Seminar Report

APL – A Programming Language Overview

Florian Auer
Florian.Auer@student.uibk.ac.at

21 February 2013

Supervisor: Prof. Dr. Georg Moser

Abstract

In this seminar report APL is discussed. The language's history, current usage and vision is covered. Furthermore a short introduction and different opinions about it are given. No previous knowledge is required.

Contents

1 Introduction and Overview

According to Kenneth E. Iverson, the mathematical notation is the best developed language used as a tool of thought. Although its wide range of covered topics shows its consistence and expressivity, it lacks universality. Symbols are differently interpreted from author to author and topic to topic.

As Iverson got in touch with computer programming languages, he saw that they too cover a wide range of applications. Nevertheless every language for itself can express all problems within one conscious notation and therefore is universal.

Now Iverson's idea was to combine both worlds, mathematical notation with its powerful operators and programming languages with their advantage of being universal. He created —to proof his idea— a sample implementation called APL – A Programming Language [7, p. 378, Ch. 5.0].

The intention of this report is to give a brief overview of the APL language. Therefore it discusses APL's past, future and language features. Additionally a first hands-on experience of the language is given by following a fictional mathematical lesson with the help of APL.

Overview First the report outlines the history of APL, from the notational (§2.1, §2.3) up to its commercial usage in the industry (§2.2). After this it discusses some of APL's features and properties (§3.1, §3.2, §3.3) and shows it's influence on other languages (§3.4). A basic introduction into APL is given by a fictional lesson (§3.5) to provide hands-on experience. To complete the overview APL's achieved goals (§4.2) and opinions (§4.1) about it are presented. Finally the report gives an answer to what would happen if APL was never invented (§4.3).

2 Past, Present, Possibilities

2.1 A Brief History

Timeline

1957	*Iverson notation* for teaching, describing and analyzing
1962	*A Programming Language* by Kenneth E. Iverson
1966	First Implementation for the System/360
1980s	Dr. Jim Brown enhanced APL to APL2
1990s	Dyalog APL, APL2000 and MicroAPL

1957 Kenneth E. Iverson developed a notation for his writing (on a book about automatic data processing) and to support his teaching at the Harvard University. Soon it was known as the *Iverson notation*, a notation major influenced by the inventor's mathematical background.

1960 Iverson joined IBM and collaborated with Adin Falkoff. They focused to use the notation to describe machines formally. Through its adaption to its new environment and the first real world tasks to solve, the notation matured. Its first step of evolution led to Kenneth E. Iverson's book *A Programming*

Language, whose title named the notation since then. Still it was a solely pen-and-paper notation without any intention to be executable or optimized for any computer architecture.

1964 the language was "*well-defined to give us* [the APL team at IBM] *some confidence in its suitability for implementation*" [4]. To prepare the language for the use on computers they first had to handle the amount of customized symbols used in the language. The solution was a character set dedicated to the use for APL and therefore an own keyboard layout. Because of several hardware-based limitations the language itself had to be adopted to its new environment.

> *"Although we expected* [...] *a deleterious effect* [...] *we now feel that the changes were beneficial."*[4]

For example the former limitation of arrays to a maximum rank of two (notated with the use of sub- and superscript indexes) had to be changed to a linearized form `a[i;j;k]`, which allows several dimensions to address.
After these customizations Lawrence M. Breed, Phil Abrams (Stanford grad students) and Roger Moore (later founder of I.P. Sharp Associates – a computer time sharing company) [11] implemented the first interpreter for APL in 1965. A year later a version for the System/360 followed. APL was from now on an executable computer programming language.

In the late 1970s and early 80s APL was commercial successful. It was taught in schools and universities, discussed at conferences and one of the most used languages [6].

1979 the Turing Award committee awarded Kenneth E. Iverson for his work on combining programming languages with mathematical notations, which resulted into APL [7].

In 1980 Kenneth E. Iverson left IBM and went back to Canada – his home country – to work at I.P. Sharp Associates. There he directed the development of Sharp APL and made it more similar to his initial vision of the language.

Back at IBM Dr. Jim Brown took over the IBM APL development. He advanced the language with nested arrays and other features to make it more competitive within the growing market of APL dialects, and called it from now on APL2.

In the 1990s APL2 got serious competitors with other interpreters like Dyalog APL, APL2000 and MicroAPL. They extended the language with all kinds of features like object-orientation, SQL database manipulation or lambda expressions. Since then, they lead the APL development and have the most advanced APL interpreters available.

2.2 Present Usage Fields

APL's ability to adapt code fast to modifications makes it useful for volatile businesses like the insurance or financial industry. There APL is still today popular or at least there are programs written in it used.

Other fields in which APL sometimes is used are simulations and mathematics, where its mathematically-inspired notation and high abstraction level are

preferred. But with the rise of other mathematical programming languages like Matlab or Octave, APL lost most of its users in these fields.

It too lost users in the industry as new spreadsheet tools like Microsoft Excel were accessible. They provided easily accessible, graphical interfaces to the most common use cases and therefore were preferred over the text-only APL. Again a lot of users migrated away from it in the 1990's as its successor language J was developed.

These are some of the reasons why today APL is sometimes called a dead language. Nevertheless it has an active community (like at `comp.lang.apl` or the British APL Association) and conferences are held to discuss its further development like the annual Dyalog Conference.

2.3 Large Motivational Vision

Kenneth E. Iverson mentioned his personal view of the language in 1991. After the creation of J, he wrote in one of his papers called *A Personal View of APL*.

> *"The initial motive for developing APL was to provide a tool for writing and teaching. Although APL has been exploited mostly in commercial programming, I continue to believe that its most important use remains to be exploited: as a simple, precise, executable notation for the teaching of a wide range of subjects."* [8]

3 The Language Itself

3.1 Some Uncommon Features

APL provides a rich set of functions to modify arrays. To express all of them and to make them easily combinable it uses symbols instead of words to identify functions. This distinguishes APL a lot from common languages like C or Java, where only the basic arithmetical functions like addition, subtraction, multiplication and division are expressed by a single symbol.

APL is much more productive as classical languages like C, Cobol or Fortran. If you believe in a productivity measure made by Kevin Weaver back in 1989, you can achieve the same task as in Cobol in a tenth of the time in APL[14].

> *"With APL, you spend your time solving the problem at hand, not translating it into the world of bit, bytes, registers, pointers, do-loops, datatypes, and dimension statements."*[1]

Most languages forces the user to distinguish different levels of precedence (the order of which operators get evaluated), like Matlab with its eleven levels of operator precedence. APL's designer also had to cope with this problem, but because of the amount of operators and functions to deal with, a hierarchy of precedence would have been unrealistic. That is why Kenneth E. Iverson decided to control the precedence with a single rule:

> *"The order is otherwise* [if not controlled by parentheses] *determined by one simple rule: the right operand of any function is the value of the entire expression following it. In particular, there is no precedence among functions; all functions, user-defined as well as primitive, are treated alike."* [3]

3.2 Turing Completeness

A Programming Language is said to be Turing-complete if it can simulate a Turing machine (a computational model capable of solving any computational problem). Therefore this means that the language is as powerful as a Turing machine.

To show whether APL is Turing-complete or not, we use an instruction set which is Turing complete. This set was introduced by Raul Rojas, in his paper *"Conditional Branching is not Necessary for Universal Computation in von Neumann Computers"* [12]. It shows that that *"any computable function can be computed using only the instructions LOAD, STORE, INC, and GOTO (unconditional branching)"* [12].

These four instructions can be expressed in APL as follows:

instruction	APL	meaning
`LOAD A`	`A`	load address `A` into accumulator
`STORE A`	`A ← X`	store accumulator into address `A`
`INC`	`A ← A + 1`	increment accumulator
`GOTO X`	`→ X`	unconditional jump to address or source line `X`

With the help of this mapping between the instruction set and APL it is possible to translate every program expressed with the help of the four instructions into semantically equivalent APL code – thus showing that APL is Turing-complete.

3.3 Higher Order Functions

Higher order functions are functions which take other functions as input or output. In APL this language property is very dominant and commonly used with operators like `/` which take a function and an array as input.

There are for example two very useful higher order functions.

The function *map* takes a list and another function and applies the function on every element of the list. In APL the *map* function is a built-in feature of the language. If you model the list as an array a and take for example the function $y = x+2$ as an argument to the *map* function, an APL implementation could be:

```
a + 2
```

If an explicit defined function is preferred it is also possible to write:

`{ ω + 2 } a`
(ω is the left parameter of the anonymous function)

The other convenient function is *fold* which in fact works exactly the same as the reduction operator. It takes a function and a list (in APL an array) and *inserts* the function between the elements of the list. So for example: $fold + [1, 2, 3, 4] = 1 + 2 + 3 + 4 = 10$ again this can be expressed in APL:

`+ / 1 2 3 4` or as `{ α + ω } / 1 2 3 4`
(α is the right parameter of the function; here the value which gets *folded* with ω which is the sum of the previous elements)

3.4 Influence On Others

The approach to write in a more mathematical notation than in a computer suitable influenced several numerical computation languages like Matlab or Mathematica. Actually Matlab creator Jack Little named APL as one of its major technology in the business plan for MathWorks, the software company behind Matlab.

> *"The product will be unique and revolutionary. Combining the technology of 1) mice and windows, 2) matrix and APL environments, and 3) direct manipulation, will do for engineers what Lotus 1-2-3 has done for the business world."*[10]

Another key feature in which APL was one of its pioneers was the interpreted development environment. Most of the other languages at its time like FORTRAN or COBOL had to compile the program first to execute it. The possibility to interact with the language and to get immediate response to an expression was a whole new experience back than. Teaching such a vivid language was much more easier, because students could try out any expression and discover the language by trial-and-error.

The concept of a workspace was part of the APL interpreter. It allowed to save user defined functions along with data in a so called workspace. So the user could later load, save or copy functions and data from one workspace to another. This feature became the standard one in the Matlab environment.

The popularity back in the 1970s, 1980s of APL makes it reasonable that APL may have helped to spread the idea of interaction in the programming language area and therefore may have partly influenced a lot interpreted languages.

Finally APL not only influenced other programming languages: Until 1962 a book called *A Programming Language* was published, the common notation for the floor function was the square bracket notation $[x]$, introduced by Karl Friedrich Gauss. With the book of Kennet E. Iverson not only a new function called ceil were established, but also his notation for the ceil $\lceil x \rceil$ as well as for the floor $\lfloor x \rfloor$ function became the standard one.

3.5 A Programming Lecture

To get some hands-on experience with the language the reader is taken back to the 18th century. The reader joins a mathematical class and discovers in doing so APL. At the end she or he will be able to solve simple mathematical problems with the help of the language.

> Paragraphs written in this style are observations on the class,

whereas paragraphs written in the standard style are here to get in touch with APL and are directed to the reader.

> "There it is". The teacher looked on the slate: "5050". "How could you come up so fast with the solution?". Karl, the boy who wrote down the number, answered: "You gave us the task of adding all numbers from 1 to 100. So, I imagined myself a row of all numbers: 1, 2, 3, 4,..."

Let's open our APL interpreter (or an online one, like `www.TryAPL.org`) and follow Karl's thoughts. Type `⍳100` into it and press [enter] . It will respond with all the numbers from 1 to 100. *Iota* ⍳ is a function who takes one operand on its right side and returns a vector (a one-dimensional array) with the desired numbers.

```
    ⍳100
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21... 98 99 100
```

There are lots of functions which are taking only one operand on their right side. For example `+`,`-` or ÷ which calculates the identity, the negation and the reciprocal (the number $\frac{1}{x}$). These functions are called monadic, which means they are taking only one operand on their right side. But let's return to Karl.

> "... 5, 6, 7 and so on, up to 100. But then I said to myself: Wow, that's a long line of numbers. Maybe it's easier if I split it into two. So I ended up with two lines, first from 1 to 50 and the second from 51 to 100."

Okay lets do this. We could now type `⍳50` to get the first row, but how do we get the second one? Instead of trying to make two rows, we simple change the shape of the existing one. APL provides therefore a function called *reshape* ⍴, which takes two operands: First the desired shape, described by the number of items or rows and columns and secondly the array to shape.

We want two rows, each containing 50 items — 50 columns. Therefore we type: `2 50 ⍴ ⍳100` to reshape the vector to a matrix (a two-dimensional array).

```
    2 50 ⍴ ⍳100
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21... 48 49 50
51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67... 98 99 100
```

Maybe you wonder why this works without parentheses; why ⍴ doesn't get first executed — after all it is the most left. That's right but APL has plenty of functions and that's why it's precedence (the order of execution) is determined by a single rule to simplify things: First execute the right most function.

To reshape the matrix again to a vector we could type: `100 ⍴ 2 50 ⍴ ⍳100`. As our expression gets longer, we would appreciate to save our progress so far. So that we can use it later for further computations. Actually there is a way to do so. We will use a variable — a placeholder for the result of the expression assigned to it. Knowing this, we store our two row matrix in the variable `m` (for matrix).

```
    m ← 2 50 ⍴ ⍳100
    m
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21... 48 49 50
51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67... 98 99 100
```

> "Doing this I thought perhaps there is some regularity that I can exploit. That's why I started to add 1 and 51, 2 and 52, 3 and 53 which would result into..."

Sitting on a computer, it should not be a problem to follow Karl's idea and summing up 1 with 51, 2 and 51, and so on.

```
    1 + 51
52
    2 + 52
54
```

Do we really have to type in all 50 pairs? There must be another, a simpler solution. And of course there is. Remember our previously defined matrix `m` ? If not, type `m` and press enter .
We already have the right numbers on top of each other. All we have to do now is to sum up each column. To do this we use the reduction `/` operator. Why an operator and not a function? The difference is this: A function operates on two values, an operator on the other side creates a new function out of a given function. In our case the reduction operator takes as its left operand a dyadic function (a function taking two operands) and an array to apply the function on it. Enough theory, let's see it in action.

```
    +/ ⍳10
55
```

This is equally to `1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10`, which is exactly what the reduction operator expresses (keep in mind the precedence).

```
    +/ ⍳m
```

```
1275 3775
```

What's that, only two numbers? This is of course not what we wanted; we expected 50 numbers. To get what we want, we need to write `+/[1]m`. *Axis* operator `[]` is dyadic and takes two operands, one on its left side and one between the brackets. In our case the 1 between the brackets forces APL to sum over the first dimension, instead over the last dimension —in our case the second— as it is default. So instead of summing over the columns, we instruct to sum over the rows.

```
    +/[1] ⍪m
52 54 56 58 60 62 64 66 68 70 72 74 76 78 80 82 84... 146 148 150
```

Does this say anything to you? Do you see some structure to exploit? Let's listen what Karl has to say.

If you think now: Hey why don't we simply write `+/⍳100` *and we are finished. Although that would return us the right result, poor Karl never heard anything about APL. So lets move on and see how he solved the problem anyway.*

> "... 52, 54, 56 and so on. Interesting, 50 is the half of 100 — our highest number. But how does 52, 54 and 56 corresponds to 1, 2 and 3? So I tried another combination of the numbers. What if the second line is reversed? So that I add 1 with 100, 2 with 99, 3 with 98 ..."

You think maybe there is a function or operator to reverse numbers? I think you finally recognize the function richness of APL. Yes, there is a monadic function called *reverse* ⌽ which does exactly what we want. Again it reverses the numbers along the last dimension of the array. This means in our case, it reverses the numbers in one row.

```
    ⌽ ⍳10
10 9 8 7 6 5 4 3 2 1
```

Now if we would simple enter `⌽m`, both lines would be reversed. To change only the second one we need first to select the second line and then assign it the reversed order of the numbers.

Selecting items of an array works with indexing. For a vector you can only select out of one dimension, hence you can write `vector[1]` to select the first and `vector[1 3 2]` to select the first, third and second item, in this order.

Matrix indexing works similar, but now you have two dimensions. Therefore you need to specify the rows and columns which you want to select. This can be achieved by first defining the rows, followed by a semicolon and finally naming the columns. The selected items are then the items at the intersection points of the rows and columns listed.

```
    3 3 ⍴ ⍳9              ⍝ after this symbol follows a comment
1 2 3
```

```
4 5 6
7 8 9
    (3 3 ⍴ ⍳9)[1 3;2 1]   ⍝ Why do we need to put parentheses
                            around 3 3 ⍴ ⍳9 ?
2 1
8 7
```

To select a whole row or column, you simply leave the row/column section blank. That's why `m[2;]` selects the second row with all its columns.

Now we know how to select a row out of an array and how to reverse a row of numbers. This means we can already do half of the task: `⌽m[2;]`. But how do we swap our new row with the existing in `m`? APL does not only allow you to select items out of an array for reading, but also allows you to use the selection as a kind of assigning mask. So if you take our selection `m[2;]` and put it on the left side of the assign arrow (`m[2;]←...`) and provide on its right side an array of the same dimension as the selection on its left side (`m[2;]←⌽m[2;]`), you assign the values according to the selection.

```
    m[2;] ←⌽m[2;]
    m
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21... 48 49 50
100 99 98 97 96 95 94 93 92 91 90 89 88 87 86 85 84 83.. 53 52 51
```

Wait, how do we know the dimension of our selection? We could simple type in the selection alone and count the rows and columns or we could use a function returning us the dimension as an array. Remember *reshape* `⍴`, the dyadic function which reshapes an array according to a given dimension? If you use `⍴` monadic (only with one operand on it's right side) it is called *shape of*. And as the name already suggests, it returns the number of rows and columns as an array. Let's try this out.

```
    ⍴ (3 3 ⍴ ⍳9)               ⍝ Can we leave out the parentheses?
3 3
    ⍴ ⍳9
9
```

Done the reversing of the second row, Karl wants us to add column after column. Because we changed `m` to fit our needs, we can again apply the same expression as we already used to sum up.

```
    +/[1] m
101 101 101 101 101 101 101 101 101 101 101 101 101... 101 101 101
```

Hm, do you see any structure in the data? Of course you do, but let's see if Karl sees it too.

> "... and so on. And do you know what the result of these additions is? It's 101, 101, 101! To be sure I also added the last numbers in both lines. 50 + 51, 49 + 52 and 48 + 53, its all 101. Now I only had to know how many 101s I have, to know what is the total sum...."

We already discussed how to get the dimension/number of columns of an array. We therefore use the *shape of* function.

```
   ⍴ +/[1] m
50
```

> "The number of 101s is of course 50, because I initially split the 100 numbers into two. Finally I multiplied 101 with 50, which results into 5050." The teacher stopped his nodding and said: "All these lines of numbers, reversing numbers, summation of pairs of numbers. Come on. You already knew the answer from somewhere else, right?". The boy, in the stood up: "No, I don't". "Then, please explain your calculation again on the blackboard. So that everyone in the class can follow your unbelievable computation.", said the teacher and took out his pen to start copying every strike the boy would do on the board.

Although this story served us to get in touch with APL, it is as you might have expected not strictly based on any historical fact. Besides the boy Karl, who is based on Karl Friedrich Gauss, a great mathematician and physician. At the age of nine, it is said that he had discovered the formula $1+2+3+...+n = \frac{(n+1)n}{2}$.

Summary of what we have learned:

- Functions take values and produce values.
- Operators take functions as operands and create new functions.
- First execute the right most function/operator.
- Monadic functions/operators take one operand on its right side.
- Dyadic functions/operators take two operands, one on its right side and one on its left.
- `⍳N` creates a vector with the numbers 1 to `N`
- `r c ⍴ a` reshapes the array `a` to have `r` rows and `c` columns
- `+/a` reduces the array with a given function (here `+`).
- `+/[1] a` reduces along the first dimension.
- `⌽a` reverses the order of the items of array `a`, along the last dimension.
- `a[3 1 2]` returns the third, first and second item of array `a` as a vector.

- `a[3 1; 2]` returns the third and first item of the second column.
- `a[3;]` returns the third row.
- `⍴a` returns the shape of the array `a` — the number of items (of a vector) or columns and rows (of a matrix)

and of course we learned $\frac{(n+1)n}{2}$ is the sum of all numbers from 1 to n.

4 Reflections On APL

4.1 Opinions About It

If you are talking about a programming language, you have to talk about its users too. What are their opinions? Do they still use APL? To cover these and similar questions we will discuss two opinions about APL. One of a supporter, who still uses APL and one of someone how doesn't prefer to use it. And finally I am briefly talking about my first impressions using it.

Pro *Why I'm Still Using APL.* Jeffrey Shallit is a computer science professor at the University of Waterloo, Canada. He gave a half an hour lecture at the APL@50 Conference (2012), York University. It celebrated the 50th anniversary of the publication of *A Programming Language* by Kenneth Iverson. He says that with APL he can get things done much faster than with other languages. Because he does not have to declare obvious properties of data and he does not need to master arcane *"ritual incantations"*[13] to do things that should be easy. After this he compares Java with APL. Therefore he wrote a simple program, reading integers and summing them up, in both languages. In Java he needed 25 lines of code, in APL he ended up with three characters: `+/⍞` (⍞ is no typesetting error, it is a function reading numbers from the user input). Then he shows more sample implementations of simple algorithms and continues to quote others about their opinion of APL.

> *"By relieving the brain of all unnecessary work, a good notation sets it free to concentrate on more advanced problems and in effect increases the mental power of the race."* [A.N. Whitehead]

In one of his last slides he also mentions *Things APL Never Did Perfectly.* For example the reading and writing of external files or the transfer of workspaces between systems. He too claims that the control structures like `do ... while` were never perfectly implemented. And finally he speaks about where he still uses APL: *"Experimentation, large-scale computations, research in automata theory, formal languages, and number theory, grade summaries for courses"* [13].

Contra Edsger Wybe Dijkstra is known for his dislike of APL. Not so well known is the reason behind his disfavour of the language. Dr. Alan Caplin was also puzzled and therefore wrote Edsger W. Dijkstra a letter[1] to ask him about his reasons. Edsger W. Dijkstra answered (dated 2 January 1982) with a letter containing some points to justify his view of APL. First he argues that *"the tool shapes the one who trains himself in its usage"*. This means he believes that the "cult" about APL drove people into the language and by practicing it they learned how to cope with it, regardless of it's suitability. And *"that most people (be it subconsciously) realize that 'ease of use' is not most significant aspect. Experience has shown that, provided people are sufficiently thrilled by a gadget, they are willing to put up with the most terrible interfaces"*. Another characteristic of APL he criticizes is its *"closeness to an implementation"*. To illustrate what he means, he mentions a professor *"trying to teach APL, bitterly complained about the absence of APL terminals. He was clearly unable to teach it without them"* —arguing that APL is working on amounts of data and with a variety of operations, which is hard to understand without seeing the intermediate results. Finally Edsger W. Dijkstra says that it may be viewed as a major shortcoming of the language to not be good enough propagateable through written word.

> *"APL is a mistake, carried through to perfection. It is the language of the future for the programming techniques of the past: it creates a new generation of coding bums."* [2]

Author's opinion Hearing the first time about the language when I was assigned to write about it, I looked it up on the Internet. I found symbols, which supposedly are able to solve common problems in one line — unbelievable or at least unreadable. After some time studying the freely available book *Mastering Dyalog APL* by Bernard Legrand, I felt a little bit more comfortable with the notation. But even after learning the ground structure, its rules and basic functions and operators, but I still can't read much of the code out there. In other programming languages you can read most of the code after some time, or at least you can think of what the function would do by its name. But in APL all you have are symbols, which doesn't help you much without knowing them. Nevertheless, even as a novice I saw the power of APL, by having the first time not to cope with much of the loop-over-some-array stuff and other bloated and trivial code parts. Which freed my mind to think more about the problem itself than its implementation.

4.2 Goals Already Achieved?

Kennet E. Iverson created the language at first with the goal to create a notation that supports his writing and teaching work.

After years of persistent reevaluation he created a notation usable as writing tool. Evidence therefore is for example the formal description of the IBM Sys-

[1] Many thanks to Edgar Graham Daylight for providing me photocopies of the letters. He too talks about them at his blog: http://www.dijkstrascry.com/node/90

tem/360, which describes all functional characteristics of the computer system in APL [5].

The second part of the goal, to support teaching with a more suitable notation was not achieved by APL as Iverson's own reflections indicate:

> *"Although APL has been exploited mostly in commercial programming, I continue to believe that its most important use remains to be exploited: as a simple, precise, executable notation for the teaching of a wide range of subjects."* [8]

Later he wanted to get APL into education [9]. Therefore he created a new language called J, which was more suitable for teaching as it was freely available. To realize this, he dropped the customized symbols, made it printable on standard printers and runnable on a variety of computes. So he could provide an APL system for use in teaching mathematics and related topics that is modern, free, and transportable.

To conclude, Kennet E. Iverson's first intention with the language was to create a notation for writing and teaching. After APL succeeded as a useful writing but not teaching language, he decided to make a new attempt and created J with the purpose to support teaching with it.

Iverson also pursued another goal with APL which he mentioned in his Turing Award winning lecture *Notation as a Tool of Though* (1979). He describes APL as an attempt to combine two types of notations. The mathematical notation with its already defined and well developed operations, but lack of universality (symbols have different meanings in different topics or from author to author). And the programming languages, providing universality and executablilty. So Iverson thought the combination of both worlds would be a desirable goal.

> *"This* [the combination] *is an important question which should receive further attention, regardless of the success or failure of this attempt to develop it in terms of APL."* [7]

4.3 What Would Happen

Now one could ask about APL's necessity of creation. Would it have made any difference whether APL was created or not? To answer this question we look at it's influence on other languages. As already mentioned, APL influenced and inspired lots of other mathematical and array oriented languages, like Matlab or J.

Take for example Matlab, a numeric computation environment used by around one million users in industry and in academic environments. MathWorks' (the company behind Matlab) initial technical business plan described their product as also containing *matrix and APL environments* [10].

J, a programming language developed by Kennet E. Iverson and Roger Hui, is strongly based on APL. This language is not only influenced by APL, but also one of its creators (Kennet E. Iverson) is the creator of APL. Which allows us to say, that without the previous experiences with APL, Iverson may not

have come to the same conclusions and design decisions, used in J.

To think about the idea of an APL free world out, we must consider the nowadays existing APL developers. What would they use instead? If you believe that J would exist in such a world, it seems to be the natural choice to take, given that APL is it's predecessor. Otherwise other mathematical or array based languages may suite the needs of such programmers. Matlab, Mathematica or Octave may seem appropriate.

5 Conclusions

I have given a brief insight into the programming language APL. Its history from the beginning as notation to its usage as programming language, its still current usage in some fields and some opinions about it. An introduction, dealing with the summation of numbers give first hands-on experience in executing simple expressions of the language. And finally the theoretical question was discussed, whether APL was necessary to be created or not.

I came to the conclusion that APL requires some time to get used to - but this time was well spent. Because it introduced me to a new way of programming and thinking about problems.

References

[1] G. A. Bergquist. The future of APL in the insurance world. *SIGAPL APL Quote Quad*, 30(1):16–21, Sept. 1999.

[2] E. W. Dijkstra. How do we tell truths that might hurt? *SIGPLAN Not.*, 17(5):13–15, May 1982.

[3] A. D. Falkoff and K. E. Iverson. The design of APL. *SIGAPL APL Quote Quad*, 6(1):5–14, Apr. 1975.

[4] A. D. Falkoff and K. E. Iverson. The evolution of APL. *SIGAPL APL Quote Quad*, 9(1):30–44, Sept. 1978.

[5] A. D. Falkoff, K. E. Iverson, and E. H. Sussenguth. A formal description of SYSTEM/360. *IBM Syst. J.*, 3(2):198–261, June 1964.

[6] R. Hui. Remembering Ken Iverson, 2004. `http://keiapl.org/rhui/remember.htm`, accessed: 2012-12-05.

[7] K. E. Iverson. Notation as a tool of thought. *Commun. ACM*, 23(8):444–465, Aug. 1980.

[8] K. E. Iverson. A personal view of APL. *IBM Syst. J.*, 30(4):582–593, Oct. 1991.

[9] C. Lathwell. Why Ken Iverson started J - with Eric Iverson and Morten Kromberg. `https://www.youtube.com/watch?v=ae94kBFCe88`, accessed: 2012-12-05.

[10] C. Moler. The growth of Matlab and the MathWorks over two decades. `http://www.mathworks.com/company/newsletters/news_notes/clevescorner/jan06.pdf`, accessed: 2012-12-05.

[11] M. S. Montalbano. A personal history of apl. `http://www.ed-thelen.org/comp-hist/APL-hist.html`, accessed: 2013-02-20.

[12] R. Rojas. Conditional branching is not necessary for universal computation in von neumann computers. *J. UCS*, 2(11):756–768, 1996.

[13] J. Shallit. Why i'm still using APL. `http://recursed.blogspot.co.at/2012/11/my-talk-at-apl50-conference.html`, accessed: 2012-12-05.

[14] K. R. Weaver. Measure productivity: use a generally accepted metric. *SIGAPL APL Quote Quad*, 19(4):377–380, July 1989.