Notes for CS 321

Paul Cull

Department of Computer Science
Oregon State University

January 7, 2016

Chapter 2

Grammars

2.1 Introduction

Grammars are the great positive victory for theoretical computer science. Grammars can be used to describe languages, and to efficiently translate from high level languages to machine languages. In this chapter, we will look at a few simple grammars, and show that different sorts of grammars generate different sorts of languages. The main result is a classification of grammars and languages. This classification is called the Chomsky Hierarchy. We will only discuss enough to establish this classification.

2.2 What Is A Grammar?

Many years ago, Mrs. Harris or Sister Mary told you that every sentence has a subject and predicate. She then went on to tell you that a predicate always has a verb and may have an object. To make things even more confusing objects came in two varieties, direct and indirect. Further, verbs were not simply single words since a sentence could have a main verb and an auxiliary (or helping) verb. This was your first introduction to grammar. And I'm sure that most of you shrugged and said "So what?" Your teacher tried to explain to you that by using grammatical categories one could determine if a sentence was grammatically correct. Of course, you shrugged again. You knew an easier way to determine if sentences were grammatically correct. You simply listened to them and if they sounded "right" they were correct. But if you introspect a little about how you knew a sentence was "right", you would realize that you had been learning English almost from the day you were born and that you had already constructed an English grammar which told you which sentences were "right" or "wrong". But this grammar is internal and you have no obvious way to verbalize your internal constructs and rules. (It is interesting to speculate that no two internal grammars are exactly the same. But the circumstances are rare in which two native speakers of English will disagree on the grammaticallity of a

sentence.) Studying grammar in school was, then, a way of learning-to describe processes that theretofore had taken place subconciously.

At some point you probably encountered sentences which sounded right but on applying the rules of grammar you found to be grammatically incorrect. (Perhaps your internal grammar was not quite correct.) On other occasions you may have found sentences which did not sound quite right but an application of the rules of grammar showed you that they were correct.

In your first programming class, your instructor may have told you that an assignment statement consists of the name of a variable, followed by the replacement symbol (perhaps :=), followed by an arithmetic expression. Or she may have said that an assignment statement consists of a replacement symbol separating a right hand side from a left hand side, and that the left hand side must be a variable name and the right hand side must be an arithmetic expression. Here you were being introduced to the grammar of a programming language. Notice the analogy to your English grammar. A sentence (an assignment statement) has a subject (left hand side) and a predicate (replacement symbol) and object (right hand side). In both cases, you need more rules to determine if the sentence or assignment statement is properly formed. What are allowed subjects? What are allowed variable names? What is the form of a properly formed object? What is the form of a properly formed arithmetic expression?

When we study the grammar of a natural (as opposed to a programming) language, we never have a complete specification of the grammar. For example, at many points in the grammar you are allowed to insert nouns, which you are told are the names of people, places, or things. This definition of a noun is ineffective. How could you use this definition to decide if "waxum" was a noun? One way to find out is to look up "waxum" in the dictionary. If it is in the dictionary and the dictionary says it is a noun, then it is. But what happens if it is not in the dictionary you have consulted? You could consult a larger dictionary or several dictionaries to see if "waxum" is a noun. If you still don't find "waxum" can you decide that it is not a noun? Hardly, because there are several reasons for a word not appearing in a dictionary. A word may not appear because it was considered impolite when the dictionary was written. (Try looking up "f—" or "c —" in a dictionary.) The word may be a borrowing from another language which has gained current use but does not yet appear in a dictionary. The word may be newly coined. This third possibility is particularly important in scientific and technical fields, in which new devices or processes are given new names. Some of these words may never appear in a dictionary since the usefulness and use of the word may disappear before the word is ever entered in a dictionary. Yet another difficulty is that if a word is formed from another word by the addition of a usual prefix or suffix, then the word with the prefix or suffix may not appear even though the word without the prefix or suffix does appear. This digression is supposed to point out that the grammar of a natural language is always incomplete. The final test for grammaticality in a natural language is: Does someone else understand what you mean?

The same test for a programming language would be: Does the computer understand what my program means? Unfortunately, this is an anthropomor-

phic question which must be operationalized for a mechanical device. What can we mean by a computer "understanding"? A reasonable definition is that a computer understands a program which computes a function, if the computer when given the program computes the same function. The difficulty with this definition is that there is an infinity (or at least a very large number) of functions which can be expressed in a programming language. How can we check that the computer handles each one correctly? Instead of attempting this infinite task, we instead completely specify the *syntax*, the grammar of the programming language, and the *semantics*, the meaning of each of the constructs in the programming language. If we can show that the computer handles both the syntax and semantics correctly, then we can conclude that the computer will correctly execute each program written in the programming language.

At this point we will not delve into the specification of the semantics of a programming language, other than to state that this is the mapping from statements in the programming language to statements in the machine's language. And we make the usual "leap of faith" and assume that statements in the machine's language operate as specified in the machine's manual. Then we will be satisfied that the computer will correctly execute programs in our programming language if we can specify a translation between every program statement and an equivalent set of machine language instructions.

This translation process is the domain of those programs we call compilers, assemblers, and interpreters. Operationalizing computer "understanding" of a programming language means writing such a translating program. To write such a program requires a precise specification of the syntax and semantics of the language. In the following we will concentrate on the problem of exactly specifying the correct form, the syntax, so that it can be automatically checked by a computer and ignore the rest of the problem of translation, the semantics.

2.3 Formal Grammars

The purpose of a formal grammar is to mathematically specify the grammatical rules which determine the correct sentences of a language. A formal grammar consists of two alphabets, a special start symbol, and a set of productions. The two alphabets are both finite sets of symbols. We call one alphabet the terminal alphabet and denote it by Σ. The terminal alphabet contains the symbols that actually occur in the language. These terminal symbols correspond to the letters or words of a natural language.

We call the other alphabet the nonterminal alphabet and denote it by V. The nonterminal symbols correspond to grammatical categories like "noun" or "verb" in a natural language. We require that these two alphabets have no symbols in common, that is $V \cap \Sigma = \emptyset$, where $\emptyset$ is the null (empty) set. The special start symbol usually denoted by S is an element of V.

2.3.1 Concatenation

Before explaining the productions, we will first explain the idea of a string and the operation of *concatenation*. A string over an alphabet A is a finite sequence of symbols from A. For example, if $A = \{a, b\}$ then *aabaabbaba* is a string over A. Similarly *baabaabb* is also a string over A. But *baacaab* is not a string over A since it contains a symbol c which is not in A. The sequence *bababa...* is not a string even though it consists solely of elements from A, because it is an infinite sequence, whereas, a string must be a finite sequence. Given two strings, x and y, we may concatenate them to form the string xy by writing down the string x and then immediately following it by the string y. Thus if $x = abaab$ and $y = babaa$ then $xy = abaabbabaa$. Notice that we represent concatenation by writing down the name of a string and following it by the name of another string. We do not use a symbol to represent the operation of concatenation. This follows the usual practice in algebra where we represent the product of two numbers by writing down the name of one number followed by the name of a second number. Of course, in a programming language we would usually use a symbol, like $*$, to represent the operation of multiplication. Similarly, if we had a programming language that had concatenation as an operation, we would use a symbol, like $|$, to represent concatenation. Thus in a programming language we might use $x|y$ to represent the concatenation of of the two strings x and y.

We are now in a position to define A^* the set of all strings over the alphabet A. A^* is the smallest set which contains the null string, Λ, and all the elements of A, and is closed under the operation of concatenation. This is an inductive (recursive) definition of A^*.
BASE: $\Lambda \in A^*$ and $A \subseteq A^*$
INDUCTIVE STEP: if $X \in A^*$ and $Y \in A^*$ then $XY \in A^*$.
Anything in A^* can be formed from the elements in the BASE by using the INDUCTIVE STEP a finite number of times.

At this point we should explain the null string, Λ. It is the string with no elements. If by the length of a string we mean the number of positions in the string then the null string is the string of length 0. The null string is the identity element for the operation of concatenation, that is $X\Lambda = \Lambda X = X$ for any string X.

Another example of an inductive definition is the definition of $\ell(X)$, the length of string X.
BASE: $\ell(\Lambda) = 0$, and if $a \in A$ then $\ell(a) = 1$.
INDUCTIVE STEP: $\ell(XY) = \ell(X) + \ell(Y)$.
A^* with the operation of concatenation forms a semigroup with identity (a monoid). That is,

1. A^* is closed under concatenation if $X \in A^*$, $Y \in A^*$ then $XY \in A^*$.
2. Concatenation is associative, for each X and Y and Z in A^*, $X(YZ) = (XY)Z$.
3. There is an identity Λ, so that for each $X \in A^*$, $\Lambda X = X\Lambda = X$.

But A^* with concatenation is not a group, since except for Λ no strings have inverses. That is, if $X \in A^*$, $X \neq \Lambda$, then for all $Y \in A^*$, $XY \neq \Lambda$.

Furthermore, concatenation is not a commutative operation. That is $XY \neq YX$ except in special cases. (See exercise 2).

2.3.2 Productions

Finally we can return to defining a production. A production is a rule of the form $W \rightarrow Z$ which tells us that the string W may be replaced anywhere it appears by the string Z. Thus if we have the string $\alpha W \beta$ we may replace it by $\alpha Z \beta$.

The set of productions P of some grammar G define a relation on the set $(V \cup \Sigma)^*$. That is two strings α and β from this set are related if there is a production in P which allows us to replace α by β.

For our purposes this relation is too weak, since we are interested in those strings which can be obtained from the start symbol using a sequence of productions. So we define strings α and β to be related if there is a sequence of productions in P that allow us to eventually replace α by β. This new relation we represent as $\Rightarrow$. So $\alpha \Rightarrow \beta$ if there is a sequence of productions in P which eventually allows us to replace α by β. Notice that this new relation is the reflexive, transitive closure of the previously defined relation which says that there is a simple production in P which allows us to replace α by β. Each different grammar defines a different relation $\Rightarrow$. It may occasionally be necessary to specify which grammar we are using by attaching another symbol to $\Rightarrow$.

For example $\alpha \stackrel{G}{\Rightarrow} \beta$ means that β can be derived from α using grammar G. We read$\Rightarrow$ as "derives".

Although derives is both reflexive and transitive, it is not in general an equivalence relation, since for many grammars $\alpha \stackrel{G}{\Rightarrow} \beta$ does not imply that $\beta \stackrel{G}{\Rightarrow} \alpha$. That is derives is not an equivalence relation because it is not symmetric. For some grammars, derives is symmetric and it is an equivalence relation for those grammars.

2.3.3 Generating Languages

Reviewing, we have that a grammar G is (Σ, V, P, S) where Σ is the terminal alphabet, V is the nonterminal alphabet, P is a set of pairs $\alpha \rightarrow \beta$ where α and β are both strings over $(V \cup \Sigma)$, and S, the start symbol, is a member of V.

A grammar should, of course, specify a language. A language is a set of strings over Σ. That is a language L is a subset of Σ^* the set of all strings over Σ, in symbols, $L \subset \Sigma^*$. The language generated by a grammar G is the set of terminal strings which can be derived from S. In symbols

$$L(G) = \{X \mid X \in \Sigma^*,\ S \stackrel{G}{\Rightarrow} X\}.$$

As a simple example, we have the following grammar G_1

$$\begin{aligned}\Sigma &= \{a, b\} \\ V &= \{S, R\} \\ P &= \{S \rightarrow aR, S \rightarrow b, R \rightarrow aa\} \\ L(G_1) &= \{aaa, b\}\end{aligned}$$

Starting from S we can derive b which is a terminal string so b is in $L(G_1)$. Alternately, we could derive aR and then derive aaa which is a terminal string and is thus in $L(G_1)$. Since these are the only possibilities we have all of $L(G_1)$.

As a grammar for a language more like English, we can consider $L(G_2)$

$$\begin{aligned}E &= \{\underline{RAH},\ \underline{NUTS}\} \\ V &= \{S, R\} \\ P &= \{S \rightarrow \underline{RAH}\ R, S \rightarrow \underline{NUTS}, R \rightarrow \underline{RAH}\ \underline{RAH}\ \} \\ L(G_2) &= \{\underline{RAH}\ \underline{RAH}\ \underline{RAH}\ , \underline{NUTS}\}\end{aligned}$$

Notice that in this example $\underline{RAH}$ is considered as a single symbol G_2 is the same grammar as G_1 except for the symbols used in the terminal alphabets.

2.3.4 Some Simple Results On Grammars and Languages

Let us make a few elementary observations about grammar and languages.

1. Every finite language has a grammar.
   To verify this observation let $L = \{l_1, l_2, \ldots, l_K\}$ be any finite language. We want to construct a grammar G such that $L(G) = L$. Let Σ be the set of symbols which appear in any of the strings in L. Let V consist solely of the start symbol S. Let $P = \{S \rightarrow l_1, S \rightarrow l_2, \ldots, S \rightarrow l_K\}$. Clearly a terminal string can be derived from S if and only if the terminal string is in L, thus $L = L(G)$.

2. Some infinite languages have grammars.
   Consider the infinite language $L = \{a, aa, aaa, aaaa, \ldots\}$.
   L can be generated by the grammar which has a as its single terminal symbol and S as its single nonterminal symbol, and which has $S \rightarrow aS$, and $S \rightarrow a$ as its only two productions.

   Although this is obvious, in order to demonstrate how proofs are done with grammars, we will prove that $L(G) = L$. First we prove that $L \subseteq L(G)$ by showing that each string in L is generated by G.

   INDUCTIVE HYPOTHESIS: for each $k \geq 1$,
   $S \Rightarrow a^k S$

   (By a^k we mean a string of k a's).

   BASE: $S \Rightarrow aS$ since $S \rightarrow aS \ \in P$.

   INDUCTIVE STEP:

$$\text{if } S \Rightarrow a^k S, \text{ then using}$$
$$\text{the production } S \rightarrow aS$$
$$S \Rightarrow a^k aS, \text{ i.e. , } S \Rightarrow a^{k+1} S.$$

Using this result, we prove $S \Rightarrow a^k$ for each $k \geq 2$. Since $S \Rightarrow a^k S$ we use the production $S \rightarrow a$ yielding $S \Rightarrow a^{k+1}$. To finish the proof that $L \subseteq L(G)$ we need to show that $a \in L(G)$, but $S \rightarrow a$ is a production so $S \Rightarrow a$ and $a \in L(G)$.

To complete the proof that $L(G) = L$, we need to show that $L(G) \subseteq L$. But each terminal string derived from S consists solely of a's since a is the only terminal symbol, so $L(G) \subseteq L$. Actually this statement is faulty since Λ, the null string, is also a string whose only terminal symbols are a's. Since $\Lambda \notin L$ we must show that $\Lambda \notin L(G)$. Looking at the productions we see that the right hand side is at least as long as the left hand side so that any string derived from S must have length at least 1, but since Λ has length 0 it cannot be derived from S in the grammar. (Perhaps from this example you can see that proving the obvious takes at least some work.)

3. There are some infinite languages which are not generated by any grammar.
Consider any finite alphabet Σ with at least one symbol. Σ^* is countable, that is the strings in Σ^* can be put in to 1-1 correspondence with the natural numbers. We order the strings by length and then alphabetically. That is all strings of length 0 come first, then all strings of length 1, then all strings of length 2, and so forth. To alphabetically order strings we place an arbitrary order on Σ, say $\Sigma = \{s_1, s_2, s_3, \ldots, s_k\}$ then the first string with r symbols is the string which consists of s_1 repeated r times. Next is the string with s_1 repeated $r-1$ times followed by s_2. More formally for two strings X and Y over Σ we have,

$$\begin{gathered} X < Y \text{ if and only if} \\ \ell(X) < \ell(Y) \\ \text{or } \ell(X) = \ell(Y) \\ \text{and } X \text{ and } Y \text{ are identical on their first } r-1 \text{ characters} \\ \text{and differ on their } r^{th} \text{ character with } X_r < Y_r \\ \text{in the ordering imposed on } \Sigma. \end{gathered}$$

Thus we have that Σ^* is countable. But a language over Σ is defined to be any subset of Σ^*, and there are an uncountable infinity of subsets of a countable infinite set. Now all we have to do is to show that there are only countable many grammars and we may conclude that since there are more languages than grammars, there must be some languages which do not have grammars. Since there are a finite number of nonterminals and a finite number of productions in a grammar, any grammar can be represented as a finite string in some finite alphabet. But we have just

seen that there are only a countable number of finite strings over a finite alphabet, so there are only a countable number of grammars and our proof is complete.

2.4 The Chomsky Hierarchy

2.4.1 Types of Grammars

In the previous section we defined grammars quite generally; we allowed productions of the form $\alpha \to \beta$ where α and β are both strings over $(\Sigma \cup V)$. But we didn't actually use productions of this generality. Our productions all had the form $A \to a\,B$ or $A \to abcd$. That is our productions only had a single nonterminal on the left hand side of the arrow and at most one nonterminal on the right hand side. These productions look simpler than general productions, like $abAAabBb \to AabB$, but it might be possible to replace a general production by several productions of restricted type. On the other hand if general productions are in fact more general we should be able to describe languages which can be generated using general productions, but which can not be generated using productions of restricted types. In fact we might consider different types of productions and define a restricted type of grammar to be a grammar which has productions of that restricted type.

This procedure becomes interesting when we can show that the restricted types of grammars form a *hierarchy* . That is, if we define grammars of several types indexed by the natural numbers, we would have that if a language can be generated by a grammar of type i, then the language can also be generated by a language of type $i + 1$, but there are languages which can be generated by grammars of type $i + 1$ which cannot be generated by grammars of type i. Said another way, a *hierarchy* is a sequence of sets $(S_1, S_2, \ldots)$ indexed by the natural numbers so that if $x \in S_i$ then $x \in S_{i+1}$ but $\exists y \in S_{i+1}$ such that $y \notin S_i$. That is S_i is properly contained in S_{i+1}. Unfortunately, for historical reasons, the hierarchy of grammars is unusual since it is finite, there are only four types of grammars, and they are numbered backwards. That is type 0 grammars are the most general and type 3 are the most restricted. Since every grammar of type 3 is also a grammar of type 0, every language generated by a type 3 grammar is also generated by a type 0 grammar. But there are languages which can be generated by type 0 grammars which cannot be generated by type 3 grammars.

The classification of grammars we will discuss is called the Chomsky hierarchy after Noam Chomsky, the **MIT** mathematical linguist (and anti-war activist) who first described this classification.

- A type 0 grammar (also called *a phrase structure grammar*) is a grammar (Σ, V, P, S) with no restrictions on the productions in P. That is each production in P has the form $\alpha \to \beta$ with a and β strings over $\Sigma \cup V$ and no further restrictions are placed on the form of the productions.

- A type 1 grammar (also called a *context sensitive grammar*) is a grammar with the restriction that if $\alpha \rightarrow \beta$ is a production then $\ell(\alpha) \leq \ell(\beta)$, i.e. the length of the string β is at least as long as the length of the string α.

- A type 2 grammar (also called a *context free grammar*) is a grammar in which the left side of each production is a single nonterminal and the right side is an nonnull string over $\Sigma \cup V$. That is the form of a production is $A \rightarrow \beta$ where $A \in V$ and β is a nonnull string over $\Sigma \cup V$.

- A type 3 grammar (also called a *regular grammar*) is a grammar in which the left hand side of each production is a single nonterminal and the right hand side is either a single terminal or a single terminal followed by a single nonterminal. That is the form of a production is either $A \rightarrow b$ where $A \in V$ and $b \in \Sigma$, or $C \rightarrow h\ D$ where $C \in V$, $D \in V$, and $h \in \Sigma$.

If we let G_i stand for the set of grammars of type i then we have the obvious result:

$$G_3 \ \subset \ G_2 \ \subset \ G_1 \ \subset G_0$$

That is each grammar of type i is also a grammar of type $i-1$ and since there are some grammars of type $i-1$ which are not grammars of type i, G_i is properly contained in G_{i-1}. We obtain a much more interesting result when we consider $L(G_i)$ to be the set of languages which can be generated by grammars of type i. (When we have an interesting result whose proof is nontrivial, we call the result a theorem.) We will call this result the Chomsky hierarchy theorem.

Theorem 1 (Chomsky Hierarchy)**.**

$$L(G_3) \ \subset \ L(G_2) \ \subset \ L(G_1) \ \subset L(G_0)$$

That is, each language which can be generated by a grammar of type i can also be generated by a grammar of type $i-1$, and there are, languages which can be generated by a grammar of type $i-1$ which cannot be generated by any grammar of type i.

Half of the proof of this theorem is trivial. It is obvious that a language which is generated by a grammar of type i can also be generated by a grammar of type $i-1$ since the grammar of type i is also a grammar of type $i-1$. The difficult part of the proof is to display a language which can be generated by a grammar of type $i-1$ and to prove that the language cannot be generated by any grammar of type i.

In the following subsections, we will show that there are languages in $L(G_2)$ which are not in $L(G_3)$, languages in $L(G_1)$ which are not in $L(G_2)$, and languages in $L(G_0)$ which are not in $L(G_1)$.

2.4.2 Regular and Context Free Languages

In the next three sections we will try to show that for $i = 0, 1, 2$ there is a language of type i which is not a language of type $i+1$. We say that a language

is of type i if there is a grammar of type i which generates the language. Thus we will try to show that there is a context free language which is not a regular language, a context sensitive language which is not a context free language, and a phrase structure language which is not a context sensitive language.

To demonstrate that a language is not of a particular type, we will show that if a particular string is in a language of a particular type, then another string must also be in the language. But we will set up the definition of the language in such a way that the extra string which must be in the language if the language is of the particular type, will not be in the language. Then we can conclude that the language we have defined is not of the particular type.

The result that if a language of a particular type contains a particular string then it must also contain another string whose form depends on the particular string, is called a *pumping lemma*. We will not attempt to state the pumping lemmas in their strongest forms, we will only give forms of pumping lemmas which suffice to prove that a language is not of a particular type.

Lemma 1 (Pumping Lemma for Regular Languages)**.** *If a regular language contains a string $\alpha_1\ \alpha_2\ \alpha_3$ whose length is greater than the number of nonterminals in the grammar generating the language then the language also contains all the strings of the form $\alpha_1\ \alpha_2^k\ \alpha_3$ where α_2 is some nonnull string and k is any positive integer.*

Proof. We first observe that any string derived from the starting symbol in a regular language contains at most one nonterminal and if there is a nonterminal then it occurs at the right hand end of the string. Obviously the start symbol by itself is a string of the desired kind. If we take a string with a nonterminal at the right hand end, then applying a regular production either replaces the nonterminal with a terminal giving a string without a nonterminal, or the nonterminal is replaced by a terminal followed by a nonterminal which again gives a string with a single nonterminal on the right hand end. It is also clear that a terminal string of length n requires n steps to derive, because each production adds one terminal to the string. Thus if a regular grammar generates a terminal string of length greater than the number of nonterminals then in the derivation sequence of this string, there will be two strings in which the same nonterminal appears, i.e.

$$S \Rightarrow \alpha_1\ X \Rightarrow \alpha_1\ \alpha_2\ X \Rightarrow \alpha_1\ \alpha_2\ \alpha_3$$

Therefore, from the nonterminal S we can derive any of the strings $\alpha_1\ \alpha_2^k\ \alpha_3$. We must also observe that α_2 is nonnull since each application of a regular production increases the number of terminals by one. □

Next we want to use this lemma to show that $\{a^n b^n\}$ is not a regular language. By the notation $\{a^n b^n\}$ we mean the set of strings which start with some number of a's (at least one) which are followed by the same number of b's. First we will show that $\{a^n b^n\}$ is a context free language by displaying a context free

grammar which generates this language.

$$G = (\{a, b\}, \{S\}, \{S \to ab, S \to aSb\}, S\}$$
$$L(G) = \{a^n b^n\}.$$

Clearly $a^n\, b^n$ can be generated by using the second production $n-1$ times and then applying the first production once. But any terminal string generated by this grammar uses the second production k times and then the first production once, so any generated terminal string has the required form. Thus $L(G) = \{a^n b^n\}$.

Theorem 2. *There is a context free language which is not a regular language. In particular, $\{a^n b^n\}$ is a context free language which is not regular.*

Proof. We have already established that $\{a^n b^n\}$ is context free, all we have to do is to show that $\{a^n b^n\}$ is not regular.

We will assume that this language is generated by regular grammar. The language contains some string $a^m b^m$ which has more symbols than the grammar has nonterminals. But $a^m b^m$ can be written as $\alpha_1\ \alpha_2\ \alpha_3$ and by the pumping lemma $\alpha_1\ \alpha_2^k\ \alpha_3$ is also in the language where α_2 is nonnull. Now α_2 cannot contain both a's and b's because then $\alpha_1\ \alpha_2^k\ \alpha_3$ would have some b's appearing before the last a. If α_2 consists solely of a's or b's then $\alpha_1 \alpha_2^k \alpha_3$ would have either more a's than b's or more b's than a's. Thus we have proved by contradiction that $\{a^n b^n\}$ is not a regular language. That is, we have assumed that $\{a^n b^n\}$ is regular and shown that this assumption leads to the contradiction that if $\{a^n b^n\}$ is regular then it must contain some strings not of the form $a^n b^n$. □

2.4.3 Context Free and Context Sensitive Languages

Next we want to demonstrate that there is a context sensitive language which is not context free. Our plan of attack is the same one we used to show that there is context free language which is not regular. We will prove a pumping lemma for context free languages, display a language which is context sensitive, and prove the language is not context free by using the pumping lemma.

Lemma 2 (Context Free Pumping Lemma)**.** *If a context free language L is infinite then for some long enough string $\alpha_1\ \alpha_2\ \alpha\ \beta_2\ \beta_1$ in L, L must also contain all the strings $\alpha_1\ \alpha_2^k\ \alpha\ \beta_2^k\ \beta_1$ where either α_2 or β_2 or both are nonnull strings.*

Proof. We will prove first that in any context free grammar generating an infinite language there is a nonterminal X such that

$$S \Rightarrow \alpha_1 X \beta_1 \text{ and } X \Rightarrow \alpha_2 X \beta_2 \text{ and } X \Rightarrow \alpha$$

where α_1, α_2, β_2, β_1 and α are terminal strings and at least one of α_2, and β_2 is nonnull. We will use a proof by contradiction; we will assume that no such X exists and show that this implies that the language generated must be finite.

If

$$S \not\Rightarrow \alpha_1 X \beta_1 \text{ or } X \not\Rightarrow \alpha$$

then no derivation of a terminal string from S ever uses the nonterminal X, so X and all productions containing X can be eliminated without affecting the language generated by the grammar.

We can now assume that we have eliminated the useless nonterminals so that for each remaining nonterminal X, $S \Rightarrow \alpha_1 X \beta_1$ and $X \Rightarrow \alpha$, with possibly different α's and β's for different nonterminals.

Next we consider if $X \Rightarrow \alpha_2 X \beta_2$ is true. From the definition of $\Rightarrow$ we always have $X \Rightarrow X$. We are interested in $X \Rightarrow \alpha_2 X \beta_2$ where at least one of α_2 and β_2 are nonnull. First, if $X \Rightarrow g_1 X g_2$ where g_1 or g_2 is nonnull but may contain nonterminals, then $X \Rightarrow \alpha_2 X \beta_2$ since we have eliminated all nonterminals which do not generate a terminal string. Now if $X \not\Rightarrow \alpha_2 X \beta_2$ for some nonterminal X, other than S, then X can be eliminated, and the X's which appear in the right hand side of productions can be replaced by each of the right hand sides of productions which have X as their left hand side. In general each production with an X will be replaced by several productions.

Thus we have argued that if there is no nonterminal X such that $S \Rightarrow \alpha_1 X \beta_1$, $X \Rightarrow \alpha$, $X \Rightarrow \alpha_2 X \beta_2$, then we can eliminate all of the nonterminals except S from the grammar. If $S \not\Rightarrow \alpha_2 S \beta_2$ then the right hand side of each production will consist entirely of terminals. The language generated by this grammar will be finite because it will only contain one terminal string for each of a finite number of productions.

Therefore, if L is infinite, any context free grammar for L must contain at least one nonterminal X such that $S \Rightarrow \alpha_1 X \beta_1$, $X \Rightarrow \alpha$, $X \Rightarrow \alpha_2 X \beta_2$, where α_2 or β_2 is nonnull. These three derivations involving X can be used to generate the infinite set of strings $\alpha_1\ \alpha_2^k\ \alpha\ \beta_2^k\ \beta_1$, which must therefore be in L. □

We will next show that $\{a^n b^n c^n\}$ is a context sensitive language which is not a context-free language. First we will give a context sensitive grammar generating $\{a^n b^n c^n\}$.

$$G = (\{a, b, c\},\ \{S, B, C\},\ P,\ S)$$

$$P = \left\{ \begin{array}{ll} S \to aBC & bB \to bb \\ S \to aSBC & bC \to bc \\ CB \to BC & cC \to cc \\ aB \to ab & \end{array} \right\}$$

To generate $a^k b^k c^k$ using this grammar we use the second production $k-1$ times and the first production once to produce $a^k (BC)^k$ We then use the third production to move the B's across the C's to obtain $a^k B^k C^k$. Then the fourth production is used and the fifth production is used $k-1$ times to obtain $a^k b^k C^k$. Finally, we use the sixth production once and the seventh production $k-1$ times to obtain $a^k b^k c^k$.

We also have to show that only terminal strings of the form $a^k b^k c^k$ are generated by this grammar. The first production must be used once to eliminate

S but thereafter, neither the first nor the second production can be used. Thus we have $k-1$ uses of the second production followed by one use of the first production giving $a^k(BC)^k$. Before the fifth, and sixth, or seventh production are used the fourth production is used exactly once to give $a^k bh$ where h contains $k-1$ B's and k C's. Of course, production three can be used to permute the B's and C's. When production six is used, as it must be to produce the first c, we have $a^k b^j cg$ where g may contain some B's and C's but if there are any B's in g they can never be changed to terminals since to be changed, they must be next to a b but there are no b's in g. Thus if a terminal string is produced it must have the form $a^k b^k c^k$.

Theorem 3. *There is a context sensitive language which is not a context free language. In particular,* $\{a^n b^n c^n\}$ *is a context sensitive language which is not context free.*

Proof. We have just seen that $\{a^n b^n c^n\}$ is context sensitive. We now want to use the pumping lemma to show that $\{a^n b^n c^n\}$, is not context free. If this language were context free, since it is an infinite language, we would have that for some large enough m, $a^m b^m c^m$ can be written as $\alpha_1\ \alpha_2\ \alpha\ \beta_2\ \beta_1$, and that $\alpha_1\ \alpha_2^k\ \alpha\ \beta_2^k\ \beta_1$ would also be in the language with at least one of α_2 and β_2 nonnull. If α_2 or β_2 is nonnull then neither of them can contain two distinct terminals because then the language would have to contain some strings with b before a or c before b. But if α_2 or β_2 consist solely of terminals of a single type then the language will contain strings with unequal numbers of a's, b's, and c's. Thus $\{a^n b^n c^n\}$ is not a context free language. □

2.4.4 Recursive and Recursively Enumerable Languages

To complete the proof of the Chomsky hierarchy theorem, we want to display a language which has a grammar but has no context sensitive grammar. The method of attack we used on the previous two levels of the hierarchy fails at this level since $\{a^n b^n c^n d^n\}$ is context sensitive, and in fact $\{a_1^n\ a_2^n\ \ldots\ a_k^n\}$ also context sensitive for each finite k. Because of these examples it is not clear what sort of pumping lemma we would be able to prove for context sensitive languages. Our approach in this section is to discuss the problem of determining whether or not a string is in a language. We will argue that if a language has a grammar then there is a procedure which tells if a string is in the language generated by the grammar, but this procedure may fail to report that a string is not in the language. We then argue that if a language has a context sensitive grammar, then there is an algorithm to determine whether or not a string is in the language. We then use the unsolvability of the halting problem for Turing machines to argue that there is a language which has a grammar but that language cannot have a context sensitive grammar.

First, we will give a few definitions. An *acceptor* for a language is a procedure which when given as input, a string from the language will **halt** and say **YES**. If the input string is **not** in the language the acceptor may either halt and say

NO or continue computing and **never halt**. A language is called *recursively enumerable* if there is an *acceptor* which accepts exactly this language.

An *algorithm* is a procedure which always **halts**. A *recognizer* for a language is an algorithm which will **halt** and say **YES** if the input string is in the language, and will **halt** and say **NO** if the input string is not in the language. A language is *recursive* if there is a *recognizer* which recognizes exactly this language. Thus a recognizer is an acceptor which is an algorithm.

Since an acceptor is a procedure, by Church's thesis, there is a Turing machine which will compute the same result as the acceptor. There are various ways in which we could have our Turing machine acceptor say **YES**. For example, we could require the Turing machine to halt with nothing but the word YES on its tape. Or we could require the Turing machine to halt in a special state q_f with a blank tape. Or, we could require the Turing machine to halt in q_f with the input string on its tape. It is easy to see that if you have a Turing acceptor of any one of these types, then you can construct a Turing acceptor of the other types so that the language accepted by your constructed Turing acceptor will be the same language accepted by the original Turing acceptor. For our purposes we prefer the third definition, that is a Turing acceptor for a language L is a Turing machine which when started in state q_0 at the right hand end marker of a tape which has some string from Σ^* bounded on right and left by end markers, will halt at the right most symbol of the input string having erased the end markers and be in state q_f if and only if the input string is in L.

We are now ready to state and prove a useful theorem.

Theorem 4. *A language has a grammar if and only if it has an acceptor.*

Proof. First we show that if a language has a grammar, it has an acceptor. If the language has a grammar, construct an acceptor which uses the grammar to generate strings. If a terminal string which matches the input string is ever generated, then your acceptor should halt and say **YES**. If the input string is in the language your acceptor will eventually generate a matching string and will halt and say **YES**. If the input string is not in the language your acceptor will continue generating strings forever and **never halt**.

If a language has a Turing acceptor, we can modify this acceptor so that it makes a copy of the input string in a work area, does its (hopefully) accepting computation on this copy while leaving the original input string alone. Also, when this modified acceptor is going to accept, we further modify it so that it erases the work area and halts in a special state, say q_f.

In the following we will assume that the Turing machine tape has "end markers" , say $\square$ and $\sharp$ so that the Turing machine can tell when it is at the left or right end of the tape.

To complete the proof we have to show how to build a grammar which simulates a given Turing machine. This will show that if a language has an acceptor, it has a grammar. First notice that we can represent the configuration of a Turing machine by a string $\alpha q_i \beta$ where α and β are strings of symbols from the machine's tape alphabet and q_i is a symbol, not in the tape alphabet,

representing the i^{th} state of the machine. The machine is looking at the symbol at the right hand end of α. To simulate a Turing machine, we have to give productions which correspond to the instructions of the machine. If we have a left moving instruction

$$q_i \ a_j \ q\prime \ a\prime \ L$$

where the a's represent tape symbols, then we put into the simulating grammar the production

$$a_j \ q_i \ \rightarrow \ q\prime \ a\prime.$$

So if we have a string $\alpha \, a_j \, q_i \, \beta$ which describes the Turing machine as being in state q_i looking at symbol a_j, we obtain the string $\alpha \, q\prime \, a\prime \ \beta$ which describes the Turing machine as having changed a_j to $a\prime$ and moved one position to the left. If we have a right moving instruction

$$q_i \ a_j \ q\prime \ a\prime \ R$$

we will put a set of productions into the grammar. We will have one production for each tape symbol a_k. These productions have the form

$$a_j \ q_i \ a_k \ \rightarrow \ a\prime \ a_k \ q\prime.$$

We need a set of productions since there are a set of things which can be immediately to the right of the state symbol. Thus if the configuration of the Turing machine is $\alpha \ a_j \ q_i \ a_k \ \beta$ then these productions allow us to obtain $\alpha \ a\prime \ a_k \ q\prime \ \beta$ which is the result of changing the a_j to $a\prime$ and moving one position to the right.

If the Turing machine ever goes off either end of its tape it should encounter blanks, so we introduce the special symbol $\overline{b}$ not in Σ, to represent a blank and add the productions

$$\square \, q_i \ \rightarrow \ \square \, \overline{b} \, q_i, \qquad q_i \, \sharp \ \rightarrow \ q_i \, \overline{b} \, \sharp$$

where we have a pair of productions for each state. Since $\overline{b}$ will not be in Σ we also have to add rules to eliminate $\overline{b}$. We add $\overline{b} q_i \rightarrow q_i$ and $q_i \overline{b} \rightarrow q_i$ again a pair of productions for each state. Finally, we have to release the terminal string, that is, when the Turing machine halts in q_f we want to remove the symbol q_f. So we have the production $q_f \rightarrow \Lambda$ where Λ is the null string. (We are assuming that the acceptor only enters q_f when it halts.) Thus we have shown that the action of a Turing machine accepting a string can be simulated by a grammar.

To complete the description of the grammar, we need some productions to set up the initial configuration of the Turing acceptor The initial production using the start symbol S of the grammar

$$S \ \rightarrow \ \square \ A$$

sets up the end markers $\square$ and a nonterminal A to generate all strings. We use the nonterminal A to generate every string over Σ, so we have a set of productions

$$A \rightarrow a_j A \quad \text{and} \quad A \rightarrow a_j \, q_0 \, \sharp$$

one pair of productions for each symbol a_j in Σ. The second of this pair of productions can only be used once because it does away with A. The second production also puts in the right hand end marker and q_0 the initial state for the Turing machine.

Once we have a string from Σ^* between the end markers, the rest of the grammar is used to simulate the acceptor.

We have now constructed a grammar which generates a string from Σ^* if and only if that string is accepted by the Turing acceptor we are simulating. □

We have just shown that a language is recursively enumerable if and only if the language is generated by some grammar. At this point we should recall that there are recursively enumerable languages which are not recursive languages. The familiar example comes from the halting problem.

The halting language $\{(M,t)\}$ is a set of pairs (M,t) where M is the description of a Turing machine and t is the description of a tape such that M halts when given tape t as input. The halting language is a recursively enumerable set which is not recursive. This language is recursively enumerable since you can use a universal Turing machine to simulate M's behaviour when given t, if M ever halts you can modify your universal Turing machine so that it will halt and say **YES**. But this language is not recursive because if it were you would have an algorithm to solve the halting problem and such an algorithm cannot exist since the halting problem is algorithmically unsolvable.

So our plan of attack to show that phrase structure languages properly contain context sensitive languages, is to show that all context sensitive languages are recursive, and thus we will know that the halting language is a phrase structure language which is not context sensitive.

Theorem 5. *There is a phrase structure language which is not context sensitive. In particular the halting language $\{(M,t)\}$ is a phrase structure language which is not context sensitive.*

Proof. The definition of context sensitive language requires the right hand side of any production to be at least as long as the left hand side of the production. Since there are only a finite number of strings of any particular length, there is an algorithm to determine when a derivation will continue producing the same strings. Thus the strings of a context sensitive language can be generated in order of their lengths. So given an input string, generate all strings of that length and compare them to the input string, if there is a match say **YES**, if there is no match, say **NO**, since any strings generated later will be longer than the input string and thus cannot match it. Thus we have an algorithm to recognize any context sensitive language and since there cannot be an algorithm to recognize the halting language, the halting language is not a context sensitive language. By the previous theorem, since the halting language has an acceptor, it is a phrase structure language and the theorem is proved. □

Corollary 1. *Every context sensitive language is recursive.*

As a result of the last theorem, one might wonder if every recursive language is context sensitive. The answer is *no*. Although we will not prove this statement, we will mention that the context sensitive are in some sense easy to recognize. That is, for any context sensitive language there is a nondeterministic Turing machine which uses only the squares occupied by the input string to determine whether or not the string is in the language. There are recursive languages which can be shown to require many more tape squares to determine if an input string is in the language. In fact no "nicely behaved" (i.e. recursive) function can give an upper bound on the number of tape squares required for recognition of all recursive languages. This fact follows from the noncompatibility of the "Busy Beaver" functions.

Since the context sensitive languages fail to capture the recursive languages, we may wonder if there is some other type of grammar which exactly specifies the recursive languages. Unfortunately there is no such type of grammar, because if we assume that we have an algorithm which determines whether or not a grammar belongs to the particular type then we could use this algorithm to solve the halting problem. But since the halting problem is algorithmically unsolvable, then the assumed algorithm for the class of recursive languages cannot exist. Thus, we have the following theorem:

Theorem 6. *No type of grammar exactly specifies the recursive languages.*

2.5 Non-Determinism and Acceptors

Usually when we deal with algorithms or effective procedures we are dealing with deterministic procedures, that is, after the execution of an instruction there is only one instruction which can be executed next. In a real computer this deterministic restriction is enforced by the program counter. The contents of the program counter is the address of the next instruction. We can change which instruction will come next by changing the contents of the program counter. But since the program counter can only contain one address it specifies a unique next instruction. For Turing machines, the restriction that no two instructions can start with the same state and tape symbol enforces determinism. That is, if the Turing machine is in a particular state looking at a particular symbol there is only one instruction which applies to this situation.

But the situation is strikingly different when we deal with grammars. For a given string there may be several productions which could be applied to that one string. In fact, a single instruction could be applied at several different locations in the string. How do we decide what to do first? We don't! Instead we do everything at once. That is, we determine all the various ways in which various productions can be applied to a given string. Then, instead of picking one of these possibilities, we do them all by generating from the given string a set of strings which contains each string that can be derived in one step from the given string. Our argument is that we can represent a nondeterministic process on strings as a deterministic process on sets of strings. Whereas we

would produce several strings from a given string, we will only produce a single set of strings from a given set of strings.

This same sort of argument can be used to prove the following theorem.

Theorem 7. *Any nondeterministic Turing machine can be simulated by a deterministic Turing machine.*

Proof. The nondeterministic Turing machine may have several instructions for a given state and symbol. But for each of these instructions the deterministic machine can make a copy of the configuration (tape plus state) of the nondeterministic machine and then simulate the use of the instruction. Thus the tape of the deterministic machine will contain all the configurations the nondeterministic machine can reach in one step from the initial configuration and the set of next configurations can be computed by the deterministic machine. This process can be continued so that at the k^{th} stage, the tape of the deterministic machine will contain all the configurations of the nondeterministic machine which can be reached in k steps from the initial configuration. □

Corollary 2. *If a language is accepted by a nondeterministic Turing acceptor then it is accepted by a deterministic Turing acceptor.*

The corollary follows since the deterministic acceptor simulates the nondeterministic acceptor. If the nondeterministic acceptor ever reaches an accepting configuration, the deterministic acceptor can determine that the configuration is accepting and halt and say **YES**. For each type of grammar there is a corresponding type of nondeterministic acceptor. One might wonder if to each nondeterministic acceptor there corresponds a deterministic acceptor of the same type. In some cases the answer is yes, in some cases the answer is no, but in most cases the answer is unknown. For regular grammars the corresponding acceptor is the nondeterministic finite automaton. This acceptor can be viewed as a Turing machine with a one way moving read head. This acceptor starts in the initial state looking at the first symbol of the input string. It then may enter anyone of several states and move to the next input symbol. If the acceptor is in an accepting state when it has read the last input symbol, then the input string is accepted. This nondeterministic acceptor can easily be converted to a deterministic acceptor since instead of remembering a single state the deterministic acceptor remembers the set of states the nondeterministic acceptor can be in. At the end of the input the deterministic acceptor accepts if any one of the states it is remembering is an accepting state, otherwise, the input string is not accepted. Notice though that if the nondeterministic finite automaton has n states then the corresponding deterministic finite automaton may have 2^n states. So there is a cost associated with the change from nondeterministic to deterministic finite automata.

Corresponding to the context free grammar is the nondeterministic pushdown automaton. This acceptor has a one-way read-only input tape and a working tape which is a pushdown stack. That is, the read/write head on the working tape may move in either direction but if it moves to the left it leaves

only blank symbols to its right. It is known that the nondeterministic pushdown automaton is strictly more powerful than the deterministic pushdown automaton.

This claim can be proved using the palindrome language. This language consists of all strings whose second half is the reverse of their first half. This language is accepted by the nondeterministic pushdown automaton which stacks the first half of the string and then compares each symbol of the second half with the corresponding symbol of the first half. The corresponding symbol will be on top of the stack. When the comparison finds a match the top symbol on the stack is popped. Nondeterminism is needed here to guess where the middle of the string is and when to stop stacking and start comparing. So here, determinism is less powerful than nondeterminism.

Corresponding to the context sensitive grammar is the nondeterministic linear bounded automaton. This acceptor is a nondeterministic Turing machine with a single tape and a two-way moving read/write head. The only restriction is that the read/write head may never move off the tape squares used to write the input string. Whether deterministic and nondeterministic linear bounded automata are equivalent has been an open question for over fifty years.

Finally, we will mention a determinism/nondeterminism problem which does not fit neatly with grammars. $\mathcal{NP}$ is the set of languages accepted by *nondeterministic* Turing acceptors whose running times are bounded by polynomials in the length of the input string. $\mathcal{P}$ is the corresponding class accepted by *deterministic* Turing acceptors whose running times are bounded by polynomials in the length of the input string. Clearly $\mathcal{NP} \supseteq \mathcal{P}$ but it has been an open question for forty years whether $\mathcal{P} \neq \mathcal{NP}$. This $\mathcal{P} \neq \mathcal{NP}$ problem does not fit neatly in the grammar framework. It is known that $\mathcal{NP} \neq L(G_1)$ that is $\mathcal{NP}$ is not the same set as the set of languages generated by context sensitive grammars, but it is unknown if $\mathcal{NP} \supset L(G_1)$ or $L(G_1) \supset \mathcal{NP}$ or whether neither of these possibilities is true.

2.6 Grammars and Acceptors

For each of the grammar classes in the Chomsky Hierarchy there is a corresponding class of acceptors. As might be expected from the nondeterminism inherent in the definition of generation by a grammar, these acceptors are, in general, nondeterministic. Since these acceptors will be described in terms of various restrictions on the allowed operations of a Turing machine, these acceptors are usually called classes of machines or classes of automata. [1]

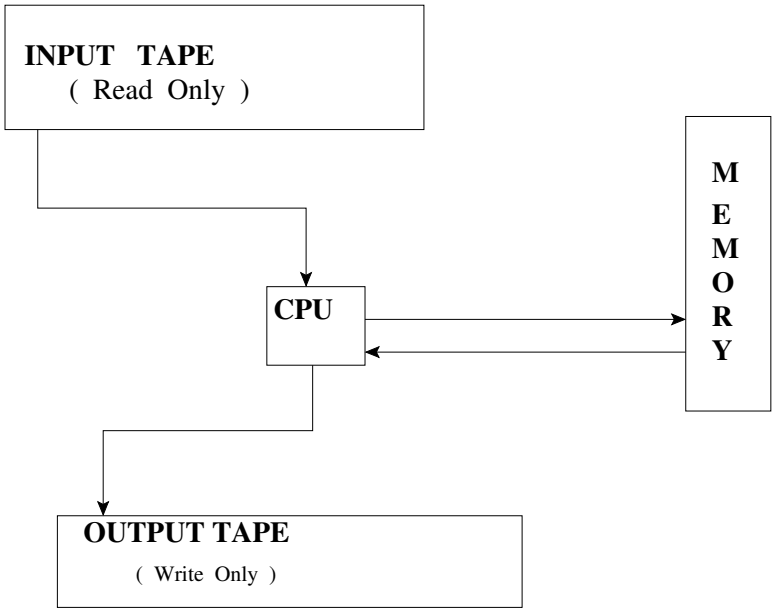
The correspondence between grammars and automata is

Regular Grammars	$\longleftrightarrow$	Nondeterministic Finite Automata
Context Free Grammars	$\longleftrightarrow$	Nondeterministic Pushdown Automata
Context Sensitive Grammars	$\longleftrightarrow$	Nondeterministic Linear Bounded Automata
General Grammars	$\longleftrightarrow$	Nondeterministic Turing Machines

[1] *Automata* is the plural of the word *automaton*. Automaton should be used for a single machine of a specific type.

2.6.1 Turing Machines

We will describe each of these automata classes in turn, starting with the most general, the Turing machine. As we've discovered in other chapters, the Turing machine is the machine model which captures the intuitive of "algorithm". The following diagram dipicts a Turing machine in a manner similar to a concentional computer.



Instead of the single tape in a typical drawing of a Turing machine, we have broken the tape into 3 tapes; an input tape (which is read-only), and output tape (which is write-only), and a memory tape (which allows both read and write). The memory tape consists of an unbounded (potentially infinite) sequence of squares, and each square contains a single symbol of the tape alphabet. This tape gives the Turing machine an unbounded memory – it never runs out of storage locations. The control box (**CPU**) contains the instructions for the machine and a bounded (local) memory, and it has a read/write head that accesses the unbounded tape memory.

An instruction of the Turing machine can change the contents of the local memory, change the contents of the memory tape square being read and move the read/write head one tape square in either direction (i.e. one square to the right or one square to the left), and read an input symbol and move the input read head one square to the right. The Turing machine may also have instructions which ACCEPT the input by writing **YES** on the output tape and terminating the computation.

This Turing machine operates by *choosing* an instruction which is consistent with the current local memory and the contents of the current squares being

read, and executes that instruction. Notice that these are nondeterministic machines, so there may be several instructions for the machine to choose from. If there is no instruction which is consistent with the local memory and the contents of the current tape squares, then this computation of the Turing machine terminates without accepting the input. If when the Turing machine is started in specified initial configuration, there is some sequence of choices that allows the Turing machine to eventually execute an ACCEPT instruction then the Turing machine accepts the input.

Ex 2.1. Show that this *nondeterministic* Turing acceptor can be turned into a *deterministic* Turing acceptor.
(NOTE: This *deterministic* Turing acceptor does **not** have to be a *recogniizer*.)

Ex 2.2. In contrast to our definition, a common definition of Turing acceptor says that the Turing machine accepts input x if when started in standard initial conditions with x on the input tape, there is some allowed computation which causes the Turing machine to halt. (Maybe the Turing machine has no further allowed instructions.)
Show that if a set (of strings) is accepted by a Turing machine according to one definition, then there is another Turing machine that accepts this same set according to the other definition.

2.6.2 LBAs (linear bounded automata)

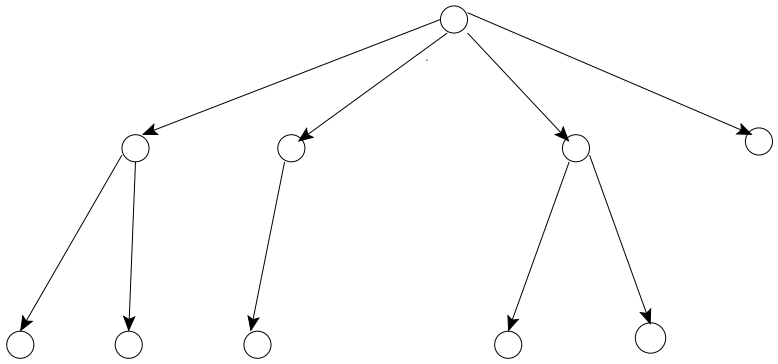
An **LBA** is a nondeterministic Turing acceptor with the limitation that the MEMORY tape has only $O(\,n\,)$ tape squares, where n is the size of the input tape. That is, as the size of the input increases the size of the MEMORY increases in proportion to the size of the input.

The **LBA** accepts the input if there is some computation which causes the **LBA** to write **YES** on the output tape. Notice that if during a computation the read/write head attempts to leave the allowed MEMORY area, then this computation is not accepting. This does not preclude another computation from being accepting.

This **LBA** can reject an input if *every* computation starting with that input string forces the read/write head to eventually leave the allowed area, or if *all* computations either leave the memory area or stay in the memory area for a very long time. Specifically, if the machine repeats a configuration twice, then the machine is in a **loop** and will continue forever. (A configuration consists of the contents of local memory, the contents of the MEMORY tape, and the positions of the heads.) If we let A be the number of symbols in the MEMORY tape alphabet (which may be much larger than the input alphabet), and let n be the length of the input string and let Q be the number of possible state of the local memory, then there are at most $A^n \cdot n^2 \cdot Q$ different configurations, and so if a computation continues longer than this number, the computation must include a repeated configuration.

Here, it may be useful to think in terms of a **TREE**. The root of the **TREE** is the initial configuration, i.e. the input is on the input tape, the read head is

at the leftmost symbol of the input, and the memory is empty. At each stage of the computation, the machine could have several possible instructions it could follow. We build the **TREE** by branching for each of these possible instructions. In the example below, the first configuration has 4 possibile instructions so there are branches to 4 possible successor configurations. At the next level, two of the configurations each have two possible successors, one configuration has only one possible successor, and the other configuration has no possible successors.



We can observe that if there is an accepting configuration then there is an accepting configuration at depth at most $A^n \cdot n^2 \cdot Q$ in the **TREE**, because any longer computation will have repeated configurations, and by removing the sequence of configurations between the repeats, we can decrease the depth of an accepting configuration. Unfortunately, keeping track of the lengths of computations may be beyond the ability of a **PDA**. So, we may be able to see that a **PDA** does not accept a string, even if the **PDA** cannot itself see so. The upshot is, if a set is accepted by a **PDA** then there is a recognizer for the set, but the recognizer may not be a **PDA**.

The instructions for a **PDA** have an input part consisting of the symbol being read on the input tape, the symbol being read on the work tape, and the contents of the local memory. On the basis of this input part, an instruction can result in one or more of the following actions:

(a) Change the position of the input read head

(b) Change the contents of local memory

(c) Write on the work tape

(d) Move the head on the work tape

(e) and, of course, ACCEPT, (write **YES**).

Ex 2.3. Sometimes an **LBA** is defined to have a single tape which functions as both the input tape and the work tape. The **LBA** is then restricted to work only on the squares on which the input was originally written, but the **LBA** is

allowed to use a tape alphabet which is much LARGER than the input alphabet. Let's call an **LBA** according to this definition, an **LBA-1**.
Show that if a set of strings is accepted by an **LBA-1** then there is an **LBA** (according to our earlier definition) which accepts the same set. (EASY)
Show that if a set of strings is accepted by an **LBA** (according to our earlier definition) then there is an **LBA-1** which accepts the same set. (a LITTLE HARDER)

2.6.3 PDAs (pushdown automata)

Context free grammars correspond to pushdown automata **PDA**s. Like the previous models, the **PDA** has an input tape and a control box, but instead of a work tape, MEMORY is a pushdown stack.

The instructions for a **PDA** consist of an input part and an action part. The input part contains the symbol being read on the input tape, the symbol on the top of the stack, and the contents of local memory. Depending on the input part, an instruction directs the **PDA** to take one or more of the following actions

(a) Remove (pop) the top symbol off the stack
(the next symbol on the stack becomes the new top symbol)

(b) Push a symbol on top of the stack

(c) Change the contents of local memory

(d) Move the input head one square (right) to see a new input symbol.

The set of possible local memory contents is called the set of *states* of the **PDA**. A selected subset of these states are called the **accepting** states. We say that the **PDA** *accepts* the input x, if when started in standard initial configuration (e.g. in the local memory state 0 with an empty stack, at the leftmost symbol of x), there is a computation which puts the automaton in an *accepting* state after the last input symbol has been read, and then the **PDA** can write **YES** on the output tape. (In the usual description of **PDA**s the output tape isn't mentioned.)

2.6.4 Finite Automata (finite state machines)

Regular grammars correspond to finite automata. A finite automaton **FA** is an automaton with its work tape cut off, that is, there is an input tape and a control box which contains local memory, but there is no extra work space, i.e. no stack, no work tape, and no work space on the input tape.

The input part of an instruction for an **FA** consists of an input symbol and the contents of local memory. Each instruction can change the contents of local memory and then move the read head to the next input symbol. The machine can still be nondeterministic because there may be several instructions with the

same input part, e.g. one instruction might read input symbol S with memory contents q and change memory contents to 17 before moving the head to the next input symbol, and there may be another instruction which has input symbol S with memory contents q and changes memory contents to 64708 before moving the head to the next input symbol. As in the **PDA**, the set of possible (local) memory contents is called the set of *states*, and a specified subset of these states are called the *accepting* states.

An **FA** accepts an input string x if when started in standard initial configuration (e.g. in the local memory state 0 at the leftmost symbol of x), there is a computation which puts the automaton in an *accepting* state after the last input symbol has been read. In conformity with other automata models, we can say that the **FA** writes **YES** on the output tape, but usual descriptions of **FA** assume that there is no output tape.

2.7 Exercises

Ex 2.4. We have just seen that any finite language has a grammar. But the grammar we constructed in the proof had the same number of productions as the number of strings in the language.

Consider the language which consists of all strings with at least **one** character and at most **ten** characters such that the first character is a letter and the remaining characters may be either letters or numbers. How many strings are in this language? If you followed the construction in the proof of observation 1, you would get a grammar with very many productions.

Find a grammar with many fewer productions which generates this language.

Give an algorithm which determines if a given string is in this language.

Ex 2.5. Show that for any two strings X and Y, $XY = YX$ if and only if there is a string W such that $X = W^{k1}$ and $Y = W^{k2}$. (By W^k we mean the result of concatenating k copies of the string W. By W^0 we mean the result of concatenating 0 copies of the string W, so $W^0 = \Lambda$.)
(HINT: Use induction on the length of the longer string.)

Ex 2.6. Consider the following grammar

$$\Sigma = \{BOY,\ GIRL,\ BALL,\ WITH,\ THE,\ PLAYS\}$$
$$V = \{S, NP, N, VP, PREP\}$$
$$P = \{S \rightarrow\ \ NP\ VP,$$
$$NP \rightarrow THE\ N,$$
$$N \rightarrow BOY,$$
$$N \rightarrow GIRL,$$
$$N \rightarrow BALL,$$
$$VP \rightarrow V\ PREP,$$
$$PREP \rightarrow WITH\ NP,$$
$$V \rightarrow PLAYS\}$$

- -

Determine whether the following strings are in the language generated by this grammar.

THE BOY PLAYS WITH THE GIRL

THE BOY PLAYS WITH THE BALL

THE BALL PLAYS WITH THE BALL

WITH THE BALL THE BOY PLAYS

THE GIRL PLAYS WITH THE PLAYS

THE BALL PLAYS WITH THE BAT

THE BOY PLAYS THE GUITAR

Ex 2.7. There is a minor hole in Theorem 6. At the finish, the end markers need to be removed. Show how to set a productions to carry out this "clean-up" phase. (You may want to assume that the acceptor is in a "special" position and a "special" state when it accepts.)

Ex 2.8. Consider the following grammar

$$
\begin{aligned}
&\Sigma = \{A, B, C, D, +, -, *,), (\ \} \\
&V = \{S,\ VAR,\ OP\} \\
&P = \{ \qquad S \to VAR, \\
&\qquad\qquad S \to VAR\ OP\ VAR, \\
&\qquad\qquad S \to (\ S\ ), \\
&\qquad\qquad VAR \to (\ S\ ), \\
&\qquad\qquad VAR \to A, \\
&\qquad\qquad VAR \to B, \\
&\qquad\qquad VAR \to C, \\
&\qquad\qquad VAR \to D, \\
&\qquad\qquad OP \to +, \\
&\qquad\qquad OP \to -, \\
&\qquad\qquad OP \to *, \qquad\qquad\qquad\qquad\qquad\qquad \}
\end{aligned}
$$

Determine which of the following strings are in the language generated by this grammar:

(A + ((B + C) - D))

(((((((B))))))

((A + (+ (B + (- C + D))))

(A + A + A)

(A + (A) + (A + (A)))

Ex 2.9. Give an example of :

1. a type 0 grammar which is not a type 1 grammar
2. a type 1 grammar which is not a type 2 grammar
3. a type 2 grammar which is not a type 3 grammar
4. Make your examples more interesting by having each of your example grammars generate a language which can also be generated by a type 3 grammar. (HINT: Try a finite language.)

Ex 2.10. What type grammars are specified by the following sets of productions?

a) { $S \to aSb$, $B \to SSa$, $aB \to Ba$ }
b) { $S \to a$, $A \to aS$, $B \to bS$, $S \to A$ }
c) { $S \to a$, $aSBa \to S$, $aaS \to aa$, $B \to b$ }
d) { $S \to A$, $A \to aSS$, $B \to SSa$, $S \to b$, $A \to B$ }

Ex 2.11. An extended regular grammar has all productions in the forms
$A \to \alpha B$ and $A \to \alpha$
where α represents a nonnull terminal string and B represents a nonterminal. Show that every extended regular grammar generates a regular language, by showing how to replace each extended regular production with regular productions.

Ex 2.12. Show that $\{a^{2n} | n > 0\}$ is a regular language.

Ex 2.13. a) Show that $\{a^n c^{2n} | n > 0\}$ is a context free language.
b) Show that the above language is not regular.

Ex 2.14. Show that $\{a^n c^m a^n | n > 0,\ m > 0\}$ is a context free language. Show that this language obeys the pumping lemma for regular languages. Does this show that this language is regular?

Ex 2.15. Prove the stronger pumping lemma:

Lemma 3 (Pumping Lemma for Regular Languages)**.** *Let p be the number of nonterminals in a grammar generating the regular language A. If A contains a string whose length is greater than p, then there is a way to write that string as three parts $\alpha_1\ \alpha_2\ \alpha_3$ so that*

1. *α_2 is* **not** *the null string,*
2. *$|\alpha_1| + |\alpha_2| \leq p$*
3. *for each $i \geq 0$,*
   $\alpha_1\ \alpha_2^i\ \alpha_3$ is also in the language A.

Ex 2.16. Show that the following grammar generates $\{a^n b^n c^n\}$

$$G = (\{a, b, c\},\ \{S, B, C\},\ P,\ S)$$

$$P = \left\{ \begin{array}{ll} S \to aBC & aB \to ab \\ S \to aCB & bB \to bb \\ S \to aSCB & bC \to bc \\ CB \to BC & cC \to cc \\ BC \to CB & \end{array} \right\}$$

Ex 2.17. If you add to the above grammar the two new productions
$B \to b$ and
$C \to c$,
does this new grammar generate the same language as the original grammar?

Ex 2.18. Show that $\{a^n b^n c^n d^n\}$ is a context sensitive language, which is not a context free language.

Ex 2.19. Give examples of several other phrase structure languages which are not context sensitive.

Ex 2.20. Show that $\{a^n b^{n^2}\}$ is a recursive language.
What is the simplest type of grammar which can generate the language?
Show that no simpler type of grammar can generate the language.

Ex 2.21. Write an algorithm which takes as input a set of regular productions and a string of terminals and gives as output a "YES" or "NO" depending on whether or not string is in the language generated by the productions.

Sketch of algorithm:
Start at the right hand end of the terminal string. Find all productions which have the last character as right hand side. Remember the set of nonterminals which are the left hand side of these productions. For each succeeding terminal find all productions whose right hand side consists of the terminal symbol and one of the remembered nonterminals, and remember the nonterminals on the left hand side of these productions. After the last (left most) nonterminal is processed, see if any of the remembered nonterminals is the start symbol of the grammar. If so, print "YES", if not print "NO". This algorithm can also fail "in the middle" if you ever find a terminal symbol such that it and none of the remembered nonterminals is the right side of any production. For such failures your algorithm should print out the terminal string and indicate the location of the "error".

Implementation:
As a first thought, you might index each production by a pair of integers (i, j) such that i is the index of the terminal symbol and j is the index of the nonterminal. If there are m nonterminals in the grammar, you could use $m + 1$ as the indication that there is no nonterminal on the right hand side. You could consider, then, storing in an array at location (i, j) the nonterminal on the left of the production. This scheme might not be very good for two reasons. First there may be several productions with the same right hand side and instead of storing a single nonterminal, you would have to store a set of nonterminals. Second, many (i, j) pairs may not correspond to any productions. Therefore, it would be more efficient to store the productions as a sorted list, sorted on i and then on j, with pointers to the first production which has an i. Then when terminal symbol i is encountered, your algorithm can go to the first i production and step through the productions to see if a remembered symbol corresponds to any of the j's in the (i, j) list. If yes, then the left side is put into a new remembered list (?), or set (?), or array (?).