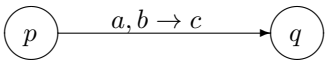
Pushdown Automata (PDA) to Context-Free Grammars (CFG)

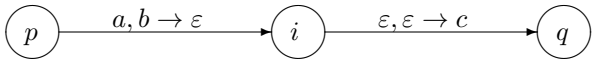
What we do in this lecture. In the previous lectures, we have showed how, for each context-free grammar, to design a pushdown automaton that accepts exactly the words generated by this grammar.

Let us show that, vice versa, for each pushdown automaton, there exists a context-free grammar that generates exactly the words accepted by the given automaton.

Algorithm: preliminary stage. If the original pushdown automaton has a transition in which we both push and pop



then we add an intermediate state i, and replace the original transition with *two* transitions, in which we first pop and then push:



Variables of the resulting grammar. Now, we introduce the grammar with the following variables:

- we have the starting variable S that represents the set of all the words accepted by the original PDA, and
- for every pair (p, q) of states p and q of the original pushdown automaton, we add a variable A_{pq}; this variable represents the set of all the words for which:
  - if we start in the state p with the empty stack,
  - we can get to state q with the empty stack.

Comments. Note that the word that enables us to go from p with the empty stack to q with the empty stack does not necessarily enable us to go from q to p.

So, in general, the variables A_{pq} and A_{qp} will be different – since they represent different languages.

How many variables we have in the resulting grammar depends on how many states the original pushdown automaton has:

- If the pushdown automaton has only one state 1, then we have two variables: S and A_{11}.
- If the pushdown automaton has two states 1 and 2, then we have 5 variables: S, A_{11}, A_{12}, A_{21}, and A_{22}.
- If the pushdown automaton has 4 states 1, 2, 3, and 4, then we have the starting state S, and we also have 4 groups of variables A_{pq} corresponding to all 4 states p:
  - four variables $A_{11}, A_{12}, A_{13}, A_{14}$ corresponding to four possible states q: $q = 1$, $q = 2$, $q = 3$, and $q = 4$;
  - four variables $A_{21}, A_{22}, A_{23}, A_{24}$ corresponding to four possible states q: $q = 1$, $q = 2$, $q = 3$, and $q = 4$;
  - four variables $A_{31}, A_{32}, A_{33}, A_{34}$ corresponding to four possible states q: $q = 1$, $q = 2$, $q = 3$, and $q = 4$;
  - four variables $A_{41}, A_{42}, A_{43}, A_{44}$ corresponding to four possible states q: $q = 1$, $q = 2$, $q = 3$, and $q = 4$.

  So overall, we have 4 groups of variables A_{pq} with 4 variables in each group, to the total of $4 \cdot 4 = 4^2$ variables of the type A_{pq}. Counting the starting state S, we thus have $4^2 + 1 = 17$ variables.
- In general, if the pushdown automaton has n states, then we have n^2 variables of type A_{pq} plus the starting variable S. So, overall, we have $n^2 + 1$ variables.

Rules of the resulting grammar. In this grammar, we have four families of rules.

First family of rules. The first family consists of the rules of the type

$$S \to A_{sf}$$

where s is the starting state of the PDA and f is its final state.

Each such rule means that if we start in the starting state s with an empty stack and end up in the final state f with an empty stack, then the word that led us from s to f is accepted by the PDA – this is the definition of what it means for a word to be accepted by the PDA.

Second family of rules. We also have rules of the type

$$A_{pp} \to \varepsilon$$

for all states p.

Each such rules means that if we are in the state p with the empty stack, and we do not read any symbol, then we remain in the state p with the empty stack.

Third family of rules. For every three states p, q, and r, we have the rule

$$A_{pr} \to A_{pq}A_{qr}$$

The meaning of this rules comes from the definition of the concatenation of two languages – as the set of all the concatenations ww', where w is a word from the first language and w' is a word from the second language. Indeed:

- if we have a word w from the language A_{pq}, this means that if we start in the state p with the empty stack, and we read the word w, then we can get to the state q with the empty stack;
- if we have a word w' from the language A_{qr}, this means that if we start in the state q with the empty stack, and we read the word w', then we can get to the state r with the empty stack.

So, if we see the word ww', we can first go from p to q, then from q to r, and we end up in the state r with the empty stack.

Fourth family of rules. If we push a symbol t into the stack, at some point we need to pop it from the stack. For each pair of rules in which the same symbol is first pushed and then popped:

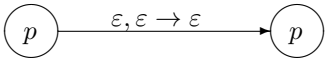
we add the rule

$$A_{ps} \to xA_{qr}y$$

The meaning of this rule is as follows. Suppose that we are in state p with an empty stack. Then:

- first, we see a symbol x and we go to state q, pushing t into the stack;
- then, we use a word from the language A_{qr} that brings us from the state q with an empty stack to the state r with the empty stack; in our case, the stack was not empty, it has the symbol t; all operations above it will not affect this symbol, so we will end up in the state r with the stack consisting of the same symbol t;
- finally, we read y, this enables us to pop t and go to state s; now we are in the state s with an empty stack.

First note. In rules from the fourth family, x, y, and/or t can be empty (ε). In particular, if $t = \varepsilon$, we can always use, as a second transition, a fictitious (but possible) transition



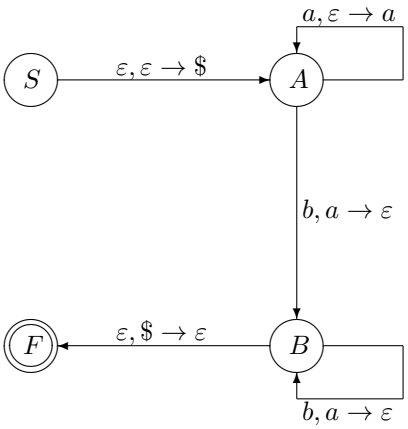
Second note. In our examples, we will not list all the rules of the resulting grammar: there are too many of them. For example, there is a rule from the third family for each triple of states (p, q, r). for an automaton with 4 states, we have 4 possible states p, there are $4^3 = 64$ triples, so we already have 64 rules – and we did not even count the rules from other families!

What we will do instead is listing only the rules which are needed to derive a given word.

Example. As an example, let us consider the very first PDA that we studied: a PDA for recognizing the language

$$\{a^n b^n : n = 1, 2, \ldots\}.$$

The corresponding PDA has the following form:



Let us recall how the word *aabb* is accepted by this automaton. We will list consequent states and the contents of the corresponding stacks, described from the top to bottom, and what symbols we see in the corresponding transitions:

- state S, stack is empty;
- state A, stack has \$;

we read a;

- state A, stack has $a, \$$;

we read a;

- state A, stack has $a, a, \$$;

we read b;

- state B, stack has $a, \$$

we read b;

- state B, stack has $\$$;
- state F, stack is empty.

We start with the state S, we end up in the final state F. Thus, the first rule we apply is the rule $S \to A_{SF}$;

The first operation that we have is pushing the dollar sign into the stack. This dollar sign is then popped at the very end. Let us list the two rules corresponding to pushing and popping:

In general, we have the two transitions

What do we need to plug in instead of p, q, etc. in the general 2-rule picture to come up with this particular picture:

- instead of p, we place S;
- instead of s we place F;
- instead of q, we place A;
- instead of r, we place B;

- instead of x and y, we place ε; and
- instead of t, we place \$.

If we make these substitutions in the general rule:

$$A_{ps} \to xA_{qr}y$$

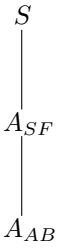
we get the rule

$$A_{SF} \to \varepsilon A_{AB}\varepsilon$$

We know that concatenation with the empty string does not change the language, so $\varepsilon A_{AB}\varepsilon = A_{AB}$, and the rule can be simplified into

$$A_{SF} \to A_{AB}$$

Thus, the derivation so far takes the form:



We have covered the transitions from S to A and from B to F, and we also covered the symbol \$ that stays in the stack between the first occurrence of the state A and the last occurrence of the state B. Let us underline what we have covered already:

- state S, stack is empty;
- state A, stack has \$;

we read a;

- state A, stack has a, \$;

we read a;

- state A, stack has a, a, \$;

we read b;

- state B, stack has a, \$

we read b;

- state B, stack has \$;

- state F, stack is empty.

The first not-yet-covered state is the first occurrence of the state A. The first thing we do when in this state is we push the symbol a into the stack. This symbol a is popped at the last transition from state B to state B. Let us write down the rules corresponding to pushing a and to popping this symbol a:

What do we need to plug in instead of p, q, etc. in the general 2-rule picture to come up with this particular picture:

- instead of p and q, we place A;
- instead of r and s, we place B;
- instead of x, we place a;
- instead of y, we place b; and
- instead of t, we place a.

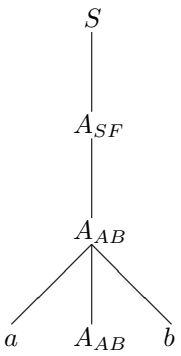
If we make these substitutions in the general rule:

$$A_{ps} \to xA_{qr}y$$

we get the rule

$$A_{AB} \to aA_{AB}b$$

Thus, the derivation so far takes the form:



We have covered two more transitions, and we also covered the first symbol a. Let us summarize what have been covered so far:

- state S, stack is empty;

- state A, stack has \$;

we read a;

- state A, stack has $a, \$$;

we read a;

- state A, stack has $a, a, \$$;

we read b;

- state B, stack has $a, \$$

we read b;

- state B, stack has \$;
- state F, stack is empty.

The first not-yet-covered state is the second occurrence of the state A. The first thing we do when in this state is we push the second symbol a into the stack. This symbol a is popped at the first transition from state A to state B. Let us write down the rules corresponding to pushing this particular symbol a and to popping this symbol a:

What do we need to plug in instead of p, q, etc. in the general 2-rule picture to come up with this particular picture:

- instead of p, q, and r, we place A;
- instead of s, we place B;
- instead of x, we place a;
- instead of y, we place b; and
- instead of t, we place a.

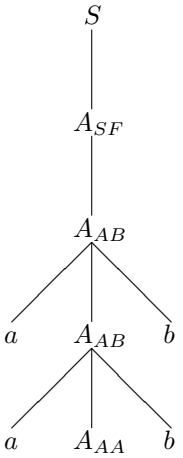
If we make these substitutions in the general rule:

$$A_{ps} \to xA_{qr}y$$

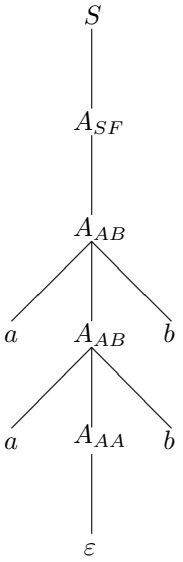
we get the rule

$$A_{AB} \to aA_{AA}b$$

Thus, the derivation so far takes the form:



Now, we have covered all the transitions, so the only thing left to do is to use the trivial rule $A_{AA} \to \varepsilon$:



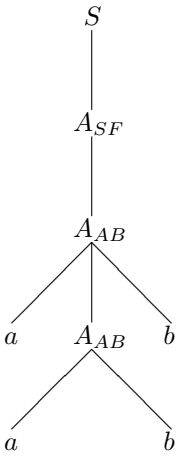
This is the desired derivation of the word $aabb$ in the resulting grammar.

Note. Interestingly, the grammar is not that bad: it is almost the same grammar that we had for this language.

A possible simplification. To make the resulting grammar simpler, we can combine the rules of the type $X \to \ell A_{aa} r$ where one of the elements in the right-hand side is A_{aa} and the rule $A_{aa} \to \varepsilon$ into a single simpler rule $X \to \ell r$.

This is exactly the same simplification as we used in Step 0 of transformation to the Chomsky normal form.

For example, if we apply this simplification to the above derivation, we will get a somewhat simpler derivation tree:



General comment. Given a word which is accepted by the given pushdown automaton, we need to find its derivation in the resulting grammar. For this, we look step-by-step at how the word is accepted.

That the word is accepted means that:

- we start in the starting state s of the pushdown automaton with the empty stack, and
- after reading all the symbols from the word, we end up in one of its final states, also with the empty stack.

So the first rule we use is the rule $S \to A_{sf}$ corresponding to this particular final state.

It can happen that at some point of the derivation, we are in some state with the empty stack. There may be several such intermediate states $q_1, \ldots, q_k$. In this case, the transition from the state s with the empty stack to the state f with the empty stack can be described as the following sequence of transitions:

- first, we go from the starting state s with the empty stack to the state q_1 with the empty stack; by definition of the languages A_{pq}, the corresponding part of the word belongs to the language A_{sq_1};
- then, we go from the state q_1 with the empty stack to the state q_2 with the empty stack; by definition of the languages A_{pq}, the corresponding part of the word belongs to the language $A_{q_1q_2}$;

- ...
- finally, we go from the state q_k with the empty stack to the final state f with the empty stack; by definition of the languages A_{pq}, the corresponding part of the word belongs to the language $A_{q_k f}$.

So:

- If there is only one such intermediate state q_1, we use the rule

$$A_{sf} \to A_{sq_1} A_{q_1 f}.$$

- If there are two such intermediate states, we apply such rules twice:

$$A_{sf} \to A_{sq_1} A_{q_1 f} \text{ and then } A_{q_1 f} \to A_{q_1 q_2} A_{q_2 f},$$

  resulting in the 2-stage derivation

$$\underline{A_{sf}} \to A_{sq_1} \underline{A_{q_1 f}} \to A_{sq_1} A_{q_1 q_2} A_{q_2 f}.$$

- Similarly, we deal with the cases when we have three or more intermediate states with the empty stack.

For each of the transitions between two states with the empty stack, we first *push* some symbol into the stack. Since at the end of this transition, we end up with the empty stack, this means that this symbol has to be *popped* at some place. So:

- we find the transitions that we were used for pushing and for popping, and
- we apply the corresponding rule of the context-free grammar – as in the above example.

Practice. Try deriving other words in the above language. Try applying the same construction to some other PDA.