Context-sensitive Languages and Linear Bounded Automata

Josh Bax
Andre Nies, Supervisor

November 15, 2010

Contents

1 Definitions of Context-Sensitive Languages

1.1 Introduction

The first complete description of the context-sensitive languages (CSLs) was developed by Noam Chomsky in the 1950's as a tool for linguistic analysis. A context-sensitive language is defined by a set of rewriting rules (called a grammar) involving symbols from a given finite set of symbols. These rules take a symbol from one distinct set, the variables, and replace it with one or more symbols from another set, the terminal symbols. The application of these rules depends on the symbols preceding and following the variable symbol, hence they are called context-sensitive. There are other forms of grammar: regular, context-free and unrestricted, classified according to the form of their productions. This classification was made by Chomsky and now forms what is known as the Chomsky hierarchy[1]. Within this hierarchy, the class of context-sensitive languages contains the class of context-free languages, and is itself contained in the recursively enumerable languages.

Over time the context-free languages have been used in parsers and compilers and the recursively enumerable languages have been found to be equivalent to the class of languages recognisable by Turing machines. Context-sensitive languages however, have not been so widely used and it is this seeming omission that spurred this project.

The aim of this text is to present theorems that show the place of the context-sensitive languages within the theory of computation. The primary source is the comprehensive work, *(*Theory of Formal Languages with Applications) by Simovici and Tenney[4], while some parts also make use of Sipser[5].

In this first section we investigate different definitions for context-sensitive languages which are useful in understanding their behaviour. We cover grammars, as mentioned above; non-contracting grammars and linear bounded automata (LBA). Then we show that all of these are equivalent. The proof of equivalence with LBA is due to Kuroda[3] and is instrumental in understanding the place of context-sensitive languages in terms of their computational complexity. In the second part we examine the properties of the class of context-sensitive languages: closure under elementary set operations, including the famous Immerman-Szelepcsényi theorem which has great importance for complexity theory. We then look at the place of the context-sensitive languages in the space complexity hierarchy and the decidability of linear bounded automata.

1.2 First Definition of Context Sensitive Languages

Definition 1.2.0 A **grammar** is a quadruple (V, Σ, S, P), such that:

V is a finite set of **variable symbols**.

Σ is the alphabet (of **terminal symbols**) of the grammar. It is required that $\Sigma \cap V = \emptyset$.

$S \in V$ is the starting variable.

P is a finite set of **productions** π_i of the form $\alpha \to \beta$, where $\alpha, \beta \in (\Sigma \cup V)^*$. $\square$

Definition 1.2.1 A grammar G defines a unique subset of $(\Sigma \cup V)^*$ called the set of **sentential forms** of G by the following rules:

- S is a sentential form of G
- $a' := a_1 \beta a_2$ is a sentential form if and only if $a := a_1 \alpha a_2$ is a sentential form, where $a_1, a_2 \in (\Sigma \cup V)^*$ and $\alpha \to \beta$ is a production of G

In this case we say a **derives** a' in G and may write this as $a \vdash a'$. $\square$

Example 1.2.2 Let $G = (\{S\}, \{a\}, S, \{S \to a, S \to aSa\})$.
Some sentential forms of G:

$$\begin{gathered} S \vdash a \\ S \vdash aSa \\ aSa \vdash aaa \end{gathered}$$

$a, aaa, aaaaa, aaaaaaa$ are all strings generated by G. Clearly $L(G)$, the language generated by G, is $\{a^n | \text{n is odd}\}$. $\square$

Example 1.2.3 Some grammars may not generate any strings at all. Let $G_1 = (\{S, A, B\}, \{c\}, S, S \to A, A \to B, c \to c)$. The only possible derivation is:

$$\begin{gathered} S \vdash A \\ A \vdash B \end{gathered}$$

But this does not end in a terminal string so $L(G_1) = \emptyset$. $\square$

It is worth noting that while still quite general, a grammar is a specialised form of *semi-Thue system* (also called a string rewriting system). These make no distinction between terminal and variable symbols and may operate as recognisers or generators. In fact they are identical in function to finite presentations of monoids.

Definition 1.2.4 Given a grammar $G = (V, \Sigma, S, P)$; G is **context-sensitive** if every production in P is of the form

$$\alpha A \beta \to \alpha \gamma \beta,$$

with $A \in V$, $\alpha, \beta \in (\Sigma \cup V)^*$ and $\gamma \in (V \cup \Sigma)^* - \lambda$. However $S \to \lambda$ is allowed provided that there is no rule in P with S on the right. A language $L \subseteq \Sigma^*$ is a **context-sensitive language** if it is generated by some context-sensitive grammar $G = (V, \Sigma, S, P)$. This means that for every string $s \in L$ there is a derivation of s from S, using the productions in P. $\square$

Note that the grammar generating L is not unique; if G generates a language $L(G)$, then we can always construct a grammar G' that also generates $L(G)$ by inserting an arbitrary number of redundant productions eg. $A \to B, B \to A$ for $A \in V, B \in V' - V$.

Example 1.2.5 Let $L := \{a^n b^n c^n | n \in N\}$ then L is a context-sensitive language.
To generate L use the following grammar $G := (\{A, B, C, H, S\}, \{a, b, c\}, S, P)$ and define

$$\begin{array}{llll} P := \{ & S \to aSBC, & CB \to HB, & aB \to ab, \\ & S \to aBC, & HB \to HC, & bB \to bb, \\ & & HC \to BC, & bC \to bc, \\ & & & cC \to cc\} \end{array}$$

To show that G does indeed generate L, we will look at the derivation scheme in general. Starting with S, the only possible productions which can be applied are $S \to aSBC$ or $S \to aBC$. Suppose we apply $S \to aSBC$ $n-1$ times to obtain the sentential form

$$a^{n-1}S(BC)^{n-1}, n > 0$$

At this point we can apply $S \to aBC$ to get

$$a^n(BC)^n$$

Now the B and C variables must be rearranged into the correct order, this is the function of the productions involving the variable H. Clearly, applying the productions $CB \to HB, HB \to HC, HC \to BC$ in sequence will transform one instance of CB to BC. Hence the sentential form $a^n(BC)^n = a^nB(CB)^{n-1}C$ becomes

$$a^nB^nC^n$$

The remaining rules in P simply rewrite variables as terminal symbols from left to right beginning with aB, leaving the desired string $a^nb^nc^n$.

But does G generate any strings outside of L? To be a terminal string of G every variable must be replaced with a terminal symbol. This can only be done by using productions from the third column above. Also notice that the left side of each of these has first a terminal, then a variable symbol and that these are dependent on previous rules from that column having been applied, eg. $cC \to cc$ requires a c, but this means at least one application of $bC \to bc$ has occurred. This means that these rules must be applied in this order: at least one application of $aB \to ab$; zero or more applications of $bB \to bb$; at least one application of $bC \to bc$; zero or more applications of $cC \to cc$. Additionally we are constrained by the fact that the first column rules specify that there must be an equal number of a, B and C symbols. So if every variable symbol is to be replaced, we can only start with sentential form $a^nB^nC^n$ as above. Hence G generates L. $\square$

1.3 The Chomsky Hierarchy

As mentioned in the introduction, context-sensitive languages are members of a hierarchy of formal languages of increasing complexity. This is very useful in terms of understanding how the class of context-sensitive languages relates to those of lesser or greater complexity and how this is classification is perhaps a natural one given our definition in terms of grammars.

L_0 - Every grammar is L_0, these characterise the *recursively enumerable* languages and are equivalent to the languages recognisable by Turing machines. An example of a recursively enumerable language is $\{< T, s > \mid < T, s >$ is an encoding of Turing machine T which accepts $s\}$.

L_1 - *Context-sensitive* grammars, with productions of the form $\alpha A\beta \rightarrow \alpha\gamma\beta$. An example is $\{a^n b^n c^n | n \in \mathbb{N}\}$ as above.

L_2 - *Context-free* grammars, with productions of the form $A \rightarrow \beta$ where $A \in V$ and $\beta \in (V \cup \Sigma)^*$. An example is $\{a^i b^j c^k | i, j, k \geq 0$ and $i = j$ or $j = k\}$.

L_3 - *Regular* grammars, with productions of the form $A \rightarrow \alpha B$ or $A \rightarrow \alpha$ for $A, B \in V$ and $\alpha \in \Sigma$. An example is $\{a^n | n$ is even $\}$.

By definition each class is contained in the preceding class. So every language generated by a context-sensitive grammar is recursively enumerable; but some context-sensitive languages are not context-free. The previously mentioned language $\{a^n b^n c^n | n \in N\}$ is an example of a CSL that is not context-free. However, to prove this we will need a property of context-free languages that does not apply to CSLs.

Theorem 1.3.6 (Pumping lemma for context-free-languages): If L is a context-free language then there is some natural number p (known as the pumping length) such that if $s \in L$ and $|s| \geq p$ then s can be written as $s = vwxyz$ where for each $i \geq 0, vw^i xy^i z \in L$, $|wy| > 0$ and $|wxy| \leq p$. □

Example 1.3.7 $L = \{a^n b^n c^n | n \in N\}$ *is not context-free.*

Suppose for contradiction that L is a context-free language and that p is its pumping length. Let $s = a^p b^p c^p$. Then, by the pumping lemma, s can be written as $s = vwxyz$. Using the conditions that $|wxy| \leq p$ and $|wy| > 0$, we see that there are five possible forms that wxy may take:

1. $wxy = a^m$ for $m \leq p$
2. $wxy = a^m b^n$ where $m + n \leq p$
3. $wxy = b^m$ for $m \leq p$
4. $wxy = b^m c^n$ where $m + n \leq p$
5. $wxy = c^m$ for $m \leq p$

Now by assumption $vw^i xy^i z \in L$ for every $i \geq 0$. Clearly, in all cases $vw^i xy^i z = a^m b^n c^o \notin L$ for $i > 1$ and $m, n, o \in \mathbb{N}$ as $m \neq n$ or $m \neq o$. Contradiction. □

Unfortunately there is no correspondingly simple pumping lemma for context-sensitive languages. To prove that a given language has no context-sensitive grammar requires a different way of looking at this class of languages.

1.4 Second Equivalent Definition of Context Sensitive Languages

Definition 1.4.8 Given a grammar $G = (V, \Sigma, S, P)$, we say that G is **length increasing** if for all productions $\alpha \to \beta$ in P we have that $|\alpha| \leq |\beta|$. $\square$

Context-sensitive languages can also be defined as the class of languages generated by length increasing grammars. It is evident from the definition of context-sensitive grammars that they are necessarily length increasing. The fact that all length increasing grammars generate context-sensitive languages is slightly more surprising.

Theorem 1.4.9 Given a length increasing grammar G_1 there is a context-sensitive grammar G_2 such that $L(G_1) = L(G_2)$.

Since G_1 is length increasing, each production can be written as

$$X_0X_1 \ldots X_m \to Y_0Y_1 \ldots Y_n,\ m \leq n,$$

where $X_i, Y_j \in (\Sigma \cap V)^*$ for all $0 \leq i \leq m$, and $0 \leq j \leq n$.
This is not necessarily a context-sensitive production. For instance $AB \to CDE$ is length increasing but not context-sensitive. Context-sensitive productions can have only one variable substituted for on the right-hand side. Hence to find an equivalent grammar, we need context-sensitive productions that derive the sentential form on the right from the one on the left. This can be done quite simply; in the above example $AB \to CDE$ can be rewritten as $AB \to XB, XB \to XDE, X \to C$, provided that X doesn't appear in any other productions. In the general case:

$$\begin{array}{rcl}
X_0X_1 \ldots X_m & \to & Z_0X_1 \ldots X_m \\
Z_0X_1 \ldots X_m & \to & Z_0Z_1 \ldots X_m \\
 & \ldots & \\
Z_0Z_1 \ldots Z_{m-1}X_m & \to & Z_0Z_1 \ldots Z_{m-1}Y_m \ldots Y_n \\
Z_0Z_1 \ldots Z_{m-1}Y_m \ldots Y_n & \to & Y_0Z_1 \ldots Z_{m-1}Y_m \ldots Y_n \\
 & \ldots & \\
Y_0Y_1 \ldots Z_{m-1}Y_m \ldots Y_n & \to & Y_0Y_1 \ldots Y_{m-1}Y_m \ldots Y_n
\end{array}$$

Where $Z_k \in (\Sigma \cap V)^*$ for all $0 \leq k \leq m-1$. Note that each Z_k appears only in the above set of productions. This and the fact that the entire sentential form is recorded in the productions (as opposed to using $Z_0 \to Y_0$ say), ensures that no derivations are possible other than the one we intend. Hence applying the above scheme to all of the non-context-sensitive productions in G_1 will yield a context-sensitive grammar G_2 with $L(G_1) = L(G_2)$. $\square$

1.5 Third Equivalent Definition of Context Sensitive Languages

A third way to define the context-sensitive languages is as the class of languages recognised by nondeterministic linear bounded automata (LBA). These automata are simply nondeterministic Turing machines that have a work tape restricted to the length of the input string or, equivalently, to a constant multiple of the length of the input. This represents a different approach to defining languages than that used above. Whereas grammars generate a language, LBAs analyse strings from Σ^* and define a language as the set of strings which have accepting runs.

The LBA construct formally represents a computer operating with limited memory. It has three components: the control unit, an input/work tape and a read/write head that scans this tape. The input, a string in Σ^*, is preloaded on the tape. The LBA operates on this input by first scanning in a symbol from the current cell (the position of the read/write head on the tape). Then, based on this input, the control unit can write a new symbol to that position on the tape or leave it unchanged. Following this it will then move the head left, right, or have it remain in place. The fact that the working space is restricted to the length of the input string means that the read/write head cannot be made to move beyond the ends of the tape. Saying that it operates nondeterministically means that it doesn't quite operate like a physical computer, rather it operates as the luckiest possible guesser. This means that at each point there may be several choices for the control unit as to the next state. If one of these choices leads to a successful result then it will choose that one, being the luckiest possible guesser.

Definition 1.5.10 A **nondeterministic linear bounded automaton (LBA)** μ is a 5-tuple (A, Q, δ, q_0, F) where the components are defined as follows:

A - The tape alphabet, defined as $\Sigma \cup \{\rhd, \lhd, \sqcup\}$. These extra symbols are the left and right end markers and empty symbol respectively. The read/write head cannot move beyond the end markers.

Q - The set of states. It is assumed that $Q \cap A = \emptyset$

$\delta : Q \times A \to \mathcal{P}(\{Q \cup F\} \times A \times \{0, L, R\})$ - The transition function. The symbols $0, L, R$ indicate that the read/write head should remain in place, move left or right respectively. If $(q', a', d) \in \delta(q, \rhd)$ then $d = R$ or $d = 0$. Similarly if $(q', a', d) \in \delta(q, \lhd)$ then $d = 0$ or $d = L$.

q_0 - The initial state, a unique element of Q.

F - The set of final states, where $Q \cap F = \emptyset$. This ensures that computation always halts when a final state is reached. $\square$

Definition 1.5.11 A **configuration** of an LBA μ is a string in $(Q \cup A)^*$ that represents the work tape, current state and the position of the read/write head. For example, μ in its initial state on input $s = a_0 a_1 \ldots a_n$ would have configuration $q_0 \rhd a_0 a_1 \ldots a_n \lhd$. Note that the tape cell currently being scanned is to the *right* of the state symbol. $\square$

Definition 1.5.12 A **computation** of μ on a string $s \in \Sigma^*$ is a sequence of configurations (also called a run) of μ; $c_0, c_1, \ldots, c_n$ such that $c_0 = q_0 \rhd s \lhd$ and if $c_i = \rhd s_1 q x s_2 \lhd$ and $c_{i+1} = \rhd s_1 x' q' s_2 \lhd$ then $(q', x', R) \in \delta(q, x)$. Similarly, the read/write head may move left or remain in place and this is reflected in

c_i, c_{i+1}. An **accepting computation** is any run on μ such that the final state q_n is in F. $\square$

Theorem 1.5.13 A language L is context-sensitive if and only if there is a linear bounded automaton μ such that $L(\mu) = L$

"$\Rightarrow$" *Given a context-sensitive grammar $G = (V, \Sigma, S, P)$ there is an LBA μ recognising $L(G)$.* What needs to be done to prove this is to construct an LBA from G that on input $s \in \Sigma^*$ reaches an accepting configuration iff $s \in L(G)$. This can be accomplished by the LBA operating as follows: On input $s \in \Sigma^*$;

1. Nondeterministically choose a production $p \in P$ where $p = \alpha \to \beta$.
2. If β is a substring of the current work tape string, s', replace β with α in s', ensuring that the string remains contiguous, otherwise leave s' unchanged.
3. Repeat 1 and 2 until the tape reads $S\sqcup^{|s|-1}$ then ACCEPT.

Clearly if there is a derivation of s using G then μ will eventually accept. Conversely for each string that μ accepts there is a derivation using G (simply apply the original productions in the reverse order to which they were guessed using μ). Since the class of languages generated by CSGs is equivalent to the class generated by length increasing grammars, it is clear that applying the inverses of length increasing productions produces sentential forms no longer than the original string. Hence this computation can be carried out in linear space.

"$\Leftarrow$" *Given a linear bounded automaton μ accepting $L(\mu)$ a grammar $G = (V, \Sigma, S, P)$ can be constructed from μ that recognises $L(\mu)$.* So given $s \in L(\mu)$ there is a derivation of s using G iff μ has an accepting computation on s. Here we should pause to note the similarities of LBAs and grammars; LBAs proceed from state to state via applications of the transition function while grammars proceed from sentential form to sentential form via applications of productions. The difference is that LBAs begin from an input string which may or may not be accepted. This is what the desired grammar must do then. We will use the symbols of G to represent the configuration of μ but also to retain the input string: each symbol in V is a pair (a, b) where $a \in \Sigma$ and $b \in T$, b is of the form a, aq_x, $q_x a$ or $q_0 \rhd a$ i.e. it may contain tape symbols but it only contains a from Σ. Now any starting configuration of μ can be represented by:

$$\begin{gathered} S \to (a, q_0 \rhd a\lhd) \\ S \to (a, q_0 \rhd a)X \\ X \to (a, a) \\ X \to (a, a\lhd) \end{gathered}$$

There are duplicate productions for every other $a \in \Sigma$ too. Next we must encode the transition function δ of μ in the productions of G:

- If $(q', b', R) \in \delta(q, b)$ then for every $a \in \Sigma$ add production $(a, qb) \to (a, b'q')$.
- If $(q', b', L) \in \delta(q, b)$ then for every $a, a_n \in \Sigma$ and $b_n \in A$ add production $(a_n, b_n)(a, qb) \to (a_n, bq')(a, b')$.

- If $(q', b', 0) \in \delta(q, b)$ then for every $a \in \Sigma$ add production $(a, qb) \to (a, q'b')$.

Finally we need productions that will recover the original string. Here we make the assumption that μ only enters a final state when the read head has moved fully to the left. Then we can include rules for every $a, a_n \in A$ and $q \in F$:

$$\begin{gathered}(a, q\rhd a\lhd) \to a \\ (a, q\rhd a) \to a \\ a(a_n, x_n) \to aa_n\end{gathered}$$

Now if $c = c_0, c_1, \ldots, c_m$ is an accepting computation of μ on $s = s_0s_1 \ldots s_n$ we must show that s can be derived by G. $c_0 = q_0 \rhd s\lhd$ is represented by the sentential form $(s_0, q \rhd s_0)(s_1, s_1) \ldots (s_n, s_n\lhd)$ which can be obtained using the first set of productions defined above. Each configuration c_i is preceded by a configuration c_{i-1} such that c_i is reached by a single application of a member of the transition function applied to the cell and state in c_{i-1}. Since the second set of productions defines productions for every possible application of the transition function, the sentential form of c_i can be derived from c_{i-1} in G. And lastly $c_m = q \rhd x_0x_1 \ldots x_n\lhd$ with sentential form $(s_0, q \rhd x_0)(s_1, x_1) \ldots (s_n, x_n)$. Using the final set of productions above, the string s can be extracted.
Conversely if $S, S_0, S_1, \ldots, S_n$ is a sequence of sentential forms of G where $s := S_n$ we must show that there is an accepting computation of s on μ. Given that $|s| = l$ then S_{l-1} encodes c_0. From there each S_i must follow by the application of a rule from the second production set. These strictly correspond to valid applications of the transition function, meaning $c_{i-(l-1)}$ entails $c_{i-(l-1)+1}$. G can only produce a terminal string if it first produces a sentential form encoding a final state configuration of μ. Hence if G produces s there is an accepting computation of s on μ. $\square$

Note that the above equivalence is only true for LBAs, not for Turing machines in general (which do not have the linear bound condition). This is because if the machine has a work tape that is longer than the input string (or a constant multiple of the length as we shall see later) then there may be configurations of the machine that are arbitrarily longer than the input string. But then when the equivalent grammar produces a terminal string (the original input string) it will be shorter than the configuration; hence the grammar is no longer length increasing.

Example 1.5.14 $\{a^p | p \text{ is prime}\}$ is a context-sensitive language. It would be difficult to find a grammar that generates this language, but it is simple to find an LBA to recognise it. Consider the LBA that on input $s = a^n$ divides n by all natural numbers m where $1 \leq m \leq n - 1$. This machine accepts only when no m divides n, otherwise it rejects after trying all $m < n$. $\square$

2 Properties of Context-Sensitive Languages

2.1 Introduction

Given any new class of sets we usually want to find out how it behaves under standard set operations. To have a useful mathematical object it is desirable that the operations of intersection, union and composition do not form new objects outside the definition of the class. It turns out that the class of context-sensitive language is closed under all of the above operations; almost all are easy to prove. Closure under complementation however, was not proven until 1987 when it was proven by both N. Immerman and R. Szelepcsényi independently[2][6].

2.2 Basic closure theorems

The first three operations considered; union, concatenation and Kleene closure, are in fact closure operations for all formal languages. These are called the regular operations.

Theorem 2.2.0 The class of context-sensitive languages is closed with respect to concatenation.

Let L_1 and L_2 be context-sensitive languages generated by $G_1 = (V_1, \Sigma, S_1, P_1)$ and $G_2 = (V_2, \Sigma, S_2, P_2)$ respectively. We can assume, without loss of generality, that $V_1 \cap V_2 = \emptyset$. Now define $G = (A_1 \cup A_2, \Sigma, S, P_1 \cup P_2 \cup \{S \to S_1S_2\})$, where $S \notin A_1 \cup A_2$. The new rule $S \to S_1S_2$ is context-sensitive as are the rules in P_1 and P_2 by definition, hence G is context-sensitive. Claim that $L(G) = L_1L_2$.
If $s \in L_1L_2$ then $s = s_1s_2$ where $s_1 \in S_1$ and $s_2 \in S_2$. A derivation of s using G uses $S \to S_1S_2$. There are derivations for s_1 and s_2 from S_1 and S_2 respectively and since $V_1 \cap V_2 = \emptyset$ applying these derivations to S_1S_2 we must get s. Conversely, if $s \in L(G)$ then the rule $S \to S_1S_2$ along with the assumption $V_1 \cap V_2 = \emptyset$ ensures that $s = s_1s_2$ where $s_1 \in L_1$ and $s_2 \in L_2$. $\square$

Theorem 2.2.1 Given two context-sensitive languages L_1 and L_2, then $L_1 \cup L_2$ is a context-sensitive language also.

The idea behind this proof is simple; take two LBAs μ_1 and μ_2 that recognise L_1 and L_2, respectively and construct a new machine μ_{cup} to recognise the union. Define a new start state and link it to the start states of the original machines. Then μ_{cup} will non-deterministically choose a machine to execute on a given input and accept only if the computation eventually reaches a final state in either of the LBAs.
Formally, if $\mu_1 = (A_1, Q_1, \delta_1, q_{(1,0)}, F_1)$ and $\mu_2 = (A_2, Q_2, \delta_2, q_{(2,0)}, F_2)$ then let $\mu_\cup = (A_1 \cup A_2, Q_1 \cup Q_2 \cup \{q_0\}, \delta', q_0, F_0 \cup F_1)$, where δ' is defined as follows:

$$\delta'(q_0, \rhd) = \{(q_{(1,0)}, \rhd, 0), (q_{(2,0)}, \rhd, 0)\}$$
$$\delta'(q, a) = \delta_1(q, a), \text{ if } a \in A_1 \text{ and } q \in Q_1$$
$$\delta'(q, a) = \delta_2(q, a), \text{ if } a \in A_2 \text{ and } q \in Q_2$$

If $s \in L_1$ then s has an accepting computation on $\mu_\cup$, namely from q_0 to $q_{(1,0)}$ from which it can reach a final state by assumption. Similarly, $s \in L_2$ has an accepting computation on $\mu_\cup$. Conversely, if s' is accepted by $\mu_\cup$ it must have an accepting computation on either μ_1 or μ_2. This can be found simply by removing the first configuration, $q_0 \rhd s' \lhd$, of the accepting computation. □

The proof of closure under the Kleene star operation requires care since the Kleene star is a unary operator. We make use of the assumption that the productions of a context-sensitive grammar can be written with terminal symbols only occurring on the right hand side of productions. An example of this is *Kuroda Normal form* described by Kuroda in [3]. The proof that every context-sensitive language has a Kuroda Normal form grammar is omitted here as it is a largely technical formalism.

Theorem 2.2.2 The set of context-sensitive languages is closed under the Kleene star operation.

Let $L := L(G)$ and $G = (V, \Sigma, S, P)$ such that L a context-sensitive language that does not contain λ. Let $L^* := \{s_1 s_2 \dots s_n : s_i \in L \text{ and } n \in \mathbb{N}\}$ be the Kleene closure of L. Define a new grammar $G^* = (V \cup \{S_1, S_2\}, \Sigma, S_1, P^*)$, assuming that $\{S_1, S_2\} \cap V = \emptyset$. The augmented set of productions of G^* is defined as:

$$P^* = P \cup \{S_1 \to \lambda, S_1 \to S, S_1 \to S_2 S\} \cup \{S_2 a \to S_2 S a, S_2 a \to S a : a \in \Sigma\}$$

The two extra variables are required because the of fact that $s^0 = \lambda$ for $s \in L$ means that we need a rule $S_1 \to \lambda$. But then, by definition, we cannot have S_1 on the right side of any other rule. The second new set of productions added above allows us to derive strings of the form $s_1 s_2 \dots s_n$ by inserting a new S on the left at any point in the derivation, eg. from $S_2 s_{n-i} \dots s_n$ we obtain $S_2 S s_{n-i} \dots s_n$ and from the new S we may derive a new string in L. This is why we require that L does not contain the empty string, otherwise G^* would not be length increasing. It is fairly clear that using these productions we can derive all strings in L^* using G^*. Now we must show that these are the only strings that are in $L(G^*)$. Suppose x is derived by G. There are only three possible ways for this to occur:

1. By an application of $S_1 \to \lambda$, so $x = \lambda \in L^*$.
2. By an application of $S_1 \to S$ then a derivation using G. Then $x \in L \subseteq L^*$.
3. By an application of $S_1 \to S_2 S$ followed by zero or more applications of $S_2 a \to S_2 S a$, for some $a \in \Sigma$. Then one application of $S2b \to Sb$, for some $b \in \Sigma$. The result is a sentential form of G^*, $X_1 X_2 \dots X_n$ where each X_i is a sentential form of G that has the form $a\alpha_1\alpha_2 \dots \alpha_n$ for $a \in \Sigma$ and $\alpha_i \in \Sigma \cup V$. This is because of our assumption that terminal symbols only occur on the left of productions of G and that $\{S_1, S_2\} \cap V = \emptyset$. This also means that from $X_1 X_2 \dots X_n$ we can only derive the string $s_1 s_2 \dots s_n$ with each $s_i \in L$.

Hence $L(G^*) = L^*$. Now suppose L contains λ. Note that $L - \{\lambda\}$ is context-sensitive; we can simply delete the rule $S \to \lambda$, since S is not on the right side of any other productions. Then form $(L - \{\lambda\})^*$. This is context-sensitive and equal to L^*. $\square$

The proof of closure under intersection is slightly more complex than for union. This is because for a string to be in $L_1 \cap L_2$ it must have an accepting computation on both μ_1 and μ_2. We must check first that μ_1 accepts the string then that μ_2 accepts. But this means that the string must be delivered unchanged to μ_2 after μ_1 has performed its accepting computation on the same tape (which is restricted to the length of the string). To do this we make an LBA with work alphabet consisting of pairs of symbols from μ_1 and μ_2 then have copies of the machines operate on two separate copies of the input string.

Theorem 2.2.3 Given two context-sensitive languages L_1 and L_2, then $L_1 \cap L_2$ is a context-sensitive language also.

The idea in this case is to have the tape store pairs of symbols ie. the tape alphabet is $A_1 \times A_2$. A modified version of μ_1 will operate on the first symbol in the pair while a version of μ_2 will operate on the second. Since each pair in $A_1 \times A_2$ occupies only one tape cell and both μ_1 and μ_2 are LBAs, the new machine is an LBA also. Let $\mu_1 = (A_1, Q_1, \delta_1, q_{(1,0)}, F_1)$ and $\mu_2 = (A_2, Q_2, \delta_2, q_{(2,0)}, F_2)$ be LBAs recognising L_1 and L_2 respectively. Now define $\mu_\cap$ as the LBA that operates as follows:

1. On input $s = s_0 s_1 ... s_n \in A_1 \cap A_2$, set the tape to $(s_0, s_0)(s_1, s_1)...(s_n, s_n)$.
2. Run μ_1 on the first symbols in the tape pairs.
3. If μ_1 accepts, run μ_2 on the unmodified second symbols in the tape pairs.
4. If μ_2 accepts then $\mu_\cap$ accepts.

Then $\mu_\cap$ accepts s if and only if $s \in L_1 \cap L_2$. $\square$

2.3 Immerman-Szelepcsényi theorem

The Immerman-Szelepcsényi theorem demonstrates that given an LBA μ recognising a certain context-sensitive language we can construct an LBA $\overline{\mu}$ that recognises all the strings not recognised by μ. The theorem originally proven by Immerman and Szelepcsényi is a lot more general; it shows that every language that is accepted by a Turing machine operating within *some* space constructable bound has a machine recognising its complement that operates in the same bound. Clearly LBAs are just a special case of this. We discuss more about the theory of space complexity after the proof, but first we introduce a crucial tool; the multitape LBA.

Definition 2.3.4 A **multi-tape LBA** is an LBA that uses several independent read/write heads, each with their own tapes of length equal to the size of the input. A k-tape LBA is defined as $\mu^k = (A, Q, \delta^k, q_0, F)$ where $\delta^k : Q \times A^k \to Q \times (A \times \{L, R, 0\})$ is the transition function. $\square$

Multi-tape machines can be simulated on single-tape machines by using k-tuples as the work tape alphabet, as in the proof of closure under intersection.

We also need special markers to keep track of the positions of the read/write heads from each tape. What is important though, is that this new machine has a work tape with the same length as the input string (as each of the individual tapes are this long). In a similar way we can see that a single tape machine with work tape restricted to at most a linear multiple of the input can be represented as an LBA. These notions allow us to simplify our representation of LBAs in proofs. If we have an algorithm that runs on a Turing machine and only uses finitely many variables, each of which can be represented in space linear in terms of the input, then we know it is able to run on an LBA. This fact is used in the following proof.

Theorem 2.3.5 The context-sensitive languages are closed under complementation.

Given a context-sensitive language $L \subseteq \Sigma^*$ with corresponding LBA $\mu = (A, Q, q_0, \delta, F)$, we wish to construct an LBA $\overline{\mu}$ which accepts a string s only if μ does not accept s. Then $L(\overline{\mu}) = \Sigma^* - L$. At first glance, simply simulating the computation of s on μ and rejecting if μ accepts seems a promising approach. However $\overline{\mu}$ must accept only if *every* possible computation of s is not accepting. Hence $\overline{\mu}$ must do the following:

1. Determine how many configurations are reachable given input s.

2. Check that for each of these μ is not in an accepting state.

Let C_i be the set of configurations of μ reachable in no more than i steps and let $m = |Q|(|s| + 2)|A|^{|s|}$, this is the total number of possible configurations of μ on s. The first task then, is equivalent to finding $|C_m|$. This can be accomplished by performing a nondeterministic breadth-first search of all the possible computations of μ on s. However we are restricted by the fact that $\overline{\mu}$ must operate within a linear bound, so we cannot store every current configuration for each possible branch of the computation. Instead, at each iteration of the algorithm we nondeterministically select the configurations from the previous iteration then count the configurations which can be reached in one further step of μ. The following is a psuedocode representation of this algorithm:

```
NBreadthFirstSearch($s$):-
   Set $CountOverall = 1$.
   For $i = 0$ to $(m - 1)$:
      Set $Count_{i+1} = 0$.
      Set $ConfigsChecked = 0$.
      For each $c_a \neq c_0$ of $\mu$:
         For each $c_b$ of $\mu$ non-deterministically choose to execute or skip the following:
            Simulate $\mu$ on input $s$.
            -REJECT if after $i$ steps $\mu$ has not reached $c_b$.
            Increment $ConfigsChecked$.
            If $c_a$ follows from $c_b$ or $c_a = c_b$ then increment $Count_{i+1}$.
                Then Goto next $c_a$.
         End For
      End For
```

```
        If $ConfigsChecked < CountOverall$ then REJECT.
        Set $CountOverall = Count_{i+1}$.
    End For
    RETURN $CountOverall$.
```

To show the correctness of this algorithm we prove by induction that for each i the value of *CountOverall* on that iteration is $|C_i|$. Use for the base case $|C_0| = 1$; at this point *CountOverall* has been initialised to 1 and so is correct. Now suppose $CountOverall = |C_p|$ for some $p \in \{0, ..m-2\}$. The $(p+1)$th iteration of the outer for-loop can succeed only if the following are true:

- Each c_b chosen to execute on must be the result of a computation of length at most i.
- Each c_a selected must follow immediately from c_b or $c_a = c_b$.
- *ConfigsChecked* must be equal to *CountOverall*, the number of configurations reachable in i steps.

Taken together these conditions imply that a configuration increments the $Count_{i+1}$ variable only if it is reachable by a computation of length at most $i+1$ and each time this occurs $|C_i| = CountOverall$ configurations must have already been checked, as recorded by *ConfigsChecked*. If the above algorithm is to run on an LBA, each of the variables must be able to be stored in space directly proportional to the input length, $|s|$. Now the maximum value of any of these variables is the number of possible states, $m = |Q|(|s|+2)|A|^{|s|}$. This value has a binary representation of length $\log_2 m = |s| \log_2 |A| + \log_2(|s|+2) + \log_2 |Q| \leq c|s|$ for some $c \in \mathbb{N}$. So an LBA implementing this algorithm will need at least seven tapes, six for the variables ($Count_{i+1}$, *CountOverall*, etc.) and one for the work tape of μ used to simulate it.

Now that we have the number of reachable configurations we can check for final states. If μ can guess $|C_m|$ distinct configurations all of which are reachable and non-accepting, then $|s|$ is in the complement of $L(\mu)$ and so $\overline{\mu}$ should accept. An algorithm for this follows:

```
Check($|C_m|$):-
    Set $Count = 0$
    For each $c \neq c_0$ of $\mu$ non-deterministically choose to execute or skip the following:
        If $c$ is a final state of $\mu$ REJECT.
        Simulate $\mu$ on $s$.
        If after $m$ steps $c$ has not been reached REJECT.
        Increment $Count$.
    If $Count = |C_m|$ ACCEPT.
```

As with the first algorithm the variables *Count*, c, $|C_m|$ and the simulation tape can all be represented on an LBAs linear bounded work tape. Since both algorithms can be carried out by an LBA the desired result follows from the equivalence of LBAs and context-sensitive languages. □

2.4 Space Complexity

The flexibility of the concept of multi-tape LBAs illustrates the fact that LBAs (equivalently CSGs and context-sensitive languages) represent an entire class of computations of Turing machines: those that can be carried out in space at most a linear multiple of the input. In addition we have seen that this class has several desirable closure properties. This leads us to define the notion of space complexity; loosely this is a classification of Turing machine algorithms by the maximum size of the work tape needed to compute them.

Definition 2.4.6 The **Space complexity** of a nondeterministic Turning machine μ is a function $f : \mathbb{N} \to \mathbb{N}$ such that $f(n)$ is the maximum number of tape cells scanned by the read/write head of μ in any branch of computation on input of length n. $\square$

In practice it is difficult to find the specific space complexity of a given algorithm. However we can still form a useful framework for space complexity by estimating the degree of scalability of the complexity function relative to its input.

Definition 2.4.7 Let $f : \mathbb{N} \to \mathbb{N}$. $O(f(n)) = \{g : \mathbb{N} \to \mathbb{N} |$ if $g(n) \leq Cf(n)$ for every $n \in \mathbb{N}$ and where C is some constant$\}$.
This is called **asymptotic** or **Big-O notation** $\square$

Definition 2.4.8 NSPACE$(f(n)) = \{L = L(\mu) | \mu$ is a non-deterministic TM with space complexity $g(n) \in O(f(n))\}$. SPACE$(f(n))$ is the corresponding class for deterministic TMs. $\square$

Using these two definitions we see that all algorithms that run on LBAs are in NSPACE(n) in other words, the problem of finding whether a given context-sensitive grammar derives a certain string is in NSPACE(n). This illustrates the close ties between complexity theory and formal language theory. As an interesting aside, there remains an open problem to do with LBAs in the realm of complexity theory: does a deterministic LBA have the same power as a normal non-deterministic LBA or, equivalently, does NSPACE(n) = SPACE(n). This problem was first posed, along with the complementation problem, by Kuroda in his paper detailing the equivalence of LBAs and context-sensitive languages. We know from *Savitch's theorem* that NSPACE$(f(n)) \subseteq$ SPACE$(f^2(n))$ and by *space hierarchy* theorems that SPACE$(n) \subsetneq$ SPACE(n^2) hence the best we can say with current knowledge is that SPACE$(n) \subseteq$ NSPACE(n).

2.5 Computability

Certain problems to do with classes of languages have the desirable property that they are able to be *decided* by a Turing machine. Decided is used in a technical sense to mean the Turing machine must halt on both accepting and rejecting input. Problems which are decidable are quite often able to be implemented on real life computers, however it must be noted that decidability says nothing about *efficiency* of computation. That is the realm of complexity theory.

Definition 2.5.9 Given a language $L \subseteq \Sigma^*$ (normally an encoding of a problem or function) a **decider** for L is a Turing machine T that for every $s \in \Sigma^*$, T accepts if $s \in L$ or halts after finitely many steps. In the non-deterministic case this means that every possible computation branch has finite length. L is then said to be **decidable**. If L does not have a decider it is **undecidable**. If T accepts when $s \in L$ but does not necessarily halt then it is a **recogniser** for L. □

The first problem we will look at is called the acceptance problem for context-sensitive languages. This is the most fundamental problem for a language: given a string $s \in \Sigma^*$ and some description of the language eg. a grammar, is s in the language? In fact this problem is decidable for all of the languages in the Chomsky hierarchy except the recursively enumerable languages.

Theorem 2.5.10 The acceptance problem for context-sensitive languages is decidable.

Given $\mu = (Q, A, \delta, q_0, F)$ we know that if $s \in L(\mu)$ then μ will halt in an accepting state by definition. Instead suppose $s \notin L(\mu)$. As noted in our proof of the Immerman-Szelépscenyi theorem there are $m = |Q|(|s|+2)|A|^{|s|}$ possible configurations of μ on s. We can use this as an upper bound on the length of a computation of μ since, by the pigeon-hole principle, if μ runs for more than m steps on s without halting then there must be at least one repeated configuration. So we know that by this stage μ must have been through every possible reachable configuration on s and so we deduce that s cannot be in $L(\mu)$. Therefore every LBA μ is a decider for $L(\mu)$ as it must either reach an accepting state or reject after m steps. □

Next we highlight a computational facet of the increase in complexity of context-sensitive languages over context-free languages in the emptiness problem. This asks, if given a description of a language, does it contain any strings? Immediately it can be seen that this problem is harder than the acceptance problem as a positive answer asserts that *all* strings in Σ^* are not in the language. However with context-free languages we can use the simple structure of the grammar to our advantage.

Theorem 2.5.11 The emptiness problem for context-free languages is decidable.

Recall that a context-free grammar is one in which each production is of the form $A \to \gamma$ where A is a variable and γ is any combination of variables and terminals. Given a string representation of a context-free grammar, first mark the terminal symbols. Then scan the right hand side of all productions for terminal only forms. If they exist, mark the corresponding left hand side variables. Now repeat but searching for marked symbols on the right this time. Repeat this until either S is marked or no new symbols are marked. Since there are only a finite number of symbols and productions this algorithm must halt in either case. □

The proof of the undecidability of the emptiness problem for context-sensitive languages makes use of the fact that the acceptance problem (sometimes called the halting problem) for Turing machines is undecidable. We form a reduction from the emptiness problem to the halting problem, showing that a decider for the emptiness problem can be used to build a decider for the halting problem.

Theorem 2.5.12 The emptiness problem for context-sensitive languages is undecidable.

Suppose for contradiction that the emptiness problem for context-sensitive languages has a decider D. So D takes input $< \mu >$, an encoding of an LBA μ, and accepts if μ does not accept any strings. We must now show that this can be used to construct a decider for the halting problem. Given a Turing machine M and input string w we may construct an LBA μ that accepts the language consisting of all accepting computations of M operating on w. For a fixed $M = (Q, A, \delta, q_0, F)$ and w, let μ operate as follows:
On input $c_0, c_1, \ldots, c_n$:

1. Check that $c_0 = q_0 w$, (w is built into μ).
2. For each of $c_1, \ldots, c_n$:
3. Check that c_{i+1} follows from c_i according to the transition function δ.
4. Check that the state in c_n is a final state of M.

This may be carried out by an LBA as it involves only reading from the work tape, no extra information need be stored.
Now if μ is not empty then there is an accepting computation of w on M, so we may define a decider for the halting problem that works by constructing μ as above, then feeds the definition of μ into the decider for the emptiness problem, D. If D accepts this input then REJECT as this means the language of μ is empty; M has no accepting computation on w. If D rejects; which it must if $< \mu >$ is not empty, then ACCEPT because there is an accepting computation of Mo on w. Hence we have a decider for the halting problem; contradiction. □

2.6 Conclusion

We have seen that far from being just an arbitrary subdivision of the set of all possible grammars, context-sensitive languages turn up in many other fields from formal language theory to complexity theory. There are three equivalent definitions of context-sensitive languages: as languages generated by context-sensitive grammars; as the class of non-contracting grammars and as the languages recognised by linear bounded automata. This last and by far most important equivalence made the connection between context-sensitive grammars and the class of non-deterministic linear space algorithms and illustrated the expressive power of context-sensitive languages.

Next we used properties of LBA and grammars to prove the closure of the class of context-sensitive languages under the union, intersection, concatenation, complementation and Kleene star operations; thus showing it is a useful mathematical object worthy of study. The Immerman-Szelepcsényi proof of closure under complementation illustrates how results in complexity theory can be applied to context-sensitive languages via the LBA equivalence. Then an important open problem in complexity theory was posed- does NSPACE(n) = SPACE(n)?

3 References and Acknowledgements

This text follows closely the development in Simovici and Tenney[4], chapter nine; significantly in the proof of the LBA/context-sensitive languages equivalence. Parts are also due to Sipser[5], namely the exposition of the Immerman-Szelepcsényi theorem and the emptiness and acceptance problems for LBAs. And finally, thanks must go to my supervisor Andre Nies for his help with this project.

1. CHOMSKY, Noam. *Three models for the description of language.* IRE Transactions on Information Theory (2), 1956, pp.113-124.

2. IMMERMAN, Neil. *Nondeterministic Space is Closed Under Complementation.* SIAM Journal on Computing 17, 1988, pp. 935-938.

3. KURODA, Sige-Yuki. *Classes of Languages and Linear-Bounded Automata.* Information and Control, 7(2), 1964, pp. 207–223.

4. SIMOVICI, Dan A., TENNEY, Richard L. *Theory of Formal Languages With Applications.* World Scientific Publishing Co., 1999.

5. SIPSER, Michael. *Introduction to the Theory of Computation, Second Edition.* Course Technology, 2006.

6. SZELEPCSÉNYI, Róbert . *The Method of Forcing for Nondeterministic Automata.* Bulletin of the EATCS 33, 1987, pp. 96-100.