APL
one of the greatest programming languages ever

Bernd Ulmann
ulmann@vaxman.de

Vintage Computer Festival Europe 2007
29th April – 1st May 2007
Munich

Kenneth E. Iverson, 17.12.1920 – 19.20.2004

The picture on the right is the last picture of Ken Iverson (with Rachel Hui, photo by Stella Hui) – four weeks before his death.

Who was Kenneth E. Iversion?

- He was a mathematician by profession.
- He loved to explore new ways of thinking. As Linda Alvord[1] once said: "*Meeting Ken Iversion could cause mental transformations.*"
- Kenneth E. Iverson had a special interest in mathematical notation – a good example is his seminal paper *Notation as a Tool of Thought* (cf. [5]).
- He found the standard notation awkward and inconsistent and thus developed a new system in 1957 while being an assistant professor in Harvard (cf. [8][pp. 1]).

[1] http://www.vector.org.uk/archive/v223/alvord.htm

The birth of APL

- In 1960 Ken E. Iverson began working for IBM where he met Adin Falkoff who became interested in Ken's new notation.
- He extended his notation to a degree which made it possible to be used for the description of systems and algorithms, see [6] (eventually this notation was used to describe the complete IBM /360 architectur in 1964, cf. [7]).
- This language was internally known as *Iverson's Better Math* – IBM did not like this name for obvious reasons, thus Iverson was forced to come up with something else:

A Programming Language
APL for short.

The birth of APL

- This name was used officially for the first time in the title of the book [3] in 1962, where it was still used, despite the title of the book, as a method for "*interpersonal communication*" (cf. [8][p. 1]), not as a computer programming language.
- Since Iversons goal was the development of a consistent new system of notation for mathematics, he used *normal* mathematical symbols, i.e. rather weird symbols which makes typing in an APL programm a bit challenging.

To be honest, APL code looks like line noise to the uninformed. . .

The birth of APL

- R. A. Brooker once asked (cf. [9][p. 11]):
  "*Why do you insist on using a notation which is a nightmare for typist and compositor. . . ?*"
- The use of very special graphical symbols in APL programs was a severe problem in the 1960s since there were no graphic displays capable of displaying these symbols. IBM overcame this problem with the introduction of the IBM selectric typewrite which could use special character balls containing the APL character set.
- Despite this particular difficulty the constistency and efficiency of Ken E. Iverson's APL quickly led to the development of interpreters for this language.

The first APL systems

1963: Hellermann implements a subset of APL as an interpreter on the IBM 1620)[2] (cf. [8][p. 1]).

1965: M. L. Breed and P. S. Abrams developed an APL implementation on an IBM 7090 running in a batch environment (cf. [8][p. 1]).

1966: Breed and Abrams create a new APL system running under an experimental time sharing system on the IBM 7090. This was the first time an APL system was used interactively as it is common today (cf. [8][p. 1]).

[2] This system was known as *CADET* – short for *C*an't *A*dd *D*oesn't *E*ven *T*ry, since the machine had no real ALU but relied on lookup tables for its operations.

APL basics

- APL is in its very nature an interpreter language and supports interactivity to a uniquely large degree.
- APL does not care too much about data types – something which was adopted in more recent languages like Perl, Python, etc.
- The basic datastructure of APL is the vector.
- APL programs make seldom use of loops, conditional execution and the like as everything is transformed into vector/matrix operations.
- APL programs tend to be incredibly short in comparison with implementations using other languages. One-liners are not uncommon.

Calculating $\sum_{i=1}^{100} i$

To give an impression of the power of APL consider the following problem: *Calculate the sum of all integers ranging from 1 to 100.* A simple C-solution might look like this:

sum.c

```
#include <stdio.h>
int main()
{
  int i, sum = 0;
  for (i = 1; i <= 100; sum += i++);
  printf("Sum:  %d\n", sum);
  return 0;
}
```

Calculating $\sum_{i=1}^{100} i$

Using APL one would transform the original problem to one requiring the calculation of the inner sum of a vector containing 100 successive integer elements ranging from 1 to 100.
The complete APL solution for calculating

$$\sum_{i=1}^{100} i$$

looks like this:

APL solution

$$+/\iota 100$$

Explanation

This program consists of two parts:

1. The generation of a vector containing 100 successive integer elements ranging from 1 to 100. This is done by using the operator ι which takes an integer argument and returns a vector of the structure described above, so $\iota 100$ yields

$$(1, 2, 3, 4, 5, 6, 7, \ldots, 98, 99, 100)$$

2. Calculating the inner sum of these vector elements: The heart of this operation is the so called *reduction operator* "/" which expects two arguments:
   1. A vector on its right side and
   2. an operand on its left side.

Explanation

- The reduction operator places the operator found on its left hand side between every two successive elements of the vector written on its right hand side.
- In the current example we have

$$+/\iota 100 = (1 + 2 + 3 + 4 + \cdots + 98 + 99 + 100)$$

  which obviously equals $\sum_{i=1}^{100} i$ and thus solves the initial problem.
- Did you see that the APL expression was evaluated from right to left? This seems quite odd, but has a reason...

Right to left evaluation

- Ken Iverson always thought of the traditional left-to-right evaluation with its idiosyncrasies of operator precedence and parentheses as a kludge.
- He preferred a strict right-to-left evaluation with no or only a few operator precedence rules. This feels odd at first but has real advantages over the traditional style of writing equations and algorithms as he shows in [4] on a polynomial evaluation using the Horner technique:
  - The conventional solution looks like this:

$$(a, b, c, d, e)\prod(x) = a + x \times (b + x \times (c + x \times (d + x \times e)))$$

while a strict right to left evaluation rule yields

$$(a, b, c, d, e)\prod(x) = a + x \times b + x \times c + x \times d + x \times e.$$

Right to left evaluation

- Every APL expression is thus evaluated from right to left!
- So $3 \times 4 + 5$ yields 27 using APL, not 17 as one might think!
- This makes the use of parenthesis in many cases superfluous which adds to the expressional power of APL.

Generate a list of prime numbers

- The following example is far more complicated than the simple calculation of a sum shown before. The goal is to print a list of all prime numbers ranging from 2 to a given number `R`.
- Using a conventional language like C a solution based on the sieve of Eratosthenes could look like this (this example is indeed a bit obfuscated, but I tried to solve the problem with as few lines of code as possible without trying too hard):

Generate a list of prime numbers

eratosthenes.c

```
#include <stdio.h>

#define R 100

int main()
{
  int i, j, v[R + 1];

  for (i = 2; i <= R; v[i++] = 1);
  for (i = 2; i * i <= R; i++)
    for (j = 2; v[i] && i * j <= R; v[i * j++]= 0);

  for (i = 2; i <= R; i++)
    if (v[i]) printf("%d ", i);

  return 0;

}
```

Generate a list of prime numbers

Solving this problem using APL results in a much shorter and more elegant program:

APL solution

$$(\sim \mathtt{R} \in \mathtt{R} \circ . \times \mathtt{R})/\mathtt{R} \leftarrow 1 \downarrow \iota \mathtt{R}$$

Explanation

How does this work?

- Create a vector of the form $(2, \ldots, R)$ and save it in R (again) by $\mathtt{R} \leftarrow 1 \downarrow \iota\mathtt{R}$.
- Create a matrix of the form

$$\begin{pmatrix} 4 & 6 & 8 & 10 & \ldots \\ 6 & 9 & 12 & 15 & \ldots \\ 8 & 12 & 16 & 20 & \ldots \\ 10 & 15 & 20 & 25 & \ldots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix}$$

by creating the outer product of the vector R like this: $\mathtt{R} \circ . \times \mathtt{R}$. This matrix contains everything but prime numbers!

Explanation

- Now create a bit vector with elements corresponding to the elements of R, containing a 1 for every value being prime (thus not being an element of the matrix above): $(\sim \mathtt{R} \in \mathtt{R} \circ . \times \mathtt{R})$.
- Finally, use this bit vector to select all prime elements from R using the selection operator $/$.

Performing a fast Fourier transformation (FFT)

```
     ∇ Z←FFT X;C;D;E;J;K;LL;M;N;O
[1]    LL←⌊2*-O-⍳M←⌊2⊛N,0⍴E←1-2×~O←⍳1.J←⍳L
       ←0,0⍴K←⍳N←¯1↑⍴X
[2]    →(M>L←L+1)/1+⍴⍴J←J,N⍴ 0 1 ∘.=(
       2*L)⍴1
[3]    Z←X[;(L←0)+(⌽LL)+.×J←(M,N)⍴J]
[4]    X← 2 1 ∘.○○(-O-K)÷¯1↑LL
[5]    Z←Z[;K-,LL[L]×J[L;]]+(⍴Z)⍴(-⌿X[;D]×
       Z[;C]),+⌿X[;D←O+N⍴LL[E+M-L]×-O-⍳
       2×LL[L]]×⊖Z[;C←K+,LL[L]×0=J[L;]]
[6]    →((M+O)>L←L+1)/5
     ∇
```

How does this work? Do not even ask! (Have a look into [2][p. 212].)

Readability ond maintainability of programs

- Being able to write APL programs does not normally imply that you are able to read APL programs – not even your own!
- This has APL earned the notation of being a *write only language*.
- APL encourages the implementation of short and powerful routines, on the other hand, which simplifies maintainance of existing APL programs.
- Getting used to the APL way of thought is hard at the beginning – especially if one is used to program in a procedural or object oriented style, but due to the expressive power of APL it is worth to have a deeper look into this language.
- One can even take advantage from the APL way of thought when using other programming languages or even when using APL as a system of notation.

APL machines

- As with LISP there were some attempts to build APL machines, i.e. implement processors with special features to speed up the execution of APL programs.
- One of the earliest attempts was an APL implementation in microcode (real programmers write microcode – Assembler is a high level language) for the IBM S/360 model 25 (described by Hassit).
- A real hardware implementation was the (commercially unsuccessful) Control Data STAR-100 vector processor shown in the next slide (a *real* computer – LISP machines are quite boring compared to that :-)).

APL machines

The future of APL

APL is far from being dead!

- There even is a recent APL implementation available for free and for most common platforms (unfortunately not for OpenVMS) from Morgan Stanley:
  `http://www.aplusdev.org`
- Ken Iverson's last brainchild, called **J**, is available for a lot of platforms, too (even OpenVMS – ported by me :-)) – it is even more weird than APL since it does not need any special symbols but (ab)uses everything available in ASCII in very inventive ways (cf. `http://www.jsoftware.com`).


[1], *A Source Book in APL*, APL PRESS, Palo Alto, 1981


[2], Wolfgang K. Giloi, "Programmieren in APL", deGruyter, Berlin, 1977


[3], Kenneth E. Iverson, *A Programming Language*, J. Wiley & Sons, New York, 1962


[4], Ken E. Iverson, "Conventions Governing Order of Evaluation", in [1][pp. 29–32]


[5], Kenneth E. Iverson, "Notation as a Tool of Thought", in [1][pp. 105–128]


[6], K. E. Iverson, "Programming notation in systems design", in *IBM Systems Journal*, June 1963, pp. 117–128

 [7], A. D. Falkoff, K. E. Iverson, E. H. Sussenguth, "A formal description of SYSTEM/360", in *IBM Systems Journal*, Vol 3, No. 3, 1964, pp. 198–261

 [8], Wolfgang H. Janko, *APL 1 – Eine Einfuehrung in die Elemente der Sprache und des Systems*, Athenaeum Verlag, Koenigstein/Ts., 1980

 [9], Eugene E. McDonnell, "Introduction", in [1][p. 11–14]