Skriptsprachen
* Ruby *

Methoden Revisited
Ein- & Ausgabe

Naumann
Sommersemester 201

7.1 Methoden Revisited

✦ Methodennamen beginnen mit einem Kleinbuchstaben oder Unterstrich, auf den weitere Zeichen, Zahlen oder Unterstriche folgen können.

✦ Ein Methodenname kann außerdem mit folgenden Zeichen enden:

?	-	Methoden, die einen booleschen Wert liefern
!	-	*destruktive* Methoden
=	-	Methoden, die auf der rechten Seite des Zuweisungsoperators auftreten können

✦ Parameter:

✦ eine Liste von lokalen Variablen, die optional geklammert werden;

✦ Standardwerte können durch Verwendung von ‚= *Wert*'-Angaben spezifiziert werden:

```
def cool_dude(arg1="Miles", arg2="Coltrane", arg3="Roach")
    "#{arg1}, #{arg2}, #{arg3}."
end
cool_dude # => "Miles, Coltrane, Roach."
cool_dude("Bart") # => "Bart, Coltrane, Roach."
cool_dude("Bart", "Elwood") # => "Bart, Elwood, Roach."
cool_dude("Bart", "Elwood", "Linus") # => "Bart, Elwood, Linus."
```

7.1 Methoden Revisited

Dabei ist es auch möglich, auf die Werte vorangegangener Parameter zu beziehen:

```
def surround(word, pad_width=word.length/2)
    "[" * pad_width + word + "]" * pad_width
end

surround("elephant") # => "[[[[elephant]]]]"
surround("fox") # => "[fox]"
surround("fox", 10) # => "[[[[[[[[[[fox]]]]]]]]]]"
```

Es ist möglich Methoden zu definieren, die eine variable Zahl von Argumenten akzeptieren: zu diesem Zweck kann ein Parameter ‚*‘ als Präfix erhalten und alle geeigneten Argumente werden in einem Array gesammelt und ihm dann zugewiesen:

```
def varargs(arg1, *rest)
    "arg1=#{arg1}. rest=#{rest.inspect}"
end
varargs("one") # => arg1=one. rest=[]
varargs("one", "two") # => arg1=one. rest=[two]
varargs "one", "two", "three" # => arg1=one. rest=[two, three]
```

7.1 Methoden Revisited

Manchmal wird dieser Mechanismus auch verwendet, um Argumente zu speichern, die nicht von der Methode selbst, sondern von der gleichnamigen Methode in der Superklasse der Klasse, in der die Methode definiert wurde, genutzt werden.

```
class Child < Parent
    def do_something(*not_used)
        # alternativ einfach: * sofern Parameter hier nicht genutzt werden
        # ...           Aktionen der Methode
        super     # Aufruf von do_something in der Superklasse
    end
end
```

Ab Ruby 1.9 kann ein beliebiger Parameter als *splat*-Argument verwendet werden:

```
def split_apart(first, *splat, last)
    puts "First: #{first.inspect}, splat: #{splat.inspect}, " +
"last: #{last.inspect}"
end
split_apart(1,2)          # => First: 1, splat: [], last: 2
split_apart(1,2,3)        # => First: 1, splat: [2], last: 3
split_apart(1,2,3,4)      # => First: 1, splat: [2, 3], last:
```

7.1 Methoden Revisited

Wie wir wissen, kann eine Methode wenn sie aufgerufen wird mit einem Block assoziiert werden, der aus der Methode heraus mit yield aufgerufen wird:

```
def double(p1)
    yield(p1*2)
end
double(3) {|val| "I got #{val}" }                    # => "I got 6"
double("tom") {|val| "Then I got #{val}" }           # => "Then I got tomtom"
```

Wenn der letzte Parameter der Methode einen ‚&'-Präfix hat, wird der Block in in **Proc**-Objekt konvertiert, das zu späterem Gebrauch gespeichert werden kann:

```
class TaxCalculator
    def initialize(name, &block)
        @name, @block = name, block
    end
    def get_tax(amount)
        "#@name on #{amount} = #{ @block.call(amount) }"
    end
end
tc = TaxCalculator.new("Sales tax") {|amt| amt * 0.075 }
tc.get_tax(100) # => "Sales tax on 100 = 7.5"
tc.get_tax(250) # => "Sales tax on 250 = 18.75"
```

7.1 Methoden Revisited

Beim Aufruf einer Methode kann optional ein Empfänger spezifiziert werden. Es folgt der Name der Methode und die Parameter, sowie wieder optional ein Block:

objekt.methode(parameter$_1$, ..., parameter$_n$) block

Für Klassen- und Modulmethoden ist der Empfänger immer die Klasse bzw. das Modul:

```
File.size("testfile") # => 66
Math.sin(Math::PI/4) # => 0.707106781186547
```

Verzichtet man auf die explizite Angabe eines Empfängers wird immer das aktuelle Objekt (self) als Empfänger verwendet.

Sofern es nicht zu Mehrdeutigkeiten führt, ist es möglich, beim Aufruf einer Methode die Klammern um die Argumente wegzulassen:

```
a = obj.hash            # identisch mit:
a = obj.hash()          # Klammern überflüssig
obj.some_method "Arg1", arg2, arg3          # Identisch mit der folgenden Zeile:
obj.some_method("Arg1", arg2, arg3)         # Klammern überflüssig
```

7.1 Methoden Revisited

Jede Methode gibt einen Wert zurück und zwar den Wert, den die zuletzt ausgeführte Anweisung produziert:

```
def meth_one
    "one"
end
meth_one                # => "one"
def meth_two(arg)
    case
        when arg > 0 then "positive"
        when arg < 0 then "negative"
        else "zero"
    end
end
meth_two(23)            # => "positive"
meth_two(0)             # => "zero"
```

Durch eine return-Anweisung kann die Methode explizit verlassen werden. Üblicherweise wird auf sie - wo möglich - in Ruby verzichtet. Sinnvoll lässt sie sich zum (vorzeitigen) Verlassen einer Schleife verwenden:

```
def meth_three
100.times do |num|
    square = num*num
    return num, square if square > 1000
end
end
meth_three # => [32, 1024]
```

7.1 Methoden Revisited

Proc-Objekte ermöglichen es, kompakteren und übersichtlicheren Kode zu schreiben:

```
print "(t)imes or (p)lus: "
operator = gets
print "number: "
number = Integer(gets)

if operator =~ /^t/
  puts((1..10).collect {|n| n*number }.join(", "))
else
  puts((1..10).collect {|n| n+number }.join(", "))
end
```

```
print "(t)imes or (p)lus: "
operator = gets
print "number: "
number = Integer(gets)

if operator =~ /^t/
  calc = lambda {|n| n*number }
else
  calc = lambda {|n| n+number }
end
puts((1..10).collect(&calc).join(", "))
```

```
(t)imes or (p)lus: t
number: 2
2, 4, 6, 8, 10, 12, 14, 16, 18, 20
```

7.1 Methoden Revisited

Einige Programmiersprachen (wie z.B. Common LISP) kennen *Schlüsselwort*-Parameter: Statt die Argumente in einer vordefinierten Reihenfolge zu übergeben wird durch die Verwendung der Parameternamen-Wert-Paaren ein größeres Maß an Flexibilität erreicht. Ruby 1.9 kennt diesen Mechanismus nicht, kann aber ähnliche Effekte durch die Verwendung von *Hashes* erzielen:

```
class SongList
    def search(name, params)
        # ...
    end
end
list.search(:titles,
            { :genre => "jazz",
              :duration_less_than => 270
            })
```

Wenn das Hash-Argument das letzte Argument ist, können die Klammern weggelassen werden:

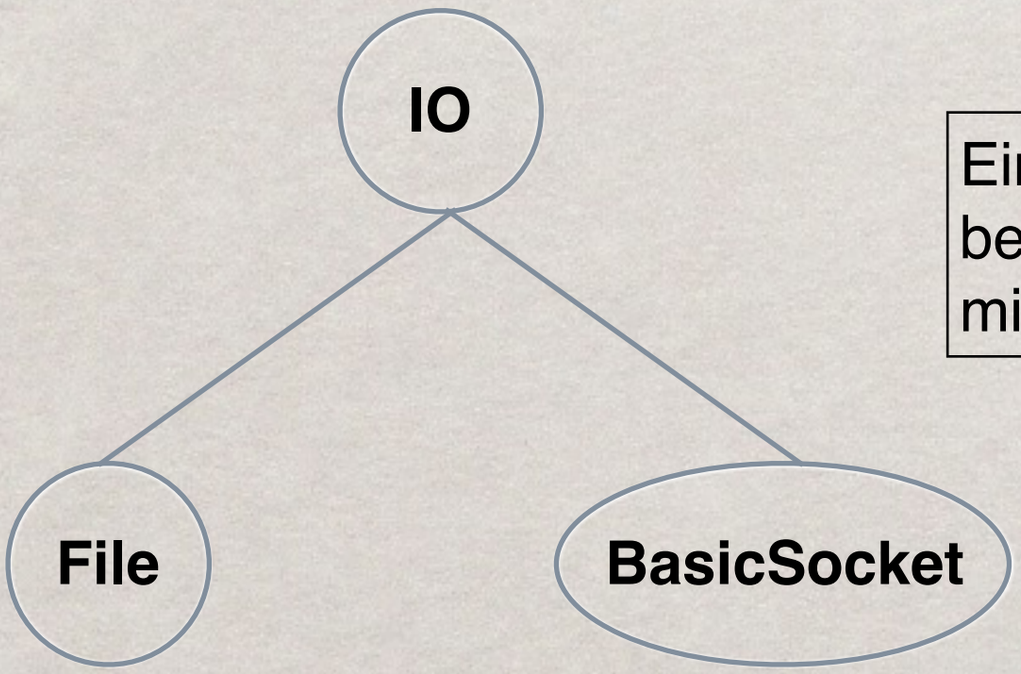
```
list.search(  :titles,
              :genre => 'jazz',
              :duration_less_than => 270)

list.search(:titles, genre: 'jazz', duration_less_than: 270)
```

7.2 Ein- und Ausgabe

- Bislang haben wir uns - wenn es um die Ausgabe oder das Einlesen von Werten ging - in der Regel auf die Methoden puts, gets, print, p, etc. beschränkt, die in Rubys Kernmodul definiert werden.

- Diese Methoden lesen (schreiben) Daten aus der (in die) Standardeingabe (-ausgabe).

- Für komplexere Ein-Ausgabeoperationen bieten sich in Ruby **IO-Objekte** an, die eine Reihe nützlicher Optionen bieten.



Ein **IO-Objekt** kann als bidirektionaler Kanal betrachtet werden, der ein Ruby-Programm mit einer externen Ressource verbindet.

7.2 Ein- und Ausgabe

Datei- und Verzeichnisnamen

✦ Die Klasse **File** stellt eine ganze Reihe von Methoden zur Verfügung, die es ermöglichen, Dateien als Einträge im Dateisystem zu behandeln (Bestimmung der Größe, der Existenz, der Art einer Datei, etc.).

✦ Die meisten dieser Methoden sind Klassenmethoden; d.h. sie operieren nicht auf Instanzen der Klasse **File**, sondern erwarten ein- oder mehrere Dateinamen (*Strings*) als Argumente.

✦ Ab Ruby 1.9 ist es allerdings auch möglich, Pfadnamen (Objekte, die durch die to_path Methode erzeugt werden) als Argumente dieser Methoden zu verwenden.

```
full = '/home/matz/bin/ruby.exe'
file=File.basename(full)          # => 'ruby.exe': just the local filename
File.basename(full, '.exe')       # => 'ruby': with extension stripped
dir=File.dirname(full)            # => '/home/matz/bin': no „/“ at end
File.dirname(file)                # => '.': current directory
File.split(full)                  # => ['/home/matz/bin', 'ruby.exe']
File.extname(full)                # => '.exe'
File.extname(file)                # => '.exe'
File.extname(dir)                 # => ''
File.join('home','matz')          # => 'home/matz': relative
File.join('','home','matz')       # => '/home/matz': absolute
```

7.2 Ein- und Ausgabe

Datei- und Verzeichnisnamen

File.expand_path konvertiert einen relativen in einen absoluten Pfad;
mit '~' wird auf das Heimatverzeichnis des aktuellen Benutzers Bezug genommen.

```
Dir.chdir("/usr/bin")                      # Current working directory is "/usr/bin"
File.expand_path("ruby")                   # => "/usr/bin/ruby"
File.expand_path("~/ruby")                 # => "/home/david/ruby"
File.expand_path("~matz/ruby")             # => "/home/matz/ruby"
File.expand_path("ruby", "/usr/local/bin") # => "/usr/local/bin/ruby"
File.expand_path("ruby", "../local/bin")   # => "/usr/local/bin/ruby"
File.expand_path("ruby", "~/bin")          # => "/home/david/bin/ruby"
```

File.identical? überprüft, ob sich zwei Dateinamen/Pfadnamen auf dieselbe Datei beziehen:

```
File.identical?("ruby", "ruby")            # => true if the file exists
File.identical?("ruby", "/usr/bin/ruby") # => true if CWD is /usr/bin
File.identical?("ruby", "../bin/ruby")     # => true if CWD is /usr/bin
File.identical?("ruby", "ruby1.9")         # => true if there is a link
```

7.2 Ein- und Ausgabe

Anzeigen von Verzeichnissen

Die einfachsten Möglichkeiten sich den Inhalt eines Verzeichnisses anzeigen zu lassen, besteht in der Verwendung der Iteratoren Dir.entries bzw. Dir.foreach.

```
# Erhalte die Namen aller Dateien im Verzeichnis config/
filenames = Dir.entries("config")          # liefert ein Array, das die Dateinamen enthält
Dir.foreach("config") {|filename| ... }    # Iteriert über die Namen der Dateien
```

Um die Dateien, deren Namen einem bestimmten Muster genügen, zu erhalten (als *Array*), kann man den Dir.[]-Operator verwenden:

```
Dir['*.data']          # Dateien mit der "data"-Extension
Dir['ruby.*']          # Jede Datei, deren Name mit "ruby" beginnt.
Dir['?']               # Jede Datei mit einem Namem der Länge 1.
Dir['*.[ch]']          # Jede Datei, deren Name mit .c oder .h endet.
Dir['*.{java,rb}']     # Jede Datei , deren Name mit .java oder .rb endet.
Dir['*/*.rb']          # Jedes Ruby-Programm in einem direkten
Subverzeichnis.
Dir['**/*.rb']         # Jedes Ruby-Programm in einem beliebigen
Subverzeichnis.
```

7.2 Ein- und Ausgabe

Überprüfen von Dateien

```
f = "/usr/bin/ruby"       # Ein Dateiname für die folgenden Beispiele

# Datei: Existenz und Typen.
File.exist?(f)            # Existiert eine Datei mit diesem Namen? (File.exists?)
File.file?(f)             # Ist es eine ‚echte' Datei?
File.directory?(f)        # Oder handelt es sich um ein Verzeichnis?
File.symlink?(f)          # Oder ist es ein symboliscjer Verweis?
```

```
# Dateigröße. Benutze File.truncate um die Dateigröße festzusetzen.
File.size(f)              # Dateigröße in Bytes.
File.size?(f)             # Größe in Bytes bzw. nil wenn die Datei leer ist.
File.zero?(f)             # True gdw. die Datei leer ist.
```

```
# Berechtigungen. Verwende File.chmod um Berechtigungen zu ändern (systemabhängig!).
File.readable?(f)         # Darf die Datei gelesen werden?
File.writable?(f)         # Darf die Datei geschrieben werden? Kein "e" in "writable".
File.executable?(f)       # Darf die Datei ausgeführt werden?
File.world_readable?(f)   # Darf jeder die Datei lesen? Ruby 1.9.
File.world_writable?(f)   # Darf jeder die Datei ausführen? Ruby 1.9.
```

7.2 Ein- und Ausgabe

Überprüfen von Dateien

```
# File times/dates. Use File.utime to set the times.
File.mtime(f)          # => Last modification time as a Time object
File.atime(f)          # => Last access time as a Time object
```

Um alle relevanten Informationen über eine Datei mit einem Aufruf zu erhalten, kann man die Methoden File.stat bzw. File.lstat verwenden. Beide Methoden liefern als Wert ein File::Stat-Objekt zurück, dass alle Metadaten über die Datei enthält:

```
s = File.stat("/usr/bin/ruby")
s.file?                # => true
s.directory?           # => false
s.ftype                # => "file"
s.readable?            # => true
s.writable?            # => false
s.executable?          # => true
s.size                 # => 5492
s.atime                # => Mon Jul 23 13:20:37 -0700 2007
```

7.2 Ein- und Ausgabe

Erzeugen, Löschen und Umbenennen von Dateien und Verzeichnissen

```
Dir.mkdir("temp")              # Erzeuge a Verzeichnis
File.open("temp/f", "w") {}    # Erzeuge eine Datei in dem Verzeichnis
File.open("temp/g", "w") {}    # Erzeuge eine weitere Datei
File.delete(*Dir["temp/*"])    # Lösche alle Dateien in dem Verzeichnis
Dir.rmdir("temp")              # Lösche das Verzeichnis
```

7.2 Ein- und Ausgabe

Öffnen und Schließen von Dateien

Neue Dateien bzw. Dateiobjekte können durch File.new erzeugt werden:

```
file = File.new("testfile", "r")
# ... Verarbeitung der Datei
file.close
```

Als erster Parameter wird der Dateiname, als zweiter die Art des Zugriffs angegeben:

Mode	Meaning
r	Read-only, starts at beginning of file (default mode).
r+	Read/write, starts at beginning of file.
w	Write-only, truncates an existing file to zero length or creates a new file for writing.
w+	Read/write, truncates existing file to zero length or creates a new file for reading and writing.
a	Write-only, starts at end of file if file exists; otherwise, creates a new file for writing.
a+	Read/write, starts at end of file if file exists; otherwise, creates a new file for reading and writing.
b	Binary file mode (may appear with any of the key letters listed earlier). As of Ruby 1.9, this modifier should be supplied on all ports opened in binary mode (on Unix as well as on DOS/Windows). To read a file in binary mode and receive the data as a stream of bytes, use the modestring "rb:ascii-8bit".

7.2 Ein- und Ausgabe

Öffnen und Schließen von Dateien

Statt die Datei auf diese Weise zu öffnen und am Ende mit close explizit zu schließen, kann man die Klassenmethode File.open verwenden, die sich wie zunächst File.new verhält, mit einem Block verbunden aber dafür sorgt, dass nach Beendigung des Blocks die Datei automatisch geschlossen wird:

```
File.open("testfile", "r") do |file|
    # ... Verarbeitung der Datei
end       # Die Datei wird automatisch geschlossen
```

Der entscheidende Vorteil dieses Vorgehens liegt darin, dass auch beim Auftreten von Fehlern die Datei in jedem Fall geschlossen wird.

```
class File
    def File.open(*args)
        result = f = File.new(*args)
        if block_given?
            begin
                result = yield f
            ensure
                f.close
            end
        end
        return result
    end
end
```

7.2 Ein- und Ausgabe

Lesen und Schreiben von Daten

Dieselben einfachen IO-Methoden, die wir bislang verwendet haben, lassen sich auch benutzen, um Daten aus Dateien zu lesen bzw. in Dateien zu speichern.

```
while line = gets
    puts line
end
```

copy.rb

```
% ruby copy.rb
These are lines
These are lines
that I am typing
that I am typing
^D
```

```
% ruby copy.rb testdatei
...     # 1. Zeile aus testdatei
...     # 2. Zeile aus testdatei
...     # 3. Zeile aus testdatei
...
```

```
File.open("testfile") do |file|
    while line = file.gets
        puts line
    end
end
```

7.2 Ein- und Ausgabe

Iteratoren zum Lesen von Daten

each_byte

```
File.open("testfile") do |file|
    file.each_byte {|ch| print "#{ch.chr}:#{ch} " }
end

T:84 h:104 i:105 s:115 :32 i:105 s:115 :32 l:108 i:105 ...
T:84 h:104 i:105 s:115 :32 i:105 s:115 :32 l:108 i:105 ...
A:65 n:110 d:100 :32 s:115 o:111 :32 o:111 n:110 .:46 ...
```

each_line

```
File.open("testfile") do |file|
    file.each_line {|line| puts "Got #{line.dump}" }
end

Got "This is line one\n"
Got "This is line two\n"
Got "This is line three\n"
Got "And so on...\n"
```

7.2 Ein- und Ausgabe

Iteratoren zum Lesen von Daten

Die Methode each_line kann beliebige Zeichen als Zeilenendmarker interpretieren (Standardwert: \n):

each_line

```
File.open("testfile") do |file|
    file.each_line("e") {|line| puts "Got #{ line.dump }" }
end

Got "This is line"
Got " one"
Got "\nThis is line"
Got " two\nThis is line"
Got " thre"
Got "e"
Got "\nAnd so on...\n"
```

7.2 Ein- und Ausgabe

Iteratoren zum Lesen von Daten

foreach

```
IO.foreach("testfile") {|line| puts line }

This is line one
This is line two
This is line three
And so on...
```

Iterator, der am Ende automatisch die Datei schließt

read

readlines

```
# Eingabe in einem String speichern
str = IO.read("testfile")
str.length # => 66
str[0, 30] # => "This is line one\nThis is line "

# Eingabe in einem Array speichern
arr = IO.readlines("testfile")
arr.length # => 4
arr[0] # => "This is line one\n"
```

7.2 Ein- und Ausgabe

Schreiben von Daten

```
# Mit "w" wird die Datei zum Schreiben geöffnet
File.open("output.txt", "w") do |file|
    file.puts "Hello"
    file.puts "1 + 2 = #{1+2}"
end
# Lesen der Datei und Ausgabe des Inhalts auf STDOUT
puts File.read("output.txt")

Hello
1 + 2 = 3
```

Nahezu jedes Objekt, das als Argument an puts und print übergeben wird, kann in einen String konvertiert werden. Ist das nicht möglich, wird ein String erzeugt, der den Name seiner Klasse und seine ID enthält.

```
str1 = "\001\002\003" # => "\x01\x02\x03"
str2 = ""
str2 << 1 << 2 << 3 # => "\x01\x02\x03"
[ 1, 2, 3 ].pack("c*") # => "\x01\x02\x03"
```

Es gibt verschiedene Wege, um binäre Daten in einen String einzufügen:
a) Literale
b) Byte für Byte einfügen
c) Array#pack

7.2 Ein- und Ausgabe

Schreiben von Daten

```
endl = "\n"
STDOUT << 99 << " red balloons" << endl

99 red balloons
```

Mit dem <<-Operator kann man ein beliebiges Objekt am Ende eines Array einfügen oder in einen Ausgabestream schieben.

Die Methoden der stringio-Bibliothek ermöglichen es, Strings wie Streams zu behandeln:

```
require 'stringio'
ip = StringIO.new("now is\nthe time\nto learn\nRuby!")
op = StringIO.new("", "w")
ip.each_line do |line|
    op.puts line.reverse
end
op.string # => "\nsi won\n\nemit eht\n\nnrael ot\n!ybuR\n"
```

7.3 Ein einfacher Lemmatisator

Das **Lemma** (von griechisch λῆμμα *lēmma* ‚Annahme'; von λαμβάνειν *lambánein* ‚nehmen') ist in der Lexikografie und Linguistik die Grundform eines Wortes, also die Wortform, unter der man es in einem Nachschlagewerk sucht (*Zitierform*, *Grundform*). Der Vorgang zur Bestimmung der genaueren Lemmata wird als *Lemmaselektion* oder auch *Lemmatisierung* bezeichnet.

Wikipedia

Prozess

flektierte Form → Grundform

kann realisiert werden mit Hílfe von
- Lexika
- morphologische Heuristiken & Analyseverfahren

verwandte Begriffe
- *Stemming*

7.3 Ein einfacher Lemmatisator

CELEX

CELEX is the Dutch Centre for Lexical Information. It was developed as a joint enterprise of the University of Nijmegen, the Institute for Dutch Lexicology in Leiden, the Max Planck Institute for Psycholinguistics in Nijmegen, and the Institute for Perception Research in Eindhoven. Over the years it has been funded mainly by the Netherlands Organisation for Scientific Research (NWO) and the Dutch Ministry of Science and Education. CELEX is now part of the Max Planck Institute for Psycholinguistics.

Englisch - Niederländisch - Deutsch

Informationen auf verschiedene Dateien verteilt
(Orthographie, Phonologie, Morphologie, Syntax für Lemmata/Wortformen, Frequenzdaten, ...)

7.3 Ein einfacher Lemmatisator

gol.cd

```
1\A\563\A\N\A\A\N
2\Ä\4\Ä\N\Ä\Ä\N
3\aalen\1\aa-len\N\aal\aal\N
4\Aal\80\Aal\N\Aal\Aal\N
5\aalglatt\0\aal-glatt\N\aalglatt\aal-glatt\N
6\Aar\2\Aar\N\Aar\Aar\N
7\Aas\6\Aas\N\Aas\Aas\N
8\Aasgeier\2\Aas-gei-er\N\Aasgeier\Aas-gei-er\N
9\ab\2171\ab\N\ab\ab\N
10\ab\2171\ab\N\ab\ab\N
11\Abakus\0\A=ba-kus\N\Abakus\A=ba-kus\N
12\abändern\16\ab-än-dern\N\abänder\ab-än-der\N
```

7.3 Ein einfacher Lemmatisator

gow.cd

```
1\abbestelle\1\35\ab-be-stel-le\N
2\abbestellst\0\35\ab-be-stellst\N
3\abbestellen\1\35\ab-be-stel-len\N
4\abbestellte\0\35\ab-be-stell-te\N
5\abbestelltet\0\35\ab-be-stell-tet\N
6\abbestellet\0\35\ab-be-stel-let\N
7\abbestellend\0\35\ab-be-stel-lend\N
8\aale\0\3\aa-le\N
9\aalst\0\3\aalst\N
10\aalt\0\3\aalt\N
```

gmw.cd

```
1\abbestelle\1\35\1SIE,13SKE
2\abbestellst\0\35\2SIE
3\abbestellen\1\35\13PIE,13PKE,i
4\abbestellte\0\35\13SIA,13SKA
5\abbestelltet\0\35\2PIA,2PKA
6\abbestellet\0\35\2PKE
7\abbestellend\0\35\pE
8\aale\0\3\1SIE,13SKE,rS
9\aalst\0\3\2SIE
10\aalt\0\3\3SIE,2PIE,rP
```