 consists of all words beginning with such a string; thus its complement, which is also regular, consists of all words that do not start with such a string.

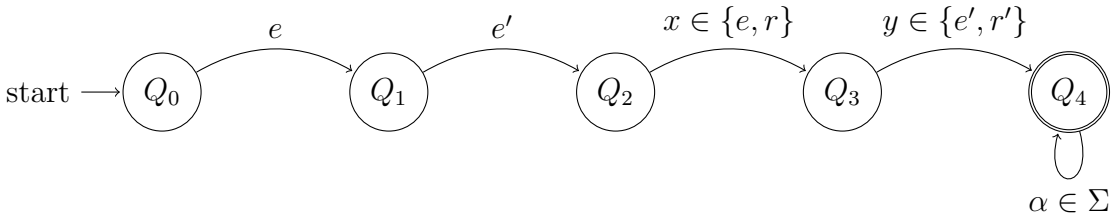

FIGURE 38. The FSA X_i, which accepts words beginning with $ee'xy$ for $x \in \{e, r\}$ and $y \in \{e', r'\}$.

The second way a pair of trees can be unreduced corresponds to a word ending with any of the following strings: $xyee'$ for $x \in \{e, r\}$ and $y \in \{e', r'\}$. The regular language accepted by the NFSA X_f in Figure 39 consists of all words ending in such a string.

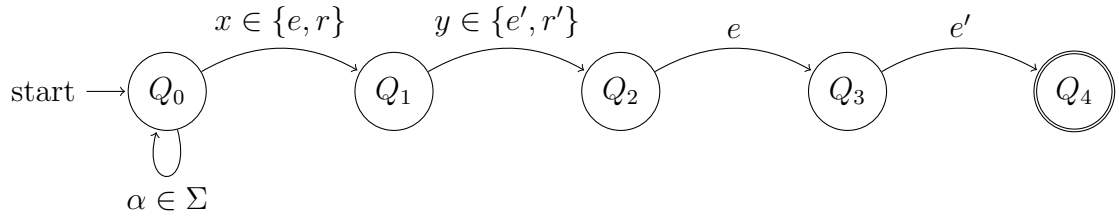

FIGURE 39. The non-deterministic FSA X_f, which accepts words ending with $xyee'$ for $x \in \{e, r\}$ and $y \in \{e', r'\}$.

The third way a pair of trees can be unreduced corresponds to a word containing any of the following strings: $xyzw$ for $x \in \{e, r, (, b\}$, $y \in \{e', r', [, b'\}$, $z \in \{), a\}$, $w \in \{], a'\}$. The regular language accepted by the non-deterministic FSA I in Figure 40 consists of all words containing such a string.

Definition 6.5. We denote $\mathcal{L}(X_i) = R_1$, $\mathcal{L}(X_f) = R_2$, and $\mathcal{L}(I) = R_3$. Then let $R = R_1^c \cap R_2^c \cap R_3^c$. So R is a regular language consisting of all and only strings that do not fail any of the three reduction criteria.

Definition 6.6. Let $\mathcal{F} = L \cap R$.

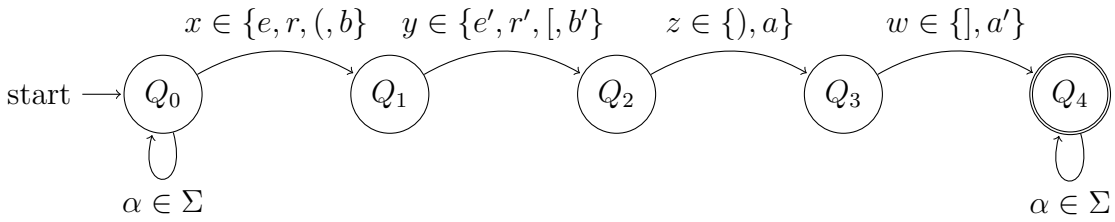

FIGURE 40. The FSA I, which accepts words containing $xyzw$ for $x \in \{e, r, (, b\}$, $y \in \{e', r', [, b'\}$, $z \in \{), a\}$, $w \in \{], a'\}$.

Since L is a 2-$\mathcal{CF}$ language, by Theorem 2.35, $\mathcal{F}$ is also 2-$\mathcal{CF}$. Further, since each word in L encodes a pair of binary trees with the same number of carets, each word in $\mathcal{F}$ encodes a reduced pair of binary trees, that is, encodes an element of F.

In this section, we defined a function

$$\tau_+ : L_+ \to \{\text{binary trees}\}$$

that is analogous to τ_- and combined τ_- and τ_+ into a mapping

$$\tau : L \to \{\text{pairs of binary trees}\}.$$

Next, we restricted τ to $\mathcal{F} \subset L$ so that τ would map onto the set of *reduced* pairs of binary trees, that is, $\mathcal{T}$. Since τ_- and τ_+ are both well-defined, clearly τ must be also. Then $\tau|_{\mathcal{F}}$ is well-defined too. So

$$\tau : \mathcal{F} \to \mathcal{T}$$

is a well-defined bijection, implying the following fact.

Fact 6.7. *$\mathcal{F}$ is a set of normal forms for Thompson's group F.*

So we have shown that $\mathcal{F}$ is a 2-context-free language that is a set of normal forms for elements of F. The following lemma shows that $\mathcal{F}$ can at best be the basis for a graph 2-context-free structure for Thompson's group F; that is, it shows that $\mathcal{F}$ is not context-free.

Lemma 6.8. *$\mathcal{F}$ is not context-free.*

Proof. Assume for contradiction that $\mathcal{F}$ is context-free. Then by the pumping lemma for context-free languages there exists some integer $p \geq 1$ such that any word $w \in \mathcal{F}$ with $|w| \geq p$ can be written as $w = qrstu$ where the substrings q, r, s, t and u satisfy the following properties.

(1) $|rst| \leq p$,
(2) $|rt| \geq 1$, and
(3) $qr^n st^n u$ is a word in $\mathcal{F}$ for all $n \geq 0$.

Consider the string

$$w = ee' \underbrace{(a'(a' \cdots (}_{2p+1} \underbrace{[a[\cdots a[}_{2p+1} \underbrace{)a')a' \cdots)}_{2p+1} \underbrace{]a] \cdots a]}_{2p+1} rr'.$$

The substring corresponding to the T_- tree is

$$w_- = e(^{p+1}a^p)^{p+1}a^p r,$$

which clearly is a word in L_- and the substring corresponding to the T_+ tree is

$$w_+ = e'a'^p[^{p+1}a'^p]^{p+1}r',$$

which clearly is a word in L_+. Further, w satisfies the three reduction criteria. So w is a word in $\mathcal{F}$. Since the string qr^nst^nu must also be a word in $\mathcal{F}$ for any natural number n, $|r|$ and $|t|$ must both be even so that qr^nst^nu still alternates symbols from Σ_- and Σ_+. So since $|rt| \geq 1$, both rt must contain at least one of the following symbols: $(,),[,]$. Since $|rst| \leq p$, it follows that rt can contain at most two of those symbols, but cannot contain both (and) or both [and]. There are two cases to consider here:

(i) rt contains just one of $(,),[,]$;
(ii) rt contains either
  (a) (and [,
  (b) [and), or
  (c)) and].

In case (i), qr^nst^nu either does not have the same number of ('s and)'s or does not have the same number of ['s and]'s for $n > 1$. In case (ii), qr^nst^nu will not have the same number of ('s and)'s *or* the same number of ['s and]'s for $n > 1$. In both cases, then, qr^nst^nu is not a word in $\mathcal{F}$ for $n > 1$, a contradiction. So $\mathcal{F}$ is not context-free. □

6.2. **Multipliers.** In this section we discuss the effect of multiplication by the generators x_0 and x_1 and present the: an FSA M_{x_0} and a PDA M_{x_1}. Given two strings $\alpha_1\alpha_2\cdots\alpha_m, \beta_1\beta_2\cdots\beta_n \in \mathcal{F}$, the automata take as input the convolution $\otimes(\alpha_1\cdots\alpha_m, \beta_1\cdots\beta_n)$. We let "$(x, x)$" be shorthand for "$(x, x)$ for $x \in \Sigma$".

For each generator, there are several cases for the effect of multiplication by the generator on a given element. For a given element x and generator s, first we construct FSA $N_1^s, N_2^s, \ldots, N_k^s$ that detect which case the element lies in. Next, for each case, we construct FSA $N_1'^s, N_2'^s, \ldots, N_k'^s$ that accept (u, v) when $f(u) = x$ and $f(v) = xs$. (In the case of x_1, one of the $N_i'^{x_1}$ is actually a NPDA.) Then our multiplier M_s accepts the regular language

$$(\mathcal{L}(N_1^s) \cap \mathcal{L}(N's_1)) \cup (\mathcal{L}(N_2^s) \cap \mathcal{L}(N_2'^s)) \cup \ldots \cup (\mathcal{L}(N_k^s) \cap \mathcal{L}(N_k'^s)).$$

In some cases, it is easier to construct an FSA that combines N_i^s and $N_i'^s$. That is, we construct an FSA B_i^s with

$$\mathcal{L}(B_i^s) = \mathcal{L}(N_i^s) \cap \mathcal{L}(N_i'^s).$$

We make it clear in which cases we construct both FSA N_i^s and $N_i'^s$ and in which cases we just construct B_i^s.

6.2.1. *Multiplication by x_0.* Generally, multiplication of an element $f \in F$ by x_0 rotates the T_- tree clockwise about the root caret, as seen in Figure 41.

More precisely, there are three cases to consider based on the binary tree representative of f.

Case 1. Suppose first that the root caret of the T_- tree has no left child. Then we must add a left child to the left-most caret of both the T_- and T_+ trees, making the pair unreduced, and then perform the rotation. An example is shown in Figure 42.

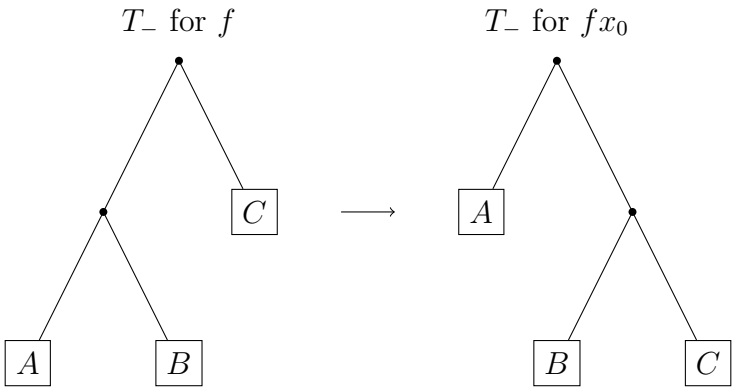

FIGURE 41. Clockwise rotation of the T_- tree of $f \in F$ induced by multiplication by x_0. Here, A, B, and C represent possibly empty subtrees.

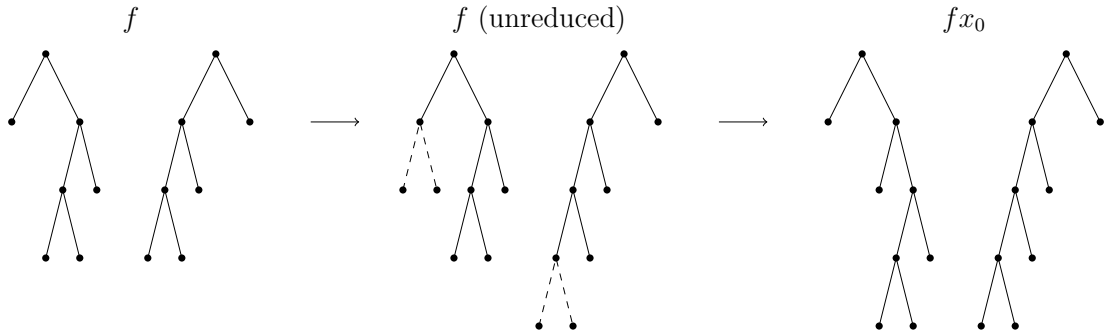

FIGURE 42. Caret addition and clockwise rotation induced by multiplication by x_0. In this case, f is represented by the string $re'ae'er'$ and fx_0 by the string $re'ee'ae'er'$.

If the root caret of the T_- tree has no left child, then the string representing the element begins with r. The FSA $N_1^{x_0}$ (Case 1 for multiplication by x_0) in Figure 43 accepts only pairs of strings (u, v) representing elements $f(u), f(v)$ when $f(u)$ lies in Case 1, that is, when u beings with "r".

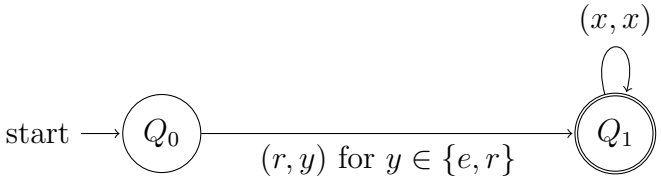

FIGURE 43. The FSA $N_1^{x_0}$, which accepts pairs of strings (u, v) representing elements $f(u), f(v)$ when $f(u)$ lies in Case 1 of multiplication by x_0.

The effect of multiplication by x_0 on the binary tree representative for f is to add an exterior caret just after the root caret of the T_- tree. Hence, the string representing the T_- tree changes from

$$ru \to reu$$

where u is some string of caret types. The effect on the T_+ tree is to add a left child to the left-most exterior caret. So the string representing the T_+ tree changes from

$$v \to e'v$$

where v is some string of caret types. In sum, the string representing f changes from

$$rw \to re'ew$$

where w is some string of caret types. The FSA that accepts the language

$$\{\otimes(w_1, w_2) : w_1 = rw, w_2 = re'ew\}$$

is extremely complicated. The key problem is to check whether two strings differ by a shift, with a particular finite string inserted. We will call an FSA that checks for this a "Shift Checker." To illustrate that this really is possible with an FSA, we construct a simple Shift Checker: an FSA over $\Sigma = \{a, b, c\}$ that accepts the language $\{\otimes(w_1, w_2) : w_1 = cw, w_2 = cabw\}$ for $w \in \{a, b\}^*$. The FSA is presented in Figure 44.

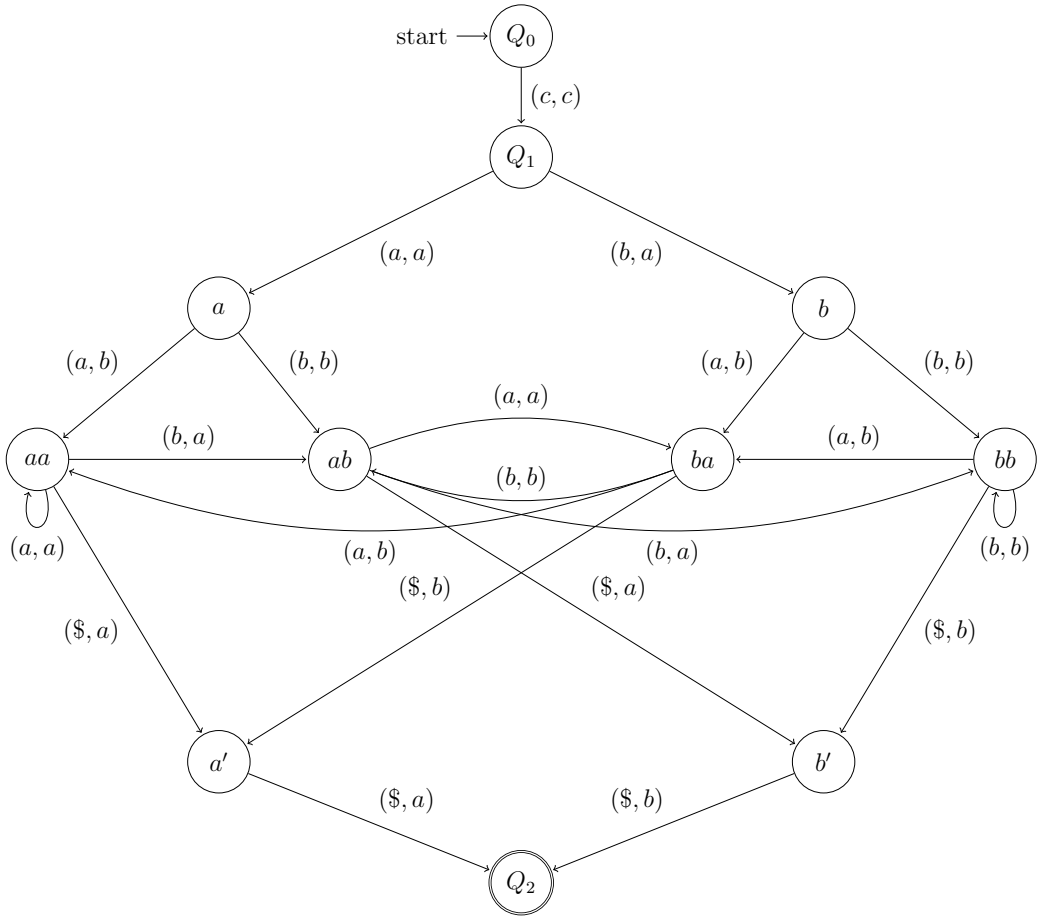

FIGURE 44. The FSA that accepts the language $\{\otimes(w_1, w_2) : w_1 = cw, w_2 = cabw\}$.

The core of this FSA lies in the four states aa, ab, ba, and bb. Each of these states records the last two symbols that have been read in w_1 and ensure that these are the next two symbols of w_2. In general, the more symbols that are possible in w and the more symbols that are inserted into the word, the more states the FSA will have. So we could construct an FSA $N_1^{'x_0}$ that would accept pairs (w_1, w_2) when inserting $e'e$ into w_1 at the beginning yields w_2, but it would have over $6^2 = 36$ states. In these cases, we will assume that we have a shift checker and represent it schematically in the FSA $N_i^{'s}$. The schematic for $N_1^{'x_0}$ is depicted in Figure 45.

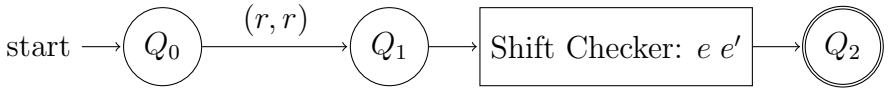

FIGURE 45. Schematic for the FSA $N_1^{'x_0}$, which recognizes Case 1 of multiplication by x_0. $N_1^{'x_0}$ accepts the language $\{\otimes(w_1, w_2) : w_1 = rw, w_2 = re'ew\}$.

Case 2. Now suppose that the root caret of the T_- tree does have a left child. If the following three conditions hold

(i) the root caret of the T_- tree has no right child;
(ii) the left child of the root caret of the T_- tree has no right child; and
(iii) the last exterior caret of the T_+ tree is exposed

then we must eliminate redundant carets after rotation, as shown in Figure 46.

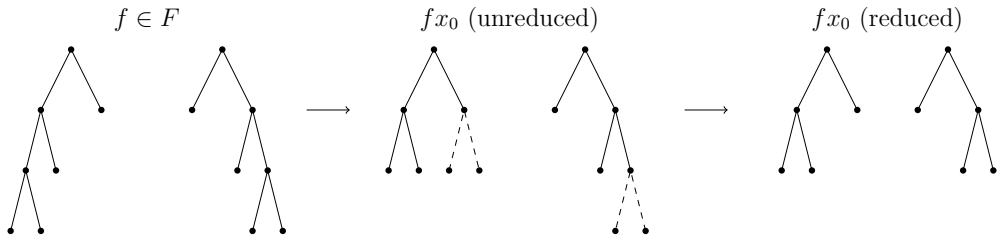

FIGURE 46. Caret elimination induced by multiplication by x_0.

Given conditions (i) and (ii), the string corresponding to the T_- tree of f must end with the substring er. Also, given condition (iii), the string corresponding to the T_+ tree must end with the substring $e'e'$ or $r'e'$. So the string representing f must end with the substring $e\gamma re'$ for $\gamma \in \{e', r'\}$. Multiplication by x_0 makes the left child (with label e) of the root caret in the T_- tree its right child and then cancels it with the final caret (with label e') of the T_+ tree. The FSA $B_2^{x_0}$ depicted in Figure 47, which combines $N_2^{x_0}$ and $N_2^{'x_0}$, captures these constraints and recognizes this case of multiplication by x_0.

Note that it is very easy to construct $N_2^{x_0}$. We must only check that given some input (u, v), u ends with the substring $e\gamma re'$ for $\gamma \in \{e', r'\}$. The FSA $N_2^{x_0}$ is depicted in Figure 48.

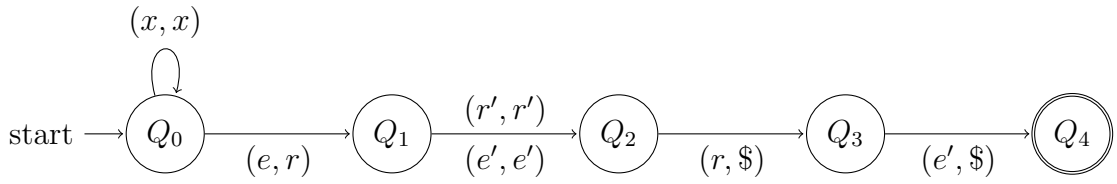

FIGURE 47. The FSA $B_2^{x_0}$ that recognizes Case 2 of multiplication by x_0.

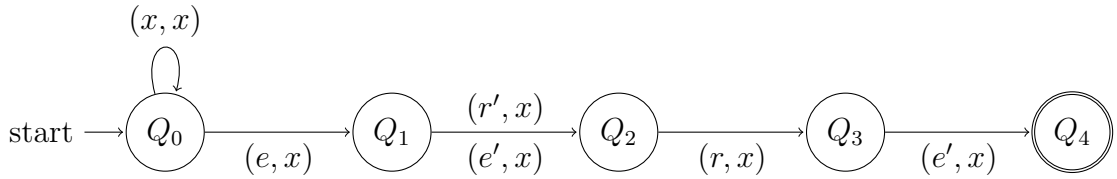

FIGURE 48. The FSA $N_2^{x_0}$ that detects Case 2 of multiplication by x_0. Here x denotes any symbol in Σ.

Case 3. If the tree pair does not belong to Case 1 or Case 2, then we know that we can construct an FSA $N_3^{x_0}$ to accept this case since the regular languages are closed under union and complement. In particular,

$$\mathcal{L}(N_3^{x_0}) = \left(L(N_1^{x_0}) \cup L(N_2^{x_0})\right)^c.$$

In this case, we perform the multiplication as in the general case, described above. That is to say, we simply rotate the T_- tree according to Figure 41. The substrings corresponding to the subtrees A, B, and C are exactly the same in the strings corresponding to the T_- trees of f and fx_0. However, in the string representing the T_- tree of f, we read

(1) the substring corresponding to A,
(2) e,
(3) the substring corresponding to B,
(4) r, and then
(5) the substring corresponding to C,

while in the string representing the T_- tree of fx_0, we read

(1) the substring corresponding to A,
(2) r,
(3) the substring corresponding to B,
(4) e, and then
(5) the substring corresponding to C.

(The string representing the T_+ does not change.) The FSA $N_3^{\prime x_0}$, depicted in Figure 49, recognizes this case of multiplication by x_0.

So for each of the three cases, we can construct FSA to detect which case elements are and FSA that test for multiplication by x_0 within that case. So we take the union of all of these languages to obtain the regular language accepted by M_{x_0}, the multiplier for x_0.

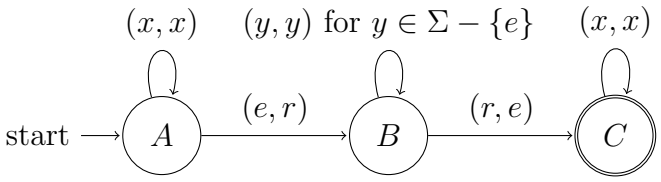

FIGURE 49. The FSA $N_3^{'x_0}$, which recognizes Case 3 of multiplication by x_0.

6.2.2. *Multiplication by x_1.* Multiplication of an element $f \in F$ by x_1 rotates the T_- tree counter clockwise about the root caret's right child, as seen in Figure 50.

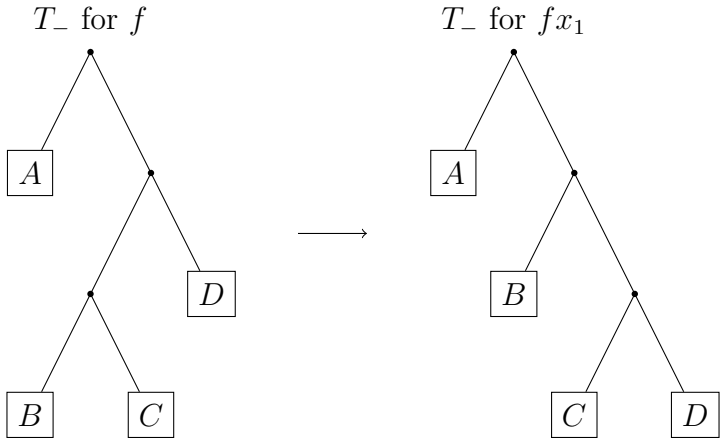

FIGURE 50. Clockwise rotation of the T_- tree of $f \in F$ induced by multiplication by x_1.

With multiplication by x_1 there are five cases to consider.

Case 1. First suppose that the root caret of the T_- tree has no right child. That is, suppose the subtrees labeled B, C, and D are all empty. Then we must add a right child to the right-most carets of the T_- and T_+ trees and we must also give this right child a left child. Then we perform the rotation. This process is shown in Figure 51.

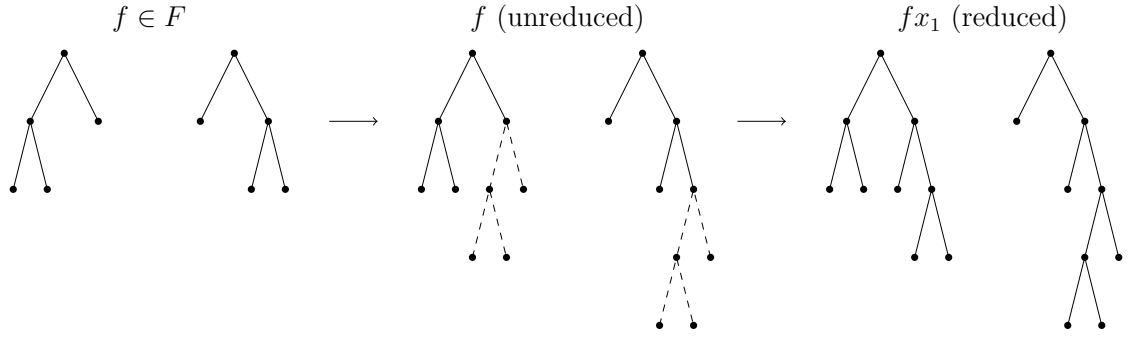

FIGURE 51. Addition of two carets and clockwise rotation induced by multiplication by x_1.

Since the root caret of the T_- tree has no right child, the string corresponding to f ends in rr' or re'. As with Case 1 of multiplication by x_0, this is easy to check with

an FSA $N_1^{x_1}$. As seen in Figure 51, multiplication by x_1 simply adds two exterior carets to the T_- tree and an interior and exterior caret to the T_+ tree. So the string corresponding to f changes from

$$w_1 r\gamma \to w_1 r\gamma e a' e e'$$

with $\gamma \in \{e', r'\}$. The FSA depicted in Figure 52, which combines $N_1^{x_1}$ and $N_1'^{x_1}$, captures the constraint and recognizes this case of multiplication by x_1.

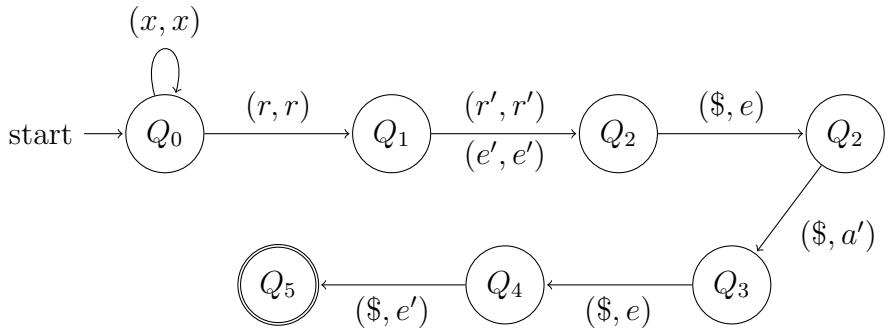

FIGURE 52. The FSA that recognizes Case 1 of multiplication by x_1.

Case 2. Next suppose that the root caret of the T_- tree has a right child, which does not have a left child. Then we must add a left child to the right child of the root and to the corresponding caret in the T_+ tree, making the pair unreduced. Then we perform the rotation. This process is shown in Figure 53.

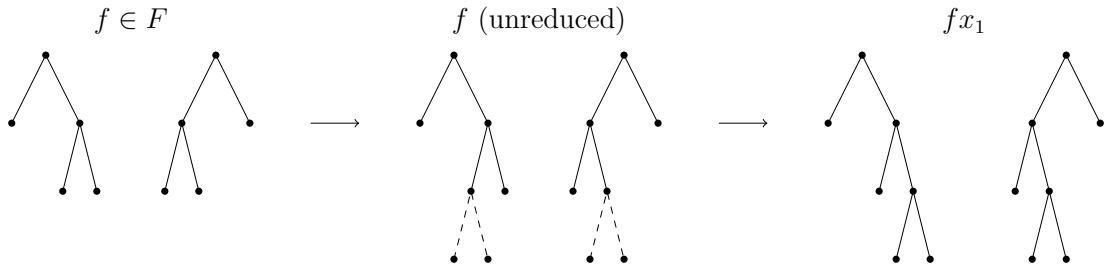

FIGURE 53. Caret addition and clockwise rotation induced by multiplication by x_1.

Since the root caret of the T_- tree has a right child, which does not have a left child, the string corresponding to the T_- tree contains the substring re. So every string corresponding to an element in this case contains the substring rxe for $x \in \Sigma_+$. It would be easy to construct an NFSA $N_2^{x_1}$ to check that a string satisfies this.

The effect on the T_- tree for f is to add an exterior caret just after the root caret. So the string corresponding to the T_- tree will change from

$$u_1 r e u_2 \to u_1 r e e u_2.$$

Let n be the infix number of the root caret of the T_- tree. Then the effect on the T_+ tree for f is to add a caret after the caret with infix number n. If the caret with infix number n is type $e', r', [$, or b', then the effect is to insert a caret of type a'. However, if the caret with infix number n is type a' or $]$, then the new caret is added

as a right child of this caret. Hence, the caret with infix number n changes to type [or b', respectively, and the new caret is type]. Thus, the string corresponding to the T_+ tree will change from

$$\gamma_1 \cdots \gamma_n \gamma_{n+1} \cdots \gamma_z \to \gamma_1 \cdots \gamma_n a' \gamma_{n+1} \cdots \gamma_z$$

if $\gamma_n \in \{e', r', b', [\}$

$$\gamma_1 \cdots a \gamma_{n+1} \cdots \gamma_z \to \gamma_1 \cdots [\,] \gamma_{n+1} \cdots \gamma_z$$

if $\gamma_n = a'$, or

$$\gamma_1 \cdots] \gamma_{n+1} \cdots \gamma_z \to \gamma_1 \cdots b'] \gamma_{n+1} \cdots \gamma_z$$

if $\gamma_n =]$.

So the string corresponding to f will change from

$$w_1 r \gamma_n w_2 \to w_1 r \gamma_n' e \sigma w_2,$$

where γ_n' and σ depend on γ_n as described above. The FSA $N_2^{'x_1}$, depicted schematically in Figure 54, recognizes this case of multiplication by x_1.

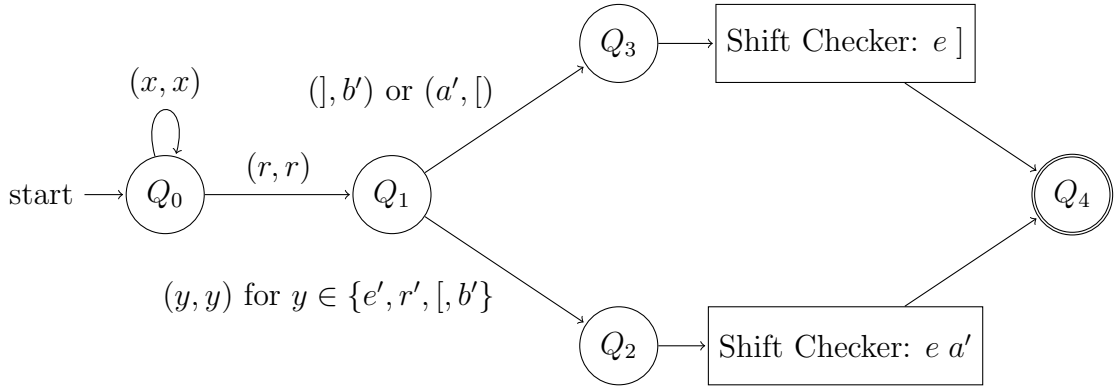

FIGURE 54. Schematic for the FSA $N_2^{'x_1}$, which recognizes Case 2 of multiplication by x_1.

Case 3. Next suppose that the root caret of the T_- tree has a right child that does have a left child. If the following three conditions hold

(i) the right child of the root caret does not have a right child;
(ii) the left child of the right child of the root caret has no children; and
(iii) the string corresponding to the T_+ tree ends in $r'e'e'$ or $e'e'e'$

then we must eliminate two redundant carets after rotation, as shown in Figure 55.

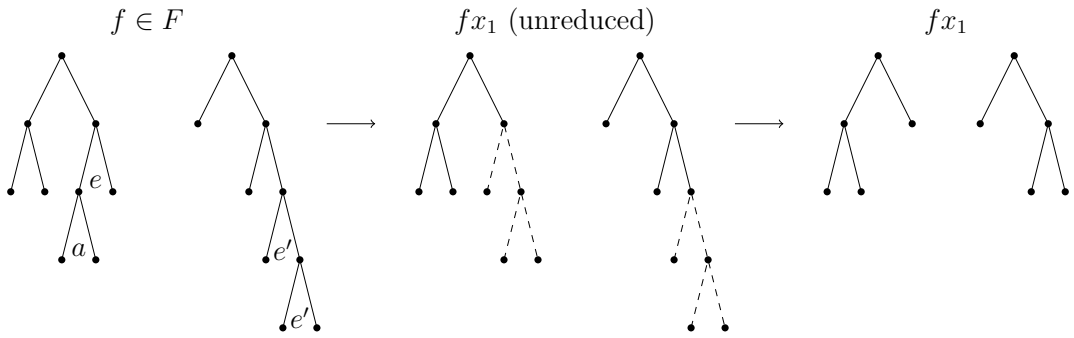

FIGURE 55. Elimination of two carets induced by multiplication by x_1.

Given conditions (i) and (ii), the string corresponding to the T_- tree ends in rae. As seen in Figure 55, multiplication by x_1 essentially eliminates the carets labeled a and e in the T_- tree and the carets labeled e' and e' in the T_+ tree. So the string corresponding to the T_- tree will change from

$$u_1 rae \to u_1 r$$

and the string corresponding to the T_+ tree changes from

$$v_1 \gamma e' e' \to v_1 \gamma$$

for $\gamma \in \{e', r'\}$. Thus, the string corresponding to f changes from

$$w_1 r \gamma a e' e e' \to w_1 r \gamma$$

for $\gamma \in \{e', r'\}$.

The FSA depicted in Figure 56, which combines $N_3^{x_1}$ and $N_3^{'x_1}$, captures these constraints and recognizes this case of multiplication by x_1.

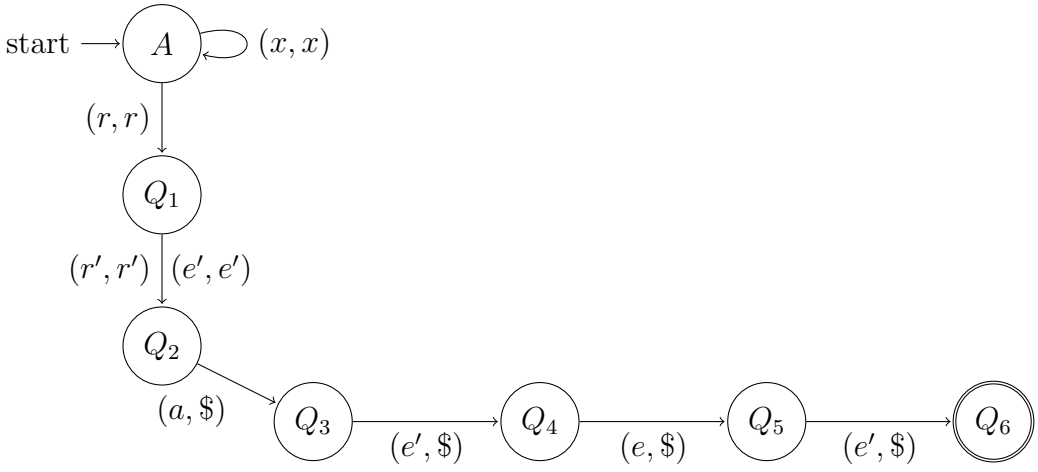

FIGURE 56. The FSA that recognizes Case 3 of multiplication by x_1.

Case 4. Next suppose that the conditions of Case 3 hold with two differences: the left child of the right child of the root caret can now have a left child; and the string corresponding to the T_+ tree ends in $r'e'$ or $e'e'$ instead of $r'e'e'$ or $e'e'e'$. Then we must eliminate a single redundant caret after rotation, as shown in Figure 57.

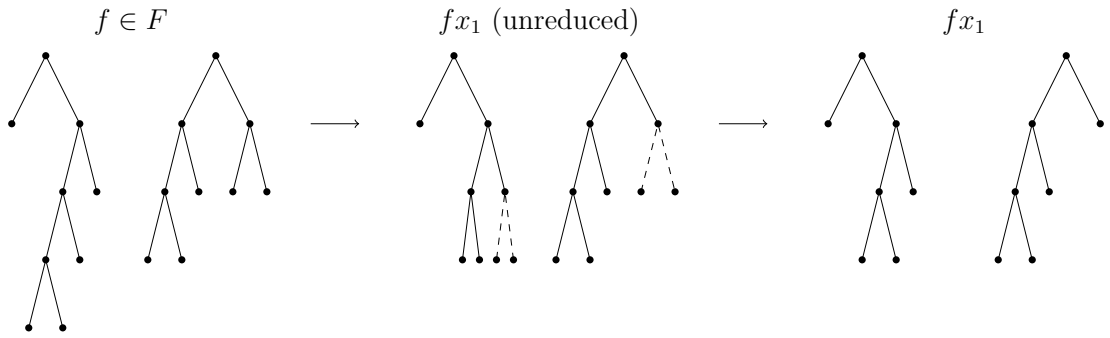

FIGURE 57. Caret elimination induced by multiplication by x_1.

In this case, the string corresponding to the T_- tree must end in ae and furthermore no other e's can appear between this e and the r. In this case, the string representing T_- changes from

$$u_1 r n_1 a e \to u_1 r n_1 e$$

where n_1 corresponds to an interior subtree. The string representing T_+ changes from

$$v_1 \gamma e' \to v_1 \gamma$$

with $\gamma \in \{e', r'\}$. Thus, the string corresponding to f changes from The FSA depicted in Figure 58, which combines $N_4^{x_1}$ and $N_4^{'x_1}$, captures these constraints and recognizes this case of multiplication by x_1.

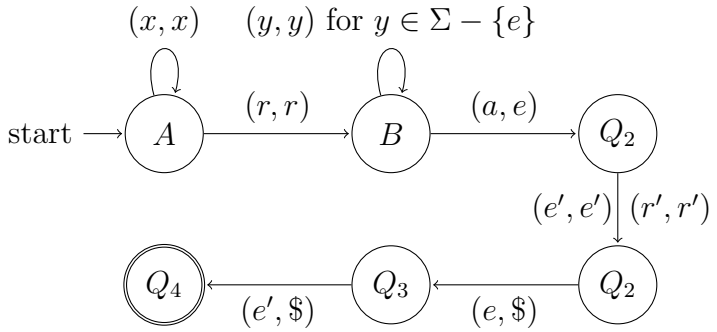

FIGURE 58. The FSA that recognizes Case 4 of multiplication by x_1.

Case 5. Finally, if the tree pair does not belong to Cases 1-4, then, as in Case 3 of multiplication by x_0, we know that we can construct an FSA $N_5^{x_1}$ to accept Case 5 since the regular languages are closed under union and complement. In this case, we perform the multiplication as in the general case, described above. That is to say, we simply rotate the T_- tree according to Figure 50.

Since multiplication does not change the T_+ tree, we do not need to construct $N_5^{'x_1}$ over the alphabet $\Sigma \times \Sigma$. For clarity, we will ignore the string corresponding to the T_+ tree and construct a simplified version of $N_5^{'x_1}$ that takes as input strings in $\Sigma_- \times \Sigma_-$. In this case, we cannot detect multiplication by x_1 with an FSA: $N_5^{'x_1}$ is a deterministic PDA.

The substrings corresponding to the subtrees A, B, C, and D are exactly the same in the strings corresponding to the T_- trees of f and fx_1. However, in the string representing the T_- tree of f, we read

(1) the substring corresponding to A,
(2) "r",
(3) the substring corresponding to B,
(4) either "(" or "a" (corresponding to the left child of the right child of the root caret),
(5) the substring corresponding to C,
(6) "e", and then
(7) the substring corresponding to D,

while in the string representing the T_- tree of fx_1, we read

(1) the substring corresponding to A,
(2) "r",
(3) the substring corresponding to B,
(4) "e",
(5) the substring corresponding to C,
(6) "e", and then
(7) the substring corresponding to D.

Each string is exactly the same up until (4). At this point, we will read the pair $\big((, e\big)$ or $\big(a, e\big)$. Now we must check that the caret represented by "(" or "a" here is in fact the left child of the right child of the root caret. First, we must use a stack that works the same as the stack in N'_-. That is, it pushes an x when it reads "(" and pops an x when it reads ")". The stack must be empty when we read the pair $\big((, e\big)$ or $\big(a, e\big)$.

If we read $\big(a, e\big)$, the subtree C is empty. Then the caret corresponding to "a" really is the left child of the right child of the root caret in f if and only if the next caret type we read is be "e". So we must read (e, e) next.

On the other hand, if we read $\big((, e\big)$, we cannot read $\big((, (\big)$ or (a, a) on an empty stack, or else the caret corresponding to this "(" or "a" would actually be the left child of the right child of the root caret. Furthermore, the root of C has type "b" or ")" in the string corresponding to f. However, this caret becomes the child of an exterior caret in fx_1, so in the string corresponding to fx_1, this caret will be type "(" or "a", respectively. So before we have read another "e" in the string corresponding to f, we must read either $\big(b, (\big)$ or $\big((, a\big)$.

The deterministic PDA $N_5^{'x_1}$, depicted in Figure 59, captures all of these constraints and recognizes this case of multiplication by x_1. $N_5^{'x_1}$ accepts by final state and has stack alphabet $\Gamma = \{Z\}$. In most transitions, the stack contents do not matter. We only reference the stack contents in transitions where they do matter: in the transitions $Q_1 \to Q_2$, $Q_1 \to Q_4$, $Q_4 \to$ FAIL, $Q_5 \to$ FAIL, and $Q_5 \to Q_6$.

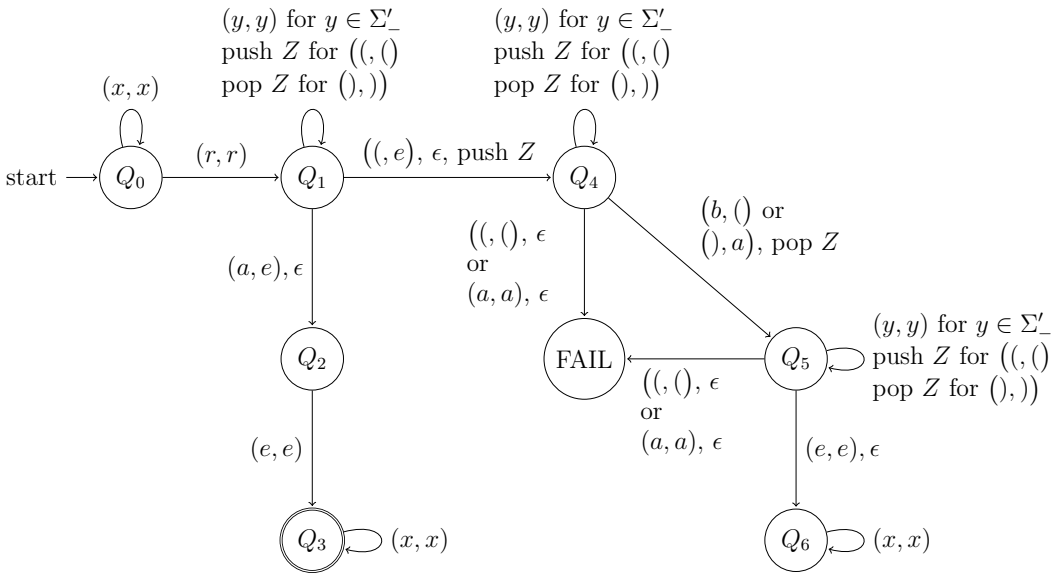

FIGURE 59. The deterministic PDA $N_5^{'x_1}$, which recognizes Case 5 of multiplication by x_1.

So for each of the five cases, we can construct FSA to detect which case the elements are and PDA that test for multiplication by x_1 within that case. So we take the union of all of these languages to obtain the context-free language accepted by M_{x_1}, the multiplier for x_1.

So we have constructed a 2-$\mathcal{CF}$ language $\mathcal{F}$ and an onto function $f : \mathcal{F} \to F$ along with an FSA multiplier M_{x_0} and a PDA multiplier M_{x_1}. This concludes our proof of Theorem 6.1.

References

[1] James Michael Belk. *Thompson's group F*. ProQuest LLC, Ann Arbor, MI, 2004. Thesis (Ph.D.)–Cornell University.

[2] Tara Brough. Groups with poly-context-free word problem, preprint. `http://arxiv.org/abs/1104.1806`, 2011.

[3] José Burillo. Quasi-isometrically embedded subgroups of Thompson's group F. *J. Algebra*, 212(1):65–78, 1999.

[4] J. W. Cannon, W. J. Floyd, and W. R. Parry. Introductory notes on Richard Thompson's groups. *Enseign. Math. (2)*, 42(3-4):215–256, 1996.

[5] Sean Cleary and Jennifer Taback. Thompson's group F is not almost convex. *J. Algebra*, 270(1):133–149, 2003.

[6] Murray Elder, Éric Fusy, and Andrew Rechnitzer. Counting elements and geodesics in Thompson's group F. *J. Algebra*, 324(1):102–121, 2010.

[7] David B. A. Epstein, James W. Cannon, Derek F. Holt, Silvio V. F. Levy, Michael S. Paterson, and William P. Thurston. *Word processing in groups*. Jones and Bartlett Publishers, Boston, MA, 1992.

[8] S. Blake Fordham. Minimal length elements of Thompson's group F. *Geom. Dedicata*, 99:179–220, 2003.

[9] V. S. Guba and M. V. Sapir. The Dehn function and a regular set of normal forms for R. Thompson's group F. *J. Austral. Math. Soc. Ser. A*, 62(3):315–328, 1997.

[10] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[11] Olga Kharlampovich, Bakhadyr Khoussainov, and Alexei Miasnikov. From automatic structures to automatic groups. `http://arxiv.org/abs/1107.3645`, 2011.