CS 3719 (Theory of Computation and Algorithms) – Pushdown automata and context-free grammars

Antonina Kolokolova[*]

January 24, 2019

1 Pushdown automata

We just proved that some languages are not regular. And, moreover, the examples we have seen are languages that can easily be computed by a simple algorithm in any modern programming language. Now a natural question is whether it is possible to add a little extra power to NFAs, so that the resulting model of computation would be able to handle languages such as $\{0^n q^n\}$. Indeed, it is possible to do so by giving NFAs a little "ability to count", some unlimited memory. There are several ways of adding memory to NFAs, and we will look at two of them, Pushdown Automata and Turing machine (in short, Pushdown Automata have an access to an unlimited stack, and Turing machines to an unlimited tape.) In this lecture, we will look at Pushdown automata and analyze its power. Later we will show that Pushdown Automata still fall short of computing many languages that we view as easily computable by our usual algorithmic techniques.

Informally, a pushdown automaton is just an NFA with a stack. So the additional part of the description of such an automaton should include transitions that operate with the stack, as well as stack alphabet.

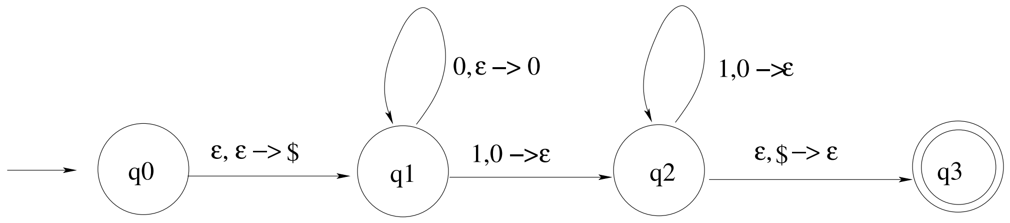
Definition 7. *A pushdown automaton (PDA) is a 6-tuple* $(Q, \Sigma, \Gamma, \delta, q_0, F)$ *where* Q *is the set of states,* Σ *the input alphabet,* Γ *the stack alphabet,* q_0 *the start state,* F *is the set of finite states and the transition function* $\delta : Q \times (\Sigma \cup \{\epsilon\}) \times (\Gamma \cup \{\epsilon\}) \to \mathcal{P}(Q \times (\Gamma \cup \{\epsilon\}))$.

That is, each transition of a PDA pops a symbol (possibly ϵ, corresponding to popping nothing) off the stack; δ specifies which set of states to go from the current state on reading an input symbol and a stack symbol, and for every such state, which, if any, symbol to push onto the stack.

[*]The material in this set of notes came from many sources, in particular "Introduction to Theory of Computation" by Sipser and course notes of U. of Toronto CS 364.

Note that this definition extend the definition of a non-deterministic finite automaton. It is possible to define deterministic pushdown automata by similarly extending the definition of a DFA. However, there are languages accepted by non-deterministic PDAs that no deterministic one can accept (such as a set of palindromes). The proof of this is beyond the scope of this course; however, it is good to remember this as a case where non-determinism actually results in a more powerful model of computation: it didn't for the finite automata. Later in the course we will talk about the P vs. NP problem, a major open problem in computer science which asks about the role of non-determinism for feasible computation.

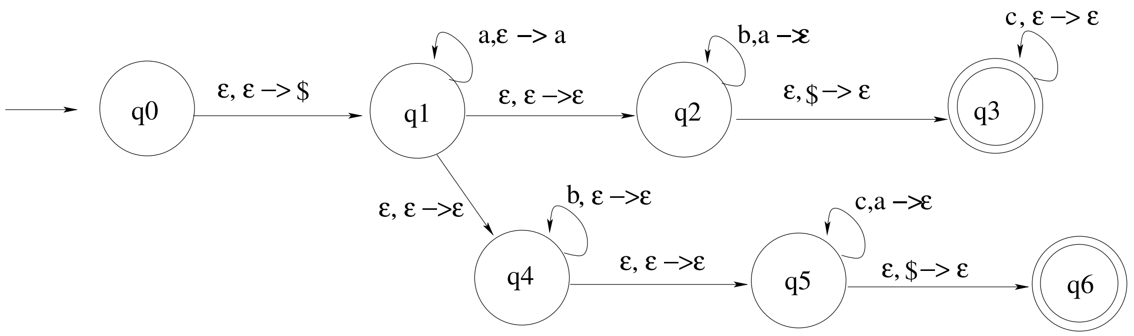
Example 1. Recall the language $\{0^n 1^n\}$ which we have shown to be non-regular in previous lecture. The following PDA accepts this language.



Here, the tape alphabet is $\Gamma = \{0, \$\}$, where \$ is a special symbol used as an empty stack marker. Since we never need to put 1 on the stack in this PDA, it is OK not to have 1 in Γ, although it is common to take $\Sigma \subseteq \Gamma$.

Let us do one more example of a Pushdown Automaton. In this case, we will consider a language where non-determinism is really unavoidable: $\{a^i b^j c^k | i = j \vee i = k\}$.

Example 2. The following PDA recognizes the language $\{a^i b^j c^k | i = j \vee i = k\}$.



Here, the tape alphabet is $\Gamma = \{a, \$\}$. Note the non-deterministic choice this PDA makes: in the state q_1 it forks between the automaton matching letters b to letters a on the stack, or matching the c's.

2 Context-Free Grammars

We have shown earlier that regular languages can be described by regular expressions. It is natural to ask is there a similar description for the languages computed by pushdown automata. Indeed there is, and it is a natural formalism that has been used for a long time on its own: context-free grammars. Noam Chomsky defined them as one of the types in his hierarchy; the context-free grammar syntax has been used in the programming language community to describe the syntax of programming languages since Algol (known there as Backus-Naur Form).

When you think of a grammar the first thing that might come to mind is studying a natural language such as French in school, and all these rules about nouns and verbs. Indeed, it is possible to do much of natural language processing (although not everything) using context-free grammar formalism.

Example 3. Consider the following rules:

$$\begin{array}{ll}
< sentence > & - > < nounphrase >< verbphrase > \\
< verbphrase > & - > < verb > \mid < verb >< nounphrase > \\
< nounphrase > & - > \texttt{Jane}|\texttt{the assignment} \\
< verb > & - > \texttt{solved}|\texttt{did}|\texttt{decided}
\end{array}$$

For example, a sentence "Jane did the assignment" can be generated by this grammar, and also "Jane solved". On the other hand, this grammar can generate "The assignment solved Jane", which might not be a desired result.

Now that we have seen an example, let's define formally what is a context-free grammar.

Definition 8. *A* context-free grammar *(CFG) is a 4-tuple* (V, Σ, R, S)*, where*

1) V is a finite set of variables, with $S \in V$ the start variable.

2) Σ is a finite set of **`terminals`** *(disjoint from the set of variables).*

3) R is a finite set of rules, with each rule consisting of a variable followed by $- >$ followed by a string of variables and terminals.

A grammar is a set of substitution rules, where a variable at the head of a rule can be substituted by the string in the body of a rule whenever that variable occurs. Let $A \to w$ be a rule of the grammar, where w is a string of variables and terminals. Then A can be replaced in another rule by w: uAv in a body of another rule can be replaced by uwv (we say uAv yields uwv,denoted $uAv \Rightarrow uwv$). If there is a sequence $u = u_1, u_2, \dots u_k = v$ such that for all i, $1 \leq i < k$, $u_i \Rightarrow u_{i+1}$ then we say that u derives v (denoted $v \overset{*}{\Rightarrow} v$.)

Definition 9. *If G is a context-free grammar, then the language of G is the set of all strings of terminals that can be generated from the start variable:* $\mathcal{L}(G) = \{w \in \Sigma^* | S \overset{*}{\Rightarrow} w\}$. *A language is called a* context-free language *(CFL) if there exists a CFG generating it.*

In the above example, variables are $< sentence >, < nounphrase >, < verbphrase >$, and $< verb >$, with the start variable $S =< sentence >$, the $\Sigma = \{$ `Jane, the assignment`, `solved,did,decided` $\}$ and R consists of the rules listed above. To be more precise, there are more rules than 4: we used a shortcut "—" for several different rules starting with the same variable. E..g, we should have had two rules $< nounphrase > - >$ `Jane`, and $< nounphrase > - >$ `the assignment`.

To see how a CFG can generate a nonregular language, consider the language $\{0^n 1^n\}$ for which we constructed a PDA in the last lecture.

Example 4. The following CFG generates the language $\{0^n 1^n\}$:

$A- > 0A1|\epsilon$

A derivation of a string 000111 in this grammar is:

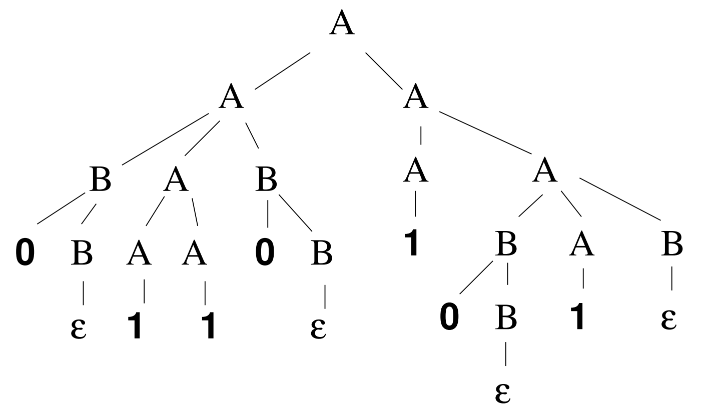
$A- > 0A1- > 00A11- > 000A111- > 000111$

The last step is because adding or removing an empty substring ϵ does not change a string.

We often visualize a derivation as a tree (a parse tree), with variables as internal nodes and terminals as leaves.

Example 5. Consider the following grammar. Here is a parse tree for a string 0110101. Note that there are several possible parse trees for this string. In this case, we say that a grammar is *ambiguous.*

$A \to AA$
$A \to BAB$
$B \to OB$
$B \to \epsilon$
$A \to 1$



2.1 Arithmetic expressions

A canonical example of use of context-grammars is in parsing. In particular, here we will see how to parse an arithmetic expression using context-free grammars.

Example 6. Consider the following grammar G_1:

$EXPR \to EXPR + EXPR | EXPR * EXPR | (EXPR) | x | y | z | 0 | 1 | 2 | 3$

This grammar generates arithmetic expressions such as $x + 2 * y$. However, there is a problem: it generates it ambiguously, ignoring precedence rules. So evaluating the expression according to the tree might give different answers, depending whether the first rule applied was multiplication or addition.

```
         EXPR                        EXPR
        / |  \                     /  |  \
     EXPR +  EXPR                EXPR *  EXPR
      |     / | \               / | \     |
      x    2  *  y             x  +  2    y
```

How would we modify the grammar to make it respect precedence rules? One way of doing it is to give different names to parts of a sum vs. parts of a product and treat them differently.

Example 7. Consider the following grammar G_1, with EXPR the start symbol.

$EXPR \to EXPR + TERM | TERM$
$TERM \to TERM * FACTOR | FACTOR$
$FACTOR \to (EXPR) | x | y | z | 0 | 1 | 2 | 3$

Note that in this case for the arithmetic expression there is only one possible parse tree:

```
           EXPR
        /   |    \
     EXPR   +     TERM
      |         /  |  \
     TERM    TERM  *   FACTOR
      |       |          |
    FACTOR  FACTOR       y
      |       |
      x       2
```

3 Pumping lemma for CFLs

Just like we proved that certain languages are not regular by defining a property (pumping lemma) which all regular languages satisfy, and then showing that some languages don't, we will define a same kind of property for context-free languages.

Consider a simple grammar $A \to aAb, A \to c$. It is clear that it generates a non-regular language: a pumping lemma proof would say that either a's or b's would have to be pumped separately, or c would be repeated multiple times, resulting in strings not in the language. However, if we could pump both a's and b's at the same time, this would work.

With this intuition, we state the pumping lemma for context-free languages.

Lemma 7. *Let L be a context-free language. Then there exists a natural number p such that $\forall s \in L$, $|s| \geq p$, $s = uvxyz$ where*

1) $\forall i \geq 0, uv^ixy^iz \in L$

2) $|vy| > 0$

3) $|vxy| \leq p$.

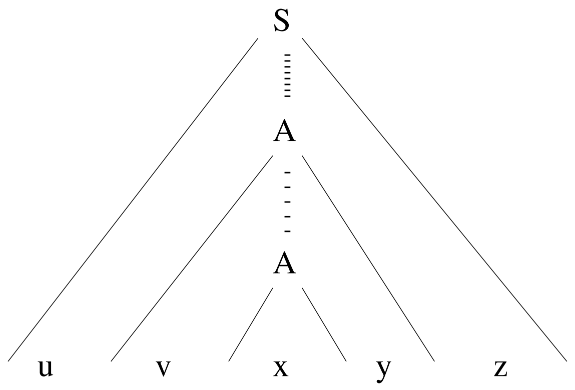
That is, if regular languages could be split into three parts so that the middle part could be iterated without creating strings not in the language, for CFLs the split is into 5 parts so that the middle and the sides stay, and the 2nd and 4th parts are iterated simultaneously. For the grammar we just saw u can consist of several a's, and y of the same number of b's; the middle part x contains the c.

Proof. In the proof for regular languages, we found a repeating part by looking at the states of the automaton and finding a repeating states. Here, we will do the same, but with parse trees.

Here is an idea. Consider a parse tree which is so high that its height is more than the number of variables in the grammar. Why should such a tree exist? It is because if the language is infinite, then there are infinitely many parse trees, at least one distinct tree for a different string in the language (remember that a string is a sequence of leaves of the parse tree). There are finitely many rules, and each rule is of finite length, so the only way to have infinitely many trees is to allow them to grow to an arbitrary height.

Let's look more carefully at the possible parse trees. Suppose that the number of symbols (variables and terminals) in the body of the longest rule is d: then our tree would be d-ary (every node having at most d children). We need to know what is the shortest string guaranteed to have a parse tree of height at least $|V| + 1$. Recall that the longest possible string produced by a d-ary tree of height h has length d^h: it corresponds to a sequence of leaves of a complete d-ary tree of height h. Therefore, if we take a string of length at least $d^{|V|}+1$ it is guaranteed to have all parse trees of height at least $|V|+1$. For convenience, we can choose even larger p; let $p = d^{|V|+1}$.

Now, why would we be interested in trees of this form? Consider the largest path from the root to a leaf in this tree. Since the height of the tree is greater than the number of variables, there would be a repeating variable, say A, on this path. A variable occurring in a parse tree means that a rule with this variable at the head was applied. But now notice that there is no way in the grammar to differentiate these occurrences and say how many there should be; thus, a subtree rooted at the first occurrence of A can be replaced by subtree in the second occurrence, and vice verse. The first one gives us $i = 0$ from the pumping lemma, and iterating the second one gives any i we desire (see the figure).



To check the second condition, suppose that we start with the smallest possible parse tree for the string. Then doing the substitution of the first occurrence by the second would give us an even smaller tree, which is a contradiction.

Finally, let's prove the third condition. Starting from the bottom of the tree, choose the first time from the bottom a variable repeats; call that variable A. A subtree under A is of height at most $|V|+1$, so it generates a string of length at most p. Here, the second occurrence of A generates vxy where the bottom occurrence generates x. $\square$

Now, let's see how this lemma can be used to show that some languages are not context-free.

Example 8. The language $\{a^nb^nc^n\}$ is not context-free.

Suppose it is. Then let p be the pumping length, and take $s = a^pb^pc^p$. Now, just like in the proof that $\{0^n1^n\}$ is not regular, no matter how we split $s = uvxyz$, either v and y are monochromatic and thus repeating them disrupts the equality of the numbers, or they are not monochromatic and repeating them disrupts the order.

Now, recall that the language $\{a^nb^nc^m\}$ is context-free: it is a special case of $\{a^ib^jc^k | i = j \text{ or } i = k\}$. Also, $\{a^nb^mc^n\}$ is context-free, for the same reason. But their intersection is exactly $\{a^nb^nc^n\}$, which we just shown to be not context-free! This leads us to a surprising corollary:

Corollary 8. *The class of context-free languages are not closed under intersection.*

From there, we can also show that there is a context-free language such that its comlement is not context-free: since union of two context-free languages is contex-free, and for any sets A and B, $A \cap B = \overline{\overline{A} \cup \overline{B}}$ by DeMorgan's law, if CFLs were closed under complementation they would also be closed under intersection.

Corollary 9. *The class of context-free languages are not closed under complementation.*

Example 9. The language $\{ww | w \in \{0,1\}^*\}$ is not context-free.

Take $s = 0^p 1^p 0^p 1^p$. By the 3rd condition of the pumping lemma, that $|vxy| \leq p$, the sequence vxy overlaps at most two out of four blocks in our string. Either it is part of $0^p 1^p$; then repeating any part of it will disrupt the equality with the second half of s. Or it overlaps the $1^p 0^p$ piece in the middle; but then either v or y are monochromatic: say y is, and it consists of just 0s. But now repeating y will increase the number of 0s in the second copy of w, but not the first, again disrupting the equality.

3.1 Regular languages vs. context-free languages

We have seen an example of a nonregular language that is context-free. But are there regular languages that are not context-free? Here, we will show that indeed the class of regular languages is a (strict) subset of the class of context-free languages, by showing that every regular language is context-free.

Note that once we show that pushdown automata accept exactly context-free languages the proof becomes easy: just construct a pushdown automaton out of a NFA which would ignore the stack; you can see that this PDA would accept the same language as the NFA. But as a warm-up for showing the equivalence between context-free languages and pushdown automata, let us show directly that every regular language can be generated by a context-free grammar.

As an example of the ideas involved in the proof, let us show that context-free languages are closed under the star operation. Suppose that G is a context-free grammar. We would like to create a grammar for $\mathcal{L}(G)^*$, where each element is a finite string consisting of 0 or more concatenated strings from $\mathcal{L}(G)$. Let S be the start symbol of G. Now, add a rule $S \to SS|\epsilon$ to the grammar. Suppose a string w consists of k occurrences of strings from $\mathcal{L}(G)$. If $k = 0$, then G generates w by the $S \to \epsilon$ part of the rule. Otherwise, if $k > 0$, then apply the first part of the rule $k - 1$ times, and then use the rest of the rules of the grammar. This shows that every string in $\mathcal{L}(G)^*$ is generated by the new grammar. But how can we make sure that no spurious strings are generated? A trick we can use here is to modify the original grammar first so that there are no more rules using S, other than the start rule. For that, just introduce a new start symbol S_0 and add a rule $S_0 \to S$. Now, think of a parse tree for this grammar. Once a symbol other than S_0 appears on a path from the root down, S_0 cannot appear anywhere below it. So the whole subtree will be generated according to the rules of the original grammar, resulting in a string from G. Since this applies to every subtree (except for ones with ϵ-leaves which can be ignored as part of the string), the resulting string will be a concatenation of strings in $\mathcal{L}(G)$.

Theorem 10. *Every regular language is context-free.*

Proof. Let A be a regular language. Then there exists a DFA $N = (Q, \Sigma, \delta, q_0, F)$ such that $\mathcal{L}(N) = A$. Build a context-free grammar $G = (V, \Sigma, R, S)$ as follows. Set $V = \{R_i | q_i \in Q\}$ (that is, G has a variable for every state of N). Now, for every transition $\delta(q_i, a) = q_j$ add a rule $R_i \to aR_j$. For every accepting state $q_i \in F$ add a rule $R_i \to \epsilon$. Finally, make the start variable $S = R_0$. □

Example 10. Consider a deterministic automaton with 3 states accepting all strings containing ab which has transitions $\delta(q_0, a) = q_1$ and $\delta(q_1, b) = q_2$, among others. Here, q_0 is a start state and $F = \{q_2\}$. Add rules $R_0 \to aR_1$, $R_1 \to bR_2$, $R_2 \to \epsilon$ (and similarly for the rest of transitions). Make $S = R_0$. You can check that the resulting grammar generates all strings with a substring ab.

Last class we saw that regular languages are a subclass of context-free languages. You may be wondering what other main types of (grammar-defined) languages there are. In mid-50s, Noam Chomsky has classified grammars into 4 main types. Type 3 is regular; type 2 context-free. The other two are context-sensitive and then unrestricted grammars. The last type is equivalent to Turing machines we will soon study; context-sensitive languages correspond to linear bounded automata which we will skip in this course.

Every context-free grammar can be transformed into one in Chomsky Normal Form: there, only rules of the form $A \to BC$ and $A \to a$ are allowed (as well as $S \to \epsilon$ for the start symbol S; however, in Chomsky Normal Form S cannot occur in the body of any of the rules). Here, A, B, C, S are variables and $a \in \Sigma$. We will skip a proof that every context-free grammar can be converted into one in normal form; see e.g. Sipser's textbook for the proof.

3.2 Equivalence between pushdown automata and context-free languages

Just like we have shown that regular expressions and finite automata recognize the same class of languages, we can show that context-free grammars and pushdown automata recognize the same larger class of languages.

Theorem 11. *A language A is context-free if and only if there exists a pushdown automata N such that $\mathcal{L}(N) = A$.*

We will skip the proof (see Sipser's book for it).

Intuitively, to construct a PDA given a grammar we build a pushdown automaton similar to one for the same number of a's and b's language: there will be one "central" state, which would contain transitions for every rule of the grammar, with variables and terminals of the grammar (plus the end marker) forming Γ. First, put the end marker \$ and then the start state onto the stack. At every iteration of the loop at the central state, non-deterministically choose one of the rules to apply as follows: if popping the stack produces a variable, push the right side of some rule for this variable onto the stack (that might require several intermediate states to push several symbols). If the top of the stack is a terminal (symbol from the alphabet), match it with the input.

The idea of the proof of the other direction is similar to the example with constructing a context-free grammar from a DFA. The extra work would be in keeping track of what is on the stack. See Sipser's book for the proof.