Script generated by TTT

Title: Simon: Compilerbau (18.04.2012)

Date: Wed Apr 18 14:16:02 CEST 2012

Duration: 87:50 min

Pages: 46

Organisatorisches

- Master oder Bachelor ab 6. Semester mit 5 ECTS
- Voraussetzungen
  - Informatik 1 & 2
  - Theoretische Informatik
  - Technische Informatik
  - Grundlegende Algorithmen
- Aufbauende Vorlesungen
  - Virtuelle Maschinen
  - Programmoptimierung
  - Programmiersprachen
  - Praktikum Compilerbau
  - Hauptseminare

Material:

- Aufzeichnung der Vorlesungen über TTT
- die Folien selbst
- Literaturliste online (→ Bibliothek)
- Tools zur Visualisierung der Virtuellen Maschinen
- Tools, um Komponenten eines Compilers zu generieren

Organisatorisches

Zeiten:

Vorlesung: Mi. 14:15-15:45 Uhr
Übung: wird per Onlineabstimmung geklärt

Abschlussprüfung

- Klausur, Termin über TUM-online
- Erfolgreiches Lösen der Aufgaben wird als Bonus von 0.3 angerechnet.

Vorläufiger Inhalt

- Grundlagen Reguläre Ausdrücke und Automaten
- Von der Spezifizierung mit mehreren Regulären Ausdrücken zur Implementation mit Automaten
- Reduzierte kontextfreie Grammatiken und Kellerautomaten
- Bottom-Up Syntaxanalyse
- Attributsysteme
- Typüberprüfung
- Codegenerierung für Stackmaschinen
- Registerverteilung
- Einfache Optimierung

Themengebiet:

Einführung

Interpreter

Vorteil:

Keine Vorberechnung auf dem Programmtext erforderlich
⇒ keine/geringe Startup-Zeit

Nachteil:

Während der Ausführung werden die Programm-Bestandteile immer wieder analysiert
⇒ längere Laufzeit

Prinzip eines Übersetzers:

Zwei Phasen:

1. Übersetzung des Programm-Texts in ein Maschinen-Programm;
2. Ausführung des Maschinen-Programms auf der Eingabe.

Übersetzer

Eine Vorberechnung auf dem Programm gestattet u.a.

- eine geschickte(re) Verwaltung der Variablen;
- Erkennung und Umsetzung globaler Optimierungsmöglichkeiten.

Nachteil

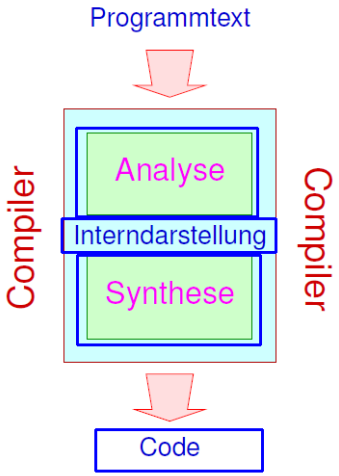
Die Übersetzung selbst dauert einige Zeit

Vorteil

Die Ausführung des Programme wird effizienter
⇒ lohnt sich bei aufwendigen Programmen und solchen, die mehrmals laufen.

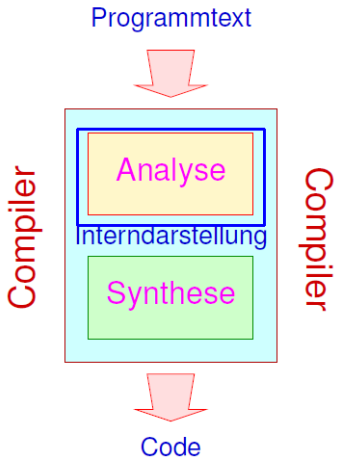
Übersetzer

allgemeiner Aufbau eines Übersetzers:

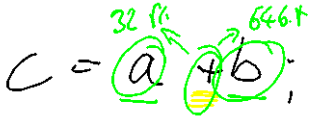


Übersetzer

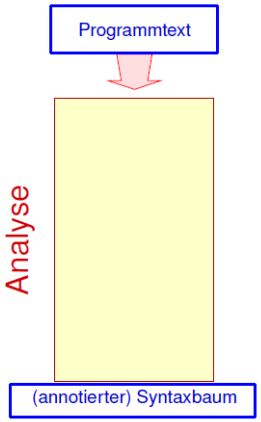
allgemeiner Aufbau eines Übersetzers:



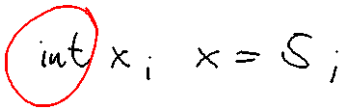
Übersetzer



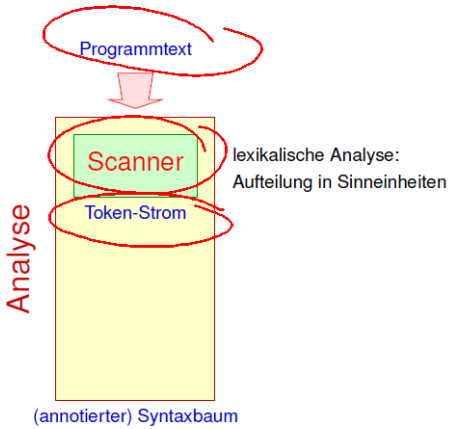
Die Analyse-Phase ist selbst unterteilt in mehrere Schritte



Übersetzer

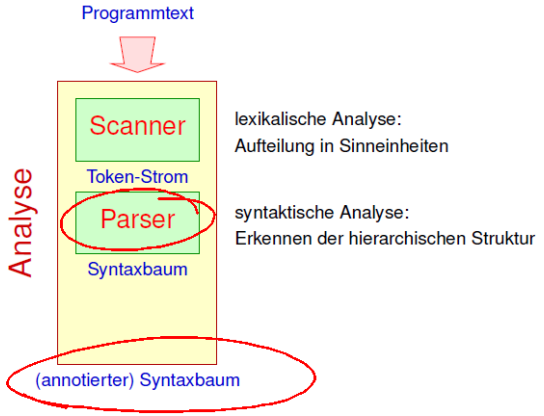


Die Analyse-Phase ist selbst unterteilt in mehrere Schritte



Übersetzer

Die Analyse-Phase ist selbst unterteilt in mehrere Schritte

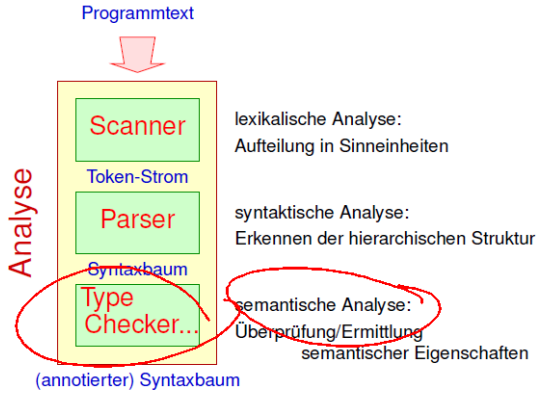


lexikalische Analyse:
Aufteilung in Sinneinheiten

syntaktische Analyse:
Erkennen der hierarchischen Struktur

Übersetzer

Die Analyse-Phase ist selbst unterteilt in mehrere Schritte



lexikalische Analyse:
Aufteilung in Sinneinheiten

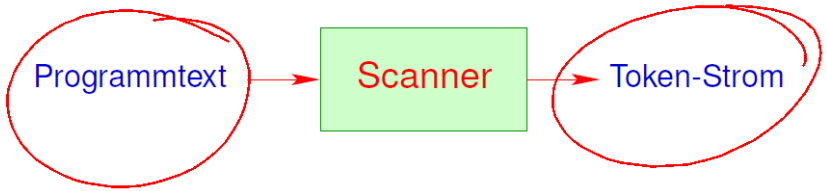
syntaktische Analyse:
Erkennen der hierarchischen Struktur

semantische Analyse:
Überprüfung/Ermittlung semantischer Eigenschaften

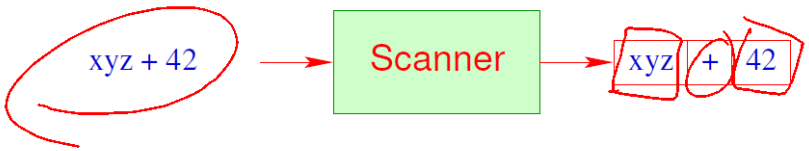
Themengebiet:

Lexikalische Analyse

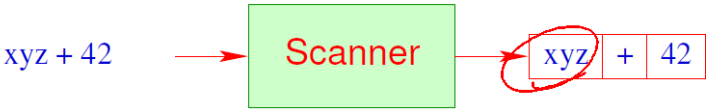
Die lexikalische Analyse



Die lexikalische Analyse



Die lexikalische Analyse



- Ein Token ist eine Folge von Zeichen, die zusammen eine Einheit bilden.
- Tokens werden in Klassen zusammen gefasst. Zum Beispiel:
  - → Namen (Identifier) wie `xyz`, `pi`, ...
  - → Konstanten wie `42`, `3.14`, `"abc"`, ...
  - → Operatoren wie `+`, ...
  - → reservierte Worte wie `if`, `int`, ...

Die lexikalische Analyse

3.1415
float-const

Sind Tokens erst einmal klassifiziert, kann man die Teilwörter vorverarbeiten:

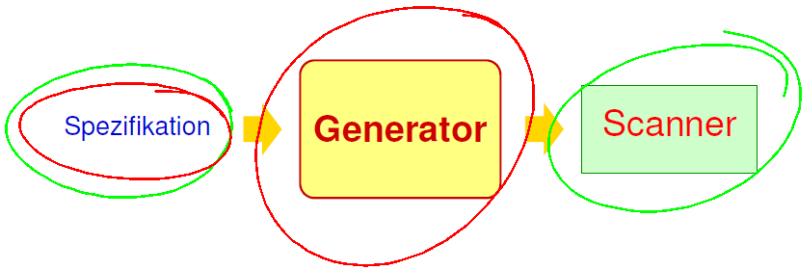
- Wegwerfen irrelevanter Teile wie Leerzeichen, Kommentare,...
- Aussondern von Pragmas, d.h. Direktiven an den Compiler, die nicht Teil des Programms sind, wie include-Anweisungen;
- Ersetzen der Token bestimmter Klassen durch ihre Bedeutung / Interndarstellung, etwa bei:
  - → Konstanten;
  - → Namen: die typischerweise zentral in einer Symbol-Tabelle verwaltet, evt. mit reservierten Worten verglichen (soweit nicht vom Scanner bereits vorgenommen) und gegebenenfalls durch einen Index ersetzt werden.

⇒ Sieber

Die lexikalische Analyse

Diskussion:

- Scanner und Sieber werden i.a. in einer Komponente zusammen gefasst, indem man dem Scanner nach Erkennen eines Tokens gestattet, eine Aktion auszuführen
- Scanner werden i.a. nicht von Hand programmiert, sondern aus einer Spezifikation generiert:



Die lexikalische Analyse - Generierung:

Vorteile

Produktivität Die Komponente lässt sich schneller herstellen

Korrektheit Die Komponente realisiert (beweisbar) die Spezifikation.

Effizienz Der Generator kann die erzeugte Programmkomponente mit den effizientesten Algorithmen ausstatten.

Die lexikalische Analyse - Generierung:

Vorteile

Produktivität Die Komponente lässt sich schneller herstellen

Korrektheit Die Komponente realisiert (beweisbar) die Spezifikation.

Effizienz Der Generator kann die erzeugte Programmkomponente mit den effizientesten Algorithmen ausstatten.

Einschränkungen

- Spezifizieren ist auch Programmieren — nur eventuell einfacher
- Generieren statt Implementieren lohnt sich nur für Routine-Aufgaben
  ... und ist nur für Probleme möglich, die sehr gut verstanden sind

Die lexikalische Analyse - Generierung:

... in unserem Fall:

Die lexikalische Analyse - Generierung:

... in unserem Fall:

Spezifikation von Token-Klassen: Reguläre Ausdrücke;

Generierte Implementierung: Endliche Automaten + X

Kapitel 1:
Grundlagen: Reguläre Ausdrücke

Reguläre Ausdrücke

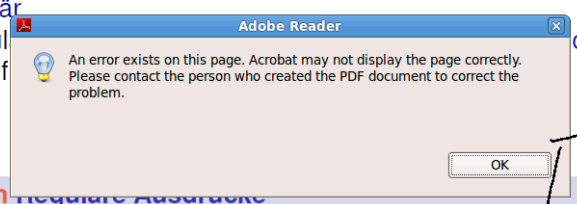
Grundlagen

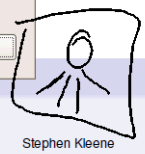
- Programmtext benutzt ein endliches Alphabet Σ von Eingabe-Zeichen, z.B. ASCII
- Die Menge der Textabschnitte einer Token-Klasse ist i.a. regulär.
- Reguläre Sprachen kann man mithilfe regulärer Ausdrücke spezifizieren.

Reguläre Ausdrücke

Grundlagen

- Programmtext benutzt ein endliches Alphabet Σ von Eingabe-Zeichen, z.B. ASCII
- Die Menge der Textabschnitte einer Token-Klasse ist i.a. regulär.
- Regul… cke
- spezif…

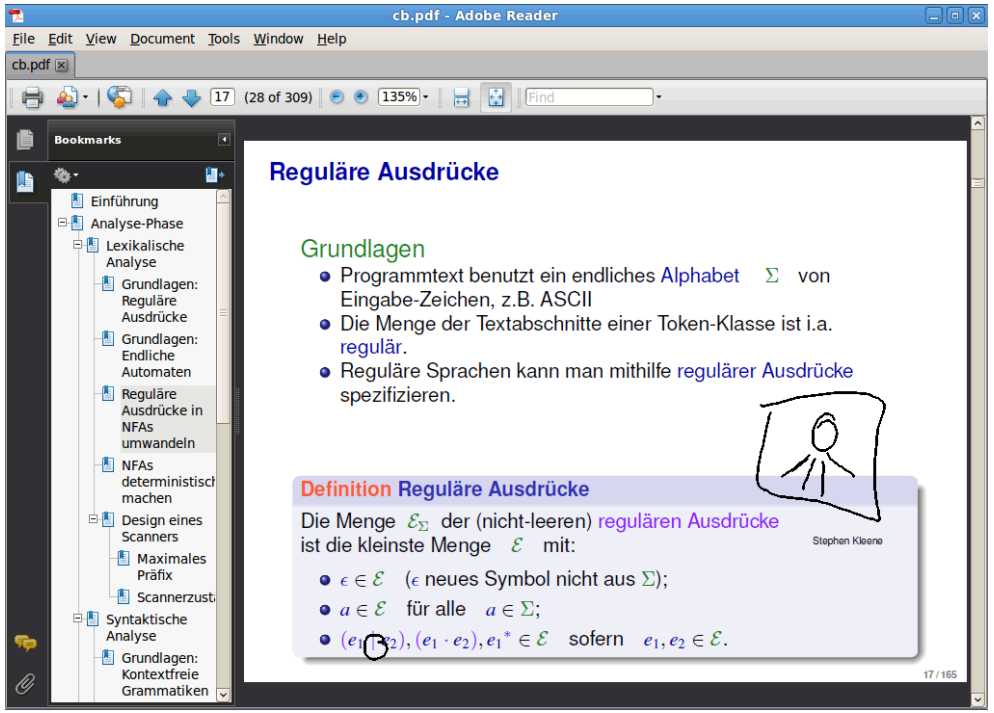


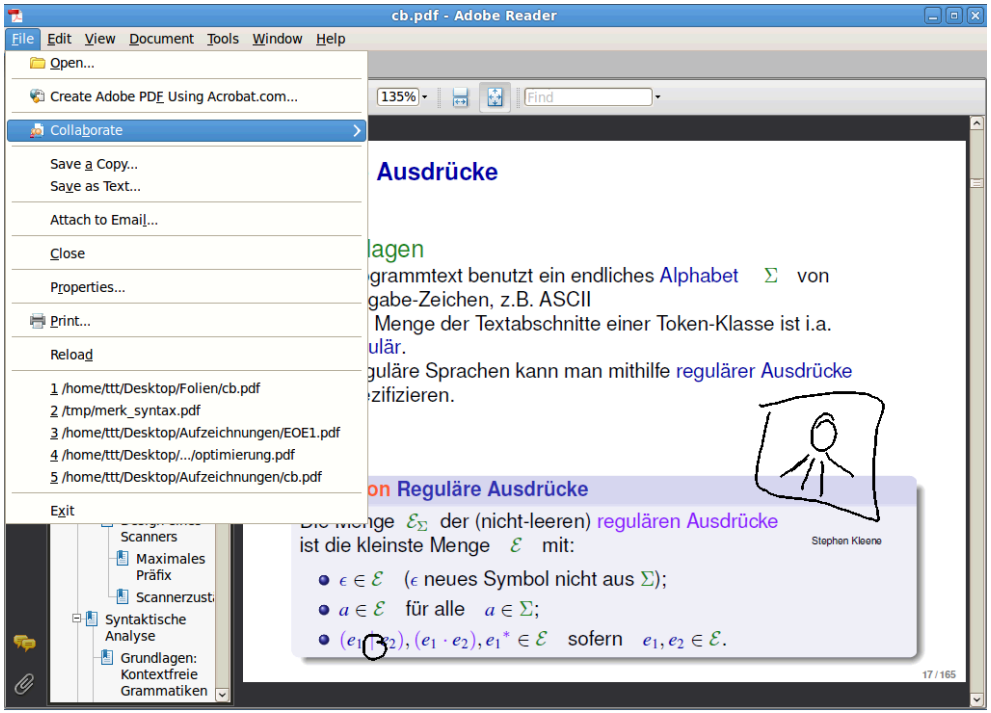

Stephen Kleene

Definition Reguläre Ausdrücke

Die Menge $\mathcal{E}_\Sigma$ der (nicht-leeren) regulären Ausdrücke ist die kleinste Menge $\mathcal{E}$ mit:

- $\epsilon \in \mathcal{E}$ (ϵ neues Symbol nicht aus Σ);
- $a \in \mathcal{E}$ für alle $a \in \Sigma$;
- $(e_1 \mid e_2), (e_1 \cdot e_2), e_1^* \in \mathcal{E}$ sofern $e_1, e_2 \in \mathcal{E}$.





Reguläre Ausdrücke

... im Beispiel:

$((a \cdot b^*) \cdot a)$
$(a \mid b)$
$((a \cdot b) \cdot (a \cdot b))$

Reguläre Ausdrücke

... im Beispiel:

$((a \cdot b^*) \cdot a)$
$(a \mid b)$
$((a \cdot b) \cdot (a \cdot b))$

Reguläre Ausdrücke

... im Beispiel:

$((a \cdot b^*) \cdot a)$
$(a \mid b)$
$((a \cdot b) \cdot (a \cdot b))$

Achtung:

- Wir unterscheiden zwischen Zeichen $a, 0, \$, \ldots$ und Meta-Zeichen $(, |,), \ldots$
- Um (hässliche) Klammern zu sparen, benutzen wir Operator-Präzedenzen:

  $^* > \cdot > |$

  und lassen "$\cdot$" weg

Reguläre Ausdrücke

... im Beispiel:

$$((a \cdot b^*) \cdot a)$$
$$(a \mid b)$$
$$((a \cdot b) \cdot (a \cdot b))$$

Achtung:

- Wir unterscheiden zwischen Zeichen $a, 0, \$$,... und Meta-Zeichen $(, |,)$,...
- Um (hässliche) Klammern zu sparen, benutzen wir Operator-Präzedenzen:

$$^* > \cdot > |$$

und lassen "$\cdot$" weg

- Reale Spezifikations-Sprachen bieten zusätzliche Konstrukte:

$$e? \equiv (\epsilon \mid e)$$
$$e^+ \equiv (e \cdot e^*)$$

und verzichten auf "ϵ"

Reguläre Ausdrücke

Spezifikationen benötigen eine Semantik
...im Beispiel:

Spezifikation	Semantik
$abab$	$\{abab\}$
$a \mid b$	$\{a, b\}$
ab^*a	$\{ab^n a \mid n \geq 0\}$

Für $e \in \mathcal{E}_\Sigma$ definieren wir die spezifizierte Sprache $[\![e]\!] \subseteq \Sigma^*$ induktiv durch:

$$\begin{aligned} [\![\epsilon]\!] &= \{\epsilon\} \\ [\![a]\!] &= \{a\} \\ [\![e^*]\!] &= ([\![e]\!])^* \\ [\![e_1 | e_2]\!] &= [\![e_1]\!] \cup [\![e_2]\!] \\ [\![e_1 \cdot e_2]\!] &= [\![e_1]\!] \odot [\![e_2]\!] \end{aligned}$$

Beachte:

- Die Operatoren $(\_)^*, \cup, \cdot$ sind die entsprechenden Operationen auf Wort-Mengen:

$$\begin{aligned} (L)^* &= \{w_1 \dots w_k \mid k \geq 0, w_i \in L\} \\ L_1 \cdot L_2 &= \{w_1 w_2 \mid w_1 \in L_1, w_2 \in L_2\} \end{aligned}$$

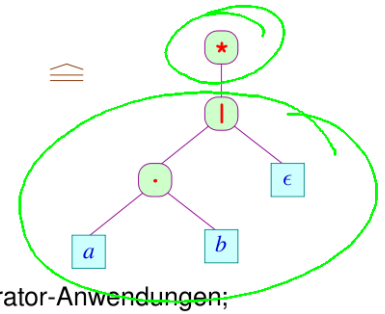
Beachte:

- Die Operatoren $(\_)^*, \cup, \cdot$ sind die entsprechenden Operationen auf Wort-Mengen:

$$\begin{aligned} (L)^* &= \{w_1 \dots w_k \mid k \geq 0, w_i \in L\} \\ L_1 \cdot L_2 &= \{w_1 w_2 \mid w_1 \in L_1, w_2 \in L_2\} \end{aligned}$$

- Reguläre Ausdrücke stellen wir intern als markierte geordnete Bäume dar:

$(ab|\epsilon)^*$ ≙



Innere Knoten: Operator-Anwendungen;
Blätter: einzelne Zeichen oder ϵ.

Reguläre Ausdrücke

Beispiel: Identifier in Java:

```
le = [a-zA-Z_\$]
di = [0-9]
Id = {le} ({le} | {di})*
```


Reguläre Ausdrücke

Beispiel: Identifier in Java:

```
le = [a-zA-Z_\$]
di = [0-9]
Id = {le} ({le} | {di})*
```

```
Float = {di}* (\.{di}|{di}\.) {di}*((e|E)(\+|\-)?{di}+)?
```


Reguläre Ausdrücke

Beispiel: Identifier in Java:

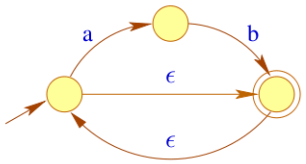
```
le = [a-zA-Z_\$]
di = [0-9]
Id = {le} ({le} | {di})*

Float = {di}* (\.{di}|{di}\.) {di}*((e|E)(\+|\-)?{di}+)?
```

Bemerkungen:

- "`le`" und "`di`" sind Zeichenklassen.
- Definierte Namen werden in "`{`", "`}`" eingeschlossen.
- Zeichen werden von Meta-Zeichen durch "`\`" unterschieden.

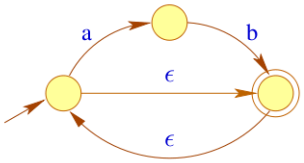
Endliche Automaten

im Beispiel:



Endliche Automaten

im Beispiel:



Knoten: Zustände;
Kanten: Übergänge;
Beschriftungen: konsumierter Input

Endliche Automaten


Michael Rabin


Dana Scott

Definition

Ein nicht-deterministischer endlicher Automat (NFA) ist ein Tupel $A = (Q, \Sigma, \delta, I, F)$ wobei:

Q	eine endliche Menge von Zuständen;
Σ	ein endliches Eingabe-Alphabet;
$I \subseteq Q$	die Menge der Anfangszustände;
$F \subseteq Q$	die Menge der Endzustände und
δ	die Menge der Übergänge (die Übergangs-Relation) ist.

Endliche Automaten


Michael Rabin


Dana Scott

Definition

Ein nicht-deterministischer endlicher Automat (NFA) ist ein Tupel $A = (Q, \Sigma, \delta, I, F)$ wobei:

Q	eine endliche Menge von Zuständen;
Σ	ein endliches Eingabe-Alphabet;
$I \subseteq Q$	die Menge der Anfangszustände;
$F \subseteq Q$	die Menge der Endzustände und
δ	die Menge der Übergänge (die Übergangs-Relation) ist.

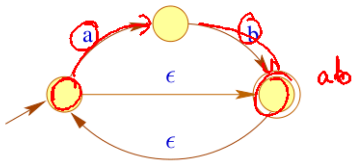
Für NFAs gilt:

Definition

Ist $\delta : Q \times \Sigma \to Q$ eine Funktion und $|I| = 1$, heißt A deterministisch (DFA).

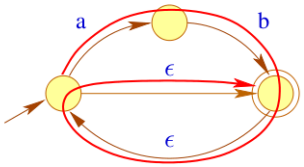
Endliche Automaten

- Berechnungen sind Pfade im Graphen.
- akzeptierende Berechnungen führen von I nach F.
- Ein akzeptiertes Wort ist die Beschriftung eines akzeptierenden Pfades ...



Endliche Automaten

- Berechnungen sind Pfade im Graphen.
- akzeptierende Berechnungen führen von I nach F.
- Ein akzeptiertes Wort ist die Beschriftung eines akzeptierenden Pfades ...



Endliche Automaten

- Dazu definieren wir den transitiven Abschluss δ^* von δ als kleinste Menge δ' mit:

  $(p, \epsilon, p) \in \delta'$ und
  $(p, xw, q) \in \delta'$ sofern $(p, x, p_1) \in \delta$ und $(p_1, w, q) \in \delta'$.

  δ^* beschreibt für je zwei Zustände, mit welchen Wörtern man vom einen zum andern kommt

- Die Menge aller akzeptierten Worte, d.h. die von A akzeptierte Sprache können wir kurz beschreiben als:

$$\mathcal{L}(A) = \{w \in \Sigma^* \mid \exists i \in I, f \in F : (i, w, f) \in \delta^*\}$$

Kapitel 3:

Reguläre Ausdrücke in NFAs umwandeln