Compiling Java in Linear Nondeterministic Space

by

Joshua Donnoe

B.S., Kansas State University, 2016

A THESIS

submitted in partial fulfillment of the
requirements for the degree

MASTER OF SCIENCE

Department of Computer Science
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2018

Approved by:

Major Professor
Torben Amtoft

Copyright

Abstract

Shannon's and Chomsky's attempts to model natural language with Markov chains showed differing gauges of language complexity. These were codified with the Chomsky Hierarchy with four types of languages, each with an accepting type of grammar and automaton. Though still foundationally important, this fails to identify remarkable proper subsets of the types including recursive languages among recursively enumerable languages. In general, with Rice's theorem, it is undecidable whether a Turing machine's language is recursive. But specifically, Hopcroft & Ullman show that the languages of space bound Turing machines are recursive. We show the converse also to be true. The space hierarchy theorem shows that there is a continuum of proper subsets within the recursive languages.

With Myhill's description of a linear bounded automata, Landweber showed that they accept a subset of the type 1 languages including the type 2 languages. Kuroda expanded the definition making the automata nondeterministic and showed that nondeterministic linear space is the set of type 1 languages. That only one direction was proven deterministically but both nondeterministically, would suggest that nondeterminism increases expressiveness. This is further supported by Savitch's theorem. However, it is not without precedent for predictions in computability theory to be wrong. Turing showed that Hilbert's Entscheidungsproblem is unsolvable and Immerman disproved Landweber's belief that type 1 languages are not closed under complementation.

Currently, a major use of language theory is computer language processing including compilation. We will show that for the Java programming language, compilability can be computed in nondeterministic linear space by the existence of a (nondeterministic) linear bounded automaton which abstractly computes compilability. The automaton uses the traditional pipeline architecture to transform the input in phases.

The devised compiler will attempt to build a parse tree and then check its semantic properties. The first two phases, lexical and syntactical analysis are classic language theory tasks. Lexical analysis greedily finds matches to a regular language. Each match is converted to a token and printed to the next stream. With this, linearity is preserved. With a Lisp format, a parse tree can be stored as a character string which is still linear. Since the tree string preserves structural information from the program source, the tree itself serves as a symbol table, which normally would be separately stored in a readable efficient manner. Though more difficult than the previous step, this will also be shown to be linear. Lastly, semantic analysis, including typechecking, and reachability are performed by traversing the tree and annotating nodes.

This implies that there must exist a context-sensitive grammar that accepts compilable Java. Therefore even though the execution of Java programs is Turing complete, their compilation is not.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I'd first like to thank my wife, Amanda, and our two boys, Ben & Sam, who kept me motivated, active, and awake respectively. We survived graduate school and are better for it.

I also want to thank Dr. Torben Amtoft for advising me in both my senior project and this thesis and the rest of my committee, Dr. Dan Andresen, who helped me with my first poster, and Dr. Robby, who along with Dr. John Hatcliff, I've had the pleasure of serving as a Teaching Assistant these last two years.

Although, I picked Computer Science over Mathematics for my Masters work, I very much benefitted from researching with Drs. Nathan Albin & Pietro Poggi-Corradini. None of this would have been possible without my readmission which only happened through the hard work and excellent advice of my undergraduate advisor Dr. Tom Muenzenberger.

Lastly, I want to thank my colleagues for helping me refine my ideas. This helped me both avoid writing and be more productive when I did write.

This thesis was composed in LaTeX using Jesse A. Tov's proof template, Till Tantau's Ti*k*Z package, and Axel Sommerfeldt's caption package.

Dedication

In life, my father would say "I'm thinking about hurting myself" whenever I tried to talk about math. I can only imagine what he would have said about this. I dedicate this thesis to Joseph Elkins Brown. My greatest hope in life is that I am as good a father to his grandsons as he was to me.

Chapter 1

Introduction

The purpose of this thesis is to improve the literature surround formal language theory in four ways

- To resolve conflicting and imprecise notation across sources
- To argue relevance of infrequently used sources
- To draw additional conclusions from existing conclusions
- To present new findings

1.1 Differences in the Literature

Between sources in the literature, there are differences in both how certain concepts are represented symbolically and how constructs are defined. This thesis can serve as a rosetta stone to translate between the notations in the various sources.

1.1.1 Differing Notations

Through the literature there are concepts that are shared between researchers but are represented differently. In some cases, the subjects of these differences are very simple including what type of construct is a member of a language. Some sources refer to these as "strings" while other prefer to use “words”. In Section 5.1, individual tokens are enumerated by finding separating whitespace and other delimiters. The term “word” implies some delimitation and is better suited to these tokens rather that the entire “string”.

Another example is how to represent the zero length string (or word). Although researchers agree that all zero length strings are identical, they disagree on what symbol to use. Early sources view languages algebraically as semi-groups where concatenation is an operation with this empty string as the identity but without inverses. Later sources simply view the empty string as the unique string with zero length.

When discussing the space hierarchy theorem, [1] uses limits to express that two functions scale are of different orders. Using asymptotic notations, this is equivalent to little-oh notation which was introduced by [2]’s first edition [3] after publication of [1]. Later discussion of the space hierarchy theorem in [4] using this notation.

1.1.2 Differing Definitions

Of the two major introductory texts on formal language, [1; 4], there are several notational differences. On some matters other sources also disagree. Between these various notations, automatic translation is difficult especially since similar notation have different meaning in the different definitions.

Since the invention of the Turing machine, researchers have modified the definition to better suit their needs. Although all the definitions are equally expressive, at face value, they appear different and notation for one may not match another. Between texts, there is no standard tuple definition of Turing machines. This is discussed in Chapter 3.

Relating back to the findings of [5; 6], it is an open question if nondeterminism grants an expressive difference in bounded. Since [4] uses the definition of a linear bounded automaton from [5] over the definition from [6], the findings of [6] are unusable.

The grammatical restrictions that created the Chomsky Hierarchy restrict the empty string from languages. This complicated discussion of languages because accepting automata have no such inherent prohibition on accepting the empty string. Section 2.2.2 lists consequences of this conflict and attempts to resolve it.

Section 4.2 discusses space complexity classes which have conflicting definitions. Where [1] defines a class in terms of machines with certain properties, [4] defines a class in terms of languages that are accepted by machines with certain properties. This subtle point is the basis of Rice's theorem, which distinguishes semantic properties of a language from concrete properties of TMs themselves [1; 4].

1.2 Underutilized Sources

Editorial choices between [1] and its third edition [7] demonstrate assumptions that certain facets of formal language are too obscure or of too little use to include in an introductory text. The discussion of bounded automaton, space complexity, and type one languages was removed from [7] even though the results of [8] answers a major open question and was awarded the Gödel Prize [9].

1.3 Additional Conclusions

1.3.1 Space Complexity and LOGSPACE

Section 4.2 lists several theorems involving space complexity which only apply to complexity classes above LOGSPACE. Across several application, this seems to be a barrier for useful behavior.

Also Section 2.2.4 shows a 2-partitioned proper subset of the regular language form a field over intersection and symmetric difference.

1.4 New Contributions

New concepts are developed and theorems proven in Chapters 4 & 5. The classification of end behavior and theory leading up to Theorem 4.1 extends [1; 4]; the first direction is proven in [1], but the second direction is a new contribution. This continues into a formalization of bounded automata and the use of asymptotic notation to populate complexity classes. Section A.4 demonstrates a technique to formally prove correctness of Turing machines, a task that is rarely attempted.

Most importantly, Chapter 5 demonstrates that Java compilability can be decided in space proportional to the input and is therefore a context-sensitive language. This is not done to manage memory scarcity but rather to show the an upper bound on the computational complexity of determining compilability.

1.5 Open Problems

Nondeterminism is a recurring theme in computer science and across the field, there are several open questions that relate to that. Limited to this literature, questions include whether the subset relationship in Savitch's Theorem is strict or not. If it is strict , then there are some deterministic problems that can not be simplified as much with nondeterminism. Another question involves comparing [5]'s findings with [6]'s. Again this reduces to whether nondeterminism grants an expressive difference.

Chapter 2

The Chomsky Hierarchy

By the 1950's, research had began on automata, initially only Markov chains[1], to describe languages [10; 11]. The finite state models are insufficient to describe natural language because they admit phrased that are "ungrammatical," i.e. "Sandwich a ate John" [10, p. 113]. More precise models can be created by looking at sliding windows of multiple previous symbols. However these models are simply larger Markov chains. For some languages, there can not exist a Markov chain that accepts the language, that is, every Markov chain generates some string not in the language or fails to generate a string in the language [10–12]. An example would be the language of strings containing some number of 0s and then an equal numbers of 1s [10; 13].

2.1 Grammars

A grammar, roughly "a finite set of 'rewriting rules'", can better approximate natural language [10, p. 140] [11]. These rules operate on two disjoint alphabets: the symbols over which the language is defined, and symbols used internally only during the rewriting process [1; 4–6; 10; 13; 14]. Each rule is of the form $\varphi \to \psi$, read φ produces ψ, where φ is a string of terminal symbols and non-terminal symbols that "can be rewritten as" [13, p. 140] another string of terminal symbols and non-terminal symbols ψ. The language of the grammar is

[1]Figure 2.1 is an example.

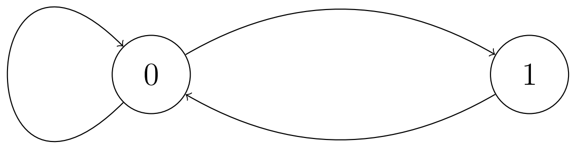

Figure 2.1: *Golden Mean Markov Chain* [12, p. 101]
Starting from either state, the machine follows a random outgoing edge and outputs the character at the new state. For this Markov chain, the sequence "11" never appears. The name comes from the fact that the entropy is the logarithm of the golden mean [12]. This measure of entropy was defined as capacity in [11].

the set of possible terminal strings[2] $\sigma_1\sigma_2\sigma_3\cdots\sigma_n$ that are the result of any finite number of derivations, applications of productions, from the start non-terminal [1; 4–6; 10; 13; 14].

The Chomsky Hierarchy is a sequence of grammar types induced by a sequence of grammatical constraints. Each grammar type is a subset of the previous. With each additional restriction, the languages in the families become simpler to describe [13]. Each language type has a corresponding type of automata that can accept languages of the type. Fittingly, as the constraints on the grammars increase, the machines become simpler and more properties of languages are decidable [1; 4–6; 13–15]. Table 2.1 gives a summary of this section.

2.1.1 Unrestricted or Type 0 grammars

The production rules of an unrestricted grammar allow for productions from substrings of terminals and non terminals [1; 13]. An important functionality of unrestricted grammars is deriving strings shorter that the original. This means that the lengths of strings in a series of derivations is not necessarily increasing[3]. When attempting to derive a string s, from the grammar G, if an intermediate derivation s', is longer than s, there may be further derivations

[2]Sometimes referred to as word [14] or sentence [6; 10; 13; 14]
[3]The terms "increasing" [16] or "monotonically increasing" [2] correspond to $\geq$ while "strictly increasing" corresponds to $>$.

that are shorter and lead to s. This indicates that it may not be possible to demonstrate that w can not be derived from G. That all members of the language can be shown to be in the language even if some candidates can not be shown to not be in the language is the defining characteristic of recursively enumerable[4] languages [1; 4; 13; 15]. The automaton for unrestricted grammars is the Turing machine which is discussed in Chapter 3 [1; 4; 13].

2.1.2 Context-sensitive or Type 1 Grammars

The productions of context-sensitive grammars[5] can only produce from non-terminals and may not produce ε[6] [1; 6; 13]. Substrings immediately left and right of the non-terminal can give context to the selection of productions that are allowed, for example $\varphi_1 A \varphi_2 \to \varphi_1 \omega \varphi_2$ even if $\varphi'_1 A \varphi'_2 \not\to \varphi'_1 \omega \varphi'_2$. Since $\omega \neq \varepsilon$, $|\omega| \geq |A|$ and therefore no application of a rule can produce a shorter string than the input [1; 4–6; 13]. Another form for CSG matches the definition of unrestricted grammars except the result of any production must not be shorter than the original [1; 6; 13].

Unlike for unrestricted grammars, if an intermediate derivation w', is longer than the desired string w, then the derivation will never lead to w. The search for a derivation for a candidate word w, can be represented by a tree similar to Figure 2.2 with a root representing the start symbol. Each node has a child for every possible direct derivation. With a finite length and finitely few production rules, every derivation has finitely few children. Ignoring the children of derivations that are longer than w yields a finite tree and any search is guaranteed to terminate [2; 17].

This shows that a non-member of a CSG can always be detected, or equivalently that a CSG is corecursively enumerable [1]. Since the intersection of recursively enumerable languages and co recursively enumerable languages is the set of recursive languages [1; 15], languages defined by CSG are recursive [1; 4; 6; 13]. However, no amount of failure to find strings in the language indicates if the language is empty. It is undecidable if a CSG is

[4]Sometimes referred to as decidable [4]
[5]Hereafter CSG
[6]Sometimes referred to as I in older literature since it is the identity for concatenation [6; 13] or Λ [15].

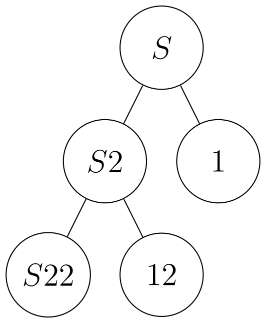

Figure 2.2: *Derivation Search Tree*
The grammar $S \to S2 \mid 1$ does not derive 21 because no such node exists in the tree and any additional nodes can not lead to a derivation of 21.

empty [1; 4]. The automaton for CSGs is the linear bounded automaton [1; 5; 6]. Chapter 4 is devoted to general bounded automata.

2.1.3 Context-free or Type 2 Grammars

Named to be distinguished from their superset, context-free grammars[7] follow the same pattern as CSG except the contexts are both ε [13]. This means that a non-terminal has the same productions, no matter where it appears, $\varphi_1 A \varphi_2 \to \varphi_1 \omega \varphi_2 \iff \varphi'_1 A \varphi'_2 \to \varphi'_1 \omega \varphi'_2$ [1; 4; 13; 14]. Since each derivation simply expands a single non terminal into one or more additional nodes, a parse tree can be constructed where the children of a non terminal are the produced symbols. Figure 2.2 also serves as a parse tree.

Because no particular context is needed, a context-free grammar can be modified to only have productions to single terminals or pairs of non-terminals and still accept the same language [1; 4; 13]. Using the pigeonhole principle which says "...that if p pigeons are placed into fewer than p holes, some hole has to have more than one pigeon in it" [4, 78], an arbitrarily long sequence of elements from a finite set must repeat an element. This fact accompanied with Chomsky normal form shows that for sufficiently long strings in a language, substrings can be removed or repeated, and the result is also in the language [1; 4].

[7]Hereafter CFG

This pumping lemma[8] shows it is decidable whether a CFL is empty, finite[9], or infinite [1] and its contrapositive can be used to prove certain languages are not context free [1; 4].

By construction of new grammars, CFLs can be shown to be closed under union. And by the pumping lemma, the intersections of some CFLs can be shown to not be CFLs[10]. This shows that the CFLs are not closed under intersection and by the closure lemma on page 38, CFLs must not be closed under complementation. Also, it is not decidable if two CFLs are the same [1; 4].

2.1.4 Regular or Type 2 Grammars

The regular grammars describe the original phenomenon of languages produced by Markov chains [10; 13]. The automaton for regular languages is the finite automaton which only stores the current state [1; 4; 13; 14; 18; 19]. These languages adhere to a restricted version of the pumping and are closed under complementation and symmetric difference. It is therefore decidable if two regular languages are equal [1; 4].

	Automaton	New Decidable Property
Type 0	Turing Machine	
Type 1	Linear Bounded Automaton	Non membership
Type 2	Push Down Automaton[11].	Emptiness
Type 3	Finite Automaton	Equality

Table 2.1: *Grammatical Comparisons* [1; 4–6; 13]

2.2 Refining The Chomsky Hierarchy

Only identifying 4 types of languages, the Chomsky hierarchy lacks precision to identify certain language families. In several of these types, there exist subtypes with additional properties.

[8]The pumping lemma for CFL is given in the proof for Lemma A.5 on pg. 45

[9]Section 2.2.4 has a detailed discussion of finite languages.

[10]Examples are given in Section 2.2.3

[11]The PDA is discussed at length in [1; 4]

2.2.1 Recursive languages among Recursively enumerable languages

Recursiveness is not particular to CSLs, the set of recursive languages is an undecidable strict subset of recursively enumerable languages [1; 4; 20] and strict superset of CSLs [1; 13]. Chapter 4 discusses recursive languages and their relationship with CSLs.

2.2.2 ε acceptance

Either formulation of CSGs quickly shows that ε can not be derived. In Section 2.1.1, it was shown that, that if in general, a grammar can produce ε, then it is unrestricted. This would mean that only unrestricted grammars can derive ε. In Table 2.1, the PDA was introduced which can accept ε [1; 4]. However, as a type 1 language, a type 2 can not contain ε [13]. This demonstrates a conflict in the literature.

There is a clever solution taken from one interpretation of Chomsky normal form where the start symbol can produce ε but the start symbol can not be produced [4]. With this, a derivation sequence may only lead to ε if the first, and only, step is to produce ε from the start symbol. By adopting this technique, the CSLs can be expanded to accept ε but not be unrestricted. In CFG and regular grammars, ε productions do not add expressiveness [1; 4].

2.2.3 Deterministic Context Free Languages

In this section, the languages in Table 2.2 will be illustrative. Techniques useful to proving the types of these languages can be found in the Appendices or [1; 4].

Name	Definition	Type
L_1	$0^i1^i0^*$	DCFL
L_2	$0^*1^i0^i$	DCFL
L_3	$L_1 \cup L_2 = 0^i1^j0^k$	CFL
	such that $i = j \vee j = k$	
L_4	$\overline{L_1}$	DCFL
L_5	$\overline{L_2}$	DCFL
L_6	$\overline{L_3} = L_4 \cap L_5$	CSL

Table 2.2: *Languages Related to Context Freedom*

In general, a string in a CFL can be derived in multiple ways, however the various derivation sequences may all correspond to the same parse tree [1; 4; 14]. This undecidable condition [1; 4] is not necessary, but CFL that meet it are described as unambiguous [1; 4; 14]. Some unambiguous CFLs are also deterministic[12] and can be accepted by deterministic pushdown automaton[13] [1; 4]. Unlike for the finite automaton, nondeterminism increases expressiveness of the pushdown automaton. In a nondeterministic finite automaton, sets of states can be combined to make deterministic behavior [1; 4; 18]. But in pushdown automata, state transitions can be accompanied with side effects to the automaton and deterministic state sets can not be constructed.

This intuition does not constitute a proof. However, the expressive difference can be shown using an additional closure of DCFL. From Table 2.2, a DPDA for L_1 is always in a single state. A companion DPDA can be constructed with a complementary accepting set to accept L_4. The generalization of this construction shows that DCFLs are closed under complementation. If the DCFLs are a subset of CFLs and have additional closure properties, then they must be a proper subset [1; 4; 14]. DCFL are not closed under union because two nondisjoint, incomparable DCFLs L_1 and L_2, will have two different parse trees for words in their intersection. To formally show this, the pumping lemma can show that the complement of the union of the DCFLs L_6 is not a CFL. In Section 4.2.1, it will be shown that the complement of a CFL is a CSL.

DCFL are important in computer science because parse trees determine semantics of a program. If two parse trees exist for the program, then it has two different meanings. By refactoring the grammar, the productions can be considered in a deterministic order and only a single correct parse tree exists [18]. Also, unlike general CFLs, it is decidable if a DCFL is regular [1, citing Stearns and Valiant].

2.2.4 Finite and Cofinite Languages among Regular Languages

There are also special types of regular languages that warrant consideration.

[12]Hereafter DCFL
[13]Hereafter DPDA

Definition 2.1. *A language is finite if and only if it contains finitely few strings.*

Definition 2.2. *A language is cofinite if and only if its complement is finite.*

Both of these families of languages are indeed regular[14] and also closed under union and intersection[15]. A further result is that the union of finite and cofinite languages is closed under union, intersection, and complementation[16]. This proper subset of the regular languages with such closure properties should enjoy its own designation in the hierarchy. The proposed refinements of types 2 and 3 can be combined to produce the fact that since DCFL are closed under complementation, it is decidable if a DCFL is cofinite.

[14] Discussion of the regularity of finite languages is limited to a vague assertion in [10, p. 115].

[15] Proven on page 41

[16] Proven on page 44

Chapter 3

Turing Machines

3.1 The Basic Turing Machine

Originally developed to study provability, "...the Turing machine has become the accepted formalization of an effective procedure" [1, p. 147][20]. The Turing machine[1] was shown to accept recursively enumerable languages [1; 4; 13; 15]. Although all equivalent, the literature has several different models for TMs. In this work, a new model will be selected as an amalgamation of these. Before an enumeration of the differences, an explanation of the functionality is warranted.

3.1.1 Explanation

The TM is an automaton consisting of a finite set of control states, Q, a tape of memory squares arbitrarily long to the right, and a read/write head that moves along the tape. The set of symbols allowed on the tape Γ includes $\textvisiblespace$ and Σ the alphabet of the input. To start, the tape is set so that the input is left aligned on the tape and every other square is $\textvisiblespace$. At each step, based on the state of the machine and the symbol at the position of the head, the machine will change control state, write a character in the square that the head is at, and move the head forward or back one square.

[1] Hereafter TM

These possible steps can be represented as a relation or set of tuples $(q,\ \gamma,\ q',\ \gamma',\ D)$ [1; 4–6; 15; 20]. When the machine enters certain accepting states[2], the machine will halt and accept the string. While only the control state and head symbol are used to determine the next action, the sequence of tape symbols is also necessary to fully describe the machine's configuration[3].

3.1.2 Head Movement and Writing

In its original presentation, the TM always moved to an adjacent square and always wrote out a symbol, even if to overwrite a symbol with itself [1; 20]. In one presentation, a TM can change control state and either move the head or write but not both [15]. This will lead to control state space twice as large but does not weaken the machine. More recent work has allowed the option of the head not moving during a step [1; 4–6]. The functionality could be build by adding two more states to a mandatory implementation [4]. The model presented will have the option of not moving.

3.1.3 Nondeterminism

By some authors, much like finite or pushdown automata, a TM may have multiple next steps to take [1; 4; 6; 8; 21]. In these cases, if there is a choice that can lead the machine to accepting state, then that choice is taken and the machine will eventually halt and accept. If there is no such choice, then the selection process is irrelevant and the selection is arbitrary[4] because the automaton will never accept.

Definition 3.1. *An execution is a possible sequences of choices a TM could make and corresponds to the sequences of configurations at each step.*

With no bound on space, nondeterminism does not increase expressiveness [1]. In Chapter 4, the choice matters because though an open question, it is likely that nondeterminism offers

[2]Sometimes referred to as final [5; 6]

[3]Sometimes referred to as instantaneous representations [1] or instantaneous description [21]

[4]This notion is equivalent to the fork & join paradigm which creates duplicates of the machine each taking a different choice and accepts if any machine halts and accepts the input [1; 4].

an expressive difference in bounded space.

An important feature of nondeterminism is that the intermediate values can be generated nondeterministically and verified instead of being computed. This reduces the problem of "finding a proof of a statement ... [to] efficiently verifying a proof" [1, p. 321]. In Figure A.2, a factor of a number is generated nondeterministically if it exists proving compositeness. Section 4.2.1 discusses using nondeterminism to quickly reject words.

3.1.4 Rejection

There is an important distinction between a string and an execution, an attempt to accept a string. In the same way that accepting states halt execution and accepts, rejecting states would halt an execution and reject [4]. If δ is allowed to be partial, rejecting states are unneeded since any transition to a rejecting state can be removed and execution "falls off"[5] the machine instead of explicitly being rejected [1; 15]. An execution can also be rejected if it attempts to read left of the start. A machine rejects a word if all executions of the word were rejected.

3.1.5 Basic Model for A Turing Machine

A Turing machine can be defined by a tuple of $(Q,\ q_0,\ \Gamma,\ \text{\textvisiblespace},\ \Sigma,\ \delta,\ Q_{accept})$ where:

1. Q the finite control state space.
2. $q_0 \in Q$ is the starting state.
3. Γ is the finite tape alphabet.
4. $\text{\textvisiblespace} \in \Gamma$ is the blank symbol.
5. $\Sigma \subset \Gamma \setminus \{\text{\textvisiblespace}\}$ is the finite input alphabet.
6. $\delta \mid Q \times \Gamma \to 2^{Q\ \times\ \Gamma\ \times\ \{L,\ R,\ S\}}$ is the non-deterministic transition function.

[5]In an object-oriented languages, this would amount to catching a `NullPointerException` and returning `false` rather than checking some condition and returning `false` when the check fails which is a bad programming habit [22].

7. $Q_{accept} \subset Q$ is the set of accepting states.

To start, the tape has the form $\Sigma^* \text{␣}^*$ where Σ^* is the input w. This justifies $\text{␣} \notin \Sigma$. A common choice in designing TM is to have a special first character so executions know where the boundary is. If starting with w on the tape, there exists some sequence of choices that lead the machine to enter an accepting state, then the machine makes those choices and eventually halts and accepts. In Section 2.1.1, the TM was said to be the accepting automaton for unrestricted grammars which were shown to not alway be able to identify non members. A famous result known as the halting problem showed that in general, it is undecidable if a given TM will halt while computing a given string [1; 4; 15; 20].

3.2 Turing Machine Modifications

With a complete definition of a TM, modified versions of TMs can be examined. As a demonstration of the power of the TM, many modification that intuitively should increase the expressiveness fail to do so. Therefore, the TM is a simple yet powerful tool to describe an algorithm. The goal is to eventually abstract away the TM definition but still understand how the underlying TM behaves. The first three modifications are basic and can be done directly while the rest rely on these basic modification. The results of Chapter 4 depend on these modification no requiring additional tape.

3.2.1 Multiple Inputs

With an additional delimiter symbol, tuples or sequences of strings can be input to a TM together. Since the specification of a TM is finite, it can be used as input into another TM. Classic examples of this include the universal TM which takes a TM spec, $\mathcal{M}$ and a string, w, and simulated $\mathcal{M}$ computing w [1; 4; 20] and the proof that it is undecidable that whether a linear bounded automaton[6] accepts the empty language [1; 4].

[6]Discussed in Section 4.2.2

3.2.2 Augmented State Space

The states of a TM can be considered to hold both control information but also an element from some other set. This partition of information and behavior is only abstract, the actual state space is the same size. This could be useful to calculate the parity for the length of a sequence. In the state corresponding to going right, the information would be the parity calculated so far and would toggle with each additional symbol read. Another example would be to compute palindrome by storing the character that should be in the other side [1]. These can be considered modes of executions. By repeatedly applying this construction, a k-tuple mode can be attached to the states.

3.2.3 Augmented Tape Alphabet and Tape Compression

Each tape square can also hold a finite amount of additional information. Elements in $\Gamma \setminus \Sigma$ including $\textvisiblespace$ can be expressed as a tuple from $(\Sigma \cup \Sigma') \times \Sigma''$ assuming $\Sigma'' \neq \emptyset$. This effectively allows two tape symbols to be written in the space of one input symbol [1]. By repeatedly applying this construction, k symbols can be written to a single tape square. This shows that computation that uses $k * n$ squares can be done in n squares. Subsequent TM modifications will rely on this to build more complex TMs [1; 4]. Section 4.2 uses this to establish tape compression.

3.2.4 Inserting or Removing Tape squares

Spare squares can be added or removed by shifting blocks of symbols. First all blocks to be moved are marked by replacing their symbol γ with a pair $(\gamma,\ marked_1)$. In the direction of the shift, the destination of the first symbol is also marked as the destination. The head starts at the first symbol and copies the symbol into its mode and blanking the square to unmark it. To see if this is the last symbol to copy, the next square is checked for the mark and the result is also stored in a mode. At the square marked as the destination, the mode symbol is copied down and the square is unmarked. If this was the last symbol to be copied then the task is complete. Otherwise while traveling back to retrieve the next symbol, the next square is marked as the destination [1].

3.2.5 Bi-Infinite Tape

Previous results can be combined to simulate a bi-infinite tape $\mathcal{T}$, where there is no left most tape square on a regular tape T. Usage of T's tape is as follows:

- $T[0] = \mathcal{T}[0]$
- $\forall i > 0, T[i] = (\mathcal{T}[i], \mathcal{T}[-i])$

Also a mode stores whether the forward or reverse portion of $\mathcal{T}$'s tape is being read. This is needed when deciding which symbol in the pair to pass to $\delta_{\mathcal{M}}$ and how to interpret head movement instructions. In the reverse mode, head movements are reversed. The mode changes when the head reaches the left end of T which is detectable since $T[0]$ is unique for storing a single symbol [1].

3.2.6 Multiple Tapes

The results of Section 3.2.3 can be considered as multiple tapes but a single head that reads all of them. A TM like machine $\mathcal{T}$, with a head on each tape can be simulated on a TM T. T's extended tape alphabet is $\Gamma' = \Gamma \times (\sigma_1 \cup \textvisiblespace) \times \Gamma \times (\sigma_1 \cup \textvisiblespace)$. The second and fourth elements of these tuple indicates the presence or absence of the head on that tape, so for only one square is the value σ_1. T's head traverses the tape left to right finding the symbols at each of $\mathcal{T}$'s heads, and after finding the symbols, stored as modes, traverses the tape right to left moving the simulated heads and rewriting the symbols [1].

Another construction involves a delimiter character separating the tapes and the two heads are marked. Whenever the left tape needs an additional blank, the right subtape is shifted to the right [4].

3.2.7 Output

Beyond accepting or rejecting, a TM has a tape configuration if it halts. For each accepting execution, the prefix of the tape until the first $\llcorner\!\lrcorner$ can be interpretted as an output of the machine. This allows for a TM M, to be treated as a partial multi valued function $\mathcal{M} \mid \Sigma^* \rightharpoonup (\Gamma \setminus \llcorner\!\lrcorner)^* \ \cup \ \emptyset$. Here $\mathcal{M}$ captures all of the behavior of M. $\mathcal{M}$'s domain is the set of string that M halts on. $L(M) = \{w \mid \mathcal{M}(w) \neq \emptyset\}$. The values of $\mathcal{M}$ can be used to set the output of other TMs or as a final output of a calculation.

Chapter 4

Bounded Automata

In Section 3.1.5, the TM was introduced with an infinite tape in one direction. With this definition, Section 3.2 generalized the TM without adding expressiveness. In this Chapter, a restriction on the tape length for the TMs will be considered.

Definition 4.1. *A bounded automaton or BA is a TM that first begins by allocating an amount of tape based on the length of the input and never exceeds that allocation. Any execution that attempts to do so is rejected.*

Definition 4.2. *A bounded languages is one that can be accepted by a BA.*

The tape length is determined by a function $s \mid \mathbb{N} \to \mathbb{N}$. From the input w, $s(|w|)$ squares are allocated between boundary characters which are not in the tape alphabet $<$ and $>$. Any state that is seeking the end, will detect the $>$ and move back left having not changed the symbol. The bulk of this Chapter is about properties of languages induced by s including the special case of CSL.

Lemma 4.1. *For a TM M, If an execution x is not the only accepting execution of M, then a machine M' identical to M except for rejecting x has the same language.*

Proof. The execution, x, is determining some string, w. As the TM is defined, w will be accepted if and only if, there exists some execution of w that accepts. If x is rejected, then

M is M'. If x is accepted, then by assumption, there exists an execution, x', which is also accepted. If x is rejected but x' is accepted then w is still in the language of M'. $\square$

4.1 Universal Machines

When a TM M computes a string w, a set of executions are generated, each with its own behavior. M's acceptance of w depends on these behaviors. If any execution accepts, then M accepts w but if any execution does not halt, then M does not reject w. Though it is undecidable which, an execution can be sorted into at least one of four categories described in Table 4.1.

Behavior	Property
Accepting	Entered an accepting state
Rejecting	Entered a configuration with no next step
Diverging	Entered a new configuration at each step
Looping	Entered the same configuration a second time

Table 4.1: *End Behaviors of TMs*

4.1.1 Detecting Loops in TM Execution

While the first three categories are disjoint, the fourth contains elements of each category since an execution could loop and then take a different path. A looping configuration could also accept, reject, or diverge or simply continue to loop.

Whether an execution has repeated is recognizable since extra untracked storage could be used to keep every configuration that has been entered. Since each configuration was entered after finite time, the amount of tape used is finite, and the list of configurations can be traversed. In the steps during the loop, other executions will have been created leaving the loop. These same children executions can be made again the second time, however the corresponding child from the first loop will steps ahead and will reach the same result. This is illustrated in Figure 4.1. Thus means that once an execution has looped, no children that it spawns can be the only accepting execution.

Definition 4.3. *A_{NoLoop} is a universal TM that takes a TM M, and a string, w and rejects looping executions but otherwise simulates w on M.*

By Lemma 4.1, A_{NoLoop} accepts $\langle M, w\rangle$ if and only if M accepts w but A_{NoLoop} may reject $\langle M, w\rangle$ even if M does not reject w. This reduces the possible behaviors to accepting, rejecting, and diverging.

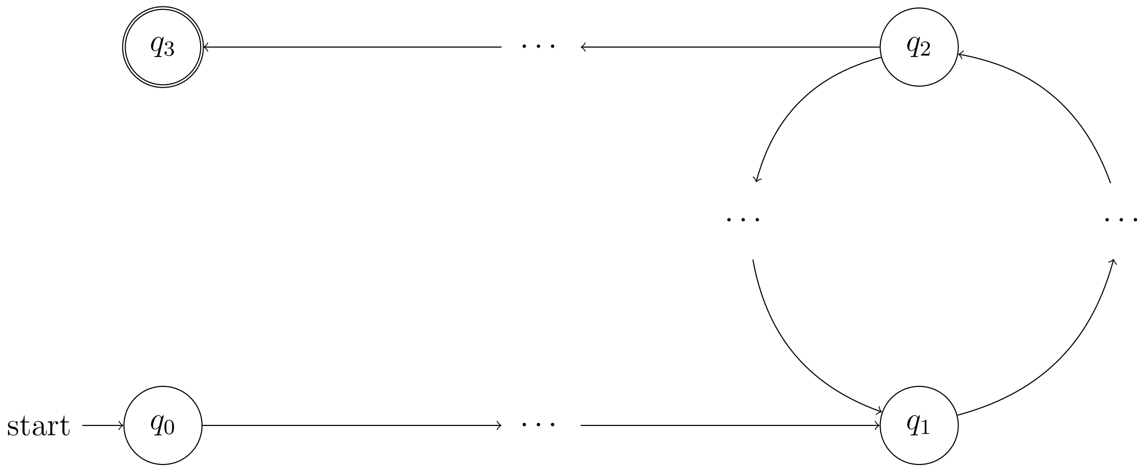

Figure 4.1: *Repeated Configuration and Nondeterminism*
When the execution follows $q_0q_1q_2$, it will both leave the loop and enter q_3 and repeat the loop and enter q_2 again. On each subsequent loop, it will spawn an execution "younger" than an existing execution and loops again.

4.1.2 The Universal Bounded Automaton

By using A_{NoLoop}, all TMs either halt or diverge. Using the contrapositive of the pigeonhole principle[1], since a divergent execution never enters the same configuration twice, it must enter infinitely many configurations and therefore use infinite tape. This means a BA can have no divergent executions.

Theorem 4.1. *Given a language L, the following are equivalent:*

1. *There exists a BA M_B, such that $L(M_B) = L$*

2. *There exists a halting TM M_H, such that $L(M_H) = L$*

[1]Discussed in Section 2.1.3

Equivalently, the set of bounded languages is the set of recursive languages.

Proof.

$1 \to 2$: Since M_B is tape bounded, when computing a string, w, if it does not halt then it must loop. M_H can be constructed using A_{NoLoop} and M_B and will halt when M_B looped. M_H therefore always halts [1; 4].

$2 \to 1$: For any length n, for any word in L $|w| = n$, there exists a nonempty set of execution that accept w on M_H. Each execution s, has a maximum amount of tape used $l(w, s)$. By the well ordered principle of the natural numbers [16], there is a minimum $l(w) = \min_s l(w, s)$. Using only $l(w)$ tape, some accepting execution that normally would accept w but will now leave the bounds and reject. Lemma 4.1 shows that w is still accepted. The value $s(n) = \max_{|w|=n} s_{min}(w)$ is the least amount of tape needed to accept any word in string in L of length n. Therefore there is a tape bound, s, in which L is computed. □

At the beginning of this Chapter, s was introduced as a function that calculates how much a BA will need as was a process that makes $s(|w|)$ squares of tape. These two can be combined into a bounding TM.

Definition 4.4. *A bounding TM is one that creates a tape allocation based on the size of a string.*

With a finite length of tape, there are finitely few distinct configurations [1; 4]. Rather than finding the first loop as A_{NoLoop} does, another approach would be to prove that any loop occurred. As an additional track to the machine, a place-value system could count the number of steps the execution has taken. Once the count exceeds the number of possible configurations, by the pigeonhole principle, a loop must have occurred [1; 4]. This requires $s \in \Omega(log)$ otherwise the configuration counting outruns the tape [2]. Logarithm is also a lower bound for Savitch's Theorem [1; 4; 21], somehow demonstrating that sub-logarithmic space complexity behaves differently.

Definition 4.5. *A_{BA} is another universal TM that given a bounding TM S, a TM intended to compute on a bounded tape M, and a string w, simulates w on M with loop detection on a tape bounded by S.*

4.2 Space Complexity

Definition 4.6. *The space complexity*[2] *of a bounding function is* $NSPACE(s) = \{L \mid \exists\ M \text{ bounded by } s \wedge L = L(M)\}$ [4].

In Section 3.2.3, it was shown that a length of n tape can simulate $k * n$ tape for any $k \in \mathbb{N}$. Obviously $k * n$ tape can simulate n tape by only using the first portion of the tape. Therefore $k * n$ tape is equally expressive as n tape. Since expressive equality is an equivalence relation, for all k_1 and k_2, $k_1 * n$ tape can simulate $k_2 * n$ tape or simply n tape can simulate $c * n$ tape for all $c \in \mathbb{R}$ [1; 8].

Theorem 4.2. *If two positive functions s_1 and s_2 are continuous and $s_1 \in \Theta(s_2)$, then s_1 and s_2 have the same space complexity.*

Proof. Θ prescribes that after some finite interval, the ratio $\frac{s_1(n)}{s_2(n)}$ is bounded [2]. The functions s_1 and s_2, being continuous over the initial interval, are bounded as is their ratio [16]. Since the ratio is bounded over the positive reals, then there exists a scalar that equates them. $\square$

The space hierarchy theorem extends the correspondence between function complexity and space complexity with the fact $f \in \Omega(log) \wedge f \in o(g) \implies NSPACE(f) \subsetneq NSPACE(g)$ [1; 4].

4.2.1 Immerman-Szelepcsényi Theorem

An important find in space complexity theory is that every complexity class at or above log, is closed under complementation, formally $f \in \Omega(log) \implies \forall\ L \in NSPACE(f),\ \overline{L} \in NSPACE(f)$ [4; 8]. Given a bounded language L, there exists a BA M that accepts L. If the complement of L is also a bounded language and can be accepted by a BA M' with the same bound as M. M' can be simulated using M. For a string w, if there exists an execution x of M that accepts w then M accepts w and M' should reject. We can consider

[2]A space complexity class is a set of TMs in [1].

x an execution of M' as well. This situation where a rejecting execution implies that all execution will be rejected will be called negative nondeterminism and is a special case of alternating TMs [1; 4]

4.2.2 The Chomsky Hierarchy Revisited

Most of the types of the Chomsky Hierarchy can be expressed using space complexity. This makes another progression that can be used to describe the languages.

Unrestricted Languages are related to TMs that may or may not only require finite tape. If an unrestricted languages is decidable, then there exists some bounding function for it, other wise no bound exists. This set of languages contains all possible space bounds.

The nondeterministic linear bounded automaton[3] is a BA with the bounding function s(n) = n and accepts the set of CSLs [1; 6]. The question of what type of language is the complement of a CSL remained open for some time [5; 6; 8]. But an immediate consequence of the Immerman-Szelepcsnyi Theorem is that the linear space is closed under complementation [1; 8]

Lemma 4.2. *The complement of a CFL is a CSL.*

Proof. This follows from the closure property just given and the fact that CFLs are a subset of CSL. □

The Immerman-Szelepcsényi theorem which also requires at least *logspace* shows that the complement of a language has the same space complexity. This means that even though CFLs are not closed under complementation, the containing space complexity is. The space complexity is exactly logarithmic since efficient coding of a PDA onto a BA will reduce storing quantities on the stack to storing a value in place-value form in the tape. Strangely, languages that are not related to CFLs are also in logspace. With the pumping lemma, $0^a1^b0^{a*b}$ can be shown to not be context free. But by construction of a log bounded BA[4], the

[3]Hereafter LBA

[4]The tape would need $2 * (\lceil log(a)\rceil + \lceil log(b)\rceil)$ to store a and b and $a + b$ in place value.

languages is in logspace. By the hierarchy theorem, logspace is in linear space, and therefore is a CSL.

When computed on a NFA, regular languages don't need storage beyond the state space. So type 3 $= NSPACE(0)$, but beyond that, with k tape a BA can split state between the state space proper and the tape configuration but still has finite states, therefore the set of type 3 languages $= NSPACE(0) = NSPACE(k)$. This shows the need for the first condition on the space hierarchy theorem as 0 and k are of different orders but have the same space complexity.

Chapter 5

Java Compilation

Language processors including compilers typically involve a pipeline that creates, modifies, annotations, and records intermediate representations [18; 19; 23]. Though not the only way to write a compiler, this methodology can serve to describe the process since compilation has no side effects and has several inherently sequential parts. Each phase can be treated as a form of analysis of the input from the previous phase and will output information for subsequent phases as a string in a potentially different alphabet. The output of the last phase is the result of compilation. These phases will be executed on the various tapes of a multitape LBA as discussed in Section 4.2.2.

It will be demonstrated that these phases iteratively refine the language of legal Java programs from type 3 to type 2 and finally to type 1. This would mean that the compiler need not be Turing complete since it reduces to acceptance of a CSL, a strictly simpler problem than acceptance of a TM. The order of types given is the reverse of the description from Chapter 2. Since languages are sets of strings, one language can be a proper subset of another regardless to their respective types.

The type 3 language used for lexical analysis is not the tightest possible type 3 language that accepts all legal Java programs, but any tighter type 3 bound would eliminate some but not all illegal token orders[1]. The pumping lemma for regular languages[2] can be invoked

[1]For example `public` must be followed by whitespace.

[2]Discussed in

to show that Type 2 refinement is still required since the set of legal Java programs is not a regular language. The actual need for type 1 refinement can be demonstrated by showing that the set of legal Java programs is not context-free with a generalization of the pumping lemma for context-free languages[3], Ogden's lemma, where nontrivial portions of the file including member declaration must be pumpable [1; 4].

The goal is to show that compilable Java programs for a CSL. This will be done by showing that compilation can be computed on an LBA. It should be emphasized that this endeavor is assess feasibility not practicality. This discussion is intentionally abstract to potentially allow it to be applied to other programming languages. The program in Figure 5.1 will serve as a demonstration through the phases.

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

Figure 5.1: *Sample Compiler Input*

5.1 Lexical Analysis

Lexical analysis[4] is the process of converting the file into a sequence of tokens including optional values. This is done by iteratively consuming the maximum match of the remainder of the file with a regular language [18; 19; 24], s, and recording a function of s. This function, T, behaves according to Table 5.1. The compiler will simulate an finite automaton and write the tokens on a second tape.

For each form, as $|s|$ increases, $|T(s)|$ increases no faster. Therefore $|T(s)| \in O|s|$ and since $\sum_{s\ in\ file} |s| = |file|$, $\sum_{s\ in\ file} |T(s)| \in O(|file|)$. The resulting token stream is given in Figure 5.2.

[3]Discussed in

[4]Sometimes referred to as scanning [19]

Semantic Information	Token	Example
None	Suppressed	Whitespace, Comment
Fixed	<Token Type>	Operators, even multi-character
Variable	<Token Type, Token Value>	Literal, Identifier

Table 5.1: *String to Token Transformations* [18; 19; 24]

```
<T_public><T_class><T_id, HelloWorld><T_openCurly><T_public><T_static><T_void>
<T_id, main><T_openParen><T_id, String><T_openSquare><T_closeSquare>
<T_id, args><T_closeParen><T_openCurly><T_id, System><T_dot><T_id, out><T_dot>
<T_id, println><T_openParen><T_stringLit, "Hello, World!"><T_closeParen>
<T_semi><T_closeCurly><T_closeCurly>
```

Figure 5.2: *Sample Token Stream*
The result of parsing the Sample Program in Figure 5.1

5.2 Syntactical Analysis

Syntactical Analysis[5] is the process of converting the token stream into a parse tree by checking conformance to a CFG [1; 4; 18; 19; 24]. Therefore, syntactical analysis recognizes a type-2 language [1; 13]. The terminals of the grammar are the tokens from lexical analysis. As mentioned in Section 2.2.3, since a parse tree corresponds to a semantic meaning, it is important to have only one correct parse tree. Since computer language grammars are used programmatically to parse programs, these grammars can be refactored to always create the same tree for any string, making the grammar deterministic.

5.2.1 Parse Sequence

Several data structures can be used to represent parses [2; 17–19; 23], but to constraint output to a string, the parenthetical structure of trees is needed. It has been shown that for an unambiguous grammar, "any two [derivations] in a [string] ... either ... are disjoint or one is part of the other." [14, p. 571] This has been extended to depth-first-search of non-tree graphs, the parenthesis theorem says that either two nodes will have an ancestor-descendant relationship where the descendant is fully explored during exploration of the ancestor or

[5]Sometimes reffered to as parsing [19]

the nodes are unrelated and the first node is fully explored before exploration of the second begins [2][6].

> "If we represent the ... [start] ... of vertex u with a left parenthesis "$(u$" and represent its finish by a right parenthesis "$u)$", then ... [the resulting string] ... makes a well-formed expression in the sense that the parentheses are properly nested" [2, p. 606]

Because these parentheses are well formed, the symbol preceding the close parenthesis is not needed since it is known which non-terminal is being closed. This inspires a string notation for parse trees. A non-terminal V having the production $\alpha_1, \alpha_2, \alpha_3$ will be represented as $(V\ \alpha_1\ \alpha_2\ \alpha_3)$. Of course if any of the αs are non-terminals, then they will be written out in the same fashion. This "s-expression" or LISP format [25] mirrors postfix notation [18]. The parse sequence of the example program is given in Figure 5.3.

5.2.2 Parser Technology

Backus-Naur form is a standard, machine readable way to express a CFG [1; 18; 19; 26]. The grammar specification for Java extends BNF with brackets to indicate a sequences appears `{zero or more times}` or `[zero times or one time]` [24]. This allows for variable-arity of nodes and shorter parse trees. Grammars with this feature are no more expressive than fixed-arity recursive grammars, only easier to develop and understand at the penalty of now needing more complex data structures to store a node's children [2; 17–19; 23].

In fact, since programming language grammars are designed rather than discovered, they could be constrained to LL grammars where the correct next production can be selected greedily. The production selection is done with a lookup table defined by the grammar and a fixed window of the token stream [18; 19; 23]. The lookup table, being defined by the language not the input, can be built into the control state of the parser rather than the memory used to manipulate the token stream. The nodes generated will be written out to

[6]Algebraically, this means that the relation "happens entirely after or entirely during" if understood to be reflexive, is a total ordering.

```
(CompilationUnit (OrdinaryCompilationUnit (TypeDeclaration
(ClassDeclaration <T_public> <T_class> <T_id, HelloWorld> (ClassBody
(<T_openCurly> ClassBodyDeclaration (ClassMemberDeclaration
(MethodDeclaration <T_public> <T_static> (MethodHeader (Result <T_void>)
(MethodDeclarator <T_id, main> <T_openParen> (FormalParameterList
(LastFormalParameter (FormalParameter (UnannType (UnannReferenceType
(UnannArrayType (AmbiguousType <T_id, String>) (Dims <T_openSquare>
<T_closeSquare>)))) (VariableDeclarationID <T_id, args>))))
<T_closeParen>)) (MethodBody (Block <T_openCurly> (BlockStatements
(BlockStatement (Statement (StatementWithoutTrailingSubstatement
(ExpressionStatement (StatementExpression (MethodInvocation
(ExpressionName (AmbiguousName <T_id, System>) <T_dot> <T_id, out>)
<T_dot> <T_id, println> <T_openParen> (ArgumentList (Expression
(AssignmentExpression (ConditionalExpression (ConditionalOrExpression
(ConditionalAndExpression (InclusiveOrExpression (ExclusiveOrExpression
(AndExpression (EqualityExpression (RelationalExpression (ShiftExpression
(AdditiveExpression (MultiplicativeExpression (UnaryExpression
(UnaryExpressionNotPlusMinus (PostfixExpression (Primary (PrimaryNoNewArray
<T_stringLit, "Hello, World!">))))))))))))))))))) <T_closeParen>))
<T_semi>))))) <T_closeCurly>))))))))))
```

Figure 5.3: *Sample Parsing sequencing*
Using a Lisp like notation, a parse tree for the sample program in Figure 5.1 can be expressed simply as a sequence of symbols.

another tape on the LBA.

5.2.3 Upper Bound on Parse Sequence Length

To show the memory bound of the parse sequence, the grammar must meet certain expectations. Besides being deterministic which was already established, the grammar may not have any ε productions. The issue with ε productions is that they could create whole subtrees full of ε leaves and internal nodes with no tokens as children. Luckily, these can be refactored out [1; 18; 19]. If a grammar meets this condition, then the number of internal nodes is linearly bounded by the number of terminal nodes[7]. With internal nodes having constant size, the size of the parse sequence is linearly bounded by the size of the file.

[7]Proven on page 44

5.3 Semantic Analysis

Chomsky gives the example “colorless green ideas sleep furiously” [10, p. 116] of a phrase that parses correctly but is meaningless. To prevent similar results in Java, a few checks must be made.

5.3.1 Parse Tree Traversal

Semantic analysis will be performed by traversing the tree and potentially adding information to the tree. To make room for additional information, the tape would have to be shifted over the appropriate amount. As opposed to traditional programming environment where a call stack can manage recursively examining nodes of the tree, the best option is to mark nodes currently being explored. By traveling left from a node, the head will find the parent of the current node before any other of its ancestor. An additional tape can be used as a stack to keep track of the depth, every time a internal node is entered, it should be marked and the stack should be incremented. Anything encountered at the wrong level to be considered will be ignored.

5.3.2 Symbol Table

Every named structure in the program must be recorded with its scope and type for references to it may be resolved [18; 19]. Though described as a table, the information and its format is not strictly tabular. Since Java is statically scoped, variables have meaning based on where they are located rather than when they are referenced [24]. Since the parse tree contains all the structural and semantic information of the program, the symbol table already exists in the tree.

In Figures 5.4 amd 5.5, while searching for a declaration of i at D, the head will reach C on the tape. However, since C is at the wrong depth, it is out of scope and ignored.

With the introduction of the keyword `var`, where local variables type can be omitted to be inferred from the assigned value, they can not be immediately typed because they may

```
{//A
    {//B
        int i;//C
    }
    i = 0;//D
}
```

Figure 5.4: *Scope Example Code*

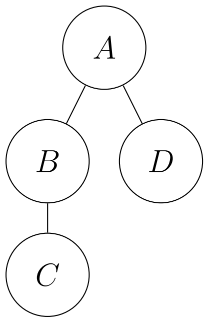

Figure 5.5: *Scope Example Tree*

depend on some value in a broader scope. A local variable will have been assigned to before any valid reference to it is made and by that time it will have a type [24]. This type can overwrite the `var` declaration in the tree at the cost of a constant number of symbols per local variable. The total of these increases is also linearly bounded by the file size.

5.3.3 Type Checking and Reachability

To enhance safety at runtime, Java requires programs expressions have a correct type for how they interact with other elements [24]. This is done by recursively examining each node to determine its type and recording it. Again this can be done by writing into each leaf node its type. With nondeterminism, each leaf is assigned a legal type if possible. A parent node then inspects the types of its children calculates its type as a and rejects the program if there is an error or stores it and proceeds to the parent. According to [24, Section 14.21.], "It is a compile-time error if a statement cannot be executed because it is unreachable." Checking this is very similar to type checking except that only information needed is whether a statement can executes normally.

5.3.4 Bytecode Generation

The bytecode generated from Java compilation is in the form of classfiles which consist of constants and then class members [27]. Constants are created as needed to represent language constructs like identifiers, literals, and method calls. As the tree is converted into a classfile, whenever a constant is needed, the partial constant pool being built can serve as a lookup table returning the index of existing constants or creating new constants.

5.4 Conclusion

This shows that a context sensitive grammar must exist that can decide compilability of a Java program. Using the theory developed in Chapter 4, it has been shown that Java programs can be compiled using a linear bounded automaton. This representation is certainly less wieldy than existing compiler theory but has the advantage of clearly defining its memory needs. Also Java compilation can be simulated on A_{BA} meaning it is not Turing complete.

Bibliography

[1] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation (Addison-Wesley series in computer science)*. Addison-Wesley Publishing Company, 1979.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (Eastern Economy Edition)*. PHI Learning Pvt. Ltd. (Originally MIT Press), 2010.

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Ronald Rivest. *Introduction To Algorithms*. Mcgraw-Hill College, 1990.

[4] Michael Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 2012.

[5] Peter S. Landweber. Three theorems on phrase structure grammars of type 1. *Information and Control*, jun 1963.

[6] S.-Y. Kuroda. Classes of languages and linear-bounded automata. *Information and Control*, jun 1964.

[7] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Pearson, 2006.

[8] Neil Immerman. Nondeterministic space is closed under complementation. *SIAM Journal on Computing*, 17(5):935–938, oct 1988.

[9] Gödel prize - 1995. URL `http://eatcs.org/index.php/component/content/article/513`.

[10] N. Chomsky. Three models for the description of language. *IEEE Transactions on Information Theory*, 2(3):113–124, sep 1956.

[11] C. E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27(3):379–423, jul 1948.

[12] Douglas Lind and Brian Marcus. *An Introduction to Symbolic Dynamics and Coding.* Cambridge University Press, 1996.

[13] Noam Chomsky. On certain formal properties of grammars. *Information and Control*, 2(2):137–167, jun 1959.

[14] Rohit J. Parikh. On context-free languages. *Journal of the ACM*, 13(4):570–581, oct 1966.

[15] Martin Davis. *Computability and Unsolvability.* McGraw-Hill, New York, 1958.

[16] Gerald G. Bilodeau, Paul R Thie, and G. E. Keough. *An Introduction to Analysis (International Series in Mathematics).* Jones & Bartlett Learning, 2009.

[17] Mark Newman. *Networks: An Introduction.* Oxford University Press, 2010.

[18] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition).* Addison Wesley, 2006.

[19] Keith Cooper and Linda Torczon. *Engineering a Compiler.* Morgan Kaufmann, 2003.

[20] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 1937.

[21] Walter J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, apr 1970.

[22] Joshua Bloch. *Effective Java (3rd Edition).* Addison-Wesley Professional, 2018.

[23] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software (Addison-Wesley Professional Computing Series).* Addison-Wesley Professional, 1994.

[24] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, and Daniel Smith. The java language specification, 2017.

[25] John McCarthy. Recursive functions symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, apr 1960.

[26] Seymour Ginsburg and H. Gordon Rice. Two families of languages related to ALGOL. *Journal of the ACM*, 9(3):350–371, jul 1962.

[27] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. The java virtual machine specification, 2017.

[28] Cliff L Stein, Robert Drysdale, and Kenneth Bogart. *Discrete Mathematics for Computer Scientists*. Addison-Wesley, 2010.

[29] Atle Selberg. An elementary proof of dirichlet's theorem about primes in an arithmetic progression. *The Annals of Mathematics*, 50(2):297, apr 1949.

Appendix A

Selected Proofs

Following are several proofs written out formally. They are separated from the main body of work because many reader will find the reasoning obvious but included vestigially to demonstrate proof techniques. Several of the justifications use basic facts about regular expressions that [2; 4; 18; 19] cite from [1].

A.1 Preliminary Lemmas

Lemma A.1 (Closure Lemma). *Given a set, X, and $F \in 2^X$, if F is closed under complementation, then it is closed under union if and only if it is closed under intersection*[1]*.*

Proof.

Forward: By DeMorgan's, the intersection of two sets can be constructed from only complementation and union [1; 2; 4; 16; 28], $A \cap B = \overline{\overline{A} \cup \overline{B}}$. By the assumed closures, the result of such a construction is in F.

Backward: Also by DeMorgan's, the union of two sets can be constructed from only complementation and intersection, $A \cup B = \overline{\overline{A} \cap \overline{B}}$. By the assumed closures, the result of such a construction is in F. □

Definition A.1. *A language is singleton if and only if it contains exactly one word*

[1]Special cases of this are used in [1; 26] and left as an exercise in [4]

Lemma A.2. *A singleton language (Definition A.1) is regular*

Proof. By induction on $|w|$

Case $|w| = 0$**.**

(1) $w = \varepsilon$	by ε being the unique string of length 0
(2) $\{\varepsilon\}$ *is regular*	by $\{\varepsilon\} = L(\varepsilon_R)$
(3) $\{w\}$ *is regular*	by substitution, (1–2)

Case $|w| > 0$**.**

(1) *Let* $w = w'a$	by choice
(2) $|w'| = n - 1$	by (1)
(3) $\{w'\}$ *is regular*	by inductive hypothesis, (2)
(4) $\{a\}$ *is regular*	by $\{a\} = L(a)$
(5) $\{w'\} \circ \{a\}$ *is regular*	by regular languages being closed under concatenation, (3–4)
(6) $\{w'\} \circ \{a\} = \{w'a\}$	by definition of concatenation
(7) $\{w\}$ *is regular*	by (1, 5–6)

□

Lemma A.3. *The language* $L = \{w \in \Sigma^* || w| \leq n\}$ *has cardinality* $|L| = \dfrac{|\Sigma|^{n+1} - 1}{|\Sigma| - 1}$

Proof. By Induction on n

Case $n = 0$.

(1) $L = \{\varepsilon\}$	by ε being the unique string of length 0
(2) $\lvert\{\varepsilon\}\rvert = 1$	by counting
(3) $1 = \dfrac{\lvert\Sigma\rvert^{0+1} - 1}{\lvert\Sigma\rvert - 1}$	by evaluation
(4) $\lvert L\rvert = 1$	by (1–3)

Case $n > 0$.

(1) $L = \{w \in \Sigma^* \mid \lvert w\rvert < n \vee \lvert w\rvert = n\}$	by definition of $\leq$
(2) $L = \{w \in \Sigma^* \mid \lvert w\rvert < n\} \cup \{w \in \Sigma^* \mid \lvert w\rvert = n\}$	by definition of $\cup$
(3) $\{w \in \Sigma^* \mid \lvert w\rvert < n\} = \{w \in \Sigma^* \mid \lvert w\rvert \leq n-1\}$	by discreteness of $\mathbb{N}$
(4) $\lvert\{w \in \Sigma^* \mid \lvert w\rvert \leq n-1\}\rvert = \dfrac{\lvert\Sigma\rvert^n - 1}{\lvert\Sigma\rvert - 1}$	by inductive hypothesis, (3)
(5) $\lvert\{w \in \Sigma^* \mid \lvert w\rvert < n\}\rvert = \dfrac{\lvert\Sigma\rvert^n - 1}{\lvert\Sigma\rvert - 1}$	by (3–4)
(6) $\{w \in \Sigma^* \mid \lvert w\rvert = n\} = \lvert\Sigma\rvert^n$	by $\lvert A^n\rvert = \lvert A\rvert^n$, [2]
(7) $\lvert L\rvert = \dfrac{\lvert\Sigma\rvert^n - 1}{\lvert\Sigma\rvert - 1} + \lvert\Sigma\rvert^n$	by Inclusion-Exclusion Principle of disjoint sets [2], (2, 5–6)
(8) $\lvert L\rvert = \dfrac{\lvert\Sigma\rvert^{n+1} - 1}{\lvert\Sigma\rvert - 1}$	by (7)

□

Lemma A.4. *A non-empty language has a longest word if and only if it is finite (Definition 2.1)*

Proof.

Forward:

(1)	Let w be the longest word in L	by assumption
(2)	$L \subset \{w' \in \Sigma^* \| \|w'\| \leq \|w\|\}$	by definition of maximum [16], (1)
(3)	$\|L\| \leq \frac{\|\Sigma\|^{\|w'\|+1} - 1}{\|\Sigma\| - 1}$	by $A \subset B \implies \|A\| \leq \|B\|$, Lemma A.3, (2)
(4)	$\|L\|$ *is finite*	by $\|L\| \in \mathbb{N}$, (3)
(5)	L *is finite*	by Definition 2.1, (4)

Backward: By Contradiction

(1)	L is finite and non empty	by assumption
(2)	$\nexists\, w \in L,\ \forall\, w' \in L\ \|w\| \geq \|w'\|$	by assumption
(3)	$\forall\, w \in L,\ \exists\, w' \in L\ \|w\| < \|w'\|$	by DeMorgan's with quantification [15; 28], (2)
(4)	$\exists\, w \in L$	by (1)
(5)	There exists an infinite sequence of words in L	by induction, (3–4)
(6)	$\perp$	by (1, 5)

□

A.2 Theorems on Finite and Cofinite Languages

Theorem A.1. *A finite language (Definition 2.1) is regular*

Proof. By induction on $|L|$

Case $|L| = 0$.

(1)	$L = \emptyset$	by $\emptyset$ being the unique language of cardinality 0
(2)	$\{\emptyset\}$ *is regular*	by $\{\emptyset\} = L(\emptyset_R)$
(3)	L *is regular*	by substitution, (1–2)

Case $|L| > 0$**.**

(1)	Let $w \in L$	by choice
(2)	Let $L' = (L \setminus \{w\}) \cup \{w\}$	by choice
(3)	$\lvert L \setminus \{w\}\rvert = \lvert L\rvert - 1$	by (1)
(4)	$L \setminus \{w\}$ *is regular*	by inductive hypothesis, (3)
(5)	$\{w\}$ *is singleton*	by Definition A.1
(6)	$\{w\}$ *is regular*	by Lemma A.2, (5)
(7)	$(L \setminus \{w\}) \cup \{w\}$ *is regular*	by regular languages being closed under union, (4, 6)
(8)	$L' = (L \cap \overline{\{w\}}) \cup \{w\}$	by $A \setminus B = A \cap \overline{B}$ [2; 16], (2)
(9)	$L' = (L \cup \{w\}) \cap (\overline{\{w\}} \cup \{w\})$	by $\cup$ being distributive over $\cap$ [2; 16; 28], (8)
(10)	$\{w\} \subset L$	by (1)
(11)	$L' = L \cap (\overline{\{w\}} \cup \{w\})$	by $A \subset B \implies A \cup B = B$, (9–10)
(12)	$L' = L \cap \mathbb{U}$	by $\overline{\{w\}} \cup \{w\} = \mathbb{U}$ [2], (11)
(13)	$L' = L$	by $\mathbb{U}$ being the identity for $\cap$ [2], (12)
(14)	L *is regular*	by substitution, (2, 7, 13)

□

Corollary A.1. *A cofinite language (Definition 2.2) is regular*

Proof. Follows immediately from Theorem A.1 and the already discussed closure of regular languages under complementation. □

Theorem A.2. *The set of finite languages is closed under intersection with any decidable language.*

Proof. $L_1 \cap L_2 \subset L_1$, and therefore $|L_1 \cap L_2| \leq |L_1|$. $|L_1 \cap L_2|$ is thus bounded by a natural number and therefore finite. □

Corollary A.2. *The set of finite languages is closed under intersection.*

Proof. Follows immediately from Theorem A.2 □

Theorem A.3. *The set of cofinite languages is closed under union with any decidable language.*

Proof.

(1)	Let A be a cofinite language	by choice
(2)	Let B be a language	by choice
(3)	Let $C = A \cup B$	by choice
(4)	$\overline{C} = \overline{A} \cap \overline{B}$	by Demorgan's, (3)
(5)	$\overline{A}$ is finite	by Definition 2.2, (1)
(6)	$\overline{C}$ is finite	by Theorem A.2, (5)
(7)	C is cofinite	by Definition 2.2, (3)

□

Corollary A.3. *The set of cofinite languages is closed under union*

Proof. Follows immediately from Theorem A.3. □

Theorem A.4. *The set of finite languages is closed under union.*

Proof. $|L_1 \cup L_2| \leq |L_1| + |L_2|$[2] [2]. Since $\mathbb{N}$ is closed under addition, $|L_1 \cup L_2|$ is bounded by a natural number and therefore finite. □

Corollary A.4. *The set of cofinite languages is closed under intersection.*

Proof. Follows immediately from Definition 2.2, Theorem A.4, and Demorgan's □

Theorem A.5. *The union of finite and cofinite languages is closed under union.*

Proof. Either both languages are finite and Theorem A.4 shows the result to be finite or at least one of the languages is cofinite and Theorem A.3 shows the result is cofinite. □

Theorem A.6. *The union of finite and cofinite languages is closed under complementation.*

Proof. Follows immediately from Definition 2.2. □

Theorem A.7. *The union of finite and cofinite languages is closed under intersection.*

Proof. Follows from the Closure Lemma and Theorems A.5 and A.6. □

A.3 Theorems on Parse Trees

Theorem A.8 (Upper Bound on Tree size)**.** *In a deterministic CFG G, that has no ε productions, if a parse tree T, has terminal nodes $n(T)$, and non-terminal nodes $m(T)$, then $|m(T)| \in O(|n(T)|)$.*

Proof. A deterministic CFG can not have unit recursion and therefore has a longest chain of unit productions with length k. By assumption, k is finite and fixed for G. By strong induction on $|n(T)|$, It can be proven that $\forall\ T,\ |m(T)| \leq k * (|n(T)| + 1)$ which would imply $|m(T)| \in O(|n(T)|)$.

[2]As a consequence of the Inclusion-Exclusion principle

Case $n(T) = 1$**.**

In Figure A.1, T_a derives a terminal in $0 \leq d \leq k$ derivations producing $d = m(T)$ internal nodes. Since $d \leq k$, $|m(T)| \leq k \times (|n(T)|)$.

Case $n(T) > 1$**.**

In Figure A.1, T_b derives the internal node I in $0 \leq d \leq k$ derivations producing d internal nodes.

(1)	$n(T_b)$ is partitioned into $\{n(T_i) \mid 0 \leq i < l\}$	by assumption
(2)	$\forall\, 0 \leq i < l, |n(T_i)| < |n(T)|$	by (1)
(3)	$|n(T_b)| = \sum_{0 \leq i < l} |n(T_i)|$	by (1)
(4)	$\forall\, 0 \leq i < l, |m(T_i)| \leq k \times (|n(T_i)|)$	by inductive hypothesis, (2)
(5)	$\sum_{0 \leq i < l} |m(T_i)| \leq k \times \sum_{0 \leq i < l} |n(T_i)|$	by (4)
(6)	$|m(T_b)| = \sum_{0 \leq i < l} |m(T_i)| + d$	by assumption
(7)	$|m(T_b)| \leq k \times |n(T_b)| + d$	by (3, 5–6)
(8)	$|m(T_b)| \leq k \times (|n(T_b)| + 1)$	by (7)

□

A.4 Example LBA

As an example of the use of the LBA, the language $C = \{1^n \mid n \textit{ is composite}\}$ will be shown to be context-sensitive but not context-free.

Lemma A.5. *C is not context-free.*

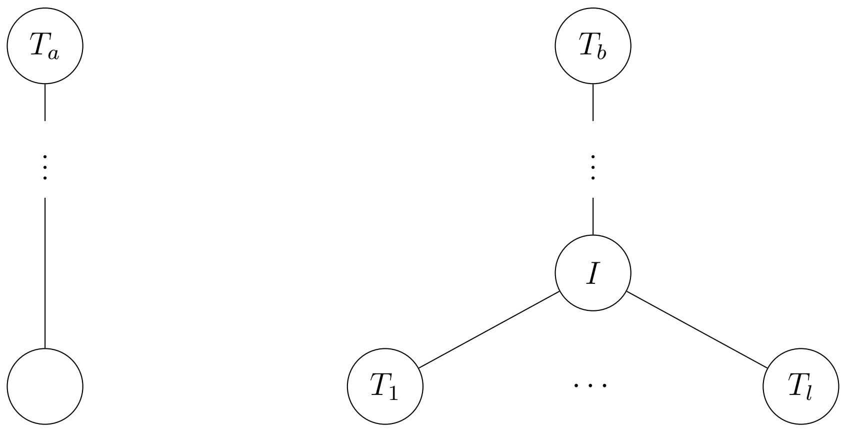

Figure A.1: *Parse Trees*

This classic problem is left as an exercise in [1] and can be solved in multiple ways. Two proofs are given that leverage number theory and language theory.

Proof. Since a CFL over a singleton alphabet is regular[3], then if C is not regular, then it is not context-free. If C were context-free, then $P = \overline{C} \setminus \{\varepsilon, 1\} = \{1^n \mid n \textit{ is prime}\}$ would be regular as well by regularity preserving steps [1; 4]. P's non context-freedom is left as an exercise in [1] and can be proven with the pumping lemma since even if $a + b + c$ is prime, $a + b(a + c) + c = (a + c) * (b + 1)$ is not. $\square$

Proof. By contradiction

(1)	C is context-free	by assumption
(2)	C is pumpable	by pumping lemma for CFL [1; 4], (1)
(3)	$\exists\ p \mid \forall\ w \in C, \lvert w\rvert \geq p \implies$ $\exists\ u, v, x, y, z \mid uvxyz = w \wedge \lvert vy\rvert > 0$ $\wedge\ \lvert vxy\rvert \leq p \wedge \forall\ i, uv^ixy^iz \in C$	by definition of pumpability for CFL [1; 4], (2)

[3]This is a trivial application to a theorem in [14], a direct corollary in [26], left as an exercise in [1] provable by the pumping lemma.

(4)	Let $q > p$ be prime	by Euclid's infinitude of primes [4; 15], choice
(5)	Let $w = 1^{q^2}$	by choice. As seen in Step (13), $\forall\ 1 < i \leq p,\ |w|$ must be coprime to i. The smallest satisfying number is q^2.
(6)	$w \in C$	by definition of C, (5)
(7)	$|w| \geq p \implies \exists\ u, v, x, y, z \mid$ $uvxyz = w \wedge |vy| > 0 \wedge\ |vxy| \leq p \wedge$ $\forall\ i, uv^ixy^iz \in C$	by (3, 6)
(8)	$|w| = q^2$	by (5)
(9)	$\exists\ u, v, x, y, z \mid$ $uvxyz = w \wedge |vy| > 0 \wedge |vxy| \leq p$ $\wedge\ \forall\ i \mid uv^ixy^iz \in C$	by (4, 7–8)
(10)	$u = 1^a, v = 1^b, x = 1^c, y = 1^d, z = 1^e$	by (5, 9)
(11)	Let $n = b + d$	by choice
(12)	$p \geq n > 0$	by (9–11)
(13)	n and q^2 are coprime	by (4, 12)
(14)	$\exists\ j \mid n \times j + q^2$ is prime	by Dirichlet's Theorem About Primes in an Arithmetic Progression [29], (13)
(15)	$uv^{j+1}xy^{j+1}z \in C$	by (9)
(16)	$|uv^{j+1}xy^{j+1}z|$ is composite	by definition of C, (15)
(17)	$|uv^{j+1}xy^{j+1}z| = n \times j + q^2$	by (10–11)
(18)	$\perp$	by (14, 16–17)

□

Lemma A.6. *C is context-sensitive.*

Proof. By Construction of an LBA

One possible certificate format[4] for compositeness is a divisor D greater than 1 [1; 2; 4] and verification is simply divisibility[5]. Figure A.2 will be shown to accept C.

First, Table A.1 shows a partitioning of the possible configurations of the machine. Each partition is a family of configuration all having the same state, head symbol, and tape pattern. The transitions between these families is given in Figure A.3. Once $F_4 \Rightarrow F_6$ is taken, the divisor is fixed. The machine then proceeds to convert a symbol from the divisor, $2 \to 3$, and then mark a symbol from the dividend, $1 \to \textvisiblespace$. When all of the symbols from the divisor are 3, detected by $F_{12} \Rightarrow F_{13}$, they are all reset back to 2 and the cycle repeats. Every time the divisor is fully marked and then reset, the dividend has been decremented by D.

If there are no more 1s, then there should be exactly a single 3. This is tested with whether the symbol left of the 3 is a 2. If so, then the machine accepts. Otherwise, it rejects. An effect of this is that $D = 1$ is rejected after the division rather than failing quickly. Here program simplicity is favored over efficiency.

Termination can be guaranteed by showing that all loops terminate. All of the self loops point in a single direction and will stop at the end. Every multinode loop contains 9 and every exit decrements the number of F_1, which is monotonic. Therefore, eventually F_6, F_7, or F_8 must transition to F_{14} rather than F_9 and the machine exits the complex division logic. All loops happen a finite number of times and so the machine does halt. □

[4]Additional information that if correct, simplifies an affirmative determination [2; 4]
[5]More precisely, $D \mid N - D$

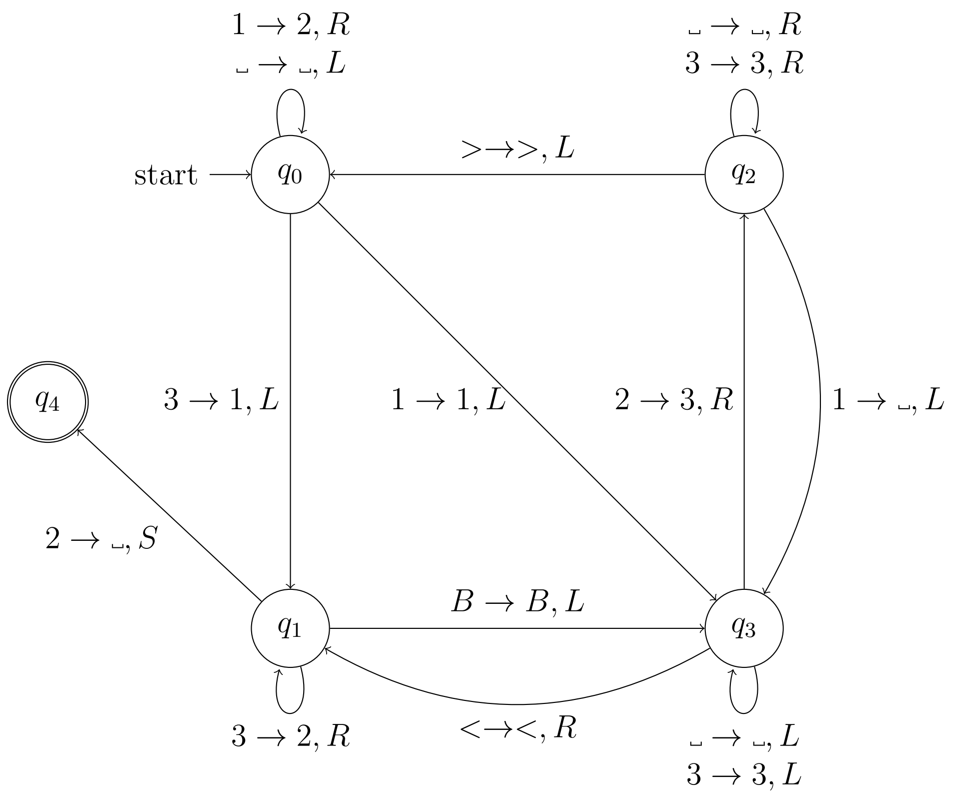

Figure A.2: *LBA for C*

Config. Fam.	Descrip.	Tape	State	Next Config. Fam.
S	Start	$\underline{<}1* >$		F_1, F_2
F_1	$N = 0$	$< \underline{>}$	q_1	Reject
F_2	$N \neq 0$	$< \underline{1}1^* >$	q_1	F_3, F_4
F_3	$D = 0$	$< \underline{1}1^* >$	q_2	Reject
F_4	Building D	$< 2^*\underline{1}1^* >$	q_1	F_4, F_5, F_6
F_5	$N = D$	$< 2^*\underline{>}$	q_1	Reject
F_6	Last $2 \rightarrow 3$	$< 2^*\underline{2}3^*\sqcup^*1^* >$	q_2	F_7, F_8, F_9, F_{14}
F_7	Skipping 3s	$< 2^*3^*\underline{3}3^*\sqcup^*1^* >$	q_3	F_7, F_8, F_9, F_{14}
F_8	Skipping $\sqcup$s	$< 2^*3^*\sqcup^*\underline{\sqcup}\sqcup^*1^* >$	q_3	F_8, F_9, F_{14}
F_9	First $1 \rightarrow \sqcup$	$< 2^*3^*\sqcup^*\underline{1}1^* >$	q_3	F_{10}, F_{11}
F_{10}	Skipping $\sqcup$s	$< 2^*3^*\sqcup^*\underline{\sqcup}\sqcup^*1^* >$	q_2	$F_6, F_{10}, F_{11}, F_{12}$
F_{11}	Skipping 3s	$< 2^*3^*\underline{3}3^*\sqcup^*1^* >$	q_2	F_6, F_{11}, F_{12}
F_{12}	No more 2s	$\underline{<}3^*\sqcup^*1^* >$	q_2	F_{13}
F_{13}	First $3 \rightarrow 2$	$< 2^*3^*\underline{3}3^*\sqcup^*1^* >$	q_4	F_{13}, F_6
F_{14}	No more 1s	$< 2^*3^*\sqcup^*\underline{>}$	q_3	F_{15}
F_{15}	Skipping $\sqcup$s	$< 2^*3^*\sqcup^*\underline{\sqcup}\sqcup^* >$	q_1	F_{15}, F_{16}
F_{16}	Last $3 \rightarrow 1$	$< 2^*3^*\underline{3}\sqcup^* >$	q_1	F_{17}, F_{19}, F_{20}
F_{17}	more 3s	$< 2^*3^*\underline{3}1\sqcup^* >$	q_4	F_{18}
F_{18}	$D \nmid N$	$< 2^*3^*2\underline{1}\sqcup^* >$	q_2	Reject
F_{19}	$N = 1$	$\underline{<}1\sqcup^* >$	q_4	Reject
F_{20}	No more 3s	$< 2^*\underline{2}1\sqcup^* >$	q_4	F_{21}
F_{21}	$D \mid N$	$< 2^*\underline{2}\sqcup 1\sqcup^* >$	q_5	Accept

Table A.1: *Configuration Families*
Underlined symbols represent head positions.

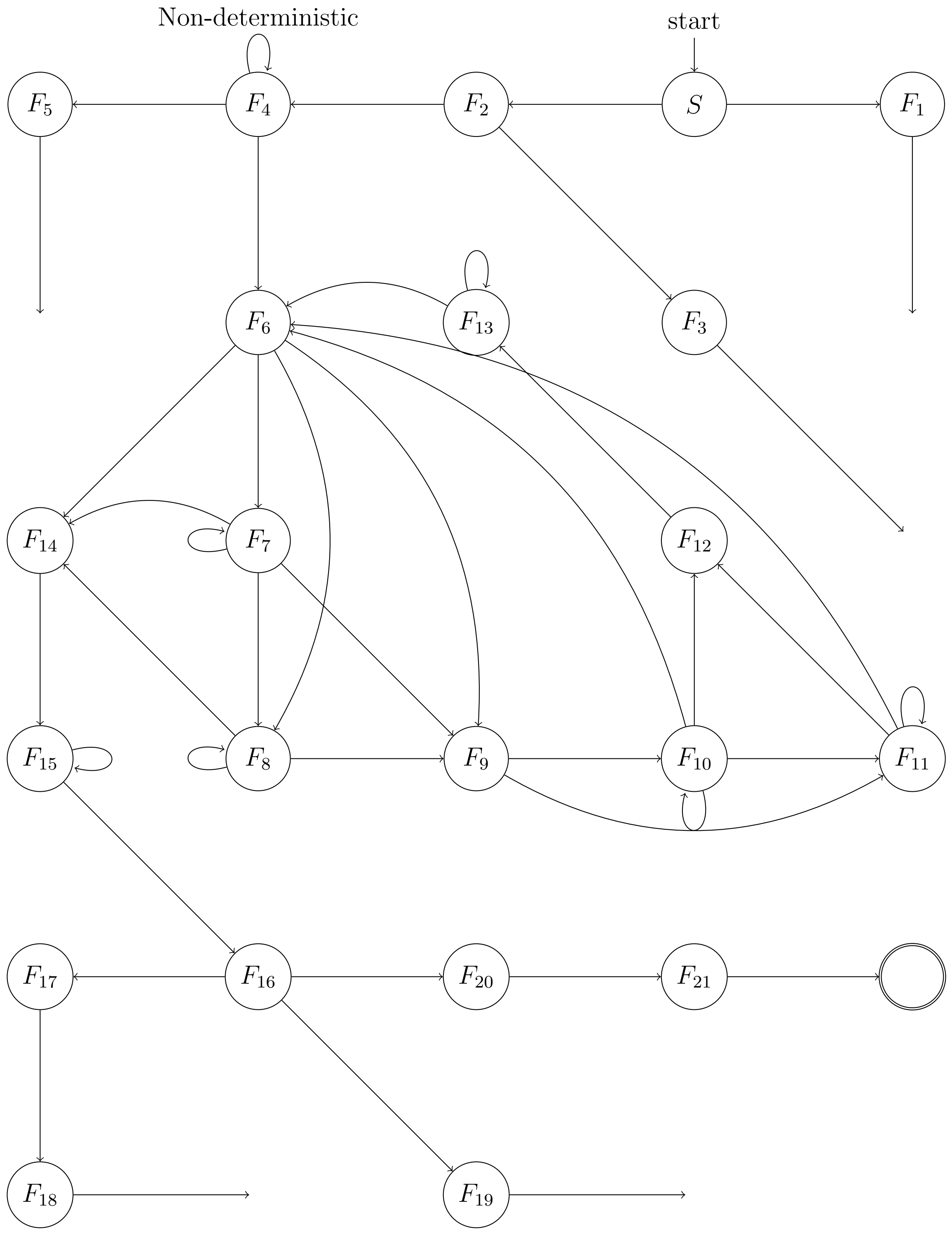

Figure A.3: *Transition Graph among Configurations Families*