APL-Journal

A Programming Language

1-2/2015

Kai Jäger
Legacy-Code, Überlebensstrategien und Fire

Helmut Engelke
Geometrische Mandalas entwerfen mit APL2/AP207

Jürgen Sauermann
GNU jetzt auch mit APL

IM BLICKPUNKT

Ralf Herminghaus
APL-Benchmark

APL-Germany e.V.
Jahrgang 34 Nr. 1-2 2015 Doppelnummer
ISSN - 1438-4531
RHOMBOS-VERLAG

Liebe APL-Freunde,

Liebe APL Freunde.

ich freue mich sehr, Ihnen das APL Journal 2015 1/2 zukommen zu lassen.

Von unseren letzten Tagungen haben wir interessante Artikel erhalten, welche ein breites Spektrum an Interessengebieten abdecken.

An dieser Stelle bedanke ich mich bei den Autoren für ihre Mühe.

Potentielle Autoren sind herzlich dazu eingeladen, einen Beitrag für das APL-Journal zu verfassen. Ausserdem sind wir stets an interessanten Vorträgen für unsere Tagungen interessiert.

Ich wünsche Ihnen viel Freude bei dem Lesen der vorliegenden Ausgabe des APL-Journals.

Ihr Dr. Reiner Nussbaum

APL Germany

INHALT

Bildnachweis:
Martin Barghoorn (Umschlagseite 1, 4, 59)

Ralf Herminghaus

APL-Benchmark (Vergleich 2012 – 2014) oder Wie schnell ist APL denn heute?

Auf dem APL-Kongress im Jahre 2012 hat der Autor einen Benchmark-Test vorgestellt, der geeignet ist, die Performance von einfachen APL-Statements für unterschiedliche Interpreter zu messen. Damals zeigten sich bereits einige Schwachpunkte der vorgelegten Untersuchung. Die verwendete Hardware etwa entsprach nicht dem aktuellen Stand der Technik. Zudem konnten nicht alle Tests auf dem gleichen Rechner durchgeführt werden, was die Vergleichsmöglichkeiten einschränkte. Auch wurden im Verlauf des damaligen Vortrages eine ganze Reihe von Erweiterungen angeregt, die den Umfang der durchgeführten Tests betrafen. Dies führte dann zu dem Vorhaben, den damaligen Test mit gewissen Abänderungen zu wiederholen, um die Ergebnisse von damals mit den heutigen zu vergleichen.

Verwendete Rechner

Anders als beim ersten Test wurde diesmal ein Rechner mit modernerer Architektur verwendet, um die Leistungen der einzelnen Interpreter unter „realistischen" Bedingungen vergleichen zu können.

- Prozessor: Intel(R) Core(TM) i5-2410M CPU
- Taktung: 2.30GHz
- RAM: 3,85 GB (DDR3)
- Betriebssystem: Windows 7

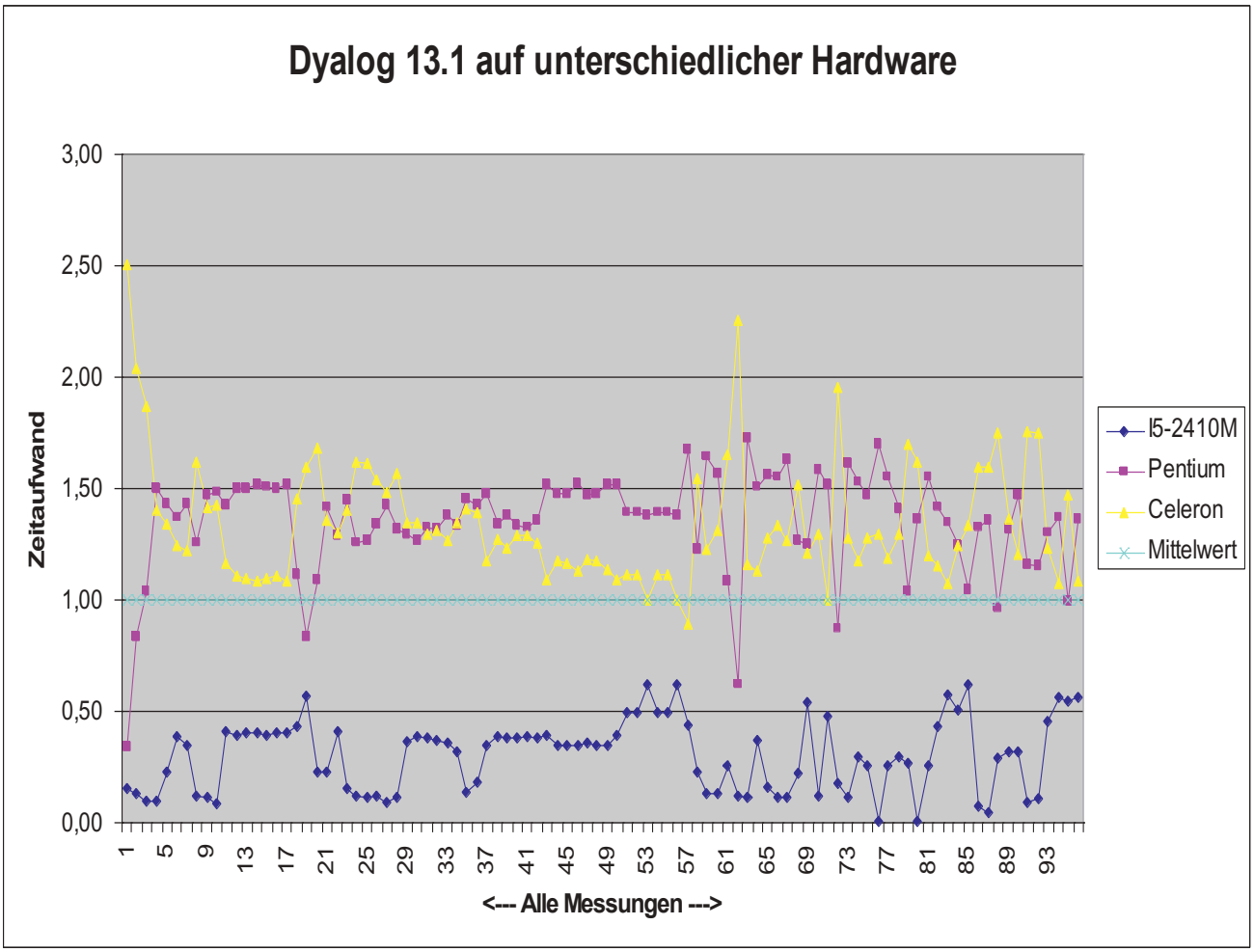
Im Vergleich dazu stehen die im damaligen Test verwendeten Prozessoren:

- Intel(R) Pentium (R) 4 mit 3.00 GHz
- RAM: 982.252 KB
- Betriebssystem: Windows 2000 / SP 3
- Celeron(R) mit 2.60 GHz
- RAM: 240 MB
- Betriebssystem: Windows XP / SP1

Einfluß der Hardware auf die Laufzeit

Am Beispiel des Dyalog 13.1- Interpreters kann man hier recht deutlich den Performance-Gewinn erkennen, der sich aus der Verwendung einer modernen Prozessor-Architektur ergeben kann. Gleichzeitig erkennt man aber auch, wie wenig homogen die erzielten Performance-Gewinne bei unterschiedlichen Prozessoren sind.

Im vorliegenden Fall werden alle Meßergebnisse des Dyalog 13.1-Interpreters



auf den drei unterschiedlichen Hardware-Plattformen dargestellt.

Als Referenzlinie (Wert 1) dient hierbei der Mittelwert über alle drei getesteten Plattformen.

Wie man an diesem Beispiel recht deutlich sieht, sind Benchmark-Messungen generell mit einer großen Unschärfe belastet, die man bei der Interpretation der Ergebnisse nie außer Acht lassen darf!

Insbesondere kann man auch erkennen, wie schwierig es ist, das Laufzeitverhalten eines Interpreters zu optimieren: Änderungen an einem Interpreter können abhängig von der verwendeten Hardware entweder positive oder negative Auswirkungen haben.

Getestete Interpreter

Beim vorliegenden Test bestand die Gelegenheit, nicht nur die Leistungsparameter der Interpreter unterschiedlicher Hersteller zu vergleichen, sondern insbesondere auch die Leistungsmerkmale von Interpretern aus unterschiedlichen Releases gegeneinander zu testen.

Dabei waren die Verbesserungen von APL2000 Rel.: 9.1 nach Rel.: 13.1 und die Verbesserungen des Dyalog-Intepreters von Rel.13.0 nach Rel.14.0 von besonderem Interesse.

Getestet wurden

- APLX Version 5.0.5 / MicroAPL Ltd
- IBM APL2 Version 2.0 / Service Level 19
- APL+WIN Rel. 9.1
- APL+WIN Rel. 13.1
- Dyalog APL 13.0
- Dyalog APL 14.0 (32-Bit)
- Dyalog APL 14.0 (64-Bit)

Test-Mechanismus

Da das vorgestellte Benchmark-Programm auf allen verwendeten Interpretern lauffähig sein muss, ist man bei der Programmierung naturgemäß auf den APL-ANSI-Standard beschränkt – soweit er bei den getesteten Interpretern denn implementiert ist. Der Test-Mechanismus ist daher bewusst sehr einfach gehalten. Der Kern des Tests wird hier kurz dargestellt werden.

Zunächst einmal betrachten wir die im Test verwendeten APL-Datenobjekte, die hier in APL-Schreibweise definiert werden.

Der eigentliche Messvorgang ist unterteilt in vier Schritte, bei denen die gesuchte Performance über die Laufzeit gemessen wird. Die Schwächen dieser Vorgehensweise wurden bereits im Zusammenhang mit dem vorangegangenen Test 2012 ausführlich diskutiert und sollen hier nicht weiter beleuchtet werden.

Anzumerken ist aber, dass für die Messungen möglichst alle störenden Einflüsse – insbesondere parallel zum Test ablaufende anderweitige Prozesse – ausgeschaltet werden müssen, um das Ergebnis möglichst wenig zu verfälschen. Trotzdem gelingt es nicht immer die Messergebnisse wirklich genau reproduzierbar zu machen.

```
⍝------------------------------------------------------------
⍝ Test-Objects
⍝------------------------------------------------------------
X←(?Nρ1000)×0.001  ⍝---NUM-Vector-(Floating Point)---
Y←(?Nρ1000)×0.001  ⍝---NUM-Vector-(Floating Point)---
XI←⌈100×X          ⍝---NUM-Vector-(Integer)----------
YI←⌈100×Y          ⍝---NUM-Vector-(Integer)----------
B←X>Y              ⍝---Boolean Vector----------------
IND_B←B/⍳⍴B        ⍝---Index Representation of B-----

sN←10              ⍝---Size of short Strings--------
L←Nρ⎕AV[65+(?(N)ρ60)]            ⍝---Long String----
S←sNρ⎕AV[65+(?(sN)ρ60)]          ⍝---Short String---
K←(N sN)ρ⎕AV[65+(?(N×sN)ρ60)]    ⍝---Character-Key-Table----
KX←⊂[2]K                         ⍝---Character-Key-Vector---

AV_FIRST←⍎⎕AV
AV_LAST←⍎¯1↑⎕AV

sN2←20             ⍝---Size of short Strings--------
S2←sN2ρ⎕AV[65+(?(sN2)ρ60)]       ⍝---Short String---
K2←(N sN2)ρ⎕AV[65+(?(N×sN2)ρ60)] ⍝---Character-Key-Table----
KX2←⊂[2]K2                       ⍝---Character-Key-Vector---

PI←○1                            ⍝---3.14159265...----------

M←X∘.+Y
MX←⊂[2]M
MC←(N N)ρ⎕AV[65+(?(N×N)ρ60)] ⍝---Character-Matrix---
H1←(+/+/M)÷(N×N)
MB←(N N)ρ(M>H1)               ⍝---Boolean-Matrix-----
```

Messung / Schritt 1

(Warten, bis die Systemuhr eine volle Sekunde zeigt)

```
TS1←⎕TS
⍝---------------------------------
⍝ 1) Wait a  second ...
⍝---------------------------------
LOOP01:
     TS2←⎕TS
     →((6⊃TS2)≠(6⊃TS1))/END_LOOP01
     →LOOP01
END_LOOP01:
```

Messung / Schritt 2

(Eine Sekunde lang immer dasselbe tun – so oft wie möglich)

```
⍝-----------------------------------------
⍝ 2) Measuring (for 1 Sec)
⍝-----------------------------------------
I←0
LOOP02:
     I←I+1
     DUMMY←⍎'OPERATION'    ⍝---Do Operation---
     TS3←⎕TS
     →(I>I)/END_LOOP02   ⍝---Dummy-Compare--
     →((6⊃TS3)≠(6⊃TS2))/END_LOOP02
     →LOOP02
END_LOOP02:
```

Messung / Schritt 3

(Genau so oft fast nichts tun)

```
⍝-----------------------------------------
⍝ 3) Measuring Loop-Overhead
⍝-----------------------------------------
J←0
LOOP03:
     J←J+1
     DUMMY←⍎'0'          ⍝---Do almost nothing---
     TS4←⎕TS
     →(J≥I)/END_LOOP03   ⍝---Non-Dummy-Compare---
     →((6⊃TS4)≠(6⊃TS3))/END_LOOP03
     →LOOP03
END_LOOP03:
```

Messung / Schritt 4

(Am Schluss die gemessenen Timestamps auswerten)

```
⍝-------------------------------------------------------------
⍝ 4) Calculating Result
⍝-------------------------------------------------------------
PATTERN←(1 1 1 1 60 60 1000)            ⍝ Timestamp-Structure
USED_TIME←0.001×PATTERN ⊥ (TS3-TS2)     ⍝ (seconds)
LOOP_OVERHEAD←0.001×PATTERN ⊥ (TS4-TS3) ⍝ (seconds)

TIME←1000000×(USED_TIME-LOOP_OVERHEAD)÷I ⍝ micro-seconds
RESULT←⌊(0.5)+TIME                       ⍝ Rounding
```

Probleme mit der Kompatibilität

Entwickelt wurde der Benchmark ursprünglich auf dem APL+WIN 9.1 – Interpreter. Dabei wurde recht sorgfältig darauf geachtet, den ANSI-Standard einzuhalten, um die Software auf allen getesteten Interpretern unverändert laufen lassen zu können. Aber bereits beim Transfer der Software zeigten sich empfindliche Schwierigkeiten:

1. Die Definition einzelner Operationen weicht etwa bei DYALOG vom ANSI-Standard ab. Abhilfe bietet hier ein Kompatibilitätsmodus („Migration-Level“), der es erlaubt, APL-Programme entsprechend des ANSI-Standards auszuführen. Obwohl das Problem der Inkompatibilität damit recht gut gelöst ist, bleibt immer noch eine gewisse Fehler-Wahrschinlichkeit infolge von Inkompatibilitäten erhalten.

2. Deutlich ärgerlicher ist die Tatsache, dass etwa das APL-Austausch-Format *. ATF durchaus nicht alle APL-Character sauber von einem Interpreter zum anderen transferieren kann. Im Großen und Ganzen klappt es zwar halbwegs,

aber einzelne Zeichen werden beim Transfer unerwartet verändert, was dann ggf. Abstürze oder – schlimmer noch – eine geänderte Bedeutung der Programme bewirkt. Betroffen sind hier mehr oder weniger alle getesteten Interpreter. Ohne hier speziell einzelne Hersteller für den Missstand verantwortlich machen zu können, muss man sagen, dass das Transfer-Format schlichtweg nicht 100% zuverlässig funktioniert.

3. Was sich bei einem kleinen Benchmark-Programm als ärgerliche Unbequemlichkeit darstellt, wird bei großen Software-Paketen leicht zum unüberwindlichen Hindernis. Allein schon das schlichte Aufschreiben eines APL-Ausdruckes im Rahmen einer Mail wird zum Problem, wenn der Empfänger nicht genau den gleichen APL-Zeichensatz verwendet wie der Sender. Neben der „steilen Lernkurve", die APL gerne bescheinigt wird, und dem Mangel an APL-Programmierern im Arbeitsmarkt, sind es sicher auch derartige Schwierigkeiten, die den Einsatz von APL bei IT-Managern als zu risikoreich in Verruf bringen.

Ich meine, dass hier die gesamte APL-Community gefordert ist, Abhilfe zu schaffen und verbindliche Standards zu definieren, um APL-Texte unbeschadet von einem System ins andere transferieren zu können.

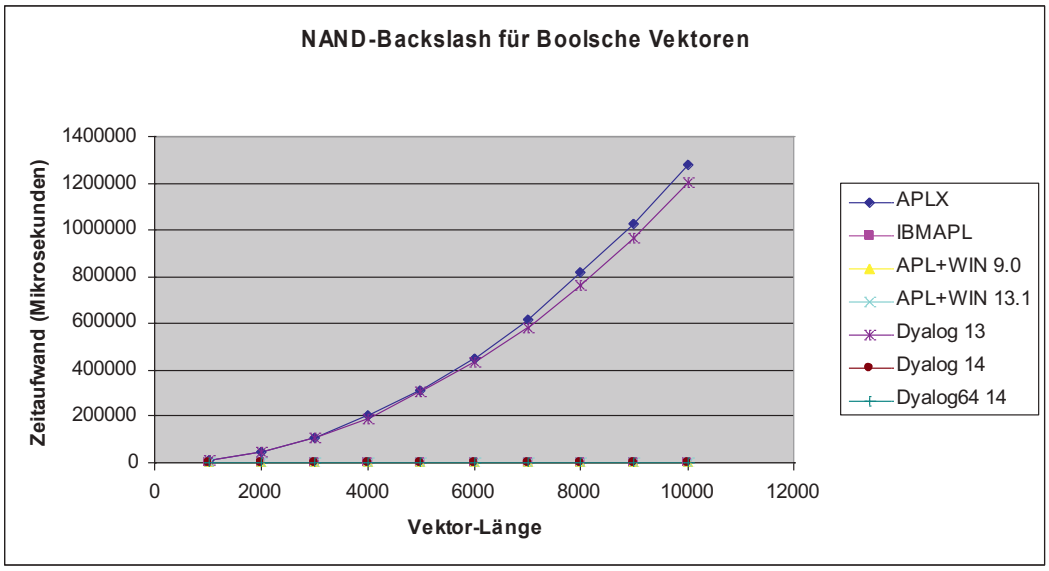
Die Ergebnisse des Benchmark-Tests stelle ich an einigen besonders interessanten Beispielen dar.

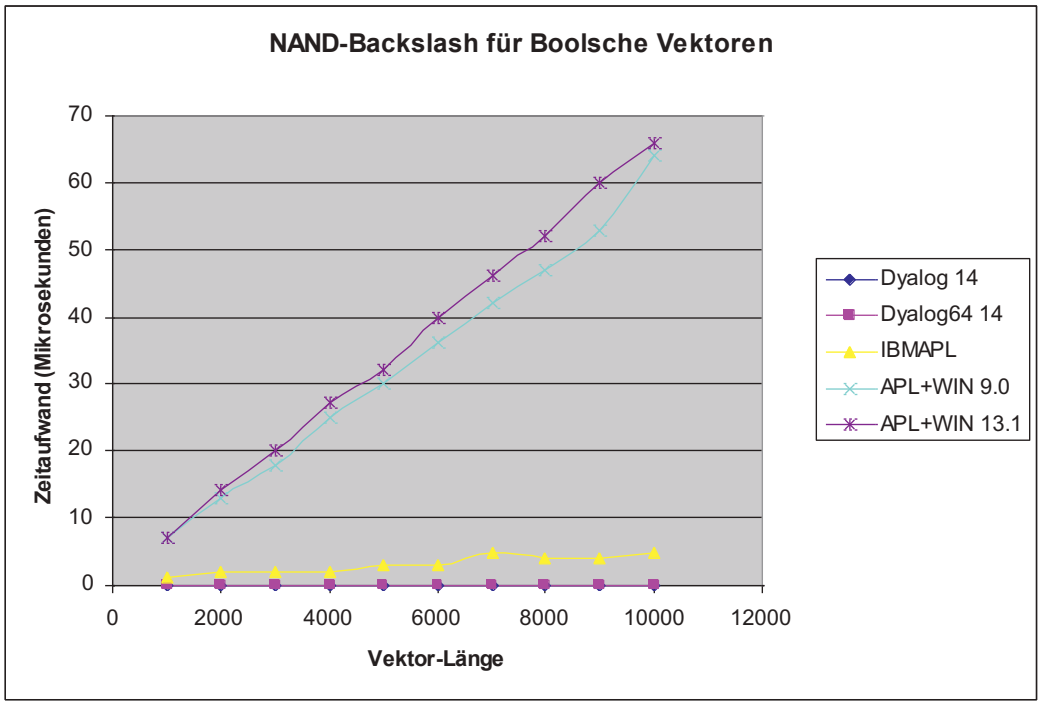
Beispiel 1: NAND-Backslash

Die Operation NAND-Backslash ist insofern interessant, als hier noch deutlicher als in anderen Funktionen Unterschiede in der Implementation sichtbar werden. So steigt der Aufwand bei den Interpretern DYALOG 13 und APLX quadratisch mit der länge der Eingabe-Vektoren. Bei den Interpretern von APL2000 (APL+WIN 9.1 und 13.1) steigt der Aufwand immerhin noch linear, während er bei IBM und den DYALOG14-Interpretern nahezu unabhängig von der Länge der Argumente bleibt.

Operation: ⍲\B

N	Dyalog 14	Dyalog64	IBMAPL	APL+WIN 9.0	APL+WIN 13	Dyalog 13	APLX
1000	0	0	1	7	7	12024	12183
2000	0	0	2	13	14	47524	49000
3000	0	0	2	18	20	107600	109111
4000	0	0	2	25	27	192333	199600
5000	0	0	3	30	32	304000	312000
6000	0	0	3	36	40	431667	447000
7000	0	0	5	42	46	585000	616000
8000	0	0	4	47	52	764000	819000
9000	0	0	4	53	60	967500	1029000
10000	0	0	5	64	66	1202000	1279000



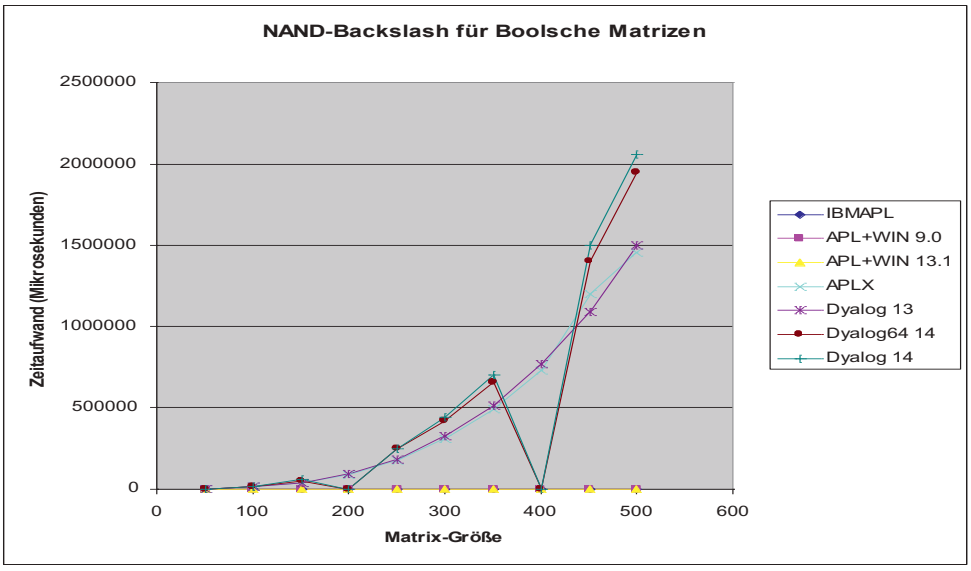


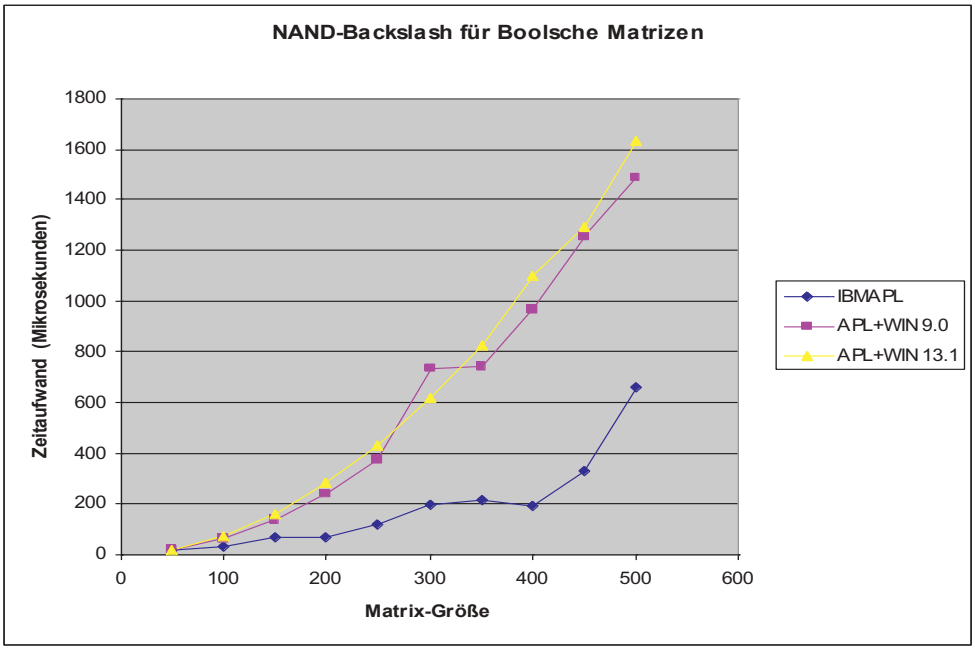
Hier zeigen sich deutlich Performance-Verbesserungen, die DYALOG bei der Weiterentwicklung von Rel.13 zu Rel.14 durchgeführt hat. Man könnte vermuten, dass ein entsprechender Test mit Boolschen Matritzen ganz analoge Ergebnisse liefern würde. Überraschenderweise zeigt der gleiche Ausdruck mit Boolschen Matrizen aber ein deutlich anderes Performance-Verhalten.

Auffällig ist hier nicht nur das im Allgemeinen schlechte Abschneiden der DYALOG14-Interpreter, sondern auch die beiden extremen Performance-Peaks bei N=200 und N=400 . Dies lässt den Schluss zu, dass der zugrundeliegende Algorithmus gerade auf Werte von N als Vielfaches von 8 hin optimiert wurde. In diesen speziellen Fällen schneidet der Interpreter deutlich besser ab als alle Konkurrenten.

Operation: ⍀\MB

N	IBMAPL	APL+WIN 9.0	APL+WIN 13.1	APLX	Dyalog 13	Dyalog64 14	Dyalog 14
50	19	18	20	1483	1521	1924	2051
100	30	63	71	11471	12183	15136	16361
150	65	137	161	38385	40560	50700	55444
200	65	240	279	92182	96455	3	7
250	115	373	428	179500	187167	253500	253500
300	194	732	618	312000	323500	421333	442000
350	217	738	826	494000	515000	655500	702000
400	189	970	1103	733000	764500	8	16
450	330	1255	1293	1201000	1092000	1404000	1497000
500	659	1487	1635	1451000	1498000	1950000	2060000





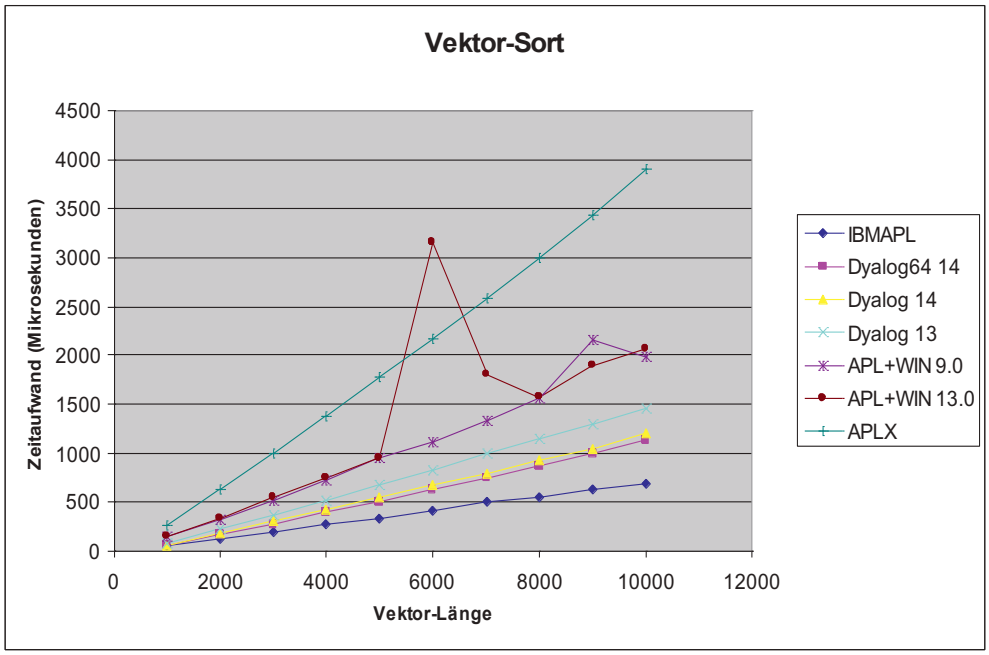
Beispiel 2: Sort

Im Gegensatz zu der oben beschriebenen NAND-Backslash-Operation ist der SORT eine in der Informatik immer wieder benötigte Grund-Funktionalität. Hier ist es nicht nur von akademischem Interesse, einen möglichst leistungsfähigen Algorithmus zur Verfügung zu stellen. Sortieralgorithmen sind ein ausführlich erforschtes Thema der Informatik. Um so erstaunlicher ist die immer noch recht breite Streuung der Performance-Werte.

Auffällig ist hier neben der breiten Streuung der Ergebnisse das nahezu lineare Laufzeitverhalten der verschiedenen Algorithmen. Folgt man der Literatur, würde man eher ein Laufzeitverhalten in der Größenordnung O(n*log(n)) erwarten. Evtl. würde dies auch erst bei deutlich größeren Eingabe-Vektoren erkennbar werden.

Operation: (⍋X)

N	IBMAPL	Dyalog64 14	Dyalog 14	Dyalog 13	APL+WIN 9.0	APL+WIN 13.0	APLX
1000	61	58	64	82	150	148	266
2000	122	164	180	222	328	337	622
3000	194	273	310	369	528	556	992
4000	269	397	422	523	724	750	1380
5000	342	509	557	680	942	954	1776
6000	417	628	673	826	1110	3154	2171
7000	498	752	801	994	1327	1804	2592
8000	555	871	939	1144	1564	1583	3006
9000	625	997	1047	1301	2162	1894	3433
10000	694	1129	1213	1453	1984	2062	3902



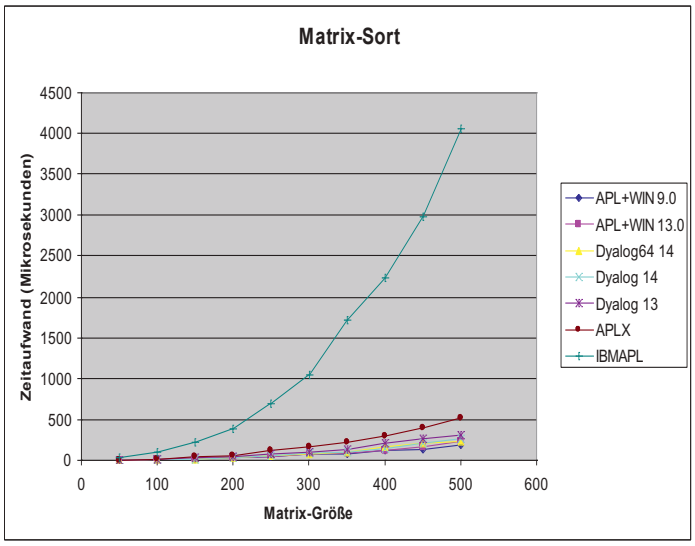
Besonders auffällig ist das Ergebnis beim Interpreter APL+WIN 13.1

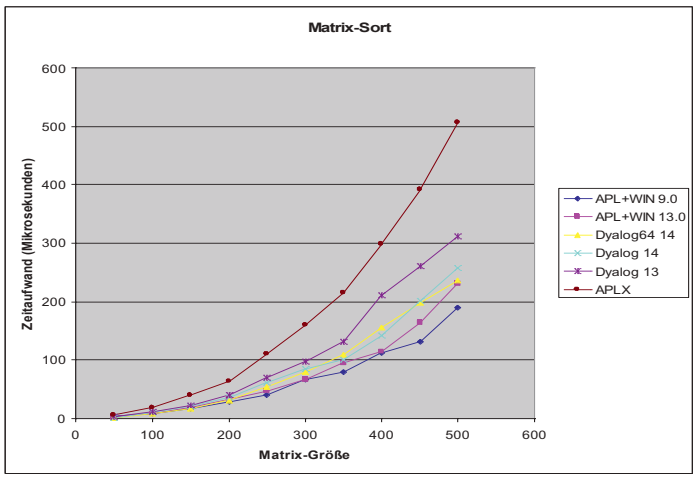
Wahrscheinlich handelt es sich hier aber um einen Effekt, der aus einer zufälligen Konstellation der Eingabe-Daten resultiert. Wie man weiß, reagieren manche Sortier-Algorithmen auf ungünstige Eingabe-Daten ein wenig allergisch .....

Der Interpreter von IBM zeigt sich in diesem Fall als der leistungsstärkste. Um so mehr muss das folgende Ergebnis erstaunen, das sich bei der Sortierung von numerischen Matrizen ergibt.

Operation: (⍋M)

N	APL+WIN 9.0	APL+WIN 13.0	Dyalog64 14	Dyalog 14	Dyalog 13	APLX	IBMAPL
50	4	3	2	2	3	6	34
100	8	9	8	10	11	19	98
150	17	19	17	21	22	40	217
200	29	32	31	36	40	63	385
250	40	46	54	61	70	111	689
300	66	66	79	85	98	160	1044
350	79	95	109	101	131	215	1709
400	112	115	156	141	211	299	2235
450	131	164	198	201	261	392	2982
500	189	231	237	258	312	506	4061





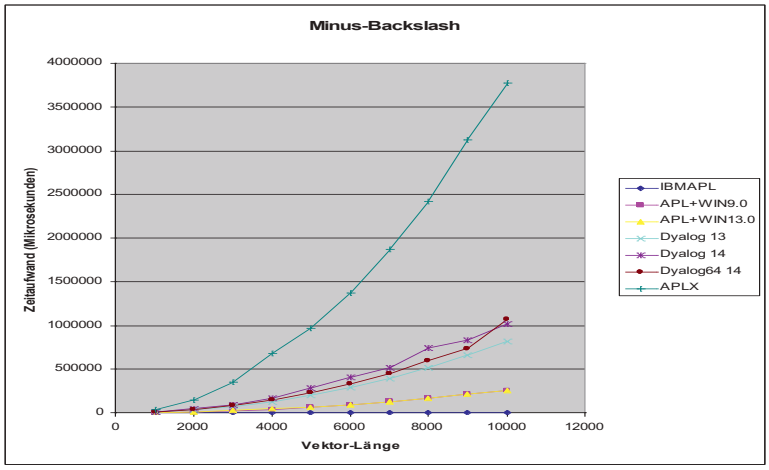
Im Gegensatz zur Sortierung von numerischen Vektoren schneidet der IBM-Sort im Falle numerischer Matrizen deutlich schlechter ab als alle seine Konkurrenten. Wie es dazu kommen kann, bleibt – vor allem angesichts der guten Ergebnisse bei der ganz analogen Sortierung von Vektoren – unklar.

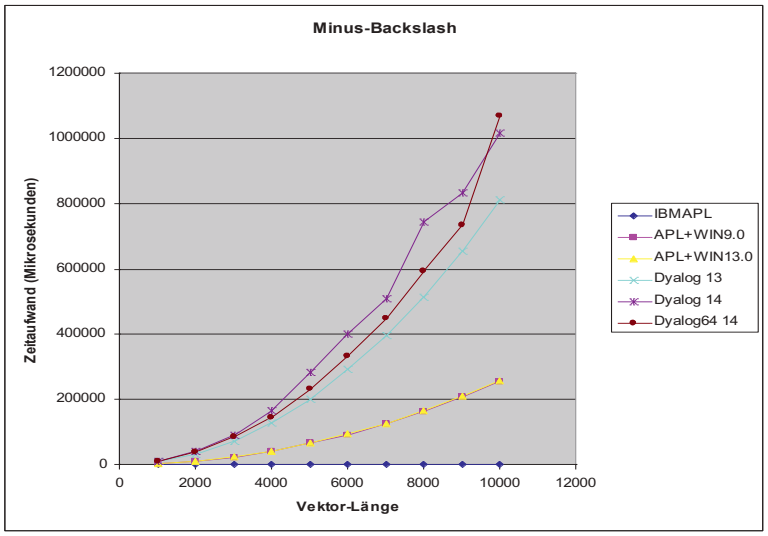
Beispiel 3: Minus-Backslash

Diese Operation ist – wenn auch nicht so wichtig wie der SORT – eine recht häufig zu findende Funktion in mathematischen Programmen. Obwohl unter Ausnutzung ihrer mathematischen Eigenschaften eine effiziente Implementation möglich wäre, wird dies ganz offensichtlich von den meisten Interpretern nicht genutzt.

Operation: "-\X"

N	IBMAPL	APL+WIN9.0	APL+WIN13.0	Dyalog 13	Dyalog 14	Dyalog64 14	APLX
1000	99	2599	2640	8048	10347	9241	38423
2000	197	10194	10406	32194	41583	36963	149286
3000	292	22682	23786	72429	90727	85833	343000
4000	389	40560	41583	128750	164857	144857	678500
5000	501	64313	64313	199800	281000	230800	967500
6000	586	92182	93545	292500	400333	332667	1373000
7000	685	126750	126750	395000	507000	447333	1872000
8000	966	162714	164857	514500	741000	593000	2418000
9000	876	205800	209000	655500	834500	733000	3120000
10000	973	253500	257250	811000	1014000	1068500	3776000





Beispiel 4: Loops

Einen interessanten Aspekt des Performance-Verhaltens, der im Benchmark 2012 noch nicht berücksichtigt worden ist, stellen verschiedene Arten von Loops dar.

Anhand von drei typischen Loop-Konstruktionen lassen sich zumindest einige interessante Erkenntnisse gewinnen.

```
[0]   Z←LABEL_LOOP N;I
[1]   I←0
[2]   LOOP: →(I>N)/END_LOOP
[3]       LABEL001:
[4]       LABEL002:
[5]       LABEL003:
[6]       LABEL004:
[7]       LABEL005:
[8]       LABEL006:
[9]       LABEL007:
[10]      LABEL008:
[11]      LABEL009:
[12]      LABEL010:
[13]      LABEL011:
[14]      LABEL012:
[15]      LABEL013:
[16]      LABEL014:
[17]      LABEL015:
[18]      LABEL016:
[19]      LABEL017:
[20]      LABEL018:
[21]      LABEL019:
[22]      LABEL020:
[23]      I←I+1
[24]      →LOOP
[25]  END_LOOP:
[26]  Z←0
```

```
[0]   Z←LABEL_LOOP2 N;I
[1]   I←0
[2]   LOOP: →(I>N)/END_LOOP
[3]       I←I+1
[4]       →LOOP
[5]   END_LOOP:
[6]   Z←0
[7]   →0
[8]
[9]   LABEL001:
[10]  LABEL002:
[11]  LABEL003:
[12]  LABEL004:
[13]  LABEL005:
[14]  LABEL006:
[15]  LABEL007:
[16]  LABEL008:
[17]  LABEL009:
[18]  LABEL010:
[19]  LABEL011:
[20]  LABEL012:
[21]  LABEL013:
[22]  LABEL014:
[23]  LABEL015:
[24]  LABEL016:
[25]  LABEL017:
[26]  LABEL018:
[27]  LABEL019:
[28]  LABEL020:
[29]
```

```
[0]   Z←LABEL_LOOP3 N;I
[1]   I←0
[2]   LOOP: →(I>N)/END_LOOP
[3]       A---Empty line---
[4]       A---Empty line---
[5]       A---Empty line---
[6]       A---Empty line---
[7]       A---Empty line---
[8]       A---Empty line---
[9]       A---Empty line---
[10]      A---Empty line---
[11]      A---Empty line---
[12]      A---Empty line---
[13]      A---Empty line---
[14]      A---Empty line---
[15]      A---Empty line---
[16]      A---Empty line---
[17]      A---Empty line---
[18]      A---Empty line---
[19]      A---Empty line---
[20]      A---Empty line---
[21]      A---Empty line---
[22]      A---Empty line---
[23]      I←I+1
[24]      →LOOP
[25]  END_LOOP:
[26]  Z←0
[27]  →0
[28]
[29]  LABEL001:
[30]  LABEL002:
[31]  LABEL003:
```

PART16= Loops		APLX	IBM-APL2	APL+WIN 9.0	APL+WIN 13.1	Dyalog 13.0	Dyalog 14.0	Dyalog64 14.0
10000	(FOR_LOOP N)	15609	14676	12808	5198	6616	6836	7085
10000	(LABEL_LOOP N)	53368	30242	32226	10967	15121	16915	19132
10000	(LABEL_LOOP2 N)	15594	14676	17483	5147	6609	6705	7141
10000	(LABEL_LOOP3 N)	58706	27405	12183	5171	16361	18107	19569

Was sagen uns die Ergebnisse ?

1. Der Vergleich zwischen LABEL_LOOP und LABEL_LOOP2 zeigt, dass Labels durchaus keine passiven Objekte darstellen, die lediglich als Markierungen im Programm stehen, sondern dass das Durchlaufen eines Labels bei allen untersuchten Interpretern Zeit kostet.
2. Eine etwas überraschende Erkenntnis liefert der Vergleich von LABEL_LOOP2 und LABEL_LOOP3: Wie man hier erkennen kann, benötigt bei den meisten Interpretern auch das Durchlaufen einer einfachen Kommentarzeile eine gwisse Zeit. Lediglich APL+WIN macht hier eine Ausnahme. Dort werden Kommentar-Zeilen offenbar vor dem Fixing der Funktion bereits entfernt.
3. Die Ergebnisse des LABEL_LOOP2–Experimentes lassen evtl. gewisse Rückschlüsse auf die allgemeine Interpreter-Geschwindigkeit zu. Hier liegen APL-WIN 13.1 und die DYALOG-Interpreter deutlich vorne. Zudem spiegeln die Unterschiede zwischen APL-WIN 9.0 und APL-WIN 13.1 deutlich die Anstrengungen wider, die APL2000 in der Vergangenheit zur allgemeinen Performance-Verbesserung unternommen hat.

Was hat sich seit 2012 verbessert ?

Diese Frage bezieht sich auf die Interpreter von APL+WIN und DYALOG, die diesmal in verschiedenen Versionen getestet wurden. Eine generelle Antwort läßt sich leider nicht geben.

Viele Einzelfunktionen wurden in ihrem Laufzeit-Verhalten verbessert. Überraschenderweise gab es aber auch einige wenige Fälle, in denen der neue Interpreter langsamer lief als der alte.

Hier einige Beispiele:

PART1= Standard-NUM-Functions		APL+WIN 9.0	APL+WIN 13.1	Dyalog 13.0	Dyalog 14.0
10000	X+Y	121	119	15	13
10000	X-Y	119	120	16	13
10000	X×Y	131	125	13	13
10000	X÷Y	87	86	295	81
10000	X*Y	1273	1159	1038	1186
10000	X!Y	3596	1947	9505	7400
10000	X⌊Y	90	86	30	32
10000	X⌈Y	94	87	34	35
10000	-X	171	145	119	98
10000	0-X	29	53	14	13
10000	÷X	232	231	379	211
10000	1÷X	88	125	257	82
10000	×X	40	41	112	107
10000	|X	144	138	121	96
10000	!X	1000	673	2541	2045

PART4= Accumulated Operations on NUM-Vectors (\)		APL+WIN 9.0	APL+WIN 13.1	Dyalog 13.0	Dyalog 14.0
10000	+\X	38	50	146	137
10000	×\X	37	47	161	154
10000	⌊\X	146	145	150	14
10000	⌈\X	140	139	146	12
10000	-\X	257500	253500	655000	1014000
10000	ΔMIN_BS X	583	592	309	271
10000	÷\X	447000	447333	2153000	1591000
10000	ΔDIV_BS X	645	626	443	341

PART8= Boolean Operations (\)		APL+WIN 9.0	APL+WIN 13.1	Dyalog 13.0	Dyalog 14.0
10000	^\B	66	69	0	0
10000	∨\B	61	66	0	0
10000	⍲\B	61	66	982500	0
10000	⍱\B	77	68	975000	0
10000	+\B	61	59	247	6
10000	~B	4	4	0	0

PART9= Matrix Operations		APL+WIN 9.0	APL+WIN 13.1	Dyalog 13.0	Dyalog 14.0
500	⌽M	463	386	4073	534
500	⊖M	3006	1998	1160	1229
500	⍉M	1147	1120	394	397
500	M+M	2997	2970	361	430
500	(M+.×M	874000	1014000	93545	92182
500	(+/M)	906	1042	778	327
500	(+⌿M)	915	1257	518	196
500	(X∘.×Y	4061	4057	242	257

PART10= Selection		APL+WIN 9.0	APL+WIN 13.1	Dyalog 13.0	Dyalog 14.0
500	B/X	11	10	0	0
500	X[IND_	2	2	0	1
500	B/M	3286	3240	1533	408
500	M[;IND_	318	370	288	284
500	B⌿M	540	573	160	163
500	M[IND_	316	343	153	171
500	⍉B⌿⍉M	4875	3283	1852	1968

PART11= Restructuring		APL+WIN 9.0	APL+WIN 13.1	Dyalog 13.0	Dyalog 14.0
400	B⊂M	8317	8387	3046	3024
400	⊂[2]M	266	507	241	266
400	⊃MX	306	464	227	240
400	⌽M	234	182	2536	247
400	⍉M	1436	871	604	623
400	⊖M	712	712	203	215

PART12= Key String search (length 10)		APL+WIN 9.0	APL+WIN 13.1	Dyalog 13.0	Dyalog 14.0
10000	∨/S⍷K	427	542	81	82
10000	(S⍷K)[	405	381	60	57
10000	K∧.=S	46	45	59	58
10000	KX⍳⊂S	578	450	401	435
10000	KX≡¨⊂S	2126	1110	1388	1270
10000	KX⍳KX	1625	1160	2879	2991

PART26= Character-Set-Check		APL+WIN 9.0	APL+WIN 13.1	Dyalog 13.0	Dyalog 14.0
10000	L∈S	68	70	196	156
10000	L∈S2	82	87	226	181
10000	L∈L	58	65	236	215
10000	L⍳S	12	9	78	84
10000	L⍳S2	12	9	104	81
10000	L⍳L	37	35	212	236

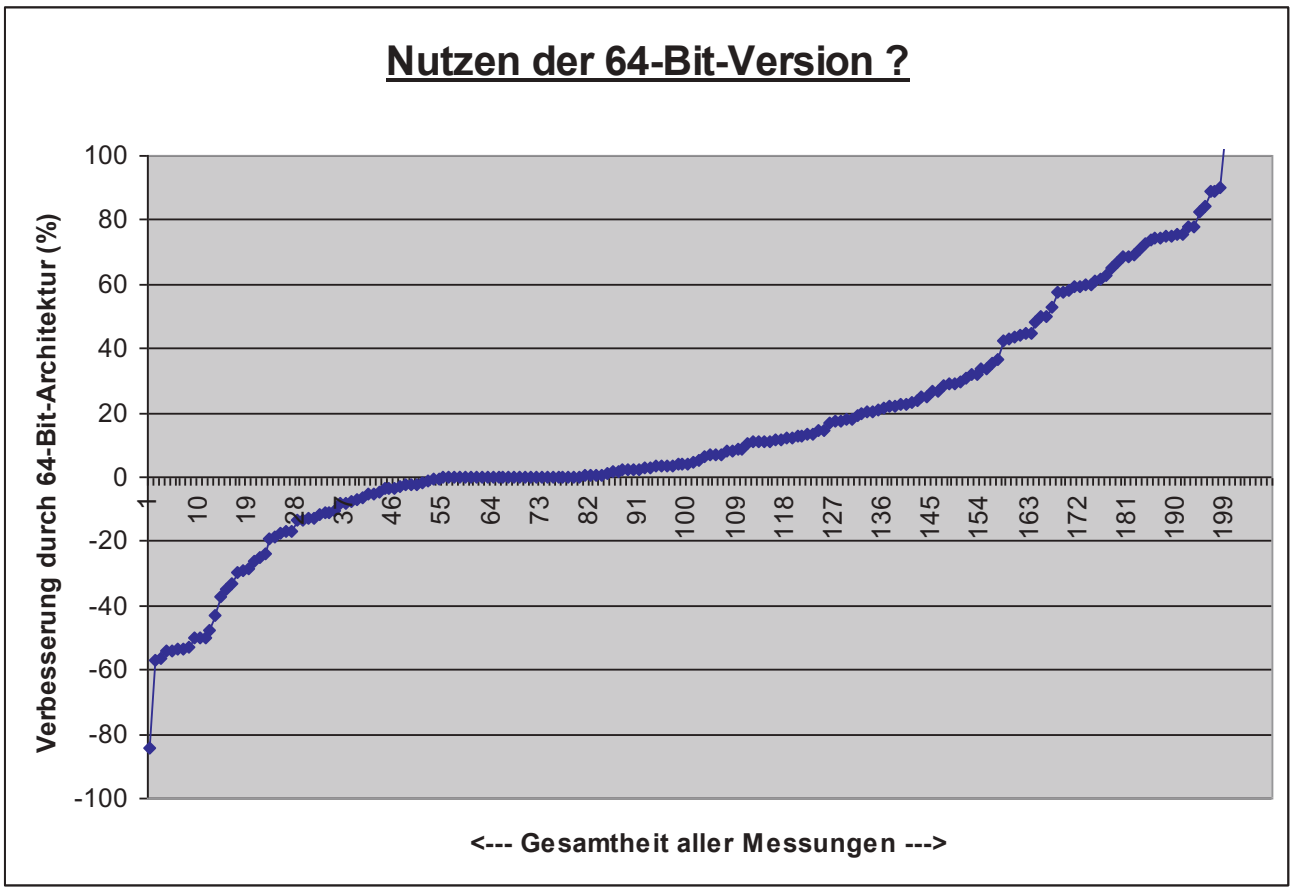
PART27= Logarithmic functions		APL+WIN 9.0	APL+WIN 13.1	Dyalog 13.0	Dyalog 14.0
10000	X*Y	1284	1186	1015	1322
10000	2*Y	1375	1389	1037	1168
10000	X*1	37	54	283	114
10000	1*Y	29	61	1040	1065
10000	⍟Y	760	470	733	613
10000	2⍟Y	648	703	1923	1013
10000	10⍟Y	698	754	1910	1007

Lohnt sich der Umstieg auf eine 64-Bit-Version?

Die Frage der Weiterentwicklung von APL in Richtung auf eine 64-Bit-Architektur wurde auf den letzten Tagungen von APL Germany immer wieder heiß diskutiert. Die Frage hat mehrere Aspekte: Einerseits ist zu erwarten, dass der Markt in absehbarer Zukunft generell nur noch Software akzeptiert, die auf 64-Bit-Hardware angepasst ist. Das hängt großenteils zusammen mit dem erweiterten Adressraum, den man mit 64-Bit ansprechen kann. Andererseits erwartet man bei der Anpassung der Software auf 64 Bit aber auch einen gewissen Performance-Gewinn, der aus dem „nahtlosen" Zusammenspiel von Software und Hardware resultieren sollte. Einziger Kandidat für eine solche Untersuchung war im vorliegenden Fall der DYALOG-14-Interpreter, der als 32-Bit- und als 64-Bit-Version zur Verfügung stand.

Um beide Interpreter zu vergleichen, sind in nachstehender Abbildung die Laufzeit-Verhältnisse bei allen durchgeführten Tests einmal gegeneinander aufgetragen.

Die Ergebnisse kann man als „durchwachsen" bezeichnen. Bei einigen Tests zeigen sich Performance-Gewinne, bei anderen Performance-Verluste. Ein deutlicher Nutzen der 64-Bit-Version lässt sich mit Blick auf die Performance jedenfalls nicht erkennen.



Die Entscheidung, auf eine 64-Bit-Version umzusteigen, sollte sich daher primär am Verhalten des Marktes bzw. an den verbesserten Möglichkeiten eines 64-Bit-Address-Raumes orientieren. Ein deutlicher Performance-Gewinn läßt sich aufgrund der Ergebnisse zum jetzigen Zeitpunkt nicht erwarten.

Wo steht eigentlich die Konkurrenz?

Die Konkurrenz zum klassischen APL ist groß geworden.

Zum einen gibt es nicht-klassische APL-Dialekte wie „J“ oder „K“ , die Marktanteile für sich beanspruchen, zum anderen bestimmen moderne Sprachen wie „Python“, „Jython“, „Ruby“ oder „R“ die aktuelle Diskussion in der Informatik-Community. Betrachtet man es vom Standpunkt der sprachlichen „Expressivität“ her, so können diese Sprachen durchaus mit APL mithalten. Zudem handelt es sich zum großen Teil um Open-Source-Produkte, die jeder ohne irgendwelche Probleme und Kosten aus dem Internet herunterladen kann. Aber wie performant sind diese Konkurrenten?

Exemplarisch habe ich hier einige Tests mit dem aktuell verfügbaren Python-Interpreter durchgeführt. Wie auch APL verfügt Python über einen „EACH-Operator“ (der dort „map“ heißt) und über ein „reduce“, das dem „/“ im APL entspricht.

Die Ergebnisse sind interessant.

PYTHON 2.7			DYALOG APL 13.0			
PART1P: Standard-NUM-Functions			Explicit EACH-Operator		Implicit EACH-Operator	
10000	'map(add,X,Y)	1674	X+¨Y	2719	X+Y	16
10000	'map(mul,X,Y)	1655	X×¨Y	2668	X×Y	12
10000	'map(minus,X,Y)	1774	X−¨Y	2722	X-Y	16
10000	'map(div,X,Y)	1668	X÷¨Y	3590	X÷Y	289
10000	'map(modulo,X,Y)	1906	Y|¨X	4775	Y|X	1109
PART2P: Integer-Functions			Explicit EACH-Operator		Implicit EACH-Operator	
10000	'map(add,XI,YI)	1452	XI+¨YI	2225	XI+YI	8
10000	'map(mul,XI,YI)	1562	XI×¨YI	2033	XI×YI	8
10000	'map(minus,XI,YI)	1547	XI−¨YI	1947	XI-YI	8
10000	'map(div,XI,YI)	1594	XI÷¨YI	3700	XI÷YI	321
10000	'map(modulo,XI,YI)	1644	YI|¨XI	2119	YI|XI	86
PART3P: Compare-Functions			Explicit EACH-Operator		Implicit EACH-Operator	
10000	'map(abs,X)	705	|¨X	1745	|X	118
10000	'map(cmp,X,Y)	1228	X cmp¨Y	12974	X cmp Y	312
PART4P: reduce-Functions			Transl.to APL			
10000	'min(X)	273	⌊/X	8		
10000	'max(X)	266	⌈	8		
10000	'reduce(add,X)	1278	+/X	38		
10000	'reduce(minus,X)	1265	−/X	276		
10000	'reduce(mul,X)	1959	×/X	21		
10000	'reduce(div,X)	2552	÷/X	556		

APL als Array-orientierte Sprache liegt deutlich vorne, wenn der implizit immer vorhandene EACH-Operator der Grundfunktionen Anwendung findet.

Der explizit implementierte EACH-Operator scheint aber im APL etwas langsamer zu arbeiten. Es stellt sich die Frage: Warum?

Fazit: Es bleibt schwierig

Im Vergleich zu Konkurrenz-Sprachen kann APL mit einigen Vorteilen punkten. Da ist zum einen die hohe Expressivität der Sprache, die eine besonders schnelle und effiziente Software-Entwicklung ermöglicht. Da ist zum anderen der Einsatz von komplexen hoch-optimierten Basis-Funktionen, der es ermöglichen sollte, bei gleicher Flexibilität die Konkurrenz in puncto Performance zu überholen. Ich hoffe, der vorliegende Benchmark trägt ein wenig dazu bei.

Allerdings: Performance-Messungen bei Interpretern sind nicht nur generell ein schwieriges Unterfangen – sie spalten sich auch schnell auf in unzählige Einzel-Messungen, die eine „Gesamt-Bewertung“ am Schluß nahezu unmöglich machen.

Einen eindeutigen „Sieger“ oder „Verlierer“ wird es nicht geben. Letztendlich sind hier aber die Hersteller gefragt, selbst Messungen in grösserem Maße durchzuführen. Nur so ist es möglich, die zahllosen Verbesserungen zu implementieren, die in ihrer Summe letztendlich ein konkurrenzfähiges Produkt ausmachen und APL langfristig einen Platz auf dem Markt der Programmiersprachen sichern.

Aber wie schon gesagt: Es bleibt schwierig.

Anhang: Alle Meßergebnisse im Detail

PART1= Standard-NUM-Functions								
		APLX	IBM-APL2	APL+WIN 9.0	APL+WIN 13.1	Dyalog 13.0	Dyalog 14.0	Dyalog64 14.0
10000	X+Y	29	15	121	119	15	13	30
10000	X-Y	28	15	119	120	16	13	26
10000	X×Y	29	15	131	125	13	13	26
10000	X÷Y	90	83	87	86	295	81	98
10000	X*Y	1023	888	1273	1159	1038	1186	1556
10000	X!Y	153714	3752	3596	1947	9505	7400	7685
10000	X⌊Y	31	65	90	86	30	32	45
10000	X⌈Y	34	67	94	87	34	35	42
10000	-X	18	11	171	145	119	98	46
10000	0-X	67	13	29	53	14	13	26
10000	÷X	88	81	232	231	379	211	111
10000	1÷X	91	82	88	125	257	82	101
10000	×X	29	19	40	41	112	107	37
10000	|X	12	16	144	138	121	96	54
10000	!X	42250	545	1000	673	2541	2045	1811

PART2= Standard-Comparison on NUM-Vectors								
		APLX	IBM-APL2	APL+WIN 9.0	APL+WIN 13.1	Dyalog 13.0	Dyalog 14.0	Dyalog64 14.0
10000	X<Y	150	124	222	222	146	151	86
10000	X≤Y	153	135	223	226	149	143	82
10000	X=Y	184	165	201	207	185	181	107
10000	X≥Y	163	132	259	225	146	148	81
10000	X>Y	149	125	219	223	147	140	83
10000	X≠Y	195	164	202	207	185	182	109
10000	X≡Y	0	0	1	1	0	0	0

PART3= Accumulated Operations on NUM-Vectors (/)								
		APLX	IBM-APL2	APL+WIN 9.0	APL+WIN 13.1	Dyalog 13.0	Dyalog 14.0	Dyalog64 14.0
10000	+/X	30	12	43	42	35	37	11
10000	×/X	1189	18	36	61	17	19	22
10000	⌊/X	32	27	11	25	8	8	8
10000	⌈/X	32	26	10	25	8	8	7
10000	-/X	29	16	72	52	272	285	221
10000	÷/X	86	108	91	91	545	357	267
10000	X+2	27	13	38	47	14	12	26
10000	X×2	27	13	37	48	14	13	28
10000	X+X	28	13	120	116	13	12	28
10000	X*2	253	655	280	89	284	124	94
10000	X×X	28	14	128	125	12	12	26

PART4= Accumulated Operations on NUM-Vectors (\)								
		APLX	IBM-APL2	APL+WIN 9.0	APL+WIN 13.1	Dyalog 13.0	Dyalog 14.0	Dyalog64 14.0
10000	+\X	774	44	38	50	146	137	101
10000	×\X	953	64	37	47	161	154	108
10000	⌊\X	474	32	146	145	150	14	19
10000	⌈\X	432	32	140	139	146	12	21
10000	-\X	4337000	775	257500	253500	655000	1014000	982500
10000	ΔMIN_E	1139	175	583	592	309	271	265
10000	÷\X	64241000	538500	447000	447333	2153000	1591000	1217000
10000	ΔDIV_E	1291	217	645	626	443	341	230

PART5= Accumulated Comparison of NUM-Vectors (^)								
		APLX	IBM-APL2	APL+WIN 9.0	APL+WIN 13.1	Dyalog 13.0	Dyalog 14.0	Dyalog64 14.0
10000	X^.<Y	1	581	342	347	155	159	91
10000	X^.≤Y	0	579	344	342	158	153	96
10000	X^.=Y	0	578	336	339	194	190	119
10000	X^.≥Y	0	586	361	339	155	159	93
10000	X^.>Y	1	988	343	341	158	155	94
10000	X^.≠Y	352	576	347	349	197	189	120
10000	X≡Y	0	0	1	1	0	0	0
10000	^/X=Y	185	164	203	208	184	176	108

PART6= Accumulated Comparison of NUM-Vectors (∨)								
		APLX	IBM-APL2	APL+WIN 9.0	APL+WIN 13.1	Dyalog 13.0	Dyalog 14.0	Dyalog64 14.0
10000	X∨.<Y	1	597	338	342	158	160	92
10000	X∨.≤Y	0	601	342	341	158	153	95
10000	X∨.=Y	245	574	367	338	196	190	120
10000	X∨.≥Y	1	610	387	337	158	157	93
10000	X∨.>Y	1	608	358	340	156	150	94
10000	X∨.≠Y	0	575	395	341	195	192	120
10000	X≡Y	0	0	1	1	0	0	0
10000	∨/X=Y	186	166	219	217	184	178	110

PART7= Boolean Operations (/)								
		APLX	IBM-APL2	APL+WIN 9.0	APL+WIN 13.1	Dyalog 13.0	Dyalog 14.0	Dyalog64 14.0
10000	∧/B	0	0	1	1	0	0	0
10000	∨/B	0	0	1	1	0	0	0
10000	⍲/B	255	75	59	60	315	320	322
10000	⍱/B	257	72	60	60	312	312	275
10000	+/B	23	3	60	63	0	0	0
10000	~B	0	1	4	4	0	0	0

PART8= Boolean Operations (\)								
		APLX	IBM-APL2	APL+WIN 9.0	APL+WIN 13.1	Dyalog 13.0	Dyalog 14.0	Dyalog64 14.0
10000	∧\B	52	4	66	69	0	0	0
10000	∨\B	52	4	61	66	0	0	0
10000	⍲\B	1248000	4	61	66	982500	0	0
10000	⍱\B	1263000	4	77	68	975000	0	0
10000	+\B	78	588	61	59	247	6	5
10000	~B	0	1	4	4	0	0	0

PART9= Matrix Operations								
		APLX	IBM-APL2	APL+WIN 9.0	APL+WIN 13.1	Dyalog 13.0	Dyalog 14.0	Dyalog64 14.0
500	⌽M	2004	786	463	386	4073	534	435
500	⊖M	14676	984	3006	1998	1160	1229	918
500	⍉M	4716	680	1147	1120	394	397	372
500	M+M	1029	704	2997	2970	361	430	365
500	(M+.×M	998000	132625	874000	1014000	93545	92182	195000
500	(+/M)	735	313	906	1042	778	327	253
500	(+⌿M)	910	287	915	1257	518	196	158
500	(X∘.×Y	1015	365	4061	4057	242	257	551

PART10= Selection								
		APLX	IBM-APL2	APL+WIN 9.0	APL+WIN 13.1	Dyalog 13.0	Dyalog 14.0	Dyalog64 14.0
500	B/X	4	4	11	10	0	0	0
500	X[IND_	0	7	2	2	0	1	0
500	B/M	1562	1215	3286	3240	1533	408	267
500	M[;IND	2235	2939	318	370	288	284	238
500	B⌿M	319	485	540	573	160	163	162
500	M[IND_	2058	197	316	343	153	171	154
500	⍉B⌿⍉M	21255	1538	4875	3283	1852	1968	1594

PART11= Restructuring		APLX	IBM-APL2	APL+WIN 9.0	APL+WIN 13.1	Dyalog 13.0	Dyalog 14.0	Dyalog64 14.0
400	B⊂M	5876	6123	8317	8387	3046	3024	2574
400	⊂[2]M	3325	6183	266	507	241	266	240
400	⊃MX	2401	7129	306	464	227	240	222
400	⌽M	1302	357	234	182	2536	247	198
400	⍉M	8832	432	1436	871	604	623	492
400	⊖M	2326	282	712	712	203	215	195

PART12= Key String search (length 10)		APLX	IBM-APL2	APL+WIN 9.0	APL+WIN 13.1	Dyalog 13.0	Dyalog 14.0	Dyalog64 14.0
10000	∨/S⍷K	507	1313	427	542	81	82	74
10000	(S⍷K)[;1]	653	620	405	381	60	57	57
10000	K∧.=S	89	36	46	45	59	58	52
10000	KX⍳⊂S	107	8056	578	450	401	435	361
10000	KX≡¨⊂S	3080	5836	2126	1110	1388	1270	1211
10000	KX⍳KX	878	14910	1625	1160	2879	2991	2930

PART13= Key String search (length 20)		APLX	IBM-APL2	APL+WIN 9.0	APL+WIN 13.1	Dyalog 13.0	Dyalog 14.0	Dyalog64 14.0
10000	∨/S2⍷K2	738	1790	598	872	102	107	96
10000	(S2⍷K2)[;1]	907	653	408	369	63	59	66
10000	K2∧.=S2	100	36	51	46	60	59	52
10000	KX2⍳⊂S2	93	11341	570	463	406	435	356
10000	KX2≡¨⊂S2	3533	5940	2208	1166	1406	1305	1284
10000	KX2⍳KX2	911	19960	2307	1412	4207	4176	4158

PART14= Character search/String search		APLX	IBM-APL2	APL+WIN 9.0	APL+WIN 13.1	Dyalog 13.0	Dyalog 14.0	Dyalog64 14.0
10000	⎕AV⍳AV_FIRST	0	1	2	1	0	0	0
10000	⎕AV⍳AV_LAST	0	2	3	2	0	0	0
10000	(⊂⎕AV)⍳¨L	4588	9082	9000	6743	3230	3148	3074
10000	(⊂L)⍳¨⎕AV	1721	2539	11483	9505	1470	1480	1487
10000	'ρ'=L	35	22	32	32	18	19	11
10000	'ρ'∈L	26	7	331	183	85	26	22
10000	S∈L	4	12	182	193	5	5	4
10000	S2∈L	3	11	188	195	3	3	2

PART15= Polynoms		APLX	IBM-APL2	APL+WIN 9.0	APL+WIN 13.1	Dyalog 13.0	Dyalog 14.0	Dyalog64 14.0
10000	(0.2)⊥X	1498	491	223	134	192	200	382
10000	(Nρ0.2)⊥X	1787	512	266	162	210	204	313
10000	X⊥Y	1591	497	234	133	198	194	309
10000	X+Y	29	15	121	122	11	13	15
10000	X×Y	29	14	138	124	13	13	14

PART16= Loops		APLX	IBM-APL2	APL+WIN 9.0	APL+WIN 13.1	Dyalog 13.0	Dyalog 14.0	Dyalog64 14.0
10000	(FOR_LOOP N)	15609	14676	12808	5198	6616	6836	7085
10000	(LABEL_LOOP N)	53368	30242	32226	10967	15121	16915	19132
10000	(LABEL_LOOP2 N)	15594	14676	17483	5147	6609	6705	7141
10000	(LABEL_LOOP3 N)	58706	27405	12183	5171	16361	18107	19569

PART17= NAND-BACKSLASH/NOR-BACKSLASH		APLX	IBM-APL2	APL+WIN 9.0	APL+WIN 13.1	Dyalog 13.0	Dyalog 14.0	Dyalog64 14.0
10000	(⍲\B)	1264000	4	62	76	975000	0	0
10000	(⍙NAND_BS B)	16	15	42	32	37	82	12
10000	(⍱\B)	1248000	4	64	70	975000	0	0
10000	(⍙NOR_BS B)	13	13	38	31	107	11	71

PART18= Alphanumeric Sort		APLX	IBM-APL2	APL+WIN 9.0	APL+WIN 13.1	Dyalog 13.0	Dyalog 14.0	Dyalog64 14.0
10000	(⎕AV⍋K)	5071	28167	2811	2742	3398	3481	2893
10000	(⎕AV⍒K)	5066	27811	3158	3083	3403	3477	2970
10000	(⎕AV⍋K2)	5071	27750	2619	2722	4803	4990	4107
10000	(⎕AV⍒K2)	5097	28514	3272	3099	4780	4965	4775
10000	(⎕AV⍋L)	5225	23762	2171	1899	174	183	160
10000	(⎕AV⍒L)	4261	23762	2520	2230	177	185	164

PART19= Numeric Sort (Vector)		APLX	IBM-APL2	APL+WIN 9.0	APL+WIN 13.1	Dyalog 13.0	Dyalog 14.0	Dyalog64 14.0
10000	(⍋X)	3828	561	2020	2041	1174	1190	1115
10000	(⍒X)	3883	587	2407	2411	1177	1185	1146
10000	(X[⍋X])	3853	1026	2060	2101	1184	1218	1129
10000	(X[⍒X])	3898	1015	2449	2458	1213	1207	1201

PART20= Numeric Sort (Matrix)		APLX	IBM-APL2	APL+WIN 9.0	APL+WIN 13.1	Dyalog 13.0	Dyalog 14.0	Dyalog64 14.0
500	(⍋M)	506	3212	183	216	265	246	267
500	(⍒M)	510	3148	211	225	248	248	257
500	(M[⍋M;])	5253	3700	796	947	614	629	636
500	(M[⍒M;])	5123	3518	829	937	600	628	597

PART21= Boolean Matrix-Operations (/ and ⌿)		APLX	IBM-APL2	APL+WIN 9.0	APL+WIN 13.1	Dyalog 13.0	Dyalog 14.0	Dyalog64 14.0
500	∧/MB	4	409	6	10	44	50	50
500	∨/MB	42	1284	28	11	77	81	56
500	⍲/MB	5982	1316	1501	1511	1247	1296	1000
500	⍱/MB	5808	1291	1496	1500	1235	1268	1001
500	∧⌿MB	5	7448	5	7	15	16	3
500	∨⌿MB	7	8248	3	21	15	16	5
500	⍲⌿MB	6000	8466	1512	1521	845	875	722
500	⍱⌿MB	5871	8387	1514	1514	853	869	712

PART22= Boolean Matrix-Operations (\ and ⍀)		APLX	IBM-APL2	APL+WIN 9.0	APL+WIN 13.1	Dyalog 13.0	Dyalog 14.0	Dyalog64 14.0
500	∧\MB	1303	378	1566	1694	37	36	37
500	∨\MB	1308	326	1490	1668	46	46	43
500	⍲\MB	1450000	262	1482	1706	1216000	2028000	1966000
500	⍱\MB	1420000	216	1505	1683	1202000	1934000	1966000
500	∧⍀MB	6085	5489	1584	1757	119	121	84
500	∨⍀MB	6048	3659	1597	1892	107	107	81
500	⍲⍀MB	1436000	3453	1514	1727	1280000	2371000	1934000
500	⍱⍀MB	1389000	3527	1574	1721	1217000	2028000	1950000

PART23= Matrix-Vector-Transformation		APLX	IBM-APL2	APL+WIN 9.0	APL+WIN 13.1	Dyalog 13.0	Dyalog 14.0	Dyalog64 14.0
500	(∈M)	576	647	1789	2791	339	339	215
500	(,M)	541	643	449	434	328	398	216
500	(∈MC)	33	31	1654	2337	29	29	36
500	(,MC)	34	33	48	48	30	28	32
500	(∈MB)	2	4	4141	4625	3	2	3
500	(,MB)	2	4	7	6	3	3	3

PART24= Brackets		APLX	IBM-APL2	APL+WIN 9.0	APL+WIN 13.1	Dyalog 13.0	Dyalog 14.0	Dyalog64 14.0
10000	X≡Y	0	0	1	1	0	0	0
10000	(X≡Y)	0	1	1	1	0	0	0
10000	((X≡Y))	0	1	2	1	0	0	0
10000	(((X≡Y)))	0	1	2	1	0	1	0
10000	((((X≡Y))))	0	1	2	2	0	1	1
10000	(((((X≡Y)))))	0	1	2	2	0	1	1

PART25= Trigonometric Functions		APLX	IBM-APL2	APL+WIN 9.0	APL+WIN 13.1	Dyalog 13.0	Dyalog 14.0	Dyalog64 14.0
10000	3.1415×X	32	14	152	133	14	12	16
10000	PI×X	30	12	127	124	12	10	12
10000	(○X)	48	12	158	157	134	107	33
10000	(1○X)	753	1014	1071	921	882	899	627
10000	(2○X)	712	1055	1103	940	877	899	628
10000	(3○X)	793	826	1018	964	964	974	712
10000	(4○X)	1028	407	700	1243	856	885	612
10000	(5○X)	1143	1148	1833	1897	1404	1401	741
10000	(6○X)	1168	1163	1896	1923	1476	1416	680
10000	(7○X)	1194	1110	1972	2079	1530	1507	643

PART26= Character-Set-Check		APLX	IBM-APL2	APL+WIN 9.0	APL+WIN 13.1	Dyalog 13.0	Dyalog 14.0	Dyalog64 14.0
10000	L∈S	42	64	68	70	196	156	165
10000	L∈S2	52	80	82	87	226	181	185
10000	L∈L	34	54	58	65	236	215	234
10000	L⍳S	25	10	12	9	78	84	86
10000	L⍳S2	24	10	12	9	104	81	85
10000	L⍳L	38	35	37	35	212	236	232

PART27= Logarithmic functions		APLX	IBM-APL2	APL+WIN 9.0	APL+WIN 13.1	Dyalog 13.0	Dyalog 14.0	Dyalog64 14.0
10000	X*Y	1013	893	1284	1186	1015	1322	1356
10000	2*Y	1011	880	1375	1389	1037	1168	1311
10000	X*1	241	381	37	54	283	114	65
10000	1*Y	883	743	29	61	1040	1065	611
10000	⍟Y	629	264	760	470	733	613	478
10000	2⍟Y	1014	684	648	703	1923	1013	861
10000	10⍟Y	1027	313	698	754	1910	1007	899

PART28= Rotate		APLX	IBM-APL2	APL+WIN 9.0	APL+WIN 13.1	Dyalog 13.0	Dyalog 14.0	Dyalog64 14.0
10000	⌽L	51	8	8	8	138	6	6
10000	1⌽L	54	2	42	43	2	2	2
10000	2⌽L	54	2	42	42	2	2	2
10000	4⌽L	54	2	42	43	2	2	2
10000	8⌽L	53	2	41	43	2	2	2
10000	16⌽L	54	2	42	43	2	2	2
10000	32⌽L	53	2	44	43	2	2	2
10000	64⌽L	77	2	43	50	2	2	2
10000	128⌽L	56	2	42	50	2	2	2
10000	256⌽L	57	2	43	52	2	2	2
10000	512⌽L	55	2	41	56	3	3	3

PART29= Standard-Integer-Functions		APLX	IBM-APL2	APL+WIN 9.0	APL+WIN 13.1	Dyalog 13.0	Dyalog 14.0	Dyalog64 14.0
10000	XI+YI	14	18	21	23	7	7	7
10000	XI-YI	11	15	20	25	8	7	7
10000	XI×YI	12	64	20	22	7	7	7
10000	XI÷YI	87	93	82	90	312	99	98
10000	XI!YI	64375	1442	2365	2170	1604	1570	1444
10000	XI⌊YI	21	44	67	76	17	17	9
10000	XI⌈YI	23	43	72	75	17	16	9
10000	-XI	11	14	38	45	44	37	36
10000	0-XI	9	14	36	76	9	9	9
10000	÷XI	85	84	240	255	333	81	113
10000	1÷XI	87	86	83	131	270	91	96
10000	×XI	11	12	43	48	50	36	36
10000	|XI	9	23	38	44	46	37	33
10000	!XI	39920	4176	1994	2749	1384	1332	1202

PART30= Mixed-Type-Standard-Functions		APLX	IBM-APL2	APL+WIN 9.0	APL+WIN 13.1	Dyalog 13.0	Dyalog 14.0	Dyalog64 14.0
10000	X+YI	66	19	37	53	22	23	23
10000	X-YI	66	20	11	16	15	16	15
10000	XI-Y	65	20	11	15	23	23	23
10000	X×YI	67	20	36	58	29	30	29
10000	X÷YI	89	87	83	95	295	89	86
10000	XI÷Y	89	90	86	101	301	92	90
10000	X!YI	199800	2727	10731	10628	8667	7448	7448
10000	X⌊YI	69	25	106	21	25	24	23
10000	X⌈YI	71	25	102	20	26	25	23

PART31= Constructors		APLX	IBM-APL2	APL+WIN 9.0	APL+WIN 13.1	Dyalog 13.0	Dyalog 14.0	Dyalog64 14.0
10000	⍳N	5	0	6	6	6	7	4
10000	N⍴X	59	10	43	47	9	8	9
10000	N⍴L	7	2	2	2	0	0	0
10000	N⍴B	2	1	6	6	0	0	0
10000	X~Y	296250	99900	4401	4024	417	427	608
10000	L~S	38	45	289	314	133	141	141
10000	L~S2	39	58	324	362	150	161	160

PART32= Matrix-Structures		APLX	IBM-APL2	APL+WIN 9.0	APL+WIN 13.1	Dyalog 13.0	Dyalog 14.0	Dyalog64 14.0
500	M,M	1222		2179	2353	626	573	656
500	M;M	1292		2205	2354	540	638	425

Anmerkung

1 Vgl. auch Herminghaus, Ralf: APL-Benchmark: Wie schnell ist APL wirklich? In: APL-Journal 2/2011, S. 11-24

Kontakt:

Ralf Herminghaus
AXA Lebensversicherung AG
Colonia-Allee 10-20, 51067 Köln,
Postanschrift: 51172 Köln
E-Mail: Ralf.Herminghaus@AXA.de

Kai Jaeger

Legacy-Code, Überlebensstrategien und Fire

Dieser Artikel erklärt, warum Fire für Dyalog APL überhaupt entstand, und warum ich insgesamt circa fünf – unbezahlte – Monate Arbeit in dieses Projekt investiert habe. Der Name „Fire“ verweist auf die zwei Hauptmerkmale, Suchen und Ersetzen: FInd and REplace. Fire ist besonders nützlich für Programmierer, die mit Legacy-Code zu tun haben: Code, der in der Regel ziemlich alt ist, mit wenig oder veralteter Dokumentation (oder auch gar keiner), ohne Testfälle, ohne jede Struktur und / oder erkennbares Design und ohne wirkliche Modularisierung; die ursprünglichen Autoren sind womöglich nicht mehr verfügbar.

Ein Legacy-System am Laufen zu halten ist immer eine Herausforderung, insbesondere deshalb, weil nur ein komplettes Resdesign vernünftig zu sein scheint, dies aber oft keine Option ist, sei es, weil der Kunde oder das Management dies nicht akzeptieren kann oder will, oder die Anwendung zu komplex ist, mit zu vielen Schnittstellen zu anderen Systemen.

Der Vorschlag eines Redesigns kann auch zu einer Gefahr für APL werden: die bestehenden Probleme werden gerne APL in die Schuhe geschoben, obwohl sie eindeutig kein Sprachmerkmal von APL sind, sondern vielmehr durch schlechte Entscheidungen von ganz normalen Menschen verursacht wurden.

In den meisten Fällen ist der Kunde zudem Teil des Problems und nicht der Lösung: fragt man nach der Bereitschaft, für eine Verbesserung der Situation zu zahlen, dann wird die Antwort oft lauten: „Wieviele neue Features bekommen wir? Keine?! Vergessen Sie es!“

Um in einer solchen Situation auf lange Sicht zu überleben gibt es nur eine Möglichkeit: Testfälle hinzufügen und Design, Modularisierung und Dokumentation Schritt für Schritt zu verbessern, wann immer man den Code anfasst. Zunächst wird dies wie eine Verschwendung von Ressourcen aussehen ohne das man viel erreicht, aber auf lange Sicht wird es die Situation verbessern und zu besserem und stabilerem Code führen.

Insbesondere das Hinzufügen von Testfällen hilft, weil man mehr und mehr Sicherheit gewinnt, daß nach einer Änderung nicht gleich alles auseinanderfällt, ein typisches Problem mit Legacy Code.

Aus diesem Grund liegt es auch im Interesse des Kunden / der Geschäftsführung, Zeit in die Verbesserung der Code-Basis

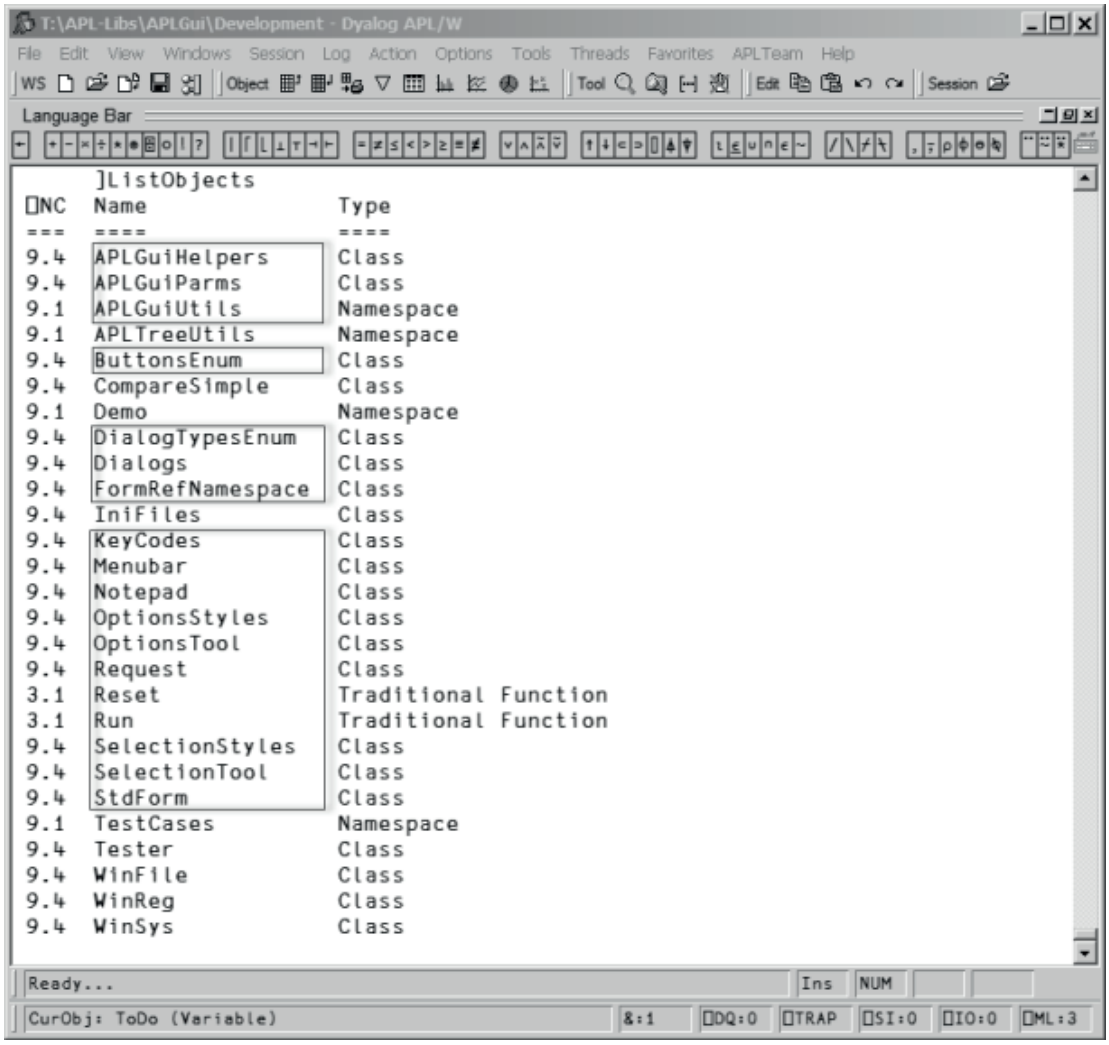
zu investieren, auch wenn der Kunde das nicht immer verstehen kann oder will. Deshalb kann es notwendig sein, seinen Mund zu halten und sich hinter einem neuen Feature oder einem Bug-fix zu verstecken.

Massnahmen zur Verbesserung der Code-Basis erfordern oft Veränderungen im großen Stil wie das Umbenennen vieler Objekte usw., etwas, das man lieber automatisch durchführt, zumindest bis zu einem gewissen Grad. Fire wurde entwickelt, um einen Programmierer in dieser Hinsicht zu unterstützen. Im Folgenden zeige ich, wie zwei typische Problemfälle mit Fire gelöst werden können.

Fallstudie I

Stellen Sie sich einen Workspace vor, in dem Sie mit der Entwicklung einer Reihe von Klassen begonnen haben, um ein bestimmtes Problem zu lösen, hier die Erstellung von GUI Utilities. Alle Klassen liegen in # (root) zusammen mit ein paar allgemeinen Klassen für die Lösung genereller Probleme, die in den GUI-Klassen benutzt werden.

Dann gibt es noch einen Namespace „Demo“, der eine ganze Reihe von Funktionen enthält, die bestimmte Aspekte der GUI-bezogenen Klassen demonstrieren. Dies illustriert, wie die Namespaces in # aussehen:



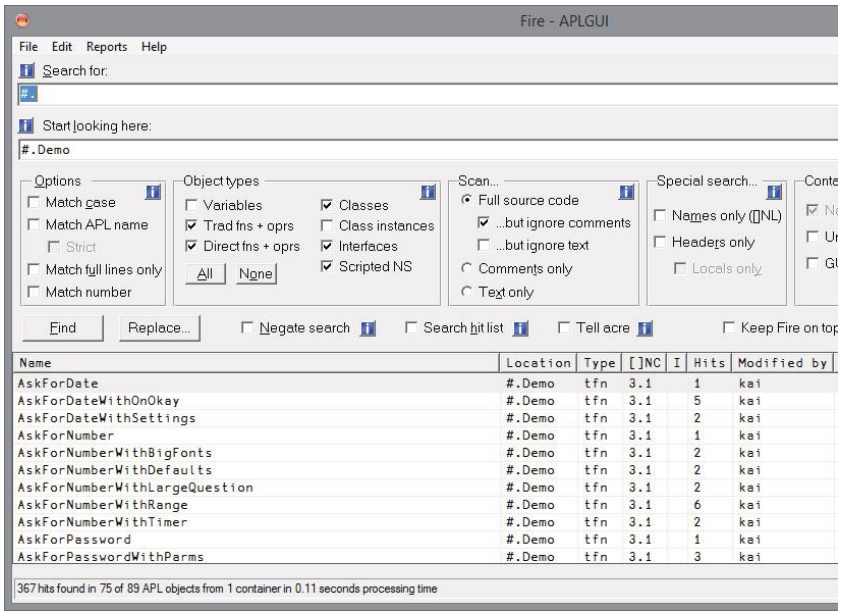
In Zukunft möchte ich Phil Lasts exzellentes Code-Management-System acre [1] verwenden. Dafür muß der Code umstrukturiert werden: alle GUI-bezogenen Klassen und Namespaceskripte sollen in einen Namespace `#.GUI` verschoben werden. Dies sind die gekennzeichneten Objekte.

Das ist relativ einfach zu erreichen, indem man die Scripte per Programm verschiebt; leider ist der WS-Explorer immer noch nicht in der Lage, dies zu tun. Danach wird # wie folgt aussehen:

```
      ]Listobjects -n=9
 NC Name              Typ
 == ====              ====
 9.1 APLTreeUtils  Namespace
 9.4 CompareSimple Class
 9.1 GUI      Namespace
 9.1 Demo      Namespace
 9.4 IniFiles      Class
 9.1 Testcases     Namespace
 9.4 Tester        Class
 9.4 WinFile       Class
 9.4 WinReg        Class
 9.4 WinSys        Class
```

Aber das ist nicht genug: die Funktionen im Namespace `#.Demo` sowie alle Testcases werden weiterhin versuchen, die GUI-Utilities in # zu finden. All diese Aufrufe müssen entsprechend angepasst werden; eine typische Aufgabe für Fire.

Zunächst gilt es herauszufinden, wie viele Funktionen und wie viele Zeilen Code geändert werden müssen. Dafür geben wir `#.` in das Feld „Suche nach“ ein und `#.Demo` in das „Start der Suche hier“ Feld wie hier gezeigt:

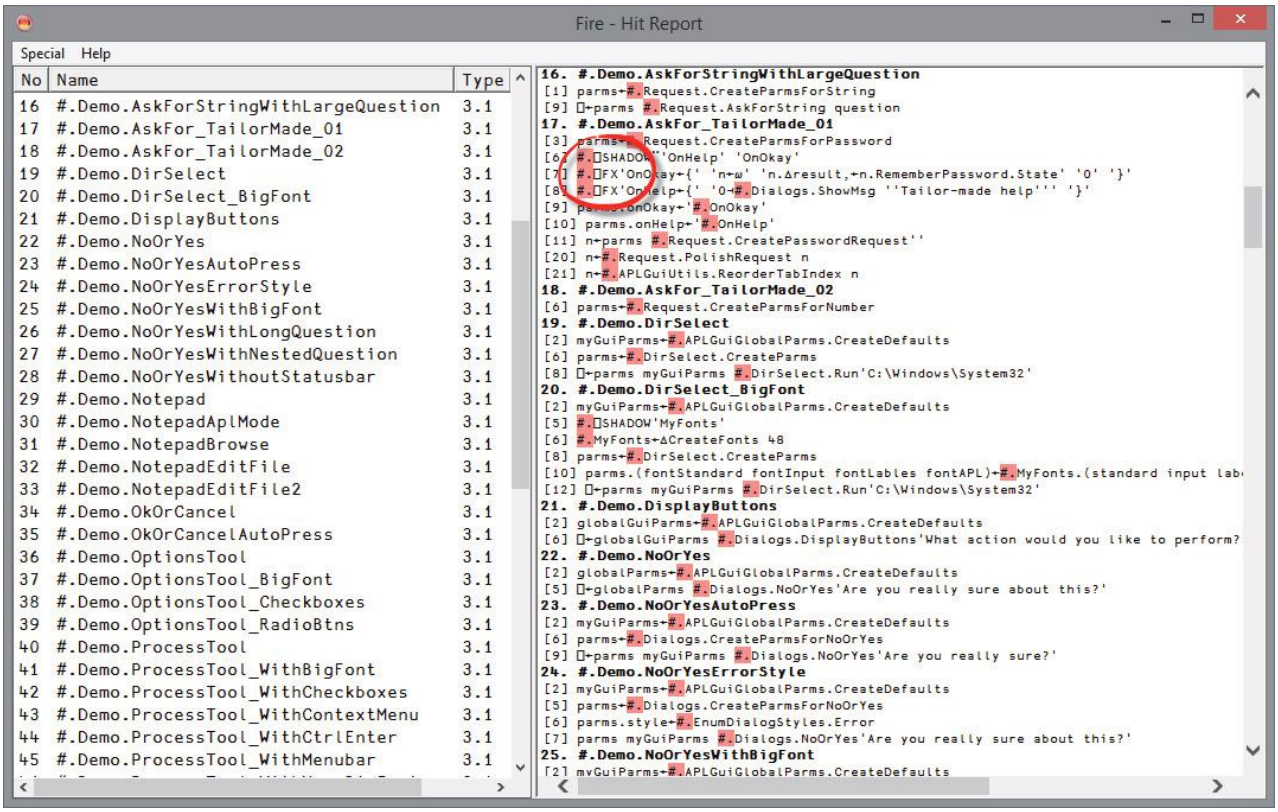


Das ergibt 367 Treffer in 75 Funktionen (siehe die Statusleiste), die möglicherweise geändert werden müssen.

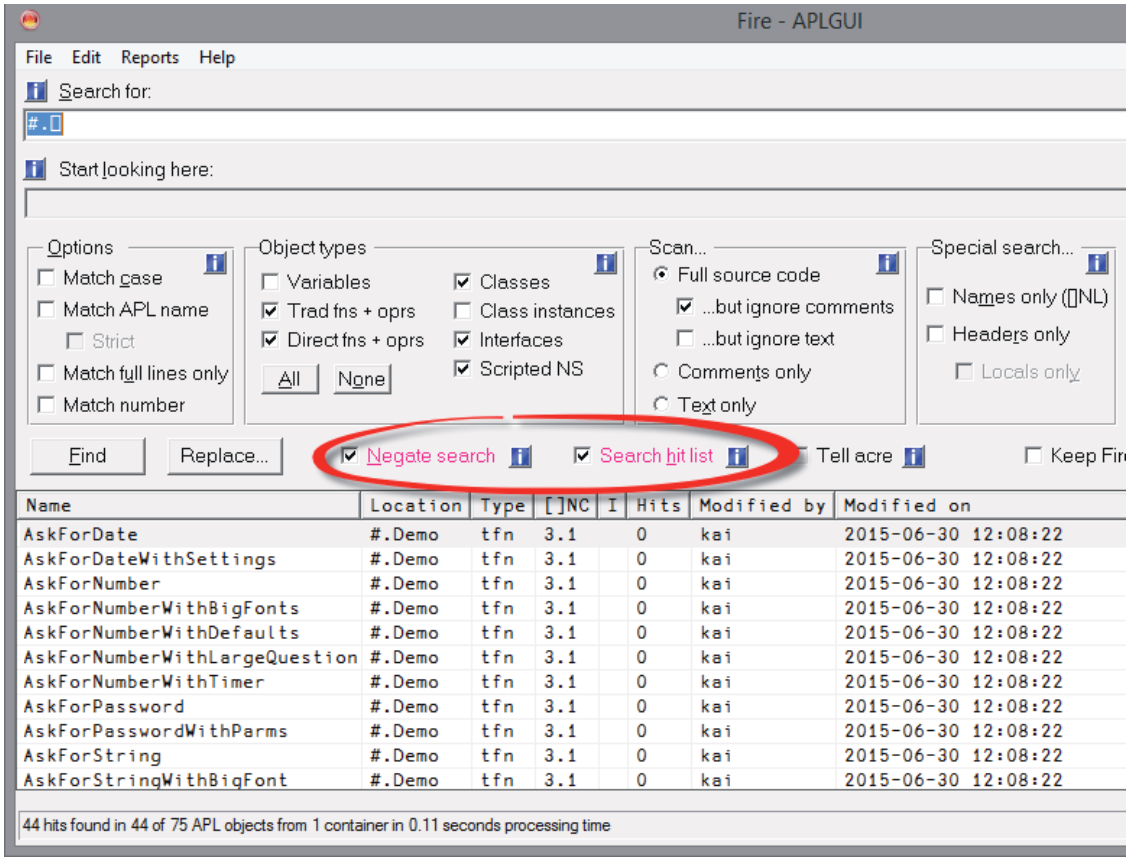
Im nächsten Schritt prüfen wir, ob dies wirklich die gewünschten Treffer sind: dafür wählen wir „Report hits“ aus dem „Reports“ Menü aus. Dieser Report (siehe nachfolgende Abbildung) gibt uns einen Überblick über die Treffer.

Die meisten angezeigten Treffer sind in Ordnung, aber es gibt auch eine Reihe von Funktionen, die Aufrufe von Systemfunktionen wie `#.SHADOW` oder `#.FX` oder `#.NEW` enthalten – diese Statements müssen unverändert bleiben. Doch gleichzeitig enthalten diese Funktionen auch Statements, die geändert werden müssen.

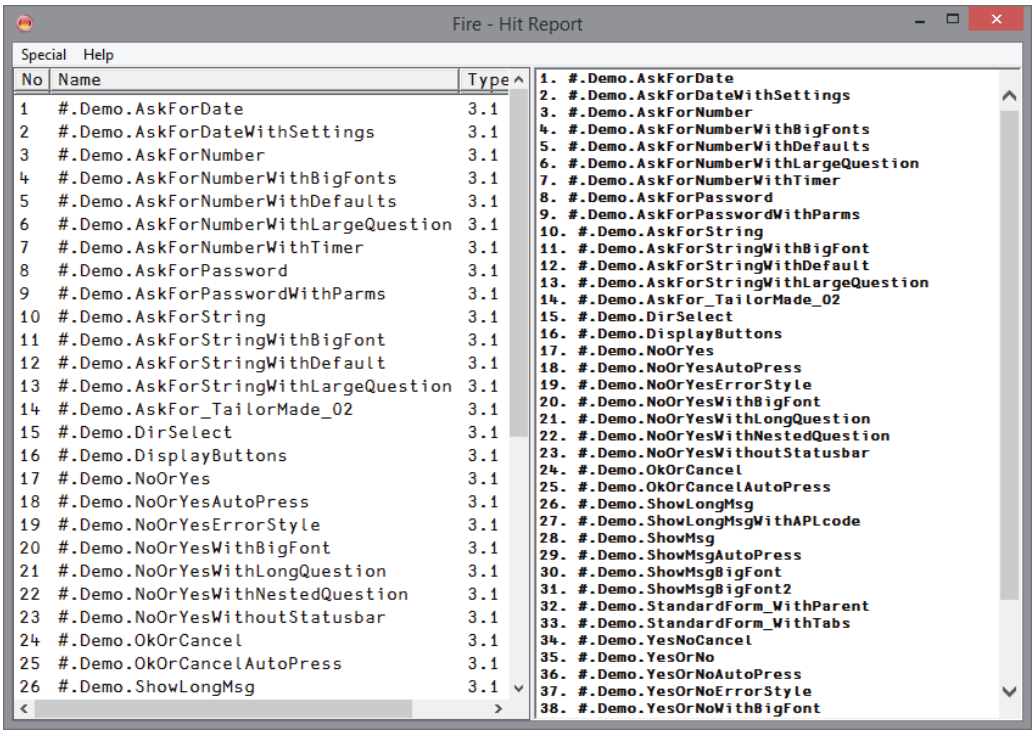
Was ist der beste Weg, um mit dieser Situation umgehen? Wir entfernen einfach alle



Funktionen aus der Liste, die den String "`#.`" enthalten; um diese Funktionen kümmern wir uns später. Dies können wir erreichen, indem wir nach "`#.`" suchen und die Optionen „Negate search" und „Search hit list" aktivieren.

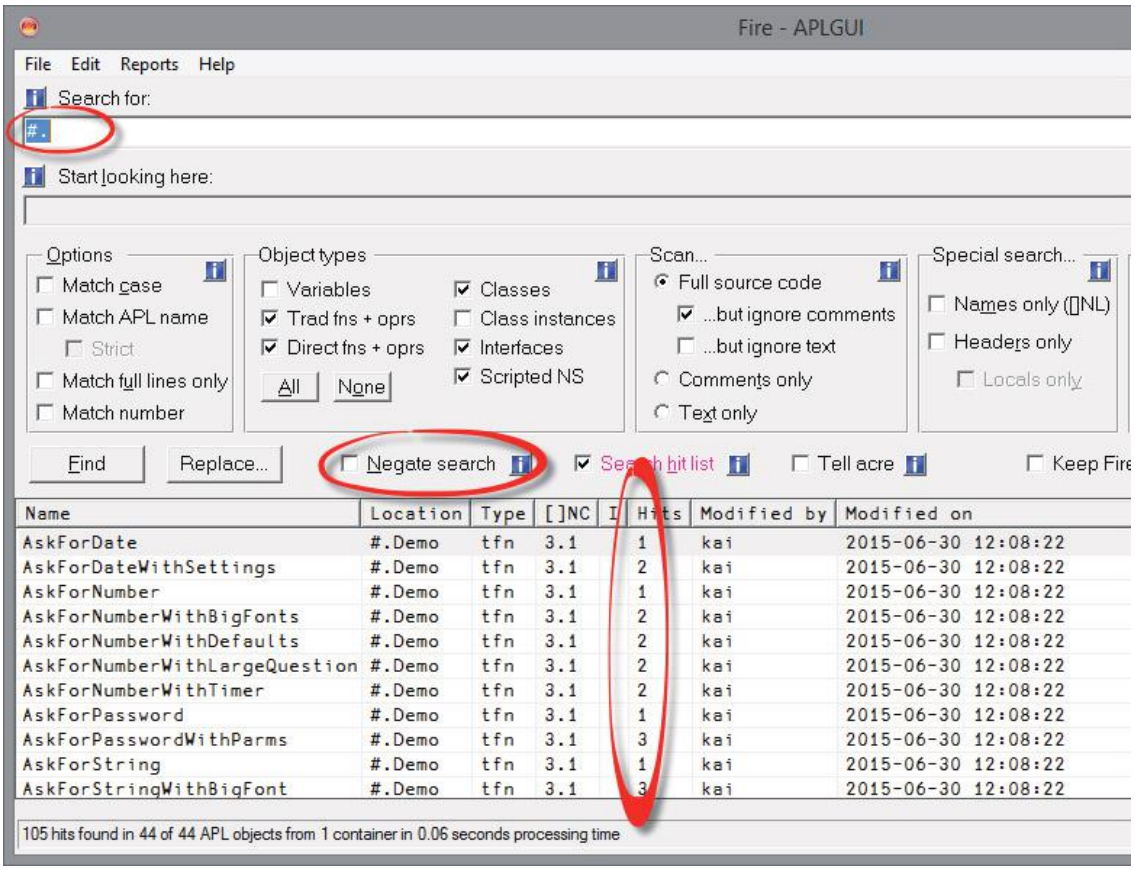


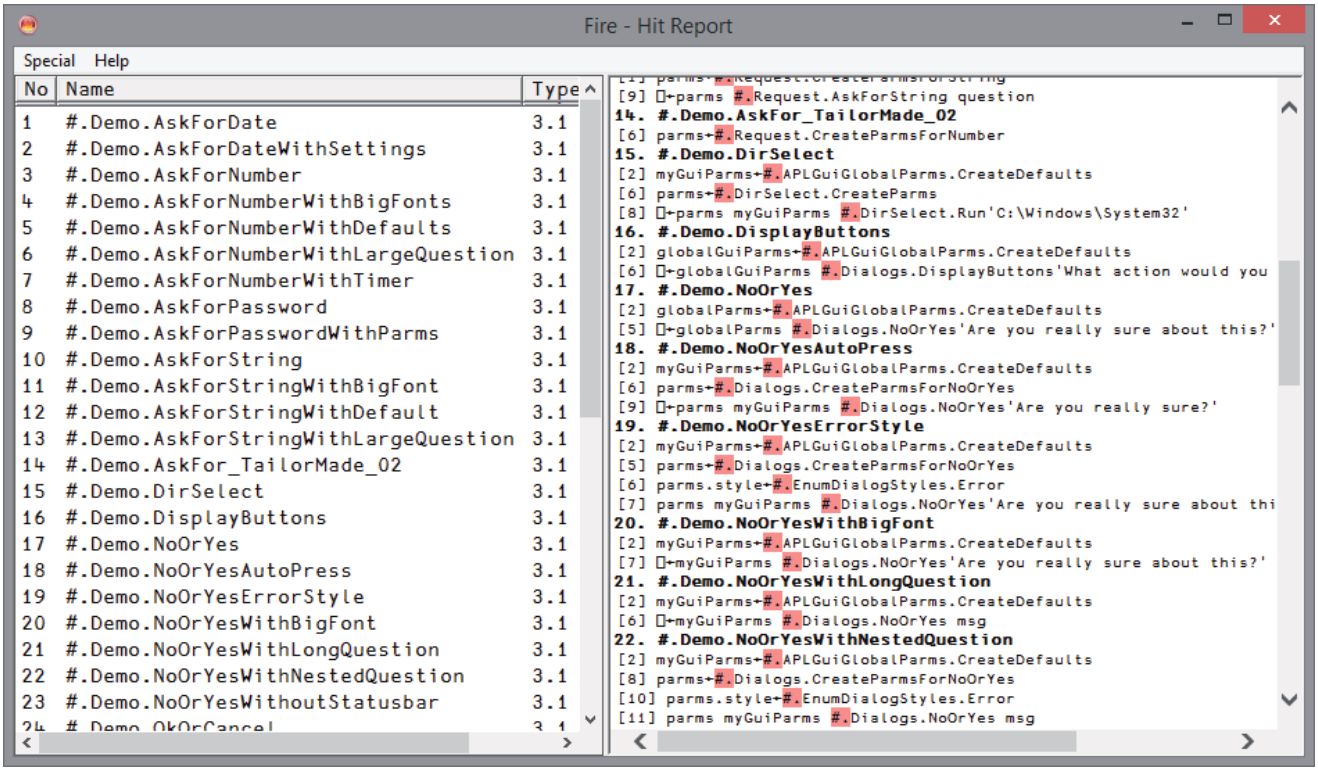
Diese Suche resultiert in einer Liste mit 44 Funktionen; dies ist der Bericht „Report hits“:



Uups. Das ist nicht, was wir erwartet haben. Der Grund für dieses Ergebnis ist die negierte Suche: die führt naturgemäß dazu, daß wir keine Treffer erhalten. Folglich kann der Report auch keine Treffer anzeigen.

Wir können die ursprüngliche Suche einfach erneut ausführen indem wir a) nach „`#.`“ suchen und b) die Option „Negate search“ ausschalten wie hier gezeigt:

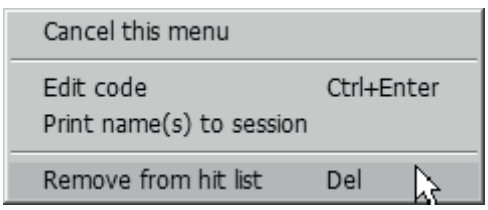




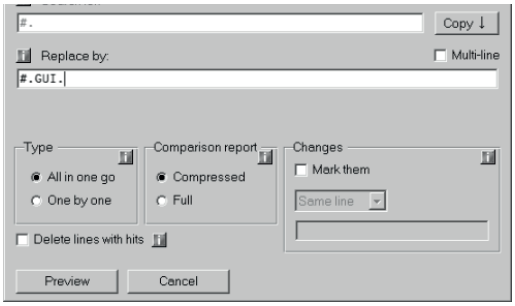
Nun zeigt der Report die erwarteten Treffer (siehe Abbildung oben).

Dies ist ein großer Schritt nach vorn: diese Liste scheint ausschließlich Funktionen zu enthalten, in denen alle Referenzen von `#.` geändert werden müssen bzw. können.

Sollte es in der Liste immer noch die eine oder andere Funktion geben, die wir nicht ändern wollen: das Kontextmenü im Hit Report offeriert die Möglichkeit, einzelne Funktionen gezielt aus der Liste zu entfernen:



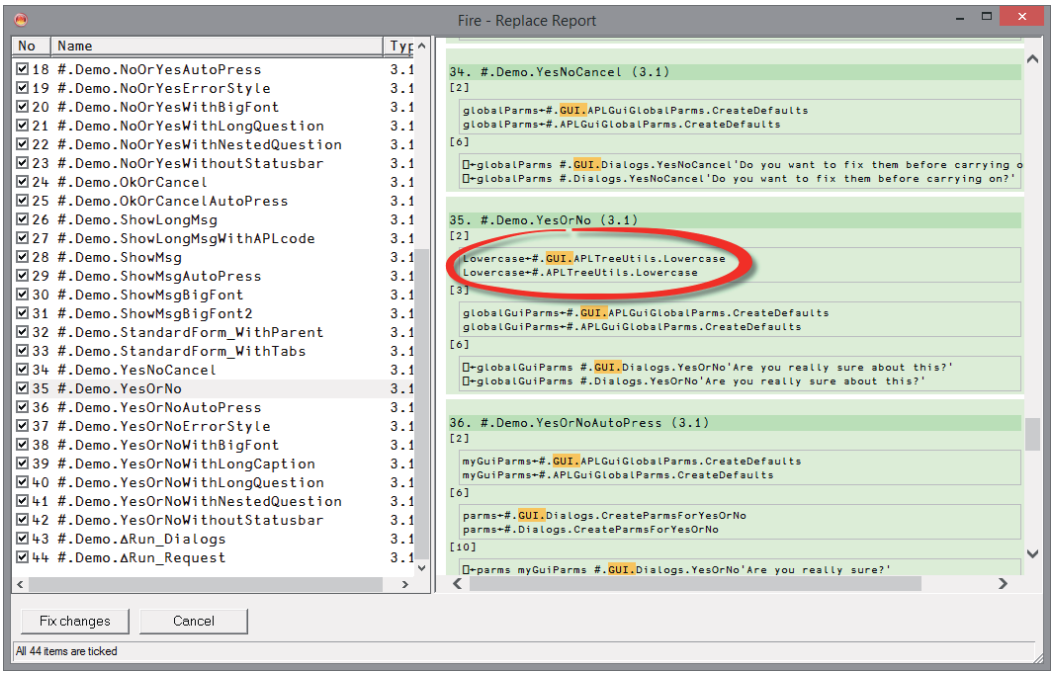
Es ist nun Zeit, den „Replace“ Button auszulösen. Dies zeigt den „Replace“ Dialog an:



Wenn wir `#.GUI` in das Feld „Replace by“ eingeben und dann auf „Preview“ klicken erhalten wir die Darstellung der nachfolgenden Abbildung (siehe nächste Seite).

Alles ist in Ordnung außer Nummer 19: die Funktion `#.Demo.YesOrNo` enthält neben zwei Zeilen, die in der Tat geändert werden müssen (3 und 6) auch eine Zeile 2, die wir nicht ändern wollen.

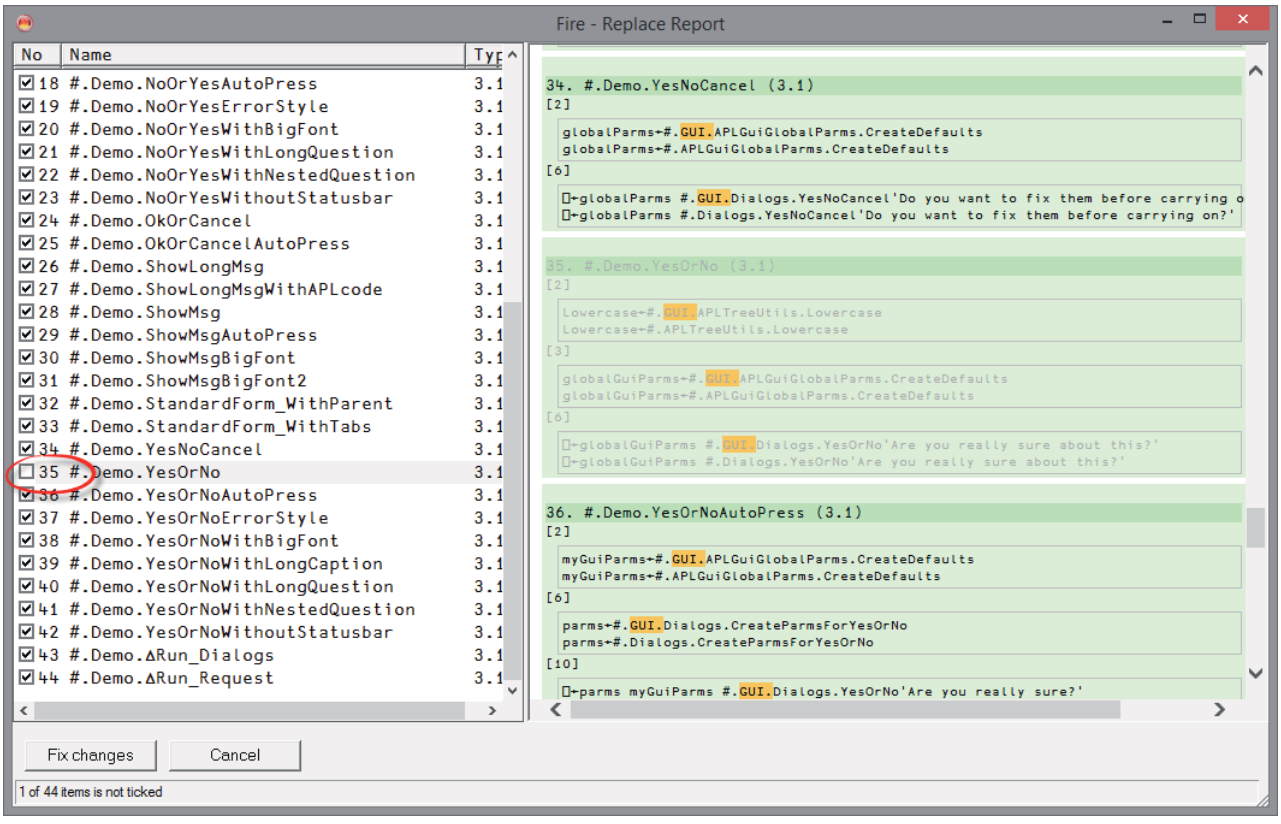
Es gibt verschiedene Strategien, mit diesem Problem umzugehen: Ein Weg ist, die Funktion von Fire abändern zu lassen und später die Zeile 2 im Editor zu berichtigen. Hier wählen wir einen anderen Ansatz, der



insbesondere dann besser geeignet ist, wenn wir eine ganze Reihe von Änderungen ausschliessen wollen: wir deaktivieren einfach das Kontrollkästchen Nummer 10 in der Baumansicht:

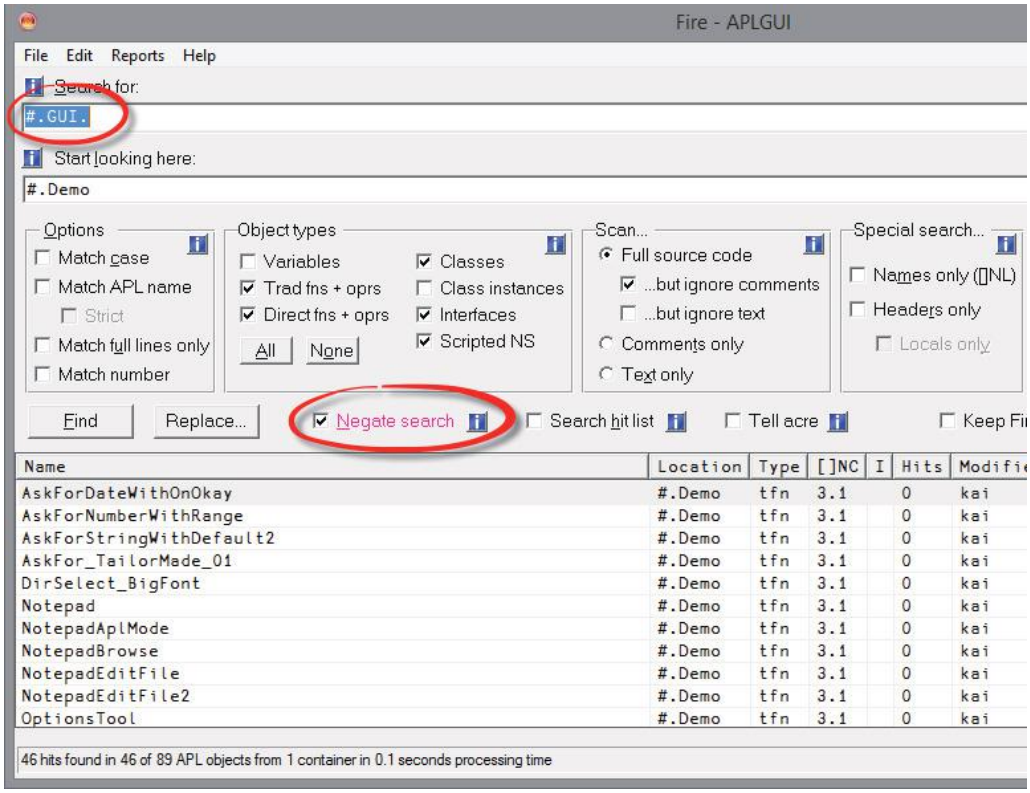
`#.Demo.YesOrNo` ist nun grau um zu verdeutlichen, daß die Funktion nicht mehr verarbeitet werden wird.

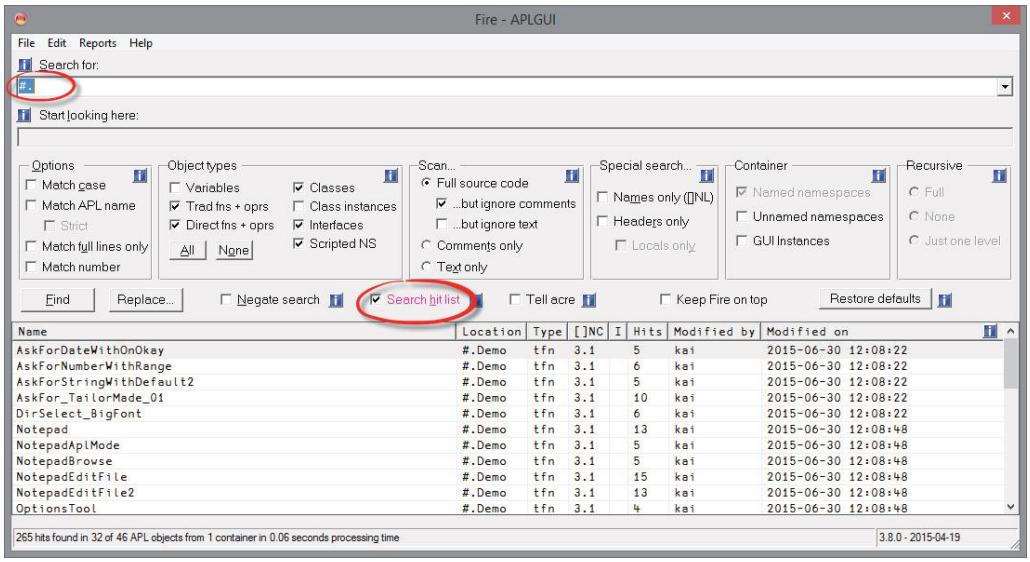
Schließlich klicken wir auf „Fix changes“. Damit ist der erste Teil der Aufgabe – und



damit der Großteil der notwendigen Änderungen – erledigt.

Um die verbleibenden Probleme anzugehen, erzeugen wir zuerst eine Liste der Objekte, die "`#.GUI`" **nicht** enthalten; damit schließen wir all jene aus, die wir gerade geändert haben. Um dies zu erreichen, setzen wir ein Häkchen in „Negate Search" und suchen dann nach „`#.GUI.`":



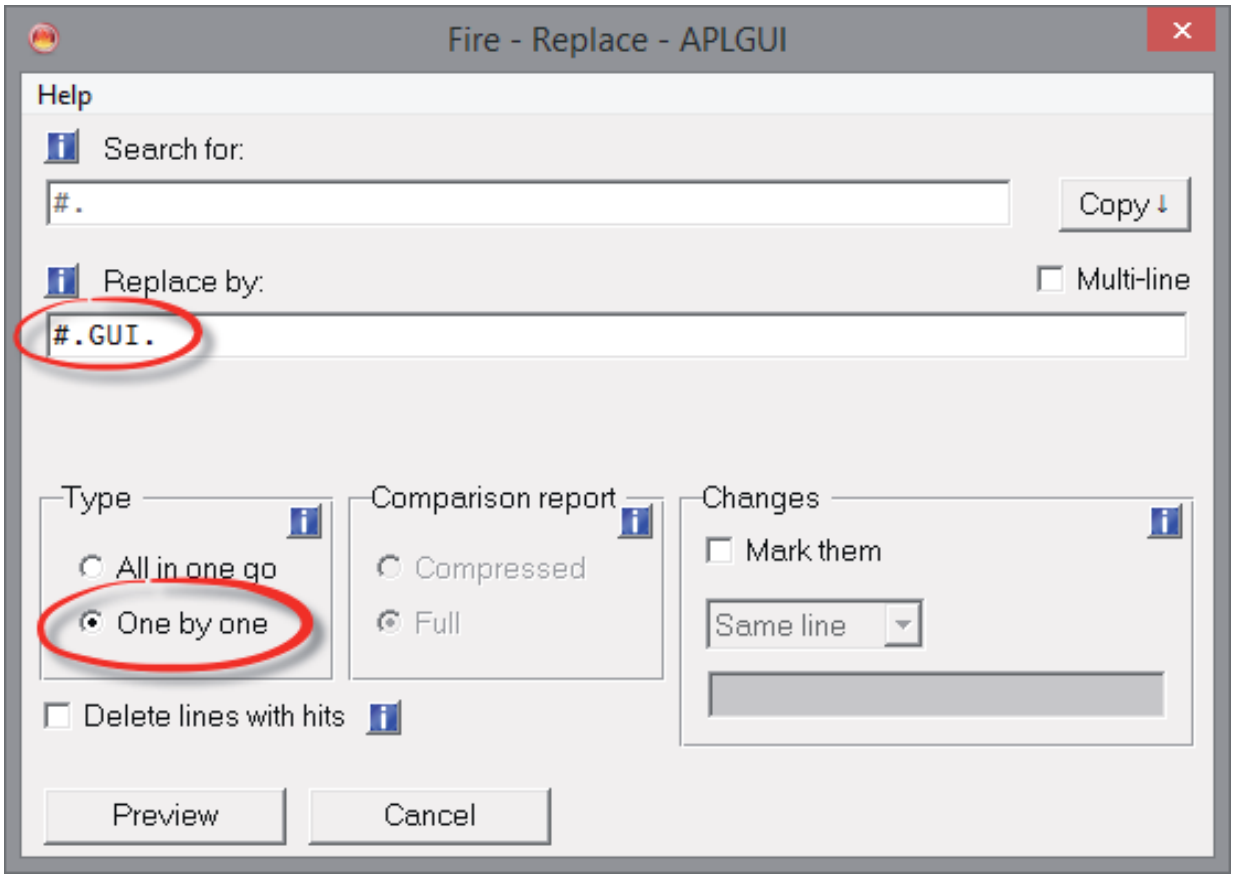


46 Objekte enthalten nicht den String `#.GUI`. Dies sind die potentiellen Kandidaten. Um nur diese zu durchsuchen, entfernen wir den Haken in „Negate search" und setzen einen in „Search hit list" (siehe Abbildung unten).

Wenn wir nun nach `#.` suchen, dann reduziert sich die Anzahl der Treffer auf 32.

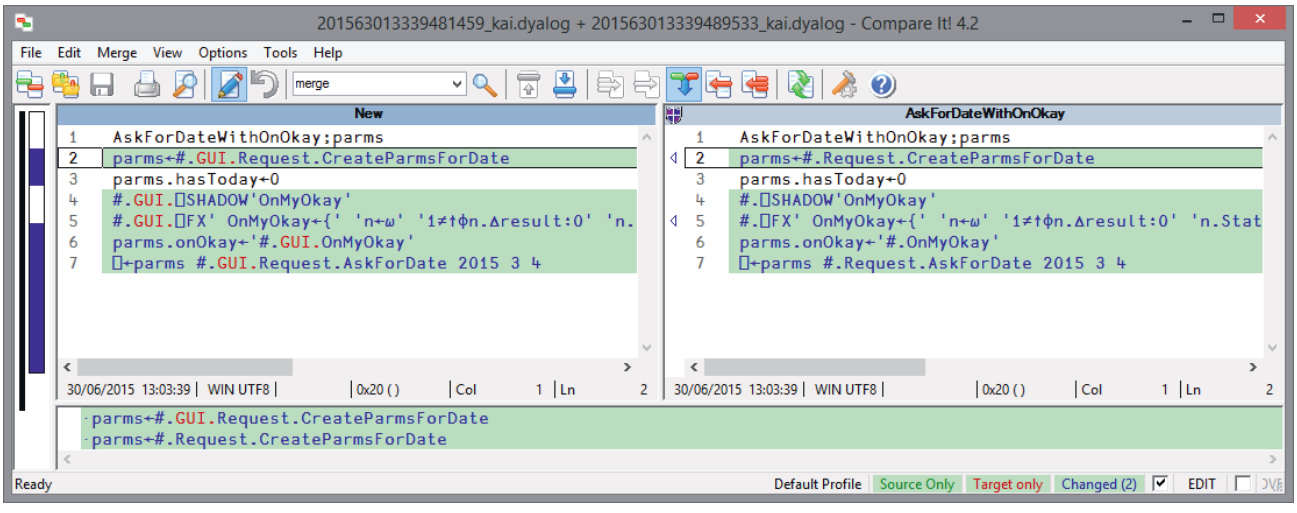
Wir wissen bereits, daß alle verbleibenden Objekte geändert werden müssen, aber wir wissen ebenfalls, daß sie auch Verweise auf `#.` enthalten, die unverändert bleiben müssen. Dieses Problem kann nicht automatisiert mit Fire gelöst werden. Allerdings kann Fire auch in dieser Situation noch eine große Hilfe sein.

Nach einem Klick auf die Schaltfläche „Replace" verändern wir die Standardeinstellungen im Dialogfeld „Replace":

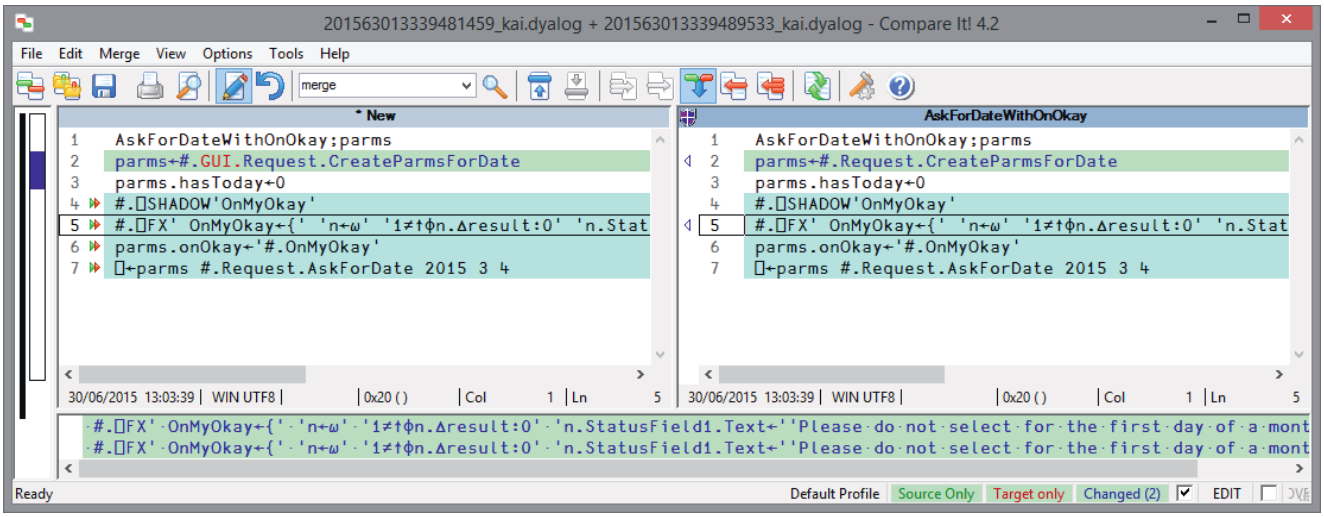


Wir unterstellen hier, daß Sie das ausgezeichnete 3rd-Party-Tool CompareIt! auf Ihrem Rechner installiert haben. Falls das nicht der Fall ist: Fire hat ein vergleichsweise beschränktes aber funktionierendes Vergleichstool eingebaut. CompareIt! ermöglicht dem Benutzer jede einzelne Änderung in einem Objekt zu bestätigen oder zu verweigern:

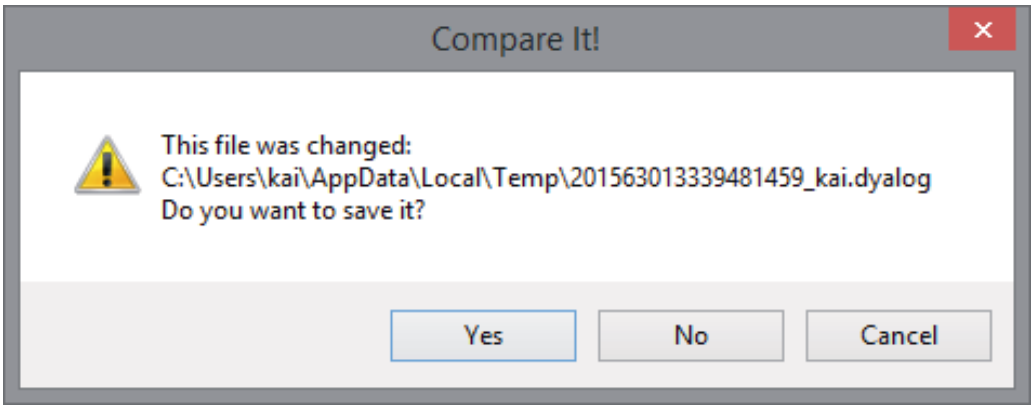
Die markierten Bereiche können aus dem rechten Fenster in das linke kopiert werden, jedoch nicht umgekehrt. In der Titelzeile des rechten Fensterbereichs wird ein Symbol angezeigt, um zu verdeutlichen, daß dieser Teil schreibgeschützt ist. Sie können den Code auf der linken Seite übrigens auch einfach bearbeiten: CompareIt! funktioniert auch wie ein normaler Editor.



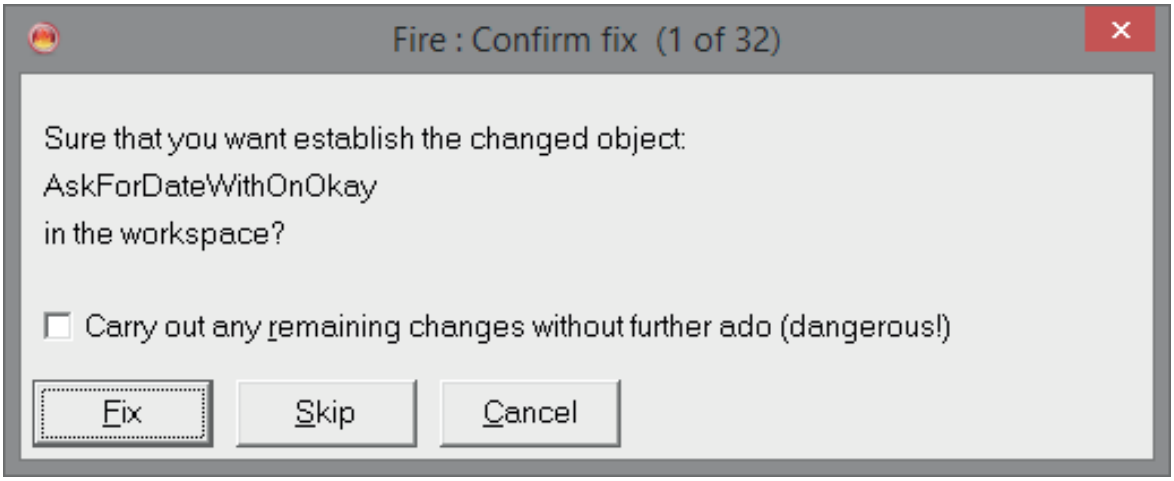
Was auch immer Sie bevorzugen, dies sollte das Ergebnis sein:



Sobald das CompareIt!-Fenster geschlossen wird, erfolgt diese Abfrage:

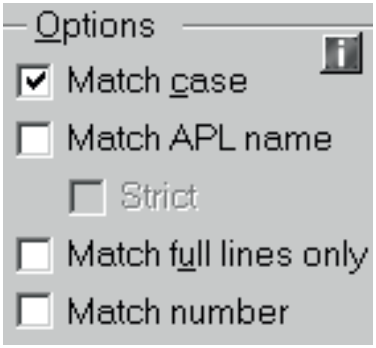


Wird diese mit „Yes“ beantwortet übernimmt Fire:



Fallstudie II

Mit Version 3.3.0 änderte sich Fires Benutzerschnittstelle: ein neues Kontrollkästchen „Strict“ steht zur Verfügung:



Die Option ist nur aktiv, wenn die Option „Match APL name“ aktiviert ist. Wenn beide Kontrollkästchen markiert sind, dann wird ein String wie „`Foo.`“ in „Search for“ von Fire nicht akzeptiert. Ist aber nur „Match APL name“ aktiviert, dann kann man nach „`.Foo`“ , „`Foo.`“ oder „`.Foo.`“ suchen. In diesem Fall würde es z.B. “`#.Foo.This`“ finden aber „`#.Foo2.This`“ ignorieren.

Im April 2014 realisierte ich, dass eine solche Option nützlich sein kann und implementierte es sofort in Fire. Das verursachte ein Problem: Zu jener Zeit verfügte Fire über 121 Testcases. Die überwiegende Mehrheit dieser Testcases erzeugt das GUI und setzt dann die gewünschten Eigenschaften. So sieht ein typischer Testfall aus:

```
R←Test_Search_001 (stopFlag
batchFlag);n;⎕TRAP
⍝ Search for “a” everywhere with
“Names only”
∘⎕TRAP←(999 ‘C’ ‘. ⍝ Deliberate
Error’)(0 ‘N’)
∘R←1
 ⍝ Preconditions
1 #.Fire.Run 0
n←#.Fire.GUI.n
n.SearchFor.Text←‘a’
n.LookIn.Text←‘#’
n.Case.State←0
 ∘n.APL_Name.State←0
∘n.FullLineOnly.State←0
∘n.AsNumber.State←0
∘n.Vars.State←1
∘n.FnsOprsTrad.State←1
∘n.FnsOprsDirect.State←1
∘n.Classes.State←1
∘n.Interfaces.State←1
 n.ScriptedNamespaces.State←1
°n.Code.State←1
°n.NoComments.State←1
°n.NoText.State←0
°n.CommentsOnly.State←0
°n.TextOnly.State←0
°n.NamesOnly.State←1
°n.HeaderOnly.State←0
°n.LocalsOnly.State←0
°n.NamedNamespaces.State←1
°n.UnnamedNamespaces.State←0
°n.GuiInstances.State←1
°n.Recursive.State←1
°n.RecursiveOneLevel.State←0
°n.RecursiveNone.State←0
°n.Negate.State←0
°n.ReuseSearch.State←0
°n.acre.State←0
°n.acre.State←0
°{} ΔSelect n.StartBtn
°ΔProcess n.Form
°→PassesIf 0<0⊃⍴n.HitList.ReportInfo
⍝ Tidy up CloseFire
°R←0        ⍝ Okay
```

Diese Zeilen sind ein Problem:

```
°n.APL_Name.State←0
°n.FullLineOnly.State←0
```

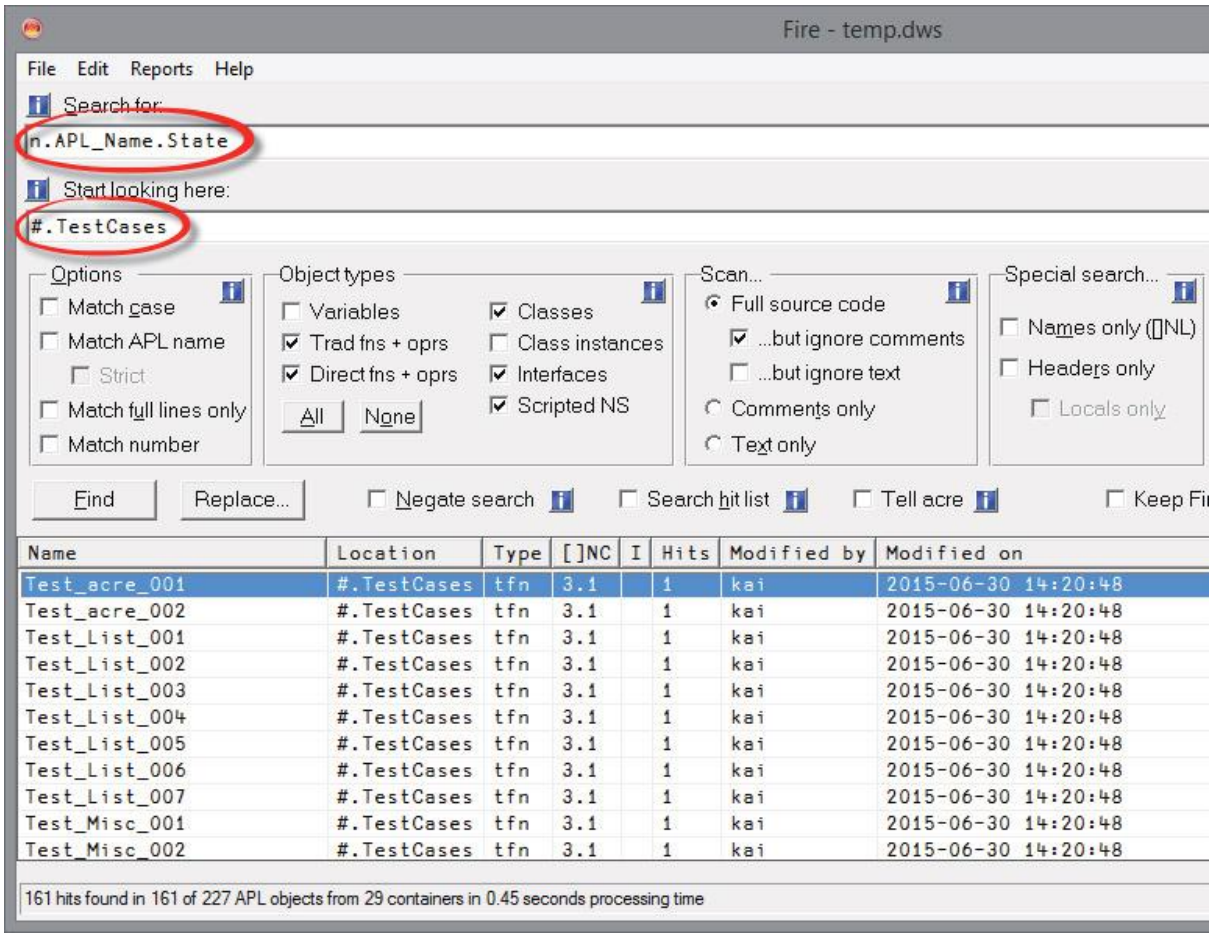
Nach der Zeile:

```
n.APL_Name.State←0
```

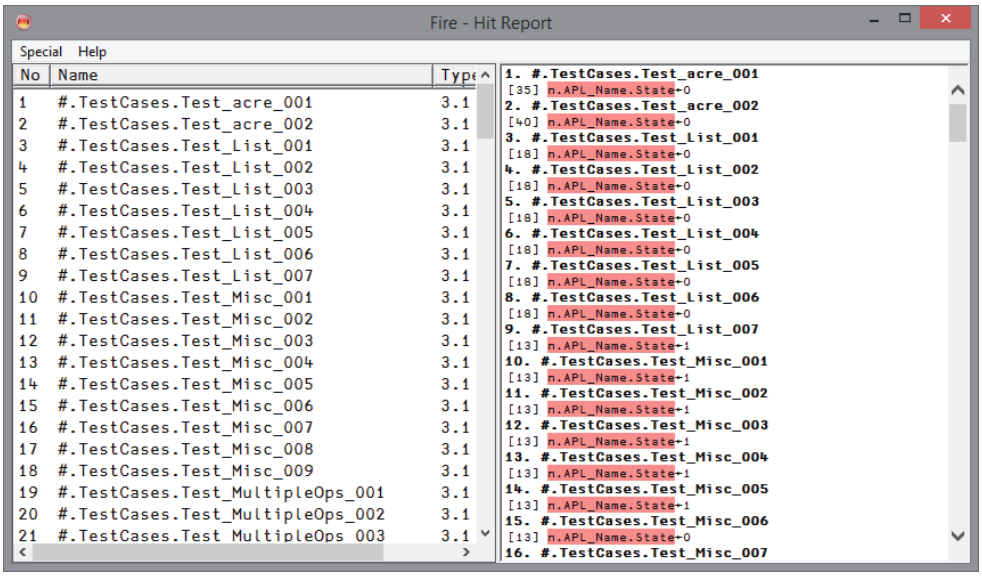
sollte diese Zeile folgen:

```
n.StrictOnNames.(Active State)←0
```

Auch dies kann mit Fire erreicht werden. Zunächst suchen wir nach „`n.APL_Name.State`“ aber nur in `#.TestCases`:



„Report hits“ bestätigt uns, dass wir auf dem richtigen Weg sind:

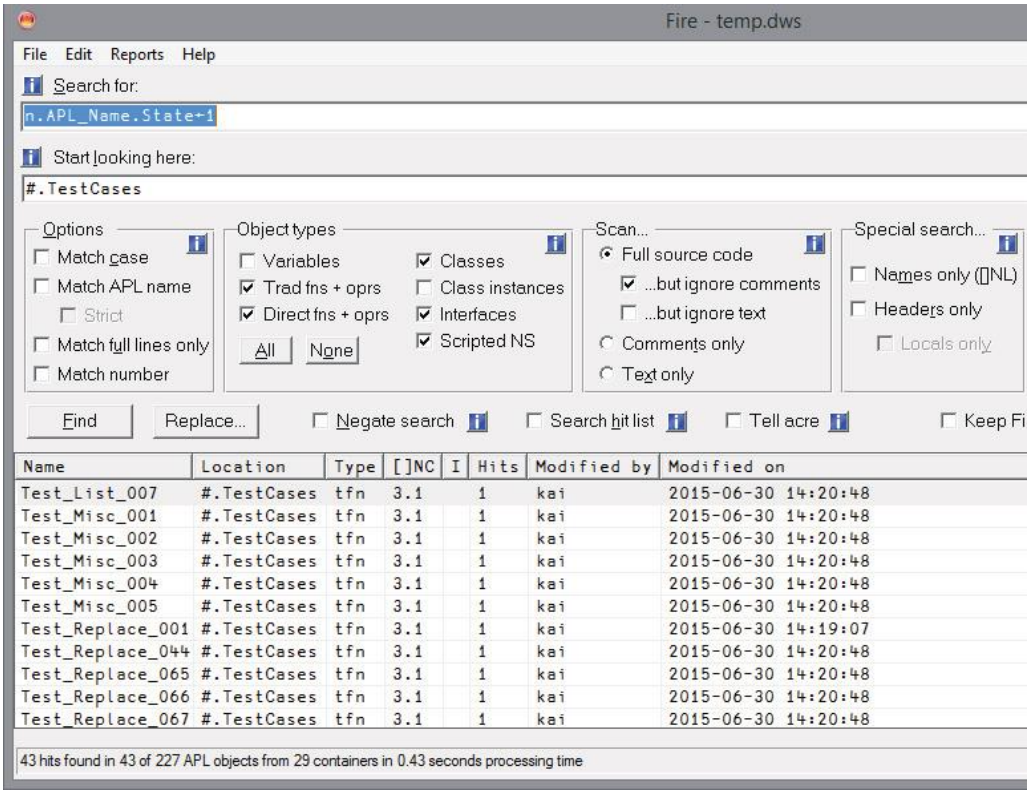


Leider zeigt der Report auch, dass manchmal eine 1 und manchmal eine 0 zugewiesen wird. Daher müssen wir die Aufgabe in zwei Schritten lösen.

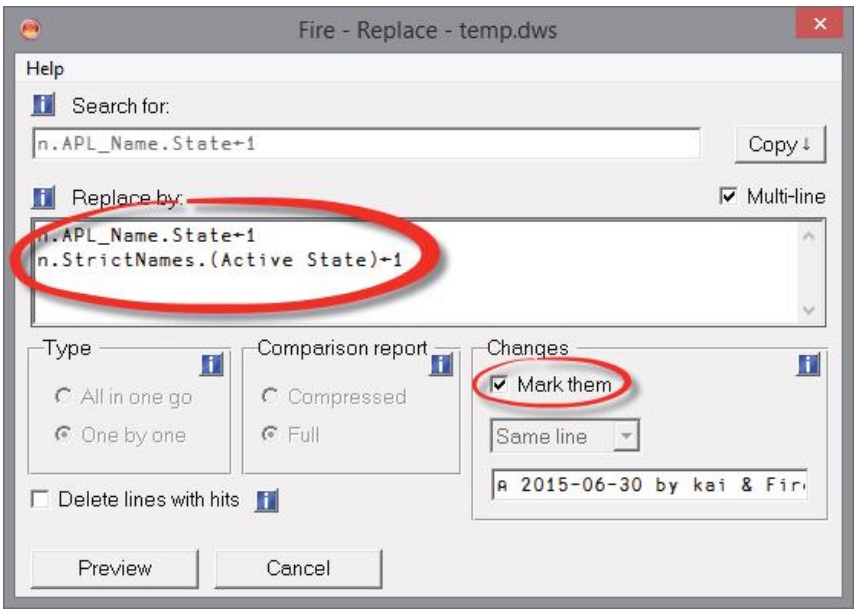
Als Erstes geben wir dies in „Search for“ ein:

```
n.APL_Name.State←1
```

und wiederholen dann die Suche. Dies muss zu weniger Treffern führen. Tatsächlich erhalten wir nur 43 Treffer:



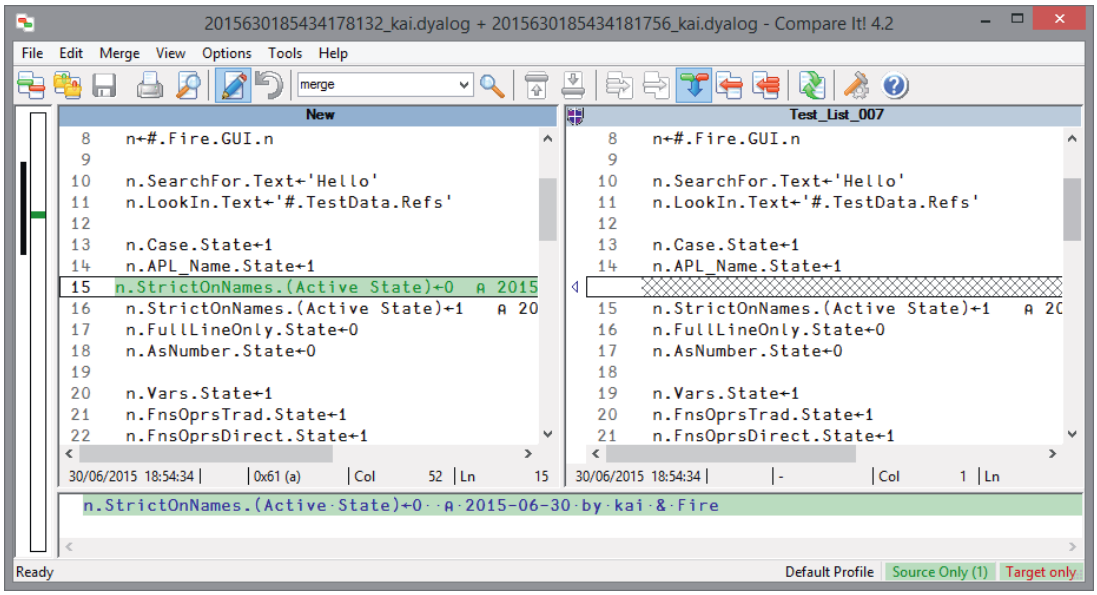
In „Replace“ kreuzen wir die Option „Multi-line“ an. Dann wiederholen wir den Suchtext gefolgt von einer zweiten Zeile mit dem Statement, welches wir hinzufügen möchten:



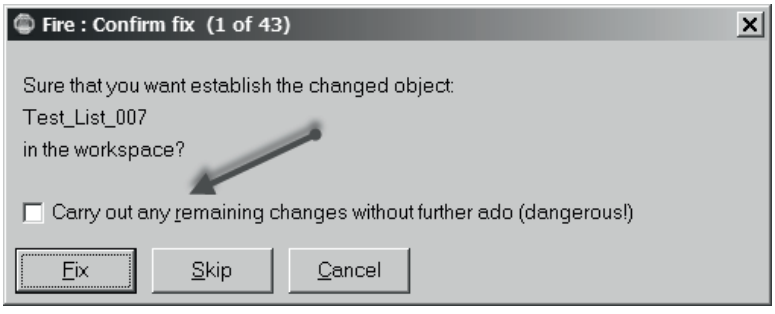
Beachten Sie, dass die „Mark them“ Option aktiviert ist und das die Combo Box darunter „Same line“ lautet. Das bedeutet, die hinzugefügte Zeile wird markiert, standardmäßig durch den String:

```
⍝ yyyy.mm.dd by {⎕AN} & Fire
```

Nach einem Klick auf „Preview“ können wir die Ergebnisse für eine Funktion nach der anderen überprüfen:



Für eine große Anzahl von Änderungen kann sich dies als lästig herausstellen. Wenn Sie sich ganz sicher sind, dass alles seine Ordnung hat (haben wird), dann können Sie das Kontrollkästchen „Carry out any remaining changes without further ado“ ankreuzen:



Natürlich ist dies gefährlich, man sollte also ein Backup haben.

In einem zweiten Schritt können wir dies wiederholen mit einer Suche nach

```
    n.APL_Name.State ← 0
```

und einer hinzugefügten Zeile

```
n.StrictOnNames.(Active State)
← 0
```

Wir haben mehr als 100 Funktionen innerhalb weniger Minuten geändert.

Die Option „Strict“ hat übrigens eine andere Bedeutung wenn Sie nach „#“ oder „##“ suchen. Schauen wir uns ein Beispiel an: In einem WS haben wir nur 4 Objekte, zwei Class Skripte (APLTreeUtils und WinFile, beides Mitglieder des APLTree Projekts [2]) und zwei Funktionen, die sich in einem gewöhnlichen Namespace `#.MyApp` befinden:

```
∇ Run; A
[1]  A←#.APLTreeUtils
[2]  ⍙WSID←A.Uppercase ⎕WSID
[3]  #.WinFile.PolishCurrentDir
[4]  Workhorse 0
     ⍝⍝⍝⍝⍝∇
∇ {R}←WorkHorse dummy
[1]  r←0
[2]  ⎕←##.WinFile.PWD
     ⍝⍝⍝⍝⍝∇
```

Der Code in den Funktionen macht nicht allzuviel Sinn, reicht aber aus, um das Thema zu illustrieren. Beachten Sie, dass `Run` sowohl `#` als auch

`##` referenziert während `Workhorse` nur `##` referenziert.

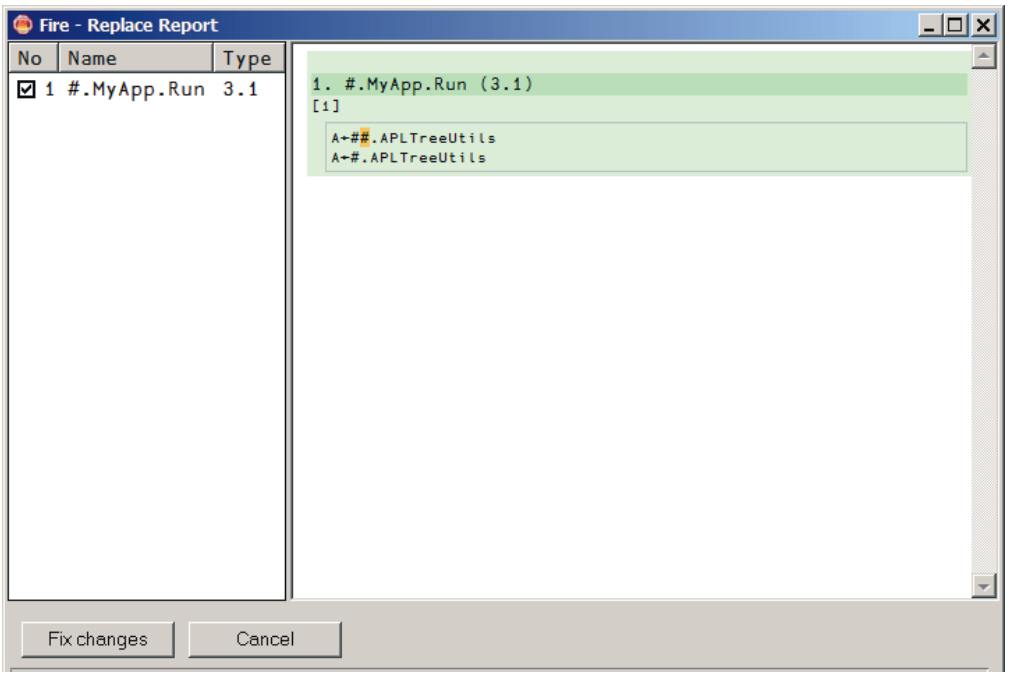
Jetzt lassen Sie uns annehmen, dass wir die beiden Funktionen in einem Dyalog User Command nutzen wollen indem wir den Code in ein User Command Script kopieren. Mit `Workhorse` funktioniert das problemlos. `Run` dagegen versucht `APLTreeUtils` und `WinFile` in `#` zu finden, und ob das funktioniert oder nicht, ist dem Zufall überlassen. In einem CLEAR WS gibt das auf jeden Fall einen VALUE ERROR.

Natürlich sollte ein User Command nicht annehmen, dass `APLTreeUtils` und `WinFile` sich in `#` befinden. Statt dessen sollte es versuchen, die notwendigen Utilities in einem übergeordenten Namespace zu finden. In der Praxis treten solche Konflikte aber trotzdem auf, z.B. weil der Autor der Funktionen nicht in Betracht gezogen hat, dass sie eines Tages als Bestandteil eines User Commands laufen würden, vielleicht gar nicht erwägen konnte, weil es zu der Zeit noch gar keine User Commands gab.

Eine andere Ursache ist, dass der Programmierer eigentlich seine Utilities mit `##.` referenzieren wollte und dies in der Regel auch tut, er es nur manchmal – versehentlich – eben nicht tut, und dies auch nicht merkt: es läuft ja alles.

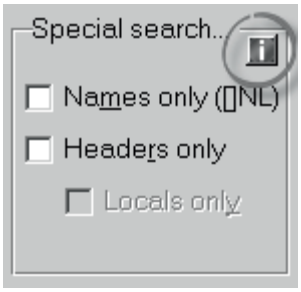
Kurz, um Code in einem User Command verwenden zu können, müssen wir alle Referenzen auf `#.` in `##.` ändern. Das Problem ist nun, daß eine normale Suche nach `#` oder `#.` nicht hilft, weil `##` und `##.` ebenfalls gefunden werden.

„Match name“ und „Strict“ lösen das Problem: Wenn beide Optionen aktiv sind, dann führt Fire eine spezielle Suche aus. Das führt in unserem Fall dazu, dass die Funktion `Workhorse` nicht gefunden wird, weil sie keine Referenz auf `#.` enthält. `Run` dagegen wird gefunden und würde geändert; hier ist ein „Preview“ auf die Änderungen:



Abschließend möchte ich Ihre Aufmerksamkeit auf die kleinen „i“s (für Informationen) lenken: Das sind Links zu den

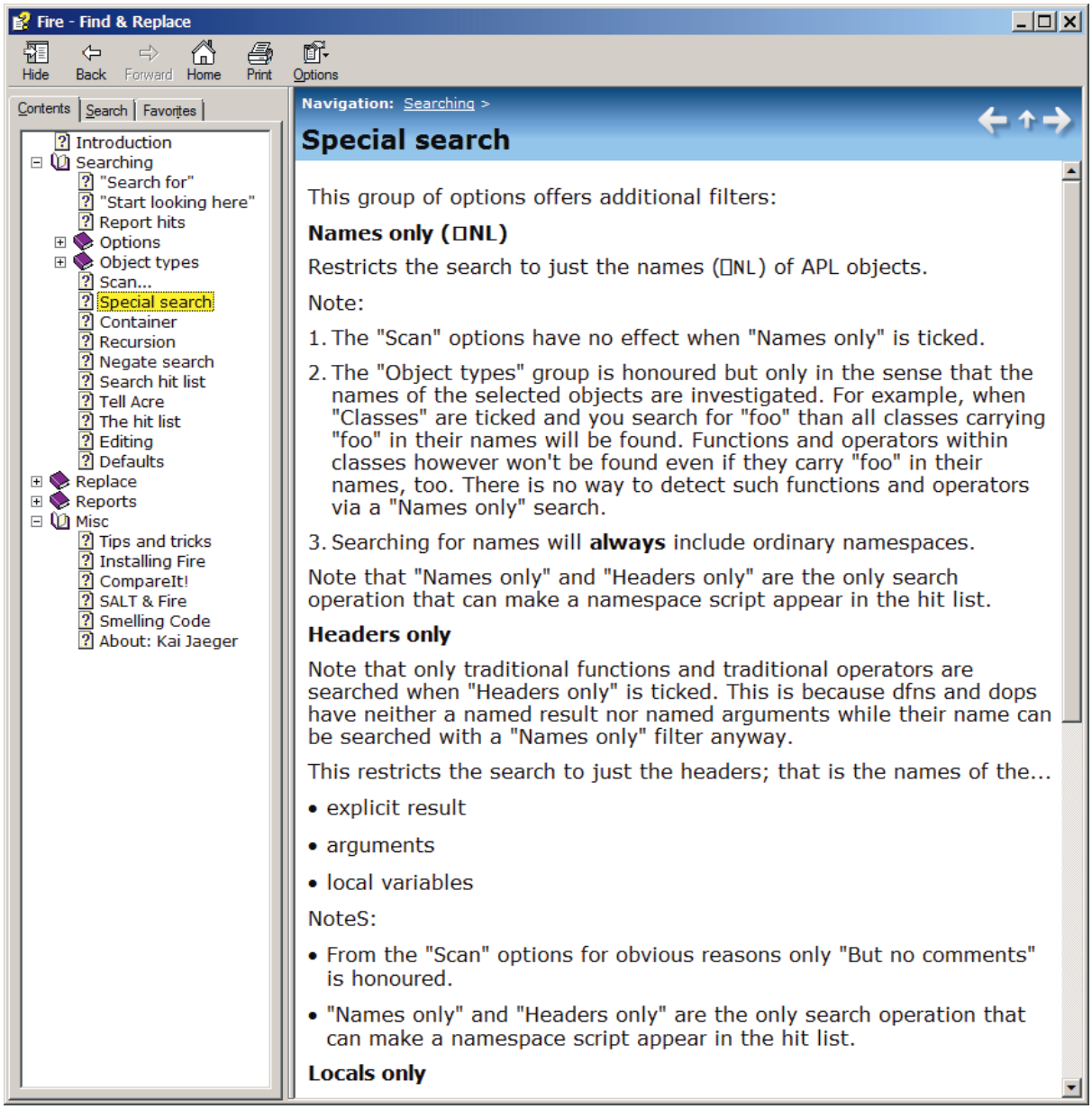
entsprechenden Seiten in der Fire Hilfe. Wenn Sie zum Beispiel dieses „i“ anklicken:



öffnet das die Seite in der Hilfe, die mit genau diesem Thema verbunden ist:

Im Laufe der Zeit könnten Sie die „i“s als überflüssig oder sogar ablenkend empfinden. Kein Problem, Haken vom Menübefehl „Help > Shw Info buttons“ entfernen und sie verschwinden.

Wenn man die Fire Hilfe in ein Word-Dokument umwandelt, dann umfasst dieses Word Dokument 33 Seiten. Das Duchsuchen eines WS und ggf. das Durchführen von Änderungen ist ein überraschend komplexes Geschäft.



Dieser Artikel hat nur einige der vielen Möglichkeiten von Fire beleuchtet. Ich hoffe aber, dass Sie dies bereits überzeugt hat, dass Fire ein mächtiges und nützliches Werkzeug ist, besonders – aber nicht nur – wenn man mit Legacy Code zu tun hat.

Fire ist ein Teil des APLTree Projekts[2,3] und als solches eine Art von Open Source [4]: Sie können es frei verwenden, können zur Codebasis beitragen oder eine Kopie nehmen und es für Ihre eigenen Zwecke ändern oder was auch immer sonst Sie mit dem Code tun wollen.

Fire hat eine eigene Seite auf der APL wiki [5] und kann von dort [6] heruntergeladen werden.

Referenzen

[1] Die acre Homepage auf dem APL wiki: http://aplwiki.com/acre

[2] „Sharing code: The APLTree Project“ von Kai Jaeger, Vector 25-3, http://archive.vector.org.uk/art10500730

[3] Alle APLTree Projekte auf dem APL wiki: http://aplwiki.com/CategoryAplTree

[4] Die APLTree Projektlizenz: http://aplwiki.com/AplTreeLicensing

[5] Die Fire Homepage auf dem APL wiki: http://aplwiki.com/Fire

[6] Die APLTree Download-Seite: http://download.aplwiki.com/apltree/

Kontakt

Kai Jaeger
IT Consultant
https://www.linkedin.com/in/kaijaeger
Email: kai@aplteam.com

Helmut Engelke

Geometrische Mandalas entwerfen mit APL2/AP207

APL2/AP207 erweist sich als flexibles Werkzeug, um spezielle Graphiken wie etwa Mandalas zu entwickeln. Der Workspace HESSE erlaubt es, die Figurenfamilie P128 als Malbuch auszudrucken oder Figuren per Programm auszumalen. Es gibt auch eine moderne Form des Malbuchs: das App auf dem Tablet PC. Tablets eignen sich wegen der graphischen Fähigkeiten hervorragend hierfür. Strukturen können durch Antippen von Farbe und Element bequem und schnell bearbeitet werden. Tatsächlich gibt es schon ein beträchtliches Angebot auch an Mandala Apps. Zu diesen könnte sich eines Tages ein App-P128 gesellen.

Geometrische Mandalas

Mandalas sind kunstvolle Bilder, die im Hinduismus und Buddhismus seit Jahrhunderten der inneren Einkehr und der Meditation dienen. Das Wort Mandala bedeutet in Sanskrit Kreis. Auch wenn der Umriss eines solchen Bildes ein Dreieck oder ein Rechteck ist, gibt es immer ein Zentrum, das mit Figuren in der Umgebung in Beziehung steht. Dieses Prinzip findet man auch in Malbüchern für Kinder verwirklicht, in denen Blumen, Kränze oder geometrischen Strukturen ausgemalt werden können. Die beruhigende und die Konzentration fördernde Tätigkeit wird mittlerweise auch von Erwachsenen geschätzt, die dem Trubel des modernen Lebens für eine kurze Zeit entgehen wollen.

Bei graphischen Anwendungen mit APL2 stieß ich zufällig auf Mandalas, als ich mit Geraden in der Hesseschen Normalform experimentierte.[1] Schon die ersten Bilder faszinierten mich (Abb. 1). Mit wenigen Zeilen APL und einem vielfachen Funktionsaufruf mit dem *Each*-Operator ergab sich eine komplexe Struktur um einen Mittelpunkt mit zahlreichen Schnittpunkten.

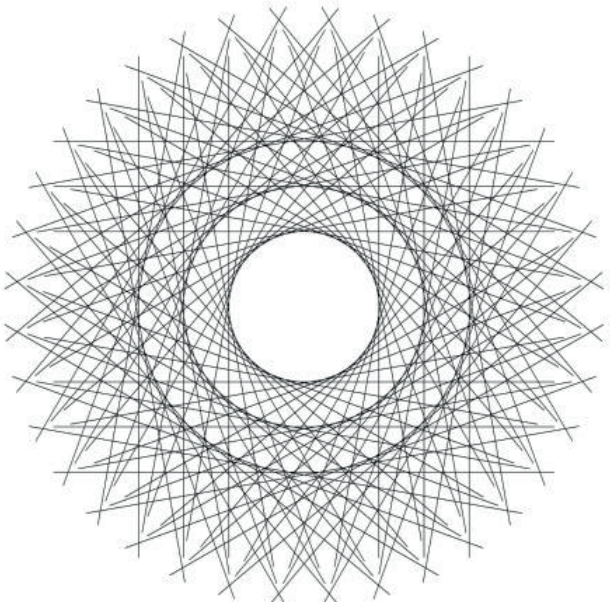


Abb. 1: Drei überlagerte Geradenscharen mit je 36 Geraden

[1] Ludwig Otto Hesse (1811 - 1874)

Danach fragte ich mich, welche Überraschungen zu erwarten wären, wenn ich den Goldenen Schnitt mit einbezöge. Mit dessen Eigenschaften hatte ich mich zwei Jahre vorher beschäftigt. Von einem Fünfeck mit einbeschriebenem Stern ausgehend, sah ich, dass zwei Geraden ausreichen, um diese Figur zu erzeugen. Hierbei werden zwei Geraden in 5 verschiedenen Richtungen gezeichnet, die sich jeweils um 72° von einander unterscheiden (Abb. 2).

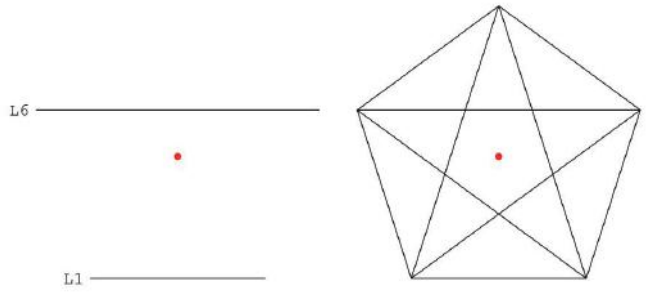


Abb. 2: Konstruktion eines Pentagons mit zwei erzeugenden Geraden L1 und L6

In mehreren Arbeitschritten verfeinerte ich nun die Struktur durch weitere Linien und erhielt mit 8 Geraden (L1 – L8) die Figur in Abb. 3.

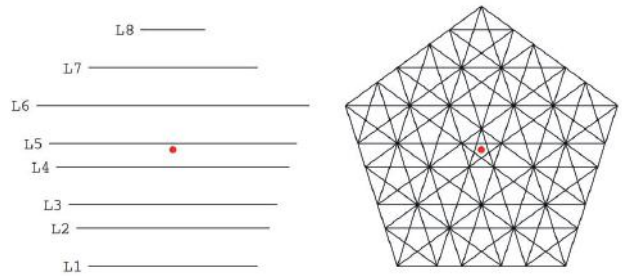


Abb. 3: Konstruktion eines Pentagons mit acht erzeugenden Geraden

Dieses Pentagon besteht aus 18 Sternen: dem Hauptstern der Abb. 2 mit 2 einbeschrieben kleineren Zentralsternen, 5 eng angrenzenden Sternen und 10 Sternen an der Peripherie. Die Flächenelemente der Figur sind regelmäßige Pentagone und Goldene Dreiecke.

Vielfalt der Figuren

Die Anzahl der Figuren steigt von 2 auf 2^8 = 256 Figuren, wenn alle möglichen Kombinationen von einer bis zu acht erzeugenden Geraden berücksichtigt werden. Bevor ich diese beschreibe, möchte ich kurz auf den Goldenen Schnitt und die Goldenen Dreiecke eingehen.

Der Goldene Schnitt

Der Goldene Schnitt wird durch die Proportion dreier Streckenlängen A, B und C definiert.

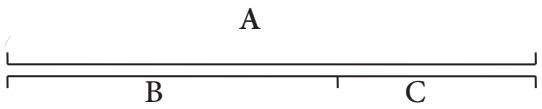


Abb. 4: Proportion dreier Streckenlängen

Die Strecke A soll so in B und C geteilt werden, dass das Verhältnis B/C = A/B ist. Setzt man C = 1 und B = x, so erhält man $x/1 = (x + 1)/x$ oder $x^2 = x+1$.

Die Lösung dieser quadratischen Gleichung ist der Goldene Schnitt:

$$\Phi = x_1 = (1 + \sqrt{5})/2 = 1.618\ldots$$

Der griechische Buchstabe Φ ist die übliche Bezeichnung.

Für Potenzen von Φ gilt die Rekursionsformel $\Phi^2 = \Phi + 1$. Diese erlaubt es, beliebige Potenzen von Φ in einen linearen Ausdruck umzuformen. Außerdem gilt die Beziehung $\Phi = 2\times\cos 36°$, die man an *Goldenen Dreiecken* bestätigt findet. Dabei handelt es sich um zwei gleichschenklige

Dreiecke. Haben die Schenkel der beiden in der Skizze gezeigten Dreiecke die Länge 1, so hat die Grundseite des *breiten* Dreiecks die Länge Φ, die des *schmalen* Dreiecks die Länge 1/Φ. In beiden Dreiecken tritt der Winkel α = 36° auf.

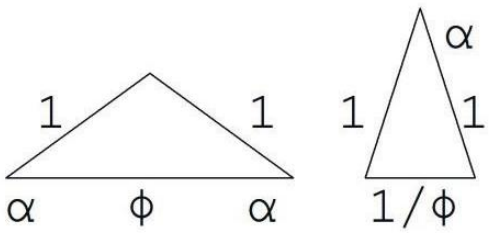


Abb. 5: Goldene Dreiecke

Nun zurück zu den 256 Figuren: Diejenigen ohne Umrandung sollen *Sterne* heißen, die mit Umrandung *Pentagone*. Die Sterne werden mit S1 bis S128 bezeichnet, Pentagone mit P1 bis P128. Die anfangs vorgestellten Figuren finden sich als P6 und P128 wieder. Da das Pentagon P128 alle 8 Erzeugenden verwendet und damit die Eigenschaften der übrigen Figuren in sich vereint, soll die ganze Familie ebenso *P128* heißen. [2] In den vier Tafeln Abb. 6 bis 9 sind alle 256 Muster als Miniaturen dargestellt.

Beschreibung der Figurenfamilie P128
Symmetrie-Eigenschaften

Alle Sterne und Pentagone sind bei ein- und mehrfachen Rotationen um 72 Grad und bei Spiegelungen an 5 Achsen deckungsgleich mit ihrer Ausgangfigur.

Da Sterne und Pentagone die meisten Eigenschaften gemeinsam haben, sollen nur die Pentagone näher beschrieben werden.

Der größte Stern S6 soll Hauptstern heißen. S6 ist eine geschlossene Kurve, die ihren Mittelpunkt zweimal umläuft. In P128 gibt es noch weitere *Ringelkurven*.

P1 - P8 aus 2 erzeugenden Geraden

In dieser Reihe nimmt der Innendurchmesser (die *Apertur* des Pentagons) von P1 bis P5 ab und bis P8 wieder zu. P4, P5 und P6 enthalten 3 konzentrische Sterne. P5 teilt das Pentagon in Trapeze und Obeliske ein, die sich teilweise überdecken.

P9 - P29 aus 3 erzeugenden Geraden

In dieser Gruppe entstehen Streifen, Bänder und radiale Strahlen. Die Pentagone P11, P12, P20 und P21 enthalten Ringelkurven mit den Umlaufzahlen 4, 4, 3, und 4.

P30 - P64 aus 4 erzeugenden Geraden

Hier gibt es Überlagerungen von Ringelkurven, Sternen und Streifen. P39 enthält 3 konzentrische Sterne. P48 ist eine Überlagerung von P6 mit Ringelkurve P21. P52 ist ein Knickband, d.h. ein Band, das am Rand der Figur wie ein Textilband schräg abgeknickt wird und unter einem Winkel zurück läuft. P64 schmückt den Hauptstern mit Außenstreifen.

P65 – P99 aus 5 erzeugenden Geraden

P66 ist eine Figur mit fünffachen Schnittpunkten von Geraden. P69 enthält 3 konzentrische Sterne und jeweils 3 Keile am Rand. P86 ist eine Überraschung: die Figur besteht aus 6 gleich großen Sternen. Bei den äußeren 5 Sternen fehlt jedoch eine Kante. Sie könnte durch eine zusätzliche Gerade L8a generiert werden. P99 enthält 4 parallele Streifen und 5x8 Rauten.

[2] S1 ist eigentlich ein leerer Stern, wird jedoch mit P1 als Grenzfall besetzt.

P100 - P120 aus 6 erzeugenden Geraden

Diese 21 Figuren enthalten weitere Überlagerungen von Mustern. P109 enthält 3 konzentrische Sterne, Ecksterne und Strahlen.

P121 - P128 aus 7 und 8 erzeugenden Geraden

P124 setzt sich aus dem Stern S4 im Zentrum und 10 äußeren Sternen zusammen. P128 ist das einzige Pentagon mit 8 erzeugenden Geraden.

Mit einer Filterfunktion SELECT kann man leicht eine näher verwandte Gruppe der 256 Figuren auswählen. Verwandtschaft heißt hier nicht Gen- sondern Linienausstattung. Zur Gruppe gehören diejenigen Mitglieder, die bestimmte Linien haben, andere Linien aber nicht.

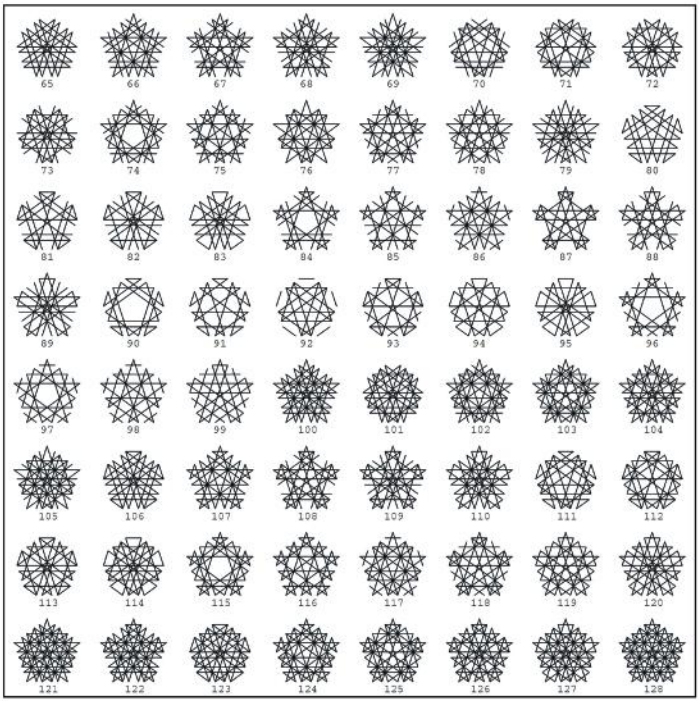


Abb. 7: Sterne mit 4 bis 7 Erzeugenden: S65 - S128

Sterne

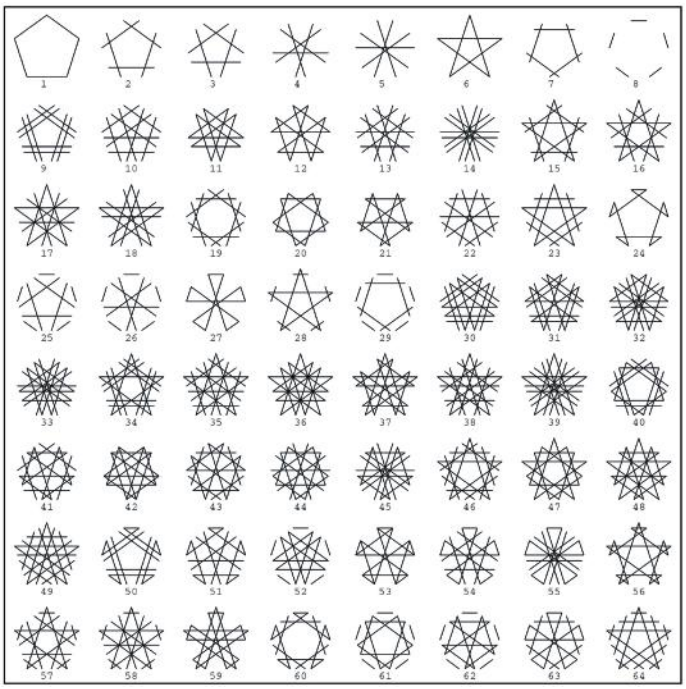


Abb. 6: Sterne mit bis zu 3 erzeugenden Linien: S1 - S64

Pentagone

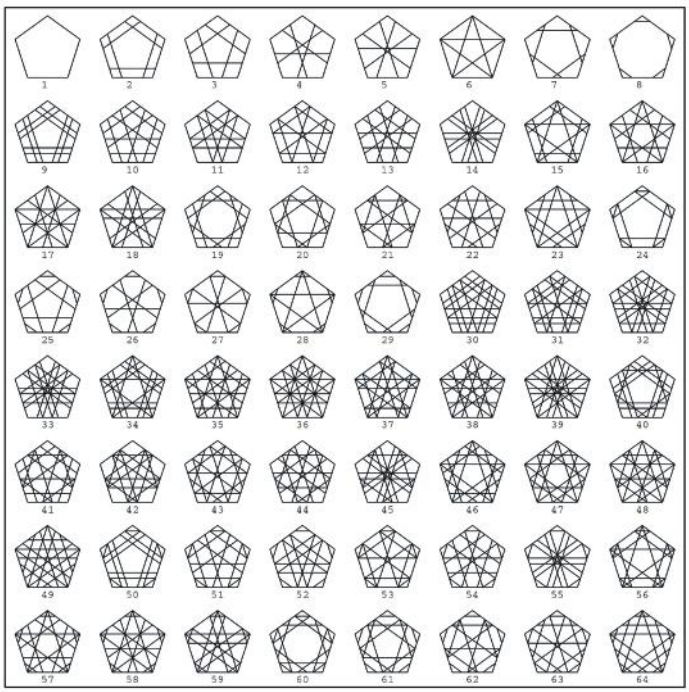


Abb. 8: Pentagone mit 2 bis 4 erzeugenden Linien: P1 – P64

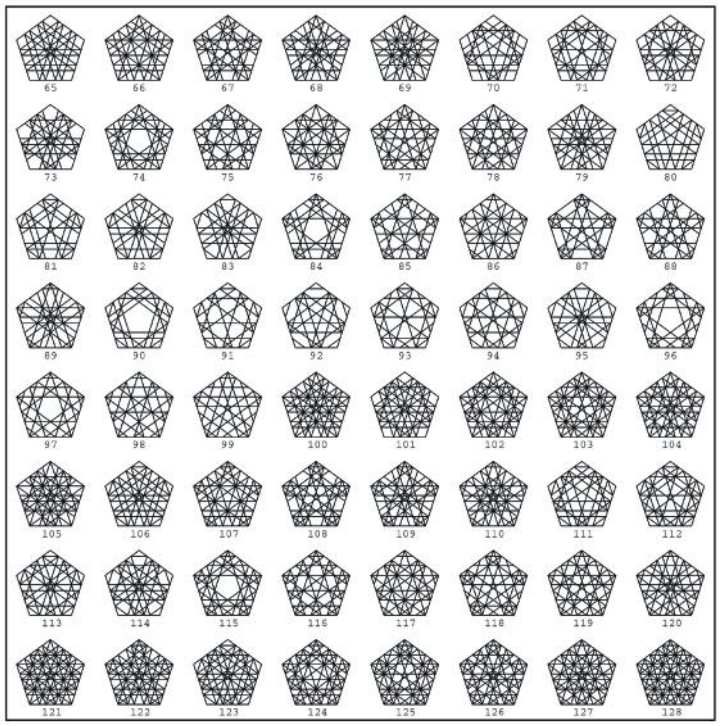


Abb. 9: Pentagone mit 5 bis 8 erzeugenden Linien: P65 - P128

Malbuch versus Malprogramm

Die Muster der Familie *P128* können in passender Größe und Auswahl als traditionelles Malbuch ausgedruckt und manuell ausgemalt werden.

Eine Alternative hierzu ist das Ausmalen mithilfe eines Graphik-Programmes. Da Bilder innerhalb weniger Sekunden mit hoher Genauigkeit berechnet und angezeigt werden können, kann der „Maler" seine ganze Aufmerksamkeit auf die Auswahl von Farben und ihre Zuordnung zu Flächenelementen konzentrieren. Er kann auf diese Weise mehr experimentieren, seine Entwürfe schnell verändern und optimieren. Die Realisierung eines *digitalen* Malbuches war das Ziel dieses Projektes.

Implementierung in APL2

Zur graphischen Darstellung wurde der Graphics Auxiliary Processor *AP207* verwendet; zum Zeichnen und Ausmalen die Funktionen DRAW, FILL und MARKER; außerdem die Funktionen COLOR und COLMAP. Um Bilder als *jpg*-Graphiken mit einer Auflösung von 4000x3000 Pixels abzuspeichern, wurden die beiden Funktionen INITCAM und SAVEPIC verwendet.

Graphik-Programm zum Ausmalen von Mandalas

Zunächst wird ein geometrisches Modell aufgestellt.

- Knoten der Struktur und Flächenelemente werden nummeriert
- Koordinaten der Knoten werden berechnet und
- Elemente werden durch zugehörigen Knoten beschrieben.

Nun können die Linien gezeichnet und die Flächen mit Farbe gefüllt werden.

Das Pentagon P128 besteht aus 125 Knoten und 191 Elementen.

Durch Ausnutzung der Symmetrieeigenschaften genügt es, eine Teilstruktur von 38 Elementen zu beschreiben. Deren geometrische Formen und Farben werden auf die vier anderen Teilstrukturen übertragen. (siehe Abb. 10)

Als Teilstruktur kann ein Trapez oder ein Obelisk mit jeweils 38 Elementen verwendet werden. Die Koordinaten der Knoten lassen sich auf einfache Weise bestimmen, da in P128 ausschließlich Goldene

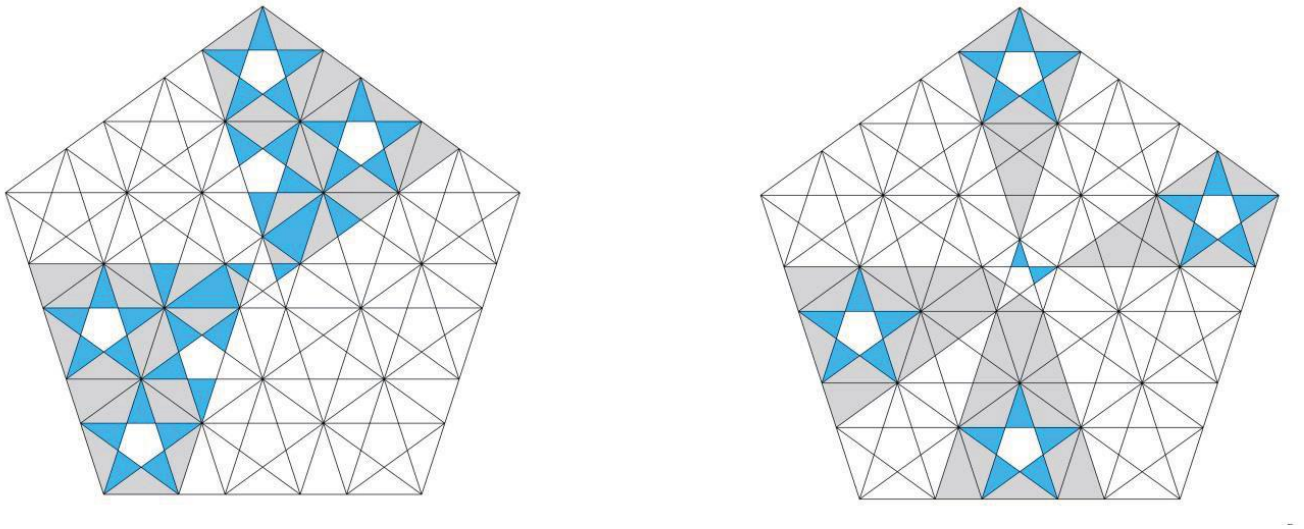


Abb. 10: Teilstrukturen von P128

Dreiecke und einzelne Pentagone auftreten. Die Knoten müssen nicht als Schnittpunkte der Geraden berechnet werden, sondern können als analytische Werte aus einer Zeichnung der Struktur abgelesen werden.

Techniken zur Auswahl von Farben und deren Zuordnung zu Elementen

In AP207 wird eine Standard-Palette mit 256 Farben angeboten, die mit Farbnummern ausgewählt werden können. 48 Farben der Palette enthalten Grundfarben und Grauwerte. 192 weitere Farben bilden einen RGB-Quader mit 6x8x4 Farben.

Um alle Elemente in P128 auszumalen, muss ein Vektor COLS191 mit 191 Farbnummern bestimmt werden. Der folgende Ausschnitt aus der Funktion GENCOLS zeigt einige Beispiele:

```
 GENCOLS
⍝   Waehle Farben aus
L3:⍝   Verwende 3 Farben
COL3←7 15 0
⍝   COL3←19    119 119
⍝   COL3←19   1 0
⍝   Definiere Farben f.    Sektor mit 38 Polygonen mit Farbindices   IC38←33⍴(10⍴1 2),3
⍝   Eckstern, Randstern, Innenstern                                  IC38←IC38,2 2 1 1 2
⍝   Randkeile, 2 Sterne und Mitte IC191←(190⍴IC38),3
⍝   Erweitere auf alle 5 Sektoren COLS191←COL3[IC191] ⋄0
L4:⍝   Generiere 191 Zufallsfarben aus 255, 38 oder 19 Farben  Z←⎕RL
⍝   Store the Random Link
⍝   Definiere Farben fuer 191 Polygone
⍝   COLS191←191?255
⍝   191 Elemente sind beliebig
⍝   COLS191←191⍴38?255
⍝   5 Sektoren mit denselben Zufallsfarben   COLS191←191⍴¯1+19 ?256
⍝   19 Zufallsfarben ⋄0
L5:⍝   Reproduziere einen frueher ausgewaehlten Farbvektor
⍝   durch Angabe eines Random Links
⍝   Bisher wurden 10 Random Links im Vektor VRL abgespeichert   ⎕RL←VRL[IRL]
⍝   Set  the random link ⍝C19←19?255
⍝   Generiere 19 Farben   C19[1 3 5 7 9]←0
⍝   Faerbe Sternzacken und weitere Pol. weiss  COLS191←(190⍴C19),0 ⋄0
```

Farben können so gewählt werden, dass die Symmetrie der Familie *P128* beachtet wird. Dies geschieht, wenn alle Sektoren (Trapeze oder Obeliske) dieselben Farben erhalten. Nimmt man jedoch 2 Farben und verteilt sie zufällig auf die 191 Elemente, wird die Symmetrie aufgegeben. Dabei können neue Formen entstehen.

Der Einfluss von Zufallszahlen auf ein Design kann sehr verschieden sein. Das Verhalten des *Zufallsmalers* kann unwissend, geschmacklos oder trivial erscheinen, andererseits auch unkonventionell, anregend, eigenwillig oder sogar kreativ. Zu große Freiheiten führen aber seltener zu guten Ergebnissen. Durch weniger Freiheiten lässt sich ein Design schneller optimieren.

Galerie

Das APL-Programm *ShowP128* malt Sterne und Pentagone aus. Es wurde bisher erst in kleinerem Umfang angewendet. Dennoch können 12 Bilder als Galerie präsentiert werden. Bei allen Bildern wurde die Struktur P128 verwendet. Die Farben können jedoch auch die Eigenschaften einfacherer Strukturen wiedergeben. So realisiert Bild 7 die Struktur P65, Bild 8 die Struktur P109.

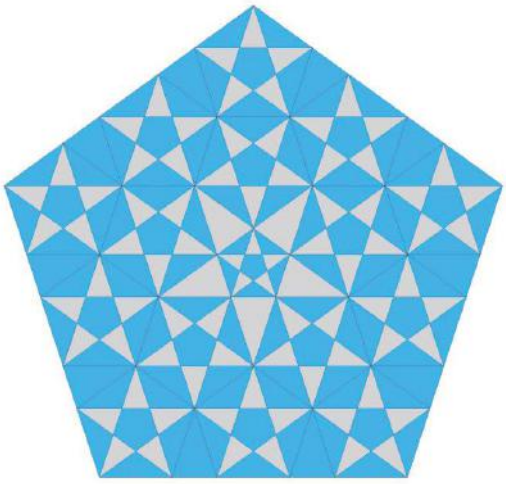


Bild 1: 2 Farben

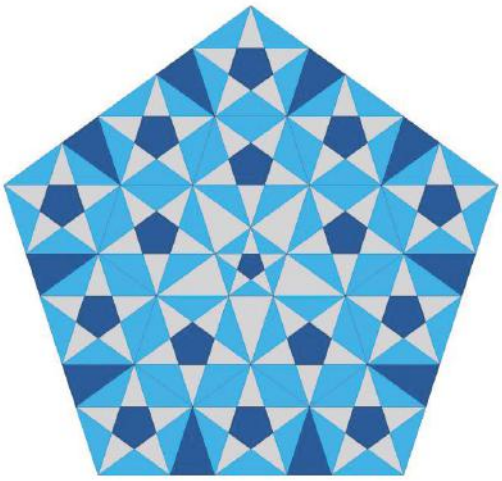


BILD 2: 3 Farben, die Keile und Sternmitten betonen

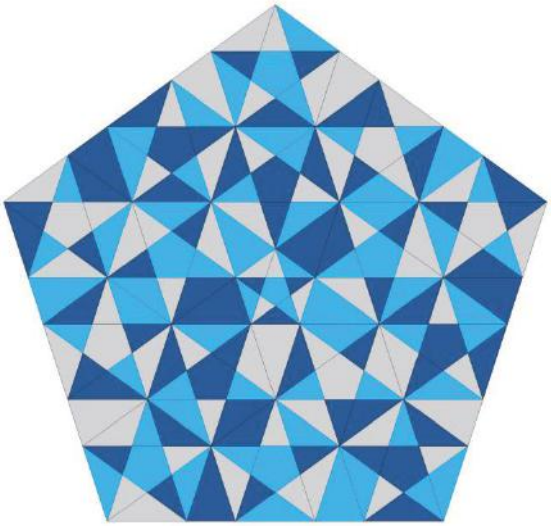


Bild 3: 3 Farben mit beliebiger Zuordnung zu Elementen b

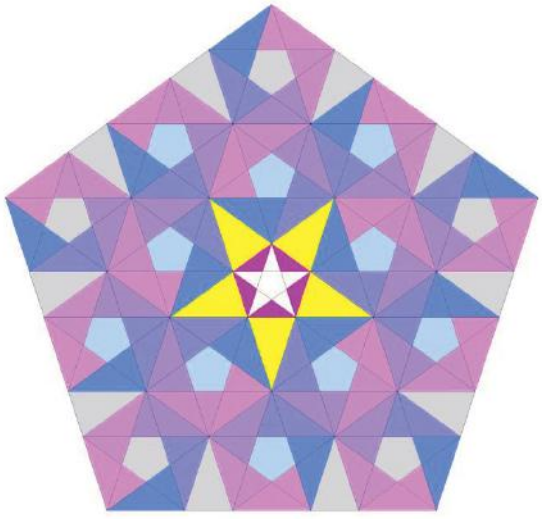


Bild 4: Manuell ausgemalt mit externem Programm

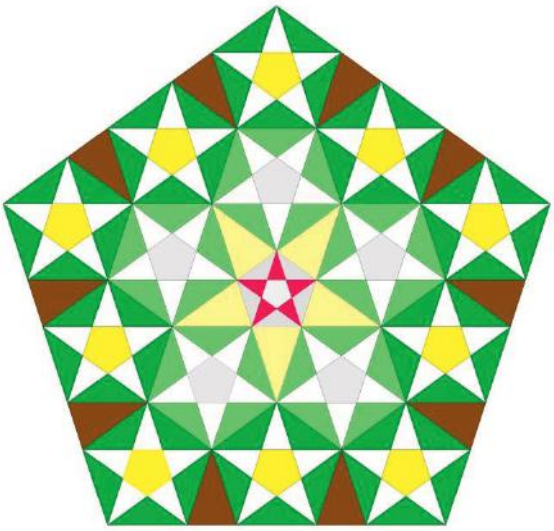


Bild 5: 7 Farben; helle Farben im Zentrum, dunklere am Rand

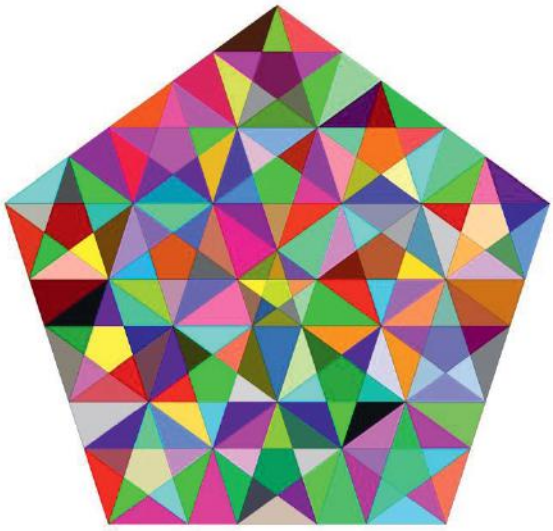


Bild 6: 38 Zufallsfarben aus Standard-Palette

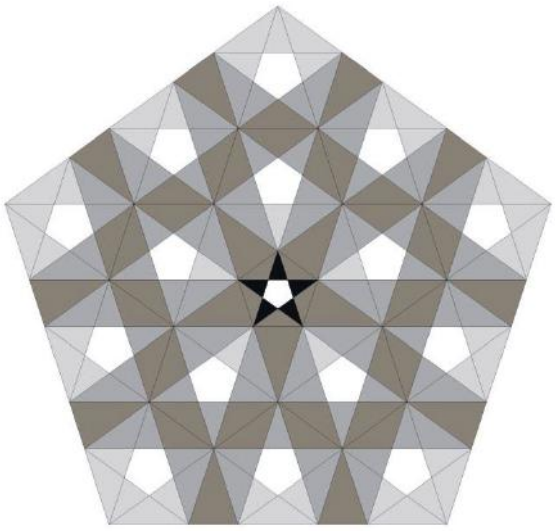


Bild 7: 3 Grautöne, die Transparenz suggerieren

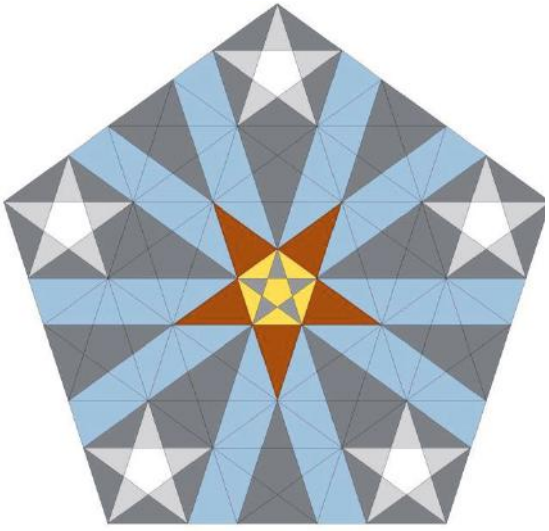


Bild 8: 6 Farben; Gelb war zuerst für die Strahlen vorgesehen

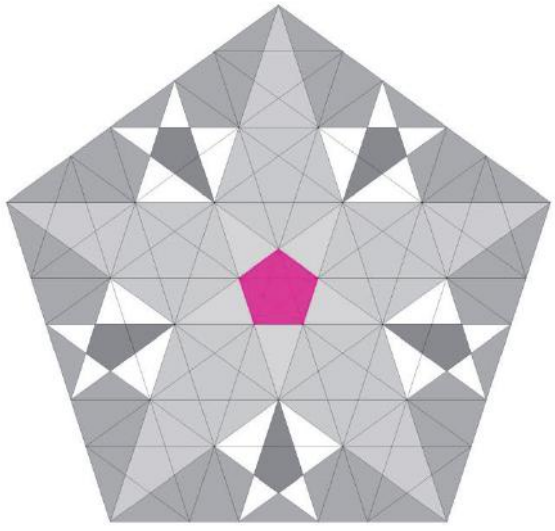


Bild 9: 6 Farben; 3D-Effekt durch Grauabstufungen

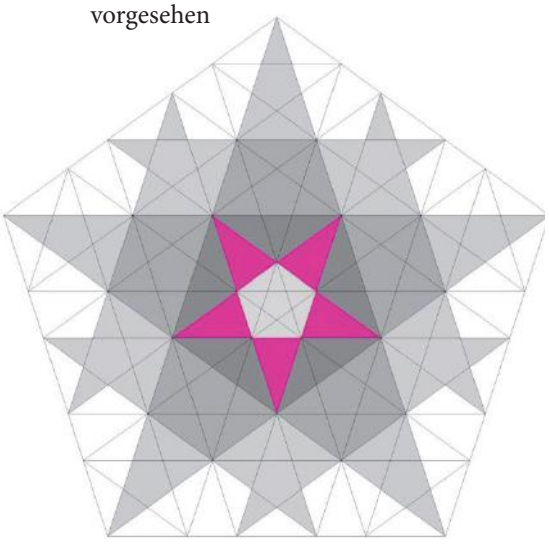


Bild 10: 6 Farben; 3D-Effekt durch Farbabstufungen

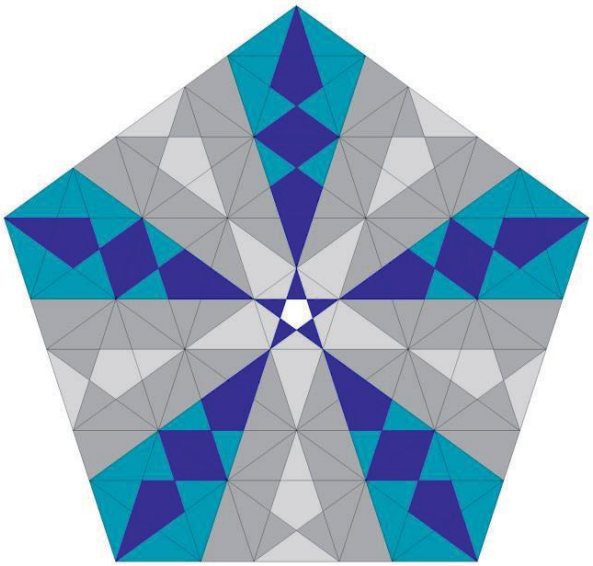


Bild 11: 6 Farben; Darstellung von 5 Obelisken

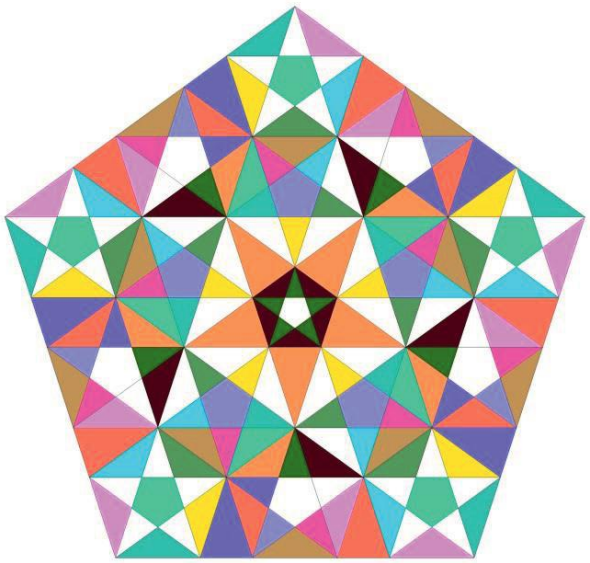


Bild 12: Gleiche Zufallsfarben für mehrere Sterne

Abb. 11: Galerie, Bilder 1 – 12

Ausblick

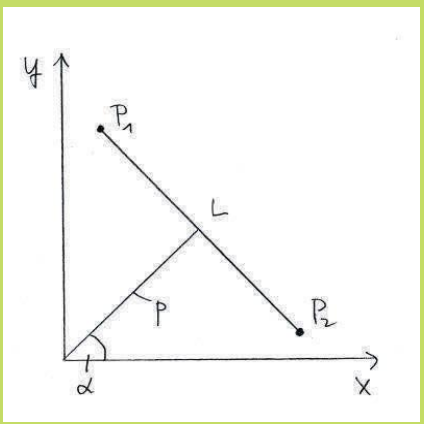
APL2/AP207 hat sich für mich wieder als flexibles Werkzeug erwiesen, um spezielle Graphiken zu entwickeln. Der Workspace HESSE erlaubt es, die Figurenfamilie P128 als Malbuch auszudrucken oder Figuren per Programm auszumalen. Beides soll in der Zukunft geschehen, zur Entspannung und zum Spaß an farbigen Objekten.

Es gibt jedoch eine moderne dritte Form des Malbuchs: das App auf dem Tablet PC. Tablets eignen sich wegen der graphischen Fähigkeiten hervorragend hierfür. Strukturen können durch Antippen von Farbe und Element bequem und schnell bearbeitet werden. Tatsächlich gibt es schon ein beträchtliches Angebot auch an Mandala Apps. Zu diesen könnte sich eines Tages ein App-P128 gesellen.

Anhang

Die Hessesche Normalform der Geraden lautet: $x \times \cos\alpha + y \bullet \sin\alpha - p = 0$.

α ist der Winkel der Geraden zur Abszisse x, p ist der Abstand der Geraden vom Koordinatenursprung.



APL2-Funktion für ein Geradenstück

```
    LINEH R;SIN;COS;X1;Y1;X2;Y2
⍝   Definiere Geradenstueck in der Hesseschen Normalform
    (ALPHA P L)←R (X0 Y0)←120  90
⍝     Ursprung des Koordinatensystems
⍝     ALPHA Winkel zwischen X-Achse und Normalen in Grad
⍝     P     Laenge der Normalen vom Koordinatenursprung aus
⍝     L     Gesamtlaenge  des  Geradenstueckes LH←L÷2
⍝     Berechne die halbe Laenge  SIN←1○○ALPHA×÷180
      COS←2○○ALPHA×÷180
⍝   Berechne Endpunkte der Linie  X1←X0+(P×COS)-LH×SIN
    Y1←Y0+(P×SIN)+LH×COS X2←X0+(P×COS)+LH×SIN Y2←Y0+(P×SIN)-LH×COS
    DRAW 2 3⍴0,X1,Y1,1,X2,Y2
⍝   Zeichne  Gerade
```

Kontakt: Dr. Helmut Engelke, E-Mail: helmut.engelke@online.de, Fon: (07031) 609432

Jürgen Sauermann

GNU jetzt auch mit APL

Das GNU Projekt – GNU ist die rekursive Abkürzung von „GNU is Not Unix“ – stellt viele freie Interpreter und Compiler für verschiedene Programmiersprachen bereit. Kurz vor dem 30. Geburtstag des GNU Projektes wurde die lange Liste von Programmiersprachen um die beiden „klassischen“ Sprachen Cobol und APL erweitert. Dieser Artikel befasst sich mit GNU APL.

GNU APL wurde in enger Anlehnung an den ISO Standard 13751 entwickelt. Die eine oder andere Lücke im ISO Standard wurde mittels Rückgriff auf die IBM APL2 Dokumentation, soweit sie auf dem Web verfügbar war, geschlossen. Das Ergebnis ist ein APL Interpreter der weitgehend APL2 kompatibel ist, so dass die meisten IBM Dokumente über APL2 auch für GNU APL verwendet werden können. Komplexe Zahlen und Shared Variables werden unterstützt.

GNU APL verzichtet bewusst auf zu viele nicht standardisierte □-Funktionen, da diese nach Ansicht des Entwicklers die Portierbarkeit von APL Programmen erschweren, was nicht im Sinne freier Software wäre. Stattdessen besitzt GNU APL einen Mechanismus – „native functions“ – der es dem Benutzer erlaubt, Funktionen oder Libraries die in anderen Programmiersprachen, typischerweise C oder C++, geschrieben wurden, aufzurufen. Dadurch ist es möglich, den Kern des Interpreters von □-Funktionen für spezielle Zwecke freizuhalten; der Benutzer kann solche Funktionen nach Belieben später hinzufügen ohne dass der Interpreter neu kompiliert werden müsste.

GNU APL verzichtet ebenfalls bewusst auf ein graphisches Benutzerinterface oder das Behandeln von APL Zeichen. Der Interpreter ist ein normales C++ Programm das seinen Input von stdin liest und seine Ausgaben auf stdout und stderr ausgibt. Die einzige Konvention dabei ist dass stdin und stdout/stderr UTF8-Kodierung verwenden. UTF8-Kodierung ist heute eine gängige Kodierung von Unicode Zeichen die auch auf Web-Seiten verwendet wird. Der vermeintliche Nachteil dieses Vorgehens ist das der Benutzer – genauer: die Plattform auf der GNU APL läuft – für die korrekte Darstellung von APL Zeichen verantwortlich ist. Für moderne Betriebssystem wir GNU/Linux ist das jedoch kein wirkliches Problem da die meisten Programme heutzutage Unicode/UTF8 direkt verstehen. Lediglich das Aufsetzen der Tastatur kann etwas trickreich sein; GNU APL wird jedoch mit einer Beschreibung verschiedener Methoden hierfür geliefert.

Der Vorteil des stdin/stdout/stderr Ansatzes liegt darin dass – dank Unicode – normalerweise keine Fonts oder ähnliches installiert werden müssen und auch das Drucken von APL Programmen macht auf halbwegs modernen Druckern keinen Ärger mehr. Ein anderer Vorteil ist die Möglichkeit Skripts in APL zu schreiben. Zum Beispiel ist die Homepage von GNU APL – http://www.gnu.org/software/apl – als CGI-Skript in APL geschrieben. Den Skript selbst kann man unter http://www.gnu.org/software/apl/APL_demo.html nachlesen. Auch wenn es zunächst etwas merkwürdig erscheint Skripts in APL zu schreiben, so gewöhnt man sich schnell daran, APL Programme, wie in anderen Programmiersprachen auch, in separaten (APL-) Text Dateien zu schreiben und den Interpreter mit diesen Dateien zu starten. Der klassische interaktive APL Modus – ∇-Editor,)LOAD,)COPY, and)SAVE – existiert jedoch weiterhin.

Unterstützte Plattformen

Der Verzicht auf ein bestimmtes graphisches Benutzerinterface macht es sehr einfach, GNU APL auf verschiedenen Plattformen zu kompilieren. Bisher wurde berichtet dass GNU APL auf Linux, Free BSD, Apple OS-X, Sun Solaris, Android und Microsoft Windows (mittels Cygwin) läuft. Auch andere CPU (Central Processing Units) wie z.B. ARM stellen kein Problem dar, solange ein C++ Compiler wie GNU gcc zur Verfügung steht. Wer Spaß am Basteln hat kann beispielsweise mit dem Raspberry Pi einen APL Rechner für unter 30 Euro (excl. Tastatur und Monitor) bauen.

Die erste Version von GNU APL benutzte noch einige Libraries die vor der Installation von GNU APL installiert werden mussten. Aufgrund von Portierungs-Problemen wurde die Anzahl dieser Libraries nach und nach reduziert, so dass die derzeit aktuelle Version von GNU APL praktisch nur noch Libraries verwendet die mit dem C/C++ Compiler geliefert werden.

Native Funktionen

GNU APL unterstützt prinzipiell das IBM APL2 Shared Variable Interface (⎕SVC, ⎕SVE, ⎕SVO, ⎕SVQ, ⎕SVR und ⎕SVS). Allerdings sind die Implementierungen der Shared Variablen nicht portabel und derzeit werden nur AP100, AP210, und zwischen Workspaces auf der gleichen Maschine gesharete Variablen von GNU APL unterstützt. Von einer Nutzung dieser Funktionen für neue Entwicklungen ist daher abzuraten. Stattdessen sollte Native Funktionen verwendet werden, um Interfaces zu Libraries zu bauen und dadurch beliebige Funktionen des Betriebssystems von APL aus aufzurufen.

Eine native Funktion wird mittels ⎕FX erzeugt, wobei das linke Argument der Name eines Shared Libraries ist, z.B: ‚lib_file_io.so' ⎕FX ‚FILE_IO' FILE_IO

Die Shared Library wird als sog. Plugin außerhalb des GNU APL Interpreters kompiliert und mittels ⎕FX geladen. Das lib_file_io.so Plugin gehört zum Lieferumfang von GNU APL und stellt die klassischen C/C++ file Operationen (open(), close(), printf(), ...) zur Verfügung.

Download und Fehlerberichte

GNU APL kann als Release oder als Snapshot heruntergeladen werden. Snapshots sind etwas aktueller als Release und Fehler die im letzten Release noch nicht bekannt waren sind normalerweise kurz nach Bekanntwerden im jeweils letzten Snapshot behoben.

GNU APL Releases können vom nächstgelegenen GNU Mirror heruntergeladen, während Snapshots mittels Subversion (SVN) geholt werden können.Die GNU APL Homepage verweist auf die entsprechenden Seiten.

Es gibt eine Mailing-liste (bug-apl@gnu.org) auf der Fehler in GNU APL berichtet und sowie Verbesserungsvorschläge gemacht werden können. Das Behandeln von Fehlern ist ein unbürokratischer und normalerweise schneller Vorgang und jede Fehlermeldung ist willkommen.

GNU APL Community

Seit dem Erscheinen von GNU APL hat sich eine kleine Community von Benutzern entwickelt die – ganz im Geiste freier Software – GNU APL eigene Anwendungen und Erweiterungen entwickelt haben und anderen Benutzern zur Verfügung stellen.

Die Seite http://www.gnu.org/software/apl/Community.html enthält Links zu den wichtigsten Beiträgen. Hierzu gehören Portierungen von GNU APL nach Android und JavaScript, Interfaces zu verschiedenen Filesystemen (SQL Datenbanken, Component File System, Keyed File System), APL Package Manager, Einbindungen von GNU APL in emacs, vi und GTK, Plotting Interfaces, und mehr.

Auf http://baruchel.hd.free.fr/apps/apl/ kann man GNU APL (mit etwas eingeschränkter Funktionalität) online ausprobieren.

Multi-Core APL

Der Autor hat bereits 1990 bewiesen, dass praktisch alle APL Funktionen und Operatoren effizient parallelisierbar sind [1]. Damals gab es an der Universität des Saarlandes den Prototypen eines parallelen APL Rechner auf dem einzelne APL Funktionen implementiert und gemessen wurden um die Prognosen der Dissertation zu bestätigen. Leider fehlte damals ein APL-Interpreter der das Ganze abgerundet hätte.

Es war daher naheliegend GNU APL so zu entwickeln dass eine Parallelisierung der primitiven APL Funktionen leicht möglich ist. Die letzten Snapshots von GNU APL, die auch im nächsten Release 1.5 enthalten sein werden, können so konfiguriert werden dass alle skalaren Funktionen sowie innere und äussere Produkte von skalaren Funktionen auf einer Multi-cor-CPU parallel berechnet werden.

Das Ganze wurde gemessen, aber die Ergebnisse waren etwas ernüchternd. Der erste Parallelisierungs-Versuch benutzte ein gängiges Parallelisierungs-Library. Das erste Problem, das sich stellte, war eine

[1] Sauermann, J. : Ein paralleler APL-Rechner, Dissertation an der Universität des Saarlandes, Saarbrücken 1990

relative hohe Start-up-Zeit von 10000-20000 CPU-Zyklen für jede parallele Ausführung. Demgegenüber beträgt die typische Rechenzeit der skalaren Funktionen beträgt einige 100 Zyklen pro Vektorelement so dass die parallele Ausführung nur für relative lange (und damit eher selten) Operanden schneller wäre.

Daher wurde die Parallelisierungs-Library durch eine vereinfachte und dadurch schnellere direkte Implementierung ersetzt, die die Start-up Zeit unter 1000 CPU-Zyklen brachte. Dadurch konnte ein messbarer Zeitgewinn schon bei kurzen Operanden erzielt werden – allerdings nur bei skalaren Funktionen, die relative viele CPU-Zyklen benötigen (z.B. ⋆ oder ⍟). Bei den meisten skalaren Funktionen war die parallele Ausführungszeit jedoch größer als die sequentielle. Der Grund hierfür ist unbekannt. Durch begleitende Messungen können Cache-Konflikte oder das Speicher-Interface als Ursache ausgeschlossen werden.

Sei N = ρ,B. Dann ist die sequentielle Implementierung einer dyadischen skalaren Funktion f, also Z←A f B gegeben durch:

for (int i = 0; i < N; ++i) Z[i] ← A[i] f B[i]

Die parallele Implementierung von f auf Core c einer C-core CPU ist dann:

for (int i = c×N÷C; i < (c+1)×N÷C; ++i)
Z[i] = A[i] f B[i]

Wenn die sequentielle Implementierung T CPU Zyklen benötigt, und wenn die Cores sich nicht gegenseitig beeinflussen, dann sollte die parallele Implementierung α + T÷C CPU Zyklen benötigen. Für hinreichen großes N (für die Messungen wurde z.B. N=1000000 gewählt, dann spielt die ohnehin relativ kurze Start-up-Zeit α keine Rolle mehr. Trotzdem wurden für einfache skalare Funktionen parallele Ausführungszeiten gemessen die sogar absolut (also nicht nur pro Vektorelement) größer als T waren während kompliziertere skalare Funktionen sich T÷C näherten.

Eine andere Beobachtung war die dass das Speed-up älterer CPU bei der parallelen Ausführung besser war als bei moderneren CPUs (die aber insgesamt weniger Zyklen benötigten als ihre Vorgänger).

Zusammenfassend lässt sich schließen, dass die parallele Ausführung von APL auf Multi-core-CPU zur Zeit an Grenzen stößt, die irgendwo innerhalb der CPU zu liegen scheinen. Es ist jedoch möglich, dass sich CPU in einer Weise weiterentwickeln, in der diese Grenzen verschwinden, so dass die parallele Ausführung von APL irgendwann interessant wird.

Die Homepage von GNU APL ist: http://www.gnu.org/software/apl. Fehler können reportet werden an: bug-apl@gnu.org. Download: ftp://ftp.gnu.org (oder Verzeichnis 'apl' auf jedem GNU mirror, siehe http://www.gnu.org/prep/ftp.html)

Kontakt

Dr. Jürgen Sauermann
Entwickler und Maintainer von GNU APL
E-Mail: juergen.sauermann@t-online.de

Leserbrief

Sehr geehrter Herr Nussbaum und sehr geehrter Herr Oswald,

im APL – Journal 1/2/2014 fand ich den Artikel „Sudoku mit APL“. Da erinnerte ich mich, daß ich dieses Spiel im Jahr 2006 im Urlaub bei meinen belgischen Freunden kennen gelernt habe und ich beschloss damals die einfachen Regeln unter dem Thema „Genuss-Mathematik“ per APL zu behandeln. Das Lösen eines Sudoku-Rätsels auf dem Papier fand ich nicht so lustbetont und ich versprach mir eine größere Befriedigung, wenn ich mir einen Algorithmus am Computer erarbeiten würde. Im Kopf war mir die Lösung schon so gut wie fertig und ich glaubte, daß ich zu Hause das Problem in kürzester Zeit gelöst haben würde.

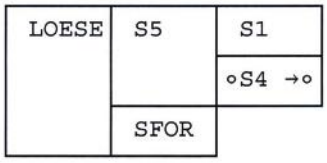
Doch schon das Erstellen einer vollständigen Sudokumatrix, aus der durch Weglassen von einigen Zahlen ein SDK-Rätsel entsteht, bereitete mir einiges Kopfzerbrechen und war nicht am ersten Tag erledigt. Ähnlich war es auch mit dem Lösungsalgorithmus, bei dem ich nicht auf menschliche Erfahrung zurückgreifen wollte, sondern es sollte konsequent die Eigenschaften eines PC und insbesondere die Strukturstärke von APL ausgenutzt werden. (Auch dachte ich nicht an eine pädagogische Verwendung bei meinen Studenten des Maschinenbaus)

Nach einigen Wochen war das Werk beendet, aber ich kann heute nicht mehr rekonstruieren, wie viele Manntage ich tatsächlich daran gearbeitet habe. Jedenfalls habe ich nun einen SDK-Rätsellöser der mit allen lösbaren SDK-Rätseln schnell fertig wird: die leichteren schafft er in Sekundenbruchteilen und die teuflischen in einer guten Sekunde. Auch Mehrdeutigkeiten sind kein Problem: So wurde beispielsweise ein SDK-Rätsel mit 19 verschiedenen Lösungen in 0.17 Sekunden gelöst. (Auf einem NoName-Rechner von 2005, mit APL*PLUS II for the 80386, Version 5.2, Serial Number 5408894, Copyright 1984-1992, Manugistics, Inc.) Auf den beiden Programmseiten sind die fünf APL-Programme gelistet, deren Zusammenspiel in einem Funktionsbaum rechts neben dem Programm LOESE gezeigt wird. Eigentlich sind aber nur die Programme S5, S1 und S4 notwendig, denn LOESE dient nur der Zeitmessung und SFOR besorgt eine schöne umrandete Ausgabe. Die einzelnen Programme, die jeweils auf einem Bildschirm darstellbar sind, sollen hier nicht erläutert werden – das wäre für mich und den Lesern zu langweilig. Ein echter APL-Sudoku-Freak sollte sich selbständig zurecht finden. Denn auch hier gilt wie in der Mathematik bekannt: „Es gibt keinen Königsweg...“

Mit herzlichen Grüßen

Dr. Joseph Specht, 69226 Nußloch

```
    ∇ M←LOESE M;T
[1]   T←⎕AI[2]
[2]   M←SFOR¨(⊂M),S5 M
[3]   ⎕AI[2]-T
    ∇
```




```
    ∇ R←S5 M;E;F;I;T;⍙S
[1]   ⍝ SPE 061104
[2]   ⍝ R = Sudoku-Matrix M deterministisch gefüllt
[3]   ⍝ T(BODO1   ) ~ 1.3  Sekunden
[4]   ⍝ T(RAINER1) ~ 0.35 Sekunden
[5]   ⍝ T("braf" ) ~ 0.16 Sekunden
[6]
[7]   E←'Keine lösbare Sudoku-Matrix'
[8]
[9]   ⍙S←S1                          ⍝ Zugriffs-Matrix: global für S4
[10]  T←(⍳9)S4¨⊂M                    ⍝ alle Möglichkeiten für ⍳9
[11]  F←,⊃1↑¨⍴¨T                     ⍝ Anzahl für Füllzahlen
[12]  ⍎(0∊F)/'R←E ◇ →0'              ⍝ Fehler-Abruch
[13]
[14]  I←2 ◇ F←⍋F                     ⍝ Füllzahlen nach aufsteigender Anzahl
[15]  R←⊃T[⍬⍴F]
[16]  L1:  R←F[I] S4¨ ⊂[2 3]R        ⍝ Möglichkeiten mit nächster Füllzahl
[17]       ⍎(⍬⍴∧/0=⊃⍴¨R)/'R←E ◇ →0'  ⍝ Fehler-Abruch
[18]       M←⍬⍴+/⊃⍴¨R
[19]       R←(M,9 9)⍴⍎,⍕((⍴R),1)⍴R  ⍝ tensorielle Umstrukturierung von R
[20]  →L1⌈⍳9≥I←I+1
[21]
[22]  R←⊂[2 3]R
    ∇
```

```
    ∇ R←S1;B;I;J;K;L;M;S;Z
[1]   ⍝ SPE 061021
[2]   ⍝ Sudoku-Zugriffsmatrix
[3]   ⍝ R[;1]       = Linear Nummer:  1  2  3 ... 40 41 42 ... 79 80 81
[4]   ⍝ R[;2]       = Matrix-Index : 11 12 13 ... 54 55 56 ... 97 98 99
[5]   ⍝ R[;3...22] = 20 Nummern der Zugriffs-Indizes
[6]   ⍝ RZ = 3.2 ms
[7]
[8]   B←3⌿3/1 4 7∘.,1 4 7                ⍝ Matrix mit Block-Start-Indizes
[9]   S←10⊥¨(⍳9)∘.,⍳9                    ⍝ Matrix mit S-Mat-Indizes
[10]  R←(⍳81),(,S),81 20⍴0               ⍝ Indizes der Einflußfelder
[11]
[12]      I←1
[13]  L1: J←1
[14]  L2: (K L)←⊃B[I;J]
[15]      M←(K+0 1 2)∘.,L+0 1 2          ⍝ Block-Lauf-Indizes
[16]      M←(S[I;],S[;J],,S[M])~S[I;J]   ⍝ Mögliche Indizes
[17]      M←((⍳⍴M)=M⍳M)/M ◇ M←M[⍋M]      ⍝ eindeutig und sortiert
[18]      R[R[;2]⍳10⊥I,J;2+⍳20]←M        ⍝ einordnen
[19]  →L2⌈⍳9≥J←J+1
[20]  →L1⌈⍳9≥I←I+1
[21]
[22]  R←R[;1 2],R[;2]⍳0 2⍴R              ⍝ Indizes in Nummern wandeln
    ∇
```

Abb. 1: Programm Listing von LOESE, S5 und S1. Im Funktionsbaum ist S4 als rekursiv markiert.

```
    ∇ T3←FZ S4 SM;M;SP;ZL
[1]   ⍝ SPE 061103
[2]   ⍝ Alle Möglichkeiten eine SM mit FZ zu füllen
[3]   ⍝ Rekursiver Loop über Spalten einer Zeile
[4]   ⍝ FZ = Füllzahl ∈ ⍳9
[5]   ⍝ T3 = (x,9 9)-Tensor mit gefüllten Matrizen
[6]
[7]   ZL←(∧/SM≠FZ)/⍳9
[8]   ⍎(0=⍴ZL)/'T3←1 9 9⍴SM ◇ →0' ⍝ keine mögliche Zeilen
[9]   ZL←⍬⍴ZL
[10]
[11]  SP←(SM[ZL;]=0)/⍳9              ⍝ mögliche Spalten
[12]  ⍎(0=⍴SP)/'T3←''Keine Sudokumatrix'' ◇ →0'
[13]
[14]  M←⍋⍒S[;2]⍳10⊥⍉ZL∘.,SP
[15]  M←0 2↓⍋⍒S[M;]
[16]  SP←(∧/FZ≠(,SM)[M])/SP          ⍝ Spalten sudoku-gesiebt
[17]
[18]  T3←0 9 9⍴⍬
[19]  →0⍳⍳0=⍴SP
[20]
[21]   L1:  M←SM ◇ M[ZL;⍬⍴SP]←FZ
[22]         T3←T3,FZ S4 M           ⍝ rekursiver Spalten-Loop
[23]  →L1⍳⍳ 0≠⍴SP←1↓SP
    ∇
```

```
    ∇ T←TEST SFOR S
[1]   ⍝ SPE 060904
[2]   ⍝ S-Matrix testen und formatieren
[3]
[4]   ⍎(0=⎕NC'TEST')/'TEST←0'        ⍝ Kein Test sondern nur formatieren
[5]
[6]   T←'Keine vollkommene Sudoku-Matrix'
[7]   ⍎(TEST=1)/'→0⍳⍳0∈45=(+/S),(+⌿S),+/S[9 9⍴⊃⍳9 3⍴⊂[2]27 3⍴(⍳9)∘.,⍳9]'
[8]
[9]   T←(19⍴0 1)⍀'|','BI2,< |>' ⎕FMT S
[10]
[11]  T[ 1;]←'
[12]  T[ 3;]←'
[13]  T[ 5;]←'
[14]  T[ 7;]←'
[15]  T[ 9;]←'
[16]  T[11;]←'
[17]  T[13;]←'
[18]  T[15;]←'
[19]  T[17;]←'
[20]  T[19;]←'
[21]
[22]  T[; 1]←'
[23]  T[;13]←'
[24]  T[;25]←'
[25]  T[;37]←'
    ∇
```

Abb. 1: Programm Listing von LOESE, S4 und SFOR

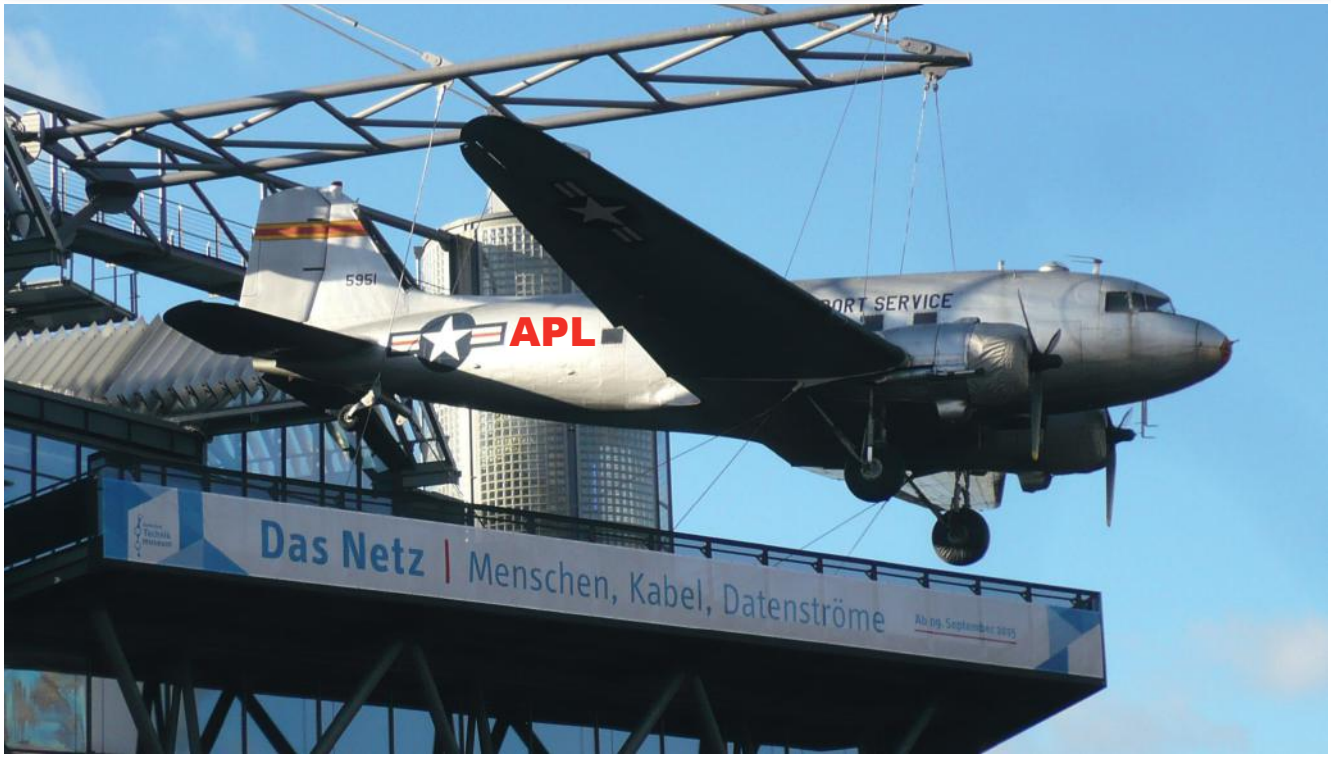
```
      LOESE SD23    A Löse sdk23 von APL-Journal 1/2/2014 (Seite 26)
2.19
```

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

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

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

Abb. 3: Demo-Aufruf von LOESE. Als Rechenzeit wurden 2.19 Sekunden ausgegeben.



APL-Journal
34. Jg. 2015, ISSN 1438-4531
Herausgeber: Dr. Reiner Nussbaum, APL-Germany e.V., Mannheim, http://www.apl-germany.de
Redaktion: Dipl.-Volksw. Martin Barghoorn (verantw.), FU Berlin, Zentraleinrichtung für Datenverarbeitung (ZEDAT), Fabeckstraße 32, 14195 Berlin, Tel. (030) 804 03 192
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: Umschlagseiten 1 und 4 sowie S. 59:
Martin Barghoorn

Beitragssätze (Jahresbeitrag)

Ordentliche Mitglieder:

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

Außerordentliche Mitglieder:

Jurist./natürl. Pers.	500,- EUR

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.,

eMail: reiner.nussbaum@t-online.de

2. Vorstandsvorsitzender:

Martin Barghoorn, FU Berlin, Zentraleinrichtung für Datenverarbeitung

(ZEDAT), Fabeckstraße 32, 14195 Berlin, eMail: Barghoorn@t-online.de

Schatzmeister

Jürgen Beckmann

Im Feudenheimer Grün 10

68259 Mannheim

Tel. 0621 7 98 08 40,

eMail: JBecki@onlinehome.de

Bankverbindung

APL-Germany e.V.

BVB Volksbank eG Bad Vilbel, DE51 5186 1325 0005 2326 94

BIC: GENODEF1BVB

Hinweis:

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

NEWS: Neues@APL-Germany.com

www.apl-germany.de

APL Germany e.V.