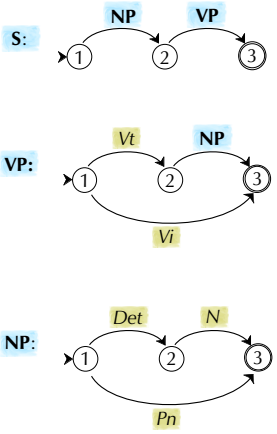
Endliche Automaten Techniken

Übersicht

- Rekursive Übergangsnetzwerke (RTN)
- RTN in Prolog
- Erweiterungen von RTNs
- Vom Akzeptor zum *Transducer*
  - *Beispiel: Lachautomat-Transduktor*
- Implementation von Transduktoren in Prolog
- Lesen oder Schreiben?
- Kompilieren oder Interpretieren?

Rekursive Transitionsnetzwerke (RTN)

Syntaxanalyse mit RTN – Erweiterungen gegenüber EA



- Kanten sind lexikalische Kategorien!
  - *Det, N, Pn, Vt, Vi*
- Kanten sind selbst Automaten!
  - Phrasen: **S**, **VP**, **NP**

Literatur:

- Gazdar/Mellish (1989: 59ff.)
- Matthews (1998: 141ff.)

RTNs in Prolog I

Lexikon

```
word(the, det).
word(man, n).
word(dog, n).
word(peter, pn).
word(smokes, vi).
word(sees, vt).
```

Die Übergänge

```
delta(s, 1, net(np), 2).
delta(s, 2, net(vp), 3).

delta(vp, 1, vi, 3).
delta(vp, 1, vt, 2).
delta(vp, 2, net(np), 3).

delta(np, 1, pn, 3).
delta(np, 1, det, 2).
delta(np, 2, n, 3).
```

Startzustände

```
start(s, 1).
start(np, 1).
start(vp, 1).
```

Endzustände

```
final(s, 3).
final(np, 3).
final(vp, 3).
```


RTN in Prolog II

Initialisierung

- Eine Zeichenkette gilt von einem Netzwerk als akzeptiert, falls ausgehend vom Startzustand des Netzwerks die ganze Zeichenkette abgearbeitet werden kann.

```
init(String, StartNet) :-
   init(String, StartNet, []).
```

- Jedes Netzwerk konsumiert soviel von der Eingabekette, wie es ausgehend von seinem Startzustand akzeptiert.

```
init(String, Net, RestString) :-
   start(Net, StartState),
   accept(String, Net, StartState, RestString).
```

RTN in Prolog III

Abarbeitung

- Fall 1: Lexikalischer Übergang konsumiert Wort!

```
accept([Word|String], Net, State, RestString) :-
   word(Word, Cat),
   delta(Net, State, Cat, NextState),
   accept(String, Net, NextState, RestString).
```

- Fall 2: Hineinspringen in Subnetzwerk ohne Zeichenkonsum!

```
accept(String, Net, State, RestString) :-
   delta(Net, State, net(SubNet), NextState),
   init(String, SubNet, RestStringSubNet),
   accept(RestStringSubNet, Net, NextState, RestString).
```

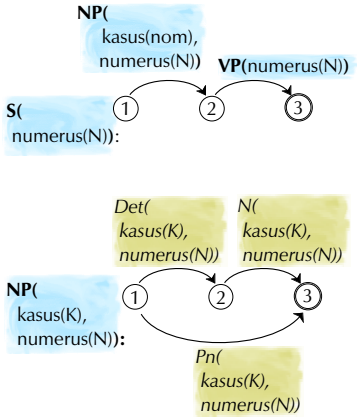
- Fall 3: Abbruchbedingung: Endzustand des Netzwerks erreicht!

```
accept(String, Net, State, String) :-
   final(Net, State).
```


Erweiterungen von RTNs

Komplexe Kanten

- Komplexe lexikalische Kategorien spezifizieren die üblichen morphosyntaktischen Beschränkungen
- Komplexe Automatenbezeichungen erlauben kantenübergreifende Kongruenzbeziehungen
  - Allerdings: Die Welt der EA haben wir damit zugunsten mächtigerer Ausdrucksmittel verlassen.
  - Aber: Warum sollen wir uns künstlich beschränken?



Endliche Automaten vs. Transduktoren

Akzeptierende EAs: Einfach, aber eingeschränkt nützlich!

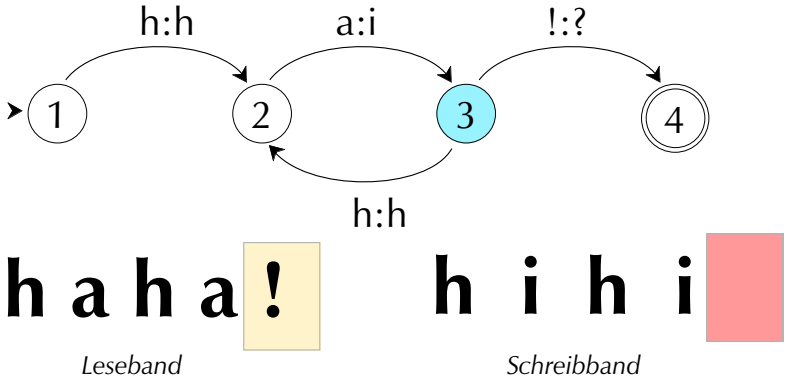
- Einfache akzeptierende Endliche Automaten sind ein gutes Modell, aber für die Praxis oft zu eingeschränkt!
- Wer mehr wissen will als nur "ja" oder "nein", muss den Formalismus aufbrechen (siehe RTN).

Transducer: Einfach, und erst noch nützlich!

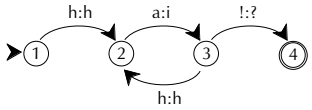
- Endliche Automaten, die zusätzlich zum Lesen auch noch Schreiben können, werden oft als Transduktoren (*transducer*) bezeichnet.
- Transduktoren können die beim Verarbeiten durchlaufenen Schritte nach Aussen kommunizieren ohne den Formalismus aufzubrechen!

Lach-Transduktor

Scanner-Interpretation mit Lese- und Schreibband



Transduktor in Prolog I




```
start(1).
```

```
delta(1, h, h, 2).
delta(2, a, i, 3).
delta(3, h, h, 2).
delta(3, !, ?, 4).
```

```
final(4).
```

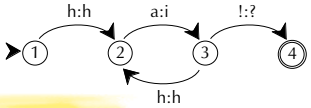
Startzustand

Übergänge

Endzustände

Transduktor in Prolog II

Transduktor



- *Initialisierung*

```
init(Input, Output) :-
  start(StartState),
  transduce(Input, Output, StartState).
```

- *Abarbeitung*

```
transduce([], [], State) :-
  final(State).
transduce([InChar|InChars], [OutChar|OutChars], State) :-
  delta(State, InChar, OutChar, NextState),
  transduce(InChars, OutChars, NextState).
```


Lesen oder Schreiben?

Wir können in beide Richtungen schreiben bzw. lesen.

```
?- init([h,a,'!'], R).
R = [h,i,'?'];
no
```

```
?- init(R, [h,i,'?']).
R = [h,a,'!'];
no
```

Wir können die Ein- *und* Ausgabesprache aufzählen lassen.

- Es gibt soviele Lösungen, wie die Sprache Elemente hat…

```
?- length(L1, _), init(L1, L2).
L1 = [h,a,'!'] L2 = [h,i,'?'];
L1 = [h,a,h,a,'!'] L2 = [h,i,h,i,'?'];
....
```


Interpretieren oder kompilieren?

Automaten müssen nicht zwangsweise interpretiert werden!

- Automaten-Struktur und Abarbeitung lassen sich zu einem effizienteren, aber spezifischeren Programm verquicken!
- lachen(+Ausgangszustand, ?Input, ?Output)

```
lachen(In, Out) :-
   lachen(1, In, Out).
```

```
lachen(1, [h|RestIn], [h|RestOut]) :-
   lachen(2, RestIn, RestOut).
lachen(2, [a|RestIn], [i|RestOut]) :-
   lachen(3, RestIn, RestOut).
lachen(3, [h|RestIn], [h|RestOut]) :-
   lachen(2, RestIn, RestOut).
lachen(3, [!], [?]).
```