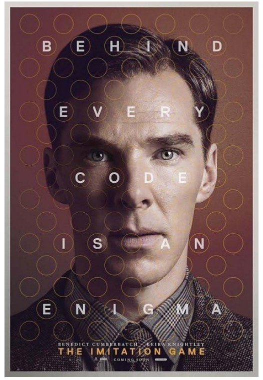
Automata & languages

A primer on the Theory of Computation



Laurent Vanbever
www.vanbever.eu

ETH Zürich (D-ITET)
October 4 2018

Part 3 out of 5

Last week, we started to learn about
closure and **equivalence** of regular languages

Last week, we started to learn about
closure and equivalence of regular languages

The class of regular languages
is closed under the

- union
- concatenation
- star

regular operations

The class of regular languages is closed under the

- union
- concatenation
- star

regular operations

if L_1 and L_2 are regular, then so are

$L_1 \cup L_2$

$L_1 . L_2$

L_1^*

Please check the companion document for the exact definition of ∅, ε, and ε-transition

Clarifications following Lecture 2

Laurent Vanbever (lvanbever@ethz.ch)

September 27, 2018

I realized that a lot of the misunderstandings today (including mine...) came from the fact that there was no formal definition of how does an NFA computes (the dynamic part of it). The same holds for the distinction between ϵ and ∅. Allow me to clarify these two points in this short document.

The difference between ϵ and ∅: Let us first consider the differences

Last week, we started to learn about closure and equivalence of regular languages

is equivalent to

DFA ≍ NFA

)(

REX

We'll finish that today then start asking ourselves whether all languages are regular

L_1	$\{0^n1^n \mid n \geq 0\}$
L_2	{w | w has an equal number of 0s and 1s}
L_3	{w | w has an equal number of occurrences of 01 and 10}

(only one of them actually is)

Advanced Automata

Thu Oct 4

1 Equivalence (the end)

- DFA
- NFA
- Regular Expression

2 Non-regular languages

3 Context-free languages

Three tough languages

1) $L_1 = \{0^n1^n \mid n \geq 0\}$

2) L_2 = {w | w has an equal number of 0s and 1s}

3) L_3 = {w | w has an equal number of occurrences of 01 and 10 as substrings}

Three tough languages

1) $L_1 = \{0^n1^n \mid n \geq 0\}$

2) L_2 = {w | w has an equal number of 0s and 1s}

3) L_3 = {w | w has an equal number of occurrences of 01 and 10 as substrings}

- In order to fully understand regular languages, we also must understand their limitations!

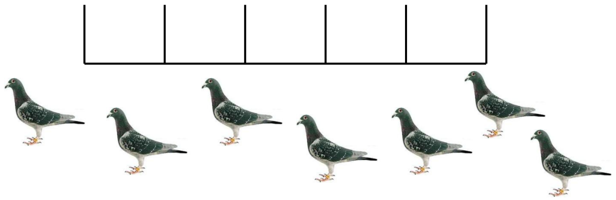
Pigeonhole principle

- Consider language L, which contains word $w \in L$.
- Consider an FA which accepts L, with $n < |w|$ states.
- Then, when accepting w, the FA must visit at least one state twice.

Pigeonhole principle

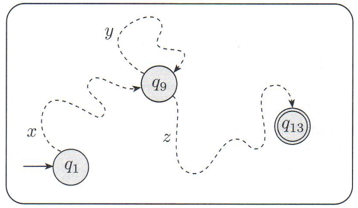
- Consider language L, which contains word w ∈ L.
- Consider an FA which accepts L, with n < |w| states.
- Then, when accepting w, the FA must visit at least one state twice.

- This is according to the pigeonhole (a.k.a. Dirichlet) principle:
  - If m>n pigeons are put into n pigeonholes, there's a hole with more than one pigeon.
  - That's a pretty fancy name for a boring observation...



Languages with unbounded strings

- Consequently, regular languages with unbounded strings can only be recognized by FA (finite! bounded!) automata if these long strings loop.



- The FA can enter the loop once, twice, …, and not at all.
- That is, language L contains all {xz, xyz, xy^2z, xy^3z, …}.

Pumping Lemma

- Theorem:

  Given a regular language *L*, there is a number *p* (the pumping number) such that:
  any string *u* in *L* of length ≥ *p* is pumpable within its first *p* letters.

Pumping Lemma

- Theorem:

  Given a regular language *L*, there is a number *p* (the pumping number) such that:
  any string *u* in *L* of length ≥ *p* is pumpable within its first *p* letters.

- A string $u \in L$ with $|u| \geq p$ is pumpable if it can be split in 3 parts *xyz* s.t.:
  - $|y| \geq 1$ (mid-portion *y* is non-empty)
  - $|xy| \leq p$ (pumping occurs in first *p* letters)
  - $xy^iz \in L$ for all $i \geq 0$ (can pump *y*-portion)

Pumping Lemma

- Theorem:

  Given a regular language L, there is a number p (the pumping number) such that:
  any string u in L of length $\geq p$ is pumpable within its first p letters.

- A string $u \in L$ with $|u| \geq p$ is pumpable if it can be split in 3 parts xyz s.t.:
  - $|y| \geq 1$ (mid-portion y is non-empty)
  - $|xy| \leq p$ (pumping occurs in first p letters)
  - $xy^iz \in L$ for all $i \geq 0$ (can pump y-portion)

- If there is no such p, then the language is not regular

Pumping Lemma Example

- Let L be the language $\{0^n1^n \mid n \geq 0\}$

- Assume (for the sake of contradiction) that L is regular
- Let p be the pumping length. Let u be the string 0^p1^p.
- Let's check string u against the pumping lemma:

- "In other words, for all $u \in L$ with $|u| \geq p$ we can write:
  - $u = xyz$ (x is a prefix, z is a suffix)
  - $|y| \geq 1$ (mid-portion y is non-empty)
  - $|xy| \leq p$ (pumping occurs in first p letters)
  - $xy^iz \in L$ for all $i \geq 0$ (can pump y-portion)"

Let's make the example a bit harder…

- Let L be the language {w | w has an equal number of 0s and 1s}

- Assume (for the sake of contradiction) that L is regular
- Let p be the pumping length. Let u be the string 0^p1^p.
- Let's check string u against the pumping lemma:

- "In other words, for all $u \in L$ with $|u| \geq p$ we can write:
  - $u = xyz$ (x is a prefix, z is a suffix)
  - $|y| \geq 1$ (mid-portion y is non-empty)
  - $|xy| \leq p$ (pumping occurs in first p letters)
  - $xy^iz \in L$ for all $i \geq 0$ (can pump y-portion)"

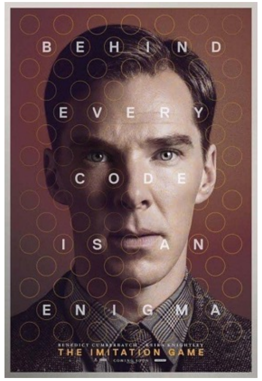
Now you try…

- Is $L_1 = \{ww \mid w \in (0 \cup 1)^*\}$ regular?

- Is $L_2 = \{1^n \mid n \text{ being a prime number } \}$ regular?

Automata & languages

A primer on the Theory of Computation



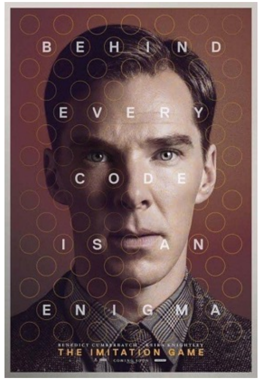
Part 1 regular language

Part 2 context-free language

Part 3 turing machine

Automata & languages

A primer on the Theory of Computation



regular language

Part 2 context-free language

turing machine

Motivation

- Why is a language such as $\{0^n1^n \mid n \geq 0\}$ not regular?!?
- It's really simple! All you need to keep track is the number of 0's…
- In this chapter we first study context-free grammars
  - More powerful than regular languages
  - Recursive structure
  - Developed for human languages
  - Important for engineers (parsers, protocols, etc.)

Example

- Palindromes, for example, are not regular.
- But there is a pattern.

Example

- Palindromes, for example, are not regular.
- But there is a pattern.

- Q: If you have one palindrome, how can you generate another?
- A: Generate palindromes recursively as follows:
  - Base case: ε, 0 and 1 are palindromes.
  - Recursion: If x is a palindrome, then so are 0x0 and 1x1.

Example

- Palindromes, for example, are not regular.
- But there is a pattern.

- Q: If you have one palindrome, how can you generate another?
- A: Generate palindromes recursively as follows:
  - Base case: ε, 0 and 1 are palindromes.
  - Recursion: If x is a palindrome, then so are 0x0 and 1x1.

- Notation: $x \rightarrow \varepsilon \mid 0 \mid 1 \mid 0x0 \mid 1x1$.
  - Each pipe ("|") is an or, just as in UNIX regexp's.
  - In fact, all palindromes can be generated from ε using these rules.

Example

- Palindromes, for example, are not regular.
- But there is a pattern.

- Q: If you have one palindrome, how can you generate another?
- A: Generate palindromes recursively as follows:
  - Base case: ε, 0 and 1 are palindromes.
  - Recursion: If x is a palindrome, then so are 0x0 and 1x1.

- Notation: $x \rightarrow \varepsilon \mid 0 \mid 1 \mid 0x0 \mid 1x1$.
  - Each pipe ("|") is an or, just as in UNIX regexp's.
  - In fact, all palindromes can be generated from ε using these rules.

- Q: How would you generate 11011011?

Context Free Grammars (CFG): Definition

- Definition: A context free grammar consists of (V, Σ, R, S) with:
  - V: a finite set of variables (or symbols, or non-terminals)
  - Σ: a finite set set of terminals (or the alphabet)
  - R: a finite set of rules (or productions)
    of the form $v \rightarrow w$ with $v \in V$, and $w \in (\Sigma_\varepsilon \cup V)^*$
    (read: "v yields w" or "v produces w")
  - $S \in V$: the start symbol.

Context Free Grammars (CFG): Definition

- Definition: A context free grammar consists of (V, Σ, R, S) with:
  - V: a finite set of variables (or symbols, or non-terminals)
  - Σ: a finite set set of terminals (or the alphabet)
  - R: a finite set of rules (or productions)
    of the form $v \rightarrow w$ with $v \in V$, and $w \in (\Sigma_\varepsilon \cup V)^*$
    (read: "v yields w" or "v produces w")
  - $S \in V$: the start symbol.
- Q: What are (V, Σ, R, S) for our palindrome example?

Derivations and Language

- Definition: The derivation symbol "$\Rightarrow$" (read "1-step derives" or "1-step produces") is a relation between strings in $(\Sigma \cup V)^*$.
  We write $x \Rightarrow y$ if x and y can be broken up as $x = svt$ and $y = swt$ with $v \rightarrow w$ being a production in R.

Derivations and Language

- Definition: The derivation symbol "$\Rightarrow$" (read "1-step derives" or "1-step produces") is a relation between strings in $(\Sigma \cup V)^*$.
  We write $x \Rightarrow y$ if x and y can be broken up as $x = svt$ and $y = swt$ with $v \rightarrow w$ being a production in R.
- Definition: The derivation symbol "$\Rightarrow^*$", (read "derives" or "produces" or "yields") is a relation between strings in $(\Sigma \cup V)^*$. We write $x \Rightarrow^* y$ if there is a sequence of 1-step productions from x to y. I.e., there are strings x_i with i ranging from 0 to n such that $x = x_0$, $y = x_n$ and $x_0 \Rightarrow x_1$, $x_1 \Rightarrow x_2$, $x_2 \Rightarrow x_3$, … , $x_{n-1} \Rightarrow x_n$.

Derivations and Language

- Definition: The derivation symbol "$\Rightarrow$" (read "1-step derives" or "1-step produces") is a relation between strings in $(\Sigma \cup V)^*$.
  We write $x \Rightarrow y$ if x and y can be broken up as $x = svt$ and $y = swt$ with $v \rightarrow w$ being a production in R.
- Definition: The derivation symbol "$\Rightarrow^*$", (read "derives" or "produces" or "yields") is a relation between strings in $(\Sigma \cup V)^*$. We write $x \Rightarrow^* y$ if there is a sequence of 1-step productions from x to y. I.e., there are strings x_i with i ranging from 0 to n such that $x = x_0$, $y = x_n$ and $x_0 \Rightarrow x_1$, $x_1 \Rightarrow x_2$, $x_2 \Rightarrow x_3$, … , $x_{n-1} \Rightarrow x_n$.
- Definition: Let G be a context-free grammar. The context-free language (CFL) generated by G is the set of all terminal strings which are derivable from the start symbol. Symbolically: $L(G) = \{w \in \Sigma^* \mid S \Rightarrow^* w\}$

Example: Infix Expressions

- Infix expressions involving {+, ×, *a*, *b*, *c*, (,)}
- *E* stands for an expression (most general)
- *F* stands for factor (a multiplicative part)
- *T* stands for term (a product of factors)
- *V* stands for a variable: *a, b,* or *c*

- Grammar is given by:
  - $E \to T \mid E + T$
  - $T \to F \mid T \times F$
  - $F \to V \mid (E)$
  - $V \to a \mid b \mid c$

- Convention: Start variable is the first one in grammar (*E*)

Example: Infix Expressions

- Consider the string *u* given by $a \times b + (c + (a + c))$
- This is a valid infix expression. Can be generated from *E*.

1. A sum of two expressions, so first production must be $E \Rightarrow E + T$
2. Sub-expression $a \times b$ is a product, so a term so generated by sequence $E + T \Rightarrow T + T \Rightarrow T \times F + T \Rightarrow^* a \times b + T$
3. Second sub-expression is a factor only because a parenthesized sum. $a \times b + T \Rightarrow a \times b + F \Rightarrow a \times b + (E) \Rightarrow a \times b + (E + T) \ldots$
4. $E \Rightarrow E + T \Rightarrow T + T \Rightarrow T \times F + T \Rightarrow F \times F + T \Rightarrow V \times F + T \Rightarrow a \times F + T \Rightarrow a \times V + T \Rightarrow a \times b + T \Rightarrow a \times b + F \Rightarrow a \times b + (E) \Rightarrow a \times b + (E + T) \Rightarrow a \times b + (T + T) \Rightarrow a \times b + (F + T) \Rightarrow a \times b + (V + T) \Rightarrow a \times b + (c + T) \Rightarrow a \times b + (c + F) \Rightarrow a \times b + (c + (E)) \Rightarrow a \times b + (c + (E + T)) \Rightarrow a \times b + (c + (T + T)) \Rightarrow a \times b + (c + (F + T)) \Rightarrow a \times b + (c + (a + T)) \Rightarrow a \times b + (c + (a + F)) \Rightarrow a \times b + (c + (a + V)) \Rightarrow a \times b + (c + (a + c))$

Left- and Right-most derivation

- The derivation on the previous slide was a so-called left-most derivation.

- In a right-most derivation, the variable most to the right is replaced.
  - $E \Rightarrow E + T \Rightarrow E + F \Rightarrow E + (E) \Rightarrow E + (E + T) \Rightarrow$ etc.

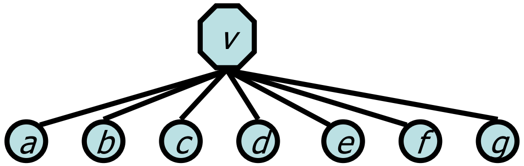
Ambiguity

- There can be a lot of ambiguity involved in how a string is derived.

- Another way to describe a derivation in a unique way is using derivation trees.

Derivation Trees

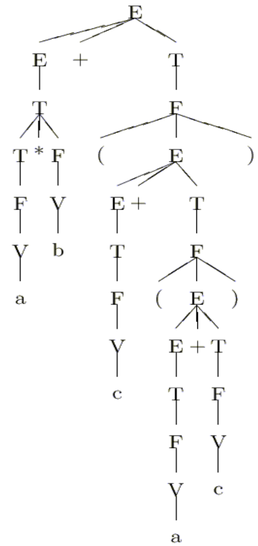
- In a derivation tree (or parse tree) each node is a symbol. Each parent is a variable whose children spell out the production from left to right. For, example *v* → *abcdefg*:



- The root is the start variable.
- The leaves spell out the derived string from left to right.

Derivation Trees

- On the right, we see a derivation tree for our string $a \times b + (c + (a + c))$
- Derivation trees help understanding semantics! You can tell how expression should be evaluated from the tree.



Ambiguity

<sentence>	→	***<action>* | *<action>*** with ***<subject>***
<action>	→	***<subject><activity>***
<subject>	→	***<noun>* | *<noun>*** and ***<subject>***
<activity>	→	***<verb>* | *<verb><object>***
<noun>	→	Hannibal | Clarice | rice | onions
<verb>	→	ate | played
<prep>	→	with | and | or
<object>	→	***<noun>* | *<noun><prep><object>***

- Clarice played with Hannibal
- Clarice ate rice with onions
- Hannibal ate rice with Clarice

- Q: Are there any suspect sentences?

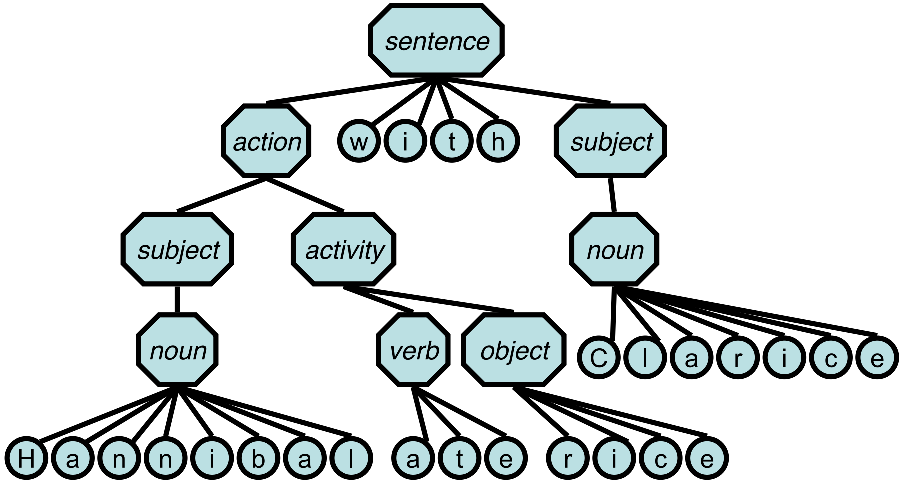
Ambiguity

- A: Consider "Hannibal ate rice with Clarice"

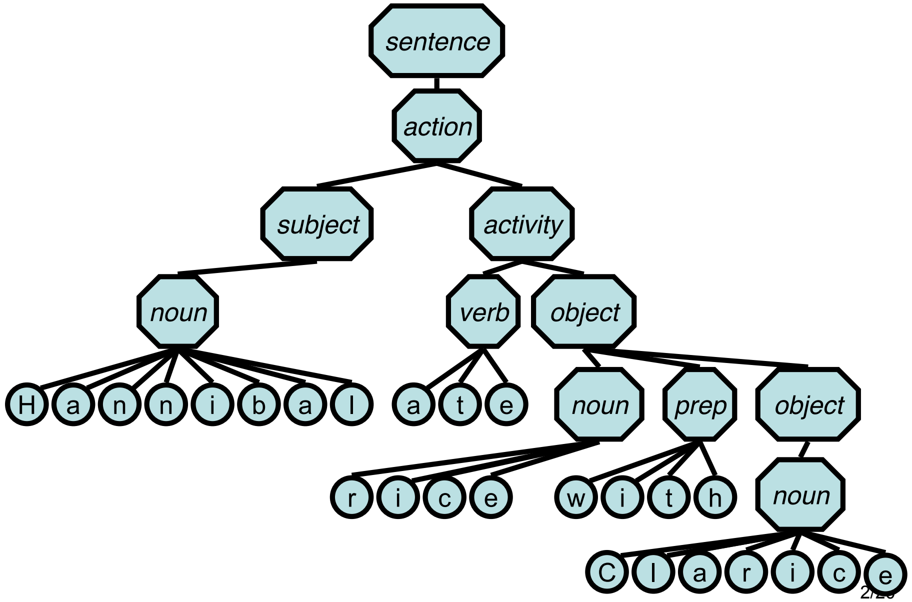
- This could either mean
  - Hannibal and Clarice ate rice *together*.
  - Hannibal ate rice and *ate* Clarice.

- This ambiguity arises from the fact that the sentence has two different parse-trees, and therefore two different interpretations:

Hannibal and Clarice Ate



Hannibal the Cannibal



Ambiguity: Definition

- Definition:

  A string *x* is said to be ambiguous relative the grammar *G*
  if there are two essentially different ways to derive *x* in *G*.
  - *x* admits two (or more) different parse-trees
  - equivalently, *x* admits different left-most [resp. right-most] derivations.

- A grammar *G* is said to be ambiguous if there is some string *x* in *L*(*G*) which is ambiguous.

Ambiguity: Definition

- Definition:

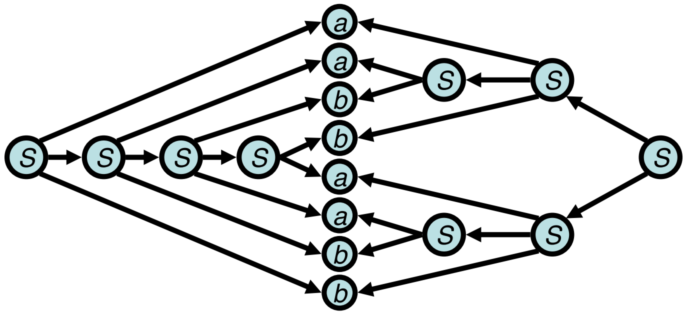
  A string *x* is said to be ambiguous relative the grammar *G*
  if there are two essentially different ways to derive *x* in *G*.
  - *x* admits two (or more) different parse-trees
  - equivalently, *x* admits different left-most [resp. right-most] derivations.

- A grammar *G* is said to be ambiguous if there is some string *x* in *L*(*G*) which is ambiguous.

- Question: Is the grammar $S \rightarrow ab \mid ba \mid aSb \mid bSa \mid SS$ ambiguous?
  - What language is generated?

Ambiguity

- Answer: $L(G)$ = the language with equal no. of *a'* s and *b'* s
- Yes, the language is ambiguous:



CFG's: Proving Correctness

- The recursive nature of CFG's means that they are especially amenable to correctness proofs.
- For example let's consider the grammar
  $$G = (S \rightarrow \varepsilon \mid ab \mid ba \mid aSb \mid bSa \mid SS)$$
- We claim that L(G) = L = $\{ x \in \{a,b\}^* \mid n_a(x) = n_b(x) \}$, where $n_a(x)$ is the number of *a'*s in *x,* and $n_b(x)$ is the number of *b'*s.
- *Proof*: To prove that L = L(G) is to show both inclusions:
  i. $L \subseteq L(G)$: Every string in L can be generated by *G*.
  ii. $L \supseteq L(G)$: *G* only generate strings of L.
    - This part is easy, so we concentrate on part i.

Proving $L \subseteq L(G)$

- $L \subseteq L(G)$: Show that every string *x* with the same number of *a'*s as *b'*s is generated by *G*. Prove by induction on the length $n = |x|$.
- Base case: The empty string is derived by $S \rightarrow \varepsilon$.
- Inductive hypothesis: Assume $n > 0$. Let *u* be the smallest non-empty prefix of *x* which is also in *L*.
  - Either there is such a prefix with $|u| < |x|$, then x = uv whereas v ∈ L as well, and we can use S → SS and repeat the argument.
  - Or x = u. In this case notice that *u* can't start and end in the same letter. If it started and ended with *a* then write *x* = *ava*. This means that *v must have* 2 more *b'*s than *a'*s. So somewhere in *v* the *b'*s of *x* catch up to the *a'*s which means that there's a smaller prefix in *L*, contradicting the definition of *u* as the *smallest* prefix in *L*. Thus for some string *v* in *L* we have *x* = *avb* OR *x* = *bva.* We can use either $S \rightarrow aSb$ OR $S \rightarrow bSa$.

Designing Context-Free Grammars

- As for regular languages this is a creative process.
- However, if the grammar is the union of simpler grammars, you can design the simpler grammars (with starting symbols S_1, S_2, respectively) first, and then add a new starting symbol/production $S \rightarrow S_1 \mid S_2$.
- If the CFG happens to be regular as well, you can first design the FA, introduce a variable/production for each state of the FA, and then add a rule $x \rightarrow ay$ to the CFG if $\delta(x,a) = y$ is in the FA. If a state x is accepting in FA then add $x \rightarrow \varepsilon$ to CFG. The start symbol of the CFG is of course equivalent to the start state in the FA.
- There are quite a few other tricks. Try yourself…