ECS 120 Lesson 14 – Non-Context-Free Languages, the Context-Free Pumping Lemma

Oliver Kreylos

Monday, April 30th, 2001

In this lecture, we will close our discussion of context-free languages by introducing a version of the Pumping Lemma for context-free languages. It will allow us to prove that certain languages are not context-free, meaning they can neither be generated by context-free grammars, nor can they be accepted by pushdown automata. Here are some examples of non-context free languages:

- $L_1 = \left\{ \texttt{0}^n\texttt{1}^n\texttt{0}^n \;\middle|\; n \geq 0 \right\}$
- $L_2 = \left\{ ww \;\middle|\; w \in \Sigma^* \right\}$
- $L_3 = \left\{ w \in \text{ASCII}^* \;\middle|\; w \text{ is a syntactically correct } \texttt{C} \text{ program} \right\}$

1 The Context-Free Pumping Lemma

For regular languages, the one observation that led us to the discovery of the Pumping Lemma was that every regular language is accepted by a finite state machine, and that every finite state machine A has a finite number p of states. From this we were able to conclude that the computation chain for any word in the language accepted by A that has at least p characters must include one of A's states twice. This leads to a loop in the computation chain, that can be taken by the machine any number of times, without the machine being able to notice this. Therefore, any word resulting from taking the loop any number of times must be in the language as well.

For context-free languages, we can observe a similar fact. In this case, however, our reasoning will be based on context-free grammars, not on push-down automata. Since the latter have an arbitrarily large stack, reasoning in a similar way as for finite automata would not work. Instead, we concentrate on the fact that each grammar can only have a finite number of variables.

Let us consider a grammar $G = (V, \Sigma, R, S)$ in Chomsky Normal Form. We know that every parse tree for any word $w \in L(G)$ in the grammar's language is a binary tree: Each rule in a CNF grammar is either of the form $A \to a$ or $A \to BC$. We will now try to find out more about the structure of those parse trees.

One important property when reasoning about trees is their *depth*. The depth of a tree is defined recursively as follows:

1. The depth $d(T)$ of a tree T consisting only of a single node (the root node) is defined to be zero.

2. If a tree T is not only a single node, it is a root node having children $T_1, T_2, \ldots, T_k$, where each child T_i is again a tree. The depth $d(T)$ of T is defined as $d(T) := 1 + \max\{\, d(T_i) \mid i = 1, 2, \ldots, k \,\}$.

We now ask how the length of a word $w \in L(G)$ and the depth of its parse trees are related. We know that the characters of w are associated with leaves of its parse tree, so we have to find out how the depth of a binary tree relates to its maximum number of leaves. We claim that every binary tree T of depth $d(T)$ has at most $\#\text{leaves}(T) \leq 2^{d(T)}$ leaves. We prove this claim by induction on the depth $d(T)$.

Induction Basis A tree T of depth $d(T) = 0$ is a single node. It has exactly one leaf, and $\#\text{leaves}(T) = 1 = 2^0 = 2^{d(T)}$.

Induction Hypothesis Assume the claim is true for all binary trees of depth not larger than n.

Induction Step Let T be a tree of depth $n + 1$. Then T consists of a root node with two children T_1 and T_2, each binary trees of depth not larger than n. From the induction hypothesis, we know that the number of leaves of T_1 is at most $\#\text{leaves}(T_1) \leq 2^{d(T_1)} \leq 2^n$, and the number of leaves of T_2 is at most $\#\text{leaves}(T) \leq 2^{d(T_2)} \leq 2^n$. The total number of leaves of T is the sum of the two numbers: $\#\text{leaves}(T) = \#\text{leaves}(T_1) + \#\text{leaves}(T_2) \leq 2^n + 2^n = 2^{n+1}$.

Let us now assume that G has exactly $|V| = n$ variables, and that $w \in L(G)$ is a word of length $|w| = 2^{n+1}$. From the last theorem we know that any parse tree for w must have a depth of at least $n + 1$ – any parse tree of height less than $n + 1$ must have less than 2^{n+1} leaves. This means that any parse tree for w has at least one branch of length $n + 1$ from the root, which means it has $n + 1$ variables $S = A_0, A_1, \ldots, A_n$ written along it[1], see Figure 1.

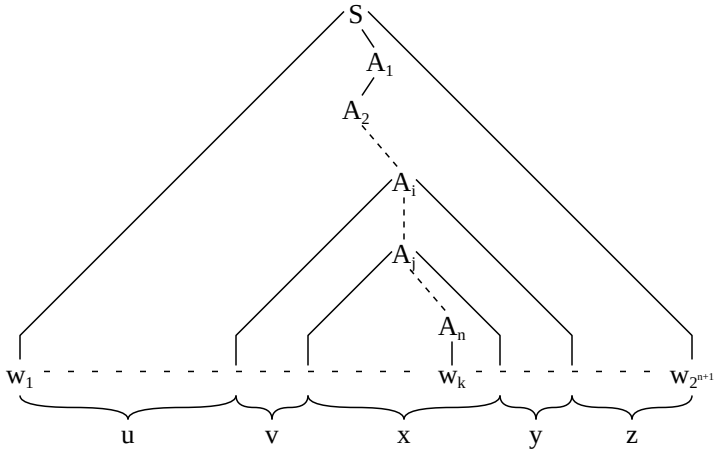

Figure 1: A parse tree of height $n + 1$ must contain at least one branch of length $n + 1$, with $n + 1$ variables labeling its nodes. The last node in the branch is a leaf labeled with a character, not a variable.

From the Pigeonhole Principle follows, that at least two of the $n+1$ variables along the branch must be identical, because G contains only n variables. Let us assume that $A_i = A_j = B$, for $0 \leq i < j \leq n$, are two occurrences of the same variable B in the branch of length $n + 1$. Splitting the branch in the parse tree at the nodes labeled with A_i and A_j introduces a split in the generated word w into five parts $w = uvxyz$. Let us denote the complete parse tree as T, the subtree underneath the node labeled A_i as T_i and the subtree underneath the node labeled A_j as T_j. Then the characters of u are the leaves of T left of all nodes in T_i, the characters of v are the leaves of T_i left of all nodes in T_j, the characters of x are the leaves of T_j, the characters of y are the leaves of T_i right of all nodes in T_j, and the characters of z are the leaves of T right of all nodes in T_i.

[1]The last node on the branch is a leaf labeled with a character, not a variable.

From our understanding of parse trees, we know that the string vxy can be generated from symbol A_i, and that the string x can be generated from A_j. Since both are the same variable B, both $B \Rightarrow^* vxy$ and $B \Rightarrow^* x$. This means, we can generate other valid parse trees by performing "parse tree surgery" on the parse tree T. The node A_i corresponds to an occurrence of the variable $A_i = B$ in some intermediate string in the derivation of w. Instead of deriving as dictated by parse tree T, we can directly replace that occurrence by the string x that can be generated from B as well. This would derive the word $w^{(0)} = uxz$ and correspond to "removing" the part of the parse tree that is generated from nodes between A_i and A_j, see Figure 2. On the other hand, the node A_j correponds to an occurrence of the variable $A_j = B$ in some other intermediate string in the derivation of w. Instead of deriving as in T, we can replace that occurrence with vxy. This would generate the word $w^{(2)} = uvvxyyz$ and "duplicate" the part of the parse tree generated from nodes between A_i and A_j. More generally, all parse trees generated by duplicating the partial branch between A_i and A_j any number of times would still generate a valid parse tree in G. Therefore, all the words $w^{(i)} = uv^ixy^iz$ generated in this way would also be in the language $L(G)$. This observation, somewhat similar to the observation that led us to the Pumping Lemma, is the basic claim of the Context-Free Pumping Lemma we are going to state.

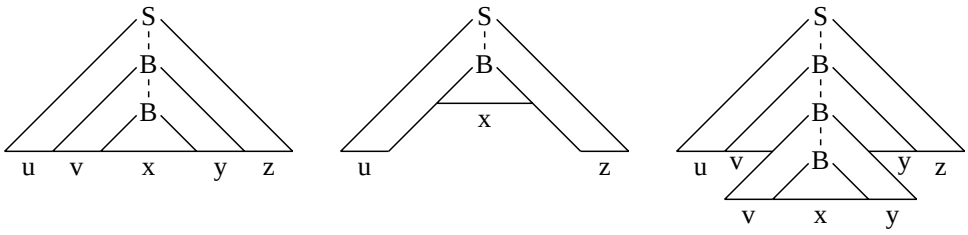

Figure 2: Performing "surgery" on parse trees. Left: The parse tree for some long word $w = uvxyz$. Middle: The parse tree obtained by "removing" the part between the two occurrences of B. Right: The parse tree obtained by "duplicating" the part between the two occurrences of B.

Next we make two observations about the lengths of vy and x. Since $i < j$, either v or y must contain at least one character. In the worst case, $j = i+1$. Then A_j is a child of A_i, but since both are variables, the rule replacing A_i by A_j must be either $A_i \to A_jC$ or $A_i \to CA_j$ for some variable C. C

must generate at least one character[2], therefore the subtree T_i must have at least one leaf either on the left or on the right of all nodes in T_j. We can write this property concisely as $|vy| \geq 1$. On the other hand, the string vxy generated by the subtree T_i can contain at most 2^{n+1} characters, because its depth is at most $d(T_i) \leq n+1$. If any word $w \in L(G)$ is generated by a parse tree T of height more than $d(T) > n+1$, we still know that the height of the subtree T_i is not more than $d(T_i) \geq n+1$. If there is a branch in the tree longer than $n+1$, the first double occurrence of a variable must appear in the first $n+1$ variables counted from the bottom, according to the Pigeonhole Principle.

We can now combine these observations and the fact that every context-free language is generated by a grammar in Chomsky Normal Form, and state

Lemma 1 (Context-Free Pumping Lemma) *Let $L \subset \Sigma^*$ be a context-free language over the alphabet Σ. Then there exists a* pumping length *$p \geq 1$, such that every word $w \in L$, with $|w| \geq p$, can be divided into five parts $w = uvxyz$, satisfying the following three conditions:*

1. *$|vy| \geq 1$,*
2. *$|vxy| \leq p$, and*
3. *$\forall i \geq 0 : uv^ixy^iz \in L$.*

2 Using the Context-Free Pumping Lemma

The Pumping Lemma for context-free languages is used in the same way as the Pumping Lemma for regular languages. A language is proven to be non-context-free by contradiction, using the following steps:

1. Assume that the language is context-free.
2. If the language is context-free, the Context-Free Pumping Lemma applies to it. This means, there must be a pumping length p.
3. Pick any word w from the language that is at least p characters long.

[2]Only the start symbol may derive to ϵ, and the start symbol cannot appear in the interior of a tree.

4. For any possible way of splitting w into parts $w = uvxyz$ that satisfy the constraints $|vy| \geq 1$ and $|vxy| \leq p$, show that at least one word $uv^i xy^i z$ is not in the language.

5. If this is the case, we have a contradiction with the assumption that the language was context-free. Therefore it is not context-free.

2.1 The Language $L_1 = \left\{ \mathtt{0}^n\mathtt{1}^n\mathtt{0}^n \;\middle|\; n \geq 0 \right\}$

Let us use the Context-Free Pumping Lemma to show that L_1 is not context-free. We assume it is, giving us a pumping length p. We now consider the word $w = \mathtt{0}^p\mathtt{1}^p\mathtt{0}^p$ (the usual suspect). In any way of splitting it into parts $w = uvxyz$ within the limits set by the Pumping Lemma, the string vxy cannot contain zeros from both the first batch and the second batch, since its total length is at most $|vxy| \leq p$. Let us assume it only contains zeros from the first batch and ones (the other case is symmetrical). Then, upon pumping w once, the number of either zeros in the first batch or ones must increase, since $|vy| \geq 1$. But the number of zeros in the second batch stays the same, therefore the pumped word cannot be in L_1. This contradicts the Pumping Lemma and hence our assumption that L_1 is context-free.

2.1.1 The Language $L_2 = \left\{ ww \;\middle|\; w \in \Sigma^* \right\}$

Again, we follow the trodden path of using the Pumping Lemma. The tricky part here is cleverly choosing word w, such that the contradiction can be constructed easily. We choose the word $ww = \mathtt{0}^p\mathtt{1}^p\mathtt{0}^p\mathtt{1}^p$. Since $|vxy| \leq p$, the string vxy can straddle at most two adjacent batches of zeros and ones. We have to consider all possible cases of splitting w:

1. vxy is part of the first copy of w. Let us assume that $|vy| = k$, where $1 \leq k \leq p$. Upon pumping once, the word $(ww)^{(2)} = \mathtt{0}^p\{\mathtt{0},\mathtt{1}\}^k\mathtt{1}^p\mathtt{0}^p\mathtt{1}^p$ is not in the language anymore – since $k \leq p$, the middle of $(ww)^{(2)}$ must occur somewhere in the first batch of ones. This means, the second half of $(ww)^{(2)}$ must begin with a one, but the first half still begins with a zero.

2. If vxy is part of the second copy of w, symmetric reasoning tells us that pumping once removes the word from the language.

3. vxy straddles the middle of the word. Since $|vy| \geq 1$, any pumping must change the second half of the first copy of w, or the first half of the second copy of w. Since neither the first half of the first copy of w nor the second half of the second copy of w change, the words $(ww)^{(i)}$ cannot consist of two identical halves w' anymore. Therefore, $(ww)^{(i)}$ cannot be in the language.

This contradicts the Pumping Lemma and hence our assumption that L_2 is context-free.

2.2 A Corollary to the Context-Free Pumping Lemma

Before we can prove that the language of correct `C` programs is not context-free, we have to make another observation about parse trees, to add a corollary to the Pumping Lemma for context-free languages:

Corollary 2 (Pumping Lemma for Substrings) *Let $L \subset \Sigma^*$ be a context-free language over the alphabet Σ. Then there exists a* pumping length *$p \geq 1$, such that for every word $swt \in L$, with $s, t \in \Sigma^*$ and $|w| \geq 2p - 1$, the center part w can be divided into five parts $w = uvxyz$, satisfying the following three conditions:*

1. $|vy| \geq 1$,
2. $|vxy| \leq p$, *and*
3. $\forall i \geq 0 : suv^ixy^izt \in L$.

This corollary allows us to "pinpoint" the occurrence of a loop inside a very long word in a context-free language: For every consecutive run w of at least $|w| \geq 2p - 1$ characters, there must be a pumping loop completely inside that run.

2.3 The Language of Syntactically Correct `C` Programs

Assume that the language of syntactically correct `C` programs is context-free, giving us a pumping length p. We can create a correct `C` program of length at least p characters as follows: Write a function `void main(void)` that first declares p different integer variables $v_1, \ldots, v_p$ and then initializes them all to zero. Let us generate valid `C` names for all variables v_i by appending the

decimal representation of i to the initial character `v`. For example, if $p = 4$, the program would be

```
void main(void)
    {
    int v1;
    int v2;
    int v3;
    int v4;

    v1 = 0;
    v2 = 0;
    v3 = 0;
    v4 = 0;
    }
```

For readability, the above program is typeset with optional whitespace. For this proof, however, we have to write the program with as little whitespace as possible: The only whitespace characters will appear between `void` and `main` and between each keyword `int` and the following variable name.

From the Pumping Lemma for Substrings, we know that there must be pumping loop vxy of length at most $|vxy| \leq p$ occurring inside the "declaration part" of the program, since it is $7p \geq 2p - 1$ characters long. We know that the pumped part of the string contains at least one character. This means, when pumping down the total length of the declaration part of the program must decrease by one. But from the way the declaration part is set up, we know that not a single one of its characters is redundant. Removing any one would either make at least one declaration statement syntactically invalid, or it would change the name of at least one of the declared variables. In the former case, the program becomes incorrect immediately; in the latter case, the program will have an undeclared variable in the initialization part. Either way, the result of pumping down can never be a syntactically correct `C` program. This implies that the language of syntactically correct `C` programs is not context-free.

This observation seems contrary to the fact that all `C` compilers are based on the description of `C`'s syntax by a context-free grammar. The reason why this is done is that `C` is *almost* context-free – the only part of `C` that is not context-free is the law that every expression and every variable in a `C` program have to have a well-known type, and that every variable and every function

have to be declared before they can be used. It can be shown that correct `C` programs can be defined by a context-sensitive (CH-1) grammar; nobody ever does this, because specifying this grammar is extremely tedious – to my knowledge, no one ever has even tried it – and parsing it is extremely expensive. It is much easier to treat `C` as context-free, and to take care of declarations and types as "special cases." In all compilers, the names and types of variables are stored in a structure called *symbol table*. In this respect, a parser for `C` is not really a pushdown automaton, but close enough for practical purposes.