Pushdown Automata

Table of Contents

[Properties of CFLs 5](#_Toc132742162)

[CFG to PDA 5](#_Toc132742163)

[The CYK Algorithm 8](#_Toc132742164)

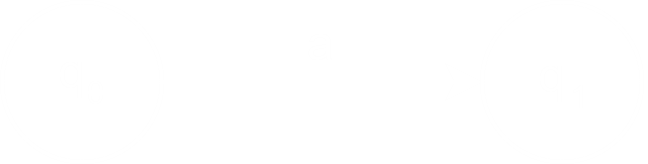
[Pumping Lemma for CFL 14](#_Toc132742165)

**Pushdown Automata** (PDA) are machines that accept CFLs. They are exactly like FSMs, except that they also have a **stack** which allows us to push and pop items, thus acting as memory. PDA can be both deterministic and non-deterministic, but we will only be examining the non-deterministic variant.

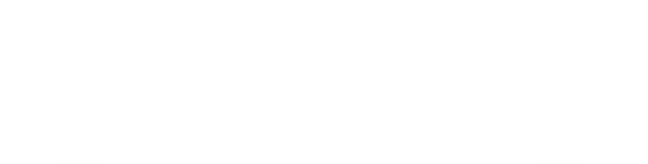
The transition from one state to another now depends on not just the input symbol, but also the top of the stack. When transitioning, we also push an item onto the stack. These two properties can be manipulated in a variety of ways for our convenience, as we will see soon. Thus, .

In the above equation, is being used to denote the **stack alphabet**. This is a set of alphabets which have special meaning with reference to the stack. They may or may not overlap with . For example, or is used to denote the initial top of the stack, i.e., to denote that the stack is empty.

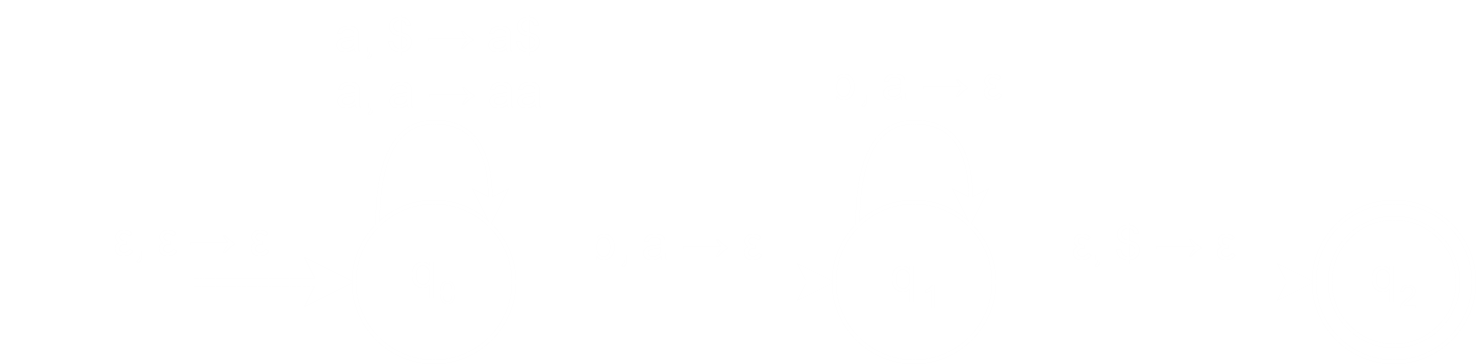
Consider the following example of an FSM:

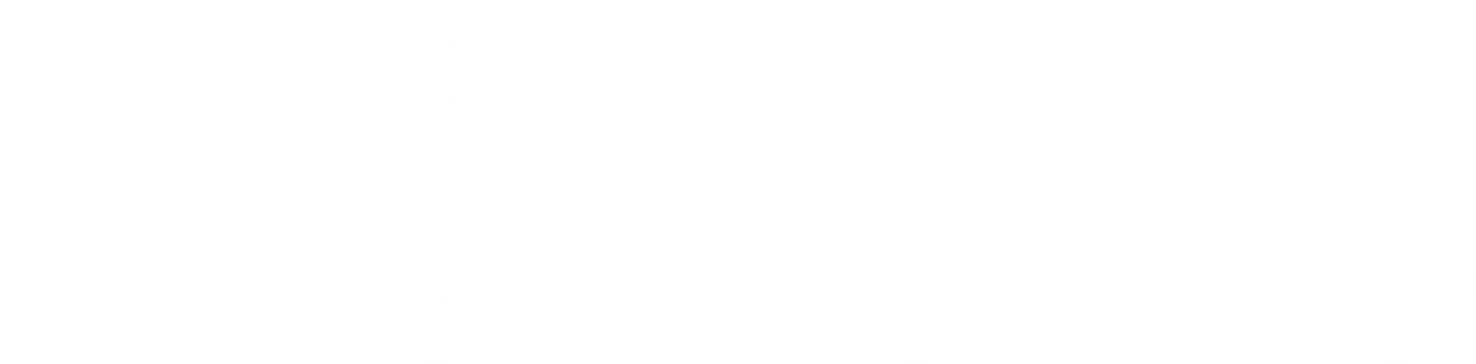


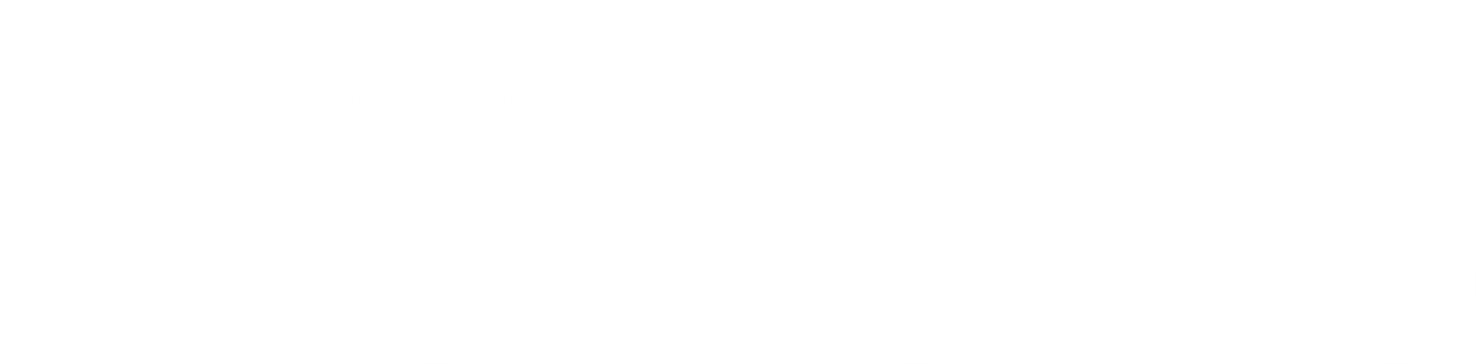
A corresponding PDA would look like this:

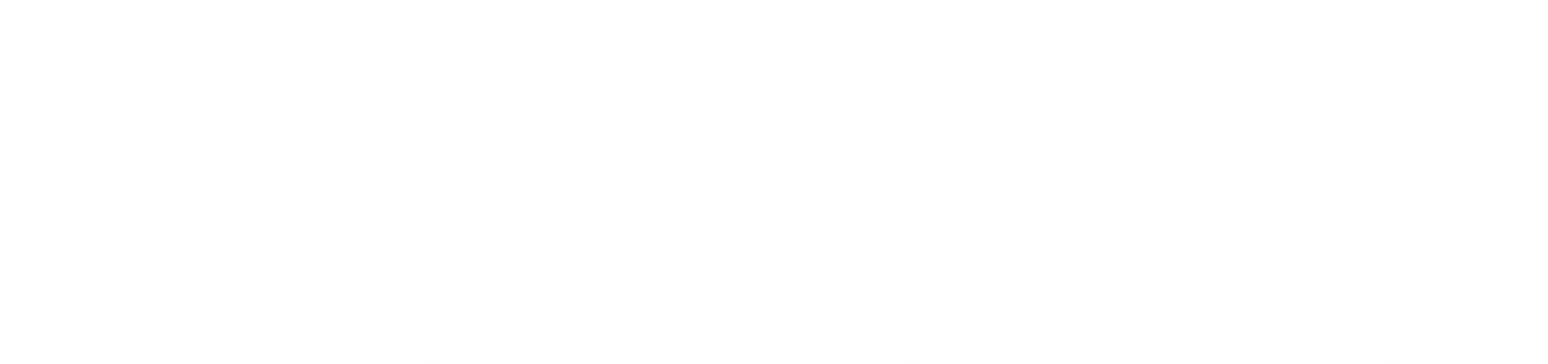


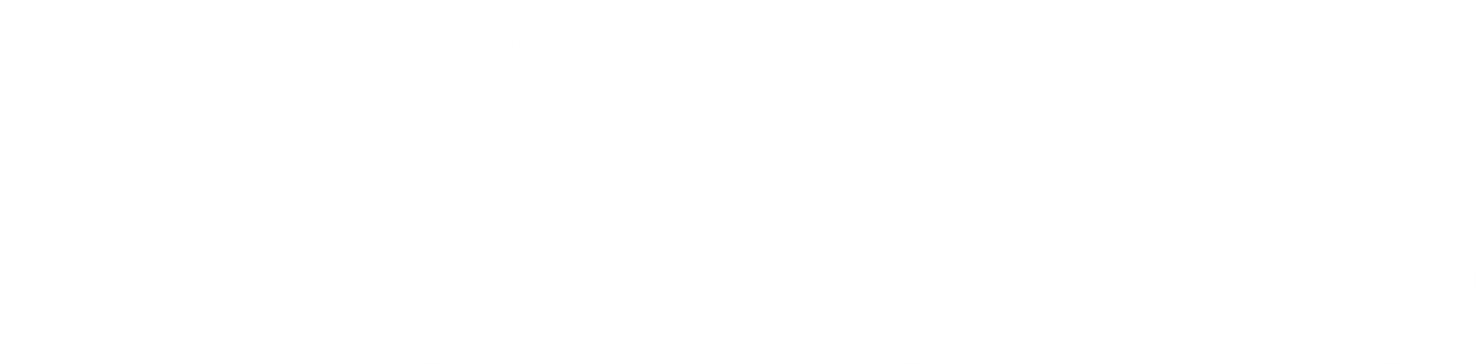
Here, is the input symbol, is the value popped from the stack and is the value to be pushed to the stack. , and can all be .

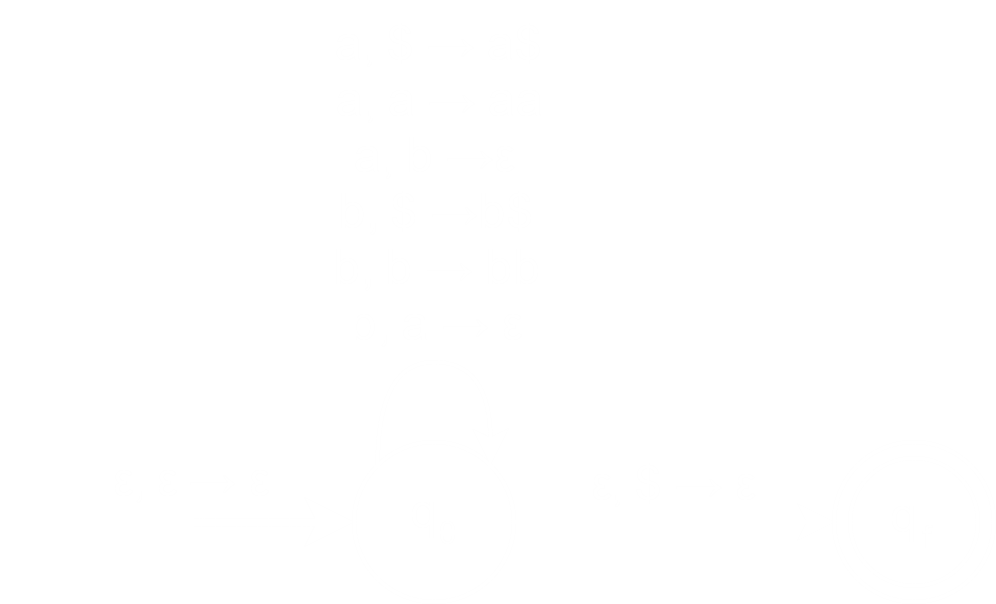


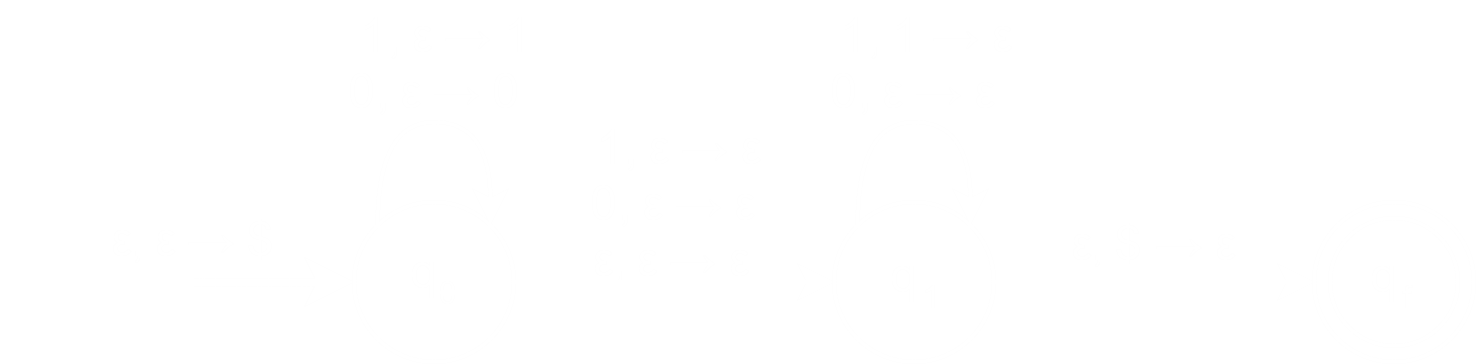












## Properties of CFLs

Let be one CFG with the rules and be another CFG with the rules .

* , where , meaning and are closed under union.
* , where , meaning and are closed under concatenation.
* and are not closed under complement.
* and are not closed under intersection.

To see why and are not closed under intersection, let the corresponding languages be and , which we have previously seen are both CFL. However, their intersection, , is not a CFL.

To see why and are not closed under complement, consider that if and are both CFLs, we know that should also be a CFL. However, cannot be a CFL, since , which we now is not a CFL. Thus, and cannot be closed under complement.

## CFG to PDA

Theorem: A language is a CFL if and only if some PDA recognizes it.

To prove this theorem, there are two parts.

1. Given a CFG, show that a PDA that recognizes it can be constructed.
2. Given a PDA, show that a CFG that it recognizes can be constructed.

The second part of this proof is a bit complicated, so we will not be looking into it here.

Suppose we have the following CFG:

This can be expanded using **left-most derivation**, where we take the left-most non-terminal symbol in the current string and expand it using one of the production rules.

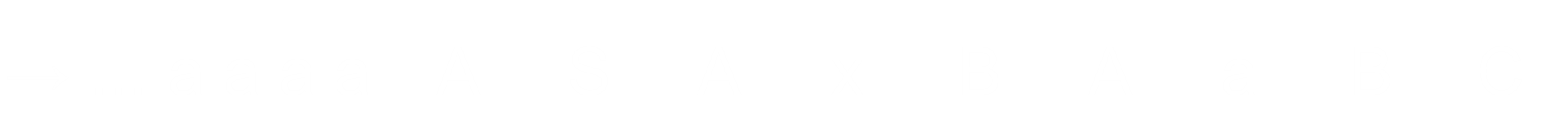
The final string we got is one of the strings in the CFL.

Any of the strings we got in one of the middle steps is said to be in a **left sentential form**. This can be written as , i.e., there is one part on the left that only contains terminal symbols while the rest of the string contains a mixture of terminal and non-terminal symbols. In the PDA, we assume that the terminal section has already been scanned, and the rest of the symbols are stored in the stack.

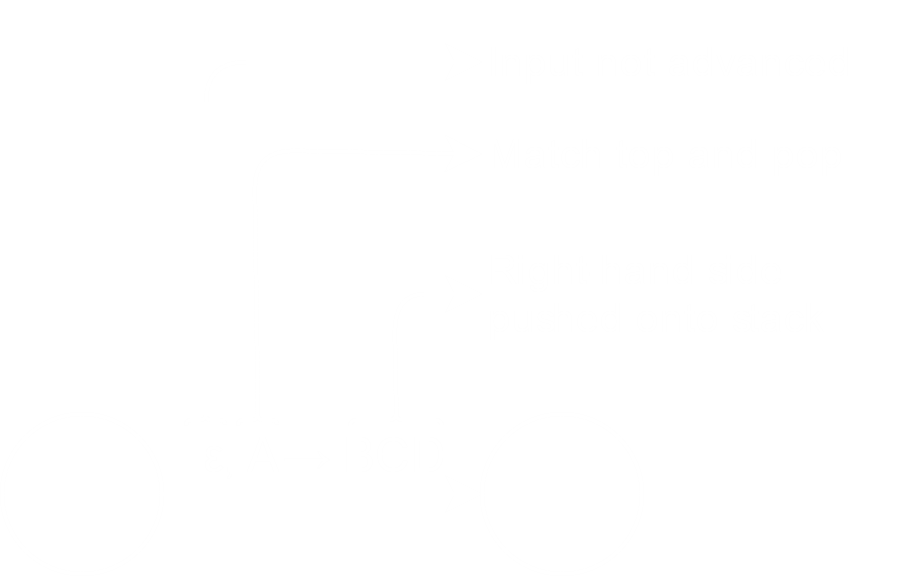


(The stack is shown horizontally here.)

Next, we take the top-most element in the stack and replace it with a rule. Suppose we have a rule . Thus, the PDA becomes:

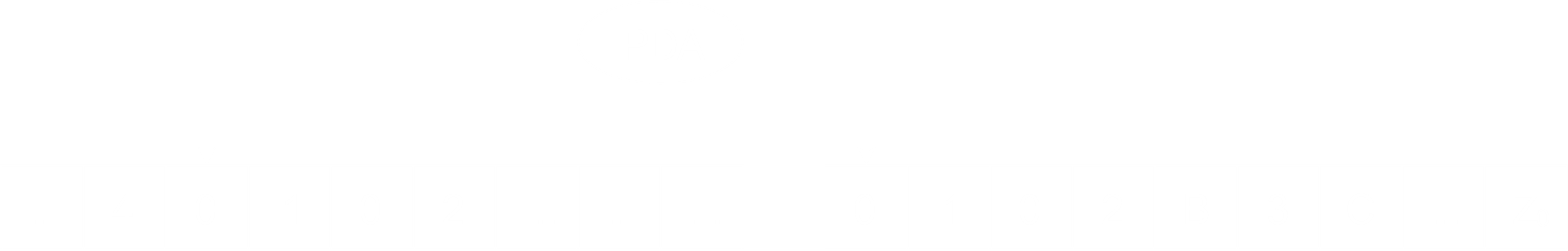


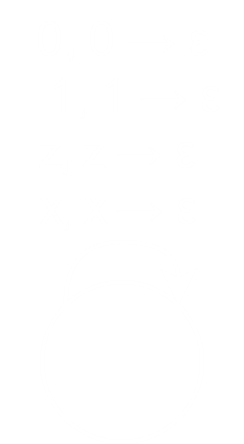
The actual transition would look like this, for a rule :



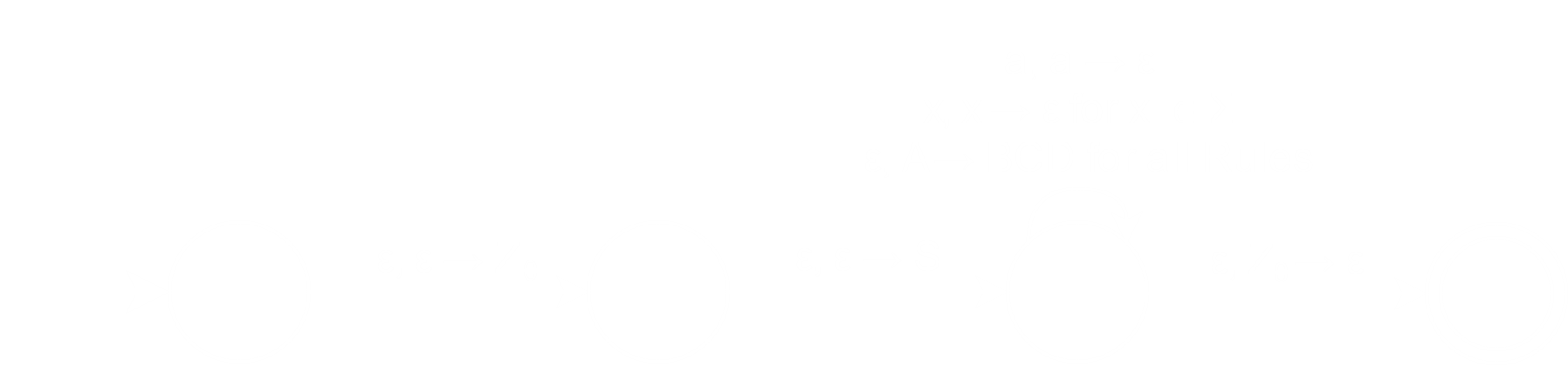
Note that nothing is read, meaning the input position remains the same.

If the top of the stack is a terminal symbol, we just read the input without pushing anything to the stack.





Using this process, we can check if other strings are part of the CFL. The final PDA looks like this:



## The CYK Algorithm

[Surprisingly, not an Indian YouTuber!](https://www.youtube.com/watch?v=VTH1k-xiswM)

Given a grammar and a string , determine if is in (i.e., the language of ).

To accomplish this, we can use the **CYK Algorithm**. This requires the grammar to be in CNF and uses a table-filling algorithm.

The table must be triangular, starting with 1 cell and increasing 1 cell in each row until the last row, which has a number of cells equal to the length of the string. For a string of length 5, the table will look like this:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

To examine how this algorithm works, we will be using an example.

Each row represents the set of substrings of length , where is the row number, starting from the bottom. We need to identify the set of non-terminal symbols that can be used to used to create that substring.

For example, the first substring in the bottom row is , which can be created using , while the second, , can be created using or .Similarly, we can fill the rest of the table.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
| {B} | {A, C} | {A, C} | {B} | {A, C} |
| b | a | a | b | a |

In the second row, the first substring of length 2 is . We know that and . There are two possible combinations of this, and . can be produced using and can be produced using .

The second substring is , and can only be produced using . Thus, the combinations we get are , , and . Amongst these, only can be produced using .

Similarly, we fill the rest of the row.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
| {S, A} | {B} | {S, C} | {S, A} |  |
| {B} | {A, C} | {A, C} | {B} | {A, C} |
| b | a | a | b | a |

The third row starts to get a little difficult. The first substring is . We can take this in two parts, followed by or followed by . From the existing diagram we know that , , and . Thus, the possible combinations are , , , or . None of these can be created using the given grammar.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
| {S, A} | {B} | {S, C} | {S, A} |  |
| {B} | {A, C} | {A, C} | {B} | {A, C} |
| b | a | a | b | a |

The second substring in this row is . Again, breaking it into two parts, we get , , and . Thus, the possible combinations are , , , and . Amongst these, can be produced using .

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  |  |  |  |  |
|  | {B} |  |  |  |
| {S, A} | {B} | {S, C} | {S, A} |  |
| {B} | {A, C} | {A, C} | {B} | {A, C} |
| b | a | a | b | a |

Similarly, we can complete the last cell in the row.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  |  |  |  |  |
|  | {B} | {B} |  |  |
| {S, A} | {B} | {S, C} | {S, A} |  |
| {B} | {A, C} | {A, C} | {B} | {A, C} |
| b | a | a | b | a |

For the first four letter substring, , we can have multiple splits:

* , which is produced by .
* , , which is produced by .
* , , which is produced by .

None of these can be produced, so the result is .

The second four letter substring is . This can be split as:

* , , which is produced by .
* , , which is produced by .
* , , which is produced by .

From these, and .

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  | {S, A, C} |  |  |  |
|  | {B} | {B} |  |  |
| {S, A} | {B} | {S, C} | {S, A} |  |
| {B} | {A, C} | {A, C} | {B} | {A, C} |
| b | a | a | b | a |

Finally, the entire string, , can be split as:

* , , given by .
* , , given by .
* , , given by .
* , , given by .

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| {S, A, C} |  |  |  |  |
|  | {S, A, C} |  |  |  |
|  | {B} | {B} |  |  |
| {S, A} | {B} | {S, C} | {S, A} |  |
| {B} | {A, C} | {A, C} | {B} | {A, C} |
| b | a | a | b | a |

Thus, is a valid string for this CFG.

## Pumping Lemma for CFL

The **pumping lemma** can also be used to prove that a language is not a context-free language.

If is a context-free language, then has a pumping length such that any string , where , can be divided into five parts, , such that:

1. is in for every .

Suppose . For , . The following cases are possible for this:

* and each contain only one type of symbol, for example , , , , . For , this does not work.
* Either or has more than one kind of symbol, for example , , , , . For , this does not work.

Thus, the language is not context-free.