Pushdown Automata

How can we go beyond finite automata? We have learned that some languages cannot be recognized by finite automata. So, a natural idea is to supplement a finite automaton with some additional structure.

In the introductory computer science classes, we learned a lot of different data structures. Most of these data structures – array, linked list, etc. – have to be artificially implemented. However, there is one data structure which is already supported – the structure of a stack. You may not think this way, but;

- stack is how variables are allocated space in the computer memory,
- stack is how method calls are handled, etc.

You may not see it, but we all got a typical error message when practicing recursion – *stack overflow*, even when our programs did not use any stacks.

From this viewpoint, a natural idea is to supplement a finite automaton with a stack.

What can we do with a stack?

- We can push something into the stack, and
- we can pop the top element of the stack.

Because of this, the resulting computational device is called a *pushdown automaton* (PDA, for short).

- In the finite automaton, when we read a symbol, all we can do is move to a different state.
- In the pushdown automaton, we can also pop a symbol from the stack and/or push a symbol into the stack.

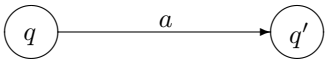
Important comment. When we study stacks in computer science classes, we usually assume that, in addition to push and pop, there is also a possibility to check whether the stack is empty. Pushdown automata were invented when stacks did not use this operation.

So how can we check whether a stack is empty? A natural idea is, from the very beginning, to push some symbol into the stack. At the end, if the only thing we see in the stack is this special symbol, this means that the stack is empty.

Which symbol should be use? On US keyboards, there is a dollar sign. It makes sense to have this sign on the keyboard, since many folks use computers to process financial data. However, in computer science, we rarely use this symbol, so it is a perfect symbol for all auxiliary purposes:

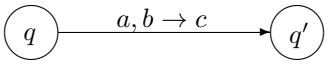
- in pushdown automata, it is used to indicate the empty stack,
- in LaTeX– software that many computer scientists use – the dollar sign is used to indicate the following text is a formula, etc.

How do we describe transitions. In a finite automaton, a transition takes the following form:



meaning that if we are in the state q and we see a symbol a, then we move to state q'.

In the pushdown automaton, transitions are described as follows:



meaning that if in the state q, we see the symbol a and the symbol b is on top of the stack, then:

- we pop b,
- we push c into the stack, and
- go to state q'.

Here:

- If we do not want to pop anything, we use $b = \varepsilon$.
- If we do not want to push anything, we use $c = \varepsilon$.

Note. In a general transition $a, b \to c$, we *read* the symbol a, we *pop* the symbol b, and we *push* the symbol c. Thus, informally, the transitions of the type $a, b \to c$ can be described as

$$read, pop \to push.$$

When do we stop? We start in the starting state with an empty stack. We stop when we are in a final state with an empty stack.

What does it mean for a word to be accepted by a pushdown automaton? In general, a word is *accepted* by a pushdown automaton if:

- once we start with the starting state with the empty stack,
- and we read this word symbol by symbol and perform the corresponding state transitions and push- and pop-operations,
- we end up in a final state with the empty stack.

Comment. Please note that the definition we use here is slightly different from the definition mentioned in our current textbook. There, the word is considered to be accepted if we end up in a final state, irrespective of whether the stack is empty or not. In most examples from the textbook, we do end up with an empty stack when the word is accepted, but there may be different situations.

Deterministic vs. non-deterministic PDA. For finite automata FA, every non-deterministic FA can be transformed into a deterministic one. For PDA, this is not the case: some non-deterministic PDAs cannot be transformed into deterministic ones.

For simplicity, in general, we will study non-deterministic PDAs – although some of them will be actually deterministic.

How can we recognize the language $\{a^n b^n\}$: idea. Let us start with the language for which we proved that this language cannot be recognized by a finite automaton.

For simplicity, let us consider the case when we only allow positive values n, so the language takes the form

$$L = \{ab, aabb, aaabbb, \ldots\}.$$

In this case, a natural idea is as follows:

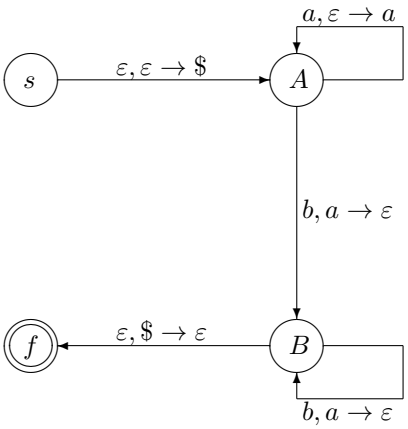
- when we see a symbol a, we push it into the stack;
- when we see a symbol b, we pop one a from the stack.

If after reading all the letters, we end up with an empty stack, this means that we have exactly as many b's as a's – which is exactly what we wanted.

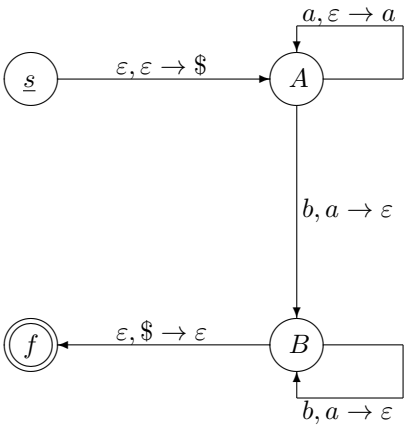
Let us describe this in precise terms.

PDA for recognizing the language $\{a^n b^n, n = 1, 2, \ldots\}$. Let us consider the following PDA, with 4 states:

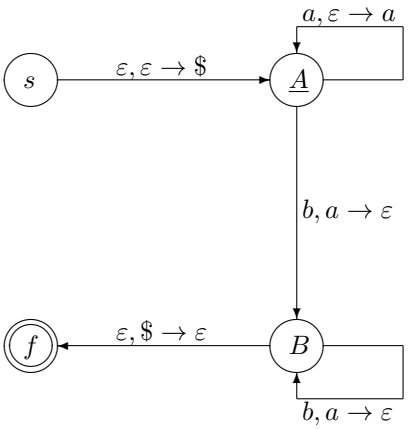
- start (s),
- "in a's" (A),
- "in b's" (B), and
- final (f):



Tracing: example. Let us show how this automaton will accept the word *aabb*. At first, we are in the starting state with an empty stack:

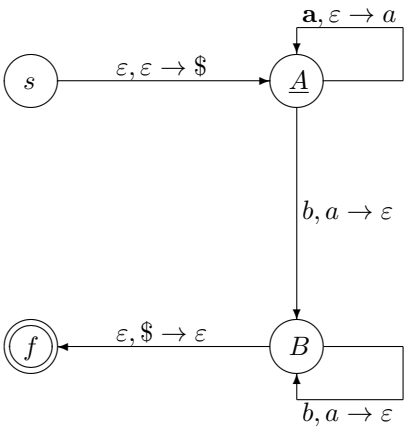


The starting state is not final, and the only way to get out of this state is to apply the rule $\varepsilon, \varepsilon \to \$$, i.e., to push the dollar sign into the stack. Then, we will be in the state A:



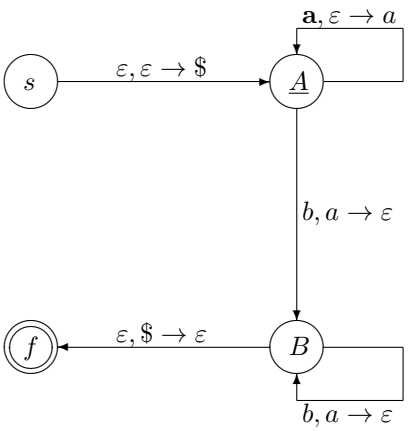
and the stack will contain the symbol \$:

Now, we read the first symbol a in the word **a***abb*, so we push a into the stack and remain in the same state A:

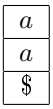


The stack will now contain the letter a on top of the dollar sign:

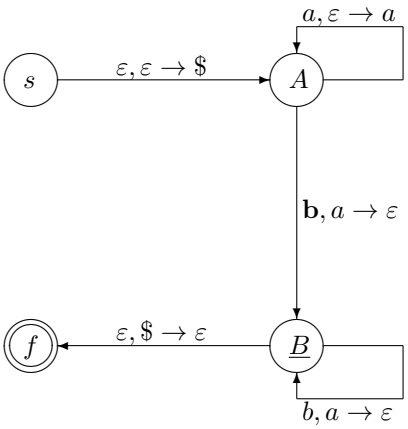
Then, we read the second letter a of the word $a\mathbf{a}bb$, so we also push a into the stack and remain in the state A:



The stack will now contain two letters a on top of the dollar sign:

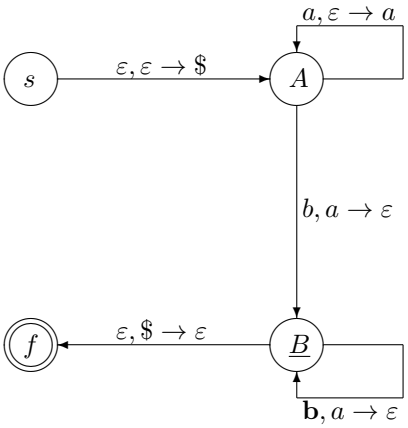


Now, we read the next symbol b of the word $aa\mathbf{b}b$. According to the rules of this PDA, we pop a from the stack and move to the state B:

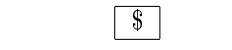


The stack will now contain only one letter a on top of the dollar sign:

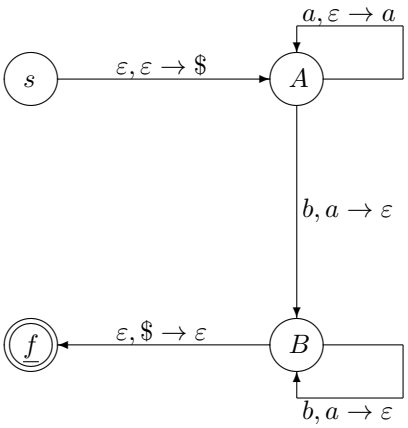
Next, we read the last letter b of the word $aab\mathbf{b}$. According to the rules, we pop a symbol a from the stack, and stay in the state B:



The stack will contain only the dollar sign:



Since we have the dollar sign on top of the stack, we can use the rule $\varepsilon, \$ \to \varepsilon$: pop the dollar sign from the stack and go to the final state:



We are in the final state with the empty stack, so the word *aabb* is accepted.

Practice. Try the same tracing for words *ab* and *aaabbb*.

Recognizing palindromes of type ww^R. Let us now consider the language of all the words of the type ww^R, i.e., all the words that can be obtained by first writing the word, and then writing its reverse. For example:

- based on *ab*, we get *abba*;
- based on the *cat*, we get *cattac*, etc.

How can we recognize such words? While we are in the first half, we push every letter we see into the stack. Once we are in the second half, we pop a letter if we see the same letter in the word. Since the stack reverses the order, this will enable us to recognize such words.

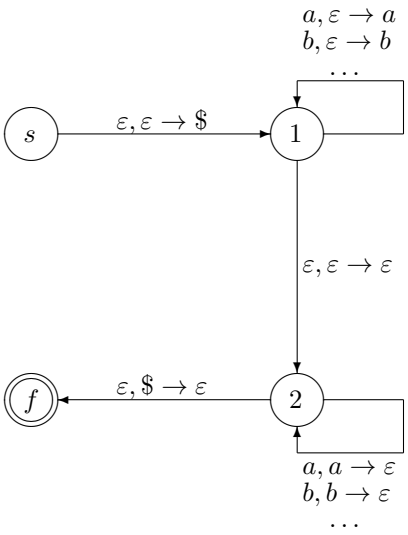
How do we know when the first half ends and the second half start? We don't, that is why the resulting PDA is non-deterministic.

Let us describe this PDA in precise terms. We have four states:

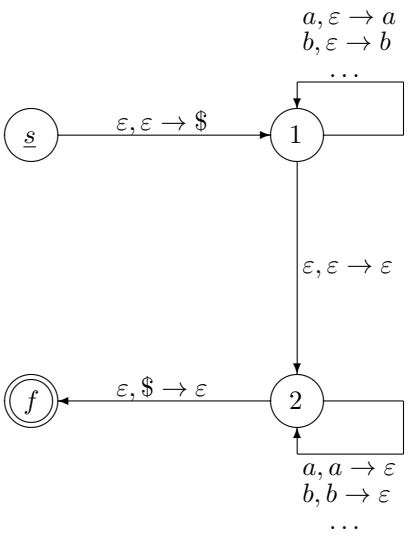
- the start state (s);
- the state "in the first half" (1);
- the state "in the second half" (2); and

- the final state (f).

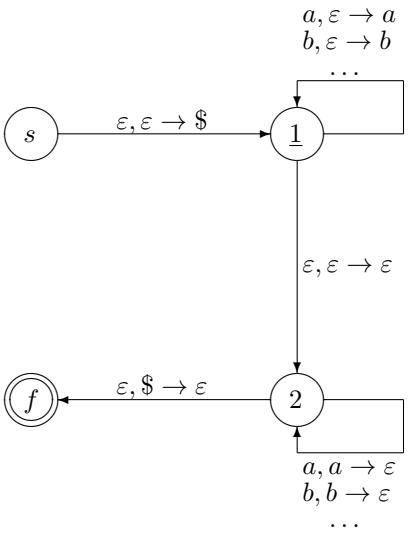
Transitions take the following form:



Tracing. Let us trace this automaton on the example of the word *abba*. In the beginning, we are in the starting state s with the empty stack:

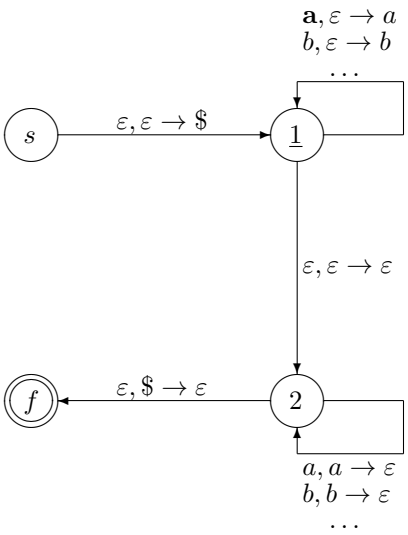


Then, we push the dollar sign into the stack and go to state 1:



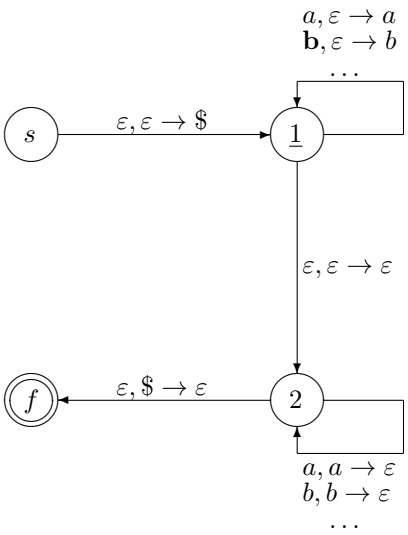
The stack now contains the dollar sign:

We read the first letter a of the word **a***bba*, so we push a into the stack, and remain in the state 1:

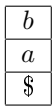


The stack now contains the letter a on top of the dollar sign:

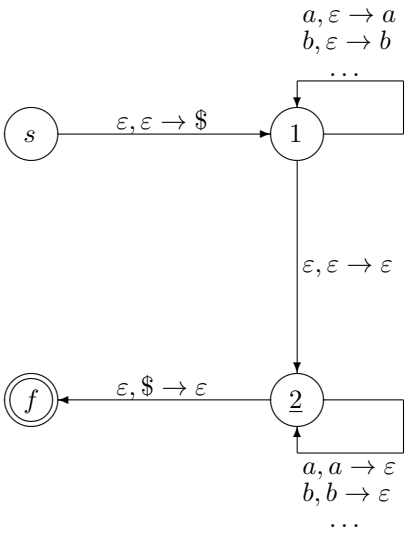
Next, we read the first letter b of the word $a\mathbf{b}ba$. Then, we push b into the stack, and remain in the state 1:



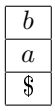
The stack not has two letters on top of the dollar sign:



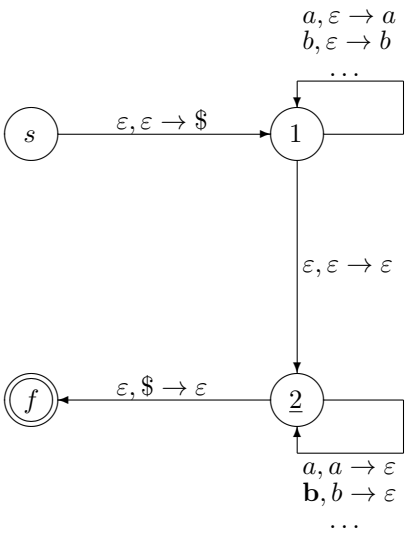
Now, we have read the first half of the word, so it is time to jump to the state corresponding to the second half, i.e., to the state 2:



The stack remains the same:

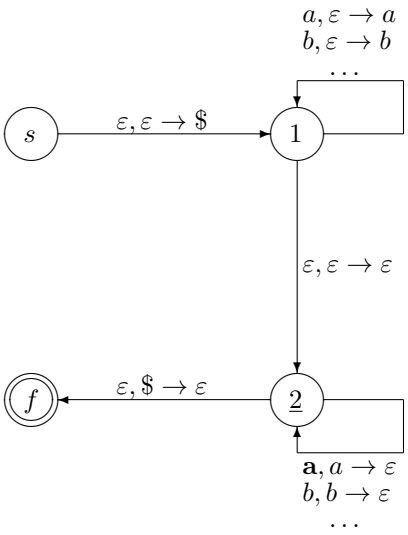


Next, we read the second letter b of the word $ab\mathbf{b}a$, then we pop the letter b from the stack and remain in state 2:



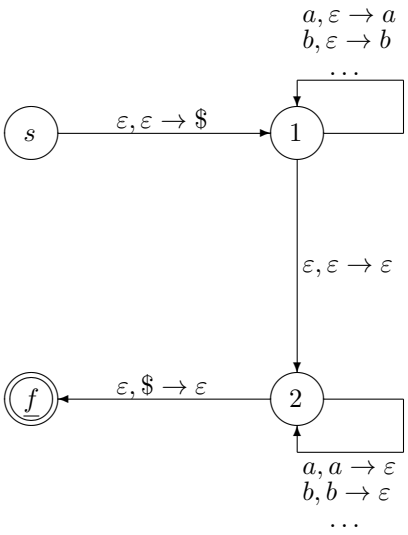
The stack is left with only letter a on top of the stack:

Finally, we read the last letter a of the word $abb\mathbf{a}$. Then, we pop the letter a from the stack and remain in the state 2:



The stack now contains only the dollar sign:

Now, we have $ on top of the stack, so we can pop the dollar sign and go to the final state:



The stack is now empty. We are in the final state with an empty stack, so the word *abba* is accepted.

Practice. Try tracing other words of the type ww^R, such as *cattac*.

Are pushdown automata sufficient? Earlier, we have learned that not every language can be recognized by a finite automaton. As an example of such a language, we had the language

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

In this lecture, we showed that this language can be recognized if use pushdown automata, i.e., if we add a stack to a finite automaton.

A natural question is: can every programming language be recognized by a pushdown automaton? Most constructions can indeed be thus recognized, but, as we will show later, there are languages that cannot be recognized by a finite automaton. An example of such a language will be the language

$$\{a^n b^n c^n : n = 0, 1, 2, \ldots\} = \{\Lambda, abc, aabbcc, \ldots\}.$$

To recognize this language, one stack is not enough. We will prove this by first proving a special pumping lemma for pushdown automata.

To recognize this language, we will need a more complicated computational device called a *Turing machine*, which is, in effect, nothing else but an automaton with *two* stacks.

Are there cases when two stacks are not enough, and we will need three, four, etc.? From the efficiency viewpoint, yes, the more stacks we allow, the faster computations, but the class of languages will not change: whatever language we can recognize by using any number of stacks we can also recognize by using only two stacks – with two stacks, recognition will be somewhat slower but still possible.