

# Grundlagen der Informatik

## VO und KV

Formale Sprachen,  
Reguläre Ausdrücke und  
Automaten



# Überblick

---

**Formale Sprachen**

**Syntaxdiagramm**

**Endliche Automaten (finite-state automaton)**

**Reguläre Ausdrücke (regular expressions)**

# REGULAR EXPRESSIONS



# Es gibt unterschiedliche Sprachen

---

## Natürliche Sprachen

hallo

$$\Sigma = \{a \dots z\}$$

## DNA

$$\Sigma = \{a, c, g, t\}$$

## Morse

-- --- .- . . . / -.- --- -.- .

$$\Sigma = \{-, \cdot, /\}$$

## Programmiersprachen

```
if temperature > 70: print('Wear shorts.')  
else: print('Wear long pants.')
```

$$\Sigma = \{\text{if}, >, \text{print}, \dots\}$$

## Binär

$$\Sigma = \{0, 1\}$$

# Formale Sprachen - Definitionen

---

<b>Sprache</b>	...	beschreibt eine beliebige Menge von Wörtern.
<b>Alphabet (<math>\Sigma</math>)</b>	...	ist eine endliche, nicht leere Menge von Zeichen. Jedes $a \in \Sigma$ heißt Buchstabe oder Terminalsymbol.
<b>Wort (<math>w</math>)</b>	...	besteht aus mindestens einem Terminalsymbol eines Alphabets, endliche Folge aus einem Alphabet.
<b>Konkatenation</b>	...	das “Hintereinanderschreiben” der Zeichen zweier Wörter $w_1, w_2$
<b>Leeres Wort (<math>\epsilon</math>)</b>	...	ist ein Wort, $\{\epsilon\}$ , $ \epsilon  = 0$ , $ \text{hallo}  = 5$

- Formale Sprachen können leer, endlich oder unendlich sein.
- Nicht Kommunikation, sondern mathematische Verwendung im Vordergrund.
- Es gibt keine Information über Bedeutung!

# Eine einfache Sprache

## Wortproblem:

“Gegeben sei eine Sprache  $S$ . Ermittle, ob ein Wort  $W$  Teil von  $S$  ist oder nicht.”

**Sprache lässt sich als Menge repräsentieren.**

**$S = \{“a”\}$**

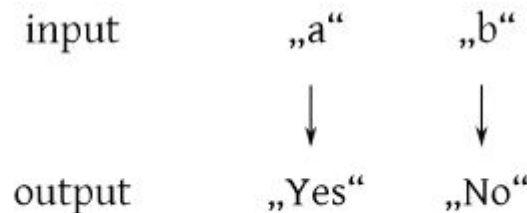


Abbildung 11.1: Zwei Beispiele für die Entscheidung des Wortproblems für die Sprache „a“.

# Formale Sprachen

---

Sie können über eine mathematische Bedingung an ihre Wörter definiert sein: „Die Sprache ... ist die Menge aller Wörter, für die gilt ...“.

$$\Sigma = \{a, b, c\}$$

$$S1 = \{aa, bb, cc, aab, cab, a\}$$

Sprache S1 hat 6 Wörter

$$S2 = \{a^n, b^n, c^n \mid n \in \mathbb{N}\}$$

abc, aabbcc, aaabbbccc

...

# Zwei grundlegende Möglichkeiten formale Sprachen zu beschreiben

---

## Über die Definition einer Grammatik

... um eindeutig festzulegen, ob ein Wort Element einer Sprache ist und zum anderen, um Eigenschaften dieser formalen Sprachen zu untersuchen bzw. zu beweisen.

[https://de.wikipedia.org/wiki/Formale\\_Grammatik](https://de.wikipedia.org/wiki/Formale_Grammatik)

## Über Automaten

... die ein Wort Zeichen für Zeichen analysieren und akzeptieren, wenn es einer Sprache zugehörig ist.

(→ Compiler)

[https://de.wikipedia.org/wiki/Automat\\_\(Informatik\)](https://de.wikipedia.org/wiki/Automat_(Informatik))



# Formale Sprache der Gleitkommazahlen

---

Erstellen wir ein **Syntaxdiagramm** (abstrakte Darstellung eines Automaten!) für alle erlaubten Wörter der Gleitkommazahlen mit wissenschaftlicher Notation.

Ein **Syntaxdiagramm** ist ein Weg, um kontextfreie Sprache (kontextfreie Grammtiken) zu repräsentieren.

Folgende Wörter sind **erlaubt** in dieser Sprache:

1.0e3   -42.   +0.43   156E+10.   .73   2.3e-5   2.3e-5.2   ...

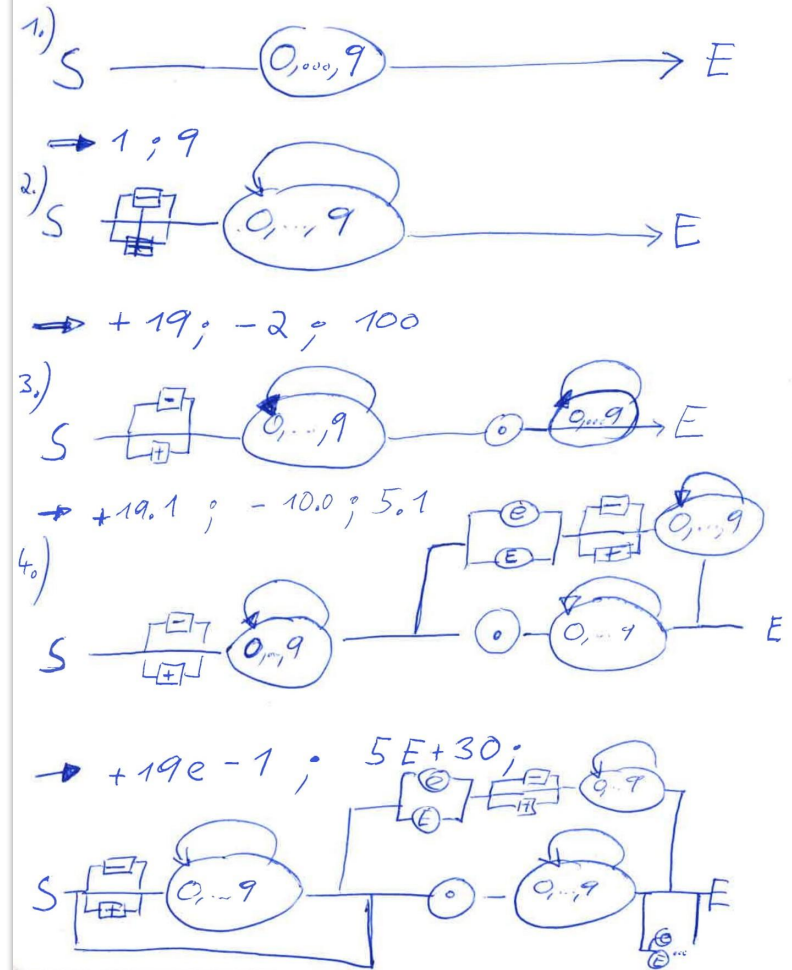
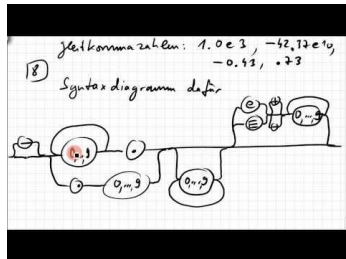
[2.3e-5 = 2.3 \* 10 hoch -5 = 0.000023]

Folgende Wörter sind **verboten** in dieser Sprache:   -e.45,   EE123,   ++,   ...

# Gleitkommazahlen

Folgende Wörter sind **erlaubt** in unserer Sprache:

1.0e3, -42. , +0.43, 156E+10., .73,  
2.3e-5

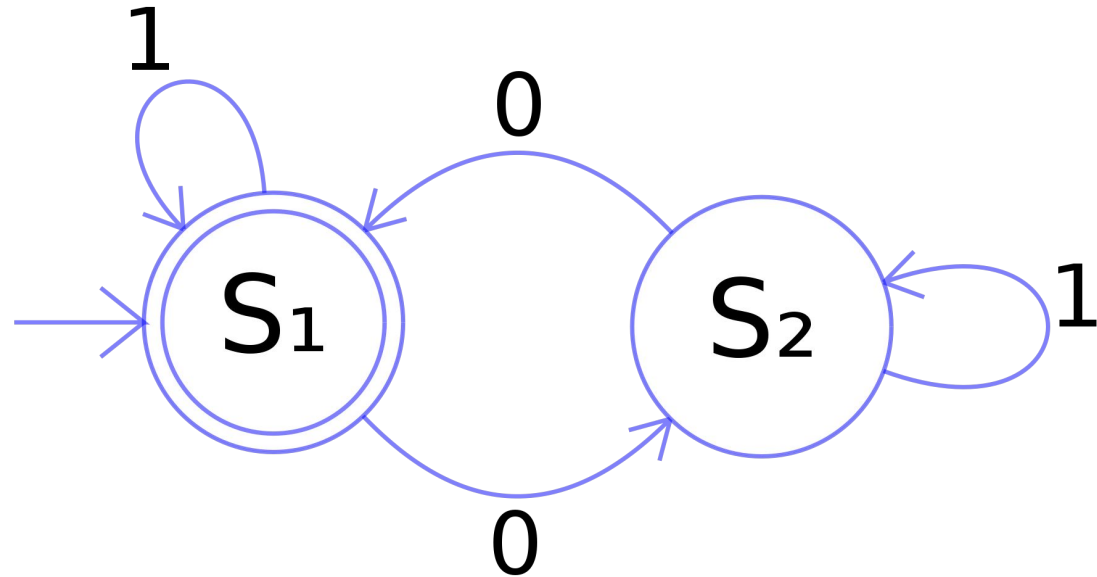


# Endlicher Automat

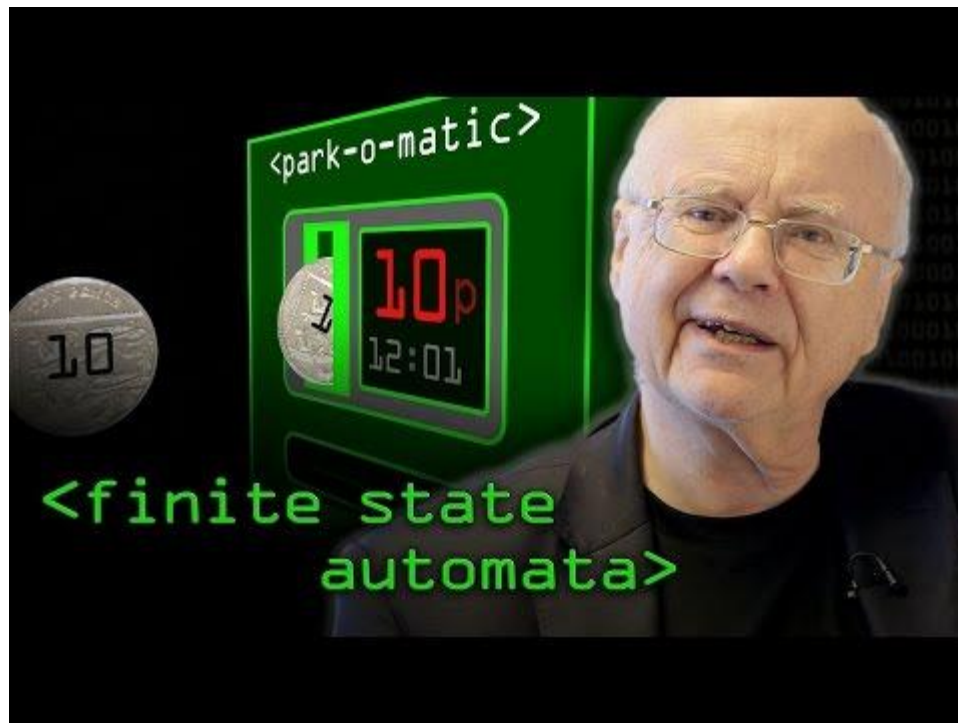
Dieser endliche Automat überprüft, ob eine Binärzahl eine gerade oder ungerade Anzahl von 0en hat:

**Input: 10001**  
**not accepted**

**Input: 1001**  
**accepted**

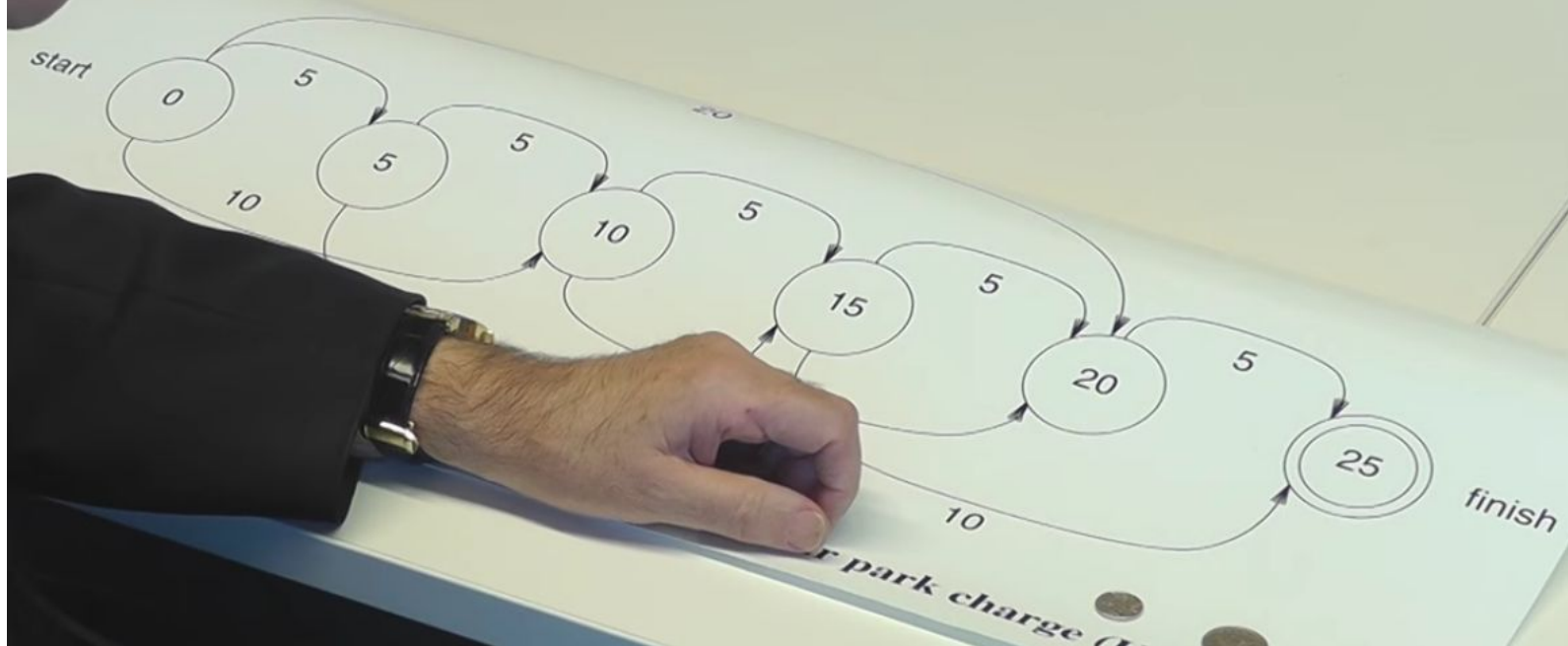


- Startzustand
- Gerichtet
- Endzustände



## Computers Without Memory - Computerphile

[https://en.wikipedia.org/wiki/Finite-state\\_machine](https://en.wikipedia.org/wiki/Finite-state_machine)



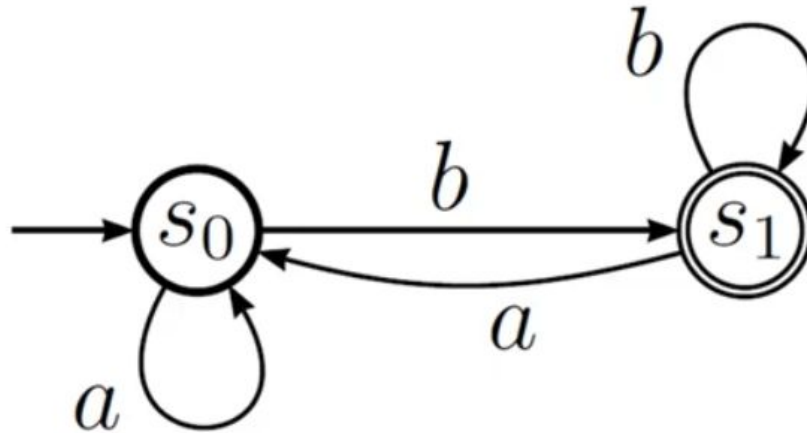
Ein **Endlicher Automat** analysiert ein Wort einer Sprache Zeichen für Zeichen und entscheidet, ob sie zur Sprache gehört oder nicht. Kein Speicher ist dafür notwendig. Wird beispielsweise zum Überprüfen von Variablennamen verwendet.

<b>Alphabet</b>	...	5,10, 20
<b>Wörter</b>	...	5 5 5 5 5, 5 10 10, 20 5, 10 5 5 5...
<b>Zustand</b>	...	0, 5, 10, 15, 20
<b>Endzustand</b>	...	25

# Deterministischer endlicher Automat

Ein deterministischer endlicher Automat (DEA, DFA) ist ein 5-Tupel.

Er besteht aus einer nichtleeren, endlichen Menge  $S$  von Zuständen, einem Alphabet, einem Startzustand, einem Endzustand und einer total definierten Überföhrungsfunktion.



Zustandsübergangsdiagramm

# Reguläre Ausdrücke: RegEx (regular expressions)

- Werden durch reguläre Grammatiken erzeugt.
- Zu jedem regulären Ausdruck existiert ein endlicher Automat, der die vom Ausdruck spezifizierte Sprache akzeptiert.
- z.B. die Sprache aller Wörter, die aus beliebig vielen a oder beliebig vielen b und der leeren Menge besteht:

**Alphabet:**  $\{a,b\}$

**Wörter:** ab, a, b, aaaaa, bbbbbb, ababab,  $\epsilon$

**RegEx:**  $(a^*b^*)^*$ ,  $a^+$ ,  $b^?$ , ab, a, b

## Verwendung von RegEx

**R1R2** wenn R1, R2 sind RegEx

**(R1|R2)** ODER

**(R1)\***

**Konkateniert**

Reguläre Ausdrücke  $\Sigma = \{a,b\}$   
 z.B.  $a^*bca^*$  bc aabca aaaaa  
 $a(bcb)a$  abca abca  
 $x \in \Sigma$  a  
 $\epsilon$



# Quantoren

---

**Beziehen sich immer auf voranstehende Ausdrücke (bzw. Klammern)**

- ?** ist optional, kommt null- oder einmal
- +** mindestens einmal vorkommen, ein- oder mehrfach
- \*** beliebig oft, kein- oder mehrfach
- ()** Zusammenfassen von Termen
- |** Alternative



# RegEx

---

$a^+$	aaaa, a, aaa
$ab^?$	a, ab
$(ab)^?$	ab, $\epsilon$
$abab^*$	ababbbbbbb, aba, ababb
$(aab^+)^*$	aab, aabbbb, aabaab, aabbaabbbaab, $\epsilon$
$(a^+b^+ \mid (bba)^+)^+$	aaabbb, bbabbabba, abbbbaabbbbabba

# RegEx als endlichen Automaten

---

Automaten für:

$a^+$

$ab^?$

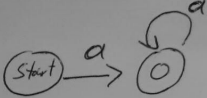
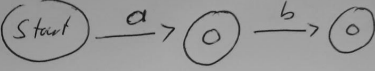
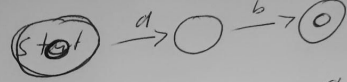
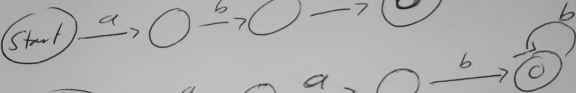
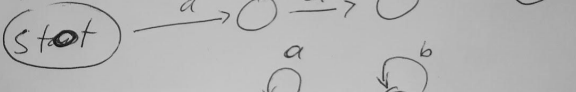
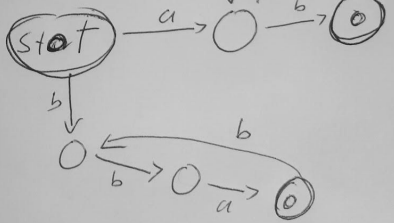
$(ab)^?$

$abab^*$

$(aab^+)^*$

$a^+b^+|(bba)^+$

# RegEx - Endliche Automaten

RegEx	endlicher Automat
$a^+$	
$ab^?$	
$(ab)^?$	
$abab^*$	
$a(ab^+)^*$	
$a^+b^+   (bba)^+$	

Automaten für:

$a^+$

$ab^?$

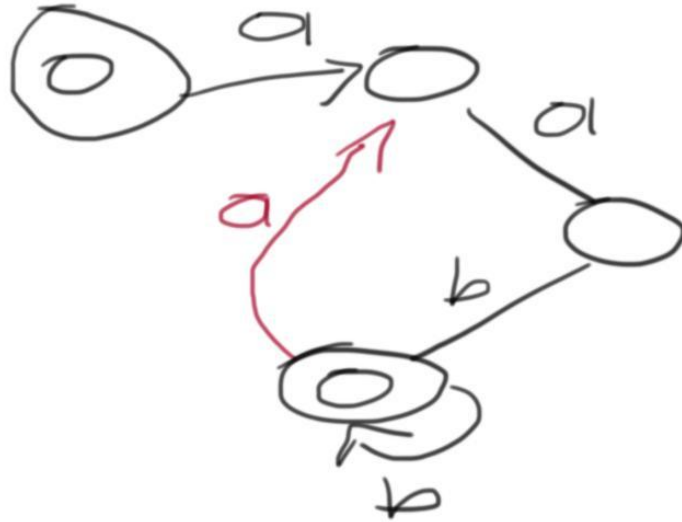
$(ab)^?$

$abab^*$

$(aab^+)^*$

$(a+b^+|(bba)^+)$

(in der Darstellung fehlt die Schleife, die durch den geklammerten Term und dem  $*$  Operator gegeben ist)

$$(a^+b^+)^*$$


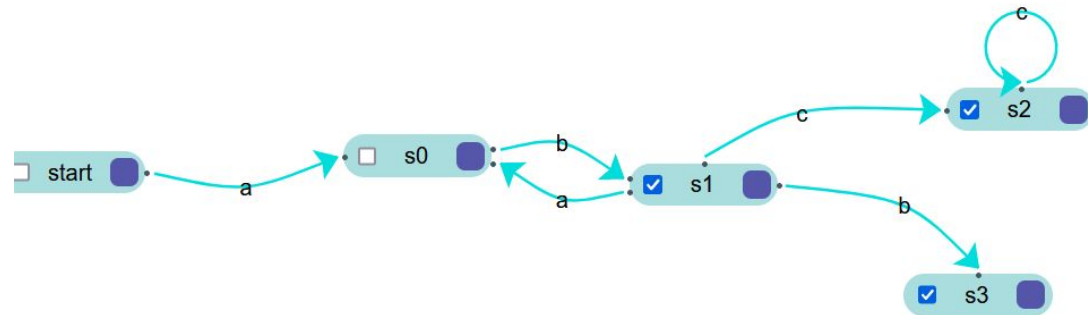
<https://automatonsimulator.com/>

**1.2 )** Zeichne einen deterministischen Automaten, der genau die Wörter akzeptiert, mit denen der in angeführte regulärer Ausdruck matcht. Vergiss nicht, mindestens einen Knoten durch einen doppelten Rand als Terminalknoten zu kennzeichnen.

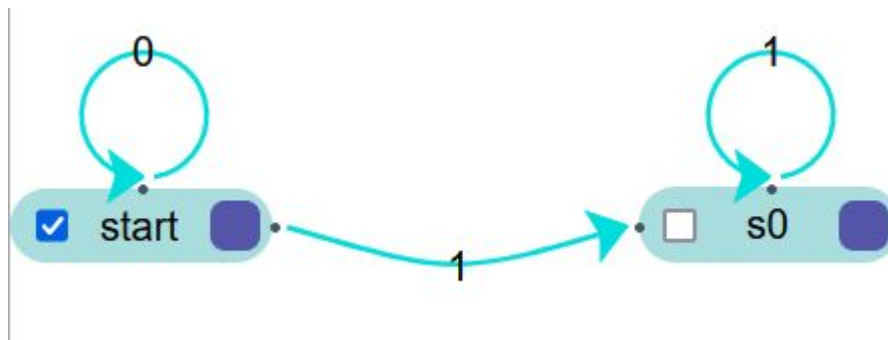
$(ab)^+(cc^*|b?)$

**Input:**

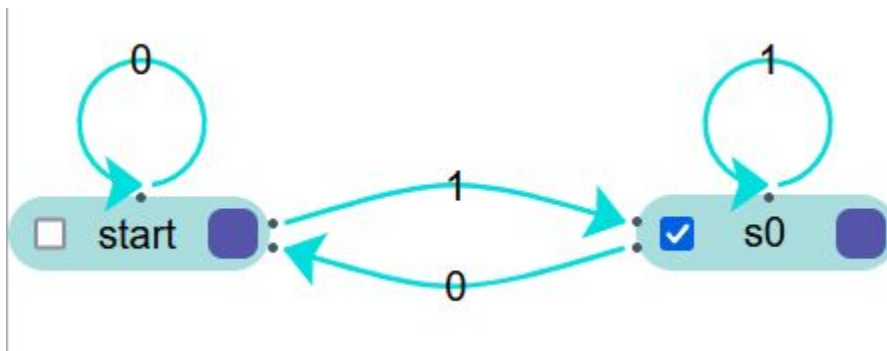
ababc  
abcccc  
abb  
ab  
abbb



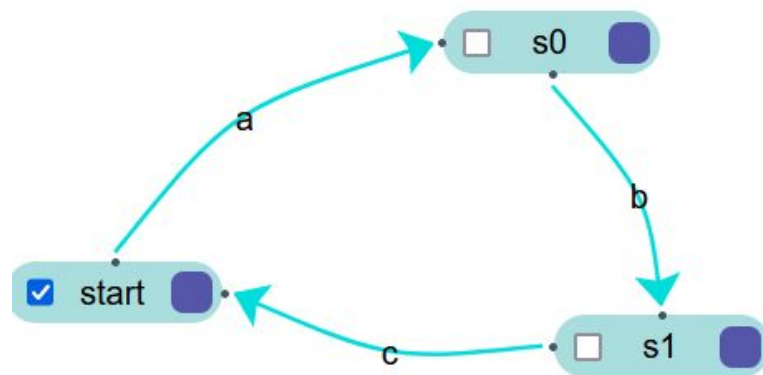
Gib einen DEA an, der alle Bitstrings akzeptiert, die keine 1 enthalten.



Welche Sprache akzeptiert dieser DEA?



Gib einen DEA an, der die Menge {leeres Wort, abc, abcabc, abcabcabc,...  
akzeptiert}






# Nichtdeterministischer Automat (NDA)

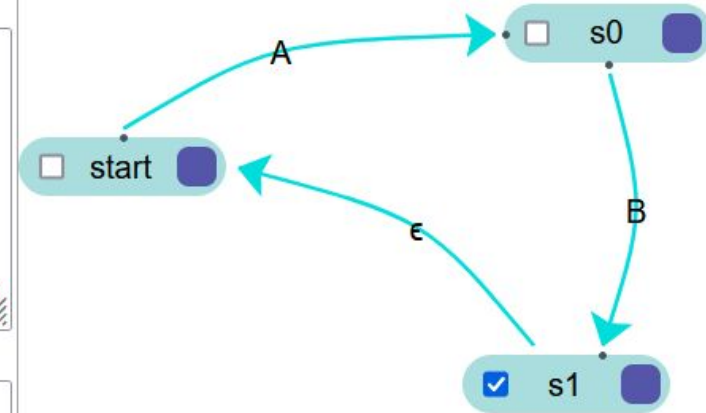
Ein nichtdeterministischer endlicher Automat (NEA; englisch nondeterministic finite automaton, NFA) ist ein endlicher Automat, bei dem es für den Zustandsübergang mehrere gleichwertige Möglichkeiten gibt. Im Unterschied zum deterministischen endlichen Automaten sind die Möglichkeiten nicht eindeutig, dem Automaten ist also nicht vorgegeben, welchen Übergang er zu wählen hat.

[https://de.wikipedia.org/wiki/Nichtdeterministischer\\_endlicher\\_Automat](https://de.wikipedia.org/wiki/Nichtdeterministischer_endlicher_Automat)

Bulk Testing 

Accept (one per line):  
AB  
ABAB  
ABABAB

Reject (one per line):  
A  
B  
ABA  
BA  
BB  
ABABB



Test Results:

# Übung RegEx matcht?

---

Gib an welche der folgenden Terme auf den angegebenen Regulären Ausdruck matchen.

Regulärer Ausdruck: **( (ac)+ | (bb)?d )+**

Term	matcht?
acacacacac	
acbbd	
d	
acb	

# RegEx konstruieren

---

Konstruiere einen RegEx der auf folgende Terme matcht:

$\varepsilon$ , eeee, de, deee, deef

Lösung:

$(d?e+f?)?$       oder       $d^*e^*f^*$       oder...

# RegEx als Automaten zeichnen

---

Zeichne einen endlichen Automaten für  **$(d^*e^+)f?$**

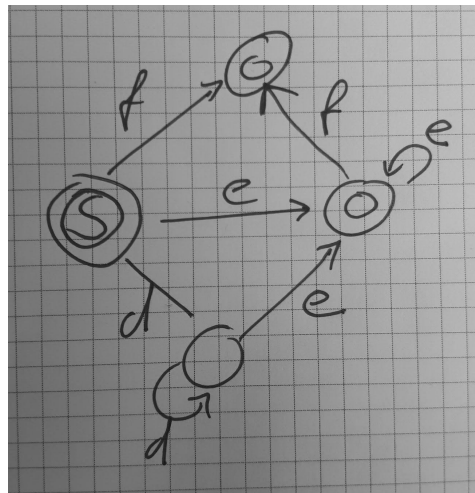
Füge sowohl Startzustand (S), als auch alle Endzustände (Doppelkreis) an.

# RegEx als Automaten zeichnen

Zeichne einen endlichen Automaten für

**$(d^*e^+)f?$**

Füge sowohl Startzustand (S), als auch alle Endzustände (Doppelkreis) an.



# Weitere RegEx Operatoren

---

## Vordefinierte Zeichenklassen:

<b>\d</b>	<b>digit</b>	eine Ziffer, [0-9] (und evtl. auch weitere Zahlzeichen in Unicode)
<b>\D</b>	no <b>digit</b>	ein Zeichen, das keine Ziffer ist, also [^\d]
<b>\w</b>	<b>word character</b>	ein Buchstabe, eine Ziffer oder der Unterstrich, also [a-zA-Z_0-9]
<b>\W</b>	no <b>word character</b>	ein Zeichen, das weder Buchstabe noch Zahl noch Unterstrich ist, also [^\w]
<b>\s</b>	<b>whitespace</b>	meist zumindest das Leerzeichen und die Klasse der Steuerzeichen \n, \r, \t
<b>\S</b>	no <b>whitespace</b>	ein Zeichen, das kein Whitespace ist, also [^\s]

[[https://de.wikipedia.org/wiki/Regul%C3%A4rer\\_Ausdruck](https://de.wikipedia.org/wiki/Regul%C3%A4rer_Ausdruck)]

# Online RegEx Check-Tool

---

<https://regexr.com>

folgende Dinge:

- Alle Zeichen
- Alle Zahlen
- Alle Wörter, alle Wörter die mit 'B' beginnen
- Alle URI's
- Alle Datumsangaben [01.01.1991]
- Alle Wörter
- Alle Zahlen

`a{}` ... so viele Zeichen

`(a|b)` ... a oder b

`[a-g]` ... Zeichen zwischen a und g

`^` ... not

# Lösungen

---

- `\w`
- `\d`
- `\w+`, `\bb\w+`
- `http:\V(\w|\d|\.|\\b|\\V)+`  
(bis zu ?, ? oft ein Parameter)
- `\\b\\d{2}\\.|\\d{2}\\.|\\d{4}\\b`



# Kurze Zusammenfassung

---

- Woher weiß ein Algorithmus, ob eine Eingabe valide ist (~Programmiersprachen und Wortproblem)
  - → Formale Sprachen
- Reguläre Ausdrücke vs. Deterministische Automaten
- Beispiel einer Sprache, die **nicht** regulär ist:  
 $L = \{ a^n b^n \mid n \in \mathbb{N} \} = \{ ab, aabb, aaabbb, aaaabbbb, \dots \}$
- Es gibt keinen regulären Ausdruck der diese Sprache beschreibt, weil man das “*es gibt genau so viele a’s wie b’s*” nicht in einem RegEx ausdrücken kann.
- $a+b+$  matcht zwar auf  $aaabbb$ , aber auch auf  $abbb$ .
- Anders gesagt: Es gibt keinen deterministischen endlichen Automaten, der genau die Sprache  $a^n b^n$  erkennt.

# Kontextfreie Sprachen

- $L = \{ a^n b^n \mid n \in \mathbb{N} \} = \{ ab, aabb, aaabbb, aaaabbbb, \dots \}$   
ist eine kontextfreie Sprache. Wir brauchen also eine kontextfreie Grammatik, um diese Sprache erzeugen zu können.

S	→ aSb	$G = (V, \Sigma, P, S)$
	→ aaSbb	V ... endliches Vokabular = Variablen = Nonterminale
	→ aaaSbbb	$\Sigma$ ... ein Alphabet (Terminale)
	→ aaaSbbb	P ... Produktionsregeln
	→ aaaSbbb	S ... Startsymbol
	→ aaabbbb	$L = \{S, \{a,b\}, \{S \rightarrow aSb, S \rightarrow \epsilon\}, S\}$

# Kontextsensitive Sprachen

$$H = \{a^n b^n c^n \mid n \in \mathbb{N}, n \geq 0\}$$

```
S → aSbB
S →
B → Bc
B →
```

Die Grammatik in Listing 11.15 ist inkorrekt. Mithilfe der Ableitung

$S \rightarrow aSbB \rightarrow abB \rightarrow abBc \rightarrow abBcc \rightarrow abcc$

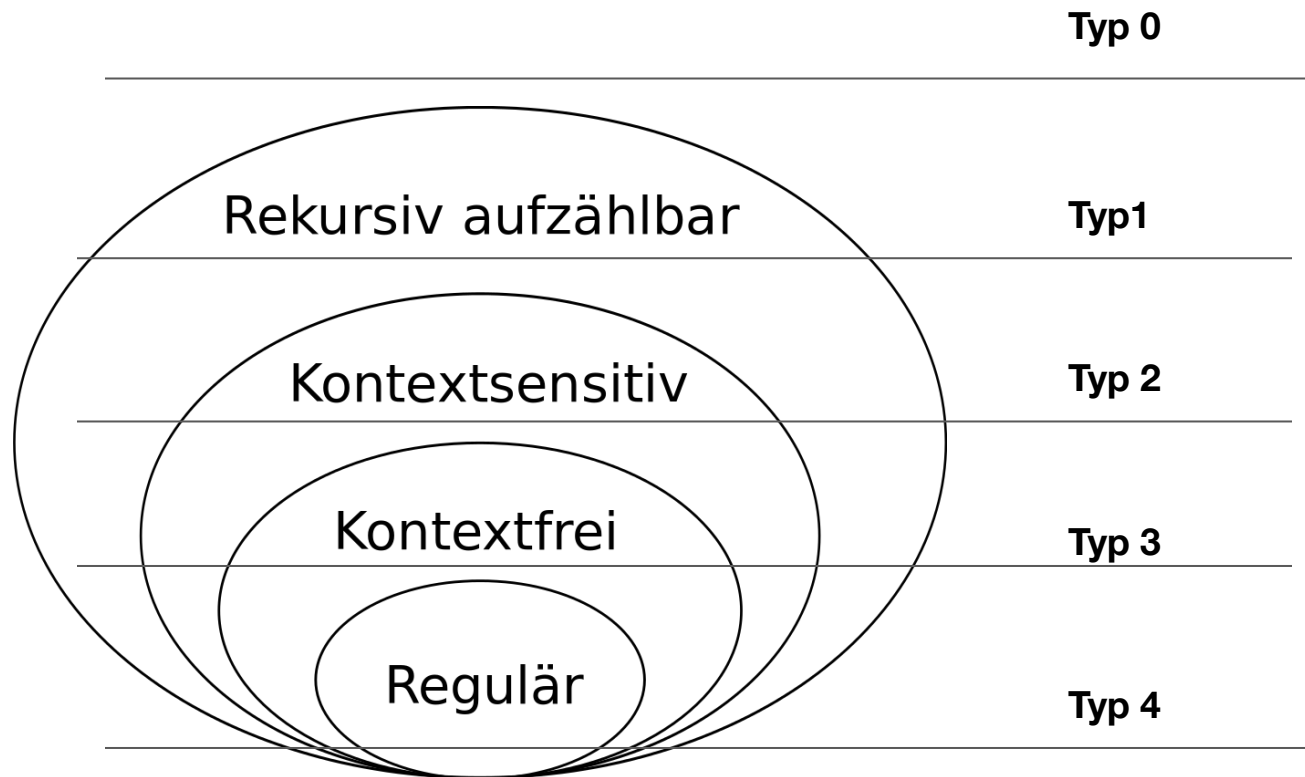
```
S → aSBC
S →
aB → ab
bB → bb
bC → bc
CB → BC
bC → bc
cC → cc
```

→ jetzt haben wir links  
und rechts Terminale  
und Nonterminale  
Symbole

→ wir können  
schleifen-artige Dinge  
bauen

→ es wird  
komplizierter

# Chomsky-Hierarchie



Grammatik	Regeln	Sprachen	Entscheidbarkeit	Automaten	Abgeschlossenheit <sup>[2]</sup>	Zeitabschätzung
<b>Typ-0</b> Beliebige formale Grammatik	$\alpha \rightarrow \beta$ $\alpha \in V^* \setminus T^*, \beta \in V^*$	rekursiv aufzählbar (nicht „nur“ rekursiv, die wären entscheidbar!)	–	Turingmaschine (egal ob deterministisch oder nicht-deterministisch)	$\circ, \cap, \cup, *$	unbeschränkt
<b>Typ-1</b> Kontextsensitive Grammatik	$\alpha A \beta \rightarrow \alpha \gamma \beta$ $A \in N, \alpha, \beta \in V^*, \gamma \in V^+$ $S \rightarrow \varepsilon$ ist erlaubt, wenn es keine Regel $\alpha \rightarrow \beta S \gamma$ in $P$ gibt.	kontextsensitiv	Wortproblem	linear platzbeschränkte nichtdeterministische Turingmaschine	$\mathbb{C}, \circ, \cap, \cup, *$	$O(2^n)$
<b>Typ-2</b> Kontextfreie Grammatik	$A \rightarrow \gamma$ $A \in N, \gamma \in V^*$	kontextfrei	Wortproblem, Leerheitsproblem, Endlichkeitsproblem	nichtdeterministischer Kellerautomat	$\circ, \cup, *$	$O(n^3)$
<b>Typ-3</b> Reguläre Grammatik	$A \rightarrow aB$ (rechtsregulär) oder $A \rightarrow Ba$ (linksregulär) $A \rightarrow a$ $A \rightarrow \varepsilon$ $A, B \in N, a \in T$ Nur links- oder rechtsreguläre Produktionen	regulär	Wortproblem, Leerheitsproblem, Endlichkeitsproblem, Äquivalenzproblem	Endlicher Automat (egal ob deterministisch oder nicht-deterministisch)	$\mathbb{C}, \circ, \cap, \cup, *$	$O(n)$

# A BNF grammar for the “Furry Dice” language

---

`<sentence> ::= <subject> <verb> <object>`

`<subject> ::= <article> <noun> | the robot`

`<article> ::= the | a`

`<noun> ::= dog | cat | man | woman | robot`

`<verb> ::= bit | kicked | stroke`

`<object> ::= <article> <noun> | two furry dice`

Kann man den Satz “*The robot stroked the dog*” mit der gegebenen Grammatik parsen? Parsen in dem Sinne: kann ich den Satz dekodieren?

`<sentence> ::= <subject> <verb> <object>`

`<subject> ::= <article> <noun> | the robot`

`<article> ::= the | a`

`<noun> ::= dog | cat | man | woman | robot`

`<verb> ::= bit | kicked | stroke`

`<object> ::= <article> <noun> | two furry dice`

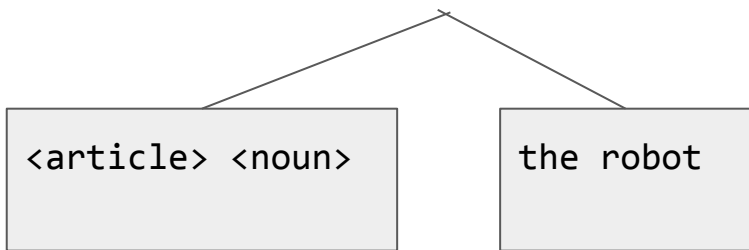
# Parse Tree

Top Down vs. Bottom Up

-

***“The robot stroked the dog”***

`<sentence> ::= <subject> <verb> <object>`



`<sentence> ::= <subject> <verb> <object>`

`<subject> ::= <article> <noun> | the robot`

**`<subject>`**  
`<verb>`  
`<object>`

Stack



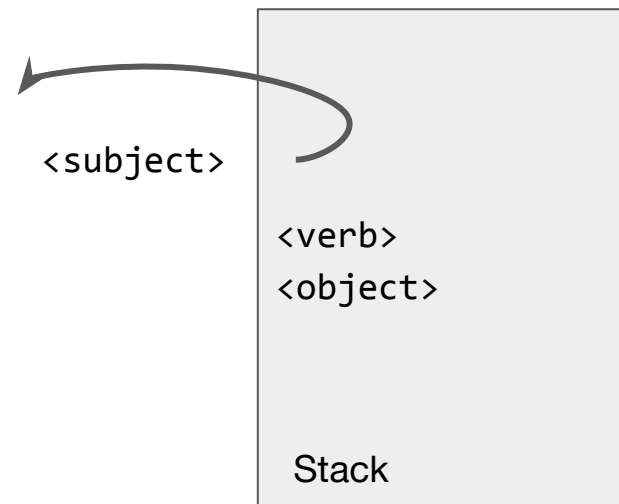
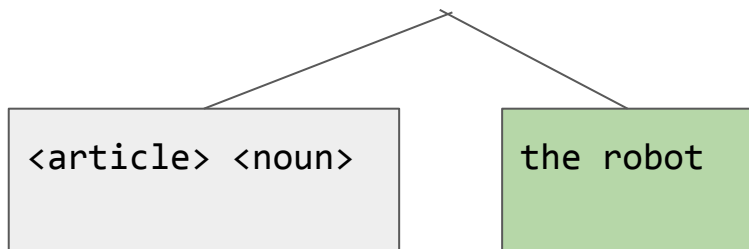
# Parse Tree

Top Down vs. Bottom Up

-

*“The robot stroked the dog”*

`<sentence> ::= <subject> <verb> <object>`



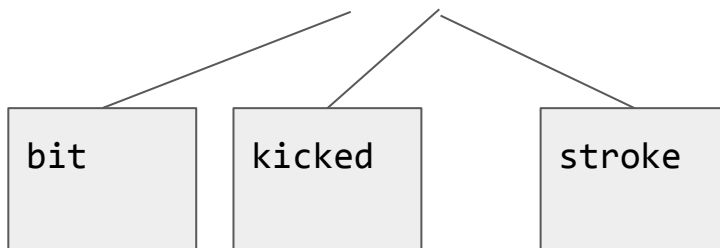
# Parse Tree

Top Down vs. Bottom Up

-

*“The robot **stroked** the dog”*

`<sentence> ::= <subject> <verb> <object>`



`<verb> ::= bit | kicked | stroke`

`<verb>`  
`<object>`

Stack

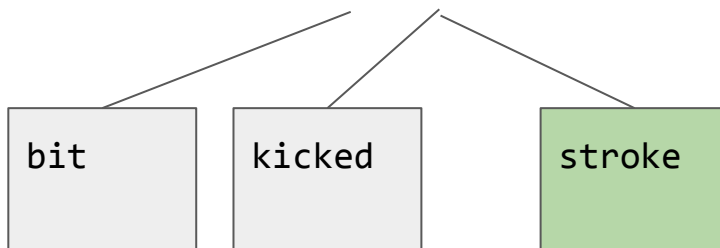
# Parse Tree

Top Down vs. Bottom Up

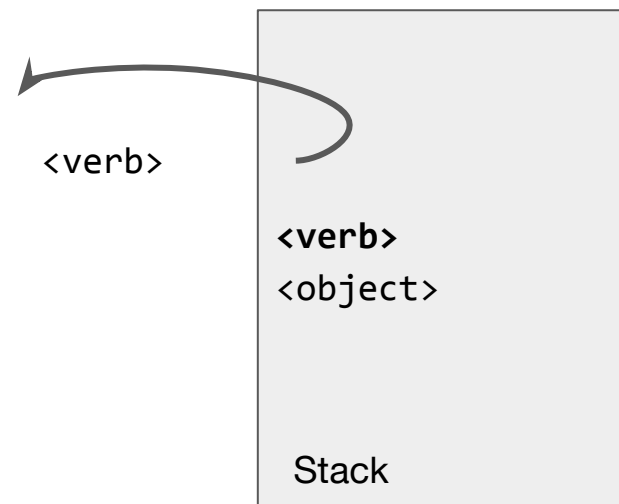
-

*“The robot **stroked** the dog”*

`<sentence> ::= <subject> <verb> <object>`



`<verb> ::= bit | kicked | stroke`



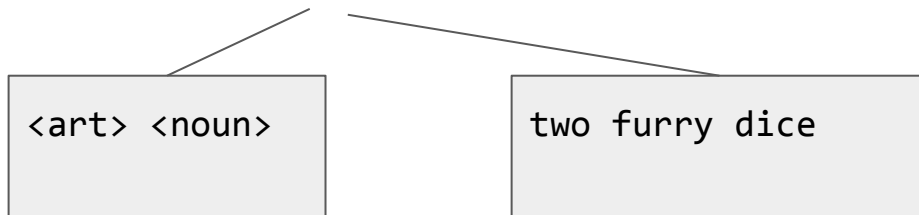
# Parse Tree

Top Down vs. Bottom Up

-

*“The robot stroked **the dog**”*

`<sentence> ::= <subject> <verb> <object>`



`<object> ::= <art> <noun> | two furry dice`

`<article> ::= the | a`

`<verb> ::= bit | kicked | stroke`



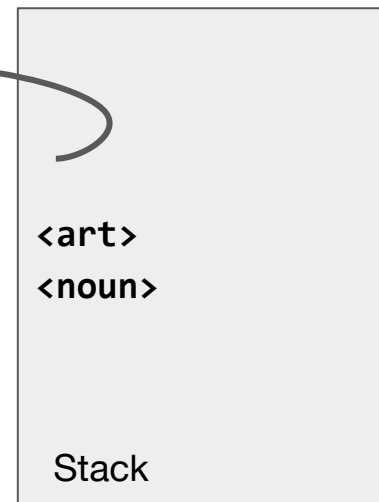
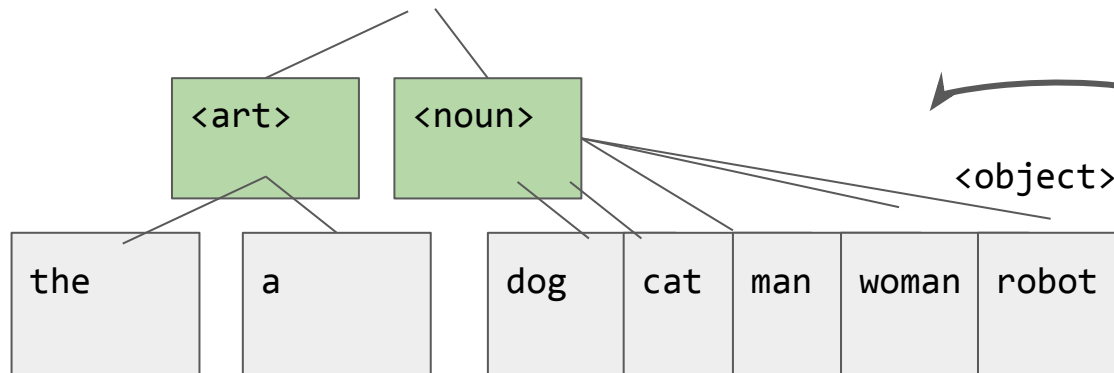
# Parse Tree

Top Down vs. Bottom Up

-

*“The robot stroked **the dog**”*

`<sentence> ::= <subject> <verb> <object>`



`<object> ::= <art> <noun> | two furry dice`

`<article> ::= the | a`

`<noun> ::= dog | cat | man | woman | robot`

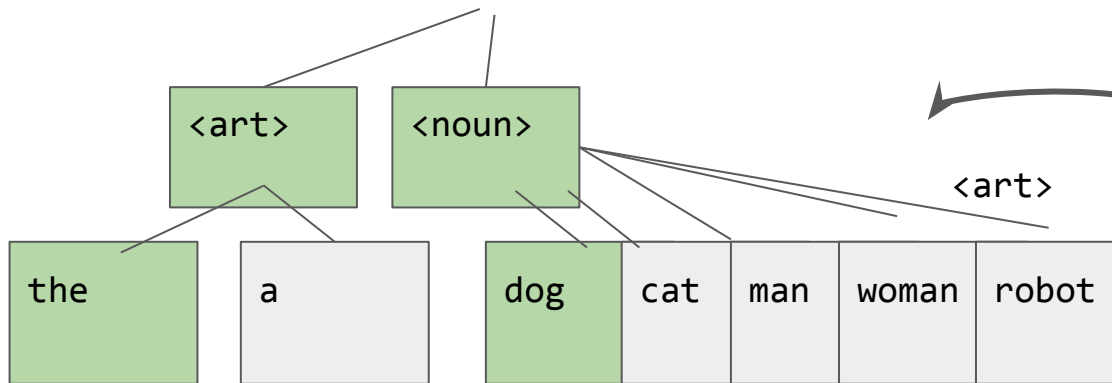
# Parse Tree

Top Down vs. Bottom Up

-

*“The robot stroked **the dog**”*

`<sentence> ::= <subject> <verb> <object>`



`<object> ::= <art> <noun> | two furry dice`

`<article> ::= the | a`

`<noun> ::= dog | cat | man | woman | robot`

`<noun>`

Stack

the robot stroked two furry dice

<subj> <verb> <obj>

<subj> <verb> <obj>

<art> <noun>

the robot stroked two furry dice