Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Talen en Compilers

2019 - 2020, period 2

Jurriaan Hage

Department of Information and Computing Sciences
Utrecht University

2021-01-11

11. Regular languages

This lecture

Regular languages

- Regular grammars
- Regular grammars vs. finite state automata
- Regular expressions
- Pumping lemma for regular languages

11.1 Regular grammars

Regular grammar

A context-free grammar G is called **regular** if all productions are of one of the following two forms:

$$A \to xB$$
$$A \to x$$

where x is a (possibly empty) sequence of terminals, and A and B are nonterminals.

In other words: Every right hand side has **at most one** nonterminal that must occur **at the end**.

Regular language

A language is called **regular** if it can be described by a regular grammar.

Question

Is every context-free language regular?

Regular language

A language is called **regular** if it can be described by a regular grammar.

Question

Is every context-free language regular?

No. The standard example is the language $\{a^n b^n \mid n \in \mathbb{N}\}$.

We will investigate later how such a negative statement (not belonging to the class of regular languages) can be proved.

Simplifying regular grammars

For every regular language, there is a regular grammar that has no productions of the form

$$A \to B$$
$$C \to \varepsilon$$

where A, B, and C are nonterminals, except that for the start symbol there may be a production

$$S \to \varepsilon$$

Simplifying regular grammars – construction

Phase 1: remove productions $A \to B$

Consider all pairs of distinct nonterminals Y and Z. If $Y \Rightarrow^* Z$:

- for every production $Z \to z$ (with z a sequence of symbols, but not a single nonterminal), add a production $Y \to z$.

Remove all productions of the form $Y \to Z$, even if Y equals Z.

Simplifying regular grammars – construction

Phase 2: remove ϵ-productions

For each production $Y \to \varepsilon$, consider all productions $Z \to xY$ (where x now can consist only of terminals) and add a production $Z \to x$.

Then remove all epsilon-productions, except $S \to \varepsilon$ if it exists.

Greibach Normal Form

We can simplify a regular grammar even further and require that all productions are of one of the following two forms

$$\begin{aligned} Y &\to \mathtt{x}Z \\ Y &\to \mathtt{x} \end{aligned}$$

where $\mathtt{x}$ is thus a **single** terminal, except for the start symbol S, for which a production of $S \to \varepsilon$ is allowed.

We say that such a regular grammar is in **Greibach Normal Form**.

Greibach Normal Form – construction

We introduce new nonterminals. For example:

$$A \to \mathtt{x}\mathtt{y}C$$

is transformed into

$$\begin{aligned} A &\to \mathtt{x}B \\ B &\to \mathtt{y}C \end{aligned}$$

11.2 Regular grammars vs. finite state automata

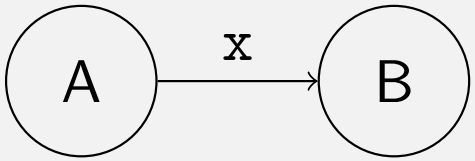
From NFA to regular grammars

We may assume the NFA has only a single start state. For each such NFA, there exists a regular grammar that describes the same language.

From NFA to regular grammars

We may assume the NFA has only a single start state. For each such NFA, there exists a regular grammar that describes the same language.

- the states become nonterminals,
- the start state becomes the start symbol,
- for each transition



we introduce a production

$$A \to \mathtt{x}B$$

- for each accepting state F we introduce a production

$$A \to \varepsilon$$

From regular grammars to an NFA

We first simplify the grammar. Then, for each production:

$$A \to \mathtt{x}$$

we introduce a new state A' and replace that production with:

$$\begin{aligned} A &\to \mathtt{x}A' \\ A' &\to \varepsilon \end{aligned}$$

- The nonterminals become states;
- The start state corresponds to the start symbol;
- We introduce a transition from A to B for every:

  $$A \to \mathtt{x}B$$

- The set of final states are those with productions:

  $$A \to \varepsilon$$

NFAs $\leftrightarrow$ regular grammars

These constructions can be performed in many different ways:

- The one in the lecture notes is slightly different.

Note: we have **not** formally **proven** that they accept the same language.

Exercise

Given two regular grammars for languages L_1 and L_2, there is another regular grammar which accepts $L_1 \cup L_2$.

Exercise

Given two regular grammars for languages L_1 and L_2, there is another regular grammar which accepts $L_1 \cup L_2$.

1. Construct two automata A_1 and A_2 which accept L_1 and L_2, respectively.
2. We know that automata are closed under union, so we can construct a third automata A which accepts $L_1 \cup L_2$.
3. Construct the regular grammar corresponding to A.

Exercise

Given two regular grammars for languages L_1 and L_2, there is another regular grammar which accepts $L_1 \cup L_2$.

1. Construct two automata A_1 and A_2 which accept L_1 and L_2, respectively.
2. We know that automata are closed under union, so we can construct a third automata A which accepts $L_1 \cup L_2$.
3. Construct the regular grammar corresponding to A.

Closure properties of NFAs translate to regular grammars and languages.

11.3 Regular expressions

Regular expressions

Regular expressions are a small **domain-specific language** for the concise description of regular languages.

Regular expressions

Regular expressions are a small **domain-specific language** for the concise description of regular languages.

Let T be an alphabet. We can define the language RE_T of regular expressions over alphabet T using a context-free grammar (terminals in green):

Regular expressions

Regular expressions are a small **domain-specific language** for the concise description of regular languages.

Let T be an alphabet. We can define the language RE_T of regular expressions over alphabet T using a context-free grammar (terminals in green):

$$
\begin{aligned}
\mathsf{RE}_\mathsf{T} &\to \emptyset \\
\mathsf{RE}_\mathsf{T} &\to \varepsilon \\
\mathsf{RE}_\mathsf{T} &\to \mathtt{a} \quad (\text{where } \mathtt{a} \in \mathsf{T}) \\
\mathsf{RE}_\mathsf{T} &\to \mathsf{RE}_\mathsf{T} + \mathsf{RE}_\mathsf{T} \\
\mathsf{RE}_\mathsf{T} &\to \mathsf{RE}_\mathsf{T}\ \mathsf{RE}_\mathsf{T} \\
\mathsf{RE}_\mathsf{T} &\to \mathsf{RE}_\mathsf{T}^* \\
\mathsf{RE}_\mathsf{T} &\to (\ \mathsf{RE}_\mathsf{T}\ )
\end{aligned}
$$

Examples of regular expressions

$(\mathtt{bc})^* + \emptyset$

$\varepsilon + \mathtt{b}(\varepsilon^*)$

Examples of regular expressions

$(\mathtt{bc})^* + \emptyset$
$\varepsilon + \mathtt{b}(\varepsilon^*)$

Operators priorities

- The star operator binds stronger than concatenation,
- Concatenation binds stronger than $+$,
- Parentheses are used for explicit grouping,
- Both concatenation and $+$ are associative.

Language of a regular expression

Every regular expression over T describes a language over T. We will show that the described language is always regular.

The mapping from regular expressions to languages is a semantic function and can be described inductively in terms of the abstract syntax (without parentheses):

$$
\begin{aligned}
L_{RE}(\emptyset) &= \emptyset \\
L_{RE}(\varepsilon) &= \{\varepsilon\} \\
L_{RE}(a) &= \{a\} \quad \text{for any } a \in T \\
L_{RE}(x + y) &= L_{RE}(x) \cup L_{RE}(y) \\
L_{RE}(xy) &= L_{RE}(x)\, L_{RE}(y) \\
L_{RE}(x^*) &= (L_{RE}(x))^*
\end{aligned}
$$

Simple regular expressions

$$
\begin{aligned}
\text{digit} &= 0 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 \\
\text{letter} &= a + b + \ldots + z + A + B + \ldots + Z \\
\text{identifier} &= \text{letter}(\text{letter} + \text{digit})^*
\end{aligned}
$$

Simple regular expressions

$$
\begin{aligned}
\text{digit} &= 0 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 \\
\text{letter} &= a + b + \ldots + z + A + B + \ldots + Z \\
\text{identifier} &= \text{letter}(\text{letter} + \text{digit})^*
\end{aligned}
$$

Generally, regular expressions are very suitable to describe the **lexical syntax** of a language, i.e., the syntax of the **tokens** of a language.

Properties of regular expressions

We define that two regular expressions are equal if they describe the same language. From the definition of L_{RE} it follows that:

$$
\begin{aligned}
\emptyset\, x &= x\, \emptyset = \emptyset \\
\varepsilon\, x &= x\, \varepsilon = x \\
\emptyset + x &= x + \emptyset = x \\
x(yz) &= (xy)z \\
x + y &= y + x \\
x + (y + z) &= (x + y) + z \\
x + x &= x \\
\emptyset^* &= \varepsilon \\
\varepsilon^* &= \varepsilon \\
(x^*)^* &= x^*
\end{aligned}
$$

Syntactic sugar

We can introduce additional familiar constructs:

$$x^+ = xx^*$$
$$x? = x + \varepsilon$$

And a wildcard for an arbitrary symbol from the alphabet $T = \{x_1, \ldots, x_n\}$:

$$\star = x_1 + \ldots + x_n$$

Unix regular expressions

our notation	`grep` *(other tools differ slightly)*
$+$	`|`
$*$	`*`
$+$	`+`
$\star$	`.`
$?$	`?`
$(\ldots)$	`( ... )`

Unix regular expressions

our notation	`grep` *(other tools differ slightly)*
$+$	`|`
$*$	`*`
$+$	`+`
$\star$	`.`
$?$	`?`
$(\ldots)$	`( ... )`

- If `*`, `+`, `.`, `(`, `)`, `?` or `\` occur as terminals in the regular expression, one has to prefix them with a `\`.
- Unix regexps support other facilities outside of regular languages. But then acceptance may not be linear.

Regular grammars for regular expressions

Theorem

For every regular expression, there is a regular grammar that describes the same language.

The proof proceeds by **induction** on the structure of regular expressions:

- Consider each case separately,
- If a regular expression contains sub-expressions, assume that we can obtain the grammar for those (**induction hypothesis**).

Regular grammars for regular expressions – contd.

- For $\emptyset$, we use the empty grammar which is regular.
- For ε, we use the grammar $S \rightarrow \varepsilon$ which is regular.
- For a terminal $\mathtt{a}$, we use the grammar $S \rightarrow \mathtt{a}$ which is regular.

Regular grammars for regular expressions – contd.

- For $x + y$, we may assume regular grammars G_x and G_y with start symbols S_x and S_y for x and y, respectively. Then for a fresh symbol S

$$\begin{aligned} S &\to S_x \\ S &\to S_y \end{aligned}$$

with all productions from G_x and G_y yields a regular grammar for $x + y$.

Regular grammars for regular expressions – contd.

- For xy, we may assume regular grammars G_x and G_y with start symbols S_x and S_y for x and y, respectively. In G_x, we replace every production of the form

$T \to x$	(where x is a terminal string)	by	$T \to xS_y$
$T \to \varepsilon$		by	$T \to S_y$

  With start symbol S_x, the result is a regular grammar for xy.

Regular grammars for regular expressions – contd.

- For x^*, we may assume a regular grammar G_x with start symbol S_x.
  We add a new start symbol S and productions

  $$S \to S_x$$
  $$S \to \varepsilon$$

  In G_x, we replace every production of the form

  $$T \to x \quad \text{(where x is a terminal string)} \quad \text{by} \quad T \to xS$$
  $$T \to \varepsilon \quad \text{by} \quad T \to S$$

  The result is a regular grammar for x^*.

NFA for regular expressions

Easy proof

NFA for regular expressions

Easy proof

1. Turn the regular expression into a regular grammar.
2. Turn the regular grammar into an NFA.

NFA for regular expressions

Easy proof

1. Turn the regular expression into a regular grammar.
2. Turn the regular grammar into an NFA.

Homework

The direct construction is very enlightening.

11.4 Pumping lemma for regular languages

How to prove that a language is not regular?

Generally, proving that a language does not belong to a certain class is much more difficult than proving that it does.

In the case of regular languages,

- **to show that a language is regular**, we have to give one regular grammar (or regular expression, finite state automaton) that describes the language;

How to prove that a language is not regular?

Generally, proving that a language does not belong to a certain class is much more difficult than proving that it does.

In the case of regular languages,

- **to show that a language is regular**, we have to give one regular grammar (or regular expression, finite state automaton) that describes the language;
- **to show that a language is not regular**, we have to prove that no regular grammar (or regular expression, finite state automaton) could ever describe that language.

The strategy

We proceed in the following steps:

1. we expose a **limitation** in the formalism (in this case, in the concept of finite state automata);
2. from this limitation, we derive a **property** that all languages in the class (in this case, regular languages) must have;
3. therefore, if a language does not have that property, it cannot be in the class.

Loops in deterministic finite state automata

Assume we have a deterministic finite state automaton, and we read a string that is accepted.

Loops in deterministic finite state automata

Assume we have a deterministic finite state automaton, and we read a string that is accepted.

How many different states do we visit...

- if the string has length 0?

Loops in deterministic finite state automata

Assume we have a deterministic finite state automaton, and we read a string that is accepted.

How many different states do we visit...

- if the string has length 0?
  – One (the start state).

Loops in deterministic finite state automata

Assume we have a deterministic finite state automaton, and we read a string that is accepted.

How many different states do we visit...

- if the string has length 0?
  – One (the start state).
- if the string has length 1?

Loops in deterministic finite state automata

Assume we have a deterministic finite state automaton, and we read a string that is accepted.

How many different states do we visit...

- if the string has length 0?
  – One (the start state).
- if the string has length 1?
  – Two or one. One if for the given terminal, the start state has a transition to itself, i.e., we walk through a loop.

Loops in deterministic finite state automata

Assume we have a deterministic finite state automaton, and we read a string that is accepted.

How many different states do we visit...

- if the string has length 0?
  – One (the start state).
- if the string has length 1?
  – Two or one. One if for the given terminal, the start state has a transition to itself, i.e., we walk through a loop.
- if the string has length 2?

Loops in deterministic finite state automata

Assume we have a deterministic finite state automaton, and we read a string that is accepted.

How many different states do we visit...

- if the string has length 0?
  – One (the start state).
- if the string has length 1?
  – Two or one. One if for the given terminal, the start state has a transition to itself, i.e., we walk through a loop.
- if the string has length 2?
  – Three or two or one. If less than three, we visit at least one state twice, i.e., walk through a loop.

Loops in deterministic finite state automata

Assume we have a deterministic finite state automaton, and we read a string that is accepted.

How many different states do we visit...

- if the string has length 0?
  – One (the start state).
- if the string has length 1?
  – Two or one. One if for the given terminal, the start state has a transition to itself, i.e., we walk through a loop.
- if the string has length 2?
  – Three or two or one. If less than three, we visit at least one state twice, i.e., walk through a loop.

Finite state automata are finite

Any finite state automaton has a finite number of states.
Assume we have one with n states.

Question

How many different states do we visit while reading a string that is accepted and has length n?

Finite state automata are finite

Any finite state automaton has a finite number of states. Assume we have one with n states.

Question

How many different states do we visit while reading a string that is accepted and has length n?

Answer

According to the previous considerations, $n + 1$ or less, and if less, we traverse a loop.

But there are only n states, so we cannot traverse $n + 1$ different states. Therefore, we **must** traverse a loop.

The strategy – revisited

We proceed in the following steps:

1. we expose a limitation in the formalism (in this case, in the concept of finite state automata);
2. from this limitation, we derive a property that all languages in the class (in this case, regular languages) must have;
3. therefore, if a language does not have that property, it cannot be in the class.

The strategy – revisited

We proceed in the following steps:

1. we expose a limitation in the formalism (in this case, in the concept of finite state automata);
2. from this limitation, we derive a property that all languages in the class (in this case, regular languages) must have;
3. therefore, if a language does not have that property, it cannot be in the class.

We have done the first step. We have found a limitation in the formalism. Now we have to derive a property for all regular languages from that.

A property of loops

Question

What can we say about a loop in a finite state automaton?

Answer

We can traverse it arbitrarily often.

A property of loops

Question

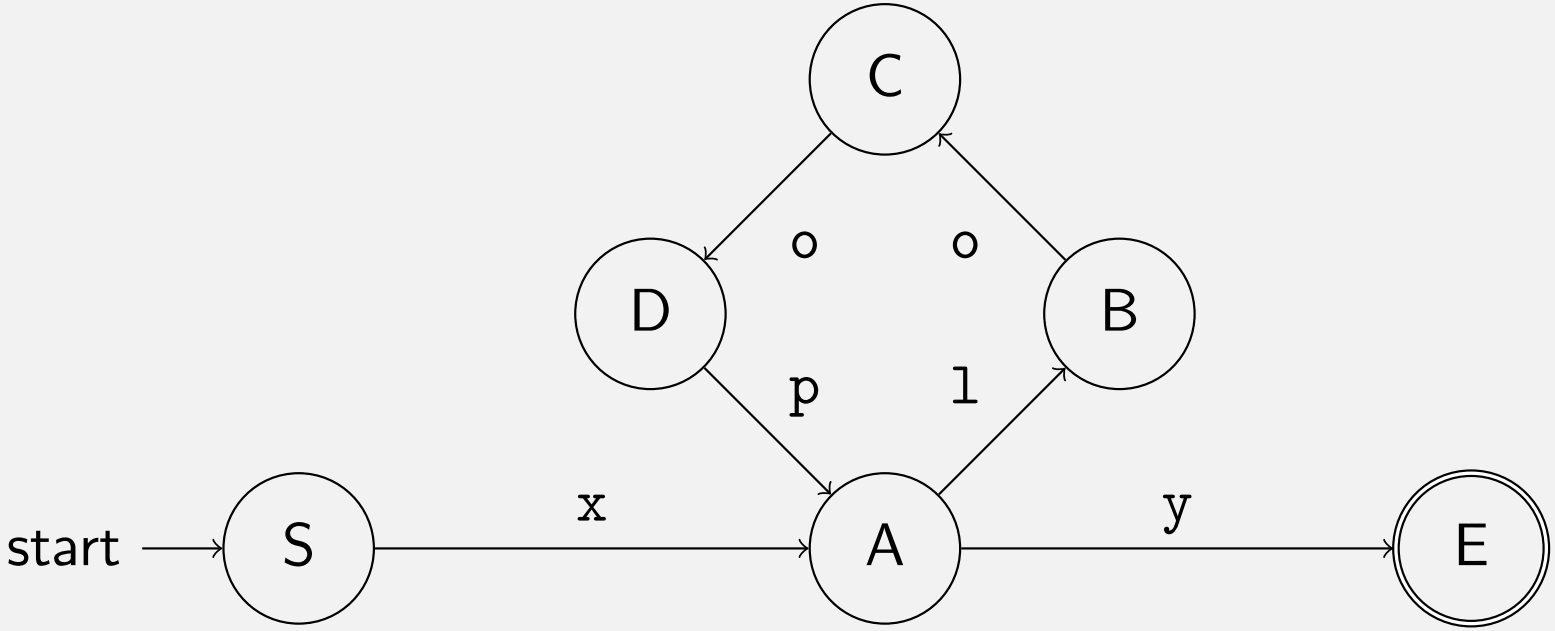
What can we say about a loop in a finite state automaton?

Answer

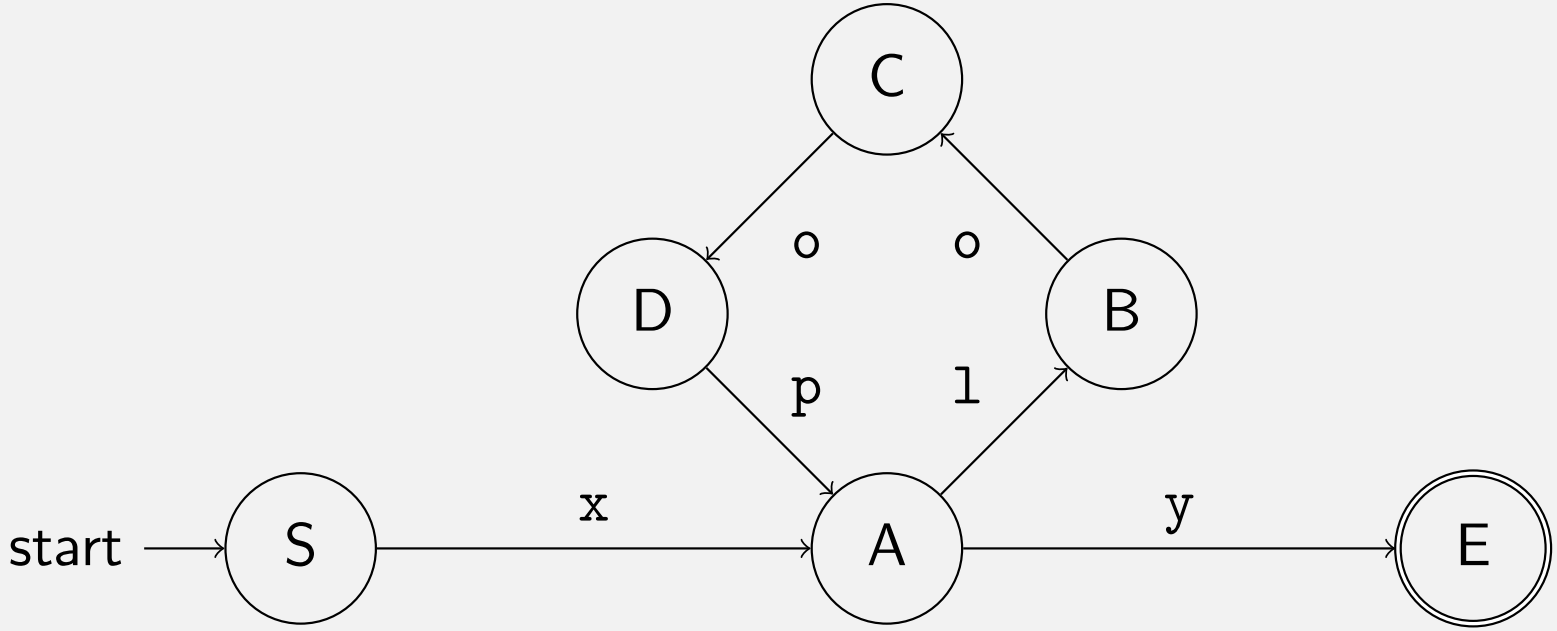
We can traverse it arbitrarily often.

To be more precise: if we have a word that is accepted and traverses the loop once, then the words that follow the same path and traverse the loop any other number of times are also accepted.

Example



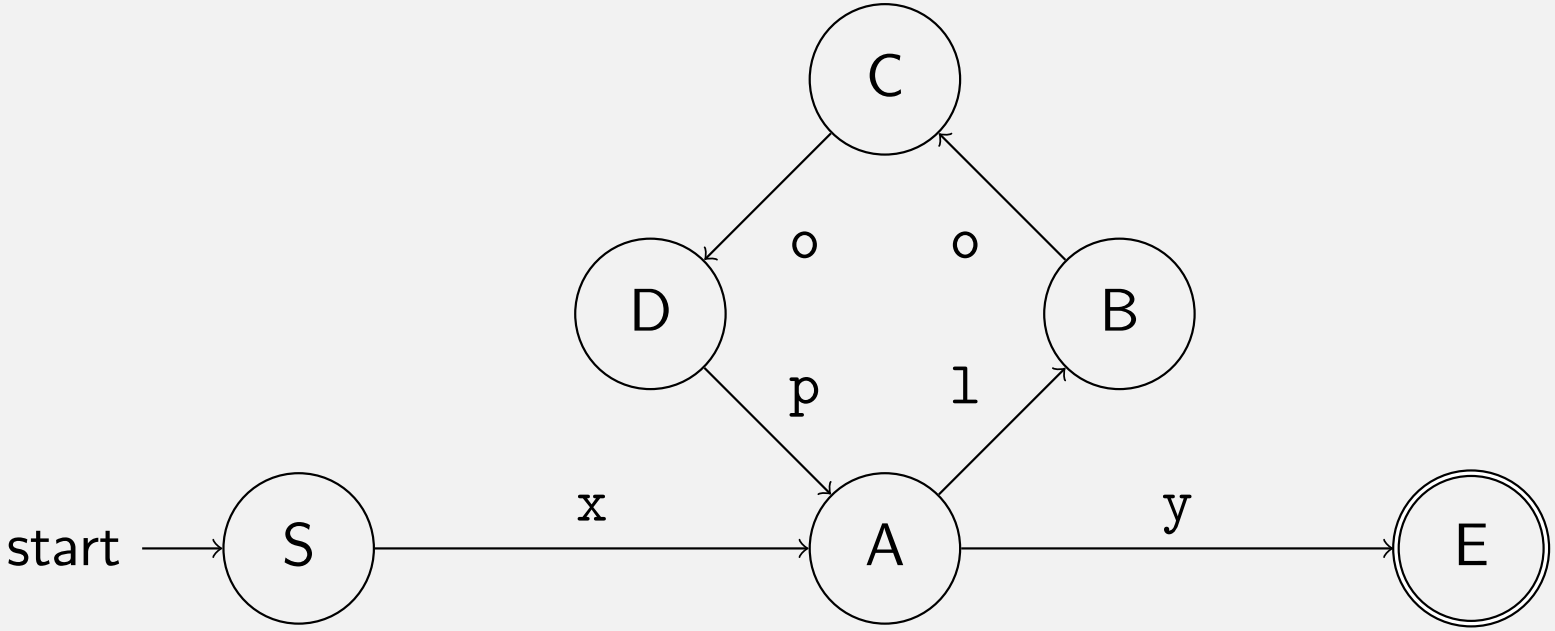
Example



The automaton accepts:

`xloopy` (1 loop traversal)

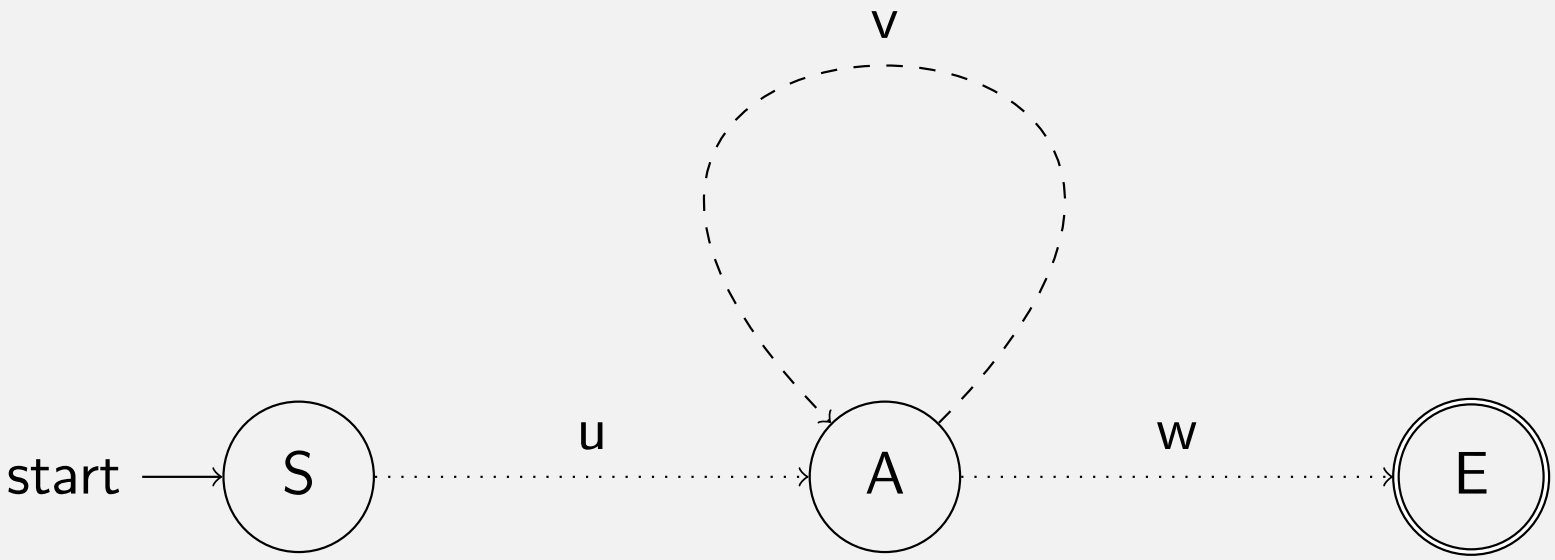
Example



The automaton accepts:

`xy`	(0 loop traversals)
`xloopy`	(1 loop traversal)
`xloopploopy`	(2 loop traversals)
`xloopploopploopy`	(3 loop traversals)
...	

The general situation



- This is an excerpt of the automaton. There may be other nodes and edges.
- Both u and w may be empty (i.e. A and S or A and E may be the same state), but v is not empty – there is a proper loop.
- All words of the form uv^iw for $i \in \mathbb{N}$ are accepted.

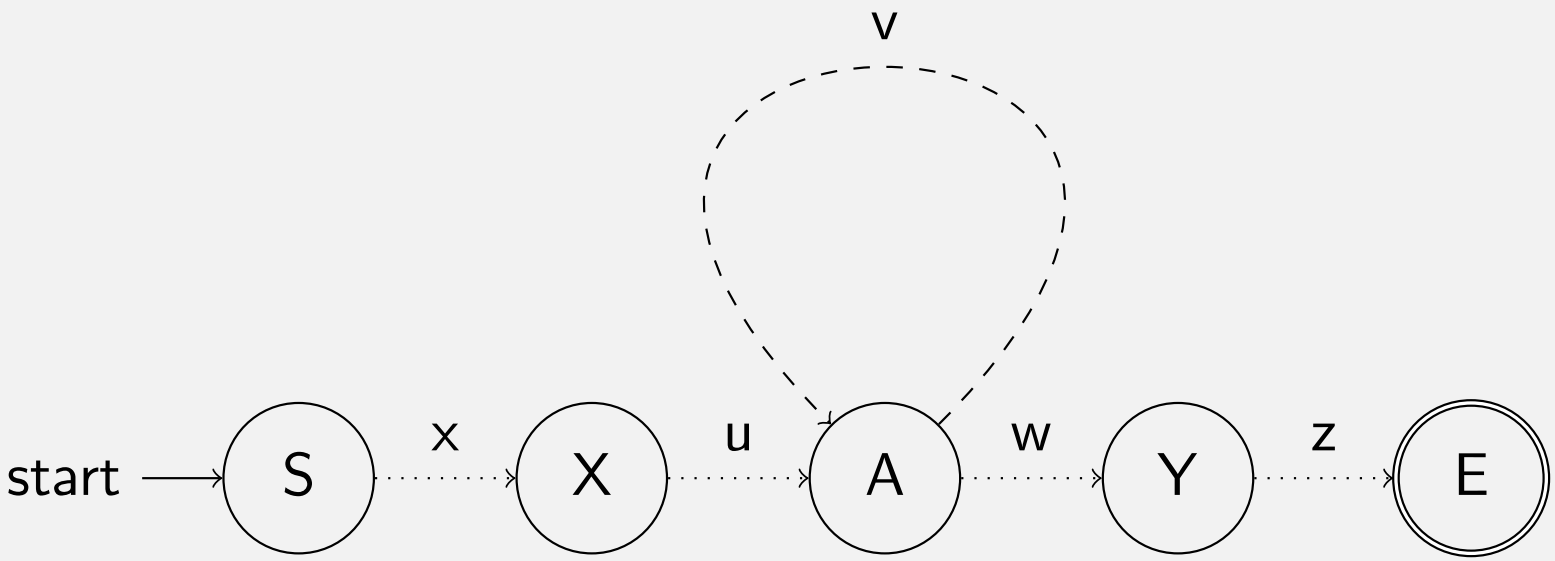
Generalizing even more

A loop has to occur in every **subword** of at least length n:

- Assume we have an accepted word xyz where subword y is of at least length n.

Generalizing even more

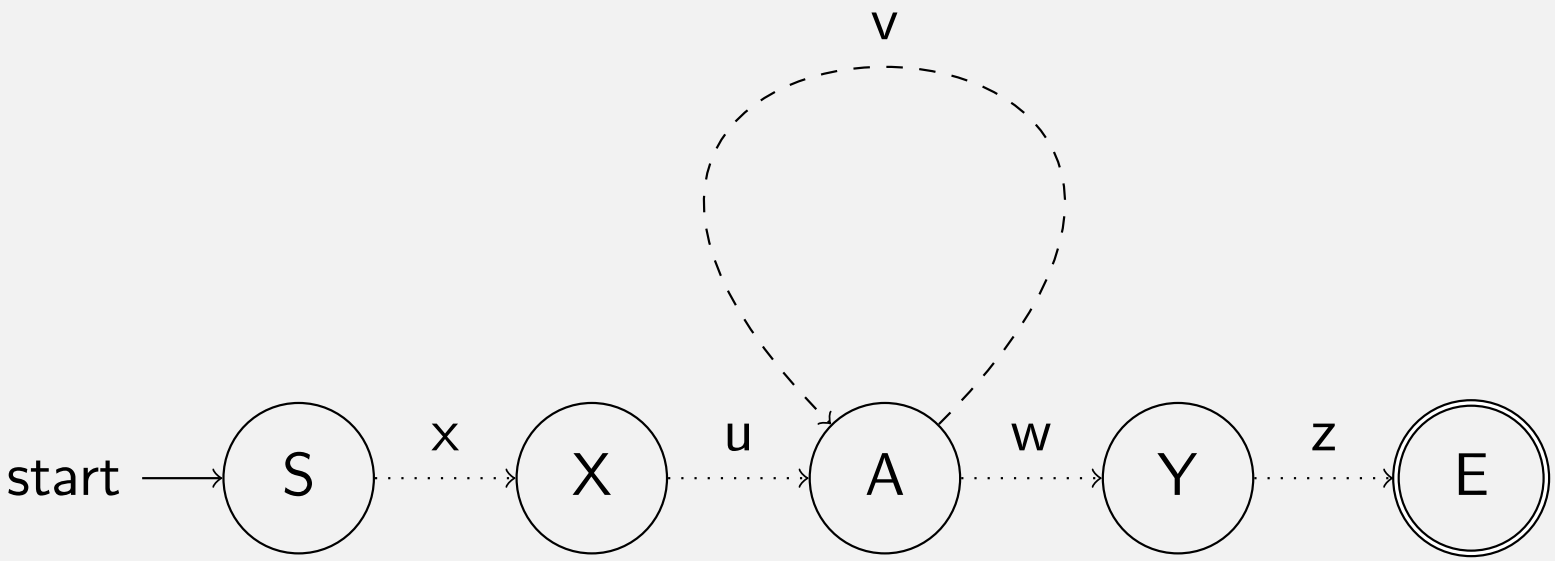
A loop has to occur in every **subword** of at least length n:



- Assume we have an accepted word xyz where subword y is of at least length n.
- Then y has to be of form uvw where v is not empty and corresponds to a loop.

Generalizing even more

A loop has to occur in every **subword** of at least length n:



- Assume we have an accepted word xyz where subword y is of at least length n.
- Then y has to be of form uvw where v is not empty and corresponds to a loop.
- All words of the form xuv^iwz for $i \in \mathbb{N}$ are accepted.

A property of all regular languages

Pumping Lemma for regular languages

For every regular language L,

A property of all regular languages

Pumping Lemma for regular languages

For every regular language L,

there exists an $n \in \mathbb{N}$

- (corresponding to the number of states in the automaton)

A property of all regular languages

Pumping Lemma for regular languages

For every regular language L,

there exists an $n \in \mathbb{N}$

such that for every word xyz in L with $|y| \geqslant n$,

A property of all regular languages

Pumping Lemma for regular languages

For every regular language L,

there exists an n $\in \mathbb{N}$

such that for every word xyz in L with $|y| \geqslant n$,

we can split y into three parts, $y = uvw$, with $|v| > 0$,

A property of all regular languages

Pumping Lemma for regular languages

For every regular language L,

there exists an n $\in \mathbb{N}$

such that for every word xyz in L with $|y| \geqslant n$,

we can split y into three parts, $y = uvw$, with $|v| > 0$,

such that for every $i \in \mathbb{N}$, we have $xuv^iwz \in L$.

The strategy – revisited

We proceed in the following steps:

1. we expose a limitation in the formalism (in this case, in the concept of finite state automata);
2. from this limitation, we derive a property that all languages in the class (in this case, regular languages) must have;
3. therefore, if a language does not have that property, it cannot be in the class.

The strategy – revisited

We proceed in the following steps:

1. we expose a limitation in the formalism (in this case, in the concept of finite state automata);
2. from this limitation, we derive a property that all languages in the class (in this case, regular languages) must have;
3. therefore, if a language does not have that property, it cannot be in the class.

We have done the first two steps. We have found a limitation in the formalism, and derived a property that all regular languages must have.

Using the pumping lemma

In order to show that a language is not regular, we show that it does not have the pumping lemma property as follows:

- We assume that the language is regular.
- We use the pumping lemma to derive a word that must be in the language, but is not:
  - find a word xyz in L with $|y| \geqslant n$,
  - from the pumping lemma there must be a loop in y,
  - but repeating this loop, or omitting it, takes us outside of the language.
- The contradiction means that the language cannot be regular.

Using the pumping lemma – strategy

- For **every** natural number n,
  - because you don't know what the value of n is
- find a word xyz in L with $|y| \geqslant n$ (**you choose** the word),
- such that for **every** splitting $y = uvw$ with $|v| > 0$,
  - because you don't know where the loop may be
- there exists a number i (**you figure out** the number),
- such that $xuv^iwz \notin L$ (you have to **prove** it).

Example

Theorem

The language $L = \{a^m b^m \mid m \in \mathbb{N}\}$ is not regular.

Example

Theorem

The language $L = \{a^m b^m \mid m \in \mathbb{N}\}$ is not regular.

Let us assume that L is regular. Then there must be a finite state automaton that describes L.

Example

Theorem

The language $L = \{a^m b^m \mid m \in \mathbb{N}\}$ is not regular.

Let us assume that L is regular. Then there must be a finite state automaton that describes L.

We do not know what the automaton may be like, but that does not matter. We do know its number of states is **finite**; assume that number is n.

Example – contd.

Consider the word $a^n b^n \in L$.

Example – contd.

Consider the word $a^n b^n \in L$.

Let $x = \varepsilon$, $y = a^n$, $z = b^n$. Then $xyz = a^n b^n \in L$ and $|y| \geqslant n$.

Example – contd.

Consider the word $a^n b^n \in L$.

Let $x = \varepsilon$, $y = a^n$, $z = b^n$. Then $xyz = a^n b^n \in L$ and $|y| \geqslant n$.

From the pumping lemma, we know there must be a loop in y.

Example – contd.

Consider the word $a^n b^n \in L$.

Let $x = \varepsilon$, $y = a^n$, $z = b^n$. Then $xyz = a^n b^n \in L$ and $|y| \geqslant n$.

From the pumping lemma, we know there must be a loop in y.

Let $y = uvw$ where $|v| > 0$ and $xuv^i wz \in L$ for all $i \in \mathbb{N}$.

Example – contd.

Consider the word $a^n b^n \in L$.

Let $x = \varepsilon$, $y = a^n$, $z = b^n$. Then $xyz = a^n b^n \in L$ and $|y| \geqslant n$.

From the pumping lemma, we know there must be a loop in y.

Let $y = uvw$ where $|v| > 0$ and $xuv^i wz \in L$ for all $i \in \mathbb{N}$.

Let $u = a^p$, $v = a^q$, $w = a^r$ where $p + q + r = n$, $q > 0$.

Example – contd.

Consider the word $a^n b^n \in L$.

Let $x = \varepsilon$, $y = a^n$, $z = b^n$. Then $xyz = a^n b^n \in L$ and $|y| \geqslant n$.

From the pumping lemma, we know there must be a loop in y.

Let $y = uvw$ where $|v| > 0$ and $xuv^i wz \in L$ for all $i \in \mathbb{N}$.

Let $u = a^p$, $v = a^q$, $w = a^r$ where $p + q + r = n$, $q > 0$.

If $i = 2$, then $xuv^2wz = a^p a^{2q} a^r b^n = a^{n+q} b^n$.

Example – contd.

Consider the word $a^n b^n \in L$.

Let $x = \varepsilon$, $y = a^n$, $z = b^n$. Then $xyz = a^n b^n \in L$ and $|y| \geqslant n$.

From the pumping lemma, we know there must be a loop in y.

Let $y = uvw$ where $|v| > 0$ and $xuv^i wz \in L$ for all $i \in \mathbb{N}$.

Let $u = a^p$, $v = a^q$, $w = a^r$ where $p + q + r = n$, $q > 0$.

If $i = 2$, then $xuv^2wz = a^p a^{2q} a^r b^n = a^{n+q} b^n$.

But $n + q \not\equiv n$ because $q > 0$! So $xuv^2wz \notin L$.

Example – contd.

Consider the word $a^n b^n \in L$.

Let $x = \varepsilon$, $y = a^n$, $z = b^n$. Then $xyz = a^n b^n \in L$ and $|y| \geqslant n$.

From the pumping lemma, we know there must be a loop in y.

Let $y = uvw$ where $|v| > 0$ and $xuv^i wz \in L$ for all $i \in \mathbb{N}$.

Let $u = a^p$, $v = a^q$, $w = a^r$ where $p + q + r = n$, $q > 0$.

If $i = 2$, then $xuv^2wz = a^p a^{2q} a^r b^n = a^{n+q} b^n$.

But $n + q \not\equiv n$ because $q > 0$! So $xuv^2wz \notin L$.

So our original assumption must have been wrong: the language L cannot be regular.

Exercise

For each of these languages:

- if it is regular, give an automaton or regular expr. for it;
- if not, use the pumping lemma to prove it.

1. $L = \{a^m b^n \mid m, n \in \mathbb{N}\}$
2. $L = \{a^m b^n \mid m, n \in \mathbb{N}, m < n\}$
3. $L = \{a^m b^n \mid m, n < 1000, m < n\}$