Theory of Formal Languages (LBA and Context Sensitive Grammars)

Lecture 6: Mar. 06, 2021

Prof. K.R. Chowdhary *: Professor of CS*

Disclaimer: *These notes have not been subjected to the usual scrutiny reserved for formal publications. They may be distributed outside this class only with the permission of the Instructor.*

6.1 Introduction

The Chomsky hierarchy, as originally defined by Noam Chomsky, comprises four types of languages and their associated grammars and machines. Table 6.1 shows the Chomsky hierarchy of grammars. These languages form a strict hierarchy, that is,

$$\begin{aligned} \textit{Regular languages} &\subset \textit{Context-free languages} \\ &\subset \textit{Context-sensitive languages} \\ &\subset \textit{Recursively enumerable languages.} \end{aligned}$$

Learning Outcomes of this Chapter:

1. Determine a language's place in the Chomsky hierarchy (regular, context-free, recursively enumerable). [Assessment]
2. Usage
3. Familiarity

6.2 Context Sensitive Grammars and Languages

We have discussed other types of languages besides those in the "classical" Chomsky hierarchy. For example, we noted that deterministic pushdown automata were less powerful than Nondeterministic Pushdown Automata (NPDA). Table 6.2 shows some of the language classes that fit readily into this hierarchy.

Not all language classes fit neatly into a hierarchy. For example, the linear languages, which (like deterministic context-free languages) fit neatly between the regular languages and the context-free languages. However, there are languages that are linear but not deterministic context-free, and there are languages that are deterministic context-free but not linear.

In fact, mathematicians have defined dozens, maybe hundreds, of different classes of languages, and how these relate to one another. We should know at least the four "classic" categories that are taught in almost every textbook on the subject.

Table 6.1: Chomsky Hierarchy

S. No.	Language	Grammar	Machine	Language Example
1.	Regular language (type-3 language)	Regular grammar : Right-linear grammar/ Left-linear grammar	Deterministic or nondeterministic finite-state acceptor	a^*
2.	Context-free language (type-2 language)	Context-free grammar	Nondeterministic pushdown automata	$a^n b^n$
3.	Context-sensitive language (type-1 language)	Context-sensitive grammar	Linear-bounded automata	$a^n b^n c^n$
4.	Recursively enumerable language (type-0 language)	Unrestricted grammar	Turing machine	Any computable function

Table 6.2: Extended Chomsky Hierarchy

S.No.	Language	Machine
1.	Regular language	Deterministic or nondeterministic finite state acceptor
2.	Deterministic context-free language	Deterministic pushdown automaton
3.	Context-free language	Nondeterministic pushdown automaton
4.	Context-sensitive language	Linear-bounded automaton
5.	Recursive language	Turing machine that halts
6.	Recursively enumerable language	Turing machine

A family of grammars $G = (, V, S, P)$ is defined by restrictions placed on the form of production rules. We have studied that a regular grammar is defined as consisting only productions of,

$$A \to aA \mid a \mid \varepsilon$$

where $A \in V$, $B \in V$, and $a \in \Sigma$. A CFG is defined as,

$$A \to \alpha$$

where $A \in V$, and $\alpha \in (V \cup \Sigma)^*$.

Definition 6.1 Unrestricted Grammar. *An unrestricted grammar $G = (\Sigma, V, S, P)$ has productions of the form $u \to v$, where, $u \in (V \cup \Sigma)^*$ and $v \in (V \cup \Sigma)^*$, with at least one variable symbol in u.*

Example 6.2 *The language $\{a^n b^n c^n \mid n \geq 0\}$, which we know that it cannot be derived using context-free grammar, can be generated by following context-sensitive grammar.*

$$\begin{aligned} V &= \{S, A, C\} \\ \Sigma &= \{a, b, c\} \\ S &\to aAbc \mid \varepsilon \\ A &\to aAbC \mid \varepsilon \\ Cb &\to bC \\ Cc &\to cc. \end{aligned}$$

Applying the above rules, we get,

$$\begin{aligned} S &\Rightarrow aAbc \\ &\Rightarrow aaAbCbc \ ; \text{ using } A \to aAbC \\ &\Rightarrow aabCbc \ ; \text{ using } A \to \varepsilon \\ &\Rightarrow aabbCc \ ; \text{ using } Cb \to bC \\ &\Rightarrow aabbcc. \ \text{ using } Cc \to cc \end{aligned}$$

Definition 6.3 Context-sensitive grammar. *A* context-sensitive *(CS) grammar $G = (\Sigma, V, S, P)$ has productions of the form $u \to v$, where $u, v \in (V \cup \Sigma)^*$ and $|u| \leq |v|$, and u has got at least one variable symbol in it.*

Theorem 6.4 *Every context-free language is context-sensitive.*

Proof: The productions of a context-free grammar have the format $A \to v$, where $v \in (\Sigma \cup V)^*$. The productions of a context-sensitive grammar have the form $xAy \to xvy$. Thus, in $A \to v$, with x, y as ε, it is context-sensitive. ■

Theorem 6.5 *There exists a context-sensitive language that is not context-free.*

Proof: The language $\{a^n b^n c^n \mid n \geq 0\}$ is not context-free (we used a pumping lemma to show this). We can show that it is context-sensitive by providing an appropriate context-sensitive grammar. Following are the productions for one such grammar. Note that it is different set than in example 6.2, as ε-production has been removed here.

$$
\begin{aligned}
&S \to aXBC \mid aBC \\
&X \to aXBC \mid aBC \\
&CB \to BC, \quad aB \to ab \\
&bB \to bb, \quad bC \to bc, \quad cC \to cc
\end{aligned}
$$

Using these rules we can generate the string *aabbcc*.

$$
\begin{aligned}
S &\Rightarrow aXBC \text{ ;using } S \to aXBC \\
&\Rightarrow aaBCBC \text{ ;using } X \to aBC \\
&\Rightarrow aabCBC \text{ ;using } aB \to ab \\
&\Rightarrow aabBCC \text{ ;using } CB \to BC \\
&\Rightarrow aabbCC \text{ ;using } bB \to bb \\
&\Rightarrow aabbcC \text{ ;using } bC \to bc \\
&\Rightarrow aabbcc \text{ ;using } cC \to cc
\end{aligned}
$$

■

Are there different grammars for the same language $a^n b^n c^n$, as demonstrated in example 6.2, and theorem 6.5? Truly not. In such situations, where the language is same, a grammar can be transformed into another grammar, or both can be transformed into some standard format such that they are identical.

Theorem 6.6 *Every context-sensitive language is recursive.*

Proof: A context-sensitive grammar is noncontracting; moreover, for any integer n there are only a finite number of sentential forms of length n. Therefore, for any string w we can set a bound on the number of derivation steps required to generate w, hence a bound on the number of possible derivations. The string w is in the language if and only if one of these derivations produces w. ■

Theorem 6.7 *There exists a recursive language that is not context-sensitive.*

Proof: The proof is left as an exercise. ■

6.3 Linear bounded Automata

These were originally developed as models for actual computers rather than models for the computational process. They have become important in the theory of computation even

though they have not emerged in applications to the extent which pushdown automata enjoy.

In 1960, Myhill introduced an automation model today known as deterministic linear bounded automaton. Shortly thereafter, Landweber proved that the languages accepted by a deterministic LBA are always context-sensitive. In 1964, Kuroda introduced the more general model of (nondeterministic) linear bounded automata, and showed that the languages accepted by them are precisely the context-sensitive languages.

A linear bounded automaton (LBA) is a multi-track Turing machine which has only one tape, and this tape is exactly the same length as the input. That seems quite reasonable. We allow the computing device to use just the storage it was given at the beginning of its computation[1]. As a safety feature, we shall employ end-markers ($<$ on the left and $>$ on the right) on our LBA tapes and never allow the machine to go past them. This will ensure that the storage bounds are maintained and help keep our machines from leaving their tapes.

Definition 6.8 Linear bounded Automata. *A Turing machine that uses only the tape space occupied by the input is called a linear-bounded automaton (LBA).*

At this point, the question of accepting sets arises. Let's have linear bounded automata accept just like Turing machines. Thus for LBA halting means accepting.

For these new machines computation is restricted to an area bounded by a constant (the number of tracks) times the length of the input. This is very much like a programming environment where the sizes of values for variables is bounded.

A linear bounded automaton (LBA) is a nondeterministic Turing machine $M = (Q, \Sigma, \Gamma, \delta, s, H)$ (see Fig. 6.1) such that:

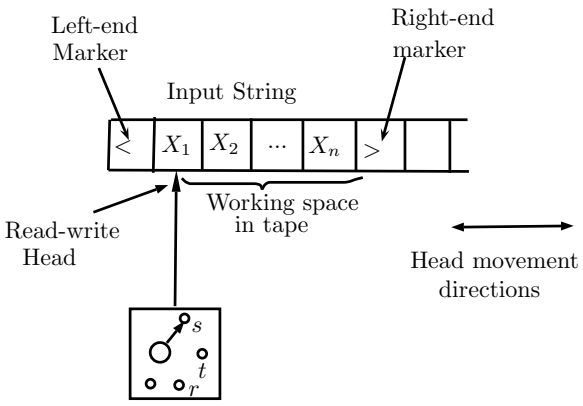

Figure 6.1: Physical model of Linear bounded Automata.

- There are two special tape symbols '<'and '>'(the left end marker and, right end marker).
- Σ is input alphabet, Γ is tape alphabet, $\Gamma = \Sigma \cup \{<, >\}$.

[1]For example, space consumed by a C program is equal to size of program code in machine language, plus the sum of size of all the variables defined, provided that program does not create dynamic storage.

- The TM begins in the configuration $(s, < x >, 0)$, where s is start state, the R/W head points to 1st symbol of input x, and there is 0 or B after $>$ end marker symbol.
- The TM cannot replace '<'or '>'with anything else, nor move the tape head left of '<'or right of '>'.
- $H = \{t, r\}$ is set of halting states, t is accept, and r is reject state.
- $\delta : (Q - \{t, r\}) \times \Gamma \to Q \times \Gamma \times \{L, R\}$.

When expressed as production, it is : $\alpha \to \beta$, with $|\alpha| \leq |\beta|$. And, the LBA is nondeterministic LBA.

Theorem 6.9 *A set of strings accepted by a linear-bounded automaton is a context-sensitive language.*

Proof: A linear-bounded automaton, as defined by Myhill (1960) and, following him, by Landweber, is a deterministic automaton specified by a tuple $(Q, \Sigma, \Gamma q_0, \delta, H)$, where the set Q of states, tape symbols Γ, and input alphabet Σ are finite sets, the initial state q_0 is an element of Q, the set H of final states is a subset of Q and, finally, the behavior function δ is a function from $(Q - H) \times \Gamma \to Q \times \Gamma \times \{L, S, R\}$ where $\Gamma = \Sigma \cup \{B, <, >\}$, satisfying the condition: if $\delta(<, s) = (<, s', k)$. Here '<'is is a symbol outside Γ and called the boundary symbol[2]. Similar is the case with right boundary symbol '>'.

The automaton functions as follows. It is given, as input, a tape blocked into squares containing a string $< x >$, where x is a string in Σ and '<', '>'are the boundary symbols. Before operating on the input it is set in the initial state q_0. At the initial stage, then, it reads the left boundary in the state q_0. In general, if it is reading a symbol a in a state q, and if $\delta(a, q) = (a', q', k)$, it prints a' in the scanned square, and moves k squares to the right (i.e., one square to the right, one square to the left, or no move at all, according as $k = 1, -1$, or 0, respectively) and enters in the state q'. Continuing the calculation in this way, if it runs off the right end of the given tape and at this time it finds itself in one of the final states of H, then by definition the string x is *accepted*, or otherwise, *rejected*, by the automaton. The set of all strings accepted is the *language accepted* by the automaton.

Now, if we allow the behavior function to be multivalued, we have a nondeterministic automaton. We understand, henceforth, by *linear-bounded automaton* a possibly nondeterministic automaton thus obtained. Only when there is possibility of confusion, we use the phrase "nondeterministic linear-bounded automaton" as a synonym of "linear-bounded automaton" . A linear-bounded automaton in Myhill's sense is referred to as a deterministic linear-bounded automaton . We then mean by a string accepted by a nondeterministic automaton M, a string for which there is a computation of M which, given the string as input, ends up off the right end of the tape in a final state. On the other hand, a string is said to be rejected by M if there is a computation of M which, given the string as input, never ends, or ends up off the left end of the tape, or, finally, ends up off the right end of the tape in a non-final state. Because of the nondeterminacy of M, a string can in general be both accepted and rejected by M . The set of all strings accepted by M is called the *languages accepted* by M. The set of all strings rejected by M is called the *language rejected* by M.

■

[2]This condition means that the symbol '<' is not effected during the computation

6.4 Space bounds of LBA

The length restriction in CSGs has some intuition. What is the motivation for the restriction $|\alpha| \leq |\beta|$ in context-sensitive rules? The Idea is that, in a context-sensitive derivation $S \Rightarrow ... \Rightarrow ... \Rightarrow w$, all the sentential forms are of length at most $|w|$. This means that if L is context-sensitive, and we are trying to decide whether $w \in L$, we only need to consider possible sentential forms of length $\leq |w|$. So intuitively, we have the problem under control, at least in principle. Note that, in the sentential form,

$$S = \alpha_a \Rightarrow \alpha_2 \Rightarrow ... \Rightarrow \alpha_i ... \Rightarrow \alpha_n = w,$$

there can be at the most only $(|\Gamma| + |V|)^{|w|}$ number of derivations, which is a finite number.

By contrast, without the length restriction, there is no upper limit on the length of intermediate forms that might appear in a derivation of w. So if we are searching for a derivation for w, how do we know when to stop looking? Intuitively, the problem here is wild and out of control. We will see this intuition made more precise as we proceed.

The machine therefore has just a finite amount of memory, determined by the length of the input string. We call this a linear bounded automaton. An equivalent definition of an LBA is that it uses only k times the amount of space occupied by the input string, where k is a constant fixed for the particular machine. However, it is possible to simulate k tape cells with a single tape cell, by increasing the size of the tape alphabet.

At the bottom level of the Chomsky hierarchy, it makes no difference: every NFA can be simulated by a DFA. At the top level, the same happens. We have defined a "deterministic" version of Turing machines – but any "nondeterministic Turing machine" can be simulated by a deterministic one.

At the context-free level, there is a difference – we need NPDA to account for all context-free languages. For example, $\Sigma^* - \{ww \mid w \in \Sigma^*\}$ is a context-free language whose complement is not context-free. However, if L is accepted by a DPDA then so is its complement – can just swap accepting and non-accepting states.

What about the context-sensitive level? Are NLBAs (nondeterministic LBAs) strictly more powerful than DLBAs? This is still an *open problem!* (Cannot use the same argument because it turns out that CSLs are closed under complementation – it was only shown in 1988.)

Example 6.10 *Maximum number of configurations of an LBA.*

Let an LBA M has Q number of states, m number of symbols in the tape alphabets, and input of length n. Then, this M can have at most $\alpha(n)$ configurations, expressed by,

$$\alpha(n) = m^n * n * Q, \tag{6.1}$$

where m^n possible number of tape contents, n is possible number of head positions for given input of length n, and Q is number of states. □

6.5 Detecting non-acceptance in LBA

Suppose T is an LBA. How might we detect that w is not in $L(T)$? Clearly, if there is an accepting computation for w, there is one that does not pass through exactly the same machine configuration twice, if it did, we could shorten it.

Since the tape is finite, the total number of machine configurations is finite (though large). So in theory, if T runs for long enough without reaching the final state, it will enter the same configuration twice, and we may as well abort.

Note that on this view, repeated configurations would be spotted "not by T itself", but by "us watching", or perhaps by some super-machine spying on T.

For Turing machines with unlimited tape space, this reasoning does not work. Is there some general way of spotting that a computation is not going to terminate?

Wider significance of Turing machines Turing machines is important because (it is generally believed that) anything that can be done by any mechanical procedure or algorithm can in principle be done by a Turing machine (Church-Turing thesis). For example,

- Any language $L \subseteq \Sigma^*$ that can be "recognized" by some mechanical procedure can be recognized by a TM.
- Any mathematical function $f : \mathbb{N} \to \mathbb{N}$ that can be computed by a mechanical procedure can be computed by a TM (for example, representing integers in binary, and requiring the TM to write the result onto the tape.)

6.6 Language acceptability by LBA

In the followings, we prove some important proofs about LBA. Because of finite number of restricted length of the tape, the total configurations under which a LBA can go are finite, hence the languages accepted by LBA (Context sensitive languages) are decidable. On the same grounds it can be concluded that for LBA, the halting problem is decidable. The LBA also, every LBA accepts context sensitive language (CSL), and for every CSL there exists an LBA which accepts it, i.e., a language is accepted by LBA if and only if it is CSL.

Theorem 6.11 *A language is accepted by an LBA iff it is context sensitive.*

Proof: Part 1. If L is a CSL, then L is accepted by some LBA.

1. Let $G = (V, \Sigma, S, P)$ be the given grammar such that $L(G) = L$, which is CSL, and we need to show that there is an LBA which recognizes L.
2. Construct LBA M with tape alphabet $\Sigma \times \{V \cup \Sigma\}$ (2- track machine).
3. First track holds input string w, and second track holds a sentential form α of G, with initial value as S.

4. if $w = \epsilon$, M halts without accepting.
5. Repeat :
   (a) Non-deterministically select a position i in α.
   (b) Non-deterministically select a production $\beta \to \gamma$ of G.
   (c) If β appears beginning in position i of α, replace β by γ there.
   If the sentential form is longer than w, LBA halts without accepting.
   (d) Compare the resulting sentential form on track 2 with w on track 1. If they match, accept. If not go to step 1.

Part 2. If there is a linear bounded automaton M accepting the language L, then there is a context sensitive grammar generating $L - \{\epsilon\}$.

Following is the approach to proof. Let the grammar of the following language simulates the moves on LBA.

1. Derivation simulates moves of LBA
2. Three can be three types of productions in this grammar:
   (a) Productions that can generate two copies of a string in Σ^*, along with some symbols that act as markers to keep the two copies separate.
   (b) Productions that can simulate a sequence of moves of M. During this portion of a derivation, one of the two copies of the original string is left unchanged; the other, representing the input tape to M, is modified accordingly.
   (c) Productions that can erase everything but the unmodified copy of the string, provided that the simulated moves of M applied to the other copy cause M to accept.

■

Example 6.12 *Show that the language* $L = \{a^n b^n c^n \mid n \geq 0\}$ *is context-sensitive.*

The production rules, we will be making in use are listed without any specific order as,

1. $S \to abc \mid aAbc$
2. $Ab \to bA$
3. $Ac \to Bbcc$
4. $bB \to Bb$
5. $aB \to aa \mid aaA$

For $w = aabbcc$, derivation can be obtained as follows:

$$S \Rightarrow aAbc \Rightarrow abAc \Rightarrow abBbcc$$
$$\Rightarrow aBbbcc \Rightarrow aabbcc$$

Note that above strings also also be generated using the set of productions: $\{S \rightarrow SBc \mid abc, cB \rightarrow Bc, bB \rightarrow bb\}$. The reason for this being – we can always transform one grammar into other through simplification, normal forms, and reductions. We note that both of these grammars are context sensitive grammars.

The limitation (of size of storage) of LBA makes the LBA a somewhat more accurate model of computers that actually exists than a Turing machine, whose definition assumes unlimited tape.

Theorem 6.13 *The halting problem is solvable for linear bounded automata.*

Proof: Our argument here will be based upon the number of possible configurations for an LBA M. Let's assume M has one track (this is allowed because one can use additional tape symbols to simulate tracks as we did with Turing machines), l instructions ($|Q| = l$), an alphabet of Γ tape symbols, and an input tape which is $|w| = n$ characters in length. The M has same configuration as a Turing machine and consists of followings:

1. an instruction (current state),
2. the tape head's position, and
3. the content of the tape.

For example, $w_i q_i w_j$, where current instruction is q_i, head position is first symbol of w_j, and content of tape is $w_i w_j$. We now ask: how many different configurations can there be there? It is not too difficult to find out. With $|Gamma \cup V| = m$, symbols and a tape which is $|w| = n$ squares long, we can have only m^n different tapes. The tape head can be on any of the n squares and, we can be executing any of the l instructions. Thus there are only $m^n * n * l$ possible different configurations for the LBA M.

Let us return to a technique we used to prove the pumping lemma for finite automata. We observe that if the LBA enters the same configuration twice then it will do this again and again, indicating that it is stuck in a loop! The theorem follows from this statement. We only need to simulate and observe the LBA for $n \times m^n \times l$ steps.

If M has not halted within $m^n * n * l$ steps, it must be in a repeating configuration, and therefore looping. We can therefore decide after this number of steps whether or not the input string will be accepted or rejected. The The halting problem is solvable for linear bounded automata, as the following is is decidable:

$$Halt_{LBA} = \{< M, w > | \ M \text{ is an LBA and } M \text{ halts on } w.\}$$

As linear-bounded automata are recognizers for context-sensitive languages, and every context-sensitive language must be *recursive*. Since, we are able to distinguish the presence of loop (which in TM we cannot !), the halting problem is solvable for LBA. ■

From the above proof, we also get two corollaries.

Corollary 6.14 *The membership problems for sets accepted by linear bounded automata are solvable.*

Proof: The above can also be expressed by,

$$A_{LBA} = \{< M, w > | \ M \text{ is an LBA and } M \text{ accepts } w\},$$

which is decidable. That is, an LBA that stops on input w must stop in at most $|w|$ steps. ■

Corollary 6.15 *The sets accepted by linear bounded automata are all recursive.*

Theorem 6.16 *Every CSL is Recursive.*

Proof: Construct a 3-tape nondeterministic TM M to simulate the derivation of G.

1. First tape holds the input string.
2. Thirds tape is for the derivation.
3. Second tape holds the sentential form generated by the simulated derivation.

■