Languages and machines

The meaning of language and limits of computation

Samuel Grayson

April 11, 2019

Goals

1. Give you an overview of language classes and machine types
2. Don't just think about computable/non-computable; Think about computable *with what*
3. Subsets of Turing Complete; next week, supersets

Mathy Definitions

Concatenation

Mathy Definitions

Concatenation $a(bc) = abc = (ab)c$

Mathy Definitions

Concatenation $a(bc) = abc = (ab)c$, $\varepsilon a = a = a\varepsilon$

Mathy Definitions

Concatenation $a(bc) = abc = (ab)c$, $\varepsilon a = a = a\varepsilon$
Concatenation of sets $\Sigma\Gamma := \{\sigma\gamma : \sigma \in \Sigma, \gamma \in \Gamma\}$

Mathy Definitions

Concatenation $a(bc) = abc = (ab)c$, $\varepsilon a = a = a\varepsilon$
Concatenation of sets $\Sigma\Gamma := \{\sigma\gamma : \sigma \in \Sigma, \gamma \in \Gamma\}$
Kleene star $\Sigma^* := \{\varepsilon\} \cup \Sigma \cup \Sigma^2 \cup \cdots$
where $\Sigma^2 = \Sigma\Sigma$ (Star means arbitrary repetitions)

Mathy Definitions

Concatenation $a(bc) = abc = (ab)c$, $\varepsilon a = a = a\varepsilon$
Concatenation of sets $\Sigma\Gamma := \{\sigma\gamma : \sigma \in \Sigma, \gamma \in \Gamma\}$
Kleene star $\Sigma^* := \{\varepsilon\} \cup \Sigma \cup \Sigma^2 \cup \cdots$
where $\Sigma^2 = \Sigma\Sigma$ (Star means arbitrary repetitions)
Language over alphabet Σ := subset of Σ^*
The strings are considered 'valid words' of the language.

Definitions

Formal Grammar :=

- finite set of terminals (convention: lowercase)
- finite set of nonterminals (convention: uppercase)
- start symbol (convention: S)
- finite set of rules (pair of strings)

Grammars generate languages.
Abstract Machine := ?
Machines can decide if a string is in a language.

Grammar semantics

- G := (a-z, A-Z, S, the following rules)
  1. S $\rightarrow$ aSb
  2. S $\rightarrow$ ε

Grammar semantics

- G := (a-z, A-Z, S, the following rules)
  1. S $\rightarrow$ aSb
  2. S $\rightarrow$ ε
- S

Grammar semantics

- G := (a-z, A-Z, S, the following rules)
  1. S $\rightarrow$ aSb
  2. S $\rightarrow$ ε
- S $\rightarrow$ aSb

Grammar semantics

- G := (a-z, A-Z, S, the following rules)
  1. S $\rightarrow$ aSb
  2. S $\rightarrow$ ε
- S $\rightarrow$ aSb $\rightarrow$ aaSbb

Grammar semantics

- G := (a-z, A-Z, S, the following rules)
  1. S $\rightarrow$ aSb
  2. S $\rightarrow \varepsilon$
- S $\rightarrow$ aSb $\rightarrow$ aaSbb $\rightarrow$ aaaSbbb

Grammar semantics

- G := (a-z, A-Z, S, the following rules)
  1. S $\rightarrow$ aSb
  2. S $\rightarrow \varepsilon$
- S $\rightarrow$ aSb $\rightarrow$ aaSbb $\rightarrow$ aaaSbbb $\rightarrow$ aaabbb

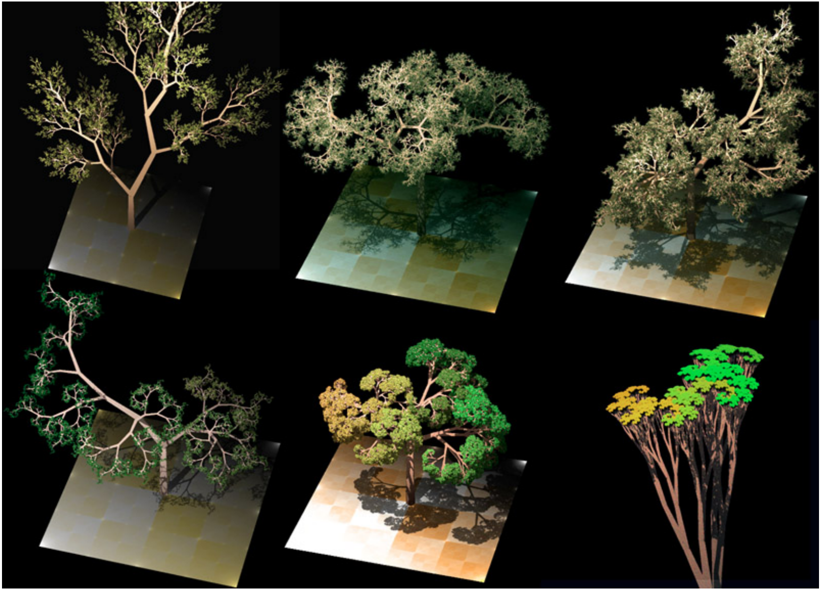
Grammar semantics

- G := (a-z, A-Z, S, the following rules)
  1. S $\rightarrow$ aSb
  2. S $\rightarrow \varepsilon$
- S $\rightarrow$ aSb $\rightarrow$ aaSbb $\rightarrow$ aaaSbbb $\rightarrow$ aaabbb
- The language generated by G is

Grammar semantics

- G := (a-z, A-Z, S, the following rules)
  1. S $\rightarrow$ aSb
  2. S $\rightarrow$ ε
- S $\rightarrow$ aSb $\rightarrow$ aaSbb $\rightarrow$ aaaSbbb $\rightarrow$ aaabbb
- The language generated by G is $\{a^n b^n | n \in \mathbb{N}\}$.

Lindenmayer system



Lindenmayer system

Grammar $\iff$ machine

Grammar generates string in the language
Machine recognizes a string that is in the language

Language relates to computing

- Consider language L consisting of factorials $\{$"1", "1", "2", "6", "24", $\ldots\}$

Language relates to computing

- Consider language L consisting of factorials $\{$"1", "1", "2", "6", "24", $\ldots\}$
- A generator/recognizer would have to compute the factorial.

Language hierarchy

simple languages, quick machines $\rightleftharpoons$ complex languages, slow machines

Language hierarchy

simple languages, quick machines $\rightleftharpoons$ complex languages, slow machines

Chomsky Hierarchy (1956):

Language class	Grammar	Type of machine
Regular	A $\rightarrow$ Yz	deterministic finite-state automata
Context-free	A $\rightarrow \gamma$	deterministic pushdown automata
Context-sensitive	$\alpha \mathrm{A} \beta \rightarrow \alpha\gamma\beta$	linear-bounded non-deterministic Turing machine
Recursively Enumerable	$\alpha \rightarrow \beta$	Turing machine

where α, β, γ: strings of terminals and non-terminals

Language hierarchy

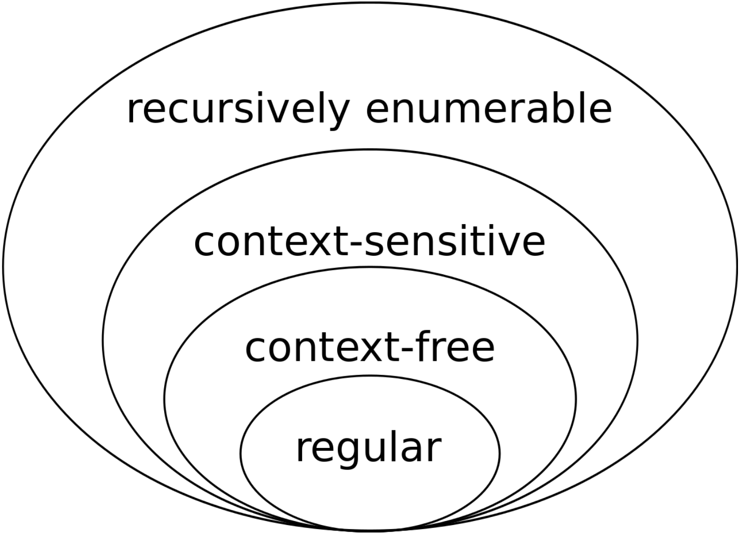


Figure from WikiMedia

Deterministic finite-state automata

Example language: Multiples of 3 in binary notation

Deterministic finite-state automata

Example language: Multiples of 3 in binary notation

Note that remainder can be computed if you slap on another digit.

remainder $x \pmod 3$	after appending 0 $x.0 \pmod 3$	after appending 1 $x.1 \pmod 3$

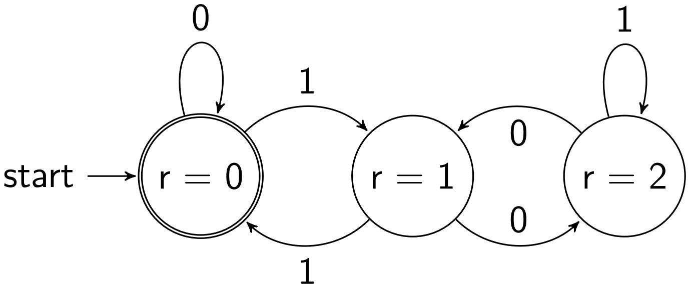
Deterministic finite-state automata

Example language: Multiples of 3 in binary notation

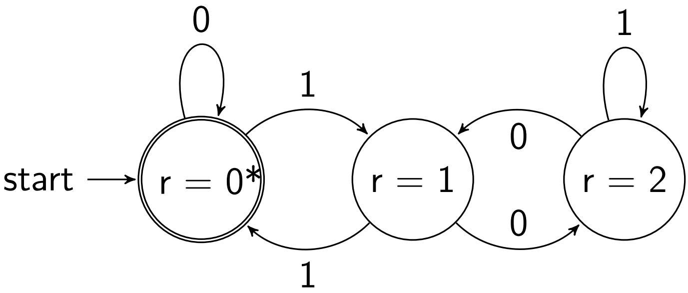
Note that remainder can be computed if you slap on another digit.

remainder $x \pmod 3$	after appending 0 $x.0 \pmod 3$	after appending 1 $x.1 \pmod 3$
0	0	1
1	2	0
2	1	2

Deterministic finite-state automata

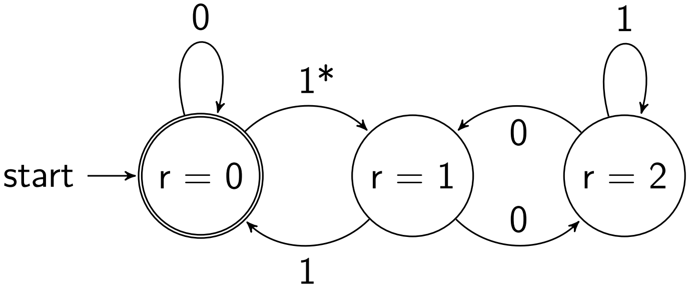


Deterministic finite-state automata



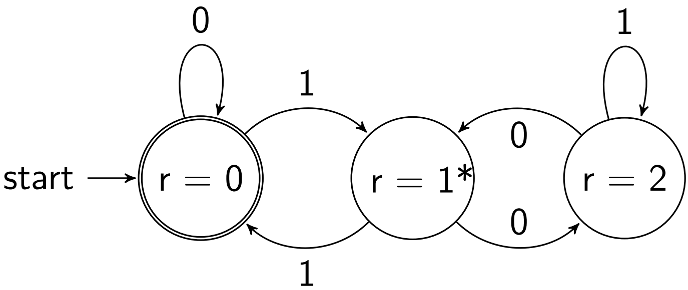
110

Deterministic finite-state automata



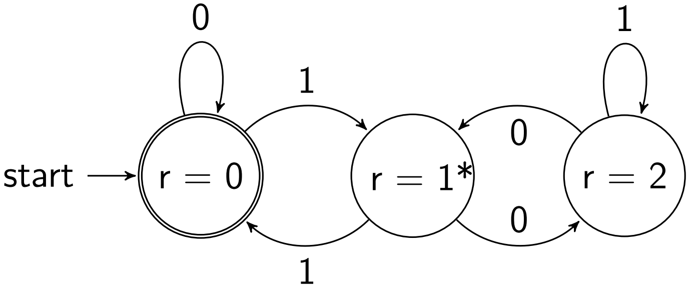
110

Deterministic finite-state automata



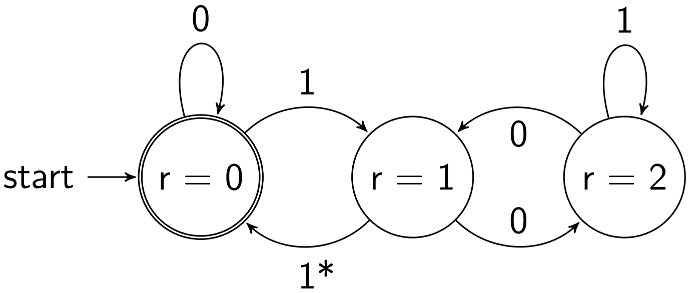
110

Deterministic finite-state automata



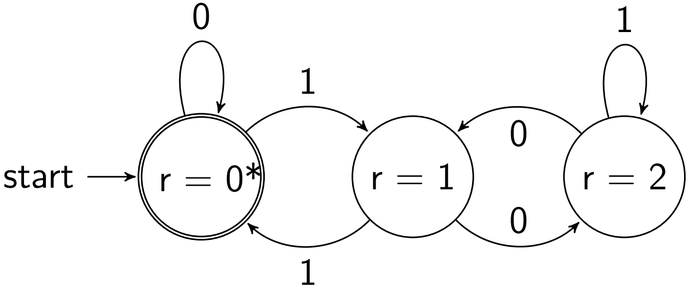
110

Deterministic finite-state automata



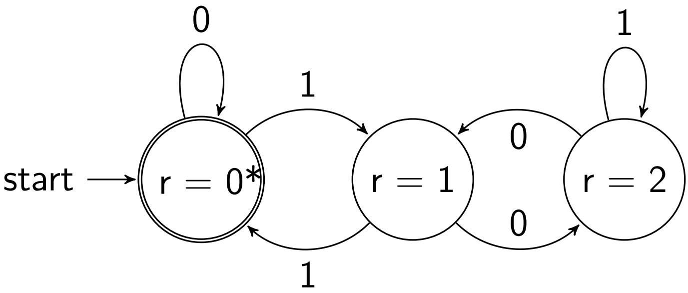
110

Deterministic finite-state automata



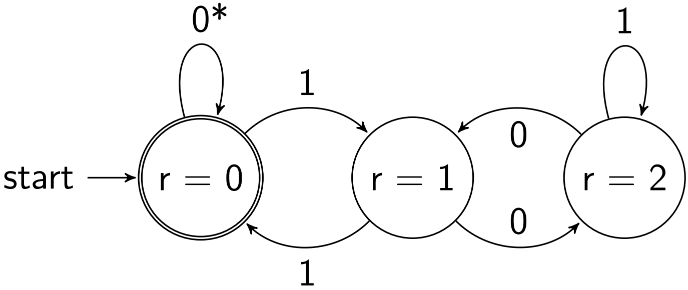
110

Deterministic finite-state automata



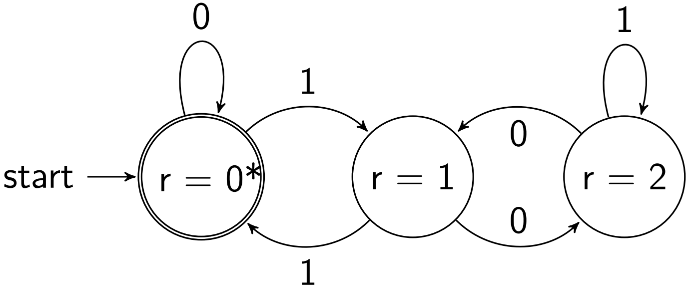
110

Deterministic finite-state automata



110

Deterministic finite-state automata



110

Deterministic Finite-state Automata

Definition

Deterministic finite state automata :=

- S: a finite set of states
- $A \subseteq S$: a set of accepting states
- $S_0 \in S$: an intial state
- Σ: a finite alphabet
- $f : S \times \Sigma \to S$: state-transition function

Nondeterministic Finite-state Automata

Definition

Nondeterministic finite state automata :=

- S: a finite set of states
- $A \subseteq S$: a set of accepting states
- $S_0 \in S$: an intial state
- Σ: a finite alphabet
- $f : S \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(S)$: state-transition function

NFA

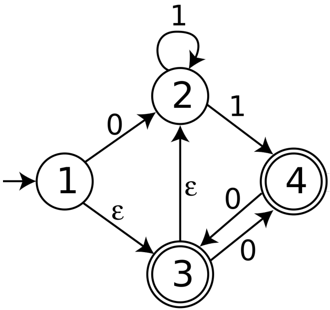
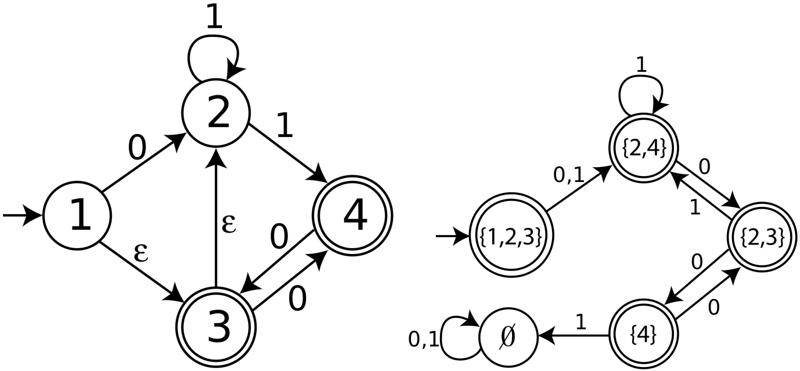
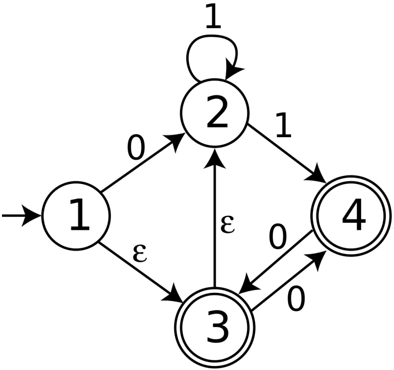


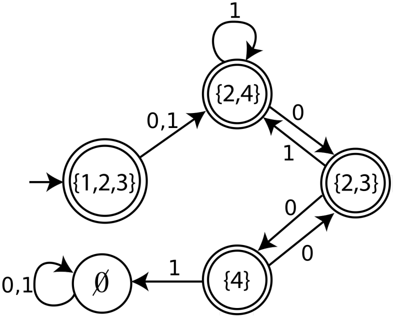
Figure from WikiMedia

NFA $\iff$ DFA



NFA $\iff$ DFA





possibly n states $\rightarrow$ 2^n states, worst-case.

Figure 1 from WikiMedia

Figure 2 from WikiMedia

Regular grammar

Definition

Regular grammar $:=$ Languages where every rule is of the form $A \rightarrow \varepsilon$, $A \rightarrow aB$ (right-regular).

Regular grammar

Definition

Regular grammar := Languages where every rule is of the form A $\rightarrow \varepsilon$, A $\rightarrow$ aB (right-regular).

Note: more than one rule could apply. A string is generated if any of these are valid.
We can construct $A \rightarrow a$ by

Regular grammar

Definition

Regular grammar := Languages where every rule is of the form A $\rightarrow \varepsilon$, A $\rightarrow$ aB (right-regular).

Note: more than one rule could apply. A string is generated if any of these are valid.

We can construct $A \rightarrow a$ by $A \rightarrow aB$ and $B \rightarrow \varepsilon$

Likewise $A \rightarrow a_1 a_2 \cdots a_n B$ by

Regular grammar

Definition

Regular grammar := Languages where every rule is of the form A $\to \varepsilon$, A $\to$ aB (right-regular).

Note: more than one rule could apply. A string is generated if any of these are valid.
We can construct $A \to a$ by $A \to aB$ and $B \to \varepsilon$
Likewise $A \to a_1 a_2 \cdots a_n B$ by $A \to a_1 A_1$, $A_1 \to a_2 A_2$, $\cdots$, $A_n \to a_n B$.

Regular Grammar example

Example language: Multiples of 3 in binary notation

remainder $x \pmod 3$	after appending 0 $x.0 \pmod 3$	after appending 1 $x.1 \pmod 3$
0	0	1
1	2	0
2	1	2

$A \rightarrow 0A$
$A \rightarrow 1B$
$B \rightarrow 0C$
$B \rightarrow 1A$
$C \rightarrow 0B$
$C \rightarrow 1C$

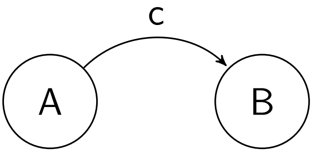
Regular Grammar example

Example language: Multiples of 3 in binary notation

remainder $x \pmod 3$	after appending 0 $x.0 \pmod 3$	after appending 1 $x.1 \pmod 3$
0	0	1
1	2	0
2	1	2

A → 0A
A → 1B
B → 0C
B → 1A
C → 0B
C → 1C
A → ε
S → A
TODO: redraw DFA

NFA $\iff$ Regular Grammar

Transition, , corresponds to rule, A $\rightarrow$ cB.

Start state = starting symbol, ending states $\rightarrow \varepsilon$

Useful for proofs.

Properties of Regular Languages

Languages with arbitrary repetition and optional elements.

Properties of Regular Languages

Languages with arbitrary repetition and optional elements.
Closed under complementation.

Properties of Regular Languages

Languages with arbitrary repetition and optional elements.
Closed under complementation. Proof: switch accepting and non-accepting states in DFA.

Properties of Regular Languages

Languages with arbitrary repetition and optional elements.
Closed under complementation. Proof: switch accepting and non-accepting states in DFA.
Closed under dis/conjunction.

Properties of Regular Languages

Languages with arbitrary repetition and optional elements.
Closed under complementation. Proof: switch accepting and non-accepting states in DFA.
Closed under dis/conjunction. Proof: consider cartesian product of two machines.

Properties of Regular Languages

Languages with arbitrary repetition and optional elements.
Closed under complementation. Proof: switch accepting and non-accepting states in DFA.
Closed under dis/conjunction. Proof: consider cartesian product of two machines.
Closed under Kleene-*.

Properties of Regular Languages

Languages with arbitrary repetition and optional elements.
Closed under complementation. Proof: switch accepting and non-accepting states in DFA.
Closed under dis/conjunction. Proof: consider cartesian product of two machines.
Closed under Kleene-*. Proof: accepting state to start state with ε transition

Properties of Regular Languages

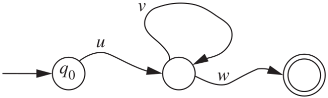
Languages with arbitrary repetition and optional elements.
Closed under complementation. Proof: switch accepting and non-accepting states in DFA.
Closed under dis/conjunction. Proof: consider cartesian product of two machines.
Closed under Kleene-*. Proof: accepting state to start state with ε transition
Algorithm to minimize (1979)
always halts; linear time

Pumping lemma (1969)

Lemma: For all DFA with n states, for all strings, $x \in L$ longer than n, ...

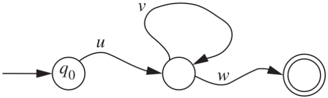
Pumping lemma (1969)

Lemma: For all DFA with n states, for all strings, $x \in L$ longer than n, ...
Proof: $|x| > n$, so visits a state twice (pigeon-hole).



Pumping lemma (1969)

Lemma: For all DFA with n states, for all strings, $x \in L$ longer than n, ...
Proof: $|x| > n$, so visits a state twice (pigeon-hole).



Now x can be decomposed into uvw where v can be repeated, so $uv^iw \in L$ also.
Note that v is non-empty and $|uv| \leq n$.

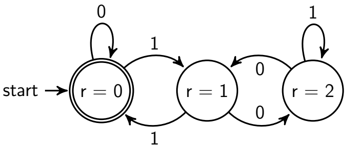
Figure from "Introduction to Languages and the Theory of Computation" by John C. Martin

Pumping lemma (1969)

Given a regular language, L,
There exists $n \in \mathbb{N}$ such that

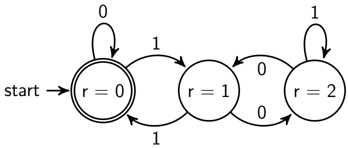
- For all $x \in L$ and $|x| > n$
  - There exists u, v, w where
    1. $x = uvw$
    2. $v \neq \varepsilon$
    3. $|uv| \leq n$
    4. $uv^i w \in L$ for all $i \in N$

Pumping lemma example



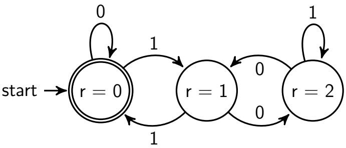
- Let's try divisble-by-3 DFA for $9 = 1001$.

Pumping lemma example



- Let's try divisible-by-3 DFA for $9 = 1001$.
- We can pump the double-zeros. $1(00)^n1$ is divisible by three.

Pumping lemma example



- Let's try divisble-by-3 DFA for $9 = 1001$.
- We can pump the double-zeros. $1(00)^n 1$ is divisible by three.
- Indeed $1 + 2^{1+2n} \pmod 3 = 1 + (-1) = 0$.

More difficult languages

- $L = \{a^n b^n : n \in \mathbb{N}\}$.

More difficult languages

- $L = \{a^n b^n : n \in \mathbb{N}\}$. No substring can be pumped up.

More difficult languages

- $L = \{a^n b^n : n \in \mathbb{N}\}$. No substring can be pumped up.
- Language of matched brackets (e.g. "(())()").

More difficult languages

- $L = \{a^n b^n : n \in \mathbb{N}\}$. No substring can be pumped up.
- Language of matched brackets (e.g. "(())()"). For every n, no less than$n$-long substring of $(^n)^n$ can be pumped up.

More difficult languages

- $L = \{a^n b^n : n \in \mathbb{N}\}$. No substring can be pumped up.
- Language of matched brackets (e.g. "(())()"). For every n, no less than$n$-long substring of $(^n)^n$ can be pumped up.
- Palindromes

More difficult languages

- $L = \{a^n b^n : n \in \mathbb{N}\}$. No substring can be pumped up.
- Language of matched brackets (e.g. "(())()").
  For every n, no less than$n$-long substring of $(^n)^n$ can be pumped up.
- Palindromes
  For every n, no less than n-long substring of $a^n b^n$ can be pumped up.

More difficult languages

- $L = \{a^n b^n : n \in \mathbb{N}\}$. No substring can be pumped up.
- Language of matched brackets (e.g. "(())()").
  For every n, no less than$n$-long substring of $(^n)^n$ can be pumped up.
- Palindromes
  For every n, no less than n-long substring of $a^n b^n$ can be pumped up.
- Language where length of strings is not "eventually linear"
- Challenge: L is not regular, but L^2 is.

Pushdown Automata

Definition

Non-/Deterministic pushdown automata $:=$

- S: a finite set of states
- $A \subseteq S$: a set of accepting states
- $S_0 \in S$: an initial state
- Σ: a finite alphabet for words
- Γ: a stack alphabet
- Γ_0: an initial stack symbol
- $f : S \times \Sigma \times \Gamma \to S \times \Gamma^*$: a transition function

Pushdown Automata

Definition

Non-/Deterministic pushdown automata :=

- S: a finite set of states
- $A \subseteq S$: a set of accepting states
- $S_0 \in S$: an initial state
- Σ: a finite alphabet for words
- Γ: a stack alphabet
- Γ_0: an initial stack symbol
- $f : S \times \Sigma \times \Gamma \to S \times \Gamma^*$: a transition function
- $f : S \times \Sigma \times (\Gamma \cup \{\varepsilon\}) \to S \times \Gamma^*$: a transition function

Pushdown Automata

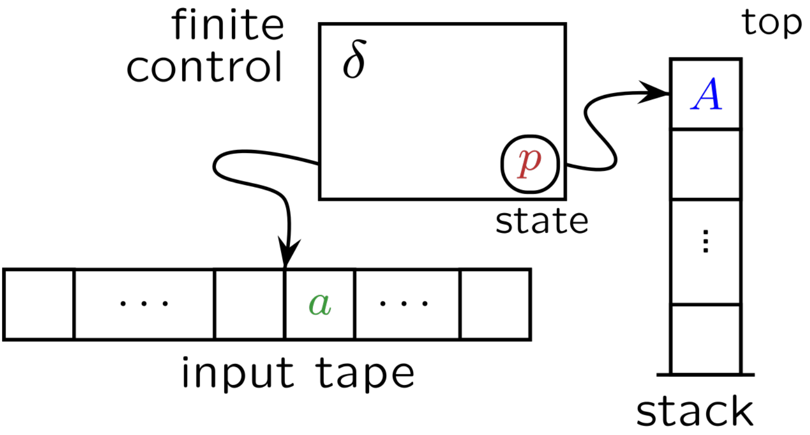
Definition

Non-/Deterministic pushdown automata :=

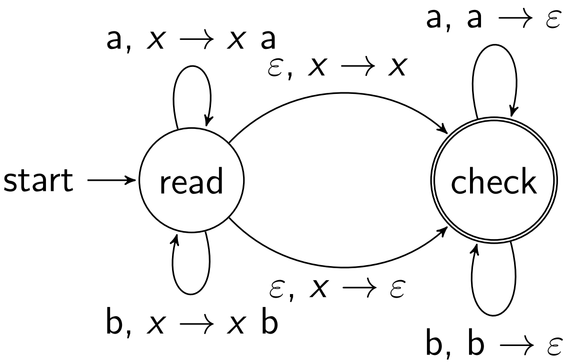
- S: a finite set of states
- $A \subseteq S$: a set of accepting states
- $S_0 \in S$: an initial state
- Σ: a finite alphabet for words
- Γ: a stack alphabet
- Γ_0: an initial stack symbol
- $f : S \times \Sigma \times \Gamma \to S \times \Gamma^*$: a transition function
- $f : S \times \Sigma \times (\Gamma \cup \{\varepsilon\}) \to S \times \Gamma^*$: a transition function

Accept by empty stack

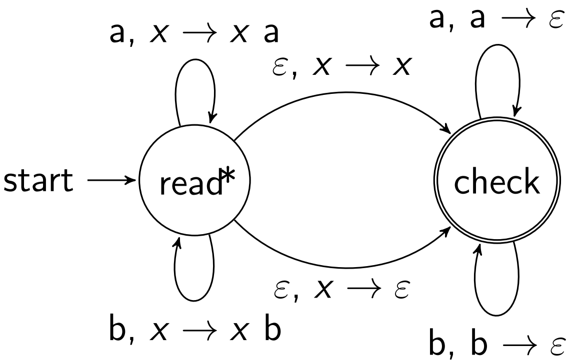
Pushdown Automata



Pushdown Automata Example

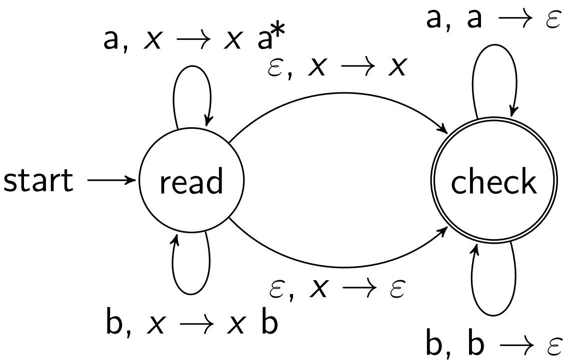


Pushdown Automata Example



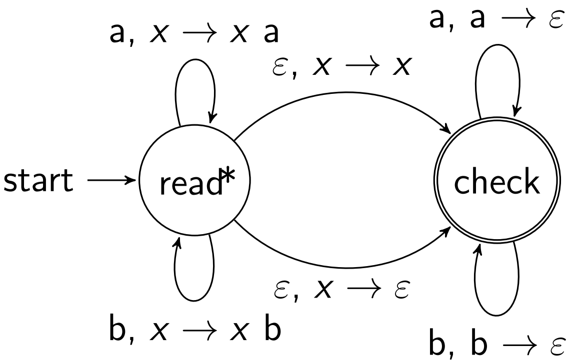
Input: abba
Stack: H

Pushdown Automata Example



Input: abba
Stack: Ha

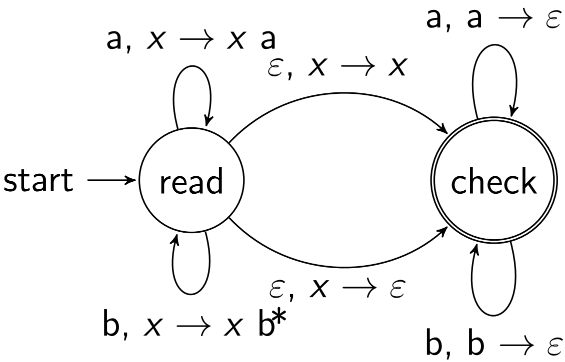
Pushdown Automata Example



Input: abba

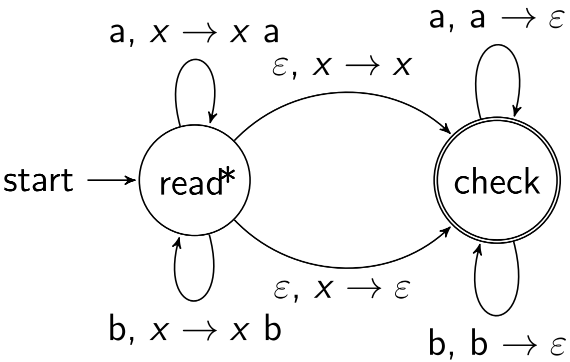
Stack: Ha

Pushdown Automata Example



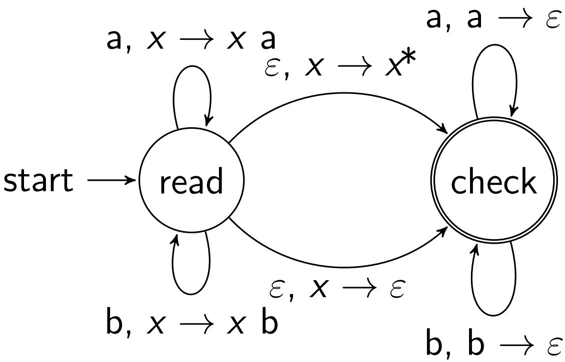
Input: abba
Stack: Hab

Pushdown Automata Example



Input: abba
Stack: Hab

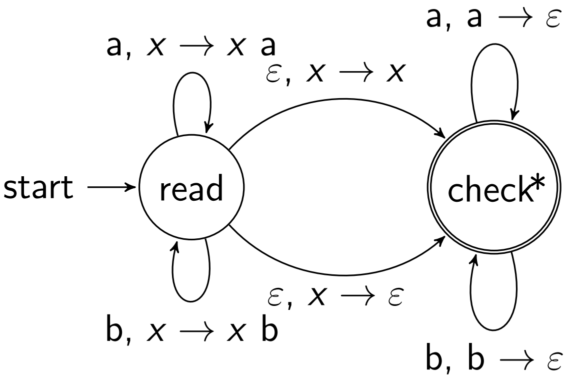
Pushdown Automata Example



Input: abba

Stack: Hab

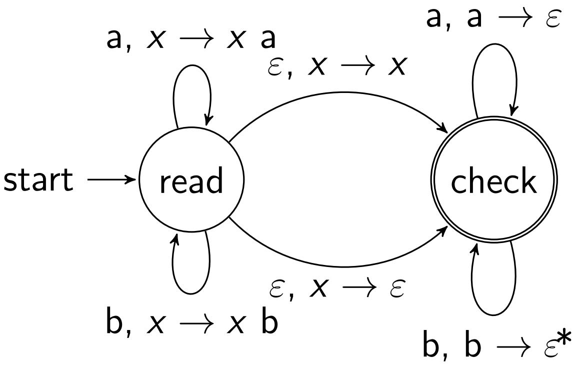
Pushdown Automata Example



Input: abba

Stack: Hab

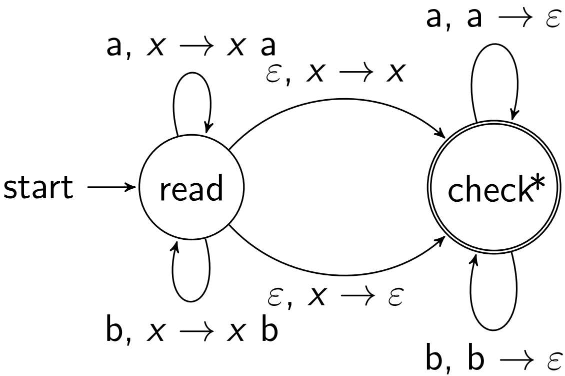
Pushdown Automata Example



Input: abba

Stack: Hab

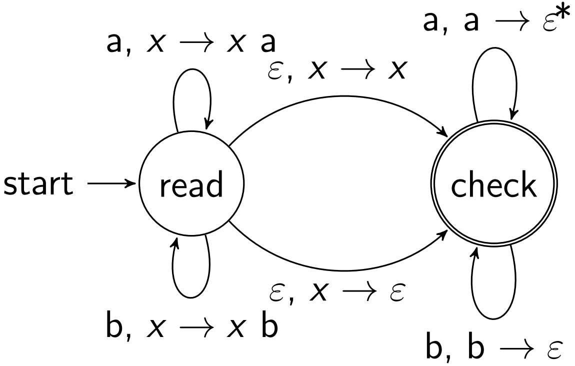
Pushdown Automata Example



Input: abba

Stack: Ha

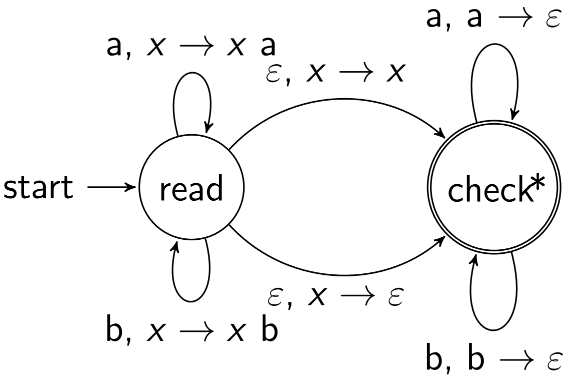
Pushdown Automata Example



Input: abba

Stack: Ha

Pushdown Automata Example



Input: abba
Stack: H

Examples

Matched named brackets

Examples

Matched named brackets

State: If left bracket, push bracket-type to stack. If right bracket and correct bracket-type on stack, pop from stack.

CFG

Definition

Context-free grammar := Grammar where all rules are of the form $A \to \alpha$, where $\alpha \in (\Sigma \cup \Gamma)^*$.

Encodes everything a regular grammar can (repetition, optionals) + "arbitrary nestedness"

- Matched-brackets

CFG

Definition

Context-free grammar $:=$ Grammar where all rules are of the form $A \to \alpha$, where $\alpha \in (\Sigma \cup \Gamma)^*$.

Encodes everything a regular grammar can (repetition, optionals) $+$ "arbitrary nestedness"

- Matched-brackets
  1. S $\to$ [S]
  2. S $\to$ S S
  3. S $\to \varepsilon$
- $\{a^n b^n : n \in \mathbb{N}\}$

CFG

Definition

Context-free grammar := Grammar where all rules are of the form $A \to \alpha$, where $\alpha \in (\Sigma \cup \Gamma)^*$.

Encodes everything a regular grammar can (repetition, optionals) + "arbitrary nestedness"

- Matched-brackets
  1. S $\to$ [S]
  2. S $\to$ S S
  3. S $\to \varepsilon$
- $\{a^n b^n : n \in \mathbb{N}\}$
  1. S $\to$ aSb
  2. S $\to \varepsilon$
- Palindromes similar

CFGs in the wild

Named brackets

CFGs in the wild

Named brackets

```
<html>
  <body>
    <h1>This is the tile</h1>
    <p>This is a paragraph.</p>
  </body>
</html>
```

CFGs in the wild

Named brackets

```
<html>
  <body>
    <h1>This is the tile</h1>
    <p>This is a paragraph.</p>
  </body>
</html>
```

Programming languages

CFGs in the wild

Named brackets

```
<html>
  <body>
    <h1>This is the tile</h1>
    <p>This is a paragraph.</p>
  </body>
</html>
```

Programming languages
95% of natural languages

Parse trees

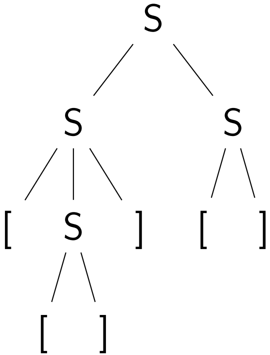
1. S $\rightarrow$ [S]
2. S $\rightarrow$ S S
3. S $\rightarrow$ ε

[[]][]

Parse trees

1. S → [S]
2. S → S S
3. S → ε

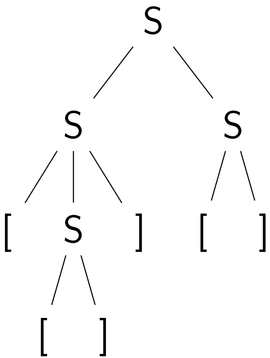
[[]][]



Parse trees

1. S → [S]
2. S → S S
3. S → ε

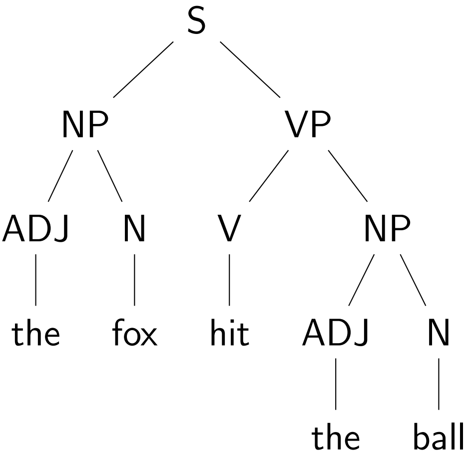
[[]][]



Parse trees for regular languages?

Ex. Natural language

- S $\rightarrow$ NP VP
- NP $\rightarrow$ ADJ N
- VP $\rightarrow$ V
- VP $\rightarrow$ V NP



Ambiguity

Example

$$\text{EXPR} \rightarrow \text{EXPR} + \text{EXPR}$$
$$\text{EXPR} \rightarrow \text{EXPR} * \text{EXPR}$$
$$\text{EXPR} \rightarrow \text{NUMBER}$$

Ambiguity

Example

$$\text{EXPR} \rightarrow \text{EXPR} + \text{EXPR}$$
$$\text{EXPR} \rightarrow \text{EXPR} * \text{EXPR}$$
$$\text{EXPR} \rightarrow \text{NUMBER}$$

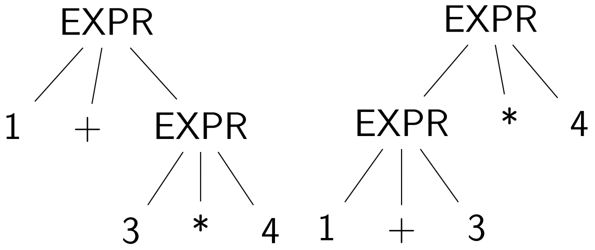
$$1 + 3 * 4$$

Ambiguity

Example

EXPR $\rightarrow$ EXPR + EXPR
EXPR $\rightarrow$ EXPR * EXPR
EXPR $\rightarrow$ NUMBER

1 + 3 * 4



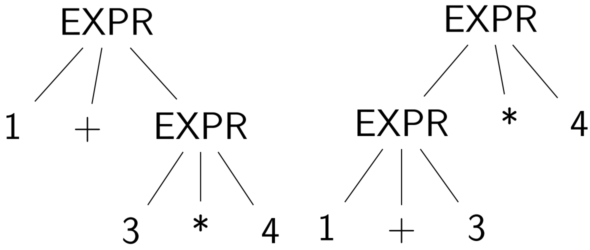
Resolution: order rules,

Ambiguity

Example

EXPR → EXPR + EXPR
EXPR → EXPR * EXPR
EXPR → NUMBER

1 + 3 * 4



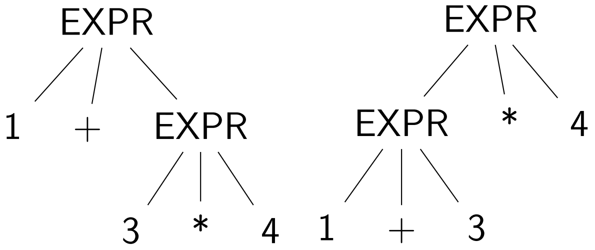
Resolution: order rules, order tokens,

Ambiguity

Example

EXPR → EXPR + EXPR
EXPR → EXPR * EXPR
EXPR → NUMBER

1 + 3 * 4



Resolution: order rules, order tokens, **don't**.

Ambiguity

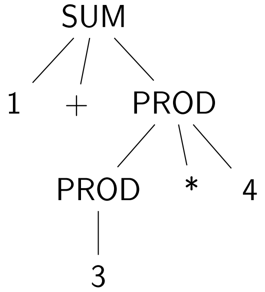
Example

SUM $\rightarrow$ SUM $+$ PROD
SUM $\rightarrow$ PROD
PROD $\rightarrow$ PROD * NUMBER
PROD $\rightarrow$ NUMBER

Ambiguity

Example

SUM → SUM + PROD
SUM → PROD
PROD → PROD * NUMBER
PROD → NUMBER



Ambiguity

Is a CFG ambiguous?

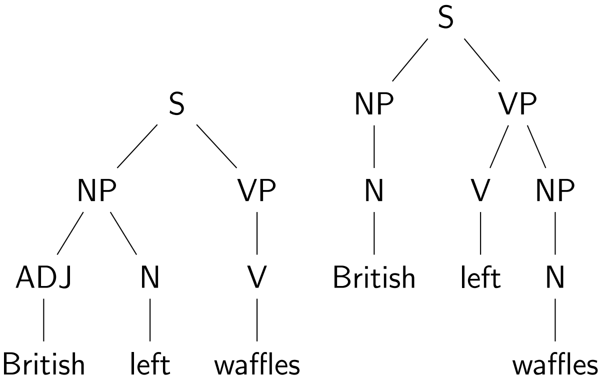
Ambiguity

Is a CFG ambiguous? Undecidable

It happens in the wild all the time!

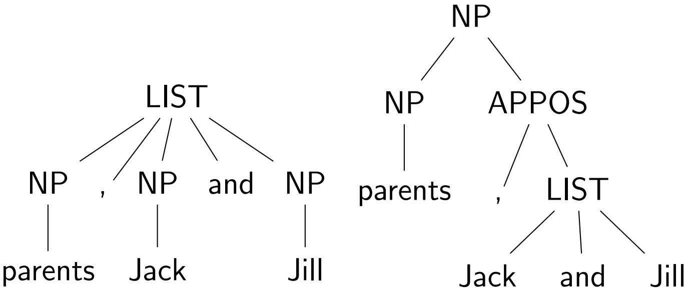
Ambiguity in natural language

British left waffles on Falklands



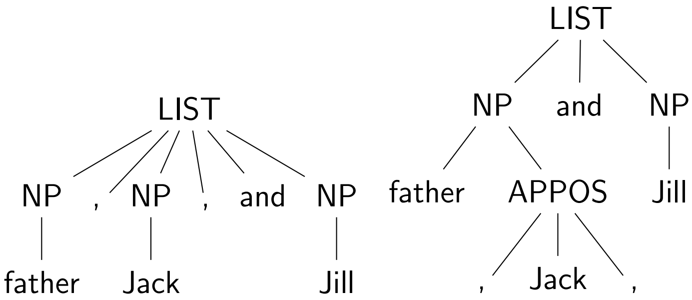
Ambiguity in natural language

"My parents, Jack and Jill" refers to 2 or 4 people.



Ambiguity in natural language

“My father, Jack, and Jill” refers to 2 or 3 people.



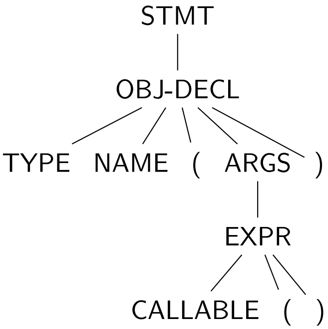
Lojban

Ambiguity in constructed languages

```
class1 obj (class2 ());
```

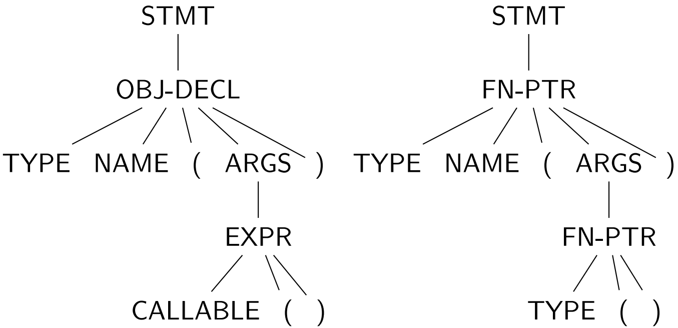
Ambiguity in constructed languages

```
class1 obj (class2 ());
```



Ambiguity in constructed languages

```
class1 obj (class2 ());
```



Public Service Announcement

```
// class1 constructed from class2
class1 obj ((class2 ()));
class1 obj {class2 {}}; // C++11

// Function pointer
class1 obj (class2 name ());
```

Bottom-up parsing

$S \rightarrow (\; S \;)$
$S \rightarrow \varepsilon$
Stack:
Input: (())()

Bottom-up parsing

$S \to (\ S\ )$
$S \to \varepsilon$
Stack: (
Input: ())()

Bottom-up parsing

$S \to (\; S \;)$
$S \to \varepsilon$
Stack: ((
Input:))()

Bottom-up parsing

$S \rightarrow (\; S \;)$

$S \rightarrow \varepsilon$

Stack: ((S

Input:))()

Bottom-up parsing

$S \to (\ S \ )$
$S \to \varepsilon$
Stack: ((S)
Input:)()

Bottom-up parsing

$S \to (\; S \;)$
$S \to \varepsilon$
Stack: (S
Input:)()

Bottom-up parsing

$S \to (\ S \ )$
$S \to \varepsilon$
Stack: (S)
Input: ()

Bottom-up parsing

$S \rightarrow (\ S \ )$
$S \rightarrow \varepsilon$
Stack: S
Input: ()

Bottom-up parsing

$S \rightarrow (\; S \;)$
$S \rightarrow \varepsilon$
Stack: S(
Input:)

Bottom-up parsing

$S \rightarrow (\; S \;)$
$S \rightarrow \varepsilon$
Stack: S(S
Input:)

Bottom-up parsing

$S \rightarrow (\ S \ )$
$S \rightarrow \varepsilon$
Stack: S(S)
Input:

Bottom-up parsing

$S \to (\ S \ )$
$S \to \varepsilon$
Stack: SS
Input:

Bottom-up parsing

$S \rightarrow (\; S \;)$
$S \rightarrow \varepsilon$
Stack: S
Input:

Properties of CFL

- Closed under disjunction (prove using PDAs)

Properties of CFL

- Closed under disjunction (prove using PDAs) ε transitions to start state of machine 1 OR 2.

Properties of CFL

- Closed under disjunction (prove using PDAs) ε transitions to start state of machine 1 OR 2.
- Closed under Kleene-* (prove using empty-stack PDAs)

Properties of CFL

- Closed under disjunction (prove using PDAs) ε transitions to start state of machine 1 OR 2.
- Closed under Kleene-* (prove using empty-stack PDAs) ε transition from accepting state to start state.

Properties of CFL

- Closed under disjunction (prove using PDAs) ε transitions to start state of machine 1 OR 2.
- Closed under Kleene-* (prove using empty-stack PDAs) ε transition from accepting state to start state.
- Closed under conjunction

Properties of CFL

- Closed under disjunction (prove using PDAs) ε transitions to start state of machine 1 OR 2.
- Closed under Kleene-* (prove using empty-stack PDAs) ε transition from accepting state to start state.
- Closed under conjunction NOT

Properties of CFL

- Closed under disjunction (prove using PDAs) ε transitions to start state of machine 1 OR 2.
- Closed under Kleene-* (prove using empty-stack PDAs) ε transition from accepting state to start state.
- Closed under conjunction NOT
- Chomsky Normal Form ($A \to BC$, $A \to a$, exception for $\varepsilon \in L$) (algorithm 1959)

Properties of CFL

- Closed under disjunction (prove using PDAs) ε transitions to start state of machine 1 OR 2.
- Closed under Kleene-* (prove using empty-stack PDAs) ε transition from accepting state to start state.
- Closed under conjunction NOT
- Chomsky Normal Form ($A \to BC$, $A \to a$, exception for $\varepsilon \in L$) (algorithm 1959)
- Chomsky–Schützenberger Theorem (1963): CFG $=$ named-bracket $\cap$ regular

Pumping lemma for CFG

For all $s \in L$ and $|s| \geq n$,

- $s = uvwxy$
- $uv^iwx^iy \in L$
- $|vx| \geq 1$
- $|vwx| \leq p$.

Pumping lemma for CFG

For all $s \in L$ and $|s| \geq n$,

- $s = uvwxy$
- $uv^iwx^iy \in L$
- $|vx| \geq 1$
- $|vwx| \leq p$.

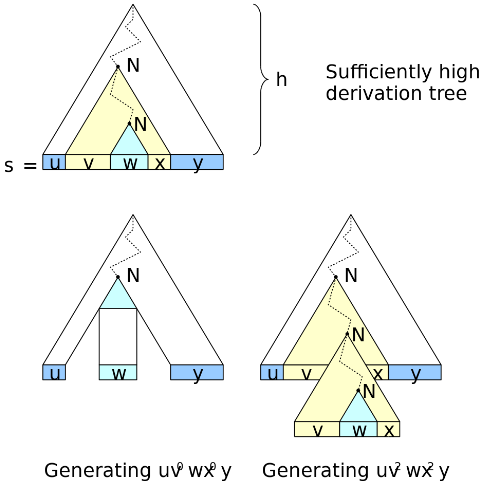


Figure from WikiMedia

Harder languages

- $\{a^n b^n c^n : n \in \mathbb{N}\}$. No 'pumpable' substrings within n.

Harder languages

- $\{a^n b^n c^n : n \in \mathbb{N}\}$. No 'pumpable' substrings within n.
  - Note that $\{a^n b^n c^n\} = \{a^n b^n c^k\} \cup \{a^n b^k c^k\} \implies$ CFLs **are not closed** under conjunction.
  - But they **are closed** under disjunction, so they **are not closed** under complementation.

Harder languages

- $\{a^n b^n c^n : n \in \mathbb{N}\}$. No 'pumpable' substrings within n.
  - Note that $\{a^n b^n c^n\} = \{a^n b^n c^k\} \cup \{a^n b^k c^k\} \implies$ CFLs **are not closed** under conjunction.
  - But they **are closed** under disjunction, so they **are not closed** under complementation.
- $\{w^2 : w \in \Sigma^*\}$ ("stutters")

Harder languages

- $\{a^n b^n c^n : n \in \mathbb{N}\}$. No 'pumpable' substrings within n.
  - Note that $\{a^n b^n c^n\} = \{a^n b^n c^k\} \cup \{a^n b^k c^k\} \implies$ CFLs **are not closed** under conjunction.
  - But they **are closed** under disjunction, so they **are not closed** under complementation.
- $\{w^2 : w \in \Sigma^*\}$ ("stutters")
- Challenge: prove $x : n_a(x) = kn_b(x)$ is not a CFL where $n_a(x)$ is the number of "a" in x.

Harder languages

- $\{a^n b^n c^n : n \in \mathbb{N}\}$. No 'pumpable' substrings within n.
  - Note that $\{a^n b^n c^n\} = \{a^n b^n c^k\} \cup \{a^n b^k c^k\} \implies$ CFLs **are not closed** under conjunction.
  - But they **are closed** under disjunction, so they **are not closed** under complementation.
- $\{w^2 : w \in \Sigma^*\}$ ("stutters")
- Challenge: prove $x : n_a(x) = kn_b(x)$ is not a CFL where $n_a(x)$ is the number of "a" in x.
- Swiss-German cross-serial dependencies
  $\mathrm{adj}_1\ \mathrm{adj}_2 \cdots \mathrm{adj}_n\ \mathrm{noun}_1\ \mathrm{noun}_2 \cdots \mathrm{noun}_n$

Harder languages

- $\{a^n b^n c^n : n \in \mathbb{N}\}$. No 'pumpable' substrings within n.
  - Note that $\{a^n b^n c^n\} = \{a^n b^n c^k\} \cup \{a^n b^k c^k\} \implies$ CFLs **are not closed** under conjunction.
  - But they **are closed** under disjunction, so they **are not closed** under complementation.
- $\{w^2 : w \in \Sigma^*\}$ ("stutters")
- Challenge: prove $x : n_a(x) = kn_b(x)$ is not a CFL where $n_a(x)$ is the number of "a" in x.
- Swiss-German cross-serial dependencies $\mathrm{adj}_1\ \mathrm{adj}_2 \cdots \mathrm{adj}_n\ \mathrm{noun}_1\ \mathrm{noun}_2 \cdots \mathrm{noun}_n$
- Programming language with no undefined terms

Turing Machine

Definition

Non-/Deterministic Turing Machine $:=$

- S: set of states
- $S_0 \in S$: initial state
- $S_r, S_a \in S$: halt reject, halt accept state
- Γ: tape alphabet
- $b \in \Gamma$: blank symbol
- $\Sigma \subseteq \Gamma \setminus \{b\}$: input alphabet
- $f : S \times \Sigma \to S \times \Sigma \times \{R, L\}$: deterministic transition function
- $f : S \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(S \times \Sigma \times \{R, L\})$: non-deterministic transition function

Turing Machine

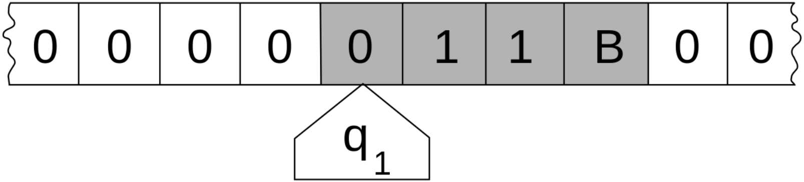


Figure from WikiMedia

Linear Bounded Turing Machine (LBTM)

Count possible 'complete-states' $|Q||n||\Sigma|^{k|n|}$.

Linear Bounded Turing Machine (LBTM)

Count possible 'complete-states' $|Q||n||\Sigma|^{k|n|}$.
Finite number of complete-states, pigeon hole. If we hit the same complete-state twice, inf. loop.

Linear Bounded Turing Machine (LBTM)

Count possible 'complete-states' $|Q||n||\Sigma|^{k|n|}$.
Finite number of complete-states, pigeon hole. If we hit the same complete-state twice, inf. loop.
Therefore inf. loops are detectable. Halting problem for LBTM is decidable.

Linear Bounded Turing Machine (LBTM)

Count possible 'complete-states' $|Q||n||\Sigma|^{k|n|}$.
Finite number of complete-states, pigeon hole. If we hit the same complete-state twice, inf. loop.
Therefore inf. loops are detectable. Halting problem for LBTM is decidable.
Nobody knows if LBNTM $\iff$ LB(D)TM

Context-Sensitive Grammar

CFG: $A \to \alpha$

Context-Sensitive Grammar

CFG: $A \to \alpha$
CSG: $\beta A \gamma \to \beta \alpha \gamma$ where $\alpha \neq \varepsilon$

Context-Sensitive Grammar

CFG: $A \rightarrow \alpha$
CSG: $\beta A \gamma \rightarrow \beta \alpha \gamma$ where $\alpha \neq \varepsilon$
$\beta \_ \gamma$ is the "context"
Repetition, optionals, nestedness

Context-Sensitive Grammar

CFG: $A \rightarrow \alpha$
CSG: $\beta A \gamma \rightarrow \beta \alpha \gamma$ where $\alpha \neq \varepsilon$
$\beta \_ \gamma$ is the "context"
Repetition, optionals, nestedness, with some memory

Context-Sensitive Grammar

CFG: $A \rightarrow \alpha$
CSG: $\beta A \gamma \rightarrow \beta \alpha \gamma$ where $\alpha \neq \varepsilon$
$\beta \_ \gamma$ is the "context"
Repetition, optionals, nestedness, with some memory
Note: Non-decreasing

LBNTM $\iff$ CSL

Suppose $\alpha \to \beta$ in a CSL with terminals Σ and non-terminals Γ.

LBNTM $\iff$ CSL

Suppose $\alpha \to \beta$ in a CSL with terminals Σ and non-terminals Γ.
Let the tape alphabet be, $\Gamma' = \Sigma \times (\Gamma \cup \{\Delta\})$.

LBNTM $\iff$ CSL

Suppose $\alpha \to \beta$ in a CSL with terminals Σ and non-terminals Γ.
Let the tape alphabet be, $\Gamma' = \Sigma \times (\Gamma \cup \{\Delta\})$.
Initially, tape has $x_0 x_1 \cdots x_n$. Convert to $(x_0, S), (x_1, \Delta) \cdots (x_n, \Delta)$.

LBNTM $\iff$ CSL

Suppose $\alpha \to \beta$ in a CSL with terminals Σ and non-terminals Γ.
Let the tape alphabet be, $\Gamma' = \Sigma \times (\Gamma \cup \{\Delta\})$.
Initially, tape has $x_0 x_1 \cdots x_n$. Convert to $(x_0, S), (x_1, \Delta) \cdots (x_n, \Delta)$.
Simulate a CSG parse $S \to \alpha \to \beta \to \cdots \to x_0 x_1 \cdots x_n$ in the *second* coordinate in each cell.
Accept if the first coordinate matches the second.

LBNTM $\iff$ CSL

Suppose $\alpha \to \beta$ in a CSL with terminals Σ and non-terminals Γ.
Let the tape alphabet be, $\Gamma' = \Sigma \times (\Gamma \cup \{\Delta\})$.
Initially, tape has $x_0 x_1 \cdots x_n$. Convert to $(x_0, S), (x_1, \Delta) \cdots (x_n, \Delta)$.
Simulate a CSG parse $S \to \alpha \to \beta \to \cdots \to x_0 x_1 \cdots x_n$ in the *second* coordinate in each cell.
Accept if the first coordinate matches the second.
CSL's are non-decreasing, so linear bounded.

LBNTM $\iff$ CSG

Construct CSG for given LBNTM:
Let configuration $:= x_0 x_1 \cdots x_{n-1} S_i x_n \cdots x_k$

LBNTM $\Longleftrightarrow$ CSG

Construct CSG for given LBNTM:
Let configuration $:= x_0x_1 \cdots x_{n-1}S_ix_n \cdots x_k$
Suppose $(S_i', x_n', L) \in f(S_i, x_n)$. Then
$S_i'x_{n-1}x_n' \to x_{n-1}S_ix_n$.

LBNTM $\iff$ CSG

Construct CSG for given LBNTM:
Let configuration $:= x_0 x_1 \cdots x_{n-1} S_i x_n \cdots x_k$
Suppose $(S_i', x_n', L) \in f(S_i, x_n)$. Then
$S_i' x_{n-1} x_n' \to x_{n-1} S_i x_n$.
This rule maps a configuration to the one configuration immediately prior. Example:
$x_0 x_1 \cdots S_i' x_{n-1} x_n' \cdots x_k$

LBNTM $\iff$ CSG

Construct CSG for given LBNTM:

Let configuration $:= x_0 x_1 \cdots x_{n-1} S_i x_n \cdots x_k$

Suppose $(S_i', x_n', L) \in f(S_i, x_n)$. Then

$S_i' x_{n-1} x_n' \to x_{n-1} S_i x_n$.

This rule maps a configuration to the one configuration immediately prior. Example:

$x_0 x_1 \cdots S_i' x_{n-1} x_n' \cdots x_k$

$x_0 x_1 \cdots x_{n-1} S_i x_n \cdots x_k$

LBNTM $\iff$ CSG

Construct CSG for given LBNTM:
Let configuration $:= x_0x_1 \cdots x_{n-1}S_ix_n \cdots x_k$
Suppose $(S_i', x_n', L) \in f(S_i, x_n)$. Then
$S_i'x_{n-1}x_n' \to x_{n-1}S_ix_n$.
This rule maps a configuration to the one configuration immediately prior. Example:
$x_0x_1 \cdots S_i'x_{n-1}x_n' \cdots x_k$
$x_0x_1 \cdots x_{n-1}S_ix_n \cdots x_k$
We can work 'backwards' to starting-tapes that the machine would accept!
We will start with a rule that generates arbitrary finishing-tapes: $b \to x_ib$ for $x_i \in \Gamma$, in an accepting state $S \to S_ab$.

CSL equivalents

- Primitive recursive (composition, recursion counting down from n)

CSL equivalents

- Primitive recursive (composition, recursion counting down from n)
- Programming with only bounded for-loops and no recursion (see BlooP)

CSL equivalents

- Primitive recursive (composition, recursion counting down from n)
- Programming with only bounded for-loops and no recursion (see BlooP)
- Kuroda Normal Form
  $(AB \to CD, A \to BC, A \to B, A \to a)$
  (exception for $\varepsilon \in L$)

Turing Machines equivalents

- Modern computers (within memory limitations)

Turing Machines equivalents

- Modern computers (within memory limitations)
- Arbitrary grammar

Turing Machines equivalents

- Modern computers (within memory limitations)
- Arbitrary grammar
- Recursively enumerable (composition, recursion, ‘count-up’)

Turing Machines equivalents

- Modern computers (within memory limitations)
- Arbitrary grammar
- Recursively enumerable (composition, recursion, ‘count-up’)
- FRACTRAN

Turing Machines equivalents

- Modern computers (within memory limitations)
- Arbitrary grammar
- Recursively enumerable (composition, recursion, 'count-up')
- FRACTRAN
- Lambda Calculus

Turing Machines equivalents

- Modern computers (within memory limitations)
- Arbitrary grammar
- Recursively enumerable (composition, recursion, ‘count-up’)
- FRACTRAN
- Lambda Calculus
- FlooP

None of these are guaranteed to halt!

P = NP

Deterministic TM **P**olynomial-time algorithm $\iff$ **N**on-deterministic TM **P**olynomial-time algorithm?

P $\neq$ NP

Deterministic TM **P**olynomial-time algorithm $\not\Leftrightarrow$ **N**on-deterministic TM **P**olynomial-time algorithm?

Universal Turing Program

Turing Machine takes encoding of a machine and its input, separated by a marker symbol.
Decides if machine accepts given input.

Unrestricted Grammars

$\alpha \to \beta$
α and β can be anything in $(\Gamma \cup \Sigma)^*$!

Unrestricted Grammars $\iff$ NTM

Task: construct NTM that recognizes strings genreated by an unrestricted grammar

$x_0 x_1 \cdots x_n$ on tape.

Unrestricted Grammars $\Longleftrightarrow$ NTM

Task: construct NTM that recognizes strings genreated by an unrestricted grammar

$x_0 x_1 \cdots x_n$ on tape.

Append $\#S$.

Unrestricted Grammars $\Longleftrightarrow$ NTM

Task: construct NTM that recognizes strings genreated by an unrestricted grammar

$x_0 x_1 \cdots x_n$ on tape.

Append $\#S$.

Start simulating a derivation, replacing α with β according to grammar. Result is unbounded, but so is tape.

Unrestricted Grammars $\Longleftrightarrow$ NTM

Task: construct NTM that recognizes strings genreated by an unrestricted grammar

$x_0 x_1 \cdots x_n$ on tape.

Append $\#S$.

Start simulating a derivation, replacing α with β according to grammar. Result is unbounded, but so is tape.

If string before $\#$ matches that after, accept.

Unrestricted Grammars $\iff$ NTM

Task: construct NTM that recognizes strings genreated by an unrestricted grammar
$x_0 x_1 \cdots x_n$ on tape.
Append $\#S$.
Start simulating a derivation, replacing α with β according to grammar. Result is unbounded, but so is tape.
If string before $\#$ matches that after, accept.
Could take forever

Unrestricted Grammars $\Longleftrightarrow$ NTM

$x_0 x_1 \cdots x_n$

Unrestricted Grammars $\iff$ NTM

$x_0 x_1 \cdots x_n$

$x_0 x_1 \cdots x_n \# S$

Unrestricted Grammars $\iff$ NTM

$x_0 x_1 \cdots x_n$

$x_0 x_1 \cdots x_n \# S$

$x_0 x_1 \cdots x_n \# \alpha\gamma$

Unrestricted Grammars $\iff$ NTM

$x_0 x_1 \cdots x_n$
$x_0 x_1 \cdots x_n \# S$
$x_0 x_1 \cdots x_n \# \alpha\gamma$
$x_0 x_1 \cdots x_n \# \beta\theta\gamma\alpha$

Unrestricted Grammars $\Longleftrightarrow$ NTM

$$x_0x_1\cdots x_n$$
$$x_0x_1\cdots x_n\#S$$
$$x_0x_1\cdots x_n\#\alpha\gamma$$
$$x_0x_1\cdots x_n\#\beta\theta\gamma\alpha$$
$$\vdots$$

Unrestricted Grammars $\iff$ NTM

$$x_0 x_1 \cdots x_n$$
$$x_0 x_1 \cdots x_n \# S$$
$$x_0 x_1 \cdots x_n \# \alpha\gamma$$
$$x_0 x_1 \cdots x_n \# \beta\theta\gamma\alpha$$
$$\vdots$$
$$x_0 x_1 \cdots x_n \# x_1 x_2 \cdots x_n$$

Unrestricted Grammars $\Longleftrightarrow$ NTM

$x_0 x_1 \cdots x_n$

$x_0 x_1 \cdots x_n \# S$

$x_0 x_1 \cdots x_n \# \alpha\gamma$

$x_0 x_1 \cdots x_n \# \beta\theta\gamma\alpha$

$\vdots$

$x_0 x_1 \cdots x_n \# x_1 x_2 \cdots x_n$

accept

Unrestricted Grammars $\iff$ NTM

Suppose $(S_i', x_n', L) \in f(S_i, x_n)$,
then $S_i' x_{n-1} x_n' \rightarrow x_{n-1} S_i x_n$.