GNU APL

- Warum GNU APL?
- Design Ziele
- Datenformate
- System-Funktionen und -Variablen
- Plotten von Daten (□PLOT)
- Interfaces (Bindings)
- Sprünge und Conditionals
- APL is Fun

Warum GNU APL?

- Ca. **1982**: Der Autor lernt APL. Damals teuer und langsam.
- **1985-1990**: Der Autor konstruiert und baut einen parallelen APL Rechner (DATIS-P 256) an der Universität des Saarlandes.
  - Effektive Parallelisierung aller APL Primitives
  - Plan: Adaption eines existierenden Interpreters (APL 68000)
  - Der Rechner funktioniert, aber der Plan scheitert am deutschen Beamtenrecht.
- Ca. **2001**: Erster Kontakt mit IBM APL2 (PC Demo Version). Immer noch zu teuer.
- Ca. **2003**: Erster Kontakt mit Unicodes
  - Erste Gedanken über einen eigenen APL interpreter.
- **2008**: GNU APL 1.0

Design Ziele

- Freier Interpreter → damit APL nicht länger teuer ist.
- Platform unabhängig → GNU/Linux, BSD, Solaris, MacOS.
- IBM APL2 Kompatibel
  - 1:1 Replacement für IBM APL2.
  - Einfache Weiter-Benutzung existierender APL2 Workspaces
- ISO 13751 Kompatibel → kam später
- Support für DATIS-P 256 Parallelität. GNU APL sollte der noch fehlende APL Intepreter für DATIS-P werden (obwohl diese Plattform (Motorola 68020 CPU) mittlerweile überholt war).

Unterstützte Datenformate #1

- Standard APL2 Typen (Integer, Float, Character, Nested)
- Rational (Integer Zähler ÷ Integer Nenner) wenn konfiguriert
- Komplex und Hex
- Float als Klasse (für Multi-Präzisions-Arithmetik (experimentell)
- Strukturierte Variablen
- Assoziative Arrays (= Spezialfall strukturierter Variablen)
- **XML** (via □XML), **JSON** via □JSON), viele andere (via □CR
- Files, Sockets, Datum, etc. (via □FIO aka. libc Interface)
- Mehrzeilige String Literals (a-la Python)

Unterstützte Datenformate #2

Beispiele:

- Rationale Zahlen: **X←1 2 3÷4 5 6**
- Hex: **X←$ABBADEAD**
- Strukturierte Variablen: **X.abc←1 2 3 4 ◊ X.abc**

1 2 3 4

- Assoziative Arrays : **X['abc']**
- **1 2 3 4**
- Mehrzeilige Strings: **Address←"""**

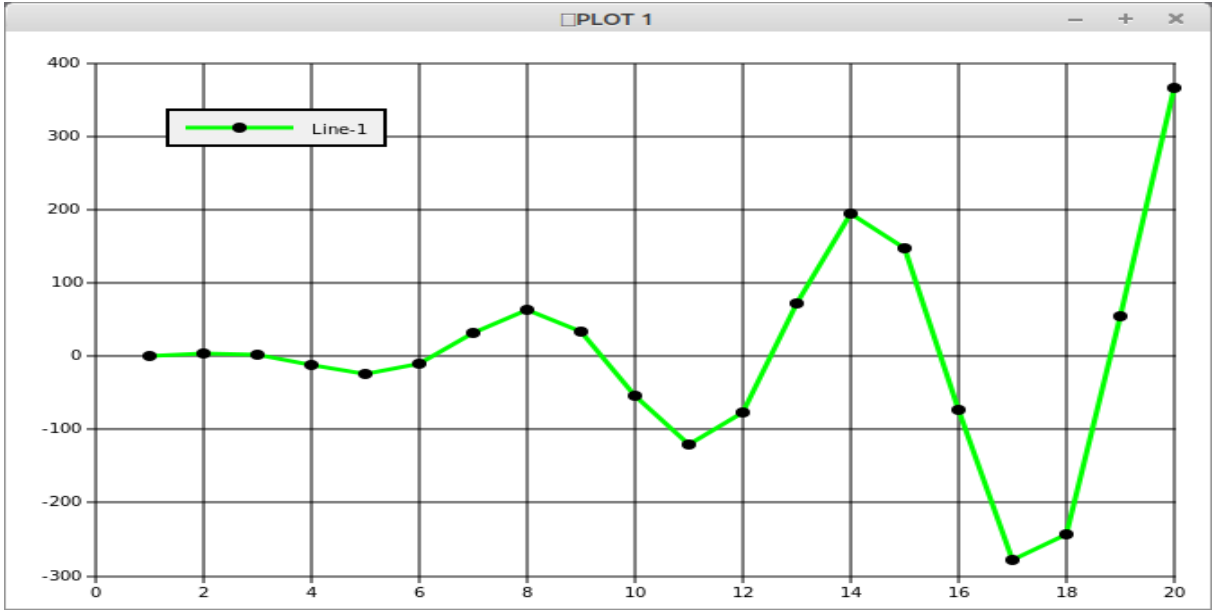
→ Jürgen Sauermann
→ 1543 Spruce St.
→ 94709 Berkeley, CA
"""

Systemfunktionen und -Variablen

- Alle IBM APL2 □-Funktionen und □-Variablen, plus:
- □DLX Exact cover aka. Knuth’s Dancing Links
- □FFT: Fast Fourier Transformation
- □GTK GTK GUI interface
- □PLOT Plotten von APL Werten
- □PNG Portable Network Graphics
- □RE Reguläre Ausdrücke
- □SQL Interface zu SQL Datenbanken (MySQL, PostgreSQL)

Plotten von Daten

```
□PLOT  J×J×1○J←⍳20
```



Interfaces (Bindings) #1

- GNU APL kann auf relativ einfache Weise mit anderen Programmiersprachen kombiniert werden. Dabei gibt es 2 Richtungen
  - Aufruf von APL Operationen (Ausdrücke, Defined Functions, Kommandos) die in einer anderen Programmiersprache als APL (meistens C oder C++) implementiert wurden, oder
  - Aufruf von APL Funktionen aus einer anderen Programmiersprache (meistens C oder C++)
- Damit können die Vorteile verschiedener Sprachen ausgenutzt werden.

Interfaces (Bindings) #2

Aufruf fremder Funktionen: so einfach wie 1-2-3.

1. Die Funktion wird in der gewünschten Sprache implementiert,
2. Die Implementierung wird in ein Shared Library gepackt, und
3. Das Shared Library wird mittels dyadic □FX in den Workspace geladen. z.B:

   my_library.so □FX 'MY_FUNCTION'

Interfaces (Bindings) #3

Aufruf von APL Funktionen aus anderen Sprachen:

1. GNU APL als library kompilieren (**./configure --with-libapl**).
2. Das fremde Programm mit **libapl.so** linken. Dadurch wird das API von **libapl** für das fremde Programm sichtbar (genauer: nach
   #include "apl/libapl.h".
3. Über dieses API werden elementare APL Operationen (Werte erzeugen, APL Funktionen aufrufen, Workspaces laden, etc. verfügbar.
4. **libapl** wurde von **Prof. Dirk Laurie** entwickelt und GNU APL zur Verfügung gestellt.

Interfaces (Bindings) #4

Existierende Bindings:

1. C/C++: libapl selbst.
2. Erlang: libapl mit dem Erlang native Interface.
3. Python: libapl mit **python_apl.cc**.

Diese Bindings sind als Beispiele (Prototypen) dafür gedacht, wie man seine eigenen Bindings herstellen kann.

Sprünge und Conditionals #1

Wir alle kennen:

LOOP: … ◊ →(MAX > N←N+1)/LOOP

Laufzeit Kosten: 5 APL Primitives **→ + ← >** und ***I*** (oder auch **⍴** statt ***I***) und 3 Variablen **LOOP**, **MAX** und **N**.

GNU APL unterstützt daneben auch relative Sprünge :

0→ MAX > N←N+1 ⍝ **0** bedeutet: 0 + ↑□LC (gleiche Zeile)

Laufzeit Kosten: 4 APL Primitives **→ + ← >** und 2 Variablen **MAX** und **N**. Und: Besser optimierbar (nur 1 Token für 0→).

Sprünge und Conditionals #2

Vergleiche (Mastering Dyalog APL, p. 186 ff.):

```
:IF Condition
   STATEMENTS
:ENDIF
```

mit (GNU APL):

```
Condition→ →STATEMENTS← ←
```

Sprünge und Conditionals #3

```
:IF Condition
   STATEMENTS_IF
:ELSE
  STATEMENTS_ELSE
:ENDIF
```

vs. (GNU APL):

```
Condition→ →STATEMENTS_IF
           ← →STATEMENTS_ELSE
           ← ←
```

APL is Fun #1

Das Exact Cover Problem:

Sei **M** eine Boole'sche Matrix. Finde eine (oder auch alle) Teilmenge(n) der Zeilen von **M**, so dass jede Spalte von **M** genau eine 1 enthält.

Warum interessiert uns das?

Weil manche Probleme mit einem Exact Cover sehr elegant gelöst werden können. Im Folgenden einige Beispiele…

APL is Fun #2

Problem: Plaziere 8 Türme so auf einem Schachbrett, dass sie sich nicht gegenseitig schlagen können.

Lösung:
1. Formuliere das Problem als Constraint Matrix **M**,
2. Finde eine (oder auch alle) Lösungen mittels **⌹DLT**.

APL is Fun #3

Wir starten mit einem leeren Schachbrett und definieren:

Ein (gültiger oder auch ungültiger) Zug besteht darin, einen einzelnen Turm auf ein (unbesetztes) Feld zu stellen.

Das Schachbrett hat **8×8=64** Felder, also gibt es (anfangs) 64 mögliche Züge.

APL is Fun #4

Die Constraint Matrix **M** für dieses Problem hat **64** Zeilen (für die **64** möglichen Züge) und **8+8** Spalten (für die **8** Zeilen und die **8** Spalten des Schachbretts).

Mit jedem (validen) Zug wird diese **64×16** Matrix immer kleiner, bis nach **8** Zügen nur noch Lösungen vorliegen. Das liegt daran, dass jeder Zug andere Züge verhindert (blockiert).

APL is Fun #5

Anders ausgedrückt: jedes Exact Cover der Matrix **M** ist eine Lösung des 8-Türme Problems.

Wenn man nun die Matrix **M** konstruiert hat (siehe später) dann berechnet:

Z← 0 ⎕DLX M ⍝ 0 ⎕DLX: return the first solution
Z←¯1 ⎕DLX M ⍝ ¯1 ⎕DLX: return all solutions

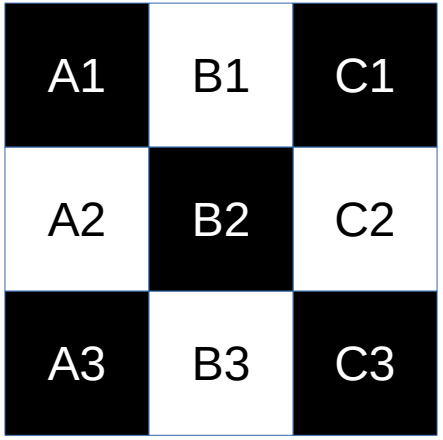
die Lösung(en) für ein (oder alle) exact cover(s) von **M**.

APL is Fun #6

Nun ist die 64×16 Matrix des 8-Türme Problems viel zu gross für diese Slide Show. Wir erklären daher alles mit 3 Türmen auf einem 3×3 Schachbrett. Hier gib es also (nur) 9 Züge und die Constraint Matrix **M** hat **9** Zeilen und **3+3=6** Spalten:

APL is Fun #7

3×3 Schachbrett:



Constraint Matrix M:

	Col.			Row		
	A	B	C	1	2	3
A1	1	0	0	1	0	0
A2	1	0	0	0	1	0
A3	1	0	0	0	0	1
B1	0	1	0	1	0	0
B2	0	1	0	0	1	0
B3	0	1	0	0	0	1
C1	0	0	1	1	0	0
C2	0	0	1	0	1	0
C3	0	0	1	0	0	1

APL is Fun #8

3×3 Schachbrett
nach Zug A2:

A1	B1	C1
A2	B2	C2
A3	B3	C3

Constraint Matrix M:

	Col.			Row		
	A	B	C	1	2	3
A1	1	0	0	1	0	0
A2	1	0	0	0	1	0
A3	1	0	0	0	0	1
B1	0	1	0	1	0	0
B2	0	1	0	0	1	0
B3	0	1	0	0	0	1
C1	0	0	1	1	0	0
C2	0	0	1	0	1	0
C3	0	0	1	0	0	1

APL is Fun #9

- Der Zug **A2** blockiert also die Züge **A1, A3, B2, und C2**. Diese Züge werden aus der Matrix M entfernt, wodurch aus der ursprünglichen 3×3 Matrix eine 2×2 Matrix für die nächsten Züge verbleibt.

- Der Algorithmus von Donald Knuth repräsentiert die Matrix **M** in einer Weise (doppelt gelinkte Listen entlang der Zeilen und Spalten von **M**) in der das Ausführen von Zügen und – noch wichtiger – die Zurücknahme von Zügen sehr effizient ist.

- In APL ist dies leider mangels entsprechender Datenstrukturen nicht möglich.

APL is Fun #10

- Aus Gründen der Gleichberechtigung wollen wir nun 8 Damen so auf dem Schachbrett plazieren, dass sie sich gegenseitig nicht schlagen können.

- Ausgangspunkt ist die Constraint Matrix für 8 Türme. An diese Matrix werden dann 15+15 Spalten für die Diagonalen angefügt.

- Die Constraint Matrix hat dann (wie zuvor) 64 Zeilen, aber nunmehr 8+8+15+15 = 46 Spalten.

- Der APL code für das (deutlich schwierigere) 8-Damen Problem ist, mit Hilfe von □DLT, ziemlich einfach:

APL is Fun #11

```
      RC←8↑'1' ◊ D←15↑¯8↑'2'   ⍝ helpers for constructing Q8
      ⍝      rows   cols   diag1    diag2
      Q8←⊃{(R⌽RC),(C⌽RC),((C-R)⌽D),((¯7-R+C)⌽D)⊣(R C)←-8 8⊤⍵-⎕IO} ¨ ⍳64
      Z←¯1 ⎕DLX Q8              ⍝ Z ← alle exact covers von Q8

      {⎕UCS (65+⌊⍵÷8)(49+8|⍵←⍵-⎕IO)} ¨ ⊃Z[1 2 3 92] ⍝ covers #1, #2, #3, and #92
A1 B5 C8 D6 G2 E3 F7 H4
A1 B6 C8 D3 E7 F4 G2 H5
A1 B7 C4 E8 D6 G5 F2 H3
A8 B4 C1 D3 G7 E6 F2 H5
```

APL is Fun #12

- Eine andere Anwendung von exact covers ist das Lösen von Sudokus. Ein Standard Sudoku hat 9×9 Felder in die die Ziffern 1..9 gesetzt werden können.
- Die Contraint Matrix des Sudokus hat dann 9×9×9=729 Zeilen
- Es gibt, ähnlich wie beim 8-Türme Problem, 9 Zeilen-Constraints und 9 Spalten-Constraints. Zusätzlich gibt es noch 9 Block Constraints. Die Matrix **M** hat daher 9×27=243 Spalten.
- Statt *einer* Option ("Turm auf B8") pro Feld gibt es nunmehr *neun* Optionen ("Ziffer 1 auf B8", "Ziffer 2 auf B8", … "Ziffer 9 auf B8".

Vielen Dank.