Array Oriented Functional Programming in APL

Morten Kromberg
CTO, Dyalog Ltd.

ACM Poughkeepsie,
Marist College, September 17th, 2019

Paradigms

- **Object Orientation**
  - Modeling of queues / events
  - Simula (1962)
  - Smalltalk (1980)
  - Java (1991), C# (2000)
- **Functional Programming**
  - Lambda Calculus (1936)
  - Lisp (1960)
  - Haskell (1992)
- **Array Oriented Programming**
  - Applied Mathematics (1950's)
  - IBM APL\360 (1966)
  - Dyalog APL (1983)
  - J (1990), k (1993)

Introduction to
A Programming Language
Kenneth E. Iverson (1962)

Applied mathematics is largely concerned with the design and analysis of explicit procedures for calculating the exact or approximate values of various functions.

Such explicit procedures are called algorithms or programs.

*Because an effective notation for the description of programs exhibits considerable syntactic structure, it is called **a programming language**.*

A Programming Language

$a\ b$

$Mat1 \cdot Mat2$

$f\ g\ x$

e^x

$\frac{x}{y}$

$\log_b a$

$\sqrt[n]{a}$

$\tan^2 x$

$\sum_{n=1}^{6} 4n$

$\prod_{i=1}^{6} 4i$

Problems:

- Wide variety of syntactical forms
- Strange and inconsistent precedence rules
- Things get worse when you deal with matrices

See http://www.jsoftware.com/papers/EvalOrder.htm

A Programming Language

$a\ b$

```
a×b
```

$\frac{x}{y}$

```
*x
```

e^x

```
x÷y
```

$\log_b a$

```
b⍟a
```

$\sqrt[n]{a}$

```
a*÷n
```

$Mat1 \cdot Mat2$

```
Mat1 +.× Mat2
```

$f\ g\ x$

```
f g x
```

$\tan^2 x$

```
(3○x)*2
```

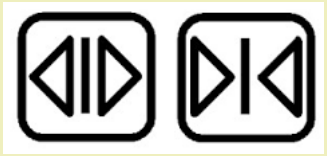
$\sum_{n=1}^{6} 4n$

```
+/4×⍳6
```

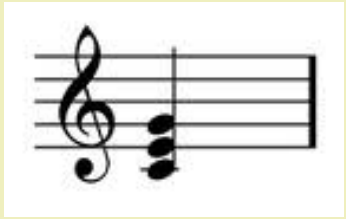
$\prod_{i=1}^{6} 4i$

```
×/4×⍳6
```

Symbols



Primitive APL functions are denoted by [Unicode] symbols

- We learn and use new symbols all the time
- Symbols help us to communicate more concisely
- Suppose I asked you to simultaneously play three notes with frequencies of 130.81, 155.56, and 196 Hertz...
-  is a more concise way to convey that information but only if you know what the symbols mean

- Sum all the elements of X

$$\sum_{i=1}^{n} X_i \qquad +/X$$

- How many combinations of k things can you make from n things?

$$\frac{n!}{k!\,(n-k)!} \qquad k\,!\,n$$

APL Symbols

- APL uses about 75 symbols
- You already know many of them

Arithmetic	+	−	×	÷	|	
Comparisons	<	≤	=	≠	≥	>
Set Operations	∩	∪	~	∈		
Logic	∧	∨	⍲	⍱	~	

- Many symbols are graphically indicative of their meaning

  ⌽ ⊖ ⍉ ≡ ≢ ⍋ ⍒ ⊥ ⌈ ⌊

- Others are mnemonically indicative

  ⍴ Rho – Reshape

  ⍳ Iota – Index

- Many symbols are complementary

  ⊂ ⊃ ↑ ↓ ⍋ ⍒ ⊥ ⊤ ⌈ ⌊

Comments

The ***lamp*** symbol (⍝) indicates the beginning of a comment (comments ***illuminate*** code).

```
      2×3 ⍝ two times three
6
```

Array Oriented Programming

Algorithms based on processing entire arrays
rather than individual elements one at a time

Example: Count the scores that are greater than or equal to 65

Pseudocode

```
count = 0
for score in scores
  if score ge 65
    count += 1
  endif
endfor
```

Dyalog APL

```
count ← 0
:For score :In scores
    :If score ≥ 65
        count +← 1
    :EndIf
:EndFor
```

Array Oriented Programming

Algorithms based on processing entire arrays
rather than individual elements one at a time

Example: Count the scores that are greater than or equal to 65

```
      scores←20 78 90 56 83
      +/ scores ≥ 65
3
      scores ≥ 65 ⍝ 1 for True, 0 for False
0 1 1 0 1

      +/ 0 1 1 0 1 ⍝ Sum = count occurrences of True
3
```

Scalar Functions

Operate on all the (scalar) items of arguments:

```
      1 2 3 + 4 5 6   ⍝ + addition
5 7 9
      10 20 30 > 25   ⍝ > greater than
0 0 1
      1 2 3 ⌈ 3 2 1   ⍝ ⌈ maximum
3 2 3
      ? 6 6 6 6       ⍝ ? roll (as in dice)
4 3 5 4
```

Indexing is also an array operation – but ***not*** a scalar function:

```
      'MISP'[1 2 3 3 2 3 3 2 4 4 2]
MISSISSIPPI
```

Operators

- In APL, an ***operator*** is a "higher order function" which takes a function as an ***operand*** and returns a ***derived function***

- Example: `+ /`

  (`+` is the *plus function* and `/` is the *reduction operator*)

- *plus reduction* is a function which computes the sum of the items in the right argument by inserting a `+` between items:

```
      +/ 1 2 3 4
→     1 + 2 + 3 + 4
→     10
```

Reduction (f/ or f⌿ or f/[n])

```
mat ← 3 4 ⍴ ⍳12
```

					+/mat	×/[2]mat
	1	2	3	4	10	24
	5	6	7	8	26	1680
	9	10	11	12	42	11880
×⌿mat	45	120	231	384		
×/[1]mat						

Scan (f\ or f⌿ or f\[n])

```
mat ← 3 4 ⍴ ⍳12
```

`+\mat`

1	2	3	4
5	6	7	8
9	10	11	12

→

1	3	6	10
5	11	18	26
9	19	30	42

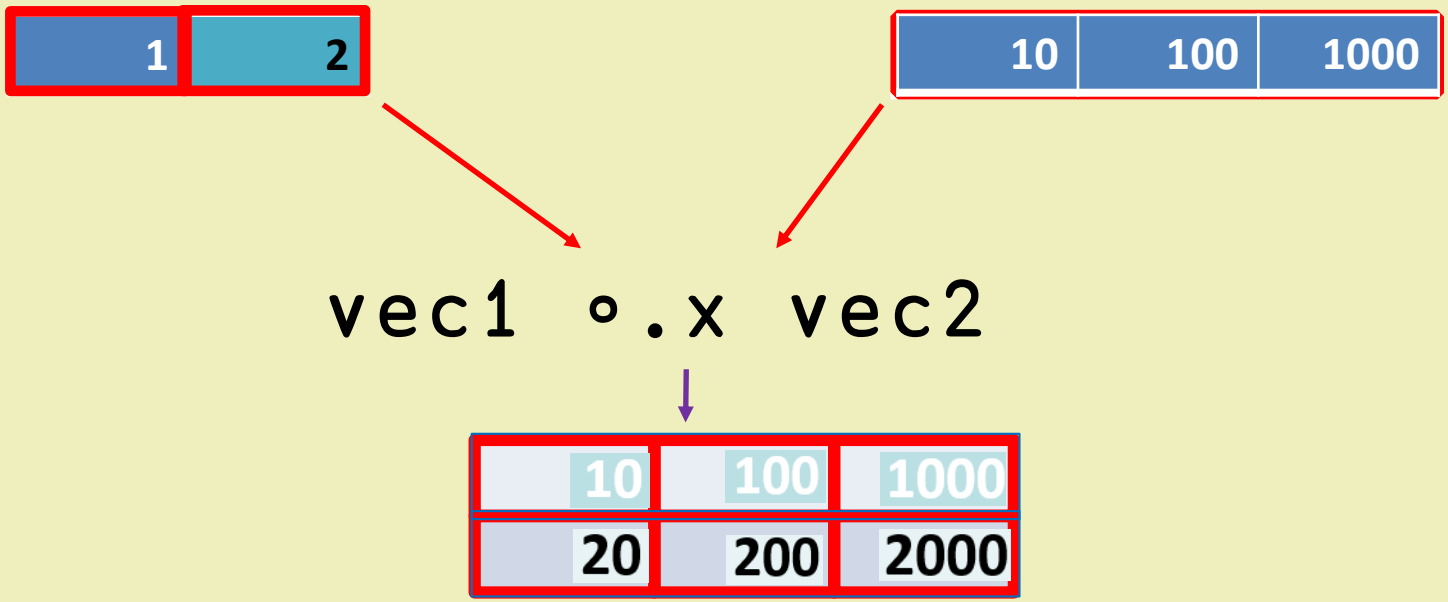
↓ ↓ ↓ ↓

`+⌿mat`

1	2	3	4
6	8	10	12
15	18	21	24

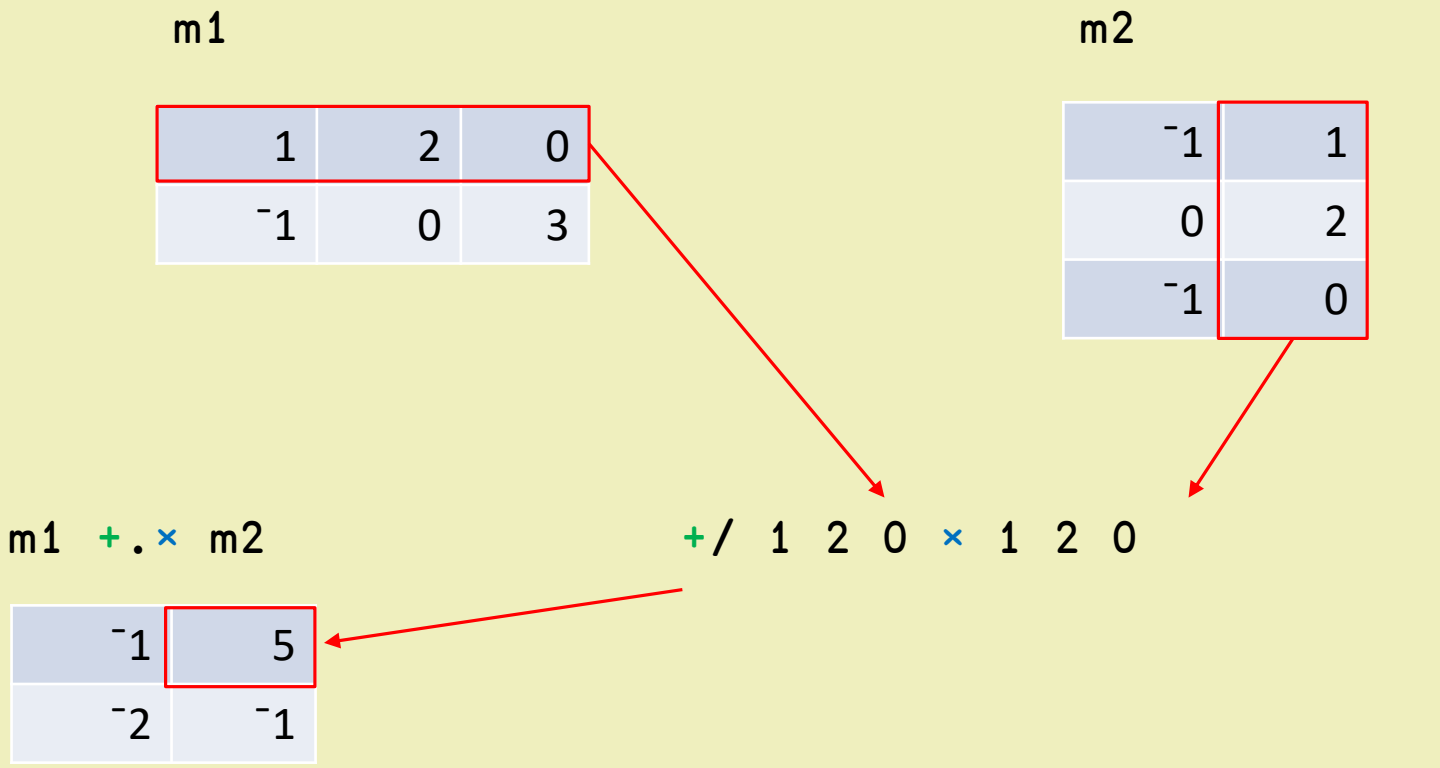
Outer Product (∘.f)

α f ω for all combinations of items from left & right arguments



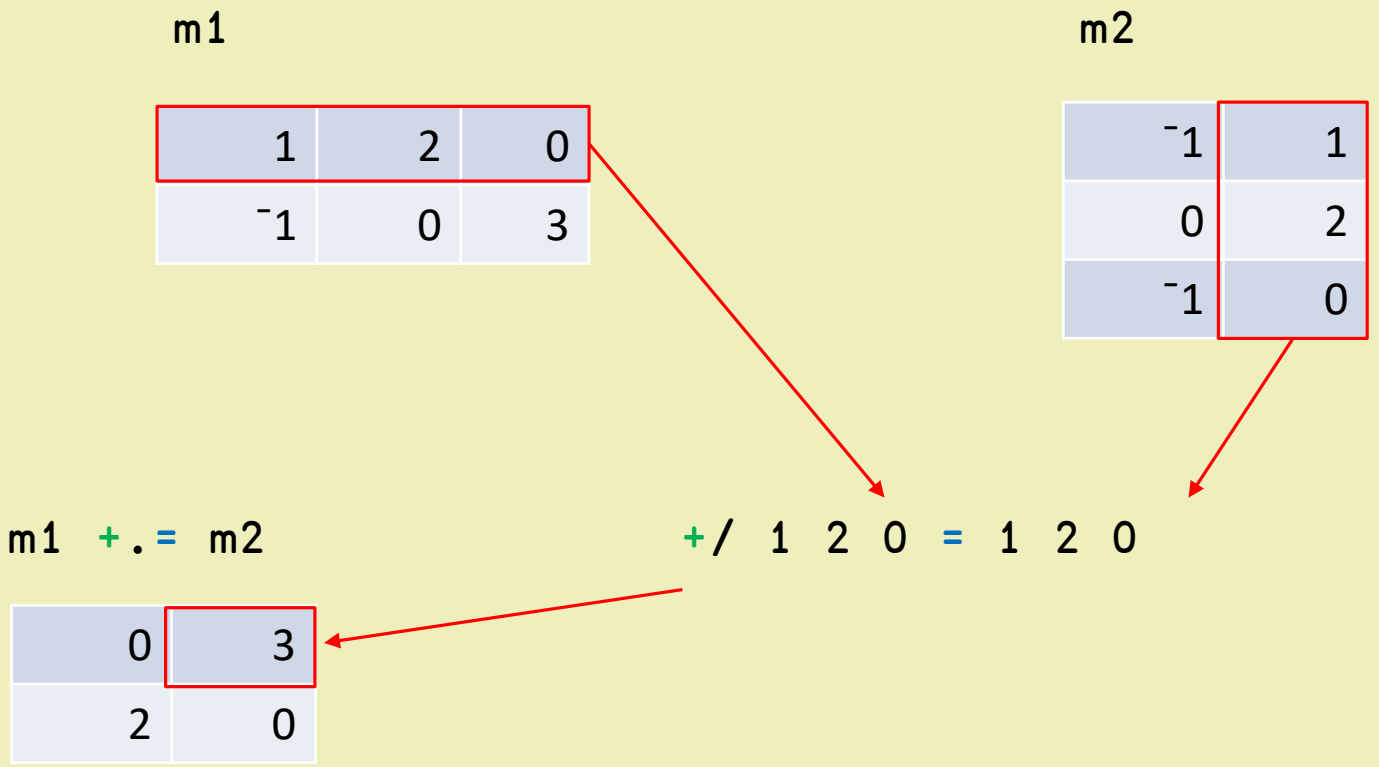
Inner Product (f . g)

`f/ row g col` for all combinations of rows left, cols right.



(+.× is vector or matrix product)

Inner Product (f . g)



Inner Product (f . g)

m1

1	2	0
¯1	0	3

m2

¯1	1
0	2
¯1	0

```
m1 ∧.= m2
```

0	1
0	0

```
∧/ 1 2 0 = 1 2 0
```

(∧.= left row identical to right col)

Summary: Syntax of APL

Syntactical Form	Example	Result
array	1 3.1415 1.2E18	
function argument	⍳ 6	1 2 3 4 5 6
left-arg **function** right-arg	1 2 3 × 1 10 100	1 20 300
operand **operator**	×/ 1 2 3 4 5 6 2 +/ 1 2 3 4 5 6	720 3 5 7 9 11
left-opnd **operator** right-opnd	1 0 2 +.× 1 2 3	7
array[indices]	'ABCDEF'[2 5 5 6]	BEEF

Mixed Functions

- ***Scalar*** functions operate on pairs of items taken from arguments
- ***Mixed*** functions operate on arrays but may combine items differently
- Many ***mixed*** functions operate on the structure of data, rather than perform computations

Rotate ϕ and ⊖

```
mat ← 3 4 ⍴ ⍳12
```

1	2	3	4
5	6	7	8
9	10	11	12

```
2 0 ¯1⌽mat
```

3	4	1	2
5	6	7	8
12	9	10	11

```
0 1 0 ¯1⊖mat
```

1	6	3	12
5	10	7	4
9	2	11	8

Examples of "Mixed" functions

IndexOf ⍳

```
      3 1 4 1 5 9 ⍳ 1 2 3
2 7 1
```

Grade (Up) ⍋

```
      digits←3 1 4 1 5 9
      ⍋digits
2 4 1 3 5 6
      digits[⍋digits] ⍝ index by grade
1 1 3 4 5 9
```

Membership ∈

```
      'HELLO WORLD'∈'AEIOU'
0 1 0 0 1 0 0 1 0 0 0
```

Reshape ⍴

```
mat ← 3 4 ⍴ ⍳12
```

1	2	3	4
5	6	7	8
9	10	11	12

```
2 2⍴mat
```

1	2
3	4

Take ↑ and Drop ↓

```
mat ← 3 4 ⍴ ⍳12
```

1	2	3	4
5	6	7	8
9	10	11	12

```
¯2 2↑mat
```

5	6
9	10

```
1↓mat
```

5	6	7	8
9	10	11	12

Transpose ⍉

```
mat ← 3 4 ⍴ ⍳12
```

1	2	3	4
5	6	7	8
9	10	11	12

```
2 1⍉mat
```

```
⍉mat
```

1	5	9
2	6	10
3	7	11
4	8	12

```
1 1⍉mat
```

1	6	11

Branch Free Logic

The fact that *map* is implicit, and *indexing* can be done using arrays, encourages logic without branches. The data itself provides flow control:

Example	Pseudo Code

`scores←20 78 90 56 83`
`+/ scores ≥ 65`
`3`
	for score in scores
	if score ge 65 count += 1

`data←2 7 15 60`
`data ⌈ 10`
`10 10 15 60`
	if data[i]>10
	then data[i] else 10

`data + data ∊ 3 7 15` `2 8 16 60`	Increment where data[i] is in [3,7,15].
`(x × mask) + y × ~mask`	If mask[i] then x[i] else y[i]

`phase←'child' 'young' '20s' 'old'`
`phase[1⌈4⌊ ⌈data÷10]`
`child child young old`
	“bucketing”

Order of Execution

No "precedence", only one rule: as `f g x` in mathematics

```
f g x ↔ f(g(x))
```

Each function takes as its right argument the result of the entire expression to its right.

If it needs a left argument, take the value immediately to the left.

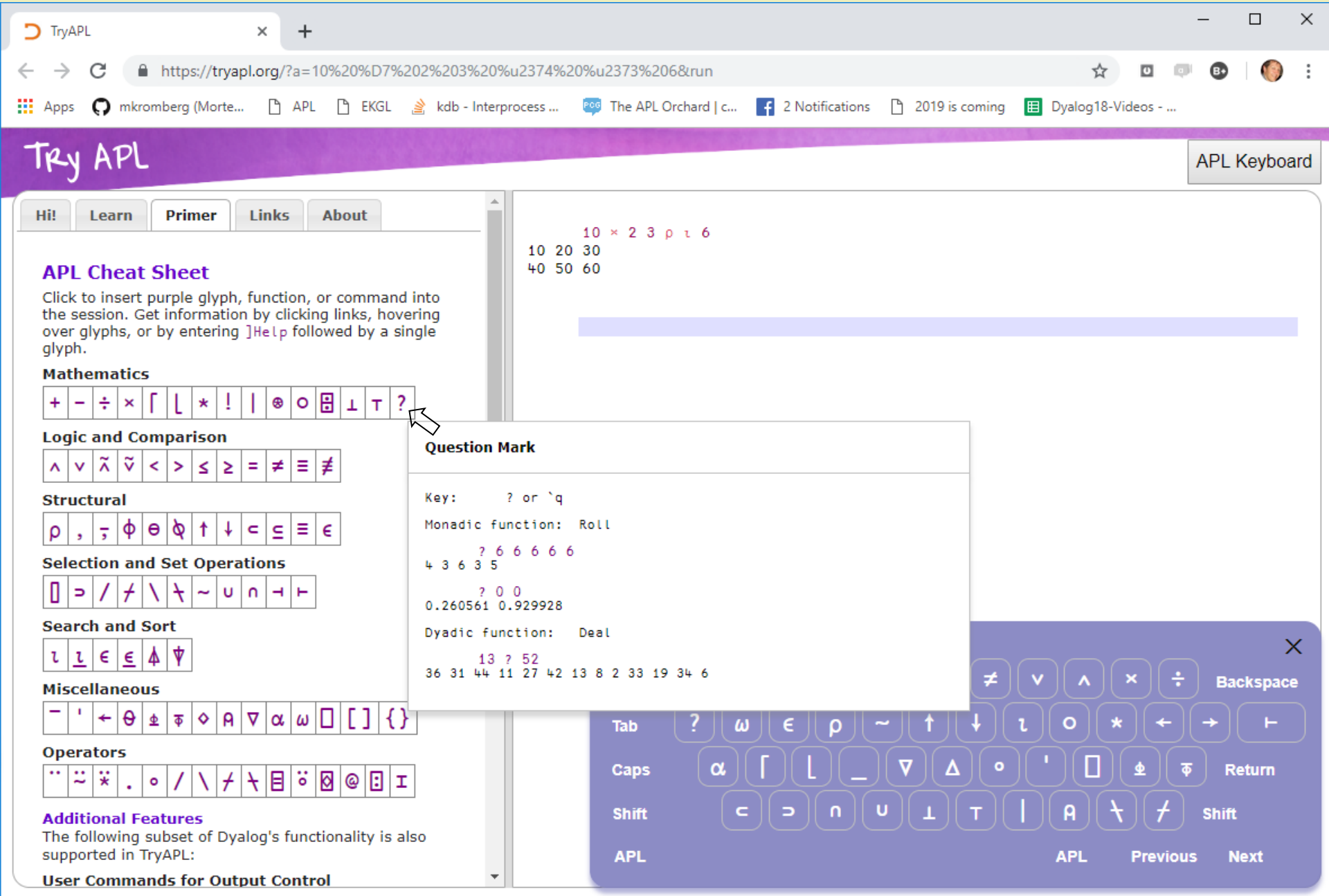
Good APL can be read from left to right by an experienced programmer, but as a beginner you may need to break it down.

```
10 × 2 3 ⍴ ⍳ 6
```

"Ten times [the] two three reshape [of the] first natural numbers."

NB: You can experiment at https://tryapl.org:

https://tryapl.org/?a=10%20%D7%202%203%20%u2374%20%u2373%206&run


TryAPL
https://tryapl.org/?a=10%20%D7%202%203%20%u2374%20%u2373%206&run
Apps
mkromberg (Morte...
APL
EKGL
kdb - Interprocess ...
The APL Orchard | c...
2 Notifications
2019 is coming
Dyalog18-Videos - ...
Try APL
APL Keyboard
Hi!
Learn
Primer
Links
About
APL Cheat Sheet
Click to insert purple glyph, function, or command into the session. Get information by clicking links, hovering over glyphs, or by entering]Help followed by a single glyph.
Mathematics
Logic and Comparison
Structural
Selection and Set Operations
Search and Sort
Miscellaneous
Operators
Additional Features
The following subset of Dyalog's functionality is also supported in TryAPL:
User Commands for Output Control
10 × 2 3 ⍴ ⍳ 6
10 20 30
40 50 60
Question Mark
Key: ? or `q
Monadic function: Roll
? 6 6 6 6 6
4 3 6 3 5
? 0 0
0.260561 0.929928
Dyadic function: Deal
13 ? 52
36 31 44 11 27 42 13 8 2 33 19 34 6
Backspace
Tab
Caps
Return
Shift
Shift
APL
APL
Previous
Next

APL in 1970

Functions:

- Scalar functions (implicit map)

```
+-×÷|! *⊗○ ⌈⌊  ⍝ math
<≤=≥>≠        ⍝ comparisons
∧∨ ⍲⍱ ~       ⍝ logic
? (roll)      ⍝ random numbers
```

- Mixed functions

```
⍴,↑↓ ⍳∊⍷⍋⍒ ⌽⊖⍉ ⊤⊥
⌹ (matrix division)
? (deal)
```

Operators:

- Reduction: `/⌿`
- Scan: `\⍀`
- Outer Product: `∘.f` (f on all combinations)
- Inner Product: `f.g` (f / rows-left g cols-right)

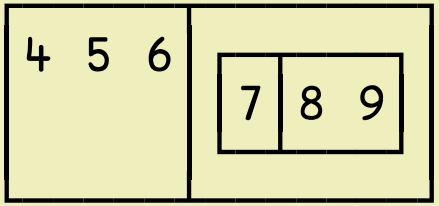
Driving Forces for Modernization

Feature	External Influence or Relation
Nested Arrays	Relational Databases / "Tuples" / Heterogenous Data
Control Structures	Structured Programming / Death of the GOTO
New Operators	Rationalization / New Insights into Array Notation
dfns	Functional Programming
Object Orientation	GUI + other interfaces (COM + Microsoft.NET)
Futures & Isolates	Multi-core CPUs and Cloud Computing

Nested Arrays

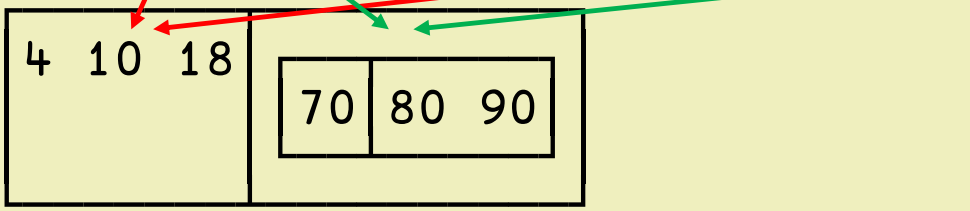
- Second Generation APL (ca 1983)
  - Each item of an array can be another array
  - Juxtapose arrays to create a list

```
(4 5 6)(7 (8 9))
```



- Scalar functions “pervade”:

```
(1 2 3) 10 × (4 5 6) (7 (8 9))
```



Each Operator (¨)

- Nested arrays demand a mechanism to apply functions to individual items
- The *Each* operator ¨ applies the operand function to each element:

```
      ⍴ (4 5 6) (7 8 9)  ⍝ shape of the array
2

      ⍴¨ (4 5 6) (7 8 9) ⍝ shape of each item
```

3	3

```
  1 5⍴¨ (4 5 6) (7 8 9) ⍝ reshape each item
```

4	7 8 9 7 8

Is APL "Functional"?

Wikipedia:

Functional programming is a programming paradigm—
a style of building the structure and elements of computer programs—
that ***treats computation as the evaluation of mathematical functions and avoids changing state and mutable data****.*

```
count ← 0
:For score :In scores
    :If score ≥ 65
        count +← 1
    :EndIf
:EndFor
```

```
+/ scores ≥ 65
```

Functional APL

APL encourages a functional style of problem-solving, but dynamically scoped APL allows procedural programming.

Since the 1990's, “dfns” provide a lexically scoped way to define functions which is a better foundation for "pure" functional definitions:

```
plusdouble←{α+2×ω} ⍝ left arg + two times right

fibonacci←{      ⍝ Tail-recursive Fibonacci.
     α←0 1       ⍝ Default left argument
     ω=0: ⊖⍴α    ⍝ If ω=0, return 1st item of α
     (1↓α,+/α) ∇ ω-1 ⍝ Tail recursion
 }
```

Syntax of APL w/ User Defined Fns

User-defined functions are infix or prefix, in exactly the same way as primitives.

Syntactical Form	Example	Result
array	**1 3.1415 1.2E18**	
function argument	⍳ 6 fibonacci 10	1 2 3 4 5 6 55
left-arg **function** right-arg	1 2 3 × 1 10 100 1 plusdouble 2 3	1 20 300 5 7
operand **operator**	×/ 1 2 3 4 5 6 2 +/ 1 2 3 4 5 6 fibonacci⍨ ⍳6 plusdouble/ 1 2 3	720 3 5 7 9 11 1 1 2 3 5 8 17
left-opnd **operator** right-opnd	1 0 2 +.× 1 2 3	7
array**[indices]**	'ABCDEF'[2 5 5 6]	BEEF

New Operators

1970:

- Reduction: `f/ f⌿`
- Scan: `f\ f⍀`
- Outer Product: `∘.f` (f on all combinations)
- Inner Product: `f.g` (f / rows-left g cols-right)

Selected new operators:

- Each: `f¨` (apply f to items)
- Power: `f⍣n` (apply f n times)
- Rank: `f⍤(l r)` (apply f to sub-arrays of given rank)

Examples of new Operators: Key (⌸)

Apply operand function to items corresponding to unique keys.

```
      keys← 'red' 'blue' 'red' 'red' 'blue'
      values← 10    20     30    40    50

      keys {⍺ ⍵}⌸ values   ⍝ Group by key
```

red	10 30 40
blue	20 50

```
      keys {⍺,+/⍵}⌸ values ⍝ Sum items by key
```

red	80
blue	70

Examples of new Operators: Stencil (⌺)

"Stencil" applies function operand to each item of an array – and selected neighbours:

```
      ⊢ 1 2 3 ⍝ ⊢ is "same": identity function
1 2 3
      (⊢ ⌺ 3) 1 2 3 ⍝ Window Size = 3
```

0	1	2
1	2	3
2	3	0

Neighbors Items

```
      ⍝ A one-dimensional "blur" stencil can be expressed:
      ({ .25 .5 .25 +.× ⍵} ⌺ 3) 0 10 0 0 10 0
2.5 5 2.5 2.5 5 2.5
```

John Conway's Game of Life

Computing the next generation:

- *Any live cell with fewer than two live neighbours dies, as if caused by under-population.*
- *Any live cell with two or three live neighbours lives on to the next generation.*
- *Any live cell with more than three live neighbours dies, as if by over-population.*
- *Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.*

```
life←{↑1 ⍵∨.∧3 4=+/,¯1 0 1∘.⊖¯1 0 1∘.⌽⊂⍵}
life←{↑1 ⍵∨.∧3 4=⊂{+/,⍵}⌺ 3 3⊢⍵}
```

Object Orientation

```
F←⎕NEW'Form' (('Caption' 'Hello World')
              ('Coord' 'Pixel')('Size'(100 400)))
F.(B1 B2 B3)←F.⎕NEW¨{'Button'
  (('Caption'('Button ',⍕⍵))('Posn'(5+50×0 ⍵)))}¨1 2 3
```


```
F.(B1 B2 B3).(Caption Posn)
```

Button 1 | 5 55	Button 2 | 5 105	Button 3 | 5 155

```
F.(B1 B2 B3).(Posn[1]←50)
```

OO: For Interfaces

```
XL←⎕NEW 'OleClient' (⊂'ClassName' 'Excel.Application')
cities←XL.Workbooks.Open ⊂'c:\...\cities.xls'
sheets←⎕cities.Sheets ⍝ Sheets collection as an array
sheets.Name
```

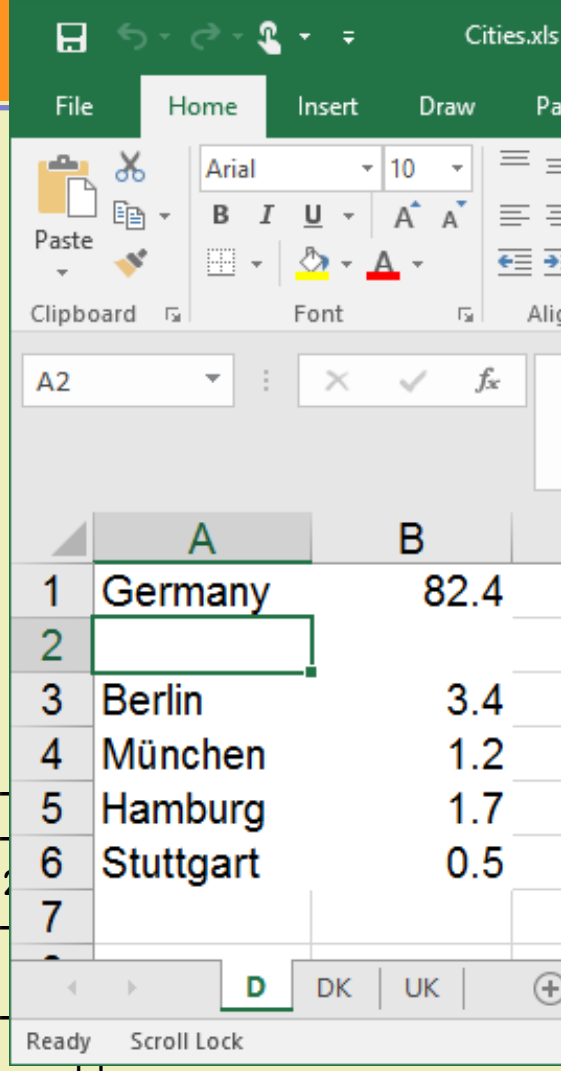
D	DK	UK

```
sheets.UsedRange.Value2
```

Germany	82.4
Berlin	3.4
München	1.2
Stuttgart	0.5

Denmark	5.4
Copenhagen	2
Helsingør	0.03

United Kingdom	60.1
London	7
Birmingham	1
Basingstoke	0.1



Arrays of Objects

- In most OO languages, everything is an object
- In Dyalog APL, arrays are a higher level of organization
  - Arrays can CONTAIN objects, but...
  - Arrays ARE not objects

- As a result, the "dot" is a parallel operation

```
array.member
```

... is a reference to member for each item of the array, not to a property of the array. For example, if sheets is a 3-element array of worksheet objects:

```
sheets.Name
```

D	DK	UK

Staying "Modern" for 50+ Years

Recap...

- **1982:** Nested Arrays (any item of an array can be an array) adopted by all APL vendors.
- **1995:** Control structures (if/then/else) adopted by most APL vendors.
- **1996:** Functional Programming: lexical scope and lambda expressions in APL ("dfns" – Dyalog).
- **2006:** Object orientation (Dyalog, MicroAPL, VisualAPL).
- **2014:** Point-free or "tacit" syntax (from J) adopted by first APL vendor (Dyalog)
- **2014:** Futures and isolates for asynchronous programming (Dyalog)

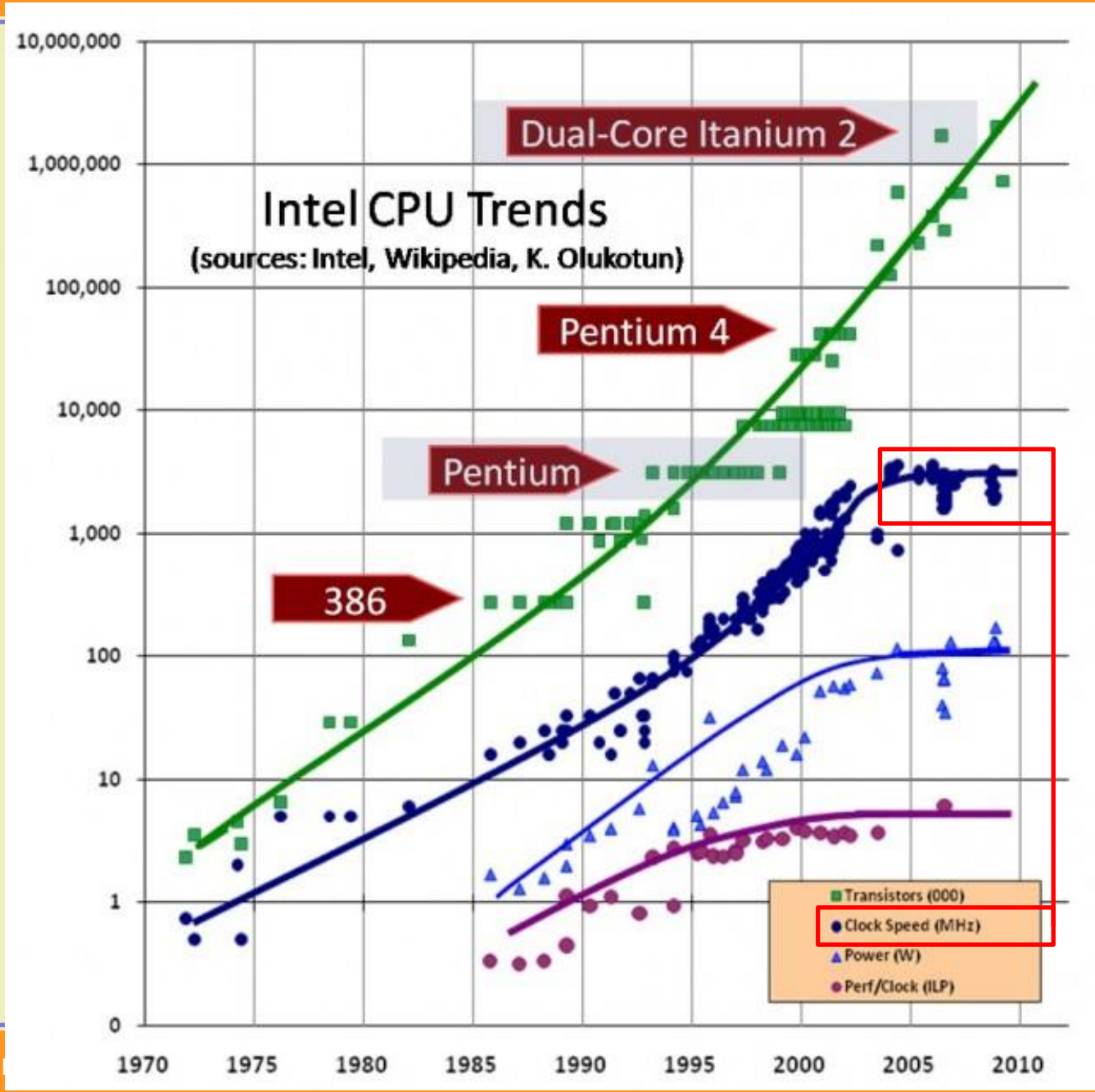
Still Modern after all those Years

"Dyalog is a modern, array-first,
multi-paradigm programming language,
which supports functional, object-oriented and
imperative programming,
based on an APL language kernel."

Why Modern Hardware Loves Arrays

- The design of APL was driven by human factors
  - Not the architecture of computer hardware
- Goals of APL Language design
  - Removes unnecessary detail from algorithms
  - Frees the mind to operate at a higher level
  - Allows math to be translated directly into code
- After 60-70 years, the hardware is also starting to appreciate arrays

The Free Lunch Is Over



CPU scaling showing transistor density, power consumption, and efficiency. Chart originally from [The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software]

The Power Problem

In the mid-2000s (at ~90nm) gates became too thin to prevent current from leaking out into the substrate. Clock speeds could no longer increase without excessive power consumption.

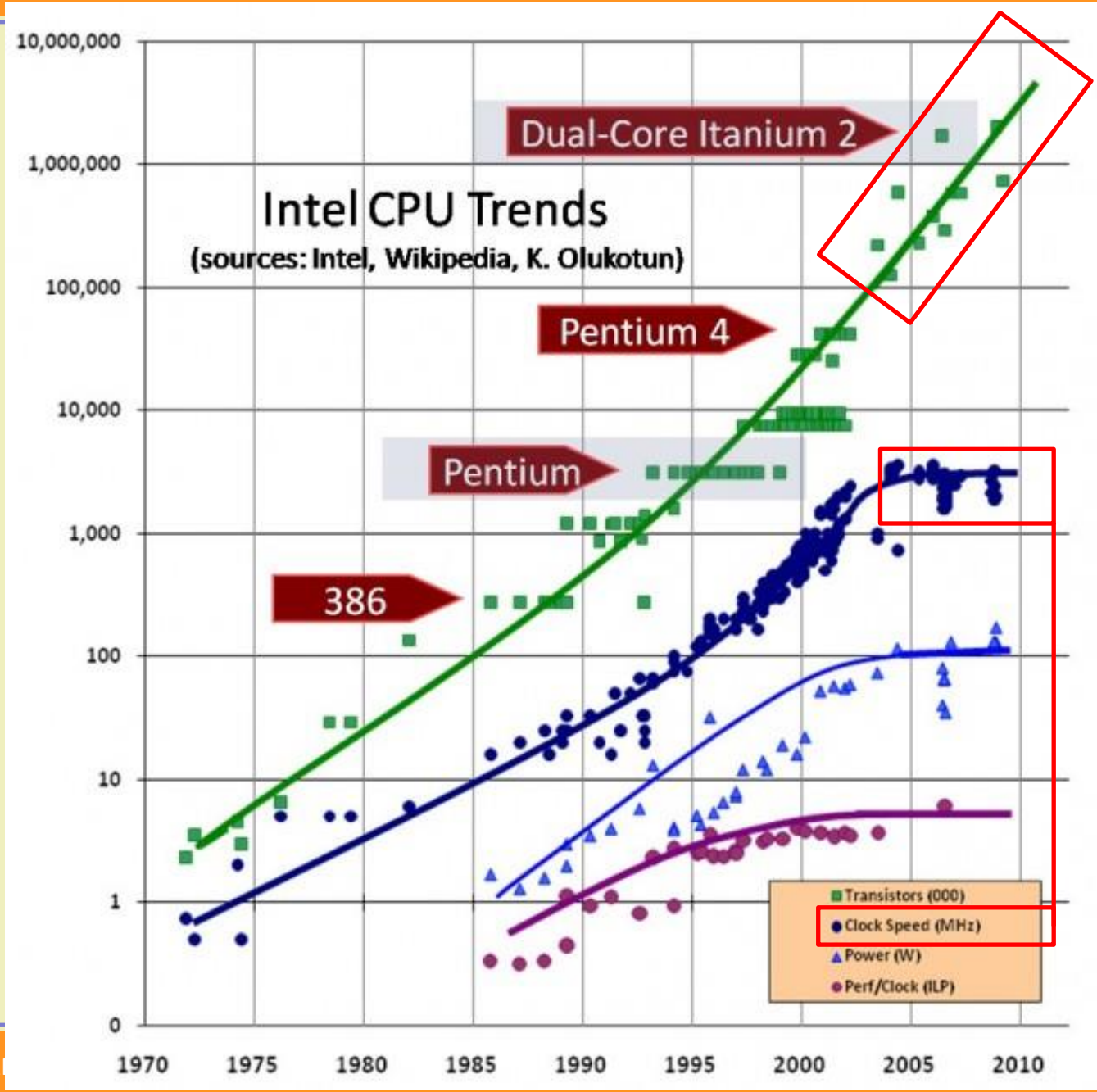
Innovation continues with technologies like: strained silicon, hi-k metal gate, FinFET, and FD-SOI have helped, but

- from 2007 to 2011, clock speeds rose 2.9GHz to 3.9GHz (33%)
- (from 1994 to 1998, CPU clock speeds rose by 300%)

Sources: http://www.extremetech.com/computing/116561-the-death-of-cpu-scaling-from-one-core-to-many-and-why-were-still-stuck

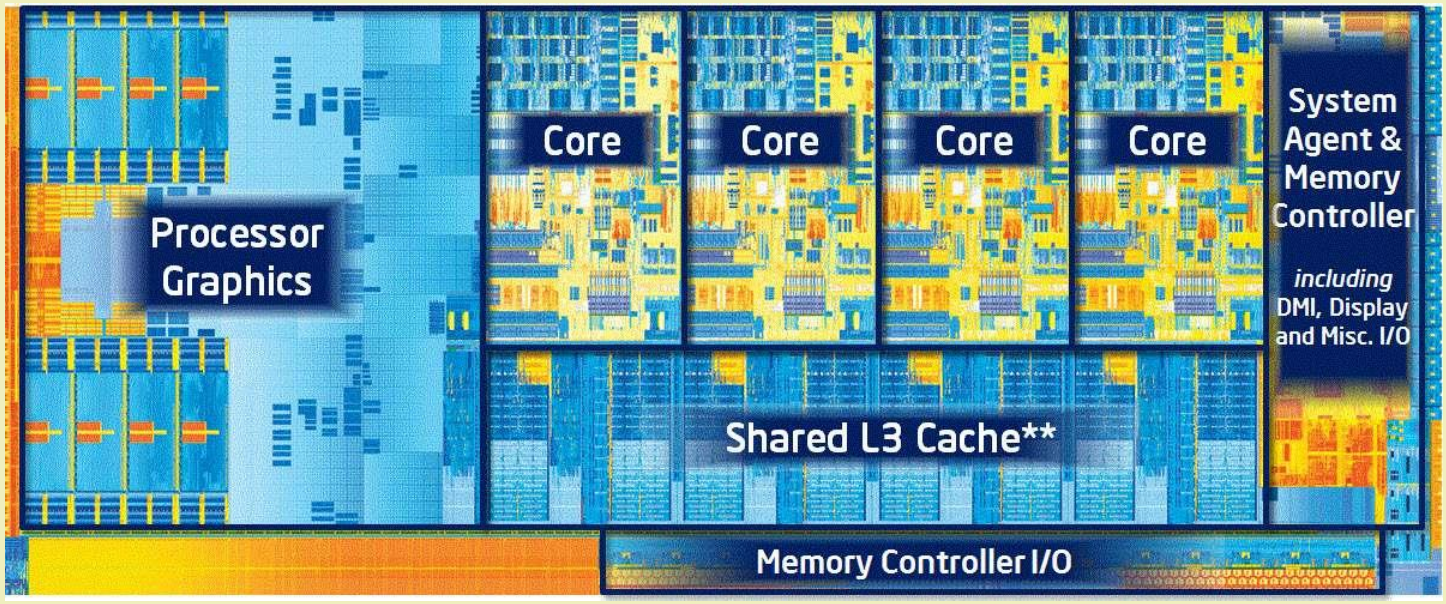
The Free Lunch Is Over

CPU scaling showing transistor density, power consumption, and efficiency. Chart originally from [The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software]



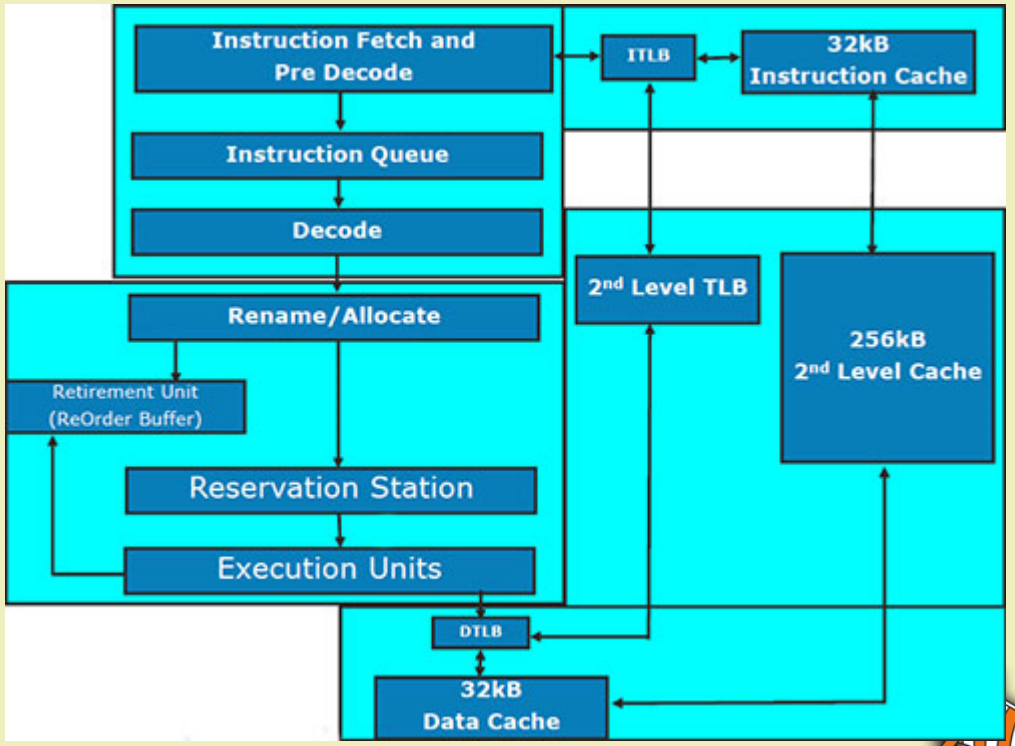
Transistor Count Still Increasing

- Cannot increase clock frequency without burning your lap and melting the North Pole
- Throttle back and do more with parallel cores



Making Use of Additional Transistors

- In addition to having parallel cores, use extra logic to do things in parallel WITHIN each core:
  - Prefetch memory into larger caches
  - Optimistically predict paths and pre-execute multiple instructions



Advantages of (APL) Arrays

Arrays are **DENSE**

- Aggressive conversion to smallest possible representations: **1-bit booleans**, 1-byte, 2-byte, 4-byte integers + 8-byte floats.
- Also 1, 2 and 4 byte Unicode characters
- ➢ Faster memory transfers
- ➢ Better cache utilization
- ➢ Vector instructions can handle many array elements at a time

Few **POINTERS**

- Sequential data access is enormously more efficient than "pointer chasing" (pointers also consume extra memory)
- ➢ More successful pre-fetching of data
- ➢ More successful branch prediction

Branch Free Logic

Decisions can be made using translate tables and vectorized computation.
Boolean arrays consume 1 bit per element (= 64 elements per instruction!).

Example	Pseudo Code

`scores←20 78 90 56 83`
`+/ scores ≥ 65`
`3`
	for score in scores
	if score ge 65 count += 1

`data←2 7 15 60`
`data ⌈ 5`
`5 7 15 60`
	if data[i]>5
	then data[i] else 5

`data + data ∊ 3 7 15` `2 8 16 60`	Increment where data[i] is in [3,7,15].
`(x × mask) + y × ~mask`	If mask[i] then x else y

`ages←'child' 'young' '20s' 'old'`
`ages[1⌈4⌊ ⌈data÷10]`
`child child young old`
	“bucketing”

Simple Example:
Replace CRLF with LF in a string

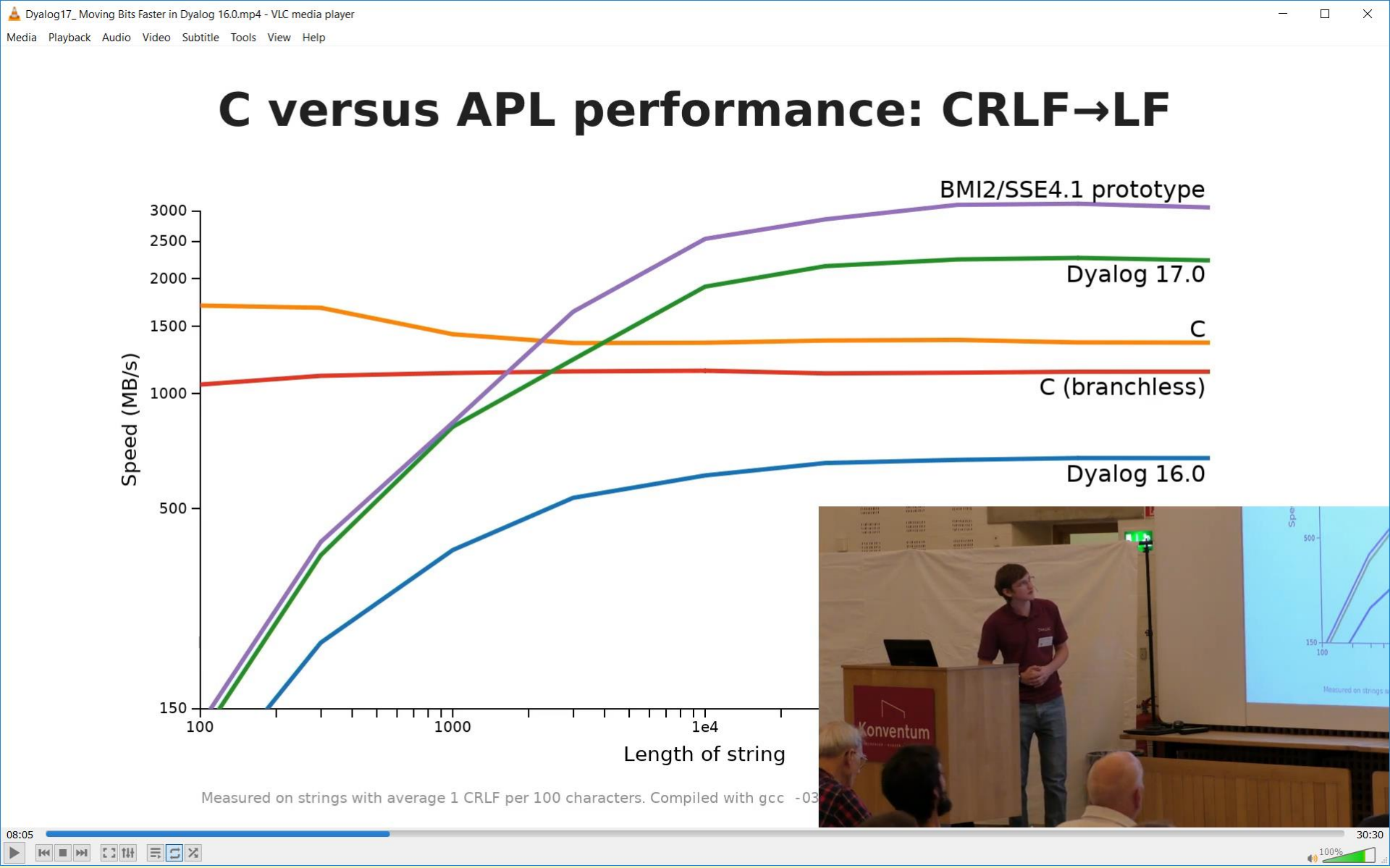
C

```
void crlf_to_lf(char* dst, char* src, size_t n)
{
        int was_cr = 0;
        for (size_t i=0; i<n; i++)
        {
                char c = src[i];
                if (was_cr && c=='\n') dst--;
                dst[i] = c;
                was_cr = (c=='\r');
        }
}
```

APL

```
CR LF ← ⎕UCS 13 10
crlf_to_lf ← {((⍵≠CR) ∨ 1⌽⍵≠LF) / ⍵}
```

For all chars: If char is not CR, or following char is not LF, keep the char



Marshall Lochbaum: Moving Bits Faster in Dyalog Version 16.0.
https://dyalog.tv/Dyalog17

Use Multicore Hardware

- Multithread primitives on large arrays
- Compile to Parallel Hardware
  - Aaron Hsu at Indiana University: co-dfns compiler
  - HyperFit / University of Copenhagen: APL to Futhark
  - Bernecky / Scholz @ Herriot Watt: Single Assignment C
- Introduce Asynchronous Features into Language
  - Futures and Isolates

[Arrays of] ***Futures***

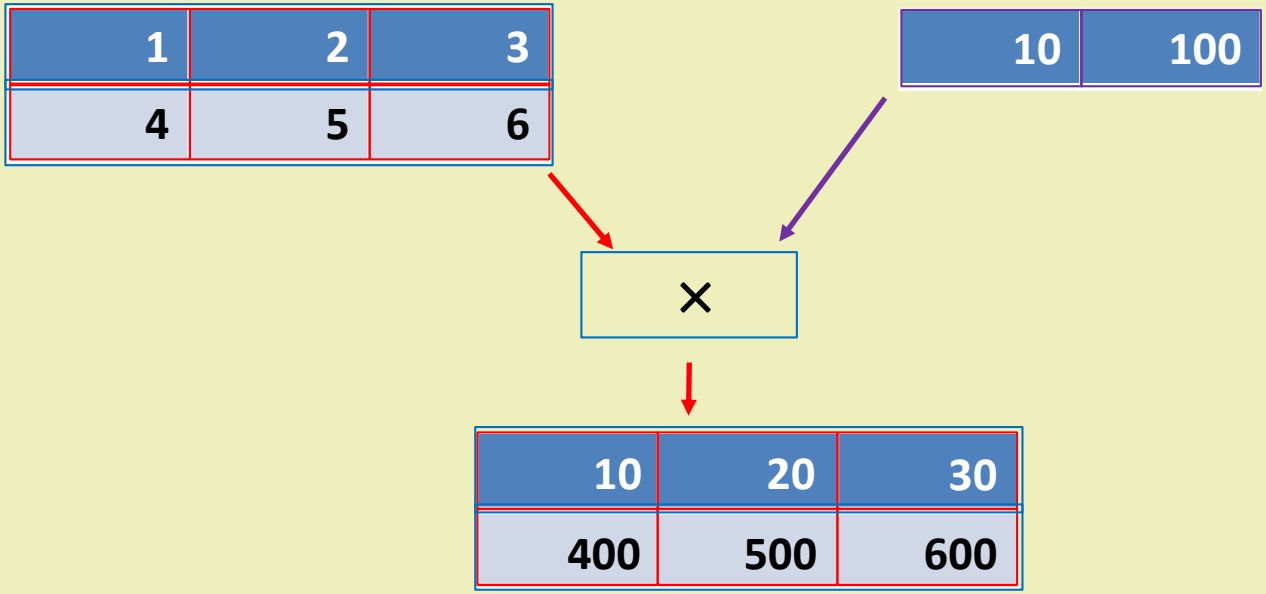
- Parallel (‖) is an operator which applies its left operand function ***asynchronously***
- A ***future*** is immediately returned, while the function runs inside an ***isolate*** (detached from the current scope)
- ***Arrays containing futures*** can be passed as arguments, and be subject to selection operations, without blocking
- The system ***blocks automatically*** until the future is resolved, if the value is required for futher computation
- The following expression computes three averages in parallel:

```
      ⎕←avgs←{(+⌿⍵)÷≢⍵}‖¨(2 3)(3 4 5)(9)
2.5 4 9
```

Rank Operator (1 of 2)

```
mat (x⍤1 0) vec
```

Multiplication with left rank 1 (vectors), right rank 0 (scalars)



Rank Operator (2 of 2)

```
mat ← 2 2 4 ⍴ ⍳16
```

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

```
((+⌿ ÷ ≢)⍤2) mat ⍝ avg cols
```

3	4	5	6
11	12	13	14

```
(+⌿ ÷ ≢) mat ⍝ average along 1st dimension
```

5	6	7	8
9	10	11	12

```
((+⌿ ÷ ≢)⍤1) mat ⍝ avg rows
```

2.5	6.5
10.5	14.5

Power Operator

- Apply a function a fixed number of times, or until the right operand function tells you to stop.

```
      2 (+ ⍤⍤ 3) 3 ⍝ Add 2 three times
9
      twice←⍤⍤2     ⍝ Bind one operand;
      2 +twice 3    ⍝   twice is a monadic opr
7
      ({1+÷⍵}⍤⍤≡)1 ⍝ … until fⁿ ≡ fⁿ⁻¹
1.618033989
```

Why Learn APL?

- APL is a powerful tool for data exploration and "algorithm mining"
- Understanding the Array Paradigm is useful even when coding in other languages
- Right now, a number of companies using APL are looking for new APL developers to replace team members who have been writing APL since 1980 (or thereabouts)

For example, Dyalog has been hiring...

Documentation, Samples, Webinars:

- Richard Park joined us late last year

United States APL Consulting Team

- Josh David
- Nathan Rogers

Interpreter Development

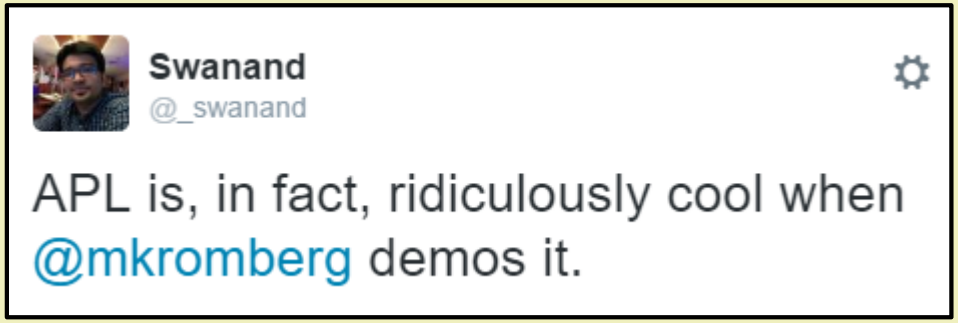
- We aim to have a new C developer on board before the end of 2019

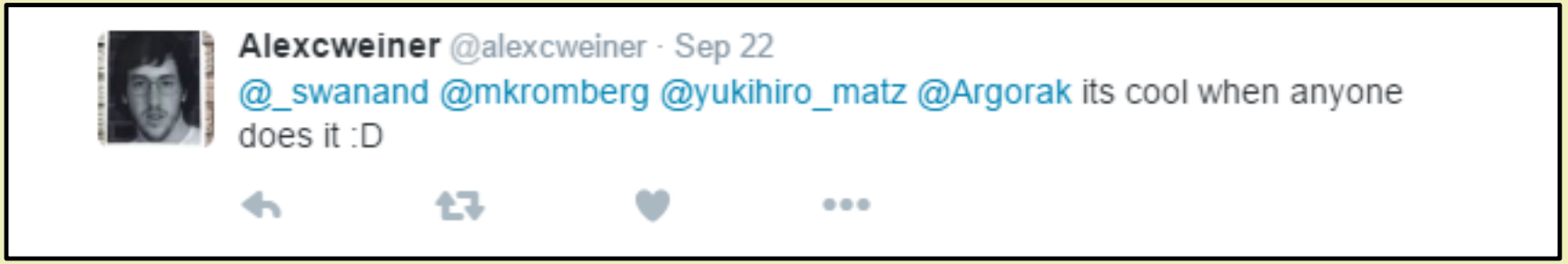



Why Learn APL?

Most importantly…

- APL is FUN ☺!





Additional Materials

- Examples of Companies using APL
- Demo of modern use of APL
  - Edit code using VS Code
  - Run APL service on Linux; query from Windows

Selected Clients

- KCI Corp (US)
  - Budgeting and Planning
- Carlisle Group (US)
  - Collateral and Securitization for Global Capital Markets
- CompuGroupMedical (Sweden)
  - Worlds Largest “Patient Record” system: 40,000 users and 2.5 million patients records at largest hospital in Scandinavia
- ExxonMobil (US)
  - Optimizes the “Cracking” of Petroleum Products using APL
- SimCorp (DK), APL Italiana (I), Fiserv Investment Services (US), Infostroy Ltd (Russia)
  - Leaders in various markets for Asset Management Systems
- A Finnish game company

Infostroy - Home

www.infostroy.com

Search

InfoStroy Ltd

Leaders in Financial Management Software

Call us on: +7 (812) 325-9797 email: info@infostroy.com

HOME ABOUT PRODUCTS SERVICES NEWS CONTACT

Search Infostroy.com Submit

GAMA

GAMA is the leading Asset Management Solution for Financial Asset Portfolio Management in the Russian market today.

GAMA+

GAMA+ is the first Russian developed software product which helps investment managers to visualise their portfolios, their

SERVICES

From Training and Technical Support to Consulting and Bespoke Solutions, InfoStroy is in a unique position to offer

Investment & Portfolio Ma...

www.simcorp.com

Search

SimCorp

Solutions Services Clients Knowledge Investors Company

News Events Contact

Search by typing

Globalization Webinar, June 17

Overcoming the challenges of managing global operations

REGISTER NOW
Join SimCorp Webinar ›

News and announcements

Share buyback program

12 June 2015

In connection with the program SimCorp A/S will repurchase shares for an amount of up to EUR 10.0m (approx. DKK 74.4m) ...

New Client Reporting ROI Portal

Having trouble proving the ROI of your client reporting project?

VISIT PORTAL NOW ›

IBOR Knowledge and Resource Portal

Make the move to an IBOR-centric investment management system

VISIT PORTAL NOW ›

APL Italiana S.p.A. - News

https://www.apl.it/eng/news.php?menuopen=primo&display=183

Search

apl italiana

SOFiA

Company ▾ SOFIA ▾ Services ▾ Customer Area ▾ Search ▾ IT | FR | ES | DE

News Archive

- **2015-06-16** - The new BPO services
- **2014-12-24** - Data Quality and Solvency II - QRT modules
- **2014-10-08** - ESMA: EMIR notifications
- **2014-06-16** - EIOPA CIRCULAR "Insurance Stress Test 2014" dated 30/04/2014
- **2014-05-29** - COVIP Circular n. 250 dated 11 January, 2013
- **2014-03-13** - Solvency II: two new SOFIA modules
- **2013-11-29** - A new look for Sofia:

News

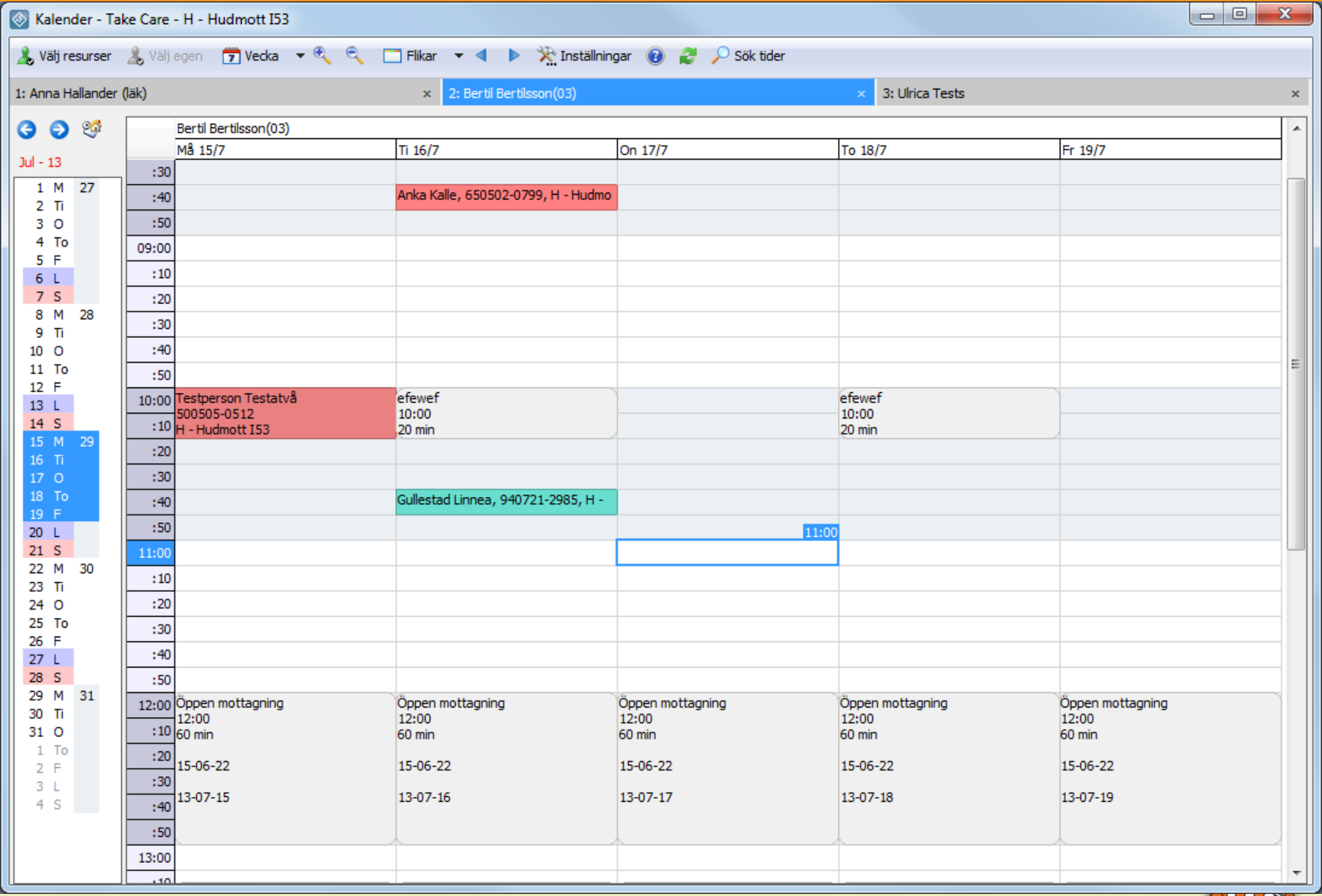
2015-06-16

The new BPO services

SOFIA Online offers new BPO – Business Process Outsourcing services, which allow to outsource to qualified consultants part of the processes concerning middle office, back office and risk management analysis for activities such as the management and the update of the assets database and market data, asset and cash reconciliation, risk management analysis and the production of the related reporting.

With the Application Management and User Administration Profile, the BPO completes the offer of services proposed by SOFIA Online.

For further information and more details, please contact our sales department.


Kalender - Take Care - H - Hudmott I53
Välj resurser
Välj egen
Vecka
Flikar
Inställningar
Sök tider
1: Anna Hallander (läk)
2: Bertil Bertilsson(03)
3: Ulrica Tests
Jul - 13
Bertil Bertilsson(03)
Må 15/7
Ti 16/7
On 17/7
To 18/7
Fr 19/7
Anka Kalle, 650502-0799, H - Hudmo
Testperson Testatvå
500505-0512
H - Hudmott I53
efewef
10:00
20 min
efewef
10:00
20 min
Gullestad Linnea, 940721-2985, H -
11:00
Öppen mottagning
12:00
60 min
15-06-22
13-07-15
Öppen mottagning
12:00
60 min
15-06-22
13-07-16
Öppen mottagning
12:00
60 min
15-06-22
13-07-17
Öppen mottagning
12:00
60 min
15-06-22
13-07-18
Öppen mottagning
12:00
60 min
15-06-22
13-07-19

https://www.youtube.com/watch?v=MzrMjpJouDs&t=306s

Resources

Supporting documentation and materials online:

- Interactive APL Session on line
  - https://tryapl.org (see the "resources" tab)
- Videos:
  - https://dyalog.tv (Webinars & User Meeting Presentations)
- Online Help and Manuals
  - https://help.dyalog.com
  - https://docs.dyalog.com
- Introduction to Dyalog APL by Bernard Legrand
  - https://http://www.dyalog.com/mastering-dyalog-apl.htm
- Google: Try for example
  - https://www.google.com/?q=dyalog+apl+power+operator
- Links to more talks by Morten Kromberg
  - https://www.dyalog.com/blog/about-the-cto/
- Look out for
  - "APL Since 1978" by Roger Hui and Morten Kromberg (HOPL IV conference, 2020)

Demo – Modern APL Environment

- Edit code using VS Code under Windows
- Push to GitHub
- Pull to Linux machine
- Run APL function as a service on Linux and call via curl

Application: Volutions

```
      iv 5
 1  2  3  4  5
16 17 18 19  6
15 24 25 20  7
14 23 22 21  8
13 12 11 10  9

      iv
{⍵ ⍵⍴⍋+\(1↓2/⌽⍳⍵)/(¯1+2×⍵)⍴(⊢,-)1 ⍵}
```

Involution, explained

```
iv←{⍵ ⍵⍴⍋+\(1↓2/⍵⍴⍳⍵)/(¯1+2×⍵)⍴(⊢,-)1 ⍵}
```

```
      iv 5
 1  2  3  4  5
16 17 18 19  6
15 24 25 20  7
14 23 22 21  8
13 12 11 10  9
```

```
involute←{
    moves←(¯1+2×⍵)⍴1 ⍵,-1 ⍵      1 5 ¯1 ¯5 1 5 ¯1 ¯5 1
    repeat←1↓2/⍵⍴⍳⍵               5 4 4 3 3 2 2 1 1
    path←+\repeat/moves    1 2 3 4 5 10 15 20 25 24 23 22 21 16 ...
    ⍵ ⍵⍴⍋path ⍝ convert positions to indices in path
}
```