AN ABSTRACT OF A THESIS

PROPERTIES OF INFINITE LANGUAGES AND GRAMMARS

Matthew Scott Estes

Master of Science in Mathematics

Traditionally, languages are described as infinite sets, in this thesis, methods for specifying infinite grammars are explored. The effects of an infinite grammar on the language specified are examined along with several proofs. The relationship between grammars and semantics to specify the behaviour of formal systems is also explored.

PROPERTIES OF INFINITE LANGUAGES AND GRAMMARS

A Thesis

Presented to

the Faculty of the Graduate School

Tennessee Technological University

by

Matthew Scott Estes

In Partial Fulfillment

of the Requirements for the Degree

MASTER OF SCIENCE

Mathematics

May 2005

CERTIFICATE OF APPROVAL OF THESIS

PROPERTIES OF INFINITE LANGUAGES AND GRAMMARS

by

Matthew Scott Estes

Graduate Advisory Committee:

Alex Shibakov, Chairperson	date
Jeff Norden	date
Allan Mills	date
Mike Rogers	date

Approved for the Faculty:

Francis Otuonye
Associate Vice-President for
Research and Graduate Studies

Date

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a Master of Science degree at Tennessee Technological University, I agree that the University Library shall make it available to borrowers under rules of the Library. Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of the source is made.

Permission for extensive quotation from or reproduction of this thesis may be granted by my major professor when the proposed use of the material is for scholarly purposes. Any copying or use of the material in this thesis for financial gain shall not be allowed without my written permission.

Signature _____

Date _____

DEDICATION

This thesis is dedicated to my girlfriend Maia and my parents, whose patience and support have enabled me to write this thesis.

ACKNOWLEDGMENTS

I would like to thank Dr. Alexander Shibakov, the Mathematics Department of Tennessee Technological University, Dr. Dick Grune of Vrije Universiteit, Amsterdam, Netherlands.

TABLE OF CONTENTS

Chapter Page

LIST OF TABLES

CHAPTER 1
INTRODUCTION

This thesis is about the study of programming languages and their properties. In order to do that mathematically, the behavior of the programming language must be completely specified. A programming language is a special case of a formal language. The study of the "meaning" of a language (formal or not) is called semantics. The structure of the language is called the syntax. The motivation for this work is to understand how much can be done mathematically to describe a language, especially the features of the language that can be manipulated by a computer from the description alone.

1.1 What Is Language?

In order to develop a formal notion of a language, only two facts are required. A language must consist of sequences of symbols, such as letters, sounds, words, punctuation marks, or numbers. This collection of symbols is called an alphabet and it must be finite. It must also have a structure, called sentences. Not every sequence of symbols is a sentence in the language.

Linguists traditionally put more constraints on what qualifies as a language. These constraints traditionally eliminate animal languages and languages which cannot represent things that are not directly present, or express new ideas. These are worthwhile things to study, but due to their subjective nature and for other practical reasons, are overly restrictive to the goal of creating a formal, mathematical model of languages.

1.2 Syntax And Semantics

Syntax consists of the rules which control how sentences are formed from words. Semantics is the study of meaning in languages. Attempts are often made to distinguish the two, but the discussion rapidly becomes philosophical. As will be shown in this thesis, a more important feature of a complete language formalism (either semantic or syntactic or both) is its total expressive power and whether it is decidable or not.

1.3 Formal Definition Of A Language

Definition 1. *Let Σ be a finite set of symbols, called the alphabet. The language, $\mathcal{L}$, is a collection of finite sequences over Σ.*

While formal languages can often seem nonsensical in nature, this definition encompasses some intuition from natural languages like English. If Σ includes the English alphabet, punctuation marks, numbers, and a symbol for spacing, it is clear that a "sentence" in English is a sequence whose elements are drawn from Σ. The fact that the sentence must be finite in length is a reasonable requirement as well, since an infinitely long sentence has never occurred in any language observed to date.

A more prototypical formal language would be a description of decimal numbers. In this case, one way to describe the langauge would be to let

$$\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ., +, -\}.$$

A decimal number consists a $+$ or $-$ followed by the digits 0-9, and optionally a decimal point, followed by more digits 0-9.

Sometimes sequences of symbols will be referred to as "strings", and ϵ is used to denote a string of length 0.

1.4 The Bit-string Description

How can a language $\mathcal{L}$ be described? One way is to order the elements of Σ (it's a finite set, so this is clearly easy to do; a phone book is an excellent example of ordering an alphabet). Now, a simple procedure will produce all the strings from the alphabet. Starting with a length of 0, list all strings of that length. This process continues for each integer.

Consider the following example. Let $\Sigma = \{a, b\}$.

Length 0

ϵ

Length 1

a b

Length 2

aa ab ba bb

Length 3

aaa aab aba abb baa bab bba bbb

...

This is a actually a very special example. It is all the finite strings on the alphabet Σ, which is called Σ^* (pronounced "sigma star"). Note that all languages are subsets of Σ^*. This leads to a first attempt at describing languages.

Since the strings of Σ^* are ordered, it is easy to check if a given string is in the language $\mathcal{L}$. If it is, write down a 1; otherwise, write down a 0. Starting with the first string on Σ, this will produce an infinitely long sequence of 0's and 1's. This "bit-string" (after a "bit" in a computer) is a description of $\mathcal{L}$. It is important to distinguish the alphabet (Σ) from the strings on the alphabet (Σ^*). The alphabet is a finite set, the strings on the alphabet form a possibly infinite set.

Besides having an infinitely long representation, bit-strings have no desirable properties for a formal description of $\mathcal{L}$. They are hard to work with; they do not take into account any repetitive structures within a language, and other than for simple examples, it is not clear how to write or generate descriptions for languages of interest.

This model is not totally useless, since it will allow one proof about languages.

Theorem 1. *There are an uncountable number of languages on an alphabet* Σ.

Proof. The argument uses diagonalization. Produce a list of all the bit-strings representing all the languages. To build a language not in this list, the k'th character of the bit-string will be opposite of the k'th bit of the k'th bit-string. The bit-string represents a language (since we can identify a string with each bit), but it's not in our list (since the k'th bit of it forces it to be different from the k'th bit-string), which contradicts our assumption that our list was complete. □

1.5 String Rewriting

A much better model of formal languages is string rewriting. In the string rewriting model, a language is described by a collection of rewrite rules. Rewrite rules use two types of symbols. The first type of symbols are called **terminals**, and

they are from an alphabet (which is usually called Σ). The other set of symbols are called **nonterminals**, these symbols must comprise a finite set, and must not come from Σ. The start symbol is a symbol arbitrarily chosen from the nonterminals by the grammar writer, and it's traditionally marked with a subscripted S.

Rewrite rules can be used to produce sentences from the language by a simple process of substitution. Write down the start symbol; this string will eventually become the sentence. Start by matching a piece of the current string with the left hand side (**lhs**) of a rewrite rule, and replace it with the right hand side (**rhs**) of that rule. This new string is used for the next stage of the process. When all nonterminals have been eliminated from the string, a string (over the alphabet of terminals, Σ) which is in the language described by the rewrite rules will be produced.

An example would help. Consider the following rewrite rules, where nonterminals are denoted by uppercase letters, and terminals by lowercase letters. The start symbol is subscripted with an S.

$$S_S \rightarrow A\ B\ C$$

$$A \rightarrow a\ A$$

$$A \rightarrow a$$

$$B \rightarrow b\ B$$

$$B \rightarrow b$$

$$C \rightarrow C\ c\ C$$

$$C \rightarrow c$$

$$C \rightarrow \epsilon$$

Note that there may be multiple rules with the same left hand side.

Now, start the production of a string with S_S.

$$S_S$$

$$A\ B\ C$$

$$a\ A\ B\ C$$

$$a\ A\ b\ B\ C$$

$$a\ a\ A\ b\ B\ C$$

$$a\ a\ a\ b\ B\ C$$

$$a\ a\ a\ b\ B\ C\ c\ C$$

$$a\ a\ a\ b\ b\ C\ c\ C$$

$$a\ a\ a\ b\ b\ C\ c\ C\ c\ C$$

$$a\ a\ a\ b\ b\ c\ c\ C\ c\ C$$

$$a\ a\ a\ b\ b\ c\ c\ c\ c\ C$$

$$a\ a\ a\ b\ b\ c\ c\ c\ c$$

To formally restate this, the following definitions are given for your reading pleasure.

Definition 2. *Let Σ be a finite set of symbols called* **terminals**, *and let $\mathcal{F}$ be a finite set of symbols called* **nonterminals** *such that $\Sigma \bigcap \mathcal{F} = \emptyset$. Let $S \in \mathcal{F}$ be a special nonterminal denoted as the* **start symbol**.

Then, let u^* be the set of all finite strings composed of symbols from $\Sigma \bigcup \mathcal{F}$. Then Γ is a rewrite rule if $\Gamma \in u^* \times u^*$. Then let $\overline{\Phi}$ be a finite subset of $u^* \times u^*$.

The language, $\mathcal{L}$ produced from $\overline{\Phi}$ is the set of all strings, α, consisting only of terminals such that a finite sequence of applications of rules from $\overline{\Phi}$ can produce α from S.

A string, $\alpha_i \in u^*$ is called a **sentential form**, *and to apply a rule $\Gamma = b \rightarrow c$ to α_i, means that $\alpha_i = dbe$, such that $d, e \in u^*$ and $\alpha_{i+1} = dce$.*

1.6 Parsers, Recognizers, And Other Terminology

A collection of rules which can be algorithmically applied to generate a language is called a grammar for that language. For instance, a set of rewrite rules which produces a language $\mathcal{L}$ is called a grammar for $\mathcal{L}$. There are usually many grammars which describe the same language.

A Recognizer is an algorithm which, given a string α and a language $\mathcal{L}$, can determine if $\alpha \in \mathcal{L}$ (i.e., a recognizer decides membership of α in $\mathcal{L}$).

A Parser is an algorithm which, given a string α and a grammar for a language, $\mathcal{L}$, can recover a sequence of rewrite rules which produces α from the start symbol, S, or determines that $\alpha \notin \mathcal{L}$. Also, by the current restrictions on rewrite rules, there may be multiple sequences of rewrite rules which produce α from S for a given language. This is called ambiguity, and is the subject of a great deal of study to determine whether a grammar describing a language can produce a string from the language in multiple ways.

Sometimes the notion of a parse tree is mentioned in the literature. The internal nodes of a parse tree are nonterminals, and the root is the start symbol. The leaves are terminals. If a symbol is produced by the replacement of a nonterminal, it is a branch of that node. The left to right ordering of nodes represents their ordering in the sentential form.

Much of the material from this section and the presentation of bit strings was adapted from [17].

CHAPTER 2
THE CHOMSKY HIERARCHY

In a series of papers published by Noam Chomsky in the late 1950's, the Chomsky Hierarchy of languages was established, along with the modern basis for the study of grammars. Chomsky's study was linguistics, but the budding field of computer science found his formalism was suitable for use in describing computer languages. The formal description of the programming language Algol 60 used the BNF (Backus-Naur Form) notation described by John Backus and Peter Naur and was based on Chomsky's work.

The Chomsky hierarchy describes an increasing series of restrictions on rewrite rules to create four sets of rewrite rules. A language is considered to be a member of a class if it can be described by a set of rewrite rules in that class. However, a language can be described (with varying degrees of difficulty) by multiple classes of grammars.

Chomsky's hierarchy is a set of increasing restrictions on the kind of rewrite rules allowed. Yet as the restrictions increase, the languages that can be described become simpler. The least restrictive grammar classes (Type 0, phrase structure grammars) allow the most refinement in deciding which strings are members of the language being described, so while all Type 1 languages can be described with a Type 1 or a Type 0 grammar, not all Type 0 languages can be described by a Type 1 grammar. When a language is classified by the hierarchy it is considered to be a member of the most restrictive class which still allows the language to be defined by a grammar of that class.

2.1 The Backus-Naur Form

There are many notations to describe rewrite rules, and many differ only in cosmetic details, but these details matter. The Backus-Naur form (BNF) is a very popular notation, and will be constantly used in examples in this thesis. In the author's opinion, it is a much more readable notation than many of the other options. The BNF notation is also the most well known notation among practicing computer scientists, and has superseded other notations used in the older literature on parsing (in particular the Wijngaarden notation used by other formalisms). For a good discussion of the Wijngaarden notation, see [5].

In the BNF notation, a nonterminal is denoted in angle brackets $\langle\ \rangle$, and nonterminals are simply Latin letters (or an appropriate symbol set for the language). The start symbol is still subscripted with an S. Also, in the case of multiple rules with the same left hand side, the various right hand sides are distinguished with a |(which can be read like a logical 'or'). So consider the example grammar from Chapter 1.

$$\langle S\rangle_S \rightarrow \langle A\rangle\ \langle B\rangle\ \langle C\rangle$$
$$\langle A\rangle \rightarrow a\ \langle A\rangle \mid a$$
$$\langle B\rangle \rightarrow b\ \langle B\rangle \mid b$$
$$\langle C\rangle \rightarrow \langle C\rangle\ c\ \langle C\rangle \mid c \mid \epsilon$$

2.2 Type 0: Phrase Structure Grammars

Type 0 grammars have no restrictions on rewrite rules. Any combination of terminals and nonterminals are permitted on both the left and right hand side of the rule. A Type 0 rule takes 1 or more symbols and replaces them with 0 or more

symbols. A rule which replaces a symbol with nothing is called an ϵ rule (remember ϵ is the string of length 0).

2.3 Type 1: Context Sensitive Grammars

Type 1 grammars come in two varieties, but both are equally powerful, and colloquially called Context Sensitive Grammars. Type 1 languages are much more interesting, if nothing else, because they can always be parsed in Exponential time, and there are no theoretical limitations on the existence of reasonably efficient parsers.

A Type 1 rule is context sensitive if only one nonterminal in the left hand side is replaced on the right hand side, while all other symbols remain unchanged. The symbols left unaltered are called the **context** of the rule. Since the context matters, replacements of nonterminals are sensitive to the context. A Type 1 rule is monotonic if the left hand side consists of fewer symbols than the right hand side, or the same number of symbols as the right hand side.

A grammar is Type 1 context sensitive if all the rules are context sensitive. A grammar is Type 1 monotonic if all the rules are monotonic.

Theorem 2. *For any Type 1 context sensitive grammar, there exists a Type 1 monotonic grammar describing the same language [17].*

The language $a^n b^n c^n$ is often cited as the "simplest" example of a context sensitive language. The exponentiation is often a shortcut in grammars to represent repetition, so this is the language of all strings with the same number of a's, b's and c's (in that order).

Type 1 Grammar for $a^n b^n c^n$.

$$\langle S \rangle_S \rightarrow abc \mid a \langle S \rangle \langle Q \rangle$$
$$b \langle Q \rangle c \rightarrow bbcc$$
$$c \langle Q \rangle \rightarrow \langle Q \rangle c$$

Of course, the language $a^n b^n c^n d^n$ is also Type 1 as well, and it is here that the shortcomings of Type 1 grammars can be seen. While some of the techniques are similar, this is not a simple addition to the previous grammar, and involves changes to almost all the rules along with several new rules.

Type 1 Grammar for $a^n b^n c^n d^n$.

$$\langle S \rangle_S \rightarrow abcd \mid a \langle S \rangle \langle Q \rangle$$
$$c \langle Q \rangle d \rightarrow \langle B \rangle ccdd$$
$$d \langle Q \rangle \rightarrow \langle Q \rangle d$$
$$b \langle B \rangle \rightarrow bb$$
$$c \langle B \rangle \rightarrow \langle B \rangle c$$

2.4 Type 2: Context Free Grammars

A rewrite rule is type 2 context free if the left hand side consists of only a single nonterminal.

Context Free languages and Context Free Grammars (hereafter CFG's) are the most popular of the Chomsky hierarchy, but several interesting (and difficult) problems remain open. One open problem is to produce an efficient linear time ($O(n)$) parser for ALL Context Free languages. It is NOT known whether a general

unrestricted CFG can be parsed in linear time. Automatic methods can create linear time parsers for subsets of CFG's such as $LL(k)$, $LALR(k)$, but the most efficient general parsers only work in $O(n^3)$, with the most efficient one being $O(n^{2.87})$. Even more frustrating is that no one has ever created a CFG for which an "ad hoc" parser could not be constructed by hand that worked in linear time. These results, along with more details are explored in depth with further references in [17].

In context of the previous example, the language $a^n b^n c^m$ is almost the language $a^n b^n c^n$. A context free language can enforce that the number of a's match the number of b's (or that the number of b's and c's match), but not all three. The reason is the Pumping Lemma for Context Free Languages, which dictates that all strings in the language can be divided into five pieces, $uvwxy$, and there is a set of basis strings which produce all the strings of the language in the form $uv^n wx^n y$. It is easily shown that all strings of the form $a^n b^n c^n$ cannot be divided in such a manner (however, the language $a^n b^n c^m$ and its variants can, and contain the language $a^n b^n c^n$ as a sublanguage).

$$\langle S \rangle_S \rightarrow a \ \langle T \rangle \ b \ \langle Q \rangle \mid ab$$

$$\langle T \rangle \rightarrow a \ \langle T \rangle \ b \mid ab \mid \epsilon$$

$$\langle Q \rangle \rightarrow c \ \langle Q \rangle \mid c$$

2.5 Type 3: Regular Grammars

A rewrite rule is type 3 regular if its left hand side consists of only 1 nonterminal, and its right hand side consists of zero or more terminals followed by at most 1 nonterminal.

Regular languages encompass what are called regular expressions by programmars, and while there are problems that remain open, they have little impact on the usability of Regular Grammars in practice, and there is no pressing interest in solving them. Regular languages are often used to describe the "lexing" aspect of many parsers, which creates some problems as well as solving others (such as ignoring white space). Most of the problems remaining with regular languages are practical engineering efforts to design optimal parsing engines that are easy to use.

At this point, given the example language $a^n b^n c^n$, it is no longer possible to restrict the number of a's to match the number of b's. This can be shown through use of another Pumping Lemma, similar to the Context Free one, except now the strings must be of the form $uv^n w$.

$$\langle S \rangle_S \rightarrow a \langle A \rangle \mid a$$
$$\langle A \rangle \rightarrow a \langle A \rangle \mid a \langle B \rangle \mid a$$
$$\langle B \rangle \rightarrow b \langle B \rangle \mid b \langle C \rangle \mid b$$
$$\langle C \rangle \rightarrow c \langle C \rangle \mid c$$

2.6 Finite Languages

Although not part of the original Chomsky hierarchy, a rewrite rule presents a "finite choice" language if the left hand side consists of only 1 nonterminal, and the right hand side consists of only terminals. Any other nonterminal besides the start symbol will be ignored, and any finite language can be described this way.

2.7 Syntax And Semantics Revisited

It is traditional to consider a feature of a language to be syntactic if it can be described by a Type 2 context free grammar. However, in the attempt to "interpret" a language, specific meanings are attached to nonterminals and terminals, and a sequence of transformations can be applied to the parse tree. The "meaning" of the tree is then the final state of the tree. The transformations could be a series of reductions, which can perform basic arithmetic, or could be a rearrangement of nodes into terminals and nonterminals from another language (which could be a translator English to French, or C++ into Java).

In this view, it is tempting to consider the nonterminals as the basic underpinning for the meaning. In fact, many grammars are written with suggestive names for the nonterminals which lead to a temptation to interpret a language by the nonterminals. But in the discussion of grammar classes, no meaning was attached to nonterminals, and if two grammars produced the same set of strings, the languages produced by the grammars are considered to be the same. In fact, many transformations can be applied to grammars to convert them into various normal forms. Some of these transformations require the introduction of new nonterminals, or the elimination of nonterminals and merging their productions. This is very inconvenient if the nonterminals are the basic unit of meaning in a language.

In this case, a language may be context free, but the grammar preserving the nonterminals may be forced to be context sensitive. This is the difficulty of semantics. Also, the ambiguous case of two sequences of productions leading to the same final sentential form is another difficulty, since different sequences would indicate different "semantics".

Also, the notion that a context free language cannot express context sensitive information is slightly misleading. A context free grammar can only express a finite number of context dependencies. For example, consider a programming language where a variable must be declared before it is used in an expression. It is possible to write context free rules that force a particular variable, like x, to be declared before it is used. Since most programming languages do not place an upper bound on the number of variables that can be declared, it is not possible to write the infinite set of rules required to express that any variable must be declared. Of course, in practice, only a small, finite number of variables will actually be used, but it would clutter the grammar with a very large number of rules (for variable names with 10 lowercase letters, the number of variables is already 26^{10}), however, a diligent reader might observe that these rules would all take very similar forms, and could be produced by another algorithm, and that is the central insight of van Wijngaarden grammars and the family of two level grammars.

The issue is further blurred by the fact that context sensitive grammars can express (albeit with much difficulty) many issues traditionally ascribed to semantics, such as declaration of variables before use, and data typing rules in programming languages. In fact, Turing machines may be constructed from Type 0 rules and vice versa, so the rewriting process, applied to Type 0 rules generated from a Turing Machine, would execute the program described by that Turing Machine. The line between grammars and meaning is very fuzzy. It is also possible to write Type 0 grammars that describe the set of all true statements from a set of axioms (although very difficult). In this case, a parser for a Type 0 grammar would generate the sequence of productions required to generate a statement, or decide that the statement

is not a member of the language. A parser for a Type 0 language would be a theorem prover! Grammars can go right to the heart of mathematics and formal logic. There are, unfortunately, strong theoretical reasons why a parser for a general Type 0 grammar cannot be constructed, especially one which would always halt, or not get stuck in loops of generating vacuously true statements like $1 = 1, 2 = 2, \ldots$.

However, Type 1 languages, which pose no such theoretical limitations, show promise of being practically implementable. A parser for a Type 1 language would be suitable as a proof checking tool (such as Mizar). However, Type 1 grammars are difficult to write, understand, and modify.

Two level systems, which include most formal systems of semantics, provide a different approach. In a two level system, the first level is often a Context Free Grammar, and the second level, the meta-system, operates on an abstract representation of the first system, through the use of metavariables and consistent substitution rules.

In fact, the quest to create a usable semantic formalism often involves the use of a two-level system whose expressive power is contained in the consistent substitution rules for metavariables. However, the complexity of most two-level systems makes computer implementation of practical parsers almost impossible if not in theory, then at least in practice.

Ultimately a formal language represents a formal system, and in that regard, the semantics and syntax of the language completely specify the behaviour of the system. The expressive power of the formalism as a whole is what is important, especially when coupled with facts about its decidability.

CHAPTER 3
COMPUTABILITY THEORY

While the issue of the dividing line between syntax and semantics is an issue of philosophy (at least in mathematics), the systems created for the specification of semantics are useful, and represent an important alternative to grammar based systems both as a benchmark, a source of ideas, and even components of other grammatical formalisms. Many basic computability results for grammars also involve the two foundational formal models of computable functions. The Lambda Calculus of Church is also a foundation for both Operational and Denotational Semantic Models as well.

3.1 Turing Machines

Turing Machines, described by Alan Turing during the 1930's, are one of the oldest models of mechanical computation.

Turing Machines have a tape which is used for the starting configuration, and is written to during the running of the machine. A finite alphabet of symbols can be written on the tape. Symbols will be denoted by S_j, where $0 \leq j \leq m$ and m is the number of symbols in the alphabet. The tape is considered to be infinitely long, but the initial configuration is of finite length, and the initial position of the head is known.

Turing machines also have a finite number of states, labeled q_i, where $1 \leq i \leq R$ where R is the number of states. The machine starts in a predetermined state.

Instructions for a Turing Machine are written as a 4-tuple:

$$(q_i, S_j, I, q_l).$$

The I is the instruction to be performed. It can be either an L, R, or S_k where $0 \leq k \leq m$. The tuple can be read as an if-then instruction. If the machine is in state q_i and the current symbol under the head is S_j, then perform the instruction I and transition to state q_l. If I is an L then the head is moved left one cell of the tape. If I is an R, the head is moved right one cell of the tape. If I is S_k, then the symbol S_k replaces the symbol S_j at the current tape position. Also, there can be at most only one tuple having q_i and S_j, and while there are variations eliminating this restriction involving nondeterminism, it is not necessary to remove that restriction, so this will keep things simple. A Turing Machine is considered to be halted if there is no q_i and S_j in the instruction table matching the current configuration of the machine. Also, blanks on the tape are represented by the symbol S_0, which is implicitly on the ends of the configuration tape, repeating off to infinity in both directions.

Table 3.1. Example of an initial configuration tape

S_1	S_2	$\overbrace{S_3}^{q_1}$	S_4	S_5	S_6

3.2 The Halting Problem

While the Halting Problem is normally phrased in terms of Turing Machines, the proof is clearer when presented in a function-based notation. The basic question is whether there exists a computable function which decides if every computable

function, on a given input will terminate with a value (in the lambda calculus, a function would not halt if it does not reduce to a value).

Theorem 3. *Given a function f and its argument x, there does not exist a function which, given f and x which evaluates to* 1 *if $f(x)$ is defined, and* 0 *if $f(x)$ is undefined.*

Proof. Assume such a function exists, $halt(f, x)$. We can define another function $contradiction(s)$ such that if $halt(s, s) = 1$ (i.e. s does not halt when given itself as its input), $contradiction(s)$ evaluates to 0, and is undefined if $halt(s, s) = 0$.

Now, what is the value of $contradiction(contradiction)$? Assume it is defined; that means:

$$halt(contradiction, contradiction) = 1$$

But that means $contradiction(contradiction)$ is undefined, which contradicts our assumption. Assume it is undefined; that means:

$$halt(contradiction, contradiction) = 0$$

But that means $contradiction(contradiction)$ is defined, which also contradicts our assumption. Therefore, no such function $halt(f, x)$ can exist. □

Not all is lost; Partially computable functions can be defined which decide if a function halts on a given input (it can simulate the execution of the function, and if the function terminates, then it can evaluate to 0), and decide in some cases it will not halt (the function repeats a previous state of simulation) and return 1, and is undefined otherwise.

3.3 Semi-Thue Processes

The notion of rewrite rules are also known as a semi-Thue process. Given a pair of words on an alphabet, g and g', a semi-Thue production is written:

$$g \rightarrow g'$$

If P is a semi-Thue production $g \rightarrow g'$, then $u \Rightarrow_P v$ means that there are words (possibly null) r and s such that by replacement of g with g' then v can be derived from u.

$$u = rgs \text{ and } v = rg's$$

A semi-Thue process is a finite set of semi-Thue productions, and if there is a finite sequence of productions such that

$$u \Rightarrow u_1 \Rightarrow \ldots \Rightarrow u_n \Rightarrow v$$

then to state that v is derivable from u write $u \Rightarrow^* v$.

A grammar is a special case of a semi-Thue process where the words are divided into two categories (terminals and nonterminals). The set of all derivations from the start symbol $S \Rightarrow^* v$ such that v contains only terminals is the transitive closure of that symbol and is equivalent to the language generated by the grammar.

Semi-Thue processes are equivalent in power to a Turing Machines. While this result may seem surprising at first, it is easy to show that a set of productions can be created which will simulate the execution of an arbitrary Turing Machine. Since

phrase structure grammars (Chomsky Type 0) are just a special case where the words are divided into two categories, this result also shows the equivalent computability of Grammars and Turing Machines (a case of the Church-Turing thesis).

Starting with a Turing Machine $\mathcal{M}$ with alphabet $s_1, \ldots, s_k$ and states $q_1, \ldots, q_n$. The semi-Thue process will have the alphabet:

$$\Sigma = \{s_0, s_1, \ldots, s_k, q_1, \ldots, q_n, q_{n+1}, h\}.$$

Machine configurations will be represented by words. For example, the configuration: will be represented by the word $hs_1s_2q_1s_3s_4s_5s_6h$.

s_1	s_2	$\overbrace{s_3}^{q_1}$	s_4	s_5	s_6

The h's are used as markers to denote the beginning and end of a configuration, and the state is placed preceding the symbol at which it is currently positioned. The s_0 letter is used to represent blanks on the configuration tape, and so multiple words may be used to represent the same configuration since it is possible to omit any number of leading and trailing blanks. The q_{n+1} state is used to denote the halt state.

Now, the 4-tuples used by the Turing Machines are of the form (q_i, s_j, I, q_l) where q_i is the current state, s_j is the current letter under the read head, I is an instruction to either advance left, right, or to write the character s_m to the tape, and q_l is the new state. Productions can now be associated with these tuples.

In the case where the tuple is of the form (q_i, s_j, s_m, q_l) add the following production:

$$q_i s_j \rightarrow q_l s_m.$$

In the case where the tuple is of the form (q_i, s_j, L, q_l) add the following productions:

$$s_m q_i s_j \to q_l s_m s_j, \text{ where } m = 0, \ldots, k$$
$$h q_i s_j \to h q_l s_0 s_j.$$

In the case where the tuple is of the form (q_i, s_j, R, q_l) add the following productions:

$$q_i s_j s_m \to s_j q_l s_m, \text{ where } m = 0, \ldots, k$$
$$q_i s_j h \to s_j q_l s_0 h.$$

Thus, each production will create new words representing the next configuration in the computation. Of course, if the Turing Machine is nondeterministic, the set of words derivable from an initial configuration in n steps will not be unique, but this should not come as a surprise and is in fact an appropriate result. Of course, all this discussion is somewhat intuitive. It will now be shown that this semi-Thue process really does simulate the given Turing Machine.

Theorem 4. *Let $\mathcal{M}$ be a nondeterministic Turing machine. A string α on the alphabet of $\mathcal{M}$ is accepted by $\mathcal{M}$ iff*

$$h q_1 s_0 \alpha h \Rightarrow^* h q_0 h.$$

Proof. $\Rightarrow$. Suppose $\mathcal{M}$ accepts α. So $\mathcal{M}$ begins with the tape empty except for α, and the tape head positioned at the blank preceding α, since it accepts α it will reach a state q_i over symbol s_k such that no quadruple in $\mathcal{M}$ matches. So, given the semi-Thue process as constructed previously, with appropriate words β, ζ, we will

have that:

$$hq_1s_0\alpha h \Rightarrow^* h\beta q_i s_k \zeta h \Rightarrow h\beta q_{n+1} s_k \zeta h \Rightarrow^* h\beta q_0 h \Rightarrow^* hq_0 h$$

Since this is what we wanted, we have proved that if $\mathcal{M}$ accepts α then our semi-Thue process will properly simulate $\mathcal{M}$.

$\Leftarrow$. Suppose $\mathcal{M}$ does not accept α. So, with our initial configuration as before, $\mathcal{M}$ will not halt. Let $w_1 = hq_1s_0\alpha h$, and suppose that:

$$w_1 \Rightarrow w_2 \Rightarrow \ldots \Rightarrow w_m$$

So, each $w_j, 1 \leq j \leq m$, must contain a symbol q_i with $1 \leq i \leq n$. So there can be no derivation from w_1 which contains q_0; in particular, we cannot derive hq_0h. □

The previous theorem was for nondeterministic Turing Machines. Since deterministic Turing Machines are just a special case of nondeterministic machines, the theorem holds for those as well. Much of this discussion was heavily influenced by [8]. Much of the literature, particularly older literature, on parsing refers to semi-Thue processes, but it is rarely defined outside of textbooks.

3.4 The Partial Correspondence Problem

The Partial Correspondence Problem for Languages is based on the Correspondence Problem from Algebra. This particular version of the problem was used by Knuth [18] to simplify decidability results for problems in languages.

> [The Partial Correspondence Problem] *Let* $(\alpha_1, \beta_1), (\alpha_2, \beta_2), \ldots, (\alpha_n, \beta_n)$ *be ordered pairs of non-empty strings. Do there exist, for all*

> $p > 0$, *ordered p-tuples of integers* $(i_1, i_2, \ldots, i_p)$ *such that the first p characters of the string* $\alpha_{i_1}, \alpha_{i_2}, \ldots, \alpha_{i_p}$ *are respectively equal to the first p characters of* $\beta_{i_1}, \beta_{i_2}, \ldots, \beta_{i_p}$*?*

The problem is strange, and so some clarifications and explanations should be made. The idea is that the sequence of i's is used to create two strings, one composed of α's and the other of β's. Since each α and β is at least one character long, the strings created by a sequence of p i's will be at least p characters long.

Now, the partial correspondence problem for a particular set of a ordered pairs is **true** means that no matter what p is chosen, a sequence of i's can always be found which will make two strings that are equal in the first p symbols. The problem is **false** means that for some p, there is no sequence of i's that can be found.

Certainly, for each p, we can try all possible combinations to see if a sequence of i's works. After all, there are only a finite number of ordered pairs and only a finite number of i's. The problem is that the Partial Correspondence Problem is true only if a sequence can be found for all possible values of p.

Another point about this is that if there's a pair such that $\alpha_k = \beta_k$, the problem is trivially true, since a sequence of p k's will always produce two strings that match. The difficulty in this problem lies in the fact that there is no algorithm which can always determine whether it is true or false, such programs will only catch increasingly larger sets of special cases.

The proof technique is particularly convenient. The Halting Problem is a well-known problem which is recursively unsolvable. Given a Turing machine and its input, it is not possible to determine if the machine will halt after a finite amount of time, other than by running the machine. If the Partial Correspondence Problem can be reduced to the Halting Problem, it too is recursively unsolvable.

The value of this fact is that many decidability problems for grammars can be reduced to the Partial Correspondence Problem.

Theorem 5. *The Partial Correspodence Problem is an unsolvable problem. In other words, a recursive algorithm can always tell if the Partial Correspondence is true if and only if a recursive algorithm can be found which always solves the Halting Problem.*

Proof. Given a particular Turing Machine, we are going to construct a Partial Correspondence Problem for that machine such that the Problem is true if and only if the Machine never halts. Let our alphabet be:

$$\vdash, q_i, \bar{q_i}, S_j, \bar{S}_j, h, \bar{h} \text{ where } 1 \leq i \leq R, 0 \leq j \leq m$$

The q_i's and S_j's are from our Turing Machine. $\vdash$, h and $\bar{h}$ are used for special conditions in our Partial Correspondence Problem.

In the initial configuration of the Turing Machine's tape, we will let q_{i_1} denote the position of the tape head and the initial state. So if our initial configuration is:

$$S_{j_1} \cdots S_{j_{k-1}} q_{i_1} S_{j_k} \cdots S_{j_x}$$

Our first pair is:

$$(\vdash, \vdash h S_{j_1} \cdots S_{j_{k-1}} q_{i_1} S_{j_k} \cdots S_{j_x} h)$$

Now, the following pairs will also be added:

$$(\bar{h}, h), (h, \bar{h}),$$

$$(\bar{S}_j, S_j), (S_j, \bar{S}_j), \text{ where } 0 \leq j \leq m$$

$$(\bar{q}_i, q_i) \text{ where } 1 \leq i \leq R$$

Also, for the tuples in the Turing Machine, we will add these pairs as well:

Tuple	Pairs, $0 \leq t \leq m$
(q_i, S_j, L, q_l)	$(hq_iS_j\ ,\ \bar{h}\bar{q}_l\bar{S}_0\bar{S}_j)$, and $(S_tq_iS_j\ ,\ \bar{q}_l\bar{S}_t\bar{S}_j)$
(q_i, S_j, R, q_l)	$(q_iS_jh\ ,\ \bar{S}_j\bar{q}_l\bar{S}_0\bar{h})$, and $(q_iS_jS_t\ ,\ \bar{S}_j\bar{q}_l\bar{S}_t)$
(q_i, S_j, S_k, q_l)	$(q_iS_j\ ,\ \bar{q}_l\bar{S}_k)$

Now, since $\vdash$ only occurs in one pair, and none of the other pairs have the same beginning symbol, if the correspondence problem is to be true, it must start with the first pair, containing the $\vdash$. Also, note that since all the other α_k's and β_k's are equal in length, the β string will always be running ahead of the α string.

Now, the h and $\bar{h}$ are used to denote the end and beginning of a stage of computation. Note that α's will be added to match the symbols of the tape, adding barred versions to the end of the β string, until a q_i is found. At the point, a pair matching the appropriate Turing Machine state will be added to the α string, and the result of that step will be added to the β string. The L and R cases have two pairs in case the head is at the left or right end of the tape, which merely inserts a blank (S_0).

After the computation step is complete, the barred α's will be brought in to play, note that since only unbarred q's occur in the pairs added for the Turing Machine states, and only barred α q's result in an unbarred string, an unbarred version of the previous computation will be appended to β while the α string matches the barred computation. Once the alpha string gets to the end of the copy process, another

barred computation stage begins, so the resulting string takes the form:

$$\vdash \alpha_0\bar{\alpha}_1\alpha_1\bar{\alpha}_2\alpha_2\cdots,$$

where the α_i's represent the state of the tape at the end of each "execution" cycle of the Turing Machine.

Now, it should be clear that, if the Turing Machine never halts, it represents a solution to the Partial Correspondence Problem for values of p, since it will continue to produce matching strings of ever increasing length.

In the case where the machine halts, a (q_i, S_j) pair will have been produced for which no tuple exists in the Turing Machine. Since that tuple did not exist, no pair matching for it would have been added for the Partial Correspondence Problem, so no α will be found matching it, so no more exploration down that route remains. Since there was only one possible starting pair, this is the only path for a solution to the Partial Correspondence Problem, so the Partial Correspondence Problem is false.

This means that the Partial Correspondence Problem is true if and only if the Turing Machine associated with it never halts. Since this is what we were looking for, the Partial Correspondence Problem is unsolvable since the Halting Problem is unsolvable. □

This particular presentation of Post's Correspondence problem was mostly influenced by [18], and was chosen for its relevance to the $LR(k)$ grammars presented in the next chapter. Knuth does not refer to semi-Thue processes in his presentation, but a separate discussion of them alleviates some of the notational burden in explaining the Correspondence problem.

3.5 Recursively Enumerable Languages

Definition 3. *A set S is Recursively Enumerable (abbreviated RE) if it is the range of a partially computable function.*

More simply, if a Turing Machine $\mathcal{M}$ halts with input x, then x is in the range of the function computed by $\mathcal{M}$. If $\mathcal{M}$ does not halt, x is not in range of the function.

How might such a set be enumerated? Since it's potentially infinite, the entire set will never be generated, but the steps of $\mathcal{M}$ can be dovetailed such that, if ran forever, all inputs will be fed to $\mathcal{M}$ so that set generated by $\mathcal{M}$ is produced.

Definition 4. *A set S is Recursive if both it and its complement are the ranges of partially computable functions.*

Interestingly enough, the two partially computable functions can be combined, such that steps are alternated (dovetailing), and make a completely computable function (it always halts) which decides if x is in $\mathcal{M}$ (e.g. by which state it halts in). Such a set is always decidable.

It is rather easy to extend these definitions to languages. Since a language is a set of strings it may be said that:

Definition 5. *A language is Recursively Enumerable means if it is generated by a partially computable function.*

Definition 6. *A language is decidable if it is a recursive set.*

Of course, with all these definitions, several questions arise about containment of RE sets and recursive sets. Any good reference on computability will provide these theorems and their proofs, but at least one interesting example will be considered, a set which is not recursively enumerable.

In the study of computable functions, it is common to convert them into numbers, using a variation on Gödel numbers, and thus it is possible to represent functions

on numbers as numbers. Of course, a function $f(x)$ on the integers is a total function if it halts for every integer x.

Let $\mathcal{T}$ be the set of all numbers p such that p is the number of a total function on one parameter ($f(x)$). First a lemma is needed, whose proof is omitted.

Theorem 6. *If S is a nonempty, recursively enumerable set, then there is a primitively recursive function $f(u)$ such that $S = \{f(n)|n \in \mathbb{N}\} = \{f(0), f(1), f(2), \ldots\}$. That is, S is the range of f.*

Theorem 7. *$\mathcal{T}$ is not recursively enumerable.*

Proof. Suppose $\mathcal{T}$ were recursively enumerable. By the previous theorem, there is a computable function $g(x)$ such that $\mathcal{T} = \{g(0), g(1), \ldots\}$.

Now, $g(x)$ gives the number of a total function, f. Now, since f is a total function of one parameter, we may apply f to any number we want. We will define a function $h(x) = f(x) + 1$, where f is the function whose number is $g(x)$. For every x, $g(x)$ is defined, and since the function f whose number is $g(x)$ is total, $f(x)$ is computable as well. Finally, since addition by one is computable, we may say that $h(x)$ is a total function, since its value for every x is computable.

Let p be the number of h, since h is a total function, $p \in \mathcal{T}$, so there is some i such that $g(i) = p$.

So what is the value of $h(i)$? Well, $h(x)$ is the function whose number is $g(i)$. Recall, $h(x) = f(x) + 1$, but in this case, $h(i)$, $f = h$, so:

$$h(i) = h(i) + 1$$

But $h(i)$ is an integer, and there is no integer which satisfies this equation, so we have a contradiction. We assumed that $\mathcal{T}$ was recursively enumerable; therefore, $\mathcal{T}$ must not be a recursively enumerable set. □

While contradiction proofs can be frustrating, an interesting example of a set which is not recursively enumerable has been obtained. The proof is also a diagonalization proof, which is common in this area of computability, and so demonstrates an important technique as well.

CHAPTER 4
SOME PROPERTIES OF CONTEXT FREE GRAMMARS

4.1 $LR(k)$ Property

In 1965, Knuth[18] identified the class of $LR(k)$ grammars and parsing algorithms which characterize the deterministic languages. In the paper, Knuth describes languages that are translatable from left to right, and explains the details of a bottom-up parsing algorithm for parsing them which, given a constant lookahead k, will not backtrack (which makes it a deterministic parsing method). He then shows how to determine whether a given grammar $\mathcal{G}$ is $LR(k)$ for a given k. He then shows that determining whether there exists a k such that a grammar $\mathcal{G}$ is $LR(k)$ is an unsolvable problem. Knuth completes his work by showing that if a grammar is deterministic, there exists a grammar which parses it which is $LR(k)$ for some k, and then he shows that if a grammar is $LR(k)$ then an $LR(1)$ grammar exists which parses the same language.

The distinction between grammars and languages has been made before. In dealing with the $LR(k)$ property it is of utmost importance to realize the distinction.

Definition 7. *A grammar $\mathcal{G}$ is said to be $LR(k)$ with integer k if every handle is uniquely determined by a lookahead of at most k characters.*

Definition 8. *A language $\mathcal{L}$ is said to be translateable from left to right if there exists a grammar which accepts $\mathcal{L}$ for which the $LR(k)$ property holds for some integer k.*

The previous qualification is incredibly important if semantics are involved. Nonterminals are often ascribed meanings for various semantic purposes. A language may be generated by an $LR(k)$ grammar, but the most convenient grammar for

semantic purposes may not be $LR(k)$ for any k. In that case, a decision must be made to modify the grammar and adapt the semantics or use a less efficient parsing method (a third option is available but it is not usually the case: hope the grammar happens to be parsable by some other linear time method such as $LL(k)$).

Definition 9. *A k-sentential form is a sentential form followed by k characters, denoted by $\dashv$ where $\dashv$ is a symbol not in the alphabet of the language in question.*

Definition 10. *(n, p) is a handle of α means that if $\alpha = X_1 \ldots X_n X_{n+1} \ldots X_{n+k} Y_1 \ldots Y_u$ means that $X_1 \ldots X_n$ is obtained by application of the p'th production of the grammar $\mathcal{G}$.*

4.2 $LR(k)$, Given k, Is Solvable

It will now be shown how to determine if a given grammar $\mathcal{G}$ is $LR(k)$, given a particular value of k.

Let I be the set of nonterminals in $\mathcal{G}$ and T be the set of terminals. In order to show that a grammar $\mathcal{G}$ is $LR(k)$ a new grammar will be constructed, $\mathcal{F}$, which produces all possible handles and the k characters to the right of it. Number the productions of $\mathcal{G}$, where A_p denotes the p'th production in $\mathcal{G}$. Let S_G be the start symbol of $\mathcal{G}$ and S_F be the start symbol of $\mathcal{F}$.

The nonterminals of $\mathcal{F}$ will be of the form $\langle A, \alpha \rangle$ where A is a nonterminal from $\mathcal{G}$ and α is a k-letter string on $T \bigcup \{\dashv\}$ where $\dashv \notin T$. The terminals of $\mathcal{F}$ will be $I \bigcup T \bigcup \{\dashv\}$ and symbols of the form $[p]$ where $[p]$ is a symbol for the p'th production of $\mathcal{G}$

Let $H_k(\sigma)$ be the set of all k length strings β over $T \bigcup \{\dashv\}$ such that $\sigma \Rightarrow^* \beta\gamma$ with respect to $\mathcal{G}$ for some γ. There is a set of strings derivable from σ by $\mathcal{G}$. $H_k(\sigma)$ gives all the strings of length k that are the first k characters of those strings.

Now, in $\mathcal{G}$, let the p'th production be of the form.

$$\langle A_p \rangle \rightarrow X_p^1 \dots X_p^n$$

Now, add to $\mathcal{F}$ all productions of the form:

$$\langle A_p, \alpha \rangle \rightarrow X_p^1 \dots X_p^{j-1} \langle X_p^j, \beta \rangle$$

where $1 \leq j \leq n$ and X_p^j is a nonterminal, and α and β are k-letter strings over $T \bigcup \{\dashv\}$ and β is in $H_k(X_p^{j+1} \dots X_p^n \alpha)$.

Also, add productions of the form:

$$\langle A_p, \alpha \rangle \rightarrow X_p^1 \dots X^n \alpha \; [p \,]$$

to $\mathcal{F}$ as well.

Now, to show that it is true with respect to $\mathcal{F}$ that

$$\langle S_G, \dashv^k \rangle \Rightarrow X_1 \dots X_n X_{n+1} \dots X_{n+k} \; [p \,]$$

if and only if there exists a k-sentential form $X_1 \dots X_n X_{n+1} \dots X_{n+k} Y_1 \dots Y_u$ with handle (n, p) and $X_{n+1} \dots Y_u$ terminals.

Now, by definition $\mathcal{G}$ will be $LR(k)$ if and only if $\mathcal{F}$ satifies the property that if

$$\langle S_G, \dashv^k \rangle \Rightarrow \theta \; [p \,]$$
$$\langle S_G, \dashv^k \rangle \Rightarrow \theta\phi \; [q \,]$$

implies $\phi = \epsilon$ and $p = q$.

Of course, since $\mathcal{F}$ is a regular language (Chomsky Type 3) it is easy to test this property of $\mathcal{F}$.

This is not the most computationally practical test for the $LR(k)$ property. In fact [18] presents two different tests. There are several others, some of which construct parsers for the language generated by $\mathcal{G}$.

4.3 $LR(k)$ Is An Unsolvable Problem

Earlier, Knuth's algorithm for determining if a grammar is $LR(k)$ for a given k was explained. Unfortunately, given a grammar, it is not possible to find a k for which that grammar is $LR(k)$. This is shown by reducing the $LR(k)$ problem to the Partial Correspondence Problem. The basic idea is that if the Partial Correspondence Problem is positive, then the grammar is not $LR(k)$ for any value of k. However, if the result is negative, the grammar is not only $LR(1)$, it is a bounded context grammar.

Now, since there is a test to determine if a given grammar is $LR(k)$ for some k, it is intuitive that the problem of finding the k is unsolvable. The test takes a finite amount of time, and so it is possible to write a loop that would start at 0 and test the grammar for each value k. If the grammar is $LR(k)$ for some k, after a while, the test will eventually find it, but if the grammar is not $LR(k)$, the test will always fail, and so the program will never halt. So, given such a program, and the solution to the halting problem, it is easy to see if such a k does not exist, and if it does, the program is executed to find the k, but since the halting problem is recursively unsolvable, so is this solution.

To restate the theorem from [18] (page 627):

Theorem 8. *The problem of deciding, for a given grammar $\mathcal{G}$, whether or not there exists $k \geq 0$ such that $\mathcal{G}$ is $LR(k)$, is recursively unsolvable.*

Proof. We shall prove this theorem by reducing the Partial Correspondence Problem to the $LR(k)$ problem for a particular grammar class. Let $(\alpha_1, \beta_1), \ldots (\alpha_n, \beta_n)$, be the pairs for the Partial Correspondence Problem, and let

$$X_1, X_2, \ldots, X_n, +$$

be $n+1$ symbols, which do not appear in the α's and the β's.

Let $\mathcal{G}$ be our grammar:

$$S \to A$$

$$S \to B$$

$$A \to X_i + \alpha_i, \text{ where } 1 \leq i \leq n$$

$$A \to X_i A \alpha_i, \text{ where } 1 \leq i \leq n$$

$$B \to X_i + \beta_i, \text{ where } 1 \leq i \leq n$$

$$B \to X_i B \beta_i, \text{ where } 1 \leq i \leq n$$

Now, the sentential forms of this grammar are:

$$\{X_{i_m} \cdots X_{i_1} A \alpha_{i_1} \cdots \alpha_{i_m}\} \cup \{X_{i_m} \cdots X_{i_1} B \beta_{i_1} \cdots \beta_{i_m}\} \cup$$
$$\{X_{i_m} \cdots X_{i_1} + \alpha_{i_1} \cdots \alpha_{i_m}\} \cup \{X_{i_m} \cdots X_{i_1} + \beta_{i_1} \cdots \beta_{i_m}\}$$

where $\langle i_1, \ldots, i_m \rangle$ are sequences of integers such that each $1 \leq i_k \leq n$.

Now, $\mathcal{G}$ is $LR(k)$ for some k if and only if the Partial Correspondence Problem has a negative answer.

Consider if the Partial Correspondence Problem is positive. That means that for each $p \in \mathbb{N}$ there exists a sequence $\langle j_1, \ldots, j_p \rangle$ such that $\alpha_{j_1} \cdots \alpha_{j_p}$ and $\beta_{j_1} \cdots \beta_{j_p}$ match in the first p characters. That means that there are sentential forms $X_{j_p} \cdots X_{j_1} + \alpha_{j_1} \cdots \alpha_{j_p}$ and $X_{j_p} \cdots X_{j_1} + \beta_{j_1} \cdots \beta_{j_p}$ such that the first p characters after the $+$ match. Let q be the maximum length of α_i's and β_i's. Now, a grammar is $LR(k)$ means that it is uniquely determined by the left hand side of the handle and the k characters following the handle. So, our handle for the rule $X_i + a_i$ will need to be q characters long, however, for each $p \in N$, there exists a sentential form such that the $p - q$ characters following the handle match, so this grammar is not $LR(p - q)$ when the Partial Correspondence Problem is true.

Now consider if the Partial Correspondence Problem is false. That means there is a p for which the first p characters of $\alpha_{j_1} \cdots \alpha_{j_p}$ and $\beta_{j_1} \cdots \beta_{j_p}$ do not match, and our sentential form is $X_{j_t} \cdots X_{j_1} + \alpha_{j_1} \cdots \alpha_{j_t}$ or $X_{j_t} \cdots X_{j_1} + \beta_{j_1} \cdots \beta_{j_t}$. We have the sequence $\langle j_1, \ldots, j_{\min(p,t)} \rangle$ such that we can determine whether there is a string of α's or β's after the $+$, but in this case, we do not need to know more than p characters to the left of the handle as well. This is not even an $LR(p-q)$ grammar, $\mathcal{G}$ is simply a bounded context grammar.

So if the Partial Correspondence Problem is true, $\mathcal{G}$ is not $LR(k)$ for any k, and if the problem is false $\mathcal{G}$ is a bounded context grammar. Since the Partial Correspondence Problem is recursively unsolvable, the problem of deciding, for a given grammar $\mathcal{G}$, whether or not there exists a $k \geq 0$ such that $\mathcal{G}$ is $LR(k)$ is recursively unsolvable. $\square$

CHAPTER 5
RESULTS

In order to talk about grammars with an infinite number of productions, an example will first be considered which demonstrates the notion and gives some basic insight into infinite grammars.

5.1 Van Wijngaarden Grammars

The theoretical and practical success of Context Free grammars was due to their conceptual simplicity and the ability to use them to automatically generate tools. Chomsky Type 1 languages, while simple, are difficult to use, and lack the conciseness of context free definitions. Seeking to remedy this, Aad van Wijngaarden created a two level system, based on context free grammars.

The essential insight is that Context Free grammar rules can express only a finite number of context conditions. It intuitively follows that an infinite collection of context free grammar rules may be able to express an infinite number of context conditions. Also, the BNF syntax can be described by a context free grammar in BNF syntax, and so it follows quite reasonably that a context free grammar meta system could describe a concrete context free grammar, with a potentially infinite rule set. This is the basis of van Wijngaarden grammars. The family of two-level grammars is a collection of various metasystems which sit atop a context free basis. It is then the hope that the theory of context free grammars can be used to make a tractable system, while retaining the conceptual simplicity of context free grammars.

The parts of a van Wijngaarden grammar are[12]:

M a finite set of **metanotions**

V a finite set of **metaterminals**, $M \bigcap V = \emptyset$.

N a finite set of **hypernotions**, a finite subset of $(M \bigcup V)^+$.

T a finite set of **terminals**

R_M a finite set of **metarules**, $X \rightarrow Y$, where $X \in M, Y \in (M \bigcup V)^*$,
such that for all $W \in M, (M, V, W, R_M)$ is a context free grammar.

R_V a finite set of **hyperrules**
$h_0 \rightarrow h_1\ h_2\ \ldots\ h_m$
where $h_0 \in N$, $h_i \in (T \bigcup N \bigcup \{\epsilon\})$, such that $1 \leq i \leq m$.

$S \in N$ the start symbol.

Given a van Wijngaarden grammar $\mathcal{G} = (M, V, N, T, R_M, R_V, S)$, the set R_S is the set of strict rules of a hyperrule (i.e. the context free rules generated by the hyperrule via consistent substitution)

$$r = h_0 \rightarrow h_1\ h_2\ \ldots\ h_m$$

containing the $n \geq 0$ metanotions $W_1, W_2, \ldots, W_n$:

$$R_S(r) = \{\phi(h_0) \rightarrow \phi(h_1)\ \phi(h_2)\ \ldots\ \phi(h_m)\}$$

ϕ is a homomorphism such that $\phi(v) = v$ for all $v \in V$ or $v = \epsilon$, $\phi(h_0) \neq \epsilon$, and $\phi(W_i) \in L((M, V, W_i, R_M))$ (i.e. $\phi(W_i)$ is in the language generated by the context free grammar (M, V, W_i, R_M). ϕ is called a consistent substitution. ϕ gives a sentence in V which has no metanotions in it.

N_S is the set of strict notions, defined:

$$N_S = \{\phi(h) | \phi \text{ is a consistent substitution and } h \in N\}$$

The language generated by $\mathcal{G}$, $L(\mathcal{G})$, is the language generated by the set of strict rules as potentially infinite context free grammar.

Less formally, a hyperrule is a template for a context free rule. Hyperrules can have metanotions embedded in them, as part of the rule name or a terminal, this is called a hypernotion. The metanotions are replaced by metaterminals generated from the metasystem grammar. Metanotions must be replaced in a hyperrule by consistent substitution with a production from the metasystem. Consistent substitution says all occurrences of a metanotion must be replaced by the same metaterminal (different productions of metaterminals create different instances of a hyperrule). A metanotion is a context free rule describing the production of a metaterminal.

If the metasystem produced a finite language, a context free grammar with a finite number of rules would be the result, and so nothing more would be produced. However, in the case that the metasystem produces an infinite set of strings, a potentially infinite collection of context free rules would be the result.

It can be shown that an infinite collection of Context Free rules is capable of producing any Type 0 language [33].

5.2 BNF Syntax Of Van Wijngaarden Grammars

It can help clear up the understanding of van Wijngaarden grammars to actually see one. Consider the simple context sensitive language $a^n b^n c^n$.

Wijngaarden Grammar for $a^n b^n c^n$.

$$N \rightarrow i \mid i\ N$$

$$A \rightarrow a \mid b \mid c$$

$$\langle text \rangle_S \Rightarrow \langle aN \rangle\ \langle bN \rangle \langle cN \rangle$$

$$\langle Ai \rangle \Rightarrow A$$

$$\langle AiN \rangle \Rightarrow A \langle AN \rangle$$

In the terms of the formal notation:

- $M = \{N, A\}$, metanotions.
- $V = \{a, b, c, i\}$, metaterminals.
- $N = \{aN, bN, cN, Ai, AiN, AN, A\}$, hypernotions.
- $T = \{a, b, c\}$, terminals
- R_M are the first two rules N, A.
- R_V are the last three rules $\langle text \rangle_S, \langle Ai \rangle, \langle AiN \rangle$
- S is $\langle text \rangle$.

For reference, here is the Type 1 Grammar for the same language.

$$S_S \rightarrow abc \mid\ aSQ$$

$$bQc \rightarrow bbcc$$

$$cQ \rightarrow Qc$$

Although any tool will show its best uses first, compared to the Type 1 grammar it is both more concise and more readable. The consistent substitution principle uses the N as a "counter" for the i's and A is used a shortcut to make three rules for

both the second and third hyperrules (Ai and AiN). The power of this example is even better when it is extended to $a^n b^n c^n d^n$:

Wijngaarden Grammar for $a^n b^n c^n d^n$.

$$N \rightarrow i \mid iN$$

$$A \rightarrow a \mid b \mid c \mid d$$

$$\langle text \rangle_S \Rightarrow \langle aN \rangle \langle bN \rangle \langle cN \rangle \langle dN \rangle$$

$$\langle Ai \rangle \Rightarrow A$$

$$\langle AiN \rangle \Rightarrow A \langle AN \rangle$$

Compared to the contortions required in the Type 1 version, the clarity is much appreciated. The book [5] is a excellent source for techniques involved in writing Wijngaarden grammars, however, another another very important technique will be considered.

In Context Free Grammars, ϵ rules reduce to nothing, and although theoretically they pose no problems, are a minor inconvenience to many parsing algorithms which must be extended to handle them, or process the grammar to eliminate them. However, hyperrules which reduce to ϵ are very special, and while some decidability restrictions eliminate, they are the major technique for writing "predicates" in the Wijngaarden grammars.

5.3 Undecidability Of Van Wijngaarden Grammars

Wijngaarden grammars are equivalent to Chomsky Type 0 grammars, and thus it follows that they are, in general, undecidable. The proof is due to [10], but also shows some common techniques in writing a Wijngaarden grammar.

Theorem 9. *For every Type-0 grammar, there is an equivalent Wijngaarden grammar.*

Proof. Let $\mathcal{G}$ be a Type-0 grammar. For any Type-0 grammar, there exists an equivalent grammar in separated normal form, such that each rule is of one of the 3 forms:

$$\langle l_1\rangle\langle l_2\rangle \ldots \langle l_n\rangle \rightarrow \langle r_1\rangle\langle r_2\rangle \ldots \langle r_m\rangle \;\; (m, n \geq 1)$$
$$\langle l\rangle \rightarrow \alpha$$
$$\langle l\rangle \rightarrow \epsilon$$

Where $\langle l_i\rangle$, $\langle r_k\rangle$, and $\langle l\rangle$ are nonterminals, and α's are terminals.

Let $\langle s\rangle$ be the start symbol of the separated version of $\mathcal{G}$. An equivalent Wijngaarden grammar is as follows. The metasystem is:

$$A \rightarrow s \mid a_1 \mid a_2 \mid \ \ldots \ \mid a_k$$
$$L \rightarrow \epsilon \mid LA$$
$$R \rightarrow L$$

A produces all the nonterminals of the Type-0 grammar.

Hyperrules. For each rule of the first form:

$$\langle L\ l_1\ l_2\ \ldots\ l_n\ R\rangle \Rightarrow \langle L\ r_1\ r_2\ \ldots\ r_m\ R\rangle$$

For each rule of the second form:

$$\langle L\ l\ R\rangle \Rightarrow \langle L\rangle\alpha\langle R\rangle$$
$$\langle l\rangle \Rightarrow \alpha$$

For each rule of the last form:

$$\langle L\ l\ R\rangle \Rightarrow \langle L\ R\rangle$$

And the start symbol is $\langle s\rangle$. □

What is perhaps more surprising is that the choice of the metasystem has little effect on the power. A regular metasystem is just as powerful as a Context Free metasystem. This can even be seen in the construction of a Wijngaarden grammar from a Chomsky Type 0 grammar, because all the metanotions are regular. Of course, the choice of a regular system may make the construction of a grammar more difficult. More usefully, one hopes to find a minimal restriction to the metasystem such that the resulting system is decidable. [10] and [16] both give exactly these sorts of restrictions. The ability to make use of those restrictions are explored in [35], [12], [14].

5.4 Infinite Grammars

Definition 11. *An infinite grammar has an infinite number of productions, each with only a finite number of different right hand sides.*

Definition 12. *An infinite grammar G that is a member of grammar class K is said to be* **nontrivial** *if the language $\mathcal{L}(G)$ is not in the class of languages generated by K.*

Consider the following example of a nontrivial infinite grammar.

Theorem 10. *The class of languages generated by regular metasystem with a regular strict grammar can generate a context-sensitive language (i.e. a nontrivial language).*

Proof. Consider the language $a^n b^n c^n$. It is an example of a context-sensitive language. If the theorem holds, then a regular metagrammar can produce an (infinite) regular grammar describing the language.

$$N \rightarrow \epsilon \mid i \mid iN$$

$$N1 \rightarrow N$$

$$N2 \rightarrow N$$

$$\langle S \rangle_S \Rightarrow a \ \langle a\_ \ i \rangle$$

$$\langle a\_ \ Ni \rangle \Rightarrow a \ \langle a\_ \ Nii \rangle \mid b \ \langle b\_ \ Ni\_ \ N \rangle$$

$$\langle b\_ \ N1\_ \ N2i \rangle \Rightarrow b \ \langle b\_ \ N1\_ \ N2 \rangle$$

$$\langle b\_ \ N\_ \rangle \Rightarrow c \ \langle c\_ \ N \rangle$$

$$\langle c\_ \ Ni \rangle \Rightarrow c \ \langle c\_ \ N \rangle$$

$$\langle c\_ \rangle \Rightarrow \epsilon$$

This grammar, while not as clear as the one presented in chapter 5, does indeed produce the language, and it is still easily seen how to extend it to produce $a^n b^n c^n d^n$. It does so by using the N metarule as a counter. The $N1$ and $N2$ are used to circumvent consistent substitution for the b rules. In fact, this grammar functions very similar to the way a naive program might attempt to parse the language $a^n b^n c^n$. Every time an a is encountered, a new nonterminal is involved, whose name incorporates the counter for the number of a's encountered so far. Upon encountering the first b, two "counters" are introduced, one for saving the result to pass on to the c's, and the other to countdown the number of b's. Once the number of b's has been counted down, the c's are counted down in the same manner.

It is clear that the metagrammar is regular, since it the only nonterminals on the righthand side are in a final position. While it does contain an ϵ rule, it is easy to eliminate if desired.

It is clear that all the hyperrules are regular as well, since again, nonterminals are only in final positions and there is only a single nonterminal per righthand side.

□

Theorem 11. *The grammar, $\mathcal{G}$ given in the prior proof is $LR(0)$.*

Proof. To define a handle, recall from Chapter 4. Number the productions of the grammar $\mathcal{H}$ (i.e. if there are j rules in the grammar, the rules will be labeled $1, \ldots, j$). Let $\alpha = X_1 \ldots X_n \ldots X_t$ be sentential form of the grammar $\mathcal{H}$. Suppose for some (partial) derivation, the leaves of a single node (which is for a production p) on the tree have the form $X_{r+1} \ldots X_n$ where $0 \leq r \leq n \leq t$ and $1 \leq p \leq j$, then (n, p) is a handle of the sentential form α. Let $\triangle$ denote the position of the handle in a sentential form.

A grammar is $LR(k)$ if the following conditions hold. Let $k \geq 0$ and let "$\dashv$" be new terminal symbol not in $I \bigcup T$ where I is the set of nonterminals and T is alphabet the language. A k-sentential form is a sentential form followed by k "$\dashv$" characters. Let $\alpha = X_1 X_2 \ldots X_n X_{n+1} \ldots X_{n+k} Y_1 \ldots Y_u$ and $\beta = X_1 X_2 \ldots X_n X_{n+1} \ldots X_{n+k} Z_1 \ldots Z_v$ be k-sentential forms in which $u \geq 0$ and $v \geq 0$ and in which none of $X_{n+1}, \ldots, X_{n+k}$, $Y_1, \ldots, Y_u$, $Z_1, \ldots, Z_v$ is a nonterminal. If (n, p) is a handle of α and (m, q) is a handle of β, we require that $m = n$ and $p = q$. In other words, a grammar is $LR(k)$ if and only if any handle is uniquely determined by the string to its left and the k terminal characters to the right.

To show $\mathcal{G}$ is $LR(0)$, we must show that any handle is uniquely determined by the string its left and 0 terminal characters to the right.

Consider a string in the language $a^k b^k c^k$. Now, every production in the grammar has two nodes, but the only production without nonterminals is $\langle c\_ \rangle \Rightarrow \epsilon$. So, before any reductions can be made, the whole string must be consumed, so the first handle which can be reduced will be $a^k b^k c^k \triangle$. Now, the string becomes $a^k b^k c^k \langle c\_ \rangle \triangle$. The only possible reduction is: $a^k b^k c^{k-1} \langle c\_ \; i \rangle \triangle$. But now, it is clear that since each production only has two nodes, that each reduction will continue one character at a time. At the boundary between the b's and c's we will have $a^k b^k \langle c\_ \; i^k \rangle \triangle$. So the only reduction is $a^k b^k \langle b\_ \; i^k \_ \rangle \triangle$. Now the reductions will start consuming the b's.

Since the handle was always the entire string, no terminals were to the right of it, thus the grammar is indeed $LR(0)$. □

An interesting phenomenon seems to be occuring in these examples. What happens when all restrictions that a grammar be finite are broken?

Theorem 12. *Any language has an infinite $LR(0)$ grammar.*

Proof. Let $\mathcal{L}$ be a language. Part 1. Consider a lexicographic ordering on strings in $\mathcal{L}$. Let $\langle S \rangle$ be the start symbol. Now, for each terminal α, if any string starts with α add the production:

$$\langle S \rangle_S \rightarrow \alpha \; \langle \alpha \rangle$$

to our grammar $\mathcal{G}$.

Now, consider each string which starts with α and whose second character is β. Add the following production to the grammar:

$$\langle \alpha \rangle \rightarrow \beta \; \langle \alpha\beta \rangle$$

In likewise manner, given the presence of a string in $\mathcal{L}$ whose lefthand portion is γ and is k characters long, and whose $k+1$ character is ϕ, add the following productions:

$$\langle \gamma \rangle \rightarrow \phi \, \langle \gamma \phi \rangle$$

For each string $\delta \in \mathcal{L}$, add the following production:

$$\langle \delta \rangle \rightarrow \epsilon$$

Each, string δ in $\mathcal{L}$ is accepted by the nonterminal $\langle \delta \rangle$, and the productions which lead to $\langle \delta \rangle$ proceed by creating a matching string in the language for the nonterminals (constructed from the same alphabet as the terminals).

Since this grammar is regular except for the empty productions, it is also $LR(0)$.

Part 2. Now, we have created an infinite grammar, $\mathcal{G}$ which is $LR(0)$, now we must show that $\mathcal{G}$ generates exactly $\mathcal{L}$.

Let a be a string $\mathcal{L}$. Let a_k denote the k'th character of a. Since $a \in \mathcal{L}$, by the construction of $\mathcal{G}$ there is a production in the grammar of the form:

$$\langle S \rangle_S \rightarrow a_1 \langle a_1 \rangle$$

Likewise, there is a production of the form:

$$\langle a_1 \rangle \rightarrow a_2 \langle a_1 a_2 \rangle$$

In fact, for each k there is a production:

$$\langle a_1 a_2 \dots a_{k-1} \rangle \rightarrow a_k \langle a_1 a_2 \dots a_{k-1} a_k \rangle$$

So, a will be accepted by the production, now, for the final production $\langle a_1 \dots a_k \rangle$ there is a production of the form (since $a \in \mathcal{L}$

$$\langle a_1 \dots a_k \rangle \rightarrow \epsilon$$

So our last production will accept a, and so a is in the language generated by $\mathcal{G}$.

We must now show that there are no "extra" strings in the language generated by our grammar. Let b be a string such that $b \notin \mathcal{L}$, where n is the length of b.

Now, if b_1 is identical to the first character in some string in $\mathcal{L}$, there will be a production which starts to accept b:

$$\langle S \rangle_S \rightarrow b_1 \langle b_1 \rangle$$

However, since $b \notin \mathcal{L}$, one of the following must be true. Either b is the first part of some string in $\mathcal{L}$, in other words, for some $a \in \mathcal{L}$, $b = a_1 \dots a_n$. The other possibility is that b shares the first $k-1$ characters with some string in $\mathcal{L}$ and $b_k \neq a_k$, where $0 \leq k \leq n$.

Now, in the first case, b will be generated by productions in $\mathcal{G}$ up to the nonterminal $\langle b \rangle$. However, since there are no strings in $\mathcal{L}$ equal to b, the only nonterminals will be of the form:

$$\langle b \rangle \rightarrow \alpha \langle b \alpha \rangle$$

So there is no rule which will finish the derivation tree for b.

The other case is similar, since there is a first character k which is different we will match up to that k with the nonterminal $\langle b_1 \dots b_{k-1} \rangle$. Since there is no string in $\mathcal{L}$ that starts with the same first $k-1$ characters and has a next character b_k there is no production of the form:

$$\langle b_1 \dots b_{k-1} \rangle \rightarrow b_k \langle b_1 \dots b_{k-1} b_k \rangle$$

Hence b is not accepted by $\mathcal{G}$, so the language generated by $\mathcal{G}$ is equal to $\mathcal{L}$. □

This proof is important because it shows that simply allowing an unstructured infinite grammar increases the expressive power far beyond the limitations of finite grammars (since even a regular grammar is capable of generating any Recursively Enumerable language). In order to make useful infinite grammars, it is desirable to create classes similar to the Chomsky hiearchy. In this respect, Wijngaarden meta-grammar generates a Context Free language which is then used to create an infinite Context Free grammar, and this was shown by Sintzoff to be capable of generating the Type-0 languages (i.e. Recursively Enumerable), but the resulting infinite grammar is not necessarily $LR(k)$ for any k. Deussen came up with several restrictions on a meta grammar and hypernotions (rule templates) to guarantee Decidable infinite grammars (i.e. Context Sensitive languages), and Fisher showed how to restrict the meta grammar and hypernotions to create an infinite $LL(1)$ grammar (Which is also decidable and allows for some standard parsing techniques to be used).

The goal of creating useful classes of infinite grammars leads very naturally to the idea of a two-level grammar. A two-level grammar consists of a metalevel which is used to generate an infinite grammar to describe a language. However, coming

up with useful restrictions on the metalevel has proven to be hard, since nearly any means of generating an infinite grammar leads to the ability to express any Recursively Enumerable language. This has lead to another common approach, restrictions on the metalevel and the rule templates which combine to make a language class which is decidable (i.e. at most context sensitive) and whose strict (infinite) grammar has nice properties (e.g. $LR(1), LL(1), \ldots$).

As an interesting aside, while not strictly more powerful, generating a large finite grammar can be useful in practice, and admits simpler (or at least shorter) descriptions of many languages than would be otherwise possible. Consider what would be possible if you generated a grammar with 100,000 productions. The ability of a context free grammar to express only a finite number of context conditions is more usable. The problem of parsing with such grammars involves problems similar to the finite case since actually generating 100,000 rules and feeding them to a parser generator will probably cause very large coefficients to come into play even on $O(n)$ parsers, if not actually breaking the algorithm to generate the parser or running the system out of memory. [23]

Corollary 1. *Any metasystem which can produce an infinite $LR(k)$ grammar can produce any language.*

It has been shown that any language can be described by an infinite $LR(0)$ grammar. While this is a nice theoretical result, it is a depressing one. Finite $LR(k)$ grammars are not only decidable, but parsable in $O(n)$ time. Since Type-0 grammars are not in general decidable here is another easy fact:

Corollary 2. *In general, an infinite $LR(k)$ grammar is not decidable.*

How does this relate to Knuth's original conception of $LR(k)$ languages? Knuth defined them as languages that are translatable from left to right with only a

finite lookahead. Of course, in the proof, a vacuous trick was used, which resolved all ambiguities only at the end which caused a cascade of reductions backwards through the parsing.

The author is now left with an interesting question. Is it possible to restore any of the nice properties of finite $LR(k)$ grammars to the infinite case? Some of those properties include decidability and time complexity and language complexity. It would be nice if it could be proved that a class of infinite $LR(k)$ grammars is $O(n)$ and accepts all of the context-sensitive languages. This result would be of major proportion because it would also imply that all of the context free languages can be parsed in $O(n)$ as well (the current bound is $O(n^{2.87})$.

Theorem 13. *Any Recursively Enumerable Language can be described by an infinite $LR(0)$ grammar which is Recursively Enumerable as well.*

Proof. If a language $\mathcal{L}$ is recursively enumerable, then it is the range of some partially computable function. By Theorem 9 from section 3.5, there exists a primitive recursive function, f, which maps $\mathbb{N}$ to $\mathcal{L}$.

Since f is computable on $\mathbb{N}$ we can construct another function $g(n)$ which creates productions to accept the n'th string in $\mathcal{L}$ in the manner used in Theorem 11. Since a grammar produced by that algorithm will accept $\mathcal{L}$, it now remains to show that in this case, the grammar is recursively enumerable.

Since all the strings produced by f will be finite (if the n'th string is infinite, $f(n)$ will not terminate, and thus $f(n)$ will not be comptuable, violating the lemma), $g(n)$ will only produce a finite number of grammar rules. So we can construct a third function which produces the n'th grammar rule, by running $g(x)$ on 1 to n (since each string will have at least 1 production), and counting the output of $g(x)$ until it

produces the n'th rule. Since the grammar, $\mathcal{G}$, can be produced by such an algorithm, $\mathcal{G}$ must also be recursively enumerable by Theorem 9. □

5.5 Infinite Languages

Given the assumption that a language is not a finite set of strings (but is on a finite alphabet), it becomes clear that for any length n, there are only finitely many strings in any language shorter than n. In other words, there is no upper bound on the length of the strings; if there were, the language would be finite.

The graph theory explored in earlier proofs will now be explored. Given the strings in a language $\mathcal{L}$, a lexicographic ordering can be provided by ordering the alphabet. A tree can be constructed which represents all the strings in the language. Given an alphabet $\{s_1, \ldots, s_n\}$, on the first level of the tree (from the root), if there is a string α which starts with the character s_k, a branch will be added labelled s_k from the root. Since there are only finitely many characters in the alphabet (n), the root will have at most n branches. Inductively proceeding, any string starting with s_k whose second character is s_{k_1} will result in adding a branch to s_k labelled with s_{k_1}. Now, since at any particular stage, there are only finitely many strings shorter than the depth of the tree (d), at each level the tree must be finite. Since the language is infinite, there must be an infinite number of leaves on this tree, so the tree is infinite, but since each vertex only has finitely many connections, it is locally finite.

König's Lemma states that a locally finite tree is infinite if and only if it has an infinite path. Since the tree is infinite, this must mean there is an infinite branch. So if $\mathcal{L}$ is an infinite language, then a string of infinite length can be associated with it.

Theorem 14. *König's Lemma A locally finite tree is infinite iff it has an infinite path.*

Proof. Case 1. Assume the tree is infinite. The degree of each vertex is finite (i.e. the tree is locally finite), so consider the root. If a vertex shares an edge with the root, it is connected to either a finite number of vertices or an infinite number. If all the vertices are connected to a finite number of vertices, the tree itself must be finite since the root has finite degree. So at least one vertex is connected to an infinite number of vertices. By induction, we may continue this process of selecting nodes on an infinite branch of the tree. A selection which leads to a branch which terminates contradicts our previous choices of the infinite branch, and so we must be able to continue this path without bound, therefore, we have found an infinite path through the tree.

Case 2. Assume the tree has an infinite path. A tree is infinite if it has an infinite number of levels. Each vertex of the path is on a level of the tree which must be finite in size (since the tree is locally finite), but the path itself is infinite, so there must be an infinite number of levels. □

Definition 13. *Given a language, $\mathcal{L}$, a string α of length n is an initial substring of $\mathcal{L}$ if there exists a string $\beta \in \mathcal{L}$ such that the first n characters of β are equal to the characters of α.*

Theorem 15. *If $\mathcal{L}$ is an infinite language on a finite alphabet, there exists a string of infinite length, α, with the property that every initial finite substring of α is also an initial substring of a string in $\mathcal{L}$.*

Proof. The initial substrings of an infinite language $\mathcal{L}$ can be represented as a tree. We may construct the tree such that particular initial substring α of length n is represented by recursively concatenating the labels of nodes, where the node on the

k'th level represents the k'th character of the initial substring. The root of the tree is empty string. Since our language has a finite alphabet of size n, at most there are n nodes on the first level, and n^2 on the second level. In general, if the language is Σ^*, the tree has at most n^k nodes on the k'th level of the tree.

Since the alphabet is finite, the tree itself is locally finite (a node has one parent and at most n children, so its maximum degree is $n + 1$).

Since the language is infinite, the tree must be infinite, since a finite number of nodes would require our language to be finite as well. Since we have an locally finite infinite tree, König's lemma applies, and we may conclude that there exists an infinite path within this tree. Since all paths through the tree are associated with initial substrings of $\mathcal{L}$, we can associate a string with this infinite path, which has infinite length. Since there is no maximal length on the strings in $\mathcal{L}$, at any finite point, the string is a substring of a sentence in $\mathcal{L}$. □

In the case of finite strings, a Turing Machine $\mathcal{M}$ is said to accept a string α if it halts when given α as the input on the tape. However, the traditional definition of a Turing Machine assumes the tape initially has only finitely many non-blank symbols on it. In order to consider how a Turing Macine $\mathcal{M}$ behaves in the presence of infinite inputs, a way to present infinite inputs to $\mathcal{M}$ must be found.

The existence of a universal Turing Machine is a well known result in computability theory. Such a machine starts with an encoding of a Turing Machine and its input on the tape, and simulates the execution of the machine with that input. A string is computable if there is a Turing Machine which will produce that string as the output of its execution. If the Turing Machine never halts, but continues to produce characters without backtracking, it will be said to produce an infinite string.

Now, given that there are only countable computable functions and an uncountable number of finite length strings, it should be clear that not all infinite strings will be computable either. However, if an infinite string is computable, a Universal Turing Machine can be used to feed its output to another Turing Machine. Since it is possible for a Turing Machine to execute multiple Turing Machines through the process of dovetailing, a Universal Turing Machine can be designed which starts with an encoding of two Turing Machines and the input of the second Turing Machine. The first machine will be called $\mathcal{C}$ since it will consume the output of the other machine $\mathcal{G}$, the generator, which is executed with input string α, also encoded on the tape.

Now, when $\mathcal{G}$ has advanced, the universal machine begins the execution of $\mathcal{C}$, however, when $\mathcal{C}$ needs to advance past $\mathcal{G}$'s current position, or after a finite number of steps, the machine switches to $\mathcal{G}$ again. If $\mathcal{C}$ ever halts in a predetermined rejection state, it has rejected its input. If $\mathcal{G}$ ever halts, and produces a finite string, and $\mathcal{C}$ halts in a predetermined acceptance state, $\mathcal{C}$ has accepted the output of $\mathcal{G}$. If $\mathcal{G}$ never halts and $\mathcal{C}$ never halts, then it cannot be determined whether $\mathcal{C}$ has accepted $\mathcal{G}$.

Theorem 16. *If $\mathcal{L}$ is an infinite language on a finite alphabet, and the infinite string from Theorem 15 is computable, then no Turing Machine which accepts $\mathcal{L}$ will reject the infinite string.*

Proof. Given the Universal Turing Machine previously described, and a machine $\mathcal{G}$ which can compute the infinite string, α, from Theorem 15, we may feed α to Turing Machine $\mathcal{C}$, which accepts $\mathcal{L}$. Since at all finite points of execution α is an initial substring of $\mathcal{L}$, $\mathcal{C}$ will never reject α. Since $\mathcal{G}$ will never halt, $\mathcal{C}$ will never be able to halt, since there is no upper bound on the length of strings in $\mathcal{L}$ (otherwise, $\mathcal{L}$ would be a finite language). □

How can this be? Consider a typical programming language. While lengths on variable names may be restricted to some finite maximum, the grammars rarely express such a constraint. Since there are a countably infinite number of variable names of finite length, a programmer may proceed to declare an infinite number of variables, and the grammar will thus accept the program, even though it never ends. Another point of contention may be that, in definition 1, it was stated that a language consists of only finite strings. It was not claimed that this infinite string is a member of a language, only that an algorithm cannot reject it. This is a strong motivation that the original definition, which rejected infinite strings, is a good one for computability theory.

An illustrative language to consider is the set of all decimal expansions of rational numbers. An irrational number would be an infinite string in such a language, which at any finite stage will not be rejected, but since the string never ends it won't be accepted either.

If the properties of context free grammars are recalled, this result may seem a little less surprising. The $uvwxy$ theorem about original sentences shows as strings in context free languages get longer, they become unoriginal and repetitive. In the case of the infinite string, it is almost assuredly very boring. Context Sensitive languages do not have a similar theorem, but a context sensitive grammar could enforce the rule that variable names would be unique in the previous example, and so even Type-1 languages show a tendency to become boring as they get longer (it is true in literature as well, where authors continue series of books long past the point where they are interesting, and degrade into soap opera like repetitions of cheap plot devices).

However, since the string will neither be accepted or rejected, does this seem to violate the fact that a Context Free Grammar is recursive? Recall that a set is

recursive if an algorithm can always reject or accept a string as a member of the set. However, those were all finite strings. Since at each finite stage, this infinite string is along a fruitful path however, so proceeding ahead the parser will never find evidence to reject the string, but it will never get to the end of the string either, so it will never reject it.

Another important point is that such a grammar will require an infinite amount of memory to process all strings in the language. Turing Machines also assume an unbounded amount of memory as well, and in fact, a finite state machine with no external memory is one of the few ideas in computability theory that does not assume an infinite memory. However, all real world computers do not have an infinite amount of memory, and so a parser on a real world computer will always be unable to accept the entire language. Or will it?

Can a computer be built with a finite memory which would accept any string from a given language? Assuming a way existed to feed the computer an infinite string, there is such a class of languages, the regular languages (which includes Σ^*) are equivalent to finite state machines without external memory (such as a tape). So a computer with finite memory can accept all regular languages, assuming it does not have to store the string itself.

CHAPTER 6
FUTURE WORK AND COMMENTS

Languages are all about meaning. They exist to allow humans to communicate. Even the most arcane proof is an attempt to communicate to another human a true fact, and the reasoning to show that fact is true. To quote two famous computer scientists, "Programs must be written for people to read, and only incidentally for machines to execute."[1]. This is true of mathematical notation as well. It is the author's opinion that the best computer programs and the better languages for writing them have a strong resemblance to executable mathematics. Others share a similar belief about the power of formal methods, and in particular their usability: "Formal methods will never have a significant impact until they can be used by people that don't understand them"[30].

6.1 Syntax And Semantics

All current mathematical models of semantics are in the end a means of describing formal systems. It has been shown in this thesis that a suitably powerful means of specifying syntax can be as powerful as any other method, and that many suitably powerful methods exist. Whether this is appropriate is perhaps a question of philosophy more than anything.

In specifying a formal system, it is as much about modularity that semantics is distinguished from syntax. This is a natural way to do it, and the presence of a "meta" level in such systems is very common. Quibbling about which level should be used to specify a fact is less important than the overall effectiveness of the system.

These systems are as much about usability as anything. Bringing the power of formal methods to a larger audience who need not be aware of the deep mathematical reasoning underlying such systems. People just want to use them.

This is not the only perspective on semantics. Denotational semantics in particular considers syntax to be incapable of properly specifiying meaning. Denotational semantics translates programs into mathematical constructs. Yet, mathematical constructs are written down in a precise, formal notation, which itself is no more powerful than the most powerful methods of expressing syntax. Given these insights, one is brought to the question of why mathematical notation is considered more fundamental. However, Denotational semantics has its uses. Others may be more aligned with the goals of this thesis.

Operational semantics is perhaps the most popular semantic formalism in use today. It is lightweight and fairly easy to read and write. Interestingly enough, its use of metavariables is similar to their use in Wijngaarden grammars.

6.2 Applications And Uses

It is also a worthy exercise to consider how these theoretical concepts have been applied. The programming language Algol 60 introduced the Backus-Naur Form to the world, and the notation was used to specify the context free portions of the language. It is easy to tell programming languages designed before this, as it is often impossible or difficult to write a context free grammar for these languages, but nearly every language since has a nice grammar.

This marked the beginning of the Compiler-Compiler dream in Computer Science as well. The goal of a compiler-compiler is that a complete formal specification for a language may be used to produce a compiler without any user intervention.

After eight years of work and revision, the Algol 68 report was released, which completely specified the language's syntax and semantics with a Wijngaarden grammar. This, like the earlier Algol 60 report, marked the introduction of the notation as well as its use. It should be noted that several competing methods to the BNF had emerged during this period, but none were significantly better or related so well to the Chomsky model of language introduced in the late 50's, and so both IBM's Vienna Definition Model and the CODASYL notation (used to specify COBOL) are now only footnotes in the literature.

Knuth introduced Attribute Grammars, which eliminate some decidability problems with Wijngaarden grammars, and they, along with Affix Grammars are used in several linguistic systems and compiler construction toolkits.

The study of Wijngaarden grammars seemed to dead-end in the mid-80's with [12] as the last paper published that could be found. Linguists still use them some, along with several other methods, but they do not always care about decidability or computational uses of their grammars. However, this study has confirmed what has been reported anecdotally and in the occasional paper. The original progenitors of Wijngaarden grammars seem to have all headed off to theorem provers such as Prolog and ML. Unification, a major part of any computational study of formal logic, is fundamentally similar to the consistent substitution rule of Wijngaarden grammars.

6.3 Future Work

The next logical step is implementing a "parser" using some of the references in this thesis. Such a parser would allow the specification of many example grammars and exploration of the problems involved in their use along with their

efficiency. It would be especially interesting to adopt the techniques used in the decidable Wijngaarden grammars to an Operational Semantics coupled with a Context Free Grammar.

In the quest to design programming languages, many of these topics lead to Type Systems, garbage collection, proof-carrying code, and other "lightweight" theorem provers. A type system is a formal system which guarantees that a properly typed program will never "go wrong".

Areas of mathematics which are commonly used in these areas include Category Theory, Abstract Algebra, and Mathematical Logic. Going deeper in these fields and their connections to formal languages is likely to yield a deeper understanding of the issues involved and how to solve the problems in current theories.

REFERENCES

[1] Abelson, Harold; Sussman, Gerald, *Structure and Interpretation of Computer Programs,* (Cambridge, MA, USA: MIT Press, 1996).

[2] Bryant, Barrett, "Two Level Grammar: Data Flow English for Functional and Logic Programming," *Proceedings of the 1988 ACM Sixteenth Annual Conference on Computer Science,* (New York, NY, USA: ACM Press, 1988): 469–474.

[3] Cheng, Edward Y.C.; Kaminski, Michael, "Context-Free Languages Over Infinite Alphabets," *Acta Informatica,* **35**(3), (Heidelberg: Springer-Verlag, 4 March 1997): 245–267.

[4] Christiansen, Henning, "A Survey of Adaptable Grammars," *Sigplan Notices,* **25**(1), (New York, NY, USA: ACM Press, November 1990): 35–44.

[5] Cleaveland, J. Craig; Uzgalis, Robert C., *Grammars for Programming Languages,* (New York, NY, USA: Elsevier North-Holland Inc., 1977).

[6] Cohen, Jacques, "A View on the Origins and Development of Prolog," *Communications of the ACM,* **31**(1), (New York, NY, USA: ACM Press, January 1988): 26–36. This paper is available online at: `http://portal.acm.org/citation.cfm?id=35045`.

[7] Crowe, David, "Generating Parsers for Affix Grammars," *Communications of the ACM,* **15**(8), (New York, NY, USA: ACM Press, August 1972): 728–734.

[8] Davis; Sigal; Weyuker, *Computability, Complexity, and Languages*, 2nd ed. (Academic Press, 1994).

[9] Dembinski, Piotr; Maluszynski, Jan, "Attribute Grammars and Two-Level Grammars: A Unifying Approach," *Mathematical Foundations of Computer Science 1978: Proceedings, 7th Symposium, Zakopane, Poland: Lecture Notes in Computer Science,* **64**, (Berlin: Springer-Verlag, September 1978): 143–154.

[10] Deussen, P., "A Decidability Criterion for van Wijngaarden Grammars," *Acta Informatica,* **5**, (Springer-Verlag Heidelberg, 1975): 353–375.

[11] Edupuganty, B.; Bryant, B.R., "Two-level Grammar as a Functional Programming Language," *Computer Journal,* **32**(1), (February 1989): 36–44.

[12] Fisher, Anthony J., "Practical LL(1)-based Parsing of van Wijngaarden Grammars," *Acta Informatica,* **21**, (Springer-Verlag Heidelberg, 1985): 559–584.

[13] Fisher, Anthony J., "Practical Parsing of Generalised Phrase Structure Grammars," *Technical Report University of York,* **YCS-95**, (University of York, 1987).

[14] Fisher, Anthony J., "A 'Yo-Yo' Parsing Algorithm for a Large Class of van Wijngaarden Grammars," *Acta Informatica,* **29**, (Springer-Verlag Heidelberg, 1992): 461–481. This article is available online at: `http://www-users.cs.york.ac.uk/~fisher/pubs/yoyovwg.ps.Z`.

[15] Ford, Bryan, "Packrat Parsing: Simple, Powerful, Lazy, Linear Time," *Proceedings of the 2002 International Conference on Functional Programming,* (October 2002). This paper is available online at: `http://citeseer.nj.nec.com/534158.html`.

[16] Greibach, Sheila A., "Some Restrictions on W-Grammars," *Annual ACM Symposium on Theory of Computing: Proceedings of the Sixth Annual ACM Symposium on Theory of Computing,* (New York, NY, USA: ACM Press, 1974): 256–265.

[17] Grune, Dick: Jacobs, Ceriel, *Parsing Techniques: A Practical Guide,* (Chichester, West Sussex, England: Ellis Horwood Limited, 1990).

[18] Knuth, Donald E., "On the Translation of Languages from Left to Right," *Information and Control,* **8** (1965): 607–639.

[19] Knuth, Donald E., "Semantics of Context-Free Languages," *Mathematical Systems Theory,* **2**(2), (June, 1968): 127–145.

[20] Knuth, Donald E., "Semantics of Context-Free Languages: Correction," *Mathematical Systems Theory,* **5**(1), (March, 1971): 95–96.

[21] Knuth, Donald E., "The Genesis of Attribute Grammars," *Attribute Grammars and their Applications,* **5**(1), (New York, NY, USA: Springer-Verlag, 1990): 1–12.

[22] Koster, C.H.A., "Affix Grammars," *Algol 68 Implementation,* (Amsterdan, Holland: North-Holland Publishing Company, 1971): 95–109. Citeseer info: `http://citeseer.nj.nec.com/context/99903/0`.

[23] Koster, C.H.A., "Affix Grammars for Programming Languages," *Attribute Grammars, Applications and Systems,* **545**, (Springer-Verlag Heidelberg, 1991): 358–373. This paper is available online at: `http://citeseer.nj.nec.com/13440.html`.

[24] Koster, C.H.A., "Affix Grammars for Natural Languages," *Attribute Grammars, Applications and Systems,* **545**, (Springer-Verlag Heidelberg, 1991): 469–484. This paper is available online at: `http://citeseer.nj.nec.com/koster91affix.html`.

[25] Koster, C.H.A., "On the Borderline Between Grammars and Programs," *Lecture Notes in Computer Science,* **528**, (Passau: Springer-Verlag, 1991): 219–230. This paper is available online at: `http://www.cs.kun.nl/~kees/home/papers/plilp.ps.gz`.

[26] Koster, C.H.A., "The Family of Affix Grammars," *Proceedings of the First Workshop on AGFL,* (Nijmegen: Nijmegen University, 1995): 358–373. This paper is available online at: `http://citeseer.nj.nec.com/koster95family.html`.

[27] Meeks, Brian, "The Static Semantics File," *Sigplan Notices,* **25**(4), (New York, NY, USA: ACM Press, April 1990): 33–42.

[28] Meersman, R.; Rozenberg, G., "Two-Level Meta-Controlled Substitution Grammars," *Acta Informatica,* **10**, (Springer-Verlag Heidelberg, 1978): 323–339.

[29] Pereira, Fernando C. N.; Shieber, Stuart M., "The Semantics of Grammar Formalisms Seen as Computer Languages," *COLING-84,* (1984): 123–129. This paper is available online at: `http://citeseer.nj.nec.com/pereira84semantics.html`.

[30] Pierce, Benjamin, *Types and Programming Languages,* (Cambridge, MA, USA: MIT Press, 2002).

[31] Russman, Arnd, "Dynamic $LL(k)$ Parsing," *Acta Informatica,* **34**(4), (Heidelberg: Springer-Verlag, 5 January 1996): 267–289. This paper is available online at: `http://www.springerlink.com/app/home/contribution.asp?wasp=bn42d3kprm5jpk91ekdw&referrer=parent&backto=issue,3,4;journal,65,84;linkingpublicationresults,id:100460,1`.

[32] Shutt, John N., "Recursive Adaptable Grammars," (Worcester, MA, USA: Worcester Polytechnic Institute, 10 Aug. 1993).

[33] Sintzoff, M., "Existence of A van Wijngaarden Syntax for Every Recursively Enumerable Set," *Annales do la Societe Scientifique de Bruxelles,* **81**(II), (1967): 115–118.

[34] Slonneger, Kenneth, *Formal Syntax and Semantics of Programming Languages,* (Reading, MA, USA: Addison-Wesley, 1995).

[35] Wegner, Lutz Micheal, "On Parsing Two-level Grammars," *Acta Informatica,* **14**, (Springer-Verlag Heidelberg, 1980): 175–193.

[36] Wijngaarden, Aad van, "Orthogonal Design and Description of a Formal Language," *Technical Report MR-76, Mathematisch Centrum,* (Amsterdam, 1965).

[37] Woods, William A., "Context-Sensitive Parsing," *Communications of the ACM,* **13**(7), (New York, NY, USA: ACM Press, July 1970): 437–445. This paper is available online at: `http://portal.acm.org/citation.cfm?id=362695`.

VITA

Matthew Estes was born August 9th, 1979, in Morristown, TN. He graduated from Morristown-Hamblen High School West in 1997 with First Honors, and was a National Merit Finalist. He received a Bachelor of Science in Electrical Engineering with an emphasis in Digital Design and a minor in Mathematics in 2003. His senior design project was the complete design and implementation of a RISC microprocessor. In the fall of that year he was admitted into the Master of Science program in Mathematics at Tennessee Technological University. He is a member of the Kappa Mu Epsilon Math Honor Society and the Alpha Lambda Delta Honor Society.

During his time in school he was an avid musician playing clarinet in Tennessee Tech's concert band and marching band. In addition, he plays harmonica and pretends to play guitar and saxophone, and has sung in the Wesley Foundation Choir, the Wesley Singers. He was an officer and active participant in the Wesley Foundation Methodist Campus Ministry and was involved in several plays performed there. He has taught Ballroom dancing, and is an experienced horseman. He loves travelling and has visited 36 states and 6 countries, and collects Hard Rock Cafe t-shirts. While attending both high school and college he ran a software company, designed webpages, and founded a newspaper. He is currently designing a programming language and preparing to start his second company, a venture to write commercial software and do engineering research and design in related areas.