VSTENO Tutorial

Marcel Maci

18. August 2018

Abstract

Dieses Tutorial führt in die "Programmierung" von VSTENO ein. Mit Programmierung ist gemeint, dass Sie in VSTENO eigene Zeichen und Regeln definieren können, um Normalschrift in Kurzschrift umzuwandeln. Das Tutorial richtet sich hauptsächlich an LinguistInnen bzw. StenografInnen, die eigene Stenografie-Systeme mit VSTENO umsetzen möchten. Die hier gezeigten Beispiele basieren auf Zeichen und Regeln zur Umsetzung der deutschen Grundschrift von Stolze-Schrey.

Einleitung

Wenn Sie das Programm VSTENO einfach zur Übertragung von deutschen Texten in die Grundschrift von Stolze-Schrey nutzen möchten (z.B. zum Lesen von Texten auf einem E-Reader oder dergleichen), dann benötigen dieses Tutorial nicht, sondern können einfach die Demoversion von VSTENO mit den vordefinierten Zeichen und Regeln verwenden. Wenn Sie aber ein eigenes Stenografie-System umsetzen möchten - z.B. die Eilschrift von Stolze-Schrey oder auch andere Sprachen wie Spanisch, Französisch, Italienisch, Englisch oder ein System, das nicht von Stolze-Schrey ist, aber ähnlich funktioniert (wie z.B. DEK) - dann benötigen Sie das folgende Tutorial, um zu verstehen, wie man selber Zeichen und Regeln formuliert, um Langschrift- in Kurzschrifttexte umzuwandeln.

Mit Drucklegung dieser Dokumentation verfügt VSTENO noch über keine Datenbankfunktionen und keinen Editor zum Erstellen, Speichern und Einlesen von Zeichen und Regeln. Besagte Daten müssen deshalb als PHP-Code eingegeben werden. Ich bin mir bewusst, dass Linguisten und Sprachwissenschaftler nicht zwangsläufig über Programmierkenntnisse verfügen (oder zumindest wenig erpicht sind, linguistische Daten als Programmcode einzugeben). Dennoch ist dies - wie ich im Folgenden zeigen möchte - auch keine Hexerei (oder zumindest nicht schwieriger als das Verfassen eigener Zeichen und Regeln). Ein separator Editor für linguistische Daten ist selbstverständlich geplant. Die Implementierung desselben wir jedoch einige Zeit in Anspruch nehmen, weshalb es mir ein Anliegen ist, die linguistischen Funktionen bereits jetzt zu erklären und zu dokumentieren. Es ist übrigens zu betonen, dass sich an der Funktionalität

von VSTENO nichts ändert: Ob mit oder ohne Editor - die Möglichkeiten des Programmes bleiben genau die gleichen. Der geplante Editor wird somit nur eine andere (selbstverständlich leichtere und intuitivere) Möglichkeit sein, dem Programm die linguistischen Daten zur Verfügung zu stellen.

Zeichen

Linguistische Betrachtung

Beginnen wir mit den stenografischen Zeichen. Vermutlich kennen Sie jene Kindermalbücher mit Zeichnungen, die aus vielen nummerierten Punkten bestehen. Die Zeichnung entsteht, indem man den Stift bei Punkt 1 ansetzt und dann alle folgenden Punkte verbindet. VSTENO funktioniert im Prinzip genau gleich: Jedes Zeichen ist als eine "Folge von Punkten" definiert. Der einzige Unterschied zu den Kinderzeichnungen besteht darin, dass VSTENO die Punkte nicht einfach mit einer geraden Linie verbindet, sondern so genannte Bezier-Kurven durch diese Punkte legt - und zwar so, dass die Übergänge je nachdem möglichst "sanft" oder "spitz" verlaufen.

Ein erstes Zeichen ...

Aber der Reihe nach. Beginnen wir mit einem ganz einfachen Zeichen: dem "T" in der Grundschrift Stolze-Schrey. Dieses besteht aus nur einem Strich "von oben nach unten". D.h. wir können zwei Punkte definieren - einen "oberen" und einen "unteren" - und diese dann durch eine gerade Linie verbinden. Der entsprechende PHP-Code hierfür sieht folgendermassen aus:

```
"T" => array( /*header*/ 6, 0.5, 0, 0, 3, 3, 0, ""
/**/ ,"","","","",0,0,0,0,
/**/ 0,0,0,0,0,0,0,0,
/*data*/ 0, 20, 0, 1, 1.0, 0, 0, 0,
/**/ 0, 0, 0, 0, 1.0, 0, 1, 0 )
```

Wenn Sie noch nie programmiert haben, dann sieht dies vermutlich reichlich verwirrend aus, deshalb vereinfachen wir die Sache gleich noch etwas. Ausserdem können Sie alles zwischen den Zeichen /* und */ ignorieren: Hierbei handelt es sich um Kommentare, die für den Programmcode bzw. das Programm selber keine Bedeutung haben[1]. In diesem Fall weisen die Kommentare /*header*/ und /*data*/ darauf hin, dass es sich bei den folgenden Zahlen um Header-Informationen (= allgemeine Angaben zum Zeichen) und um eigentliche Daten (hier: Punkte, die das Zeichen definieren) handelt. Der Übersicht halber lassen wir den Header vorerst mal weg:

[1] Eine weitere Möglichkeit sind Kommentare mit //: diese sind im Unterschied zu /* und */ jedoch auf eine Zeile beschränkt.

```
"T" => array( /*data*/ 0, 20, 0, 1, 1.0, 0, 0, 0,
/**/ 0, 0, 0, 0, 1.0, 0, 1, 0 )
```

Damit verbleiben also zwei Datentupel - eines nach /*data*/ und eines nach /**/, die je aus 8 Werten bestehen. Das erste Datentupel, welches dem ersten Punkt entspricht, enthält die folgenden Werte:

```
 0, 20,  0,  1, 1.0,  0,  0,  0
x1, y1, t1, d1,  th, dr, d2, t2
```

Auf der zweiten Zeile habe ich die Bedeutung der einzelnen Werte notiert. Wichtig sind für uns im Moment vor allem die Werte x1 und y1, welche die Koordinaten des ersten Punktes (0,20) markieren. Der Vollständigkeit halber dokumentieren wir aber gleich alle Bedeutungen[2]:

- x1, y1: x- und y-Koordinate des Punktes
- t1, t2: Spannungen (tensions) im Anschluss an den Punkt (t1) und vor dem folgenden Punkt (t2)[3]
- d1, d2: Art des Punktes (der Wert d1 = 1 bedeutet, dass es sich um einen so genannten entry-point handelt - also den ersten Punkt des Zeichens).
- th: Dicke (thickness) - dieser Wert wird vor allem für Schattierungen verwendet und hat im Moment keine weitere Bedeutung.
- dr: Der so genannte draw-Wert (Zeichnungswert), der bestimmt, ob der Punkt verbunden oder abgesetzt gezeichnet werden soll (der Wert 0 bedeutet, dass der Punkt verbunden wird).

Es mag sein, dass die vielen Informationen im Moment etwas verwirrlich sind, aber wie bereits erwähnt geht es im Moment nur um die x- und y-Koordinaten. Wie man sehen kann wird im ersten Datentupel der Punkt 1=(0,20) und im zweiten Datentupel der Punkt 2=(0,0) definiert. Dies bedeutet nichts anderes als die Umsetzung dessen, was wir weiter oben vermerkt haben: Das Zeichen "T" ist die Verbindung zwischen einem "oberen" Punkt (0,20) und einem "unteren" Punkt (0,0).

An dieser Stelle weise ich gleich auf zwei weitere wichtige Aspekte hin: (1) Stenografie-Zeichen werden senkrecht (also ohne Neigung) definiert[4] und (2) das Koordinatensystem von VSTENO verläuft auf der x-Achse von links nach rechts und auf der y-Achse von unten nach oben. Die Grösse der Zeichen können Sie als LinguistIn im Prinzip frei wählen (da es sich um Vektorkoordinaten

[2] Aber bitte tun Sie mir den Gefallen: Vergessen Sie Werte, die wir nicht benötigen, gleich wieder - wir haben später Gelegenheit darauf zurückzukommen!

[3] Die Bedeutung der tension wird später erklärt. Sie gibt bei einer Bezier-Kurve an, ob der Punkt "spitz" oder "sanft" (rund) verbunden werden soll. Der Wert 0 gibt hier an, dass die Verbindung spitz sein soll, wie es das Zeichen "T" verlangt.

[4] Wenn sie, wie im System Stolze-Schrey, geneigt sein sollen, so kann VSTENO das geneigte Zeichen anhand des senkrechten selbständig berechnen.

handelt, lassen sich die Zeichen später beliebig und verlustfrei vergrössern oder verkleinern). Es wird aber empfohlen, eine gut lesbare und intuitiv verständliche Standardgrösse zu verwenden. Im vorliegenden Fall verwenden wir 10 Punkte für eine Stufe des Systems Stolze-Schrey. Da das Zeichen "T" zwei Stufen hoch ist, ergibt dies für die y-Koordinate den Wert 20. Beachten Sie, dass alle Werte Fliesskommazahlen sind, d.h. Sie können auch Koordinaten mit Kommastellen - z.B. 19.5 oder 19.999999 - verwenden.

Wenn Sie sehen möchten, wie das Zeichen "T" aussieht, dann gehen Sie am besten zur Demoversion und geben Wörter mit "T" ein, z.B. "beten". In der Demoversion ist eine Neigung von 60° voreingestellt, weshalb das Zeichen nicht senkrecht, sondern um 60 Grad geneigt erscheint. Wenn Sie das Zeichen so sehen möchten, wie wir es oben definiert haben, dann geben Sie im Formular unter Optionen 90° ein: Sowohl der Buchstabe "B" als auch "T" werden nun senkrecht dargestellt. Machen Sie sich im Moment noch keine Gedanken über den Vokal "E" oder die Endkürzung "EN". Widmen wir uns nun als nächstes dem Buchstaben "B". Dieser weist im Unterschied zu "T" eine Rundung am unteren Ende auf.

Rund oder spitzig?

Im ersten Abschnitt haben wir den denkbar einfachsten Fall behandelt: ein Zeichen, das mit zwei Punkten definiert wird, die spitzig miteinander verbunden werden sollen. Die Art und Weise, wie die Punkte verbunden werden sollen, ist im Datenfeld-Tension definiert. Betrachten wir noch einmal die zwei Punkte von "T":

```
"T" => array( 0, 20, /*p1t1*/ 0, 1, 1.0, 0, 0, /*p1t2*/ 0,
/**/ /*p2t1*/ 0, 0, 0, 0, 1.0, 0, 1, /*p2t2*/ 0 )
```

Zur Veranschaulichung habe ich Kommentare eingefügt. Wie man sieht, enthält jeder Punkt (p1 und p2) zwei Tensions (t1 und t2). Bevor wir über die Bedeutung dieser Tensions weiterreden, empfehle ich Ihnen am besten, die folgende Internet-Seite zu besuchen: Sie enthält eine interaktive Demo so genannter Splines. Ein Spline ist nichts anderes als eine Folge von Punkten (wie wir sie in unseren Zeichen, z.B. "T", definieren). Sie können nun mit den Tension der verschiedenen Punkte (im Englischen manchmal auch knots genannt) herumspielen und so ein intuitives Verständnis dafür bekommen, welchen Einfluss die Spannung auf den Verlauf von Bezier-Kurven haben:

http://scaledinnovation.com/analytics/splines/aboutSplines.html

Vereinfacht gesagt ergibt die Spannung mit dem Wert 0 eine spitze Verbindung, die Spannung mit dem Wert 0.5 eine runde Verbindung (es sind natürlich auch andere Werte - also "stärkere" und "schwächere" Spannungen - möglich). Wichtig zu wissen ist, dass die Kurve zwischen zwei Punkten P1 zu P2 durch zwei Spannungen definiert wird: Die Spannung p1t1 gibt die Spannung nach dem

Punkt P1 (Richtung P2) an, die Spannung p1t2 gibt die Spannung vor dem Punkt P2 (aus Richtung P1) an. Wir verwenden diese Spannungen nun, um das Zeichen "B" zu definieren:

```
"B" => array( /*data p1*/ 0, 10, 0, 1, 1.0, 0, 0, 0.5,
/*p2*/ 0, 2, 0.5, 0, 2.5, 0, 0, 0.5,
/*p3*/ 2.5, 0, 0.5, 0, 1.0, 0, 0, 0.5,
/*p4*/ 5, 2, 0.5, 0, 1.0, 0, 1, 0)
```

Wiederum haben wir aus Gründen der Übersichtlichkeit den Header weggelassen. Wenn wir nur die Koordinaten aus den Tupeln anschauen, wird das Zeichen "B" mit 4 Punkten definiert: P1(0,10), P2(0,2), P3(2.5,0), P4(5,2). Zur Veranschaulichung können Sie z.B. ein Blatt Papier nehmen, die Punkte auf einem Koordinatensystem eintragen und diese dann - wie eingangs anhand der Kinderzeichnungen erwähnt - verbinden. Beachten Sie, dass dieses Zeichen nur 10 Punkte hoch ist (im Unterschied zu "T" ist "B" einstufig).

Der ganze mathematische Hokuspokus - bzw. die Magie ;-) - liegt nun in den Tensions. Sie betragen für die Punkte 1-4: T1(0,0.5), T2(0.5,0.5), T3(0.5,0.5), T4(0.5,0). Das bedeutet, dass beim Zeichen "B" nur der erste Punkt spitz (Wert 0) verbunden wird. Alle anderen Punkte werden danach rund (mit der Spannung 0.5) verbunden. Die letzte Spannung T4 enthält in diesem Beispiel wieder den Wert 0: Dieser hat keine Bedeutung, denn wenn wir das Zeichen "B" mit einem weiteren Zeichen verbinden, so können wir nicht wissen, ob die Verbindung mit dem folgenden Zeichen rund oder spitzig sein muss (im Falle von "T" spitzig, im Falle von "M" hingegen rund). Anders gesagt: Dieser Wert hängt vom Folgezeichen ab - und es ist somit völlig egal, welchen Wert Sie hier eintragen (er wird später überschrieben).

Schattierungen

Bis jetzt haben wir Zeichen als Folge von Punkten definiert, die spitz oder rund miteinander verbunden werden. Damit lässt sich schon einiges machen! Allerdings verlangen gewisse Stenografie-Systeme - darunter auch Stolze-Schrey -, dass man Zeichen schattieren kann. Auch diese Funktion können wir durch Setzen der entsprechenden Werte innerhalb der Datentupel erreichen. Nehmen wir noch einmal unser einfaches Zeichen "T", wie wir es definiert haben:

```
"T" => array( /*data p1*/ 0, 20, 0, 1, 1.0, 0, 0, 0,
/*p2*/ 0, 0, 0, 0, 1.0, 0, 1, 0 )
```

An der 5. Stelle sehen wir hier den Wert 1.0. Dieser Wert entspricht der Dicke (thickness) und bedeutet also, dass das Zeichen immer mit der Standarddicke gezeichnet werden soll. Genau genommen ist der Wert 1.0 ein Multiplikationsfaktor in Bezug zu einer (vom/von der Nutzer/in vorgegebenen) Grunddicke: Werte >1.0 geben eine dickere, Werte <1.0 eine dünnere Linie an. Es können - wie für alle anderen Werte - Fliesskommazahlen verwendet werden (also auch 0.77 ist eine gültige Liniendicke). Wir passen diesen Wert nun an:

```
"T" => array( /*data p1*/ 0, 20, 0, 1, 2.5, 0, 0, 0,
/*p2*/ 0, 0, 0, 0, 1.0, 0, 1, 0 )
```

Wie man sieht, habe ich im ersten Datentupel (= Punkt 1) als Dicke den Wert 2.5 eingetragen. Dies bedeutet nun Folgendes: Wenn das Zeichen schattiert werden soll, dann wird ausgehend vom ersten Punkt P1 die Linie (genauer: Bezier-Kurve) zum Punkt P2 mit der 2.5-fachen Dicke gezeichnet. Bitte beachten Sie: Für das Zeichen "T" genügt es, nur diesen einen Wert zu ändern, um das Zeichen zu schattieren. Würden wir auch im Punkt P2 den Wert auf 2.5 erhöhen, so würde dies bedeuten, dass auch die Verbindungslinie zum nächsten Zeichen schattiert würde (was wir nicht wollen). Sie können diese Schattierung sehen, indem Sie in der Demoversion das Wort "Tat" eingeben: Das erste "T" wird normal gezeichnet, das zweite schattiert.

Leider ist es aber nicht immer so einfach mit den Schattierungen wie beim Zeichen "T", das mit einem spitzen Punkt beginnt und endet. Speziell bei Zeichen, die mit Rundungen beginnen und/oder aufhören, sieht es unschön aus, wenn wir einfach ab einem bestimmten Punkt eine wesentlich dickere Linie definieren. Bei runden Zeichen empfiehlt es sich also, die Schattierung in mehrere Schritte abgestuft beginnen und/oder enden zu lassen. Zur Illustration zeigen wir dies an unserem zweiten Beispielzeichen "B", welches einen spitzen Anfang und ein rundes Ende aufweist:

```
"B" => array( /*data p1*/ 0, 10, 0, 1, 2.5, 0, 0, 0.5,
/*p2*/ 0, 2, 0.5, 0, 1.75, 0, 0, 0.5,
/*p3*/ 2.5, 0, 0.5, 0, 1.0, 0, 0, 0.5,
/*p4*/ 5, 2, 0.5, 0, 1.0, 0, 1, 0)
```

Da unser Zeichen mit einem spitzen Punkt beginnt, können wir hier problemlos direkt die Dicke 2.5 einsetzen. Die Kurve von P1 zu P2 wird also "maximal dick" gezeichnet. Danach setzen wir ab Punkt P2 eine mittlere Dicke von 1.75 ein. Die Kurve von P2 zu P3 wird also "weniger dick" gezeichnet. Schliesslich bleibt noch die Verbindung von P3 zu P4: Hier kehren wir zur "normalen" Dicke von 1.0 zurück.

Intermediate shadow points

Die soeben dargestellte Abstufung der Schattierungen funktioniert relativ gut, wenn das Zeichen selbst bereits aus mehreren Punkten besteht, die eine abgestufte Definition der Schattierung zulassen. Schwieriger wird es, wenn das Zeichen als solches zu wenige Punkte enthält, um eine optisch einigermassen gelungene Abstufung zu erreichen. In diesem Fall besteht zwar die Möglichkeit, zusätzliche Punkte in die Zeichendefinition einzufügen, um mehr Zwischenschritte in der Schattierung zu erhalten. Diese Zwischenpunkte können jedoch den Nachteil haben, dass sie das optische Bild in der unschattierten Variante stören, da der kontinuierliche Lauf der Bezier-Kurve unterbrochen wird. Die von VSTENO verwendete Lösung besteht heir in so genannten "intermediate shadow points". Dies sind Zwischenpunkte, die nur dann gezeichnet werden, wenn das Zeichen

schattiert dargestellt werden soll. In der normalen Zeichendarstellung werden diese Punkte ignoriert. Als Beispiel zeigen wir das Zeichen "R"[5]:

```
"VR" => array( /*data*/  /*1*/ 2.5, 5, 0.5, 1, 1.5, 0, 0, 0.5,
/*2*/ 3.75, 4, 0.7, 5, 2.5, 0, 0, 0.7,
/*3*/ 5, 2.5, 0.7, 0, 3.0, 0, 0, 0.7,
/*4*/ 4.5, 0.5, 0.7, 5, 2, 0, 0, 0.7,
/*5*/ 3.25, 0.15, 0.7, 5, 1.5, 0, 0, 0.7,
/*6*/ 2.5, 0, 0.7, 0, 1.0, 0, 0, 0.5,
/*7*/  0, 2.5, 0.7, 0, 1.0, 0, 0, 0.5,
/*8*/ 2.5, 5, 0.5, 0, 1.0, 0, 1, 0.0)
```

Der Einfachheit halber wurden die Punkte 1-9 innerhalb von Kommentaren nummeriert. Für die Definition des nicht schattierten Zeichens "R" reichen im Prinzip die folgenden Punkte aus: 1=(2.5,5), 3=(5,2.5), 6=(2.5,0), 7=(0,2.5), 8=(2.5,5). Wie man sehen kann, markieren diese 5 Punkte, die Eckpunkte eines geschlossenen Kreises, die jeweils rund - d.h. mit Tensions zwischen 0.5-0.7 - verbunden werden. Würde man allerdings nur diese 5 Punkte für das schattierte Zeichen verwenden, so stünden für die Schattierung lediglich die Strecken von Punkt 1 - 3 und von Punkt 3 - 6 zur Verfügung. Aus diesem Grund wurden die übrigen Punkte - also 2, 4, 5 - als intermediate shadow points eingefügt. Erkennbar ist dies am Wert 5 an der 4. Stelle im Datatupel. Wie wir weiter oben schon angedeutet hatten, steht der Wert in der 4. Position für den "Typ" des Punktes. Wir haben hier schon den entry point (mit Wert 1) und den normalen Punkt (mit Wert 0) kennen gelernt. Der intermediate shadow point ist also einfach ein weiterer Typ, den ein Punkt annehmen kann.

Wie bereits erwähnt würde das Zeichen "VR" unschön aussehen, wenn diese Zwischenpunkte auch für das unschattierte Zeichen verwendet werden (dies ist hier besonders deutlich, weil Kreise kontinuierliche Kurven sind, wo es besonders auffällt, wenn sie durch weitere Punkte unterbrochen werden). Aus diesem Grund lässt der Zeichenalgorithmus von VSTENO diese Zusatzpunkte weg, wenn das Zeichen nicht schattiert ist.

Punkttypen

An dieser Stelle ist es nun an der Zeit, die verschiedenen Punkttypen in ihrer Vollständigkeit vorzustellen. Wiederum: Vergessen Sie bitte sämtliche Punkttypen, die hier vorgestellt werden und die noch nicht anhand von Beispielen erklärt werden. Im Moment genügt es, wenn Sie ein erstes Mal gehört haben, dass diese Punkttypen existieren.

Wichtig im Zusammenhang mit diesen Punkttypen ist zu wissen, dass es im Datentuplet zwei Stellen gibt, wo diese definiert werden. Hier noch einmal das Datentuplet des ersten Punktes, das wir für das Zeichen "T" definiert hatten:

[5] Da der Laut "R" im System Stolze-Schrey zwei Ausführungsvarianten hat, unterscheiden wir zwischen "AR" (= Anlaut-R) und "VR" (= vokalisches R). Das "vokalische R" ist somit jenes, das im Uhrzeigersinn ausgeführt wird und nach Vokal steht.

```
0,  20,  0,  1, 1.0,  0,  0,  0
x1, y1, t1, d1,  th, dr, d2, t2
```

Das Feld d1 steht für die Eingangs-Information (entry information), d.h. die Information, die angibt, wie das Zeichen verbunden wird. Es kann folgende Werte annehmen:

Wert	Bedeutung
0	normaler Punkt
1	entry point (Anfangspunkt = erster Punkt des Zeichens)
2	pivot point (Drehpunkt)
3	conditional pivot point (bedingter Drehpunkt)[6]
4	connecting point (Verbindungspunkt)
5	intermediate shadow point (Zwischenschattierungspunkt)
98	late entry point (später Anfangspunkt)

Analog steht das Feld d2 für die Ausgangs-Information (exit information), d.h. die Information, die angibt, wie das Zeichen "beendet" wird. Die Werte sind ähnlich wie bei d1, aber mit einigen kleinen Unterschieden:

Wert	Bedeutung
0	normaler Punkt
1	exit point (Endpunkt = letzter Punkt des Zeichens)
2	pivot point (Drehpunkt)
3	conditional pivot point (bedingter Drehpunkt)[7]
99	early exit point (früher Endpunkt)

Wie gesagt: Zerbrechen Sie sich nicht jetzt schon den Kopf über alle diese Punkttypen, wir werden sie einzeln mit Beispielen erläutern. Beginnen werden wir mit dem pivot point, also dem "Drehpunkt", mit dem Wert 2 in d1 oder d2.

Verbundene Rundungen

Zwei Stenozeichen mit spitzen Anfangs- und Endpunkten zu verbinden, ist trivial: Eine gerade Linie genügt! Schwieriger wird es jedoch, wenn eines - oder beide Zeichen - Rundungen als Anfangs- oder Endpunkt aufweisen. Möglich sind hier die Kombinationen spitz + rund ("T" + "M" im Wort "Thema"), rund + spitz ("B" + "T" im Wort "beten") oder rund + rund ("M" + "M" im Wort "Mumie"). Zusätzlich kann im System Stolze-Schrey (das wir immer als Beispiel nehmen) das folgende Zeichen eng oder weit verbunden und hoch oder tiefgestellt werden. Während die Verbindung "eng" vs "weit" noch einigermassen überschaubar ist (2 Möglichkeiten) ergibt sich bei der Hoch- und Tiefstellung eine Vielzahl von Fällen: Zum einen bedeutet Hoch- und Tiefstellung nicht bei jedem Zeichen dasselbe (beim Zeichen "B" bedeutet Hochstellung eine halbe Stufe höher, beim Zeichen "SCH" hingegen beträgt der Unterschied eine ganze Stufe), zum anderen können Zeichen halb-, ein-, zwei- oder dreistufig sein, was zu einer Unzahl von Verbindungsarten (damit meinen wir vor allem Länge, Winkel und allenfalls Verlauf der Verbindung) führt.

Aus diesem Grund besteht die Möglichkeit bei gerundeten Zeichen einen Drehpunkt zu definieren. Dieser Drehpunkt sollte sich im Fuss- oder Scheit-

elpunkt des Zeichens befinden und somit so gelegen sein, dass er das Zeichen in 2 (Zeichen mit einer Rundung) oder 3 Teile (Zeichen mit 2 Rundungen) auftrennt. D.h. dass solche Zeichen über einen fixen Mittel- oder Kernteil verfügen, an den sich die variablen Rundungen anschliessen. Der Drehpunkt markiert also den Übergang zwischen dem fixen und dem Variablen Teil eines Zeichens. Wir illustrieren dies am Beispiel "B", welches eine Rundung am Fusse aufweist:

```
"B" => array( /*data*/ /*1*/0, 10, 0, 1, 1.0, 0, 0, 0.5,
/*2*/ 0, 2, 0.5, 0, 2.5, 0, 0, 0.5,
/*3*/ 2.5, 0, 0.5, 0, 1.0, 0, 2, 0.5,
/*4*/ 5, 2, 0.5, 0, 1.0, 0, 1, 0)
```

Diese Definition ist identisch mit jener, die wir bereits weiter oben gegeben haben, mit dem einzigen Unterschied, dass der 3. Punkt als Drehpunkt definiert wird (Wert 2 im vorletzten Datentupel). Dies bedeutet nun, dass die Punkte 1-3 fix (also unveränderlich) sind, während der Punkt 4 (= exit point) variabel ist. Betrachten wir hier das Beispiel "Bohne": Hier trifft die Rundung von "B" mit der Tiefstellung des einstufigen Zeichens "N" zusammen, welches eng verbunden wird. Wie man leicht erkennen kann, befindet sich somit der Anschlusspunkt des Zeichens "N" unterhalb (!) des Endpunktes des Zeichens "B" (in Koordinaten gesprochen: der Endpunkt von "B" liegt bei y=2, der Anschlusspunkt von "N" liegt bei y=0). Mit anderen Worten: Würde das Zeichen "B" ohne Anpassung verbunden, dann entstünde ein hässlicher "Schnörkel" anstelle einer kontinuierlichen Linie.

VSTENO passt in diesem Fall die Endpunkte beider Zeichen (auch "N" ist ein Zeichen, welches mit einer Rundung beginnt) an, sodass ein "sanfter" Übergang entsteht.[8]

Early exit und late entry

Und es geht weiter mit einem neuen Punkttyp: dem early exit point oder dem vorzeitigen Endpunkt. Dieser wird im System Stolze-Schrey für Zeichen mit Unterschlaufe verwendet, die am Ende eines Wortes ohne Schlaufe geschrieben werden. Beispiele sind hier die Zeichen "NG" oder "NS" (als einstufige Variante) oder "SCH", "CH", "SCHW", "ZW" usw. (als zweistufige Variante). Der early exit point bedeutet also nichts anderes, als dass das Zeichen an diesem Punkt "vorzeitig" endet, wenn es am Ende eines Wortes steht. Hier das Zeichen "NS":

```
"NS" => array( /*data*/ 0.75, 5, 0.5, 1, 1.5, 0, 0, 0.5,
/**/ 3.75, 8.5, 0.5, 2, 2.0, 0, 0, 0.5,
```

[8] Die Anpassung erfolgt im Moment als simple Gerade, d.h. VSTENO nimmt den Drehpunkt von "B" und den Drehpunkt von "N", zieht eine Gerade zwischen den beiden und passt den Endpunkt von "B" und den Anfangspunkt von "N" so an, dass sie auf dieser Geraden liegen.

```
/**/ 2.65, 10, 0.5, 0, 2.5, 0, 0, 0.5,
/**/ 1.75, 9, 0.5, 0, 3.0, 0, 0, 0.5,
/**/ 1.75, 1, 0.5, 0, 3.0, 0, 0, 0.5,
/*late entry point*/ 0.75, 0, 0.5, 0, 2.5, 0, 99, 0.5,
/**/ 0, 2.25, 0.5, 0, 1.5, 0, 2, 0.5,
/**/ 1.75, 3, 0.5, 0, 1.0, 0, 1, 0.5 )
```

Einen ähnlichen Fall gibt es bei Zeichen, die mit einer Rundung beginnen und wo in Verbindung mit anderen Zeichen ein Teil der Rundung zweimal gezeichnet wird. Nehmen wir als Beispiel das Zeichen "V": In Zusammensetzungen wie "davon" - wo das Zeichen vom Fusspunkt der Kürzung "DA" - von unten her verbunden werden muss, wird der oberste Teil des Zeichens "V" zweimal gezeichnet (einmal beim Hochfahren, einmal beim Herunterfahren). Dies ist in diesem Fall - d.h. wenn das Zeichen verbunden ist - richtig und somit kein Problem. Nehmen wir nun aber nur die Kürzung "VON" (welche dem Zeichen "V" alleine entspricht). Auch hier würde VSTENO den obersten Teil des Zeichens zweimal zeichnen - was nicht schön ist (wenn die beiden Kurven beim Hoch- und Herunterfahren nicht exakt übereinander liegen, dann erscheint das Kopfende dunkler). Deshalb existiert auch hier der late entry point oder eben der "späte Eintrittspunkt":

```
"V" => array( /*data*/ 1, 16, 0.5, 1, 1.0, 0, 0, 0.5,
/**/ 2, 18, 0.6, 2, 1.0, 0, 0, 0,
/*late entry point*/ 6, 20, 0, 98, 0, 0, 0, 0.5,
/**/ 2, 18, 0.6, 0, 1.5, 0, 0, 0.5,
/**/ 1, 16, 0.5, 0, 2.5, 0, 0, 0.5,
/**/ 0, 14, 0.5, 0, 3.0, 0, 0, 0.5,
/**/ 0, 5, 0.5, 0, 3.0, 0, 0, 0.5,
/**/ 1, 2, 0.5, 0, 2.5, 0, 0, 0.5,
/**/ 3, 0, 0.5, 4, 1.5, 0, 0, 0.5,
/**/ 3, 0, 0.5, 0, 1.5, 0, 2, 0.5,
/**/ 5, 2, 0.5, 0, 1.0, 0, 1, 0.5 )
```

Der Wert 98 bedeutet hier also, dass die Punkte 1 und 2 nicht gezeichnet werden, wenn das Zeichen am Anfang steht. In diesem Fall wird der late entry point (also der 3. Punkt) als Anfangspunkt gesetzt.

Header

Bei all unseren Definitionen haben wir bis jetzt den Header der Einfachheit halber weggelassen. Dieser enthält Informationen, die das ganze Zeichen (also nicht nur einzelne Punkte) betreffen. In der aktuellen Version ist der Header 24 Felder lang, welche folgende Bedeutungen haben:

Offset	Bedeutung
0	token width (Zeichenbreite)
1	delta-y before (Delta-Y vorher)
2	delta-y after (Delta-Y nachher)
3	tension before (Spannung vorher)
4	additional x before (zusätzliches x vorher)
5	additional x after (zusätzliches x nachher)
6	additional delta-y (zusätzliches Delta-Y)
7	nicht verwendet
8	nicht verwendet
9	nicht verwendet
10	nicht verwendet
11	nicht verwendet
12	token type (Zeichentyp)
13	inconditional delta-y before (unbedingtes Delta-Y vorher)
14	inconditional delta-y after (unbedingtes Delta-Y nachher)
15	alternative exit point x (alternativer Endpunkt x)
16	alternative exit point y (alternativer Endpunkt y)
17	exit point to use (zu verwendender Endpunkt)
18	interpretation y coordinates (Interpretation y-Koordinaten)
19	vertical (vertikale Höher-/Tieferstellung)
20	distance (horizontaler Abstand)
21	shadowed (schattiert)
22	don't connect (nicht verbinden)
23	nicht verwendet

Beachten Sie bitte, dass wir die Feldnummer als Offset bezeichnen und mit der Nummerierung bei 0 beginnen. Auch hier werden wir wieder Schritt für Schritt vorgehen und die Bedeutung der einzelnen Werte anhand von Beispielen zeigen. Wir beginnen mit den Feldern 19-21, welche die Höher-/Tieferstellung (Offset 19), den horizontalen Abstand (Offset 20) und die Schattierung (Offset 21) betreffen.

Vokale

Diese drei Felder können wir verwenden, um im System Stolze-Schrey die Vokale zu definieren. Im Unterschied zu Konsonantenzeichen entsprechen diese nämlich keinen real geschriebenen Zeichen, sondern werden durch die Art, wie Zeichen verbunden werden, dargestellt. Mit anderen Worten: Ein Vokal entspricht im System Stolze-Schrey einem "leeren Zeichen" (= keine Punkte im Data-Bereich), das nur aus einem Header besteht. Als Beispiel der Vokal "E", der aus einer weiten Verbindung ohne Hochstellung und ohne Schattierung besteht:

```
"E" => array( /*header0-7*/ 0, 0, 0, 0, 0, 0, 0, "",
/*8-15*/ "","","","", 2, 0, 0, 0,
/*16-23*/ 0,0,0,"no","wide","no",0,0,
```

```
/*data*/ )
```

Zunächst einmal weisen wir auf den Offset 12 hin, der dem Zeichentyp (token type) entspricht. Hier wurde der Wert 2 gesetzt, der "virtuelles Zeichen" bedeutet (dagegen bezeichnet der Wert 0 ein "normales" und der Wert 1 ein "unbedingt schattiertes" Zeichen, wie wir später sehen werden). "Virtuell" bedeutet in diesem Fall nichts anderes, als dass das Zeichen keine grafische Entsprechung (als "Punktezeichnung") hat, sondern nur aus einem Header besteht.

Weiter weisen wir im Header-Untertupel 16-23 auf die Werte "no" (Offset 19), "wide" (Offset 20) und "no" (Offset 21) hin. Diese entsprechen somit der Angabe "keine Höher-/Tieferstellung" (also horizontale Verbindung), "weite Verbindung" und "nicht schattiert". Anbei die Liste der Werte, die Felder annehmen können:

Offset	Wert & Bedeutung
19	"no" (horizontale Verbindung), "up" (Höherstellung), "down" (Tieferstellung)
20	"wide" (weite Verbindung), "narrow" (enge Verbindung), "no" (kein Abstand)
21	"no" (keine Schattierung), "yes" (Schattierung)

Ein weiteres Beispiel: Der Diphtong "AU" wird als "enge Verbindung", "hochgestellt" und "schattiert" definiert:

```
"AU" => array( /*header0-7*/ 0, 0, 0, 0, 0, 0, 0, "",
/*8-15*/ "","","","", 2, 0, 0, 0,
/*16-23*/ 0,0,0,"up","narrow","yes",0,0,
/*data*/ ),
```

Bitte beachten Sie, dass in beiden Fällen der Data-Bereich leer bleibt: Die Vokale enhalten also wie bereits erwähnt keine Punkte!

Hochstellung

Mit den Offsets 19-21 im Header können wir also die Verbindung von Zeichen und insbesondere deren Hoch- oder Tiefstellung (Offset 19) bestimmen. Allerdings bedeutet "Hochstellung" nicht in allen Fällen dasselbe: Bei den meisten Zeichen im System Stolze-Schrey bedeutet es, dass das Folgezeichen eine halbe Stufe höher (d.h. dass sich die y-Koordinate sich um den Wert 5 erhöht) geschrieben wird. Es gibt jedoch auch Zeichen - z.B. "SCH", "CH", "Z", "ZW" etc. -, welche eine ganze Stufe (als 10 Punkte) höher geschrieben werden müssen. Ausserdem unterscheiden sich diese Zeichen darin, wie ein weiteres Folgezeichen angeschlossen wird: Bei den meisten Zeichen werden weitere Folgezeichen ebenfalls eine halbe Stufe höher geschrieben, bei "SCH", "CH", "Z", "ZW" etc. hingegen, muss nach der Hochstellung wieder zur Grundlinie zurückgekehrt werden. All dies kann im Header mit den Offsets 1 (delta-y before) und 2 (delta-y after) definiert werden:

```
"SCH" => array( /*header0-7*/ 9, 1,-1, 0.5, 0, 0, 0, "",
/*8-15*/ "","","","", 0, 0, 0, 0,
/*16-23*/ 0, 0, 0, 0, 0, 0, 0, 0,
/*data*/ /*...*/ )
```

Hier wird also im Offset 1 ein delta-y before mit dem Wert 1 (= 1 Stufe höher) und im Offset 2 ein delta-y after mit dem Wert -1 (= eine Stufe tiefer) definiert. Mit anderen Worten: Nachdem das Zeichen "SCH" eine Stufe höher an das Vorzeichen angeschlossen wird, wird die Schreiblinie danach wieder um den Wert -1 auf die ursprüngliche Linie zurückgesetzt. Im Vergleich dazu das Zeichen "B":

```
"B" => array( /*header0-7*/ 5, 0.5, 0, 0, 1, 1, 0, "",
/*8-15*/ "","","","", 0, 0, 0, 0,
/*16-23*/ 0, 0, 0, 0, 0, 0, 0, 0,
/*data*/ /*...*/ )
```

Hier wird die Schreiblinie vor dem Zeichen um eine halbe Stufe (Wert 0.5) erhöht und danach gleich belassen (der Wert 0 bedeutet, dass keine Veränderung vorgenommen wird).

Zeichenbreite

Bis jetzt haben wir Zeichen definiert, ohne uns Gedanken über deren Breite zu machen. Das sollten wir nun tun, denn es ist offensichtlich, dass jedes Zeichen eine Breite hat bzw. haben muss, um eine vernünftige und saubere Aneinanderreihung zu erreichen. Betrachten wir zum Beispiel noch einmal unser allererstes Zeichen "T", das aus einem einzigen senkrechten Strich besteht:

```
"T" => array( /*header0-7*/ 6, 0.5, 0, 0, 3, 3, 0, "",
/*8-15*/ "","","","", 0, 0, 0, 0,
/*16-23*/ 0, 0, 0, 0, 0, 0, 0, 0,
/*data p1*/ 0, 20, 0, 1, 1.0, 0, 0, 0,
/*p2*/ 0, 0, 0, 0, 1.0, 0, 1, 0 )
```

Da dieses Zeichen nur aus einem senkrechten Strich besteht, der in einer bestimmten Dicke gezeichnet wird, ist es per se nur so breit wie der Strich dick ist. Das macht aber wenig Sinn, da das Zeichen so unmittelbar auf ein vorhergehendes oder ein nachfolgendes Zeichen angeschlossen wird. Selbst bei "engem" bzw. "keinem" Abstand zwischen den Zeichen, sollten sie ein Minimum voneinander entfernt sein. Dies können wir durch die Werte in den Offsets 4 (additional x before) und Offset 5 (additional x after), sowie dem Offset 1 (token width) erreichen. Der Wert in Offset 4 entspricht dabei dem "linken", Offset 5 dem "rechten" Leerraum, der auf das Zeichen folgt. Im obigen Beispiel wurde hier 3 Pixel für den linken und 3 Pixel für den rechten Abstand definiert, was für das Zeichen eine Gesamtbreite von 6 Pixeln (Offset 0) ergibt.

Bitte beachten Sie, dass die Werte in den Offsets 4 und 5 zusätzlich zur jener Breite addiert werden muss, welche sich aus den Punkten, die das Zeichen definieren, ergibt:

```
"Y" => array( /*header0-7*/ 14, 0.5, 0, 0, 2, 2, 0, ""
/*8-15*/ "","","","", 0, 0, 0, 0,
/*16-23*/ 0, 0, 0, 0, 0, 0, 0, 0,
```

```
/*data p1*/ 0, 10, 0, 1, 3.0, 0, 0, 0,
/*p2*/ 10, 0, 0, 0, 1.0, 0, 1, 0 /**/ ),
```

Das Zeichen "Y" - in Stolze-Schrey als gerader Strich von links oben P1(0,10) nach rechts unten P2(10,0) definiert, ist - rein von den Punkten her - bereits 10 Pixel breit. Wird nun in Offsets 4 und 5 ein zusätzlicher Leerabstand links und rechts von 2 Pixeln (also insgesamt 4 Pixel) definiert, so ist das Zeichen insgesamt 14 Pixel breit.

Es empfiehlt sich, beim Erstellen neuer Zeichen, mit diesen Möglichkeiten etwas herumzuspielen, um optimale Werte zu finden, die in Verbindungen mit anderen Zeichen am besten aussehen.

Schreiblinienverschiebung

Es gibt auch Zeichen, bei welchen sich die Schreiblinie in jedem Fall - also unabhängig von Hoch- oder Tiefstellung - verschiebt. Im System Stolze-Schrey ist dies z.B. bei "RR" der Fall: Hier müssen Folgezeichen eine Stufe höher angeschlossen werden. Wir erreichen dies durch Setzen der Offsets 13 (inconditional y before) und 14 (inconditional y after):

```
"RR" => array( /*header0-7*/ 10, 0.5, 0, 0.5, 0, 0, 0, "",
/*8-15*/ "","","","", 0, 0, 0, 0,
/*16-23*/ 0, 0, 0, 0, 0, 0, 0, 0,
/*data p1*/ 1, 7.75, 0.5, 1, 1.0, 0, 0, 0.5,
/*p2*/ 5, 10, 0.7, 0, 3.0, 0, 0, 0.8,
/*p3*/ 10, 5, 0.8, 0, 3.0, 0, 0, 0.7,
/*p4*/ 5, 0, 0.7, 0, 1.0, 0, 0, 0.5,
/*p5*/ 0, 5, 0.5, 0, 1.0, 0, 0, 0.5,
/*p6*/ 1, 7.75, 0.5, 0, 1.0, 0, 0, 0.5,
/*p7*/ 5, 10, 0.5, 0, 1.0, 0, 1, 0.5, /**/ )
```

In diesem Zeichen sehen wir gleich zwei Phänomene: (1) Die Schreiblinie verschiebt sich nach dem Zeichen in Normalstellung um 1 Stufe nach oben (Offset 14 mit Wert 1) und (2) Wird das Zeichen höher gestellt, so beträgt die Höherstellung eine halbe Stufe (Wert 0.5 im Offset 1) und die Schreiblinie liegt 1.5 (d.h. die Summe aus Offset 1 und Offset 14) höher![9]

Unbedingte Schattierung

Und weiter geht's mit Besonderheiten des Systems Stolze-Schrey. Nebst allen präsentierten Anforderungen an stenografische Zeichen, gibt es auch Abkürzungen, in jedem Fall eine Schattierung verlangen. Ein Beispiel dafür ist die Abkürzung "AUCH", welche aus einem höher gestellt, schattierten "CH" besteht:

[9] Falsch wäre hier, die Schreiblinienverschiebung in Offset 3 - z.B. mit dem Wert 0.5 - zu definieren: Dies würde zwar für höher gestellte "RR" gelten, normal gestellte "RR" würden jedoch falsch geschrieben, da sich bei "RR" die Schreibline in jedem Fall um eine Stufe erhöht!

```
"AUCH" => array( /*header0-7*/ 5, 1,-1, 0.5, 0.5, 0.5, 0, "",
/*8-15*/ "","","","", 1, 1, 1, 0,
/*16-23*/ 0, 0, 0, 0, 0, 0, 0, 0,
/*data p1*/ 0, 8.5, 0.5, 1, 1.3, 0, 0, 0.5,
/*p2*/ 2.5, 10, 0.7, 2, 2.5, 0, 0, 0.8,
/*p3*/ 5, 7, 0.8, 0, 3.0, 0, 0, 0.5,
/*p4*/ 5, -8, 0.5, 0, 2.5, 0, 0, 0.5,
/*p5*/ 3, -10, 0.5, 0, 2, 0, 0, 0.5,
/*p6*/ 1.5, -9, 0.5, 0, 1.5, 0, 99, 0.5,
/*p7*/ 0, -7, 0.5, 0, 1.0, 0, 2, 0.5,
/*p8*/ 5, 0, 0.5, 0, 1.0, 0, 1, 0.5 ),
```

Definiert wird die Schattierung hier im Offset 12 durch den Wert 1. Wie wir bereits bei den Vokalen gesehen haben, steht der Offset 12 für den Zeichentyp (token type). Bei Vokalen haben wir hier den Wert 2 (virtuelles Zeichen) gesetzt, für alle anderen Zeichen den Wert 0 (normales Zeichen). Der Wert 1 nun bedeutet für VSTENO, dass das Zeichen in jedem Fall schattiert werden soll (also unabhängig davon, welcher Vokal oder welches Zeichen vorausgeht).

Bitte beachten Sie in diesem Zeichen "AUCH" eine weitere Besonderheit: Da "AUCH" einem hoch gestellten "CH" entspricht, konnte hier einfach die Definition von "CH" kopiert und in den Offsets 13 und 14 (inconditional y before/after), die wir bereits gesehen haben, der Wert +1 gesetzt werden (was bedeutet, dass das ursprüngliche Zeichen "CH" einfach um eine Stufe nach oben verschoben wird). Definitionen dieser Art sind sehr effizient, da man bereits definierte Zeichen wiederverwenden (und wie hier für eine Kürzung gebrauchen) kann. Wir werden später noch weitere Möglichkeiten sehen, um aus bereits definierten Zeichen zusätzliche - kombinierte oder verschobene - Zeichen zu erstellen.

Das Kopieren eines Zeichens hat aber auch den Nachteil, dass die Definition u.U. nicht optimal ist: Im Falle von "AUCH" verwendet das urprüngliche Zeichen "CH" z.B. einen early exit point in Punkt P6 (Wert 99 an vorletzter Stelle im Datentupel). Da die Kürzung "AUCH" immer alleine steht, kann man den early exit point auch durch einen normalen Endpunkt ersetzen und die Punkte P7 und P8 löschen:

```
"AUCH" => array( /*header0-7*/ 5, 1,-1, 0.5, 0.5, 0.5, 0, "",
/*8-15*/ "","","","", 1, 1, 1, 0,
/*16-23*/ 0, 0, 0, 0, 0, 0, 0, 0,
/*data p1*/ 0, 8.5, 0.5, 1, 1.3, 0, 0, 0.5,
/*p2*/ 2.5, 10, 0.7, 2, 2.5, 0, 0, 0.8,
/*p3*/ 5, 7, 0.8, 0, 3.0, 0, 0, 0.5,
/*p4*/ 5, -8, 0.5, 0, 2.5, 0, 0, 0.5,
/*p5*/ 3, -10, 0.5, 0, 2, 0, 0, 0.5,
/*p6*/ 1.5, -9, 0.5, 0, 1.5, 0, 1, 0.5 )
```

Eine letzte Bemerkung zur Kürzung "AUCH": Es ist VSTENO vollkommen egal, ob Sie nun eine Kürzung oder ein Zeichen definieren. VSTENO betrachtet alles, was gezeichnet werden kann, als Zeichen. Die Namen der Zeichen können einen oder beliebig viele Buchstaben lang sein (auch die Kürzung "VIELLEICHT" wird - obwohl es in der Langschrift 10 Buchstaben lang ist - von VSTENO als 1 Zeichen betrachtet). Auch das Aneinanderreihen von Buchstaben und/oder Kürzungen macht für VSTENO keinen Unterschied: Das Wort "dafür" zum Beispiel wird von VSTENO als zwei Kürzungen betrachtet, welche als zwei Zeichen (schattiertes D + normales F) aneinandergereiht wird. Wir werden später bei den Regeln sehen, wie Kürzungen - und die Übertragung derselben aus der Langschrift - definiert werden können.

Alternative exit points

Man würde nun vielleicht denken, dass wir langsam alle Besonderheiten stenografischer Zeichen abgedeckt haben, aber dem ist nicht so: Es gibt weitere Zeichen, die nach Spezialfunktionen verlangen und dazu gehören jene, welche Folgezeichen auf zwei verschiedene Arten anschliessen können: entweder (1) "normal": also auf der gleichen Schreiblinie wie bei 95% der Fälle) oder aber (2) "anders": in ganz wenigen Fällen. Zu diesen Zeichen gehört z.B. das vokalische R "VR", welches Folgezeichen normalerweise auf der Grundlinie anschliesst (vgl. "gern": die Zeichen r und n stehen auf der gleichen Linie), die Endungskürzungen "(D)EN" und "(D)EM" hingegen am oberen Ende anschliesst (vgl. "äusseren": das Zeichen r steht auf der Grundlinie, die Endung en hingegen eine halbe Stufe höher. Dies können wir mit dem Offset 16 alternative exit point (alternativer Endpunkt) im Header definieren[10]:

```
"VR" => array( /*header0-7*/ 5, 0.5, 0, 0.5, 2, 2, 0, "",
/*8-15*/ "","","","", 0, 0, 0, 2.5,
/*16-23*/ 5, 0, 0, 0, 0, 0, 0, 0,
/*data p1*/ 2.5, 5, 0.5, 1, 1.5, 0, 0, 0.5,
/*p2*/ 5, 2.5, 0.7, 0, 3.0, 0, 0, 0.7,
/*p3*/ 2.5, 0, 0.7, 0, 1.0, 0, 0, 0.5,
/*p4*/ 0, 2.5, 0.7, 0, 1.0, 0, 0, 0.5,
/*p5*/ 2.5, 5, 0.5, 0, 1.0, 0, 1, 0.0),
```

Wie wir sehen können, enthält der Header im Offset 16 den Wert 5. Dies bedeutet nun, dass der "alternative Endpunkt" auf der y-Achse 5 Pixel höher liegen soll. Dies entspricht dem y-Wert des letzten Punktes P5 - also dem Kopfende des Zeichens "VR". Die Definition einer x-Koordinate ist nicht nötig, da diese automatisch beim Aneinanderfügen der Zeichen errechnet wird.

Die Frage ist nun: Wann kommt dieser alternative Endpunkt zum Einsatz. Hierzu sehen wir uns die Definition der Endung "EN" an:

[10] Der Einfachkeit halber werden in der Definition die intermediate shadow points weggelassen.

```
"EN" => array( /*header0-7*/ 5, 0, 0, 0.5, 0, 0, 0, "",
/*8-15*/ "","","","", 0, 0, 0, 0,
/*16-23*/ 0, 1, 0, 0, 0, 0, 0, 0,
/*data*/ 5, 0, 0, 1, 1.0, 0, 0, 0,
/**/ 5, 0, 0, 0, 1.0, 0, 1, 0 ),
```

Der entscheidende Wert steht hier im Offset 17 des Headers: Dieser Wert entspricht dem exit point to use (zu verwendender Endpunkt). VSTENO handhabt dies nun folgendermassen: Verlangt ein Zeichen - wie in diesem Fall die Kürzung "EN" - nach einem alternativen Endpunkt, so prüft VSTENO, ob das vorhergehende Zeichen einen solchen definiert. Ist dies der Fall (wie beim Zeichen "VR"), dann wird er verwendet. Bietet das vorhergehende Zeichen keinen alternativen Endpunkt an, so wird der normale Endpunkt verwendet (was das richtige Ergebnis liefert, vgl. z.B. "laufen": hier wird die Endung "EN" auf der Grundlinie angeschlossen).

Bietet ein Zeichen einen alternativen Endpunkt an und das Folgezeichen verlangt nicht danach, so wird der alternative Endpunkt ignoriert.

Absolute Koordinaten

Damit nähern wir uns allmählich doch dem Ende der Erklärungen zum Header. Es bleiben uns allerdings noch drei letzte Dinge zu klären. Das erste betrifft den Offset 18, der für die "Interpretation y-Koordinate" steht. Dieses Feld kann zwei Werte annehmen:

Wert	Bedeutung
0	y-Koordinaten sind relativ (standard)
1	y-Koordinaten sind absolut

Diese Einstellung tut genau das, was der Name sagt: Sämtliche Zeichen, die wir bis jetzt definiert haben, verwenden relative Koordinaten (Standardeinstellung, Wert 0), was bedeutet, dass die Zeichen bei Höher- oder Tieferstellung automatisch "mitverschoben" werden. Wird hier der Wert 1 gesetzt, so wird die y-Koordinaten in den Punkten absolut gesetzt. Dies kann verwendet werden, wenn Zeichen unter keinen Umständen verschoben werden sollen[11].

Unverbundene Zeichen

Der zweitletzte Hinweis zum Header betrifft unverbundene Zeichen: Dies können zum Beispiel Zahlen sein. Diese werden - auch in einem stenografischen Text - als normale Zahlen geschrieben (und dürfen somit nicht an das vorangehende Zeichen angeschlossen werden). Als Beispiel hier die Definition der Zahl 1 (die wir wiederum deshalb wählen, weil sie nur aus zwei geraden Strichen besteht und damit sehr einfach ist):

[11] Wir fügen hier kein Beispiel an, weil das Phänomen in der Grundschrift kaum vorkommt. In der Eilschrift hingegen gibt es die Kürzung "Ding(e)", welche dem Zeichen "NG" in unschattierter Höherstellung entspricht. Dieses Zeichen muss unbedingt abgetrennt und in Höherstellung geschrieben werden.

```
"1" => array( /*header0-7*/ 7, 0, 0, 0, 4, 0, 0, "",
/*8-15*/ "","","","", 0, 0, 0, 0,
/*16-23*/ 0, 0, 1, 0, 0, 0, 1, 0,
/*data p1*/ 0, 11, 0, 1, 1.0, 0, 0, 0,
/*p2*/ 7, 19, 0, 0, 1.0, 0, 0, 0,
/*p3*/ 7, 1, 0, 0, 1.0, 0, 1, 0 )
```

Der Wert 1 im Offset 22 bedeutet also "dieses Zeichen nicht verbinden" (der Wert 0 - den wir bis jetzt immer verwendet haben - bedeutet hingegen "dieses Zeichen verbinden").

Alternativ können Sie auch das Feld dr (draw) im Datentupel des ersten Punktes P1 auf den Wert 5 (= don't connect) setzen:

```
"1" => array( /*header0-7*/ 7, 0, 0, 0, 4, 0, 0, "",
/*8-15*/ "","","","", 0, 0, 0, 0,
/*16-23*/ 0, 0, 1, 0, 0, 0, 0, 0,
/*data p1*/ 0, 11, 0, 1, 1.0, 5, 0, 0,
/*p2*/ 7, 19, 0, 0, 1.0, 0, 0, 0,
/*p3*/ 7, 1, 0, 0, 1.0, 0, 1, 0 )
```

Zu guter Letzt: Wenn Sie auf Nummer sicher gehen wollen, können Sie sowohl 1 im Header also auch 5 im draw-Feld des Punktes eintragen ... ;-)[12]

Anfangsspannung

Damit kommen wir zum definitiv letzten Punkt des Headers, dem Offset 3, welcher der Bedeutung tension before (Anfangsspannung betrifft). Wie wir bei der Definition einzelner Punkte gesehen haben, enthält jedes Datentupel zwei Spannungen: t1 für die Spannung die auf den Punkt folgt und Spannung t2 vor dem nächsten Punkt. Wenn die Spannungswerte auf diese Art gespeichert werden, ist schnell klar, dass für den ersten Punkt keine Anfangsspannung definiert werden kann (da ihm ja dann ein Punkt vorausgehen müsste, der ein Feld tension before enthält). Dies wird nun so gelöst, dass die Spannung vor dem ersten Punkt in den Offset 3 des Headers geschrieben wird.

Die Standardregel für diesen Wert lautet: Zeichen, die mit einer Rundung beginnen (z.B. "M"), sollten hier den Wert 0.5 (oder ähnlich) enthalten, Zeichen die spitz beginnen (z.B. "B" oder "T"), den Wert 0.

Kombinierte Zeichen

Nachdem wir den Header nun vollständig kennen, widmen wir us noch einmal den Punkten, um auch hier ein paar letzte, sehr hilfreiche Funktionen zu erläutern.

[12] Zu bevorzugen ist in diesem Fall die Variante "Header": Es handelt sich hier ja um eine Information, die das ganze Zeichen betrifft. Das draw-Feld des Punktes hingegen sollte verwendet werden, wenn INNERHALB eines Zeichens gewisse Punkte nicht miteinander verbunden werden.

Im System Stolze-Schrey gibt es bekanntlich Zeichen, welche sich mit anderen "kombinieren" können. Es sind dies vor allem R und L in Verbindung mit verschiedenen Konsonantenzeichen wie z.B. T, B etc. Wir könnten nun für diese Kombinationen einfach neue Zeichen definieren. D.h. in unserer Gesamtliste an Zeichen würden wir zuerst ein Zeichen "T" definieren, dann ein Zeichen "T+R" und schliesslich ein Zeichen "T+L". Das Problem hierbei: Wir betreiben damit doppelt und dreifachen Aufwand! Es wäre viel effizienter, wenn VSTENO eine Funktion anböte, um gewisse Zeichen automatisch zu kombinieren - und eine solche Funktion existiert in der Tat: Sie nennt sich TokenCombiner.

Damit der TokenCombiner zwei Zeichen verbinden kann, muss er jedoch wissen, wo (d.h. an welcher Stelle) Zeichen miteinander verbunden werden können und auf welche Zeichen dies angewandt werden soll. VSTENO stellt hier die Funktion so genannter connection points (Verbindungspunkte) zur Verfügung: Ein Verbindungspunkt ist ein Punkt eines Zeichens, der im Datentupel-Feld d1 (= Offset 3) den Wert 4 enthält. Als Beispiel zeigen wir wieder unser Zeichen "T":

```
"T" => array( /*header*/ 0, 0.5, 0, 0,4,2.5, 0, "",
/**/ "","","","", 0, 0, 0, 0,
/**/  0, 0, 0, 0, 0, 0, 0, 0,
/*data p1*/ 0, 20, 0, 1, 3.0, 0, 0, 0,
/*p2*/ 0, 0, 0, 0, 1.0, 0, 1, 0,
/*p3*/ 0, 2.5, 0, 4, 1.0, 0, 0, 0.5 )
```

Wir haben hier das Zeichen "T" also um einen zusätzlichen dritten Punkt P3 ergänzt, welcher die Koordinanten (0,2.5) aufweist. Dieser Punkt liegt also auf halber Höhe eines halbstufigen Zeichens - und ist somit der ideale Ansatzpunkt, um ein "R" einzufügen.

Das Problem ist hier jedoch, dass wir nicht die bereits definierten r (d.h. "VR" oder "AR") verwenden können, da diese entweder oben bzw. bei 12 Uhr ("VR") oder unten bzw. bei 6 Uhr ("AR") beginnen. Deshalb definieren wir hier ein spezielles R, welches bei 3 Uhr - also rechts auf Viertelhöhe, d.h. genau bei der Koordinate (0,2.5) - beginnt:

```
"@R" => array( /*header0-7*/ 5, 0.5, 0, 0.5, 0, 1, 0, "",
/*8-15*/ "","","","", 0, 0, 0, 0,
/*16-23*/ 0, 0, 0, 0, 0, 0, 0, 0,
/*data p1*/ 0, 0, 0.7, 0, 1.0, 0, 0, 0.7,
/*p2*/ -2, 2, 0.7, 0, 1.0, 0, 0, 0.5,
/*p3*/ -4, 0, 0.7, 0, 1.0, 0, 0, 0.5,
/*p4*/ -2, -2, 0.5, 0, 1.0, 0, 2, 0.5,
/*p5*/ 0, 0, 0.5, 0, 1.0, 0, 1, 0.7 )
```

Beachten Sie nun, dass wir dieses spezielle R - das wir "@R" nennen, nun nicht an der Koordinate (0,2.5) - wo das Zeichen letztlich eigentlich hinkommen soll -, sondern an der Ursprungskoordinate (0,0) beginnen. Mit anderen

Worten: Sämtliche Koordinaten dieses "Verbundzeichens" sind relativ zu verstehen. Beim Einfügen werden hier also jeweils die Koordinaten des connection point - im dem Falle also (0,2.5) - dazuaddiert. Die Punkte P1-P5 definieren einen Kreis im Gegenuhrzeigersinn, der im Ursprung endet (wo er begonnen hat). Beachten Sie auch, dass der Punkt P4 als Drehpunkt definiert wird: Dadurch wird sichergestellt, dass Zeichen, die tiefergestellt werden (wie z.B. in "Thron") später elegant angeschlossen werden.

Nachdem wir unsere beiden Einzel-Zeichen nun definiert haben, brauchen wir VSTENO nur noch zu sagen, dass wir die Zeichen kombinieren wollen. Wir tun dies, indem wir dem TokenCombiner vier Informationen übergeben:

```
$token_combiner = array(
    array( "T", "@R", 0, 0 )
);
```

Machen Sie sich wiederum nicht zu viele Gedanken um die Klammern, die Variable $token_combiner und den Strichpunkt (es ist PHP-Code). Es geht hier im Moment nur um die 4 Informationen, die der TokenCombiner benötigt: (1) erstes Zeichen: "T", (2) zweites Zeichen: "@R", (3) Vorgängiges Delta-y: 0, (4) Nachträgliches Delta-y: 0.

Diese Zeile genügt also, um das neue Zeichen "T@R" zu generieren. Beachten Sie hierbei, dass der Name des kombinierten Zeichens immer die Zusammenfügung von "Zeichen1" + "Zeichen2" ist (hier also "T@R"). Die Werte vorgängiges und nachträgliches Delta-y werden in den Header des neuen Zeichens geschrieben. Genauer gesagt: Der TokenCombiner verwendet den Header des ersten Zeichens als Basis und ersetzt dann nur diese beiden Felder durch die angegebenen Werte. Ebenfalls passt der TokenCombiner die Breite des neuen Zeichens automatisch an (die Breite des neuen Zeichens entspricht der Summe der beiden Zeichenbreiten). Zu guter Letzt: Wenn wir zwei Zeichen kombinieren, besteht natürlich die Gefahr, dass das neue Zeichen mehrere Anfangs- und Endpunkte hat. Der TokenCombiner analysiert deshalb das neue Zeichen, löscht überflüssige Anfangs- oder Endpunkte (indem er sie in "normale" Punkte umwandelt) und behält nur den ersten und letzten Punkt des Zeichens als Anfangs- und Endpunkt.

Der TokenCombiner ist ein ungemein praktisches Werkzeug, um im Handumdrehen neue Zeichenkombinationen zu generieren. Z.B. genügt es, beim Zeichen "B" einen Verbindungspunkt P3 = 0, 2.5, 0, 4, 1.0, 0, 0, 0.5 wie im Zeichen "T" einzusetzen und den TokenCombiner zu ergänzen:

```
$steno_tokens_master = array(
     "D" => array( /*header*/ 0, 0.5, 0, 0, 2 ,3, 0, "",
     /**/ "","","","",0,0,0,0,
     /*+++*/ 0,0,0,0,0,0,0,0,
     /*data*/ 0, 10, 0, 1, 3.0, 0, 0, 0,
     /**/ 0, 0, 0, 0, 1.0, 0, 1, 0,
     /*connection point*/ 0, 2.5, 0, 4, 1.0, 0, 0, 0.5 )   // neue Zeile
);
$combiner_table = array(
```

```
    array( “T”, “@R”, 0, 0 ),
    array( “D”, “@R”, 0, 0 )     // neue Zeile
);
```

Zwei Zeilen genügen somit, um das neue Zeichen “D@R” zu generieren. Da dieses - wie alle übrigen Grundzeichen - ebenfalls in die Variable $steno_tokens_master geschrieben wird, können Sie es danach ohne Einschränkung wie jedes andere Zeichen verwenden!

Verschobene Zeichen

Ähnlich wie der TokenCombiner funktioniert auch der TokenShifter: Dieser ermöglicht es, Zeichen horizontal oder vertikal zu verschieben. Im System Stolze-Schrey kann dies z.B. dazu verwendet werden, Anschlüsse von Zeichen an ein Aufstrich-T zu definieren, ohne das entsprechende Zeichen noch einmal neu eingeben zu müssen:

```
$shifter_table = array(
    array("N", "&TN", 4, 15, 0, 1.5)
);
```

Der TokenShifter benötigt 6 Informationen, die folgendes bedeuten: (1) zu verschiebendes Zeichen: “N”, (2) Name des neuen Zeichens: “&TN”, (3) Verschiebung auf x-Achse: 4 Pixel nach rechts, (4) Verschiebung auf y-Achse: 15 Pixel nach oben, (5) vorgängiges Delta-y: 0 (Schreibzeile verschiebt sich nicht), (6) nachträgliches Delta-y: 15 (Schreibzeile verschiebt sich um 15 Pixel bzw. 1.5 Stufen nach oben). Das Grundzeichen “N” wird durch diese Anweisung also um 1.5 Stufen nach oben und 4 Pixel nach rechts verschoben. Das neue Zeichen heisst “&TN” und die Schreiblinie verschiebt sich nach dem Zeichen um 1.5 nach oben. Wenn nun dieses Zeichen im Wort “Zentner” vom Grundlinien-N aus verbunden wird, so entspricht “&TN” einem Aufstrich-T. Die neue Schreiblinie befindet sich am Fusspunkt des Zeichens “N”.

Auch dies ist eine sehr effiziente Art, neue Zeichen zu generieren. Für das neue Zeichen “&TENS”, welches einem Aufstricht-T verbunden mit dem Grundzeichen “NS” entspricht, genügt z.B. die folgende zusätzliche Zeile:

```
$shifter_table = array(
    array("N", "&TN", 4, 15, 0, 1.5),
    array("NS", "&TENS", 4, 10, 0, 1)
);
```

Soll statt dem Aufstrich-T die Kürzung “HEIT” verwendet werden, genügt eine grössere Verschiebung auf der x-Achse:

```
$shifter_table = array(
    array("N", "&TN", 4, 15, 0, 1.5),
    array("NS", "&TENS", 4, 10, 0, 1),
    array("NS", "&EITENS", 18, 10, 0, 1)
);
```

Beachten Sie dass die vertikale Verschiebung hier nur 10 Pixel beträgt, da das Zeichen "NS" im Unterschied zu "N" einstufig ist. Aus diesem Grund kommt auch die neue Schreiblinie nur 1 Stufe höher zu liegen (also eine halbe Stufe tiefer als bei "N").

Zeichen als PHP-Code

Wie eingangs erwähnt verfügt VSTENO im Moment noch über keinen Editor, um Zeichen grafisch zu editieren und automatisch einzulesen. Deshalb bleibt bis auf Weiteres nichts anderes übrig als alle Daten direkt als PHP-Code einzugeben. Ich habe weiter oben geschrieben, dass Sie kein/e PHP-Programmierer/in sein müssen, um VSTENO als Linguist/in zu verwenden. Das stimmt insofern, als dass Sie keine eigenen Algorithmen oder Programmteil erstellen müssen, um VSTENO z.B. für ein neues System anzupassen. Das "Programmieren" beschränkt sich also streng genommen auf das Definieren bestimmter Variablen: Diese enthalten dann die linguistischen Daten, auf die VSTENO zugreift.

Dieser Aufgabe widmen sich nun also die folgenden Kapitel: Ein Mini-Crash-Kurs in PHP sozusagen, bei dem es lediglich darum geht, dass Sie lernen, wie und welche Variablen Sie in PHP setzten müssen, damit VSTENO funktioniert. Da wir bis jetzt erst Zeichen (und noch keine Regeln) behandelt haben, beschränken wir uns zunächst nur darauf. Die Frage somit: Wie geben Sie die oben erläuterten Zeichendefinitionen in PHP ein?

Grundstruktur

Eine PHP-Datei trägt den Namen dateiname.php und lässt sich mit jedem beliebigen Texteditor editieren. Die linguistischen Daten von VSTENO befinden sich in der Datei data.php. Wenn Sie also eigene Zeichen und Regeln definieren wollen, dann öffnen Sie am besten diese Datei, löschen Sie sämtlichen Inhalt (machen Sie vorher ein Backup, wenn Sie die Original-Datei später noch verwenden möchten) und beginnen Sie mit folgenden Zeilen[13]:

```
<?php
        // hier können Sie Kommentarzeilen
        /* oder PHP-Code einfügen */
?>
```

Diese Zeilen bedeuten nichts anderes, als dass zwischen <?php und ?> unser ganzer PHP-Code stehen wird. Auch Kommentarzeilen können Sie, wie bereits weiter oben beschrieben, mit // (eine Zeile) oder /* und */ (eine oder mehrere Zeilen) einfügen.

[13] Alternativ können Sie auch die Datei data_template.php verwenden, die auf der Webseite zum Download bereitssteht. Diese enthält die Grundstruktur, in die Sie nur noch die linguistischen Daten einsetzen müssen.

Stenozeichen

Um nun unsere Stenozeichen einzufügen, speichern wir diese direkt in die Variable $steno_tokens_master (beachten Sie, dass Variablen in PHP immer durch ein Dollarzeichen markiert werden). Die Variable $steno_tokens_master soll nun eine "Sammlung" von Zeichen sein (die ihrerseits eine "Sammlung" von Informationen sind, welche diese Zeichen definieren). Eine "Sammlung" von Daten in PHP nennt man "Array". Wenn ich in PHP also schreibe:

```
$meine_variable = array( "dies", "ist", "meine", "Sammlung" );
```

So enthält die Variable $meine_variable eine "Sammlung" mit 4 Werten (Wörtern): (1) Dies, (2) ist, (3) meine, (4) Sammlung. Beachten Sie bitte, dass in PHP eine Befehlszeile durch ein Semikolon (Strichpunkt) abgeschlossen wird.

Für unsere "Sammlung" an Stenozeichen können wir nun also schreiben:

```
$steno_tokens_master = array( "T", "B" );
```

Dadurch enthält unsere Sammlung an Stenozeichen zwei Zeichen: (1) T, (2) B.

Leider lässt sich damit noch nichts anfangen, denn unsere Zeichen sind ja noch nicht weiter definiert ... Deshalb fügen wir diese Definition nun ein, indem wir PHP wieder darauf hinweisen, dass die Definition eine "Sammlung" an weiteren Daten (in unserem Fall Header und Punktedaten) ist:

```
$steno_tokens_master = array(
      "T" => array( /* hier folgen die Daten des Zeichens "T" */ ),
      "B" => array( /* hier folgen die Daten des Zeichens "B" */ )
);
```

Wie Sie sehen, habe ich die Definition der Variable nun auf mehrere Zeilen aufgeteilt, weil dies übersichtlicher ist. Wichtig ist hier: (1) Verschiedene Elemente einer Sammlung (bzw. eines Arrays) müssen durch Kommas abgetrennt werden und (2) Wir verwenden hier das Symbol =>, welches unser Zeichen "T" zu einem so genannten Key definiert. Mit diesem Key kann VSTENO später ganz einfach auf die Daten innerhalb des Arrays zugreifen.

Bezüglich der Kommas: Ich kann es nicht genug betonen - vergessen Sie diese (und auch die Semikolons) nicht! Wenn in Ihrer Datei auch nur ein solches Zeichen fehlt, dann läuft das ganze Programm nicht (PHP ist da SEHR pingelig und wird Ihnen auch nicht unbedingt verraten, WO ein Komma fehlt). Weniger haarspalterisch ist PHP hingegen, falls Sie innerhalb von Arrays ein Komma zu viel schreiben:

```
$steno_tokens_master = array(
      "T" => array( /* hier folgen die Daten des Zeichens "T" */, ),
      "B" => array( /* hier folgen die Daten des Zeichens "B" */, ),
);
```

Die drei zusätzlichem Kommas in diesem File sind eigentlich überflüssig - PHP ignoriert diese aber. Das Problem ist meistens folgendes: Wenn Sie die erste Schreibweise verwenden und nachträglich noch eine zusätzliche Zeile einfügen, dann ist das Risiko gross, dass Sie das Komma vergessen:

```
$steno_tokens_master = array(
      “T” => array( /* hier folgen die Daten des Zeichens “T” */ ),
      “B” => array( /* am Ende dieser Zeile fehlt ein Komma */ )
      “K” => array( /* nach dieser Zeile braucht es kein Komma */ )
);
```

Im Zweifelsfall setzen Sie am Ende einer Sammlung von Elementen (in einem Array) somit lieber ein Komma, statt keines!

Nun brauchen wir anstelle der Kommentare nur noch echte Daten einzufügen. Wir verwenden hier - einmal mehr - die Zeichen “T” und “B”:

```
$steno_tokens_master = array(
      "T" => array( /*header*/ 0, 0.5, 0, 0,4,2.5, 0, "",
      /**/ "","","","", 0, 0, 0, 0,
      /**/ 0 , 0, 0, 0, 0, 0, 0, 0,
      /*data*/ 0, 20, 0, 1, 3.0, 0, 0, 0,
      /**/ 0, 0, 0, 0, 1.0, 0, 1, 0,
      /**/ 0, 2.5, 0, 4, 1.0, 0, 0, 0.5, ),
      "B"  => array( /*header*/ 5, 0.5, 0, 0, 1, 1, 0, "",
      /**/ "","","","", 0, 0, 0, 0,
      /**/  0, 0, 0, 0, 0, 0, 0, 0,
      /*data*/  0, 10, 0, 1, 3.0, 0, 0, 0.5,
      /**/  0,  2, 0.5, 0, 2.5, 0, 0, 0.5,
      /**/ 2.5,  0, 0.5, 4, 1.5, 0, 0, 0.5,
      /**/  2.5,  0, 0.5, 0, 1.5, 0, 2, 0.5,
      /**/ 5,  2,0.5, 0, 1.0, 0, 1,   0),
);
```

Ich brauche wohl an dieser Stelle nicht zu sagen, dass die Sache relativ schnell unübersichtlich wird (und gerade darum wird ein visueller Editor hier eine enorme Erleichterung bringen). Es hilft somit ungemein, die Daten beim Eingeben möglichst sauber zu strukturieren (indem man z.B. Dateneinheiten auf verschiedene Zeilen schreibt). Zusätzliche Zeilenumbrüche, Leerzeichen und Einrückungen (Tabulatoren) sind PHP egal - solange Sie die Klammern und Kommas nicht vergessen -, also machen Sie davon Gebrauch!

TokenCombiner

Genau gleich dazu können Sie nun den TokenCombiner definieren. Den eigentlichen Code hierfür haben wir ja schon früher präsentiert und fügen ihn hier nun noch einmal ein:

```
$steno_tokens_master = array(

      "T" => array( /*header*/ 0, 0.5, 0, 0,4,2.5, 0, "",
      /**/ "","","","", 0, 0, 0, 0,
      /**/ 0 , 0, 0, 0, 0, 0, 0, 0,
      /*data*/ 0, 20, 0, 1, 3.0, 0, 0, 0,
      /**/ 0, 0, 0, 0, 1.0, 0, 1, 0,
      /**/ 0, 2.5, 0, 4, 1.0, 0, 0, 0.5, ),
      "B"  => array( /*header*/ 5, 0.5, 0, 0, 1, 1, 0, "",
      /**/ "","","","", 0, 0, 0, 0,
      /**/  0, 0, 0, 0, 0, 0, 0, 0,
      /*data*/  0, 10, 0, 1, 3.0, 0, 0, 0.5,
      /**/  0,  2, 0.5, 0, 2.5, 0, 0, 0.5,
      /**/ 2.5,  0, 0.5, 4, 1.5, 0, 0, 0.5,
      /**/  2.5,  0, 0.5, 0, 1.5, 0, 2, 0.5,
      /**/ 5,  2,0.5, 0, 1.0, 0, 1,   0),
);
$combiner_table = array(
    array( "T", "@R", 0, 0 ),
    array( "D", "@R", 0, 0 )
);
```

Beginnen Sie also ausserhalb der Zeichendefinitionen eine neue Definition für die Variable $combiner_table. Der TokenCombiner verwendet somit diese Variable $combiner_table, um sämtliche darin stehenden Zeichen automatisch zu erzeugen (der Name muss genau so lauten, damit VSTENO die Daten findet).

TokenShifter

Genau gleich funktioniert dies mit dem TokenShifter:

```
$steno_tokens_master = array(

      "T" => array( /*header*/ 0, 0.5, 0, 0,4,2.5, 0, "",
      /**/ "","","","", 0, 0, 0, 0,
      /**/ 0 , 0, 0, 0, 0, 0, 0, 0,
      /*data*/ 0, 20, 0, 1, 3.0, 0, 0, 0,
      /**/ 0, 0, 0, 0, 1.0, 0, 1, 0,
      /**/ 0, 2.5, 0, 4, 1.0, 0, 0, 0.5, ),
      "B"  => array( /*header*/ 5, 0.5, 0, 0, 1, 1, 0, "",
      /**/ "","","","", 0, 0, 0, 0,
      /**/  0, 0, 0, 0, 0, 0, 0, 0,
      /*data*/  0, 10, 0, 1, 3.0, 0, 0, 0.5,
      /**/  0,  2, 0.5, 0, 2.5, 0, 0, 0.5,
      /**/ 2.5,  0, 0.5, 4, 1.5, 0, 0, 0.5,
      /**/  2.5,  0, 0.5, 0, 1.5, 0, 2, 0.5,
```

```
      /**/ 5,  2,0.5, 0, 1.0, 0, 1,   0),
);
$combiner_table = array(
    array( "T", "@R", 0, 0 ),
    array( "D", "@R", 0, 0 )
);
$shifter_table = array(
    array("N", "&TN", 4, 15, 0, 1.5),
    array("NS", "&TENS", 4, 10, 0, 1),
    array("NS", "&EITENS", 18, 10, 0, 1)
);
```

Auch hier wird die Variable $shifter_table vom TokenShifter dazu verwendet, automatisch sämtliche darin enthaltenen Zeichen zu generieren!

Regeln

Nach den Zeichendefinitionen kommen wir nun zu den so genannten Regeln, welche einzelne Wörter der Langschrift Schritt um Schritt so umwandeln, dass sie am Schluss durch Aneinanderfügen der definierten Zeichen als Stenogramme dargestellt werden können. Diese "Übertragung" findet in mehreren Schritten statt und um diese sichtbar zu machen, können Sie in der Demoversion die Funktion "Debug" anwählen. Für das Wort "baten" sehen Sie dann:

```
ORIGINAL: baten
[1] WORD: bat{EN} FROM: rule: (?<!(^[Ww])|i)en$ => {EN}
[2] WORD: bAt{EN} FROM: rule: a => A
[3] WORD: BAt{EN} FROM: rule: b => B
[4] WORD: BAT{EN} FROM: rule: t => T
NUMBER OF RULES APPLIED: 4
```

Beim Wort "baten" wird also zuerst die Endkürzung -en erkannt und durch {EN} markiert (Schritt 1), danach wird der Vokal a erkannt und mit dem Grossbuchstaben A markiert (Schritt 2). Analog dazu werden auch die Konsonanten b und t erkannt und mit den Grossbuchstaben B und T markiert (Schritte 3+4). Die so entstanden Folge - die VSTENO intern als TokenList (Zeichenliste) bezeichnet - kann danach sehr simpel zu einem Stenogramm verarbeitet werden, indem die Zeichendefinitionen aus der Variable $steno_tokens_master ausgelesen und die einzelnen Zeichen aneinandergefügt werden.

Das obige Beispiel ist natürlich relativ simpel, da alle Zeichen ohne grosse Veränderungen verwendet werden können. Gewisse Zeichen müssen aber je nach Kontext anders geschrieben werden. So z.B. das Aufstrich-T, wie es in "bunt" vorkommt:

```
ORIGINAL: bunt
```

```
[1] WORD: bun[&T] FROM: rule: ([bcdfghjklmnpqrvwxyz])t => $1[&T]
[2] WORD: bUn[&T] FROM: rule: u => U
[3] WORD: BUn[&T] FROM: rule: b => B
[4] WORD: BUN[&T] FROM: rule: n => N
NUMBER OF RULES APPLIED: 4
```

Alle diese Dinge müssen VSTENO anhand von Regeln genauestens "erklärt"[14] bzw. beigebracht werden. Damit VSTENO die Anweisungen versteht, müssen müssen sie in einer klar definierten Formelsprache abgefasst werden. Im Falle von VSTENO ist dies REGEX, eine Formelsprache die standardmässig in PHP integriert ist. Regex ist ein sehr mächtiges Instrument, das einige Tücken aufweist ... das aber - richtig angewandt - sämtlichen linguistischen Bedürfnissen gerecht werden kann.

Beachten Sie bitte, dass Regeln nach dem Format "Wenn A, dann B" funktionieren. In den obigen Beispielen bedeutet "b => B" also: "Wenn du innerhalb des Wortes den Kleinbuchstaben b findest, dann erstetze ihn durch den Grossbuchstaben B".

REGEX

Die Möglichkeiten von REGEX (Abkürzung für so genannte "regular expressions") auch nur ansatzweise darzustellen, würde den Rahmen dieses Tutorial sprengen - schliesslich gibt es ganze Bücher, die sich ausschliesslich mit REGEX beschäftigen! Wir werden uns also damit begnügnen, hier nur einige wesentliche Elemente zu erklären. Für den Rest verweisen wir Sie auf die folgenden Seiten, die Ihnen weiterhelfen können:

Ein guter Start, um einen Überblick zu REGEX zu erhalten, sind die beiden Wikipedia-Seiten auf Deutsch und auf Englisch:

- Deutsch: https://de.wikipedia.org/wiki/Regul%C3%A4rer_Ausdruck
- Englisch: https://en.wikipedia.org/wiki/Regular_expression

Ebenfalls sehr zu empfehlen ist der folgende REGEX-Tester:

- Online-REGEX-Tester: https://regex101.com/

Er erlaubt es Ihnen, einzelne REGEX-Ausdrücke direkt im Webbrowser zu testen und so Fehler in bzw. falsch forumlierte Formeln zu finden und zu korrigieren.

REGEX kann man durchaus als "learning by doing" lernen: Spielen Sie also mit den Elementen die wir im Folgenden vorstellen im REGEX-Tester etwas herum und Sie werden bald ein relativ intuitives Verständnis dafür erlangen, wie REGEX funktioniert!

[14] Wenn man dieses Verb im Zusammenhang mit einem Computer verwenden kann ... ;-) Aber in gewissem Sinn ist die Analogie triftig: Der Computer bzw. das Programm VSTENO ist unser Schüler, dem wir als Linguisten Stenografie beibringen (und "erklären").

Wortgrenzen und Lookaround-Expressions

Hier nun also einige wenige Grundregeln und Prinzipien von REGEX, die Sie für linguistische Regeln mit Sicherheit benötigen werden. Beginnen wir mit den Zeichen ^ und $. Sie markieren den Anfang und das Ende eines Wortes:

```
“^hab$” => “{HAB}”
```

Bedeutet also: Falls das Wort genau der Zeichenfolge “hab” entspricht (bzw. sich vor “hab” der Wortanfang und nach “hab” das Wortende befindet), dann ersetze die Zeichenfolge “hab” durch “{HAB}”. Beachten Sie, dass die Ausdrücke ^hab$ und {HAB} innerhalb von Anführungszeichen “” gesetzt werden müssen, da es sich dabei um Strings (Zeichenfolgen) handelt. Diese Regel kann also verwendet werden, um die Kürzung HAB zu definieren. Sie ist allerdings schlecht formuliert: Sie berücksichtigt z.B. nicht, dass das Verb “haben” auch am Wortanfang stehen kann (was bedeutet, dass “hab” dann mit einem Grossbuchstaben beginnt). Ausserdem wird die Kürzung HAB auch in längeren Wörtern angewendet (wie z.B. “Inhaber”, “habt”, “haben” etc.). Auch diese Fälle würden nicht berücksichtigt, da die Regel verlangt, dass das Wort nach “hab” zu Ende ist. Wir formulieren die Regel deshalb um:

```
“[Hh]ab” => “{HAB}”
```

Die eckige Klammer [] bedeutet hier: “einer der aufgelisteten Buchstaben” (die Liste kann beliebig lang sein - z.B. [aeiou] für die 5 Grundvokale - oder man kann auch weitere Ausdrücke verwenden wie z.B. [a-z] oder [A-Z] für Kleinbuchstaben und Grossbuchstaben, [0-9] für Ziffern etc.). Da wir die Zeichen ^ und $ (Wortanfang und Wortende) entfernt haben, trifft diese Regel nun auf die oben genannten Fälle zu: “Inhaber”, “habt” und “haben” werden zum Beispiel in “In{HAB}er”, “{HAB}t” und “{HAB}en” umgewandelt. Allerdings ist auch diese Regel zu ungenau formuliert, da auch Wörter wie “schaben”, “Haber”[15] in “sc{HAB}en” und “{HAB}er” umgewandelt werden, was wiederum falsch ist ...

Dies können wir korrigieren, indem wir eine so genannte Lookaround-Expression (zu Deutsch etwa: Schau-dich-um-Ausdruck) verwenden. Es gibt zwei Arten von Lookaround-Expressions: die Lookbehind-Expression (sie sucht nach Zeichenfolgen vor dem Wort) und die Lookahead-Expression (für Zeichenfolgen nach dem Wort). Ausserdem können Lookaround-Expressions positiv (= die Zeichenfolge muss vorkommen) oder negativ (= die Zeichenfolge darf nicht vorkommen) sein. Im Folgenden verwenden wir eine negative Lookbehind-Expression. Diese steht in einer runden Klammer vor dem Wort und beginnt mit ?<!:

```
“(?<![Ss]c)[Hh]ab” => “{HAB}”
```

Diese Regel bedeutet also: Ersetze “hab” - egal ob mit Gross- oder Kleinbuchstaben beginnend - an einer beliebigen Stelle des Wortes durch {HAB}, sofern nicht “sc” oder “Sc” vorausgeht. Oder anders formuliert: Ersetze “hab” nur

[15] Wir meinen hier das umganssprachliche Wort für “Hafer”.

dann, wenn das H nicht Teil von SCH ist. Diese Regel löst also das Problem von "schaben", nicht aber jenes von "Haber". Wir könnten natürlich auch hier eine negative Lookahead-Expression verwenden:

```
“[Hh]ab(?=!er)” => “{HAB}”
```

Allerdings würde dann das Wort "Inhaber" ebenfalls nicht mehr in "In{HAB}er" umgewandelt.

Man sieht also: Das Formulieren von präzisen linguitischen Regeln - die wirklich nur auf jene Wörter angewandt wird, auf die die Regel wirklich zutrifft - ist zum Teil recht anspruchsvoll. Es lohnt sich aber, hier Zeit zu investieren, denn je präziser die Regel ist, umso weniger "Ausnahmen" müssen später zusätzlich (in einem Wörterbuch beispielsweise) definiert werden.

Klammern, Quantoren und Variablen

Quantoren sind Zeichen, welche angeben, wie oft ein Zeichen vorkommen muss. REGEX kennt hier z.B. die Zeichen ? (= 0 oder 1 Mal vorkommend), + (= mindestens 1 Mal vorkommen), * beliebig oft vorkommend. Diese können auf einzelne Zeichen angewandt werden oder es können mehrere Zeichen zu einer Gruppe zusammengefasst werden, für die der Quantor gilt. Als Beispiel betrachten wir folgende Regel:

```
“^([Uu]n)?[Zz]uver” => “$1{ZU}{VER}”
```

Hier wurde also der Ausdruck [Uu]n durch die Klammern () zu einer Gruppe zusammengefasst. Der Quantor ? nach ([Uu]n) bedeutet, dass die Vorsilbe un- (die am Anfang des Wortes - markiert durch das Zeichen ^ - gross oder klein geschrieben werden kann) 0 oder 1 Mal vorkommen kann. Dieses "Muster" (oder pattern, wie es auf Englisch genannt wird) trifft z.B. auf Wörter wie "unzuverlässig", "Unzuverlässigkeit" zu (hier kommt un- 1 Mal vor), aber auch auf Wörter wie "zuverlässig", "Zuversichtlichkeit" usw.

Auf der rechten Seite (der Folge-Seite der Regel) steht der Ausdruck $1 für die "Variable in der ersten Position". Mit Position ist gemeint: "Der x-te Klammerausdruck von links beginnend", in unserem Fall also ([Uu]n).

Diese Regel nimmt somit folgende Ersetzungen vor: "unzuverlässig" => "un{ZU}{VER}lässig", "Unzuverlässigkeit" => "Un{ZU}{VER}lässigkeit", "Zuversicht" => {ZU}{VER}sicht, "zuverlässig" => "{ZU}{VER}lässig. Beachten Sie hierbei, dass die Variable $1 den Klammerausdruck exakt so wiedergibt, wie er im Wort vorgefunden wird, als "Un" (mit grossem Anfangsbuchstaben) im Substantiv "Unzuverlässigkeit" und "un" (mit Kleinbuchstaben) im Adjektiv "unzuverlässig". Nicht so hingegen beim Ausdruck "[Zz]u" (wo wir keine Variable verwenden): Hier wird also sowohl "Zu" und "zu" durch "{ZU}" ersetzt.

Da Variablen nummeriert sind, kann man auch mehrere hintereinander verwenden:

```
“^([Uu]n)?([Zz]u)ver” => “$1$2{VER}”
```

In dieser Regel z.B. wird nur das Präfix ver- durch {VER} ersetzt. Wie bei unweiter oben wird nun auch in diesem Beispiel die zweite Vorsilbe "zu" exakt so übertragen, wie sie im Wort steht (also "Zu" bei einem Substantiv, "zu" bei klein geschriebenen Wörtern).

Im Unterschied zu ? bezeichnet der Quantor + einen Ausdruck der mindestens 1 Mal (oder mehr) vorkommen muss. Die Regel

```
“^([Zz]u)+” => “$1|”
```

Trennt eine oder mehrere Vorsilben zu- durch einen vertikalen Strich vom Rest des Wortes ab: "zugeben" => "zu|geben", "zuzugeben" => "zuzu|geben". Bitte beachten Sie, dass dies wiederum eine linguistisch zu unpräzis formulierte Regel ist, da sie auch auf Wörter wie "Zuzug" zutrifft (hier findet die Regel 2 Vorsilben zu-, dabei enthält das Wort nur 1 Vorsilbe - der zweite Teil gehört zum Stamm des Wortes). Auch trifft die Regel theoretisch auf inexistente Wörter zu wie "zuzuzukumi" (hier fände die Regel 3 Vorsilben).

Wildcards und Greediness

REGEX definiert den Punkt . als so genannte Wildcard[16]: Dieses Zeichen kann also für "irgend ein" Zeichen stehen. Auch den Punkt können wir mit Quantoren kombinieren, wobei hier ein neuer Aspekt ins Spiel kommt: die so genannte greediness (zu Deutsch in etwa: Gefrässigkeit). Quantoren können also "greedy" (gefrässig) oder non-greedy (ungefrässig oder genügsam) sein. Im ersten Fall versucht REGEX den Ausdruck auf die grösstmögliche Zeichenfolge anzuwenden, im letzten Fall hingegen wird der kürzeste Ausdruck gesucht, der auf das Muster zutrifft.

```
[1] Bett(.*)en => Bett$1{EN}
[2] Bett(.*?)en => Bett$1{EN}
```

Im Wort "Bettenkapazitäten" (welches 2 x die Endung -en enthält), findet die erste Regel (mit dem "gefrässigen" Ausdruck *) die LETZTE Endung und ersetzt das Wort somit durch "Bettenkapazität{EN}". In der zweiten Regel (mit dem "genügsamen" Ausdruck *?) findet REGEX hingegen der ERSTEN Ausdruck, sodass das Resultat hier "Bett{EN}kapazitäten" lautet.

Passen Sie deshalb mit Quantoren besonders auf: Sie können eine total andere Bedeutung haben als man - von einer intuitiven, sprachlichen Logik her denkend - glauben würde ...

Logisches Oder

Wir haben bereits die eckigen Klammern [] kennengelernt, welche eine Reihe von Zeichen zusammenfasst, von denen wenigtens eines zutreffen muss. In der Logik entspricht dieses einem Oder: Im Ausdruck [abcd] muss also entweder a

[16] Wir auch Joker genannt.

oder b oder c oder d vorkommen. Ein weiterer Ausdruck um ein logisches Oder wiederzugeben ist der vertikale Strich:

```
“(ck|kk)” => “[CK]”
```

Diese Regel bedeutet also, dass “Zucker” zu “Zu[CK]er” und “Mokka” zu “Mo[CK]a” wird. Beachten Sie in diesem Zusammenhang unbedingt, dass die eckige Klammer auf der linken Seite (Bedingung) der Regel nicht die gleiche Bedeutung hat wie auf der rechten Seite (Folge):

```
“[Ää]” => “[Ä]”
“(Ä|ä)” => “[Ä]”
```

Diese beiden Regeln sind gleichbedeutend![17] Die eckige Klammer auf der linken Seite bedeutet also wie erwähnt “eines dieser Zeichen” (also Ä oder ä) auf der rechten Seite bedeutet es jedoch “ersetzen durch das Klammerzeichen” (offen oder geschlossen). Diese Regel transformiert die Wörter “Kläger” und “KlÄger” zu “Kl[Ä]ger”!

Wenn Sie auf der linken Seite nach einer eckigen Klammer suchen möchten, dann müssen Sie das Zeichen “escapen” (wir übersetzen mal frei: “seiner REGEX-Bedeutung entheben”). Dies geschieht indem der Backslash \ vorangestellt wird:

```
“\[Ää\]” => “{Ää}”
```

Diese Regel würde die Zeichenfolge “ABC[Ää]XYZ” zu “ABC{Ää}XYZ” umschreiben!

Auch hier: Passen Sie bei Zeichen, die in REGEX eine spezielle Bedeutung haben, sehr gut auf, ob Sie sie wörtlich (als Literal oder Buchstabe) oder in der Bedeutung von Regex verwenden wollen. Dies trifft prinzipiell auf alle Zeichen zu, die in REGEX eine spezielle Bedeutung haben. Sicherheitshalber listen wir die wichtigsten im Folgenden noch einmal auf.

Escaping

Folgende Zeichen MÜSSEN in PHP-Regex “escaped” werden, wenn Sie “wörtlich” (= als genau dieses Zeichen) verwendet werden sollen:

```
. als \.
^ als \^
$ als \$
* als \*
+ als \+
? als \?
```

[17] ... theoretisch. Aus irgendeinem unerfindlichen Grund - vermutlich hat es mit dem Sonderzeichen Ä zu tun - funktioniert, zumindest in meiner Systemkonfiguration, nur die Variante (Ä|ä). Dasselbe gilt auch für Üü, Öö, ÄU ... Ich empfehle deshalb, für diese Sonderzeichen, die Variante in runden Klammern zu verwenden.

```
( als \(
) als \)
[ als \[
\ als \\
| als \|
```

Die geschwungenen Klammern {} KÖNNEN in PHP-REGEX "escaped" werden (oder nicht)[18]. Speziell weisen wir auch auf das Zeichen ^ hin. Es hat innerhalb der eckigen Klammern die Bedeutung von "nicht" (logische Negation):

```
"^[Zz]u([^m])" => "{ZU}$1"
```

Dieses Regelbeispiel bedeutet also: Ersetze die Vorsilbe zu- am Anfang eines Wortes durch {ZU}, sofern das darauffolgende Zeichen kein m ist (in Stolze-Schrey wird in diesem Fall keine Kürzung verwendet). Beachten Sie, dass hier die beiden Zeichen ^ verschiedene Bedeutungen haben: Das erste bedeutet "Wortanfang", das zweite bedeutet "nicht m".

Wenn Sie das Zeichen ^wortwörtlich (also als Zeichen) suchen möchten, müssen Sie es "escapen":

```
"ABC\^XYZ" => "ABC{circonflex}XYZ"
```

Wandelt die Zeichenkette "ABC^XYZ" in "ABC{circonflex}XYZ" um!

Es kann nicht genug betont werden, wie wichtig diese unscheinbaren Unterschiede sind: REGEX ist deshalb so "kompliziert", "tricky", "kryptisch" - oder wie immer Sie es bezeichnen möchten - weil ein einziges Zeichen je nach Art und Weise, wie es verwendet wird, eine total andere Bedeutung haben kann! Verwenden Sie in diesem Fall den erwähnten REGEX-Tester und vergewissern Sie sich, dass Ihre Regel auch wirklich das heisst, was Sie meinen ...

Regeln in VSTENO

Die Regeln, die wir uns bis jetzt angesehen haben, entsprachen dem Standard-Schema der REGEX-Replace-Funktion, nämlich: Wenn A, dann (ersetze durch) B. VSTENO erweitert den Regelformalismus um eine weitere Möglichkeit: Wenn A, dann B, ausser C (oder D):

```
A => B                          normale REGEX-Regel
A => { B, C, D ..., X }         erweiterte Regel
```

In PHP wird dieser Regeltyp umgesetzt, indem anstelle der Folge (B) ein Array (also eine Liste an Werten bzw. Wörtern) gesetzt wird. Wir werden später noch genauer auf die Notation in PHP eingehen, aber hier vorweg schon einmal ein Beispiel als PHP-Code. Wir verwenden noch einmal die Kürzung "HAB", die wir bereits weiter oben gesehen haben:

[18] Ich empfehle sie nicht zu escapen, da die Regeln dadurch etwas übersichtlicher werden.

```
“(?<![Ss]c)[Hh]ab” => array( “{HAB}”, “^Haber” )
```

Diese Regel bedeutet also: Ersetz die Zeichenfolge “hab” (oder “Hab”) an einer beliebigen Stelle des Wortes durch “{HAB}”, AUSSER das Muster “Haber” (mit Grossbuchstaben) steht am Anfang des Wortes. Diese Regel nimmt in den folgenden Beispielwörtern folgende Transformationen vor:

```
Inhaber => In{HAB}er
haben => {HAB}en
habt => {HAB}t
schaben => schaben
Haber => Haber
Habermacher => Habermacher
```

Beachten Sie, dass das zweite Element des Folgearrays “^Haber” von VSTENO ebenfalls als REGEX-Muster interpretiert wird, weshalb es auch auf “Habermacher” zutrifft (sonst müsste dort “^Haber$” stehen).

Auch diese Kürzungsregel für “HAB” ist nicht perfekt, da z.B. das Wort “Habicht” ebenfalls zu “{HAB}icht” gekürzt wird. Mit dem erweiterten Regelformalismus von VSTENO können wir weitere Ausnahmen aber sehr einfach im Folge-Array hinzufügen:

```
“(?<![Ss]c)[Hh]ab” => array( “{HAB}”, “^Haber”, “Habicht” )
```

Auch hier hat die REGEX-Notation wieder den Vorteil, dass sämtliche Fälle (z.B. Genetiv “Habichts”) erfasst werden.

PHP-Code

Nach all diesen Ausführungen zur Regelnotation widmen wir uns nun - analog zum Kapitel Zeichen - der Implementierung der Regeln in PHP. Genau wie die Zeichen müssen die Regeln innerhalb der Datei data.php definiert werden. Bevor wir jedoch die entsprechenden PHP-Variablen setzen können, müssen wir wissen, wie der Parser funktioniert. Der Parser ist nämlich jener Teil, welcher die Regeln liest und verarbeitet. Um die Sache einigermassen übersichtlich zu halten, wurde der Pars-Prozess in verschiedene, linguistisch sinnvolle Unterschritte unterteilt.

Funktionsweise des Parsers

Um zu verstehen, wie der Parser funktioniert, betrachten wir die Debug-Ausgabe des Wortes “kannten”:

```
ORIGINAL: kannten
[1] WORD: kannt{EN} FROM: rule: (?<!(^[Ww])|i)en$ => {EN}
[2] WORD: ka[NN]t{EN} FROM: rule: nn => [NN]
[3] WORD: ka[NN][&T]{EN} FROM: rule: ([BCDFGHJKLMNPQRVWXYZS])\]t => $1][&T]
```

```
[4] WORD: ka[NN][&T][&E][EN] FROM: rule: \[&T\]{EN}$ => [&T][&E][EN]
[5] WORD: kA[NN][&T][&E][EN] FROM: rule: a => A
[6] WORD: KA[NN][&T][&E][EN] FROM: rule: k => K
[7] WORD: KA[NN][&T&E][EN] FROM: rule: \[&T\]\[?&E\]? => [&T&E]
NUMBER OF RULES APPLIED: 7
```

Zunächst einmal halten wir fest, dass wir immer von einem Ursprungswort ausgehen (hier: "kannten"), das VSTENO dem Parser "übergibt". VSTENO nimmt diese Wörter - eins nach dem anderen - aus dem Text, den wir via Webformular eingeben.

Anschliessend beginnt der Parser, ausgehend vom Ursprungswort, die verschiedenen Regeln anzuwenden. Jedes Mal, wenn eine Regel zutrifft, modifiziert der Parser das Wort gemäss der Regel und verwendet danach das Resultat als neues Wort.

Sobald der Parser alle Regeln abgearbeitet hat, nimmt er das letzte Resultat (das wir als PrintShorthand oder "DruckSteno" bezeichnen) - in unserem Fall "KA[NN][&T&E][EN]" - und sendet dieses der TokenEngine ("Zeichenmaschine").

Die TokenEngine wiederum tut nichts anderes, als links (also mit dem Zeichen "K") beginnend für jedes einzelne Zeichen die entsprechenden grafischen Vektordaten einzusetzen und diese zu verbinden. Die TokenEngine kennt dabei nur zwei Arten von Zeichen: einzelne Zeichen (die aus nur 1 Buchstaben bestehen) und so genannte bundled (gebündelte) Zeichen (die aus mehreren Buchstaben bestehen und in eckigen oder geschweiften Klammern stehen). In unserem Beispiel enhält das Wort "KA[NN][&T&E][EN]" 5 Stenozeichen: K-A-NN-&T&E-EN.

Die meisten Zeichen der DruckSteno-Form (z.B. K, A und NN) sind sicher intuitiv verständlich. Die Zeichen &T&E und EN sind vielleicht etwas ungewohnter und bedeuten "Aufstrich-T mit Häkchen" und "Endkürzung EN"[19].

Unterteilung des Parsers

Wenn wir uns den Weg vom Ursprungswort zur DruckSteno-Form im obigen Beispiel anschauen, dann scheint es, als würden die 7 Regeln alle aus dem gleichen Regel-Pool stammen. Dem ist aber nicht so: Wie bereits erwähnt, unterteilt VSTENO den Pars-Prozess in verschiedene Unterschritte, die linguistisch logischen Operationen entsprechen. Chronologisch durchläuft ein Wort parserintern die folgenden Schritte:

1. Filter & Globalizer: In diesem Teilen können Sonderzeichen umgeschrieben werden (z.B. können hier verschiedene Arten von Anführungszeichen oder

[19] Im Grunde ist es egal, wie Sie Ihre Stenozeichen "taufen": einem "Aufstrich-T mit Häkchen" könnte man auch den Namen "/T&E" oder "/E" (oder was auch immer) geben. Wichtig ist nur, dass das Endresultat des Parsers (also die DruckSteno-Form) Zeichennamen enthält, die Sie tatsächlich (und in dieser Form) definiert haben! Im vorliegenden Beispiel wurde das Zeichen &T&E mit dem TokenShifter aus dem Zeichen &E (Häkchen auf Grundstufe) erstellt.

Umlaute so umgeschrieben werden, dass sie Zeichen in der TokenEngine entsprechen). In unserem Beispiel kommen Globalizer und Filter nicht zum Zug.

2. Shortener: In diesem Teil, werden Kürzungen und Präfixe erkannt und markiert. Im obigen Beispiel stammt die erste Transformation (Endkürzung -en) aus dem Shortener. Beachten Sie, dass wir Kürzungen innerhalb geschweifter Klammern markieren.

3. Normalizer: Hier werden ortografische Vereinfachungen vorgenommen (z.B. th => t, ie => i, ah => a etc.). In unserem Beispiel kommt der Normalizer nicht zum Zug.

4. Bundler: Hier werden Stenozeichen, die aus mehreren Buchstaben in der Langschrift bestehen erkannt und innerhalb eckiger Klammern "gebündelt". Im obigen Beispiel trifft dies auf die zweite Transformation (nn => [NN]) zu.

5. Transcriptor: Hier alle Stenozeichen so umgeschrieben, dass Sie direkt von der TokenEngine verarbeitet werden können. Im obigen Beispiel sind dies die Schritte 3-7. Beachten Sie, dass die Zeichen der TokenEngine Grossbuchstaben verwenden. In den meisten Fällen werden deshalb Kleinbuchstaben in Grossbuchstaben umgeschrieben (a => A, k => K). Der Transcriptor erkennt hier aber auch, dass das T nach einem Doppel-N als Aufstrich-T geschrieben werden muss und dass dieses Aufstrich-T zusätzlich ein Häkchen benötigt, weil daran die Endung -en angeschlossen wird. Beachten Sie, dass der Transkriptor hier die geschweiften Klammern der Kürzung in eckige Klammern umschreibt.

6. Substituter: Ersetzt Kürzungen (in geschweifter Klammer) durch entsprechende Stenozeichen (in eckiger Klammer, falls nötig). In unserem Beispiel kommt der Substituter nicht zum Zug, da die Endung -en bereits durch den Transcriptor umgeschrieben wurde.

Da der Parser diese Teile einzeln nacheinander wie die Glieder einer Kette durchläuft, nennen wir diese Abfolge auch ParserChain (ParserKette). Wichtig ist nun zu wissen, dass jedem dieser Teile eine PHP-Variable mit dem Beinamen "table" (Tabelle) entspricht. Die Variablen heissen also:

```
Globalizer = $globalizer_table
Filter = $filter_table
Shortener = $shortener_table
Normalizer = $normalizer_table
Bundler = $bundler_tabler
Transcriptor = $transcriptor_table
Substituter = $substituter_table
```

In diese "Tables" werden wir jetzt dann gleich unsere Regeln füllen. Aber zunächst müssen wir noch zwei, drei weitere Besonderheiten des Parsers erklären.

Gross- und Kleinbuchstaben

Dieses Thema sind wir bis jetzt stillschweigend übergangen bzw. haben es nur hie und da kurz angedeutet. Es geht um die Frage, wie Klein- und Grossbuchstaben in VSTENO verwendet werden.

Tatsache ist, dass Gross- und Kleinschreibung durchaus einen Unterschied machen können: das Substantiv "Waren" und das Verb "waren" werden in Stolze-Schrey z.B. völlig unterschiedlich geschrieben. Dies ist der Grund, warum VSTENO zu Beginn der ParserChain die Gross- und Kleinschreibung beibehält - und zwar bis und mit Shortener. Wenn Sie also im Shortener die Regel:

```
"war" => "{WAR}"
```

definieren, so wird diese nur auf das Verb "waren" (ergibt "{WAR}en") nicht aber auf "Waren" (bleibt "Waren") angewendet[20]. Da es bei den meisten Wörtern jedoch keinen Unterschied macht, ob sie gross oder klein geschrieben werden - und es mühsam wäre, in den Regeln jedesmal sämtliche Varianten in eckigen Klammern anzugeben - wandelt VSTENO nach dem Shortener sämtliche Gross- in Kleinbuchstaben um! Alle nachfolgenden Schritte (Normalizer, Bundler, Transcriptor, Substituter) arbeiten somit mit Kleinbuchstaben. Ausgenommen von der Umwandlung in Kleinbuchstaben sind allerdings Veränderungen, die der Shortener selbst vornimmt: Wenn der Shortener im Wort "wahrhaftig" also die Kürzung -haft zu "wahr{HAFT}ig" ersetzt, so verbleibt der Teil "{HAFT}" auch nach dem Shortener in Grossbuchstaben.[21]

Die konsequente Kleinschreibung - mit Ausnahme der vom Shortener eingesetzten Teile, wie gesagt - hat den Vorteil, dass wir Grossbuchstaben nun dazu benutzen können, jene Teile des Wortes zu markieren, die von bestimmten Regeln bereits umgeschrieben wurden. Wenn also der Shortener im Wort "Andenken" das Prefix an- erkennt und gross markiert (also "ANdenken"), dann "weiss" der Bundler später, dass er "gross N" und "klein d" nicht zum Zeichen [ND] zusammenfassen darf. Die Regel für den Bundler lautet nämlich:

```
"nd" => "[ND]"
```

Der Bundler soll also nur die unbearbeitete (d.h. klein geschriebene) Zeichenfolge "nd" bündeln.

Dictionary und Trickster

Zusätzlich zu den bereits vorgestellten Parserteilen gibt es noch einen Wörterbuch und einen sogenannten "Trickster". Beide verwenden entsprechende Tables als PHP-Variablen:

[20] Eine andere Sache ist natürlich, wenn das Verb "Waren" dummerweise am Satzanfang steht und ebenfalls gross geschrieben wird ... ;-)

[21] Wir weisen hier - en passant - noch auf einen weiteren Aspekt hin: Wichtig ist in diesem Beispiel auch, dass der Normalizer (der den Vokal "ah" zu "a" umschreibt) erst NACH dem Shortener abgearbeitet wird, da sonst die Kürzung "war" auch auf das Wort "wa(h)rhaftig" angewandt würde.

```
Dictionary = $dictionary_table
Trickster = $trickster_table
```

Um es gleich zu sagen: Sie sollten diese Tables im Prinzip nicht verwenden! Der Grund hierfür: Die $dictionary_table wird in künftigen Versionen verschwinden und durch eine Datenbank ersetzt. Und was den Trickster anbelangt: Er ist innerhalb eines geordneten Systems - wie es linguistische Regeln sein sollten - ein etwas ungemütlicher und fast furchteinflössender Zeitgenosse, weil er nämlich die Macht hat, gewisse Regeln zu überlisten ...

Da beide jedoch (noch) Teil der jetzigen Version sind - und man (solange noch keine Datenbankfunktionen bestehen) vielleicht doch Bedarf hat, eine Art Wörterbuch zur Verfügung zu haben, werde ich die Funktionen kurz dokumentieren:

- Dicionary: Ist im Grunde ein Array mit Einträgen, wobei jeder Eintrag zwei Informationen enthält: Das Ursprungswort (wie es dem Parser von VSTENO übergeben wird) und das Endresultat (also die PrintSteno-Form, die von der TokenEngine direkt verarbeitet werden kann). Der Dictionary ist der ParserChain vorgeschaltet und unterscheidet zwischen Gross- und Kleinbuchstaben. Findet VSTENO im Dictionary einen Eintrag, der dem Ursprungswort entspricht, dann verzichtet das Programm auf die Abarbeitung der ParserChain. Stattdessen wird die DruckSteno-Form aus dem Wörterbuch direkt an die TokenEngine geschickt. Ebenfalls wichtig: Der Wörterbucheintrag muss exakt mit dem Ursprungswort übereinstimmen: Enthält das Wörterbuch z.B. einen Eintrag für das Wort "Haus", dann gilt dieser nur für "Haus" (und nicht für andere Forme des gleichen Wortes wie "Hauses", "Hause" etc.). Die Einträge im Wörterbuch werden also nicht als REGEX-Muster behandelt!

- Trickster: Ein unangenehmer Zeitgenosse, wie gesagt, der sich an keine Regeln hält ... Wenn Sie z.B. feststellen, dass ein Wort von den Regeln falsch umgeschrieben wird (sagen wir z.B. die Kürzung "ET" wird versehentlich auf das Wort "beten" angewandt), dann können sie den Trickster dazu veranlassen, z.B. den Vokal E in "beten" gross zu schreiben. Dies wiederum bewirkt, dass der Shortener die Kürzung "ET" nicht anwenden wird, weil das E als "bereits bearbeitet" markiert ist (auch der Trickster umgeht also - wie der Shortener - die Umwandlung in Kleinbuchstaben). Sagen wir es so: Ich dachte anfänglich, der Trickster sei ein spannende und sogar brauchbare Idee ... Aber leider bringt er sehr viel Unordnung in die Regeln, was schliesslich der Grund war, den Regel-Formalismus um die Form "Wenn A, dann B - AUSSER ..." zu erweitern. Mit andern Worten: Es ist besser die Ausnahme von der Regel gleich bei der Regel zu notieren als "irgendwo anders" (i.e. im Trickster), wo sie dann ihr Unwesen treibt ... Letzte Anmerkung: Es ist geplant, den Parser komplett umzuschreiben - und ich denke nicht, dass ich dem Trickster in der neuen Version eine Daseinsberechtigung einräumen werde ... ;-)

Programmcode

Damit sind wir nun aber definitiv reif für den letzten Schritt: Das Schreiben der Regeln als PHP-Code. Da der Parser in "linguistische Unterschritte" unterteilt ist, beginnen Sie am besten mit dem Shortener (also den Kürzungen). Öffnen Sie also die Datei data.php und setzen sie dort die Variable $shortener_table. Wie bei den Zeichen ist diese Variable eine "Sammlung" (Array) an Regeln[22]:

```
$shortener_table = array(
      "war" => "{WAR}",
      "hab" => "{HAB}",
      "vielleicht" => "{VIELLEICHT}"
);
```

Beachten Sie - wie bei den Zeichendefinitionen - dass die Variablendefinition mit einem Strichpunkt (Semikolon) abgeschlossen und die einzelnen Regeln durch Kommas abgetrennt werden müssen[23]. Wichtig: Alle "Kürzungen", die sie hier verwenden, müssen entweder als einzelne Stenozeichen definiert sein (z.B. kann in der Grundschrift Stolze-Schrey das lange Wort "vielleicht" tatsächlich durch ein einzelnes Zeichen wiedergegeben werden) oder die markierten Kürzungen müssen später vom Substituter so umgeschrieben werden, dass es sich um Zeichen handelt, die von der TokenEngine verarbeitet werden können.

Nach den Kürzungen werden Sie vermutlich den Normalizer und dann den Bundler definieren. Hier ein paar Beispiele für den Bundler:

```
$bundler_table = array(
  "nn" => "[NN]",
  "nk" => "[NK]",
  "sp" => "[SP]"
);
```

Später geht es dann zum Transcriptor:

```
$transcriptor_table = array(
   "a" => "A",
   "b" => "B",
   "m" => "M"
);
```

Und schliesslich zum Substituter:

```
$substituer_table = array(
   "{MIT}" => "M",
   "{ZU}" => "Z",
```

[22] Die Regeln sind natürlich stark vereinfacht und können so in der Praxis nicht verwendet werden, es geht hier aber nur darum zu zeigen, wie die Regeln als Array notiert werden.

[23] Auch hier gilt: Im Zweifelsfall lieber ein Komma zu viel - weitere Details hierzu finden Sie bei den Zeichendefinitionen

```
    "{HAB}" => "[HAB]",
    "{VIELLEICHT} => "[VIELLEICHT]",
    "{HIN}" => "HN"
);
```

Wie bereits erwähnt können Sie auch Gebrauch vom erweiterten Formalismus "Wenn A, dann B - ausser C oder D etc." machen. Das Beispiel hatten wir schon früher gegeben, fügen es hier aber noch einmal vollständig ein:

```
$shortener_table = array(
    "(?<![Ss]c)[Hh]ab" => array( "{HAB}", "^Haber", "Habicht" )
);
```

Kurz gesagt bedeutet diese Regel: Kürze "HAB", ausser in "Haber" und "Habicht". Wie Sie sehen definieren wir hier die Regel als Array im Array: Es handelt sich also um eine rekursive Struktur - was ja bekanntlich erst die die Magie eines jeden Formalismus ausmacht ... ;-)

VSTENO lokal installieren

Damit Sie Ihre neuen Zeichen und Regeln ausprobieren können, müssen Sie VSTENO lokal installieren und die Datei data.php durch Ihre eigene ersetzen. Es wird empfohlen, hierfür eine freie Linux-Distribution (z.B. Trisquel oder Debian) zu verwenden, da diese die Philosophie der Freien Software unterstützen und von Hause aus einen Webserver (z.B. Apache) mit sich bringen.

1. Der Apache-Webserver kann in Trisquel oder Debian installiert werden mit: sudo apt-get install apache2. Der Apache-Server ist bereits vorkonfiguriert und das home-Verzeichnis von localhost befindet sich in /var/www/html/.

2. Installieren Sie nun PHP mit: sudo apt-get install php5[24] (anschliessend sollten Sie Ihren Webserver neu starten; falls dies nicht automatisch geschieht, können Sie es manuell tun mit: sudo service apache2 restart).

3. Wechseln Sie ins Verzeichnis /var/www/html und "klonen" Sie VSTENO mit: git clone https://github.com/marcelmaci/vsteno.

4. Ersetzen Sie nun die Datei /var/www/html/vsteno/php/data.php durch ihre eigenen Zeichen und Regeln!

5. Öffnen Sie Ihren Webbrowser und geben Sie die Adresse localhost/vsteno/php/input.php ein - that's it!

[24] Verwenden Sie PHP in der Version 7, falls Ihre Distribution ein entsprechendes Paket anbieten: Die Version ist gegenüber der Version 5 ca. 80% schneller und verbraucht weniger Speicher.

Zum Schluss

Damit sind wir am Ende dieses "linguistischen Tutorials", welches mit einem einigermassen knappen Zeitbudget erstellt wurde. Ich hoffe, dass trotzdem die wesentlichsten Aspekte abgedeckt werden konnten. Bei Unklarheiten stehe ich gerne zur Verfügung (m.maci@gmx.ch).