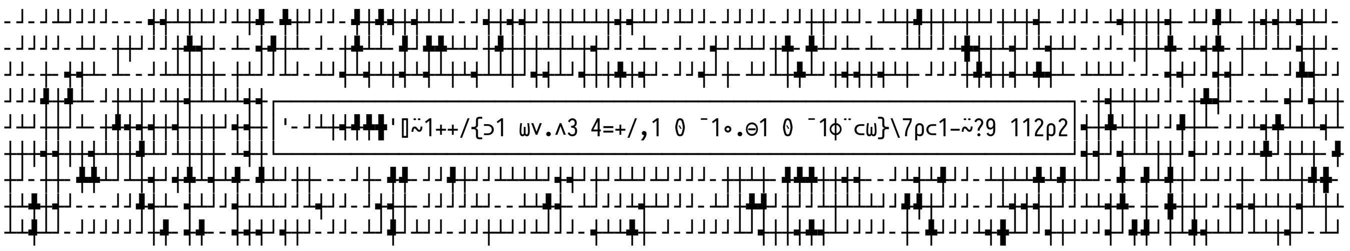
April: APL Compiling to Common Lisp

Andrew Sengul
asengul@fastmail.fm



```
'-┘┴┼┼╋╋╋'⎕⍨1++/{⊃1 ⍵∨.∧3 4=+/,1 0 ¯1∘.⊖1 0 ¯1⌽¨⊂⍵}\7⍴⊂1-⍨?9 112⍴2
```

Figure 1: An APL expression framed by its output

ABSTRACT

This paper demonstrates the April APL compiler (code hosted at https://github.com/phantomics/april). April compiles a subset of the APL language into Common Lisp, allowing APL's terse, efficient syntax to be leveraged for array processing and mathematical operations within a Common Lisp program. Along with the compiler April includes a suite of specification tools making it easy to extend the language, allowing for a uniquely flexible development approach. Released under the permissive Apache 2.0 license, April has been leveraged in a graphical display hardware startup and a variety of applications including statistical analysis, vector graphics and terminal interfaces.

CCS CONCEPTS

• **Software and its engineering** → *Software design engineering*; • **Computing methodologies** → **Computer algebra systems**; **Representation of mathematical functions**.

KEYWORDS

demonstration, compiler, array, DSL, APL, Lisp, linear algebra, vector languages, interoperability

ACM Reference Format:
Andrew Sengul. March 2022. April: APL Compiling to Common Lisp. In *Proceedings of the 15th European Lisp Symposium (ELS'22).* ACM, New York, NY, USA, 5 pages. https://doi.org/10.5281/zenodo.6349745

1 INTRODUCTION

APL is known for its exotic character set and minimalist style. Like Lisp the language was originally designed as a mathematical notation [7] and creator Ken Iverson didn't anticipate that APL expressions could be evaluated by a computer. His colleagues built the first APL interpreter using a variant of Iverson's notation simplified for use with a teletype terminal [4], just as John McCarthy's students traded M-expressions for S-expressions to develop the original IBM 704 Lisp interpreter. APL followed an evolutionary path somewhat similar to that of Lisp as the language grew to prominence on mainframes, functioning as a complete operating system for the machines where it ran [5].

Work on April began in late 2017 and it has since gone through multiple development iterations of its core compiler, functions and specification macros. A previous talk I gave on April can be viewed at https://youtube.com/watch?v=AUEIgfj9koc. Since then April has evolved considerably, incorporating tacit function composition, inline operators and multithreading support for almost all functions.

2 USING APRIL

The simplest way to use April is to pass APL strings to the `(april)` macro. An example is `(april "1+1 2 3")`, which returns the vector `#(2 3 4)` – APL composes addition and other scalar functions over arrays, so the 1 is added to each element of `#(1 2 3)`. APL's core functions are all just one character long, like +, -, × and ÷. April can also take files of APL code as input and it has a wide variety of configuration options. April's parameters may be passed as the first argument to the `(april)` macro inside a `(with)` form.

A complete introduction to the APL language is far beyond the scope of this section but a good starting point is April's README file, located at the link in the abstract. The README has guidelines on ways of entering APL characters and links to online resources including tutorials from Dyalog and other sources. April is included in Quicklisp and installing it is as simple as evaluating `(ql:quickload 'april)`.

April runs its character input through a lexer, converting the characters to tokens which are then fed to a compiler that generates Common Lisp code. The `(:print-tokens)` parameter prints tokens output by the lexer before they are passed to the compiler:

```
* (april (with (:print-tokens)) "1+1 2 3")
(3 2 1 (:FN #\+) 1)
#(2 3 4)
```

Note that the lexer accumulates the tokens in reverse order; this is natural since APL code is evaluated from right to left and the tokens are thus fed to the compiler starting from the end of each line read.

The `(:compile-only)` parameter causes April to print its compiled output instead of evaluating it:

```
* (april (with (:compile-only)) "1+1 2 3")
(IN-APRIL-WORKSPACE COMMON
  (LET ((OUTPUT-STREAM *STANDARD-OUTPUT*))
    (DECLARE (IGNORABLE OUTPUT-STREAM))
    (SYMBOL-MACROLET
        ((INDEX-ORIGIN ⊑*INDEX-ORIGIN*)
         (PRINT-PRECISION ⊑*PRINT-PRECISION*)
         (COMPARISON-TOLERANCE
           ⊑*COMPARISON-TOLERANCE*)
         (DIVISION-METHOD ⊑*DIVISION-METHOD*)
         (RNGS ⊑*RNGS*))
      (A-OUT (A-CALL (APL-FN-S +)
                     (AVEC 1 2 3) 1)
             :PRINT-PRECISION
             PRINT-PRECISION))))
```

Note the ⊑ reader macro. It works to intern symbols in the proper workspace packages in tandem with the `(in-april-workspace)` macro. Like other APLs April stores its functions and variables in named workspaces, which are implemented as Common Lisp packages. When the macro `(in-april-workspace common ...)` is expanded, an instance of `⊑symbol` within is transformed into the symbol `april-workspace-common::symbol`. Considerable work has been done to make April's compiled output human-readable, with many macros abbreviating common structures that would otherwise bloat the code.

Compared to other APL implementations April stands out for its seamless interoperability with Common Lisp, and through CL other languages and systems. APLs have traditionally been implemented as monolithic interpreters, and communication with external APIs must be done through a plugin to the interpreter. The most popular APL implementation, Dyalog APL[1], is proprietary and thus any such plugin must be created by Dyalog. Other free software APLs exist, but their implementation in Algol descendants like C++ and Java makes extension an ordeal.

The simplest way to pass values from CL into April is to use the `(april-c)` macro. Here, the number 10 is passed as the second argument to `(april-c)` and is represented by `⍵`, which stands for the right argument, within the APL function.

```
* (april-c "{⍵+5}" 10)
15
```

April's `(:state)` parameter with the sub-parameters `(:in)` and `(:out)` can be used for more complex input and output.

```
* (april (with (:state :in ((a 3) (b 5))
                       :out (a c)))
         "c←a+⍳b")
3
#(4 5 6 7 8)
```

Variables named `a` and `b` are passed in, and the variables named `a` and `c` are returned. The `[⍳ index]` function seen here produces a vector of numbers from 1 to its argument, and ← assigns the result of the vector's addition to `a` to the variable `c`.

Passing functions into April is likewise simple:

```
* (april (with (:store-fun
               (add-ten (lambda (x)
                          (+ x 10)))))
         "")
NIL ;; nothing is evaluated, so nil is returned

* (april "addTen 20")
30
```

Dash-separated variable names are converted to camelCase within April, since the `-` character expresses the subtraction function in APL.

April does not have any stock functions for system interaction, but using the `(:store-fun)` parameter they can easily be added as required. Here is an example using the `uiop`[2] library:

```
* (april (with
          (:store-fun
           (sh (lambda (s)
                 (uiop:run-program
                  (coerce s 'string)
                  :output :string)))))
         "")
NIL

* (april "' GOODBYE',⍨sh 'echo HELLO'")
"HELLO
 GOODBYE"
```

In just a few lines, April can thus be extended to support running terminal commands. Recurring questions addressed to other vector language projects like "When will we get JSON support?" and "When will we be able to make HTTP requests?" can be addressed by April users within minutes.

3 IMPLEMENTATION

Common Lisp has powerful tools for working with arrays but their syntax is often cumbersome. APL can build and transform arrays with only a handful of characters, making tasks that take a large amount of code in Common Lisp much simpler to write. This led to my interest in leveraging APL within Common Lisp, and CL is one of the best choices of language to implement APL because it has almost all of the necessary array faculties inbuilt. With support for nested arrays, high-rank arrays and zero-rank arrays, it's easy to work with April's array output using standard CL code. This section outlines some of the more interesting challenges encountered in the course of developing April.

3.1 The Core Specification

Building a programming language is a complex task. I wrote the `(specify-vex-idiom)` macro to mitigate this complexity, implementing a core specification for April that can be seen in the source file `april/spec.lisp`[3]. This large macro specifies all of April's lexical functions and operators along with its language utilities, putting all the language's significant configuration in one central

[1]https://www.dyalog.com/

[2]https://gitlab.common-lisp.net/asdf/asdf/-/tree/master/uiop

[3]https://github.com/phantomics/april/blob/master/spec.lisp

location. Information like the inverse forms of functions and their alternative character representations can be found here, along with all of their unit tests.

This centralized organization has made the development of the language significantly faster than would have been possible with a different style. For example, the recent addition of an inverse form for the `[⍸ where]` function required just one new line in the spec along with an 18-line function added to April's main library.

Moreover, April's specification macros can be used to augment the language with new functional characters in just a handful of lines.

```
(extend-vex-idiom
 april
 (functions
  (with (:name :extra-functions))
  (⍛ (has :title "Add3")
     (ambivalent
      (scalar-function
       (lambda (omega) (+ 3 omega)))
      (scalar-function
       (lambda (alpha omega) (+ 3 alpha omega))))
     (tests (is "⍛77" 80)
            (is "8⍛7" 18)))))
```

In this code, functional character ⍛ is added to the April language, implementing a rather silly function called `Add3` that adds three to its argument (if given one argument) or to the sum of its arguments (if given two arguments). A pair of unit tests for this function are added to the main test sequence as well. The `(extend-vex-idiom)` macro can also be used to overload April's utilities, like the functions that strip comments from code and parse numeric strings.

With this macro skilled developers can patch the language for specific applications, creating custom variants of April with no need to fork its main codebase. The specification macros are implemented in April's sub-package `vex`[4], which contains a set of general tools for implementing vector languages. In the future other vector languages may be implemented based on the `vex` model.

3.2 Array Prototypes

The only significant array feature APL has that CL lacks is empty array prototypes. The prototype of an APL array is its first row-major element [1]. Prototypes are used by functions like `[↑ take]` and `[/ expand]` to fill the empty space resulting when an array is made larger. When an APL array is reduced to size 0, as with functions like `[⍴ shape]` and `[↓ drop]`, it retains the "memory" of its prototype so that if it is expanded to a nonzero size, the prototype will be used to populate the space in the new array. For a character array the prototype is a blank space, and for a numeric or mixed array the prototype is 0. For an array whose first row-major element is a nested array, the prototype is an array of the same shape whose elements are the prototype of the nested array. Thus if the first element in the array is the matrix `#2A((1 2)(3 4))`, the array's prototype will be `#2A((0 0)(0 0))`.

[4]https://github.com/phantomics/april/blob/master/vex/vex.lisp

The Common Lisp array model does not include a "prototype" value, but for non-zero-sized arrays it's unnecessary since the prototype is simply an "empty" version of the first element. Functions that output a zero-sized array will displace the array to a one-element vector containing a list of metadata with the prototype. The `(array-displacement)` function can be used to fetch the metadata from any context, making it straightforward to get the empty array's prototype for functions that use it.

3.3 Multithreading

One of April's recent development priorities has been to use multithreading wherever possible. April uses macros called `(xdotimes)` and `(ydotimes)` to accomplish this, leveraging the `lparallel`[5] library. These macros' definitions can be found in the source file `april/aplesque/aplesque.lisp`[6]. The `(xdotimes)` macro is used for algorithms that iterate across the elements in a function's output array in row-major order. This macro splits an array processing task into appropriately-sized segments to divide between threads. Most CL implementations have been observed to use registers with sizes equal to the sizes of array elements when modifying arrays of elements 8 bits in size or larger. When dealing with arrays that have integer elements smaller than 8 bits, 64-bit registers are usually used to hold the values of elements being processed. This means that when operating on arrays with sub-8-bit integer elements, threads must work upon sub-vectors of elements with a length of `(/ 64 element-size)` to stop elements from being clobbered as multiple threads try to write to the same location in memory.

The `(ydotimes)` macro is like `(xdotimes)` but it doesn't support sub-8-bit elements; in the case of arrays with elements smaller than 8 bits, `(ydotimes)` will perform a task synchronously. April uses `(ydotimes)` in cases where it's impractical to iterate over an output array in row-major order and thus the operation can't be divided into 64-bit segments for small integer elements.

Most of April's array-transforming functions have a similar design pattern. Based on the dimensions of the input array and the arguments passed to the function, the shape of the output array is determined. Then, April iterates over the output or input array using `(xdotimes)` or `(ydotimes)` and performs arithmetic on the row-major index of each output element to determine the corresponding row-major element in the input, and finally copies the elements from the input array to the output array.

4 APPLICATIONS

April has been used for image editing, statistical analysis, web development, terminal interfaces and more. In my experience, while Lisp is unmatched as a general-purpose language APL enables the most intuitive development within its domain of array processing and discrete algorithms. April shares in the interactive features of Common Lisp environments, enabling developers to re-evaluate individual closures in source code, which standard APL environments don't allow - their interactivity is limited to the REPL and to reinterpreting entire discrete functions.

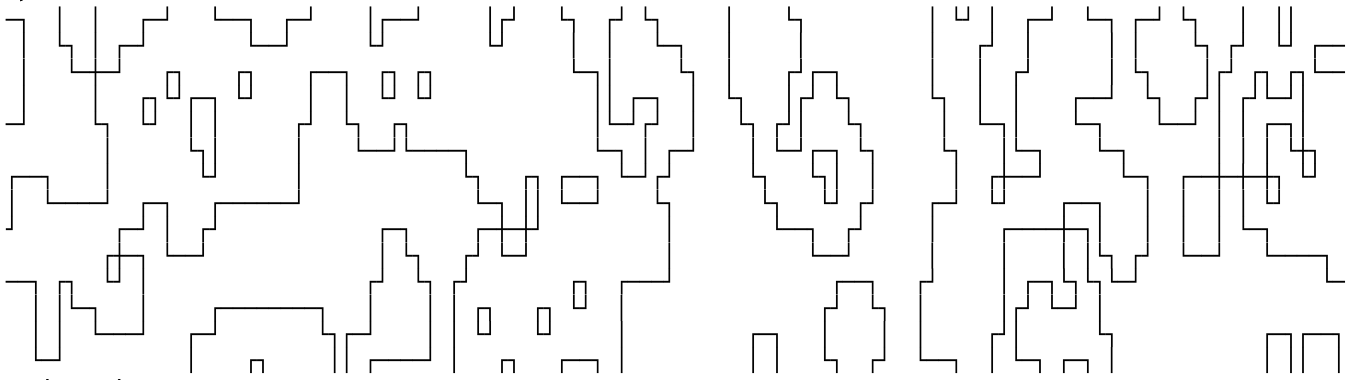
[5]https://lparallel.org/
[6]https://github.com/phantomics/april/blob/master/aplesque/aplesque.lisp

```
* (april "
random ← {⎕IO-⍨?2⍴⍨|α ω}
       ⍝ create a randomized binary matrix
life   ← {⊃1 ⍵∨.∧3 4=+/,1 0 ¯1∘.⊖1 0 ¯1⌽¨⊂⍵}
       ⍝ the classic Conway's Game of Life function
trace  ← {α[;1]⌷⍨⊂α[;2]{1⌈α{⍵×⍵≤≢α}α⍳⍵}{2⊥,⍵}⌺3 3⊢⍵}
       ⍝ use [⌺ stencil] to outline cells according to matrix decoding maps

chars     ← '─  ─   │   │   ┌  ┌   └   └   ┐  ┐   ┘   ┘  '
ints      ←  48 384 144 288 16 416 128 304 32 400 256 176
xEncInts  ←  68 69 257 261 321 324
decodings ← ints⍪⍨{⍵⍴⍨1,⍴⍵} chars~' '

H ← 14  ⍝ height
W ← 112 ⍝ width
I ← 5   ⍝ iterations of life function to perform before printing
M ← ' ' {(α,0)⍪⍵[1;] {(⍪⍸×⍵),⍨⍪α⌷⍨⊂⍵~0} ⍵[2;] {α{⍵×⍵≤≢α}α⍳⍵} {2⊥,(2 2⍴0)⍷3 3⍴(9⍴2)⊤⍵}¨⍳2*9} decodings
  ⍝ map of binary decodings of stencil matrices to box-drawing characters
M⍪← '┼',⍪xEncInts ⍝ add cross-line character values to decoding map

⎕←M trace life⍣I⊢H random W ⋄ (⍕I),' iterations'
")
```



```
"5 iterations"
```

Figure 2: APL evaluated via April implementing the Game of Life function with a convolution kernel to outline cells

4.1 Terminal Graphics

An example using April to generate text-based visuals is shown in Figure 2. This code includes a classic APL function used to implement mathematician John Conway's Game of Life [2]. Rather than displaying the cells themselves, it uses the `[⌺ stencil]` operator to draw boxes around the locations of the cells. This operator can implement convolution kernels, a common technology in computer graphics [3][6]. Convolution kernels are used to blur and sharpen images, for pattern-matching (to detect faces, for example) and in this case to find edges.

The `trace` function uses `[⌺ stencil]` to process 3x3 submatrices of the binary matrix generated by the `life` function, producing 9-bit integers decoded from the binary vector displaced to each submatrix. In other words, matrix `#2A((1 1 0)(1 0 0)(0 0 0))` is displaced to vector `#(1 1 0 1 0 0 0 0 0)` which decodes in binary to 416. This number corresponds to the box-drawing character ┌ stored in the table `M`. For decoded values like 416 that indicate the presence of adjacent cells while no cell is actually present at the position, box-drawing characters are placed in a character matrix of the same shape as the Game of Life matrix.

A more complex variant of this function is used in April's ncurses demo application (`april/demos/ncurses/`[7] in the repository). It's integrated with the `croatoan`[8] CL ncurses binding library to implement a terminal application displaying the cell outlines generated by the code in Figure 2. It also varies each character's background color to reflect the presence or absence of cells in those spaces over time. Building these graphical algorithms in Common Lisp would have required much more code than could fit on one page.

Developing with ncurses has been regarded as a tedious and painful enterprise since using conventional languages means writing dozens of nested loops. April offers the potential to quickly and intuitively specify terminal interface elements and even add some animation and color to keep things fun.

[7]https://github.com/phantomics/april/tree/master/demos/ncurses
[8]https://github.com/McParen/croatoan

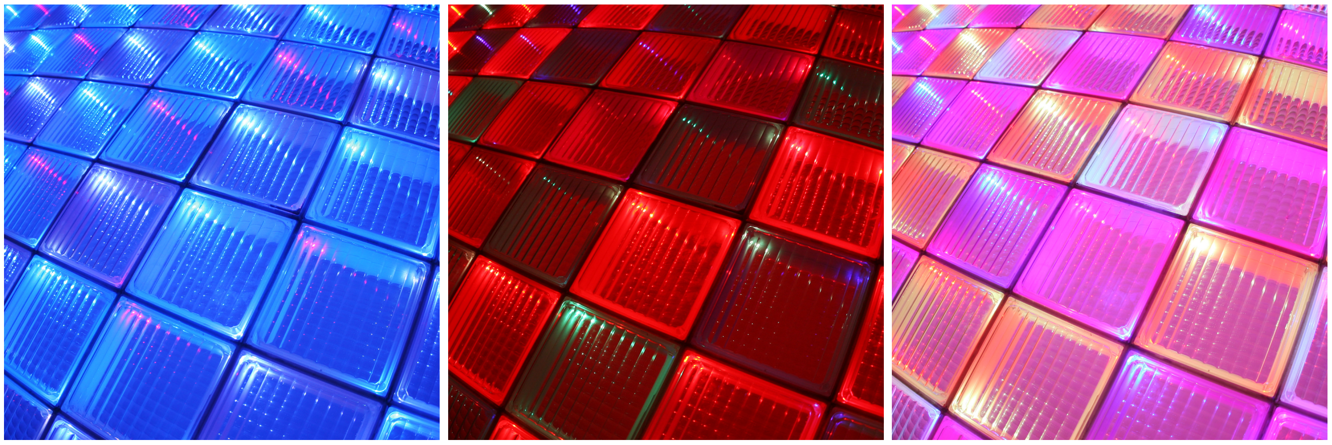


Figure 3: Three color combinations shown on the Bloxl display

4.2 Speaking of Color...

The April compiler’s most prominent application is designing pixel animations for use with a custom LED display built by a hardware startup called Bloxl[9]. Raster graphics are a natural fit for APL; for instance, this code using the `opticl`[10] image processing library is all that’s needed to produce a matrix of the unique colors (one set of RGB values per row) in a .png image:

```
(april-c "{⍉E⊤∪,(E←3⍴2*8)⊥2 3 1⍉⍵}"
         (opticl:read-png-file "/path/to/image.png"))
```

While designing patterns for display on the LED device I experimented with different methods to generate appealing color combinations. This is one of the simpler algorithms I wrote:

```
(april-c "{(⍺×3)⍴1-⍨2*?⍵⍴8}" segment leds)
```

Figure 3 shows three of the resulting color schemes. A vector of random numbers between 1 and 8 of length `segment` is created, 2 is raised to the power of each element, 1 is subtracted from each result, and the output vector is repeated to fill a vector of length `leds` times 3 (3 RGB integer values for each LED). The resulting 8-bit integers will manifest a color series on an LED array. Varying the length of the segment will produce different patterns. In the span of about 10 minutes I wrote this code and used it to build a library of palettes for use with the Bloxl display, generating dozens of RGB vectors and saving the ones that looked good.

April has been a unique boon to the development of Bloxl since building animations often requires custom code for each animation that isn’t used anywhere else. Using a more verbose language, I would be faced with the choice of either placing the custom code directly inside the spec for an individual animation and bloating it by many lines or collecting all custom animation functions in another part of the codebase, adding the cognitive overhead and technical debt of many more functions that are each only used for one task. April makes it possible to express sophisticated custom effects in just a line or two, negating complexity in a way that wouldn’t otherwise be possible.

[9]https://bloxl.co

[10]https://github.com/slyrus/opticl/

5 ACKNOWLEDGEMENTS

Justin Dowdy and Nikolai Matiushev for many bug reports, Kevin Jones, Jérôme Ibanes and Nathan Rogers for steadfast support, Jan Münch, Elias Mårtenson, Marshall Lochbaum and many others on IRC, Matrix and Github for conversation and commentary.

6 CONCLUSIONS AND FUTURE WORK

I consider April to have substantially fulfilled my initial design goal: an alloy of two languages with complementary strengths. April has reduced the time needed to accomplish many tasks and made things possible that weren’t before. At events featuring the Bloxl device I have live coded effects that would have taken hours to assemble using conventional methods.

Upcoming design priorities for April include further speedups through the use of SIMD and even possible GPU acceleration through integration with ArrayFire[11]. April remains slower than Dyalog APL but the compiler has a multifaceted framework for optimization, including its parallelizing macros and a pattern matching system for code that can be implemented in a faster way than the compiler normally would (`april/patterns.lisp`[12]).

REFERENCES

[1] APLWiki.com. Prototype, 2020. URL https://aplwiki.com/wiki/Prototype.

[2] APLWiki.com. John scholes’ conway’s game of life, September 2021. URL https://aplwiki.com/wiki/John_Scholes%27_Conway%27s_Game_of_Life.

[3] Thiago Carvalho. Basics of kernels and convolutions with opencv, 2020. URL https://towardsdatascience.com/basics-of-kernels-and-convolutions-with-opencv-c15311ab8f55.

[4] Adin Falkoff. Apl 360 history. In *Proceedings of the Conference on APL*, APL ’69, page 8–15, New York, NY, USA, 1969. Association for Computing Machinery. ISBN 9781450373784. doi: 10.1145/800012.808128. URL https://doi.org/10.1145/800012.808128.

[5] H. Hellerman. Experimental personalized array translator system. *Commun. ACM*, 7(7):433–438, Jul 1964. ISSN 0001-0782. doi: 10.1145/364520.364573. URL https://doi.org/10.1145/364520.364573.

[6] Roger Hui. Towards improvements to stencil, 2020. URL https://www.dyalog.com/blog/2020/06/towards-improvements-to-stencil/.

[7] Kenneth E. Iverson. Notation as a tool of thought. *Commun. ACM*, 23(8):444–465, Aug 1980. ISSN 0001-0782. doi: 10.1145/358896.358899. URL https://doi.org/10.1145/358896.358899.

[11]https://github.com/arrayfire/arrayfire

[12]https://github.com/phantomics/april/blob/master/patterns.lisp