Python 1

Vorlesung “Computerlinguistische Techniken”

Alexander Koller

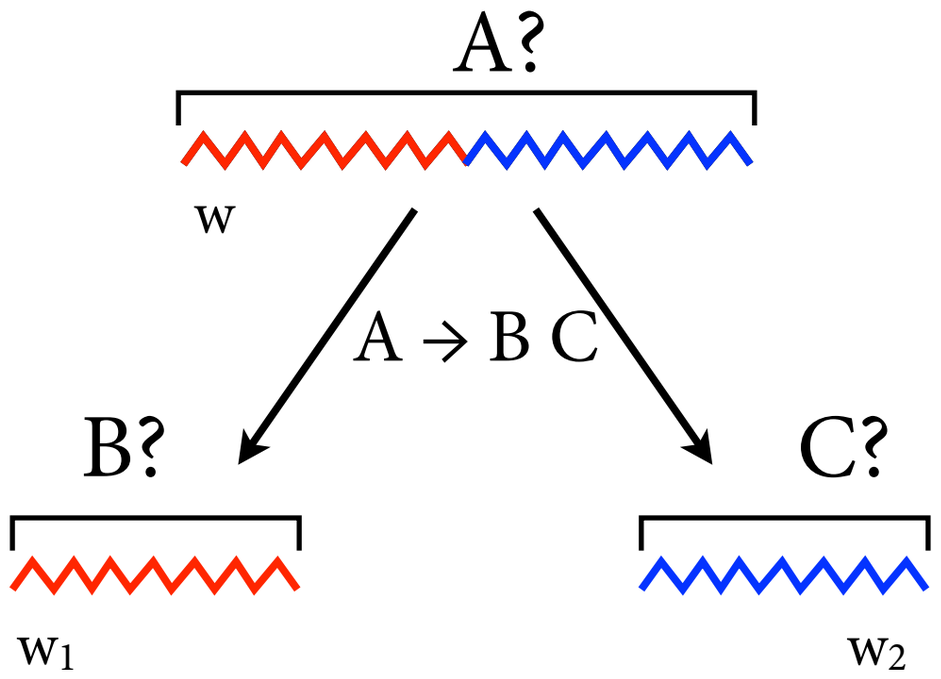
27. Oktober 2014

Übersicht

- Rest von Shift-Reduce
- Was ist Python und warum lernen wir es?
- Ausdrücke und Anweisungen
- Komplexe Datentypen und Anweisungen
- Funktionen

Recursive-Descent-Parsing

- Rekursiver Algorithmus, der für A und w die Frage "A $\Rightarrow^*$ w?" entscheidet.



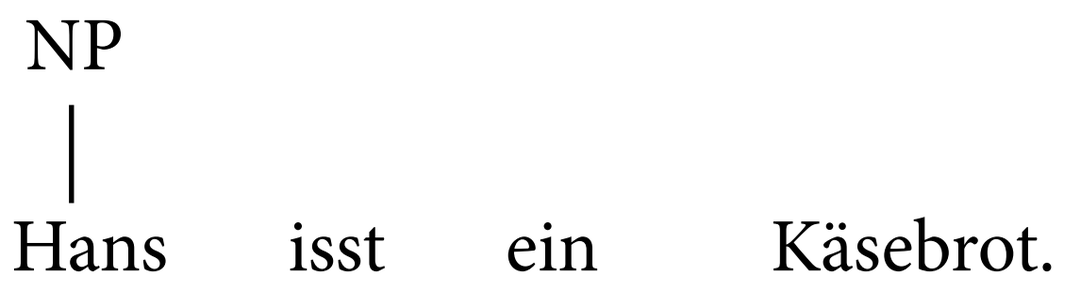
Bottom-Up-Parsing

S → NP VP	V → isst	Det → ein
NP → Det N	NP → Hans	N → Käsebrot
VP → V NP		

Hans isst ein Käsebrot.

Bottom-Up-Parsing

S → NP VP	V → isst	Det → ein
NP → Det N	NP → Hans	N → Käsebrot
VP → V NP		



Bottom-Up-Parsing

S → NP VP	V → isst	Det → ein
NP → Det N	NP → Hans	N → Käsebrot
VP → V NP		

Bottom-Up-Parsing

S → NP VP	V → isst	Det → ein
NP → Det N	NP → Hans	N → Käsebrot
VP → V NP		

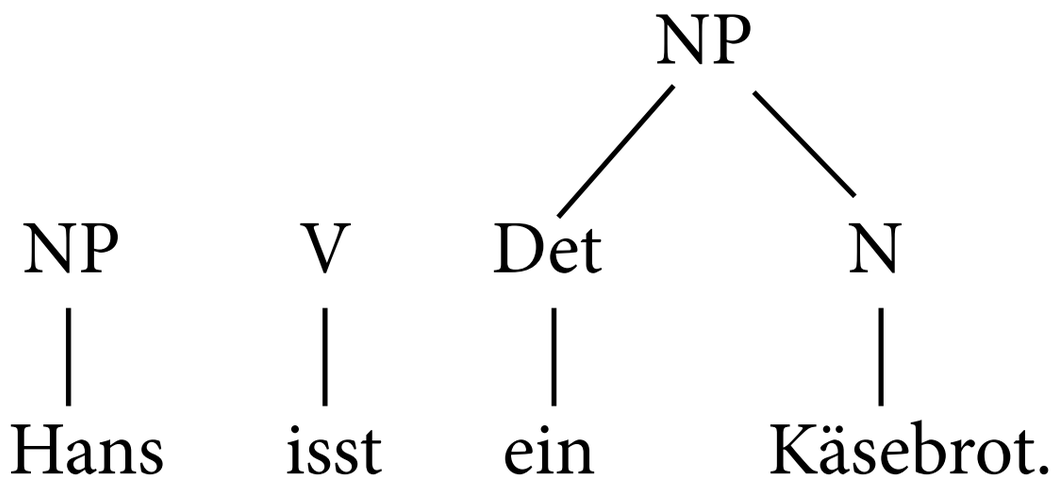
NP	V	Det	
|	|	|	
Hans	isst	ein	Käsebrot.

Bottom-Up-Parsing

S → NP VP	V → isst	Det → ein
NP → Det N	NP → Hans	N → Käsebrot
VP → V NP		

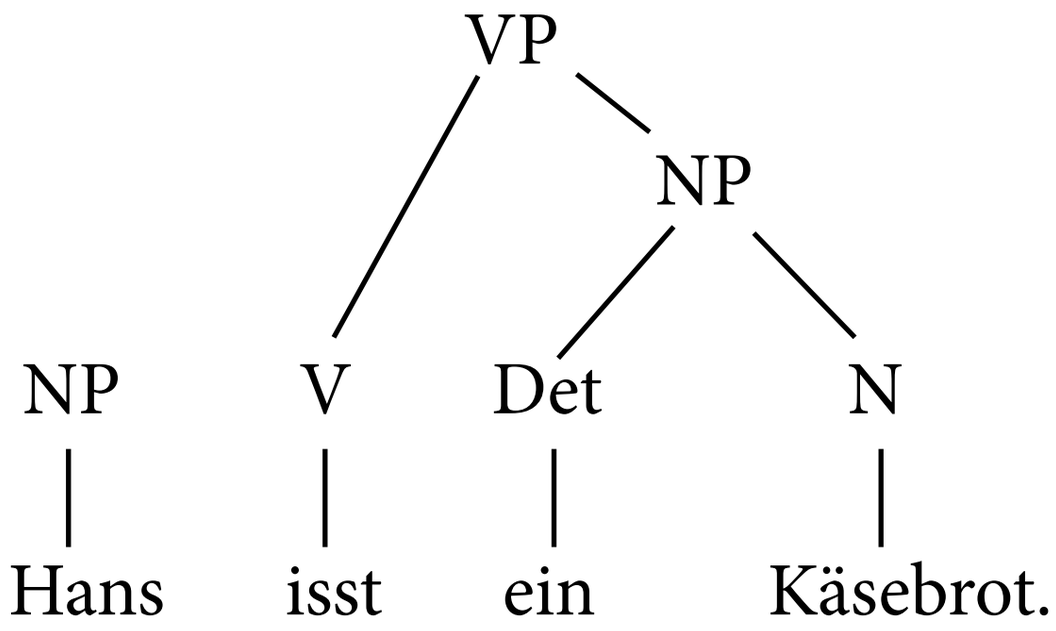
Bottom-Up-Parsing

S → NP VP	V → isst	Det → ein
NP → Det N	NP → Hans	N → Käsebrot
VP → V NP		



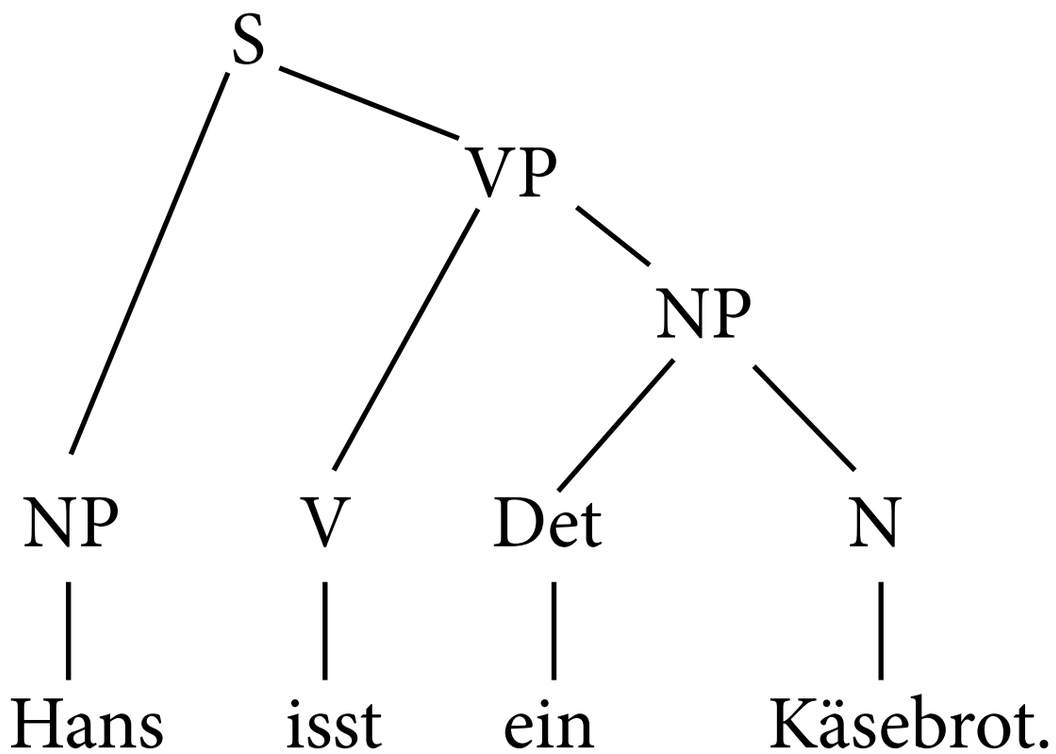
Bottom-Up-Parsing

S → NP VP	V → isst	Det → ein
NP → Det N	NP → Hans	N → Käsebrot
VP → V NP		



Bottom-Up-Parsing

S → NP VP	V → isst	Det → ein
NP → Det N	NP → Hans	N → Käsebrot
VP → V NP		



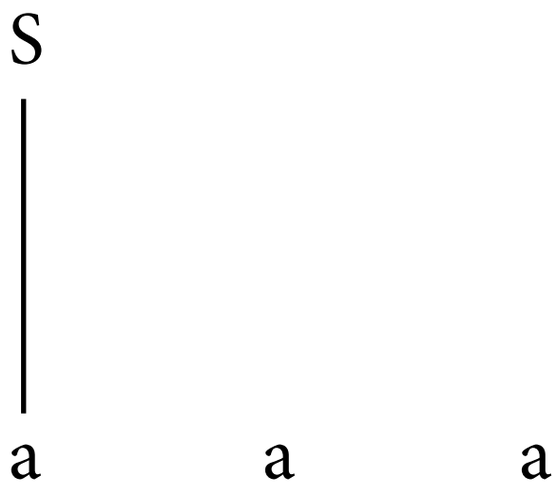
Bottom-up-Parsing

$S \rightarrow S\ S$ $S \rightarrow a$

a a a

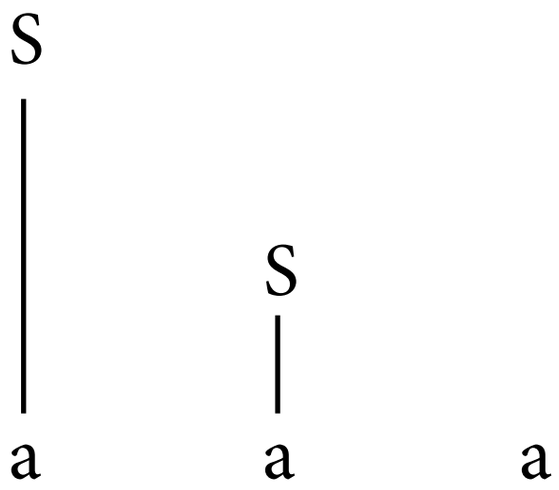
Bottom-up-Parsing

$S \rightarrow S\ S$ $S \rightarrow a$



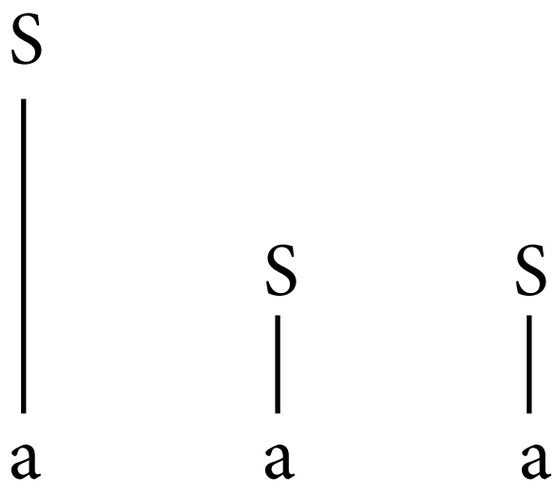
Bottom-up-Parsing

$S \rightarrow S\ S$ $S \rightarrow a$



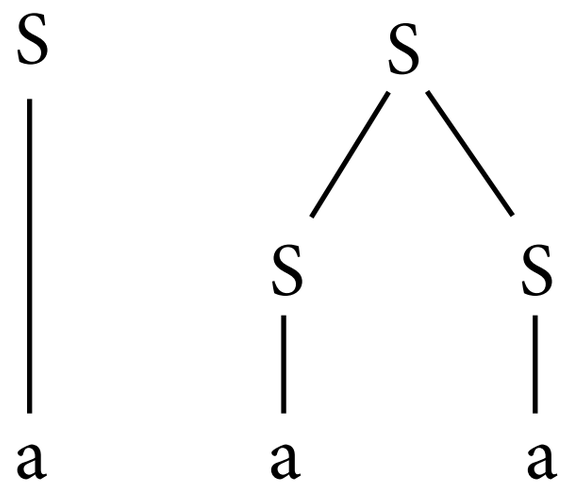
Bottom-up-Parsing

$S \rightarrow S\ S$ $S \rightarrow a$



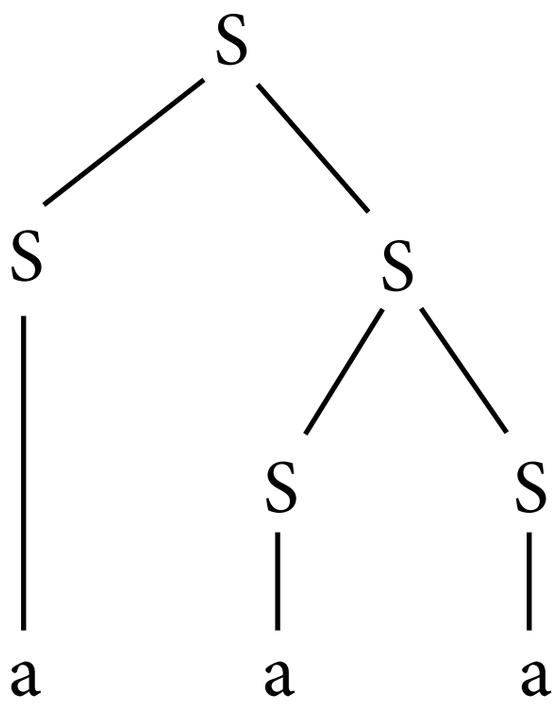
Bottom-up-Parsing

$S \rightarrow S\ S$ $S \rightarrow a$



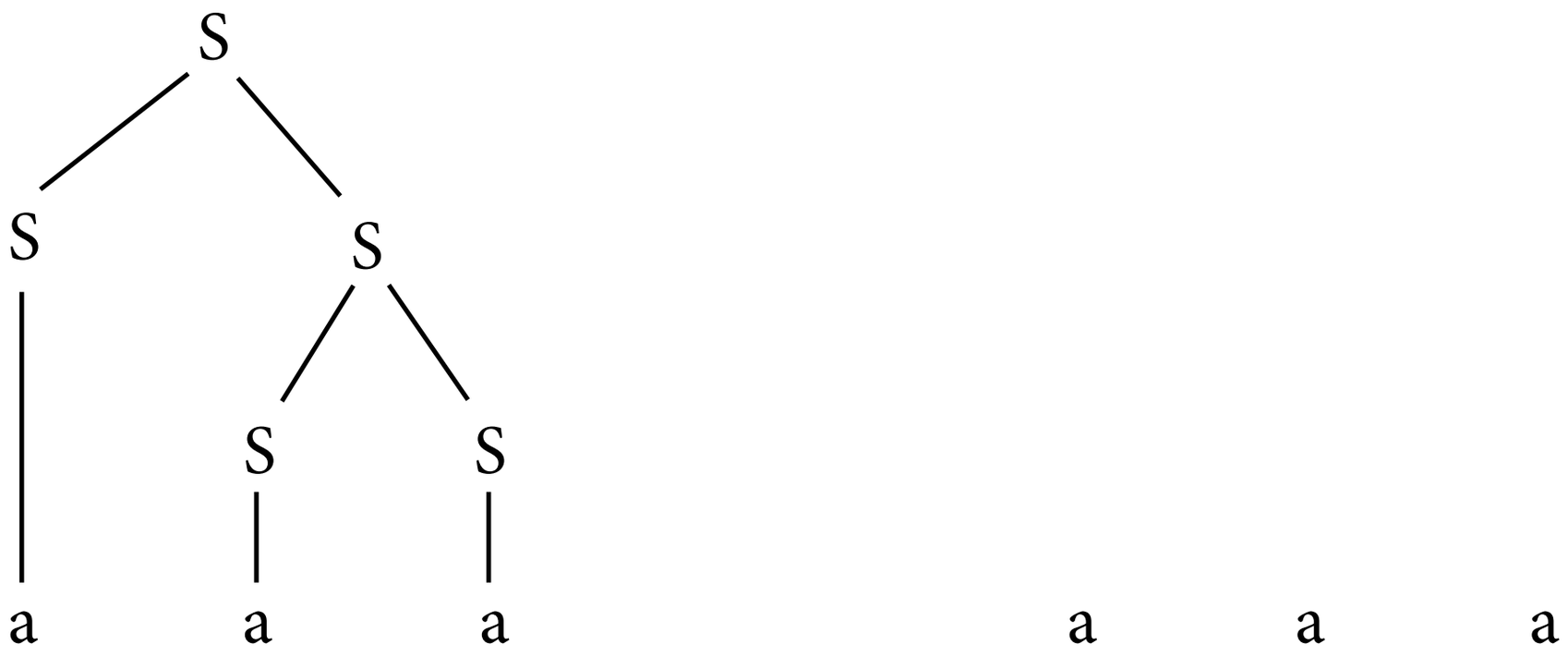
Bottom-up-Parsing

$S \rightarrow S\ S$ $S \rightarrow a$



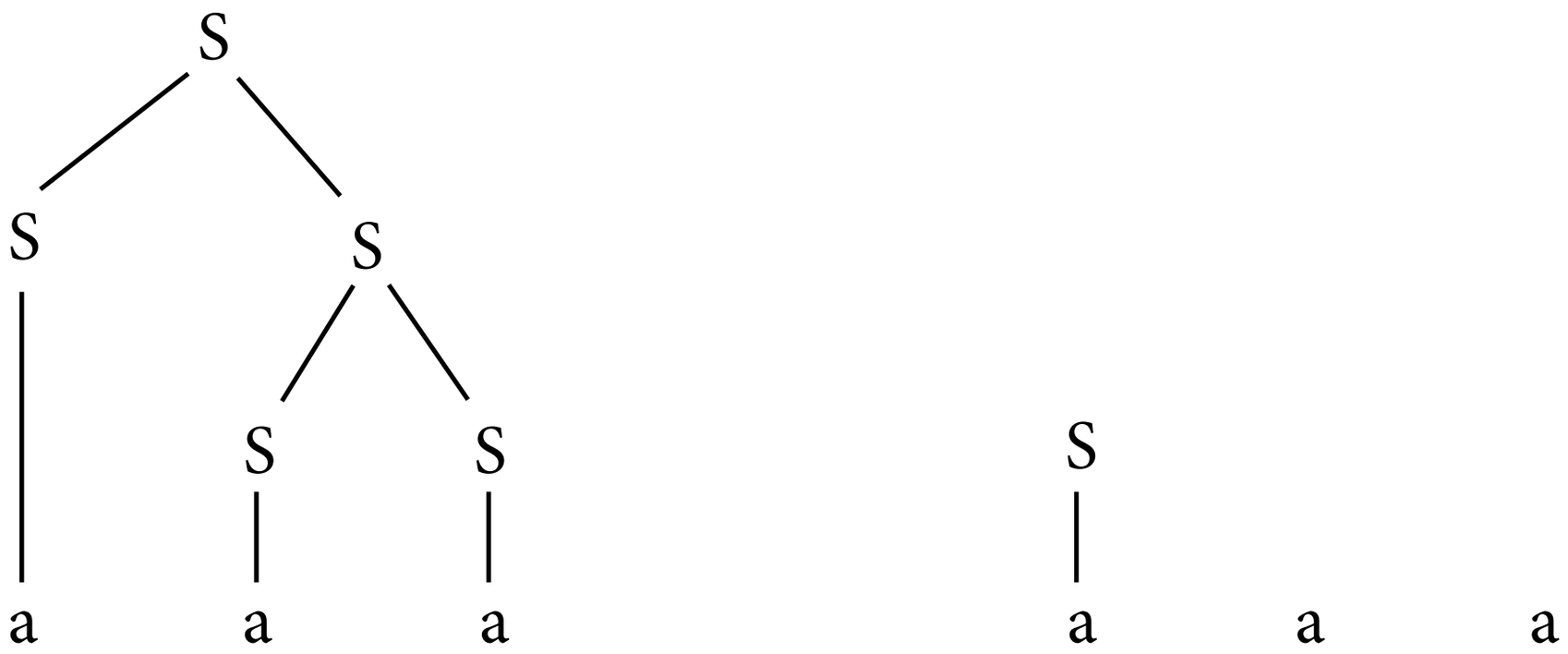
Bottom-up-Parsing

$S \rightarrow S\ S$ $S \rightarrow a$



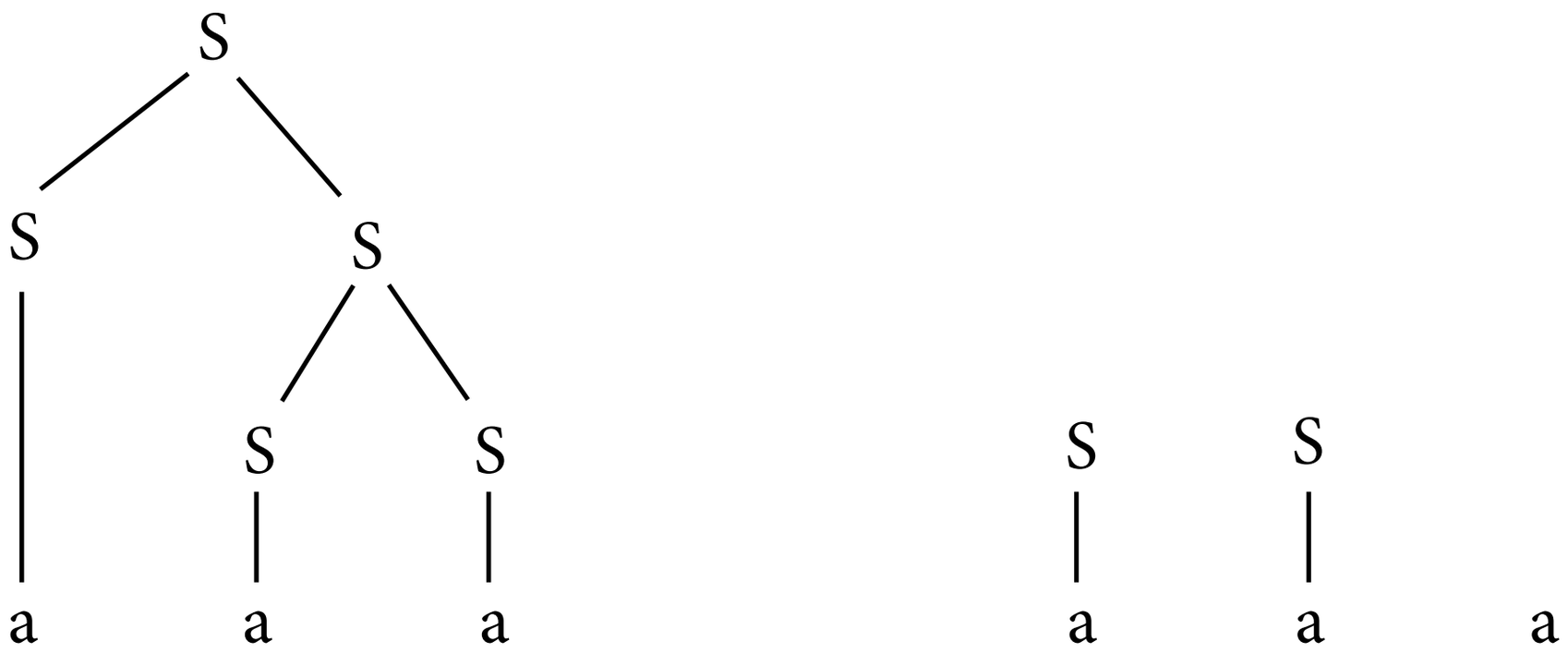
Bottom-up-Parsing

$S \rightarrow S\ S$ $S \rightarrow a$



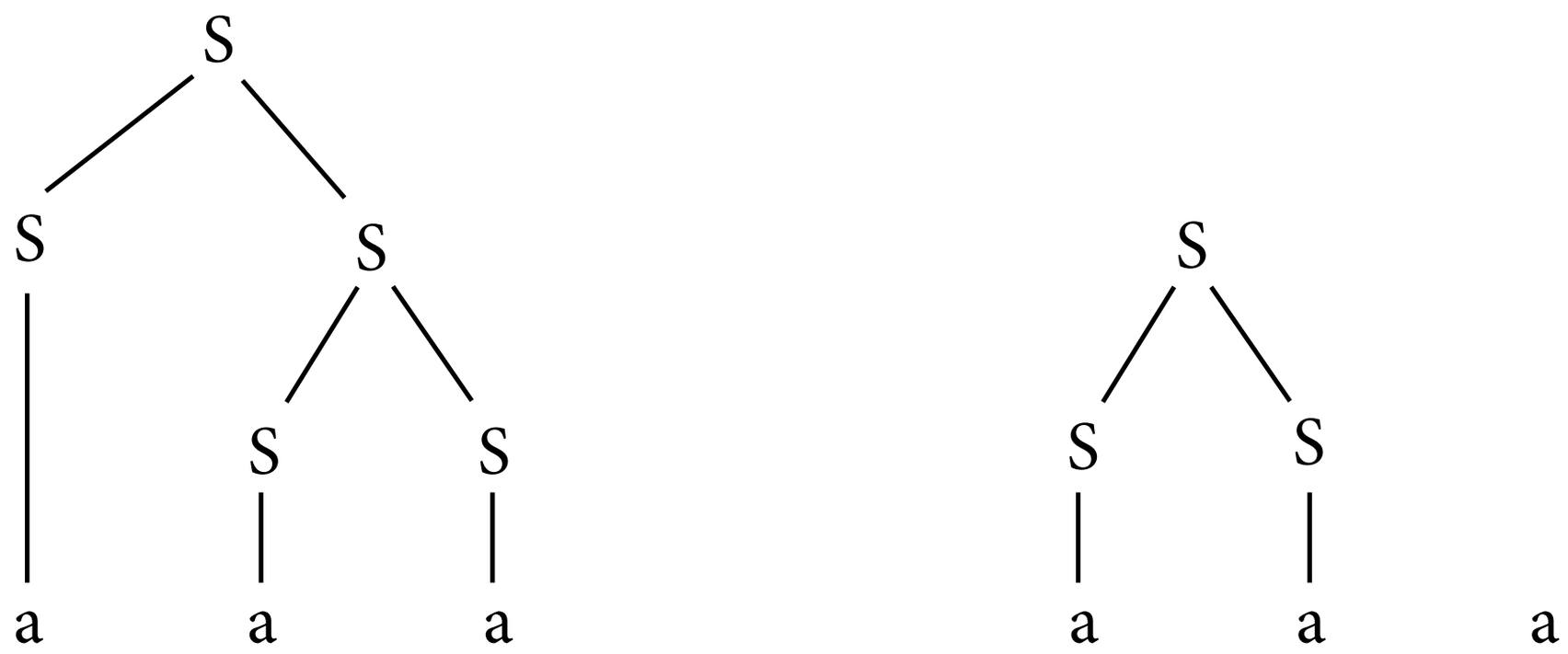
Bottom-up-Parsing

$S \rightarrow S\ S$ $S \rightarrow a$



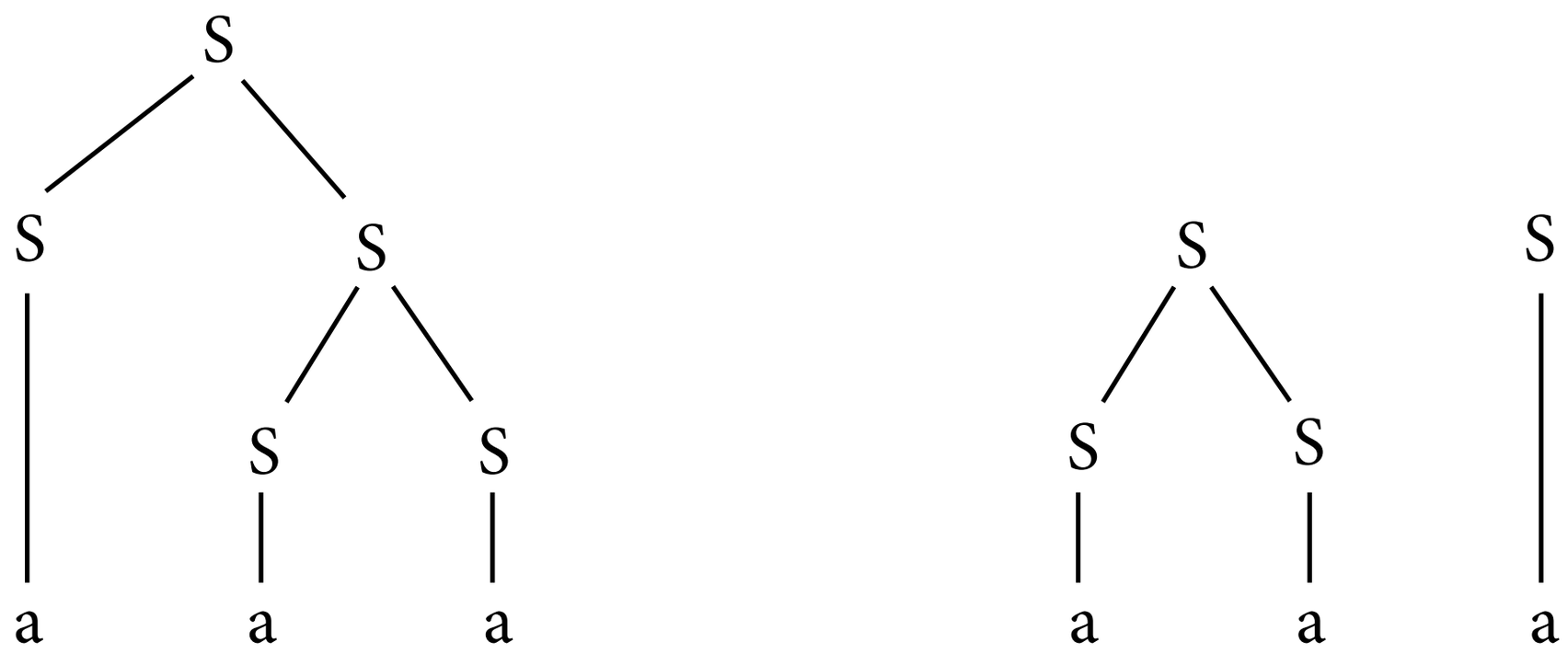
Bottom-up-Parsing

$S \rightarrow S\ S$ $S \rightarrow a$



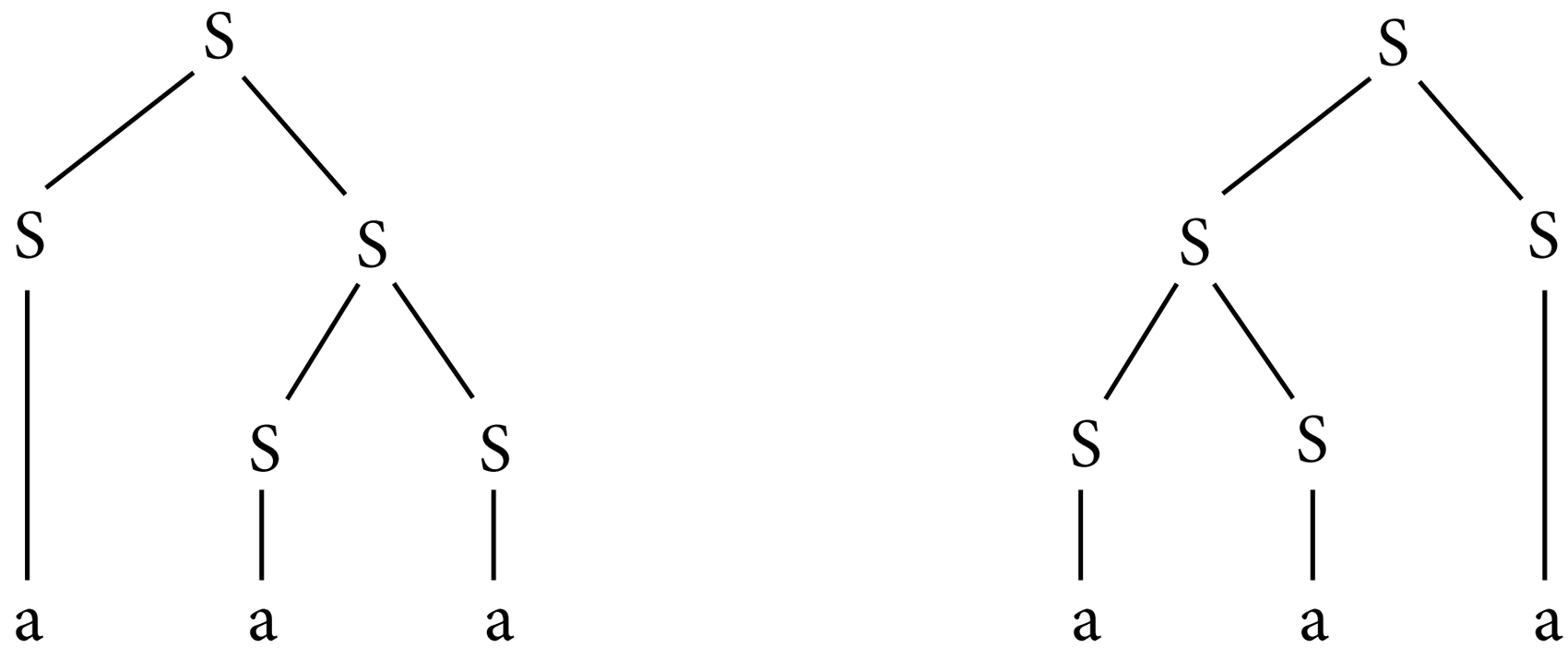
Bottom-up-Parsing

$S \rightarrow S\ S$ $S \rightarrow a$



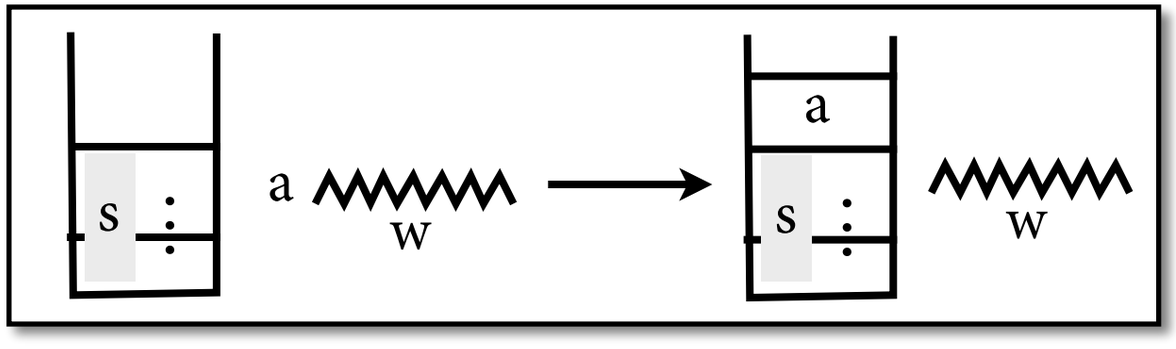
Bottom-up-Parsing

$S \rightarrow S\ S$ $S \rightarrow a$

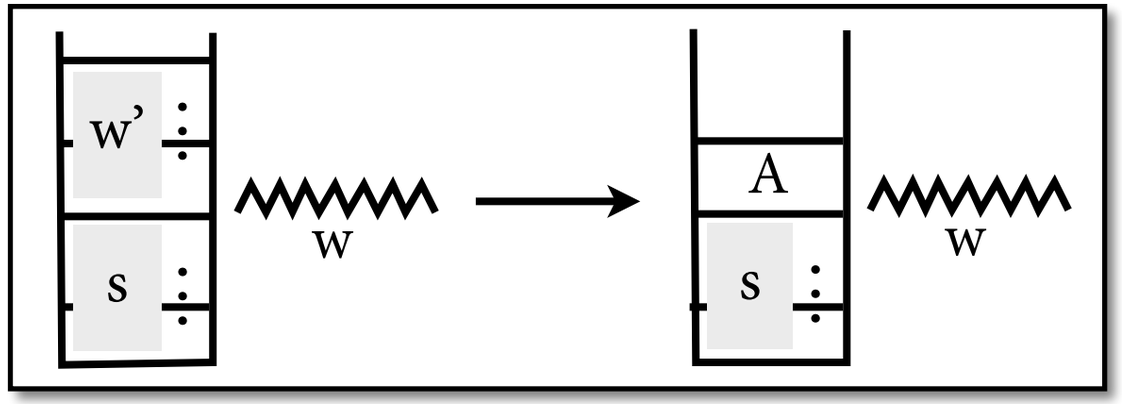


Shift-Reduce-Parsing

- Shift-Regel:
  (s, a · w) → (s · a, w)



- Reduce-Regel:
  (s · w', w) → (s · A, w) falls A → w' in P



- Start: (ε, w)

- Wende Regeln nichtdeterministisch an. Algorithmus sagt “ja”, wenn er Konfiguration (S, ε) erreicht (d.h. erreichen kann).

Shift-Reduce: Beispiel

Shift: $(s, a \cdot w) \rightarrow (s \cdot a, w)$
Reduce: $(s \cdot w', w) \rightarrow (s \cdot A, w)$

Hans isst ein Käsebrot.

Shift-Reduce: Beispiel

Shift: $(s, a \cdot w) \rightarrow (s \cdot a, w)$
Reduce: $(s \cdot w', w) \rightarrow (s \cdot A, w)$

Hans isst ein Käsebrot.

(ε, Hans isst ein K.)

Shift-Reduce: Beispiel

Shift: (s, a · w) → (s · a, w)
Reduce: (s · w', w) → (s · A, w)

(ε, Hans isst ein K.) → (Hans, isst ein K.)

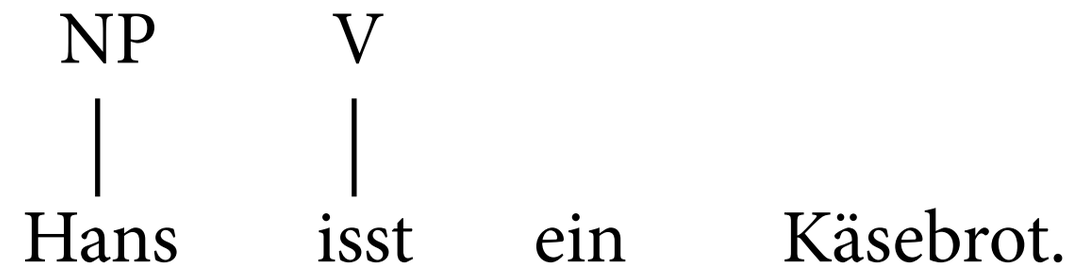
Shift-Reduce: Beispiel

Shift: (s, a · w) → (s · a, w)
Reduce: (s · w', w) → (s · A, w)

(ε, Hans isst ein K.) → (Hans, isst ein K.) → (NP, isst ein K.)

Shift-Reduce: Beispiel

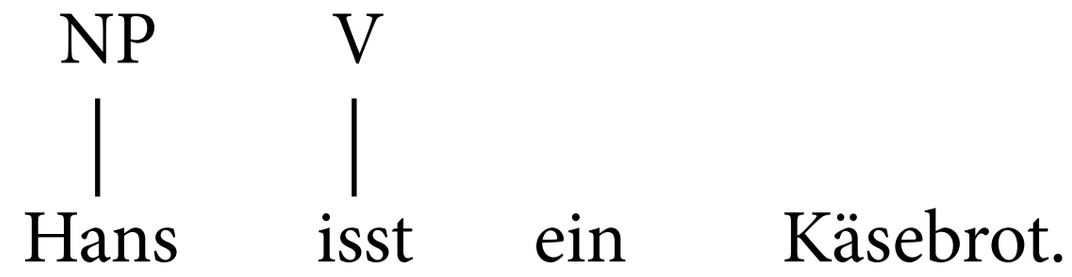
Shift: (s, a · w) → (s · a, w)
Reduce: (s · w', w) → (s · A, w)



(ε, Hans isst ein K.) → (Hans, isst ein K.) → (NP, isst ein K.)
→ (NP isst, ein K.)

Shift-Reduce: Beispiel

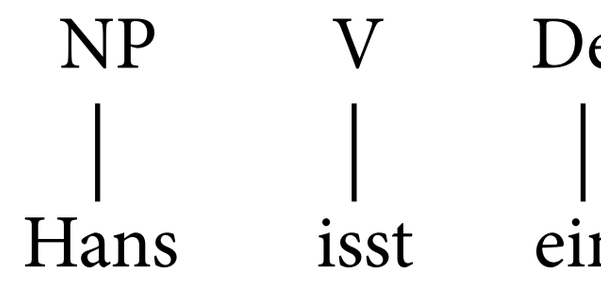
Shift: (s, a · w) → (s · a, w)
Reduce: (s · w', w) → (s · A, w)



(ε, Hans isst ein K.) → (Hans, isst ein K.) → (NP, isst ein K.)
→ (NP isst, ein K.) → (NP V, ein K.)

Shift-Reduce: Beispiel

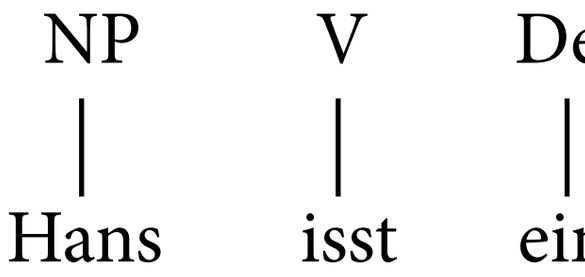
Shift: (s, a · w) → (s · a, w)
Reduce: (s · w', w) → (s · A, w)



(ε, Hans isst ein K.) → (Hans, isst ein K.) → (NP, isst ein K.)
→ (NP isst, ein K.) → (NP V, ein K.) → (NP V ein, K.)

Shift-Reduce: Beispiel

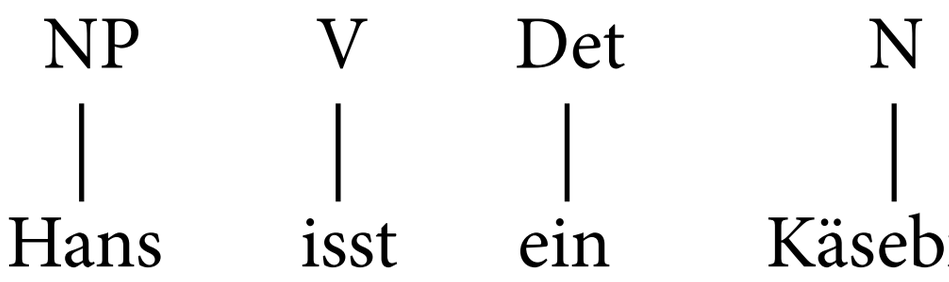
Shift: (s, a · w) → (s · a, w)
Reduce: (s · w', w) → (s · A, w)



(ε, Hans isst ein K.) → (Hans, isst ein K.) → (NP, isst ein K.)
→ (NP isst, ein K.) → (NP V, ein K.) → (NP V ein, K.)
→ (NP V Det, K.)

Shift-Reduce: Beispiel

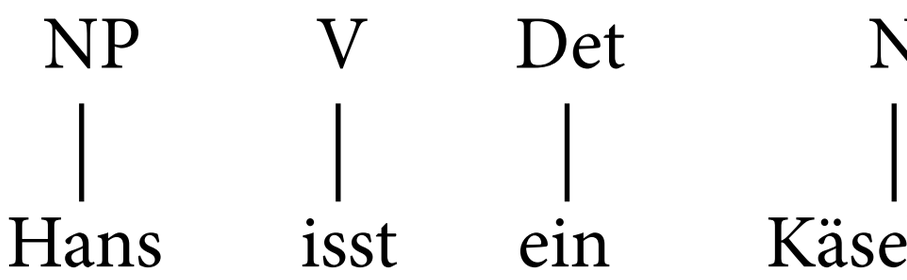
Shift: (s, a · w) → (s · a, w)
Reduce: (s · w', w) → (s · A, w)



(ε, Hans isst ein K.) → (Hans, isst ein K.) → (NP, isst ein K.)
→ (NP isst, ein K.) → (NP V, ein K.) → (NP V ein, K.)
→ (NP V Det, K.) → (NP V Det K., ε)

Shift-Reduce: Beispiel

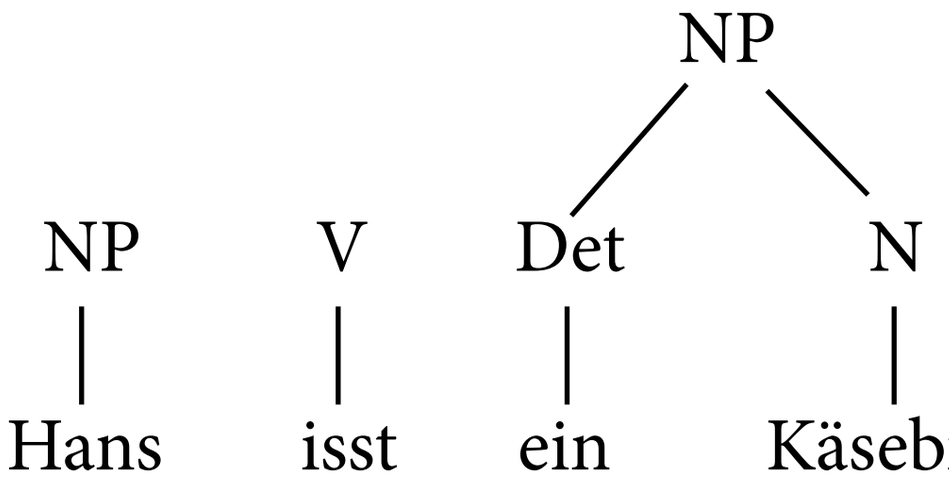
Shift: (s, a · w) → (s · a, w)
Reduce: (s · w', w) → (s · A, w)



(ε, Hans isst ein K.) → (Hans, isst ein K.) → (NP, isst ein K.)
→ (NP isst, ein K.) → (NP V, ein K.) → (NP V ein, K.)
→ (NP V Det, K.) → (NP V Det K., ε) → (NP V Det N, ε)

Shift-Reduce: Beispiel

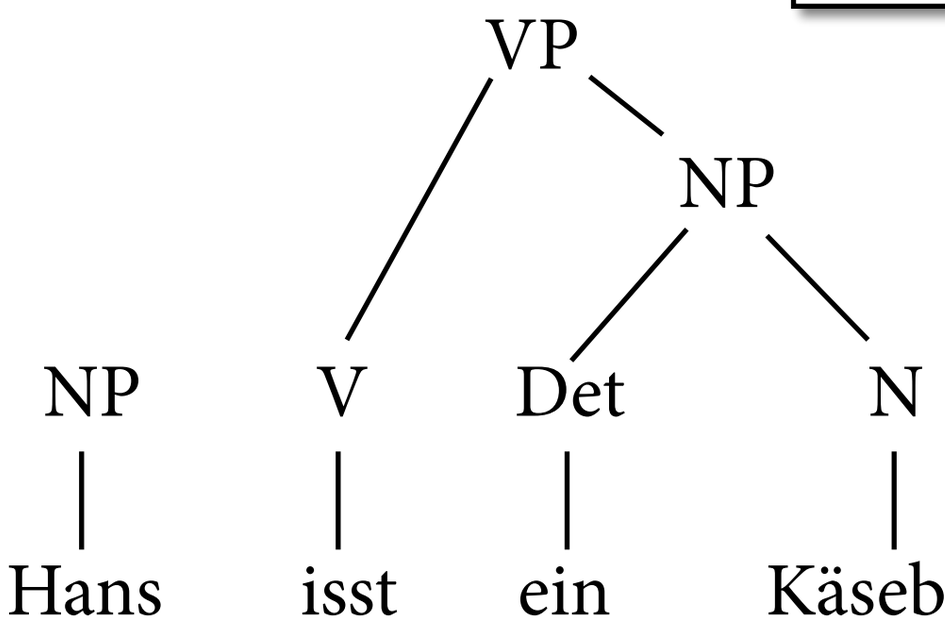
Shift: (s, a · w) → (s · a, w)
Reduce: (s · w', w) → (s · A, w)



(ε, Hans isst ein K.) → (Hans, isst ein K.) → (NP, isst ein K.)
→ (NP isst, ein K.) → (NP V, ein K.) → (NP V ein, K.)
→ (NP V Det, K.) → (NP V Det K., ε) → (NP V Det N, ε)
→ (NP V NP, ε)

Shift-Reduce: Beispiel

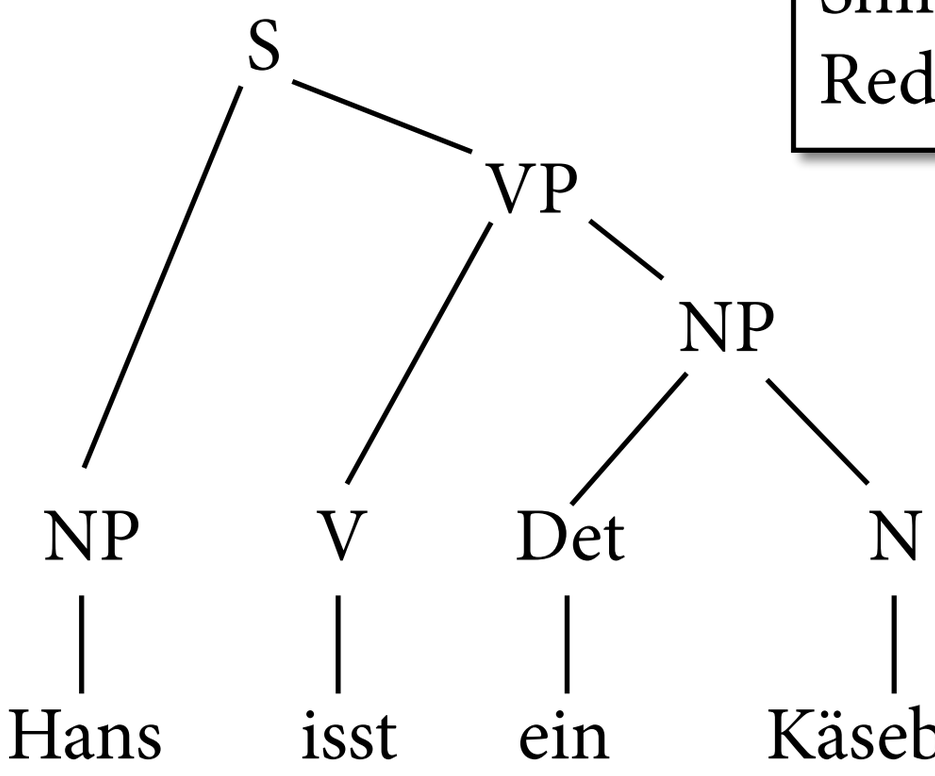
Shift: (s, a · w) → (s · a, w)
Reduce: (s · w', w) → (s · A, w)



(ε, Hans isst ein K.) → (Hans, isst ein K.) → (NP, isst ein K.)
→ (NP isst, ein K.) → (NP V, ein K.) → (NP V ein, K.)
→ (NP V Det, K.) → (NP V Det K., ε) → (NP V Det N, ε)
→ (NP V NP, ε) → (NP VP, ε)

Shift-Reduce: Beispiel

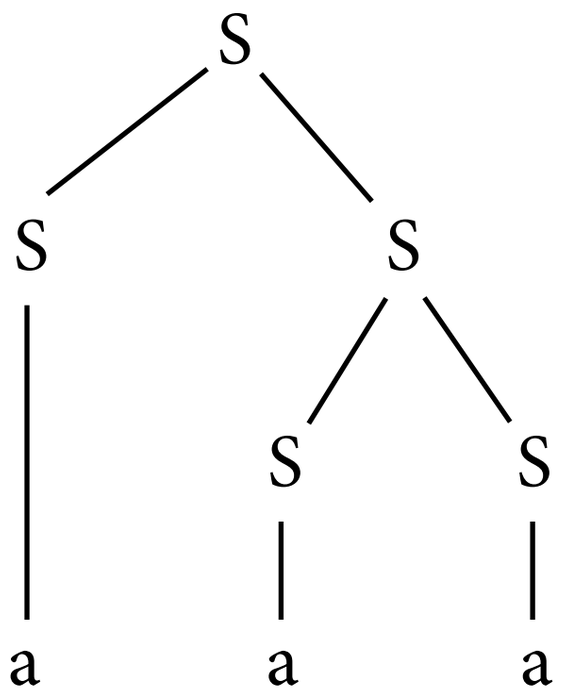
Shift: (s, a · w) → (s · a, w)
Reduce: (s · w', w) → (s · A, w)

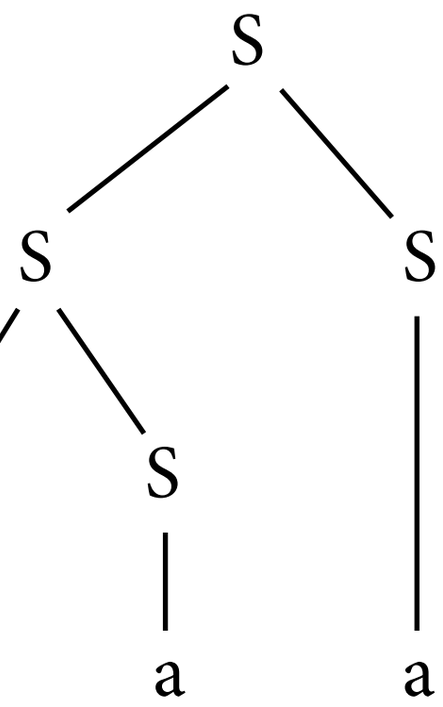


(ε, Hans isst ein K.) → (Hans, isst ein K.) → (NP, isst ein K.)
→ (NP isst, ein K.) → (NP V, ein K.) → (NP V ein, K.)
→ (NP V Det, K.) → (NP V Det K., ε) → (NP V Det N, ε)
→ (NP V NP, ε) → (NP VP, ε) → (S, ε)

Shift-Reduce: Beispiel

$S \rightarrow S\,S$ $\qquad$ $S \rightarrow a$





$(\varepsilon, aaa) \rightarrow (a, aa) \rightarrow (S, aa)$
$\rightarrow (Sa, a) \rightarrow (SS, a) \rightarrow (SSa, \varepsilon)$
$\rightarrow (SSS, \varepsilon) \rightarrow (SS, \varepsilon) \rightarrow (S, \varepsilon)$

$(\varepsilon, aaa) \rightarrow (a, aa) \rightarrow (S, aa)$
$\rightarrow (Sa, a) \rightarrow (SS, a) \rightarrow (S, a)$
$\rightarrow (Sa, \varepsilon) \rightarrow (SS, \varepsilon) \rightarrow (S, \varepsilon)$

Shift-Reduce in Prolog

restlicher Satz

Stack (Top = Links)

```
parse(Satz) :- step(Satz, []).

step([], [s]).
step(Satz, Stack) :-
   reduce(Satz, Stack, Satz1, Stack1), step(Satz1,Stack1).
step(Satz, Stack) :-
   shift(Satz, Stack, Satz1, Stack1), step(Satz1, Stack1).

shift([W|Rest], Stack, Rest, [W|Stack]).

reduce(Satz, [B,C|Rest], Satz, [A|Rest]) :- rule(A, C, B).
reduce(Satz, [B|Rest], Satz, [A|Rest]) :- lex(A, B).

rule(s, np, vp).    rule(vp, tv, np).

lex(np, hans).  lex(n, kaesebrot).  lex(det, ein).  lex(v, isst).
```

Vom Erkenner zum Parser

```
parse(Satz, T) :- step(Satz, [], T).

step([], [(s,T)], T).
step(Satz, Stack, T) :- reduce(Satz, Stack, Satz1, Stack1),
       step(Satz1,Stack1, T).
step(Satz, Stack, T) :- shift(Satz, Stack, Satz1, Stack1),
       step(Satz1, Stack1, T).

shift([W|Rest], Stack, Rest, [(W,W)|Stack]).

reduce(Satz, [(B,T1), (C,T2) | Rest], Satz, [(A,T)|Rest]) :-
       rule_with_trees(A, C, B, T2, T1, T).

reduce(Satz, [(B,T1)|Rest], Satz, [(A,T)|Rest]) :-
       lex_with_trees(A, B, T1, T).

rule_with_trees(A, B, C, T1, T2, T) :-
   rule(A, B, C), T =.. [A, T1, T2].

lex_with_trees(A, B, T1, T) :- lex(A,B), T =.. [A, T1].
```

Probleme: Übersicht

- Parsertypen haben komplementäre Probleme:

	top-down	bottom-up
Regeln raten	$A \rightarrow w$ vs. $A \rightarrow w$	$A \rightarrow w$ vs. $B \rightarrow w$
Zerlegung raten	String aufteilen	Shift vs. Reduce entscheiden

- Allgemein nicht zu vermeiden (Ambiguität).

Imperatives Programmieren

- Programm besteht aus Anweisungen, die der Reihe nach abgearbeitet werden.

- Rekursion ist möglich, wird aber häufig durch Schleifen ersetzt.

- Variablen werden nicht gebunden, sondern bekommen Werte zugewiesen.

- Historisch ältestes Programmierparadigma.

Python

- Beliebte Programmiersprache, u.a. weil sie einfach zu lernen ist.
  - erste Version 1991 von Guido van Rossum
  - aktuell: Version 3.4; wir verwenden Python 2.7

- Python erlaubt imperatives Programmieren; unterstützt auch funktional und objektorientiert.

- Viele Bibliotheken, z.B. NLTK.

Woher kriege ich Python?

- Möglichkeiten:
  - https://www.enthought.com/products/canopy/academic/
  - http://www.python.org/getit/
  - Linux: über eingebauten Paket-Manager
  - mitgeliefertes Python 2.6 von MacOS hat Bugs

- Verwenden Sie dazu eine IDE, z.B. Aptana Studio.

- Siehe Links im Piazza.

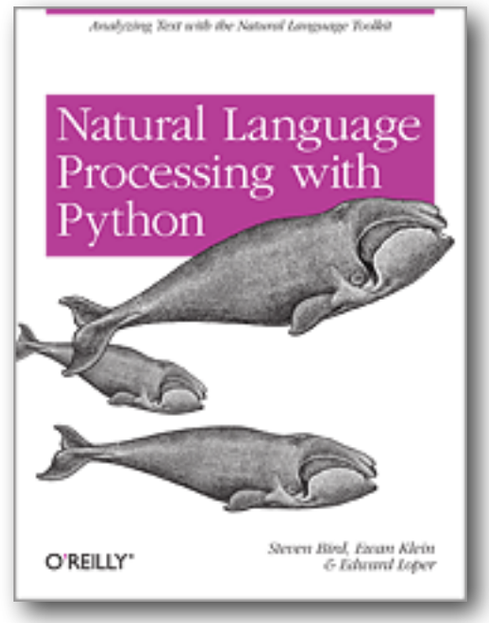
Literatur

- Original-Dokumentation incl. Tutorial: http://docs.python.org/2.7/

- Diverse Bücher, u.a. kostenloses “A Byte of Python”: http://swaroopch.com/notes/Python

- Online-Kurse, z.B. bei Codecademy: http://www.codecademy.com/en/tracks/python

- NLTK-Buch: http://www.nltk.org/book



Python-Programme

- Interaktive Python-Shell:

```
koller$ python
>>>
```

- Python-Programm aus einer Datei ablaufen lassen:

```
koller$ python programm.py
```

- Innerhalb einer Entwicklungsumgebung (IDE), z.B. Aptana Studio.

Werte

- Werte sind Elemente von bestimmten Wertebereichen.

- Jeder Wert hat einen Datentyp:
  - ganze Zahlen
  - Fließkommazahlen
  - Wahrheitswerte
  - Strings
  - Listen
  - usw.

Ausdrücke

- Ausdrücke (= Terme) denotieren Werte.
  Jeder Ausdruck hat einen Datentyp.

- Literale:

  `5` (int), `-2.3` (float), “`hallo`” (String), `True` (Boolean) ...

- Komplexe Ausdrücke werden durch Operatoren gebaut:

  `(2+3)*4`, `True and not False`, “`hallo`”`[3]`

Wichtige Operatoren

- Die üblichen arithmetischen Operationen +, -, *, / und Vergleiche ==, !=, <, <= usw.

- Für Ganzzahlen (int) ist “/” ganzzahlige Division. Rest mit “%” (modulo-Operator).

- Für Strings: “a”+”b”, “a”*3; außerdem Teilstrings mit `s[2:5]`, `s[:5]`, `s[5:]`

- Für Boolean: and, or, not

Datentypen konvertieren

- Man kann zwischen Datentypen konvertieren.

- int(x) gibt einen Ganzzahlwert zurück:

```
int(2.3) == 2
int(-2.3) == -2
int(“2”) == 2
```

- str(x) gibt String-Repräsentation zurück:

`str(2) = “2”, str(True) = “True”` usw.

Variablen

- Auch Variablen haben Werte. Der Wert kann sich aber im Lauf des Programms durch eine Zuweisung ändern.

  Verwendung in Ausdrücken: `x`, `(y+z)*3`, `str[i]`

  Zuweisung: `x = 27`

- Variablennamen: Beginnen mit Buchstabe oder _, keine Umlaute, keine Schlüsselwörter.

Variablen

- Der Wert einer Variablen hat einen Datentyp.
- Die Variable an sich hat keinen: Man darf ihr nacheinander Werte von verschiedenen Typen zuweisen.
  `x = 27`; dann `x = “hallo”`
- Variable existiert erst ab der ersten Zuweisung; vorher Fehler bei Verwendung.

Listen

- Sequenz von beliebigen Werten.
- Literale: `[]`, `[1,2,3]`, `["hallo", True, -5.0]`
- Hinten an Liste anhängen: `L.append(27)`
- Länge ist `len(L)`

Slices

- Zugriff auf Teile:

  `L[2]`: drittes Element der Liste (0 ist erstes)

  ```
  L[2:5] = [L[2],...,L[4]]
  L[2:] = L[2:len(L)]
  L[:2] = L[0:2]
  ```

- Man kann Elemente und Slices überschreiben (auch verkürzen und verlängern):

  ```
  L = [1,2,3]   L[1] = 7 => L == [1,7,3]
  L = [1,2,3]   L[0:1] = [5] => L == [5,2,3]
  L = [1,2,3]   L[2:] = [] => L == [1,2]
  ```

Mengen

- Wie mathematische Menge: ungeordnet, jedes Element kommt nur einmal vor

- Erzeugen mit `set()` oder `set([2,3,4])`

- Erweitern mit `s.add("hallo")`

- Mitgliedschaft (effizient) testen mit `2 in s`

Dictionaries

- Dictionary (Wörterbuch) besteht aus einer Menge von Schlüsseln, denen Werte zugewiesen werden.

- Literale: `{“hallo”:3, “welt”:False}`, `{}`

- Hinzufügen mit `dict[“foo”] = 5;`
  Auslesen mit `dict[“foo”];`
  Test, ob Schlüssel existiert, mit `“foo” in dict;`
  Schlüsselmenge mit `dict.keys()`

Anweisungen

- Mit einer Anweisung bringt man Computer dazu, eine bestimmte Aktion auszuführen.
  - Zuweisung ist Anweisung: Wert einer Variable ändern
  - `print(x)`: Zeige Wert des Ausdrucks x an
  - `sys.exit()`: Programm beenden
- Anweisungen sind das zentrale Werkzeug beim imperativen Programmieren.

Kontrollstrukturen

- Ein Programm ist eine Sequenz von Anweisungen.
- In den meisten Programmen sollen bestimmte Anweisungen nur manchmal bzw. mehrmals ausgeführt werden.
- Das erreicht man mit *Kontrollstrukturen*.

If

- Mit der if-Kontrollstruktur wird ein Programmblock nur ausgeführt, wenn ein Ausdruck (die Bedingung) zu `True` evaluiert:

```
if len(L) > 0:
    print L[0]
```

- Wenn die Bedingung zu `False` evaluiert, wird der ganze if-Block übersprungen.

Einrücken

- In Python ist es wichtig, dass Anweisungen innerhalb einer Kontrollstruktur eingerückt werden.

- Beachte die Leerzeichen vor `print L[0]`!

- Wenn Bedingung zu `False` evaluiert, überspringt if genau die eingerückten Zeilen.

If-Else

- Man kann einen else-Block angeben, der ausgeführt wird, wenn die Bedingung zu `False` evaluiert:

```
if len(L) > 0:
    print L[0]
else:
print “Liste ist leer”
```

While

- Manchmal möchte man eine Anweisung mehrfach ausführen, z.B. so lange, bis eine Bedingung `False` wird:

```
while len(L) > 0:
    print L[0]
    L = L[1:]
```

(Dieses Beispiel ist eine schrecklich ineffiziente Art, um eine Liste auszugeben!)

For

- Alternativ kann man auch einen Block von Anweisungen für jedes Element einer Liste oder Menge ausführen:

```
for x in L:
    print x
```

- x kann wie eine Variable verwendet werden.
  Es macht aber normalerweise keinen Sinn, x etwas zuzuweisen.

For mit Zahlen

- Eine häufige Verwendung von for ist es, Anweisungen für alle Zahlen von 0 bis n-1 auszuführen.

- Dazu verwendet man `range(n)`: gibt Liste [0,...,n-1] zurück.

```
sum = 0
for i in range(n):
    sum = sum + i
```

Schleifen

- Anweisungen, die mit den Konstrukten for und while aufgebaut werden, heißen *Schleifen*.

- In imperativen Sprachen kann Rekursion durch Iteration (= Verwendung von Schleifen) ersetzt werden.

- Wahl zwischen Rekursion und Iteration hängt von Lesbarkeit und Effizienz ab.

Funktionen

- Blöcke von Anweisungen, die immer wieder verwendet werden, kann man in *Funktionen* zusammenfassen:

```
def printxpluszwei(zahl):
    output = zahl+2
    print(output)

>>> printxpluszwei(5)
7
```

Funktionen

- Eine Funktion kann 0, 1 oder mehrere Parameter haben:
  - `def keineparameter(): ...`
  - `def einparameter(x): ...`
  - `def zweiparameter(x, y): ...`

- Innerhalb der Funktion kann ein Parameter wie eine Variable verwendet werden.

- Wert des Parameters: durch Aufruf bestimmt.

Funktionen

- Funktionen können mit einer `return`-Anweisung Werte zurückgeben. Dieser Wert wird dann der Wert des Funktions-Ausdrucks.

```
def plus(x,y):
    return x + y
```

```
>>> print plus(3,4)
7
```

Rekursion

- Auch in Python dürfen Funktionen rekursiv sein, d.h. sich selbst aufrufen.

```
def ggT(klein, gross):
    if klein == 0:
        return gross
    else:
        rest = gross % klein
        return ggT(rest, klein)
```

- Aber Rekursion viel seltener als in Prolog: häufig stattdessen Schleifen.

Zusammenfassung

- Schnelltour durch Python:
  - Werte und Ausdrücke
  - Variablen
  - Anweisungen
  - Listen, Mengen, Dictionaries
  - Kontrollstrukturen und Schleifen
  - Funktionen
- Morgen: Fortgeschrittenes Python