Einführung in die Computerlinguistik und Sprachtechnologie

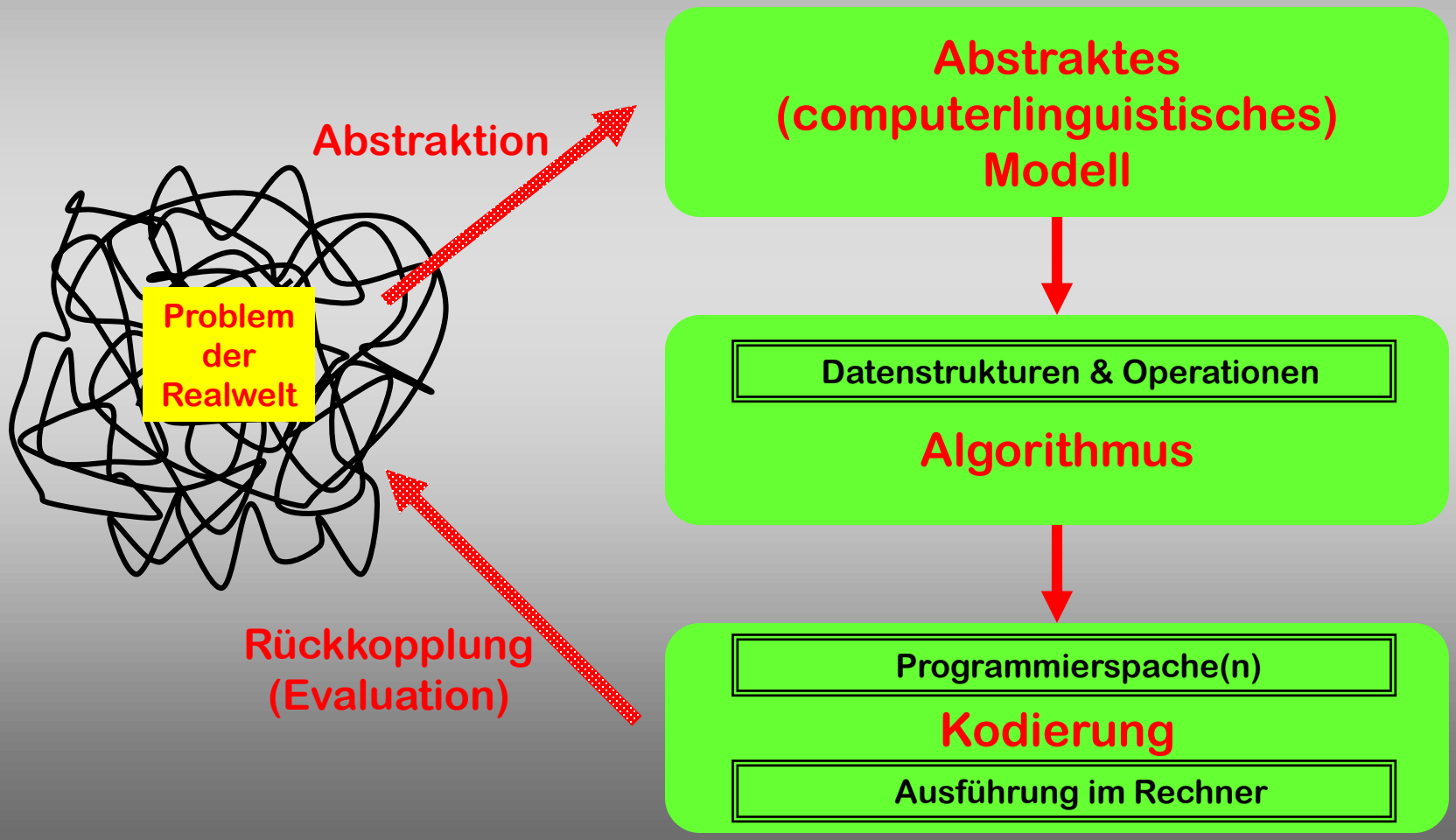
Vorlesung im WiSe 2018/19
(B-GSW-12)

Prof. Dr. Udo Hahn

Lehrstuhl für Computerlinguistik
Institut für Germanistische Sprachwissenschaft
Friedrich-Schiller-Universität Jena

http://www.julielab.de

Informatischer Problemlösungszyklus



Informatischer Problemlösungszyklus

- **Modellbildung**
  - **Abstraktion von allen unwesentlichen Details der Problemstellung im Hinblick auf die algorithmische Lösung**
  - **Spezifikation der logischen Abhängigkeiten zwischen problemlösungsrelevanten Objekten**
  - **(computer)linguistisches Wissen**

Informatischer Problemlösungszyklus

- Algorithmisierung
  - Übersetzung der modellbezogenen Spezifikation in
    - eine Menge von Objekten (Datenstrukturen) mit bestimmten Eigenschaften und Beziehungen zueinander
    - die erlaubten Operationen auf diesen Objekten
  - Algorithmus: (möglichst präzise) Beschreibung einer Folge zulässiger Operationen auf den Objekten, um das Problem zu lösen
  - Computerlinguistische Kernexpertise – verlangt informatisches Grundlagenwissen

Informatischer Problemlösungszyklus

- Kodierung (Programmierung)
  - Übersetzung der algorithmischen Spezifikation in Konstrukte einer (geeigneten) Programmiersprache
- Ausführung des Programms
  - Hier erst Bezug auf konkrete Maschinen (Datenstrukturen und Algorithmen sind abstrakte Konstruktionen)
  - Test-Modifikationszyklus ... Dokumentation !
  - Informatisches Know-How

Morphologische Prozesse: Flexion - Deflexion

- Kombination von Grundformen mit Flexionsaffixen (Kasus, Numerus, Tempus)
  - Deklination
    - Land: Land, Landes, Lande, Länder, Ländern
  - Konjugation
    - landen: lande, landest, landet, landeten , gelandet
- primär syntaktische, nur minimale semantische Information, kein grundlegender Wortartwechsel

Morphologische Prozesse: Derivation - Dederivation

- Kombination von Grundformen mit Derivationsaffixen
  - Land: landen, verlanden, anlanden,
  - Land: Landung, Verlandung , Anlandung
  - Land: ländlich, verländlichen, Verländlichung
- modifizierende semantische Information, häufig mit Wortartwechsel verbunden

Morphologische Prozesse: Komposition - Dekomposition

- Kombination von Grundformen mit Grundformen (mittels Fugeninfixen)
  - Land: Landnahme, Landflucht, Landgang
  - Land: Heimatland, Ausland, Bauland
  - Land: Landesrekord, Landesverrat, Landsmann
  - Land: Inlandsflug, Landesratspräsidentengattin
- starke semantische Modifikation, fast keine Wortartwechsel
  - … aber: Rotkehlchen, Weichteile

Lemmatisierung

Eingabe	Lemma	
Töchtern	Tochter	
Hauses	Haus	
sagte	sagen	
Spiegelungen	Spiegelung	
leichter	leicht	
verlängerte	verlängert	
	verlängern	

Lemmatisierung vs. Stemming

Eingabe	Lemma	Stem/Stemming
Töchtern	Tochter	Töchter
Hauses	Haus	Haus
sagte	sagen	sagen, sag
Spiegelungen	Spiegelung	Spiegel
leichter	leicht	leicht
verlängerte	verlängert	läng
	verlängern	läng

Lemmatisierung vs. Stemming

Eingabe	Lemma	Stem/Stemming
Töchtern	Tochter	Töchter
Hauses	Haus	Haus
sagte	sagen	sagen, sag
Spiegelungen	Spiegelung	Spiegel
leichter	leicht	leicht
verlängerte	verlängert	läng
	verlängern	läng

Bestandteile der Problemlösung für morphologisches Stemming

- Linguistisches Wissen
  - Morphologische Struktur von Wörtern:

    $\underline{\text{WORT}} \eqsim \text{AFFIX}_1 \otimes \ldots \otimes \text{AFFIX}_k \otimes \text{STAMM} \otimes$

    $\otimes \text{AFFIX}_{k+1} \otimes \ldots \otimes \text{AFFIX}_n$

  - $\text{AFFIX}_{1..k}$ heißen Präfixe, $\text{AFFIX}_{k+1..n}$ Suffixe
  - Deklarativ (Strukturbeschreibung)
- Computerlinguistisches Wissen
  - Suffixabtrennungsalgorithmus (suffix stripping):

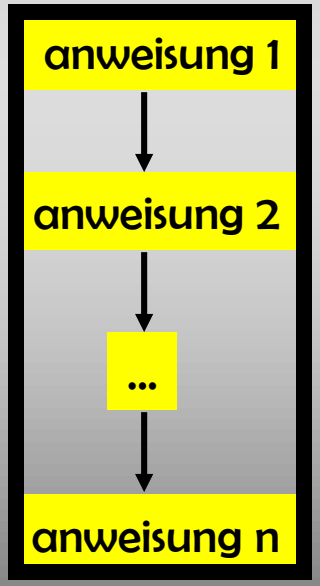
    $\text{STAMM} \otimes \text{AFFIX}_{k+1} \otimes \ldots \otimes \text{AFFIX}_n \rightarrow \text{STAMM}$
  - Prozedural (Aktionsbeschreibung)

Algorithmische Sprachkonstrukte
Anweisungsfolge

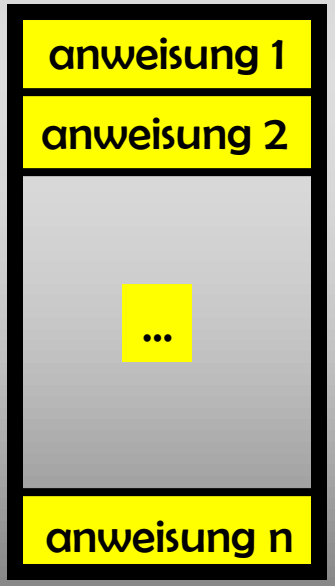
PSEUDOCODE

```
anweisung 1;
anweisung 2;
...
...
anweisung n;
```

FLUSSDIAGRAMM



STRUKTOGRAMM

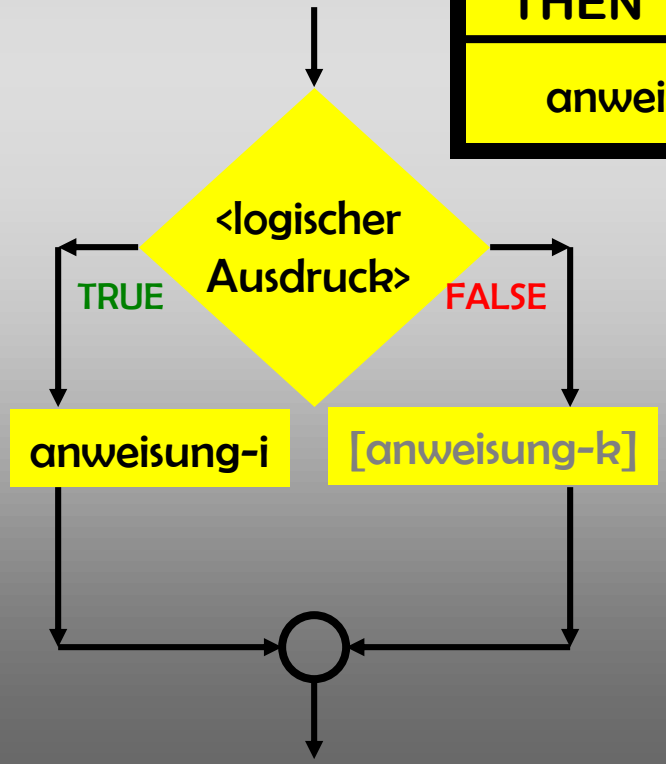


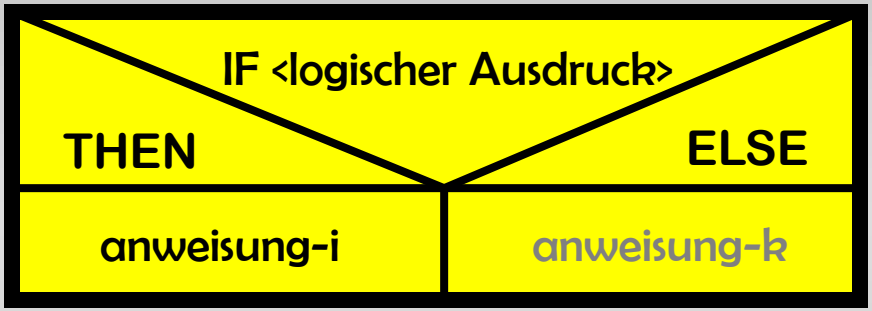
Algorithmische Sprachkonstrukte
Bedingte Anweisungen (IF)

PSEUDOCODE **FLUSSDIAGRAMM** **STRUKTOGRAMM**

IF <logischer Ausdruck>
THEN anweisung-i;
(ELSE anweisung-k;)

„*falls* <logischer Ausdruck>
TRUE *führe aus:*
anweisung-i;
(sonst *führe aus:*
anweisung-k;)“



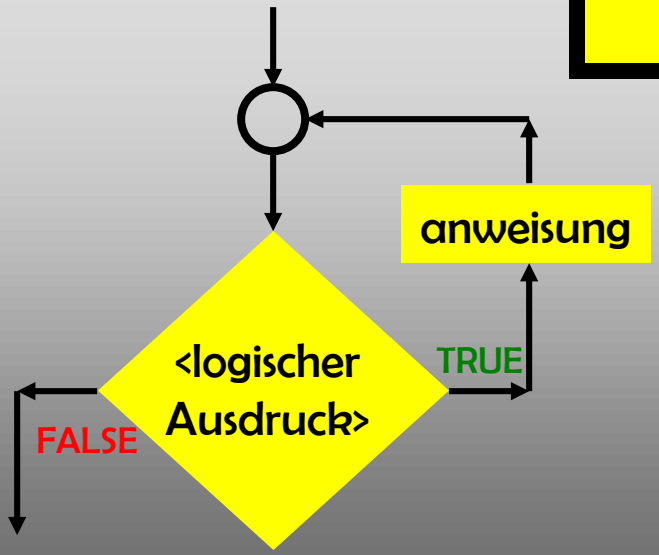


Algorithmische Sprachkonstrukte
Repetierte Anweisungen (WHILE)

PSEUDOCODE **FLUSSDIAGRAMM** **STRUKTOGRAMM**

WHILE <logischer Ausdruck> DO
anweisung;

„*solange* <logischer Ausdruck>
TRUE
führe aus:
anweisung“



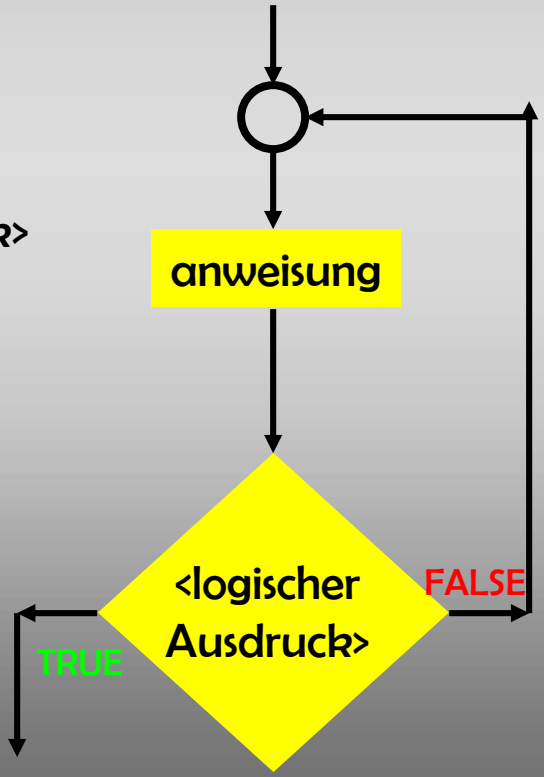
Algorithmische Sprachkonstrukte
Repetierte Anweisungen (REPEAT)

PSEUDOCODE

REPEAT anweisung;
UNTIL <logischer Ausdruck>

„ führe aus:
anweisung
solange <logischer Ausdruck>
FALSE“

FLUSSDIAGRAMM



STRUKTOGRAMM

REPEAT	anweisung
UNTIL	<logischer Ausdruck>

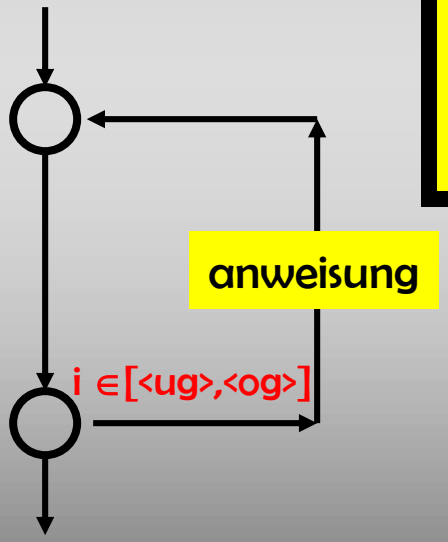
Algorithmische Sprachkonstrukte
Repetierte Anweisungen (FOR)

PSEUDOCODE

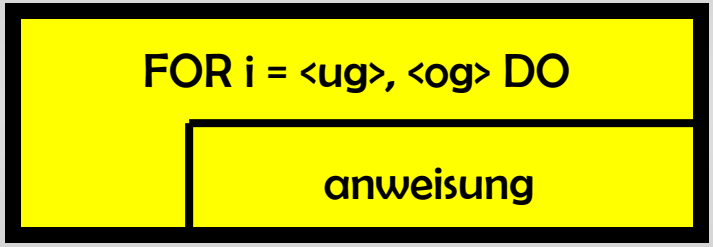
FOR i=<ug>,<og> DO
anweisung;

„ führe aus:
anweisung
solange i ∈[<ug>,<og>]“

FLUSSDIAGRAMM



STRUKTOGRAMM



Porter-Stemmer

[Porter 1980]

- (Vereinfachte) morphologische Struktur von englischen Wörtern
  - Ein Stamm, gefolgt von einem oder mehreren morphologischen Affixen ($\text{STAMM} \otimes \text{AFFIX}_{k+1} \otimes \ldots \otimes \text{AFFIX}_n$)
- (Porter-)Stemming
  - Verfahren zur Reduktion eines beliebigen englischen Eingabeworts auf seinen (morphologisch oft nicht-kanonischen) Stamm durch Eliminierung/Transformation aller Affixe
  - Regelbasierter, heuristischer Ansatz
    - *abate, abatements, abated* → *abat*
    - 6 Regelsätze in sequentieller Ordnung

      (<condition>) IF →THEN-Regeln

Porter-Stemmer – Definitionen I

[Porter 1980]

- VOKAL
  - Die Buchstaben ‚A‘, ‚E‘, ‚I‘, ‚O‘, ‚U‘ und ‚Y‘
- KONSONANT
  - Jeder andere Buchstabe
- Damit haben alle Wörter die Form
  - $(C)\ (VC)^m\ (V);\ m \geq 0$
  - C : Folge von einem oder mehreren Konsonanten
  - V : Folge von einem oder mehreren Vokalen
  - Beispiel: *troubles* = CVCVC

Porter-Stemmer – Definitionen I

[Porter 1980]

- VOKAL
  - Die Buchstaben ‚A‘, ‚E‘, ‚I‘, ‚O‘, ‚U‘ und ‚Y‘
- KONSONANT
  - Jeder andere Buchstabe
- Damit haben alle Wörter die Form
  - $(C)\ (VC)^m\ (V);\ m \geq 0$
  - C : Folge von einem oder mehreren Konsonanten
  - V : Folge von einem oder mehreren Vokalen
  - Beispiel: *troubles* = CVCVC

Porter-Stemmer – Definitionen I

[Porter 1980]

- VOKAL
  - Die Buchstaben ‚A‘, ‚E‘, ‚I‘, ‚O‘, ‚U‘ und ‚Y‘
- KONSONANT
  - Jeder andere Buchstabe
- Damit haben alle Wörter die Form
  - $(C)\ (VC)^m\ (V);\ m \geq 0$
  - C : Folge von einem oder mehreren Konsonanten
  - V : Folge von einem oder mehreren Vokalen
  - Beispiel: *troubles* = CVCVC

Porter-Stemmer – Definitionen I

[Porter 1980]

- VOKAL
  - Die Buchstaben ‚A‘, ‚E‘, ‚I‘, ‚O‘, ‚U‘ und ‚Y‘
- KONSONANT
  - Jeder andere Buchstabe
- Damit haben alle Wörter die Form
  - $(C)\ (VC)^m\ (V);\ m \geq 0$
  - C : Folge von einem oder mehreren Konsonanten
  - V : Folge von einem oder mehreren Vokalen
  - Beispiel: *troubles* = CVCVC

Porter-Stemmer – Definitionen I

[Porter 1980]

- VOKAL
  - Die Buchstaben ‚A‘, ‚E‘, ‚I‘, ‚O‘, ‚U‘ und ‚Y‘
- KONSONANT
  - Jeder andere Buchstabe
- Damit haben alle Wörter die Form
  - $(C)\ (VC)^m\ (V);\ m \geq 0$
  - C : Folge von einem oder mehreren Konsonanten
  - V : Folge von einem oder mehreren Vokalen
  - Beispiel: *troubles* = CVCVC

Porter-Stemmer – Definitionen I

[Porter 1980]

- VOKAL
  - Die Buchstaben ‚A‘, ‚E‘, ‚I‘, ‚O‘, ‚U‘ und ‚Y‘
- KONSONANT
  - Jeder andere Buchstabe
- Damit haben alle Wörter die Form
  - (C) $(VC)^m$ (V); $m \geq 0$
  - C : Folge von einem oder mehreren Konsonanten
  - V : Folge von einem oder mehreren Vokalen
  - Beispiel: *troubles* = CVCVC

Porter-Stemmer – Definitionen I

[Porter 1980]

- VOKAL
  - Die Buchstaben ‚A', ‚E', ‚I', ‚O', ‚U' und ‚Y'
- KONSONANT
  - Jeder andere Buchstabe
- Damit haben alle Wörter die Form
  - $(C)\ (VC)^m\ (V);\ m \geq 0$
  - C : Folge von einem oder mehreren Konsonanten
  - V : Folge von einem oder mehreren Vokalen
  - Beispiel: *troubles* = CVCVC
- *Longest matching* hat Priorität bei Regelauswahl innerhalb einer Klasse

Porter-Stemmer - Definitionen II

[Porter 1980]

- m()
  - Gibt die Anzahl von Vokal-Konsonantensequenzen im aktuellen Stamm zurück (<..> optional)
    - <c><v> ergibt ‚0' (cry ← cry-ing)
    - <c>vc<v> ergibt ‚1' (care ← car-ing, scare ← scar-ing)
    - <c>vcvc<v> ergibt ‚2' (probab ← probab-ility)
- *χ: Stamm endet mit Buchstaben χ (χ aus A..Z)
- *v*: Stamm enthält Vokal
- *d : Stamm enthält Doppelkonsonant (z.B. ‚LL')
- *o : Stamm endet in der Form Konsonant-Vokal-Konsonant; 2. Konsonant nicht ‚W', ‚X' oder ‚Y'

Porter-Stemmer – Schritte für das Englische

[Porter 1980]

1. Eliminierung von Pluralendungen und 3PS
2. Eliminierung von Past Tense und Verlaufsform bei Verben
3. Y→I Transformation
4. Derivationsmorphologie I: Doppelsuffixe
5. Derivationsmorphologie II: Einzelsuffixe
6. Clean-up (Aufräumen)

Schritt 1: Plurale, 3.PS Verb

SSES	→	SS	caresses	→	caress
IES	→	I	ponies	→	poni
			ties	→	ti
SS	→	SS	caress	→	caress
S	→	ε	cats	→	cat

Schritt 2:
Verb Past Tense und Verlaufsform

(m> 1)	EED	→	EE	feed	→	feed
				agreed	→	agree
(*v*)	ED	→	ε	plastered	→	plaster
				bled	→	bled
(*v*)	ING	→	ε	motoring	→	motor
				sing	→	sing

	AT	→	ATE	conflat(ed)	→	conflate
	BL	→	BLE	troubl(ing)	→	trouble
	IZ	→	IZE	siz(ed)	→	size
(*d & !(*L or *S or *Z))		→	single letter	hopp(ing)	→	hop
				tann(ed)	→	tan
				fall(ing)	→	fall
				hiss(ing)	→	hiss
				fizz(ed)	→	fizz
(m=1 & *o)		→	E	fail(ing)	→	fail
				fil(ing)	→	file

Schritt 3:
‚-y‘ → ‚-i‘

(*v*) Y → I	happy → happi sky → sky

Schritt 4: Doppel- in Einzelsuffixe

(m > 0)	ATIONAL	→ ATE	relational	→	relate
(m > 0)	TIONAL	→ TION	conditional	→	condition
			rational	→	rational
(m > 0)	ENCI	→ ENCE	valenci	→	valence
(m > 0)	ANCI	→ ANCE	hesitanci	→	hesitance
(m > 0)	IZER	→ IZE	digitizer	→	digitize
(m > 0)	ABLI	→ ABLE	conformabli	→	conformable
(m > 0)	ALLI	→ AL	radicalli	→	radical
(m > 0)	ENTLI	→ ENT	differentli	→	different
(m > 0)	ELI	→ E	vileli	→	vile
(m > 0)	OUSLI	→ OUS	analogousli	→	analogous
(m > 0)	IZATION	→ IZE	vietnamization	→	vietnamize
(m > 0)	ATION	→ ATE	predication	→	predicate
(m > 0)	ATOR	→ ATE	operator	→	operate
(m > 0)	ALISM	→ AL	feudalism	→	feudal
(m > 0)	IVENESS	→ IVE	decisiveness	→	decisive
(m > 0)	FULNESS	→ FUL	hopefulness	→	hopeful
(m > 0)	OUSNESS	→ OUS	callousness	→	callous
(m > 0)	ALITI	→ AL	formaliti	→	formal
(m > 0)	IVITI	→ IVE	sensitiviti	→	sensitive
(m > 0)	BILITI	→ BLE	sensibiliti	→	sensible

(m > 0)	ICATE	→ IC	triplicate	→	triplic
(m > 0)	ATIVE	→ ε	formative	→	form
(m > 0)	ALIZE	→ AL	formalize	→	formal
(m > 0)	ICITI	→ IC	electriciti	→	electric
(m > 0)	FUL	→ ε	hopeful	→	hope
(m > 0)	NESS	→ ε	goodness	→	good

Schritt 5: Einzelsuffixe

(m > 1)	AL	→	ε	revival	→	reviv
(m > 1)	ANCE	→	ε	allowance	→	allow
(m > 1)	ENCE	→	ε	inference	→	infer
(m > 1)	ER	→	ε	airliner	→	airlin
(m > 1)	IC	→	ε	gyroscopic	→	gyroscop
(m > 1)	ABLE	→	ε	defensible	→	defens
(m > 1)	ANT	→	ε	irritant	→	irrit
(m > 1)	EMENT	→	ε	replacement	→	replac
(m > 1)	MENT	→	ε	adjustment	→	adjust
(m > 1)	ENT	→	ε	dependent	→	depend
(m > 1)	(*S or *T) & ION	→	ε	adoption	→	adopt
(m > 1)	OU	→	ε	homologou	→	homolog
(m > 1)	ISM	→	ε	communism	→	commun
(m > 1)	ATE	→	ε	activate	→	activ
(m > 1)	ITI	→	ε	angulariti	→	angular
(m > 1)	OUS	→	ε	homologous	→	homolog
(m > 1)	IVE	→	ε	effective	→	effect
(m > 1)	IZE	→	ε	bowdlerize	→	bowdler

Schritt 6:
Clean-up (Aufräumen)

(m > 1)	E → ε	probate → probat rate → rate
(m = 1 & ! *o)	E → ε	cease → ceas

(m > 1 & *d *L)	→ [single letter]	controll → control roll → roll

Porter-Stemmer

[Porter 1980]

- Eine kodierungsnähere Lösung des gleichen Problems …

Porter-Stemmer - Hilfsfunktionen

[Porter 1980]

- m()
  - Gibt die Anzahl von Vokal-Konsonantensequenzen im aktuellen Stamm zurück
    - <c><v> ergibt ‚0' (cry ← cry-ing)
    - <c>vc<v> ergibt ‚1' (care ← car-ing, scare ← scar-ing)
    - <c>vcvc<v> ergibt ‚2' (probab ← probab-ility)
- r(String str)
  - Falls die m-Funktion > 0, setze das aktuelle Suffix auf „str"
- cvc(Int pos)
  - Prüft, ob die vorangehenden drei Buchstaben vor ‚pos' Konsonant-Vokal-Konsonant sind

Schritt 1: Plurale, ‚-ing', ‚-ed'

```
if b[k] == 's'
  if ends("sses")
    k -= 2
  else if ends("ies")
    setto("i")
  else if b[k-1] != 's'
    k--
if ends("eed")
  if m() > 0
    k--
else if ends("ed") || ends("ing") &&
vowelinstem()
  k = j
  if ends("at")
    setto("ate")
  else if ends("bl")
    setto("ble")
  else if ends("iz")
    setto("ize")
  else if doublec(k)
     k--
     int ch = b[k]
     if ch=='l' || ch=='s' || ch=='z'
        k++
  else if (m() == 1 && cvc(k))
     setto("e")
```

k : aktuelle Endposition des Prüftokens b

Steht für: k ← k-2

!= steht für: =/=

Steht für: k ← k-1

|| steht für: ODER;
&& steht für: UND

j : Position vor Endung „ed" oder „ing"

Steht für: k ← k+1

- Possesses → possess
- Ponies → poni
- Operatives → operative
- **Markedly → markedly**
- Interesting → interest
- **Confess → confess**
- Consumables → consumable
- Realizes → realize
- Infuriating → Infuriate
- Fables → fable
- Fated → fate

Schritt 2:
‚-y‘ → ‚-i‘, falls Vokal im Stamm

```
if ends("y") && vowelinstem()
   b[k] = 'i'
```

- Coolly → coolli
- Furry → furri
- Fry → fry
- Grey → grei
- **Interestingly → Interestingli**

Schritt 3: Doppel- in Einzelsuffixe

```
switch b[k-1]
  case 'a':
    if ends("ational") -> r("ate")
    if ends("tional") -> r("tion")
  case 'c':
    if ends("enci")) -> r("ence")
    if ends("anci")) -> r("ance")
  case 'e':
    if ends("izer") -> r("ize")
  case 'l':
    if ends("bli") -> r("ble")
    if ends("alli") -> r("al")
    if ends("entli") -> r("ent")
    if ends("eli") -> r("e")
    if ends("ousli") -> r("ous")
  case 'o':
    if ends("ization") -> r("ize")
    if ends("ation") -> r("ate")
    if ends("ator") -> r("ate")
  case 's':
    if ends("alism") -> r("al")
    if ends("iveness") -> r("ive")
    if ends("fulness") -> r("ful")
    if ends("ousness") -> r("ous")
  case 't':
    if ends("aliti") -> r("al")
    if ends("iviti") -> r("ive")
    if ends("biliti") -> r("ble")
   case 'g':
    if ends("logi") -> r("log")
```

- **Rational → rational**
- Optional → option
- Operational → operate
- Possibly→ possibli → possible
- Really → realli → realli
- Realization → realize
- Feudalism → feudal
- Playfulness → playful
- **Liveness → liveness**

Schritt 4:
Suffixe wie ‚-ness', ‚-full'

```
switch b[k]
  case 'e':
    if ends("icate") -> r("ic")
    if ends("ative") -> r("")
    if ends("alize") -> r("al")
  case 'i':
    if ends("iciti") -> r("ic")
  case 'l':
    if ends("ical") -> r("ic")
    if ends("ful") -> r("")
  case 's':
    if ends("ness") -> r("")
```

- Authenticate → authentic
- Predicate → predic
- **Realize → realize**
- Felicity → feliciti → felic
- Practical → practic
- Playful → play
- **Gleeful → gleeful**
- Largeness → large

Schritt 5: Formen wie ‚-ant', ‚-ence'

```
switch b[k-1]
 case 'a': b[k-1] == 'a'
   if ends("al") -> break
 case 'c': b[k-1] == 'c'
   if ends("ance") -> break
   if ends("ence") -> break
 case 'e': b[k-1] == 'e'
   if ends("er") -> break
 case 'i': b[k-1] == 'i'
   if ends("ic") -> break
 case 'l': b[k-1] == 'l'
   if ends("able") -> break
   if ends("ible") -> break
 … more rules here...
 default: return
if m() > 1
  k = j;
```

break : lösche Endung

- Precedent → preced
- Operational → operate → oper
-
- Interestingly
- Infuriating
- Fable → fable
- Parable →parable
- Controllable → controll

Schritt 6: Entfernung finales ‚-e‘

```
if b[k] == 'e'
  int a = m()
  if a > 1 || a == 1 && !cvc(k-1)
    k--
if b[k]=='l' && doublec(k) && m() > 1
  k--
```

! steht für die Negation des nachfolgenden Ausdrucks

- Parable → parabl
- Fate → fate (cvc)
- Deflate → deflat
- Bee → bee
- Controllable → controll → control
- Petrol → petrol
- Stall → stall
- Resell → resel

Walkthrough

- **Schritt-für-Schritt-Analyse für**
  - *semantically*
  - *recognizing*
  - *destructiveness*

Schritt 1: Plurale, ‚-ing‘, ‚-ed‘

```
if b[k] == 's'
  if ends("sses")
    k -= 2
  else if ends("ies")
    setto("i")
  else if b[k-1] != 's'
    k--
if ends("eed")
  if m() > 0
    k--
else if ends("ed") || ends("ing") &&
vowelinstem()
  k = j
  if ends("at")
    setto("ate")
  else if ends("bl")
    setto("ble")
  else if ends("iz")
    setto("ize")
  else if doublec(k)
     k--
     int ch = b[k]
     if ch=='l' || ch=='s' || ch=='z'
        k++
  else if (m() == 1 && cvc(k))
     setto("e")
```

- Semantically → ?
- Destructiveness → ?
- Recognizing → ?

Schritt 1: Plurale, ‚-ing', ‚-ed'

```
if b[k] == 's'
  if ends("sses")
    k -= 2
  else if ends("ies")
    setto("i")
  else if b[k-1] != 's'
    k--
if ends("eed")
  if m() > 0
    k--
else if ends("ed") || ends("ing") &&
vowelinstem()
  k = j
  if ends("at")
    setto("ate")
  else if ends("bl")
    setto("ble")
  else if ends("iz")
    setto("ize")
  else if doublec(k)
     k--
     int ch = b[k]
     if ch=='l' || ch=='s' || ch=='z'
        k++
  else if (m() == 1 && cvc(k))
     setto("e")
```

- Semantically → semantically
- Destructiveness → destructiveness
- Recognizing → recognize

Schritt 2:
‚-y‘ → ‚-i‘, falls Vokal im Stamm

```
if ends("y") && vowelinstem()
   b[k] = 'i'
```

- Semantically → semantically → ?
- Destructiveness → destructiveness → ?
- Recognizing → recognize → ?

Schritt 2:
‚-y‘ → ‚-i‘, falls Vokal im Stamm

```
if ends("y") && vowelinstem()
   b[k] = 'i'
```

- Semantically → semantically → semanticalli
- Destructiveness → destructiveness → destructiveness
- Recognizing → recognize → recognize

Schritt 3: Doppel- in Einzelsuffixe

```
switch b[k-1]
  case 'a':
    if ends("ational") -> r("ate")
    if ends("tional") -> r("tion")
  case 'c':
    if ends("enci")) -> r("ence")
    if ends("anci")) -> r("ance")
  case 'e':
    if ends("izer") -> r("ize")
  case 'l':
    if ends("bli") -> r("ble")
    if ends("alli") -> r("al")
    if ends("entli") -> r("ent")
    if ends("eli") -> r("e")
    if ends("ousli") -> r("ous")
  case 'o':
    if ends("ization") -> r("ize")
    if ends("ation") -> r("ate")
    if ends("ator") -> r("ate")
  case 's':
    if ends("alism") -> r("al")
    if ends("iveness") -> r("ive")
    if ends("fulness") -> r("ful")
    if ends("ousness") -> r("ous")
  case 't':
    if ends("aliti") -> r("al")
    if ends("iviti") -> r("ive")
    if ends("biliti") -> r("ble")
   case 'g':
    if ends("logi") -> r("log")
```

- Semantically → semantically → semanticalli → ?
- Destructiveness → destructiveness → destructiveness → ?
- Recognizing → recognize → recognize → ?

Schritt 3: Doppel- in Einzelsuffixe

```
switch b[k-1]
  case 'a':
    if ends("ational") -> r("ate")
    if ends("tional") -> r("tion")
  case 'c':
    if ends("enci")) -> r("ence")
    if ends("anci")) -> r("ance")
  case 'e':
    if ends("izer") -> r("ize")
  case 'l':
    if ends("bli") -> r("ble")
    if ends("alli") -> r("al")
    if ends("entli") -> r("ent")
    if ends("eli") -> r("e")
    if ends("ousli") -> r("ous")
  case 'o':
    if ends("ization") -> r("ize")
    if ends("ation") -> r("ate")
    if ends("ator") -> r("ate")
  case 's':
    if ends("alism") -> r("al")
    if ends("iveness") -> r("ive")
    if ends("fulness") -> r("ful")
    if ends("ousness") -> r("ous")
  case 't':
    if ends("aliti") -> r("al")
    if ends("iviti") -> r("ive")
    if ends("biliti") -> r("ble")
   case 'g':
    if ends("logi") -> r("log")
```

- Semantically →
  semantically →
  semanticalli →
  semantical
- Destructiveness →
  destructiveness →
  destructiveness →
  destructive
- Recognizing →
  recognize →
  recognize →
  recognize

Schritt 4:
Suffixe wie ‚-ness‘, ‚-full‘

```
switch b[k]
  case 'e':
    if ends("icate") -> r("ic")
    if ends("ative") -> r("")
    if ends("alize") -> r("al")
  case 'i':
    if ends("iciti") -> r("ic")
  case 'l':
    if ends("ical") -> r("ic")
    if ends("ful") -> r("")
  case 's':
    if ends("ness") -> r("")
```

- Semantically →
  semantically →
  semanticalli →
  semantical → ?
- Destructiveness →
  destructiveness →
  destructiveness →
  destructive → ?
- Recognizing →
  recognize →
  recognize →
  recognize → ?

Schritt 4:
Suffixe wie ‚-ness', ‚-full'

```
switch b[k]
  case 'e':
    if ends("icate") -> r("ic")
    if ends("ative") -> r("")
    if ends("alize") -> r("al")
  case 'i':
    if ends("iciti") -> r("ic")
  case 'l':
    if ends("ical") -> r("ic")
    if ends("ful") -> r("")
  case 's':
    if ends("ness") -> r("")
```

- Semantically →
  semantically →
  semanticalli →
  semantical →
  semantic
- Destructiveness →
  destructiveness →
  destructiveness →
  destructive →
  destructive
- Recognizing →
  recognize →
  recognize →
  recognize →
  recognize

Schritt 5: Formen wie ‚-ant', ‚-ence'

```
switch b[k-1]
 case 'a':
   if ends("al") -> break
 case 'c':
   if ends("ance") -> break
   if ends("ence") -> break
 case 'e':
   if ends("er") -> break
 case 'i':
   if ends("ic") -> break
 case 'l':
   if ends("able") -> break
   if ends("ible") -> break
 case 'v':
   if ends("ive") -> break
 … more rules here...
 default: return
if m() > 1
  k = j;
```

- Semantically →
  - semantically →
  - semanticalli →
  - semantical →
  - semantic → ?
- Destructiveness →
  - destructiveness →
  - destructiveness →
  - destructive →
  - destructive → ?
- Recognizing →
  - recognize →
  - recognize →
  - recognize →
  - recognize → ?

Schritt 5: Formen wie ‚-ant', ‚-ence'

```
switch b[k-1]
 case 'a':
   if ends("al") -> break
 case 'c':
   if ends("ance") -> break
   if ends("ence") -> break
 case 'e':
   if ends("er") -> break
 case 'i':
   if ends("ic") -> break
 case 'l':
   if ends("able") -> break
   if ends("ible") -> break
 case 'v':
   if ends("ive") -> break
 … more rules here...
 default: return
if m() > 1
  k = j;
```

- Semantically →
  semantically →
  semanticalli →
  semantical →
  semantic → semant
- Destructiveness →
  destructiveness →
  destructiveness →
  destructive →
  destructive → destruct
- Recognizing →
  recognize →
  recognize →
  recognize →
  recognize → recogn

Schritt 6: Entfernung finales ‚-e‘

```
if b[k] == 'e'
  int a = m()
  if a > 1 || a == 1 && !cvc(k-1)
    k--
if b[k]=='l' && doublec(k) && m() > 1
  k--
```

- Semantically →
  semantically →
  semanticalli →
  semantical →
  semantic →
  semant → ?
- Destructiveness →
  destructiveness →
  destructiveness →
  destructive →
  destructive →
  destruct → ?
- Recognizing →
  recognize →
  recognize →
  recognize →
  recognize →

Schritt 6: Entfernung finales ‚-e‘

```
if b[k] == 'e'
  int a = m()
  if a > 1 || a == 1 && !cvc(k-1)
    k--
if b[k]=='l' && doublec(k) && m() > 1
  k--
```

- Semantically →
  semantically →
  semanticalli →
  semantical →
  semantic →
  semant → **semant**
- Destructiveness →
  destructiveness →
  destructiveness →
  destructive →
  destructive →
  destruct → **destruct**
- Recognizing →
  recognize →
  recognize →
  recognize →
  recognize → **recognize**

Hinweise

- Martin F. Porter (1980). An algorithm for suffix stripping. In: Program, 14:130-137.
- Martin Porter's Stemming Page:
  - http://tartarus.org/~martin/PorterStemmer/
- Demo Page:
  - http://9ol.es/porter_js_demo.html
  - http://textanalysisonline.com/nltk-porter-stemmer
- Folien dieser Passage z.T. übernommen von:
  - http://www.deepsky.com/~merovech/voynich/voynich_manchu_reference_materials/PDFs/jurafsky_martin.pdf
  - https://www.eecis.udel.edu/~trnka/CISC889-11S/lectures/dan-porters.pdf