Automatentheorie und formale Sprachen
Satz von Kleene

Dozentin: Wiebke Petersen

3.6.2009

Satz von Kleene

(Stephen C. Kleene, 1909 - 1994)

Jede Sprache, die von einem deterministischen endlichen Automaten akzeptiert wird ist regulär und jede reguläre Sprache wird von einem deterministischen endlichen Automaten akzeptiert.

Wiederholung: reguläre Sprachen

RE: syntax

The set of regular expressions RE_Σ over an alphabet $\Sigma = \{a_1, \dots, a_n\}$ is defined by:

- $\underline{\emptyset}$ is a regular expression.
- ϵ is a regular expression.
- $a_1, \dots, a_n$ are regular expressions
- If a and b are regular expressions over Σ then
  - $(a + b)$
  - $(a \bullet b)$
  - $(a^\star)$

  are regular expressions too.

Wiederholung: reguläre Sprachen

RE: semantics

Each regular expression r over an alphabet Σ describes a formal language $L(r) \subseteq \Sigma^*$.

Regular languages are those formal languages which can be described by a regular expression.

The function L is defined inductively:

- $L(\underline{\emptyset}) = \emptyset$, $L(\epsilon) = \{\epsilon\}$, $L(a_i) = \{a_i\}$
- $L(a + b) = L(a) \cup L(b)$
- $L(a \bullet b) = L(a) \circ L(b)$
- $L(a^\star) = L(a)^*$

Finite-state automatons accept regular languages

Theorem (Kleene)

Every language accepted by a DFSA is regular and every regular language is accepted by some DFSA.

Finite-state automatons accept regular languages

Theorem (Kleene)

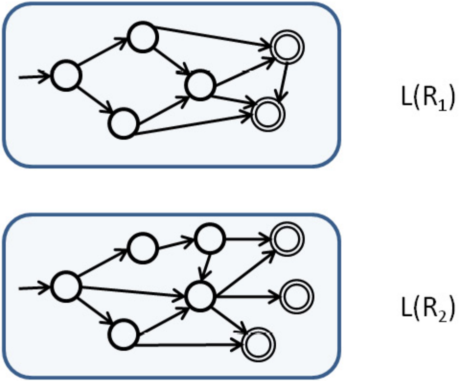
Every language accepted by a DFSA is regular and every regular language is accepted by some DFSA.

proof idea (one direction): Each regular language is accepted by a NDFSA (and therefore by a DFSA):

$L(\emptyset)$ $L(\epsilon)$ $L(a_i)$

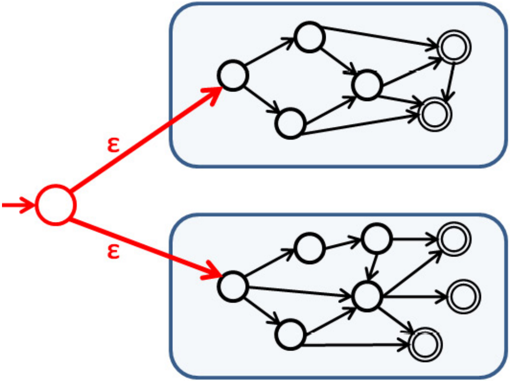
Proof of Kleene's theorem (cont.)

If R_1 and R_2 are two regular expressions such that the languages $L(R_1)$ and $L(R_2)$ are accepted by the automatons $\mathcal{A}_1$ and $\mathcal{A}_2$ respectively, then $L(R_1 + R_2)$ is accepted by:



Proof of Kleene's theorem (cont.)

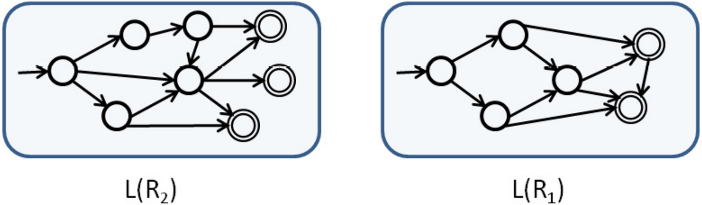
If R_1 and R_2 are two regular expressions such that the languages $L(R_1)$ and $L(R_2)$ are accepted by the automatons $\mathcal{A}_1$ and $\mathcal{A}_2$ respectively, then $L(R_1 + R_2)$ is accepted by:



$L(R_1+R_2)$

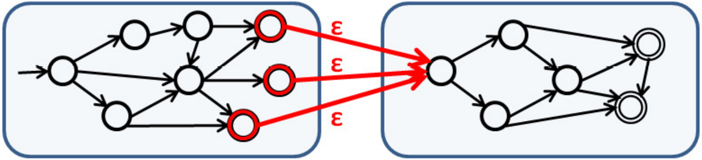
Proof of Kleene's theorem (cont.)

$L(R_1 \bullet R_2)$ is accepted by:



Proof of Kleene's theorem (cont.)

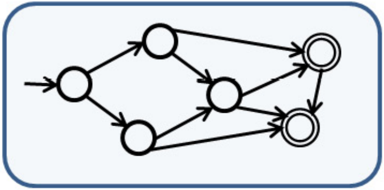
$L(R_1 \bullet R_2)$ is accepted by:



L(R₂•R₁)

Proof of Kleene's theorem (cont.)

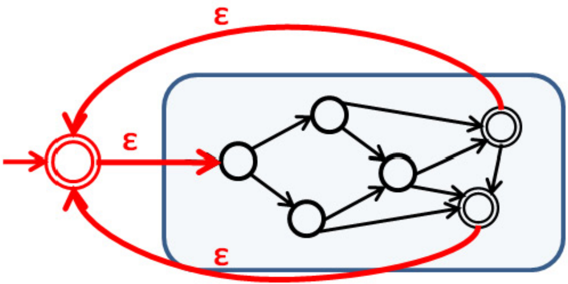
$L(R_1^*)$ is accepted by:



$L(R_1)$

Proof of Kleene's theorem (cont.)

$L(R_1^*)$ is accepted by:



L(R₁*)

Exercise 1

Beschreiben sie mit ihren eigenen Worten, wie die Automaten für die Sprachen $L(R_1 + R_2)$, $L(R_1 \bullet R_2)$, und $L(R_1^\star)$ systematisch aus den Automaten für die Sprachen $L(R_1)$ und $L(R_2)$ konstruiert werden können.

Closure properties of regular languages

Theorem

1. *If L_1 and L_2 are two regular languages, then*
   - *the union of L_1 and L_2 ($L_1 \cup L_2$) is a regular language too.*
   - *the intersection of L_1 and L_2 ($L_1 \cap L_2$) is a regular language too.*
   - *the concatenation of L_1 and L_2 ($L_1 \circ L_2$) is a regular language too.*
2. *The complement of every regular language is a regular language too.*
3. *If L is a regular language, then L^* is a regular language too.*

Exercise 2

Überglegen sie sich, warum obenstehende Aussagen gelten.

Implementierung endlicher Automaten in Prolog

Prolog: the basics

- facts: state things that are unconditionally true of the domain of interest.
  `human(sokrates).`
- rules: relate facts by logical implications.
  `mortal(X) :- human(X).`
  - head: left hand side of a rule
  - body: right hand side of a rule
  - clause: rule or fact.
  - predicate: collection of clauses with identical heads.
- knowledge base: set of facts and rules
- queries: make the Prolog inference engine try to deduce a positive answer from the information contained in the knowledge base.
  `?- mortal(sokrates).`

Prolog: some syntax

- facts: `fact.`
- rules: `head :- body.`
- conjunction: `head :- info1 , info2.`
- atoms start with small letters
- variables start with capital letters

lists in Prolog

- Lists are recursive data structures: First, the empty list is a list. Second, a complex term is a list if it consists of two items, the first of which is a term (called first), and the second of which is a list (called rest).
- `[mary|[john|[alex|[tom|[]]]]]`
- simpler notation: `[mary,john,alex,tom]`

```
function D-RECOGNIZE (tape, machine) returns accept or reject
  index ← Beginning of tape
  current-state ← Initial state of machine
  loop
    if End of input has been reached then
      if current-state is an accept state then
        return accept
      else
        return reject
    elsif transition-table [current-state, tape[index]] is empty then
      return reject
   else
     current-state ← transition-table [current-state, tape[index]]
     index ← index + 1
end
```

Jurafsky & Martin 2000

function D-RECOGNIZE (*tape*, *machine*) **returns** accept or reject
index ← Beginning of tape
current-state ← Initial state of machine
loop
if End of input has been reached **then**
if current-state is an accept state **then**
return accept
else
return reject
elsif *transition-table [current-state, tape[index]]* is empty **then**
return reject
else
current-state ← *transition-table [current-state, tape[index]]*
index ← *index* + 1
end

Jurafsky & Martin 2000

```
% Finite state automaton.

fsa(Tape):-
   initial(S),
   fsa(Tape,S).

fsa([],S):- final(S).

fsa([H|T],S):-
   trans_tab(S,H,NS),
   fsa(T,NS).

% FSA transition table:
% trans_tab/3
% trans_tab(State, Input, New State)

trans_tab(1,a,1).
trans_tab(1,b,2).
trans_tab(2,a,2).

initial(1).
final(2).
```

```
function ND-RECOGNIZE(tape, machine) returns accept or reject

  agenda ← {(Initial state of machine, beginning of tape)}
  current-search-state ← NEXT(agenda)
  loop
    if ACCEPT-STATE?(current-search-state) returns true then
       return accept
    else
       agenda ← agenda ∪ GENERATE-NEW-STATES(current-search-state)
    if agenda is empty then
       return reject
    else
       current-search-state ← NEXT(agenda)
  end

function GENERATE-NEW-STATES(current-state) returns a set of search-states

  current-node ← the node the current search-state is in
  index ← the point on the tape the current search-state is looking at
  return a list of search states from transition table as follows:
    (transition-table[current-node, ε], index)
    ∪
    (transition-table[current-node, tape[index]], index + 1)

function ACCEPT-STATE?(search-state) returns true or false

  current-node ← the node search-state is in
  index ← the point on the tape search-state is looking at
  if index is at the end of the tape and current-node is an accept state of machine
  then
    return true
  else
    return false
```