APL-Journal

A Programming Language

1/2/2014

Urs Oswald
APL, Unicode und UTF-8

Frank Schmidt-Kelletat
IBM-APL2 und die COM-Schnittstelle von WORD 2010 unter WINDOWS 7

Rolf Erbe
Der Huffman-Code

Im Blickpunkt

Urs Oswald
Sudoku mit APL

APL-Germany e.V.
Nr. 1/2 2014 Doppelnummer
Jahrgang 33
ISSN - 1438-4531
RHOMBOS-VERLAG

APL-Journal

33. Jg. 2014, ISSN 1438-4531

Herausgeber: Dr. Reiner Nussbaum, APL-Germany e.V., Mannheim, http://www.apl-germany.de

Redaktion: Dipl.-Volksw. Martin Barghoorn (verantw.), TU Berlin, Franklinstr. 28, D-10587 Berlin, Tel. (030) 314 24392, Fax (030) 314 25901

Verlag: RHOMBOS-VERLAG, Berlin, Kurfürstenstr. 15/16, D-10785 Berlin, Tel. (030) 261 9461, eMail: verlag@rhombos.de, Internet: www.rhombos.de

Erscheinungsweise: halbjährlich

Erscheinungsort: Berlin

Satz: Rhombos-Verlag

Druck: dbusiness.de GmbH, Berlin

Fotonachweis Titelseite: Martin Barghoorn

Liebe APL-Freunde,

es ist mir eine Freude, Ihnen das vorliegende APL-Journal zukommen zu lassen.

Im Jahr 2011 hatten wir zwei Tagungen: Die Frühjahrstagung fand in Köln bei der Deutschen Krankenversicherung (DKV) statt und die Herbsttagung bei der Allianz Deutschland AG in Stuttgart. Den Gastgebern der Veranstaltungen sei an dieser Stelle noch einmal herzlich gedankt.

Viel Vergnügen bei dem Lesen dieser APL-Ausgabe wünscht Ihnen

Ihr Reiner Nussbaum

P.S.: Im Frühjahr 2013 finden bei APL-Germany e.V. Vorstandswahlen statt. Insbesondere für die Positionen Kassenwart und Schriftführer streben die derzeitigen Vorstandsmitglieder eine personelle Neubesetzung an. Interessierte Mitglieder sind aufgerufen, sich für diese Positionen zu engagieren.

INHALT

Bildnachweis:
Martin Barghoorn (Umschlagseite 1)

Urs Oswald

Sudoku mit APL

Das APL-Programm, das hier vorgestellt wird, hat nicht zum Ziel, Sudokus möglichst schnell zu lösen. Vielmehr soll es aufzeigen, wie ein vorgelegtes Sudoku mit den sieben Regeln *F, N, B, T, W, Y, X* gelöst werden kann, wie sie in FOWLER [2] dargestellt werden. Der Lösungsweg wird detailliert protokolliert und kann mit Papier und Bleistift nachvollzogen werden. So kann der passionierte Sudoku-Löser überprüfen, ob seine Lösung unnötig kompliziert ist, oder wie er dort weiterkommt, wo er steckengeblieben ist. Allerdings reichen diese sieben Regeln nicht immer aus. Sudokus, bei welchen man mit diesen Regeln durchkommt, nennt FOWLER [2] *constrained*. Nicht alle, aber die meisten der publizierten Sudokus sind von dieser Art. Mit einem *rekursiven* Programm, welches mit *backtracking* arbeitet, lassen sich alle Sudokus lösen. Solche Programme haben den Charakter einer *blackbox*. Sie geben keine Anleitung, wie man allein mit Papier und Bleistift vorgehen könnte. Sie stützen sich auf systematisches Durchprobieren aller Möglichkeiten, was theoretisch auch allein mit Papier und Bleistift möglich, aber mühselig und langweilig ist.

Der APL-WS `APLSUDO` enthält gut 100 APL-Funktionen, welche der guten Ordnung halber sämtliche in OSWALD [4] in alphabetischer und in OSWALD [5] in Aufruf-Ordnung dokumentiert sind. Der WS ist in APL+Win von APL2000 programmiert. Um die Portabilität zu erhöhen, habe ich auf Kontrollstrukturen und die proprietären ASMFNS-Funktionen verzichtet, ebenso auf Systemfunktionen, welche nicht in IBM-APL2 vorkommen. Der WS ist als Komponentenfile gespeichert. Er kann von jeder Version von APL+Win eingelesen und aktiviert werden. Einzelheiten gebe ich auf Anfrage gerne bekannt. Es sollte wohl auch möglich sein, die Funktionen zu portieren, z.B. in IBM-APL2 oder Dyalog APL.

Die sieben Regeln bewirken „*constraint propagation*“, d.h. die fortgesetzte Einschränkung einer Kandidatenliste, so wie man es beim Lösen mit Papier und Bleistift gewohnt ist. Sie setzen nicht voraus, dass es sich um ein eigentliches Sudoku handelt. Sie funktionieren auch im Falle von *Pseudo-Sudokus*, welche wie Sudokus aussehen, aber möglicherweise keine oder aber mehr als eine Vervollständigung haben. Bei unlösbaren Pseudo-Sudokus führt die fortgesetzte Einschränkung zu einer leeren Zelle ohne übrigbleibende Kandidaten. Ist das Pseudo-Sudoku mehrdeutig, so endet die fortgesetzte Einschränkung vorzeitig in einem Stillstand, wo keine der Regeln zu einer weiteren Einschränkung führt. Ein rekursives Programm führt in diesem Fall jedoch weiter und liefert alle möglichen Vervollständigungen. In OSWALD [3] werden die Regeln anhand vieler Beispiele erläutert.

Sowohl das nachfolgende rekursive Programm `rek_prim` als auch das weiter unten beschriebene Programm `CPROP_FNBTWYX` arbeiten mit einer Kandidatenliste in boolescher Form (BKL). BKL ist ein boolescher Würfel; B[i;;] hält fest, in welchen Zellen der Kandidat `i` vorkommt. `CPROP_FNBTWYX` arbeitet analog zum Lösungsvorgang mit Papier und Bleistift. Es führt parallel zueinander das schon teilweise ergänzte Sudoku (nm: numerische Matrix) und die Kandidatenliste in boolescher Form.

2 Ein rekursives Programm vorweg

Zunächst erlaubt es das relativ kurze, rekursive Programm `rek_prim`, beliebige Sudokus zu lösen (mindestens theoretisch). Darüberhinaus liefert es bei Pseudo-Sudokus sämtliche Vervollständigungen. Zur Erinnerung: Unter einem Pseudo-Sudoku verstehen wir etwas, das wie ein Sudoku aussieht, aber keine oder aber mehr als eine Vervollständigung hat. In den folgenden Beispielen bedeutet dabei die erste Zahl unter dem Aufruf immer die Anzahl der Vervollständigungen. Die Zahl darunter ist die auf Sekunden gerundete Laufzeit des Programms.
Sudokus werden normalerweise mit dem Programm `zeig` wiedergegeben. Anschließend an das Sudoku folgen drei Einsen, welche bestätigen, dass die Block-, Zeilen- und Spaltenbedingungen (jede Ziffer höchstens einmal) erfüllt sind, und eine vierte Zahl, welche die Anzahl der Vorgaben wiedergibt.

```
      t0←sec ⋄ ⍴v←rek_prim ⍋sdk9_ta_170810 ⋄ sec-t0
1
0
      zeig¨⍋sdk9_ta_170810 (1⊃v)
 2 0 0 | 0 0 0 | 0 0 7        2 9 1 | 3 4 5 | 8 6 7
 0 4 0 | 6 0 8 | 0 9 0        7 4 5 | 6 2 8 | 3 9 1
 0 0 0 | 1 0 9 | 0 0 0        3 6 8 | 1 7 9 | 4 2 5
 ---------------------        ---------------------
 0 8 3 | 0 0 0 | 7 4 0        6 8 3 | 5 9 1 | 7 4 2
 0 0 0 | 0 0 0 | 0 0 0        5 2 4 | 7 3 6 | 1 8 9
 0 7 9 | 0 0 0 | 6 5 0        1 7 9 | 4 8 2 | 6 5 3
 ---------------------        ---------------------
 0 0 0 | 9 0 4 | 0 0 0        8 3 2 | 9 1 4 | 5 7 6
 0 5 0 | 8 0 3 | 0 1 0        9 5 7 | 8 6 3 | 2 1 4
 4 0 0 | 0 0 0 | 0 0 8        4 1 6 | 2 5 7 | 9 3 8
 1 1 1 24                     1 1 1 81
```

Es gibt genau eine Vervollständigung. (Also liegt wirklich ein Sudoku vor.) Die gerundete Laufzeit beträgt 0 Sekunden, und das gegebene Sudoku hat 24 Vorgaben. Die Zahl 81 unter dem Sudoku rechts zeigt an, dass die Vervollständigung gelungen ist.

```
      t0←sec ⋄ ρv←rek_prim ⍋sdk9_viel85 ⋄ sec-t0
85
15
      zeig¨(⊂⍋sdk9_viel85),2↑v
 0 0 0 | 0 0 0 | 0 0 0        9 2 8 | 6 5 4 | 7 1 3        9 2 8 | 6 5 4 | 7 1 3
 0 0 3 | 0 9 0 | 6 4 0        7 5 3 | 2 9 1 | 6 4 8        7 5 3 | 2 9 1 | 6 4 8
 0 1 0 | 7 0 8 | 0 2 0        6 1 4 | 7 3 8 | 5 2 9        6 1 4 | 7 3 8 | 5 2 9
 ---------------------        ---------------------        ---------------------
 0 0 9 | 0 0 0 | 4 0 0        2 3 9 | 1 8 6 | 4 7 5        2 7 9 | 1 8 3 | 4 6 5
 0 8 0 | 4 0 5 | 0 9 0        1 8 6 | 4 7 5 | 3 9 2        1 8 6 | 4 7 5 | 3 9 2
 0 0 5 | 0 0 0 | 8 0 0        4 7 5 | 9 2 3 | 8 6 1        4 3 5 | 9 2 6 | 8 7 1
 ---------------------        ---------------------        ---------------------
 0 6 0 | 8 0 2 | 0 5 0        3 6 7 | 8 1 2 | 9 5 4        3 6 7 | 8 1 2 | 9 5 4
 0 4 1 | 0 6 0 | 2 3 0        8 4 1 | 5 6 9 | 2 3 7        8 4 1 | 5 6 9 | 2 3 7
 0 0 0 | 0 0 0 | 0 0 0        5 9 2 | 3 4 7 | 1 8 6        5 9 2 | 3 4 7 | 1 8 6
 1 1 1 25                     1 1 1 81                     1 1 1 81
```

Hier erfordert das Programm 15 Sekunden. Das gegebene Pseudo-Sudoku hat 25 Vorgaben und genau 85 Vervollständigungen. Mit `rek_prim` können auch 4×4-Pseudo-Sudokus vervollständigt werden:

```
      t0←sec ⋄ ρv←rek_prim 4 4↑2 2ρ1 2 3 4 ⋄ sec-t0
12
0
      2 6ρv
 1 2 3 4    1 2 3 4    1 2 3 4    1 2 3 4    1 2 3 4    1 2 3 4
 3 4 1 2    3 4 1 2    3 4 1 2    3 4 1 2    3 4 2 1    3 4 2 1
 2 1 4 3    2 3 4 1    4 1 2 3    4 3 2 1    2 1 4 3    4 3 1 2
 4 3 2 1    4 1 2 3    2 3 4 1    2 1 4 3    4 3 1 2    2 1 4 3

 1 2 4 3    1 2 4 3    1 2 4 3    1 2 4 3    1 2 4 3    1 2 4 3
 3 4 1 2    3 4 1 2    3 4 2 1    3 4 2 1    3 4 2 1    3 4 2 1
 2 1 3 4    4 3 2 1    2 1 3 4    2 3 1 4    4 1 3 2    4 3 1 2
 4 3 2 1    2 1 3 4    4 3 1 2    4 1 3 2    2 3 1 4    2 1 3 4
```

Das 4×4-Pseudo-Sudoku mit dem oberen linken Block 2 2 ρ 1 2 3 4 hat genau 12 Vervollständigungen. Eine theoretische Herleitung dieses Sachverhalts findet man in FELGENHAUER/JARVIS[1] oder OSWALD[3]:

```
      t0←sec ⋄ ρv←rek_prim 4 4ρ0 ⋄ sec-t0
288
1
```

Das leere 4×4-Pseudo-Sudoku hat genau 288 Vervollständigungen; mit anderen Worten gibt es genau 288 vollständige 4×4-Sudokus. Siehe dazu FELGENHAUER/JARVIS[1] oder OSWALD[3]. Das sind 24 Mal so viele Lösungen wie beim vorgängigen Beispiel. 24 ist die Anzahl der Permutationen der Zahlen 1 bis 4. Das Programm rek_prim braucht zur Lösung 1 Sekunde.

Wenn wir dasselbe mit dem leeren 9×9-Sudoku versuchen würden, hätten wir keine Chancen. Das Programm müsste nicht 288, sondern nach FELGENHAUER/JARVIS[1] 6 670 903 752 021 072 936 960 Vervollständigungen erzeugen. Das Programm `rek_prim` erzeugt theoretisch zu jedem Pseudo-Sudoku die Menge aller Vervollständigungen. Praktisch sind ihm aber Grenzen gesetzt, die möglicherweise von einer Beschränkung der Rekursionstiefe herrühren. Man erhält jedoch ein sehr leistungsfähiges Programm, wenn man `rek_prim` mit einigen der 7 erwähnten Regeln kombiniert. Wir kommen darauf in Abschnitt 5 zurück. Anschließend folgt das Programm rek prim mit seinen Unterprogrammen sowie das Programm `zeig`.

```
 [0] v_nm←rek_prim nm;side;a;min;Report;c;r;BKL;w;N;L;I;nmi;cc
 [1] ⍝------------------------------------------------------------------------
 [2] ⍝   lv: Pseudoudoku-Name
 [3] ⍝ v_nm: alle Vervollständigungen
 [4] ⍝------------------------------------------------------------------------
 [5]  Report←0
 [6]  v_nm←⍳0
 [7] →(~∧/konformZSB nm)/0              ⍝ Zeilen-/Spalten-/Block-Bedingung?
 [8] →(~∧/,nm>0)/L_0                    ⍝ noch nicht vollständig?
 [9]  v_nm←v_nm,⊂nm ⋄ →0                ⍝ vollständig: Lösung hinzufügen
[10] L_0:
[11]  side←↑⍴nm ⋄ BKL←genBKL nm
[12]  a←,+/[1]BKL                       ⍝     a: Kandidatenzahlen
[13]  min←⌊/a~0                         ⍝   min: minimale (ausser 0)
[14]  (c r)←1+(side,side)⊤¯1+a⍳min      ⍝ (c r): "row,column" erste Zelle mit
[15]                                    ⍝  dieser minimalen Kandidatenzahl
[16]  cc←w/⍳⍴w←BKL[;c;r]                ⍝    cc: Kandidaten in dieser Zelle
[17]  N←⍴cc ⋄ →(N=0)/0
[18]  L←(N⍴LOOP),0 ⋄ I←1
[19] LOOP:
[20]  nmi←nm                            ⍝ Kandidaten einen nach dem andern
[21]  nmi[c;r]←I⊃cc                     ⍝ in (c,r) einsetzen
[22]  v_nm←v_nm,rek_prim nmi            ⍝ rekursiver Aufruf ("back tracking")
[23] →L[I←I+1]

 [0] b3←konformZSB nm;w;u;side
 [1] ⍝------------------------------------------------------------------------
 [2] ⍝ prüft, ob pro Block/Zeile/Spalte jede Ziffer höchstens einmal vorkommt
 [3] ⍝ b3[1]: Zeilen  ZSB-konform?
 [4] ⍝   [2]: Spalten ZSB-konform?
 [5] ⍝   [3]: Bloecke ZSB-konform?
 [6] ⍝------------------------------------------------------------------------
 [7]  side←↑⍴nm
 [8]  b3←3⍴0
 [9]  b3[1]←∧/,⊃side pruefe¨⊂[2]nm                    ⍝ Zeilen
[10]  b3[2]←∧/,⊃side pruefe¨⊂[2]⍉nm                   ⍝ Spalten
[11]  b3[3]←∧/,⊃side pruefe¨⊂[2]bloeckeZUzeilen nm    ⍝ Bloecke


 [0] bs←pruefe nv
 [1] ⍝------------------------------------------------------------------------
 [2] ⍝ prüft, ob in nv jede Ziffer≠0 höchstens einmal vorkommt
 [3] ⍝ l: side
 [4] ⍝------------------------------------------------------------------------
 [5]  bs←(⍴nv~0)=+/(⍳side)∊nv

 [0] bm3D←genBKL nm;side;N;L;I;bm;w
 [1] ⍝------------------------------------------------------------------------
 [2] ⍝   nm: Sudoku (numerisch)
 [3] ⍝ bm3D: BKL (boolesche Kandidatenliste)
 [4] ⍝------------------------------------------------------------------------
 [5]  side←1⊃⍴nm ⋄ bm3D←(3⍴side)⍴0 ⋄ N←side ⋄ L←(N⍴LOOP),L_ENDE ⋄ I←1
 [6] LOOP:
 [7]  bm←nm=0
 [8]  bm←bm∧(~∨/nm=I)∘.∧~∨/[1]nm=I                    ⍝ Zeilen u. Spalten
 [9] ⍝ Blöcke (werden vorübergehend in Zeilen verwandelt)
[10]  bm←bm∧bloeckeZUzeilen(~∨/I=bloeckeZUzeilen nm)∘.+side⍴0          ⍝ Blöcke
[11]  bm3D[I;;]←bm ⋄ →L[I←I+1]
[12] L_ENDE:

 [0] nm←bloeckeZUzeilen nm;w;base
 [1] ⍝------------------------------------------------------------------------
 [2] ⍝ verwandelt Blöcke in Zeilen, ist identisch mit seiner Umkehrung
 [3] ⍝ nm: quadratische Matrix
 [4] ⍝------------------------------------------------------------------------
 [5]  base←1 2 3[1 4 9⍳↑⍴nm] ⋄ nm←⊃,,¨⊃¨(base/⍳base)⊂[1](base/⍳base)⊂[2]nm

 [0] lm←zeig nm;e;nm;ss;w;base
 [1] ⍝------------------------------------------------------------------------
 [2] ⍝ für 4×4 und 9×9
 [3] ⍝------------------------------------------------------------------------
 [4]  lm←1 0⍕nm
 [5]  base←1 2 3[1 4 9⍳↑⍴nm] ⋄ e←(¯1+base×1+base)⍴(base⍴1),0 ⋄ lm←e\[1]e\lm
 [6]  ss←(base+1)×⍳base-1 ⋄ lm[;ss]←'|'
 [7]  w←(⍴lm)[2] ⋄ e←(¯1+w+w)⍴1 0 ⋄ lm←e\lm ⋄ lm[ss;]←'-'
 [8]  lm←(0 5+⍴lm)↑lm ⋄ lm←lm,[1](1↓⍴lm)↑⍕(konformZSB nm),+/,0<nm
```

3 Die sieben Regeln des fortgesetzten Einschränkens

Die sieben Arten des fortgesetzten Einschränkens sind:

F In einer bestimmten Zelle ist nur eine Ziffer möglich (F: “Field”)

N In einem Block/einer Zeile/einer Spalte ist eine Ziffer nur in einer einzigen Zelle möglich (N: “only”)

B Block-Zeile- und Block-Spalt-Koppelung

T Offene und versteckte Tupel (T: “Tupel”)

X X-Ketten (Einkandidaten-Ketten)

Y Y-Ketten (Paarketten)

W W-Muster (x-wing, swordfisch)

3.1 Regel F

	2	3		9		6	4	
	1		7		8		2	
		9		8		4		5
	8		4		5		9	
4		5				8		
	6		8		2		5	
	4	1		6		2	3	

sdk 1

$= F \Rightarrow$

	2	3		9	1	6	4	
	1		7		8		2	
		9		8		4		5
	8		4		5		9	
4		5				8		
	6	7	8		2		5	
	4	1		6		2	3	

sdk 2

- In Zelle (2,6) ist nur eine Ziffer möglich, nämlich 1.
- In Zelle (7,3) ist nur eine Ziffer möglich, nämlich 7.

3.2 Regel N - N_B, N_R, N_C

	1	2	8	4				
	9	3				4		
	4				3			
1				3	4			
	5					1	3	4
			1	7				9
			6				2	
		1	3			7	4	
			4		5	8	1	

sdk 3

$= N_B \Rightarrow$

	1	2	8	4				
	9	3				4		
	4				3			
1			5	3	4			
	5					1	3	4
			1	7				9
			6		7	9	2	
		1	3			7	4	
			4		5	8	1	

sdk 4

- (N_B) In Block $B_{2,2}$ (Mitte) ist die Ziffer 5 nur in der Zelle (4,4) möglich.
- (N_B) In Block $B_{3,2}$ ist die Ziffer 7 nur in der Zelle (7,6) möglich.
- (N_B) In Block $B_{3,3}$ ist die Ziffer 9 nur in der Zelle (7,7) möglich.

				4	8	9		
	8	4	2			5	1	
	9		3			2	8	4
9						6	4	
	4	5				7		
	3	8			4	1		2
4	6				1	8	2	
8	2		4		5	3		
		9	8	2		4		

sdk 5

$= N_R \Rightarrow$

				4	8	9		
	8	4	2			5	1	
	9		3			2	8	4
9						6	4	
	4	5				7		
	3	8			4	1		2
4	6				1	8	2	5
8	2		4		5	3		
		9	8	2		4		

sdk 6

- (N_R) In Zeile 7 ist die Ziffer 5 nur in der letzten Spalte möglich.

					1	2		3
	4			5			7	
								6
1					7			
	8						9	
			3					2
5		8					2	
				9			8	
3		2	6					

sdk 7

$= N_C \Rightarrow$

					1	2		3
	4			5			7	
								6
1					7		3	
	8						9	
			3					2
5		8					2	
				9			8	
3		2	6					

sdk 8

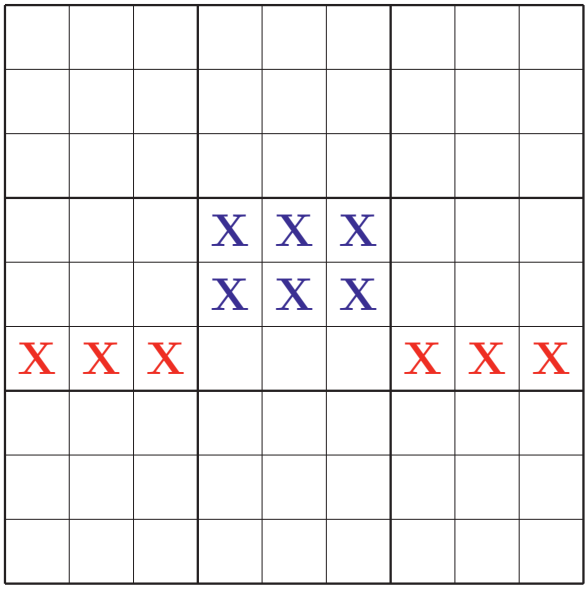
- (N_C) In Spalte 8 ist die Ziffer 3 nur in der 4. Zeile möglich.

Bemerkung. Diese vier Beispiele zeigen, dass die Regeln F, N_B, N_R und N_C voneinander unabhängig sind. Denn in jedem Fall führt jeweils von diesen Regeln nur die angegebene weiter.

3.3 Regel B

Die Regel B zerfällt in die 4 Subregeln B ➢ R, B ➢ C, R ➢ B und C ➢ B. Im Bild rechts werden B ➢ R und R ➢ B erläutert (bezogen auf einen bestimmten Kandidaten). Die Kreuze bedeuten, dass der Kandidat in jenen Zellen nicht vorkommt.

Ist nun zum Beispiel der Kandidat in der 6. Zeile auf den mittleren Block beschränkt, so kann er in diesem Block sonst nicht vorkommen. Ist er umgekehrt im Block auf die 6. Zeile beschränkt, so kann er in dieser Zeile nur innerhalb dieses Blocks vorkommen. Analog wirken die Regeln B ➢ C und C ➢ B.



sdk 9

Regel B ➢ R Wenn innerhalb eines Blocks ein Kandidat auf eine einzige Zeile beschränkt ist, kann dieser Kandidat außerhalb des Blocks in dieser Zeile gestrichen werden.

Regel B ➢ C Wenn innerhalb eines Blocks ein Kandidat auf eine einzige Spalte beschränkt ist, kann dieser Kandidat außerhalb des Blocks in dieser Spalte gestrichen werden.

Regel R ➢ B Wenn innerhalb einer Zeile ein Kandidat auf einen einzigen Block beschränkt ist, kann dieser Kandidat außerhalb der Zeile in diesem Block gestrichen werden.

Regel C ➢ B Wenn innerhalb einer Spalte ein Kandidat auf einen einzigen Block beschränkt ist, kann dieser Kandidat außerhalb der Spalte in diesem Block gestrichen werden.

Die folgenden Beispiele zeigen Sudokus, welche fast allein mit den Regeln F und N gelöst werden können. Pro Beispiel muss jeweils eine der vier Subregeln B ➢ R, B ➢ C, R ➢ B und C ➢ B angewendet werden.

Beispiel (Regel B ➢ R)

	9				8	1	6	
		1	7			9	2	
2								3
3		8	4		1	5		6
	4	9	8			2	3	
6					7	8		4
8			6	4				9
				7	2	6		
	6	7	1	8			5	2

sdk 10

4 5 7	9	4 5 3	2 3 5	2 3 5	8	1	6	7 5
4 5	3 5 8	1	7	3 5 6	3 4 5 6	9	2	5 8
2	5 7 8	6	5 9	1	4 5 9	4 7	4 7 8	3
3	2 7	8	4	2 9	1	5	7 9	6
1 5 7	4	9	8	5 6	5 6	2	3	1 7
6	1 2 5	2 5	2 3 5 9	2 3 5 9	7	8	1 9	4
8	1 2 3 5	2 3 5	6	4	3 5	3 7	1 7	9
1 4 5 9	1 3 5	3 4 5	3 5 9	7	2	6	1 4 8	1 8
4 9	6	7	1	8	3 9	3 4	5	2

sdk 11

Durch zweimalige Anwendung der Regel N_B erhält man das rechte Sudoku aus dem linken. Nun ist im Block $B_{1,3}$ der Kandidat 4 auf Zeile 3 beschränkt. Deshalb kann nach Regel B ➢ R Kandidat 4 in Zelle (3, 6) gestrichen werden. Nun kann nach Regel N_B 4 in die Zelle (2, 6) gesetzt werden.

Zur Vervollständigung genügt jetzt Regel F. In diesem Beispiel ist eine Kandidatenliste nicht nötig.

In vielen Fällen führen die Regeln B ➢ R and B ➢ C zusammen mit den Regeln F und N direkt zu einer neuen besetzten Zelle.

Beispiel (Regel C ➢ B)

		3				8		
					1			
6				7				5
2	7		3		8			
		1		2		4		
			6		4		7	9
3				8				7
			4					
		2	5			1		

sdk 12

7	1 2 4 5 ~~8~~	3	2 9	4 5 6	2 6 ~~8~~	8	1 2 4 6 9	1 2 4 6
4 5 8	2 4 5 8 ~~9~~	5 8 9	2 8 9	3 4 5 6	1	7	2 3 4 6 9	2 3 4 6
6	1 2 4 8 ~~9~~	8 9	2 8 9	7	2 3 ~~8~~	3 9	1 2 3 4 9	5
2	7	4	3	9	8	5 6	1 5 6	1 6
9	6	1	7	2	5	4	3 8	3 8
5 8	3	5 8	6	1	4	2	7	9
3	4 5 9	6	1	8	2 9	5 9	2 4 5 9	7
1	5 8 9	7	4	3 6	2 3 6 9	3 5 6 9	2 3 5 6 8 9	2 3 6 8
4 8	4 8 9	2	5	3 6	7	1	3 4 6 8 9	3 4 6 8

sdk 13

Wiederum erhält man das rechte Sudoku mit den Regeln F und N aus dem linken. Nun ist der Kandidat 9 in Spalte 3 auf den Block $B_{1,1}$ beschränkt. Deshalb kann nach Regel C ➢ B der Kandidat 9 im Rest des Blocks gestrichen werden. In der Spalte 4 sind beide Kandidaten 2 und 9 auf Block $B_{1,2}$ beschränkt und können deshalb in den Zellen (1, 6) und (3, 6) gestrichen werden.

3.4 Regel T

Offene und versteckte Tupel

oT-2 {2, 8} offenes 2-Tupel
hT-4 {3, 4, 7, 9} verstecktes 4-Tupel

In Zeile 4 tritt ein offenes Tupel auf: In den Zellen (4, 2) und (4, 6) kommen nur die Kandidaten 2 und 8 vor. Das hat automatisch zur Folge, dass die übrigen Kandidaten, 3, 4, 7 und 9, nur in den übrigen Zellen vorkommen und in diesen ein *verstecktes* (*hidden*) *Quadrupel* bilden. Offene und versteckte Tupel treten immer gemeinsam auf, aber ein verstecktes Paar ist unter Umständen leichter zu entdecken als ein offenes Quadrupel.

In der Folge können die Kandidaten 2 und 8 außerhalb der Zellen (4, 2) und (4, 6) gestrichen werden.

5	2 8	1	⊗2 3 4 ⊗8	3 7 9	2 8	6	⊗2 3 7	3 4 ⊗8 9

sdk 14

OffeneTupel. Wenn in n unbesetzten Zellen eines Blocks (einer Zeile, einer Spalte) nur gerade n verschiedene Kandidaten auftreten, so sagt man, in diesem Block (in dieser Zeile, in dieser Spalte) liege ein offenes n-Tupel vor.

Regel T. Wenn in n unbesetzten Zellen eines Blocks (einer Zeile, einer Spalte) nur gerade n verschiedene Kandidaten auftreten, so können diese Kandidaten in den übrigen unbesetzten Zellen dieses Blocks (dieser Zeile, dieser Spalte) gestrichen werden.

3.5 Regel W

X-Wing
Swordfish

Das Schema rechts hat "dualen Charakter", d.h. es kann in zwei Richtungen gelesen werden. Die Kreuze bedeuten, dass dort der betreffende Kandidat nicht vorkommt.

Ist auf den Zeilen 2, 5 und 8 der Kandidat auf die Spalten 3, 5 und 9 beschränkt, so kann er in diesen Spalten außerhalb der genannten Zeilen gestrichen werden.

Dual: Ist in den Spalten 3, 5 und 9 der Kandidat auf die Zeilen 2, 5 und 8 beschränkt, so kann er in diesen Zeilen außerhalb der genannten Spalten gestrichen werden.

X-Wing (5)
Spalten 5, 9 / Zeilen 3, 7

In den Spalten 5 und 9 ist der Kandidat 5 auf die Zeilen 3 und 7 beschränkt. Auf diesen Zeilen kann deshalb der Kandidat außerhalb der Spalten 5 und 9 gestrichen werden.

		X		X				X
X	X		X		X	X	X	
		X		X				X
		X		X				X
X	X		X		X	X	X	
		X		X				X
		X		X				X
X	X		X		X	X	X	
		X		X				X

sdk 15

2	1 9	7	1 5	4	1 3	6	3 5 9	8
4	3	5	9	6	8	7	1	2
1 9	6	8	7	2 5	1 2 3	3 8 9	3 4 8 9	4 5 9
5	4	1	6	3	9	2	8	7
6	7	2	1 4	8	1 4	5 9	5 9	3
3	8	9	2	7	5	4	6	1
1 9	1 2 9	3 4	4 8	2 5 9	6	3 8	7	5 9
8	5	6	3	2 9	7	1	2 4 9	4 9
7	2 9	3 4	4 5 8	1	2 4	3 5 8 9	2 3 5 9	6

sdk 16

Regel W. Ist ein Kandidat auf n Zeilen auf n Spalten beschränkt, so kann er in diesen Spalten außerhalb der betreffenden Zeilen gestrichen werden. Analoges gilt beim Vertauschen von "Zeilen" und "Spalten".

Swordfish (9)
Zeilen 1, 5, 9

In den Zeilen 1, 5, 9 ist der Kandidat 9 auf die Spalten 2, 7 und 8 beschränkt. Deshalb kann der Kandidat in diesen Spalten außerhalb der Zeilen 1, 5, 9 gestrichen werden.

2	1 (9)	7	1 5	4	1 3	6	3 5 (9)	8
4	3	5	9	6	8	7	1	2
1 9	6	8	7	2 5	1 2 3	3 (9)	3 4 (9)	4 5 9
5	4	1	6	3	9	2	8	7
6	7	2	1 4	8	1 4	5 (9)	5 (9)	3
3	8	9	2	7	5	4	6	1
1 9	1 2 (9)	3 4	4 8	2 5 9	6	3 8	7	5 9
8	5	6	3	2 9	7	1	2 4 (9)	4 9
7	2 (9)	3 4	4 5 8	1	2 4	3 5 8 (9)	2 3 5 (9)	6

sdk 17

3.6 Regel X

Starke und schwache Kanten. Kommt ein Kandidat innerhalb eines Blocks zweimal vor, so nennt man die Verbindung der betreffenden Zellen eine *Kante* (in Bezug auf diesen Kandidaten). Kommt der Kandidat innerhalb des Blocks kein weiteres Mal vor, so heißt die Kante *stark*, sonst *schwach*. Analog sind Kanten in Bezug auf Zeilen und Spalten definiert. Im Folgenden unterscheiden wir zwischen X_1-Ketten und X_2-Zyklen.

3.6.1 X_1-Ketten

Unter einer *einfachen* X_1-Kette versteht man eine zusammenhängende Folge aus einer ungeraden Anzahl starker Kanten.

einfache X_1-Kette (2)
einfache X_1-Kette (7)

Die rote Kette besteht aus 3 aufeinanderfolgenden starken Kanten bezüglich des Kandidaten 2. In der Vervollständigung (genauer: in jeder Vervollständigung) wird deshalb die Ziffer 2 in genau einer der beiden Endzellen (4, 1) und (8, 5) stehen. Somit kann der Kandidat 2 in Zelle (4, 5) gestrichen werden. Analog bewirkt die blaue Kette, dass der Kandidat 7 in Zelle (8, 6) gestrichen werden kann.

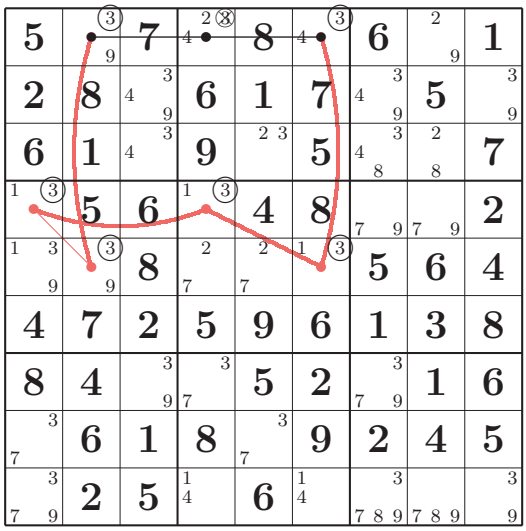
7 6	4	9	8	5	7 6	3	2	1
7 8	3 5 8	7 8 5	7 3	1	2	4	9	6
1	3 6	2	9	4 3 6	4 3	5	7	8
2 8 6	7	5 8	2 3 6	2 3 4	4 3 8	1	5 3	9
9	2 5 6 8	1	7 2 3 6	7 2 3	8 3	2 6	5 3	4
3	2 6	4	1	9	5	2 6	8	7
4	9	3	5	8	1	7	6	2
5	2 8	7 8	4	2 7 6	7 6	9	1	3
2 7	1	6	7 2 3	7 2 3	9	8	4	5

sdk 18

X_1-Ketten, allgemein. Sind zwei einfache X_1-Ketten gemäß obiger Definition durch eine schwache Kante verbunden, so hat die (jede) Vervollständigung des Sudokus die Eigenschaft, dass der Kandidat in mindestens einer Endzelle als Ziffer gesetzt wird. Somit kann in jeder Zelle, die mit beiden Endzellen assoziiert ist (mit jeder in einem gemeinsamen Block, einer gemeinsamen Zeile oder einer gemeinsamen Spalte liegt), der Kandidat gestrichen werden.

X_1-Kette (3) aus zwei einfachen X_1-Ketten
zweimal je ungerade Anzahl starker Kanten, verbunden durch eine schwache Kante

Die rote X_1-Kette für den Kandidaten 3 besteht aus zwei einfachen X_1-Ketten, welche mit den Zellen (4, 1) und (5, 2) verbunden sind (schwach, weil der Kandidat 3 im Block $B_{2,1}$ (Mitte links) noch ein drittes Mal vorkommt).



sdk 19

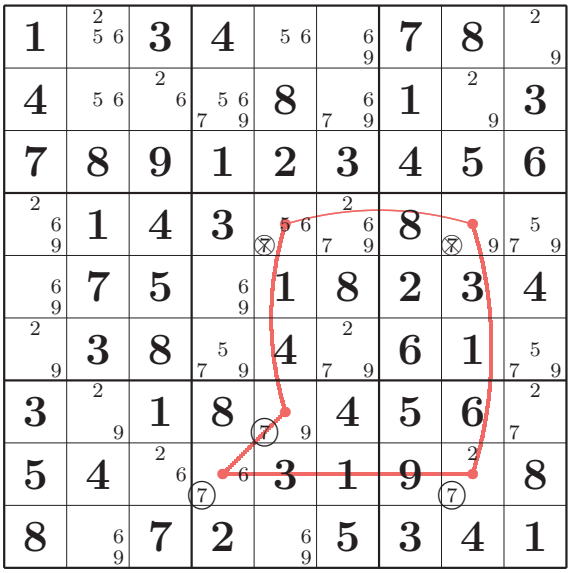
Regel X_1. Liegt für einen Kandidaten eine X_1-Kette vor, so kann der Kandidat in jeder Zelle gestrichen werden, die mit beiden Endzellen der Kette assoziiert ist.

3.6.2 X_2-Zyklen

Ein einfacher X_2-Zyklus besteht aus einer geraden Anzahl starker Kanten und einer schwachen Kante.

einfacher X_2-Zyklus (7)
gerade Anzahl starker Kanten und eine schwache Kante

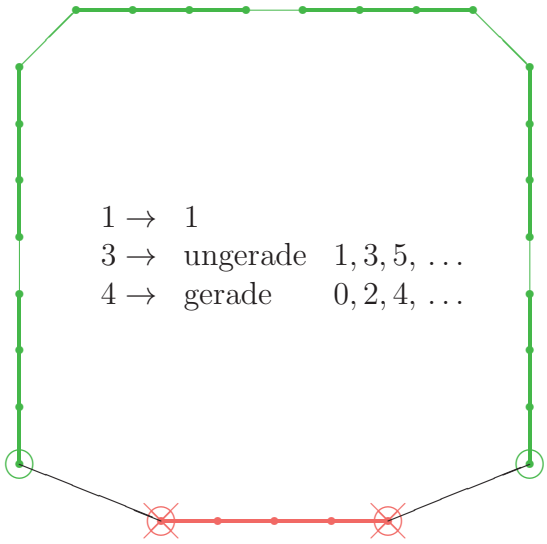
Da bei jeder starken Kante der Kandidat an einem und nur einem Ende in die Vervollständigung eingehen wird, gibt es bei einer Kette aus einer geraden Anzahl Kanten nur zwei Möglichkeiten: Der Kandidat erscheint in der Vervollständigung in beiden oder in keiner der beiden Endzellen. Der erste Fall ist jedoch wegen der schwachen Kante ausgeschlossen. Also kann der Kandidat in beiden Endzellen gestrichen werden.



sdk 20

Allgemeines Schema eines X_2-Zyklus

X_1-Kette
Verbindungskanten
gerade Anzahl starker Kanten



Der allgemeine X_2-Zyklus setzt sich zusammen aus einer X_1-Kette und einer Kette aus einer geraden Anzahl starker Kanten, mit zwei schwachen Kanten als Verbindungsstücken.
Im Schema rechts steht die Kette aus 4 starken Kanten für eine Kette aus einer beliebigen geraden Anzahl starker Kanten. (Sogar 0 ist zulässig; dann mutiert die Regel zur X_1-Regel.) Jede Kette aus genau 3 starken Kanten steht für eine Kette aus einer beliebigen ungeraden Anzahl starker Kanten. Jede schwache Kante (dünn) steht für eine und nur eine schwache Kante.

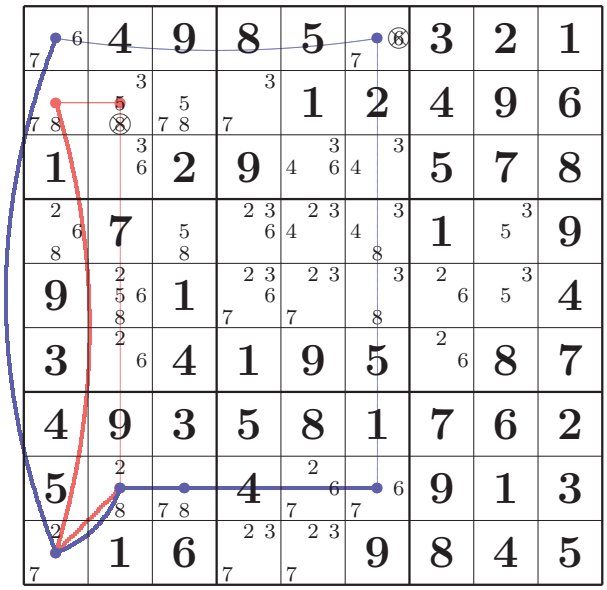
Regel X_2. In einem X_2-Zyklus kann der Kandidat an den Enden der Kette aus einer geraden Anzahl starker Kanten gestrichen werden.

3.7 Regel Y

Y-Ketten bestehen aus Kanten, welche Zellen mit je genau 2 Kandidaten verbinden. Eine Y-Kette für den Kandidaten c_0 liegt vor, wenn die Folge der Kandidatenpaare in folgender Form geschrieben werden kann:

$$\{c_0, c_1\}, \{c_1, c_2\}, \ldots, \{c_{n-1}, c_0\}$$

Im Sudoku rechts liegt eine Y-Kette für den Kandidaten 8 und eine für den Kandidaten 6 vor.



sdk 21

Y (8): {8, 7}, {7, 2}, {2, 8}
Y (6): {6, 7}, {7, 2}, {2, 8}, {8, 7}, {7, 6}

Regel Y. Liegt eine Y-Kette für einen Kandidaten c_0 vor, so kann dieser in jeder Zelle, die mit beiden Endzellen der Kette assoziiert ist, gestrichen werden.

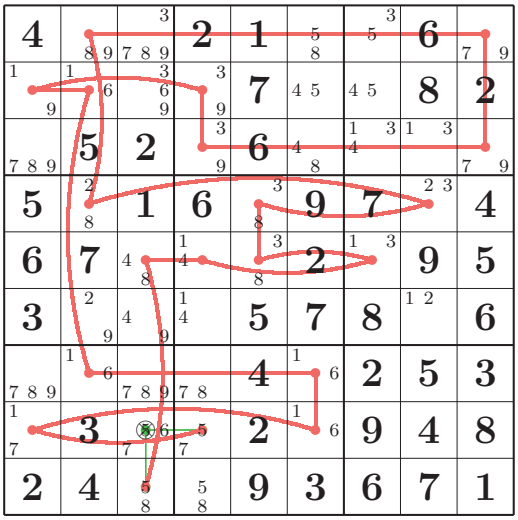
Im obigen Beispiel kann der Kandidat 8 in der Zelle (2, 2), der Kandidat 6 in der Zelle (1, 6) gestrichen werden.

In Fowler[2] ist eine sehr komplizierte Y-Kette für den Kandidaten 5, welche es gestattet, diesen in Zelle (8, 3) zu streichen:

Y (5):

{5, 7}, {7, 1}, . . ., . . . , {4, 8}, {8, 5}

Aber: Man kann das Sudoku auch anders lösen . . .



sdk 22

4 Fortgesetztes Einschränken mit APL

4.1 Boolesche Kandidatenliste (BKL)

Man erhält üblicherweise eine Kandidatenliste, indem man in jede leere Zelle sämtliche Kandidaten schreibt, die weder im betreffenden Block, noch in der betreffenden Zeile, noch in der betreffenden Spalte vorkommen.
Eine solche Kandidatenliste lässt sich durch einen Booleschen Würfel darstellen. Die erste Würfelebene drückt das Vorkommen des Kandiaten 1 aus, die 2. Ebene die des Kandidaten 2 usw.

Das folgende Beispiel illustriert die Boolesche Kandidatenliste (BKL) im Fall eines 4×4-Sudokus (`Δsdk4_001`). Die BKL besteht aus vier Booleschen Matrizen.

```
      lm1←⎕ucmd'display osDEB¯1 0↓zeig nm←Δsdk4_001'
      lm2←⎕ucmd'display osDEB¨(⊂¯1 0)↓¨zeig¨⊂[2 3]genBKL nm'

      (9 15↑' ',[1]lm1,[1]' '),lm2
                 .→-----------------------------------------------.
.→--------.      |.→--------..→--------..→--------..→--------.|
↓2 0 | 1 0|      |↓0 0 | 0 0|↓0 0 | 0 0|↓0 1 | 0 0|↓0 0 | 0 1||
|0 0 | 3 0|      ||1 1 | 0 0||0 0 | 0 1||0 0 | 0 0||1 0 | 0 1||
|---------|      ||---------||---------||---------||---------||
|0 4 | 0 0|      ||1 0 | 0 1||0 0 | 1 1||1 0 | 0 1||0 0 | 0 0||
|0 0 | 0 0|      ||1 1 | 0 1||0 1 | 1 1||1 1 | 0 1||0 0 | 1 1||
'---------'      |'---------''---------''---------''---------'|
                 '∊-----------------------------------------------'
```

Zum Beispiel eliminiert die 1 in Zelle (1, 3) in der ersten Matrix den Kandidaten 1 in Block $B_{1,2}$ (oben rechts), Zeile 1 und Spalte 3. Die 4 in Zelle (3, 2) eliminiert in der 4. Matrix den Kandidaten 4 in Block $B_{2,1}$ (unten links), Zeile 3 und Spalte 2. Die übrigen Nullen in den 4 Booleschen Matrizen zeigen die schon besetzten Zellen an.

Das Sudoku und die BKL erzeugen eine Partition der Zellen, wie unten ersichtlich wird:

```
      BKL←genBKL ⍋sdk4_001

      (∨/[1]clues∧BKL) (∨/[1]clues∨BKL)
0 0 0 0    1 1 1 1
0 0 0 0    1 1 1 1
0 0 0 0    1 1 1 1
0 0 0 0    1 1 1 1
```

Wir fügen noch das Programm `osDEB` an, welches ein Ersatz für das ASMFNS-Programm `DEB` von APL+Win ist:

```
[0] lm←osDEB lm;bv1;bv2;MAT;lm0;lm1;w
[1] ⍝------------------------------------------------------------------------
[2] ⍝ Ersatz für ASM-Funktion DEB in klassischem APL
[3] ⍝ Eliminiert Anfangs- und Endleerspalten und reduziert
[4] ⍝  im Innern mehrfache Leerspalten auf einfache
[5] ⍝------------------------------------------------------------------------
[6]  MAT←2=⍴⍴lm
[7]  lm←(¯2↑1 1,⍴lm)⍴lm ⋄ lm←' ',lm,' ' ⋄ lm←0 1↓0 ¯1↓(~1 1⍷∧/[1]' '=lm)/lm
[8] →MAT/0 ⋄ lm←,lm
```

4.2 Die Regel F

Das folgende Programm `cpropF` reduziert ein gegebenes Sudoku `nm` und die zugehörige BKL gemäß Regel *F* :

```
 [0] v2←cpropF v2;bm;BKLunique;BKL0;BKL;w;nm;cm;side;v;t
 [1] ⍝------------------------------------------------------------------------
 [2] ⍝ Aufgrund der gemäss BKL feststehenden Ziffern
 [3] ⍝ werden BKL und nm (Sudoku) nachgeführt.
 [4] ⍝ (d,r,c):  digit,row,column
 [5] ⍝------------------------------------------------------------------------
 [6]  t←'F ---'                          ⍝ Bildschirm: Regel
 [7]  (BKL nm)←v2
 [8]  side←1⊃⍴BKL
 [9]  BKL0←BKL                           ⍝ zu Protokollzwecken
[10]  bm←1=+/[1]BKL                      ⍝ Wo ist die Ziffer eindeutig?
[11]  BKLunique←BKL∧(⍴BKL)⍴bm            ⍝ In den andern Zellen BKL=0
[12]  v←numeriereBKL BKLunique           ⍝ l in BKLunique --> (d,r,c)
[13]  BKL←BKL∧⊃∧/side gen3DFilter¨v      ⍝ Ausputz von BKL
[14]  w←,nm ⋄ w[1+side⊥¯1+⍉⊃1↓¨v]←1⊃¨v  ⍝ nm gemäss v nachführen
[15]  nm←(side,side)⍴w
[16]  BKL←BKL∧genBKL nm                  ⍝ BKL anpassen
[17] w←÷(nm=0)≡∨/[1]BKL                  ⍝ Kompatibilität von nm/BKL testen
[18] v2←BKL nm
[19] →(Report=0)/0
[20] →(0=⍴v)/0
[21] ⍝ Protokoll nachführen: Anfang ------------------------------
[22] ⍝=================================================================
[23]  w←,' ',' →'subLin 1 0 2 0 1 0⍕⊃v  ⍝ Zellenangabe
[24]  cpropProgress BKL0 BKL (cm←t,w)   ⍝ l: nm t0R Report
[25] ⍝=================================================================
[26] ⍝ Protokoll nachführen: Ende -------------------------------
```

Es folgen die zugehörigen Unterprogramme:

```
 [0] v_nv3←numeriereBKL bm3D;side
 [1] ⍝------------------------------------------------------------------------
 [2] ⍝ gibt zu jeder 1 in bm3D die 3 Koordinaten an (d,r,c)
 [3] ⍝------------------------------------------------------------------------
 [4]  side←1⊃⍴bm3D ⋄ v_nv3←⊂[2]⍉1+(3⍴side)⊤¯1+(,bm3D)/⍳⍴,bm3D
```

```
 [0] bm3D←ns gen3DFilter nv3;side;d;r;c;base
 [1] ⍝------------------------------------------------------------------------
 [2] ⍝ ns: side
 [3] ⍝ Wenn an der Stelle nv3 (Ziffer, Zeile, Spalte) eine 1 gesetzt ist:
 [4] ⍝ - keine andere Ziffer
 [5] ⍝ - Ziffer nicht möglich in derselben Zeile
 [6] ⍝ - Ziffer nicht möglich in derselben Spalte
 [7] ⍝ - Ziffer nicht möglich im selben Block
 [8] ⍝------------------------------------------------------------------------
 [9]  side←ns
[10]  base←1 2 3[1 4 9⍳side]
[11]  (d r c)←nv3                  ⍝ d,r,c: digit, row, column
[12]  bm3D←(3⍴side)⍴1
[13]  bm3D[;r;c]←0                 ⍝ keine andere Ziffer in der Zelle (r,c)
[14]  bm3D[d;r;]←0                 ⍝ Ziffer nicht möglich in derselben Zeile
[15]  bm3D[d;;c]←0                 ⍝ Ziffer nicht möglich in derselben Spalte
[16]  bm3D←bloeckeZU3zeilen bm3D
[17]  bm3D[d;base welcherBlock r,c;]←0  ⍝ Ziffer nicht möglich im selben Block
[18]  bm3D←bloeckeZU3zeilen bm3D
[19]  bm3D[d;r;c]←1                    ⍝ ursprünglicher Wert wird gerettet
```

```
 [0] na←bloeckeZU3zeilen na;base
 [1] ⍝------------------------------------------------------------------------
 [2] ⍝ verwandelt Blöcke in Zeilen in 3D-Matrix mit ⍴ = x,side,side
 [3] ⍝ (analog zu bloeckeZUzeilen) - identisch mit seiner Umkehrung
 [4] ⍝------------------------------------------------------------------------
 [5]  base←1 2 3[1 4 9⍳⌈/⍴na] ⋄ na←(⍴na)⍴∊⊂[3](base/⍳base)⊂[2](base/⍳base)⊂na
```

```
 [0] ns←ns welcherBlock nv2;base;side
 [1] ⍝------------------------------------------------------------------------
 [2] ⍝  ns: base (Default 3)
 [3] ⍝ nv2: r,c  "row" "column"
 [4] ⍝ findet den Block, in dem sich die Zelle (r,c) befindet
 [5] ⍝ Blöcke zeilenweise durchgezählt (1 bis side)
 [6] ⍝------------------------------------------------------------------------
 [7] →(2=⎕nc'ns')/L_0 ⋄ ns←3
 [8] L_0: base←ns ⋄ side←base×base ⋄ ns←1+base⊥¯1+1+⌊(¯1+nv2)÷base
```

4.3 Die Regel N

Für die Regel *F* wird BKL in Richtung der 1. Achse auf Eindeutigkeit abgesucht. Für die Regel N wird BKL analog in Richtung der Achsen 2 und 3 abgesucht. Zur Ermittlung der Block-Eindeutigkeit müssen zuerst die Blöcke in Zeilen verwandelt werden. Das Programm cpropN ist im Wesentlichen dasselbe wie cpropF, nur dass einige Achsenpermutationen vorgenommen werden.

```
 [0] v2←ns cpropN v2;BOX;BKL0;BKL;nm;t;p;bm;BKLunique;v;w;cm;side
 [1] ⍝-----------------------------------------------------------------------
 [2] ⍝ Aufgrund der Regel N werden BKL und nm (Sudoku) nachgeführt.
 [3] ⍝ ns: (1) Box (2) Column (3) Row
 [4] ⍝ v2: BKL nm
 [5] ⍝ (d,r,c): digit,row,column
 [6] ⍝-----------------------------------------------------------------------
 [7]  BOX←ns=1
 [8]  t←ns⊃'N N_B' 'N N_C' 'N N_R'           ⍝ Bezeichnung
 [9]  p←ns⊃(3 2 1) (2 1 3) (3 2 1)           ⍝ Achsenpermutationen (=Umkehrung)
[10]  (BKL nm)←v2
[11]  side←1⊃⍴BKL
[12]  BKL0←BKL
[13] →(~BOX)/L_0
[14]  BKL←bloeckeZU3zeilen BKL    ⍝ speziell für Block-Reduktion
[15] L_0:
[16]  BKL←p ⍉BKL                  ⍝ p: Achsenpermutation BKL
[17]  bm←1=+/[1]BKL               ⍝ In welchen Zellen ist die Ziffer eindeutig?
[18]  BKLunique←BKL∧(⍴BKL)⍴bm     ⍝ In den andern Zellen wird BKL auf 0 gesetzt
[19]  BKL←p ⍉BKL                  ⍝ p: Achsenpermutation BKL
[20]  BKLunique←p ⍉BKLunique      ⍝ p: Achsenpermutation BKLunique
[21] →(~BOX)/L_1
[22]  BKL←bloeckeZU3zeilen BKL               ⍝ speziell für Block-Reduktion
[23]  BKLunique←bloeckeZU3zeilen BKLunique   ⍝ speziell für Block-Reduktion
[24] L_1:
[25]  v←numeriereBKL BKLunique               ⍝ l in BKLunique --> (d,r,c)
[26]  BKL←BKL∧⊃∧/side gen3DFilter¨v          ⍝ Ausputz von BKL
[27]  w←,nm ⋄ w[1+side⊥¯1+⍉⊃1↓¨v]←1⊃¨v       ⍝ nm gemäss v nachführen
[28]  nm←(side,side)⍴w
[29]  BKL←BKL∧genBKL nm                      ⍝ BKL anpassen
[30] w←÷(nm=0)≡∨/[1]BKL           ⍝ Kompatibilität von nm/BKL testen 16.05.14
[31] v2←BKL nm
[32] →(Report=0)/0
[33] →(0=⍴v)/0
[34] ⍝ Protokoll nachführen: Anfang ----------------------------
[35] ⍝=========================================================
[36]  w←,' ',' →'subLin 1 0 2 0 1 0⍕⊃v       ⍝ Zellenangabe
[37]  cpropProgress BKL0 BKL (cm←t,w)        ⍝ l: nm t0R Report
[38] ⍝=========================================================
[39] ⍝ Protokoll nachführen: Ende ------------------------------
```

4.4 Zur Regel Y

Sowohl für die Regel `X` als auch für die Regel `Y` ist die Ermittlung von Ketten nötig. Im Programm `cpropY` werden die verbundenen Zellen durchnummeriert. Eine Kette wird dann durch einen numerischen Vektor dargestellt. Da unser Programm die Ketten, die zur Streichung von Kandidaten führen, im Protokoll vollständig wiedergeben soll, muss es sie ständig mitführen. Das Programm wird dadurch langsamer und kann gelegentlich lange bei der vergeblichen Suche nach Ketten verharren.

Im folgenden Aufrufdiagramm geschieht die Verkettung auf den Zeilen [9] und [10]. Verwendet wird in `joinESn` ein äußeres, in `joinESn_v` ein inneres Produkt.

Letztgenanntes hat als bekannteste Anwendung die gewöhnliche Matrixmultiplikation. In der Form `∨.∧` dient es zur Verkettung von Pfaden in Graphen.

Führt z.B. in einem Graphen mit 7 Ecken ein Pfad von Punkt 2 nach Punkt 5 und einer von Punkt 5 nach Punkt 3, so können diese Verbindungen durch Matrizen `M` und `N` dargestellt werden, und das innere Produkt zeigt dann, dass Punkt 2 mit Punkt 3 verbunden ist:

```
      N←M←7 7⍴0 ⋄ M[2;5]←N[5;3]←1
      ]display M N '   ' (M∨.∧N)
.→-----------------------------------------------.
|.→------------..→------------..→-..→------------.|
|↓0 0 0 0 0 0 0|↓0 0 0 0 0 0 0||  |↓0 0 0 0 0 0 0||
||0 0 0 0 1 0 0||0 0 0 0 0 0 0|'--'|0 0 1 0 0 0 0||
||0 0 0 0 0 0 0||0 0 0 0 0 0 0|   |0 0 0 0 0 0 0||
||0 0 0 0 0 0 0||0 0 0 0 0 0 0|   |0 0 0 0 0 0 0||
||0 0 0 0 0 0 0||0 0 1 0 0 0 0|   |0 0 0 0 0 0 0||
||0 0 0 0 0 0 0||0 0 0 0 0 0 0|   |0 0 0 0 0 0 0||
||0 0 0 0 0 0 0||0 0 0 0 0 0 0|   |0 0 0 0 0 0 0||
|'~------------''~------------'   '~------------'|
'∊-----------------------------------------------'
```

Will man die Pfade (Ketten) mitführen, so muss man sich anstelle der 1 in M[2;5] alle Pfade vorstellen, die von Punkt 2 nach Punkt 5 führen, z.B.

```
      ]display vv1←(2 5) (2 3 7 5) (2 4 5) (2 6 5)
.→---------------------------.
|.→--..→------..→----..→----.|
||2 5||2 3 7 5||2 4 5||2 6 5||          ⍝ Alle Streckenzüge führen von 2 nach 5.
|'~--''~------''~----''~----'|
'∊---------------------------'
```

Und statt der 1 in N[5;3] stehen dann Pfade, die von Punkt 5 nach Punkt 3 gehen, wie beispielsweise:

```
      ]display vv2←(5 7 3) (5 4 3) (5 3)
.→------------------.
|.→----..→----..→--.|
||5 7 3||5 4 3||5 3||                  ⍝ Alle Streckenzüge führen von 5 nach 3.
|'~----''~----''~--'|
'∊------------------'
```

Das Programm joinESn verkettet jeden der Pfade der ersten Sorte mit jedem der zweiten Sorte und ergibt deshalb Pfade, die vom Punkt 2 über Punkt 5 nach Punkt 3 führen:

```
      ]display vv1 joinESn vv2    ⍝ Alle Streckenzüge führen von 2 über 5 nach 3.
.→-------------------------------------------------------------------------------.
|.→------..→------..→----..→--------..→------..→--------..→--------..→------.|
||2 5 7 3||2 5 4 3||2 5 3||2 4 5 7 3||2 4 5 3||2 6 5 7 3||2 6 5 4 3||2 6 5 3||
|'~------''~------''~----''~--------''~------''~--------''~--------''~------'|
'∊-------------------------------------------------------------------------------'
```

Das Programm joinESn verknüpft nun aber lediglich *einen* Punkt in der Matrix M mit *einem* Punkt in der Matrix N. Um aber sämtliche Pfade vom Punkt 2 nach dem Punkt 3 zu erhalten, muss die ganze 2. Zeile von M mit der ganzen 3. Spalte von N verbunden werden, was zusätzlich durch das Programm joinESn_v ausgeführt wird. In den klassischen Fällen +.× und ∨.∧ ist dies nicht nötig, da die verwendetetn APL-Funktionen Vektoren von sich aus elementweise verarbeiten. Es folgen die beiden Programme joinESn und joinESn_v sowie der Aufrufbaum von cpropY:

```
[0] vv←vv1 joinESn vv2;bv;vv0
[1] ⍝------------------------------------------------------------------------
[2] ⍝ ES: "edge sequence" (Streckenzug)
[3] ⍝ vv1,vv2,vv : Vektoren von Streckenzügen (edge sequences)
[4] ⍝ kateniert jeden Streckenzug von vv1 mit jedem von vv2
[5] ⍝ Alle Streckenzüge von vv1 führen von A nach B, und alle von vv2
[6] ⍝ führen von B nach C. Nach der Katenation führen alle Streckenzüge
[7] ⍝ von A nach C.
[8] ⍝------------------------------------------------------------------------
[9]  vv←⍳0
[10] →(vv1≡⍳0)/0
[11] →(vv2≡⍳0)/0
[12]  vv0←,vv1∘.,1↓¨vv2            ⍝ inneres Produkt bezüglich ","
[13]  bv←,⊃distinct¨vv0            ⍝ ohne Wiederholungen
[14] →(~∨/bv)/0
[15]  vv←bv/vv0                    ⍝ Zyklen vermeiden
```

```
[0] v←v1 joinESn_v v2;N;L;I;w
[1] ⍝------------------------------------------------------------------------
[2] ⍝ verarbeitet v1 und v2 elementweise mit "joinESn"
[3] ⍝ Dieser Schritt ist bei +.× und bei ∨.∧ nicht nötig, weil +, ×, ∨, ∧
[4] ⍝  an sich schon elementweise arbeiten.
[5] ⍝ v←v1 joinESn¨ v2
[6] ⍝------------------------------------------------------------------------
[7]  v←⍳0
[8]  N←⍴v1 ⋄ →(N=0)/0 ⋄ v←N⍴⊂⍳0 ⋄ L←(N⍴LOOP),0 ⋄ I←1
[9] LOOP: v[I]←⊂(I⊃v1)joinESn I⊃v2 ⋄ →L[I←I+1]
```

```
[0] cpropY
[1] |Clos_ES_cprop
[2] |-getEdgesY
[3] |--associated
[4] |--indexFRC
[5] |--nub
[6] |-redFRC_PathsY_pp_bmVrb
[7] |--nub
[8] |-getCE<<
[9] |-joinESn_v
10] |--joinESn
11] |-vvExtractFROMm<<
[12] |-pp_Kanten_ij
[13] |--pp_Punkte_ij
[14] |-getCandidates_n
[15] |--indexFRC
[16] |--nub
[17] |-redPathsYorder_cprop
[18] |--getCandidates_n
[19] |--nub
[20] |--getSignatures
[21] |-evalPathsYcprop
[22] ∘--elimCands_cprop
```

Das Programm `Clos_ES_cprop` bringt die Verlängerung der Kantenfolgen (ES: *edge sequence*) zum Abschluss (*closure*). Auf Zeile [9] wird `joinESn_v` aufgerufen.

Der Aufruf ist im nachfolgenden Programm auf Zeile [22] zu finden. Es handelt sich um ein inneres Produkt mit der Rechtsfunktion `joinESn_v` und der Katenation als Linksfunktion. Die verlängerten Streckenzüge (Ketten) müssen anschließend einzeln ausgepackt werden. Das ist weder beim gewöhnlichen Matrixprodukt noch bei ∨.∧ nötig, weil der Einschluss (enclose) von Skalaren keine Wirkung zeigt.

```
 [0] bm3D←Clos_ES_cprop bm3D;BKL;vv2;BKL0;pp;bmVrb;CE;m;m0;vvI;vvRC;cc;w;M;side
 [1] ⍝------------------------------------------------------------------------
 [2] ⍝ Clos: "Closure"
 [3] ⍝   ES: "edge sequence" Streckenzug
 [4] ⍝ bm3D: BKL
 [5] ⍝  vv2: Vektor der Kanten (RC)
 [6] ⍝  vvI: Vektor der möglichen Y-Ketten
 [7] ⍝       (mit pp-Indizes als Ecken) erhalten aus bmVrb
 [8] ⍝ vvRC: Vektor der Streckenzüge (mit (R,C) als Ecken)
 [9] ⍝    m: ⊃m[i;j] ist Vektor sämtlicher Streckenzüge von P_i nach P_j
[10] ⍝       (pp-Indizes) falls keine: ⊂⍳0
[11] ⍝    l: ZAEHLER_SUP_Y  VVlen_SUP_Y DOMINO_EIN
[12] ⍝------------------------------------------------------------------------
[13]  BKL←bm3D ⋄ side←↑⍴BKL
[14]  vv2←2 getEdgesY BKL          ⍝ vv2=vSE4: sequence of edges Form (R,C)
[15]  (pp bmVrb)←redFRC_PathsY_pp_bmVrb vv2
[16] →(0=⍴pp)/L_ENDE
[17]  CE←m←getCE bmVrb             ⍝ CE: "connecting edges" Verbindungsstrecken
[18]  m0←(⍴m)⍴⊂⍳0                  ⍝     jede Kante ein Paar von pp-Indizes
[19]  M←⍳0
[20]  BKL0←BKL
[21] LOOP:
[22]  m←⊃¨m ,.joinESn_v CE         ⍝ Verlängerung der Ketten um 1 Kante (CE)
[23]                               ⍝ m enthält nur Ketten einer best. Länge
[24]  M←M,⊂m
[25]  →(m≡m0)/L_ENDE
[26] ⍝ BKL aufgrund von m anpassen: Anfang ------------------------------------
[27] ⍝=====================================================================
[28]  vvI←vvExtractFROMm m              ⍝ Übergang m --> vvI
[29]  vvRC←pp pp_Kanten_ij vvI side     ⍝ vvI: pp-Indizes vvRC: (R,C)
[30]  cc←(⊂BKL)getCandidates_n¨vvRC
[31] w←÷∧/2=,⊃⍴¨,⊃cc
[32]  vvRC←BKL redPathsYorder_n vvRC    ⍝ vvRC reduzieren und ordnen
[33]  →(0=⍴vvRC)/L_ENDE
[34] ⍝ Kandidaten streichen: Anfang ---------------------------------------
[35] ⍝=====================================================================
[36]  BKL←evalPathsYcprop BKL vvRC      ⍝ entspr. Kandidaten streichen
[37] ⍝=====================================================================
[38] ⍝ Kandidaten streichen: Ende -----------------------------------------
[39] ⍝=====================================================================
[40] ⍝ BKL aufgrund von m anpassen: Ende ---------------------------------
[41] →(BKL≡BKL0)/LOOP
[42] L_ENDE: bm3D←BKL
```

4.5 Alle 7 Regeln

Das folgende Programm `CPropFNBTWYX` wendet die 7 Regeln in der aufgeführten Reihenfolge an. Es kehrt jedesmal zur einfachsten Regel zurück, wenn irgendeine der Regeln zu einer Reduktion von BKL geführt hat. Erfahrung zeigt, dass es sich empfiehlt, die Regel *Y* vor der Regel *X* anzuwenden. Durch Weglassen von Unterprogrammen oder Umstellen der Reihenfolge kann das Programm leicht an eine beliebige Auswahl der Regeln und eine beliebige Reihenfolge angepasst werden. Es liest oder ändert die Variablen `Report`, `RPPR` und `t0R`, welche die Protokollierung betreffen. Diese Variablen werden vom anschließend aufgeführten Programm `CPROP_FNBTWYX` bereitgestellt. Dieses bildet gewissermaßen den Rahmen für die Lösung des Sudokus.

```
    ∇ v2←nv3 CPropFNBTWYX V2;BKL0;tt;BKL;text;nm;BKL0;BKL;cm;side
[1]   ⍝-----------------------------------------------------------------
[8]   ⍝ nv3: [1] obere Grenze für offene Tupel       (Default: 2)
[9]   ⍝      [2] obere Grenze für verborgene Tupel   (Default: 2)
[10]  ⍝      [3] obere Grenze für W                  (Default: 2)
[11]  ⍝  V2: BKL nm
[12]  ⍝ att: F field
[13]  ⍝      N oNly cell (R/C/B)
[14]  ⍝      B box/box interaction (bb1, bb2)
[15]  ⍝      T tupel (inkl. h2)
[16]  ⍝      X X13 x2 (x2: nur Streckenz. mit gerader Anzahl starker Kar
[17]  ⍝      Y y gc   (ohne identische Paare)  gc: "golden chain"
[18]  ⍝      W        SF (inkl. X-Wing)
[19]  ⍝   n: "neu" (neues "red_Yn" statt "red_Y")
[20]  ⍝-----------------------------------------------------------------
[21]   (BKL nm)←V2
[22]   side←1⊃⍴nm
[24]  LOOP:
[26]   nr1←(3 0⍕+/,BKL)subL' 0' ⋄ nr2←(2 0⍕+/,nm>0)subL' 0'
[27]   ⍎'BKL_',nr1,'_',nr2,'←BKL' ⋄ ⍎'nm_',nr1,'_',nr2,'←nm'
[29]  L_F:
[30]   BKL0←BKL ⋄ (BKL nm)←cpropF BKL nm ⋄ nm←nmAUS_BKLnm BKL nm
[31]  →(0 0≡⍴nm)/L_ENDE ⋄ →(0=+/,BKL)/L_ENDE ⋄ →(~BKL≡BKL0)/LOOP
[33]  L_N_B:                            ⍝ 1 cpropN: Box
[34]   BKL0←BKL ⋄ (BKL nm)←1 cpropN BKL nm ⋄ nm←nmAUS_BKLnm BKL nm
[35]  →(0 0≡⍴nm)/L_ENDE ⋄ →(0=+/,BKL)/L_ENDE ⋄ →(~BKL≡BKL0)/LOOP
[37]  L_N_R:                            ⍝ 3 cpropN: Row
[38]   BKL0←BKL ⋄ (BKL nm)←3 cpropN BKL nm ⋄ nm←nmAUS_BKLnm BKL nm
[39]  →(0 0≡⍴nm)/L_ENDE ⋄ →(0=+/,BKL)/L_ENDE ⋄ →(~BKL≡BKL0)/LOOP
[41]  L_N_C:                            ⍝ 2 cpropN: Column
[42]   BKL0←BKL ⋄ (BKL nm)←2 cpropN BKL nm ⋄ nm←nmAUS_BKLnm BKL nm
[43]  →(0 0≡⍴nm)/L_ENDE ⋄ →(0=+/,BKL)/L_ENDE ⋄ →(~BKL≡BKL0)/LOOP
[45]  L_B: BKL0←BKL ⋄ BKL←cpropB BKL ⋄ nm←nmAUS_BKLnm BKL nm
[46]  →(0 0≡⍴nm)/L_ENDE ⋄ →(0=+/,BKL)/L_ENDE ⋄ →(~BKL≡BKL0)/LOOP
[48]  L_T: BKL0←BKL ⋄ BKL←(2↑nv3) cpropT BKL nm ⋄ nm←nmAUS_BKLnm BKL nm
[49]  →(0 0≡⍴nm)/L_ENDE ⋄ →(0=+/,BKL)/L_ENDE ⋄ →(~BKL≡BKL0)/LOOP
[51]  L_W: BKL0←BKL ⋄ BKL←nv3[3] cpropW BKL ⋄ nm←nmAUS_BKLnm BKL nm
[52]  →(0 0≡⍴nm)/L_ENDE ⋄ →(0=+/,BKL)/L_ENDE ⋄ →(~BKL≡BKL0)/LOOP
[54]  L_Y: 0 cpropProgress BKL ((3/side)⍴0) (cm← ' L_Y:')
[55]   BKL0←BKL ⋄ BKL←cpropY BKL ⋄ nm←nmAUS_BKLnm BKL nm
[56]  →(0 0≡⍴nm)/L_ENDE ⋄ →(0=+/,BKL)/L_ENDE ⋄ →(~BKL≡BKL0)/LOOP
[58]  L_X: 0 cpropProgress BKL ((3/side)⍴0) (cm← ' L_X:')
[59]   BKL0←BKL ⋄ BKL←cpropX BKL ⋄ nm←nmAUS_BKLnm BKL nm
[60]  →(0 0≡⍴nm)/L_ENDE ⋄ →(0=+/,BKL)/L_ENDE ⋄ →(~BKL≡BKL0)/LOOP
[62]  L_ENDE: v2←BKL nm
```

```
 [0] v4←nv3 CPROP_FNBTWYX lv;t0R;RPPR;BKL;nm;BKL0;nm0;tupel;Report
 [1] ⍝-------------------------------------------------------------------------
 [2] ⍝ nv3: [1] obere Grenze für offene Tupel       (Default: 2)
 [3] ⍝      [2] obere Grenze für verborgene Tupel   (Default: 2)
 [4] ⍝      [3] obere Grenze für W-Muster           (Default: 2)
 [5] ⍝  lv: Sudoku-Name
 [6] ⍝ Regeln gemäss "att": www.afjarvis.staff.shef.ac.uk/sudoku/sudoku.pdf
 [7] ⍝                      J.S. Fowler: A 9×9 sudoku solver and generator
 [8] ⍝ att: F field
 [9] ⍝      N oNly cell (B/R/C)
[10] ⍝      B box/row box/column interaction
[11] ⍝      T tupel
[12] ⍝      X X13 X_2
[13] ⍝      Y
[14] ⍝      W (inkl. X-Wing)
[15] ⍝-------------------------------------------------------------------------
[16]  Report←1                              ⍝ Initialisierung Report
[17]  t0R←sec                               ⍝ Initialisierung t0R
[18]  RPPR←0 80⍴''                          ⍝ Initialisierung Protokoll
[19]  →(2=⎕nc'nv3')/L_0 ⋄ nv3←7 7 7         ⍝ 7 7 7: Default für nv3
[20] L_0:
[21]  (nv3 BKL nm)←nv3 cpropStart lv        ⍝ BKL aufgrund von nm (Sudoku)
[22]  (BKL0 nm0)←BKL nm
[23]  tupel←genTupel 1 2 3 [1 4 9⍳↑⍴nm]     ⍝ TUPEL (für Regeln T und W)
[24] ⍝ Constraint Propagation: Anfang ---------------------------------
[25] ⍝=================================================================
[26]  (BKL nm)←nv3 CPropFNBTWYX BKL nm      ⍝ l: Report t0R  ä: RPPR
[27] ⍝=================================================================
[28] ⍝ Constraint Propagation: Ende -----------------------------------
[29]  lv cpropEnde nm0 nm BKL0 BKL
[30] v4←(nm0 nm BKL RPPR)
```

Das folgende Protokoll wird vom Programm CPROP_FNBTWYX hergestellt. Es wird während des Ablaufs auf den Bildschirm geschrieben und ist zugleich in der Variablen RPPR enthalten. Es wird nachfolgend stückweise wiedergegeben:

```
    (nm0 nm BKL RPPR)←CPROP_FNBTWYX '∆sdk9_tbz_071112'

[0] ∆sdk9_tbz_071112                                          0.0 cpropStart[22]------
[1] ========================================                  0.0 cpropStart[23]------
[2] N N_B 3→79 4→39 5→47                    28 189  ¯18       0.0 cpropN[48]----------
[3] N N_B 7→29                              29 182   ¯7       0.0 cpropN[48]----------
[4] B B>R cn=1/b=13/r=1   17 18                               0.0 cpropBR[48]---------
[5]   12-1 13-1 15-1                        29 179   ¯3       0.0 cpropBR[50]---------
```

[2]: In Zelle (7, 9) kann 3 gesetzt werden, weil der Kandidat 3 im betreffenden Block nur einmal vorkommt. Aus demselben Grund kann in Zelle (3, 9) die Ziffer 4 und in Zelle (4, 7) die Ziffer 5 gesetzt werden. [4], [5]: In Block $B_{1,3}$ konzentriert sich der Kandidat 1 auf die 1. Zeile, wo er in den Zellen (1, 7) und (1, 8) vorkommt. Deshalb kann er nach Regel B ➢ R in den Zellen (1, 2), (1, 3) und (1, 5) gestrichen werden.

Ganz am rechten Rand ersieht man das jeweils aktuelle Unterprogramm und die Zeilennummer. Die vier Zahlkolonnen davor bedeuten: die Anzahl der bisher bestimmten Ziffern, die Anzahl der in der BKL verbleibenden Kandidaten sowie die jeweilige Anzahl der gestrichenen Kandidaten, und schließlich die Zeit in Sekunden, die seit dem Start des Programms verflossen ist.

```
 [6] N N_B 1→35                             30 172   ¯7       0.0 cpropN[48]----------
 [7] B B>R cn=6/b=13/r=1   17 18 19                             0.0 cpropBR[48]---------
 [8]   12-6                                 30 171   ¯1       0.0 cpropBR[50]---------
 [9] B B>R cn=8/b=13/r=1   17 18 19                             0.1 cpropBR[48]---------
[10]   12-8 14-8 15-8                       30 168   ¯3       0.1 cpropBR[50]---------
[11] B B>R cn=9/b=31/r=9   92 93                               0.1 cpropBR[48]---------
[12]   95-9 96-9 98-9                       30 165   ¯3       0.1 cpropBR[50]---------
[13] B B>C cn=2/b=33/c=7   77 97                               0.1 cpropBC[53]---------
[14]   57-2                                 30 164   ¯1       0.1 cpropBC[55]---------
[15] B B>C cn=5/b=31/c=1   71 81                               0.1 cpropBC[53]---------
[16]   21-5                                 30 163   ¯1       0.1 cpropBC[55]---------
[17] T hT2 (c=2) 22 92 {16}                                    0.2 cprop_hTupel[36]----
[18]   22-8 22-9 92-7 92-9                  30 159   ¯4       0.2 cprop_hTupel[38]----
```

In Spalte 2 liegt ein verstecktes Paar (h: „hidden“) vor. Es handelt sich um das Paar {1; 6} in den Zellen (2; 2) und (9; 2). Gemäß Regel *T* können in diesen beiden Zellen die übrigen Kandidaten gestrichen werden: In Zelle (2;2) sind es die Kandidaten 8 und 9, in Zelle (9; 2) die Kandidaten 7 und 9.

```
[19] N N_B 9→93                             31 154   ¯5       0.2 cpropN[48]----------
[20] N N_B 9→12                             32 150   ¯4       0.2 cpropN[48]----------
[21] N N_C 7→33                             33 146   ¯4       0.2 cpropN[48]----------
[22] N N_C 7→42                             34 142   ¯4       0.2 cpropN[48]----------
[23] B B>R cn=2/b=11/r=3   31 32                               0.2 cpropBR[48]---------
[24]   36-2                                 34 141   ¯1       0.2 cpropBR[50]---------
[25]  L_Y:                                                     0.6 CPropFNBTWYX[54]----
[26]  L_X:                                                     0.6 CPropFNBTWYX[58]----
[27] X X13 (3) 36=31=13=63                                     7.4 cpropX_bm[83]-------
[28]   66-3                                 34 140   ¯1       7.4 cpropX_bm[85]-------
```

[25]: Das Programm sucht nach Y-Ketten,- findet aber keine.
[26]-[28]: Das Programm sucht nach X-Ketten und findet auch eine für den Kandidaten 3. Es handelt sich um eine einfache X_1-Kette aus den Zellen (3, 6), (3, 1), (1, 3) und (6, 3). Die 3 Gleichheitszeichen stehen für 3 starke Kanten. Da Zelle (6, 6) mit beiden Enden der Kette assoziiert ist, kann dort der Kandidat 3 gestrichen werden.

```
[29]  L_Y:
[30]  L_X:
[31] X X13 (3) 63=13=31=36=96=95
[32]   65-3
[33] N N_R 3→63
.... F F F F F F F
[41] N N_B 4→68
[42] N N_B 4→87
[43] N N_B 1→88
[44] N N_B 1→17
[45] N N_C 3→96
[46] B C>B cn=2/b=22/c=6   56 66
[47]   54-2 65-2
[48] F --- 5→65
[49] T oT2 (c=6) 26 86 {59}
[50]   56-9
```

```
                    7.7 CPropFNBTWYX[54]----
                    7.7 CPropFNBTWYX[58]----
                   17.4 cpropX_bm[83]-------
34 139   ¯1        17.4 cpropX_bm[85]-------
35 135   ¯4        17.4 cpropN[48]----------

48  88   ¯3        17.5 cpropN[48]----------
49  85   ¯3        17.5 cpropN[48]----------
50  80   ¯5        17.5 cpropN[48]----------
51  77   ¯3        17.5 cpropN[48]----------
52  74   ¯3        17.5 cpropN[48]----------
                   17.5 cpropCB[50]---------
52  72   ¯2        17.5 cpropCB[52]---------
53  69   ¯3        17.5 cpropF[30]----------
                   17.5 cprop_oTupel[38]----
53  68   ¯1        17.5 cprop_oTupel[40]----
```

[29]-[32]: Erneut sucht das Programm vergeblich nach einer Y-Kette. Dann findet es für den Kandidaten 3 eine weitere (ebenfalls einfache) X_1-Kette mit 5 starken Kanten, welche es erlaubt, den Kandidaten in der Zelle (6, 5) zu streichen.

[49],[50]: In Spalte 6, nämlich in den Zellen (2, 6) und (8, 6), befindet sich das offene Tupel {5, 9}. Also kann nach Regel *T* der Kandidat 9 in Zelle (5, 6) gestrichen werden.

```
[51] W X2C cn=9/cc=58/rr=47
[52]   44-9 74-9
[53]  L_Y:
[54] Y (9) 48-57-77-75
[55]       98 86 62 29
[56]   45-9 78-9
[57] N N_B 9→54 9→89
.... F F F F F F F F
[66] Anfang und Schluss:
```

```
                   17.8 cpropSF_R_bm_t[46]--
53  66   ¯2        17.8 cpropSF_R_bm_t[48]--
                   18.1 CPropFNBTWYX[54]----
                   18.2 elimCands_cprop[24]-
                   18.2 elimCands_cprop[26]-
53  64   ¯2        18.2 elimCands_cprop[28]-
55  56   ¯8        18.2 cpropN[48]----------

81   0 ¯207        18.3 cpropEnde[20]-------
```

[51], [52]: Der Kandidat 9 kommt in den Spalten 5 und 8 nur in den Zeilen 4 und 7 vor. Er kann deshalb nach Regel *W* in den Zellen (4, 4) und (7, 4) gestrichen werden.

[53]-[56]: Nun ist die erneute Suche nach einer Y-Kette erfolgreich. Sie wird aus den Zellen (4, 8), (5, 7), (7, 7), (7, 5) gebildet, und die Kandidatenpaare, richtig angeordnet, lauten: {9, 8}, {8, 6}, {6, 2} und {2, 9}.

Es handelt sich somit um eine Y-Kette für den Kandidaten 9. Gemäß Regel *Y* kann nun der Kandidat in den Zellen (4, 5) und (7, 8) gestrichen werden.

```
[67] 4 0 0 | 0 0 7 | 0 0 0 || 4 9 5 | 2 3 7 | 1 8 6
[68] 0 0 0 | 0 4 0 | 3 2 0 || 8 6 1 | 5 4 9 | 3 2 7
[69] 0 0 0 | 6 0 0 | 9 5 0 || 3 2 7 | 6 1 8 | 9 5 4
[70] --------------------- || ---------------------
[71] 0 0 6 | 0 0 4 | 0 0 1 || 2 7 6 | 3 8 4 | 5 9 1
[72] 0 5 0 | 0 7 0 | 0 3 0 || 1 5 4 | 9 7 2 | 6 3 8
[73] 9 0 0 | 1 0 0 | 7 0 0 || 9 8 3 | 1 5 6 | 7 4 2
[74] --------------------- || ---------------------
[75] 0 4 8 | 0 0 1 | 0 0 0 || 5 4 8 | 7 9 1 | 2 6 3
[76] 0 3 2 | 0 6 0 | 0 0 0 || 7 3 2 | 8 6 5 | 4 1 9
[77] 0 0 0 | 4 0 0 | 0 0 5 || 6 1 9 | 4 2 3 | 8 7 5
[78] 1 1 1 25              || 1 1 1 81
```

Im weiteren genügen die Regeln *N* und *F* zur Vervollständigung. Am Ende des Protokolls erscheinen das gegebene und das vervollständigte Sudoku nebeneinander. Es zeigt sich, dass das gegebene 25, das vervollständigte 81 eingefüllte Ziffern aufweist.

5 Noch ein rekursives Programm

Das Pseudo-Sudoku rechts ist seit längerem dafür bekannt, dass es genau zwei Lösungen hat. Es hat nur 16 Vorgaben, und trotz langer Suche hat man nie ein Sudoku mit weniger als 17 Vorgaben gefunden. Heute sind etwa 50 000 solcher 17er-Sudokus bekannt; die meisten sind auf ROYLE[7] zusammengestellt. Ende 2012 ist in MCGUIRE/TUGEMANN/CIVARIO[6] ein computergestützter Beweis erschienen, wonach es keine Sudokus mit weniger als 17 Vorgaben gibt.

Mit dem Sudoku rechts kommt unser Programm `rek_prim` nicht zurande. Das leicht erweiterte Programm `rek_FNBWT` hingegen, welches zusätzlich zur Rekursion fortgesetzte Einschränkung mit den Regeln *F* , *N* , *B*, *T* , *W* benützt, hat keine Probleme:

5		2				4		
			7	1				3
					4	6		
	7		2					
	1							
6					2			
				3			1	
4								

sdk 23

```
      t0←sec⋄ osDEB¨zeig¨(⊂⍋sdk9_16_2),v←rek_FNBTW ⍋sdk9_16_2 ⋄osDEB 8 1⍕sec-t0
  5 0 2 | 0 0 0 | 4 0 0   5 6 2 | 3 8 9 | 4 7 1   5 6 2 | 3 9 8 | 4 7 1
  0 0 0 | 7 1 0 | 0 0 3   8 4 9 | 7 1 6 | 2 5 3   9 4 8 | 7 1 6 | 2 5 3
  0 0 0 | 0 0 0 | 0 0 0   1 3 7 | 4 2 5 | 8 9 6   1 3 7 | 4 2 5 | 9 8 6
  ---------------------   ---------------------   ---------------------
  0 0 0 | 0 0 4 | 6 0 0   3 5 8 | 1 9 4 | 6 2 7   3 5 9 | 1 8 4 | 6 2 7
  0 7 0 | 2 0 0 | 0 0 0   9 7 4 | 2 6 3 | 1 8 5   8 7 4 | 2 6 3 | 1 9 5
  0 1 0 | 0 0 0 | 0 0 0   2 1 6 | 8 5 7 | 3 4 9   2 1 6 | 9 5 7 | 3 4 8
  ---------------------   ---------------------   ---------------------
  6 0 0 | 0 0 2 | 0 0 0   6 9 1 | 5 4 2 | 7 3 8   6 8 1 | 5 4 2 | 7 3 9
  0 0 0 | 0 3 0 | 0 1 0   7 2 5 | 6 3 8 | 9 1 4   7 2 5 | 6 3 9 | 8 1 4
  4 0 0 | 0 0 0 | 0 0 0   4 8 3 | 9 7 1 | 5 6 2   4 9 3 | 8 7 1 | 5 6 2
  1 1 1 16                1 1 1 81                1 1 1 81
 0.7


      osDEB¨zeig¨(≠/v)×v
0 0 0 | 0 8 9 | 0 0 0   0 0 0 | 0 9 8 | 0 0 0
8 0 9 | 0 0 0 | 0 0 0   9 0 8 | 0 0 0 | 0 0 0
0 0 0 | 0 0 0 | 8 9 0   0 0 0 | 0 0 0 | 9 8 0
---------------------   ---------------------
0 0 8 | 0 9 0 | 0 0 0   0 0 9 | 0 8 0 | 0 0 0      ⍝ Die beiden Lösungen
9 0 0 | 0 0 0 | 0 8 0   8 0 0 | 0 0 0 | 0 9 0      ⍝ entstehen auseinander durch
0 0 0 | 8 0 0 | 0 0 9   0 0 0 | 9 0 0 | 0 0 8      ⍝ Vertauschen von 8 und 9
---------------------   ---------------------
0 9 0 | 0 0 0 | 0 0 8   0 8 0 | 0 0 0 | 0 0 9
0 0 0 | 0 0 8 | 9 0 0   0 0 0 | 0 0 9 | 8 0 0
0 8 0 | 9 0 0 | 0 0 0   0 9 0 | 8 0 0 | 0 0 0
1 1 1 18                1 1 1 18
```

```
  [0] v_nm←rek_FNBTW nm;side;a;min;Report;c;r;BKL;w;N;L;I;nmi;cc;base;tupel;nv3
  [1] ⍝-----------------------------------------------------------------------
  [2] ⍝   lv: Pseudoudoku-Name (lv: literaler Vektor)
  [3] ⍝ v_nm: alle Vervollständigungen (Vektor von numerischen Matrizen)
  [4] ⍝-----------------------------------------------------------------------
  [5]  Report←0
  [6]  tupel←genTupel 1 2 3[1 4 9⍳↑⍴nm]
  [7]  v_nm←⍳0
  [8] →(~∧/konformZSB nm)/0                ⍝ Zeilen-/Spalten-/Block-Bedingung?
  [9] →(~∧/,nm>0)/L_0                      ⍝ noch nicht vollständig?
 [10]  v_nm←v_nm,⊂nm ⋄ →0                  ⍝ vollständig: Lösung hinzufügen
 [11] L_0:
 [12]  side←↑⍴nm ⋄ BKL←genBKL nm
 [13]  nv3←7 7 7⌊side-2
 [14]  (BKL nm)←nv3 CPropFNBTW BKL nm   ⍝ constraint propagation
 [15] →(~∧/,nm>0)/L_1                      ⍝ noch nicht vollständig?
 [16]  v_nm←v_nm,⊂nm ⋄ →0                  ⍝ vollständig: Lösung hinzufügen
 [17] L_1:
 [18]  a←,+/[1]BKL                         ⍝     a: Kandidatenzahlen
 [19]  min←⌊/a~0                           ⍝   min: minimale (ausser 0)
 [20]  (c r)←1+(side,side)⊤¯1+a⍳min        ⍝ (c r): "row,column" erste Zelle mit
 [21]                                      ⍝  dieser minimalen Kandidatenzahl
 [22]  cc←w/⍳⍴w←BKL[;c;r]                  ⍝    cc: Kandidaten in dieser Zelle
 [23]  N←⍴cc ⋄ →(N=0)/0
 [24]  L←(N⍴LOOP),0 ⋄ I←1
 [25] LOOP:
 [26]  nmi←nm                              ⍝ Kandidaten einen nach dem andern
 [27]  nmi[c;r]←I⊃cc                       ⍝ in (c,r) einsetzen
 [28]  v_nm←v_nm,rek_FNBTW nmi             ⍝ rekursiver Aufruf ("back tracking")
 [29] →L[I←I+1]
```

Mit diesem Programm kann man auch nachweisen, dass es keine 4×4-Sudokus mit weniger als 4 Vorgaben gibt. Hingegen fehlt es nicht an 4 × 4-Sudokus mit 4 Vorgaben.

6 LaTeX mit APL

APL eignet sich ausgezeichnet, um mit der `picture`-Umgebung von LaTeX Sudoku-Darstellungen zu erzeugen. Um z.B. das Sudoku `sdk_11` samt Kandidatenliste zu erhalten, genügen die untenstehenden Befehle. Dabei bedeutet `BKL_pos` die Liste der normal dargestellten, `BKL_neg` die der ausgekreuzten und `BKL_cir` die der eingekreisten Kandidaten. Das Programm `ZeigZiffernKandidatenLaTeX` erzeugt die gesamte `picture`-Umgebung. Das erste Linksargument (hier 8.5) bestimmt die Zellengrösse in mm, das zweite die Art der Fusszeile.

```
      zeig sdk11
0 9 0 ¦ 0 0 8 ¦ 1 6 0
0 0 1 ¦ 7 0 0 ¦ 9 2 0
2 0 6 ¦ 0 1 0 ¦ 0 0 3
---------------------
3 0 8 ¦ 4 0 1 ¦ 5 0 6
0 4 9 ¦ 8 0 0 ¦ 2 3 0
6 0 0 ¦ 0 0 7 ¦ 8 0 4
---------------------
8 0 0 ¦ 6 4 0 ¦ 0 0 9
0 0 0 ¦ 0 7 2 ¦ 6 0 0
0 6 7 ¦ 1 8 0 ¦ 0 5 2
1 1 1 40
```

Anschließend folgt ein Ausschnitt aus der erzeugten Textmatrix `lm`:

```
[0]       BKL_pos←genBKL sdk11
[1]       BKL_neg←9 9 9⍴0 ⋄ BKL_neg[4;3;6]←1
[2]       BKL_cir←9 9 9⍴0 ⋄ BKL_cir[4;3;7]←BKL_cir[4;3;8]←1
[3]       lm←8.5 2 ZeigZiffernKandidatenLaTeX 'sdk11' BKL_pos BKL_neg BKL_cir
```

```
[0] \setlength{\unitlength}{8.5mm}
[1] \begin{minipage}[t]{9\unitlength}
[2] \begin{picture}(9,9)
[3]   \linethickness{.1mm}
[4]   \multiput(0,0)(0,1){10}{\line(1,0){9}}
[5]   \multiput(0,0)(1,0){10}{\line(0,1){9}}
[6]   \linethickness{.3mm}
[7]   \multiput(0,0)(0,3){4}{\line(1,0){9}}
[8]   \multiput(0,0)(3,0){4}{\line(0,1){9}}
[9] % Feld (1,1)
[10]   \put(0.1666666666666666,8.5){\makebox(0,0){\scriptsize 4}}
[11]   \put(0.5,8.5){\makebox(0,0){\scriptsize 5}}
[12]   \put(0.1666666666666666,8.166666666666656){\makebox(0,0){\scriptsize 7}}
[13] % Feld (1,2)
[14]   \put(1.5,8.5){\makebox(0,0){\textbf{\Large 9}}}
[15] ..........................................................................
[16] % Feld (9,9)
[17]   \put(8.5,0.5){\makebox(0,0){\textbf{\Large 2}}}
[18] \end{picture}
[19] \vspace{2mm}
[20] \textbf{sdk \thesdk} \stepcounter{sdk} \hfill\rule{0pt}{0pt}
[21] \end{minipage}
```

Literatur

[1] Bertram Felgenhauer and Frazer Jarvis: Enumerating possible Sudoku grids. URL: http://www.afjarvis.staff.shef.ac.uk/sudoku/sudoku.pdf

[2] J. S. Fowler: A 9×9 sudoku solver and generator. URL: http://www2.research.att.com/~gsf/sudoku/sudoku.html

[3] Urs Oswald: Sudoku - A Tutorial. URL: http://www.ursoswald.ch/download/TUTORIAL.pdf

[4] Urs Oswald: Sudoku-Programme in Aufrufordnung. URL: http://www.ursoswald.ch/download/APLSUDOPRGaufruf.pdf

[5] Urs Oswald: Sudoku-Programme in alphabetischer Ordnung. URL: http://www.ursoswald.ch/download/APLSUDO PRGalphab.pdf

[6] Gary McGuire, Bastian Tugemann, Gilles Civario: There is no 16-Clue Sudoku: Solving the Sudoku Minimum Number of Clues Problem via Hitting Set Enumeration. URL: http://www.arxiv.org/pdf/1201.0749.pdf

[7] Gordon Royle: A collection of 49,151 distinct sudoku configurations with 17 entries. URL: http://www.csse.uwa.edu.au/~gordon/sudokumin.php

Kontakt:
Dr. sc. math. (ETH) Urs Oswald
Internet: http://www.ursoswald.ch/
eMail: osurs@bluewin.ch

Urs Oswald

APL, Unicode und UTF-8

1 Unicode

Nach der Idee von *Unicode* soll jedem auf der Welt gebräuchlichen *Zeichen* (englisch: *character*) eine natürliche Zahl, also eine der Zahlen 0, 1, 2, . . . zugeordnet werden. Diese Zahl wird der *Codepunkt* (englisch: *codepoint*) des Zeichens genannt. Dabei ist „Zeichen" im abstrakten Sinne zu verstehen. Zum Beispiel sieht der Großbuchstabe *A* verschieden aus, je nachdem, ob er in Arial, Courier, Tahoma, Fraktur oder von Hand geschrieben ist; es handelt sich aber immer um ein und dasselbe Zeichen. Unterschiedlich sind nur die *Glyphen*, mit denen es graphisch dargestellt wird. Anderseits werden in Unicode auch Zeichen einbezogen, die keine Glyphen haben, wie zum Beispiel der Tabulator (Codepunkt 9), der Zeilenvorschub (Line Feed, LF, Codepunkt 10), der Zeilenumbruch (Carriage Return, CR, Codepunkt 13) oder das Leerzeichen (Blank, Codepunkt 32). Für mehr Einzelheiten siehe zum Beispiel [2].

Die Zuordnung der Zeichen zu ihren Codepunkten soll umkehrbar eindeutig (bijektiv) erfolgen. Das heißt, dass nicht nur jedem Zeichen eindeutig ein Codepunkt, sondern auch zwei verschiedenem Zeichen immer zwei verschiedene Codepunkte zugeordnet werden. So führt jeder Codepunkt eindeutig zurück zu einem Zeichen. Die Sache wird aber dadurch kompliziert, dass ein und dieselbe Glyphe verschiedene Zeichen ausdrücken kann. Zum Beispiel kommt die Glyphe H sowohl im lateinischen als auch im kyrillischen Alphabet vor. Der betreffende Buchstabe wird aber im kyrillischen Sprachbereich als N ausgesprochen. Man hat deshalb beschlossen, diese Glyphe als Darstellung zweier verschiedener Zeichen zu betrachten, welchen man die Codepunkte 72 und 1053 zuordnet.

Seit der Veröffentlichung von Unicode 2.0 im Juli 1996 wird für Codepunkte das Intervall 0, . . . , 1 114 111 vorgesehen. Von diesen 1 114 112 Zahlen sind 2 114 für Spezialzwecke blockiert, so dass schließlich 1 111 998 Codepunkte für Zeichen verbleiben.

Die Zahl 1 114 112 ergibt sich aus der Aufteilung des gesamten Bereiches in 17 *Ebenen* (englisch: *planes*) zu je 2^{16} aufeinanderfolgenden Zahlen: $1\,114\,112 = 17 \times 65\,536$.[1]

Die Codepunkte der von den verschiedenen APL-Produkten verwendeten Zeichen liegen alle in der ersten Ebene; der höchste Codepunkt ist 9675 für das Kreissymbol ∘.

1 Für mehr Einzelheiten siehe zum Beispiel [1].

2 APL und Unicode

Die verschiedenen APL-Produkte ordnen dem Atomic Vector die Unicode-Codepunkte in unterschiedlicher Weise zu. Wir bezeichnen die entsprechenden Vektoren gemäß ihrer Herkunft, zum Beispiel mit ΔavucAPX (APLX), ΔavucIBM (IBM-APL2), ΔavucWIN (APL+WIN). In APL+Win erhält man:

```
      ΔavucWIN[⎕av⍳'A○⍴◇']
65 9675 9076 8900

      ⎕av[ΔavucWIN⍳65 9675 9076 8900]
A○⍴◇
```

Durch die Codepunkte 65, 9675, 9076 und 8900 ist nun die Zeichenfolge eindeutig bestimmt. Wer in APLX arbeitet, erhält mit denselben Codepunkten und mit ΔavucAPX

```
      ⎕av[ΔavucAPX⍳65 9675 9076 8900]
A○⍴◇
```

In IBM-APL2 ergeben dieselben Codepunkte mit ΔavucIBM

```
      ⎕av[ΔavucIBM⍳65 9675 9076 8900]
A○⍴◇
```

Um also zum Beispiel eine APL-Funktion von APL+Win nach APLX zu übertragen, braucht man nur den entsprechenden Vektor der Codepunkte auf irgendeine Weise (etwa als Text) zu übertragen. Die Abbildungsvektoren ΔavucAPX, ΔavucIBM und ΔavucWIN lassen sich mit den folgenden drei APL-Funktionen genUC_APX, genUC_IBM, und genUC_WIN erzeugen:

```
  [0] nv256←genUC_APX;w
  [1] ⍝------------------------------------------------------------------------
  [2] ⍝ Quelle: http://www.microapl.com/apl_help/ch_210.htm
  [3] ⍝------------------------------------------------------------------------
  [4]  w←⍳0
  [5]  w←w,    0    1    2    3    4    5    6 9040    8 9047   10 9031
  [6]  w←w, 9032   13 9073 9074 9042 9035 9021 9033 8854 9055 9014 9067
  [7]  w←w, 9038 9045 9024 9023 9053 9054   33 9017   32  168   41   60
  [8]  w←w, 8804   61   62   93 8744   94 8800  247   44   43   46   47
  [9]  w←w,   48   49   50   51   52   53   54   55   56   57   40   91
 [10]  w←w,   59  215   58   92  175 9082 8869 8745 8970 8714   95 8711
 [11]  w←w,  123 9075 8728   39 9109  124 8868 9675   42   63 9076 8968
 [12]  w←w,  126 8595 8746 9077 8835 8593 8834 8592 8866 8594 8805   45
 [13]  w←w, 8900   65   66   67   68   69   70   71   72   73   74   75
 [14]  w←w,   76   77   78   79   80   81   82   83   84   85   86   87
 [15]  w←w,   88   89   90 8710 8867 9066   36  125  128  129  130  131
 [16]  w←w,  132  133  134  135  136  137  138  139  140  141  142  143
 [17]  w←w, 9484 9488 9492 9496 9472 9474 9532 9500 9508 9524 9516   27
 [18]  w←w,   28  205   30   31   34   35   37   38   64  163   96 8801
 [19]  w←w, 8802 9079 9080 9019 9026 9060 9061 9015  196  197  199  201
 [20]  w←w,  209  214  220  225  224  226  228  227  229  231  233  232
 [21]  w←w,  234  235  237  236  238  239  241  243  242  244  246  245
 [22]  w←w,  250  249  251  252  192  195  213  338  339  198  230 9068
 [23]  w←w,  216  248  191  161  223  255    0    0    0   97   98   99
 [24]  w←w,  100  101  102  103  104  105  106  107  108  109  110  111
 [25]  w←w,  112  113  114  115  116  117  118  119  120  121  122 9049
 [26]  w←w,  200 8364    0  127    0    0    0    0    0    0    0    0
 [27] nv256←256↑w
```

```
  [0] nv256←genUC_IBM;w
  [1] ⍝------------------------------------------------------------------------
  [2] ⍝ Quelle: erzeugt mit "Quad-UCS" in APL2
  [3] ⍝------------------------------------------------------------------------
  [4]  w←⍳0
  [5]  w←w,     0     1     2     3     4     5     6     7     8     9    10    11
  [6]  w←w,    12    13    14    15    16    17    18    19    20    21    22    23
  [7]  w←w,    24    25    26    27    28    29    30    31    32    33    34    35
  [8]  w←w,    36    37    38    39    40    41    42    43    44    45    46    47
  [9]  w←w,    48    49    50    51    52    53    54    55    56    57    58    59
 [10]  w←w,    60    61    62    63    64    65    66    67    68    69    70    71
 [11]  w←w,    72    73    74    75    76    77    78    79    80    81    82    83
 [12]  w←w,    84    85    86    87    88    89    90    91    92    93    94    95
 [13]  w←w,    96    97    98    99   100   101   102   103   104   105   106   107
 [14]  w←w,   108   109   110   111   112   113   114   115   116   117   118   119
 [15]  w←w,   120   121   122   123   124   125   126   127   199   252   233   226
 [16]  w←w,   228   224   229   231   234   235   232   239   238   236   196   197
 [17]  w←w,  9109  9054  9017   244   246   242   251   249  8868   214   220   248
 [18]  w←w,   163  8869  8359  9014   225   237   243   250   241   209   170   186
 [19]  w←w,   191  8968   172   189  8746   161  9045  9038  9617  9618  9619  9474
 [20]  w←w,  9508  9055  8710  8711  8594  9571  9553  9559  9565  8592  8970  9488
 [21]  w←w,  9492  9524  9516  9500  9472  9532  8593  8595  9562  9556  9577  9574
 [22]  w←w,  9568  9552  9580  8801  9080  9079  8757  9015  9026  9019  8866  8867
 [23]  w←w,  8900  9496  9484  9608  9604   166   204  9600  9082   223  8834  8835
 [24]  w←w,  9053  9074  9076  9073  9021  8854  9675  8744  9075  9033  8714  8745
 [25]  w←w,  9023  9024  8805  8804  8800   215   247  9049  8728  9077  9067  9035
 [26]  w←w,  9042   175   168   160     0     0     0     0     0     0     0     0
 [27] nv256←256↑w
```

```
  [0] nv256←genUC_WIN;w
  [1] ⍝------------------------------------------------------------------------
  [2] ⍝ Quelle: http://aplwiki.com/AplToUnicode
  [3] ⍝ korrigierte Version
  [4] ⍝------------------------------------------------------------------------
  [5]  w←⍳0
  [6]  w←w,     0     0  8364  9079  8900   168  8592     7     8     9    10  8834
  [7]  w←w,    12    13  8835  9055   229   230   236  9067   249   242  9068  9077
  [8]  w←w,  8593  8595  8594    27  8867  8866  9035  9042    32    33    34    35
  [9]  w←w,    36    37    38    39    40    41    42    43    44    45    46    47
 [10]  w←w,    48    49    50    51    52    53    54    55    56    57    58    59
 [11]  w←w,    60    61    62    63    64    65    66    67    68    69    70    71
 [12]  w←w,    72    73    74    75    76    77    78    79    80    81    82    83
 [13]  w←w,    84    85    86    87    88    89    90    91    92    93    94    95
 [14]  w←w,    96    97    98    99   100   101   102   103   104   105   106   107
 [15]  w←w,   108   109   110   111   112   113   114   115   116   117   118   119
 [16]  w←w,   120   121   122   123   166   125   126  8802   199   252   233   226
 [17]  w←w,   228   224  8800   231   234   235   232   239   238  8968   196  8970
 [18]  w←w,   201  8710   215   244   246  9109   251  9054  9017   214   220   162
 [19]  w←w,   163   165  9066  9064   225   237   243   250   241   209  9053  9024
 [20]  w←w,   191  9015   337   248   253   161   171   187  9109  9109  9109     0
 [21]  w←w,     0     0     0    43    43     0     0    43    43    43    43    43
 [22]  w←w,   192   193   194   195   196   197   198   199   200   201   202   203
 [23]  w←w,   204   205   206   207    45   209   210   211   212   213   214    43
 [24]  w←w,   216   217   218   219   220   221     0   255  9082   223  9075  9060
 [25]  w←w,   227  9073  8869  8868  9021  8854  9074  9023  8711  9033  8714  8745
 [26]  w←w,  8801  9049  8805  8804  9045  9038   247    34  8728  9675  8744  9076
 [27]  w←w,  8746   175   124     0     0     0     0     0     0     0     0     0
 [28] nv256←256↑w
```

Da der Diamant an unterschiedlichen Stellen des Atomic Vector untergebracht ist, kann mit der folgenden Funktion automatisch der richtige Zuordnungsvektor erzeugt werden:.

```
  [0] nv256←genUC
  [1] ⍝------------------------------------------------------------------------
  [2] →(69 216 4 = (-⎕io)+⎕av⍳'◇')/L_APX,L_IBM,L_WIN
  [3] L_APX: nv256←genUC_APX ◇ →0
  [4] L_IBM: nv256←genUC_IBM ◇ →0
  [5] L_WIN: nv256←genUC_WIN
```

2.1 Vergleich der Codepunkt-Vektoren

Wie unten ersichtlich ist, enthalten ΔavucAPX, ΔavucIBM und ΔavucWIN insgesamt 316 verschiedene Codepunkte. Diese können in 7 paarweise elementfremde Gebiete Δavuc_001, Δavuc_010,Δavuc_011,Δavuc_100, Δavuc_101, Δavuc_110, Δavuc_111 aufgeteilt werden. Die Indizes 1, 0, 1 bedeuten zum Beispiel, dass Δavuc_101 genau die Codepunkte enthält, welche in ΔavucAPX und ΔavucWIN, nicht aber in ΔavucIBM vorkommen.

```
      ⍴nub ⍋avucAPX,⍋avucIBM,⍋avucWIN
316
      av,anz←,⊃⍴¨⍋¨⊂[2]av←'⍋avuc_'ANFANG ⎕nl 2
⍋avuc_001        20                    ⍝ nur   ⍋avucWIN
⍋avuc_010        39                    ⍝ nur   ⍋avucIBM
⍋avuc_011         5                    ⍝ nicht ⍋avucAPX
⍋avuc_100        24                    ⍝ nur   ⍋avucAPX
⍋avuc_101        16                    ⍝ nicht ⍋avucIBM
⍋avuc_110        25                    ⍝ nicht ⍋avucWIN
⍋avuc_111       187                    ⍝ allen gemeinsam
      anz
20 39 5 24 16 25 187
      +/anz
316
```

Die Indizes 1, 1, 0 bedeuten, dass Δavuc_110 genau diejenigen Codepunkte enthält, welche in ΔavucAPX und ΔavucIBM, nicht aber in ΔavucWIN vorkommen, was sich unten bestätigt.

```
      ∧/⍋avuc_110∊avuc←((⍋avucAPX∊⍋avucIBM)/⍋avucAPX)~⍋avucWIN
1
      ∧/avuc∊⍋avuc_110
1
```

Wichtig für die möglichst uneingeschränkte Portierbarkeit ist Δavuc_111. Die darin enthaltenen 187 Codepunkte, bzw. die zugehörigen Zeichen können in allen drei Produkten verwendet werden. Allerdings ist dies nur eine notwendige, aber keine hinreichende Bedingung. Es ist zu berücksichtigen, dass sich bekanntlich die Produkte APLX, IBM-APL2 und APL+Win auch in Bezug auf viele weitere Eigenschaften unterscheiden, wie zumBeispiel

- Kontrollstrukturen,
- Sytemfunktionen oder
- Hilfsprozessoren (Auxiliary Processors)

Die folgende Tabelle zeigt die in Δavuc_111 enthaltenen Codepunkte mit Ausnahme von 0, 8, 10, 13 und 27 (9 ist nicht in ΔavucAPX enthalten), welche keine Glyphen darstellen. Sie umfasst 187 – 5 = 182 Codepunkte. Die 5 Spalten bedeuten:

(1) Codepunkt, (2) Zeichen, (3) AV-Index in APX, (4) AV-Index in IBM, (5) AV-Index in WIN. Die AV-Indizes sind auf ⎕io=0 bezogen.

32		32	32	32	78	N	110	78	78	124	|	77	124	254	8711	∇	71	183	236
33	!	30	33	33	79	O	111	79	79	125	}	127	125	125	8714	∈	69	238	238
34	"	160	34	34	80	P	112	80	80	126	~	84	126	126	8728	∘	74	248	248
35	#	161	35	35	81	Q	113	81	81	161	¡	219	173	173	8744	∨	40	235	250
36	$	126	36	36	82	R	114	82	82	163	£	165	156	156	8745	∩	67	239	239
37	%	162	37	37	83	S	115	83	83	168	¨	33	254	5	8746	∪	86	172	252
38	&	163	38	38	84	T	116	84	84	175	¯	64	253	253	8800	≠	42	244	134
39	'	75	39	39	85	U	117	85	85	191	¿	218	168	168	8801	≡	167	207	240
40	(	58	40	40	86	V	118	86	86	196	Ä	176	142	142	8804	≤	36	243	243
41	)	34	41	41	87	W	119	87	87	197	Å	177	143	197	8805	≥	94	242	242
42	⋆	80	42	42	88	X	120	88	88	199	Ç	178	128	128	8834	⊂	90	226	11
43	+	45	43	43	89	Y	121	89	89	209	Ñ	180	165	165	8835	⊃	88	227	14
44	,	44	44	44	90	Z	122	90	90	214	Ö	181	153	153	8854	⊖	20	233	233
45	−	95	45	45	91	[	59	91	91	215	×	61	245	146	8866	⊢	92	214	29
46	.	46	46	46	92	\	63	92	92	220	Ü	182	154	154	8867	⊣	124	215	28
47	/	47	47	47	93	]	39	93	93	223	ß	220	225	225	8868	⊤	78	152	231
48	0	48	48	48	94	^	41	94	94	224	à	184	133	133	8869	⊥	66	157	230
49	1	49	49	49	95	_	70	95	95	225	á	183	160	160	8900	⋄	96	216	4
50	2	50	50	50	96	`	166	96	96	226	â	185	131	131	8968	⌈	83	169	141
51	3	51	51	51	97	a	225	97	97	228	ä	186	132	132	8970	⌊	68	190	143
52	4	52	52	52	98	b	226	98	98	229	å	188	134	16	9015	⌶	175	211	169
53	5	53	53	53	99	c	227	99	99	231	ç	189	135	135	9017	⍐	31	146	152
54	6	54	54	54	100	d	228	100	100	232	è	191	138	138	9021	⍒	18	232	232
55	7	55	55	55	101	e	229	101	101	233	é	190	130	130	9023	⍓	27	240	235
56	8	56	56	56	102	f	230	102	102	234	ê	192	136	136	9024	⍔	26	241	167
57	9	57	57	57	103	g	231	103	103	235	ë	193	137	137	9033	⍜	19	237	237
58	:	62	58	58	104	h	232	104	104	236	ì	195	141	18	9035	⍝	17	251	30
59	;	60	59	59	105	i	233	105	105	237	í	194	161	161	9038	⍞	24	175	245
60	<	35	60	60	106	j	234	106	106	238	î	196	140	140	9042	⍢	16	252	31
61	=	37	61	61	107	k	235	107	107	239	ï	197	139	139	9045	⍣	25	174	244
62	>	38	62	62	108	l	236	108	108	241	ñ	198	164	164	9049	⍦	251	247	241
63	?	81	63	63	109	m	237	109	109	242	ò	200	149	21	9053	⍩	28	228	166
64	@	164	64	64	110	n	238	110	110	243	ó	199	162	162	9054	⍪	29	145	151
65	A	97	65	65	111	o	239	111	111	244	ô	201	147	147	9055	⍫	21	181	15
66	B	98	66	66	112	p	240	112	112	246	ö	202	148	148	9067	⍬	23	250	19
67	C	99	67	67	113	q	241	113	113	247	÷	43	246	246	9073	⍱	14	231	229
68	D	100	68	68	114	r	242	114	114	248	ø	217	155	171	9074	⍲	15	229	234
69	E	101	69	69	115	s	243	115	115	249	ù	205	151	20	9075	⍳	73	236	226
70	F	102	70	70	116	t	244	116	116	250	ú	204	163	163	9076	⍴	82	230	251
71	G	103	71	71	117	u	245	117	117	251	û	206	150	150	9077	⍵	87	249	23
72	H	104	72	72	118	v	246	118	118	252	ü	207	129	129	9079	⍷	169	209	3
73	I	105	73	73	119	w	247	119	119	8592	←	91	189	6	9082	⍺	65	224	224
74	J	106	74	74	120	x	248	120	120	8593	↑	89	198	24	9109	⍽	76	144	149
75	K	107	75	75	121	y	249	121	121	8594	→	93	184	26	9675	○	79	234	249
76	L	108	76	76	122	z	250	122	122	8595	↓	85	199	25					
77	M	109	77	77	123	{	72	123	123	8710	∆	123	182	145					

Die Codepunkte in den übrigen 6 Gebieten der Partition sind unten ersichtlich:

```
------------------------------------------------------------------------ ∆avuc_110
      ⎕←⍴cp←∆avuc_110 ⋄ cp                  ⍝ nicht in APL+Win
25
1 2 3 4 5 6 28 30 31 127 9014 9019 9026 9080 9472 9474 9484 9488 9492 9496 9500
      9508 9516 9524 9532
      ⎕←⍴cp←(∆avuc_110>126)/∆avuc_110 ⋄ cp                    ⍝ darstellbar
16
127 9014 9019 9026 9080 9472 9474 9484 9488 9492 9496 9500 9508 9516 9524 9532
⌂   ⌶    ⍰    ⍱    ⍳    ─    │    ┌    ┐    └    ┘    ├    ┤    ┬    ┴    ┼
------------------------------------------------------------------------ ∆avuc_101
      ⎕←⍴cp←∆avuc_101 ⋄ cp                  ⍝ nicht in IBM-APL  alle darstellbar
16
192 195 198 200 201 205 213 216 227 230 255 8364 8802 9060 9066 9068
À   Ã   Æ   È   É   Í   Õ   Ø   ã   æ   ÿ   €    ≢    ⍨    ⍮    ⍬
------------------------------------------------------------------------ ∆avuc_100
      ⎕←⍴cp←∆avuc_100 ⋄ cp                  ⍝ nur APLX
24
128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 245 338 339 9031
      9032 9040 9047 9061
      ⎕←⍴cp←(∆avuc_100>143)/∆avuc_100 ⋄ cp                    ⍝ darstellbar
8
245 338 339 9031 9032 9040 9047 9061
õ   Œ   œ   ⍈    ⍉    ⍐    ⍗    ö
------------------------------------------------------------------------ ∆avuc_011
      ⎕←⍴cp←∆avuc_011 ⋄ cp                  ⍝ nicht in APLX
5
7 9 12 166 204
      ⎕←⍴cp←∆avuc_011~7 9 12 ⋄ cp                             ⍝ darstellbar
2
166 204
¦   Ì
------------------------------------------------------------------------ ∆avuc_010
      ⎕←⍴cp←∆avuc_010 ⋄ cp                     ⍝ nur IBM-APL2
39
11 14 15 16 17 18 19 20 21 22 23 24 25 26 29 160 170 172 186 189 8359 8757 9552
      9553 9556 9559 9562 9565 9568 9571 9574 9577 9580 9600 9604 9608 9617 9618
      9619
      ⎕←⍴cp←(∆avuc_010>160)/∆avuc_010 ⋄ cp                    ⍝ darstellbar
23
170 172 186 189 8359 8757 9552 9553 9556 9559 9562 9565 9568 9571 9574 9577 9580
ª   ¬   º   ½   ₧    ∵    ═    ═    ╖    ╗    ╜    ╝    ╞    ╡    ╥    ╨    ╬
      9600 9604 9608 9617 9618 9619
      ▀    ▄    █    ░    ▒    ▓
------------------------------------------------------------------------ ∆avuc_001
      ⎕←⍴cp←∆avuc_001 ⋄ cp                  ⍝ nur APL+Win     alle darstellbar
20
162 165 171 187 193 194 202 203 206 207 210 211 212 217 218 219 221 253 337 9064
¢   ¥   «   »   Á   Â   Ê   Ë   Î   Ï   Ò   Ó   Ô   Ù   Ú   Û   Ý   ý   ő   ⍨
```

2.2 Zeichen in Codepunkte, Codepunkte in Zeichen

Es folgen zwei Funktionen, welche Zeichen in Codepunkte und diese wiederum zurück in Zeichen verwandeln können.

```
[0] na←nv UCausCHARS la
[1] ⍝------------------------------------------------------------------------
[2] ⍝ nv: ∆avucWIN / ∆avucIBM / ∆avucX    (Default: ∆avucWIN)
[3] ⍝ la: literal array
[4] ⍝ na: numerical array (Unicode-Codepunkte von la)
[5] ⍝  l: ∆avucWIN
[6] ⍝------------------------------------------------------------------------
[7] →(2=⎕nc'nv')/L_0 ⋄ nv←∆avucWIN
[8] L_0: na←nv[⎕av⍳la]
```

```
[0] la←nv CHARSausUC na
[1] ⍝------------------------------------------------------------------------
[2] ⍝ nv: ∆avucWIN / ∆avucIBM / ∆avucX  (Default: ∆avucWIN)
[3] ⍝ na: numerical array (Codepunkte)
[4] ⍝ la: literal array  (APL-Zeichen aufgrund von nv; Default ∆avucWIN)
[5] ⍝  l: ∆avucWIN
[6] ⍝------------------------------------------------------------------------
[7]  la←'' ⋄ →(0=⍴na)/0
[8] →(2=⎕nc'nv')/L_0 ⋄ nv←∆avucWIN
[9] L_0: la←⎕av[nv⍳na]
```

Zunächst könnte es vielleicht scheinen, dass diese Programme überflüssig sind. Ihr Nutzen liegt aber vor allem darin, dass sie zusammen mit `each` angewendet werden können.

2.3 Bemerkungen zu `∆avutf8`

In [4] wird für APL+Win der Vektor `∆avutf8` präsentiert, welcher dieselbe Funktion wie `∆avucWIN` hat. Die Benützung führt jedoch zu ernsthaften Problemen. Es geht mir dabei nicht um Kritik, sondern um Schwierigkeiten, welche bei der Koppelung von `⎕av` mit Unicode entstehen können.

Zunächst ist die Bezeichnung irreführend, da UTF-8 bei der Koppelung von APL-Zeichen mit Codepunkten nicht im Spiel ist. Probleme können dadurch entstehen, dass die Zuordnung `⎕av`1→Unicode zwar eindeutig, aber nicht umkehrbar eindeutig ist. Im Folgenden wird das Programm `timestamp` über eine Textkette in einen nummerischen Vektor von Codepunkten und anschließend zurück in die ursprüngliche `⎕cr`-Matrix verwandelt, worauf diese mit `⎕fx` wieder aktiviert wird. Der Prozess scheint gelungen:

```
      timestamp
13.09.2014/18:36

      ⍴lv0←,⎕av[⎕io+13],⎕cr'timestamp'
1275
      ⍴uc←∆avutf8[⎕av⍳lv0]
1275
      lv←⎕av[∆avutf8⍳uc]
      ⎕fx ⊃splitSS lv
timestamp
```

Bei der Ausführung tritt jedoch ein Fehler auf:

```
      timestamp
SYNTAX ERROR
timestamp[13]  TIME← 2 0 ⍕100|⎕TS[4 5]  ⋄
                                       ^
      (lv≠lv0)/lv ⋄ (lv≠lv0)/lv0
|
|
```

Am Bildschirm ist der Fehler nicht ersichtlich. Erst die zusätzliche Abfrage macht klar, dass ⎕av[⎕io+254] in ⎕av[⎕io+179] verwandelt worden ist:

```
      (-⎕io)+⎕av⍳(lv≠⍳lv0)/lv0          ⍝ in der ursprünglichen Funktion: 254
254
      (-⎕io)+⎕av⍳(lv≠⍳lv0)/lv           ⍝ in der transformierten Funktion: 179
179
      (-⎕io)+⎕av⍳'|'                    ⍝ Eingabe ab Tastatur
254
      ∆avutf8[⎕av⍳'|']                  ⍝ Codepunkt: 124 ...
124
```

Die beiden intern verschiedenen Zeichen sehen am Bildschirm gleich aus. Die Eingabe am Bildschirm produziert ⎕av[⎕io+254]. ∆avutf8 bildet beide auf den Codepoint 124 ab. Insgesamt werden 8 Zeichen auf 124 abgebildet:

```
      (-⎕io)+(∆avutf8=124)/⍳⍴∆avutf8    ⍝ ... kommt 8mal vor in ∆avutf8
179 180 181 182 185 186 222 254

      ⎕←uc124←⎕av[⎕io+179 180 181 182 185 186 222 254]
||||||||
```

Am Bildschirm sehen alle gleich aus, aber intern sind sie alle voneinander verschieden:

```
      uc124∘.=uc124                     ⍝ Zeichen sehen alle gleich aus,
1 0 0 0 0 0 0 0                         ⍝   sind aber intern alle
0 1 0 0 0 0 0 0                         ⍝   voneinander verschieden!
0 0 1 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 1 0 0 0
0 0 0 0 0 1 0 0
0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 1
```

Der Bildschirm ist ein Gleichmacher, doch execute stolpert über die interne Verschiedenheit:

```
      i←1 ⋄ '5',(i⊃uc124),'42'
5|42
      5|42
2
      i←1 ⋄ ⍎'5',(i⊃uc124),'42'
SYNTAX ERROR
      ⍎ 5|42
        ^
```

Durch Ausprobieren zeigt sich, dass sich execute nur mit ⎕av[⎕io+254] zufrieden gibt:

```
      i←8 ⋄ '5',(i⊃uc124),'42'
5|42
5|42
2
      i←8 ⋄ ⍎'5',(i⊃uc124),'42'   ⍝ Nur 8⊃uc kann zusammen mit ⍎ verwendet werden
2
```

Da das verwendete iota bei der Rückverwandlung den *ersten* der 8 Codepunkte 124 berücksichtigt, kann das Problem gelöst werden, indem man an den vorgängigen Stellen (⎕io+179 180 181 182 185 186 222) ∆avutf8 auf 0 setzt. Wiederholungen in ∆avutf8 sorgen auf dem Bildschirm für dieselben Glyphen. Das Additionszeichen erscheint 9 mal, ⎕ 4 mal. Auch das Minus- und das Fragezeichen wiederholen sich:

```
      ⎕av[av⍳uc],[1]uc,[.5]+/⍉av∘.=uc←nub (1<+/av∘.=av)/av←∆avutf8
    ⍎   "   +   -   ?   Ç   Ä   É   ⎕   Ö   Ü   Ñ   |
0 9067 34 43 45 63 199 196 201 9109 214 220 209 124
2    2  2  9  2  2   2   2   2    4   2   2   2   8
```

Es zeigt sich aber, dass hier nichts zu korrigieren ist. Denn diese Zeichen sind, sofern APL-wirksam, je an der ersten Stelle mit dem betreffenden Codepunkt untergebracht:

```
      (-⎕io)+⎕av⍳'+-?⎕'                   ⍝ +-?⎕: über die Tastatur eingegeben
43 45 63 149                              ⍝ "richtige" ⎕av-Indizes von +, -, ?, ⎕

      ∆avutf8[⎕av⍳'+-?⎕']                       ⍝ entsprechende Codepunkte
43 45 63 9109
      uc,[.5]+/(uc←43 45 63 9109)∘.=∆avutf8    ⍝ Vielfachheiten
 43 45 63 9109
  9  2  2    4

      (-⎕io)+∆avutf8⍳ 43 45 63 9109      ⍝ ⍳ erwischt die "richtigen" Stellen
43 45 63 149
```

Wiederum erweist sich der Bildschirm als Gleichmacher, der nur auf die Glyphen reagiert. Wenn wir eine Zeile kopieren und mit der Glyphe □ vergleichen, spielt der interne Unterschied keine Rolle:

```
      ind←(¯1+⍳256)~0 7 8 9 10 12 13 27 255
      ⎕av[⎕io+ind]
⍀€⍷⋄¨←⊂⊃⍟åæì⍒ùò⍬⍵↑↓→⊣⊢⍋⍒ !"#$%&'()★+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVW
      XYZ[\]∧_`abcdefghijklmnopqrstuvwxyz{¦}~≢Çüéâäà≠çêëèïî⌈Ä⌊É∆×ôö⎕û⍞⌹ÖÜ¢£¥⍪⍨áí
      óúñÑ⍝⍀¿⌷õøý¡«»□□□||||++||+++++ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏ-ÑÒÓÔÕÖ+ØÙÚÛÜÝ|ÿαßι⍤ã⍱⊥⊤⌽⊖⍲≁
      ∇⍉∊∩≡⍙≥≤⍕⍎÷"∘○∨⍴∪¯|

      +/'óúñÑ⍝⍀¿⌷õøý¡«»□□□||||++||+++++ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏ-'='⎕'
3
```

Es folgt eine Liste sämtlicher nötiger Korrekturen:

```
 w[⎕io+157]←165                            ⍝ ¥
 w[⎕io+179 180 181 182 185 186 222]←0      ⍝ '|' ist ⎕av[⎕io+254]
 w[⎕io+239]←8745                           ⍝ ∩
 w[⎕io+252]←8746                           ⍝ ∪
 w[⎕io+1 2 16 17 18 20 21 127]←0 8364 229 230 236 249 242 8802
⍝                                  €    å   æ   ì   ù   ò   ≢
```

Als nächstes testen wir den korrigierten Vektor, ∆avucWIN, an

Als nächstes testen wir den korrigierten Vektor, ∆avucWIN, an einem zufallsgenerierten Textvektor:

```
      ⎕←lv0←⎕av[⎕io+ii0←¯1+17?256]
z?b0»⍱ZBJÊP:mì+\>
      ⎕←uc←∆avucWIN[⎕av⍳lv0]
122 63 98 48 187 9073 90 66 74 202 80 58 109 236 43 92 62
      lv0≡lv←⎕av[∆avucWIN⍳uc]
0
      lv0,[.5]lv
z?b0»⍱ZBJÊP:mì+\>
z?b0»⍱ZBJÊP:mì+\>
      ii0,[.5](-⎕io)+⎕av⍳lv
 122 63 98 48 175 229 90 66 74 202 80 58 109 18 190 92 62
 122 63 98 48 175 229 90 66 74 202 80 58 109 18  43 92 62
```

Wir haben die uns schon bekannte Situation, dass lv0 und lv auf dem Bildschirm gleich aussehen, aber intern verschieden sind. Es folgen noch die Details:

```
      ⎕av[⎕io+190 43]
++
      =/⎕av[⎕io+190 43]
0
      ∆avucWIN[⎕io+190 43]
43 43
      (-⎕io)+(∆avucWIN=43)/⍳⍴∆avucWIN
43 183 184 187 188 189 190 191 215
      ⎕av[⎕io+43 183 184 187 188 189 190 191 215]
+++++++++
```

Durch unsere Korekturen haben wir aber erreicht, dass ΔavucWIN diejenigen Zeichen (und nicht nur die Glyphen) richtig abbildet, welche APL-wirksam sind. Bei ΔavucAPX und ΔavucIBM entstehen keine vergleichbaren Probleme. Im ersteren tritt nur der Codepunkt 0 mehrmals auf, im letzteren gibt es überhaupt keine Wiederholungen:

```
      acuv←ΔavucAPX~0
      ∧/1=+/acuv∘.=acuv
1
      +/ΔavucAPX=0
5
      ∧/1=+/ΔavucIBM∘.=ΔavucIBM
1
```

Wir fügen noch die Funktion timestamp sowie die Unterfunktion subL an:

```
[0] lv16←timestamp;d;t
[1] ⍝------------------------------------------------------------------------
[2] ⍝ lv16: z.B. 17.09.2014/06:57
[3] ⍝------------------------------------------------------------------------
[4]  d← 2 0 2 0 4 0 ⍕⎕ts[3 2 1] ⋄ d←d subL ' 0'  ⍝ d: date
[5]  d← 1 1 0 1 1 0 1 1 1 1\d ⋄ d[3 6]←'.'
[6]  t← 2 0 ⍕100|⎕ts[4 5] ⋄ t←t subL ' 0'        ⍝ t: time
[7]  t← 1 1 0 1 1 \t ⋄ t[3]←':'
[8]  lv16←d,'/',t
```

```
[0]  la←la subL lv2;dim;ind
[1] ⍝------------------------------------------------------------------------
[2] ⍝ ersetzt in la das Zeichen lv2[1] durch lv2[2]
[3] ⍝  la: literal array (simple)
[4] ⍝ lv2: literal vector of length 2
[5] ⍝------------------------------------------------------------------------
[6]  dim←⍴la ⋄ la←,la ⋄ ind←(la ⎕SS lv2[1])/⍳⍴la ⋄ la[ind]←lv2[2] ⋄ la←dim⍴la
```

3 UTF-8

3.1 Die UTF-8-Kodierung

UTF-8 ist eine Kodierungsmethode für natürliche Zahlen (inklusive 0), welche je nach Zahlgröße ein bis vier Bytes erfordert. Im folgenden Kodierungsschema bezeichnet x die für die Binärdarstellung zur Verfügung stehenden Stellen; deren Anzahl stimmt mit dem Exponenten der Zweierpotenz überein. Jeder Codepunkt muss mit möglichst wenigen Bytes dargestellt werden. Ohne diese einschränkende Vorschrift könnte man zum Beispiel 128 sowohl mit 2 Bytes als 11000010 10000000 (korrekt), als auch mit 3 Bytes als 11100000 10000010 10000000 (unzulässig) darstellen. (Auch 11110000 10000000 10000010 10000000 wäre möglich.) Das erste Byte einer Zeichendarstellung wird *Startbyte* genannt, die übrigen heißen *Folgebytes*. Folgebytes sind dadurch gekennzeichnet, dass sie mit 10 beginnen; Startbytes beginnen entweder mit 0 oder mit mehr als einer Eins.

Mit diesen 4 Bytes können Zahlen von 0 bis $2^{21} - 1 = 2\,097\,151$ (1FFFFF HEX) kodiert werden, also rund doppelt so viel als die 1 114 112, die (inklusive Sonderfunktionen) für Unicode vorgesehen sind.

Bytes	Schema	Maximum (DEZ)	Maxim. (HEX)
1	0xxxxxxx	$2^7 - 1 = 127$	7F
2	110xxxxx 10xxxxxx	$2^{11} - 1 = 2\,047$	7FF
3	1110xxxx 10xxxxxx 10xxxxxx	$2^{16} - 1 = 65\,535$	FFFF
4	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx	$2^{21} - 1 = 2\,097\,151$	1FFFFF

3.2 UTF-8 aus Codepunkten

Bei gegebenem Codepunkt `cp` wird zunächst die Anzahl `n` der benötigten Bytes bestimmt. Dann wird für jedes Byte die Anzahl der führenden Einsen sowie der für die Binärdarstellung zur Verfügung stehenden Stellen berechnet:

```
      cp←8900                          ⍝ cp: Codepunkt
      ⎕←n←1++/128 2048 65536≤cp        ⍝  n: #Bytes
3
      ⎕←e←1+n↑(n-1)                    ⍝  e: #führende Einsen
3 1 1
      ⎕←x←8-1+e                        ⍝  x: #Binärstellen
4 6 6
```

Als Nächstes werden die Köpfe der Bytes sowie die schon verteilten Binärstellen erhalten:

```
      ]display 8↑¨(e+1)↑¨(e/⍳n)⊂(+/e)⍴1
┌→──────────────────────────────────────────────────────┐
│┌→──────────────┐┌→──────────────┐┌→──────────────┐│
│└1 1 1 0 0 0 0 0┘└1 0 0 0 0 0 0 0┘└1 0 0 0 0 0 0 0┘│
└∊──────────────────────────────────────────────────────┘
      ]display ¯8↑¨(x/⍳n)⊂((+/x)⍴2)⊤cp
┌→──────────────────────────────────────────────────────┐
│┌→──────────────┐┌→──────────────┐┌→──────────────┐│
│└0 0 0 0 0 0 1 0┘└0 0 0 0 1 0 1 1┘└0 0 0 0 0 1 0 0┘│
└∊──────────────────────────────────────────────────────┘
```

Die beiden Teile werden vereinigt:

```
      ]display (8↑¨(e/⍳n)⊂(+/e)⍴1)∨¯8↑¨(x/⍳n)⊂((+/x)⍴2)⊤cp
┌→──────────────────────────────────────────────────────┐
│┌→──────────────┐┌→──────────────┐┌→──────────────┐│
│└1 1 1 0 0 0 1 0┘└1 0 0 0 1 0 1 1┘└1 0 0 0 0 1 0 0┘│
└∊──────────────────────────────────────────────────────┘
```

Die APL-Funktion `UTF8ausUC 4` führt diese Schritte aus und verwandelt am Ende noch den Vektor von booleschen Vektoren mit ∊ in einen einzigen booleschen Vektor.

```
  [0] bv←UTF8ausUC_4 ns;cp;n;e;x;w
  [1] ⍝------------------------------------------------------------------------
  [2] ⍝ ns: cp Codepunkt
  [3] ⍝ bv: UTF-8-Kodierung von cp
  [4] ⍝------------------------------------------------------------------------
  [5]  n←1++/128 2048 65536≤cp←ns                ⍝ cp: Codepunkt  n: #Bytes
  [6] →(n>1)/L_0 ⋄ bv←(8⍴2)⊤cp ⋄ →0              ⍝ mehr als 1 Byte?
  [7] L_0:
  [8]  e←1+n↑n-1                                 ⍝ anfängliche Einsen
  [9]  x←8-1+e                                   ⍝ bedeutsame Stellen
 [10] bv←∊(8↑¨(e/⍳n)⊂(+/e)⍴1)∨¯8↑¨(x/⍳n)⊂((+/x)⍴2)⊤cp
```

Die Zahlen von 0 bis 127 brauchen 1 Byte:

```
      cp←0 ⋄ b←(÷8)×n←⍴bv←UTF8ausUC_4 cp ⋄ zeigV_BV (8/⍳b)⊂bv
00000000
      cp←127 ⋄ b←(÷8)×n←⍴bv←UTF8ausUC_4 cp ⋄ zeigV_BV (8/⍳b)⊂bv
01111111
```

Die Zahlen von 128 bis 2047 brauchen 2 Byte:

```
      cp←128 ⋄ b←(÷8)×n←⍴bv←UTF8ausUC_4 cp ⋄ zeigV_BV (8/⍳b)⊂bv
11000010 10000000
      cp←2047 ⋄ b←(÷8)×n←⍴bv←UTF8ausUC_4 cp ⋄ zeigV_BV (8/⍳b)⊂bv
11011111 10111111
```

Die Zahlen von 2048 bis 65535 brauchen 3 Byte:

```
      cp←2048 ⋄ b←(÷8)×n←⍴bv←UTF8ausUC_4 cp ⋄ zeigV_BV (8/⍳b)⊂bv
11100000 10100000 10000000
      cp←65535 ⋄ b←(÷8)×n←⍴bv←UTF8ausUC_4 cp ⋄ zeigV_BV (8/⍳b)⊂bv
11101111 10111111 10111111
```

Die Zahlen von 65536 bis 2 097 151 brauchen 4 Byte:

```
      cp←65536 ⋄ b←(÷8)×n←⍴bv←UTF8ausUC_4 cp ⋄ zeigV_BV (8/⍳b)⊂bv
11110000 10010000 10000000 10000000
      cp←2097149 ⋄ b←(÷8)×n←⍴bv←UTF8ausUC_4 cp ⋄ zeigV_BV (8/⍳b)⊂bv
11110111 10111111 10111111 10111101
      cp←2097150 ⋄ b←(÷8)×n←⍴bv←UTF8ausUC_4 cp ⋄ zeigV_BV (8/⍳b)⊂bv
11110111 10111111 10111111 10111110
      cp←2097151 ⋄ b←(÷8)×n←⍴bv←UTF8ausUC_4 cp ⋄ zeigV_BV (8/⍳b)⊂bv
11110111 10111111 10111111 10111111
```

3.3 Codepunkte aus UTF-8

Als Beispiel nehmen wir die UTF-8-Binärfolge

```
        ¯                                       ¯
      bv
0 1 1 0 0 0 0 1 1 1 1 0 0 0 1 0 1 0 0 0 0 1 1 0 1 0 0 1 0 0 0 0 0 0 1 1 0 0 0 1
      0 0 1 1 0 1 0 1 0 0 1 0 0 0 0 0 1 1 1 0 0 0 1 0 1 0 0 0 1 0 1 1 1 0 0 0 0
      1 0 0 0 0 1 0 0 0 0 0 0 1 1 0 0 0 1 0 1 1 1 0 0 0 1 0 1 0 0 0 0 1 1 0 1 0
      0 1 0 0 0 0 0 0 1 1 0 0 1 0 1 1 1 0 0 0 1 0 1 0 0 1 0 1 1 1 1 0 0 0 1 0 1
      1 1 1 1 0 0 0 1 0 1 0 0 1 0 1 1 1 1 0 0 0 1 0 1 1 0 0 1 0 1 1 1 0 0 0 1 1
      0 0 1 0 0 0 1 1 0 1 0 1
```

Zunächst wird die Anzahl der führenden Einsen pro Byte bestimmt. Es handelt sich dann (und nur dann) um ein Startbyte (`s`), wenn diese Anzahl von 1 verschieden ist.

```
      bm←(⌽8,(⍴bv)÷8)⍴bv                  ⍝ bm: jede Zeile von bm ein Byte
      ⎕←e←+/∧\bm                          ⍝  e: je #anfängliche Einsen
0 3 1 1 0 0 0 3 1 1 0 0 3 1 1 0 3 1 1 3 1 1 0 0 0

      ⎕←s←e≠1                             ⍝ Ist das Byte ein Startbyte?
1 1 0 0 1 1 1 1 0 0 1 1 1 0 0 1 1 0 0 1 0 0 1 1 1
```

Nun wird die Zuordnung der Byte und der Bit zu den Zeichen ermittelt. Aufgrund dieser Zuordnung wird `bv` in einen Vektor `v_bv` von Binärfolgen verwandelt, von denen jede ein Zeichen bestimmt.

```
      +\s                                 ⍝ Zuteilung Byte -> Zeichen
1 2 2 2 3 4 5 6 6 6 7 8 9 9 9 10 11 11 11 12 12 12 13 14 15

      nv_part←8/+\s                       ⍝ nv_part: Partitionsvektor für Bits
      zeigV_BV v_bv←nv_part⊂bv
01100001 11100010100001101001 0000 00110001 00110101 00100000 1110001010001011100
      00100 00100000 01100010 11100010100001101001 0000 00110010 1110001010010111
      10001011 11100010100101111000 1011 00101110 00110010 00110101
```

Zu jeder dieser Binärfolgen gehört ein Codepunkt, und zu jedem Codepunkt ein Zeichen:

```
      UCausUTF8¨nv_part⊂bv
97 8592 49 53 32 8900 32 98 8592 50 9675 9675 46 50 53

      CHARSausUC UCausUTF8¨nv_part⊂bv
a←15 ⋄ b←2○○.25
```

Die Funktion `UCausUTF8` führt die oben erläuterten Schritte aus:

```
  [0] ns←UCausUTF8 bv;UTF8;n;x;w
  [1] ⍝-------------------------------------------------------------------------
  [2] ⍝ bv: UTF-8-Darstellung  e i n e s  Zeichens (1 - 4 Bytes)
  [3] ⍝ ns: unicode            e i n e s  Zeichens
  [4] ⍝-------------------------------------------------------------------------
  [5]  ns←0 ⋄ UTF8←bv
  [6]  n←+/∧\UTF8                         ⍝ n: #Bytes
  [7] →(n>0)/L_0 ⋄ ns←2⊥8↑UTF8 ⋄ →0       ⍝ 1-Byte Darstellungen tanzen
  [8] L_0:                                ⍝  aus der Reihe
  [9]  UTF8←(8/⍳n)⊂UTF8                   ⍝ Aufteilung in einzelne Byte
 [10]  x←8-1+1+n↑n-1                      ⍝ x: bedeutsame Stellen
 [11] ns←2⊥∊(-x)↑¨UTF8
```

3.4 Funktionen nach UTF-8 und zurück

Wir zeigen an einem Beispiel, wie die sämtlichen Funktionen eines WS (es sind 950) in eine einzige UTF-8-Binärfolge `bv` kodiert und aus dieser wieder zurückgewonnen werden können.

```
      )wsid
IS C:\APLWIN13\PRGS\SUDOKU\SUDOKU

      )copy ..\utf8\utf8 sec UCausCHARS ∆avucWIN
SAVED Freitag, 19. September 2014 16:10:34
```

Wir brauchen zwei Trennzeichen, `sep` und `SEP`. Das erste leitet jeweils die Zeilen einer Funktion, das zweite die Funktion selbst ein. Eigentlich handelt es sich nicht um Zeichen, nicht einmal um Codepunkte im eigentlichen Sinne. Es sind einfach zwei Zahlen, welche mit den Codepunkten der zu verschlüsselnden Funktionen nicht in Konflikt kommen.

```
      (sep SEP)←9998 9999                ⍝ APL-Codepunkte ≤ 9675 (○)
      ⍴fns0←VausLM FNS0←⎕nl 3
950
      t0←sec ⋄ ⍴nv0←∊SEP,¨,¨sep,¨(⊂⍋avucWIN)UCausCHARS¨v_lm0←⎕cr¨fns0 ⋄ sec-t0
2908584
0
```

Wie man sieht, beginnt man mit einem Vektor, dessen Komponenten die CR-Matrizen der 950 Funktionen sind. Dieser wird direkt in einen Vektor von nummerischen Matrizen, bestehend aus den Codepunkten der CR-Darstellungen, verwandelt. Dann werden die Zeilenanfänge mit `sep` versehen, die Matrizen in Vektoren verwandelt und die Funktionsanfänge mit `SEP` gekennzeichnet. Mit ∊ wird die Vektorschachtelung aufgehoben. Man hat nun die Funktionen in den nummerischen Vektor `nv0` gesteckt und kann sie löschen.

```
      ⎕erase ⎕nl 3
      ⍴⎕nl 3
0 0
      )copy ..\utf8\utf8 sec UTF8ausUC_4 splitUTF8 UCausUTF8 CHARSausUC splitSS
SAVED Freitag, 19. September 2014 16:10:34

      t0←sec ⋄ ⍴bv←1=∊UTF8ausUC_4¨nv0 ⋄ sec-t0
24623552
16
```

Nachdem man die zur weiteren Umwandlung benötigten Funktionen wieder einkopiert hat, wird zunächst der Codepunkt-Vektor `nv0` in die UTF-8-Binärfolge `bv` kodiert.
Nun beginnt die Rückgewinnung der Funktionen. Die UTF-8-Folge muss als nächstes in Binärvektoren aufgeteilt werden, welche den einzelnen Zeichen entsprechen.

```
      t0←sec ⋄ ⍴v_bv←splitUTF8 bv ⋄ sec-t0
2908584
1.7999999999992724
      t0←sec ⋄ nv←UCausUTF8¨v_bv ⋄ sec-t0
14.099999999998546
      nv≡nv0
1
```

Die einzelnen Binärvektoren werden in je einen Codepunkt verwandelt. Der Codepunkt-Vektor `nv` stimmt mit dem Ausgangsvektor `nv0` überein. Nun wird mit `splitSS` zweimal nacheinander aufgeteilt: zunächst in einzelne Funktionen (wobei `SEP` als Trennzeichen wirkt), dann jede Funktion in einzelne Zeilen (mit `sep` als Trennzeichen. Mit `disclose` erhält man einen Vektor von Matrizen, und schließlich `v_lm`, welcher Vektor von literalen Matrizen mit dem ursprünglichen `v_lm0` übereinstimmt.

```
      t0←sec ⋄ ⍴v_lm←(⊂⍋avucWIN)CHARSausUC¨⊃¨splitSS¨splitSS nv ⋄ sec-t0
950
0.39999999999963624
      v_lm≡v_lm0
1
      ⍴fns←⎕fx¨v_lm
950
      FNS0≡⎕nl 3
1
```

Mit `⎕fx` erhält man schließlich die ursprünglichen 950 Funktionen zurück. Die Rückverwandlung innerhalb desselben APL-Produkts dient hier als Test der Korrektheit. Will

man aber die 950 Funktionen zum Beispiel in IBM-APL2 laufenlassen, so braucht man bei der Dekodierung von `bv` nur `ΔavucWIN` durch `ΔavucIBM` zu ersetzen.
Es folgen die oben verwendeten Funktionen `VausLM`, `sec`, `splitUTF8`, `splitSS` sowie die Unterfunktion `osDEB`, welche Anfangs- und Endleerstellen beseitigt und Zwischenstücke von Leerstellen auf eine einzige Leerstelle reduziert. Hier wird sie allerdings nur verwendet, um Endleerstellen zu beseitigen.

```
  [0] v_lv←VausLM lm
  [1] ⍝------------------------------------------------------------------------
  [2] ⍝ verwandelt lm (literale Matrix) in Vektor von literalen Vektoren
  [3] ⍝ In diesen werden Anfangs- und Endleerstellen eliminiert und leere
  [4] ⍝ Zwischenstücke auf je eine Leerstelle reduziert
  [5] ⍝------------------------------------------------------------------------
  [6]  v_lv←0⍴⊂'' ⋄ →(0=↑⍴lm)/0
  [7] v_lv←osDEB¨⊂[2]lm
```

```
  [0] ns←sec
  [1] ⍝------------------------------------------------------------------------
  [2] ⍝ ns: Zeit seit Anfang Sitzung, auf Zehntelsekunden gerundet
  [3] ⍝------------------------------------------------------------------------
  [4]  ns←.1×⌊.5+10×2⊃⎕ai
```

```
  [0] v_bv←splitUTF8 bv;bm;a;a2;nv_part;w
  [1] ⍝------------------------------------------------------------------------
  [2] ⍝   bv: UTF-8-Binärfolge
  [3] ⍝ v_bv: Jede Komponente des Vektors ist wiederum eine UTF-8-Binärfolge
  [4] ⍝        und entspricht einem Zeichen
  [5] ⍝ zerlegt bv (UTF-8) in Abschnitte, die den einzelnen Zeichen entsprechen
  [6] ⍝------------------------------------------------------------------------
  [7]  bm←(⍋8,(⍴bv)÷8)⍴bv
  [8]  a←+/∧\bm                ⍝ a: Anzahl anfänglicher Einsen pro Byte
  [9]  nv_part←1++\a≠1         ⍝ a≠1: Beginn des nächsten Zeichens
 [10] v_bv←(8/nv_part)⊂bv
```

```
  [0] v_v←splitSS v;sep;bv;nv_part
  [1] ⍝------------------------------------------------------------------------
  [2] ⍝ v: nv oder lv (numerical / literal vector)
  [3] ⍝    1⊃v ist das Trnnzeichen (sep)
  [4] ⍝ partitioniert v, v_v ist Vektor von Vektoren
  [5] ⍝------------------------------------------------------------------------
  [6]  sep←1⊃v ⋄ v←1↓v ⋄ bv←v=sep             ⍝ sep: Trennzeichen
  [7]  nv_part←(~bv)×1++\bv                   ⍝ nv_part: Partitionsvektor
  [8] v_v←nv_part⊂v                           ⍝ Partition gemäss nv_part
```

```
  [0] lm←osDEB lm;bv1;bv2;MAT;lm0;lm1;w
  [1] ⍝------------------------------------------------------------------------
  [2] ⍝ Ersatz für ASM-Funktion DEB durch klassisches APL
  [3] ⍝ Eliminiert Anfangs- und Endleerspalten und reduziert
  [4] ⍝  im Innern mehrfache Leerspalten auf einfache
  [5] ⍝------------------------------------------------------------------------
  [6]  MAT←2=⍴⍴lm
  [7]  lm←(¯2↑1 1,⍴lm)⍴lm ⋄ lm←' ',lm,' ' ⋄ lm←0 1↓0 ¯1↓(~1 1⍷∧/[1]' '=lm)/lm
  [8] →MAT/0 ⋄ lm←,lm
```

Literatur

[1] *Unicode*, siehe http://de.wikipedia.org/wiki/Unicode

[2] *Glyphe*, siehe http://de.wikipedia.org/wiki/Glyphe

[3] *APLX Character set*, siehe http://www.microapl.com/apl help/ch 210.htm

[4] *APL to Unicode*, siehe http://aplwiki.com/AplToUnicode

Kontakt:
Dr. sc. math. (ETH) Urs Oswald
Internet: http://www.ursoswald.ch/
eMail: osurs@bluewin.ch

Frank Schmidt-Kelletat

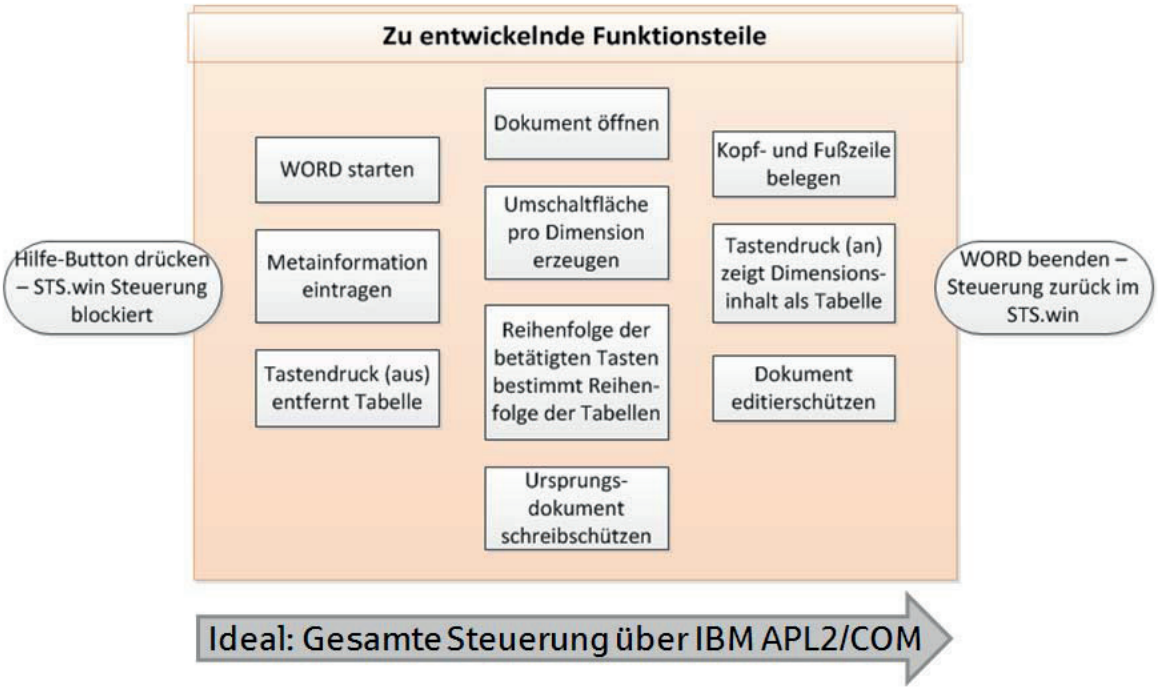
IBM-APL2 und die COM-Schnittstelle von WORD 2010 unter WINDOWS 7

Zum Thema IBM-APL2 und die COM-Schnittstelle von WORD 2010 unter WINDOWS 7 gibt es bislang keine dokumentierte Hilfe. Der folgende Beitrag soll ein kleines Fundament schaffen, das Interessierten eine strukturierte Starthilfe an die Hand gibt, die es leichter macht, sich in dieses Thema einzuarbeiten.

Status Quo

Ausgangspunkt meiner Konfrontation mit diesem Thema war die Problemstellung, wie es aus Sicht eines potentiellen Endanwenders einer Applikation gelingen kann, eindeutige Datenbank-IDs mit ihren Bezeichnungen komfortabel zur Ansicht zu bringen, um sie zu speichern oder zu drucken.

Im Rahmen des Informationssystems des Rheinischen Sparkassen- und Giroverbandes wird dieser Dienst zum Zeitpunkt der Datenselektion für granulare Abfragen unter der Beschreibung „Datenbankdokumentation anzeigen" für jede Datenbank angeboten. Abhängig von der gewünschten Auswahl wird auf Anforderung ein



WORD-Dokument geladen, das aus einem einleitenden, beschreibenden und einem tabellarisch aufgebauten Teil besteht, der die Schlüsselinformationen enthält. Diese Dokumente, generiert durch die Fachabteilung, liegen statisch im System vor.

Aus Anwendersicht ist die Forderung selbstverständlich, dass der Tabellenteil immer aktuell ist. Auch aus Sicht der Fachabteilung ist dies selbstverständlich, aber leider nicht immer realisierbar, da Anpassungen manuell nachgepflegt werden müssen. Dies erfordert aber einen permanent reibungslos funktionierenden Informationsaustausch unter verschiedenen Beteiligten, der aus unterschiedlichen Gründen nicht immer gewährleistet ist.

Ziel

Die optimale Lösung muss also sein, die aktuell statischen WORD-Dokumente soweit es geht zu dynamisieren. Nach einer Absprache mit der Fachabteilung kann man sehr schnell zu dem Ergebnis kommen, dass die Vorlagendokumente nur noch den einleitenden, beschreibenden Teil enthalten sollen, gefolgt von zwei Absatzzeichen, die den Übergang zum dynamischen Teil markieren. Schlüsselinformationen sowie Kopf- und Fußzeileninformationen sollen nur noch dynamisch erzeugt werden, womit wir aus Entwicklungssicht bei der COM-Technologie wären.

Wunschvorstellung ist, dass sich zwischen Aufruf und Schließen der Dokumentation verschiedene Elemente, Eigenschaften und Ereignisdefinitionen vom APL2 über die COM-Schnittstelle WORD übergeben lassen, die den User in den Stand versetzen, in einem editiergeschützten Dokument über Umschaltflächen zielgerichtet die gewünschten Datenbankinformationen anzuzeigen, das Dokument zu drucken oder zu speichern, ohne dass die Gefahr besteht, dass das Ursprungsdokument überschrieben oder andere geöffnete WORD-Instanzen beeinträchtigt werden.

Hilfen

Die MSDN Developer Page (WORD-Objektmodell), wobei insbesondere der Enumerationsteil von großer Bedeutung ist, der APL2 User's Guide (Supplied External Routines: `COM` und `COMBROWSE`) sowie der VBA-Makroaufzeichnungsmodus.

Im Detail

Das Starten von WORD hat keine negativen Auswirkungen auf bestehende WORD-Sitzungen – es wird immer eine neue, eigenständige Instanz (durch nachfolgenden Befehl unsichtbar) geöffnet.

- Befehl zum Kreieren einer WORD-Instanz:

```
WORD←COM 'CREATE' 'Word.Application'
```

Die gesamte Generierung des Dokumentes soll so ruhig wie möglich für das Auge des Anwenders ablaufen. Dazu kann sich der Entwickler zwei Techniken zunutze machen: Die Bildschirmaktualisierung deaktivieren und die Generierung im minimierten Zustand durchführen – Hilfe bietet dabei die Enumeration wdWindowState:

- Befehl für Bildschirmaktualisierung aus:

```
COM 'PROPERTY' WORD 'APPLICATION.SCREENUPDATING' 0
```

- Befehl für Ausführung im minimierten Zustand:

```
COM 'PROPERTY' WORD 'APPLICATION.WINDOWSTATE' 2
```

Nun kann man sich dem zu ladenden Vorlagendokument widmen. Da man das Ursprungsdokument vor dem Überschreiben absichern muss, ist es bereits beim Öffnen notwendig, die entsprechende Readonly-Eigenschaft mitzugeben. Da WORD unsichtbar ist, kann man das Dokument unbesorgt zusätzlich sichtbar machen:

- Befehl zum sichtbaren und schreibgeschützten Öffnen des Vorlagendokumentes ΔR:

```
DOK←COM 'METHOD' WORD 'Documents.Open[]'('Filename' ΔR)('ReadOnly' 1)('Visible' 1)
```

Man beachte, dass nun mit dem Handle DOK ein Repräsentant des WORD-Dokumentes zur Verfügung steht.
WORD bietet verschiedene Ansichtsmodi an. Wir entscheiden uns für die Variante Seitenlayout mit ausgeblendeten Formatierungssymbolen, weil es für unseren beabsichtigten Dienst nach unserer Meinung die passende Darstellungsform ist:

- Befehl für Ansicht Seitenlayout:

```
COM 'PROPERTY' WORD 'ActiveWindow.View.Type' 3
```

- Befehl für Ausblenden der Formatierungssymbole:

```
COM 'PROPERTY' WORD 'ActiveWindow.ActivePane.View.ShowAll' 0
```

Da diese Einstellungen mit Beenden von WORD beibehalten werden, wir aber nicht möchten, dass der Anwender seine gewohnte Umgebung dauerhaft verändert vorfindet, merken wir uns den Ursprungszustand, um ihn beim Beenden von WORD wieder herzustellen. Zum Auslesen verwendet man die gleiche Syntax, jedoch unter Verzicht der rechten Parameter.
Arbeiten wir uns nun von oben nach unten durch das zu erzeugende Dokument und passen zunächst die Kopfzeile an, da wir dort ein Logo mit Hilfe einer Tabelle rechts platzieren möchten. WORD unterscheidet

hier die aktuelle Kopfzeile, die gerade Kopfzeile, die erste Kopfzeile oder die primäre Kopfzeile. Wir entscheiden uns für die primäre Kopfzeile und ermitteln die zu verwendende Konstante über die Enumeration wdSeekView:

- Befehl zum Anwählen der primären Kopfzeile:
  ```
  COM 'PROPERTY' WORD 'ACTIVEWINDOW.ACTIVEPANE.VIEW.SEEKVIEW' 1
  ```
- Befehl zum Einfügen einer einzeiligen Tabelle mit drei Spalten ist zweiteilig, da man zunächst einen Handle als Repräsentanten der Tabelle benötigt:
  ```
  TABLE←COM 'PROPERTY' WORD 'SELECTION.RANGE'
  COM 'METHOD' WORD 'SELECTION.TABLES.ADD' TABLE 1 3
  ```
- Befehl zum Ermitteln des Handles der rechten Zelle:
  ```
  ZELLE←COM 'PROPERTY' WORD 'ACTIVEDOCUMENT.SECTIONS().HEADERS().RANGE.TABLES().
  CELL().RANGE' 1 1 1(1 3)
  ```
- Befehl zum Einfügen der Grafik in die gewünschte Zelle:
  ```
  PICTURE←COM 'METHOD' ZELLE 'INLINESHAPES.ADDPICTURE'('Logo.bmp')
  ```
- Befehl zum Rechtsausrichten der Grafik innerhalb der Zelle mit Hilfe der Enumeration wdParagraphAlignment:
  ```
  COM 'PROPERTY' ZELLE 'PARAGRAPHFORMAT.ALIGNMENT' 2
  ```
- Befehl zur Höhenskalierung der Grafik in Prozent:
  ```
  COM 'PROPERTY' PICTURE 'SCALEHEIGHT' 15
  ```
- Befehl zur Breitenskalierung der Grafik in Prozent:
  ```
  COM 'PROPERTY'PICTURE 'SCALEWIDTH' 20
  ```

Nach Beenden der Kopfzeilendefinition wenden wir uns dem Body des Dokumentes zu. Um den Cursor an das Ende des Dokumentes zu steuern, verwendet man die folgende Methode in Kombination mit der entsprechenden wdUnits-Konstanten:

- Befehl zur Platzierung des Cursors an das Ende des Textes:
  ```
  COM 'METHOD' WORD 'SELECTION.ENDOF' 6
  ```
- Befehl zum Einfügen einer neuen Zeile durch Einfügen eines Absatzzeichens:
  ```
  COM 'PROPERTY' WORD 'SELECTION.TYPEPARAGRAPH'
  ```
- Befehl zum Einfügen einer Textzeile mit aktuellem Datum:
  ```
  COM 'PROPERTY' WORD 'SELECTION.TEXT' 'Datenbankstruktur mit Stand vom TT.MM.JJJJ'
  ```
- Befehle zum Unterstreichen und Kursivsetzen dieser Zeile:
  ```
  COM 'PROPERTY' WORD 'SELECTION.FONT.UNDERLINE' 1
  COM 'PROPERTY' WORD 'SELECTION.FONT.ITALIC' 1
  ```
- Befehlsfolge zum Aussetzen dieser Formatierungen:
  ```
  COM 'METHOD' WORD 'SELECTION.ENDOF' 6
  COM 'PROPERTY' WORD 'SELECTION.FONT.UNDERLINE' 0
  COM 'PROPERTY' WORD 'SELECTION.FONT.ITALIC' 0
  ```

Nun werden die Umschaltflächen (Toggle Buttons) erzeugt, die dafür genutzt werden, auf Tastendruck die Informationen der gewünschten Datenbankdimension tabellarisch darzustellen:

- Befehl zur Erzeugung eines Toggle-Buttons:
```
TOGGLE←COM 'METHOD' WORD 'SELECTION.INLINESHAPES.ADDOLECONTROL' 'Forms.ToggleButton.1'
```
- Befehle zur Formatierung des Toggle-Buttons (kein Zeilenumbruch, Buttonbezeichnung setzen, automatische Buttonbreite und Schriftart definieren):
```
COM 'PROPERTY' TOGGLE 'OLEFORMAT.OBJECT.WORDWRAP' 0
COM 'PROPERTY' TOGGLE 'OLEFORMAT.OBJECT.CAPTION' 'Bezeichnung'
COM 'PROPERTY' TOGGLE 'OLEFORMAT.OBJECT.AUTOSIZE' 1
COM 'PROPERTY' TOGGLE 'OLEFORMAT.OBJECT.FONT.NAME' 'Tahoma'
```

Die Toggle-Buttons sind nun kreiert, aber noch ohne Funktion – dazu bedarf es der Definition eines Events, in diesem Fall eines CLICK-Events, das durch einen Tastendruck ausgelöst wird. APL benötigt dieses Signal, um die Anweisung zu erhalten, die entsprechende Tabelle mit den gewünschten Datenbankinformationen ein- oder auszublenden. Aufgrund eines aktuell noch ungeklärten, schwerwiegenden Problems bei der Definition dieses Events, existiert bei Drucklegung dazu nur ein Workaround, auf dessen Darstellung an dieser Stelle verzichtet wird. Um die Zeit für das Formatieren der Tabellen für die anzuzeigenden Datenbankdimensionen zu minimieren, werden sie einmalig aufgebaut und dann jeweils in einem unsichtbaren Dokument abgelegt. Beim gewünschten Einblenden einer Tabelle wird lediglich das entsprechende Dokument angesprochen, der gesamte Inhalt kopiert und in das Vorlagendokument eingefügt. Dabei muss aber immer der vorhandene Editierschutz berücksichtigt werden, der das Dokument vor Manipulationen bewahren soll:

- Befehl zum Erzeugung eines unsichtbaren Dokumentes:
```
DOKUN←COM 'METHOD' WORD 'Documents.ADD[]'('Visible' 0)
```
- Befehl zum Einfügen der Datenbankinformationen ERG (Format: Vektor mit Trennung der Attribute durch Semikolon und Verwenden von AV[14] um Zeilen umzubrechen):
```
COM 'PROPERTY' WORD 'SELECTION.TEXT' ERG
```
- Befehl zum Umwandeln des Textes in eine Tabelle mit der wdTableFieldSeparator-Konstanten 2, die das Semikolon als Spaltentrennzeichen berücksichtigt:
```
COM 'METHOD' WORD 'SELECTION.CONVERTTOTABLE' 2
```
- Befehlsfolge zur Absatzformatierung (wdLineSpacing):
```
COM 'PROPERTY' WORD 'SELECTION.PARAGRAPHFORMAT.SPACEBEFORE' 0
COM 'PROPERTY' WORD 'SELECTION.PARAGRAPHFORMAT.SPACEAFTER' 0
COM 'PROPERTY' WORD 'SELECTION.PARAGRAPHFORMAT.LINESPACINGRULE'3
COM 'PROPERTY' WORD 'SELECTION.PARAGRAPHFORMAT.LINESPACING' 14
```
- Befehl zum Verwenden von einer automatischen Formatierung der Tabelle (wdTableFormat):
```
COM 'METHOD' WORD 'ACTIVEDOCUMENT.TABLES().AUTOFORMAT' 1 16
```

- Befehl zum Setzen der optimale Spaltenbreite (wdAutoFitBehaviour):
```
COM 'METHOD' WORD 'ACTIVEDOCUMENT.TABLES().AUTOFITBEHAVIOUR' 1 1
```
- Befehl zum Setzen einer Textmarke, die später zum Wiederfinden der Tabelle dient:
```
COM 'METHOD' WORD 'ACTIVEDOCUMENT.BOOKMARKS.ADD'('_Dimension_X')
```
- Befehl zum Ermitteln des Handles der Tabelle:
```
TABLE←COM 'PROPERTY' WORD 'ACTIVEDOCUMENT.TABLES().RANGE' 1
```
- Befehlsfolge zur weiteren Formatierung der Tabelle (Zeilenhöhe, Text vertikal zentrieren[wdCellVerticalAlignment], Schriftart und Schriftgröße setzen):
```
COM 'PROPERTY' TABLE 'ROWS.HEIGHT' 16
COM 'PROPERTY' TABLE 'CELLS.VERTICALALIGNMENT' 1
COM 'PROPERTY' TABLE 'FONT.NAME' 'Sparkasse Rg'
COM 'PROPERTY' TABLE 'FONT.SIZE' 11
```
- Befehl zum Ermitteln des Handles der ersten Tabellenzeile:
```
ROW←COM 'PROPERTY' WORD 'ACTIVEDOCUMENT.TABLES().ROWS().RANGE' 1 1
```
- Befehlsfolge zur weiteren Formatierung der Tabellenzeile (verbundene Zellen, Rahmenlinien links, oben und rechts entfernen [wdBorderType]):
```
COM 'PROPERTY' ROW 'CELLS.MERGE' COM 'PROPERTY' ROW 'BORDERS().LINESTYLE' ¯2 0
COM 'PROPERTY' ROW 'BORDERS().LINESTYLE' ¯1 0
COM 'PROPERTY' ROW 'BORDERS().LINESTYLE' ¯4 0
```
- Befehl zum Ermitteln des Handles der zweiten Tabellenzeile:
```
ROW←COM 'PROPERTY' WORD 'ACTIVEDOCUMENT.TABLES().ROWS().RANGE' 1 2
```
- Befehl zum Setzen der Hintergrundfarbe Grau in der zweiten Zeile (wdColorIndex):
```
COM 'PROPERTY' ROW 'SHADING.BACKGROUNDPATTERNCOLORINDEX' 16
```
- Befehl zum Setzen einer fetten Schriftart in der zweiten Zeile:
```
COM 'PROPERTY' ROW 'FONT.BOLD' 1
```
- Befehl zum Setzen eines Blattschutzes (wdProtectionType):
```
COM 'METHOD' WORD 'ACTIVEDOCUMENT.PROTECT[]'('Password' 'test') ('Type' 3)
```

Das Einfügen und Entfernen der Tabellen auf Tastendruck steuert man auf entsprechendes Signal hin wie folgt aus der APL2-Umgebung:

- Befehlsfolge zum Einfügen einer Tabelle in das sichtbare WORD-Dokument (unsichtbares Quelldokument aktivieren – markierte Tabelle in die Zwischenablage kopieren – sichtbares Zieldokument aktivieren – Ende des Dokumentes aufsuchen [wdUnits] – Blattschutz aufheben – Absatz einfügen – Tabelle im Ursprungsformat einfügen [wdRecoveryType] – Ende des Dokumentes aufsuchen [wdUnits] – weiteren Absatz einfügen – zur letzten Tabelle gehen [wdGoToItem und wdGoToDirection] – den gesamten Text in der ersten Zeile der Tabelle markieren [wdUnits, Count, wdExtend]):
```
COM 'PROPERTY' WORD 'DOCUMENTS().ACTIVATE' Quelldokument
COM 'PROPERTY' WORD 'SELECTION.COPY'
COM 'PROPERTY' WORD 'DOCUMENTS().ACTIVATE' Zieldokument
COM 'METHOD' WORD 'SELECTION.ENDOF' 6
COM 'METHOD' WORD 'ACTIVEDOCUMENT.UNPROTECT'('test')
COM 'PROPERTY' WORD 'SELECTION.TYPEPARAGRAPH'
```

```
COM 'METHOD' WORD 'SELECTION.PASTEANDFORMAT' 16
COM 'METHOD' WORD 'SELECTION.ENDOF' 6
COM 'PROPERTY' WORD 'SELECTION.TYPEPARAGRAPH'
COM 'METHOD' WORD 'SELECTION.GOTO' 2 ¯1
COM 'METHOD' WORD 'SELECTION.MOVERIGHT' 3 1 1
```

- Befehlsfolge zum Entfernen einer Tabelle aus sichtbaren WORD-Dokument:

```
COM 'METHOD' WORD 'Selection.GoTo[]'('What' ¯1)('Name' '_Dimension_X')
COM 'METHOD' WORD 'ACTIVEDOCUMENT.UNPROTECT'('test')
COM 'PROPERTY' WORD 'Selection.Rows.Delete'
COM 'METHOD' WORD 'Selection.Delete[]'('Unit' 1)('Count' 2)
```

Der Einfügen- und Entfernenprozess wird abgerundet durch Löschen der durch das Drücken des Buttons angelegten Textmarke und dem Wiederherstellen des Blattschutzes:

- Befehl zum Entfernen der Textmarke:

```
COM 'PROPERTY' WORD 'ACTIVEDOCUMENT.BOOKMARKS().DELETE' (⊂TABINDEX⊃DIM1)
```

- Befehlsfolge zum Entfernen einer Tabelle aus sichtbaren WORD-Dokument:

```
COM 'METHOD' WORD 'ACTIVEDOCUMENT.PROTECT[]'('Password' 'test') ('Type' 3)
```

Für die Hinterlegung userspezifischer Informationen wie User-ID und Institut eignet sich die Belegung der Fußzeile:

- Befehl zum Anzeigen der Fußzeile im Druckvorschaumodus (wdSeekView):

```
COM 'PROPERTY' WORD 'ACTIVEWINDOW.ACTIVEPANE.VIEW.SEEKVIEW' 4
```

- Befehl zum Eintragen der User-ID (mit Zeilenumbruch):

```
COM 'METHOD' WORD 'SELECTION.INSERTAFTER'(USER-ID, 1↓ TC)
```

- Befehl zum Eintragen der Institutsbezeichnung darunter:

```
COM 'METHOD' WORD 'SELECTION.INSERTAFTER' INSTITUT
```

- Befehl zum Wechseln der Anzeige in Druckvorschaumodus des Dokumentes:

```
COM 'PROPERTY' WORD 'ACTIVEWINDOW.ACTIVEPANE.VIEW.SEEKVIEW' 0
```

Wenn der Anwender die strukturellen Datenbankinhalte gesehen hat, schließt er das WORD-Dokument. Um die systemimmanente Nachfrage „Soll das Dokument gespeichert werden?“, - die von jedem geöffneten Dokument gestellt wird - zu unterdrücken, muss man WORD entweder suggerieren, dass das Dokument bereits gespeichert wurde oder es tatsächlich automatisch speichern (leider lässt sich die erste Variante nicht anwenden, wenn Toogle Buttons im Dokument vorliegen). Dies erreicht man mit folgender Technik, die sich VBA-Codes bedient:

- Formulierung des VBA-Codes für die unsichtbaren Dokumente:

```
VB_CODE←⊂'Private Sub Document_Close()'
VB_CODE←VB_CODE,⊂('ActiveDocument.Saved = True')
VB_CODE←VB_CODE,⊂('End Sub')
```

- Formulierung des VBA-Codes für das sichtbare Dokument:
- VB_CODE←⊂'Private Sub Document_Close()'
- VB_CODE←VB_CODE,⊂('ActiveDocument.SaveAs FileName:='Datei mit Pfad')
- VB_CODE←VB_CODE,⊂('End Sub')
-
- Anweisungen zum Platzieren des VBA-Codes in den VBA-Projekten der Dokumente:
- VBCOMPONENTS←COM 'PROPERTY' DOK 'VBProject.VBComponents'
- ITEM←COM 'PROPERTY' VBCOMPONENTS 'ITEM()' 1
- CODEMODULE←COM 'PROPERTY' ITEM 'CODEMODULE'
- COM 'METHOD' CODEMODULE 'InsertLines' 2(VB_CODE,¨⊂1↓ TC)

Ich hoffe, mit meinen Ausführungen dem einen oder anderen Interessierten den Einstieg in das Thema zukünftig deutlich erleichtern zu können.

Kontakt:
Frank Schmidt-Kelletat

Rheinischer Sparkassen- und Giroverband, Düsseldorf

eMail: frank.schmidt-kelletat@rsgv.de

Rolf Erbe

Der Huffman-Code

1 Einleitung

Zur Klärung des Begriffs vorweg eine kurze Einleitung: In einem deutschen oder englischen Text kommt der Buchstabe e sehr viel häufiger vor als beispielsweise der Buchstabe q. Um den Text mit möglichst wenigen Bits zu codieren, liegt die Idee nahe, häufig vorkommende Zeichen durch möglichst kurze Codewörter zu codieren. Diese Idee ist auch beim Morse-Code verwirklicht, in dem das Zeichen e durch ein Codewort der Länge 1, nämlich einen ·, das Zeichen q dagegen durch ein Codewort der Länge 4, nämlich – – · – codiert wird.

Problematisch bei Codes mit unterschiedlichen Codewortlängen ist, dass im allgemeinen eine eindeutige Dekodierung von Zeichenfolgen nicht möglich ist: Im Morse-Code könnte beispielsweise die Folge · · – · · · · – sowohl zu usa als auch idea dekodiert werden.

Das Problem kommt dadurch zustande, dass gewisse Codewörter Anfangswörter von anderen Codewörtern sind. Dadurch kann bei der Dekodierung nicht entschieden werden, wo ein Codewort endet und wo das nächste anfängt. Die o.a. Folge könnte mit e (·) oder mit i (· ·) oder mit u (· · –) oder sogar mit f (· · – ·) anfangen.

Gilt dagegen die folgende Bedingung, so lassen sich codierte Zeichenfolgen direkt und eindeutig dekodieren. Im Morse-Code wird die Bedingung erfüllt, indem hinter jedem Codewort als Trennzeichen eine kurze Pause eingefügt wird.

Satz: (Fano-Bedingung)

Wenn kein Codewort Anfangswort (=Prefix) eines anderen Codewortes ist, dann ist jede codierte Zeichenreihe eindeutig dekodierbar.

Gegeben sei ein Alphabet A und ein Text $t \in A^+$. Ziel des Verfahrens von Huffman ist die systematische Konstruktion eines Codes $c(A) \subseteq B^+$, der die Fano-Bedingung erfüllt und der den Text mit möglichst wenigen Bits codiert. Anwendung findet die Huffman-Codierung nicht nur bei der Kompression von Texten, sondern u.a. in der Fax-Übertragung und im Bilddaten-Kompressionsverfahren JPEG.

Beispiel:

Der zu codierende Text lautet: ‘Im Westen nichts Neues‘. In Bild 1 ist die Situation nach Schritt 1 des Algorithmus dargestellt.

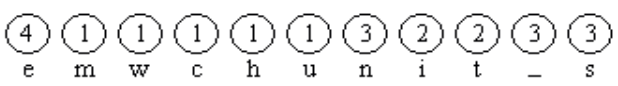


Bild 1: In Schritt 1 erzeugte Knoten

Die nachfolgenden Bilder 2 bis 5 zeigen weitere Stadien im Verlauf der Konstruktion des Baumes in Schritt 2. Die Kanten sind von oben nach unten gerichtet; die Kantenmarkierungen sind zunächst weggelassen.

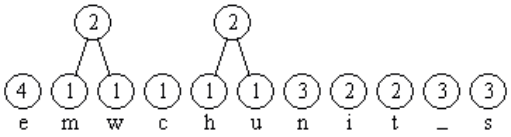


Bild 2: Die ersten beiden in Schritt 2 neu erzeugten Knoten

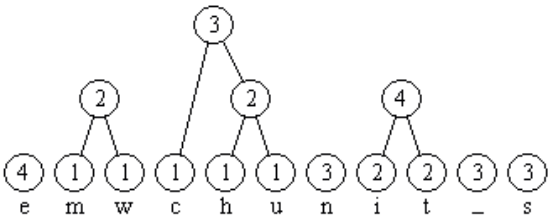


Bild 3: Weitere neu erzeugte Knoten

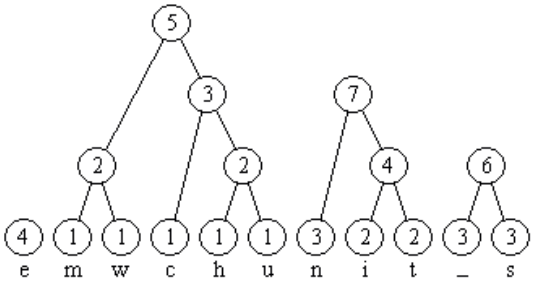


Bild 4: Weitere neu erzeugte Knoten

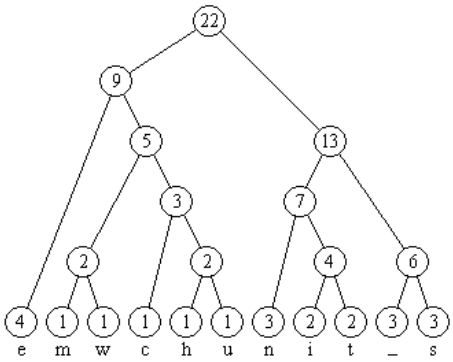


Bild 5:Der fertige Huffman-Baum

2 Implementierung des Huffman-Codes in APL

Als Ursprungstext (DEMOTXT) wirdhier die E-Mail-Einladung von Herrn Geißelhardt zur APL-Tagung am 12./13.05.2014 in Stuttgart verwendet (siehe Bild 6).

Die Liste (FREQS) der (73) verschiedenen in dieser Nachricht enthaltenen und zu codierenden Zeichen ergibt sich damit zu:

Lieb APKolg(n),usrTaück-
täh.WzfVSIdE-MmwRN@
DpvBO15CGq209:3UßQ78+46Z;FHöx/y

Bild 7 zeigt eine Liste der entsprechenden Häufigkeiten ihres Auftretens im Ursprungstext (hier nur ein Ausschnitt).Das heißt beispielsweise, das „o“ kommt 30 mal vor, der Punkt 31 mal etc. Aus dieser Liste wird nun mittels der Routine HUFFMAN ein sog. Huffman-Tree (DEMOtree) aufgebaut:

```
DEMOtree ← HUFFMAN FREQS
```

Dieser Huffman-Tree wird als Vektor der Länge 145 dargestellt. Bild 8 gibt dessen sechs erste Elemente wieder. Als Graphik stellt sich der Baum dann wie in Bild 9 dar (hier nur die obersten 5 Ebenen):

Zur Codierung eines Textes wird nun aus diesem Huffman-Tree die entsprechend Code-Tabelle (DEMOtab) erzeugt:

```
DEMOtab ⎕ HUFFtabGEN DEMOtree
```

Die ersten 5 Eintäge in dieser Code-Tabelle werden in Bild 10 präsentiert.

```
Liebe APL Kolleg(inn)en,
unsere Tagung rückt näher.
Wir haben noch Platz für Vorträge. Wenn Sie eine nette Idee für eine kurze Präsentation haben,
 schicken Sie doch bitte eine E-Mail an mich bzw. an Reiner Nussbaum (reiner@nussbaum-gift.de)
.
Der Vortrag muss nicht in einer perfekten Präsentationsvorlage verfügbar sein, es kann z.B. au
ch
Online direkt im APL vorgeführt werden. Der zeitliche Rahmen ist natürlich offen, wir freuen u
ns
 auch über einen 15-minütigen Vortrag.
Das Thema Matter of Concern (MoC) wird uns während dieser GSE-Tagung wieder beschäftigen. Wir
 werden bei der Requirement-Session über das weitere Vorgehen diskutieren müssen.
Die Tagung beginnt am 12.5. um 09:30 Uhr. Am Sonntagabend werden wie immer einen Vorabendtreff
 organisieren.
Weitere Informationen werde ich in Kürze versenden.
Beste Grüße

Bernd Geißelhardt

APL Benutzer Service
Allianz Deutschland AG
IT Personen-, Sachversicherung, Querschnitt (D-IT-PSQ3-K3)
Reinsburgstr.19
70178 Stuttgart

Tel:   +49 711 663 3968
Allianz Deutschland AG
Vorsitzender des Aufsichtsrats: Dr. Werner Zedelius.
Vorstand: Dr. Markus Rieß, Vorsitzender; Dr. Wolfgang Brezina, Dr. Markus Faulhaber, Bernd Hei
nemann,
 Burkhard Keese, Dr. Manfred Knof, Dr. Birgit König, Andree Moschner, Dr. Alexander Vollert.
Für Umsatzsteuerzwecke: USt-ID-Nr.: DE 814 580 981
Finanz- und Versicherungsleistungen i.S.d. UStG / MwStSystRL sind von der Umsatzsteuer befreit
.
Sitz der Gesellschaft: München
Registergericht: Amtsgericht München HRB 158078
```

Bild 6: Ursprungstext (DEMOTXT)

```
DISPLAY  ⍬¨⍉(⍬ ,[⍳10]FREQS)[59+⍳10;]
```

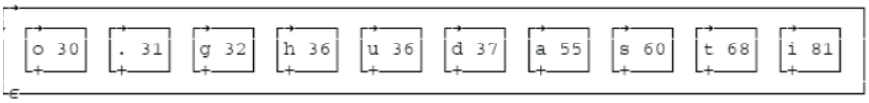
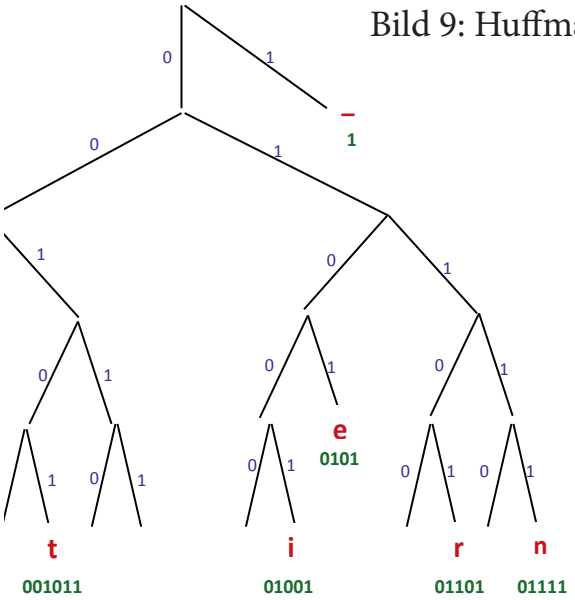


Bild 7: Zeichenhäufigkeit im Ursprungstext (DEMOTXT)

```
DISPLAY 6↑DEMOtree
```

Bild 8: Zeichenhäufigkeit im Ursprungstext (DEMOTXT)

Bild 9: Huffman-Tree




```
DISPLAY ⊂[2](⍉DEMOtab)[⍳5;]
```

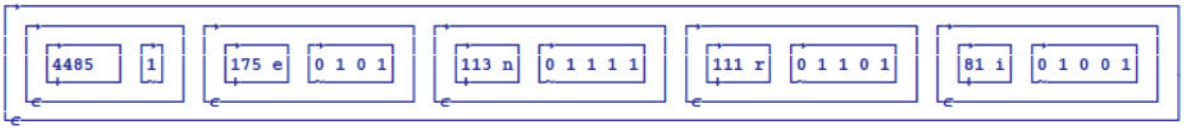


Bild 10: Code-Tabelle

```
DISPLAY 50↑DEMOCipher
0 1 0 0 0 1 1 1 1 0 1 0 0 1 0 1 0 1 0 0 1 1 1 0 0 0 1 0 1 1 0 1 1 0 0 0 1 1 0 0 1 0 0 0 0 0 0 1 0 0
```

Bild 11: Codierter Text (Anfang)

```
0100011110100101010011100010110110001100100000 ...
```

Bild 12: Decodierung des Textes

```
      DISPLAY ⎕CR 'HUFFencode'
Z←L HUFFencode R;N;S;SYMBOL;T;TEXT;i;n
⍝ Encode a plaintext L into Huffman-Ciphertext
⍝ L: T ; encoding table (output of HUFFtable)
⍝ R: text to be HUFFMAN-encoded (e.g.:plaintext)
⍝ Z: encoded (binary) ciphertext
T←L
TEXT←R
Z←⍳0
L1:
→(0=⍴TEXT)/END
Z←Z,((↑TEXT)=↑¨⍬¨T[1;])/T[2;]
TEXT←1↓TEXT
→L1
END:
Z←∊Z
      DISPLAY ⎕CR 'HUFFdecode'
Z←L HUFFdecode R;BIT;T;Tcurr;cipher;code;pl;pr
⍝ Decode HUFFMAN-encoded ciphertext
⍝ L: ciphertext to be decoded
⍝ R: coding-tree (e.g.: HUFFtree)
(cipher T)←L R
Z←⍳0
L0:
Tcurr←↑T
code←⍳0
L1:
→(0=⍴cipher)/0
BIT←↑cipher
cipher←1↓cipher
→(BIT=0 1)/BIT0 BIT1
BIT0:              ⍝ process bit 0:
code←code,BIT
⍎(2=⍴Tcurr)/'Z←Z,2⌷Tcurr ⋄ →L0' ⍝ if it's a leaf
pl←3⌷Tcurr     ⍝ proceede to left child
Tcurr←↑(pl=2⌷¨T)/T
⍎(2=⍴Tcurr)/'Z←Z,2⌷Tcurr ⋄ →L0' ⍝ if it's a leaf
→L1              ⍝ handle next bit
BIT1:              ⍝ process bit 1:
code←code,BIT
⍎(2=⍴Tcurr)/'Z←Z,2⌷Tcurr ⋄ →L0' ⍝ if it's a leaf
pr←4⌷Tcurr      ⍝ proceede to right child
Tcurr←↑(pr=2⌷¨T)/T
⍎(2=⍴Tcurr)/'Z←Z,2⌷Tcurr ⋄ →L0' ⍝ if it's a leaf
→L1              ⍝ handle next bit
```

Bild 13: APL2-Code der beiden Routinen HUFFencode und HUFFdecode

Das heißt, das Blank kommt im Text 4485 mal vor und hat den Huffman-Code 1, das e kommt 175 mal vor und hat den Huffman-Code 0101 , das n kommt 113 mal vor und hat den Code 01111 etc.

Mit dieser Code-Tabelle (DEMOtab) wird nun mit der Routine HUFFencode der Ursprungstext (DEMOTXT) codiert:

DEMOCipher ⎕

DEMOtab HUFFencode (,DEMOTXT)

Den Anfang des 5.775 Stellen langen codierten Textes zeigt Bild 11, die Dekodierung Bild 12.

Wird dieser gesamte mit der Routine HUFFencode verschlüsselte Text (DEMOCipher) nun mit der Routine HUFFdecode wieder zurückübersetzt, ergibt sich:

```
RUECKTXT←DEMOCipher
HUFFdecode DEMOtree
```

Der Test, ob die Huffman-Codierung tatsächlich verlustfrei war erfolgt durch

```
      (25 231⍴RUECKTXT) ≡ DEMOTXT
1
```

Was damit bestätigt wird.

Der Originaltext (DEMOTXT) hat eine Länge von 5775 Zeichen zu je 8 Bit = 46.200 Bit.

Der codierte Text (DEMOCipher) hat eine Länge von 12.264 Bit.

Der Kompressionsfaktor beträgt somit 46.200/12.264 = 3,77:1. *)

*) Ein solches Ergebnis kann natürlich nicht als repräsentativ gelten, da es aus der sehr hohen Anzahl der Blanks im Ursprungstext resultiert

Kontakt:

Rolf Erbe
Nordweststr. 89
63128 Dietzenbachne.de

E-Mail: rwerbe@t-online.de

APL-Journal
33. Jg. 2014, ISSN 1438-4531
Herausgeber: Dr. Reiner Nussbaum, APL-Germany e.V., Mannheim, http://www.apl-germany.de
Redaktion: Dipl.-Volksw. Martin Barghoorn (verantw.), TU Berlin, Franklinstr. 28, D-10587 Berlin, Tel. (030) 314 24392, Fax (030) 314 25901
Verlag: RHOMBOS-VERLAG, Berlin, Kurfürstenstr. 15/16, D-10785 Berlin, Tel. (030) 261 9461, eMail: verlag@rhombos.de, Internet: www.rhombos.de
Erscheinungsweise: halbjährlich
Erscheinungsort: Berlin
Satz: Rhombos-Verlag
Druck: dbusiness.de GmbH, Berlin

Allgemeine Informationen

(Stand 2011)

APL-Germany e.V. ist ein gemeinnütziger Verein mit Sitz in Düsseldorf. Sein Zweck ist es, die Programmiersprache APL, sowie die Verbreitung des Verständnisses der Mensch-Maschine Kommunikation zu fördern. Für Interessenten, die zum Gedankenaustausch den Kontakt zu anderen APL-Benutzern suchen, sowie für solche, die sich aktiv an der Weiterverbreitung der Sprache APL beteiligen wollen, bietet APL-Germany den adäquaten organisatorischen Rahmen.

Auf Antrag, über den der Vorstand entscheidet, kann jede natürliche oder juristische Person Mitglied werden. Organe des Vereins sind die mindestens einmal jährlich stattfindende Mitgliederversammlung sowie der jeweils auf zwei Jahre gewählte Vorstand.

1. Vorstandsvorsitzender
Dr. Reiner Nussbaum
Dr. Nussbaum gift mbH, Buchenerstr. 78, 69259 Mannheim,
Tel. (0621) 7152190.

2. Vorstandsvorsitzender:
Martin Barghoorn
Sekr. FR 6-7, Technische Universität Berlin, Franklinstr. 28,
10587 Berlin,
Tel. (030) 314 24392

Schatzmeister
Uwe Janza
Janza Software Consulting,
Mittlere Kanalstraße 4a
90429 Nürnberg
Tel. 0911-327075,
Fax -327076

Ordentliche Mitglieder:

Natürliche Personen	32,- EUR*
Studenten / Schüler	11,- EUR*

Außerordentliche Mitglieder:

Jurist./natürl. Pers.	500,- EUR*

* Jahresbeitrag

Bankverbindung
BVB Volksbank eG Bad Vilbel
BLZ 518 613 25
Konto-Nr. 523 2694

Hinweis:
Wir bitten alle Mitglieder, uns Adressänderungen und neue Bankverbindungen immer sofort mitzuteilen. Geben Sie bei Überweisungen den namen und/oder die Mitgliedsnummer an.

Einzugsermächtigung

Mitglieds-Nr.: _____

Ich erkläre mich hiermit widerruflich damit einverstanden, daß APL Germany e.V. den jeweils gültigen Jahres-Mitgliedsbeitrag von meinem unten angegebenen Konto abbucht. Mit der Datenübermittlung an das oben genannte Kreditinstitut bin ich einverstanden. Einen eventuell bestehenden Dauerauftrag habe ich bei meiner Bank gelöscht.

Bankbezeichnung: _____

BLZ: _____ Konto-Nr.: _____

Datum: _____ Unterschrift: _____