CS112: Theory of Computation (LFA)

Lecture 11: Non-context-free Languages

Dumitru Bogdan

Faculty of Computer Science
University of Bucharest

May 13, 2025

Table of contents

Section 1

Previously on CS112

CFG Formal definition

Definition

A **context-free grammar** is a 4-tuple (V, Σ, R, S) where:

- V is a finite set called the **variables**
- Σ is a finite set, disjoint from V called the **terminals**
- R is a finite set of rules, with each rule being a variable and a string of variables and terminals
- $S \in V$ is the start variable

Chomsky normal form

- Working with context-free grammars, it is often convenient to have them in simplified form
- The simplest and most useful forms is called **the Chomsky normal form**
- Chomsky normal form is useful in giving algorithms for working with context-free grammars

CNF Formal definition

Definition

A context-free grammar is in Chomsky normal form if every rule is of the form

$$A \to BC$$
$$A \to a$$

where a is any terminal and A, B, and C are any variables — except that B and C may not be the start variable. In addition, we allow the rule $S \to \epsilon$, where S is the start variable

Equivalence PDA - CFG

Theorem

A language is context free if and only if some pushdown automaton recognizes it

As usual for "if and only if" theorems, we have two directions to prove. We formulate both directions as lemmas.

Lemma

If a language is context free, then some pushdown automaton recognizes it

Lemma

If a pushdown automaton recognizes some language, then it is context free

Proof idea $\Rightarrow$

- Let A be a CFL. From the definition we know that A has a CFG, G generating it. We show how to convert G into an equivalent PDA, which we call P
- The PDA P that we now describe will work by accepting its input w, if G generates that input, by determining whether there is a derivation for w
- Recall that a derivation is simply the sequence of substitutions made as a grammar generates a string
- Each step of the derivation yields an intermediate string of variables and terminals
- We design P to determine whether some series of substitutions using the rules of G can lead from the start variable to w
- One of the difficulties in testing whether there is a derivation for w is in figuring out which substitutions to make
- The PDA's nondeterminism allows it to guess the sequence of correct substitutions
- At each step of the derivation, one of the rules for a particular variable is selected nondeterministically and used to substitute for that variable

Proof idea $\Rightarrow$

- The PDA P begins by writing the start variable on its stack. It goes through a series of intermediate strings, making one substitution after another
- Eventually it may arrive at a string that contains only terminal symbols, meaning that it has used the grammar to derive a string
- Then P accepts if this string is identical to the string it has received as input
- Implementing this strategy on a PDA requires one additional idea. We need to see how the PDA stores the intermediate strings as it goes from one to another
- Simply using the stack for storing each intermediate string is tempting. However, that doesn't quite work because the PDA needs to find the variables in the intermediate string and make substitutions
- The PDA can access only the top symbol on the stack and that may be a terminal symbol instead of a variable
- The way around this problem is to keep only part of the intermediate string on the stack: the symbols starting with the first variable in the intermediate string
- Any terminal symbols appearing before the first variable are matched immediately with symbols in the input string

Proof idea $\Rightarrow$

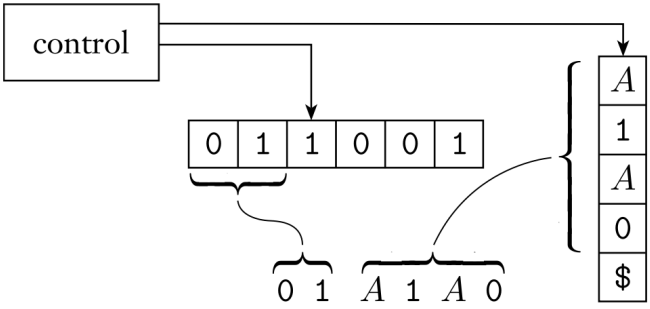

Figure: P representing the intermediate string $01A1A0$

Proof idea $\Rightarrow$

The following is an informal description of P

1. Place the marker symbol \$ and the start variable on the stack
2. Repeat the following steps forever
   - 2.1 If the top of stack is a variable symbol A, nondeterministically select one of the rules for A and substitute A by the string on the right-hand side of the rule
   - 2.2 If the top of stack is a terminal symbol a, read the next symbol from the input and compare it to a. If they match, repeat. If they do not match, reject on this branch of the nondeterminism
   - 2.3 If the top of stack is the symbol \$, enter the accept state. Doing so accepts the input if it has all been read

Proof idea $\Leftarrow$

- We have a PDA P and we want to make a CFG G that generates all the strings that P accepts. In other words, G should generate a string if that string causes the PDA to go from its start state to an accept state
- We design a grammar that does somewhat more. For each pair of states p and q in P, the grammar will have a variable A_{pq} . This variable generates all the strings that can take P from p with an empty stack to q with an empty stack. Observe that such strings can also take P from p to q, regardless of the stack contents at p, leaving the stack at q in the same condition as it was at p
- We simplify our task by modifying P slightly to give it the following three features:
  1. It has a single accept state, q_{accept}
  2. It empties its stack before accepting
  3. Each transition either pushes a symbol onto the stack (a *push* move) or pops one off the stack (a *pop* move), but it does not do both at the same time

Proof idea $\Leftarrow$

- Giving P features 1 and 2 is easy.
- To give it feature 3, we replace each transition that simultaneously pops and pushes with a two transition sequence that goes through a new state, and we replace each transition that neither pops nor pushes with a two transition sequence that pushes then pops an arbitrary stack symbol
- To design G so that A_{pq} generates all strings that take P from p to q, starting and ending with an empty stack, we must understand how P operates on these strings
- For any such string x, P's first move on x must be a push, because every move is either a push or a pop and P can't pop an empty stack. Similarly, the last move on x must be a pop because the stack ends up empty

Proof idea $\Leftarrow$

- So, two possibilities occur during P's computation on x. Either the symbol popped at the end is the symbol that was pushed at the beginning, or not
- If that's the case then the stack could be empty only at the beginning and end of P 's computation on x
- If not, the initially pushed symbol must get popped at some point before the end of x and thus the stack becomes empty at this point
- We simulate the former possibility with the rule $A_{pq} \rightarrow aA_{rs}b$, where a is the input read at the first move, b is the input read at the last move, r is the state following p, and s is the state preceding q
- We simulate the latter possibility with the rule $A_{pq} \rightarrow A_{pr}A_{rq}$, where r is the state when the stack becomes empty

Section 2

Context setup

Context setup

Corresponding to Sipser 2.3

Context setup

- In this lecture we present a technique for proving that certain languages are not context free
- Recall that one previous lecture we introduced the pumping lemma for showing that certain languages are not regular
- Here we study a similar pumping lemma for context-free languages. It states that every context-free language has a special value called the pumping length such that all longer strings in the language can be "pumped"
- In this case the meaning of pumped is a bit more complex. It means that the string can be divided into five parts so that the second and the fourth parts may be repeated together any number of times and the resulting string still remains in the language

Section 3

Pumping Lemma

Pumping lemma for context-free languages

Theorem

If A is a context-free language, then there is a number p (the pumping length) where, if s is any string in A of length at least p, then s may be divided into five pieces $s = uvxyz$ satisfying the conditions

1. *for each $i \geq 0$, $uv^ixy^iz \in A$*
2. $|vy| > 0$
3. $|vxy| \leq p$

- When s is being divided into $uvxyz$, condition 2 says that either v or y is not the empty string. Otherwise the theorem would be trivially true
- Condition 3 states that the pieces v, x and y together have length at most p. This technical condition sometimes is useful in proving that certain languages are not context free

Proof idea

- Let A be a CFL and let G be a CFG that generates it. We must show that any sufficiently long string s in A can be pumped and remain in A. The idea behind this approach is simple
- Let s be a very long string in A. (later will be clear what we mean by "very long"). Because s is in A, it is derivable from G and so has a parse tree
- The parse tree for s must be very tall because s is very long. That is, the parse tree must contain some long path from the start variable at the root of the tree to one of the terminal symbols at a leaf
- On this long path, some variable symbol R must repeat because of the pigeonhole principle
- This repetition allows us to replace the subtree under the second occurrence of R with the subtree under the first occurrence of R and still get a legal parse tree
- So, we may cut s into five pieces $uvxyz$ as the figure indicates, and we may repeat the second and fourth pieces and obtain a string still in the language
- In other words we have uv^ixy^iz in A for any $i \geq 0$

Proof idea

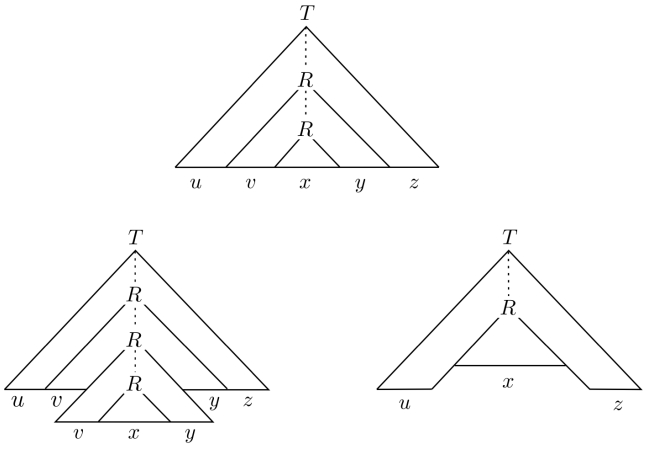

Figure: Surgery on parse trees

Formal proof

Proof.

Let G be a CFG for CFL A. Let b be the maximum number of symbols in the right-hand side of a rule, we assume at least 2. In any parse tree using this grammar, we know that a node can have no more than b children. In other words, at most b leaves are 1 step from the start variable; at most b^2 leaves are within 2 steps of the start variable; and at most b^h leaves are within h steps of the start variable. So, if the height of the parse tree is at most h, the length of the string generated is at most b^h. Conversely, if a generated string is at least $b^h + 1$ long, each of its parse trees must be at least $h + 1$ high.

Say $|V|$ is the number of variables in G. We set p, the pumping length, to be $b^{|V|+1}$. Now if s is a string in A and its length is p or more, its parse tree must be at least $|V| + 1$ high, because $b^{|V|+1} \geq b^{|V|} + 1$. □

Formal proof

Proof.

To see how to pump any such string s, let τ be one of its parse trees. If s has several parse trees, choose τ to be a parse tree that has the smallest number of nodes. We know that τ must be at least $|V| + 1$ high, so its longest path from the root to a leaf has length at least $|V| + 1$. That path has at least $|V| + 2$ nodes; one at a terminal, the others at variables. Hence that path has at least $|V| + 1$ variables. With G having only $|V|$ variables, some variable R appears more than once on that path. For convenience later, we select R to be a variable that repeats among the lowest $|V| + 1$ variables on this path. □

Formal proof

Proof.

We divide s into $uvxyz$ as seen in previous image. Each occurrence of R has a subtree under it, generating a part of the string s. The upper occurrence of R has a larger subtree and generates vxy, whereas the lower occurrence generates just x with a smaller subtree. Both of these subtrees are generated by the same variable, so we may substitute one for the other and still obtain a valid parse tree. Replacing the smaller by the larger repeatedly gives parse trees for the strings uv^ixy^iz at each $i > 1$. Replacing the larger by the smaller generates the string uxz. That establishes condition 1 of the lemma. □

Formal proof

Proof.

To get condition 2, we must be sure that v and y are not both ϵ. If they were, the parse tree obtained by substituting the smaller subtree for the larger would have fewer nodes than τ does and would still generate s. This result isn't possible because we had already chosen τ to be a parse tree for s with the smallest number of nodes. That is the reason for selecting τ in this way.

To get condition 3, we need to be sure that vxy has length at most p. In the parse tree for s the upper occurrence of R generates vxy. We chose R so that both occurrences fall within the bottom $|V| + 1$ variables on the path, and we chose the longest path in the parse tree, so the subtree where R generates vxy is at most $|V| + 1$ high. A tree of this height can generate a string of length at most $b^{|V|+1} = p$. □

Section 4

Examples

Example

- Le us use the pumping lemma to show that the language $B = \{a^n b^n c^n | n \geq 0\}$ is not context free
- We assume that B is a CFL and obtain a contradiction
- Let p be the pumping length for B that is guaranteed to exist by the pumping lemma
- Select the string $s = a^p b^p c^p$. Clearly s is a member of B and of length at least p
- The pumping lemma states that s can be pumped, but we show that it cannot
- We show that no matter how we divide s into $uvxyz$, one of the three conditions of the lemma is violated

Example

- Condition 2 stipulates that either v or y is nonempty. Then we consider one of two cases, depending on whether substrings v and y contain more than one type of alphabet symbol
  1. When both v and y contain only one type of alphabet symbol, v does not contain both a's and b's or both b's and c's, and the same holds for y. In this case, the string uv^2xy^2z cannot contain equal numbers of a's, b's, and c's. Therefore, it cannot be a member of B. That violates condition 1 of the lemma and is thus a contradiction
  2. When either v or y contains more than one type of symbol, uv^2xy^2z may contain equal numbers of the three alphabet symbols but not in the correct order. Hence it cannot be a member of B and a contradiction occurs.
- Since one of these cases must occur and because both cases result in a contradiction, a contradiction is unavoidable
- So the assumption that B is a CFL must be false. Thus we have proved that B is not a CFL

Example

- Let $C = \{a^i b^j c^k | 0 \leq i \leq j \leq k\}$. We use the pumping lemma to show that C is not a CFL
- Assume that C is a CFL and obtain a contradiction
- Let p be the pumping length given by the pumping lemma. We use the string $s = a^p b^p c^p$ that we used earlier, but this time we must "pump down" as well as "pump up"
- The pumping lemma states that s can be pumped, but we show that it cannot
- We show that no matter how we divide s into $uvxyz$, one of the three conditions of the lemma is violated
- Condition 2 stipulates that either v or y is nonempty. Then we consider one of two cases, depending on whether substrings v and y contain more than one type of alphabet symbol

Example

1. When both v and y contain only one type of alphabet symbol, v does not contain both a's and b's or both b's and c's, and the same holds for y. Now, we must analyze the situation more carefully to show that s cannot be pumped. Observe that because v and y contain only one type of alphabet symbol, one of the symbols a, b, or c doesn't appear in v or y. We further subdivide this case into three subcases according to which symbol does not appear:
   - 1.1 The a's do not appear. Then we try pumping down to obtain the string $uv^0xy^0z = uxz$. That contains the same number of a's as s does, but it contains fewer b's or fewer c's. Therefore, it is not a member of C, and a contradiction occurs
   - 1.2 The b's do not appear. Then either a's or c's must appear in v or y because both can't be the empty string. If a's appear, the string uv^2xy^2z contains more a's than b's, so it is not in C. If c's appear, the string uv^0xy^0z contains more b's than c's, so it is not in C. Either way, a contradiction occurs
   - 1.3 The c's do not appear. Then the string uv^2xy^2z contains more a's or more b's than c's, so it is not in C, and a contradiction occurs

Example

2. When either v or y contains more than one type of symbol, uv^2xy^2z will not contain the symbols in the correct order. Hence it cannot be a member of C, and a contradiction occurs

- Thus we have shown that s cannot be pumped in violation of the pumping lemma and that C is not context free