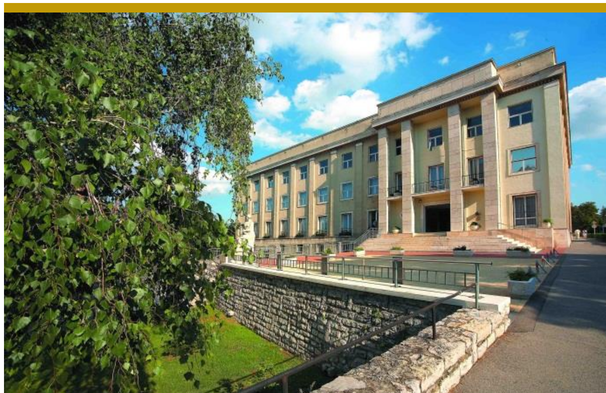
Dr. Heckl István:

Theory of Digital Computation lecture notes



A felsőfokú informatikai oktatás minőségének fejlesztése, modernizációja

TÁMOP-4.1.2.A/1-11/1-2011-0104

Főkedvezményezett:
Pannon Egyetem
8200 Veszprém
Egyetem u. 10.

Kedvezményezett:
Szegedi Tudományegyetem
6720 Szeged
Dugonics tér 13.

2014

ÚJ SZÉCHENYI TERV
MAGYARORSZÁG MEGÚJUL
A projekt az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósul meg.

Elements of the Theory of Computation

Lesson 1

1.1. Sets

1.2. Relations and functions

1.3. Special types of binary relations

University of Pannonia

Dr. István Heckl, Istvan.Heckl@gmail.com

Subject information

- Lecturer: Dr. István Heckl, Istvan.Heckl@gmail.com
- http://oktatas.mik.uni-pannon.hu/
  - registration
  - course: Theory of the elements of computation

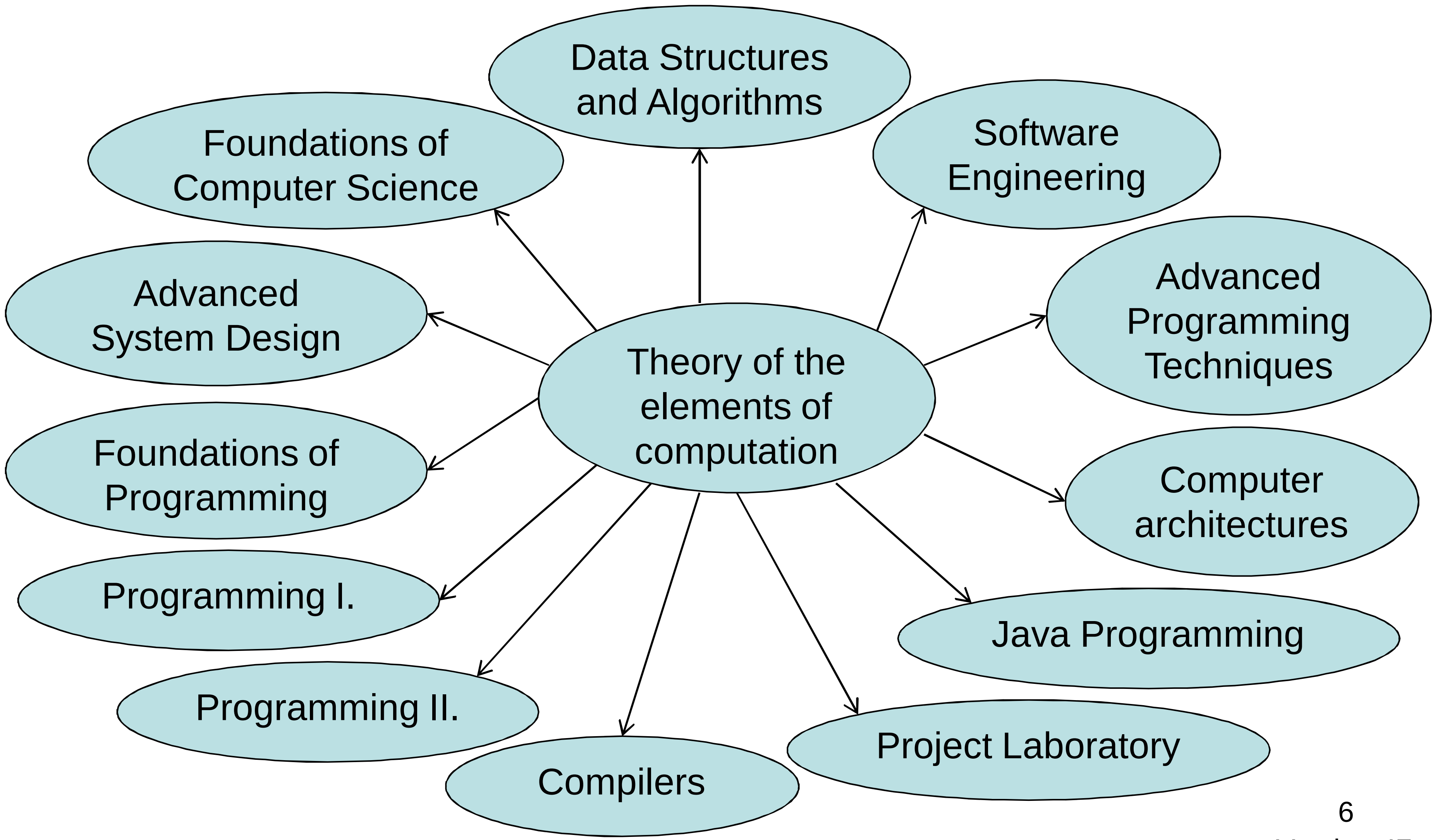
Subject information

- Exercise book:
  - the whole lecture should be written down
  - use exercise book (not sheets) and pen
  - number each page
  - write date, lecture number, signature for each lecture
  - each lecture should start on a new page

Subject information

- Subject code:
  - VEMISA3244D
- Signature: at least 50% result at ZH
- Subject name in Hungarian: A digitális számítás elmélete
- Literature: Harry R. Lewis, Christos H. Papadimitriou: Elements of the Theory of Computation, Prentice Hall, Inc., 1998. (second edition)
  - this presentation is based on this book
- Irodalom: Bach Iván, Formális nyelvek

Subject information



Scope

- The theory of computations tries to answer the question: what is an algorithm?
  - algorithm theory examine given algorithms
  - we would like to know what is algorithm in general
  - e.g.:

```
input x
while x > 10
    x = x – 3
end
```

Scope

- Does it halt?

```
input x
x = x * 2
while x is even
   x = x * 2
end
```

- Does it halt?

```
input x
repeat
   if x is even x = x /2
   else x = x*3+1
until x > 1
```

Scope

- Any algorithm can be seen as a language
  - the words of a language: (input1, output1), (input2, output2), ...
  - a language can be recognized by an automata
  - we keep learning more and more complex classes of languages
- The subject
  - shows how automata (e.g.: computers) work
  - is the basis for writing compilers
    - e.g.: C++ compiler

Scope

- ADC (Automata Drawing and Converting Tool) can be found in the Moodle

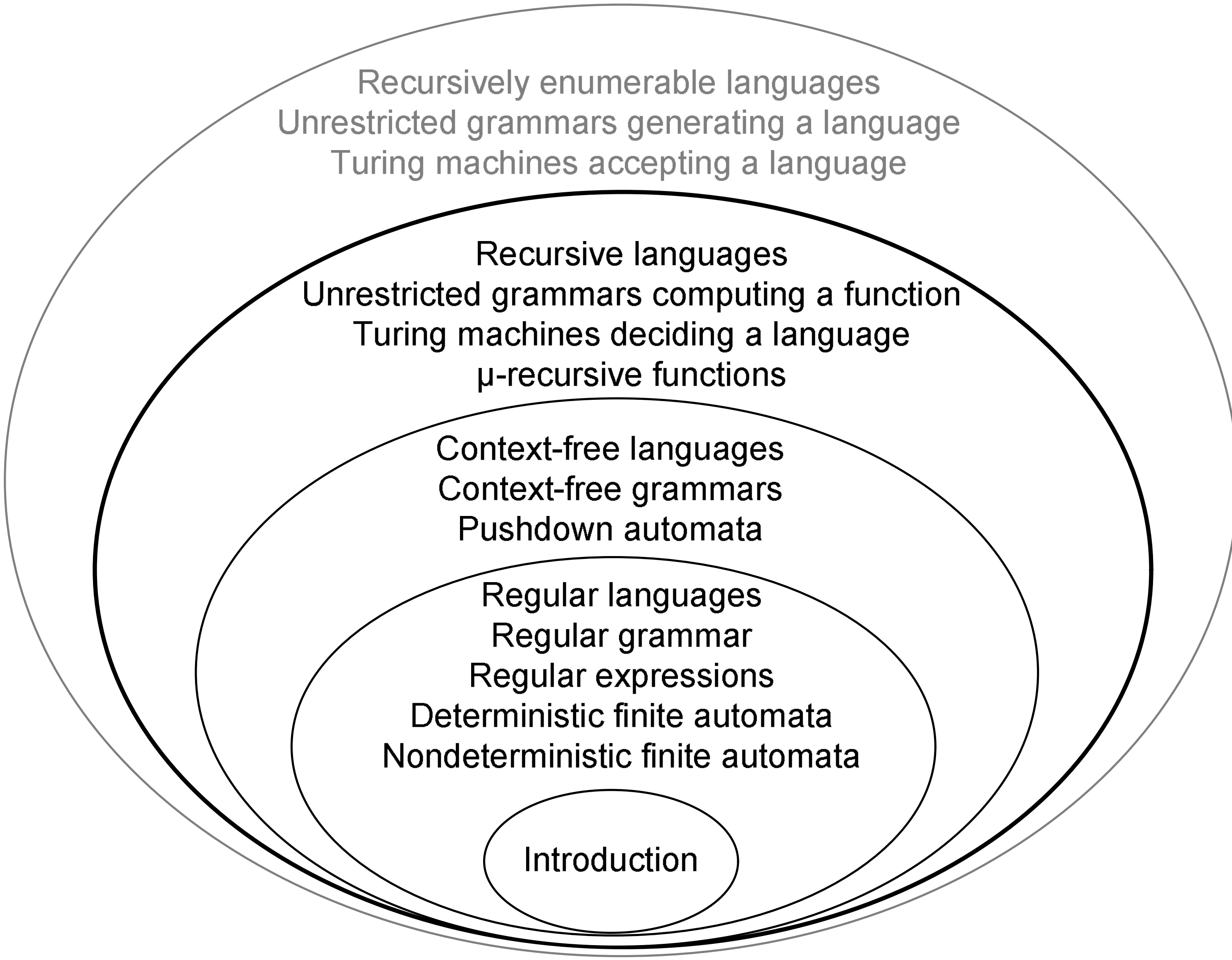
Scope

- We need exact terms for languages, grammars, computation, algorithms, ...
  - no exact term for algorithms
- We need to know what the unsolvable problems are, what the very hard problems are, what problems can be solved easily
  - halting problem is unsolvable
  - traveling salesman is NP complete

Scope for programmers

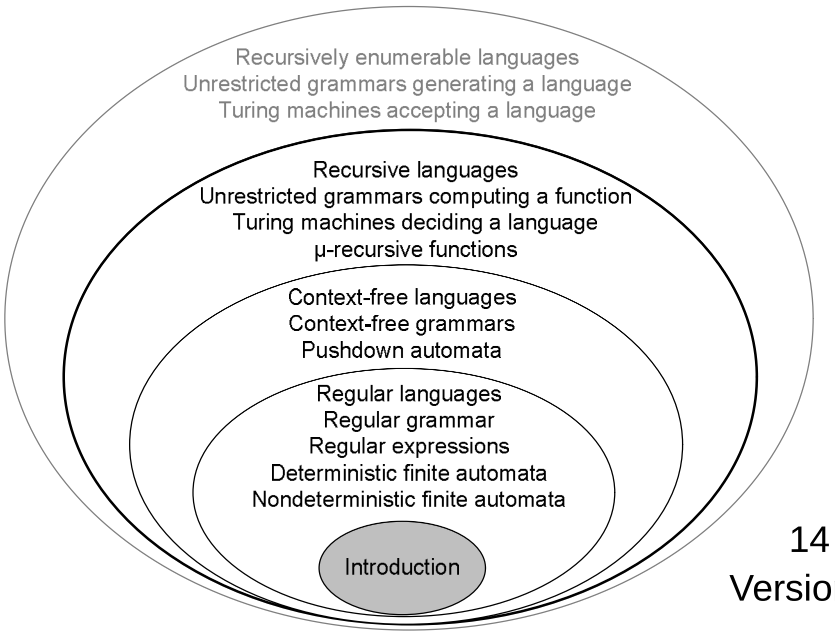
- Simple decisions: based solely on inputs
  - e.g.: if the outside temperature is lower than 6 °C I take a hat
  - there can be many inputs
- Complex decisions: based on inner state and on inputs
  - e.g.: the outside temperature is lower than 6 °C but I also know that dad will take me to school by car so I do not take a hat

Onion diagram of topics



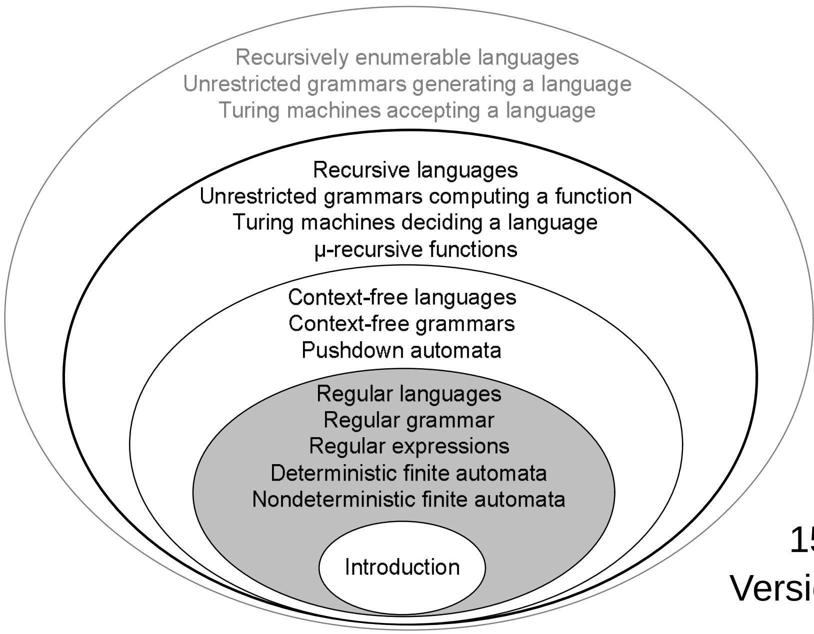
Content: Introduction

1. Sets, Relations and functions, Special binary relations
2. Finite and infinite sets, Three fundamental proof techniques, Closures and algorithms
3. Alphabets and languages, Finite representations of languages



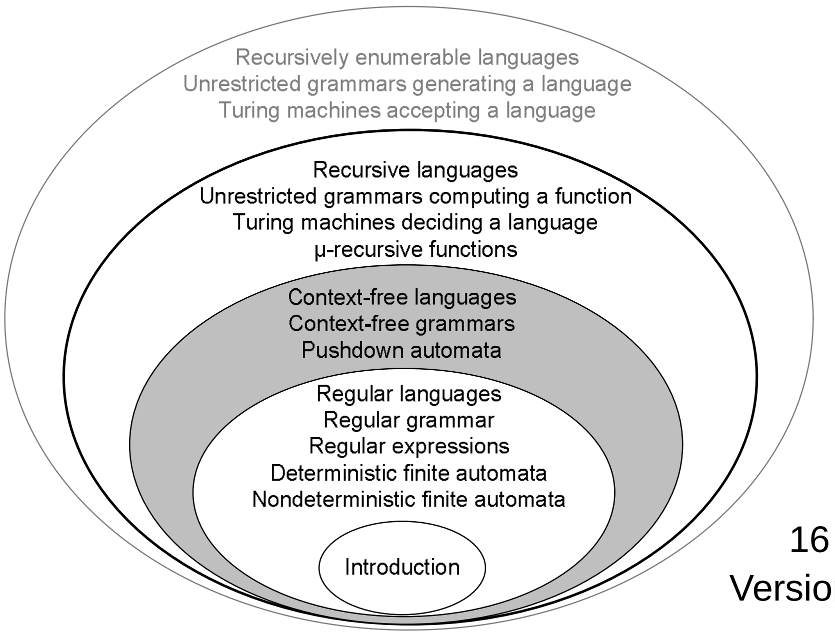
Content: Finite automaton

4. Deterministic finite automata
5. Non-deterministic finite automata
6. Finite automata and regular expressions, Languages that are and are not regular



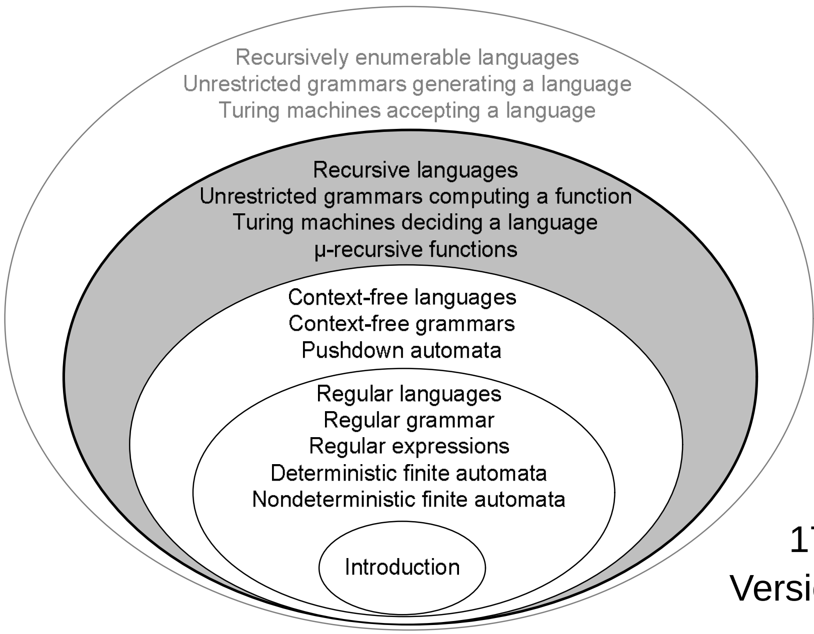
Content: Context-free languages

7. Context-free grammars
8. Pushdown automata
9. Pushdown automata and context-free grammars, Languages that are and are not context free



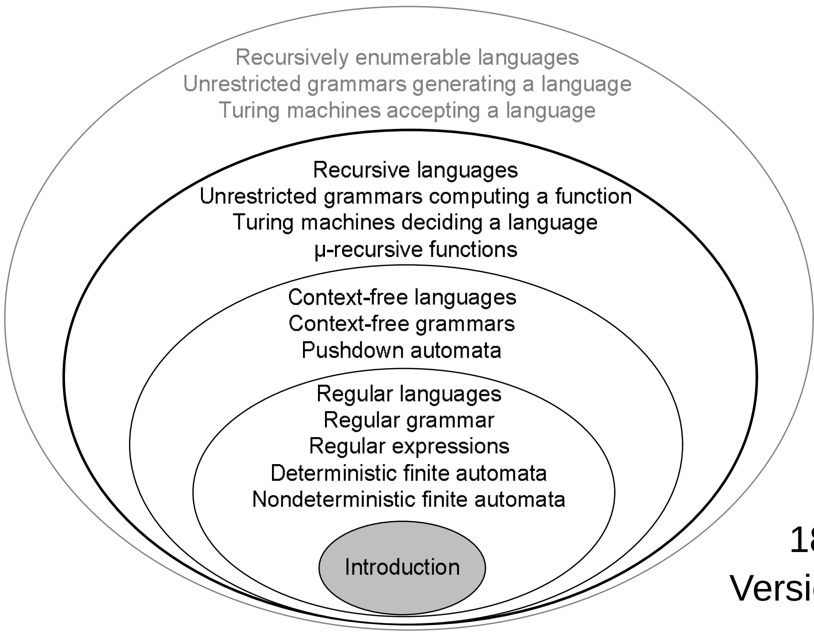
Content: Turing machines

10. The definition of a Turing machine
11. Computing with Turing machines
12. The Church-Turing thesis, Universal Turing machine, The halting problem



Introductions 1

- Boolean algebra
- Sets
- Sets operations
- Relations and functions
- Special types of binary relations



Boolean algebra

- Statements can be: true or false
- Examples:
  - the word "watermelon" has more e than o: true
  - the word "watermelon" starts with z: false
- George Boole (1815-1864)
  - English mathematician and philosopher
  - the inventor of Boolean logic, the basis of modern digital computer logic

Boolean algebra

- Boolean operators:
  - combines two statements or modify a single statement
  - and, or, not, xor, xand (=), nor, nand, implication

Boolean algebra

a	b	~a	and	or	xand, =	xor, ^	nand	nor	impl, →
0	0	1	0	0	1	0	1	1	1
0	1	1	0	1	0	1	1	0	1
1	0	0	0	1	0	1	1	0	0
1	1	0	1	1	1	0	0	0	1

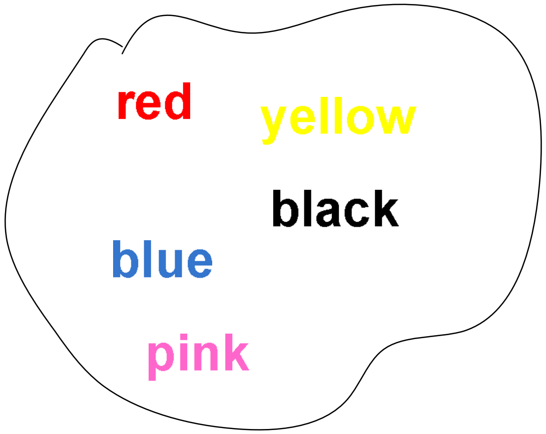
Boolean algebra

- Symbols:
  - a = blue-eyed
  - b = long-haired
  - c= blonde
- Formulate your statement:
  - S1 = b or (a and c)
  - S2 = a and b and c

a	b	c	S1	S2
0	0	0	0	0
0	0	1	0	0
0	1	0	1	0
0	1	1	1	0
1	0	0	0	0
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Sets

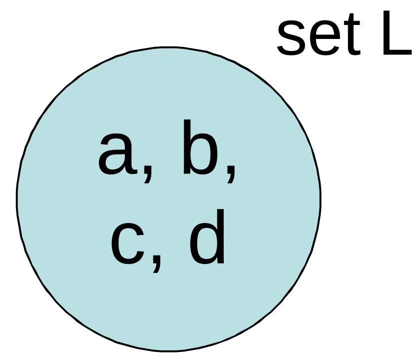
- Description of set: collection of objects
  - collection = set
  - objects = elements
  - e.g.: L = {a, b, c, d}, S = {colors}
- Sets do not contain repetitions of elements
  - {red, blue, red} is not a proper set
- Order of elements is unimportant
  - {1, 3, 9} = {9, 3, 1} = {3, 1, 9}
- Elements can be sets too:
  - {2, red, {blue, d}}
- Automata are defined with sets



Sets

- A set can be specified:
  - listing all its elements
    - infinite sets cannot be defined in this way
    - e.g.: M = {xx, yy, zz}
  - giving a property which holds for every element
    - such property does not always exist
    - e.g.:
      - $K = \{x \in N : x \text{ is not divisible by } 2\}$
      - A = {words, that contain 'a'}

Nomenclature



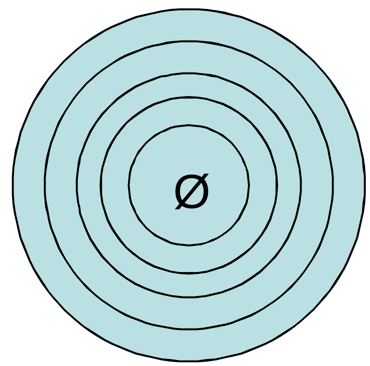
- Sets:
  - b $\in$ L:
    - b is an element of set L
  - z $\notin$ L:
    - z is not an element of set L
  - |L| is the cardinality of set L
- Definition:
  - read '$\rightarrow$' as then
  - read ',' as and
  - read '$\leftrightarrow$' as if and only if

Sets

- Two sets are equal:
  - if and only if they have the same elements
- Definition of singleton: a set with one element
  - |L| = 1
  - e.g.: L = {a}
- Definition of empty set: a set with no element
  - |L| = 0
  - e.g.: L = Ø or L = {}
  - beware: Ø = {} ≠ { {} }

Sets

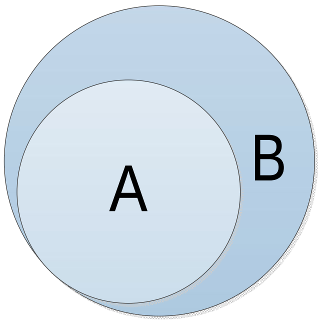
- Neumann onions:
  - 0 = {}
  - 1 = { {} }
  - 2 = { {{}} }
  - ...



- János Neumann (1903 –1957)
  - Hungarian mathematician

Sets

- Definition of subset: A is the subset of B, if each element of A is also in B
  - notation: $A \subseteq B$



- Properties:
  - any set is subset to itself
  - if $A \subseteq B$, $A \neq B \rightarrow$ A is a proper subset of B
    - notation: $A \subset B$
  - $A = B \leftrightarrow A \subseteq B, B \subseteq A$
  - Ø is the subset of every set

Sets

- Give an algorithm for checking if x is an element of A!

```
elementTest(x, A)
for i = 0 to |A|-1
     if A[i] == x
          return true
return false
```

$$\in$$

Sets

- True or false
  - $5 \in \{5, 6, 7\}$
  - $6 \in \{5, 7, 9\}$
  - $\{5, 6\} \in \{5, 6, \{5, 6\}\}$
  - $\{5, 6\} \in \{5, 6, 7\}$
  - $a \in \{\{a\}\}$
  - $\{a, b\} \in \{a, b\}$
  - $\{a, b\} \in \{a, \{a, b\}, b\}$
  - $\varnothing \in \varnothing$
  - $\varnothing \in \{\varnothing\}$

Sets

- Give an algorithm for checking if A is a subset of B!

```
subsetTest(A, B)
for i = 0 to |A|-1
     if elementTest(A[i], B) == false
          return false
return true
```

$$\subseteq$$

Sets

- True or false
  - $\{5, 6\} \subseteq \{5, 6, 7\}$
  - $\{6, 8\} \subseteq \{5, 6, 7\}$
  - $\{a, b\} \subseteq \{a, b\}$
  - $\{a, b\} \subseteq \{a, b, \{a, b\}\}$
  - $a \subseteq \{a, b, \{a, b\}\}$
  - $\varnothing \subseteq \varnothing$
  - $\varnothing \subseteq \{\varnothing\}$

Set operations

- Definition of union: a set which contains all the elements of two sets
  - $A \cup B = \{x : x \in A \text{ or } x \in B\}$
  - e.g.:
    - {red, green} $\cup$ {blue} = {red, green, blue}
    - $\{1, 3, 9\} \cup \{3, 5, 7\} = \{1, 3, 5, 7, 9\}$
  - an important property: the finite automata is closed under the union operation

Set operations

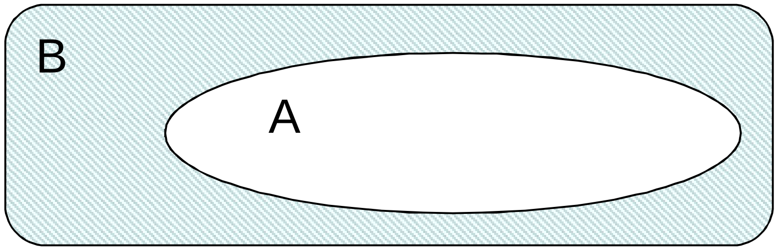
- Definition of intersection: a set which contains the elements which are common in two sets
  - $A \cap B = \{x : x \in A \text{ and } x \in B\}$
  - e.g.:
    - $\{1, 3, 9\} \cap \{3, 5, 7\} = \{3\}$
    - $\{red, green\} \cap \{blue\} = \emptyset$

Set operations

- Definition of difference between A and B: a set which contains all elements of A that are not in B
  - $A \setminus B = \{x : x \in A \text{ and } x \notin B\}$
  - e.g.:
    - $\{1, 3, 9\} \setminus \{3, 5, 7\} = \{1, 9\}$
    - {red, green} \ {blue} = {red, green}

Set operations

- Definition of disjoint sets: sets with no common element
  - $A \cap B = \emptyset$
  - e.g.:
    - A = {1, 4, 33}, B = {2, 6, 12}
    - A = {dogs}, B = {cats}
- Definition of complementer set:
  - A^C = {x: x is element of base set, but x is not element of A}
  - e.g.: B = {1, 3, 5}, A = {1, 3}, $A \subseteq B$ and A^C= {5}



Set operations

- Set operations with more than two sets:
  - $\cup$L: the set, whose elements are the elements of the sets in L
    - L = {{a, b}, {b, c}, {c, d}}
    - $\cup$L = {a, b} $\cup$ {b, c} $\cup$ {c, d} = {a, b, c, d}
  - $\cap$L: the set, whose elements are the common elements of the sets in L
    - L = {{a, b}, {b, c}, {b, d}}
    - $\cap$L = {a, b} $\cap${b, c} $\cap$ {c, d} = {b}

Set operations

- Questions:
  - A = {1, 3, 5, 6, 7}
  - B = {2, 3, 4, 5, 7}

  - $A \cap B$ = {3, 5, 7}
  - $A \cup B$ = {1, 2, 3, 4, 6, 7}
  - $A \setminus B$ = { 1, 4, 6}
  - $(A \setminus B) \cup (A \cap B) = A$
  - $(A \setminus B) \cap (A \cup B) = A \setminus B$

Set operations

- Properties of the set operations:
  - idempotency: $A \cap A = A$; $A \cup A = A$
    - for unary operator it means: $f(f(A))=f(A)$
  - commutativity: $A \cup B = B \cup A$; $A \cap B = B \cap A$
  - associativity: $(A \cup B) \cup C = A \cup (B \cup C)$;
    $(A \cap B) \cap C = A \cap (B \cap C)$
  - distributivity: $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$
    $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$
  - absorption: $A \cap (A \cup B) = A$; $A \cup (A \cap B) = A$

Set operations

- Properties of the set operations:
  - De' Morgan's Laws:

$$\overline{(A \cup B)} = \overline{A} \cap \overline{B}$$

$$\overline{(A \cap B)} = \overline{A} \cup \overline{B}$$

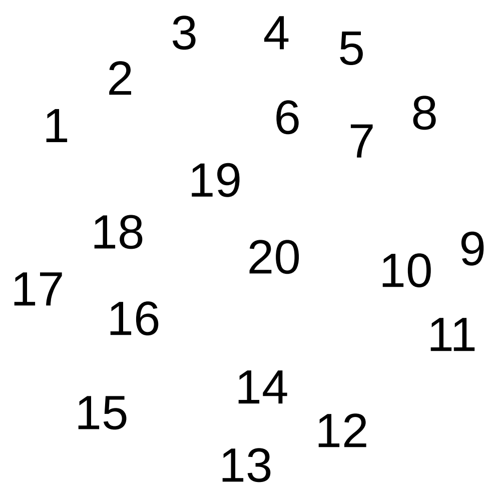
- The proof will use it which says that NFA is closed under intersection ➡
- Augustus De Morgan (1806 –1871)
  - British mathematician and logician

Set operations

- Definition of power set: collection of all subsets of a set
  - P(A), 2^A
  - $|P(A)| = 2^{|A|}$
  - e.g.:
    - P(Ø) = {Ø}
    - P({a}) = {Ø, {a}}
    - P({b, c}) = {Ø, {b}, {c}, {b, c}}
    - P({d, e, f}) = {Ø, {d}, {e}, {f}, {d, e}, {d, f}, {e, f}, {d, e, f}}
    - P({g, h, i, j}) = {Ø, {g}, {h}, {i}, {j},
      {g, h}, {g, i}, {g, j}, {h, i}, {h, j}, {i, j},
      {g, h, i}, {g, h, j}, {g, i, j}, {h, i, j}, {g, h, i, j}}

Set operations

- Definition of partition: Π is a partition of A if
  - $\Pi \subseteq P(A)$
  - $\varnothing \notin \Pi$
  - the members of Π are disjoint
  - $\cup\Pi = A$



Set operations

- Example for power sets:
  - P({a, b, c}) = { Ø, {c}, {b}, {a}, {a, b}, {a, c}, {b, c}, {a, b, c} }

a	**b**	**c**	**P({a, b, c})**
0	0	0	Ø
0	0	1	{c}
0	1	0	{b}
0	1	1	{b, c}
1	0	0	{a}
1	0	1	{a, c}
1	1	0	{a, b}
1	1	1	{a, b, c}

Set operations

- Examples for power sets
  - $P(\emptyset) = \{\emptyset\}$
  - $P(\{\emptyset\}) = \{ \emptyset, \{\emptyset\} \}$
  - $P(\{\emptyset, \{\emptyset\}\}) = \{ \emptyset, \{\emptyset\}, \{\{\emptyset\}\}, \{\emptyset, \{\emptyset\}\} \}$

- True or false
  - $\emptyset \in P(\emptyset)$
  - $\emptyset \subseteq P(\emptyset)$
  - $\{a, b\} \subseteq P(\{a, b\})$
  - $\{a, b\} \in P(\{a, b\})$

Set operations

- Examples for operation with sets:

  – $(\{1, 3, 5\} \cup \{3, 1\}) \cap \{3, 5, 7\} =$

  $= \{1, 3, 5\} \cap \{3, 5, 7\} =$

  $= \{3, 5\}$

  – $(\{1, 2, 5\} \setminus \{5, 7, 9\}) \cup (\{5, 7, 9\} \setminus \{1, 2, 5\}) =$

  $= \{1, 2\} \cup \{7, 9\} =$

  $= \{1, 2, 7, 9\}$

Set operations

- Examples for operation with sets:

  – $\{3, 5\} \cup \{3, \{3, 5\}, \{7\}\} \cup (\bigcap\{\{1, 2, 3\}, \{2, 3, 4\}\}) =$

  $= \{3, 5, \{3, 5\}, \{7\}\} \cup \{2, 3\} =$

  $= \{2, 3, 5, \{3, 5\}, \{7\}\}$

  – $P(\{2, 3, 5\}) \setminus P(\{3, 5\}) =$

  $= \{\{2\}, \{2, 3\}, \{2, 5\}, \{2, 3, 5\}\}$

Relations and functions

- Definition of ordered n-tuple: $(a_1, \ldots, a_n)$ an object made of other objects, $a_1, \ldots a_n$, where the order of the components is important
- Ordered 2-, 3-, 4-, 5-, 6-tuples are called
  - pairs, triples, quadruples, quintuples and sextuples
  - context free languages are quadruples
- n-tuples can be defined with sets
  - e.g.: $(a, b) = \{\{a\}, \{a, b\}\}$
- Properties:
  - the order matters: $(a, b) \neq (b, a)$
  - $(a, b) = (c, d) \leftrightarrow a = c, b = d$

Relations and functions

- Definition of Cartesian product:
  - $A \times B = \{(a, b) : a \in A, b \in B\}$
  - e.g.: $\{1, 3\} \times \{b, c\}=\{(1, b), (1, c), (3, b), (3, c)\}$
- n-fold Cartesian product $A_1 \times \ldots \times A_n$: $\{(a_1, ..., a_n) : a_i \in A_i\}$
  - if $A_1 = A_2 = A_n \rightarrow A_1 \times \ldots \times A_n = A^n$
  - e.g.: $N \times N = N^2$
- René Descartes (1596 –1650)
  - French mathematician
  - latinized form: *Renatus Cartesius*

Relations and functions

- Examples for Cartesian product:
  - $\{1, 3, 9\} \times \{b, c, d\} =$

    $= \{(1, b), (1, c), (1, d), (3, b), (3, c),$
    $(3, d), (9, b), (9, c), (9, d)\}$

  - $\{1\} \times \{1, 2\} \times \{1, 2, 3\} =$

    $= \{(1, 1, 1), (1, 1, 2), (1, 1, 3), (1, 2, 1), (1, 2, 2), (1, 2, 3)\}$

  - $P(\{1, 2\}) \times \{1, 2\} =$

    $= \{\emptyset, \{1\}, \{2\}, \{1, 2\}\} \times \{1, 2\} =$

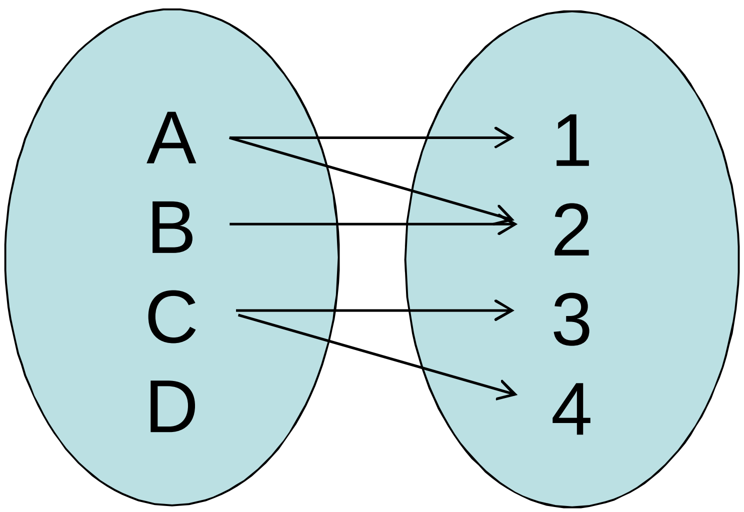
    $= \{(\emptyset, 1), (\emptyset, 2), (\{1\}, 1), (\{1\}, 2), (\{2\}, 1),$
    $(\{2\}, 2), (\{1, 2\}, 1), (\{1, 2\}, 2)\}$

Relations and functions

- True or false:
  - $(a, b) \in \{(a, b)\} \times \{a, b\}$ $= \{((a, b), a), ((a, b), b)\}$
  - $\{a, b\} \in \{b, a\} \times \{b\}$ $= \{(b, b), (a, b)\}$
  - $\{a, b\} \in \{a\} \times \{b\}$ $= \{(a, b)\}$
  - $(a, b) \in \{a\} \times \{b\}$ $= \{(a, b)\}$
  - $\{(a, b)\} \subseteq \{a\} \times \{b\}$ $= \{(a, b)\}$
  - $\{a, b\} \subseteq \{a\} \times \{b\}$ $= \{(a, b)\}$

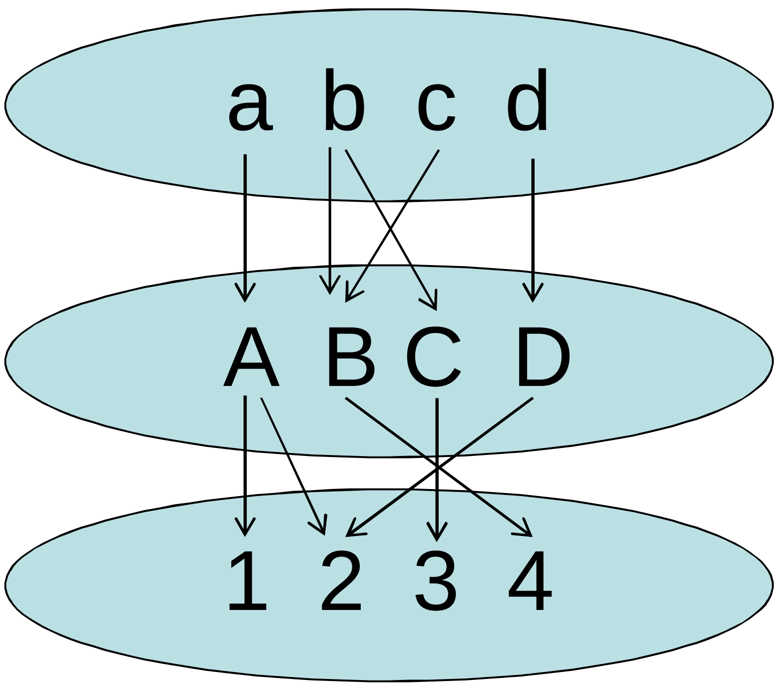
Relations and functions

- Definition of binary relation R on sets A and B:
  - a subset of $A \times B$
  - e.g.: less than relation
    - $A=B=N$, $R = \{(i, j) \in N^2 : i<j\} = \{(0, 1), (0, 2), (0, 3), (0, 4), ..., (1, 2), (1, 3), (1, 4), ... \}$
    - $(a, b) \in R \leftrightarrow a < b$



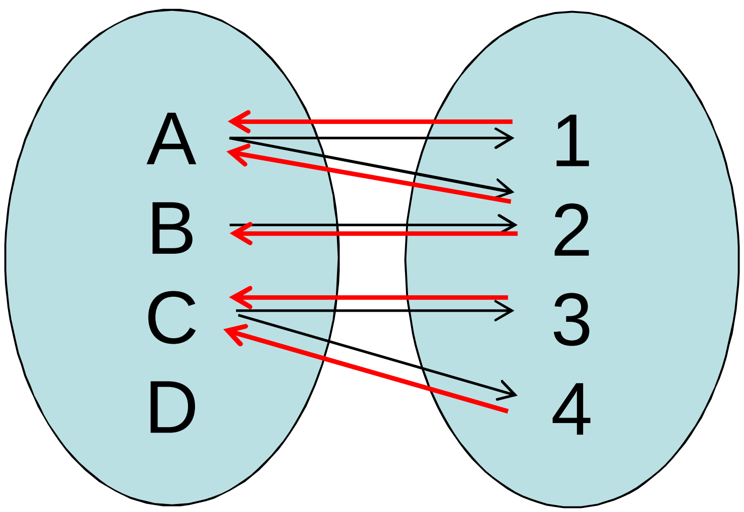
Relations and functions

- N-ary relation is a subset of $A_1 \times \ldots \times A_n$
  - R = {(a, A, 1), (a, A, 2), (b, B, 4), (b, C, 3), ...}



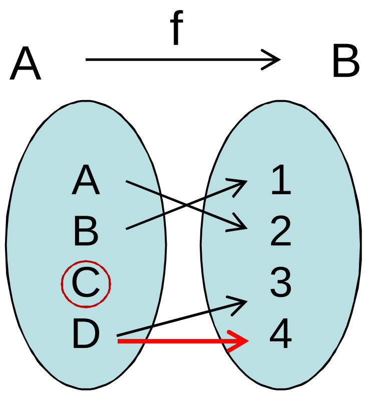
Relations and functions

- Definition of inverse of binary relation R^{-1} :
  - $R^{-1} = \{(b, a) : (a, b) \in R\}$
  - $R \subseteq A \times B$ binary relation
  - e.g.: $R^{-1} = \{(1, A), (2, A), (2, B), (3, C), (4, C)\}$



Relations and functions

- Definition of function, f: A $\rightarrow$ B: f $\subseteq$ A×B (f is a relation) where for $\forall$ a $\in$ A, $\exists$ exactly one pair in f with first component 'a'
  - (a, b) $\in$ f $\leftrightarrow$ f(a) = b
  - an association of each element of set A with an element of set B
    - A: domain of f
    - f(a) is the image of 'a' under f
    - range: the image of the domain



Relations and functions

- $R = \{(x, y) : x \in C, y \in S$, x is a city in state y$\}$
  - is a function, $C \to S$
    - R = {(Pest, HU), (Szeged, HU), (Austin, USA), ...}
    - R(Pest) = HU, R_1(Szeged) = HU, ...

Relations and functions

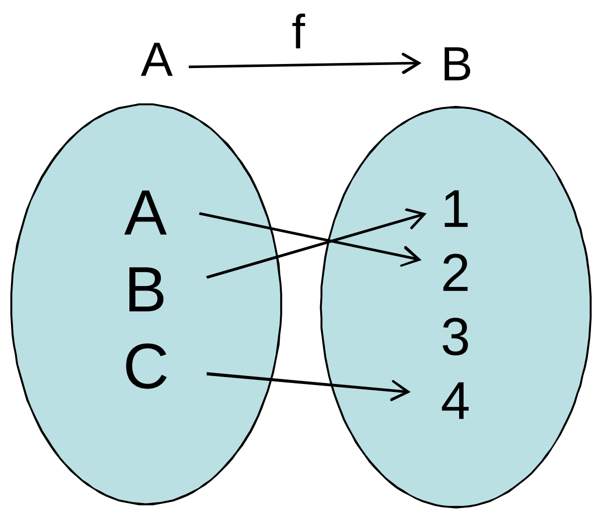
- $R^{-1} = \{(y, x) : x \in C, y \in S$, x is a city in state y$\}$
  - is not a function, $S \rightarrow C$
    - R^{-1} = {(HU, Pest), (HU, Szeged), (USA, Austin), ...}
  - but $F: S \rightarrow P(C)$ is a function
    - F(HU) = {Pest, Szeged, ...}, ...

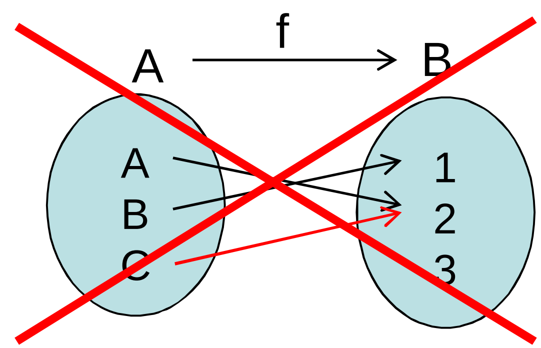
Relations and functions

- Function with multiple arguments: $f(a_1, \ldots, a_n) = b$
  - $a_1, \ldots, a_n$ are the arguments of f
  - b is the value of f
  - we can write $f((a_1, \ldots, a_n)) = b$
    - or define functions with multiple arguments
- The transition of a DFA is defined by a function (state, letter) → new state

Relations and functions

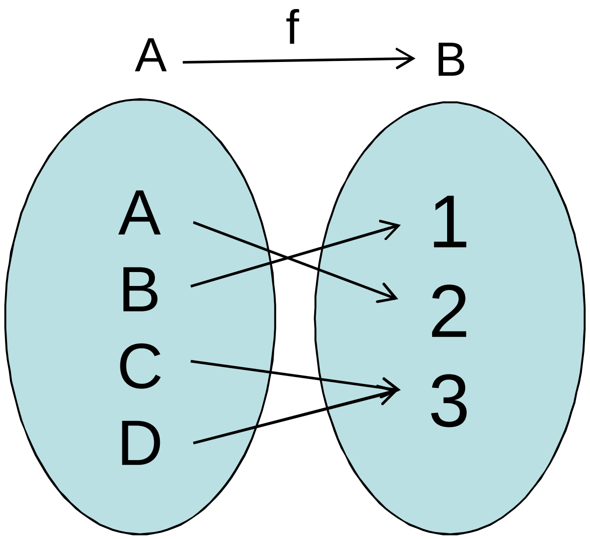
- Properties of f: A → B:
  - one-to-one or injective: if a ≠ a' → f(a) ≠ f(a')
    - every element of B is mapped to at most one element of A
    - e.g.: S = {states}, C = {cities}
      f: S → C; f(s) = capital of state s

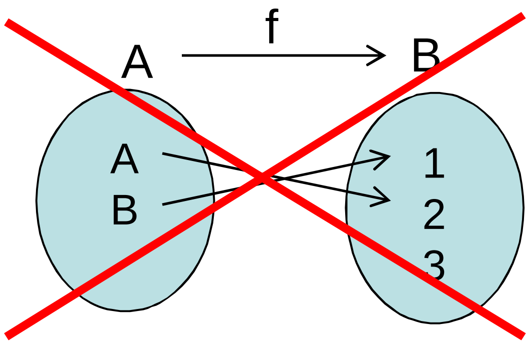




Relations and functions

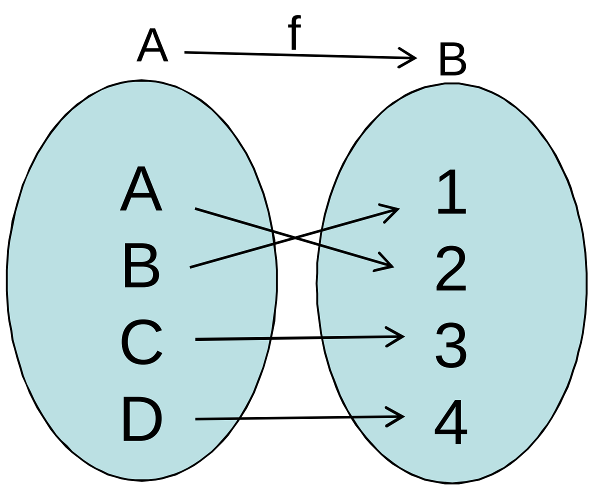
– onto or surjective function:
  - every element of B is mapped to at least one element of A
  - e.g.: C = {cities}, S = {states}
    f: C $\rightarrow$ S; f(c) = state of city c

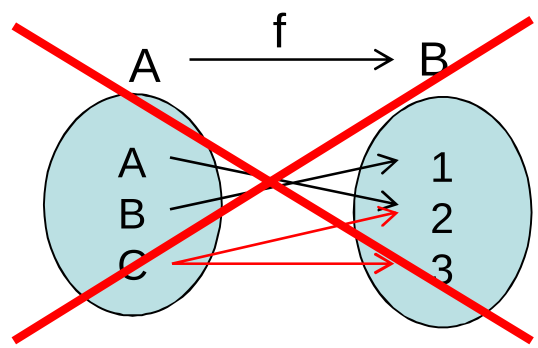




Relations and functions

– one-to-one correspondence or bijective function:
  - every element of B is mapped to exactly one element of A
    – one-to-one and onto function also
  - e.g.: S = {states}, C = {capital cities}
    f: S → C; f(s) = capital of state s



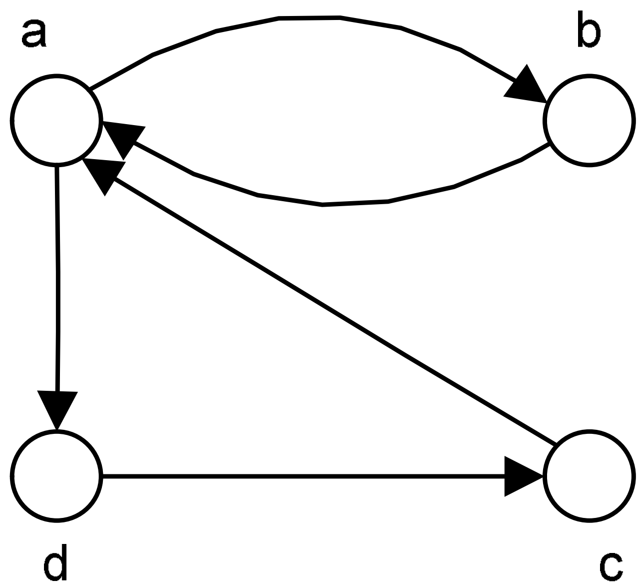


Relations and functions

- Questions: injective, surjective, or bijective
  - g: {vehicle type} → {car brand} surjective, injective
  - h: {people} → {fingerprints of people} bijective
  - i: {ID card} → {people} injective
  - j: {wives} → {husbands} bijective in ideal case

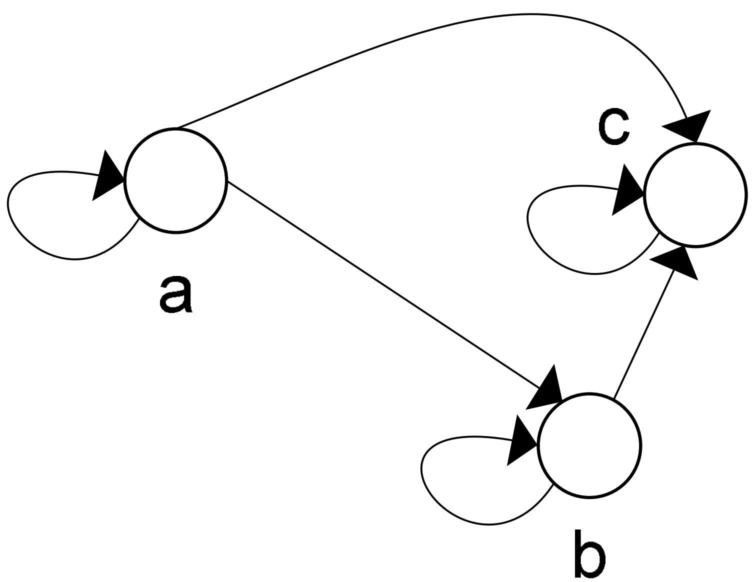
Special types of binary relations

- A binary relation R $\subseteq$ A×A can be represented in a directed graph
  - each elements of A are represented by a node
  - an arc is drawn from a to b if (a, b) $\in$ R
  - e.g.: R={(a, b), (a, d), (b, a), (c, a), (d, c)}



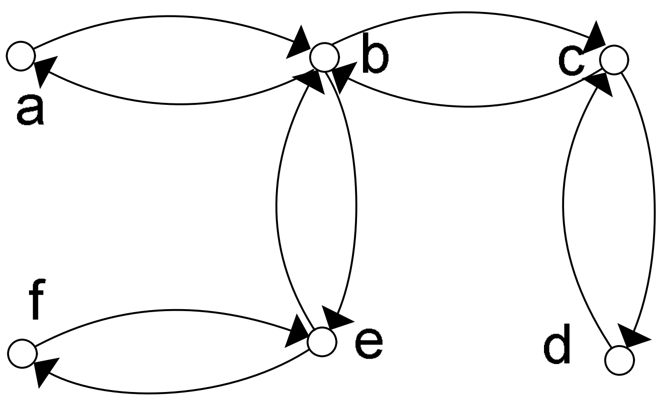
Special types of binary relations

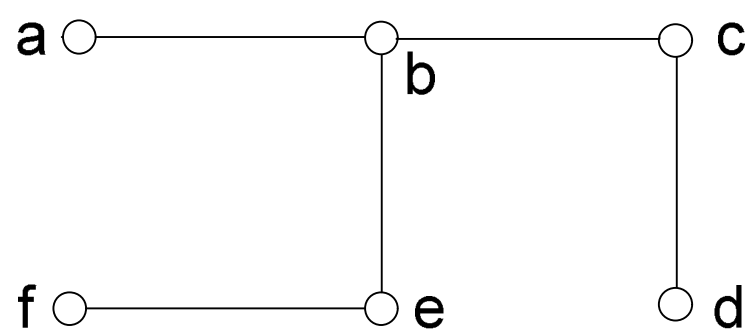
- Properties of binary relations $R \subseteq A \times A$:
  - reflexive: $(a, a) \in R$ for all $a \in A$
  - e.g.: $\{(a, b) : a \leq b\}$



Special types of binary relations

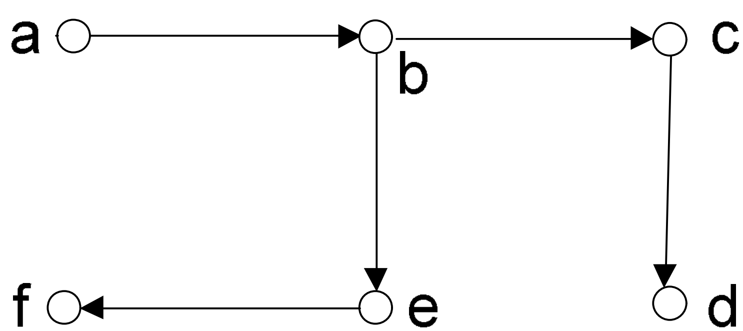
- Properties of binary relations $R \subseteq A \times A$:
  - symmetric: if $(a, b) \in R \rightarrow (b, a) \in R$
    - there are arcs in both directions between the nodes
    - a single undirected arc can be used
    - e.g.: {(a, b) : a is a friend of b}





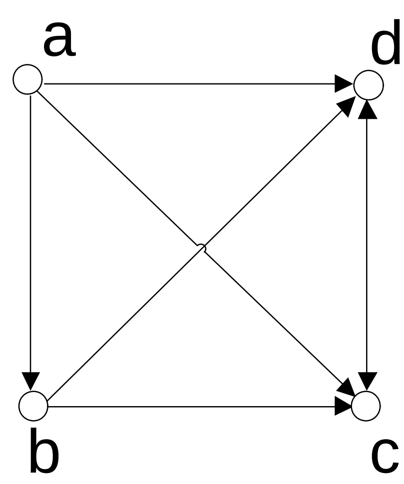
Special types of binary relations

- Properties of binary relations $R \subseteq A \times A$:
  - anti-symmetric: if $(a, b) \in R$, $a \neq b \rightarrow (b, a) \notin R$
    - e.g.: P = set of all persons, {(a, b) : a, b $\in$ P, 'a' is the father of b}



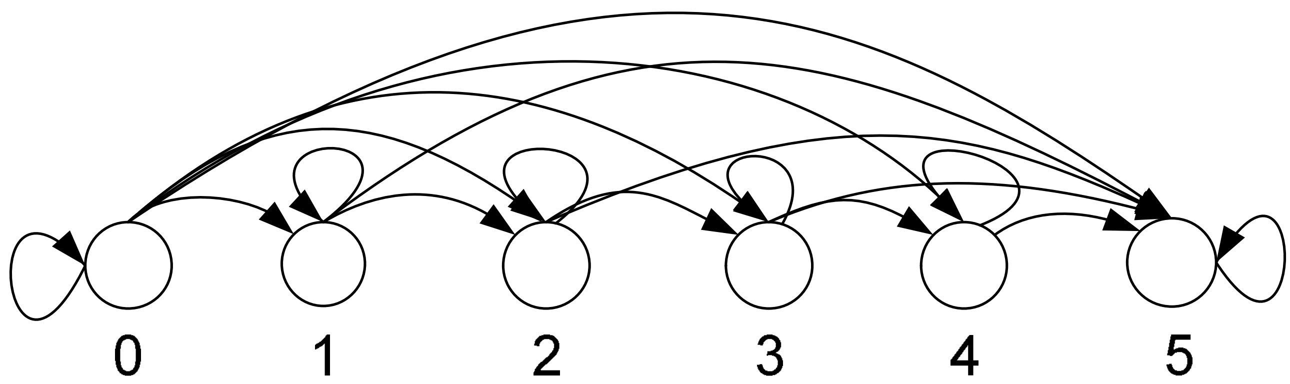
Special types of binary relations

- Properties of binary relations $R \subseteq A \times A$:
  - transitive: if (a, b), (b, c) $\in$ R $\rightarrow$ (a, c) $\in$ R
    - e.g.: {(a, b) : a, b $\in$ P, a is an ancestor of b}



Special types of binary relations

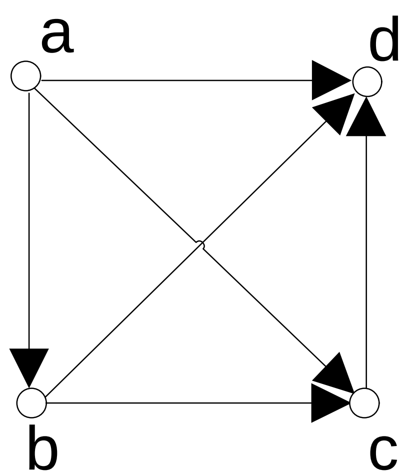
- Which properties are true?



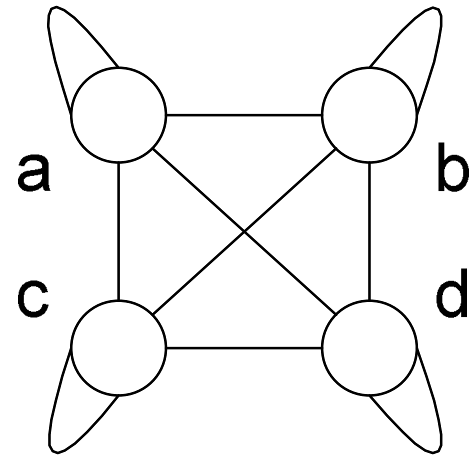
  - reflexive
  - anti-symmetric

Special types of binary relations

- Which properties are true?



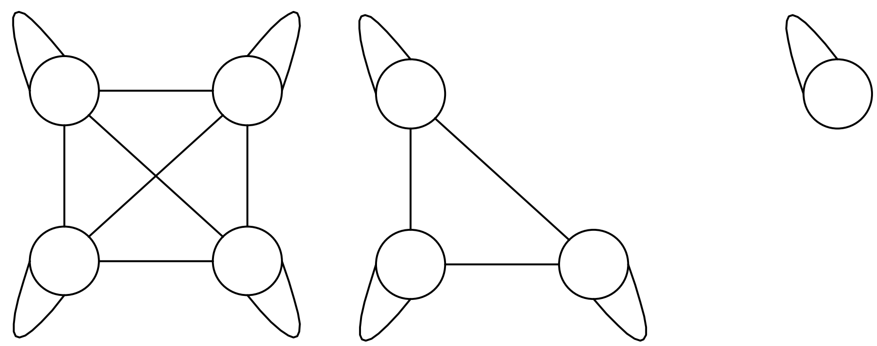
- anti-symmetric
- transitive



- reflexive
- symmetric
- transitive

Special types of binary relations

- Properties of binary relations $R \subseteq A \times A$:
  - equivalence relation: R is reflexive, symmetric, and transitive



Special types of binary relations

- R consists clusters
  - the clusters are not connected
  - within a cluster every node is connected
- the clusters are called equivalence classes
- e.g.: $\{(a, b) : a = b\}$, each class is a singleton
- this will be used at algorithm complexity ➡

Special types of binary relations

- Properties of binary relations $R \subseteq A \times A$:
  - partial order: R is reflexive, anti-symmetric, transitive
    - e.g.: {(a, b): a, b are persons, a is an ancestor of b}
  - total order: R is partial order and either $(a, b) \in R$ or $(b, a) \in R$
- Theorem: If R is an equivalence relation on a set A $\rightarrow$ the equivalence classes of R constitute a partition of A

Summary

- Introduction, Scope, Content
- Basic: Boolean algebra and notation
- Sets, Power sets, Descartes product
- Relations and functions
- Special type of binary relations

Next time

- Finite and infinite sets
- Three fundamental proof techniques
- Closures and algorithms

Elements of the Theory of Computation

Lesson 2

1.4. Finite and infinite sets

1.5. Three fundamental proof techniques

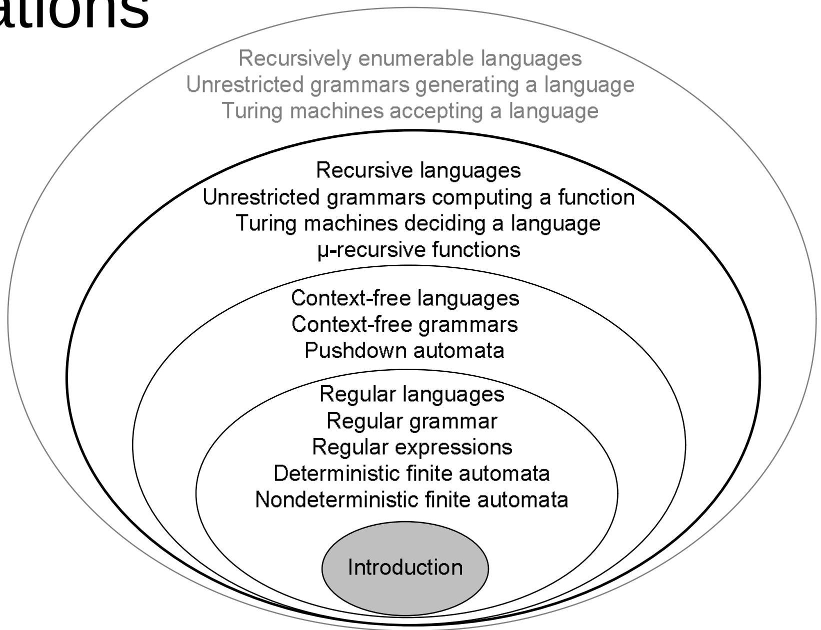
1.6. Closures and algorithms

University of Pannonia

Dr. István Heckl, Istvan.Heckl@gmail.com

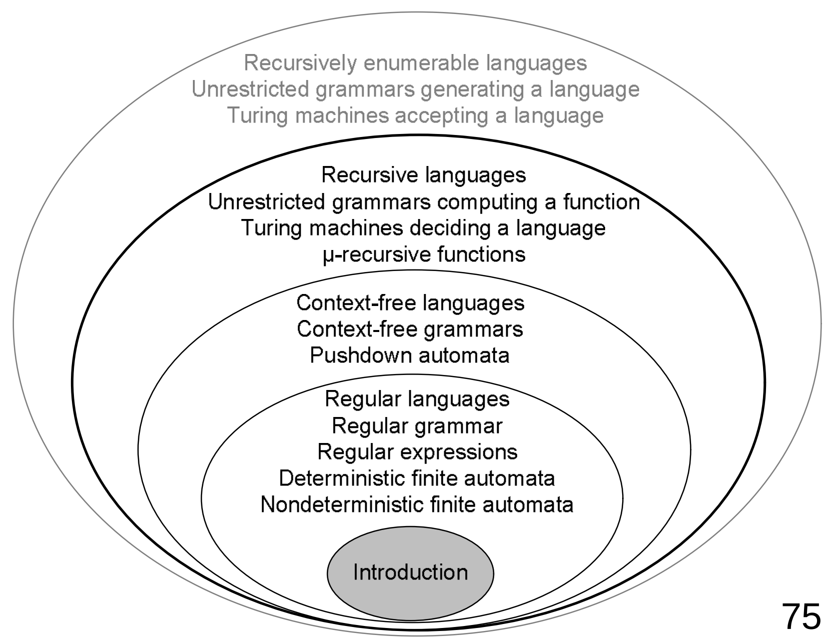
Last time

- Boolean algebra
- Sets
- Sets operations
- Relations and Functions
- Special types of binary relations



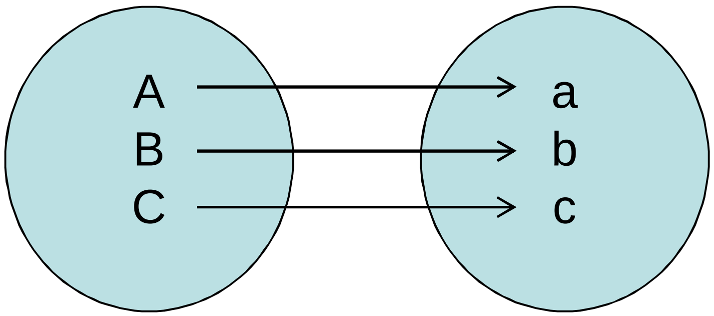
Introductions 2

- Finite and infinite sets
- Three fundamental proof techniques
  - Mathematical induction
  - The Pigeonhole principle
  - Diagonalization principle
- Algorithm complexity
- Reflexive, transitive closure



Finite and infinite sets

- The cardinality of set: the number of elements in it
  - this definition is problematic with infinite sets
- Definition of equinumerous: set A and B is called equinumerous if there is a bijection f: A → B
  - e.g.: A = {8, red, {Ø, b}}, B = {1, 2, 3}
    f(8) = 1; f(red) = 2; f({Ø, b}) = 3



Finite and infinite sets

- Definition of finite set: the set is equinumerous with $\{1, 2, \dots, n\}$, $n \in N$
  - A is a finite set, if $\exists$ bijection $f: A \rightarrow \{1, 2, \dots, n\}$
- Definition of infinite set: a set that is not finite
  - e.g.: N, R

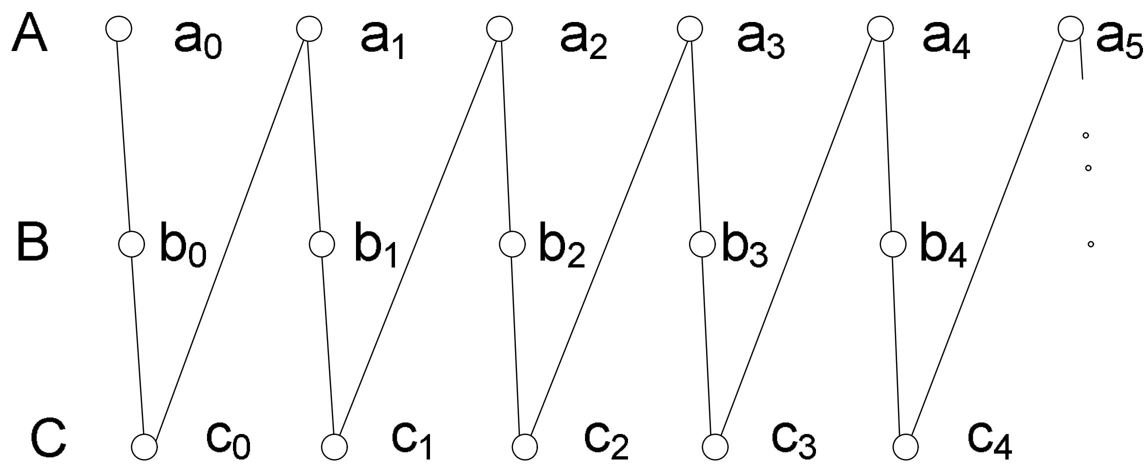
Finite and infinite sets

- Definition of countably infinite set: equinumerous with N
  - the set can be listed using ... only once
  - e.g.: Z

    - there is as much integer as much positive integer
- Definition of countable: finite or countably infinite
- Definition of uncountable: a set that is not countable
  - e.g.: R

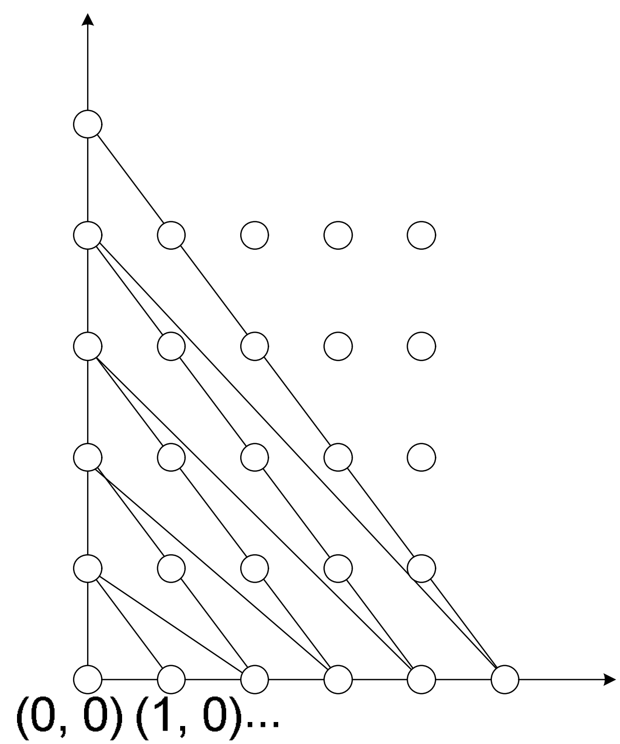
Finite and infinite sets

- Theorem: the union of finite number of set, each set is countably infinite, is also countably infinite
- Proof:
  - a bijection must be given
  - a clever listing of the elements of the sets is needed
  - e.g. for set A, B, and C



Finite and infinite sets

- Theorem: the Cartesian product of finite number of set, each set is countably infinite is also countably infinite
  - it is the union of countably infinite number of set, each set is countably infinite
- Proof:
  - a bijection must be given
  - a clever listing of the elements of the sets is needed



Finite and infinite sets

- Questions:
  - {number of divisor of a}
    $a \in N$ — countable - finite
  - {words} — countable - infinite
  - {points in the coordinate system} — uncountable - infinite

Three fundamental proof techniques

- Mathematical induction
- The Pigeonhole principle
- Diagonalization principle

Mathematical induction

- Idea:
  - if for set A the following are true:
    - $A \subseteq N$
    - $0 \in A$
    - $\forall n \in N$, if $n \in A \rightarrow n+1 \in A$
  - then $A = N$
- Intuitive proof: if the conditions are true for A: A can be increased one element at a time
  - {0}, {0,1}, {0,1, 2}, ...
  - the series converges to N
- NFA to DFA conversion uses it

Mathematical induction

- We would like to show that property P is true for $\forall$ n $\in$ N
  - basis step: we show that for 0 P is true
  - induction hypothesis:
    - for some n P is true
  - induction step:
    - we prove that P is true for n+1 if P is true for n

Example

- Theorem: $n \geq 0, 1 + 2 + \ldots + n = \dfrac{n^2 + n}{2}$
- Proof:
  - basis step: n = 0
    - the sum on the left is zero, there is nothing to add
    - the expression on the right is also zero
  - induction hypothesis:

$$n \geq 0, 1 + 2 + \ldots + n = \frac{n^2 + n}{2}$$

Example

- Proof:
  - induction step:

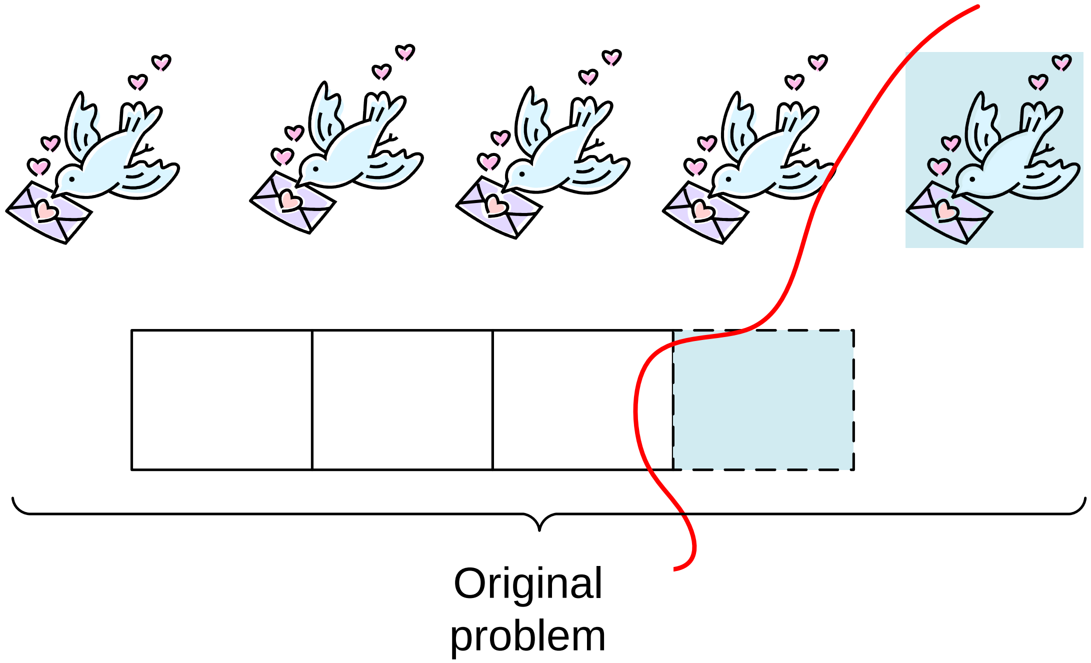
The pigeonhole principle

- Theorem: the Pigeonhole principle: if A and B are finite sets and $|A| > |B| \rightarrow$ there is no one-to-one function $f: A \rightarrow B$
  - there will be a pigeon without pigeonhole
- Proof by induction for |B|:
  - basis step: $n = 0 \rightarrow B = \emptyset$, f (with any property) does not exist
  - induction hypothesis: for $|B| = n$ there is no one-to-one f
    - $f: A \rightarrow B$, $|A| > |B|$, $|B| = n$, $n \geq 0$

The pigeonhole principle

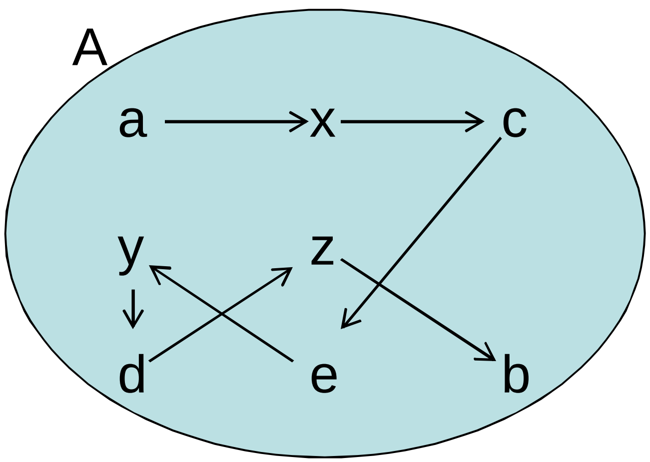
– induction step: |B| = n+1, proof by indirection
  - suppose ∃ one-to-one f: A → B, |A| > |B|
  - choose some a ∈ A
  - if ∃ a' ∈ A, f(a) = f(a') → f is not one-to-one → contradiction
  - else construct g: A-{a} → B-{f(a)} such that f=g except at 'a'
  - the induction hypothesis is true for g, so g does not exist, consequently, neither does f → contradiction

The pigeonhole principle



The pigeonhole principle

- Theorem:
  - R is a binary relation on finite set A, a, b $\in$ A
  - if there is a path from 'a' to b in R $\rightarrow$ $\exists$ such a path whose length is at most |A|

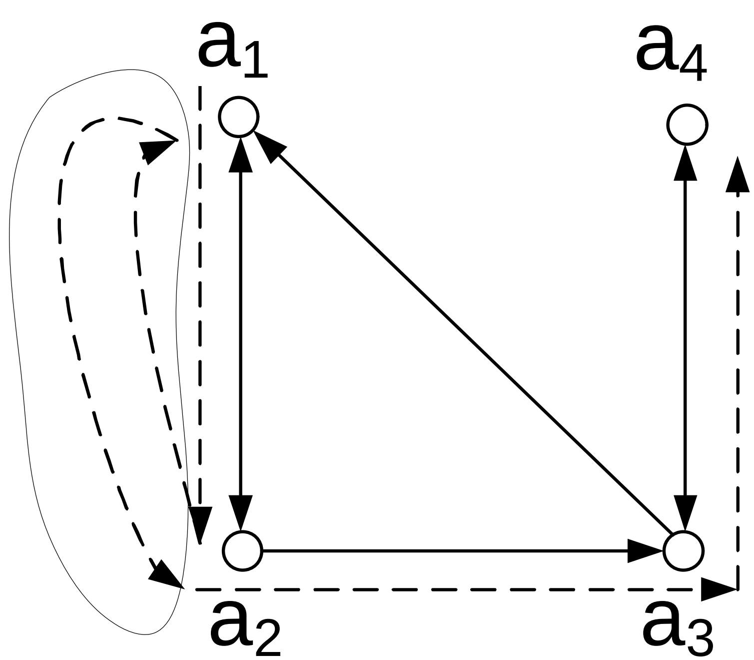


The pigeonhole principle

- Proof by indirection:
  - suppose the shortest path from 'a' to b is ($a=a_1$, a_2, ..., $a_n=b$) and $n>|A|$
  - function f: $\{1, 2, ... n\} \rightarrow A$, f is no one-to-one according to the pigeonhole principle
    - e.g.: $f = \{(1, a_1), (2, a_2), ...\}$
  - if f is no one-to-one $\rightarrow \exists\ a_i = a_j$ ($i<j$)
  - (a_1, a_2, ..., a_i, a_{j+1}, ... a_n) is a shorter path than the original (omit the nodes between a_i, a_j), contradiction is reached

The pigeonhole principle

- If there is a path between 'a' to b → in the worst case you travel through each node once
  - if you travel through a node (a_2) twice then you can shorten the path by cutting the loop



Diagonalization principle

- Theorem, Diagonalization principle:
  - if
    - R is binary relation on set A
    - D is diagonal set for R
      - $D = \{a : a \in A, (a, a) \notin R\}$
    - for each $a \in A$, $R_a = \{b : b \in A, (a, b) \in R\}$
  - then D is distinct from each R_a
- Proof: R_c differs from D in terms of c
  - if $(c, c) \in R \rightarrow c \notin D, c \in R_c$
  - if $(c, c) \notin R \rightarrow c \in D, c \notin R_c$
  - c is selected arbitrary

Diagonalization principle

- Halting problem uses the diagonalization principle ➡
- Visualization:
  - if A is finite, R is pictured as an array
  - rows and columns are labeled with the elements of A
  - if (x, y) $\in$ R $\rightarrow$ square (x, y) is checked in the array
  - D: complementary of the diagonal of the array:
  - R_a: corresponds to the row 'a' of the array

Diagonalization principle

- Diagonalization principle in other words: the complement of the diagonal is different from each row
- The Diagonalization principle also holds for infinite sets
- D, R_a are sets but an ordering can be introduced based on the next figure

Diagonalization principle

R relation

	a	b	c	d	e	f
a		x		x		
b		x	x			
c			x			
d		x	x		x	x
e					x	x
f	x		x	x	x	

(a, a) $\in$ R

	x	x		x	

(a, a) $\notin$ R

x			x		x

Diagonalization principle

- Questions:

	a	b	c	d	e	f
a		x		x		
b		x	x			
c			x			
d		x	x		x	x
e					x	x
f	x		x	x	x	

- R = {(a, b), (a, d), (b, b), (b, c), (c, c), (d, b), (d, c), (d, e), (d, f), (e, e), (e, f), (f, a), (f, c), (f, d), (f, e)}

- R_a = {b, d}
- R_b = {b, c}
- R_c = {c}
- R_d = {b, c, e, f}
- R_e = {e, f}
- R_f = {a, c, d, e}

Diagonalization principle

- Theorem: P(N) is uncountable
- Proof by indirection:
  - suppose P(N) is countably infinite
    - there is a way to enumerate all the subsets of N
      $P(N) = \{R_1, R_2, ...\}$
    - e.g.: $R_1 = \{1\}$, $R_2 = \{1, 2\}$, $R_3 = \{2, 3\}$, $R_4 = \{1, 2, 3\}$, $R_5 = \{3, 4\}$, $R_6 = \{2, 3, 4\}$, ...
  - build relation R
    - R_1 should be the 1st row, R_2 the 2nd, and so on

Diagonalization principle

R relation

	1	2	3	4	5
R_1	x				
R_2	x	x			
R_3		x	x		
R_4	x	x	x		
R_5			x	x	
R_6		x	x	x	

Diagonalization principle

– let $D = \{n : (n, n) \notin R\}$
  - e.g.: $D = \{1, 6, ...\}$
– D is a set of natural numbers
  - according to its definition
– D is not a set of natural numbers
  - according to the diagonalization principle there is no $i \in N$ such that $D = R_i$
  - R_i's are all the possible subsets of N

Algorithm complexity

- Definition of the complexity of an algorithm: f(n) is an upper bound on the number of elementary steps required for the algorithm if the size of the input is n
  - average number cannot be used because it requires a known distribution for the inputs

Algorithm complexity

- Definition of order of f, $O(f)$:
  - let $f: N \rightarrow N$
  - $O(f)$ is a set of such functions which increase at most as fast as f disregarding some constants (informal)
  - for $\forall g \in O(f)$, $g: N \rightarrow N$
    - $\exists c \geq 0, d \geq 0$ constants such that for $\forall n \in N$, $g(n) \leq c \cdot f(n)+d$
    - e.g.: $O(n^3) = \{n, n+1, n+2, \ldots, 2n, 2n+1, \ldots, n^2, \ldots, n^3, \ldots\}$

Algorithm complexity

- Definition of relation $f \approx g$: f, g: $N \rightarrow N$, $f \in O(g)$, $g \in O(f)$
  - $\approx$ is an equivalence relation of the $N \rightarrow N$ functions
    - reflexive: $f \in O(f)$, with constants 1 and 0
    - symmetric: the roles of f and g are interchangeable
    - transitive
- The $N \rightarrow N$ functions are partitioned by $\approx$ into equivalence classes
- Definition of rate of growth of f: the equivalence class of f with respect to the $\approx$ relation

Example

- $f(n) = 31n^2 + 17n + 3$
  - is it true that $f(n) \in O(n^2)$? (i.e., $f(n) \le cn^2+d$)
    - notice $n^2 \ge n$; $f(n) \le 31n^2 + 17n^2 + 3 = 48n^2 + 3$
    - c=48, d=3
  - is it true that $n^2 \in O(f)$?
    - yes, with c=1, d=0
  - hence $n^2 \approx 31n^2 + 17n + 3$, so the two functions have the same rate of growth

Example

- Let $f(n) = 10n^2 + 5n+7$
  - is it true: $f(n) \in O(n^2)$
  - $f(n) = 10n^2 + 5n+7 < 10n^2 + 5n^2+7 = 15n^2 + 7$
  - $f(n) \leq c * n^2 + d$
    - $c = 15, d = 7$

Example

- $f(n) = a_d n^d + a_{d-1} n^{d-1} + \ldots + a_1 n + a_0$, $a_i \geq 0$ for $\forall$ i, $a_d > 0$
  - $f(n) \in O(n^d)$
  - all polynomials of the same degree have the same rate of growth

Algorithm complexity

- Lemma: for $\forall\, n \in N$, $n \leq 2^n$
- Proof:
  - basis step: $0 \leq 2^0 = 1$
  - induction hypothesis: suppose that $n \leq 2^n$
  - induction step: $n+1 \leq 2^n+1 \leq 2^n + 2^n = 2^{n+1}$
    - add 1 to both sides
    - replace 1 with 2^n on the right side, $1 \leq 2^n$

Algorithm complexity

- Theorem: for $\forall$ i $\in$ N, $n^i \in O(2^n)$
- Proof:
  - $n^i \leq c2^n + d$
    - $c=(2i)^i$, $d=(i^2)^i$
  - if $n \leq i^2$
    - $n^i \leq (i^2)^i$, use the power function
      - $(i^2)^i = d$, see definition of d
    - $n^i \leq d$
    - $n^i \leq c2^n + d$, the added term is positive
  - for small n the d in $c2^n + d$ makes sure that n^i is smaller

Algorithm complexity

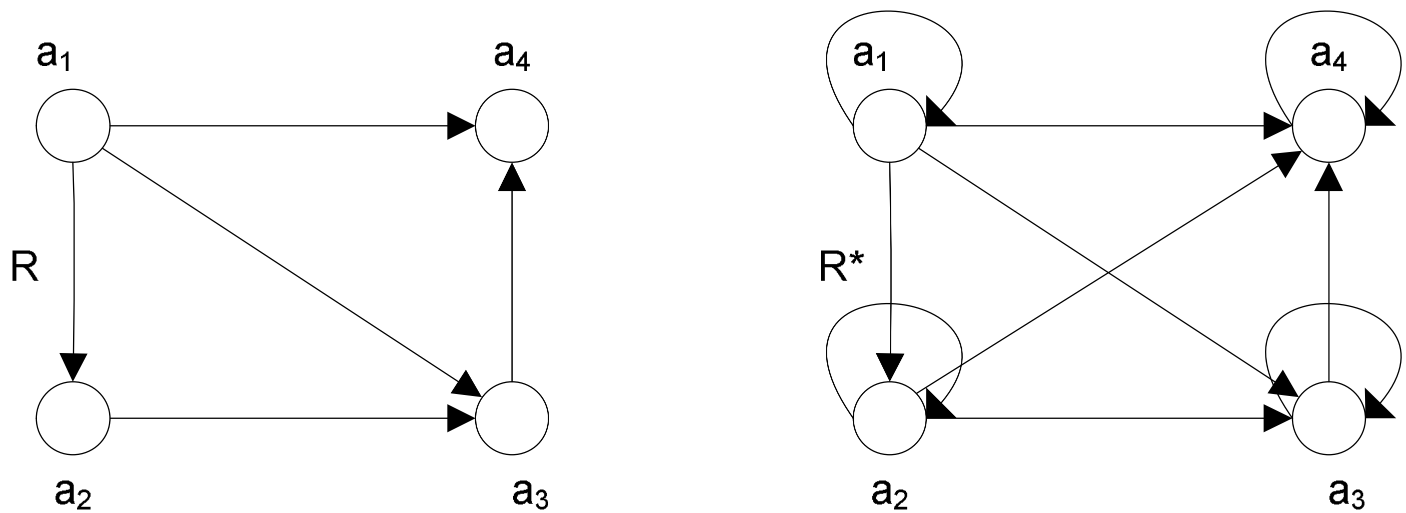
- if $n \geq i^2$
  - let $m = \lfloor n / i \rfloor \rightarrow m*i \leq n < (m+1)*i$
  - $n < i \cdot (m+1)$
  - $n^i \leq i^i \cdot (m+1)^i \leq i^i \cdot (2^{m+1})^i$ (because of the lemma)
  - $n^i \leq i^i \cdot (2^{m+1})^i = i^i \cdot (2 \cdot 2^m)^i = (2 \cdot i \cdot 2^m)^i = (2i)^i \cdot 2^{mi} = c \cdot 2^{mi} \leq c2^n \leq c2^n + d$
- for large n the c makes sure that n^i is smaller
- the rate of growth of any polynomial is no faster than 2^n

Algorithm complexity

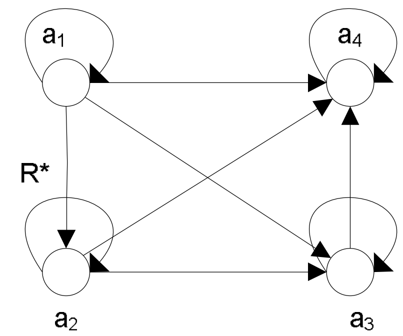
- Theorem: for $\forall$ i $\in$ N, $2^n \notin O(n^i)$
  - 2^n does grow faster than n^i
- Proof by indirection:
  - suppose $2^n \in O(n^i)$ for $\forall$ i $\in$ N
  - $n^i \in O(2^n)$, see the previous theorem
  - $2^n \in O(n^i)$, $n^i \in O(2^n) \rightarrow n^i \approx 2^n$
  - select i1 ≠ i2
  - $n^{i1} \approx 2^n$, $n^{i2} \approx 2^n \rightarrow n^{i1} \approx n^{i2}$
    - transitive property of $\approx$
  - $n^{i1} \approx n^{i2}$ is not true, contradiction

Reflexive, transitive closure

- Definition of "reflexive, transitive closure of R" = R*:
  - let $R \subseteq A^2$ be represented by a directed graph defined on a set A
  - R* is the smallest relation that contains R and is reflexive and transitive



Reflexive, transitive closure



- Algorithm 1 for determining R*:

```
R* := 0
for i=1, …, n do
    for each i-tuple (b1, …, bi) ∈ A^i do
      if (b1, …, bi) is a path in R →
        add (b1, bi) to R*
```

- Informal definition of algorithm:
  – sequence of instructions that produces a result
  – halt after a finite number of steps

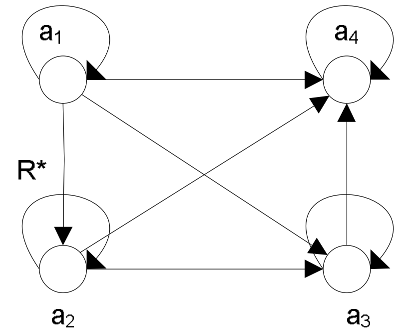
Reflexive, transitive closure

- The operation of the algorithm:
  - initially R* is empty
  - all paths of R (with all the possible length) are considered
  - for each path a direct connection is added to R*

Reflexive, transitive closure

- Complexity of the algorithm 1:
  - the input size is IAI = n
  - number of i-tuples if IAI = n: n^i
    - e.g.: IAI = 10, number of 5-tuples: 10^5
  - number of steps to check if an i-tuple is a path: n
  - $f(n) = n*(1+n+n^2+\ldots+n^n)$
  - $f \in O(n^{n+1})$
    - n^{n+1} has even higher rate of growth than 2^n
    - this algorithm is not efficient

Reflexive, transitive closure



- Algorithm 2 for determining R*:

  R* := R ∪ {(a_i, a_i) : a_i ∈ A}

  for all (a_i, a_j, a_k) ∈ A^3

    if (a_i, a_j),(a_j, a_k) ∈ R*, (a_i, a_k) ∉ R* → add (a_i, a_k) to R*, restart

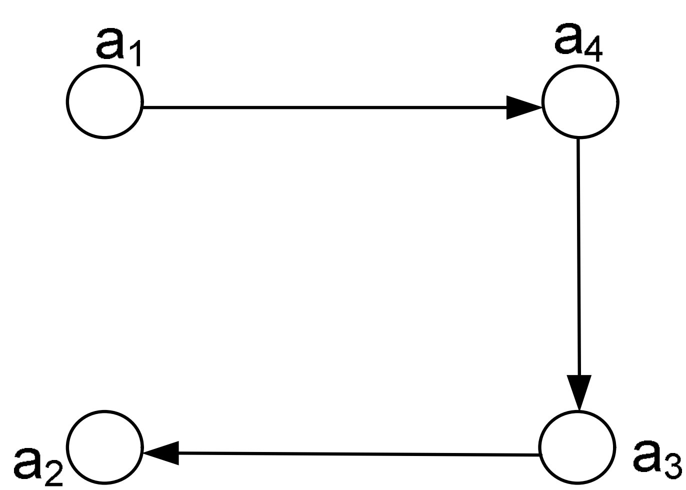
- R* will certainly contain R, and it will be reflexive

Reflexive, transitive closure

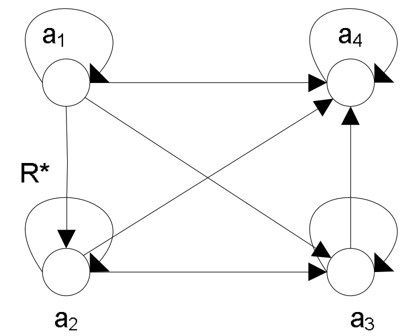
- Complexity of the algorithm 2:
  - the add statement is executed at most n^2 times
  - after each addition the search for a suitable triplet must be restarted, there are n^3 triplets
  - $f(n) = n^5$

Example

- Visiting order of the triplets: (a_1, a_1, a_1), (a_1, a_1, a_2), ..., (a_1, a_1, a_4), (a_1, a_2, a_1), (a_1, a_2, a_2), ..., (a_4, a_4, a_4)
- First violation at (a_1, a_4, a_3)
  - new edge: (a_1, a_3)
- If the search is not restarted → the next violation at (a_1, a_3, a_2) is missed



Reflexive, transitive closure



- Algorithm 3 for determining R*:

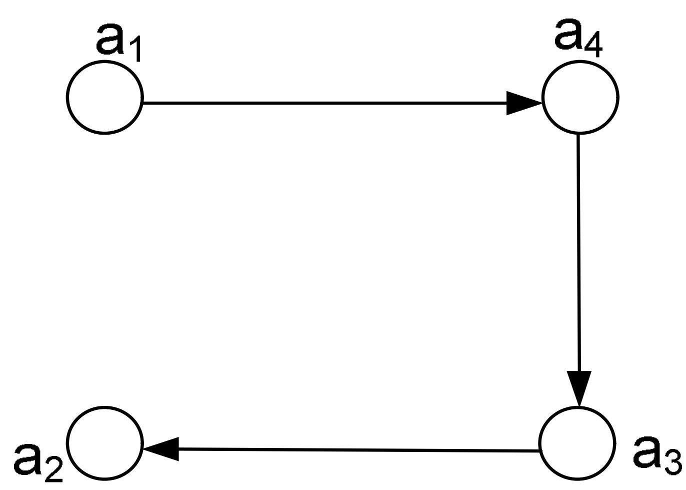
```
R* := R ∪ {(a_i, a_i) : a_i ∈A}
for j=1, 2, …, n do
   for i=1, 2, …, n, k=1, 2, …, n do
    if (a_i, a_j), (a_j, a_k) ∈ R* →
       add (a_i, a_k) to R*
```

Reflexive, transitive closure

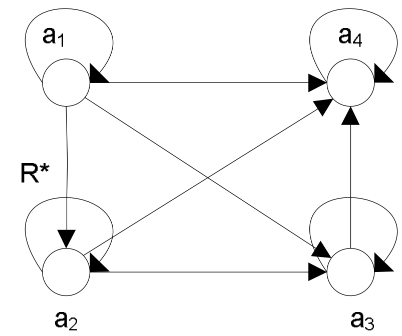
- Algorithm 3 for determining R*:
  - this is a modification of algorithm 2
  - it searches the triplets in such an order that the newly added arcs do not introduce such violation which cannot be rectified later
    - restart is not needed
  - $f(n) = n^3$

Example

- Visiting order of the triplets: (a_1, a_1, a_1),..., (a_1, a_1, a_4), (a_2, a_1, a_1), ..., (a_2, a_1, a_4),..., (a_1, a_2, a_1), ..., (a_4, a_4, a_4)
- First violation: (a_4, a_3, a_2)
  - new arc: (a_4, a_2)
- The new violation, (a_1, a_4, a_2), will be dealt with later
- Last violation: (a_1, a_4, a_3)



Reflexive, transitive closure



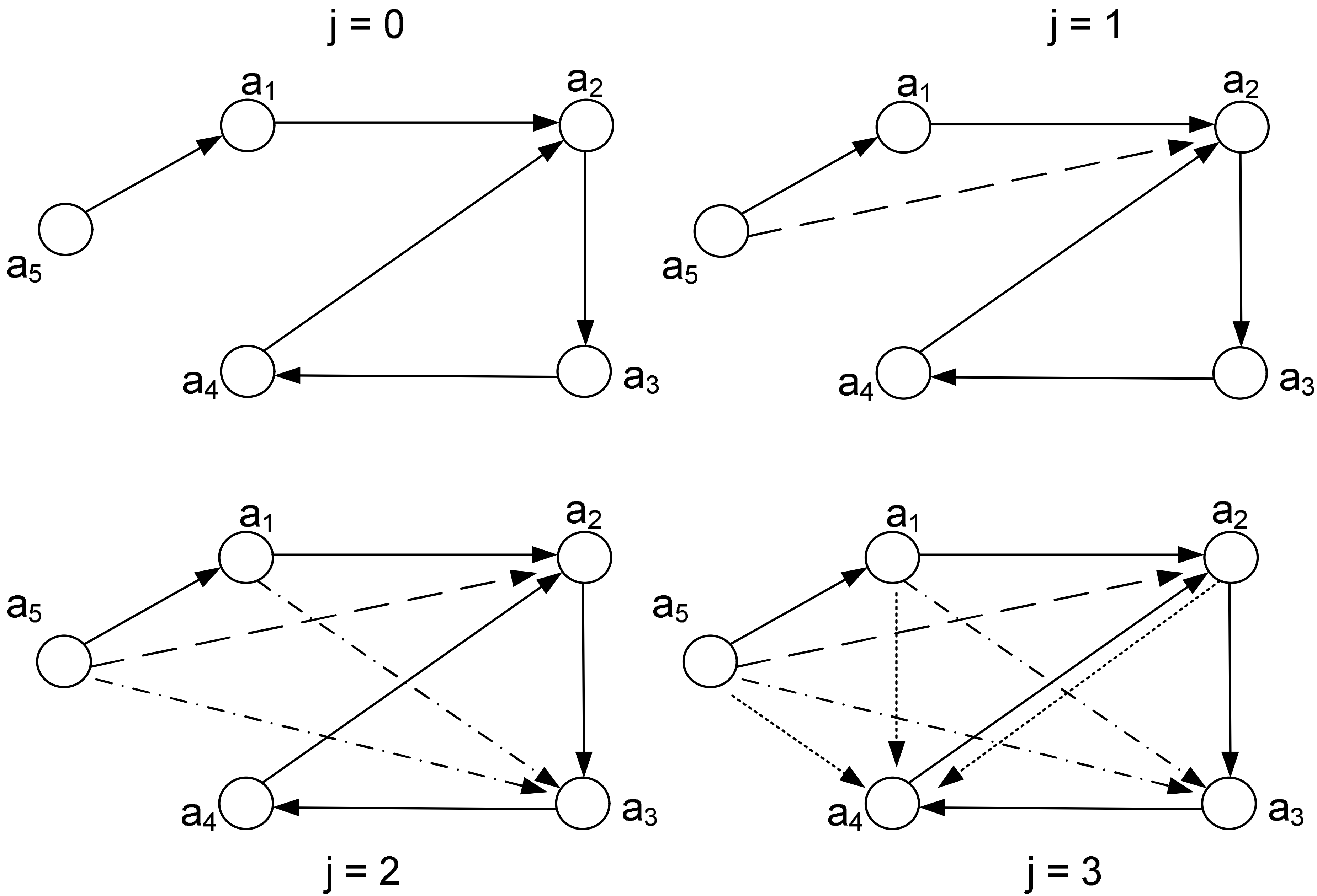
- Definition of the rank of a path (a_{i0}, a_{i1}, ..., a_{ik}): the largest integer among i_1, ..., i_{k-1} (the indexes of the inner nodes)
  - trivial path: a single arc, rank = 0, no inner node
- Theorem: the j^{th} iteration adds those pairs to R* that are connected in R by paths of rank j
  - in other words: after the j^{th} iteration, R* contains all pairs (a_i, a_k) which are joined by a path of rank j or less in R (we prove this)
  - if j = n → the statement is: after the n^{th} iteration (at the end) R* contains all pairs which are joined by a path of rank n or less (any path) in R
    - it is the definition of R*

Reflexive, transitive closure

- Proof by induction on j:
  - basis step: j=0
    - trivial paths of R are the arcs of R
    - the arcs of R is already in R*
  - induction step:
    - select any nodes a_i, a_k which are connected by a path of rank j+1
      - they are also connected by such a path in which a_{j+1} appears exactly once
      - if a_{j+1} appears more than once $\rightarrow$ delete the portion of the path between the first and last occurrences

Reflexive, transitive closure

- paths $(a_i, ..., a_{j+1})$ and $(a_{j+1}, ..., a_k)$ have rank j or less
  - the algorithm regard triplets and not paths
- $(a_i, a_{j+1}), (a_{j+1}, a_k) \in R^*$ according to the induction hypothesis
  - these arcs are added in a previous iteration
- we add (a_i, a_k) to R* according to the algorithm, so now R* contains all pairs (a_i, a_k) which are joined by a path of rank j+1 or less


j = 0
a_1
a_2
a_5
a_4
a_3
j = 1
a_1
a_2
a_5
a_4
a_3
a_1
a_2
a_5
a_4
a_3
j = 2
a_1
a_2
a_5
a_4
a_3
j = 3

Set closed under a relation

- Definition of set A is closed under relation R:
  - let
    - D set
    - $n \geq 0$
    - $A \subseteq D$
    - $R \subseteq D^{n+1}$ an (n+1)-ary relation
      - R is called the closure property of set A, R does not go out from A
  - if $\forall\ (b_1, \ldots, b_{n+1}) \in R,\ b_1, \ldots, b_n \in A, \rightarrow b_{n+1} \in A$
- If R is a function then $R(b_1, \ldots, b_n) = b_{n+1}$
- The result is in the same set as the parameters

Example

- Natural numbers are closed under addition
  - D=Z, A=N, n=3, R={..., (0, -1, -1), (0, 0, 0), (0, 1, 1), ... , (3, 4, 7), ...}
  - the sum of two natural number is also a natural number
- Natural numbers are not closed under subtraction
  - D=Z, A=N, n=3, R={..., (0, -1, 1), (0, 0, 0), (0, 1, -1), ... , (3, 4, -1), ...}
  - the difference of two natural numbers is not always a natural number

Closures

- Theorem: Let $A \subseteq D$, $R \subseteq D^{n+1}$ an (n+1)-ary relation, there is a unique minimal (in terms of cardinality) set A* such that $A \subseteq A^*$, and A* is closed under R
  - A* is called the closure of A under R
  - "set A is closed under R" is a property of set A
  - "R closure of set A" is a set operation of A
    - A can be any set, e.g., a relatic

Closures

- There are several possible closures, and there are polynomial algorithms for computing all of these closures
  – conversely any polynomial algorithm can be interpreted as the computation of the closure of a set under some relation

Closures

- R is a relation, $R \subseteq D^{r+1}$, $A \subseteq D$
- Computation of A* under R

```
A* := A
while ∃ elements a_j1, ..., a_jr ∈ A*,
    a_jr+1 ∉ A*, and (a_j1, ..., a_jr, a_jr+1)∈ R
          add a_jr+1 to A*
```

$A^* := A$

while $\exists$ elements $a_{j1}, \ldots, a_{jr} \in A^*$, $a_{jr+1} \notin A^*$, and $(a_{j1}, \ldots, a_{jr}, a_{jr+1}) \in R$

add a_{jr+1} to A^*

Closures

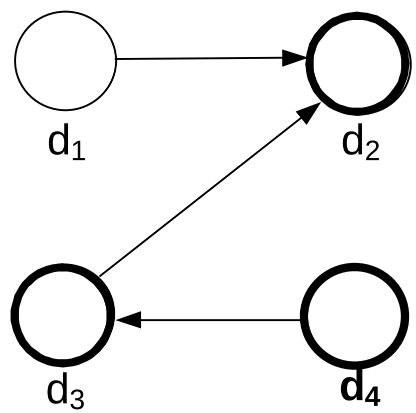
- Transitivity:
  - let $A \subseteq D \times D$
    - A can be seen as a relation and as a set (of arcs)
  - $R = \{((a, b), (b, c), (a, c)) : a, b, c \in D\}$
    - all possible transitive triplets
    - $R \subseteq (D \times D)^3$, ternary relation
  - A is closed under R $\leftrightarrow$ A is transitive
  - A* is completed by adding the 3^d component of R which is calculated from the first and second ones

Closures

- Reflexivity:
  - let $A \subseteq D \times D$
    - A can be seen as a relation and as a set (of arcs)
  - $R = \{((a, a)) : a \in D\}$
    - all possible loop
    - $R \subseteq (D \times D)$, unary relation: relates nothing with (a, a)
  - A is closed under $R \leftrightarrow$ A is reflexive
  - A* is completed by adding the first component of R which is calculated from nothing

Examples

- A binary relation is given on D × D, give the closure of set A on this relation!
  - $D = \{d_1, d_2, d_3, d_4\}$
  - $A = \{d_4\}$
  - $(d_4, d_3) \in R, d_4 \in A^* \rightarrow d_3 \in A^*$
  - $(d_3, d_2) \in R, d_3 \in A^* \rightarrow d_2 \in A^*$
  - $A^* = \{d_2, d_3, d_4\}$



Summary

- Finite and infinite sets
- Mathematical induction
- The Pigeonhole principle
- Diagonalization principle
- Algorithm complexity
- Reflexive, transitive closure

Next time

- Alphabets and languages
- Finite representations of languages

Elements of the Theory of Computation

Lesson 3

1.7. Alphabets and languages

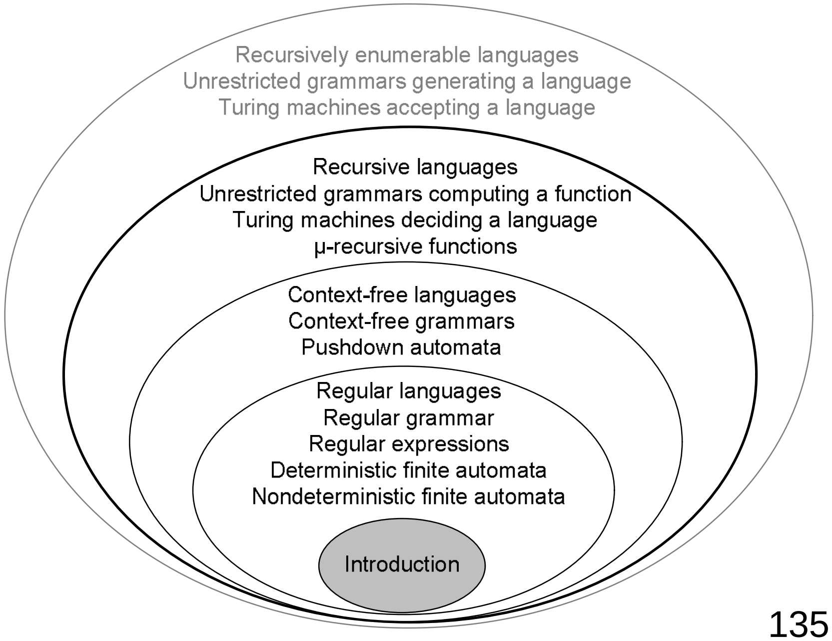
1.8. Finite representations of languages

University of Pannonia

Dr. István Heckl, Istvan.Heckl@gmail.com

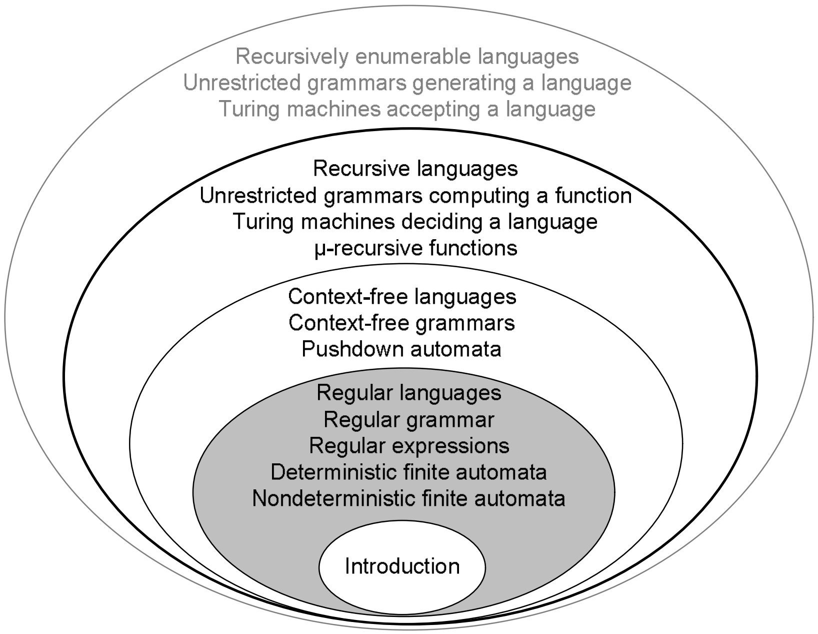
Last time

- Finite and infinite sets
- Mathematical induction
- The Pigeonhole principle
- Diagonalization principle
- Algorithm complexity
- Reflexive, transitive closure



Alphabets and languages

- Alphabets, strings, and languages
- Finite representation of language
- Regular expressions
- Properties of RE
- Regular languages



Alphabets, strings, and languages

- Definition of alphabet, Σ: finite set of symbols
- E.g.:
  - the Roman alphabet: {a, b, c,…, z}
  - the binary alphabet: {0, 1}
  - unary alphabet: {I}
- Definition of string: finite sequence of symbols from the alphabet

Alphabets, strings, and languages

- Examples for strings:
  - watermelon, water, chain are strings over the alphabet {a, b, c, …, z}
  - 0111011, 1, 100, 0 are strings over the alphabet {0, 1}

Alphabets, strings, and languages

- Definition of empty string, *e*: string containing 0 symbol
  - do not confuse with symbol e
  - {*e*} ≠ Ø
- Definition of Σ*: the set of all strings over alphabet Σ
  - in other words: all such string which can be created by using elements of Σ
  - contains the *e*
  - it is called sigma star
  - w ∈ Σ*, means any string from that alphabet

Alphabets, strings, and languages

- Definition of length of string, |w|: the number of letters in a string
- E.g.:
  - |apple| = 5
  - |101| = 3
  - |*e*| = 0
  - |szpsz| = 5 in English
  - |szpsz| = 3 in Hungarian
- w(j) is the jth letter in string w
  - e.g.: w=fun, w(1)=f, w(2)=u, w(3)=n

Alphabets, strings, and languages

- Definition of concatenation of two strings, $x \circ y$: string operation resulting in a new string
  - also denoted by: xy
  - if w = xy
  - then
    - $|w| = |x|+|y|$
    - $w(j) = x(j), j=1, \dots |x|$
    - $w(|x|+j) = y(j), j=1, \dots |y|$

Alphabets, strings, and languages

- Examples for concatenation:
  - beach∘boy = beachboy
  - 01∘001 = 01001
  - $w \circ e = e \circ w = w, \forall\ w \in \Sigma^*$
- Concatenation is associative: (wx)y = w(xy)
  - but not commutative

Alphabets, strings, and languages

- Definition of substring: v is a substring of w $\leftrightarrow$ $\exists$ x, y such that w = xvy
  - w, x, v, y $\in \Sigma^*$
  - both x and y could be *e*, so every string is a substring of itself
  - if w = vy, v is the prefix of w
  - if w = xv, v is the suffix of w

Alphabets, strings, and languages

- Definition of w^i: $w \in \Sigma^*$, $i \in N$
  - $w^0 = e$
  - $w^{i+1} = w^i \circ w$, $i \geq 0$
  - e.g.: $(re)^1 = re$, $(do)^2 = dodo$
- Definition of the reversal of a string, w^R:
  - if $|w| = 0$, $w^R = w = e$
  - if $|w| > 0 \rightarrow \exists\, a \in \Sigma$, $u \in \Sigma^*$ such that $w = ua \rightarrow w^R = au^R$
  - e.g.:
    - $(car)^R = rac$
    - (A man a plan a canal Panama)^R =
      = A man a plan a canal Panama

…

Alphabets, strings, and languages

- Theorem: for any strings w and x, $(wx)^R = x^R w^R$
  - e.g.: $(walnut)^R = (nut)^R(wal)^R = tunlaw$
- Proof:
  - basis step: $|x| = 0 \rightarrow x = e$, and $(wx)^R = (we)^R = w^R = ew^R = e^R w^R = x^R w^R$
  - induction hypothesis for n
    - if $|x| \leq n \rightarrow (wx)^R = x^R w^R$

Alphabets, strings, and languages

- Proof:
  - induction step: $|x| = n+1 \rightarrow x = ua$, $u \in \Sigma^*$, $a \in \Sigma$, $|u| = n$
    - $(wx)^R = (w(ua))^R$ since $x=ua$
    - $= ((wu)a)^R$ since concatenation is associative
    - $= a(wu)^R$ by the definition of reversal of $(wu)a$
    - $= a(u^R w^R)$ by the induction hypothesis
    - $= (au^R)w^R$ since concatenation is associative
    - $= (ua)^R w^R$ by the definition of the reversal of ua
    - $= x^R w^R$ since $x=ua$

Alphabets, strings, and languages

- Definition of language, L: a set of strings over Σ
- Special languages:
  - Ø: a language with 0 string
  - Σ: a language with |Σ| one letter strings
  - Σ*: contains all possible string over Σ

Alphabets, strings, and languages

- Defining languages:
  - listing all its items, e.g.: L = {aba, czr, d, f} is a language over {a, b, c, ….., z}
  - specify a property which is true for all strings in the language
    - infinite languages can be defined in this way
    - e.g.: L = {w $\in$ Σ^* : w starts with ab}

Alphabets, strings, and languages

- Definition of union of languages:
  - $L_1 \cup L_2 = \{w : w \in L_1 \text{ or } w \in L_2\}$
    - someone uses | instead of $\cup$
- Definition of concatenation of languages:
  - then $L_1 \circ L_2 = \{w \in \Sigma^* : w = x \circ y, x \in L_1, y \in L_2\}$
    - L_1L_2 also means the concatenation
- An important property: finite automata are closed under union ➡

Alphabets, strings, and languages

- L_1L_2 is similar to the Descartes product
  - $|L_1L_2| \leq |L_1| * |L_2|$
- E.g.:
  - $\Sigma = \{a, b\}$, $L_1 = \{a, aa\}$, $L_2 = \{bb, a\}$
  - $L_1L_2 = \{abb, aa, aabb, aaa\}$
- E.g.:
  - $\Sigma = \{a, b\}$, $L_1 = \{ab, a\}$, $L_2 = \{a, ba\}$
  - $L_1L_2 = \{aba, abba, aa\}$

Alphabets, strings, and languages

- Definition of Kleene star of a language, L*:
  - $L^* = \{w \in \Sigma^* : w = w_1 \circ ..... \circ w_k,\ k \geq 0, w_1, \ldots\ldots, w_k \in L\}$
    - set of all strings obtained by concatenating zero or more strings from L
  - the concatenation of zero strings is *e* and the concatenation of one string is the string itself
  - $L^+ = LL^*$, $L? = L \cup \{e\}$
- Stephen Cole Kleene (1909 –1994)
  - American mathematician
  - helped to lay the foundations for theoretical computer science

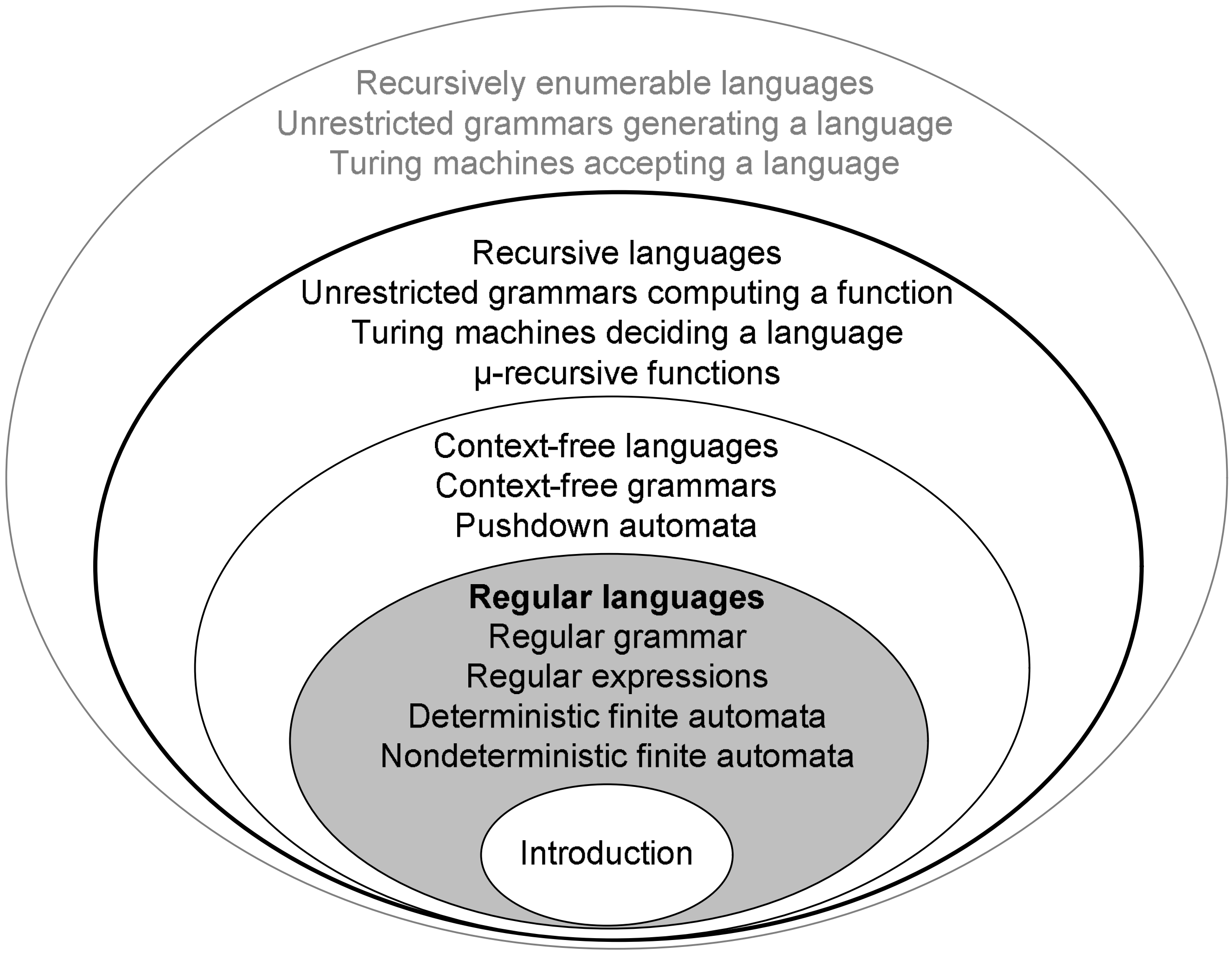
Examples

- Examples and questions:
  - if L = {01,1,100} → 110001110011 ∈ L*, since 110001110011 = 1∘100∘01∘1∘100∘1∘1
  - if L = {ab, ba, acb} → abacbab ∈ L*
  - if L = Ø → L* = {*e*}
    - the only possible concatenation $w_1 \circ ..... \circ w_k$ with k = 0
  - 100011100 = 100∘01∘1∘100
  - 1011001 = 1∘01∘100∘1
  - acbbaab = acb∘ba∘ab
  - baacbab = ba∘acb∘ab

Alphabets, strings, and languages

- Lemma: if $L_1 \subseteq L_2 \rightarrow L_1^* \subseteq L_2^*$ from the definition of Kleene star
- Theorem: if $L = \{w \in \{0, 1\}^* : w$ has an unequal number of 0 and 1$\} \rightarrow L^* = \{0, 1\}^*$
- Proof:
  - $\{0, 1\} \subseteq L$, since both 0 and 1 has an unequal number of 0 and 1 $\rightarrow \{0, 1\}^* \subseteq L^*$ by the lemma
  - $L^* \subseteq \{0, 1\}^*$
    - $B \subseteq \Sigma^* = \{0, 1\}^*$ is true for each language B
  - $L^* = \{0, 1\}^*$, the subset is true for both directions

Finite representation of language



Finite representation of language

- Theorem: only a small portion of the languages can be represented finitely
- Proof:
  - Σ is an alphabet with all possible letter
  - Σ*, the set of all possible words, is countably infinite
  - P(Σ*), the number of all possible language, is uncountable
  - a language representation is a word
    - does not matter if the elements are listed or a common property is given
  - there are only countably infinite language representation but there are uncountable languages

Regular expressions

- Motivating example:
  - L = {w $\in$ {0, 1}* : w has two or three occurrences of 1 and the first and second are not consecutive}
  - this language can be described with only singleton sets and language operations

Regular expressions

- L = {0}*∘{1}∘{0}*∘{0}∘{1}∘{0}*∘(({1}∘{0}*)∪{0}*)
  - {0} = language containing string 0
  - {0}* = Kleene star of the previous language
  - {0}*∘{1} = concatenation of the previous language and language {1}
- it is more simple to omit the braces and write L=0*10*010*(10*∪0*)
  - we need an exact definition what this expression does mean

Regular expressions

- Definition of regular expression, RE over alphabet Σ: strings over $\Sigma \cup \{\underline{\cup}, {}^{\Theta}, \underline{\emptyset}, \underline{(}, \underline{)}\}$ that can be obtained as
  - $\underline{\emptyset}$ and any element of Σ is a regular expression
  - $\underline{(}\alpha\beta\underline{)}$ is a regular expression
    - α and β are regular expressions
  - $\underline{(}\alpha \,\underline{\cup}\, \beta\underline{)}$ is a regular expression
  - α^{Θ} is a regular expression
  - nothing is regular expression unless it follows the previous four points

Regular expressions

- It is a recursive definition
- E.g.: $\Sigma = \{x, y\} \rightarrow \underline{\emptyset}, x, y, \underline{(}xy\underline{)}, \underline{(}xy\underline{)}^{\Theta}, \underline{((}xy\underline{)}^{\Theta} \underline{\cup} z\underline{)} \in \mathrm{RE}$
- For simplicity $\underline{(}, \underline{)}$ can be omitted
  - e.g.: $\underline{((}xy\underline{)}z\underline{)} = xyz, \underline{(}x \underline{\cup} y\underline{)} = x \underline{\cup} y$
  - beware: $\underline{(}xy\underline{)}^{\Theta} \neq xy^{\Theta}$

Regular expressions

- Regular expressions:
  - are language generators
    - describe how a generic specimen in the language is produced
    - language generators are not algorithms
  - represent a new way to define a language
  - $\underline{\cup}$, $^{\Theta}$, $\underline{Ø}$, $\underline{(}$, $\underline{)}$ are new symbols without meaning at the moment
    - these symbols appear only in regular expression
    - we will see that these symbol correspond to $\cup$, *, Ø, (,) so only these regular symbols will be used

Regular expressions

- Definition of function L: RE → languages:
  - α and β are regular expressions
  - $L(\underline{\emptyset}) = \emptyset$, $L(a) = \{a\}$, $\forall\, a \in \Sigma$
  - $L(\underline{(}\alpha\beta\underline{)}) = L(\alpha)L(\beta)$
  - $L(\underline{(}\alpha \,\underline{\cup}\, \beta\underline{)}) = L(\alpha) \cup L(\beta)$
  - $L(\alpha^{\Theta}) = L(\alpha)^*$
- Now the meaning of the new symbols are defined
  - from now on we use $\cup$ instead of $\underline{\cup}$, ...

Regular expressions

- $L(\emptyset^*) = L(\emptyset)^* = \emptyset^* = \{e\}$
- Nota bene: 'a' can be
  - symbol
  - string
  - language
  - RE

Properties of RE

- Commutative: $r \cup s = s \cup r$
- Associative:
  - $(r \cup s) \cup t = s \cup (r \cup t)$
  - $(rs)t = r(st)$
- Distributive:
  - $r(s \cup t) = rs \cup rt$
  - $(s \cup t)r = sr \cup tr$

Properties of RE

- Ø identity element:
  - Ør = r
  - rØ = r
- Idempotent: r** = r*
- Precedence in increasing order: $\cup$, $\circ$, *
- All these operators are left associative
  - if the same operator is at both sides of an operand → the left one must be performed first
- E.g.: (a) $\cup$ ((b)*(c)) is equivalent with a $\cup$ b*c

Example

- E.g.: $L(\ \underline{(}\underline{(}a \ \underline{\cup}\ b\underline{)}^{\Theta}a\underline{)}\ ) = ?$

$$
\begin{aligned}
&= L(\ \underline{(}\underline{(}a \ \underline{\cup}\ b\underline{)}^{\Theta}a\underline{)}\ ) = L(\underline{(}a \ \underline{\cup}\ b\underline{)}^{\Theta})L(a) \\
&= L(\underline{(}a \ \underline{\cup}\ b\underline{)}^{\Theta})\{a\} \\
&= L(\underline{(}a \ \underline{\cup}\ b\underline{)})^*\{a\} \\
&= (L(a) \cup L(b))^*\{a\} \\
&= (\{a\} \cup \{b\})^*\{a\} \\
&= \{a, b\}^*\{a\} \\
&= \{w \in \{a, b\}^* : w \text{ ends with 'a'}\}
\end{aligned}
$$

Example

- $L(a \cup ab)L(cd \cup dc) = ?$

  $= L(a \cup ab)L(cd \cup dc) =$

  $= (L(a) \cup L(ab))(L(cd) \cup L(dc)) =$

  $= (\{a\} \cup \{ab\})(\{cd\} \cup \{dc\}) =$

  $= \{a, ab\}\{cd, dc\} =$

  $= \{acd, adc, abcd, abdc\}$

Example

- $L(a \cup \emptyset)L(ab \cup ba) = ?$

  $= L(a \cup \emptyset)L(ab \cup ba) =$

  $= (L(a) \cup L(\emptyset))(L(ab) \cup L(ba)) =$

  $= (\{a\} \cup \emptyset)(\{ab\} \cup \{ba\}) =$

  $= \{a\}\{ab, ba\}=\{aab, aba\}$

Example

- True or false?
  - baa $\in$ L(a*b*a*b*)
  - L(b*a*) $\cap$ L(a*b*) = L(a* $\cup$ b*)
  - L(a*b*) $\cap$ L(c*d*) = Ø
  - abcd $\in$ L((a(cd)*b)*)
    - false because the first iteration of the outermost * can generate "ab" but after that there is a compulsory "a"

Regular languages

- Definition 1 of regular languages, $\Re$: the set of languages satisfying the following properties
  - $\emptyset \in \Re$, $\{a\} \in \Re$, $\forall$ $a \in \Sigma$
  - if $A, B \in \Re \rightarrow A \cup B \in \Re$, $A \circ B \in \Re$, $A^* \in \Re$
  - if S is a set of languages and it satisfies the first two points $\rightarrow \Re \subseteq S$ ($\Re$ is minimal)
- $\Re$ is the closure of the basic languages respect to union, concatenation, and Kleene star

Regular languages

- Nota bene:
  - $\Re$ is a set of languages, a language is a set of strings
  - don't confuse language with grammar

Example

- Give regular expression RE such that L(RE) = {w $\in$ {a, b}*}
  - RE = (a*b*)* or RE = (a $\cup$ b)*
- Give regular expression RE such that L(RE) = {w $\in$ {a, b}* | abba is a substring of w}
  - RE = (a*b*)*abba(a $\cup$ b)*
- Give regular expression RE such that L(RE) = {w $\in$ {a}* | #a is odd}
  - RE = a(aa)*
- Give regular expression RE such that L(RE) = {w $\in$ {a, b}* | #a is odd}
  - RE = b*ab*(b*ab*ab*)*

Example

- Give regular expression RE such that $L(RE) = \{w \in \{a, b\}^* \mid \#a \text{ is even or } \#a \bmod 3 = 0\}$
  - $RE_1 = (b^*ab^*ab^*)^* \cup b^*$
  - $RE_2 = (b^*ab^*ab^*ab^*)^* \cup b^*$
  - $RE = RE_1 \cup RE_2 = (b^*ab^*ab^*)^* \cup b^* \cup (b^*ab^*ab^*ab^*)^*$

Regular languages

- Theorem: every finite language is regular
- Proof:
  - let $|L| = n$, $w_i \in \Sigma^*$ the possible strings in L
  - let RE $R = w_1 \cup w_2 \cup ... \cup w_n$
  - $L = L(R)$

Regular expressions

- Definition 2 of regular languages: every language which can be described by a regular expression
- We cannot describe some languages by regular expressions though they have very simple descriptions by other means
  - $L = \{a^n b^n : n \geq 0\}$ not regular

Summary

- Alphabets, strings, and languages
- Finite representation of language
- Regular expressions
- Properties of RE
- Regular languages

Next time

- Deterministic finite automata

Elements of the Theory of Computation

Lesson 4

2.1. Deterministic finite automata

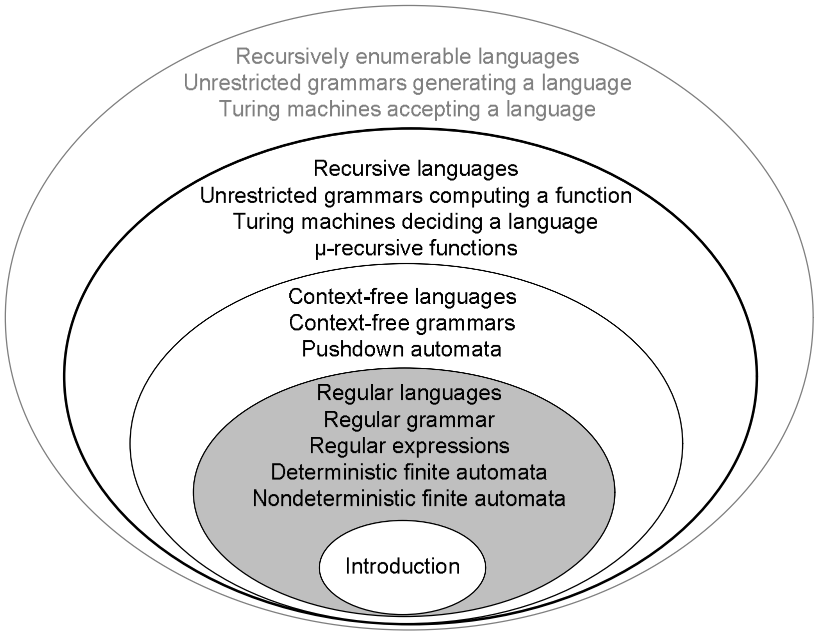
University of Pannonia

Dr. István Heckl, Istvan.Heckl@gmail.com

Version 47

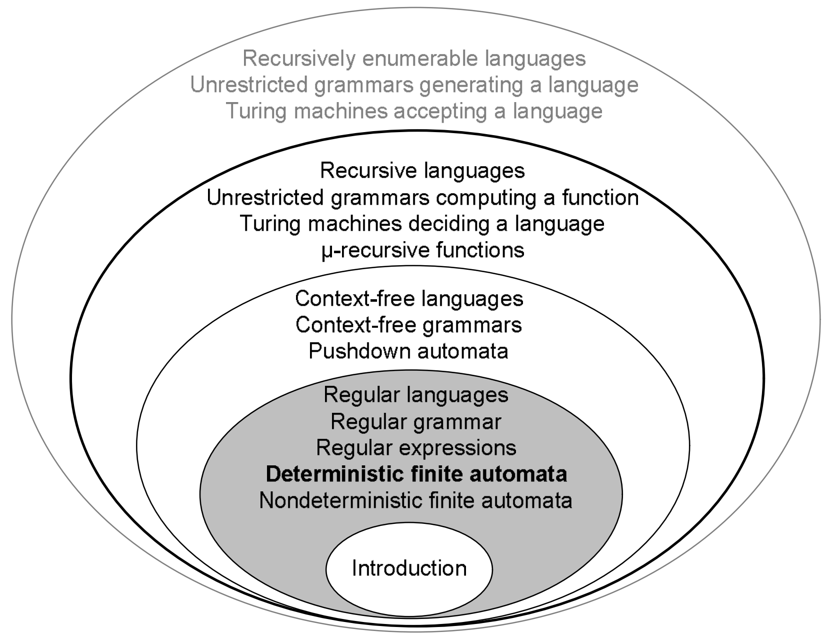
Last time

- Alphabets, strings, and languages
- Finite representation of language
- Regular expressions
- Properties of RE
- Regular languages



Deterministic finite automata

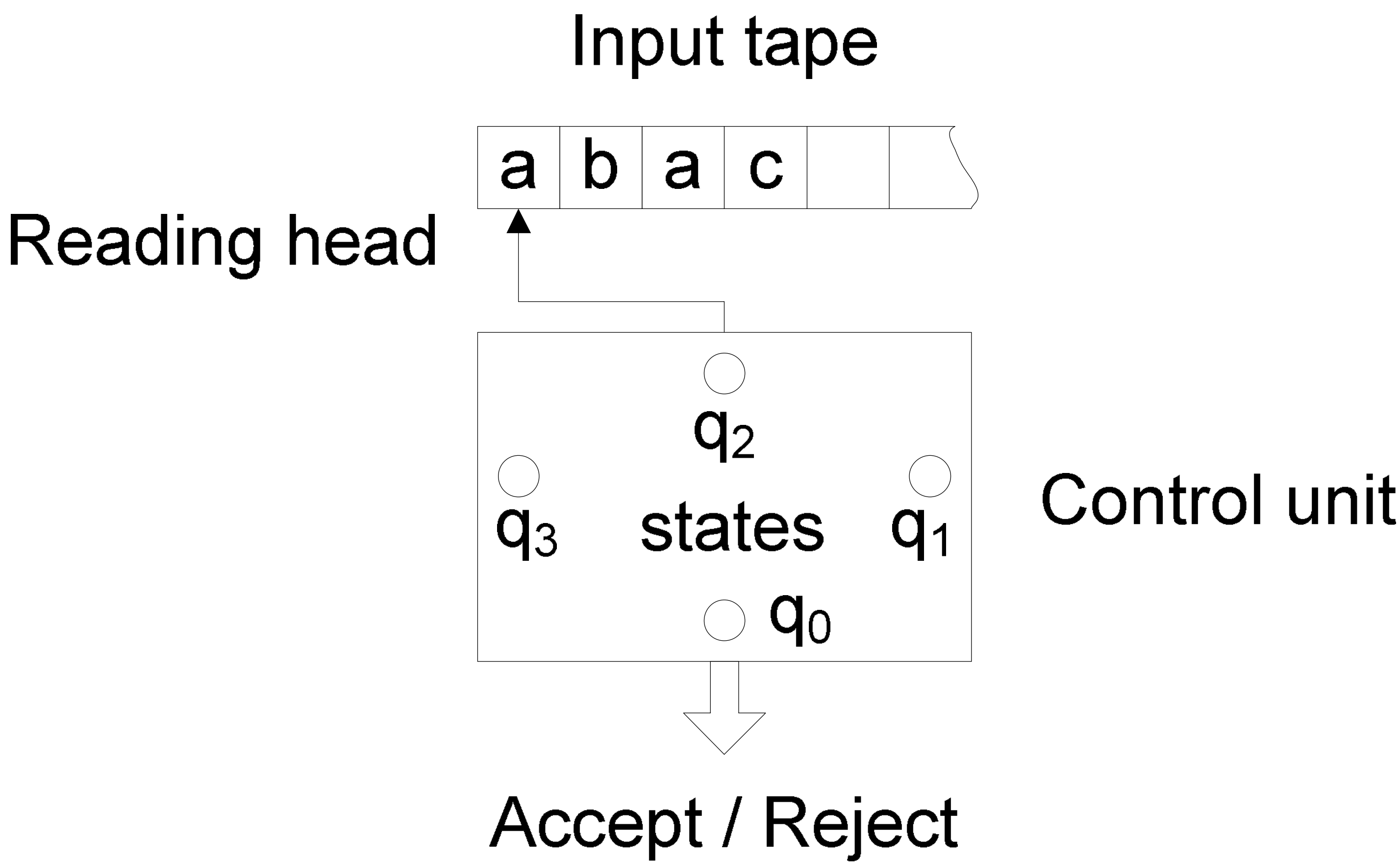
- Structure of DFA
- The operation of DFA
- State diagram
- Configuration
- Yield in one step
- Computation
- Yield
- String accepted by DFA
- Language accepted by DFA



Deterministic finite automata

- Deterministic finite automaton, DFA: mathematical model for a machine that can accept certain types of languages
  – it is called a language recognizer
- DFA is
  – deterministic because it is unambiguous what to do next
  – finite because it is defined with finite sets
  – automaton because does not need user interaction

Structure of DFA



Deterministic finite automata

- Definition of deterministic finite automaton, M: a quintuple $(K, \Sigma, \delta, s, F)$, where:
  - K set of states (finite)
  - Σ alphabet (finite)
  - δ transition function, $K \times \Sigma \rightarrow K$
    - δ is defined for all pair in $K \times \Sigma$
  - $s \in K$, initial state
  - $F \subseteq K$, the set of final states
    - F could be called accepting states

The operation of a DFA

- A DFA begins
  - in state s
  - reading the first symbol in the input tape
- The DFA changes state
  - if
    - M is in state q
    - reading symbol $\sigma \in \Sigma$

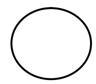
The operation of a DFA

- then
  - M passes to state $\delta(q, \sigma)$
    - the new state is determined uniquely as δ is a function
  - the reading head steps one to the right
- After reading the last symbol, DFA halts
  - the input is accepted if DFA is in $q \in F$
  - otherwise the input is rejected

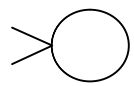
State diagram

- State diagram is a representation of a DFA
  - it is a directed graph
    - nodes represent states
      - the outdegree of each node $|\Sigma|$
      - name the states
    - arrows are labeled with elements of δ
- Sink: a node with only reflexive outgoing arcs

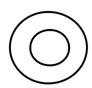
State diagram



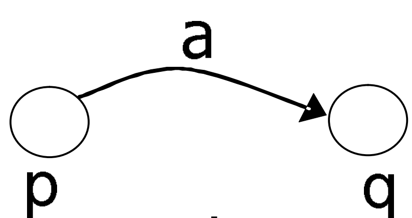
state



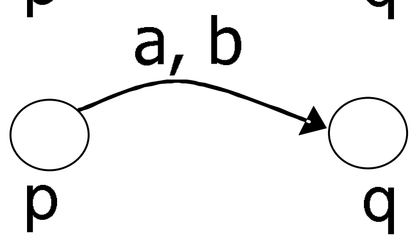
initial state

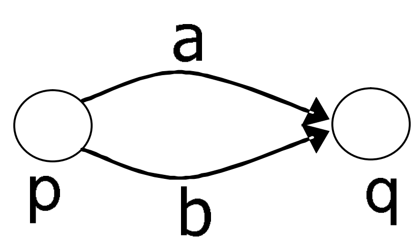


final state



$\delta(p, a)=q$

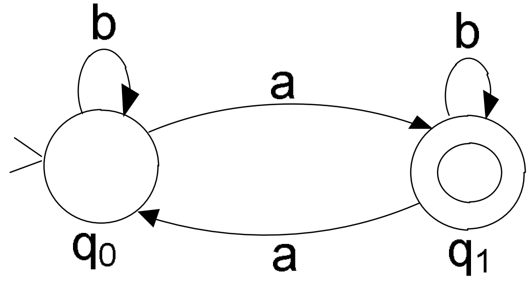




Deterministic finite automata

- DFA is denoted as either $M = (K, \Sigma, \delta, s, F)$ or $M(K, \Sigma, \delta, s, F)$
- Give the state diagram of $M = (K, \Sigma, \delta, s, F)$!
  - $K = \{q_0, q_1\}$
  - $\Sigma = \{a, b\}$
  - $s = q_0$
  - $F = \{q_1\}$

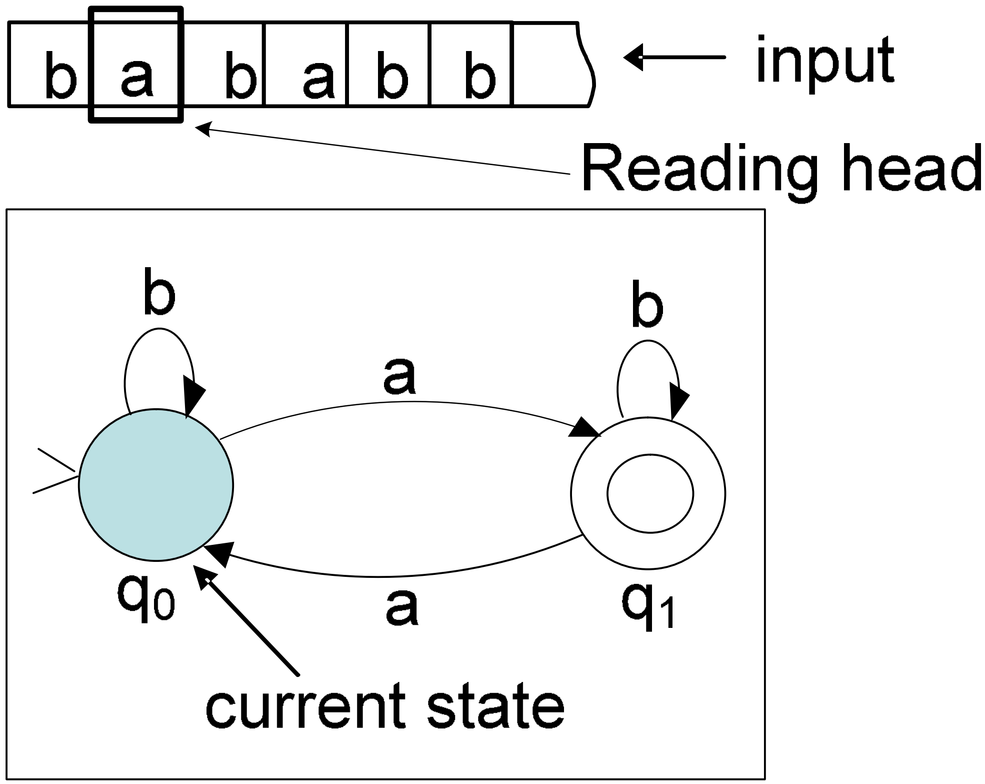
q	**σ**	**δ(q, σ)**
q_0	a	q_1
q_0	b	q_0
q_1	a	q_0
q_1	b	q_1



Configuration

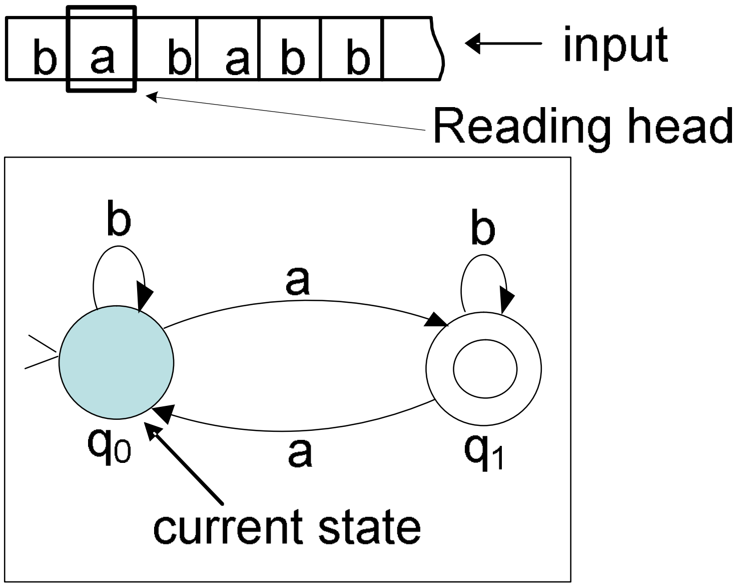
- Definition of configuration of a DFA $M = (K, \Sigma, \delta, s, F)$: an ordered pair of the current state of M and the unread part of the input
  - it is an element of $K \times \Sigma^*$
  - there is no need to store the whole input because the reading head cannot go to the left, so the already read input cannot affect the result
  - the effect of the already read input is in the current state
  - e.g.: $(q_5, aaabb)$

Example



Configuration of DFA: (q_0, ababb)

Example



Configuration of DFA: (q_0, ababb)

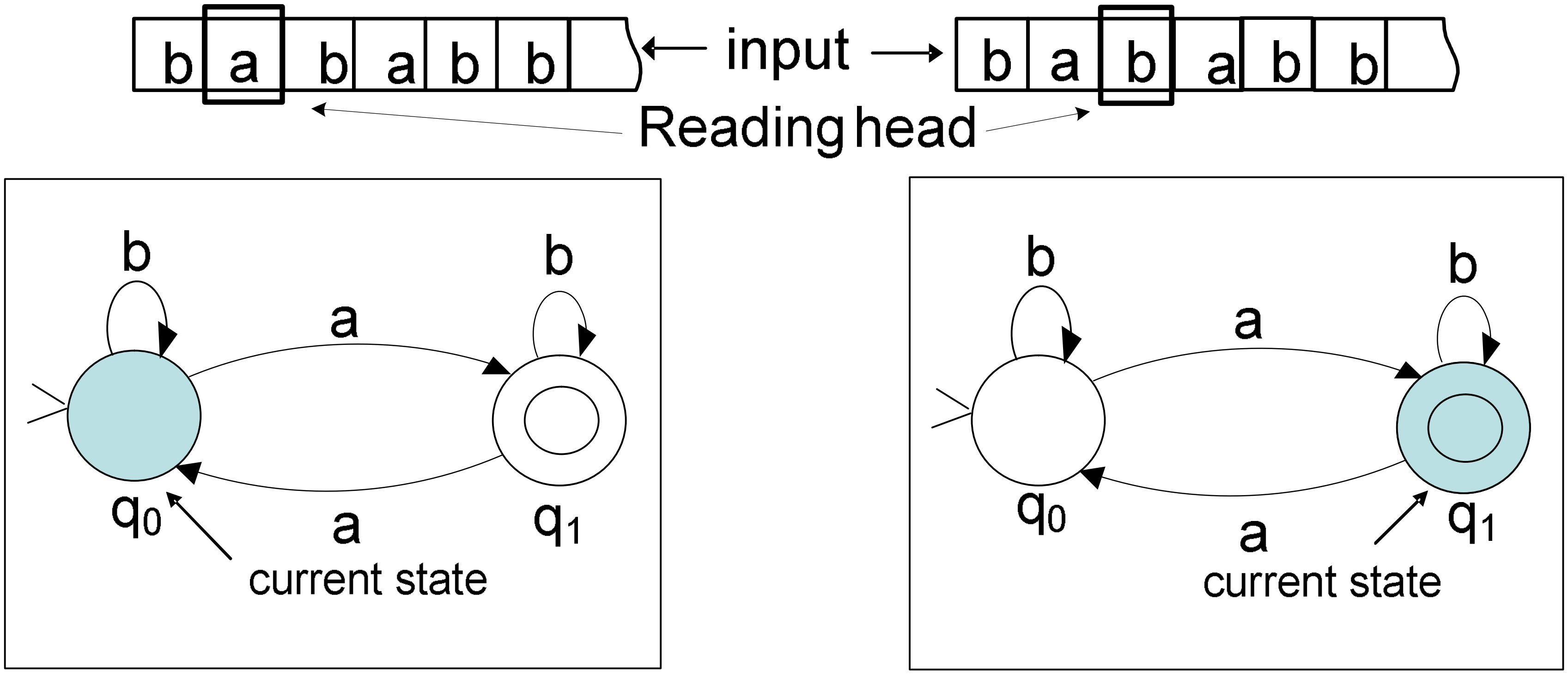
- Is it a valid configuration if w = bababb
  - (q_0, abbba)
  - (q_0, bb)
  - (q_1, b)
  - (q_1, abb)
  - (q_2, babb)

- M accepts L = {w : the number of 'a' in w is odd}
  - q_0 - the number of 'a' is even
  - q_1 - the number of 'a' is odd

Yield in one step

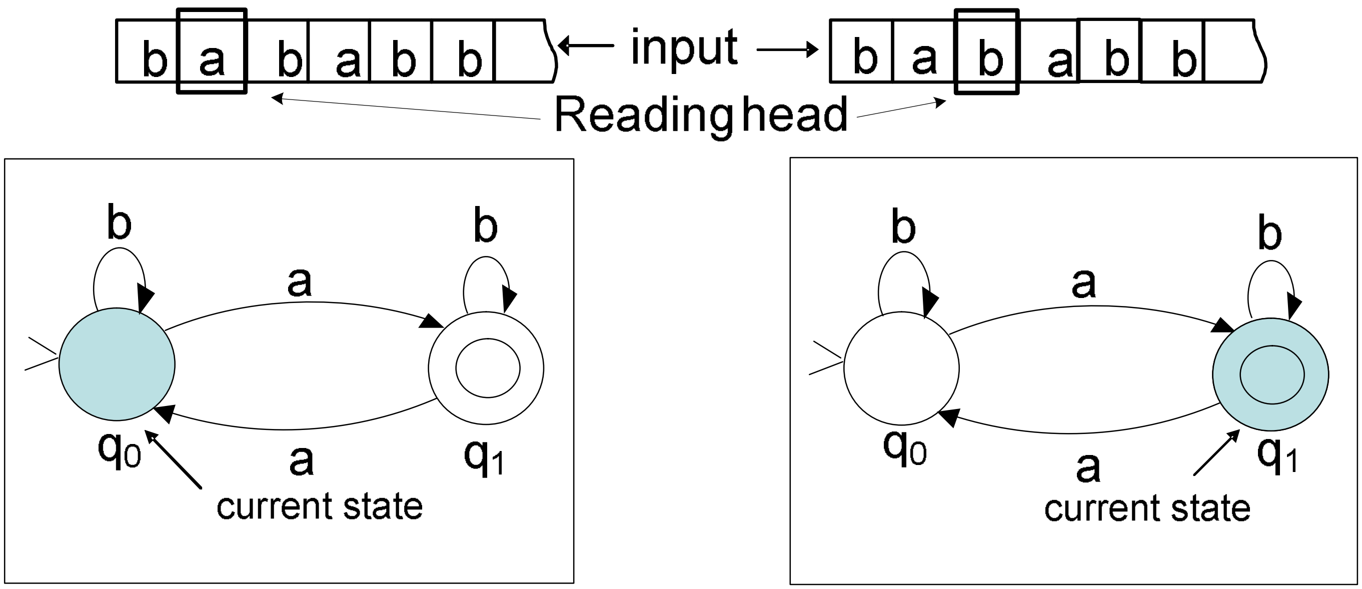
- Definition of yield in one step of a DFA, $|-_M$: a relation between two "neighboring" configurations
  - formally:
    - if $a \in \Sigma$, $y \in \Sigma^*$, $q, p \in K$, $\delta(q, a) = p$
    - then $((q, ay), (p, y)) \in |-$ or $(q, ay) |- (p, y)$
  - we say: (q, ay) yields (p, y) in one step
    - there is an appropriate transition between the two configurations
  - $|-_M \subseteq (K \times \Sigma^*)^2$
- If it is unambiguous that the yield corresponds to which DFA then the subscript M may be omitted

Example



Yield in one step: $(q_0, ababb) \vdash (q_1, babb)$

Example



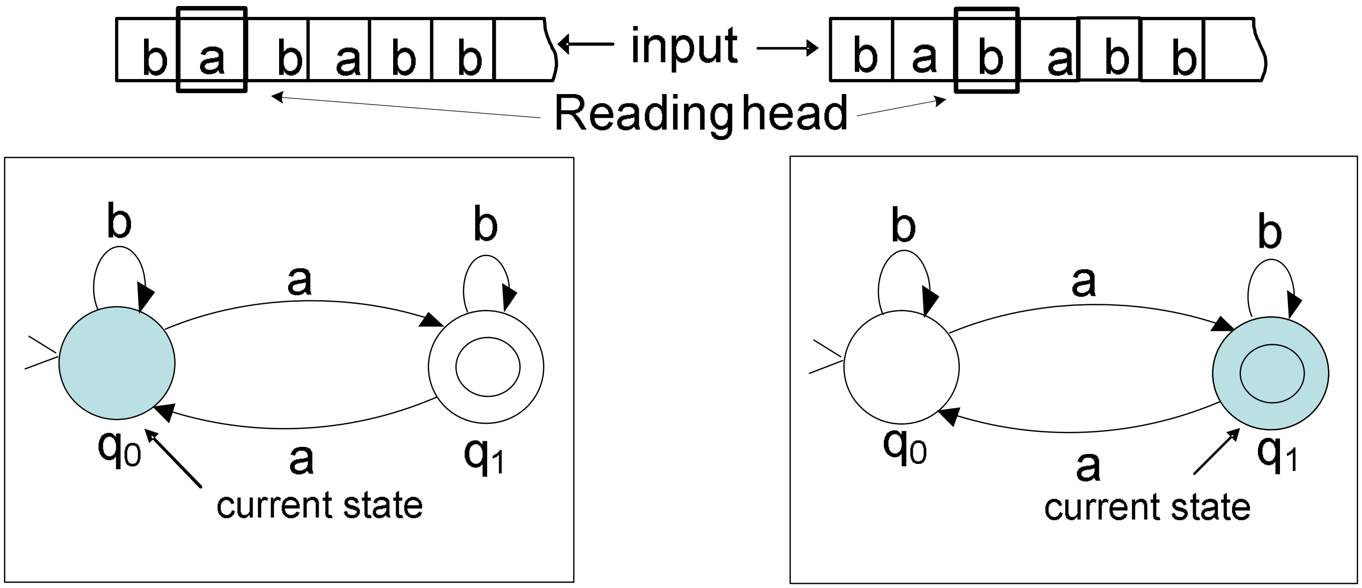
Yield in one step: (q_0, ababb) |- (q_1, babb)

- Is it a valid yield?
  - (q_0, abbba) |- (q_1, abbba)
  - (q_0, aba) |- (q_1, ba)
  - (q_0, abb) |- (q_0, bb)
  - (q_0, bab) |- (q_0, ab)

Computation

- Definition of computation by DFA M: a sequence of configuration C_0, C_1, ... C_n such that C_0 |- C_1 |- ... |- C_n
  - e.g.: (q_1, abaa) |- (q_2, baa) |- (q_1, aa) |- (q_3, a)
  - the length of a computation is the number of yield in one step
  - the first and the last configuration can be connected with the yield in n steps relation, signed as $|-^n$
    - e.g.: (q_1, abaa) $|-^3$ (q_3, a)
- We will use computation at NFA

Example



Yield in one step: (q_0, ababb) |- (q_1, babb)

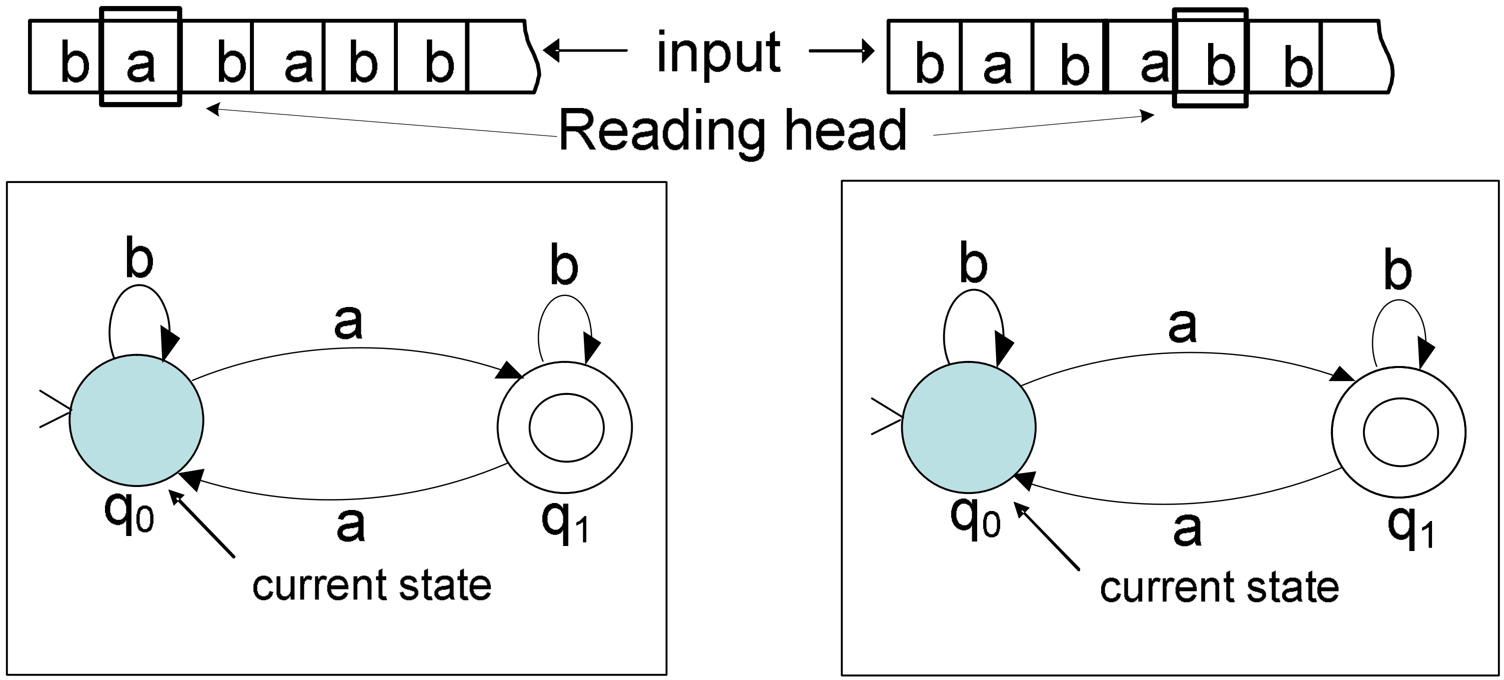
Are they valid yields?

(q_0, bababa) |- (q_0, ababa) |- (q_1, baba) |- (q_1, aba) |- (q_1, ba)

Yield

- Definition of yield of a DFA, $|-_M*$: the reflexive, transitive closure of $|-_M$
  - if (q', w') can be reached from (q, w) through a number of yield in one step operation then the yield operation holds between (q, w) and (q', w')
    - denote as: (q, w) |-* (q', w')
  - zero step is possible: (q, w) $|-_M*$ (q, w)

Example



Yield: (q_0, ababb) |-* (q_0, bb)

(q_0, bababa) |-* (q_1, aba)

(q_0, bababa) |-* (q_1, ba)

(q_0, bababa) $|-^3$ (q_1, aba)

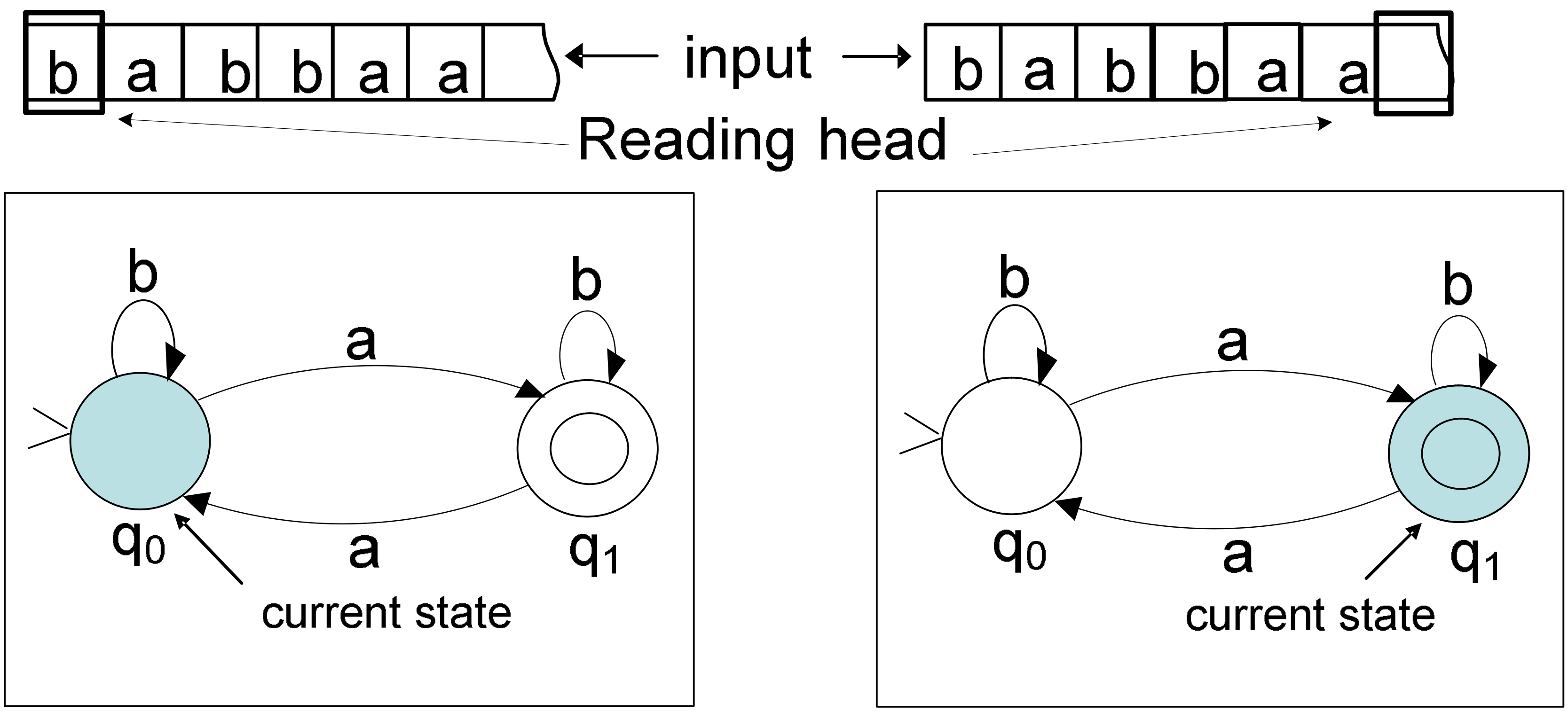
(q_0, bababa) $|-^5$ (q_0, ba)

String accepted by DFA

- Definition of word accepted by DFA: $w \in \Sigma^*$ is accepted by M if $(s, w) \vdash_M^* (q, e), q \in F$
  – if an accepting configuration is reachable from the initial configuration through yield operation
    - initial configuration: (s, w) = (starting state, whole input)
    - accepting configuration: the state of the configuration belongs to the final states, and w = *e*

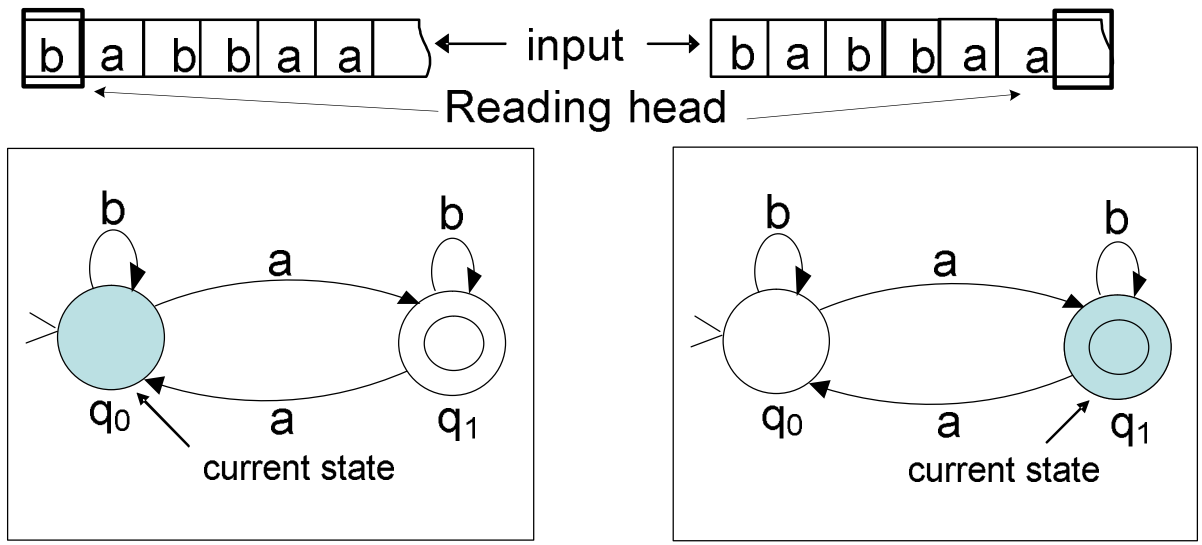
Example

- babbaa is accepted by DFA



(q_0, babbaa) (q_1, *e*)

Example



(q0, babbaa) (q1, *e*)

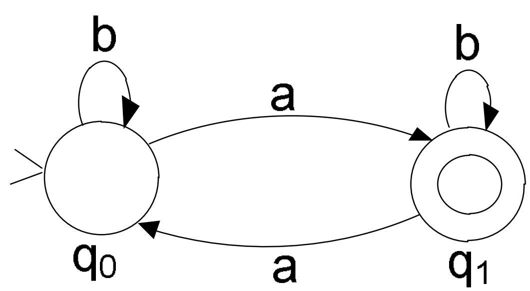
- Are the following strings accepted?
  - abbba
  - bbbabbb
  - bababab

Language accepted by DFA

- Definition of language accepted by DFA M, L(M): the set of strings accepted by M
  - $L(M) = \{w \in \Sigma^* : (s, w) \mid\!-_M^* (q, e), q \in F\}$
- The number of steps required to decide if $w \in L(M)$ or not: $|w|$
  - one symbol is processed in every step

Example

- Give the computation of bbabaa and aabaab by DFA M!
  - L(M) = {w : in w the number of 'a' are odd}



Example

- Input: bbabaa

$(q_0, \text{bbabaa}) \mid\!-_M (q_0, \text{babaa})$

$\mid\!-_M (q_0, \text{abaa})$

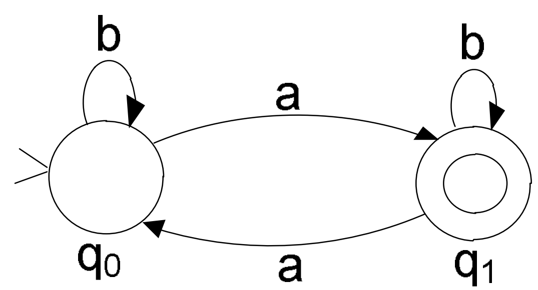
$\mid\!-_M (q_1, \text{baa})$

$\mid\!-_M (q_1, \text{aa})$

$\mid\!-_M (q_0, \text{a})$

$\mid\!-_M (q_1, e)$

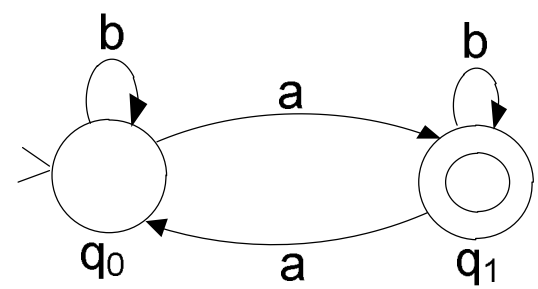
Accepted



Example

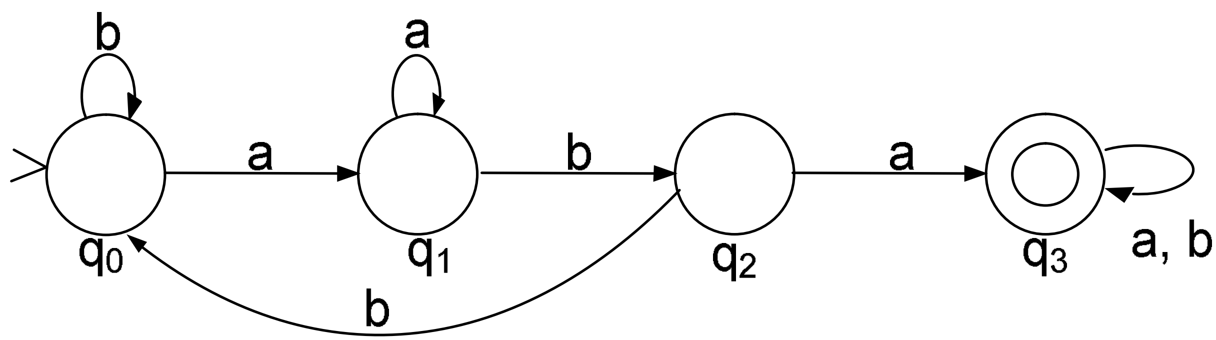
- Input: aabaab

$(q_0, \text{aabaab}) \vdash_M (q_1, \text{abaab})$
$\vdash_M (q_0, \text{baab})$
$\vdash_M (q_0, \text{aab})$
$\vdash_M (q_1, \text{ab})$
$\vdash_M (q_0, \text{b})$
$\vdash_M (q_0, e)$
Rejected



Example

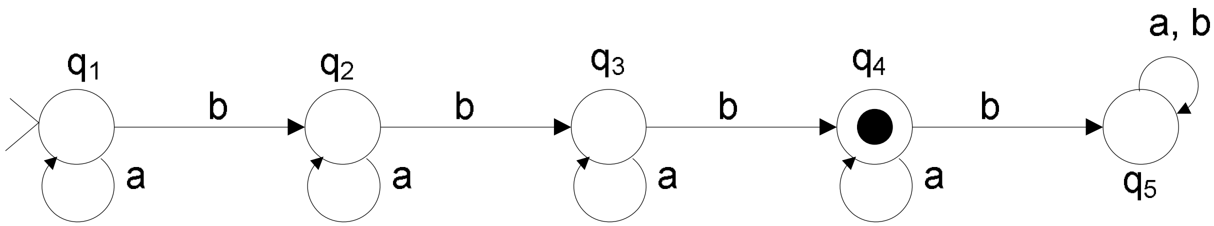
- $L(M) = \{w : w \in \{a, b\}^*$ and w contains the string aba$\}$



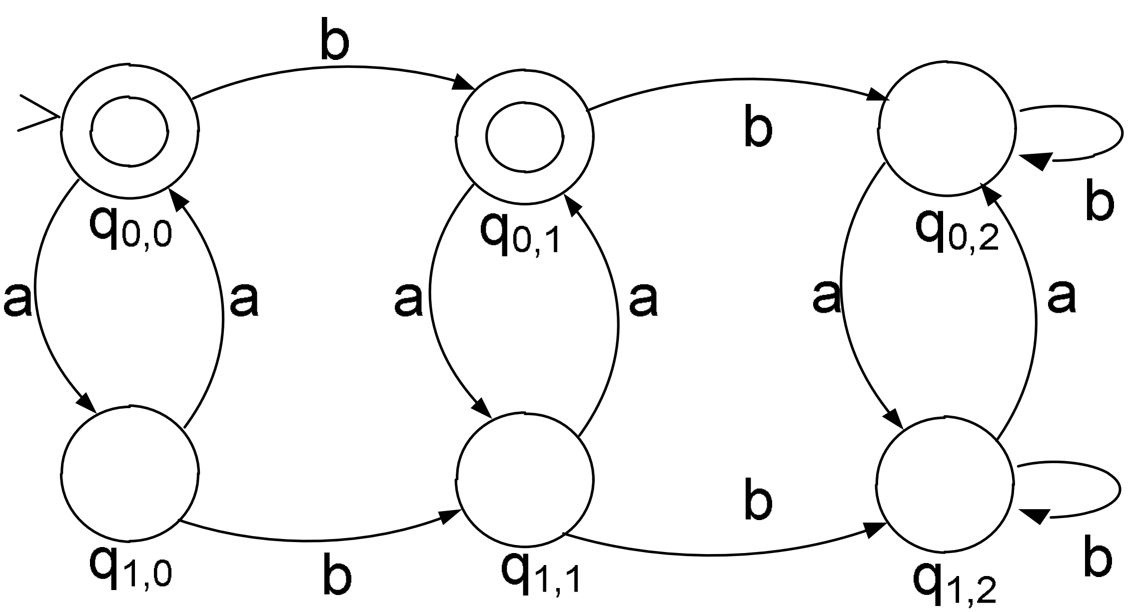
- It is important to give the meaning of the states
  - q_0: 0 symbol (*e*) is read from aba
  - q_1: 1 symbol (a) is read from aba
  - q_2: 2 symbol (ab) is read from aba
  - q_3: 3 symbol (aba) is read from aba

Example

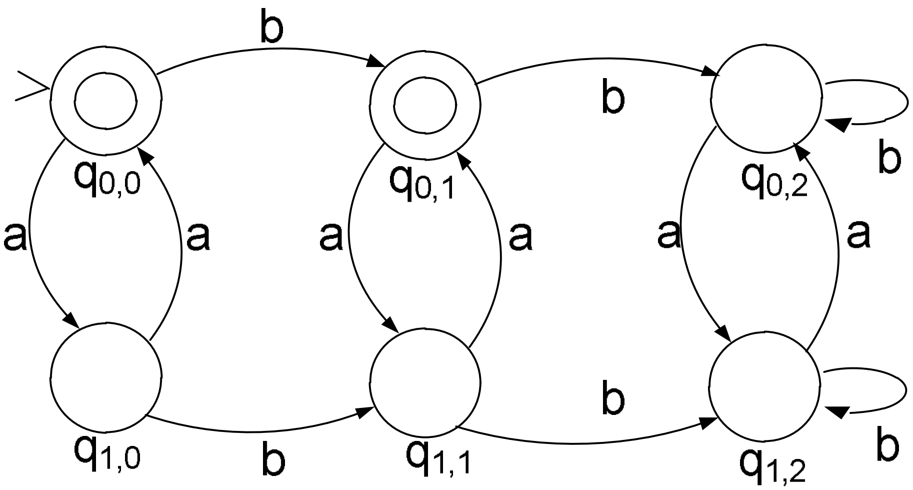
- Define DFA M such that L(M) = {w ∈ {a, b}* | #b = 3}!
- States:
  - q_1: #b = 0
  - q_2: #b = 1
  - q_3: #b = 2
  - q_4: #b = 3 – final state
  - q_5: #b ≥ 4

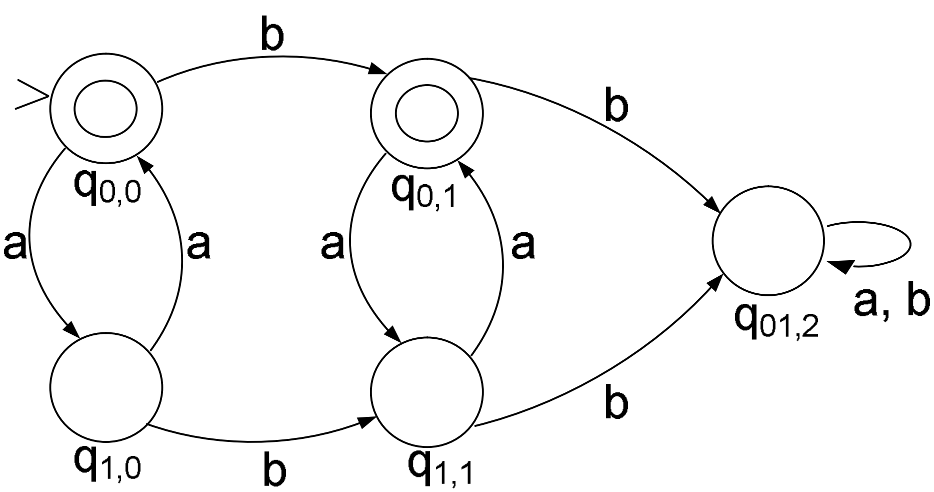


- L(M) = {w ∈ {a, b}* : in w the number of 'a' is even and there is at most one b in w}
  - $q_{0,0}$: #a is even, no b yet
  - $q_{1,0}$: #a is odd, no b yet
  - $q_{1,1}$: #a is odd, 1 b occurred
  - $q_{0,1}$: #a is even, 1 b occurred
  - $q_{0,2}$: #a is even, more than 1 b occurred
  - $q_{1,2}$: #a is odd, more than 1 b occurred



- The two DFAs are equivalent





Summary

- Structure and operation of DFA
- State diagram
- Yield in one step
- Yield
- String accepted by DFA
- Language accepted by DFA

Next time

- Non-deterministic finite automata

Elements of the Theory of Computation

Lesson 5

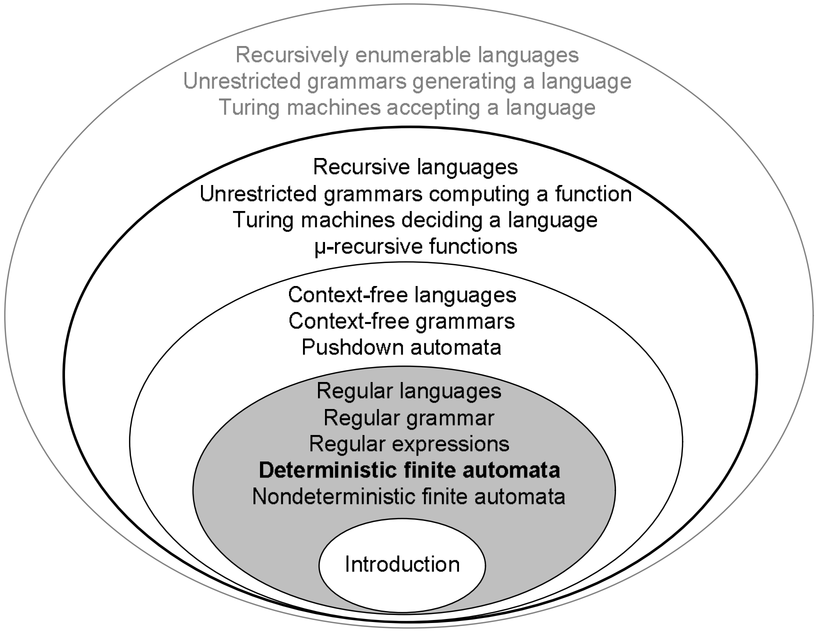
2.2. Non-deterministic finite automata

University of Pannonia

Dr. István Heckl, Istvan.Heckl@gmail.com

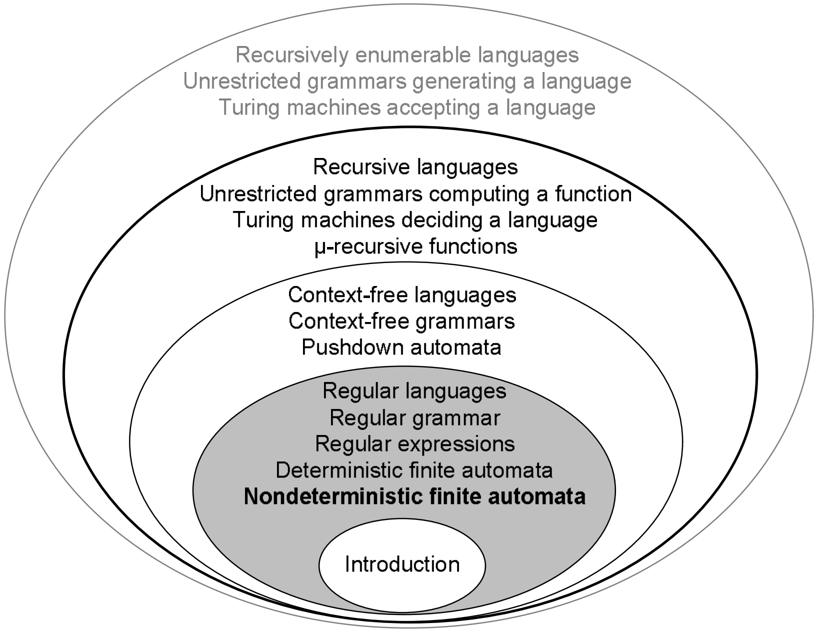
Last time

- Structure of DFA
- The operation of DFA
- State diagram
- Configuration
- Yield in one step
- Computation
- Yield
- String accepted by DFA
- Language accepted by DFA



Non-deterministic finite automata

- Non-deterministic behavior
- NFA
- Difference of DFA and NFA
- Yield in one step
- Yield
- String accepted by NFA
- Language accepted by NFA
- Automata equivalence
- DFA $\leftrightarrow$ NFA



Non-deterministic behavior

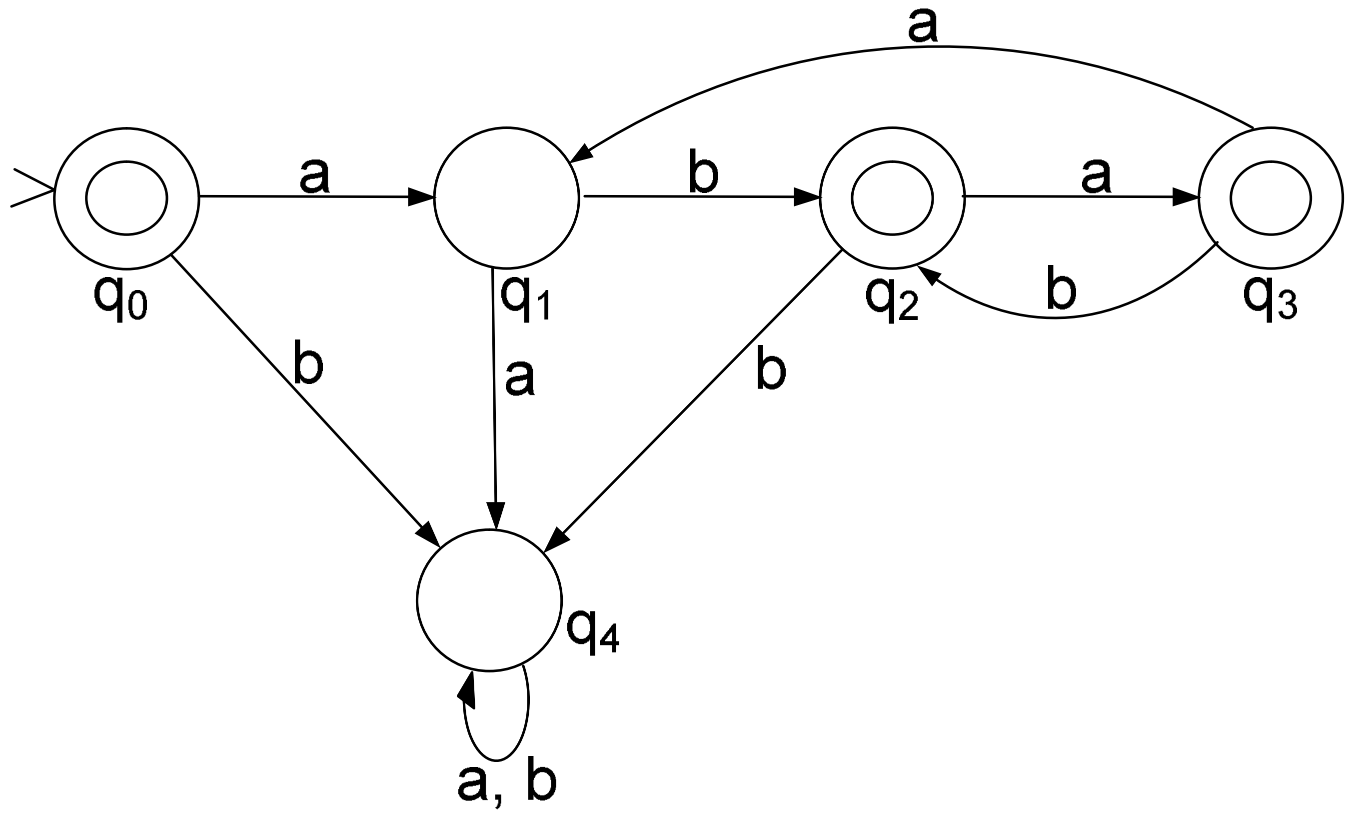
- Definition of non-deterministic behavior: the operation such a way that there is a number of possible next step and there is no way to decide between them
  - at finite automaton: change the actual state in such a way that is only partially determined by the current state and input symbol

Non-deterministic behavior

- In other words:
  - there are several possible next states for a given input
    - our model does not determine which state should be chosen
  - it is not a realistic model as there is no way to implement it directly
    - though it can be simulated by taking into account every possibility

Motivating example

- Consider the language L = (ab ∪ aba)*, which is accepted by the next DFA

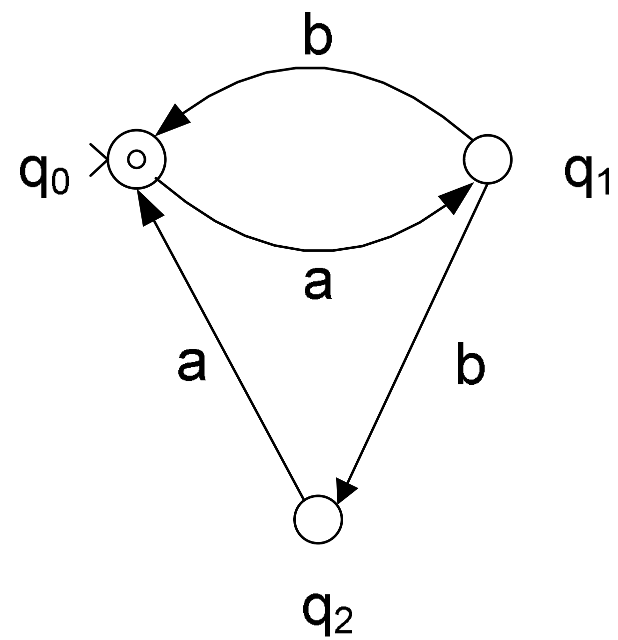


Motivating example

- The previous figure is quite complex
  - it is hard to check if it is DFA at all
  - it is hard to check if it accept L
  - there is no simpler DFA that can accept L

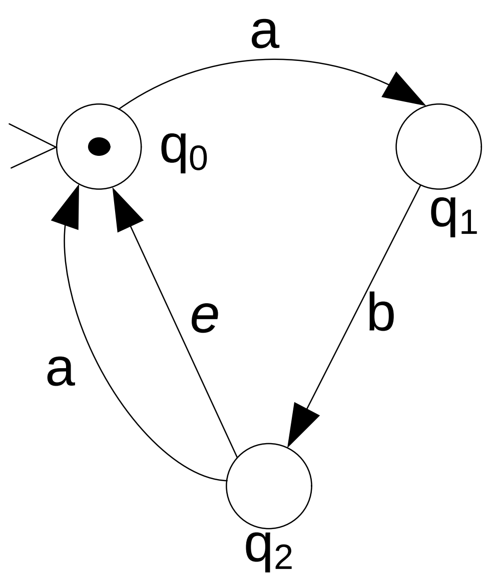
Motivating example

- The current automata is simpler though it is not DFA
  - from q_0 there is no b arrow
  - from q_1 there is two b arrow but not 'a' arrow
  - from q_2 there is no b arrow
- Operation
  - ab: $q_0 \rightarrow q_1 \rightarrow q_0$
  - aba: $q_0 \rightarrow q_1 \rightarrow q_2 \rightarrow q_0$
- At q_1 we might choose the wrong way
  - let us suppose we always guess correctly



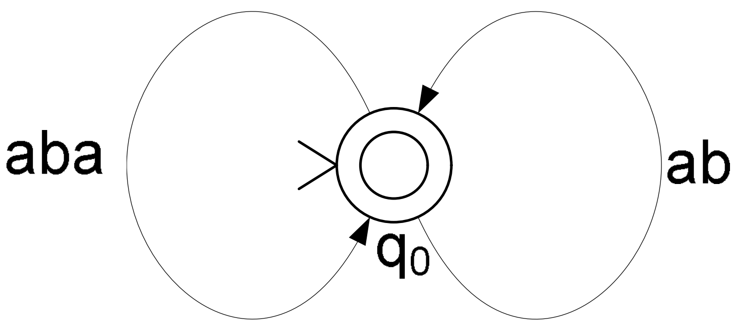
Motivating example

- The current automata also accepts L but it is not DFA either
  - there is an empty transition, *e*, from q_2
    - we might go to q_0 without moving the head
- Operation
  - ab: $q_0 \rightarrow q_1 \rightarrow q_2 \rightarrow$(reading *e*) q_0
  - aba: $q_0 \rightarrow q_1 \rightarrow q_2 \rightarrow$ (reading 'a') q_0



Motivating example

- The current automata also accepts L but it is not DFA either
  - a whole string is read in a single transition
    - the labels "ab" and "a, b" on an arc has different meanings
- Operation
  - ab: $q_0 \rightarrow$(reading ab) q_0
  - aba: $q_0 \rightarrow$(reading aba) q_0



Non-deterministic finite automata NFA

- Definition of non-deterministic finite automata, M: a quintuple (K, Σ, Δ, s, F), where
  - K set of states (finite)
  - Σ alphabet (finite)
  - $s \in K$ initial state
  - $F \subseteq K$ the set of final states
  - $\Delta \subseteq K \times \Sigma^* \times K$ the transition relation
    - the 2nd edition book define $\Delta \subseteq K \times (\Sigma \cup e) \times K$
- The push down automaton is similar to NFA
  - it is also non-deterministic

Non-deterministic finite automata, NFA

- Michael Oser Rabin (1931)
- Dana Stewart Scott (1932)

Differences between DFA and NFA

- Δ is a relation and not a function:
  - for one state several next states may be reached reading the same input
  - Δ may not be defined for all K×Σ
  - there are transition with strings instead of symbols
  - there are *e* transitions

Configuration

- Definition of configuration of a NFA M = (K, Σ, Δ, s, F): an ordered pair of the current state of M and the unread part of the input
  - it is an element of K×Σ*
  - there is no need to store the whole input because the reading head cannot go to the left, so the already read input cannot affect the result
  - e.g.: (q_8, aaba)

Yields in one step

- Definition of yield in one step of an NFA, $|\!-_M$: a relation between two "neighboring" configurations
  – formally:
    - if $x, y \in \Sigma^*$, $q, p \in K$, $(q, x, p) \in \Delta$
    - then $((q, xy), (p, y)) \in |\!-$ or $(q, xy) |\!- (p, y)$
  – we say: (q, xy) yields (p, y) in one step
    - there is an appropriate transition between the two configurations
  – $|\!-_M \subseteq (K \times \Sigma^*)^2$
- If it is unambiguous that the yield corresponds to which NFA then the subscript M may be omitted

Computation

- Definition of computation by NFA M: a sequence of configuration C_0, C_1, ... C_n such that C_0 |- C_1 |- ... |- C_n
  - e.g.: (q_1, abaa) |- (q_2, aa) |- (q_1, aa) |- (q_3, a)
  - the length of a computation is the number of yield in one step applied
  - the first and the last configuration can be connected with the yield in n steps relation, signed as $|-^n$
    - e.g.: (q_1, abaa) $|-^3$ (q_3, a)

Yield

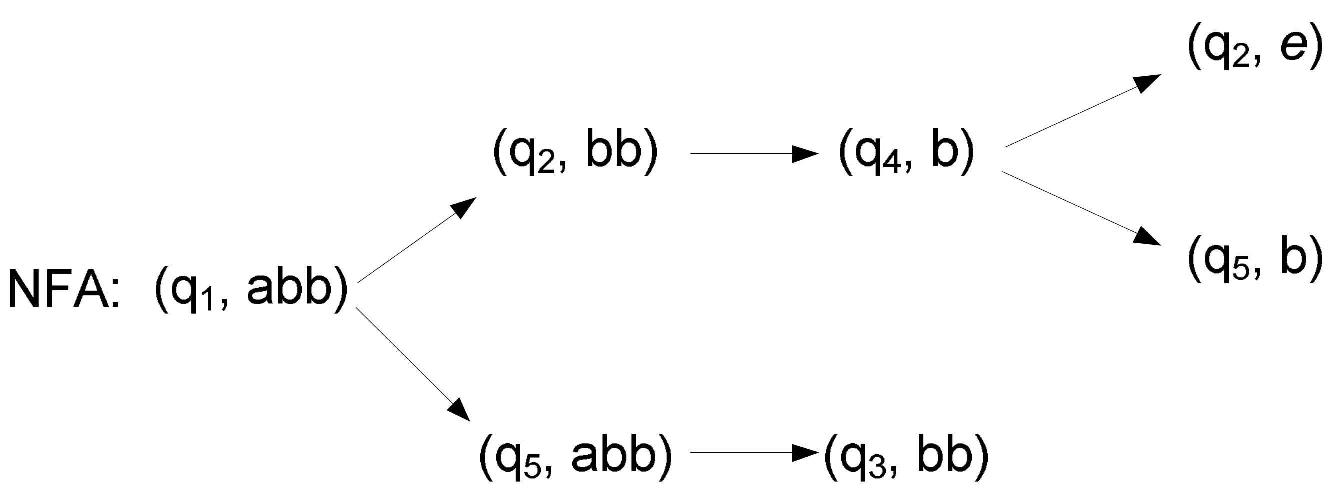
- Definition of yield of an NFA, $|-_M*$: the reflexive, transitive closure of $|-_M$
  - if (q', w') can be reached from (q, w) through a number of yield in one step operation then the yield operation holds between (q, w) and (q', w')
    - denote as: (q, w) $|-_M*$ (q', w')
  - zero step is possible: (q, w) $|-_M*$ (q, w)

String accepted by NFA

- Definition of strings accepted by NFA: $w \in \Sigma^*$ is accepted by M if (s, w) $\vdash^*$ (q, *e*), $q \in F$
  - the automaton is in final state
  - the whole input is read
- If NFA M cannot process the whole input because of the missing transitions then w is rejected

String accepted by NFA

- The yield in NFA can lead to different configurations reading the same input
  – there are possible branching at the computation of w
  – if there is as much as one path from (s, w) to (q, *e*) such that $q \in F$ then w is accepted

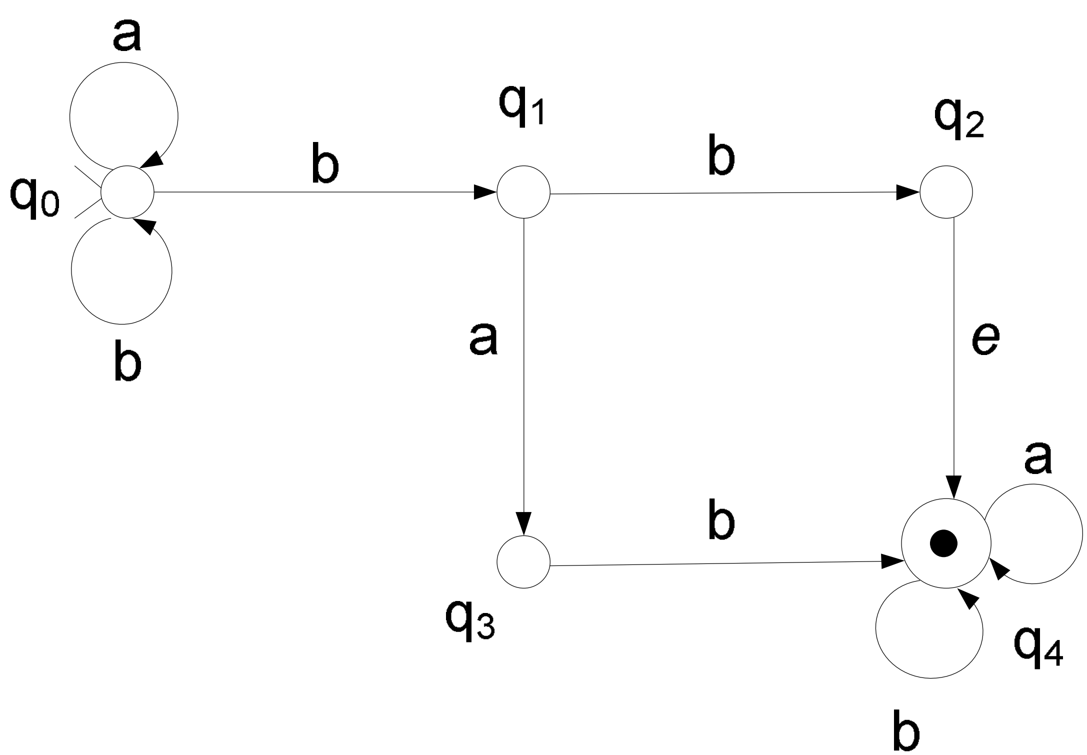


Language accepted by NFA

- Definition of language accepted by NFA M, L(M): the set of strings accepted by M
  - $L(M) = \{w \in \Sigma^* : (s, w) \vdash_M^* (q, e), q \in F\}$

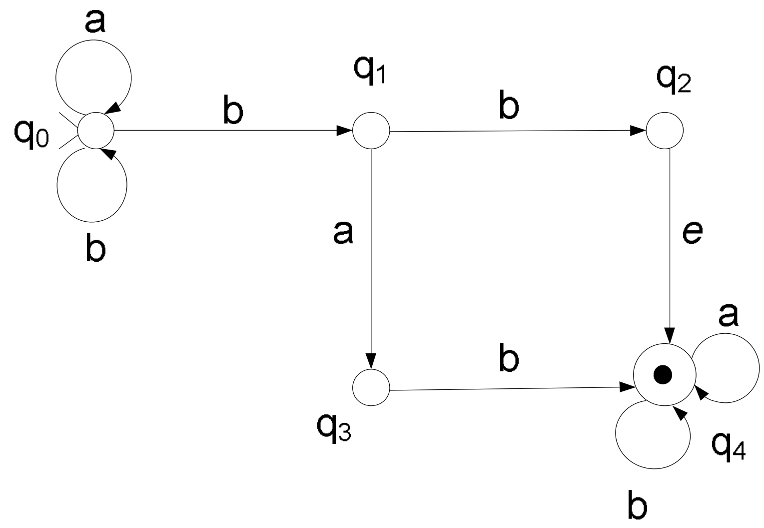
Example

- NFA that accept all strings containing bb or bab

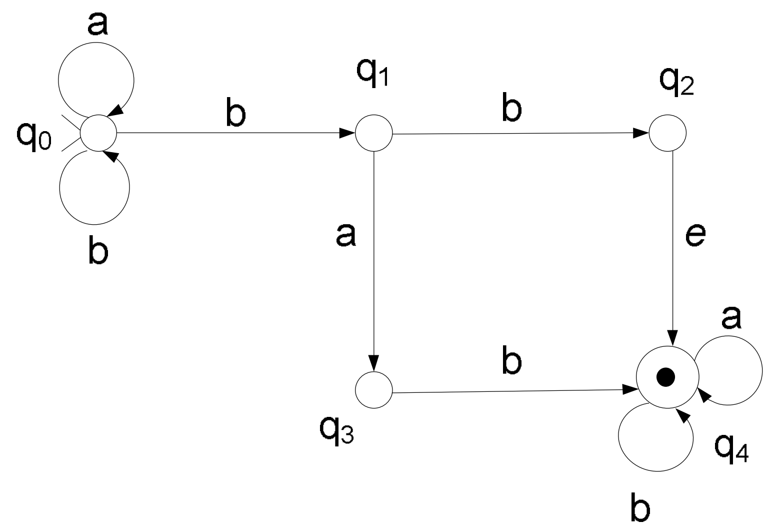


Example

- Formally (K, Σ, Δ, s, F), where:
  - $K = \{q_0, q_1, q_2, q_3, q_4\}$
  - $\Sigma = \{a, b\}$
  - $\Delta = \{(q_0, a, q_0), (q_0, b, q_0), (q_0, b, q_1), (q_1, b, q_2), (q_1, a, q_3), (q_2, e, q_4), (q_3, b, q_4), (q_4, a, q_4), (q_4, b, q_4)\}$
  - $s = q_0$
  - $F = \{q_4\}$



Example



- Input: bababab
- Case 1:
  - (q_0, bababab) |- (q_0, ababab) |- (q_0, babab) |- (q_0, abab) |- … |- (q_0, *e*)
  - this computation ended in a non-final state
- Case 2:
  - (q_0, bababab) |- (q_1, ababab) |- (q_3, babab) |- (q_4, abab) |- (q_4, bab) |- (q_4, ab) |- (q_4, b) |- (q_4, *e*)
- The string is accepted because there is such a computation which leads to a final (accepting) configuration

Non-deterministic finite automata

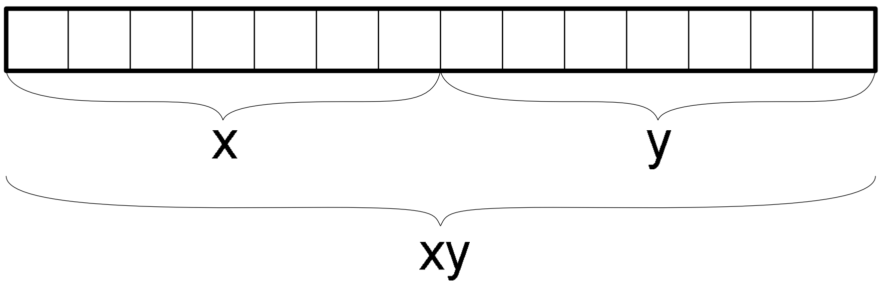
- End of input theorem: (q, x) |-* (p, *e*) ↔ (q, xy) |-* (p, y)
  - the end of the input, y, does not effect the operation of M until it is read
  - e.g.: (q_3, alma) |-* (q_7, *e*) ↔ (q_3, almafa) |-* (q_7, fa)

Non-deterministic finite automata

- Proof:
  - (q, x) |-* $(p, e) \leftrightarrow (q, x) = (q_0, x_0)$ |- (q_1, x_1) |- (q_2, x_2) |- … |- $(q_n, x_n) = (p, e)$ by detailing the yield
    - $q_0, q_1, \ldots, q_n \in K, x_0, x_1, \ldots, x_n \in \Sigma^*$
  - (q_i, x_i) |- $(q_{i+1}, x_{i+1}) \leftrightarrow \exists (q_i, u_i, q_{i+1}) \in \Delta, u_i \in \Sigma^*$ such that $x_i = u_i x_{i+1}$, by the definition of yield in one step
  - $\exists (q_i, u_i, q_{i+1}) \in \Delta \leftrightarrow (q_i, u_i x_{i+1} y)$ |- $(q_{i+1}, x_{i+1} y)$ by the definition of yield in one step
  - $(q_i, u_i x_{i+1} y)$ |- $(q_{i+1}, x_{i+1} y) \leftrightarrow (q, xy)$ |-* (p, y)
    - by the transitive property of yield
    - $(q, xy) = (q_0, x_0 y), (q_n, x_n y) = (p, y)$

Non-deterministic finite automata

- Theorem: if (q, x) |-* (p, *e*), (p, y) |-* (r, *e*) → (q, xy) |-* (r, *e*)
  - example: (q_3, alma) |-* (q_7, *e*), (q_7, fa) |-* (q_4, e) ↔ (q_3, almafa) |-* (q_4, *e*)
  - let:
    - M = (K, Σ, Δ, s, F) be a NFA
    - q, r, p ∈ K
    - x, y ∈ Σ*



Non-deterministic finite automata

- Proof:
  - (q, x) |-* (p, *e*) → (q, xy) |-* (p, y) by the previous theorem
  - if (q, xy) |-* (p, y), (p, y) |-* (r, *e*) ⟶ (q, xy) |-* (r, *e*) by the transitive property of |-*

Automata equivalence

- Definition of the equivalence of finite automata M_1, M_2: $L(M_1) = L(M_2)$
  - the automata can have different states and transitions

NFA $\leftrightarrow$ DFA

- A DFA can be seen as a special type of NFA
  - there are no *e* transitions
  - one symbol is read in one transitions
  - the current state and symbol determines the next state uniquely
  - from each state there are exactly $|\Sigma|$ transitions

NFA $\leftrightarrow$ DFA

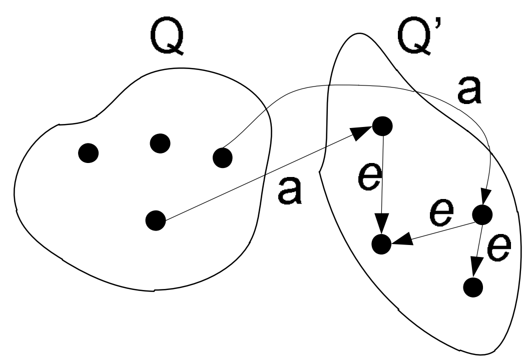
- Construction:
  - NFA is signed with q, Δ, F
  - DFA is signed with Q', δ', F'
  - q $\in$ K is one of the states which can be reached from s by consuming the input so far
  - idea: Q $\in$ K' is the set of states from K which can be reached from s by consuming the input so far

NFA $\leftrightarrow$ DFA

- Construction:
  - Q may have a label such as $\{q_1, q_5, q_{21}\}$ but it is a single state of K'
  - $\delta'(Q, a) = Q'$ is the set of states (of K) which can be reached from one state of Q by reading 'a'
    - possibly followed by a number of *e* transitions

NFA $\leftrightarrow$ DFA

- Construction:
  - formally:
    - $E(q) = \{p \in K, (q, e) \vdash^*_M (p, e)\}$
      - the set of states that can be reached from q by zero or more *e* transition
    - $K' = P(K)$
      - we may not need all of them
    - $\Sigma' = \Sigma$
    - $s' = E(s)$
    - $F' = \{Q \subseteq K : Q \cap F \neq \emptyset\}$
    - $\delta'(Q,a) = \bigcup_{q \in Q, (q,a,p) \in \Delta} E(p)$



NFA $\leftrightarrow$ DFA

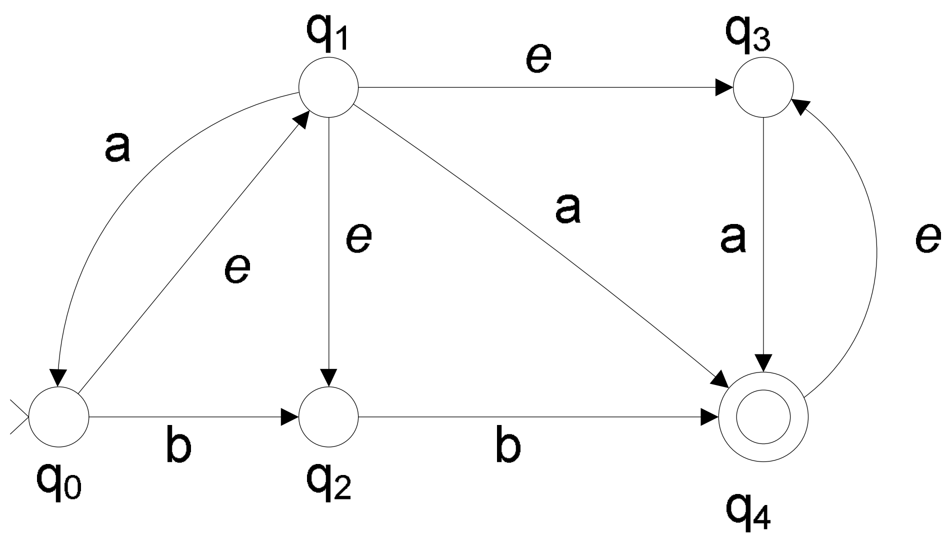
- Remarks:
  - M' is deterministic because the 4 properties to differentiate DFA from NFA holds
  - $\emptyset \in K'$
  - the cost to resolve determinism is to introduce $2^{|K|}$ new state
    - the increase is exponential

NFA $\leftrightarrow$ DFA

- E(q) is the closure of the set {q} under the relation
  {(p, r) : there is a transition (p, *e*, r) $\in \Delta$}
- E(q) can be computed by the following algorithm:

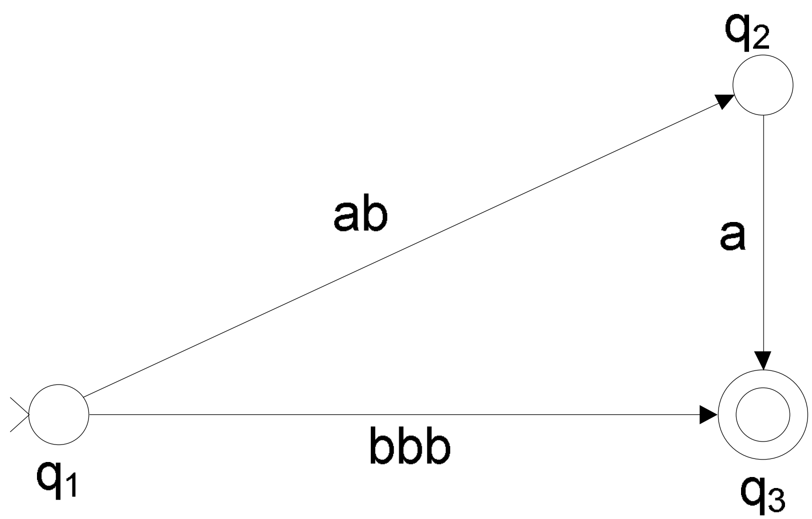
```
E(q) := {q};
while there is a transition (p, e, r) ∈ Δ
  with p ∈ E(q) and r ∉ E(q) do
   E(q) := E(q) ∪ {r};
```

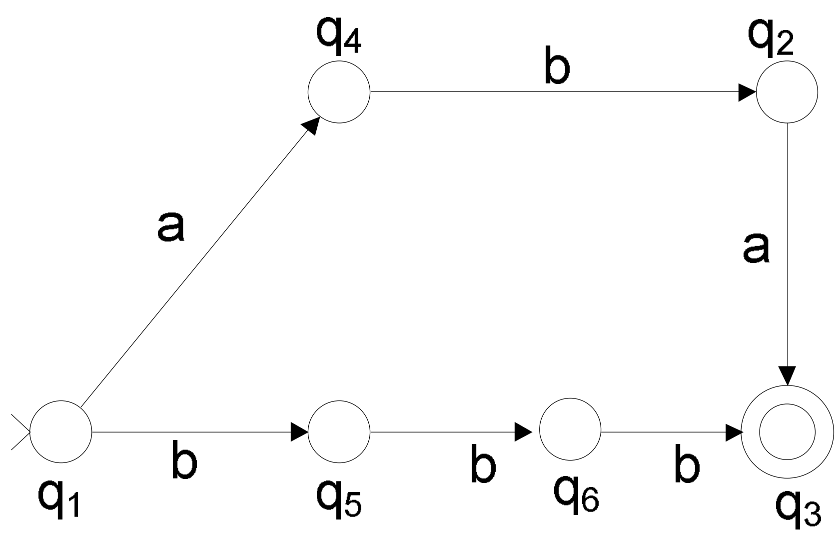
Example



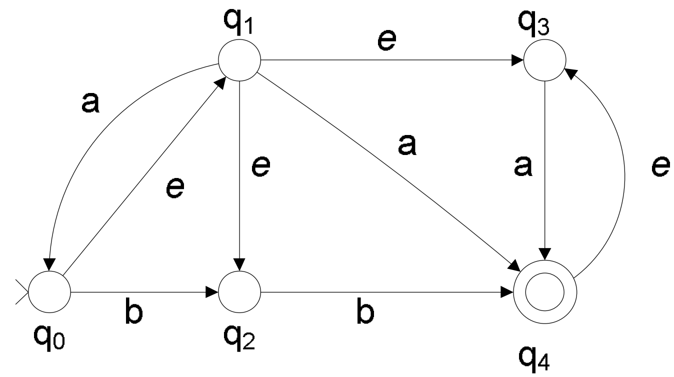
- $E(q_0) = \{q_0, q_1, q_2, q_3\}$
- $E(q_1) = \{q_1, q_2, q_3\}$
- $E(q_2) = \{q_2\}$
- $E(q_3) = \{q_3\}$
- $E(q_4) = \{q_3, q_4\}$

NFA ↔ DFA





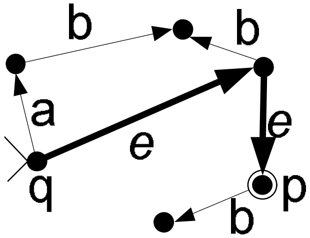
Example



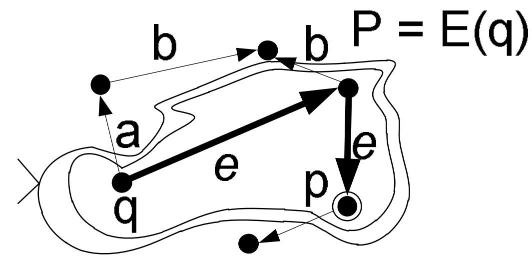
- Defining δ':
  - δ'(Q, a) = the set of all states of M that can be reached from one state of Q by reading 'a'
    - followed possibly by several *e* transitions
  - s' = $E(q_0) = \{q_0, q_1, q_2, q_3\}$
  - $\delta'(\{q_0, q_1, q_2, q_3\}, a) = E(q_0) \cup E(q_4) = \{q_0, q_1, q_2, q_3, q_4\}$
    - there are 'a' transitions only from q_1 to q_0 and q_4; and from q_3 to q_4

NFA $\leftrightarrow$ DFA

- Lemma: $(q, w) \vdash_{M}^{*} (p, e), p \in F \leftrightarrow (E(q), w) \vdash_{M'}^{*} (P, e)$, $p \in P \in F'$
  - NFA M and DFA M' accept the same w words
  - $w \in \Sigma^*$, $q, p \in K$
  - read "$p \in P$" as: some (not defined) P containing p

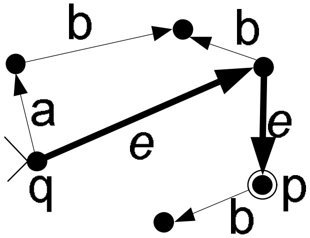


NFA ↔ DFA

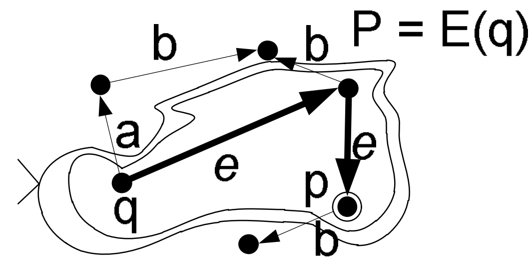


- Proof by induction on |w|:
  - basis step:
    - |w| = 0 ↔ w = *e*, we must show that (q, *e*) $|\!-_M^*$ (p, *e*), p ∈ F ↔ (E(q), *e*) $|\!-_{M'}^*$ (P, *e*), p ∈ P ∈ F'
    - let P = E(q)
    - (E(q), *e*) $|\!-_{M'}^*$ (E(q), *e*) by the reflexive property of $|\!-_{M'}^*$
    - (E(q), *e*) $|\!-_{M'}^*$ (P, *e*) by the previous two points

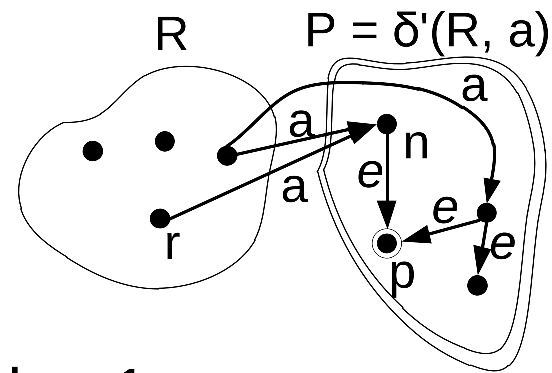
Exam: P1



NFA ↔ DFA



- (q, *e*) $|-_M^*$ (p, *e*) ↔ p ∈ E(q) by the definition of E(q)
- p ∈ P by the previous point and P = E(q)
- P∈ F' by p ∈ P, p ∈ F and the construction of F'
- comment:
  - M can go from q to p (a final state) through *e* arcs
  - M' performs 0 step while processing *e*
    - » E(q) is both initial and final state of M'

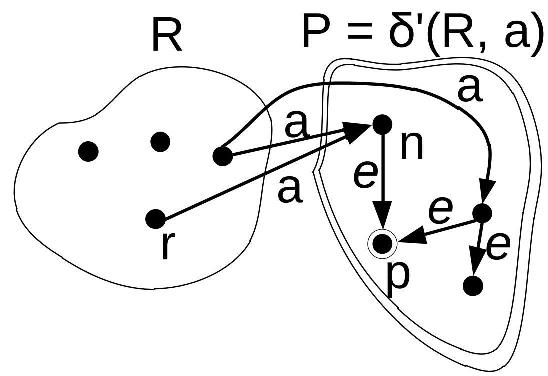


NFA ↔ DFA

– induction step: we prove the claim for k + 1
  - let w = va, $v \in \Sigma^*$, $a \in \Sigma$

– →
  - suppose: (q, va) $|\!-_M^*$ (p, *e*), $p \in F$ → (q, va) $|\!-_M^*$ (r, a) $|\!-_M$ (n, *e*) $|\!-_M^*$ (p, *e*), $r, n \in K$
    - detailing the yield
    - r, n exist but not defined exactly
  - (q, va) $|\!-_M^*$ (r, a) ↔ (q, v) $|\!-_M^*$ (r, *e*) by the end of input theorem
  - (q, v) $|\!-_M^*$ (r, *e*) ↔ (E(q), v) $|\!-_{M'}^*$ (R, *e*), $r \in R$ by the induction hypothesis
    - R is not defined, but we know that $r \in R$

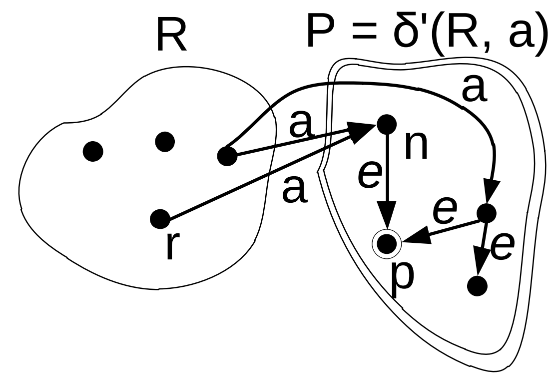
NFA ↔ DFA



- $(E(q), v) \vdash_{M'}^* (R, e)$, $r \in R \leftrightarrow$ $(E(q), va) \vdash_{M'}^* (R, a)$, $r \in R$ by the end of input theorem
- let $P = \delta'(R, a)$, P is not defined exactly
- $p \in P$ because $r \in R$, $(r, a) \vdash_M (n, e) \vdash_M^* (p, e)$, the construction of δ'
- $P = \delta'(R, a) \rightarrow (R, a) \vdash_{M'} (P, e)$ by the definition of the yield in one step
- $p \in P \in F'$ by the construction of F'
- $(E(q), va) \vdash_{M'}^* (R, a)$, $(R, a) \vdash_{M'} (P, e) \rightarrow (E(q), va) \vdash_{M'}^* (P, e)$, $P \in F'$ by the transitive property of yield

Exam: P2

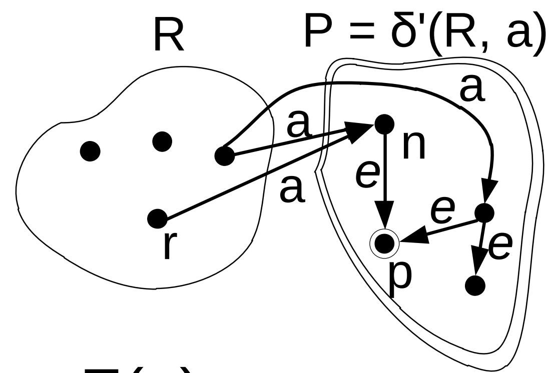
NFA ↔ DFA



– ←

- suspose: (E(q), va) $|\text{-}_{M'}^*$ (P, *e*), p ∈ P ∈ F' → (E(q), va) $|\text{-}_{M'}^*$ (R, a) $|\text{-}_{M'}$ (P, *e*), p ∈ P, R ∈ F'
  - detailing the yield
  - R exists but not defined exactly
- (E(q), va) $|\text{-}_{M'}^*$ (R, a) ↔ (E(q), v) $|\text{-}_{M'}^*$ (R, *e*) by the end of input theorem
- (E(q), v) $|\text{-}_{M'}^*$ (R, *e*) ↔ (q, v) $|\text{-}^*_{M}$ (r, *e*), r ∈ R by the induction hypothesis
- (q, v) $|\text{-}_{M}^*$ (r, *e*) ↔ (q, va) $|\text{-}_{M}^*$ (r, a) by the end of input theorem
- (R, a) $|\text{-}_{M'}$ (P, *e*) ↔ δ'(R, a) = P by the definition of the yield in one step

NFA ↔ DFA

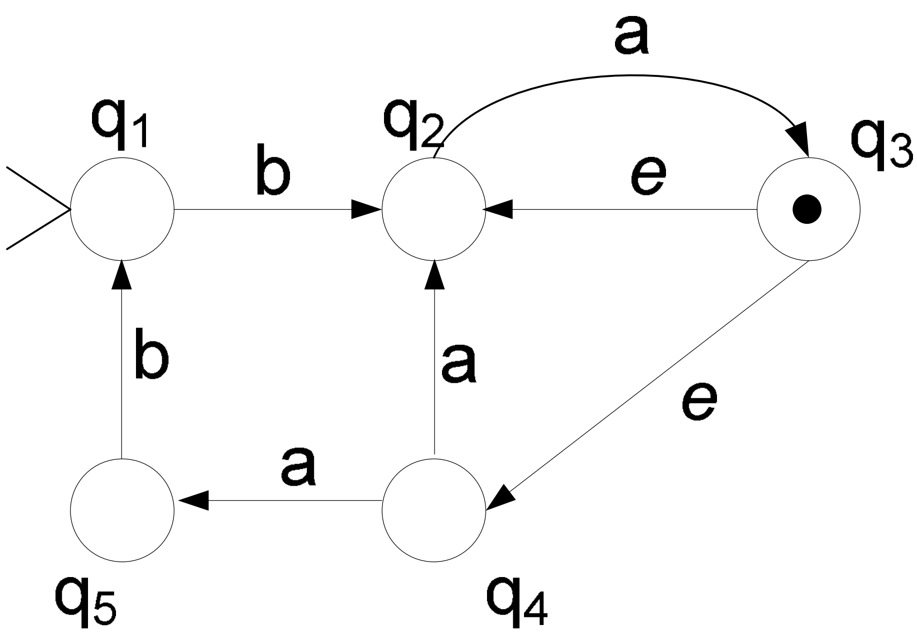


- δ'(R, a) = P → ∃ (r, a, n) ∈ Δ and p ∈ E(n), r ∈ R, p, n ∈ P by the construction of δ'
  - $\delta'(R, a) = \bigcup_{r \epsilon R, (r, a, n) \epsilon \Delta} E(r)$
  - p and n exist but not defined exactly
- (r, a) $|-_M$ (n, *e*) $|-_M^*$ (p, *e*) by the definition of the yield in one step, the definition of the E(n) set, and the previous point
- (q, va) $|-_M^*$ (r, a), (r, a) $|-_M$ (n, *e*) $|-_M^*$ (p, *e*) ↔ (q, va) $|-_M^*$ (p, *e*) by the transitive property of yield
- P ∈ F' ↔ ∃ p ∈ P ∩ F by the construction of F'
- p ∈ P ∩ F → p ∈ F

NFA ↔ DFA

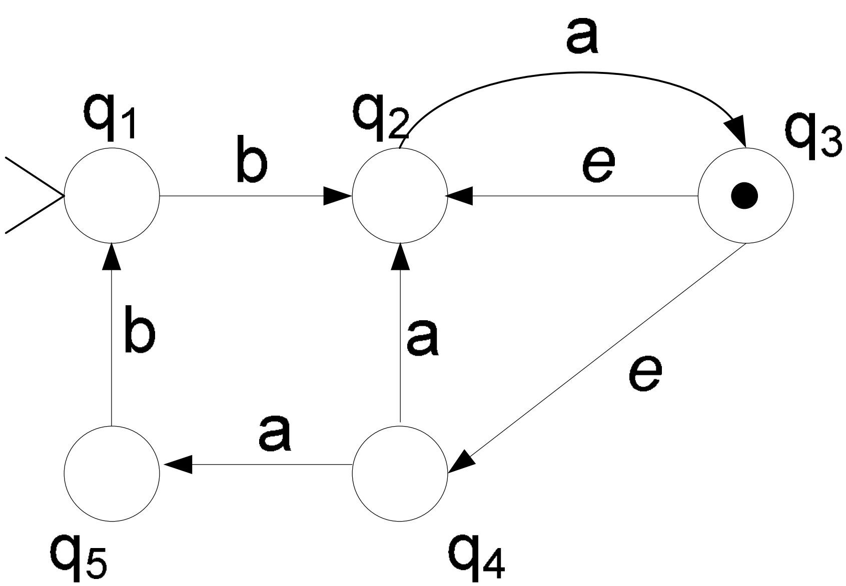
- Theorem: for each NFA M = (K, Σ, Δ, s, F), there is an equivalent DFA M' = (K', Σ', δ', s', F')
- Proof: $w \in \Sigma^*$
  - $w \in L(M) \leftrightarrow (s, w) \vdash_M^* (p, e), p \in F$
    - by definition of acceptance
  - $(s, w) \vdash_M^* (p, e) \leftrightarrow (E(s), w) \vdash_{M'}^* (P, e), p \in P \in F'$
    - by the lemma
  - $(E(s), w) \vdash_{M'}^* (P, e), P \in F' \leftrightarrow w \in L(M')$ by definition of the acceptance

Example

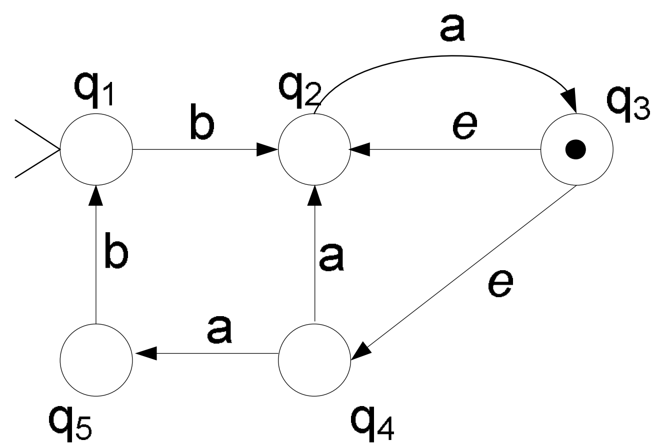


Example

- Defining E sets
  - $E(q_1) = \{q_1\}$
  - $E(q_2) = \{q_2\}$
  - $E(q_3) = \{q_2, q_3, q_4\}$
  - $E(q_4) = \{q_4\}$
  - $E(q_5) = \{q_5\}$
- Initial state
  - $s' = E(q_1) = \{q_1\} = Q_0$

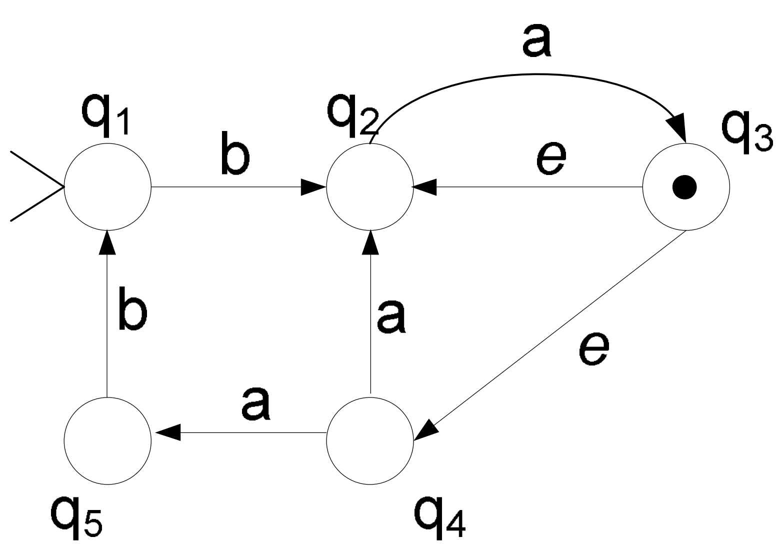


- Defining $\delta'(P, \sigma)$
  - $\delta'(\{q_1\}, a) = \varnothing = Q_1$
  - $\delta'(\{q_1\}, b) = E(q_2) = \{q_2\} = Q_2$
  - $\delta'(\varnothing, a) = \varnothing = Q_1$
  - $\delta'(\varnothing, b) = \varnothing = Q_1$
  - $\delta'(\{q_2\}, a) = E(q_3) = \{q_2, q_3, q_4\} = Q_3$
  - $\delta'(\{q_2\}, b) = \varnothing = Q_1$
  - $\delta'(\{q_2, q_3, q_4\}, a) = E(q_2) \cup E(q_3) \cup E(q_5) = \{q_2, q_3, q_4, q_5\} = Q_4$
  - $\delta'(\{q_2, q_3, q_4\}, b) = \varnothing = Q_1$
  - $\delta'(\{q_2, q_3, q_4, q_5\}, a) = E(q_2) \cup E(q_3) \cup E(q_5) = \{q_2, q_3, q_4, q_5\} = Q_4$
  - $\delta'(\{q_2, q_3, q_4, q_5\}, b) = E(q_1) = \{q_1\} = Q_0$

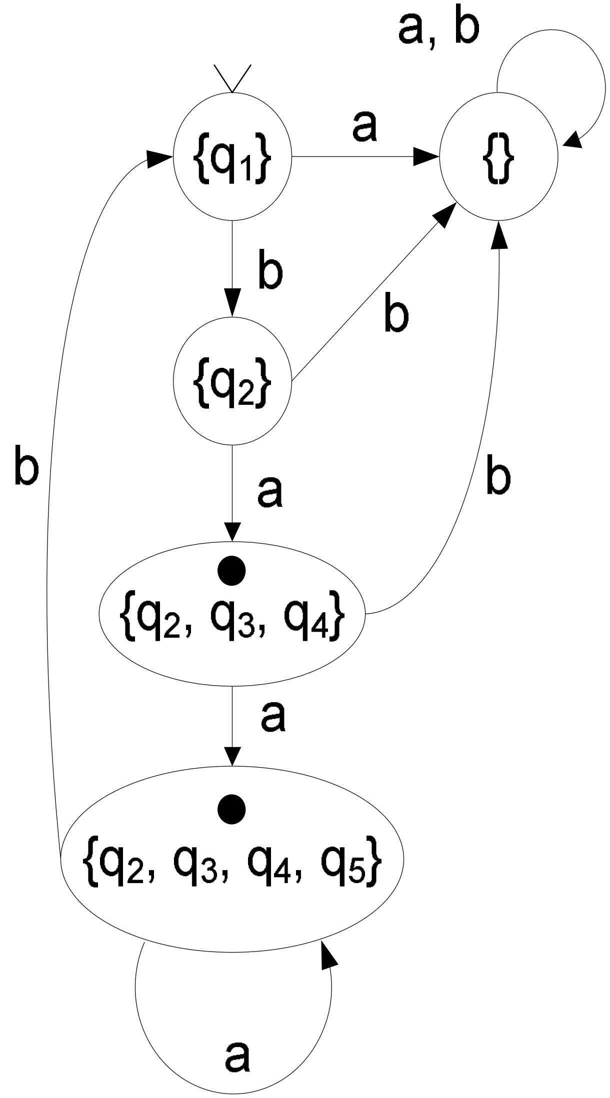


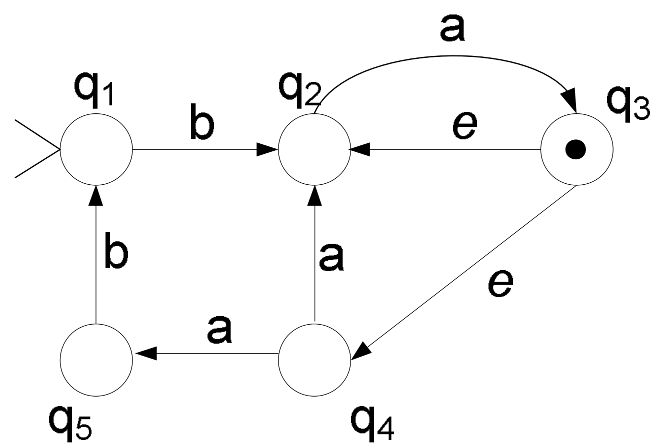
Example

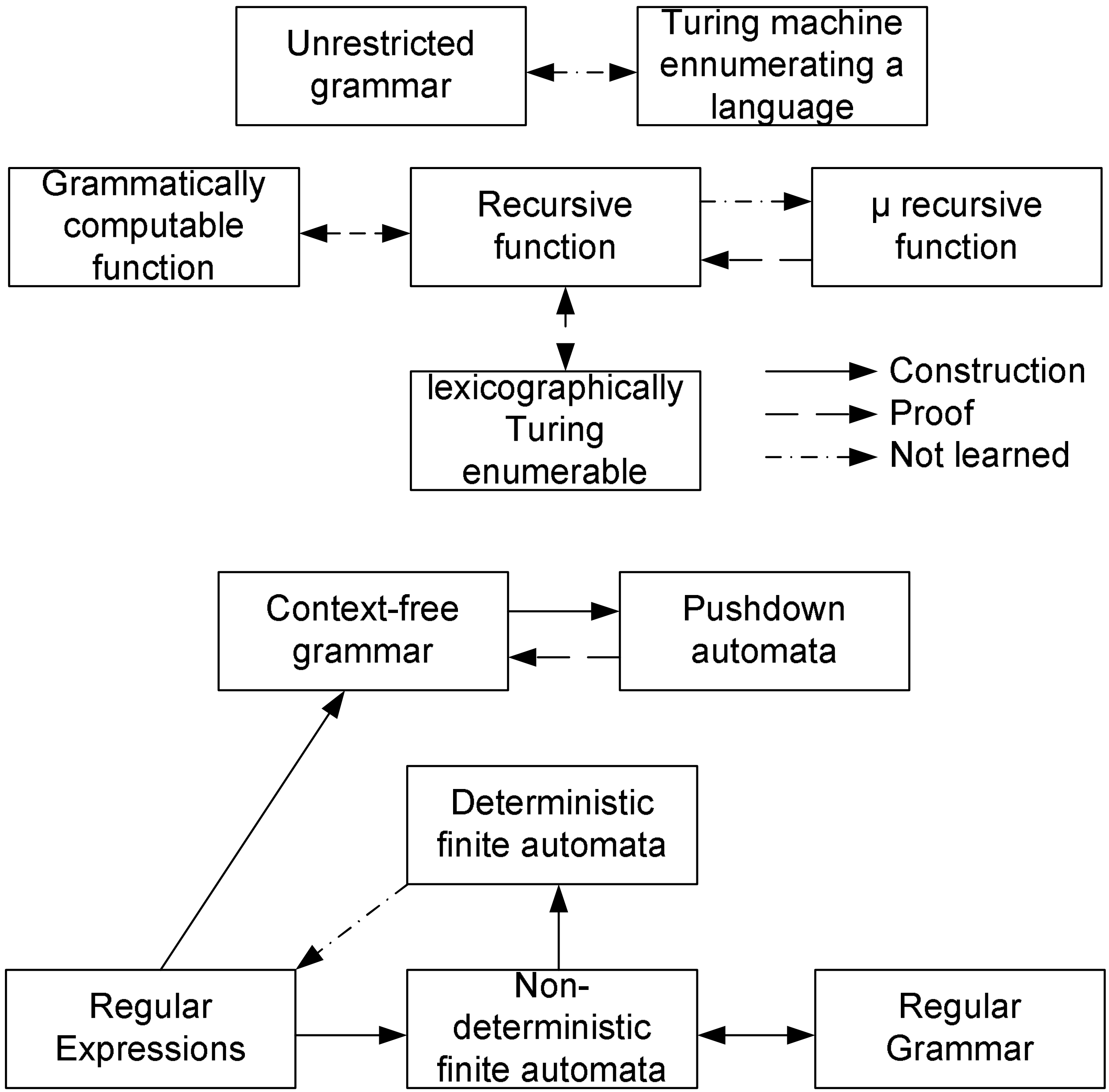
- Defining final states:
  - $F' = \{Q_3, Q_4\} = \{\{q_2, q_3, q_4\}, \{q_2, q_3, q_4, q_5\}\}$
  - the final states contain q_3



Example






Unrestricted grammar
Turing machine ennumerating a language
Grammatically computable function
Recursive function
μ recursive function
lexicographically Turing enumerable
Construction
Proof
Not learned
Context-free grammar
Pushdown automata
Deterministic finite automata
Regular Expressions
Non-deterministic finite automata
Regular Grammar

Summary

- Non-deterministic behavior, NFA
- Difference of DFA and NFA
- Yield in one step, Yield
- String and language accepted by NFA
- Automata equivalence
- DFA $\leftrightarrow$ NFA, construction, proof

Next time

- Non-deterministic finite automata
- Finite automata and regular expressions

Elements of the Theory of Computation

Lesson 6

2.3. Non-deterministic finite automata

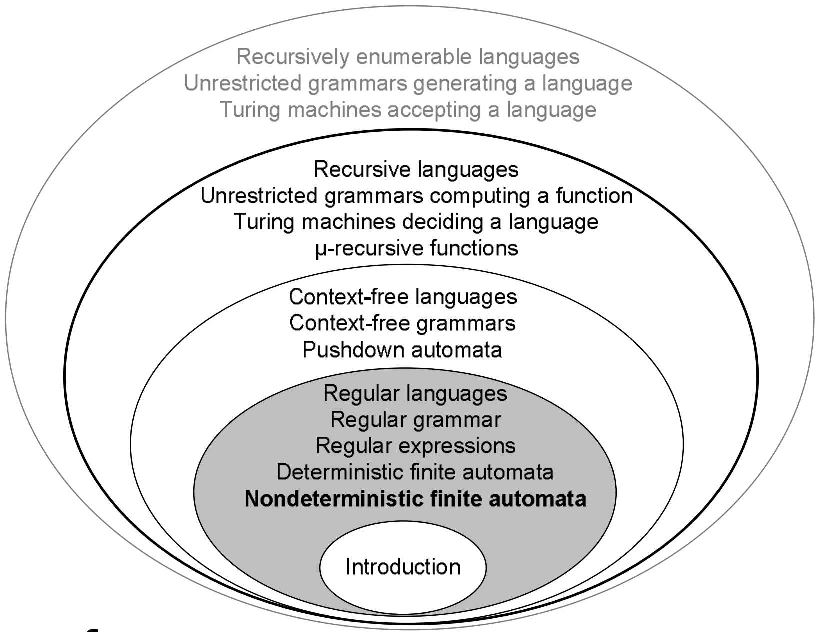
2.4. Finite automata and regular expressions

University of Pannonia

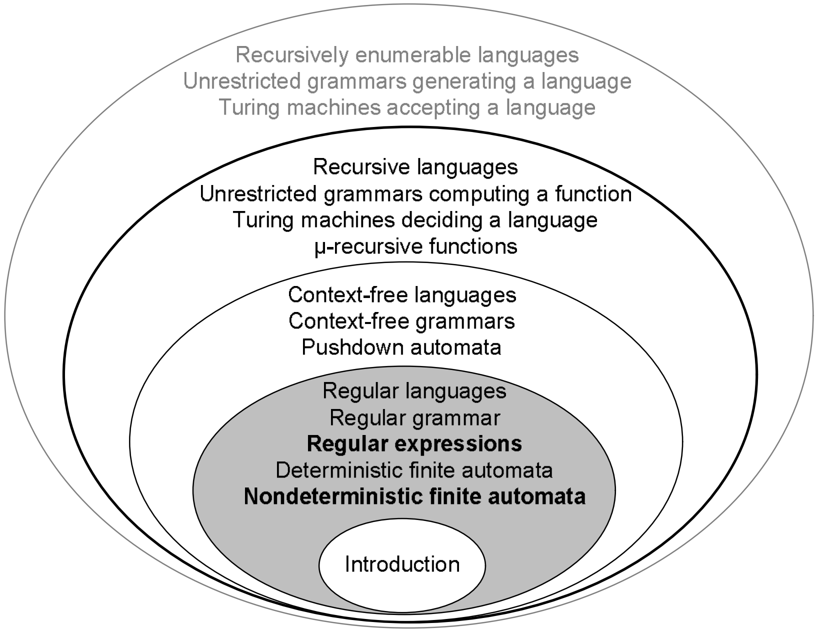
Dr. István Heckl, Istvan.Heckl@gmail.com

Last time

- Non-deterministic behavior
- NFA
- Difference of DFA and NFA
- Yield in one step
- Yield
- String accepted by NFA
- Language accepted by NFA
- Automata equivalence
- DFA ↔ NFA, construction, proof



Finite automata



- RE → NFA
- Closure properties
  - union
  - concatenation
  - Kleene star
  - complementation
  - intersection
- Algorithms for automata
- RE ↔ NFA
- Pumping theorem 1
- Languages that are not regular

RE → NFA

- Some theorems help us to create NFA which is equivalent with some regular expression
  - Thomson's construction
  - regular expressions define regular languages
- The theorems include construction, i.e., they not only prove the existence but provide the required automaton

RE → NFA

- These theorems helps to prove that RE and NFA are equivalent
  - for a given regular expression an NFA always exists which accepts the same language what the regular expression generates
  - for a given NFA always exists a regular expression which generates the same language what the NFA accepts

Thompson

- Kenneth Lane Thompson (1943)
  - American pioneer of computer science
  - his work:
    - the B programming language
    - the C programming language
    - one of the creators and early developers of the Unix and Plan 9 operating systems
    - regular expressions
    - early computer text editors QED and ed

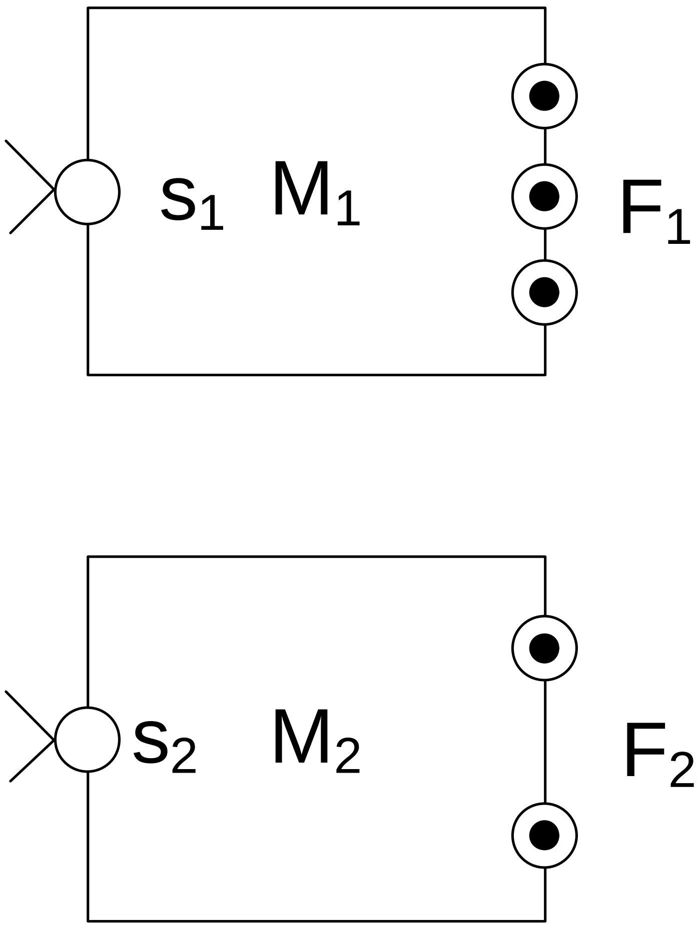
Union

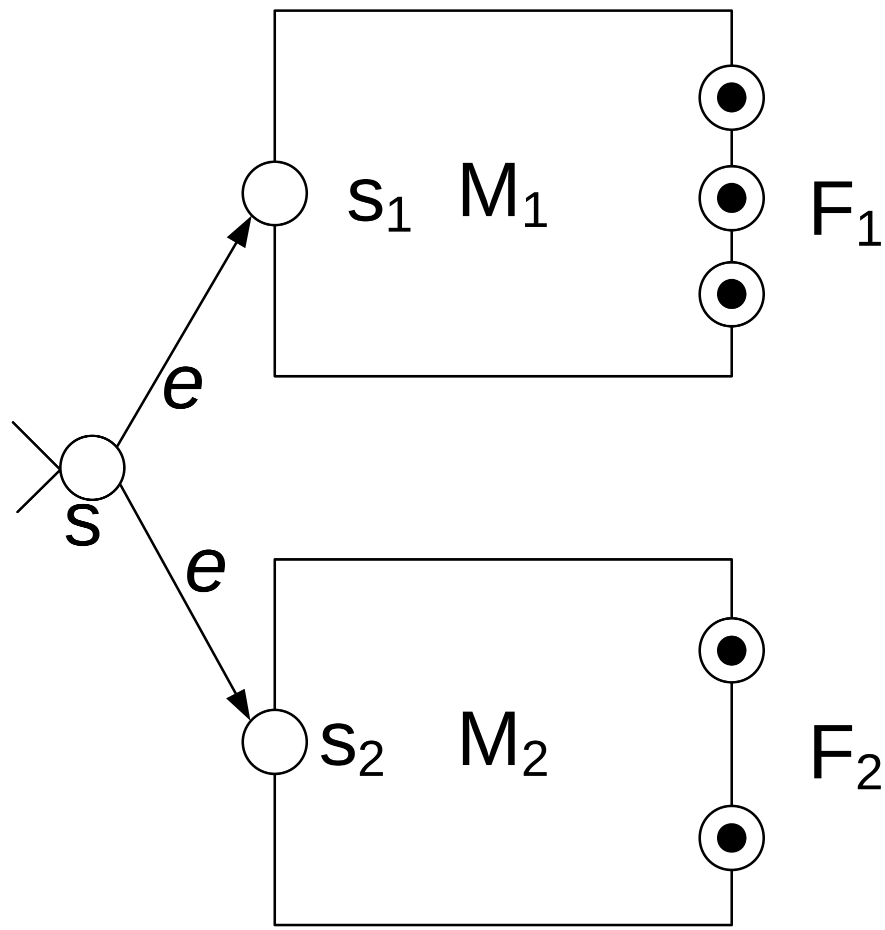
- Theorem: languages accepted by finite automata are closed under union
  - if $L(M_1)$, $L(M_2)$ are languages accepted by finite automata M_1 and M_2 $\rightarrow \exists$ a finite automata M such that $L(M) = L(M_1) \cup L(M_2)$
    - the term closed is used because the new automata M is the same type as M_1, M_2, finite automata
  - the union of two regular languages is also regular

Union

- Comments:
  - M_1, M_2, are NFAs
  - $L(M_1)$, $L(M_2)$ are languages accepted by finite automata M_1 and M_2
  - $L(M_1) \cup L(M_2)$ is also a language
  - $L(M) = L(M_1) \cup L(M_2)$ is accepted by automaton M
  - M is a finite automation (the same type as M_1 and M_2)

Union



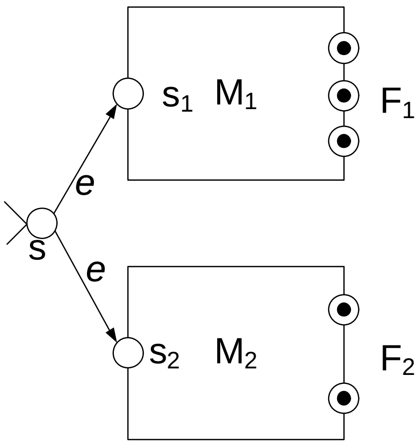


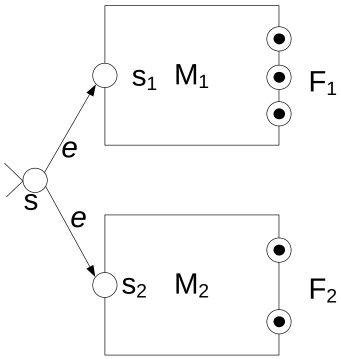
Union

- Construction:
  - NFA M_1, M_2 are known
  - $M_1 = (K_1, \Sigma, \Delta_1, s_1, F_1)$, $M_2 = (K_2, \Sigma, \Delta_2, s_2, F_2)$
    - K_1 and K_2 are disjoint
  - $M = (K, \Sigma, \Delta, s, F)$
    - $K = K_1 \cup K_2 \cup \{s\}$
      - s is a new state not in $K_1 \cup K_2$
    - Σ are the same for the 3 automata
    - $\Delta = \Delta_1 \cup \Delta_2 \cup \{(s, e, s_1), (s, e, s_2)\}$
    - $F = F_1 \cup F_2$

Union

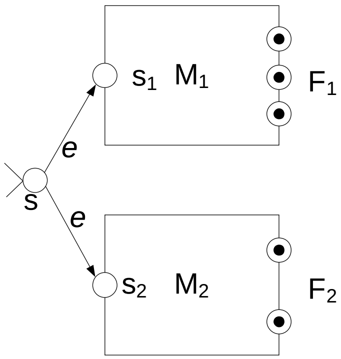
- Proof:
  - suppose $w \in L(M)$
    - $w \in L(M) \rightarrow (s, w) \vdash_M^* (q, e)$, $q \in F$ by the definition of acceptance
    - $(s, w) \vdash_M (s_1, w) \vdash_M^* (q, e)$, $q \in F_1$ or $(s, w) \vdash_M (s_2, w) \vdash_M^* (q, e)$, $q \in F_2$ by the construction of M and the previous point





Union

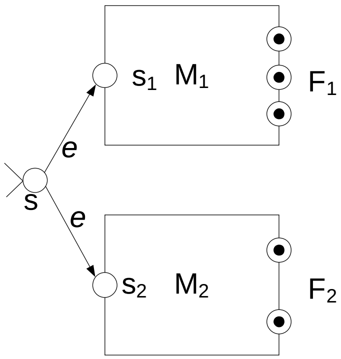
- $(s_1, w) \vdash_M^* (q, e) \rightarrow (s_1, w) \vdash_{M1}^* (q, e), q \in F_1$ by the construction of M
  - $(s_1, w) \vdash_{M1}^* (q, e), q \in F_1 \rightarrow w \in L(M_1)$ by the definition of acceptance
- or $(s_2, w) \vdash_M^* (q, e) \rightarrow (s_2, w) \vdash_{M2}^* (q, e), q \in F_2$ by the construction of M
  - $(s_2, w) \vdash_{M2}^* (q, e), q \in F_2 \rightarrow w \in L(M_2)$ by the definition of acceptance
- $w \in L(M) \rightarrow w \in L(M_1)$ or $w \in L(M_2) \rightarrow$ $L(M) \subseteq L(M_1) \cup L(M_2)$



Union

– suppose $w \in L(M_1) \cup L(M_2)$

- $w \in L(M_1) \cup L(M_2) \rightarrow (s_1, w) \mid\!-_{M1}^* (q, e)$, $q \in F_1$ or $(s_2, w) \mid\!-_{M2}^* (q, e)$, $q \in F_2$ by the definition of acceptance
- $(s_1, w) \mid\!-_{M1}^* (q, e) \rightarrow (s_1, w) \mid\!-_{M}^* (q, e)$, $q \in F_1$ by the construction of M
- $(s_2, w) \mid\!-_{M2}^* (q, e) \rightarrow (s_2, w) \mid\!-_{M}^* (q, e)$, $q \in F_2$ by the construction of M
- $(s, e, s_1), (s, e, s_2) \in \Delta$ by the construction of M



Union

- $(s, e, s_1), (s, e, s_2) \in \Delta \rightarrow (s, w) \vdash_M (s_1, w)$, $(s, w) \vdash_M (s_2, w)$
- $(s, w) \vdash_M (s_1, w), (s_1, w) \vdash_M^* (q, e) \rightarrow$ $(s, w) \vdash_M^* (q, e), q \in F_1$ by the transitivity of $\vdash_M^*$
- $(s, w) \vdash_M (s_2, w), (s_2, w) \vdash_M^* (q, e) \rightarrow$ $(s, w) \vdash_M^* (q, e), q \in F_2$ by the transitivity of $\vdash_M^*$
- $(s, w) \vdash_M^* (q, e), q \in F_1$ or $q \in F_2 \rightarrow w \in L(M)$ by the definition of acceptance
- $w \in L(M_1) \cup L(M_2) \rightarrow w \in L(M) \rightarrow$ $L(M_1) \cup L(M_2) \subseteq L(M)$

– $L(M) \subseteq L(M_1) \cup L(M_2), L(M_1) \cup L(M_2) \subseteq L(M) \rightarrow$ $L(M) = L(M_1) \cup L(M_2)$

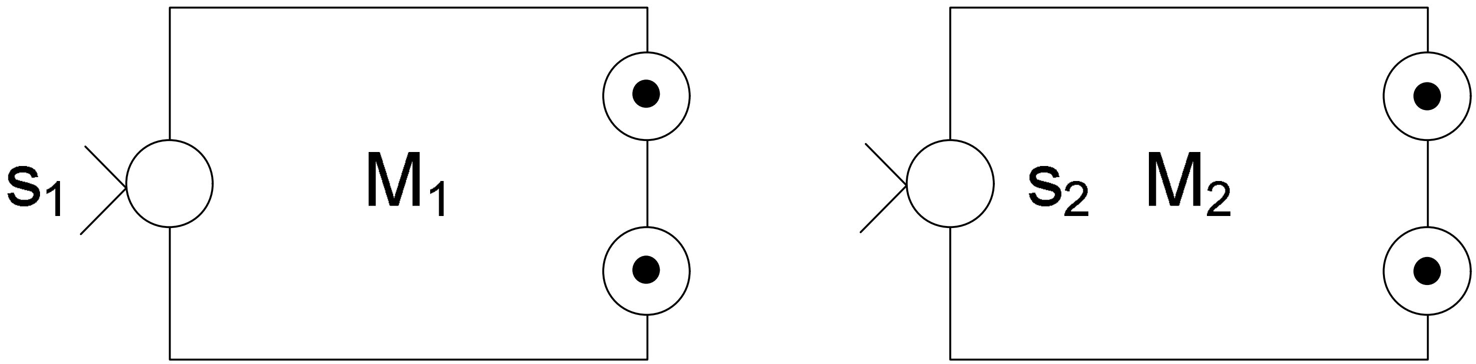
Union

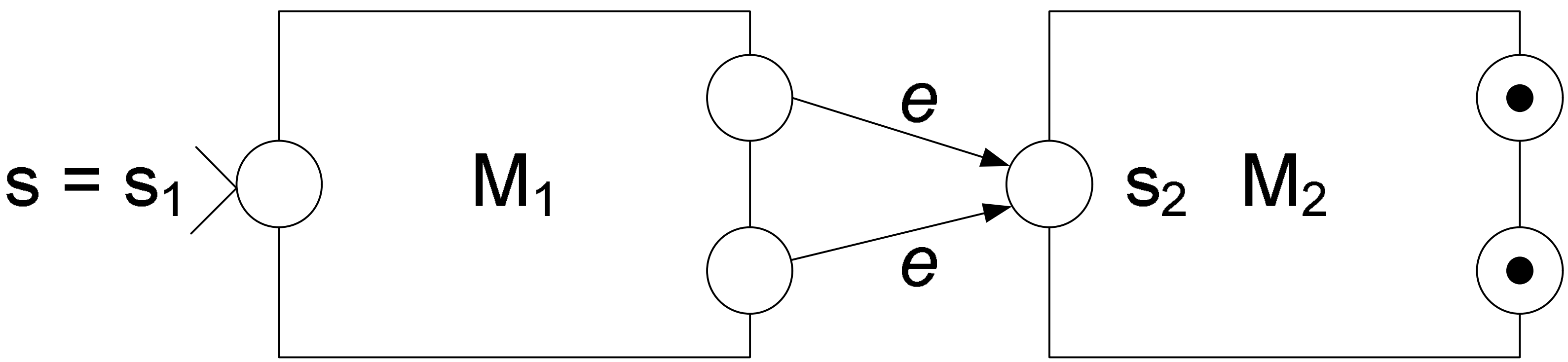
- M uses non-deterministic behavior to guess which direction is correct
- M is finite automaton because we started from two finite automata and added 1 new state and 2 transitions

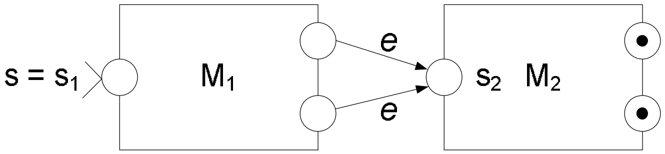
Concatenation

- Theorem: languages accepted by finite automata are closed under concatenation
  - if $L(M_1)$, $L(M_2)$ are languages accepted by finite automata M_1 and M_2 $\rightarrow \exists$ a finite automata M such that $L(M) = L(M_1) \circ L(M_2)$

Concatenation







Concatenation

- Construction:
  - NFA M_1, M_2 are known
  - $M_1 = (K_1, \Sigma, \Delta_1, s_1, F_1)$, $M_2 = (K_2, \Sigma, \Delta_2, s_2, F_2)$
    - K_1 and K_2 are disjoint
  - $M = (K, \Sigma, \Delta, s, F)$
    - $K = K_1 \cup K_2$ (K_1 and K_2 are disjoint)
    - Σ are the same for the 3 automata
    - $\Delta = \Delta_1 \cup \Delta_2 \cup (F_1 \times \{e\} \times \{s_2\})$
    - $s = s_1$
    - $F = F_2$

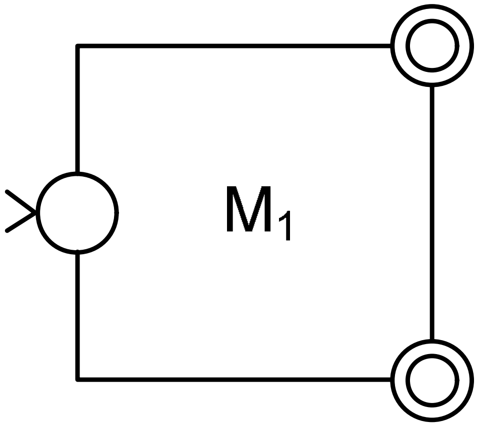
Kleene star

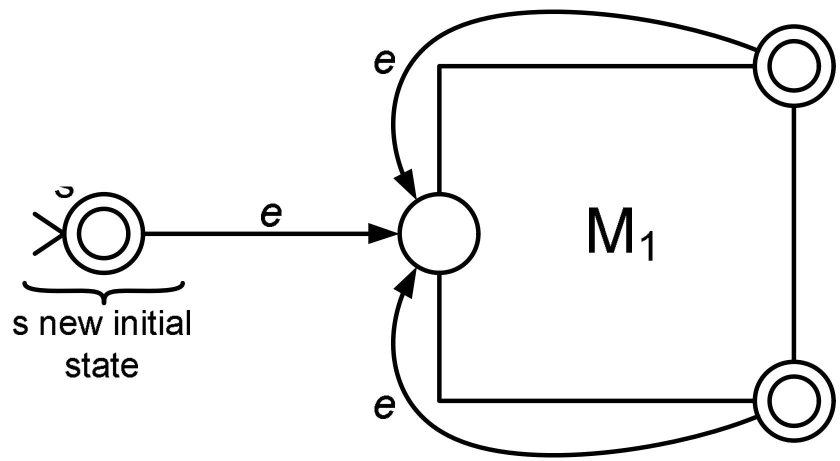
- Stephen Cole Kleene (1909 –1994)
  - American mathematician
  - helped to lay the foundations for theoretical computer science
  - a number of mathematical concepts are named after him:
    - Kleene hierarchy
    - Kleene algebra
    - the Kleene star (Kleene closure)
    - Kleene's recursion theorem
    - Kleene fixpoint theorem

Kleene star

- Theorem: languages accepted by finite automata are closed under Kleene star
  - if $L(M_1)$ is a language accepted by finite automata M_1 $\rightarrow \exists$ a finite automata M such that $L(M) = L(M_1)^*$

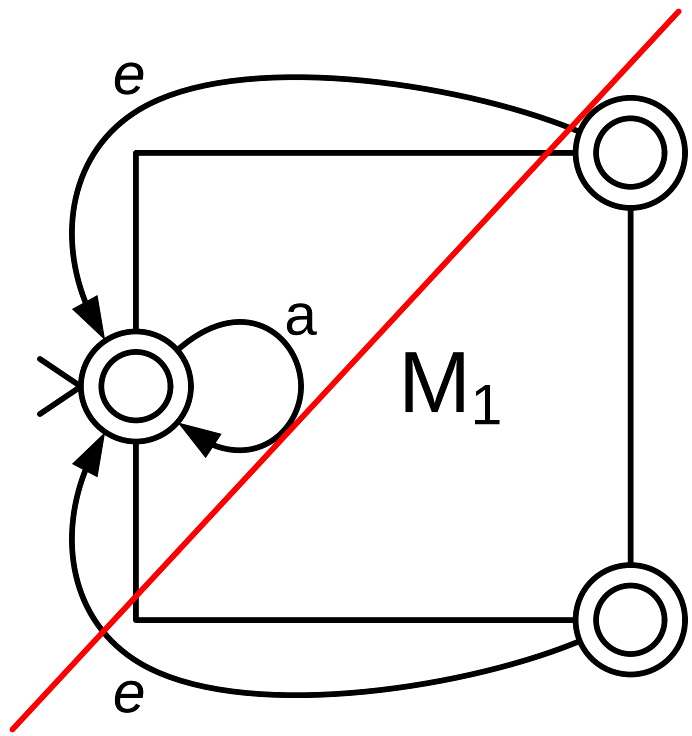
Kleene star



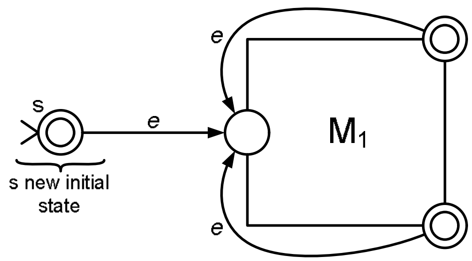


- Starting state must be final because $L(M_1)^*$ contains *e*

Kleene star



- This state diagram is not correct
  - the automaton may accept wrong word if it halts in s_1



Kleene star

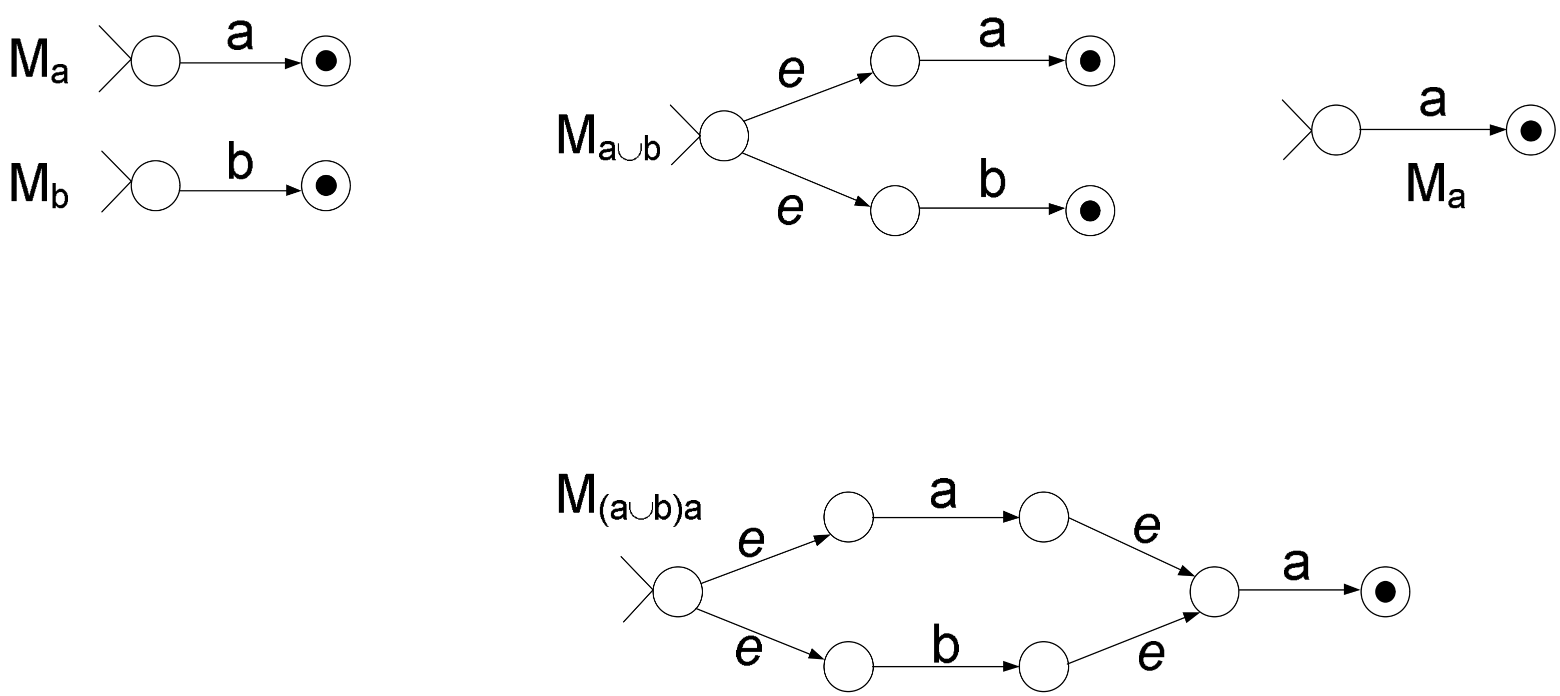
- Construction:
  - NFA M_1 is known
  - $M_1 = (K_1, \Sigma, \Delta_1, s_1, F_1)$
  - $M = (K, \Sigma, \Delta, s, F)$
    - $K = K_1 \cup \{s\}, s \notin K_1$
    - $\Delta = \Delta_1 \cup \{(s, e, s_1)\} \cup (F_1 \times \{e\} \times \{s_1\})$
    - $s = s$
    - $F = F_1 \cup \{s\}$

Example

- Construct NFA M such that $L(M) = (a \cup b)a$!
  - create a basic machine for every $\sigma \in \Sigma$ in the regular expression
  - use the constructions stated before to connect the machines
  - at union the machines should be ordered vertically, at concatenation horizontally

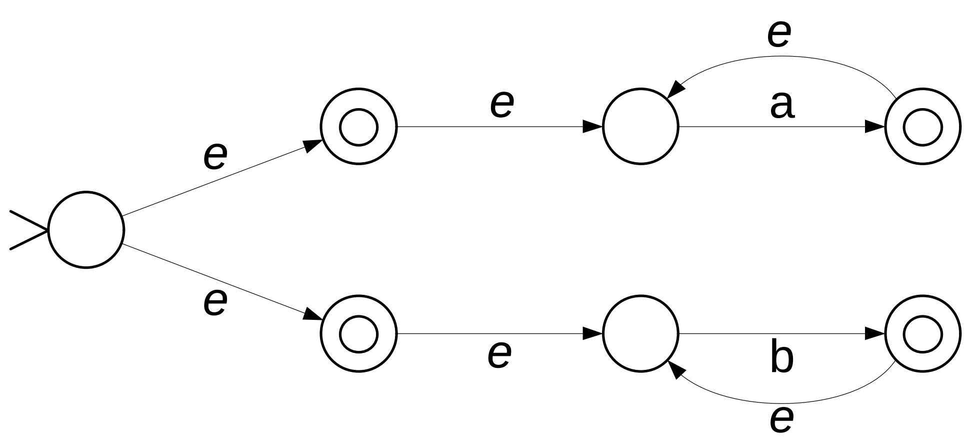
Example

- L(M) = (a ∪ b)a



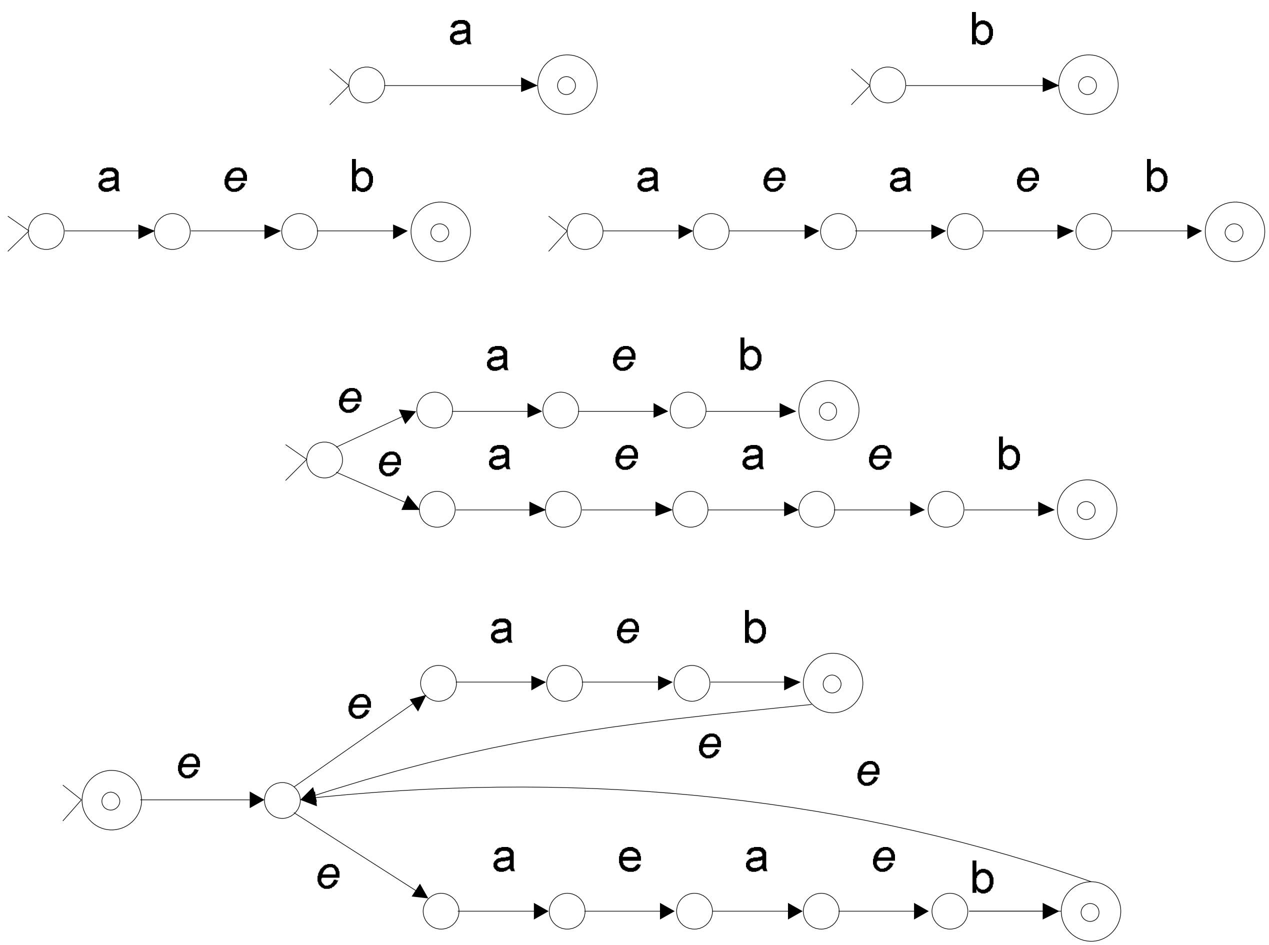
Example

- Construct NFA M such that L(M) = a* ∪ b* !



Example

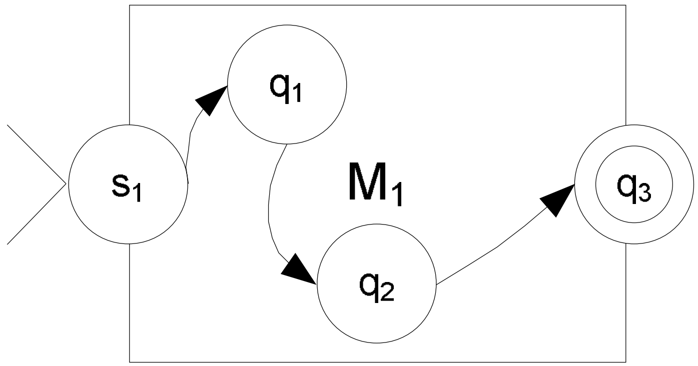
- Construct NFA M such that L(M) = (ab ∪ aab)* !

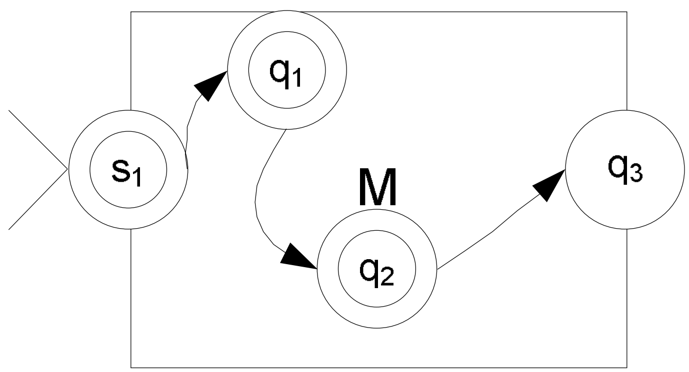


Complementation

- Theorem: languages accepted by finite automata are closed under complementation
  - if $L(M_1)$ is a language accepted by finite automata M_1 $\rightarrow \exists$ a finite automaton M such that $L(M) = \Sigma^* - L(M_1) = L(M_1)^C$

Complementation





Complementation

- Construction:
  - DFA M_1 is known
    - $M_1 = (K_1, \Sigma, \delta_1, s_1, F_1)$
    - if M_1 is not DFA then it must be converted
  - $M = (K, \Sigma, \delta, s, F)$
    - $K = K_1$
    - $\delta = \delta_1$
    - $s = s_1$
    - $F = K_1 - F_1$
- CFG is not closed under complementation ➡

Intersection

- Theorem: languages accepted by finite automata are closed under intersection
  - if $L(M_1)$, $L(M_2)$ are languages accepted by finite automata M_1 and M_2 $\rightarrow \exists$ a finite automata M such that $L(M) = L(M_1) \cap L(M_2)$
    - the intersection of two languages accepted by finite automata can be also accepted by a finite automata

Intersection

- Construction:
  - apply the previous constructions for M_1 and M_2
    - NFA $\rightarrow$ DFA twice
    - complementation theorem twice
    - union once
    - NFA $\rightarrow$ DFA
    - complementation theorem once again

Intersection

- Proof:
  - languages accepted by finite automata are closed under union and complementation
  - intersection can be expressed by these two operation
  - $L(M) = L(M_1) \cap L(M_2) = (L(M_1)^C \cup L(M_2)^C)^C$
    - De'Morgan identity

$L(M) = \Sigma^*$?

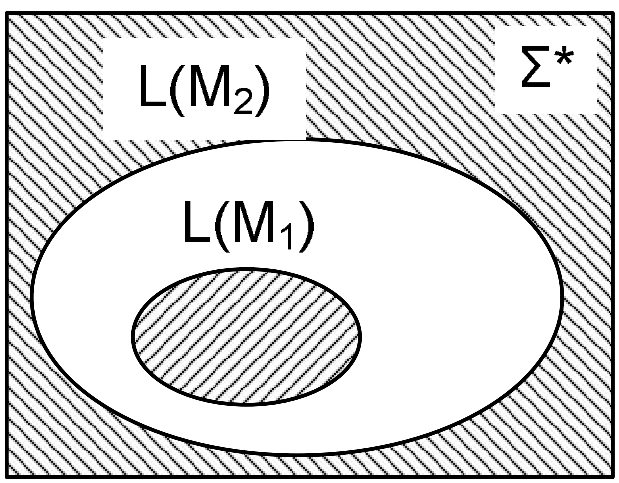
- Theorem: there is an algorithm for deciding if $L(M) = \Sigma^*$
  - finite automaton M accepts each possible string
- Proof:
  - $L(M) = \Sigma^* \leftrightarrow L(M)^C = \emptyset$
    - construct M_1 such that $L(M_1) = L(M)^C$
    - use theorem about complementation
  - $L(M_1) = \emptyset \leftrightarrow$ if there is no directed path from s_1 to any element of F_1 on the state diagram of M_1

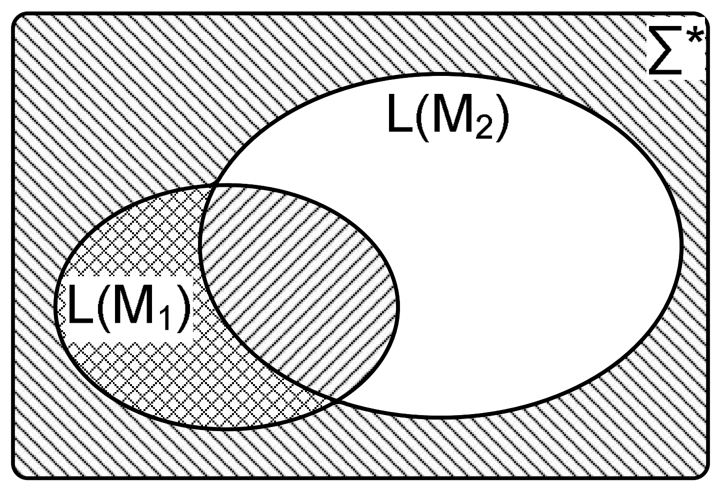
Algorithm for deciding if there is a direct path

```
denote(A: point, N: number of points)
  sign(A)
  for i = 1 to N do
     if isEdge(A, p_i) and !isSigned(p_i) then
          denote(i, N)
end
bool isDirectedPath(A: startPoint,
  B: endPoint, N: number of points)
  denote(A, N)
  if isSigned(B) then return true
  else return false
end
```

$L(M_1) \subseteq L(M_2)$?

- Theorem: there is an algorithm for deciding if $L(M_1) \subseteq L(M_2)$
  - M_1 and M_2 are finite automata
- Proof:
  - $L(M_1) \subseteq L(M_2) \leftrightarrow L(M_1) \cap L(M_2)^C = \varnothing$
  - we know how to check if $L = \varnothing$





$L(M_1) = L(M_2)$?

- Theorem: there is an algorithm for deciding if $L(M_1) = L(M_2)$
  - M_1 and M_2 are finite automata
- Proof:
  - $L(M_1) = L(M_2) \leftrightarrow L(M_1) \subseteq L(M_2)$ and $L(M_2) \subseteq L(M_1)$
    - we know how to check if $L_1 \subseteq L_2$
  - it is an algorithm and not the definition of the equivalence of two automata

RE $\leftrightarrow$ NFA

- Theorem: a language is regular $\leftrightarrow$ it is accepted by a finite automaton
- Proof: $\rightarrow$
  - recall that $\mathfrak{R}$ is the set of regular languages
    - $\emptyset \in \mathfrak{R}$, $\{a\} \in \mathfrak{R} \; \forall \; a \in \Sigma$
    - if $A, B \in \mathfrak{R} \rightarrow A \cup B \in \mathfrak{R}$, $A \circ B \in \mathfrak{R}$, $A^* \in \mathfrak{R}$
    - $\mathfrak{R}$ is minimal
  - there are finite automata to accept the empty set and the singleton languages
  - languages accepted by finite automata are closed under union, concatenation, and Kleene star

RE ↔ NFA

- Proof: ←
  - for each NFA an equivalent RE can be constructed, it is not proved here

Pumping theorem 1

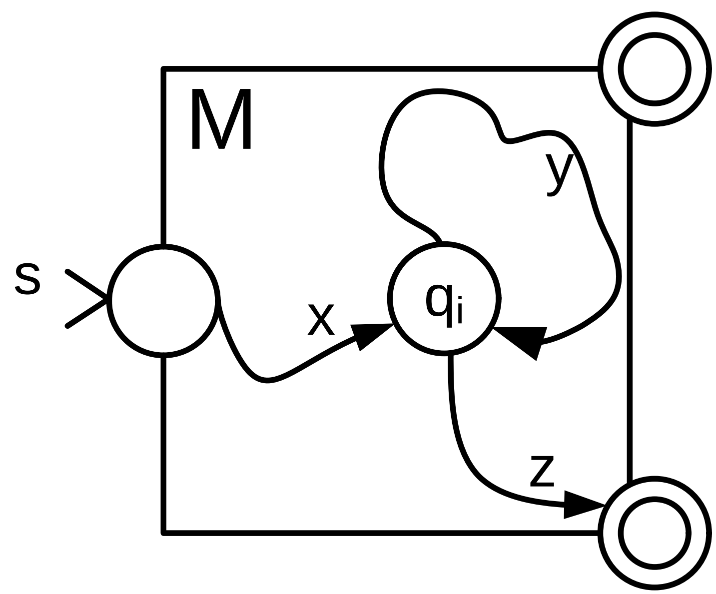
- Theorem: let M(K, Σ, δ, s, F) is a DFA, long enough words in L(M) ($|w| \geq |K|$) has a form, w = xyz, y ≠ *e*, such that $xy^nz \in L(M)$, $\forall\ n \geq 0$
- Proof:
  - idea: if L(M) is infinite then the state diagram of M must contain a loop

Pumping theorem 1

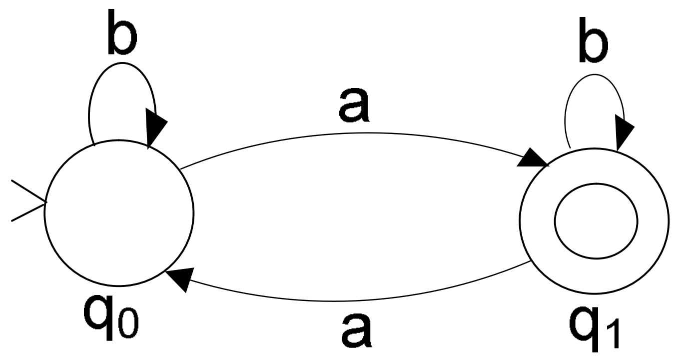
- let $w \in L(M)$ such that $|w| = k \geq |K|$
  - w exists since L is infinite
  - $w = \sigma_1\sigma_2\ldots\sigma_k$
- $(q_0, \sigma_1\sigma_2\ldots\sigma_k)$ |- $(q_1, \sigma_2\ldots\sigma_k)$ |- ... |- (q_{k-1}, σ_k) |- (q_k, e)
  - $q_0 = s$, $q_k \in F$
  - the number of yield in one step is k
- since $k \geq |K|$, $\exists\ q_i, q_j$, such that $q_i = q_j$, $i \neq j$, $(i<j)$

Pumping theorem 1

- $\sigma_{i+1}\sigma_{i+2} \ldots\sigma_j$ string moves M from state q_i to state q_j
- $\sigma_{i+1}\sigma_{i+2} \ldots\sigma_j$ can be removed or repeated without affecting the acceptance of w
- $\sigma_1\sigma_2\ldots\sigma_i(\sigma_{i+1}\sigma_{i+2} \ldots\sigma_j)^n\sigma_{j+1}\ldots\sigma_k \in L(M)$ for $n \geq 0$
  - $x = \sigma_1\sigma_2\ldots\sigma_i$
  - $y = \sigma_{i+1}\sigma_{i+2} \ldots\sigma_j$
  - $z = \sigma_{j+1}\ldots\sigma_k$



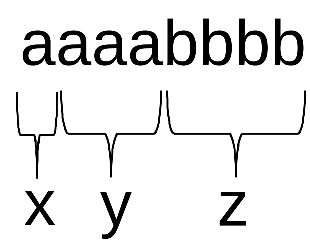
Example



- L(M) = {w $\in$ (ab)*: #a odd in w}
  - |K| = 2
  - w = bbabaa is long enough
  - the previous theorem does not tell how to construct x, y, z, it states only their existence
  - let us say the revisited node is q_0
    - x = b, y = baba, z = a are valid strings
      - xy^0z = ba $\in$ L(M), xy^1z = bbabaa $\in$ L(M), xy^2z = bbabababaa $\in$ L(M), ...
    - x = *e*, y = bbaba, z = a are valid strings too
  - we can say that the revisited node is q_1

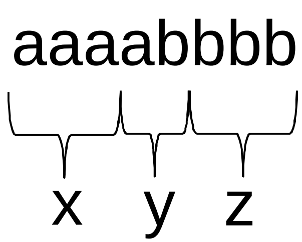
Languages that are not regular

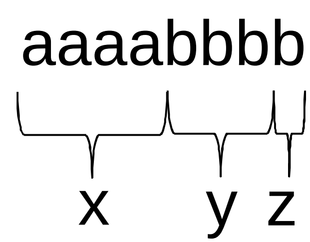
- Theorem: $L = \{a^n b^n : n \geq 0\}$ is not regular
- Proof by indirection:
  - assume that L is regular and apply the pumping theorem for a long enough string $a^k b^k$, where k is a fix number
  - $xy^n z = a^k b^k$
  - $x = a^{n1}$, $y = a^{n2}$, $z = b^{n3}$
    - n1, n2, n3 $\in$ N are fix numbers
    - $xy^n z = a^{n1} a^{n \cdot n2} b^{n3}$
    - $n1 + n \cdot n2 = n3$ for $\forall$ n, contradiction



Languages that are not regular

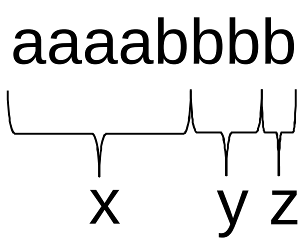
- $x = a^{n1}$, $y = a^{n2}b^{n3}$, $z = b^{n4}$
  - $xy^nz = a^{n1}(a^{n2}b^{n3})^nb^{n4} \notin L$ as b precedes 'a' if $n > 1$
- $x = a^{n1}$, $y = b^{n2}$, $z = b^{n3}$
  - $xy^nz = a^{n1}b^{n \cdot n2}b^{n3}$
  - $n1 = n \cdot n2 + n3$ for $\forall n$, contradiction

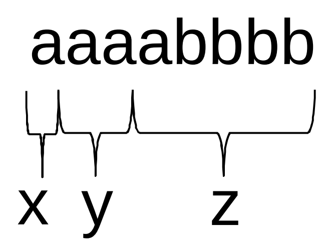




Languages that are not regular

- $x = a^{n1}b^{n2}$, $y = b^{n3}$, $z = b^{n4}$
  - $xy^nz = a^{n1}b^{n2}b^{n\cdot n3+n4}$
  - $n1 + n\cdot n2 + n3 = n4$ for $\forall n$, contradiction
- $x = a^{n1}$, $y = a^{n2}$, $z = a^{n3}b^{n4}$
  - $xy^nz = a^{n1}a^{n\cdot n2+n3}b^{n4}$
  - $n1 + n\cdot n2 + n3 = n4$ for $\forall n$, contradiction
- L is not regular because with finite state we cannot keep in mind the number of 'a' symbols if this value has no upper limit





Languages that are not regular

- Theorem: L = {a^n : n is prime} is not regular
- Proof by indirection:
  - assume that L is regular and apply the pumping theorem for a long enough string a^k, where k is a fix number

Languages that are not regular

- $xy^nz = a^k$
  - $x = a^p$, $y = a^q$, $z = a^r$ for some $p, q, r \in N$, $q \neq 0$
  - $p+nq+r$ is prime for $\forall$ n
    - let $n = p+2q+r+2$
    - $p+nq+r = p+(p+2q+r+2)q+r = p+pq+2qq+rq+2q+r = (q+1)(p+2q+r)$, contradiction
- L is not regular because there is no simple periodicity in the set of prime numbers

Summary

- RE → NFA
- Closure properties
- Algorithms for automata
- Pumping theorem 1
- Languages that are not regular

Next time

- Context-free grammars

Elements of the Theory of Computation

Lesson 7

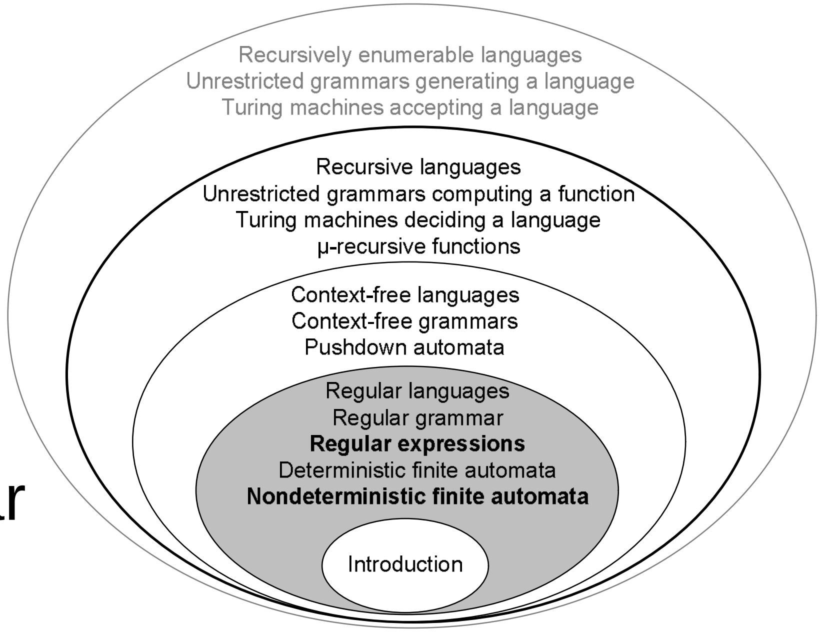
3.1. Context-free grammars

University of Pannonia

Dr. István Heckl, Istvan.Heckl@gmail.com

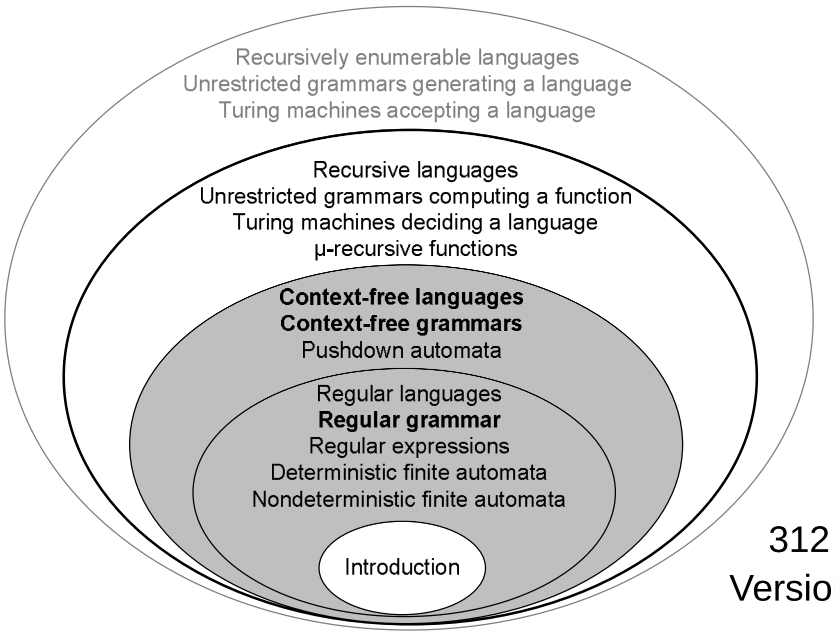
Last time

- RE → NFA
- Closure properties
- Algorithms for automata
- Pumping theorem 1
- Languages that are not regular



Context-free grammars

- Context-free languages
- Context-free grammars
- Derivation
- Language generated by CFG
- Regular grammars
- NFA ↔ RG



Context-free languages

- Language recognizer
  – a device that accepts valid strings
  – e.g.: NFA, DFA
- Language generator
  – a device that are capable of producing valid strings
  – e.g.: regular expressions

Regular expressions

- Regular expressions can be viewed as a language generator
  - $RE_1 = a(a^* \cup b^*)b$
    - first output 'a'
    - then output a number of 'a' or output a number of b
    - finally output b

Context-free grammars

- There are more complex sorts of language generators, called context-free grammars (CFG)
- They apply rules to generate a string
  – it's not completely determined which rule to use
- Let us generate the same language as before with CFG
  – $RE_1 = a(a^* \cup b^*)b$

$RE_1 = a(a^* \cup b^*)b$

Context-free grammar

- Introduce new symbols
  - S: a string in the language
  - M: middle part of the string
  - A: a number of consecutive 'a'
  - B: a number of consecutive b

$RE_1 = a(a^* \cup b^*)b$

Context-free grammar

- Introduce rules to express the meaning of the new symbols
  - $S \rightarrow aMb$
    - where $\rightarrow$ is read as "can be"
    - this rule says that a string in the language starts with 'a' then comes a middle part and ends with b
  - $M \rightarrow A$
  - $M \rightarrow B$
    - the middle part can be a number of consecutive 'a' or 'b'
    - the or relation is expressed with two rules

$RE_1 = a(a^* \cup b^*)b$

Context-free grammar

- $A \rightarrow e$
  - a number of consecutive 'a' can be *e*
- $A \rightarrow aA$
  - a number of consecutive 'a' can be 1 'a' followed by a number of consecutive 'a'
- $B \rightarrow e$
- $B \rightarrow bB$

$RE_1 = a(a^* \cup b^*)b$

Context-free grammar

- It is easy to see that RE_1 and the newly introduced CFG generates the same language
- Algorithm for generating a string with a CFG:

```
start with the string S
while the string contains new symbols
    select a new symbol
    select a corresponding rule (the left
      side of the rule = new symbol)
    replace the new symbol with the right
      side of the rule
```

$RE_1 = a(a^* \cup b^*)b$

Example

- To generate the string aaab
  - start with S
  - apply the rules
    - S → aMb resulting in aMb
    - M → A resulting in aAb
    - A → aA resulting in aaAb
    - A → aA resulting in aaaAb
    - A → *e* resulting in aaab

$RE_1 = a(a^* \cup b^*)b$

Context-free grammar

- Consider the string aaAb, which was an intermediate stage in the generation of aaab
  - we call the strings aa and b, which surround the symbol A, the context of A in this particular string
    - 'a' and *e* are also the context of A
  - the rule A → aA says that we can replace A by the string aA no matter what is the context of A
    - that is why the current grammar is called context free
    - example for a rule which cannot be in CFG: aaAb → abA, SaS → bbA

Context-free grammar

- Definition of context-free grammar, G: a quadruple (V, Σ, R, S) where
  - V an alphabet
  - $\Sigma \subseteq V$ the set of terminals
    - a string of a language contains only terminals
    - $V-\Sigma$ is the set of non-terminals (the new symbols)
  - $R \subseteq (V-\Sigma) \times V^*$ set of rules
    - the left side of a rule is always a single non-terminal
    - we can write a rule $(A, u) \in R$ in the next form $A \rightarrow_G u$
  - $S \in V-\Sigma$ start symbol

Derivation

- Definition of the one step derivation, $=>_G$:
  - $u = xAz$, $v = xyz$, $x, y, z \in V^*$, $A \in V-\Sigma$
  - if $A \rightarrow_G y \in R \rightarrow xAz =>_G xyz$
- When the grammar to which we refer to is obvious, we can write $A \rightarrow w$ and $xAz => xyz$ instead of $A \rightarrow_G w$ and $xAz =>_G xyz$
- Definition of derivation, $=>_G^*$: the reflexive, transitive closure of $=>_G$

Derivation

- We call the sequence $w_0 =>_G w_1 =>_G \dots =>_G w_n$ a derivation of w_n from w_0 in G
  - $w_0, w_1, \dots w_n \in V^*$
    - we term w_i as a partially defined string because it can contain a non-terminal
  - if the derivation has exactly $n \in N$ steps then it can be emphasized as $w_0 =>^n w_n$
- E.g.: S => aMb => aAb => aaAb => aaaAb => aaab
  - see the CFG introduced previously

Language generated by CFG

- Definition of language generated by CFG G, L(G): the set of strings generated by G
  - $L(G) = \{w \in \Sigma^* : S \Rightarrow_G^* w\}$
- Definition of context-free language L: $\exists$ context-free grammar G such that L = L(G)
  - nota bene: language and grammar are different concepts

Example

- Give grammar G(V, Σ, R, S) such that $L(G) = \{a^n b^n : n \geq 0\}$!
  - V = {S, a, b}
  - Σ = {a, b}
  - R = {S → aSb, S → *e*}
    - S → aSb | *e* is a shorthand for the two rules above
- A possible derivation is

  S => aSb => aaSbb => aabb

  - the first two steps used the rule S → aSb and the last used the rule S → *e*

Example

- Which words can derived at most in 4 steps with G = {V, Σ, R, S} grammar from S ?
  - V = {a, b, A, B, S}
  - Σ = {a, b}
  - R = {S → A, S → abA, S → aB, A → a, B → Sb}
- Solution:
  - S $=>_G$ A $=>_G$ a
  - S $=>_G$ aB $=>_G$ aSb $=>_G$ aAb $=>_G$ aab
  - S $=>_G$ abA $=>_G$ aba
  - S $=>_G$ aB $=>_G$ aSb $=>_G$ aabAb $=>_G$ aabab

Example

- Create a (partial) grammar for the English language
  - $V = \{S, A, N, V, P\} \cup \Sigma$
    - S stands for sentence, A for adjective, N for noun, V for verb, and P for phrase
  - Σ = {Jim, big, green, cheese, ate}
    - beware: here the elements of Σ are strings
  - R = {P → N | AP, S → PVP, A → big | green, N → cheese | Jim, V → ate}

Example

- The following are some strings in L(G)
  - Jim ate cheese
  - big Jim ate green cheese
  - big cheese ate Jim
- Unfortunately, these are also strings in L(G)

  - big cheese ate green green big green big cheese

  - green Jim ate green big Jim

Example

- Create grammar G which can generate mathematical statements such as (id*id+id)*(id+id)!
  - id stands for any identifier such as variable name, reserved words of the language, or numerical constants

Example

- G(V, Σ, R, E), where
  - V = {+ ,* , (,), id, T, F, E}
    - E - expression, T - term, F - factor
  - Σ = {+ , *, (,), id}
  - R = {E → E + T, (R1)

    E → T, (R2)

    T → T * F, (R3)

    T → F, (R4)

    F → (E), (R5)

    F → id} (R6)

Example

- Generation of (id*id + id) * (id + id)
  - the course compilers helps to determine which rule should be used

E => T	by Rule R2
=> T * F	by Rule R3
=> T * (E)	by Rule R5
=> T * (E + T)	by Rule R1
=> T * (T + T)	by Rule R2
=> T * (F + T)	by Rule R4
=> T * (id + T)	by Rule R6
=> T * (id + F)	by Rule R4

Example

=> T * (id + id)	by Rule R6
=> F * (id + id)	by Rule R4
=> (E) * (id + id)	by Rule R5
=> (E + T) * (id + id)	by Rule R1
=> (E + F) * (id + id)	by Rule R4
=> (E + id) * (id + id)	by Rule R6
=> (T + id) * (id + id)	by Rule R2
=> (T*F + id) * (id + id)	by Rule R3
=> (F*F + id) * (id + id)	by Rule R4
=> (F*id + id) * (id + id)	by Rule R6
=> (id*id + id) * (id + id)	by Rule R6

Regular grammars

- Definition of regular grammars, RG: such a CFG for which $R \subseteq (V\text{-}\Sigma) \times \Sigma^*((V\text{-}\Sigma) \cup \{e\})$
  - there can be at most one non-terminal at right side of a rule, if there is, it must be at the right end
  - R is reduced from $(V\text{-}\Sigma) \times V^*$
- Example: $G = (V, \Sigma, R, S)$ is RG
  - $V = \{S, A, B, a, b\}$
  - $\Sigma = \{a, b\}$
  - $R = \{S \rightarrow bA \mid aB \mid e, A \rightarrow abaS, B \rightarrow babS\}$

NFA $\leftrightarrow$ RG

- Theorem: a language is regular $\leftrightarrow$ it can be created by a RG
- Construction: $\rightarrow$
  - suppose that L is regular
  - L is accepted by some NFA $M(K, \Sigma, \Delta, s, F)$
  - construct RG $G(V, \Sigma, R, S)$ such that $L(M) = L(G)$
    - $V = K \cup \Sigma$
      - K will be the non-terminals of G
    - $R = \{q \rightarrow xp : (q, x, p) \in \Delta \} \cup \{q \rightarrow e : q \in F\}$
      - for each transition from q to p on input $x \in \Sigma^*$ we have in R the rule $q \rightarrow xp$
    - $S = s$

NFA $\leftrightarrow$ RG

- Proof:
  - $\forall$ w $\in$ L(M) $\leftrightarrow$ (s, w) |-* (p, *e*), p $\in$ F by the definition of acceptance
  - (s, w) |-* (p, *e*), p $\in$ F $\leftrightarrow$ (p_0, $w_1 w_2 \ldots w_n$) |- (p_1, $w_2 \ldots w_n$) |- … |- (p_n, *e*), $p_n \in$ F by the definition of the yield
    - w = $w_1 w_2 \ldots w_n$, p_0, ... $p_n \in$ K, p_0 = s, p_n = p
  - (p_0, $w_1 w_2 \ldots w_n$) |- (p_1, $w_2 \ldots w_n$), (p_1, $w_2 \ldots w_n$) |- (p_2, $w_3 \ldots w_n$), … $\leftrightarrow$ $\exists$ transitions (p_0, w_1, p_1), (p_1, w_2, p_2), ... $\in \Delta$ by the definition of the yield in one step

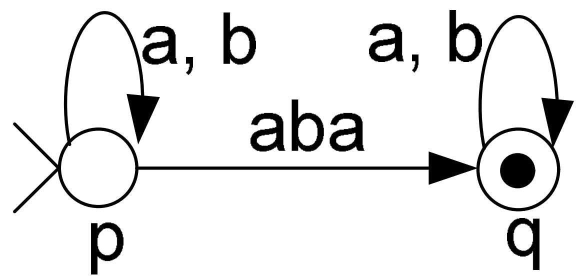
Exam: P3

NFA $\leftrightarrow$ RG

- $\exists$ $(p_0, w_1, p_1), (p_1, w_2, p_2), \ldots \in \Delta \leftrightarrow \exists$ rules: $p_0 \rightarrow w_1p_1$, $p_1 \rightarrow w_2p_2$, ... by the construction of G
- $\exists$ rules: $p_0 \rightarrow w_1p_1$, $p_1 \rightarrow w_2p_2$, ... $\leftrightarrow$ $p_0 \Rightarrow w_1p_1 \Rightarrow w_1w_2p_2 \Rightarrow \ldots \Rightarrow w_1w_2\ldots w_np_n \leftrightarrow s \Rightarrow^* wp_n$ by the definition of the one step derivation and the transitivity of yield
  - $p_0, \ldots p_n \in V\text{-}\Sigma$
- $p_n \in F \leftrightarrow$ $\exists$ rule $p_n \rightarrow e$ by the construction of G
- $s \Rightarrow^* wp_n$, $\exists$ rule $p_n \rightarrow e$ $\leftrightarrow$ $s \Rightarrow^* w$ by the transitive property of yield
- $s \Rightarrow^* w \leftrightarrow w \in L(G)$ by the definition of acceptance

Example

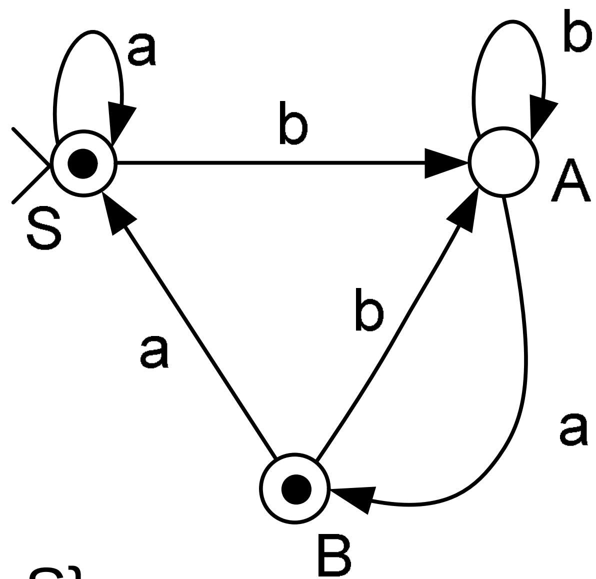
- Construct such a RG G = (V, Σ, R, S) which is equivalent with the given NFA!



  - V = {a, b, P, Q}
  - Σ = {a, b}
  - R = {P → aP, P → bP,
    P → abaQ, Q → aQ, Q → bQ, Q → *e*}
  - S = P
- Give the computation and derivation for w = ababb !
  - (p, ababb) |- (q, bb) |- (q, b) |- (q, *e*)
  - S = P => abaQ => ababQ => ababbQ => ababb

Example

- Construct such a RG G = (V, Σ, R, S) which is equivalent with the given NFA!



  - V = {a, b, A, B, S}
  - R = {S → aS | bA, A → aB | bA, B → aS | bA, B → *e*, S → *e*}

NFA $\leftrightarrow$ RG

- Construction: $\leftarrow$
  - suppose L is generated by some RG G(V, Σ, R, S)
  - construct NFA M(K, Σ, Δ, s, F) such that L(M) = L(G)
    - $K = (V-\Sigma) \cup \{f\}$, where $f \notin V$
    - $\Delta = \{(A, w, B) : A \rightarrow wB \in R, A, B \in V-\Sigma, w \in \Sigma^*\} \cup \{(A, w, f) : A \rightarrow w \in R, A \in V-\Sigma, w \in \Sigma^*\}$
    - $s = S$
    - $F = \{f\}$

NFA $\leftrightarrow$ RG

- Proof:
  - $\forall$ w $\in$ L(G) $\leftrightarrow$ S =>* w by the definition of acceptance
  - S =>* w $\leftrightarrow$ $A_1 \Rightarrow w_1A_2 \Rightarrow w_1w_2A_3 \Rightarrow \ldots \Rightarrow w_1w_2\ldots w_{n-1}A_n \Rightarrow w_1w_2\ldots w_n$ by the definition of yield
    - if w can be reached in n steps then these steps can be written down one by one
    - $A_1, \ldots, A_n \in V\text{-}\Sigma$, $A_1 = S$, $w = w_1w_2\ldots w_n$
  - $A_1 \Rightarrow w_1A_2 \Rightarrow w_1w_2A_3 \Rightarrow \ldots \Rightarrow w_1w_2\ldots w_{n-1}A_n \Rightarrow w_1w_2\ldots w_n \leftrightarrow \exists\, A_1 \rightarrow w_1A_2,\ A_2 \rightarrow w_2A_3, \ldots \in R$ by the definition of the one step derivation

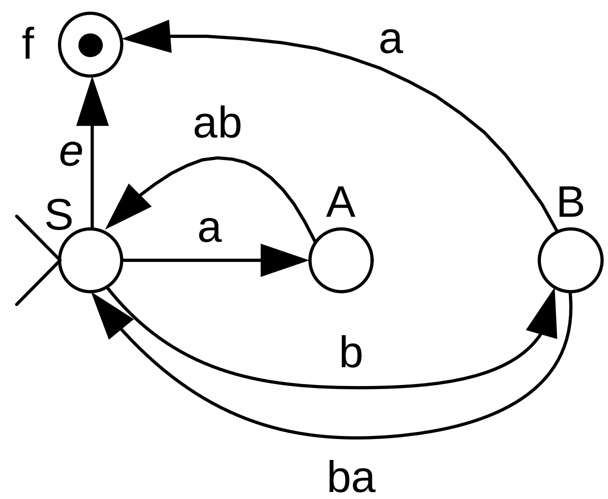
Exam: P4

NFA $\leftrightarrow$ RG

- $\exists$ rules $A_1 \rightarrow w_1A_2$, $A_2 \rightarrow w_2A_3$, ... $\leftrightarrow$ $\exists$ (A_1, w_1, A_2), (A_2, w_2, A_3), ... $\in \Delta$ by the construction of M
- $\exists$ (A_1, w_1, A_2), (A_2, w_2, A_3), ... $\in \Delta \leftrightarrow (A_1, w_1w_2\ldots w_n)$ |- $(A_2, w_2w_3\ldots w_n)$ |- ... |- $(A_n, w_n) \leftrightarrow (S, w)$ |-* (A_n, w_n) by the definition of the yield in one step
- $\exists$ rule $A_n \rightarrow w_n$ $\leftrightarrow$ $\exists$ $(A_n, w_n, f) \in \Delta$ by the construction of M
- $\exists$ $(A_n, w_n, f) \in \Delta \leftrightarrow (A_n, w_n)$ |- (f, e) by the definition of the yield in one step
- (S, w) |-* (A_n, w_n), (A_n, w_n) |- (f, e) $\leftrightarrow$ (S, w) |-* (f, e) by the transitive property of yield
- (S, w) |-* (f, e), $f \in F \leftrightarrow w \in L(M)$ by the definition of acceptance

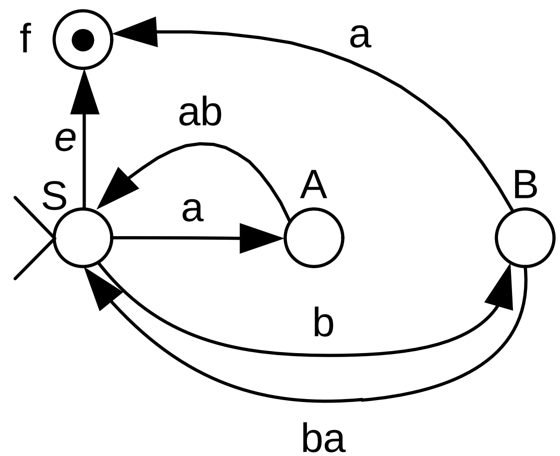
Example

- Construct such NFA M which is equivalent with the given RG G=(V, Σ, R, S)!
  - V = {a, b, A, B, S}
  - Σ = {a, b}
  - R = {S → aA | bB | *e*, A → abS, B → baS | a}
  - S = S
  - L(G) = (aab ∪ bba)*(ba ∪ *e*)



Example

- Give the derivation and computation for w = aabbba !
  – S => aA => aabS => aabbB =>
    => aabbbaS => aabbba
  – (S, aabbba) |- (A, abbba) |-
    |- (S, bba) |- (B, ba) |- (S, *e*) |- (f, *e*)



- Give the derivation and computation for w = bbaaab !
  – S => bB => bbaS => bbaaA => bbaaabS => bbaaab
  – (S, bbaaab) |- (B, baaab) |- (S, aab) |- (A, ab) |-
    |- (S, *e*) |- (f, *e*)

Summary

- Context-free languages
- Context-free grammars
- Derivation
- Language generated by CFG
- Regular grammars
- NFA $\leftrightarrow$ RG

Next time

- Pushdown automata

Elements of the Theory of Computation

Lesson 8

3.3. Pushdown automata

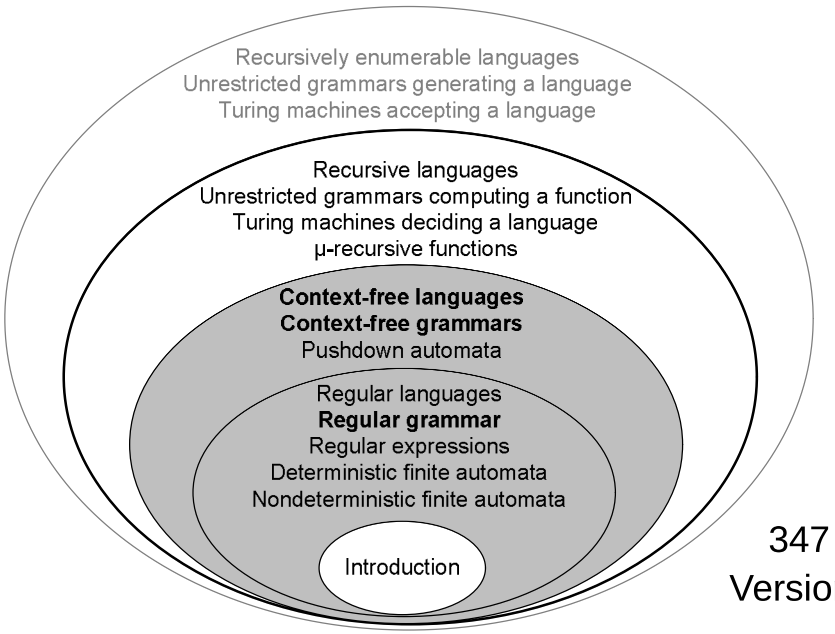
University of Pannonia

Dr. István Heckl, Istvan.Heckl@gmail.com

Version 47

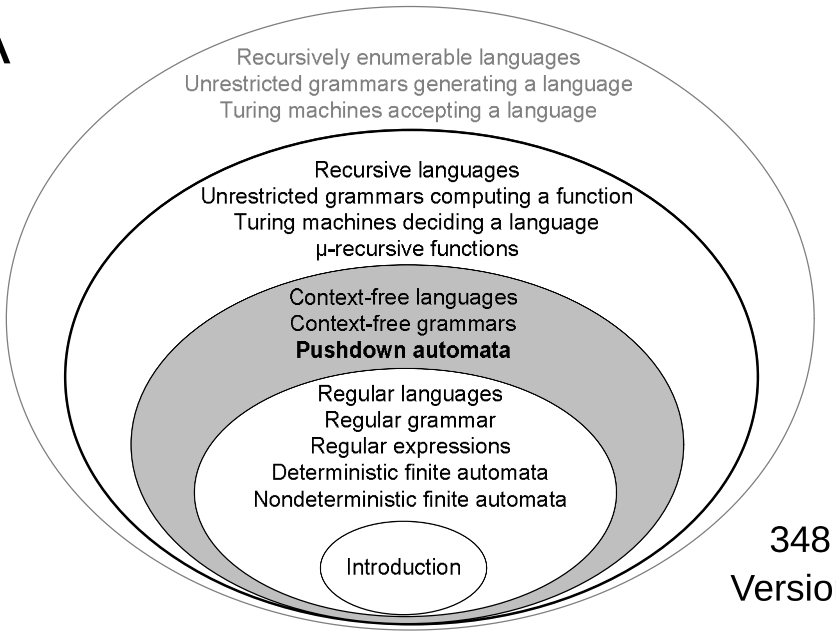
Last time

- Context-free languages
- Context-free grammars
- Derivation
- Language generated by CFG
- Regular grammars
- NFA ↔ RG



Pushdown automata

- Pushdown automata
- Configuration
- Yield in one step
- Yield
- String accepted by PDA
- Language accepted by PDA
- State diagram



Case		E N
Upper	Lower	
Α	α	a
Β	β	
Γ	γ	g
Δ	δ	
Ε	ϵ	e
Ζ	ζ	
Η	η	
Θ	θ	
Ι	ι	
Κ	κ	k

Pushdown automata

- Not every context-free language can be recognized by a finite automaton
  - some context-free languages are not regular
  - e.g.: $\{a^n b^n : n \in N\}$
- What extra features do we need to add to the finite automata so that they accept any context-free language?

Pushdown automata

- Consider $L = \{wcw^R : w \in \{a, b\}^*\}$!
  - L can be generated by a CFG containing rules:

    $S \rightarrow aSa$, $S \rightarrow bSb$, $S \rightarrow c$
  - it seems any device capable accepting L must remember the first half of the input string so it can check it against the second half

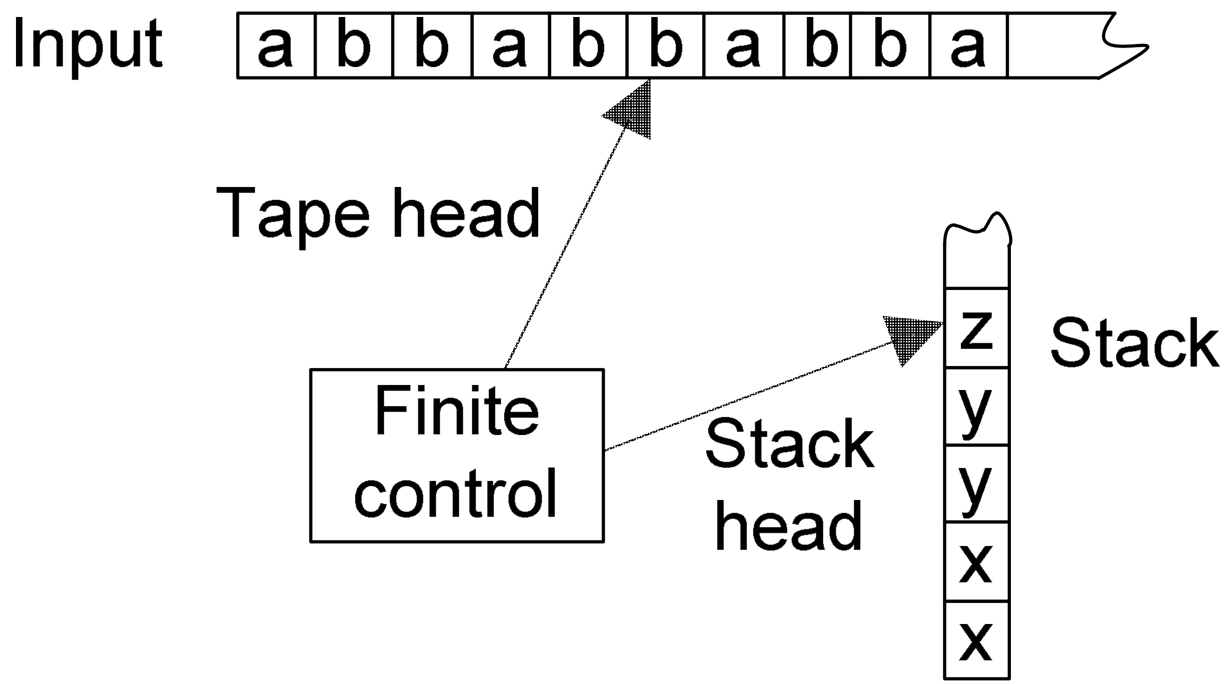
Pushdown automata

- The language of balanced parenthesis:
  - G = ({S, (,)}, {(,)}, {S → *e* | SS | (S)}, S)
- What algorithm can decide this language?
  - start counting at zero
  - add one for every left parentheses
  - subtract one for every right parenthesis
  - reject a string if the count either goes negative at any time or ends up different from zero
    - otherwise it should be accepted

Pushdown automata

- The counter can be considered as a special case of a stack, on which only one kind of symbol can be written
  - states cannot be used because the input can be longer than the number of states
- Rules of the regular grammar, e.g., $A \rightarrow aB$, are easy to simulate by a finite automaton, as follows:
  - if in state A reading 'a' go to state B
  - see RG $\leftrightarrow$ NFA
- What about a rule whose right-hand side is not a terminal followed by a non-terminal?

Pushdown automata



Pushdown automata

- Components of a pushdown automata (PDA):
  - input tape with reading head
    - each tape cell contains a symbol from Σ
    - the tape is infinite to the right
  - control unit
    - finite number of states
  - stack with reading head
    - last in first out (LIFO) data structure
    - infinite capacity

Pushdown automata

- Definition of pushdown automata, M: a six-tuple (K, Σ, Γ, Δ, s, F) where
  - K set of states (finite)
  - Σ alphabet of the input symbols (finite)
  - Γ alphabet of the stack symbols
    - can be different from Σ
  - $\Delta \subseteq (K \times \Sigma^* \times \Gamma^*) \times (K \times \Gamma^*)$ transition relation
    - according to book $\Delta \subseteq (K \times (\Sigma \cup \{e\}) \times \Gamma^*) \times (K \times \Gamma^*)$
  - $s \in K$ initial state
  - $F \subseteq K$ set of final states

Pushdown automata

- The meaning of transition relation:
  - if $((p, \alpha, \beta), (q, \gamma)) \in \Delta$
  - then from state p, reading α, popping β M goes to state q while pushing γ
    - if $\alpha = e$, then the input is not consulted
    - replaces β by γ on the top of the stack
- The PDA described here is non-deterministic
  - there is deterministic PDA but it is not equivalent with the non-deterministic PDA

Pushdown automata

- Stack operations
  - push: a symbol is added to the top of the stack
    - ((p, u, *e*), (q, a)) pushes 'a'
  - pop: a symbol is removed from the top of the stack
    - ((p, u, a), (q, *e*)) pops 'a'

Pushdown automata

- Every finite automaton can be viewed as a pushdown automaton
  - let M = (K, Σ, Δ, s, F) be an NFA
  - let M' = (K, Σ, Ø, Δ', s, F) be a PDA
    - Δ' = {((p, u, *e*), (q, *e*)) : (p, u, q) ∈ Δ}
  - M' does not consult its stack otherwise simulates the transition of M
  - L(M) = L(M')

Configuration

- Definition of configuration of a PDA $M = (K, \Sigma, \Gamma, \Delta, s, F)$: an ordered triple of the current state of M, the unread part of the input, and the whole stack
  - it is an element of $K \times \Sigma^* \times \Gamma^*$
  - there is no need to store the whole input because the reading head cannot go to the left, so, the already read input cannot affect the result
  - e.g.: (q, bbb, abc)
    - 'a' is at the top of the stack

Yields in one step

- Definition of yield in one step of a PDA, $|-_M$: a relation between two "neighboring" configurations
  - formally:
    - if $x, y \in \Sigma^*$, $q, p \in K$, $\beta, \eta, \gamma \in \Gamma^*$, $((p, x, \beta), (q, \gamma)) \in \Delta$
    - then $((p, xy, \beta\eta), (q, y, \gamma\eta)) \in |-$ or $(p, xy, \beta\eta) |-_M (q, y, \gamma\eta)$
    - we say: $(p, xy, \beta\eta)$ yields $(q, y, \gamma\eta)$ in one step
- If it is unambiguous that the yield corresponds to which PDA then the subscript M may be omitted

Computation

- Definition of computation by PDA M: a sequence of configuration C_0, C_1, ... C_n such that C_0 |- C_1 |- ... |- C_n
  - e.g.: (q_1, abaa, *e*) |- (q_2, aa, xx) |- (q_1, *e*, *e*)
  - the length of a computation is the number of yield in one step applied
  - the first and the last configuration can be connected with the yield in n steps relation, signed as $|-^n$
    - e.g.: (q_1, abaa) $|-^3$ (q_3, a)

Yield

- Definition of yield of a PDA, $|-_M*$: the reflexive, transitive closure of $|-_M$
  - if (q', w', α') can be reached from (q, w, α) through a number of yield in one step operation then the yield operation holds between (q, w, α) and (q', w', α')
    - denoted as: (q, w, α) |-* (q', w', α')

String accepted by PDA

- Definition of string accepted by PDA M: $w \in \Sigma^*$ is accepted by M if (s, w, *e*) |-* (q, *e*, *e*), $q \in F$
  - the automaton is in final state
  - the whole input is read
  - the stack is empty

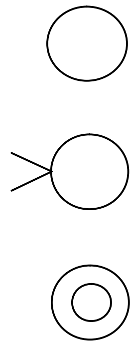
String accepted by PDA

- The yield in PDA can lead to different configurations reading the same input
  - there are possible branching at the computation of w
  - if there is as much as one path to (q, *e*, *e*), q $\in$ F then w is accepted
- If PDA M cannot process the whole input because the missing transitions then w is rejected

Language accepted by PDA

- Definition of language accepted by PDA M, L(M): the set of strings accepted by M
  - $L(M) = \{w \in \Sigma^* : (s, w, e) \vdash_M^* (q, e, e), q \in F\}$

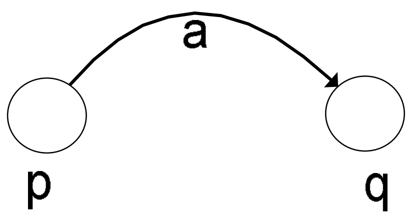
State diagram



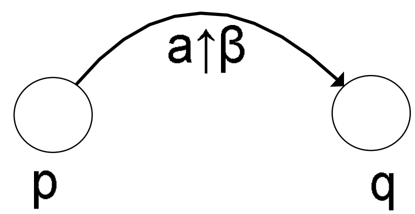
state

initial state

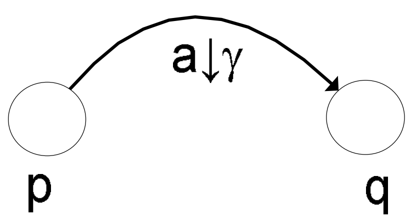
final state



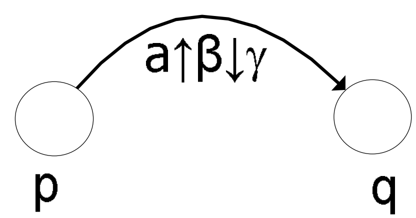
$((p, a, e), (q, e))$



$((p, a, \beta), (q, e))$



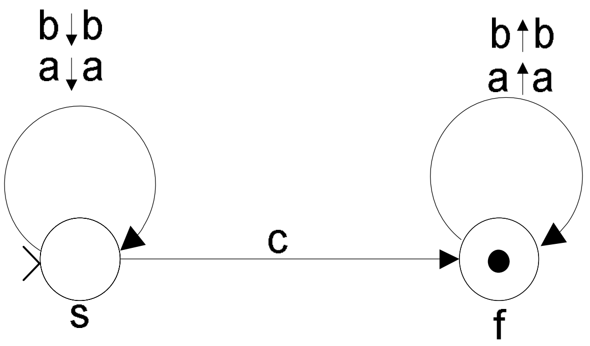
$((p, a, e), (q, \gamma))$



$((p, a, \beta), (q, \gamma))$

Example

- Design a pushdown automaton M to accept the language L = {wcw^R : w $\in$ {a, b}*}!
  - e.g.: ababcbaba $\in$ L, abcab, cbc $\notin$ L
  - M = (K, Σ, Γ, Δ, s, F) where K = {s, f}, Σ = {a, b, c}, Γ = {a, b}, F = {f} and Δ is
    - (1) ((s, a, *e*), (s, a))
    - (2) ((s, b, *e*), (s, b))
    - (3) ((s, c, *e*), (f, *e*))
    - (4) ((f, a, a), (f, *e*))
    - (5) ((f, b, b), (f, *e*))
      - you may omit the inner parenthesis in a transition



Example

- Transitions:
  - 1, 4 corresponds to rule: S → aSa
  - 2, 5 corresponds to rule: S → bSb
  - 3 corresponds to rule: S → c
- Operation:
  - in state s reads the first half of its input
    - transitions 1 and 2 read w while pushing a corresponding stack symbols into the stack for each input symbol
      - 'a' corresponds to 'a', b corresponds to b now

Example

- switches state from s to f without consulting its stack, when M sees c in the input string
- in state f reads the second half of its input
  - transitions 4 and 5 remove the top symbol from the stack, if the corresponding input symbol is read

Example

- The input is accepted if
  - the automaton can reach configuration (f, *e*, *e*)
- The input is rejected if
  - not exactly one c is encountered
  - in the second phase of operation the top stack symbol and the next input symbol does not match
  - the stack and the input is not finished at the same time

Example

- The emphasis is shifted from the meaning of the states to the meaning of the stack symbols
  - there are fewer states but they still have meaning
    - state s: we are before c
    - state f: we are after c
  - a symbol in the stack means that the same symbol must be at the corresponding position
    - e.g.: ba in stack (b at the top) means that the word must finished with ba

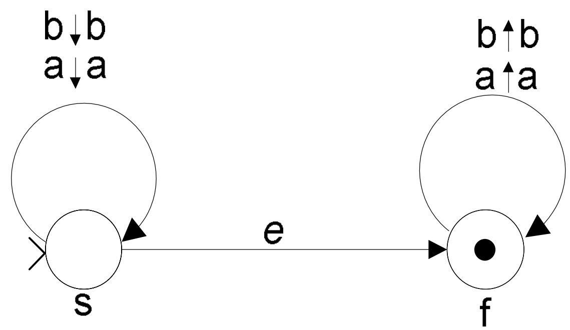
Example

State	Unread input	Stack	Transition used
s	abbcbba	*e*	-
s	bbcbba	a	1
s	bcbba	ba	2
s	cbba	bba	2
f	bba	bba	3
f	ba	ba	5
f	a	a	5
f	*e*	*e*	4

Example

- Design a pushdown automaton M to accept the language L = {ww^R : w ∈ {a, b}*} !
  - M = (K, Σ, Γ, Δ, s, F), where K = {s, f}, Σ = {a, b},

    F = {f} and Δ is the set of the following five transitions
    - (1) ((s, a, *e*), (s, a))
    - (2) ((s, b, *e*), (s, b))
    - (3) ((s, *e*, *e*), (f, *e*))
    - (4) ((f, a, a), (f, *e*))
    - (5) ((f, b, b), (f, *e*))



Example

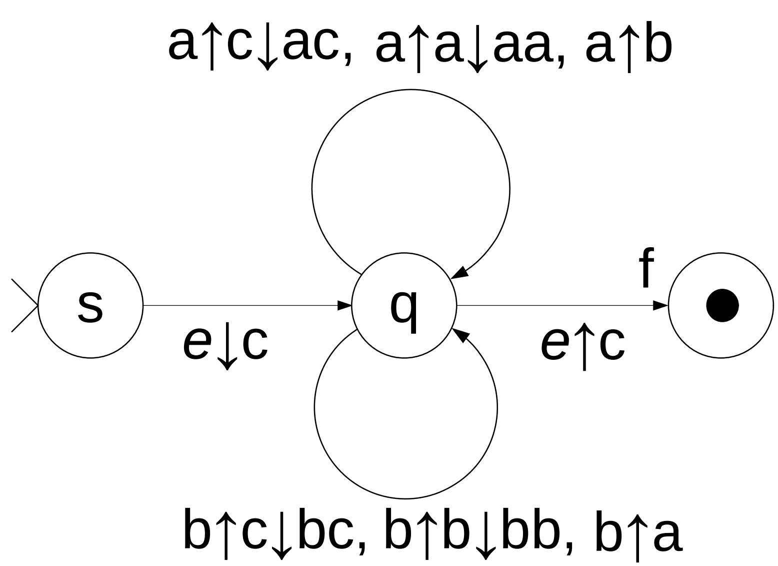
- The language is very similar to the previous one but there is no way to determine the middle of the string
  - with two complete read of the input it could be done easily because then you know the length of w
- In state s, M can non-deterministically choose either
  - to push the next input symbol onto the stack
  - to switch to state f without consuming any input
    - middle point has been reached
- Therefore even starting from a string of the form ww^R, M has computations that do not lead it to the accepting configuration (f, *e*, *e*)
  - but there is at least one that does

Example

- Design a pushdown automaton M to accept
  L = {w ∈ {a, b}* : w has the same number of 'a' and b} !
- M = (K, Σ, Γ, Δ, s, F), where K = {s, q, f}, Σ = {a, b},

  Γ = {a, b, c}, F = {f}, and Δ is listed below

  – (1) ((s, *e*, *e*), (q, c))
  – (2) ((q, a, c), (q, ac))
  – (3) ((q, a, a), (q, aa))
  – (4) ((q, a, b), (q, *e*))
  – (5) ((q, b, c), (q, bc))
  – (6) ((q, b, b), (q, bb))
  – (7) ((q, b, a), (q, *e*))
  – (8) ((q, *e*, c), (f, *e*))



Example

- Stack:
  - there is a c on the bottom as a marker
  - an 'a' in the stack indicates the excess of 'a' over b thus far read on the input tape
  - b in the stack indicates the excess of b over 'a' thus far read on the input tape

Example

- Operation:
  - transition 1 perform initialization
    - puts M in state q and places c on the bottom of the stack
  - in state q, when M reads 'a', M may
    - push 'a' onto c (transition 2)
    - push 'a' onto another 'a' (transition 3)
    - pop b (transition 4)
  - when reading a b from the input, M may
    - push b onto c (transition 5)
    - push b onto another b (transition 6)
    - pop 'a' (transition 7)
  - transition 8 ends the computation by popping c

Example

State	Unread input	Stack	Transition	Comments
s	abbbabaa	*e*	-	Initial configuration
q	abbbabaa	c	1	Bottom marker
q	bbbabaa	ac	2	Start a stack of 'a'
q	bbabaa	c	7	Remove one 'a'
q	babaa	bc	5	Start a stack of b
q	abaa	bbc	6	
q	baa	bc	4	
q	aa	bbc	6	
q	a	bc	4	
q	*e*	c	4	
f	*e*	*e*	8	Accepts

Example

- Both transitions 2 and 3 pushes 'a' into the stack (similarly transitions 5 and 6 pushes b) so why not just use transition ((q, a, *e*), (q, a)) instead?
  - because then M would be non-deterministic
  - e.g., at (q, abaa, bc) both ((q, a, b), (q, *e*)) and ((q, a, *e*), (q, a)) would be applicable
    - the first transition is correct in the given configuration

Summary

- Pushdown automata
- Configuration
- Yield in one step
- Yield
- String accepted by PDA
- Language accepted by PDA
- State diagram

Next time

- Pushdown automata and context-free grammars
- Languages that are and are not context-free

Elements of the Theory of Computation

Lesson 9

3.4. Pushdown automata and context-free grammars

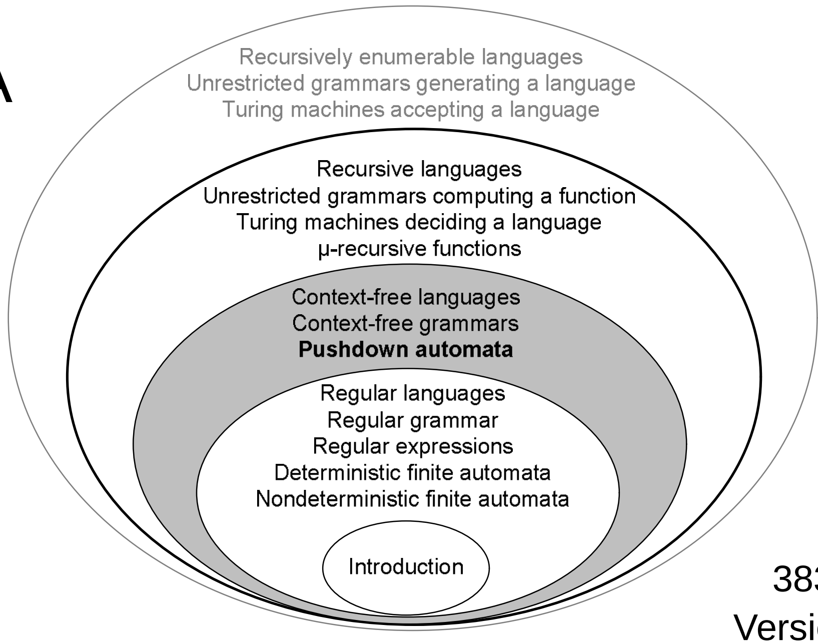
3.5. Languages that are and are not context-free

University of Pannonia

Dr. István Heckl, Istvan.Heckl@gmail.com

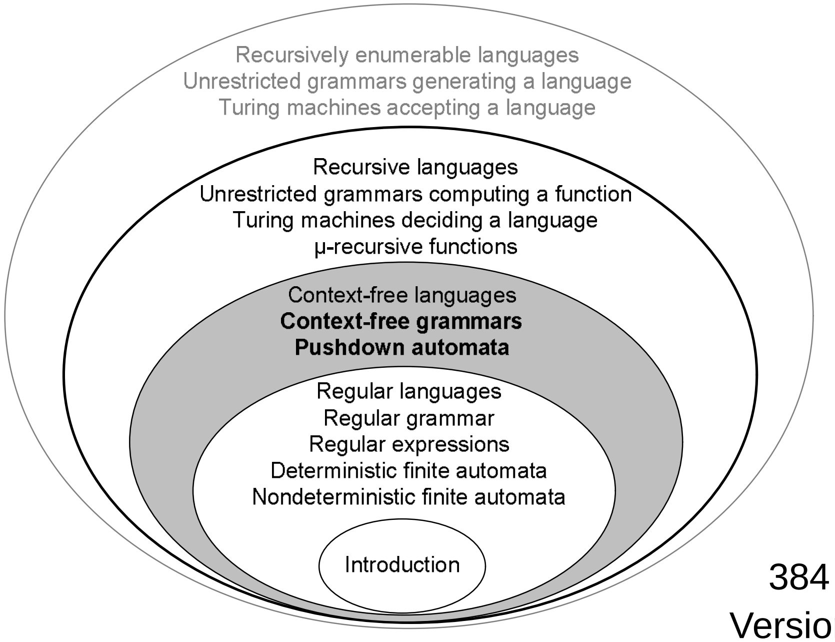
Last time

- Pushdown automata
- Configuration
- Yield in one step
- Yield
- String accepted by PDA
- Language accepted by PDA
- State diagram



Pushdown automata and context-free grammars

- CFG → PDA
- Simplicity
- PDA → CFG
- Closure properties
- Pumping theorem 2



CFG $\rightarrow$ PDA

- Definition of leftmost derivation: such a derivation in which always the leftmost non-terminal is selected for substitution
  - denote with: $=>^L$
  - e.g.: $R = \{S \rightarrow AB, S \rightarrow aA, A \rightarrow a, B \rightarrow Sb\}$
    $S =>^L AB =>^L aB =>^L aSb =>^L aABb =>^L aaBb =>^L$
    $=>^L aaSbb =>^L aaaAbb =>^L aaaabb$

CFG $\rightarrow$ PDA

- Theorem: for $\forall$ CFG G = (V, Σ, R, S) $\exists$ PDA M such that L(M) = L(G)
- Construction:
  - M uses V as the stack symbols
  - M mimics the leftmost derivation of G
  - M = ({p, q}, Σ, V, Δ, p, {q})

CFG $\rightarrow$ PDA

- Δ contains:
  - (1) $((p, e, e), (q, S))$
    - M begins by pushing S (start symbol of G)
    - M enters into state q
  - (2) $((q, e, A), (q, x))$, for $\forall$ rule $A \rightarrow x \in R$
    - if the topmost symbol, A, on the stack is non-terminal then it is replaced by the right-hand side, x, of some rule $A \rightarrow x \in R$
  - (3) $((q, a, a), (q, e))$ for $\forall$ $a \in \Sigma$
    - pops the topmost symbol from the stack provided that it is a terminal symbol that matches the next input symbol

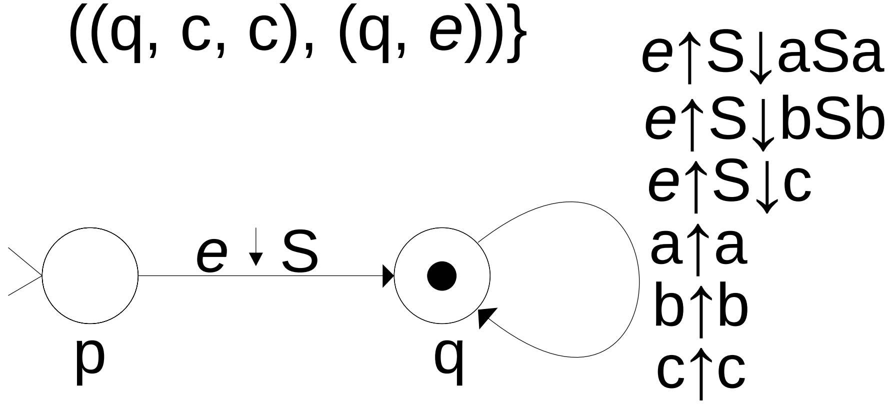
Example

- Give CFG G such that L(G)= {w $\in$ {a, b}* : w = xcx^R} and give the equivalent PDA!
  - V = {S, a, b, c}
  - Σ = {a, b, c}
  - R = {S $\rightarrow$ aSa | bSb | c}
  - remark: we have already constructed a PDA for this language
    - let us call the previous PDA "plain" and the current "constructed"

R = {S → aSa | bSb | c}
Σ = {a, b, c}

Example

- M = ({p, q}, Σ, V, Δ, p, {q}), with
  - Δ = {((p, *e*, *e*), (q, S)), (T1)
    ((q, e, S), (q, aSa)), (T2)
    ((q, *e*, S), (q, bSb)), (T3)
    ((q, *e*, S), (q, c)), (T4)
    ((q, a, a), (q, *e*)), (T5)
    ((q, b, b), (q, *e*)), (T6)
    ((q, c, c), (q, *e*))} (T7)



State	Unread Input	Stack	Transition Used
p	abbcbba	*e*	-
q	abbcbba	S	T1
q	abbcbba	aSa	T2
q	bbcbba	Sa	T5
q	bbcbba	bSba	T3
q	bcbba	Sba	T6
q	bcbba	bSbba	T3
q	cbba	Sbba	T6
q	cbba	cbba	T4
q	bba	bba	T7
q	ba	ba	T6
q	a	a	T6
q	*e*	*e*	T5

- S =>^{L} aSa =>^{L} abSba =>^{L} abbSbba =>^{L} abbcbba

CFG $\rightarrow$ PDA

- Lemma: $S \Rightarrow^{L*} w\alpha \leftrightarrow (q, w, S) \vdash^* (q, e, \alpha)$
  - α starts with non-terminal, $w \in \Sigma^*$, $\alpha \in (V - \Sigma)V^* \cup \{e\}$
  - consider the original grammar and the constructed PDA M, $q \in F$
- Proof: $\rightarrow$, induction for the number of steps in the derivation
  - basis step:
    - 0 step derivation: $S \Rightarrow^{L*} S \rightarrow w = e, \alpha = S$
    - $(q, w, S) \vdash^* (q, w, S)$ by the reflexivity of $\vdash^*$
    - $w = e, \alpha = S \rightarrow (q, w, S) \vdash^* (q, e, \alpha)$ by the pervious points

Exam: P5

– induction step:

- $S \Rightarrow^{L*} w\alpha \rightarrow$
  $S = u_0 \Rightarrow^{L*} u_n = xA\beta \Rightarrow^{L} u_{n+1} = x\gamma\beta = w\alpha$ by the transitivity of the derivation
  - A is the leftmost non-terminal in u_n, $x \in \Sigma^*$
  - $A \rightarrow \gamma \in R$
- $S \Rightarrow^{L*} xA\beta \rightarrow (q, x, S) \vdash^* (q, e, A\beta)$ by the induction hypothesis, $w_2 = x$, $\alpha_2 = A\beta$
- $A \rightarrow \gamma \in R \rightarrow \exists\ ((q, e, A), (q, \gamma)) \in \Delta$ by the construction, rule type 2
- $\exists\ ((q, e, A), (q, \gamma)) \in \Delta \rightarrow (q, e, A\beta) \vdash (q, e, \gamma\beta)$ by the definition of yield in one step
- $(q, x, S) \vdash^* (q, e, A\beta), (q, e, A\beta) \vdash (q, e, \gamma\beta) \rightarrow$
  $(q, x, S) \vdash^* (q, e, \gamma\beta)$ by the transitivity of yield

CFG → PDA

- (q, x, S) |-* (q, *e*, γβ) → (q, xy, S) |-* (q, y, γβ) by the end of input theorem
  - y can be any string, but we select it such as w = xy ∈ Σ*
- xγβ = wα (first row), w = xy → xγβ = xyα → γβ = yα
  - y is the starting terminal part of γβ
- (q, w, S) |-* (q, y, yα) by the previous two point
- (q, y, yα) |-* (q, *e*, α) by the construction, rule type 3
- (q, w, S) |-* (q, y, yα), (q, y, yα) |-* (q, *e*, α) → (q, w, S) |-* (q, *e*, α) by the transitivity of yield

CFG $\rightarrow$ PDA

- Proof: if $\leftarrow$, induction for the number of type 2 transitions
  - suppose (q, w, S) |-* (q, *e*, α)
  - α starts with non-terminal, $w \in \Sigma^*$, $\alpha \in (V - \Sigma)V^* \cup \{e\}$
  - basis step: 0 type 2 transition
    - in (q, w, S) only type 2 transition is applicable (replacing S) so there can be only 0 total transition
    - (q, *e*, S) |-* (q, *e*, S) $\rightarrow$ w = *e*, α = S
    - $S \Rightarrow^{L*} S$ by the reflexivity of |-*
    - $S \Rightarrow^{L*} w\alpha$ by the previous two points

Exam: P6

CFG $\rightarrow$ PDA

– induction step:
  - (q, w, S) |-* (q, *e*, α) $\rightarrow$
    (q, xy, S) |-* (q, y, Aβ) |- (q, y, $\gamma\beta$) |-* (q, *e*, α) by the transitivity of yield
    – w = xy $\in \Sigma^*$
    – (q, y, Aβ) |- (q, y, $\gamma\beta$) the $(n+1)^{th}$ type 2 transition
    – $\exists$ A $\rightarrow \gamma \in$ R by the construction
  - (q, xy, S) |-* (q, y, Aβ) $\leftrightarrow$ (q, x, S) |-* (q, *e*, Aβ) by the end of input theorem

CFG $\rightarrow$ PDA

- $(q, x, S) \vdash^* (q, e, A\beta) \leftrightarrow S \Rightarrow^{L*} xA\beta$ by the induction hypothesis $w_2 = x$, $\alpha_2 = A\beta$
- $A \rightarrow \gamma \in R \rightarrow xA\beta \Rightarrow^{L*} x\gamma\beta$ by the definition of the one step derivation
- $(q, y, \gamma\beta) \vdash^* (q, e, \alpha)$ with type 3 transitions (the type 2 transitions are already used up) $\rightarrow$ $(q, y, y\alpha) \vdash^* (q, e, \alpha)$, $\gamma\beta = y\alpha$, $y \in \Sigma^*$ by the construction
- $S \Rightarrow^{L*} xA\beta$, $xA\beta \Rightarrow^{L*} x\gamma\beta \rightarrow S \Rightarrow^{L*} x\gamma\beta$ by the transitivity of derivation
- $S \Rightarrow^{L*} x\gamma\beta$, $\gamma\beta = y\alpha \rightarrow S \Rightarrow^{L*} xy\alpha$
- $S \Rightarrow^{L*} xy\alpha$, $w = xy \rightarrow S \Rightarrow^{L*} w\alpha$

CFG $\rightarrow$ PDA

- Proof of the theorem:
  - each language generated by a CFG is accepted by some PDA
  - $w \in L(G) \leftrightarrow S =>^{L}* w$ by the definition of acceptance
  - $S =>^{L}* w \leftrightarrow (q, w, S)$ |-* (q, e, e) by the lemma with $\alpha = e$
  - (p, w, e) |- (q, w, S) |-* $(q, e, e) \leftrightarrow w \in L(M)$ by the definition of acceptance and the construction of M
    - use transition type 1 for the first step
    - $q \in F$ according to the construction of M

Simplicity

- Definition of simple PDA: for $\forall$ ((q, a, β), (p, γ)) $\in$ Δ when q is not the starting state $\rightarrow$ β $\in$ Γ, |γ| ≤ 2
  - the automaton always change the topmost stack symbol with *e*, one, or two other symbols
  - "when q is not the start state" condition is important to start the computation when the stack is empty

Simplicity

- Theorem: for $\forall$ PDA M(K, Σ, Γ, Δ, s, F) $\exists$ a simple PDA M' such that L(M) = L(M')
- Construction:
  - M' = (K', Σ, Γ $\cup$ {Z}, Δ', s', {f'})
  - s', f' are new states
  - Z is a new stack symbol signaling the bottom of the stack
  - Δ' contains
    - ((s', e, e), (s, Z))
    - ((f, e, Z), (f', e)), $\forall$ f $\in$ F
    - all transitions of Δ (some violate simplicity)

Simplicity

- eliminating transitions when more than one stack symbol is popped
  - replace $((q, a, \beta), (p, \gamma)) \in \Delta'$, $\beta = C_1C_2\ldots C_n$
  - with:

$$((q, e, C_1), (r_1, e)),$$
$$((r_1, e, C_2), (r_2, e)),$$
$$\ldots$$
$$((r_{n-2}, e, C_{n-1}), (r_{n-1}, e)),$$
$$((r_{n-1}, a, C_n), (p, \gamma))$$

  - $r_1, r_2, \ldots r_{n-1}$ are new states
  - pop C_i one by one

Simplicity

– eliminating transitions when more than 1 stack symbols are pushed

- replace $((q, a, \beta), (p, \gamma)) \in \Delta'$, $\gamma = C_1 \ldots C_n$
- with

$$((q, a, \beta), (r_1, C_n)),$$
$$((r_1, e, e), (r_2, C_{n-1})),$$
$$\ldots$$
$$((r_{n-2}, e, e), (r_{n-1}, C_2)),$$
$$((r_{n-1}, e, e), (p, C_1)),$$

- $r_1, \ldots, r_{n-1}$ are new state
- push C_i one by one
- simplicity would allow that $n = 2$

Simplicity

- eliminating transitions when the topmost stack symbol is not popped
  - replace $((q, a, e), (p, \gamma)) \in \Delta'$, $q \neq s'$
  - with $((q, a, A), (p, \gamma A))$, $\forall A \in \Gamma \cup \{Z\}$
    - popping and pushing A before pushing γ
      - » each potential transition is produced though probably only some of them is used
    - in the previous step we made sure that only 1 stack symbol is pushed (γ), now at most two can be pushed (γA)

PDA → CFG

- Theorem: the language of each pushdown automaton is generated by some context-free language
- Construction :
  - let M a PDA and M' the corresponding simple PDA
  - we shall construct G(V, Σ, R, S) such that L(G) = L(M')
  - V = S ∪ Σ ∪ <q, A, p>, q, p ∈ K', A ∈ Γ ∪ {*e*, Z}
    - <q, A, p> is a non-terminal representing a portion of the input string that might be read while M' moves from state q to state p and the net effect of the stack is popping A
    - lots of these non-terminals will not be used

PDA → CFG

– R contains

- S → <s, Z, f'>
  - S can be any such string which is read by M' while moving from s to f' and while the net effect on the stack is popping Z
  - M' contains Z in the stack in state s because ((s', e, e), (s, Z)) ∈ Δ'
  - M' does not contain Z in the stack in state f' because ((f, e, Z), (f', e)) ∈ Δ', ∀ f ∈ F

PDA $\rightarrow$ CFG

- <q, B, p> $\rightarrow$ a<r, C, p> for ((q, a, B), (r, C)) $\in$ Δ', for each p $\in$ K'
  - transition ((q, a, B), (r, C)) has to be simulated
  - we know that at state q, there is B at the top of the stack
    - » if another symbol is at the top of the stack then it is handled by another transition
  - p is not defined by the transition so we regard each possibility

<q, B, p> → a<r, C, p>

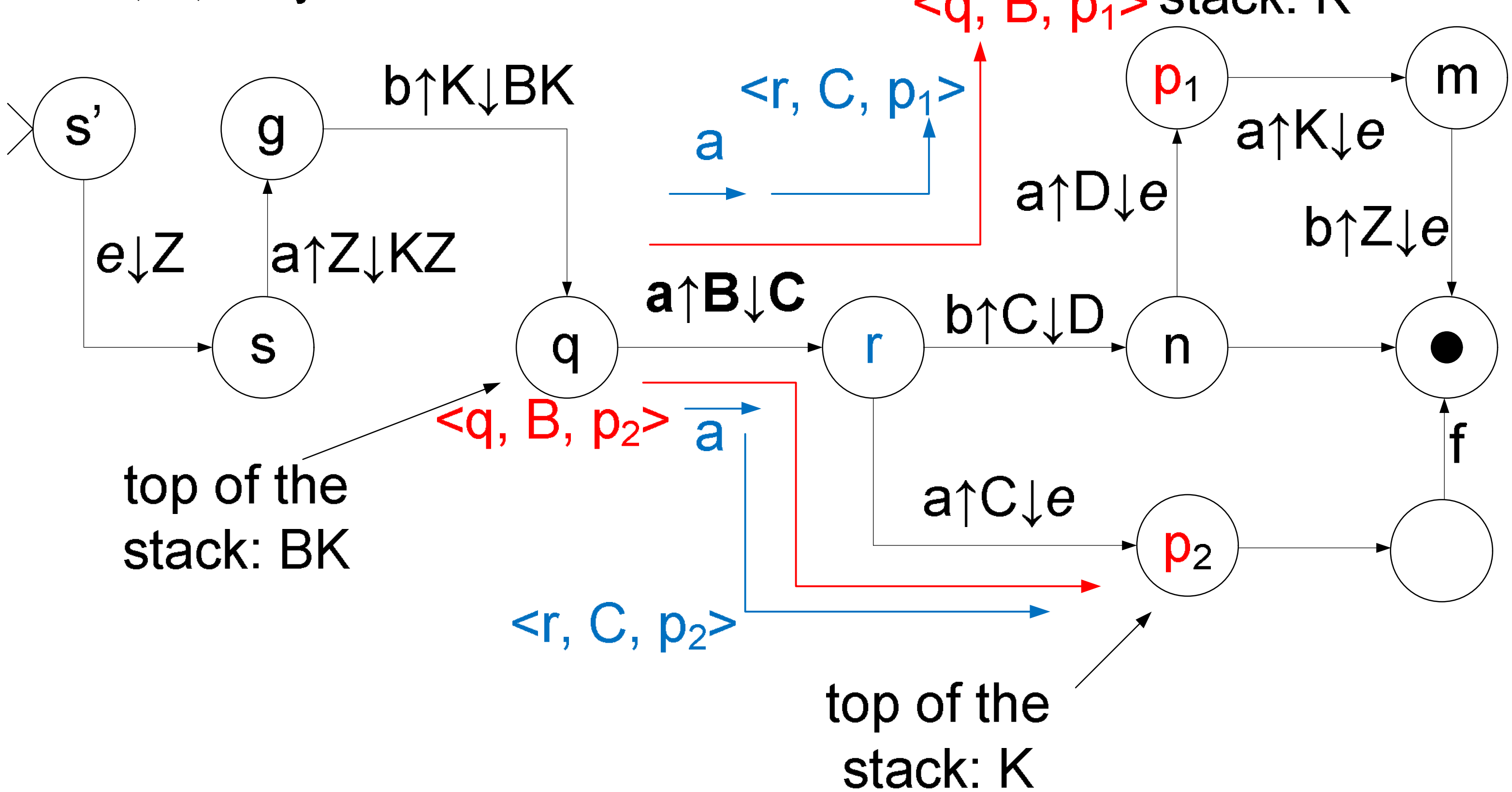
PDA → CFG

- left side: string that is read while moving from state q to p and the net effect is popping B
- right side: 'a' concatenated by a string that is read while moving from state r to p and the net effect is popping C
- we arrive to state p in both cases
- the net effect is popping B in both cases
  - » in the second case B is changed to C first as ((q, a, B), (r, C)) dictates
- the same string is read in both cases
  - » the beginning of the string is 'a' as the transition dictates

PDA $\rightarrow$ CFG

- $\langle q, B, p\rangle \rightarrow a\langle r, C_1, p'\rangle\langle p', C_2, p\rangle$ for $((q, a, B), (r, C_1C_2)) \in \Delta'$, for each $p, p' \in K'$
  - we handled each potential transition of a simple PDA
- $\langle q, e, q\rangle \rightarrow e, \forall q \in K$
  - while remaining in state q without consulting the stack nothing is read
  - eliminating the extra non-terminals

Simple PDA: pop 1 symbol, push 0, 1, 2 symbols



- Constructed rules: ..., $\langle q, B, p_1\rangle \rightarrow a\langle r, C, p_1\rangle$, $\langle q, B, p_2\rangle \rightarrow a\langle r, C, p_2\rangle$, ...

PDA $\rightarrow$ CFG

- Lemma: q, p $\in$ K', A $\in \Gamma \cup \{e\}$, x $\in \Sigma^*$,

  $(q, x, A) \vdash_{M'}^{*} (p, e, e) \leftrightarrow \langle q, A, p\rangle \Rightarrow_G^{*} x$
- Proof: induction on the length of the derivation of G or computation of M'

PDA $\leftrightarrow$ CFG

- Theorem: the class of languages accepted by PDA is exactly the class of languages generated by CFG
- Proof:
  - the language of each CFG is accepted by some PDA
  - the language of each PDA is generated by some CFG

Closure properties

- Theorem: context-free languages are closed under union
  - the union of such languages which are generated by two CFGs can be also generated by a CFG
- Construction:
  - let $G_1(V_1, \Sigma_1, R_1, S_1)$, $G_2(V_2, \Sigma_2, R_2, S_2)$ are known CFGs
    - $V_1 - \Sigma_1$, $V_2 - \Sigma_2$ are disjoint
  - construct G such that $L(G) = L(G_1) \cup L(G_2)$
    - $V = V_1 \cup V_2 \cup \{S\}$
    - $\Sigma = \Sigma_1 \cup \Sigma_2$
    - $R = R_1 \cup R_2 \cup \{S \rightarrow S_1 \mid S_2\}$

Closure properties

- Proof:
  - the theorem uses the term closed because the constructed G is CFG as the two initial grammars
    - R_1, R_2 and the new rules are all CFG rules
  - suppose $w \in L(G_1)$
    - we could have supposed that $w \in L(G_2)$
    - $w \in L(G_1) \leftrightarrow S_1 \Rightarrow_{G1}^* w$ by the definition of acceptance
    - $S_1 \Rightarrow_{G1}^* w \leftrightarrow S_1 \Rightarrow_{G}^* w$ by the construction, R contains R_1

Closure properties

- $S \rightarrow S_1 \in R$ by the construction
- $S \rightarrow S_1 \in R \leftrightarrow S \Rightarrow_G S_1$ by the definition of $\Rightarrow_G$
- $S \Rightarrow_G S_1, S_1 \Rightarrow_G^* w \leftrightarrow S \Rightarrow_G^* w$ by the transitivity of $\Rightarrow_G^*$
- $S \Rightarrow_G^* w \leftrightarrow w \in L(G)$ by definition of acceptance

Closure properties

- Theorem: context-free languages are closed under concatenation
  - the concatenation of such languages which are generated by two CFGs can can be also generated by a CFG
- Construction:
  - let $G_1(V_1, \Sigma_1, R_1, S_1)$, $G_2(V_2, \Sigma_2, R_2, S_2)$ are known CFGs
    - $V_1-\Sigma_1$, $V_2-\Sigma_2$ are disjoint
  - construct G such that $L(G) = L(G_1)L(G_2)$
    - $V = V_1 \cup V_2 \cup \{S\}$
    - $\Sigma = \Sigma_1 \cup \Sigma_2$
    - $R = R_1 \cup R_2 \cup \{S \to S_1S_2\}$

Closure properties

- Proof:
  - the theorem uses the term closed because G is CFG as the two initial grammars
    - R_1, R_2 and the new rules are all CFG rules
  - suppose $w_1 \in L(G_1)$, $w_2 \in L(G_2)$
    - $w_1 \in L(G_1) \leftrightarrow S_1 =>_{G1}^* w_1$ by the definition of acceptance
    - $w_2 \in L(G_2) \leftrightarrow S_2 =>_{G2}^* w_2$ by the definition of acceptance
    - $S_1 =>_{G1}^* w_1, S_2 =>_{G2}^* w_2 \leftrightarrow S_1 =>_{G}^* w_1, S_2 =>_{G}^* w_2$ by the construction, R contains R_1 and R_2

Closure properties

- $S \rightarrow S_1S_2 \in R$ by the construction
- $S \rightarrow S_1S_2 \in R \leftrightarrow S \Rightarrow_G S_1S_2$ by the definition of $\Rightarrow_G$
- $S \Rightarrow_G S_1S_2$, $S_1 \Rightarrow_G^* w_1$, $S_2 \Rightarrow_G^* w_2 \leftrightarrow S \Rightarrow_G^* w_1w_2$ by the transitivity of $\Rightarrow_G^*$
- $S \Rightarrow_G^* w_1w_2 \leftrightarrow w_1w_2 \in L(G)$ by the definition of acceptance

Closure properties

- Theorem: context-free languages are closed under Kleene star
  - the Kleene star of such a language which is generated by a CFG can be also generated by a CFG
- Construction:
  - let $G_1(V_1, \Sigma_1, R_1, S_1)$ a known CFG
  - construct G such that $L(G) = L(G_1)^*$
    - $V = V_1 \cup \{S\}$
    - $\Sigma = \Sigma_1$
    - $R = R_1 \cup \{S \rightarrow SS_1 \mid e\}$
  - the theorem uses the term closed because G is CFG as the initial grammar
    - R_1 and the new rules are all CFG rules

Closure properties

- Proof:
  - suppose $w_1, ..., w_n \in L(G_1)$
    - $w_1 \in L(G_1) \leftrightarrow S_1 =>_{G1}* w_1$ by the definition of acceptance

    ...

    - $w_n \in L(G_1) \leftrightarrow S_1 =>_{G1}* w_n$ by the definition of acceptance
    - $S_1 =>_{G1}* w_1, ..., S_1 =>_{G1}* w_n \leftrightarrow$
      $S_1 =>_G* w_1, ..., S_1 =>_G* w_n$ by the construction, R contains R_1

Closure properties

- $S \rightarrow SS_1 \mid e \in R$ by the construction
- $S \rightarrow SS_1 \mid e \in R \leftrightarrow S =>_G SS_1 =>_G * SS_1...S_1 =>_G S_1...S_1$ by the definition of $=>_G$
- $S =>_G S_1...S_1, S_1 =>_G* w_1, ..., S_1 =>_G* w_n \leftrightarrow S =>_G* w_1...w_n$ by the transitivity of $=>_G*$
- $S =>_G* w_1...w_n \leftrightarrow w_1...w_n \in L(G)$ by the definition of acceptance

Closure properties

- Theorem: the intersection of a context-free language with a regular language is a context-free language
- Construction:
  – L is a context-free language, R is a regular language
  – $\exists$ PDA $M_1(K_1, \Sigma, \Gamma_1, \Delta_1, s_1, F_1)$ such that $L = L(M_1)$
  – $\exists$ DFA $M_2(K_2, \Sigma, \delta, s_2, F_2)$ such that $R = L(M_2)$
  – idea: construct PDA M which carries out the computation of M_1 and M_2 in parallel and accept w if both automata would have accepted w
    - M works as M_1 but also keeps track the state of M_2

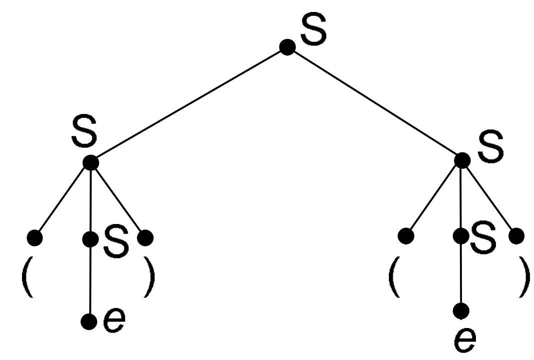
Closure properties

- let $M(K, \Sigma, \Gamma, \Delta, s, F)$
  - $K = K_1 \times K_2$
  - $\Sigma = \Sigma_1 \cup \Sigma_2$
  - $\Gamma = \Gamma_1$
  - $s = (s_1, s_2)$
  - $F = F_1 \times F_2$

Closure properties

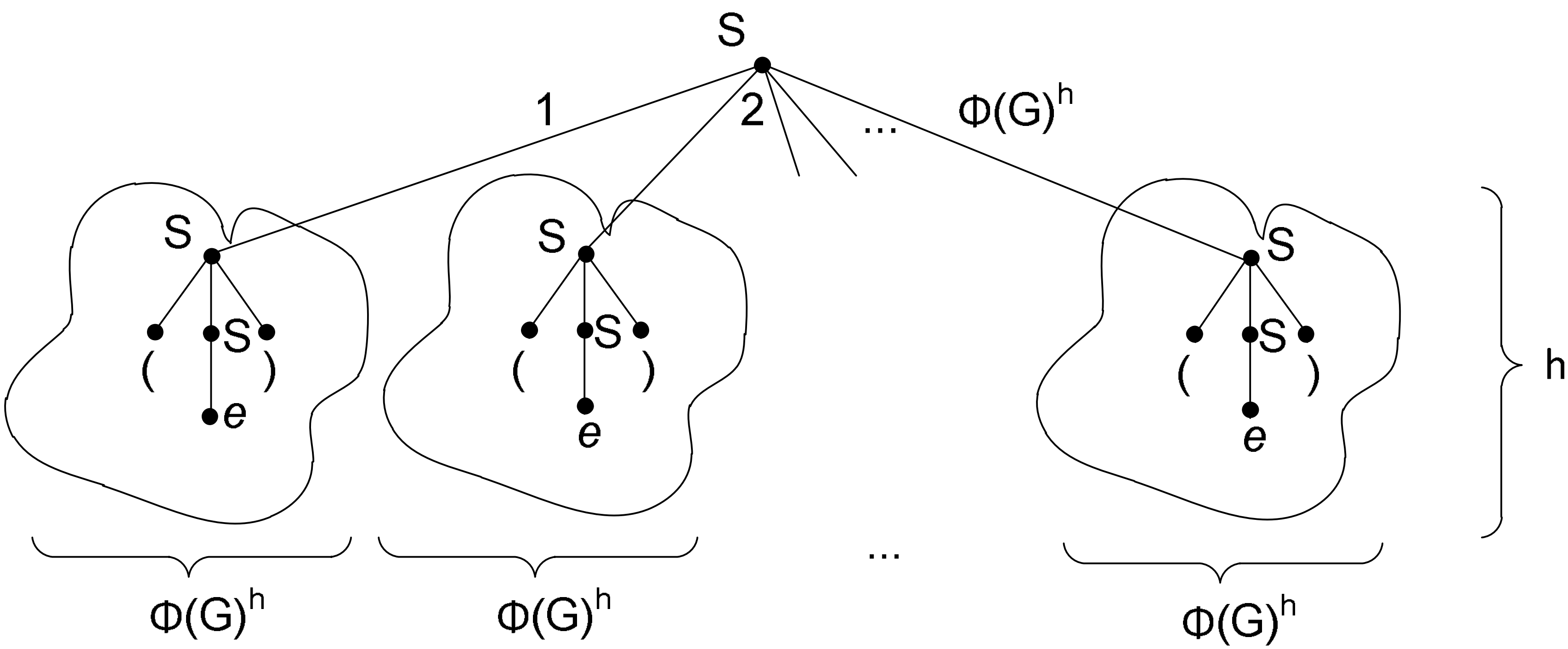
- Δ is defined by
  - $(((q_1, q_2), a, \beta), ((p_1, \delta(q_2, a)), \gamma))$
    - » $\forall$ $((q_1, a, \beta), (p_1, \gamma)) \in \Delta_1, q_2 \in K_2$
    - » the 2nd component of the new state is determined by δ
  - $(((q_1, q_2), e, \beta), ((p_1, q_2), \gamma))$
    - » $\forall$ $((q_1, e, \beta), (p_1, \gamma)) \in \Delta_1, q_2 \in K_2$
    - » the 2nd component of the new state does not change if the head does not move

Pumping theorem 2

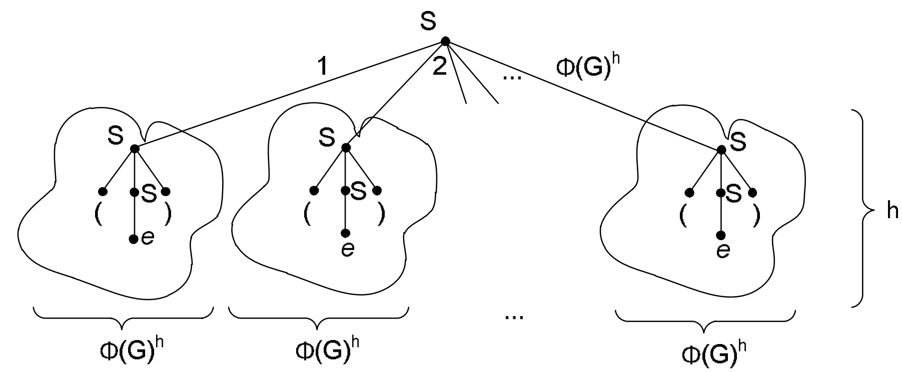


- Definition of fanout of CFG G, Φ(G): the largest number of symbol at the right side of any rule in G
  - e.g.: R = {S → AB | a, B → AAA | ab, A → ABBA | *e*}, Φ(G) = 4
- Parse tree: a graphical way to represent the derivation of a string
  - the inner nodes are non-terminals, the root is S
  - the arcs indicate the rules
  - the leaves gives the string, it is also called the yield of the tree
- Theorem: the length of the yield of any parse tree with height h is at most $\Phi(G)^h$

Pumping theorem 2



Pumping theorem 2



- Proof by induction on h:
  - basis step: h = 1, 1 rule is applied so the maximum yield is Φ(G)
  - induction step:
    - the root of a parse tree with height h+1 connects to at most Φ(G) smaller parse trees with height h
    - according to the induction hypothesis the length of the yield of the smaller parse trees is no more than $\Phi(G)^h$
    - the length of the yield of the original parse tree is $\Phi(G) * \Phi(G)^h = \Phi(G)^{h+1}$

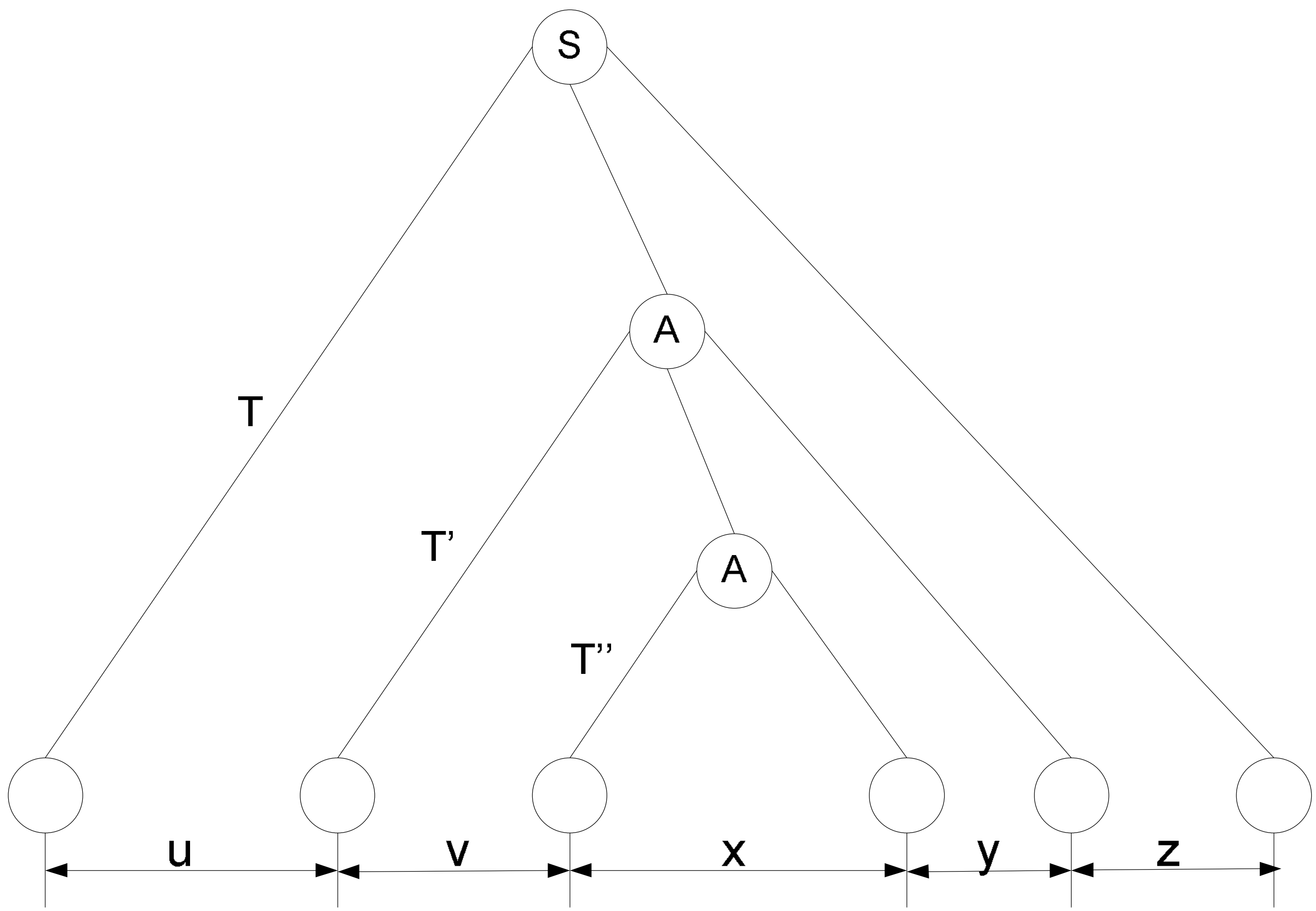
Pumping theorem 2

- Corollary: the height of the parse tree of $w \in L(G)$ where $|w| > \Phi(G)^n$ is greater than n
  - n can be computed using |w| and Φ(G)
  - the greatest n is interesting for us
  - e.g. $\Phi(G) = 4$, $|w| = 65 \rightarrow$ height $> 3 = n$, $4^3 = 64$

Pumping theorem 2

- Pumping theorem 2: let G be a CFG, long enough words in L(G) ($|w| > \Phi(G)^{|V-\Sigma|}$), has a form, $w = uvxyz$, $v \neq e$, $y \neq e$, such that $uv^n xy^n z \in L(G)$, $\forall\, n \geq 0$
- Proof:
  - let $w \in L(G)$ such that $|w| > \Phi(G)^{|V-\Sigma|}$ and let T the parse tree of w with the smallest number of leaves
  - according to the previous corollary the height of T is at least $|V-\Sigma|+1$ so the longest path has $|V-\Sigma|+2$ nodes

Pumping theorem 2



Pumping theorem 2

- only the end of a path can be terminal so the longest path contains |V-Σ|+1 non-terminal
- the longest path contains at least one non-terminal twice
  - let this non-terminal signed with A
- there is a derivation of w

$$S \Rightarrow^* uAz \Rightarrow^* uvAyz \Rightarrow^* uvxyz$$

  - where $u, v, x, y, z \in \Sigma^*$, $A \in V-\Sigma$

Pumping theorem 2

– there is also a derivation in G: A =>* vAy which can be repeated several times (including 0) to generate new strings in L(G)
  - the new strings has the form uv^nxy^nz
  - S =>* uAz =>* uxz
  - S =>* uAz =>* uvAyz =>* uvxyz
  - S =>* uAz =>* uvAyz =>* uv^2Ay^2z =>* uv^2xy^2z

– if vy = *e* → ∃ parse tree for w with smaller number of leaves than T which contradicts the initial assumption
  - if vy can be *e* → the theorem states nothing

Example

- G = ({S, A, B, a, b}, {a, b}, {S → bBa, A → aB | aa, B → aAb | bb}, S)
  - S => bBa => baAba => baaBba => baabbba
  - the theorem does not define u, v, x, y, z only states their existence
  - the string is not long enough ($\Phi(G)^{|V-\Sigma|} = 3^3$) but the derivation contains B twice, thus the theorem holds anyway
  - B =>* aaBb can be repeated
  - u = b, v = aa, x = bb, y = b, z = a
  - uv^0xy^0z = bbba, uv^1xy^1z = baabbba, uv^2xy^2z = baaaabbbba, ...

Languages that are not context-free

- Theorem: $L = \{a^n b^n c^n : n \geq 0\}$ is not context-free
- Proof by indirection:
  - let $n > \Phi(G)^{|V-\Sigma|} / 3$
  - $w = a^n b^n c^n$ can be written in the form uvxyz, $v \neq e$, $y \neq e$
  - according to the pumping theorem $uv^i xy^i z \in L$, $\forall\ i \geq 0$
  - if either v or y contains two or three symbols from {a, b, c} $\rightarrow uv^2xy^2z$ contains letters in wrong order
    - e.g.: $a(ab)^2bb(bc)^2c$
  - if both v and y contain one type of symbol from {a, b, c} $\rightarrow uv^ixy^iz$ can't contain equal number of 'a', b, c for some i
    - e.g.: $a(a)^3bb(c)^3c$

Closure properties

- The class of context-free languages is not closed under complementation or intersection
  - complementation was applied on DFA
    - DFA is equivalent with NFA
    - non-deterministic PDA (what we have used) is not equivalent with a deterministic PDA
  - remember that finite automata intersection property used complementation

Closure properties

- Theorem: context-free languages are not closed under intersection
- Proof by indirection:
  - suppose context-free languages are closed under intersection
  - $L_1 = \{a^n b^n c^m : m, n \geq 0\}$, $L_2 = \{a^m b^n c^n : m, n \geq 0\}$ are context-free
  - according the assumption $L_1 \cap L_2 = \{a^n b^n c^n : n \geq 0\}$ is also context free, but we have shown it is not

Closure properties

- Theorem: context-free languages are not closed under complementation
- Proof by indirection:
  - suppose languages are closed under complementation
  - we have already proved that context-free languages languages are closed under union
  - according to the De'Morgan identity and the previous two points context-free languages are closed under intersection which is not
    - $L_1 \cap L_2 = (L_1^C \cup L_2^C)^C$

Languages that are not context-free

- L = {$a^n b^m c^n d^m$: n ≥ 0} is not context-free
  - the subscripts are in a wrong order, you would need two stacks
- L = {wcw : |w| ≥ 0} is not context-free
  - PDA uses stack not queue

Summary

- CFG → PDA
- Simplicity
- PDA → CFG
- Closure properties
- Pumping theorem 2

Next time

- The definition of a Turing machine

Elements of the Theory of Computation

Lesson 10

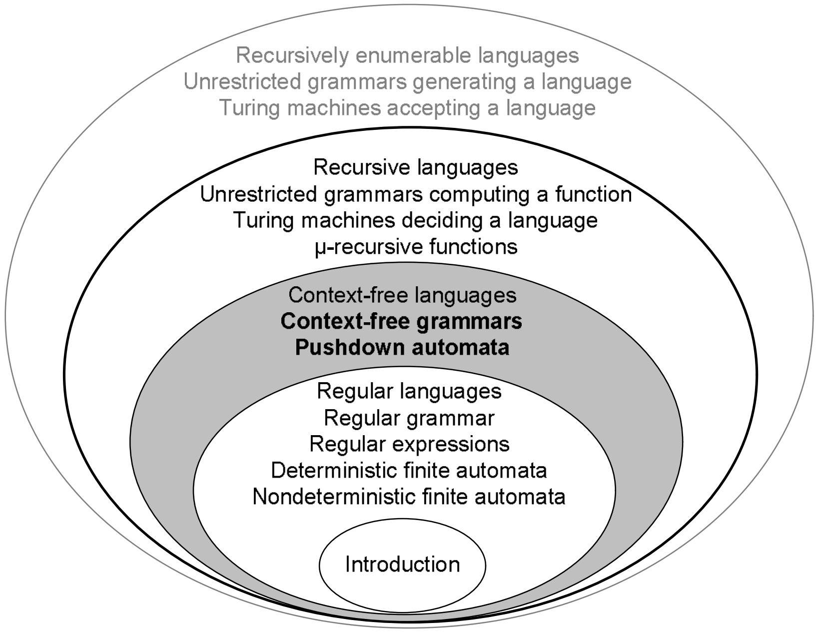
4.1. The definition of a Turing machine

University of Pannonia

Dr. István Heckl, Istvan.Heckl@gmail.com

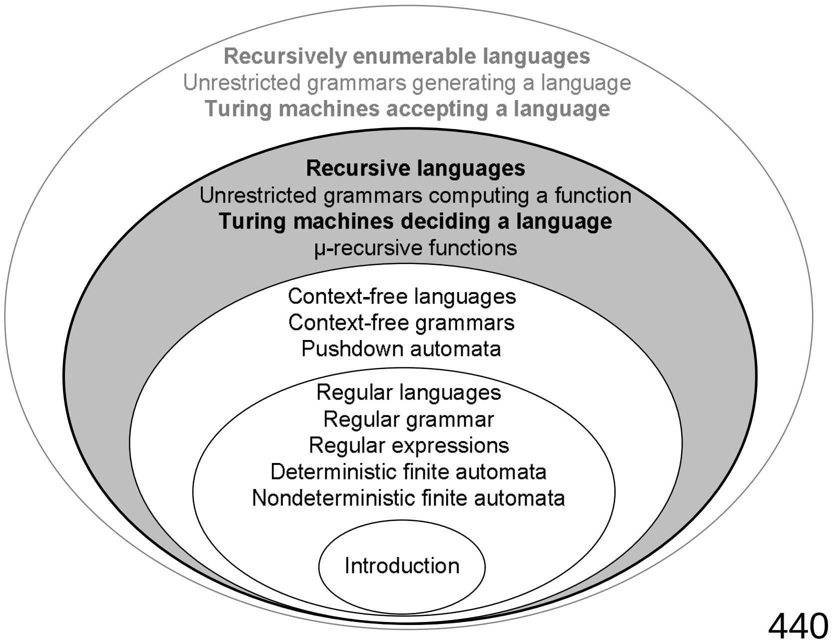
Last time

- CFG → PDA
- Simplicity
- PDA → CFG
- Closure properties
- Pumping theorem 2



The definition of a Turing machine

- Turing machine, TM
- Configuration
- Yield in one step
- Computation
- Yield
- Machine schema
- The basic machines
- Tape
- Other important machines



Turing machine, TM

- Alan Turing (1912 –1954)
  - English mathematician, logician, cryptanalyst, and computer scientist
  - he was highly influential in the development of computer science
  - providing a formalization of the concepts of "algorithm" with the Turing machine

Turing machine, TM

- We have seen that some language cannot be accepted by PDA, e.g.:
  - $L = \{a^n b^n c^n : n \geq 0\}$
  - $L = \{a^n : n \text{ is prime}\}$
  - L = {w $\in \Sigma$: w has an equal number of 'a', b and c}
- Let us enhance the PDA to be able accept the previous languages

Turing machine

- We will see that TM is the strongest automaton in terms of computing power
  - any computation that can be carried out on a fancier type of automaton can be also carried out on a TM
- TM is designed to satisfy simultaneously the following criteria:
  - should be automata
  - should be simple to define formally and reason about
  - should be the strongest in terms of computing power

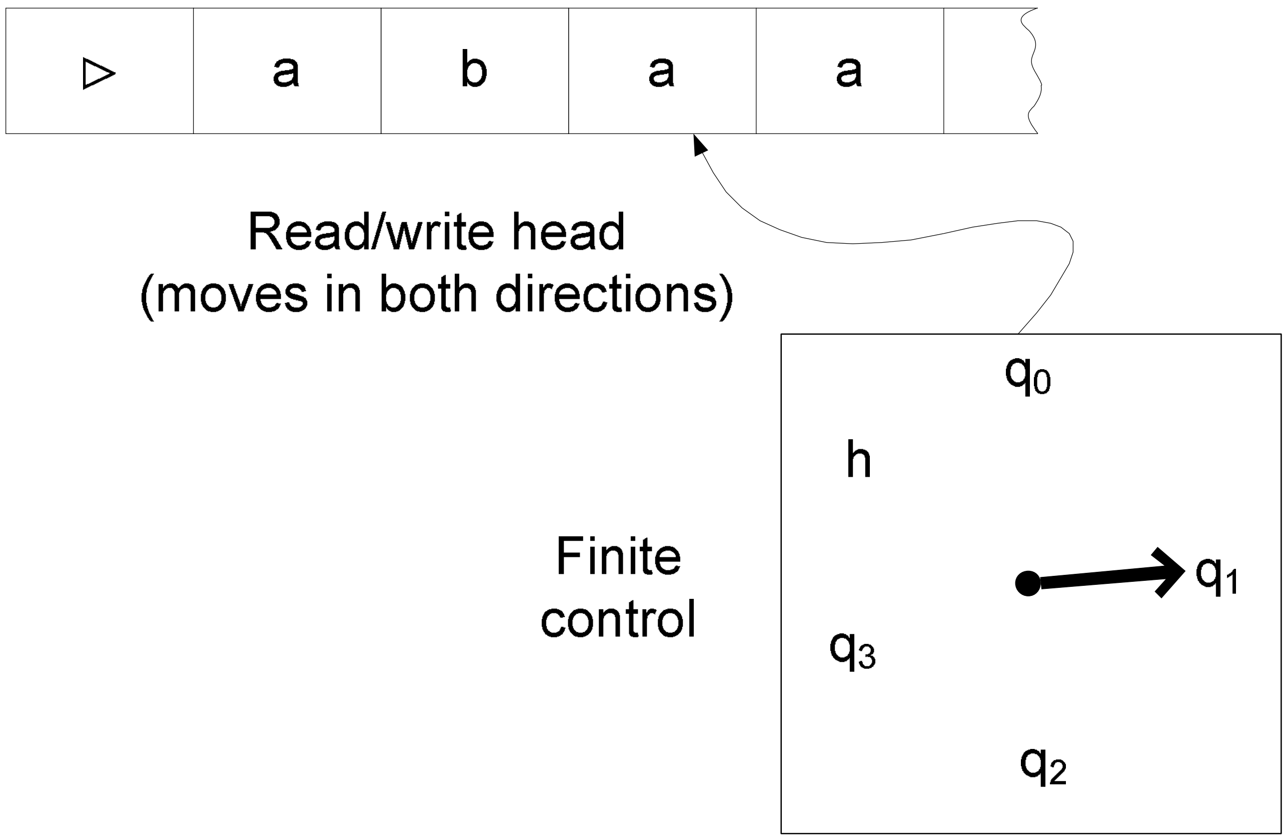
Turing machine

- Components of a TM:
  - finite-state control unit
  - tape, infinite to the right
  - head for reading and writing, able to move in both directions
- Differences between PDA and TM:
  - the head of a TM can move to the left
  - TM can write on the tape
  - TM does not have a stack
    - though it can store data in the tape

Turing machine in action

- http://www.youtube.com/watch?v=cYw2ewoO6c4
- http://www.youtube.com/watch?v=E3keLeMwfHY

Turing machine



Turing machine

- Operation of a TM:
  - the control unit operates in discrete steps
  - each step performs two functions:
    - put the control unit in a new state
    - either:
      - write a new symbol
        - » may be the same as the old one
      - move the head one tape square to the left or to the right
  - if a halting state is encountered then the TM stops
    - does not matter if the whole input is read or not
    - NFA can go on from final state

Turing machine

- Special symbols
  - $\leftarrow$, $\rightarrow$ denote the movement of the head
    - these symbols are not members of any alphabet we consider
  - $\rhd$ marks the leftmost end of the tape
    - when the head reads a $\rhd$, it immediately moves to the right
    - $\rhd \in \Sigma$
  - $\sqcup$ marks the blank symbol
    - the end of the tape is filled with $\sqcup$
    - $\sqcup \in \Sigma$

Turing machine

- Definition of Turing machine M: a quintuple $(K, \Sigma, \delta, s, H)$, where
  - K set of states (finite)
  - Σ alphabet (finite)
    - containing $\sqcup$, $\triangleright$, not containing $\leftarrow$, $\rightarrow$
  - $s \in K$ the initial state
  - $H \subseteq K$ the set of halting states (finite)
    - some say there is only one halting state
  - δ transition function, $(K - H) \times \Sigma \rightarrow K \times (\Sigma \cup \{\leftarrow, \rightarrow\})$ such that
    - $\forall\, q \in K - H$, if $\delta(q, \triangleright) = (p, b) \;\rightarrow\; b = \rightarrow$
    - $\forall\, q \in K - H$, $a \in \Sigma$, if $\delta(q, a) = (p, b) \;\rightarrow\; b \neq \triangleright$

Turing machine

- TM is deterministic
- TM stops only when the machine enters a halting state
- $\triangleright$ appears only at the left end of the tape
  - it is never erased
  - TM never writes $\triangleright$

Example

- Create TM M which changes all 'a' to ⊔ as it goes to the right, until it finds a tape square already containing ⊔!
  – changing a nonblank symbol to the blank symbol is called erasing
  – M = (K, Σ, δ, s, {h}), where
    - $K = \{q_0, q_1, h\}$
    - Σ = {a, ⊔ , ▹}
    - $s = q_0$
    - δ is given by the following table

Example

q	σ	$\delta(q, \sigma)$
q_0	a	$(q_1, \sqcup)$
q_0	$\sqcup$	$(h, \sqcup)$
q_0	$\rhd$	$(q_0, \rightarrow)$
q_1	a	(q_0, a)
q_1	$\sqcup$	$(q_0, \rightarrow)$
q_1	$\rhd$	$(q_1, \rightarrow)$

- Notice that in state q_1 the input symbol is always blank nonetheless $\delta(q_1, a)$ must be defined as the domain of δ is $(K - H) \times \Sigma$

Example

- Create TM M which scans to the left until it finds ⊔, if starts from ⊔ then halt at once!
  - M = (K, Σ, δ, s, H), where
    - K = {q_0, h}
    - Σ = {a, ⊔, ▹}
    - s = q_0
    - H = {h}
    - δ is given by the following table

Example

q	σ	$\delta(q, \sigma)$
q_0	a	$(q_0, \leftarrow)$
q_0	$\sqcup$	$(h, \sqcup)$
q_0	$\rhd$	$(q_0, \rightarrow)$

- Unlike the previous automata, M may never stops
  - it happens if there is no $\sqcup$ to the left
  - in that case the head goes back and forth between the first and second symbol of the tape

Configuration

- Definition of a configuration of TM $M = (K, \Sigma, \delta, s, H)$: an ordered triple of the current state of M and the whole tape
  - the tape is partitioned into 2 parts
    - until the head (including the head)
    - after the head
  - it is an element of $K \times \triangleright\Sigma^* \times (\Sigma^*(\Sigma - \{\sqcup\}) \cup \{e\})$
    - the head position is defined by the second and third components

Configuration

– the description of the tape always starts with ▹ and never ends with ⊔

  • (q, ▹baa, bc⊔), (q, ⊔aa, ba) are not valid configurations

– the last character of the second element of the configuration is the head position

  • e.g.: (q, ▹a, aba), (h, ▹⊔⊔⊔, ⊔a), (q, ▹⊔a⊔b, *e*) the head is on 'a', ⊔, b respectively

Configuration

- the second and third component of the configuration may be merged then the head position is marked with underline
  - (q, wa, u) = (q, w$\underline{a}$u)
  - (q, ▷⊔a⊔⊔, *e*) = (q, ▷⊔a⊔$\underline{⊔}$)
- halted configuration: such configuration in which the state component is in H
- some partitions the tape into 3 parts: before the head, under the head, after the head

Yield in one step

- Definition of yield in one step of a TM, $|-_M$: a relation between two "neighboring" configurations
  - let
    - M = (K, Σ, δ, s, H) be a Turing machine
    - $(q_1, w_1\underline{a}_1u_1)$, $(q_2, w_2\underline{a}_2u_2)$ are configurations of M, $a_1, a_2 \in \Sigma$
  - then $(q_1, w_1\underline{a}_1u_1) |-_M (q_2, w_2\underline{a}_2u_2)$ if and only if

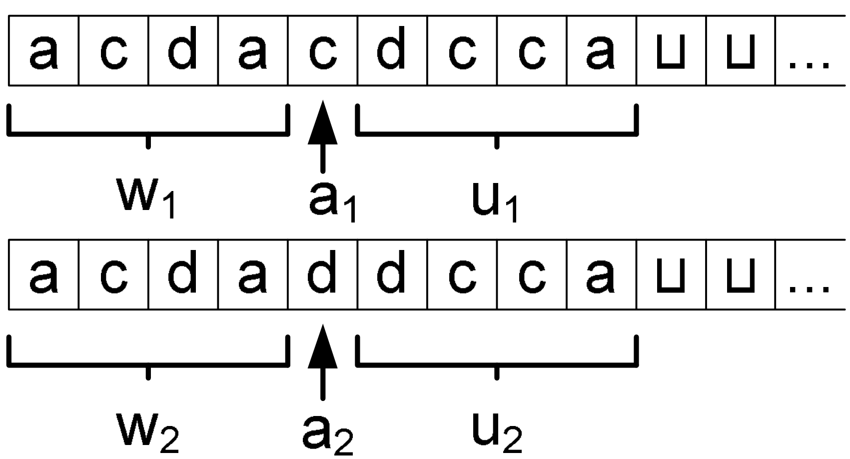
$(q_1, w_1\underline{a}_1u_1) \vdash_M (q_2, w_2\underline{a}_2u_2)$

Yield in one step

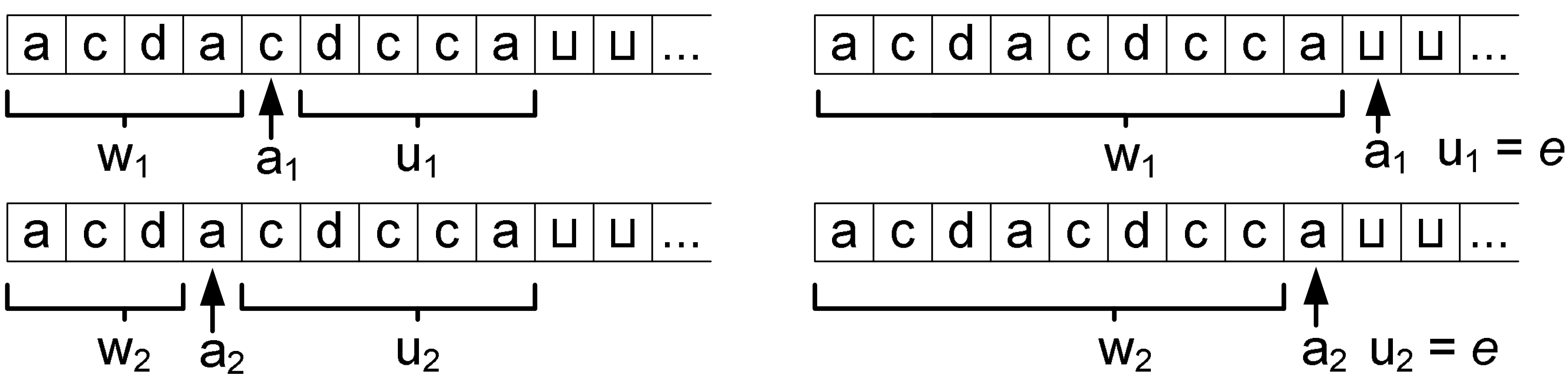
- $\exists\ \delta(q_1, a_1) = (q_2, b)$, $b \in \Sigma \cup \{\leftarrow, \rightarrow\}$, and one of the following holds:
  - $b \in \Sigma$, $w_1 = w_2$, $u_1 = u_2$, $a_2 = b$
  - $b = \leftarrow$, $w_1 = w_2a_2$, either
    - $u_2 = a_1u_1$, if $(a_1 = \sqcup, u_1 = e)^C$ or
    - $u_2 = e$, if $a_1 = \sqcup$, $u_1 = e$
  - $b = \rightarrow$, $w_2 = w_1a_1$, either
    - $u_1 = a_2u_2$, if $u_1 \neq e$
    - $u_1 = u_2 = e$, $a_2 = \sqcup$, if $u_1 = e$

Yield in one step

- Let $a \in \Sigma$, $w, u \in \Sigma^*$, u does not end with ⊔; the yield in one step relation may hold between two configurations if
  - M rewrites a symbol without moving its head
    - $b \in \Sigma$, $w_1 = w_2$, $u_1 = u_2$, $a_2 = b$
    - $\delta(q_1, c) = (q_2, d)$, a_1 = 'c', b = 'd'
    - e.g.: $(q_1, \triangleright acda\underline{c}dcca) \vdash_M (q_2, \triangleright acda\underline{d}dcca)$

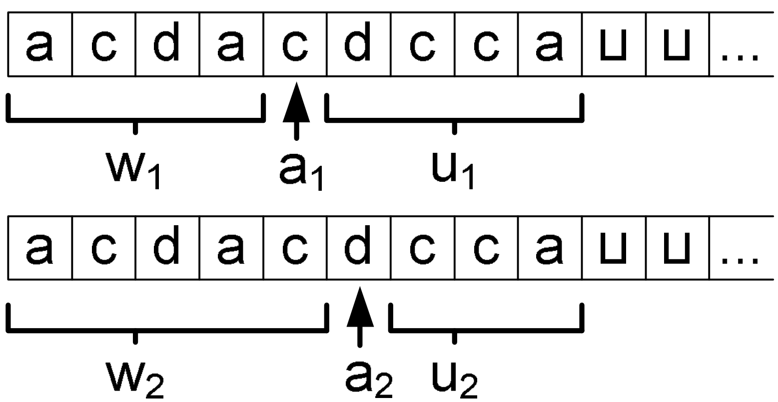


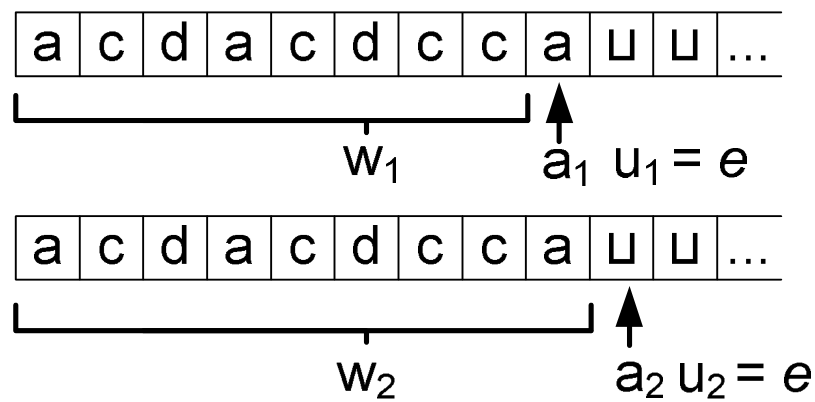
- M moves its head one square to the left
  - $b = \leftarrow$, $w_1 = w_2a_2$, either
    - $u_2 = a_1u_1$, if $(a_1 = \sqcup, u_1 = e)^C$ is true, or
    - $u_2 = e$, if $a_1 = \sqcup$, $u_1 = e$
  - $\delta(q_1, c) = (q_2, \leftarrow)$
  - e.g.: $(q_1, \triangleright \text{acda}\underline{\text{c}}\text{dcca}) \vdash_M (q_2, \triangleright \text{acd}\underline{\text{a}}\text{cdcca})$
  - $w_2a_2u_2$ can be shorter than $w_1a_1u_1$
    - e.g.: $(q_1, \triangleright \text{acdacdcca}\underline{\sqcup}) \vdash_M (q_2, \triangleright \text{acdacdcc}\underline{\text{a}})$



– M moves its head one square to the right

- $b = \rightarrow$, $w_2 = w_1 a_1$, either
  - $u_1 = a_2 u_2$, if $u_1 \neq e$
  - $u_1 = u_2 = e$, $a_2 = \sqcup$, if $u_1 = e$
- $\delta(q_1, a) = (q_2, \rightarrow)$
- e.g.: $(q_1, \triangleright acda\underline{c}dcca) \vdash_M (q_2, \triangleright acdac\underline{d}cca)$
- $w_2 a_2 u_2$ can be longer than $w_1 a_1 u_1$
  - e.g.: $(q_1, \triangleright acdacdcc\underline{a}) \vdash_M (q_2, \triangleright acdacdcca\underline{\sqcup})$





Computation

- Definition of computation by TM M: a sequence of configuration C_0, C_1, ... C_n such that C_0 |- C_1 |- ... |- C_n
  - e.g.: (q_1, ▷$\underline{a}$baa) |- (q_2, ▷$\underline{b}$baa) |- (q_1, ▷b$\underline{b}$aa) |- (q_3, ▷b$\underline{a}$aa)
  - the length of a computation is the number of yield in one step operation applied
  - the first and the last configuration can be connected with the yield in n steps relation, denoted as $|-^n$
    - e.g.: (q_1, ▷$\underline{a}$baa) $|-^3$ (q_3, ▷b$\underline{a}$aa)

Yield

- Definition of yield of a TM, $|\text{-}_M{}^*$: the reflexive, transitive closure of $|\text{-}_M$
  - if (q', w', u') can be reached from (q, w, u) through a number of yield in one step operation then the yield operation holds between (q, w, u) and (q', w', u')
  - denote as: (q, w, u) |-* (q', w', u')

Example

- Consider again TM M which changes all 'a' to ⊔ as it goes to the right, until it finds a tape square already containing ⊔!
  - $(q_1, \triangleright\underline{\sqcup}aaaa)$, $(q_0, \triangleright\sqcup\underline{a}aaa)$, $(q_1, \triangleright\sqcup\underline{\sqcup}aaa)$, $(q_0, \triangleright\sqcup\sqcup\underline{a}aa)$, $(q_1, \triangleright\sqcup\sqcup\underline{\sqcup}aa)$ is a computation of length 4
  - $(q_1, \triangleright\underline{\sqcup}aaaa) \vdash^* (q_1, \triangleright\sqcup\underline{\sqcup}aaa)$
  - $(q_1, \triangleright\underline{\sqcup}aaaa) \vdash^3 (q_0, \triangleright\sqcup\sqcup\underline{a}aa)$
  - $(q_1, \triangleright\underline{\sqcup}aaaa) \vdash^5 (q_1, \triangleright\sqcup\sqcup\underline{\sqcup}aa)$
  - $(q_1, \triangleright\underline{\sqcup}aaaa) \vdash^2 (q_0, \triangleright\sqcup\underline{a}aaa)$

Machine schema

- Defining TM as a quintuple is cumbersome and hard to understand
  - the table of the transition function is usually big
- A machine schema is such a TM which is constructed using already defined TMs as building blocks
- The notation is similar to the state diagram but instead of states the already defined TMs appear as nodes
  - the arrows connecting the sub-TMs tell which sub-TM is to start after the current one stopped

The basic machines

- Symbol writing and head moving machines:
  - for each $a \in \Sigma \cup \{\rightarrow, \leftarrow\} - \{\triangleright\}$, we define TM $M_a = (\{s, h\}, \Sigma, \delta, s, \{h\})$
  - $\forall\, b \in \Sigma - \{\triangleright\}$, $\delta(s, b) = (h, a)$
    - but $\delta(s, \triangleright) = (s, \rightarrow)$
  - symbol writing machine if $a \in \Sigma$
  - head moving machine if $a \in \{\rightarrow, \leftarrow\}$
  - these machines perform one step and halt
    - except $M_{\leftarrow}$ if it started from the second head position, right after $\triangleright$
  - shorthand: $M_a = a$, $M_{\leftarrow} = L$, $M_{\rightarrow} = R$

The basic machines, a

q	σ	δ(q, σ)
s	a	(h, a)
s	b	(h, a)
s	c	(h, a)
s	⊔	(h, a)
s	⊳	(h, →)

The basic machines, b

q	σ	δ(q, σ)
s	a	(h, b)
s	b	(h, b)
s	c	(h, b)
s	⊔	(h, b)
s	⊳	(h, →)

The basic machines, c

q	σ	$\delta(q, \sigma)$
s	a	(h, c)
s	b	(h, c)
s	c	(h, c)
s	$\sqcup$	(h, c)
s	$\triangleright$	(h, $\rightarrow$)

The basic machines, L

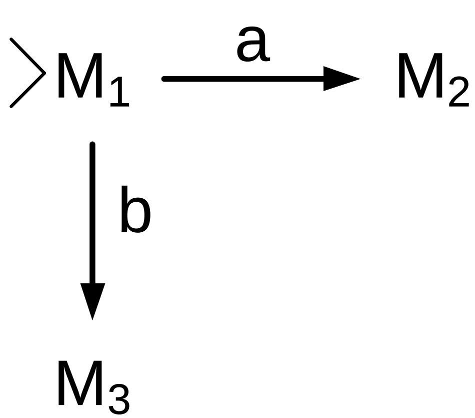
q	σ	δ(q, σ)
s	a	(h, ←)
s	b	(h, ←)
s	c	(h, ←)
s	⊔	(h, ←)
s	⊳	(h, →)

The basic machines, R

q	σ	$\delta(q, \sigma)$
s	a	(h, $\rightarrow$)
s	b	(h, $\rightarrow$)
s	c	(h, $\rightarrow$)
s	$\sqcup$	(h, $\rightarrow$)
s	$\rhd$	(h, $\rightarrow$)

Example

- Operation of M:
  - start at M_1, operate as M_1 would until M_1 would halt
    - if the currently scanned symbol is 'a', initiate M_2 and operate as M_2 would operate
    - if the currently scanned symbol is b, initiate M_3 and operate as M_3 would operate
    - if the currently scanned symbol is neither 'a' nor b then halt
  - if M_2 or M_3 would halt then M halt



Machine schema

- Definition of machine schema: a triplet $M = (m, \eta, M_s)$, where
  - m set of TMs (finite)
    - common alphabet Σ and disjoint sets of states
    - $m = \{M_1, M_2, ..., M_n\}$
      - $M_i = (K_i, \Sigma, \delta_i, s_i, H_i)$
  - $\eta \in m \times \Sigma \times m$, defines the next TM
  - $M_s \in m$, starting TM

Machine schema

- $M = (m, \eta, M_s) = (K, \Sigma, \delta, s, H)$
  - $K = K_0 \cup ... \cup K_n \cup \{r_0, r_1, \ldots, r_n, h\}$
    - $r_0, r_1, \ldots, r_n$ are new states
    - $|m| = n$
  - $s = s_s$
  - $H = \{h\}$
    - h is a new state

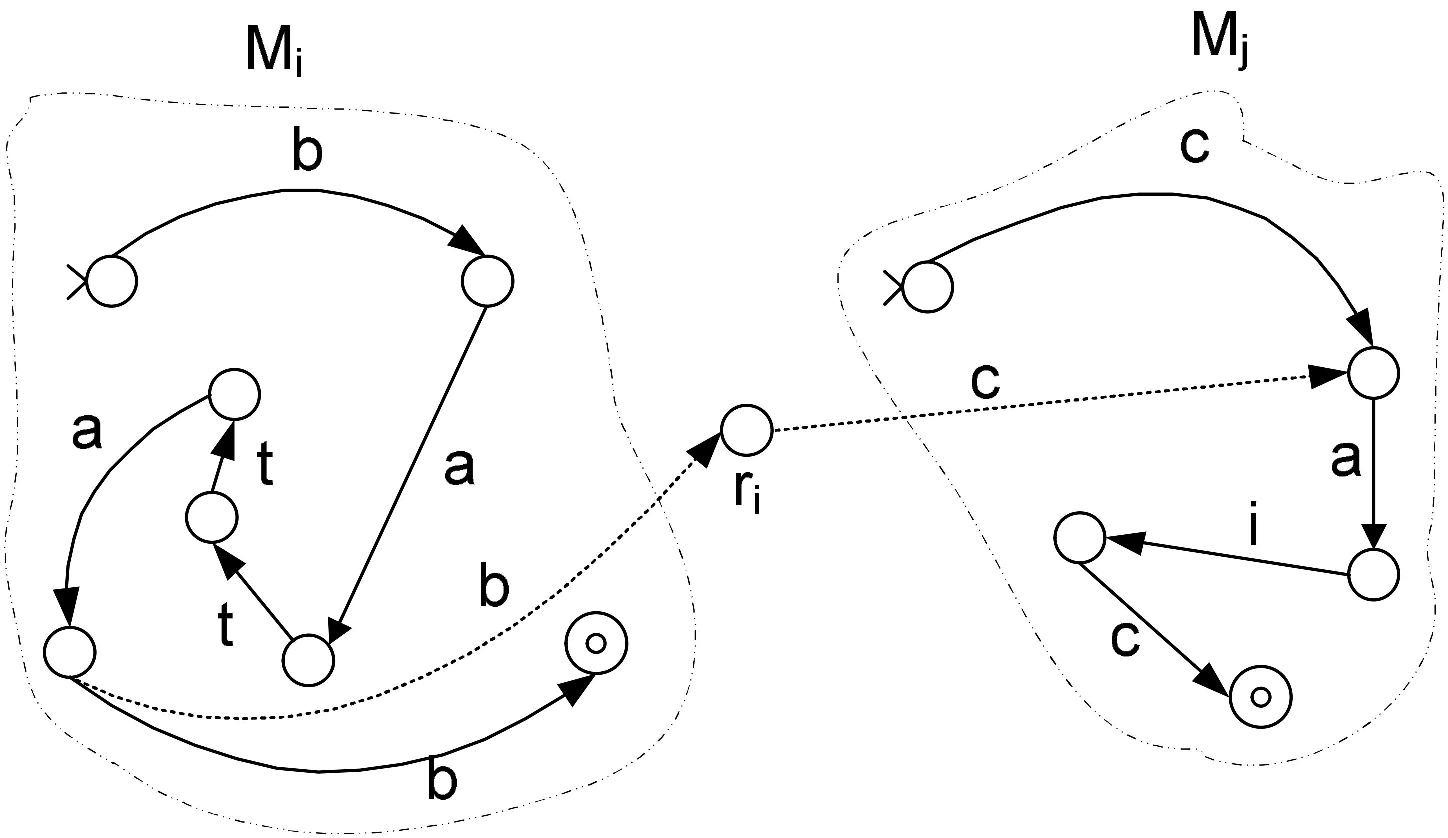
Machine schema

- δ
  - when imitating M_i
    - if $q \in K_i$, $a \in \Sigma$, $\delta_i(q, a) = (p, b)$, $p \notin H_i$
    - then $\delta(q, a) = (p, b)$
  - instead of halting M_i we go to a new state
    - if $q \in K_i$, $a \in \Sigma$, $\delta_i(q, a) = (p, b)$, $p \in H_i$
    - then $\delta(q, a) = (r_i, b)$
      - » reading the last symbol of what M_i would have read

Machine schema

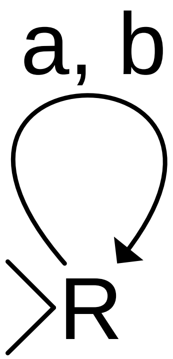
- if η don't define a new TM then M halts
  - if $r_i \in K$ (r_i is a new state), $a \in \Sigma$, $\eta(M_i, a)$ is not defined
  - then $\delta(r_i, a) = (h, a)$
- if η defines a new TM then M starts to imitate it
  - if $r_i \in K$ (r_i is a new state), $a \in \Sigma$, $\eta(M_i, a) = M_j$, $\delta_j(s_j, a) = (p, b)$
  - then (s_j is skipped because r_i acted as s_j)
    - » $\delta(r_i, a) = (p, b)$ if $p \notin H_j$
    - » $\delta(r_i, a) = (r_j, b)$ if $p \in H_j$

Machine schema



Example

- $R_\sqcup = (\{R\}, \eta, R)$ is a machine schema
  - $R = (\{q, h\}, \Sigma, \delta_R, q, \{h\})$
    - $\delta_R(q, a) = (h, R)$, for $\forall\, a \in \Sigma$
  - $\eta(R, a) = R$, $\eta(R, b) = R$,
    $\eta(R, \sqcup)$ = undefined
- $R_\sqcup = (\{q, r_0, h\}, \Sigma, \delta, q, \{h\})$ is a TM
  - r_0 is the new state
  - $\delta(q, a) = (r_0, R)$ for $a \in \Sigma$ (b rule)
  - $\delta(r_0, a) = (r_0, R)$ if $a \neq \sqcup$ (d2 rule)
    (h, a) if $a = \sqcup$ (c rule)



Tape

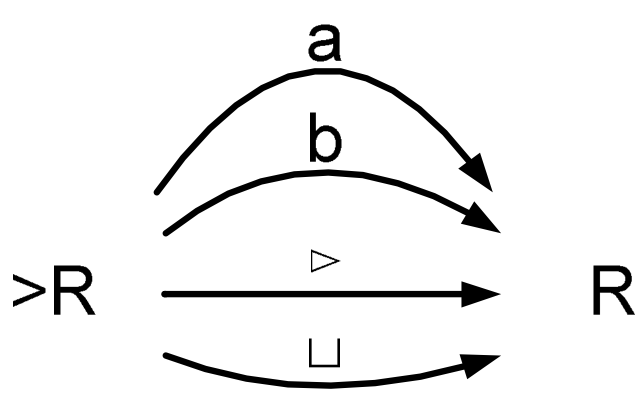
- When a schema transfer control from one TM to another the content of the tape and the position of the head does not change
- Standard form of a tape: the head is after the rightmost non-blank symbol
  - e.g.: $\sqcup w \underline{\sqcup}$ where $w \in (\Sigma - \sqcup)^*$

Tape

- At constructing machine schema it is useful to leave the tape in a standard form so another schema may assume that it can start from this standard form
  - not all machine apply this convention
- Definition of the initial configuration of TM M on input $w \in (\Sigma - \{\sqcup, \triangleright\})^*$: $(s, \triangleright\underline{\sqcup}w)$

Example

- Construct a machine schema which moves the head to the right by 2 squares!
  - the schema moves its head right one square then if that square contains an 'a', b, ▷, or ⊔, it moves its head one square further to the right



Example

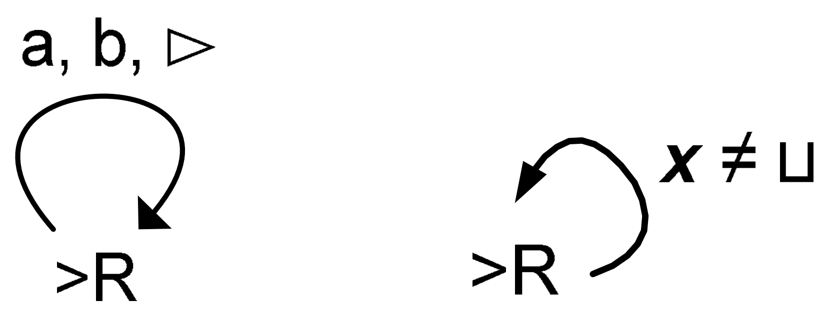
- An arrow labeled with several symbols is the same as several parallel arrows

$$>R \xrightarrow{a,b,\sqcup,\triangleright} R$$

- If an arrow is labeled by all symbols in the alphabet Σ, then the labels can be omitted, so M can be signed as
  - $>R \rightarrow R$
  - $>RR$
  - $>R^2$

Example

- Construct a machine schema that scans its tape to the right until it finds a blank!
  - denote this machine by $R_{\sqcup}$
  - we can eliminate multiple arrows and labels by using label $\boldsymbol{x} \neq \sqcup$ ($\boldsymbol{x}$ is not the letter 'x' but the currently read letter)

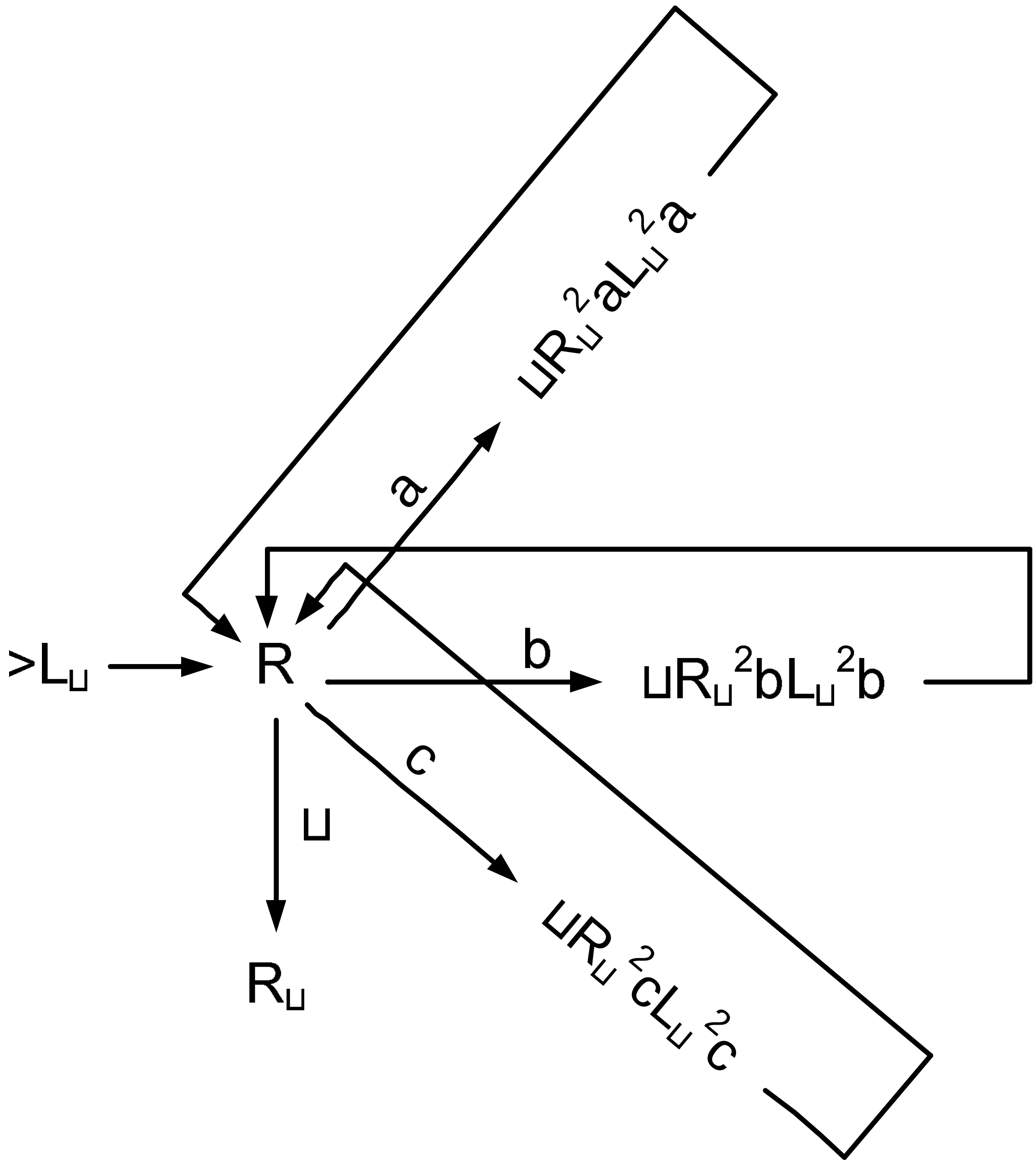


Example

- Construct a machine schema that scans its tape to the left until it finds a blank!
  - denote this machine by $L_{\sqcup}$
  - this TM never stops if there is no ⊔ to the left

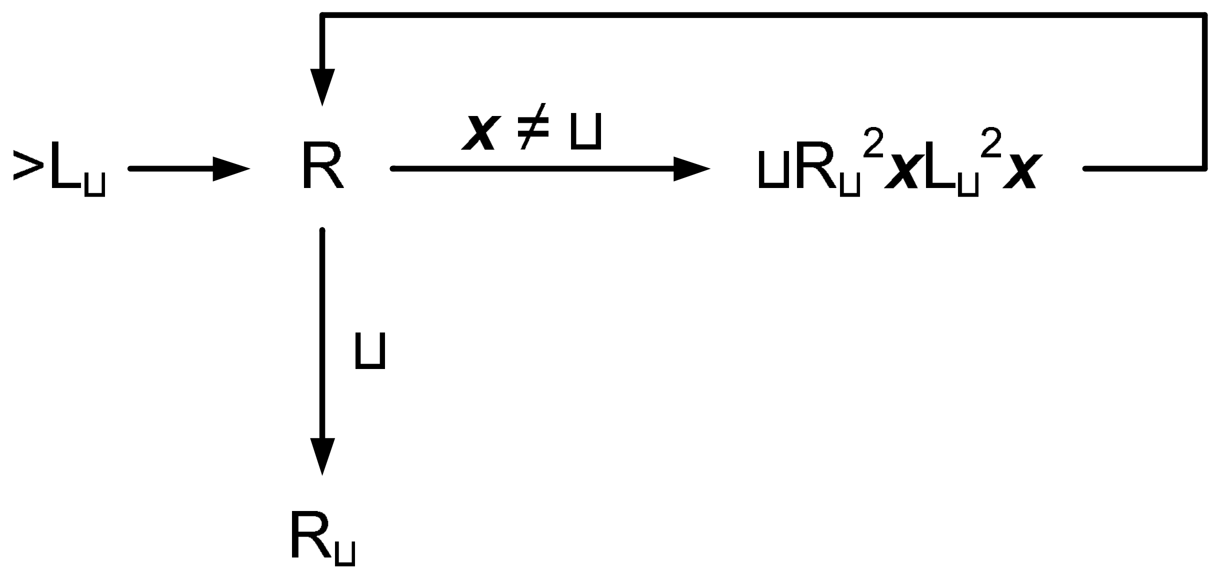
Copy machine

- Construct a machine schema that copies a string not containing ⊔!
  - denote this machine by C
  - C transforms ⊔w<u>⊔</u> into ⊔w⊔w<u>⊔</u>, ⊔ $\notin$ w
    - other strings may precede w
    - blank denotes the beginning of the string

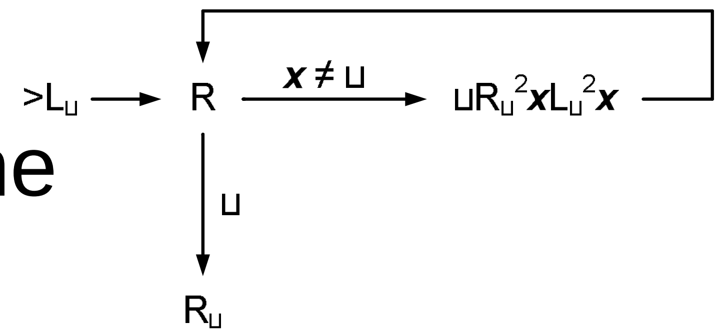

>L⊔
R
a
⊔R⊔²aL⊔²a
b
⊔R⊔²bL⊔²b
c
⊔R⊔²cL⊔²c
⊔
R⊔

Copy machine

– remember that C has several loop
  - in each loop there is concrete symbol writing machine instead of ***x***



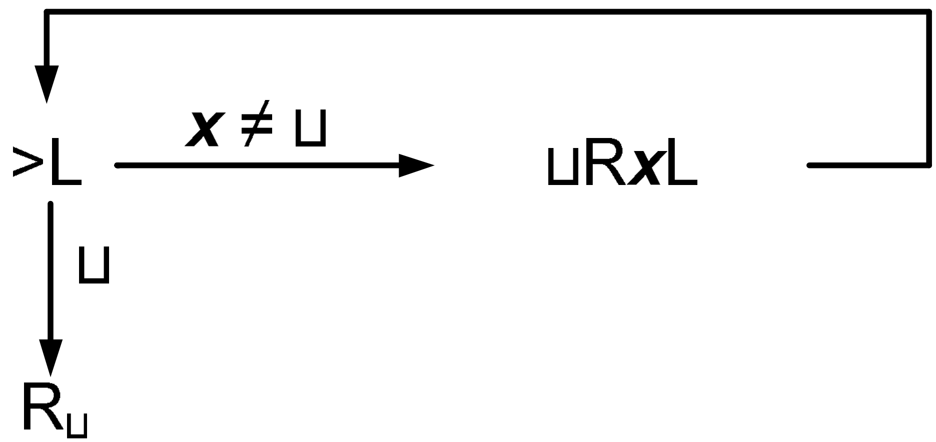
Copy machine



$\sqcup abc\underline{\sqcup}$	$\sqcup a\underline{b}c\sqcup a$	$\sqcup ab\sqcup\underline{\sqcup}ab\sqcup$
$\underline{\sqcup}abc\sqcup$	$\sqcup a\underline{\sqcup}c\sqcup a$	$\sqcup ab\sqcup\sqcup ab\underline{\sqcup}$
$\sqcup\underline{a}bc\sqcup$	$\sqcup a\sqcup c\underline{\sqcup}a$	$\sqcup ab\sqcup\sqcup ab\underline{c}$
$\sqcup\underline{\sqcup}bc\sqcup$	$\sqcup a\sqcup c\sqcup a\underline{\sqcup}$	$\sqcup ab\sqcup\underline{\sqcup}abc$
$\sqcup\sqcup bc\underline{\sqcup}$	$\sqcup a\sqcup c\sqcup a\underline{b}$	$\sqcup ab\underline{\sqcup}\sqcup abc$
$\sqcup\sqcup bc\sqcup\underline{\sqcup}$	$\sqcup a\sqcup c\underline{\sqcup}ab$	$\sqcup ab\underline{c}\sqcup abc$
$\sqcup\sqcup bc\sqcup\underline{a}$	$\sqcup a\underline{\sqcup}c\sqcup ab$	$\sqcup abc\underline{\sqcup}abc$
$\sqcup\sqcup bc\underline{\sqcup}a$	$\sqcup a\underline{b}c\sqcup ab$	$\sqcup abc\sqcup abc\underline{\sqcup}$
$\sqcup\underline{\sqcup}bc\sqcup a$	$\sqcup ab\underline{c}\sqcup ab$	
$\sqcup\underline{a}bc\sqcup a$	$\sqcup ab\underline{\sqcup}\sqcup ab$	

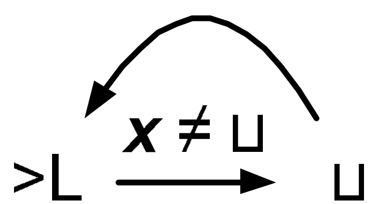
Example

- Construct a machine schema that shift a string not containing ⊔ to the right!
  - denote this machine by $S_{\rightarrow}$
  - $S_{\rightarrow}$ transforms ⊔w<u>⊔</u> into ⊔⊔w<u>⊔</u>, ⊔ ∉ w
    - other strings may precede w
    - blank denotes the beginning of the string



Example

- Construct a machine schema that deletes a string (not containing ⊔)!
  - D transforms ⊔w<u>⊔</u> into <u>⊔</u>, ⊔ ∉ w



Other important machines

- $S_{\leftarrow}$ shift a string to the left
- L_a find the first occurrence of 'a' to the left
- R_a find the first occurrence of 'a' to the right

Summary

- Turing machine, TM
- Configuration
- Yield in one step
- Computation, Yield
- Machine schema
- The basic machines, Tape
- Other important machines

Next time

- Computing with Turing machines

Elements of the Theory of Computation

Lesson 11

4.2. Computing with Turing machines

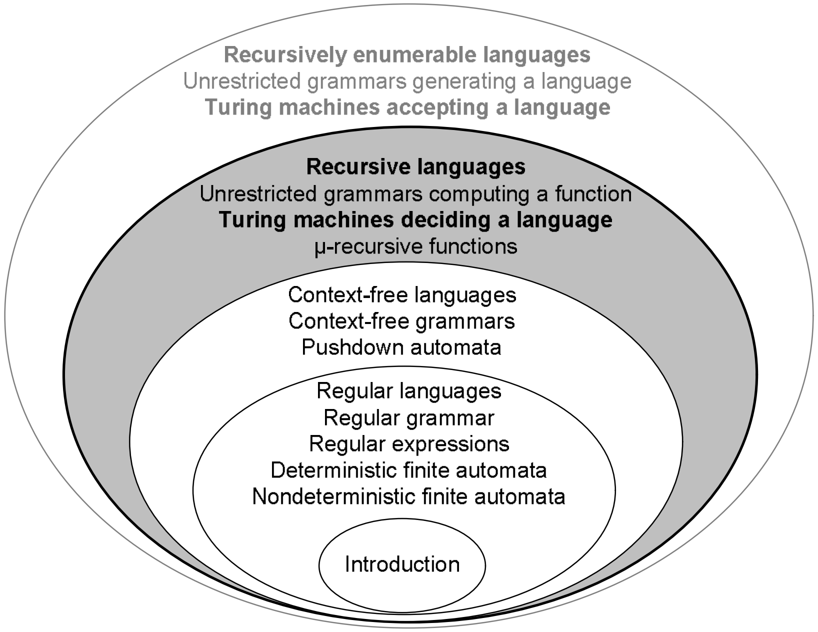
University of Pannonia

Dr. István Heckl, Istvan.Heckl@gmail.com

Version 47

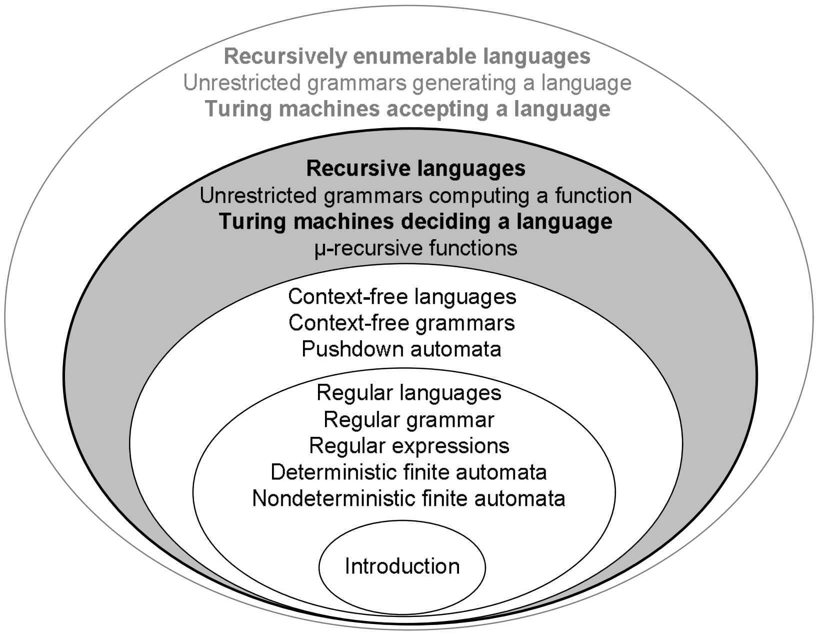
Last time

- Turing machine, TM
- Configuration
- Yield in one step
- Computation
- Yield
- Machine schema
- The basic machines
- Tape
- Other important machines



Computing with Turing machines

- Turing computable function
- Representation of numbers with strings
- String accepted by TM
- Language accepted by TM
- Turing acceptable
- Turing decidable
- Algorithm



Turing computable function

- Definition of the output of TM M on input w, M(w):
  if (s, $\triangleright\underline{\sqcup}w$) |-* (h, $\triangleright\underline{\sqcup}y$) $\rightarrow$ M(w) = y
  - $\Sigma_0 \subseteq \Sigma - \{\sqcup, \triangleright\}$, w, y $\in \Sigma_0^*$, h $\in$ H
  - M(w) is defined only if M halts on input w
    - it is supposed that M leaves the tape in a specified format
    - M(w) = ↗ if M fails to halt on input w
  - the output of DFA, NFA, and PDA was binary, they halted or not

Turing computable function

- Definition of Turing computable function, f: $\Sigma_0^* \rightarrow \Sigma_0^*$:
  $\exists$ TM M such that M(w) = f(w), $\forall$ w $\in \Sigma_0^*$
  - if M is started with input w, then when it halts, its tape contains f(w)
  - we say that M computes f
  - a Turing computable function is also called recursive function

Turing computable function

- Is κ Turing computable? If it is, give the TM which computes it!
  - $\kappa: \Sigma^* \rightarrow \Sigma^*, \kappa(w) = ww$
  - κ is computed by $R_{\sqcup} C S_{\leftarrow}$
    - position the head: $\triangleright\underline{\sqcup}w \rightarrow \triangleright\sqcup w\underline{\sqcup}$
    - copy the string: $\triangleright\sqcup w\underline{\sqcup} \rightarrow \triangleright\sqcup w\sqcup w\underline{\sqcup}$
    - shift the copied string: $\triangleright\sqcup w\sqcup w\underline{\sqcup} \rightarrow \triangleright\sqcup ww\underline{\sqcup}$

Representation of numbers with strings

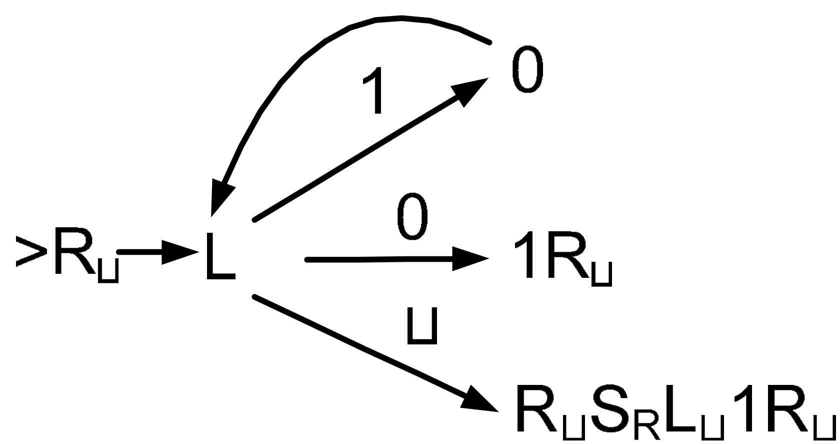
- Unary representation
  - one type of symbol is used to describe any number
  - numUni: $\{I\}^* \rightarrow N$, numUni(I^n) = n
    - e.g.: numUni(III) = numUni(I^3) = 3
- Binary representation
  - numBin: $0 \cup 1\{1, 0\}^* \rightarrow N$,
    numBin($a_1 a_2 \ldots a_n$) = $a_1 2^{n-1} + a_2 2^{n-2} + \ldots + a_n$
    - e.g.: numBin(110) = 1*4+1*2+0 = 6

Turing computable function

- Definition of Turing computable function, $f: N^k \rightarrow N$: $\exists$ TM M such that $\forall\ w_1,\ldots,w_k \in \Sigma^*$, $num(M(w_1;\ldots;w_k)) = f(num(w_1), \ldots, num(w_k))$
  - e.g.: $num(M_{add}("5"; "3")) = add(num("5"), num("3"))$
  - if M is started with the representations of the integers $n_1,\ldots,n_k$ as input, then when it halts, its tape contains a string that represents number $f(n_1 ,\ldots, n_k)$
  - we say that M computes f
  - a Turing computable function is also called recursive function

Example

- Is succ Turing computable? If it is, give the TM which computes it!
  – succ: N → N, succ(n) = n + 1



Example

- Operation:
  - M finds the right end of the input
  - goes to the left as long as it sees 1, changing all of them to 0
  - when M sees a 0, it changes it into 1, goes to the right and halts
  - if M sees ⊔ while looking for 0 then it shifts the whole string one position to the right, writes 1 at the left end, goes to the right end, and halts

String accepted by TM

- Definition of string accepted by TM: $w \in \Sigma^*$ is accepted by M if $(s, \triangleright\underline{\sqcup}w)$ |-* (h, x, y), $w \in \Sigma_0^*$, $h \in H$
  - w is accepted if M the computation halts
  - w is rejected if M the computation never halts
    - e.g.: M is in an infinite loop
  - $\Sigma_0 \subseteq \Sigma - \{\sqcup, \triangleright\}$
  - the value of x and y is unimportant

Language accepted by TM

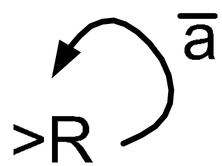
- Definition of language accepted by TM M, L(M):
  $L(M) = \{w \in \Sigma_0^* : (s, \triangleright\underline{\sqcup}w) \vdash^* (h, x, y), w \in \Sigma_0^*, h \in H\}$
  - the set of strings accepted by M
  - $\Sigma_0 \subseteq \Sigma - \{\sqcup, \triangleright\}$

Turing acceptable

- Definition of Turing acceptable language, L: $\exists$ TM M such that L = L(M)
  - we say M accepts or semi-decides L
  - a Turing acceptable language is also called recursively enumerable
  - M halts for $\forall$ w $\in$ L, M(w) = $\nearrow$ for $\forall$ w $\notin$ L

Example

- Is L Turing acceptable? If it is, give the TM which accepts it!
  - L = {w $\in$ {a, b}* : w contains at least one 'a'}



- Operation:
  - M scans right until 'a' is encountered and then halts
  - if no 'a' is found, the machine goes on forever into blanks that follow its input

Turing decidable

- Definition of Turing decidable language, L: $\exists$ TM M such that
  $\forall$ w $\in$ L, (s, $\rhd\underline{\sqcup}$w) |-* (h, $\rhd\sqcup$Y$\underline{\sqcup}$),
  $\forall$ w $\notin$ L, (s, $\rhd\underline{\sqcup}$w) |-* (h, $\rhd\sqcup$N$\underline{\sqcup}$), h $\in$ H
  - Y and N are new symbols
  - we say M decides L
  - M always halts, L(M) = Σ*
  - a Turing decidable language is also called recursive

Turing decidable

- Turing decidable can be also defined by introducing two new halting states: y, n
  - accepting configuration: its halting state is y
    - M accepts w if the initial configuration yields an accepting configuration
  - rejecting configuration: its halting state is n
    - M rejects w if the initial configuration yields a rejecting configuration
  - M decides a language L if for $\forall$ w $\in$ Σ_0^*
    - if w $\in$ L then M accepts w
    - if w $\notin$ L then M rejects w

Characteristic function

- Definition of the characteristic function, χ_L of language L: $\chi_L(w) = Y$ if $w \in L$, $\chi_L(w) = N$ otherwise
  - $\Sigma_0 \subseteq \Sigma - \{\sqcup, \triangleright\}$, $L \subseteq \Sigma_0^*$
  - $\chi_L: \Sigma_0^* \rightarrow \{Y, N\}$
    - χ_L - Greek chi
    - $Y, N \notin \Sigma_0$
  - e.g.: $\chi_{\{aa, bb, cc\}}(aa) = Y$, $\chi_{\{aa, bb, cc\}}(ab) = N$
- Theorem: a Turing machine with 2 or more tapes is equivalent with a simple Turing machine

Turing decidable

- Theorem: L is Turing decidable $\leftrightarrow$ χ_L Turing computable
- Proof:
  - if L is decided by M (which is $w \in L \rightarrow M(w) = Y$)
  - then $\chi_L(w) = Y$, so the same M computes χ_L (which is $M(w) = \chi_L(w)$)
- Theorem: $f: \Sigma_0^* \rightarrow \Sigma_0^*$ is Turing computable $\leftrightarrow$ $L_f = \{ x,f(x) : x \in \Sigma_0^*\}$ is Turing decidable
  - e.g.: f = plus1, L_{plus1} = { "0,1", "1,2", "2,3", ...}

Turing decidable

- Proof: →, example
  - L_f = {opposite pairs, e.g.: black-white, good-bad, ...}
  - Adam can give you the opposite for a given word
  - Bell can decide if a pair, e.g., boy-tablet, is in L_f in the following way
    - she asks Adam about the opposite of boy, it is girl
    - girl is not tablet, so boy-tablet is not in L_f

Turing decidable

- Proof: →
  - suppose M computes f
  - M' (a 2-tape TM) decides L_f in the following way
    - search for ',' on the tape 1, T1: x,z, T2: *e*
    - move z to tape 2, T1: x, T2: z
    - simulate M on tape 1, T1: M(x), T2: z
      - M(x) = f(x) because M computes f
    - compare tape 1 and 2, (f(x) and z), write Y if they match, write N otherwise

Turing decidable

- Proof: ←, example
  - idea: all possible output is considered so M' will halt eventually
  - L_f = {w-w^R, e.g., ba-ab, baa-aab, ...}
  - Bell can decide if a pair is in L
  - Adam can compute the reverse of a word, e.g., ab, in the following way
    - Adam systematically asks Bell about each potential pair, where the first component is given, e.g., ab-*e*, ab-a, ab-b, ab-aa, ab-ab, ab-ba

Turing decidable

- Proof: ←
  - suppose M decides L_f
  - M' (a 3-tape TM) computes f in the following way
    - write w to T2 (original w, never changes)
    - initialize T3 with *e* (the result possibilities)
    - copy T2 ∘ , ∘ T3 to T1 (the work tape)
    - simulate M on T1
    - if M says Y → copy T3 to T1 and halt
    - T3 := lexicographically next string of T3
      - *e*, a, b, aa, ab, bb, aaa, ...
    - go back to give new value to T1

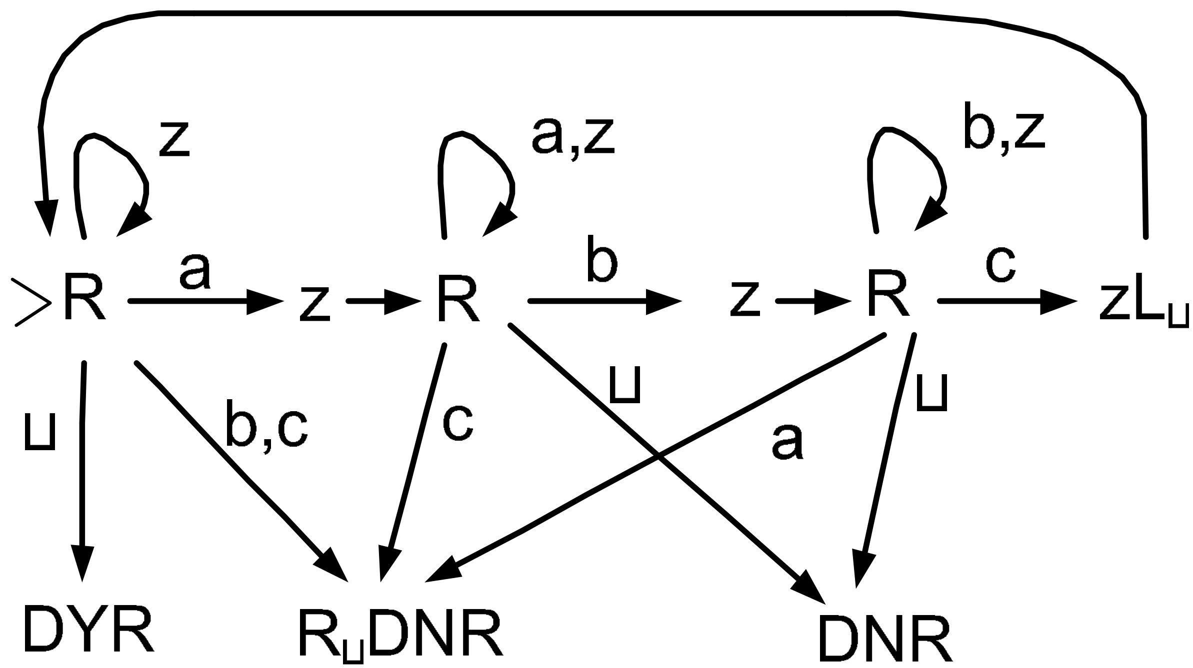
Turing decidable

- For each language L there is an equivalent function χ_L
- For each function f there is an equivalent language {x,f(x)}
- Corollary: Turing computable functions and Turing decidable languages are equivalent
  – recursive functions and recursive languages are equivalent
  – functions are an alternative way to describe languages (see μ-recursive functions)

Example

- Construct a machine schema that decides
  $L = \{a^n b^n c^n : n \geq 0\}$!
  - we proved that L is not context-free

Example



Example

- Operation:
  - on input $a^n b^n c^n$ it will operate in n stages
  - in each stage
    - M starts from the left end of the string
    - moves to the right in search of 'a'
    - when it finds 'a', it replaces it by z
    - looks further to the right for b
    - when it finds b, it replaces it by z
    - looks further to the right for c
    - when it finds c, it replaces it by z
    - returns to the left end of the input

Example

- Operation:
  - if at any point the machine schema does not find the proper symbol then delete the input and write N to the tape
    - e.g.: if M finds b when looking for 'a' → there is more b than 'a'
- It is easy to construct such a TM which accepts $L = \{a^n b^n c^n d^n : n \geq 0\}$

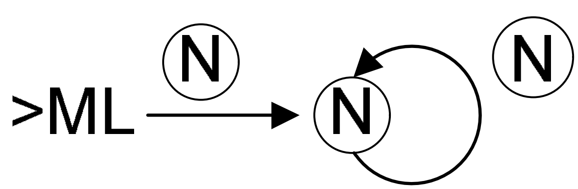
Algorithm

- Description of algorithm: a finite set of well-defined instructions for accomplishing some task which will terminate after a final number of steps
  - there is no formal definition
- TM M that accepts a language L cannot be usefully employed for telling whether w is in L
  - reason: if $w \notin L$ $\rightarrow$ we will never know when we have waited enough for an answer
  - M is not a representation of an algorithm
- TM M that decides a language L can be perceived as an algorithm

Turing decidable languages

- Theorem: if language L is Turing decidable → L is Turing acceptable
- Proof:
  - TM M decides L
  - the machine schema below accepts L
  - if M results in $\triangleright \sqcup Y \underline{\sqcup}$ → the schema halts
  - if M results in $\triangleright \sqcup N \underline{\sqcup}$ → the schema does not halt

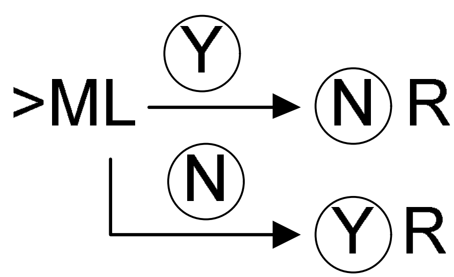
M’:



Turing decidable languages

- Theorem: if language L is Turing decidable $\leftrightarrow$ L^C is Turing decidable
- Proof:
  - TM M decides L
  - the machine schema below decides L^C
  - if M results in $\triangleright\sqcup Y\underline{\sqcup}$ $\rightarrow$ the schema results $\triangleright\sqcup N\underline{\sqcup}$
  - if M results in $\triangleright\sqcup N\underline{\sqcup}$ $\rightarrow$ the schema results $\triangleright\sqcup Y\underline{\sqcup}$

M':



Turing decidable languages

- Theorem: if both L and L^C are Turing acceptable $\leftrightarrow$ L is Turing decidable
- Proof:
  - $\rightarrow$
    - TM M_1 accepts L, M_2 accepts L^C
    - construct a 2-tape TM which simulates M_1 on tape 1 and M_2 on tape 2 in parallel
      - execute one step of M_1 then one step of M_2, and so on (time sharing)
      - if M_1 is to performed fully first and the input is in $L^C \rightarrow M_1$ never stops, so this is a wrong strategy

Turing decidable languages

- Proof:
  - →
    - if M_1 halts, write ⊔Y$\underline{⊔}$ to the tape and halt
    - if M_2 halts, write ⊔N$\underline{⊔}$ to the tape and halt
  - ←
    - if L Turing decidable → L Turing acceptable
    - if L Turing decidable → L^C Turing decidable
    - if L^C Turing decidable → L^C Turing acceptable

Summary

- Turing computable function
- Representation of numbers with strings
- String accepted by TM
- Language accepted by TM
- Turing acceptable, decidable

Next time

- The Church-Turing thesis
- Universal Turing machines
- The halting problem

Element of the Theory of Computation

Lesson 12

5.1. The Church-Turing thesis

5.2. Universal Turing machines

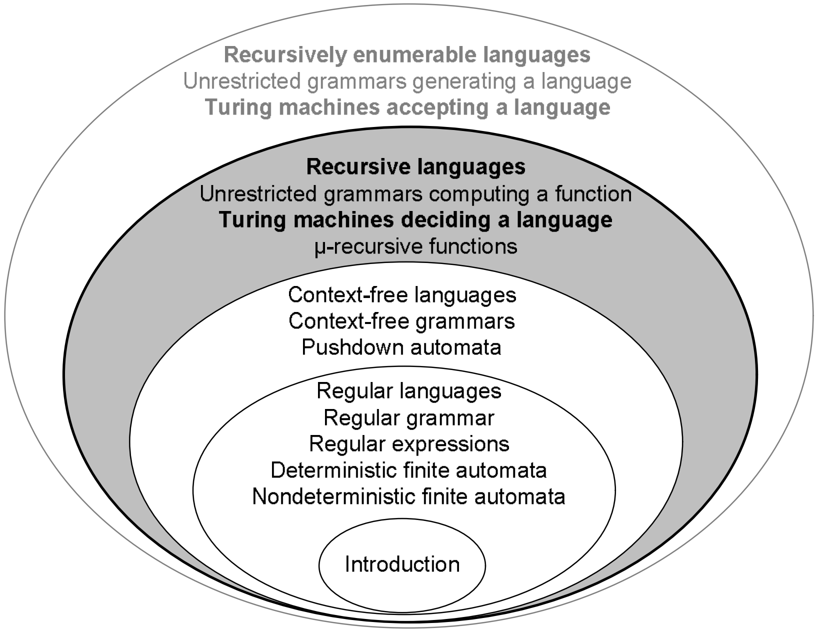
5.3. The halting problem

University of Pannonia

Dr. István Heckl, Istvan.Heckl@gmail.com

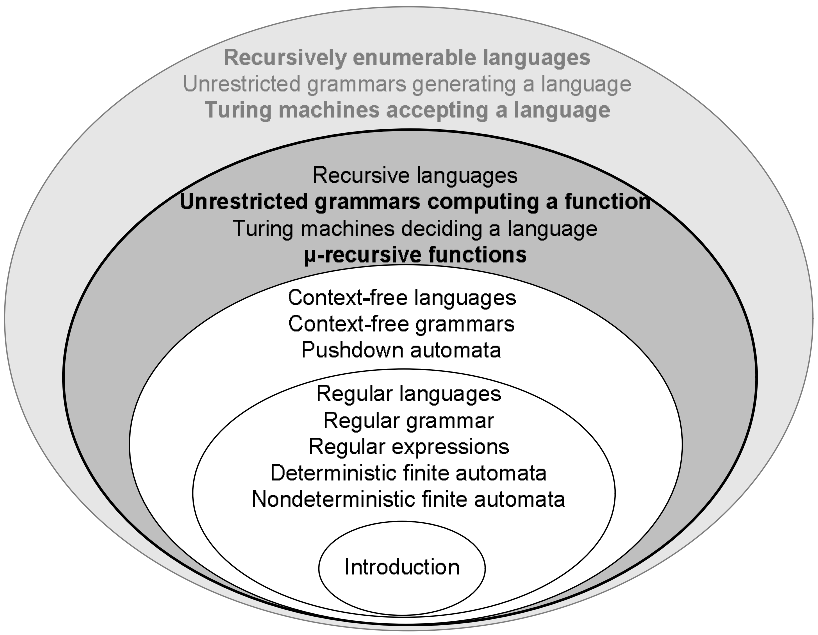
Last time

- Turing computable function
- Representation of numbers with strings
- String accepted by TM
- Language accepted by TM
- Turing acceptable
- Turing decidable
- Algorithm



TM

- The Church-Turing thesis
- Universal TM
- Unary encoding
- Binary encoding
- The halting problem



The Church-Turing thesis

- By keeping extending the language acceptors we have reached the TM
  - we have demonstrated the wide range of tasks solvable by TM
  - several enchantment (multiple tape, random access memory, non-deterministic behavior) do not increase the computational capability of the TM

The Church-Turing thesis

- By keeping extending the language generators the unrestricted grammars can be reached
- μ-recursive functions is also a representation of languages
  - μ-recursive functions, TMs, and unrestricted grammars are equivalent

The Church-Turing thesis

- Church-Turing thesis: any algorithm can be performed by a TM provided that sufficient time and storage space are available
  - it is a thesis and not a theorem because TM is a mathematical concept but algorithm is not
    - it cannot be proved
    - could be disproved by introducing such a reasonable machine which is capable to solve such problems which cannot be done with TM

The Church-Turing thesis

- We regard something as algorithm if it can be represented by such TMs which halt on every input
  - TMs accepting languages cannot be regarded as algorithms
    - they do not halt on every input

The Church-Turing thesis

- We have shown previously that there are uncountable languages but only countably infinite representation
  - not every language can be represented
  - deciding if w is such a language is an unsolvable problem

The Church-Turing thesis

- The cardinality argument (there are countably infinite language representation but uncountable languages) proves only the existence of unsolvable problems
  - finding an actual unsolvable problem is our current aim

Universal TM

- TM cannot be programmed
  - its program is hardwired into the transition function
- Definition of universal TM, U: such a TM which is capable of simulating any TM
  - U can be programmed as any computer
  - the program and the input of the program can be given on the tape of U
  - U is still a TM

Universal TM

- The program of U is the encoding of a TM
  - hardware and software are equivalent (Neumann principle)
  - ρ(M) the encoding of TM M (rho of M)
  - ρ(w) the encoding of string w
- U(ρ(M)ρ(w)) = ρ(M(w))
  - U gives the same result in encoded form what its program (M) would give processing the program's input (w)
  - beware: M's input is w, U's input is the encoded form of M and w

q	σ	$\delta(q, \sigma)$
q_1	a	$(q_0, \leftarrow)$
q_1	$\sqcup$	$(h, \sqcup)$
q_1	$\rhd$	$(q_0, \rightarrow)$
q_2	a	$(q_1, \rightarrow)$
…		

Unary encoding

- If TM $M = (K, \Sigma, \delta, s) \rightarrow$
  $\rho(M) = cS_0cS_{q1,a1}S_{q1,a2} \ldots S_{q1,a|\Sigma|}S_{q2,a1}S_{q2,a2} \ldots S_{q|K|,a|\Sigma|}$
  - S_0 encodes the initial state, $S_0 = \lambda(s)$
  - $S_{qp,ar}$ encodes values of the transition function $\delta(q_p, a_r) = (q_p', a_r')$
    - $S_{qp,ar} = cw_1cw_2cw_3cw_4$, where
      - $w_1 = \lambda(q_p)$
      - $w_2 = \lambda(a_r)$
      - $w_3 = \lambda(q_p')$
      - $w_4 = \lambda(a_r')$
- The encoding of string $w = b_1b_2 \ldots b_n$:
  $\rho(w) = c\lambda(b_1)c\lambda(b_2)c \ldots c\lambda(b_n)c$

Unary encoding

- Encoding of the alphabet and states of the program:
  - to decide what I^3 does mean, its position must be checked
    - at w_1, w_3 it is q_2
    - at w_2, w_4 it is a_1

σ	λ(σ)
states	
q_i	I^{i+1}
h	I
alphabet	
L	I
R	II
a_i	I^{i+2}

Unary encoding

σ	λ(σ)
states	
q_i	I^{i+1}
h	I
alphabet	
L	I
R	II
a_i	I^{i+2}

- Example:
  - M = {K, Σ, δ, s, {h}}
    - K = {h, q_2}
    - Σ = {a_1, a_3, a_6}
    - s = q_2 ↔ III
    - transition function
      - δ(q_2, a_1) = (h, a_3) ↔ cIIIcIIIcIcIIIIIc
      - δ(q_2, a_3) = (q_2, R) ↔ cIIIcIIIIIcIIIcIIc
      - δ(q_2, a_6) = (q_2, R) ↔ cIIIcIIIIIIIIcIIIcIIc
  - ρ(M) = $cI^3ccI^3cI^3cIcI^5ccI^3cI^5cI^3cI^2ccI^3cI^8cI^3cI^2cc$
    - cc signals the start of some $S_{qp,ar}$

Unary encoding

- In the example Σ is not $\{a_1, a_2, a_3\}$, why?
  - suppose a machine schema is to be executed by U
  - each TM of the schema may have different Σ
  - U has to represent every possible character
    - create the union of all Σ and number the elements
    - create new indices to the elements
    - these new indices are used

Unary encoding

- In general, U can execute any TM so it has to represent every possible character
  - the Σ of U still {c, I}
    - λ is the unary encoding
- Similar argument holds for K

Binary encoding

- If TM $M = (K, \Sigma, \delta, s, H) \rightarrow$
  $\rho(M) = S_{q1,a1} S_{q1,a2} \ldots S_{q1,a|\Sigma|} S_{q2,a1} S_{q2,a2} \ldots S_{q|K|,a|\Sigma|}$
  - $\exists$ i, j $\in$ N such that, $2^i > |K|$, $2^j > |\Sigma|$
  - $S_{qp,ar}$ encodes values of the transition function $\delta(q_p, a_r) = (q_p', a_r')$
    - $S_{p,r} = (w_1, w_2, w_3, w_4)$ where
      - $w_1 = \lambda(q_p)$
      - $w_2 = \lambda(a_r)$
      - $w_3 = \lambda(q_p')$
      - $w_4 = \lambda(a_r')$

Binary encoding

- Encoding of the alphabet and states of the program:
  - q_k q*numBin(k)*
    - q followed by a binary number of length i
    - the actual encoding is not given here
    - the start state is always $q0^i$
  - ⊔ $a0^j$
  - ▹ $a0^{j-1}1$
  - ← $a0^{j-1}10$
  - → $a0^{j-1}11$
  - a_k a*numBin(k+3)*
    - 'a' followed by a binary number of length j

Binary encoding

- Example:
  - consider TM M = (K, Σ, δ, s, {h})
    - K = {s, q, h}, Σ = {⊔, ▻, a}
    - δ and the state, symbol encoding are given in these tables (i = 2, j = 3)

state/ symbol	represen- tation
s	q00
q	q01
h	011
⊔	a000
▻	a001
→	a010
←	a011
a	a100

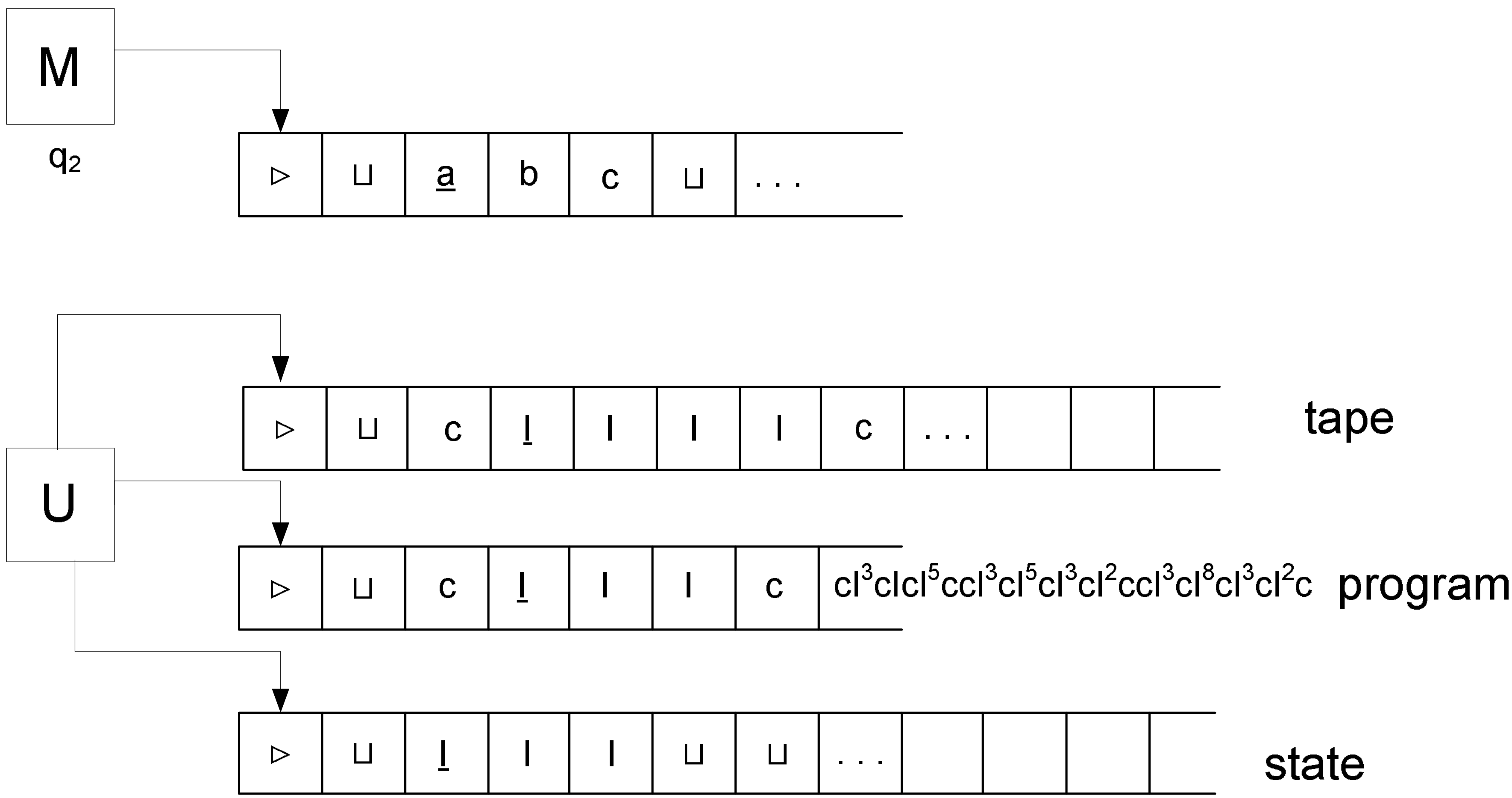
Universal TM

- U is implemented with a 3-tape TM
  - tape 1: encoding of the tape of TM M to be simulated
    - initially: ρ(w), the input of the algorithm
  - tape 2: encoding of the TM M to be simulated
    - ρ(M) is the program
  - tape 3: encoding of the current state of TM M during the simulation

Universal TM

- Operation:
  - initially the input of U, ρ(M)ρ(w), is on the tape 1
  - ρ(M) is copied to the tape 2, ρ(w) is shifted to the left
  - the starting state is written onto tape 3
  - head 1 moves in accordance with the head of M
    - initially it is on the 2nd square where the encoding of the 1st symbol of w starts
  - head 2 searches for such a transition which corresponds to the actual simulated state and the actual scanned simulated symbol
  - according to the transition either the scanned simulated symbol is changed or head 1 is moved

Universal TM



The halting problem

- Remember the diagonalization principle
  - the complement of the diagonal differs from each row
- Some seemingly correct definition can be contradictory
  - e.g.: (Bob) the barber cuts the beard for those people (condition:) who does not do it for himself
    - Adam cuts his own beard so Bod does not do it
    - Clarence does not cut his own beard so Bob cuts it

The halting problem

  - does Bob cut his own beard?
    - suppose no: the condition is true for Bob, thus, the barber cuts his beard according to the definition, contradiction is reached
    - suppose yes: the condition is false for Bob, thus, the barer does not cut his beard according to the definition of barber, contradiction is reached
- The set of people whose beard is cut by Bob
  - the above definition is contradictory
  - a mathematical system is either incomplete or contradictory

The halting problem

- halts(P, X): such a program which returns yes if the program P would stop on input X, otherwise it returns no
  - halts(P, X) always stops
  - halts(P, X) would be very useful for debuging
- Theorem: halts(P, X) does not exist
  - function "halts" is not Turing computable
- Proof by indirection:
  - assume halts(P, X) does exist

The halting problem

- construct diagonal(X)
  - such a program which loops forever if program X would stop on input X, otherwise it stops

```
diagonal(X)
a: if halts(X, X) then goto a:
   else stop
```

- let X = diagonal
  - start diagonal(diagonal)
  - will diagonal(diagonal) stop?

The halting problem

- if halts(diagonal, diagonal) = true →
  - diagonal does not stop because the goto statement loops forever
  - diagonal stops according to the definition of halts(P, X)
- if halts(diagonal, diagonal) = false→
  - diagonal stops in the else branch
  - diagonal does not stop according to the definition of halts(P, X)

– contradiction reached in both cases, so, halts(P, X) does not exist

The halting problem

- Theorem: language H corresponding to halts(P, X) is not Turing decidable
  - H = {ρ(M)ρ(w) : TM M halts on input w}
    - H contains strings with two components, the first is the encoding of a program, the second is the encoding of its input, moreover the program halts on the given input
  - H is Turing acceptable as it is accepted by U
    - the input of U is a program and its input
    - according to U's definition if the program halts then U also halts

The halting problem

- Proof:
  - assume H is Turing decidable
  - H_2 = {ρ(M)ρ(M) : TM M halts on input w} is a subset of H
  - H_1 = {ρ(M) : TM M stops on input ρ(M)}
  - H_2 can be transformed into H_1 by halving the string
  - H_1 is a subset of H
  - if H is Turing decidable → H_1 also Turing decidable
    - H_1 corresponds to halts(X, X)

The halting problem

- if H_1 Turing decidable → H_1^C also Turing decidable (theorem)
  - H_1^C = {w : w is not the encoding of a TM, or w = ρ(M) but M does not halt on input ρ(M)}
    - corresponds to diagonal(X)
- H_1^C is not even Turing acceptable
  - suppose M* accepts H_1^C

The halting problem

- is it true that $\rho(M^*) \in H_1^C$ (will diagonal(diagonal) stop?)
  - $\rho(M^*) \in H_1^C$
    - » M* does not halt on input $\rho(M^*)$ according to the definition of H_1^C
    - » M* accepts $H_1^C \rightarrow$ M* does halt input on $\rho(M^*)$ by the definition of acceptance

The halting problem

  - $\rho(M^*) \notin H_1^C$
    - » M* halt on input $\rho(M^*)$ according to the definition of H_1^C
    - » M* accepts $H_1^C \rightarrow$ M* does not halt input on $\rho(M^*)$ by the definition of acceptance
  - contradiction reached $\rightarrow$ M* does not exist
- H_1^C seems to be nicely defined by a property but its property cannot be checked

The halting problem

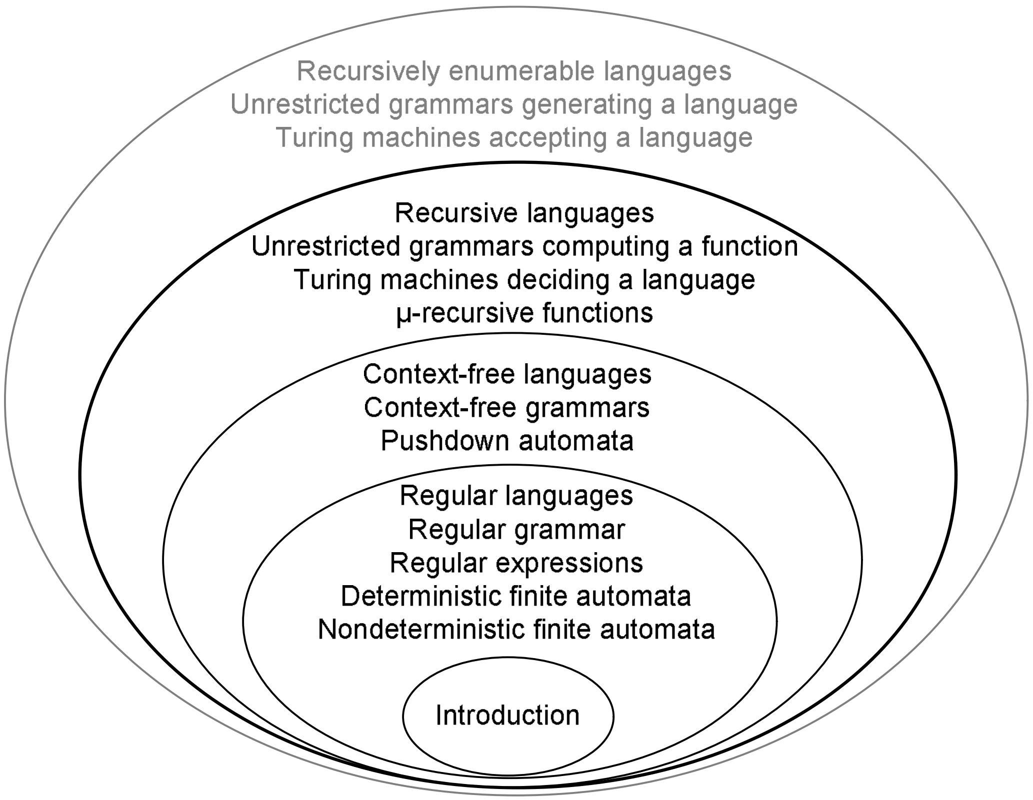
- Theorem: the class of Turing decidable languages is a strict subset of the class of Turing acceptable languages
- Proof: H is Turing acceptable but not Turing decidable
- Theorem: the class of Turing acceptable languages is not closed for complementation
- Proof: H_1 is Turing acceptable but H_1^C is not

The halting problem

- Definition of undecidable problems: such problems for which no algorithm exists
  - no TM M exists which can decide if w is in L or not
- The most famous undecidable problem is the halting problem
  - both the TM and its input is arbitrary
  - if a fixed TM is considered then it may be decidable
- Other undecidable problems:
  - deciding whether multivariable polynomial equation has a solution in integers (Hilbert's tenth problem)
  - tiling problem

Summary

- The Church-Turing thesis
- Universal TM
- Unary encoding
- Binary encoding
- The halting problem



The exam

- One of six initial questions:
  - RG $\leftrightarrow$ NFA (two proofs)
  - NFA $\rightarrow$ DFA (two proofs)
  - CFG $\rightarrow$ PDA (two proofs)
- The exam is failed if the initial question is failed

The exam

- Check the download material in moodle
- Do not be surprised when I ask what grade is your aim
- Have a favorite question
- Have the lecture notes, sometimes it is enough to explain something, so you do not have to write it down
- Be ready for questions from the last lecture too
- If you failed the exam, next time know what you did not know before
- Be prepared for examples