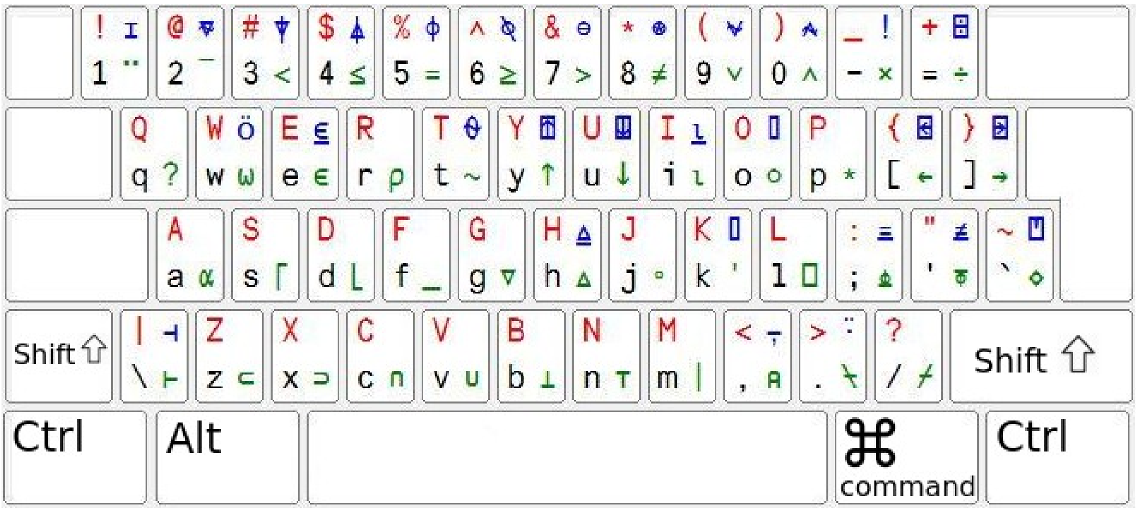
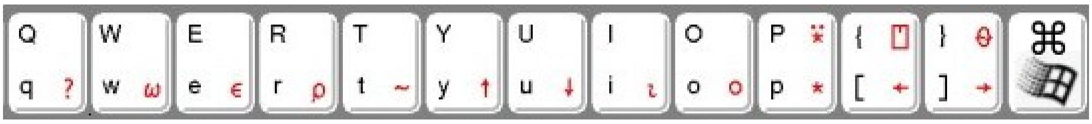
Avril APL
An introduction

Eduardo Costa
Maybe Marcus Santos



Chapter 1

apl keyboard



The command key ⌘ (win key) is represented as # in the comments. To input ω, one should type `#w`, i.e., hold the command key down and strike w. In the figure above, it is the last key on the right-hand side of the QWERTY row of the keyboard.

```
~$ setxkbmap -layout us_intl,apl -option grp:win_switch
~$ rlwrap sbcl --noinform
CL-USER(1): (ql:quickload 'april :silent t)
(APRIL)
CL-USER(2): (in-package :april)

#<PACKAGE "APRIL">
APRIL(3): (april-f "V ← 3.4 8.5 9.2 4.0") ; #[ ←

#(3.4d0 8.5d0 9.2d0 4.0d0)
APRIL(4): (april-f "{+/ω ÷ ρ ω} V") ; #wω, #=÷, #r ρ
6.275000000
6.2749999999999995d0
APRIL(5): (cl-user::exit)
~$
```

On page 1, you gained insights into utilizing Quicklisp for loading April APL into Steel Bank Common Lisp (hereinafter referred to as sbcl). I assume you are already familiar with the installation process of sbcl and the configuration of Quicklisp.

After quickloading the `'april` compiler, I accessed its package using the command `(in-package :april)`. The following command assigned a vectorial quantity to the variable `V`:

```
APRIL(3): (april-f "V ← 3.4 8.5 9.2 4.0")
```

In the sexpr `(april-f "{+/⍵ ÷ ρ ⍵} V")`, the subexpression `{+/⍵ ÷ ρ ⍵}` represents a function that calculates the average of the elements of `V`. The symbol `⍵` denotes the right-hand side argument of the function. The subexpression `+/` inserts the `+` operator between the elements of `⍵`=`V`, resulting in `3.4+ 8.5+ 9.2+ 4.0`. Since `ρ⍵` calculates the length of `⍵`, the entire expression computes the average.

Given that we are within the `APRIL` package, attempting to exit using the sexpressions `(exit)` or `(quit)` proves ineffective for leaving SBCL. Consequently, it becomes essential to execute `(cl-user::exit)`, prefixing the procedure `exit` with the name of the package where it is defined.

To type APL programs, special characters are required. On GNU-Linux, the command below facilitates the inclusion of the APL set of symbols. Whenever you wish to work with APL, open a terminal and execute the `setxkbmap` command.

```
~$ setxkbmap -layout us_intl,apl -option grp:win_switch
```

To type an APL symbol, one needs a prefix-key for producing it. For this purpose, the option `grp:win_switch` selects the command key ⌘, identifiable by the Microsoft Windows flag.

In the comments of page 1, the symbol `#` denotes the command key ⌘ as the prefix. For instance, `#w` (⌘w) signifies holding down the command key ⌘ and then pressing the `w` key to generate the APL symbol ω. Similarly, `#r` (⌘r) indicates that you should keep the window key pressed and strike the `r` key to obtain the APL symbol ρ.

1.1 Star operator

On page 1, you learned how to calculate the average of the elements in a vector. Listing 1.1 shows how to calculate the standard deviation.

Listing 1.1: Standard Deviation

```
APRIL(8): (april-f "V← 4.5 8 3.6 12")

#(4.5d0 8 3.6d0 12)
APRIL(9): (april-f "{((+/(ω - +/ω ÷ ρω)*2) ÷ ρω)*0.5} V")
3.309361721
#(3.3093617209365314d0)

%\caption{Standard Deviation}
%\label{fig:star-operator}
```

In listing 1.1, the star operator `v * n` elevates `v` to the `n-th` power. To obtain the star symbol, you must type ⌘p by holding down the win key and striking `p`. Below, you can see what happens when one calculates `X*2`, where `X` is a vector.

```
APRIL(12): (april-f "X← 3 4 5 8")

#(3 4 5 8)
APRIL(13): (april-f "X * 2")
9 16 25 64
#(9 16 25 64)
```

The expression (ω - +/ω ÷ ρω) subtracts the average of each element of the ω right hand side argument, where the average is calculated by +/ω ÷ ρω (see page 1). Therefore, (ω - +/ω ÷ ρω)*2 is a vector whose elements are represented in equation 1.1 as $(\omega_i - \overline{\omega})^2$.

$$s = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(\omega_i - \overline{\omega})^2} \tag{1.1}$$

The expression /+(ω - +/ω ÷ρω)*2 performs the summation of the elements of (ω - +/ω ÷ ρω)*2.

Let us denote s as the expression ((+/(ω - +/ω ÷ ρω)*2) ÷ ρω). Then s*0.5 elevates this expression to the power of 0.5, which is equivalent to extracting the square root: `{s←((+/(ω - +/ω ÷ ρω)*2)÷ ρω) ◊ s*0.5} V`

1.2 Loading files

The macro `april-load` allows you to write files containing pure APL code, rather than passing strings to the `april-f` macro within Lisp code.

Let us test the program in listing 1.2 on page 4. The program is stored in the file `#P"src/test.apl"`, located in the subdirectory `src`.

Listing 1.2: File src/test.apl – Function definition

```
⍝ This is a comment.
⍝ Use⌘, to introduce a comment.
⍝ Use ⌘l for the output variable ⎕

avg ← {+/ω ÷ ρω} ⍝ ⌘wω, ⌘r ρ , ⌘= ÷
sd ← {((+/(ω - +/ω ÷ ρω)*2) ÷ ρω)*0.5} ⍝ ⌘p *

⎕ ← 'Test functions avg and sd'
⎕ ← 'Test vector V= ' , V ← 4.5 8 3.6 12
⎕ ← 'Average= ' , avg V
⎕ ← 'Standard Deviation= ' , sd V
' '
```

The Lisp command `(april:april-load #P"src/test.apl")` loads the file and executes all APL expressions stored in it.

```
~/apl/apldocs$ rlwrap sbcl --noinform
CL-USER(1): (ql:quickload :april :silent t)

(:APRIL)
CL-USER(2): (april:april-load #P"src/test.apl")
Test functions avg and sd
Test vector V=  4.5 8 3.6 12
Average=  7.025
Standard Deviation=  3.309361721
#\
```

The APL expression `⎕ ← avg V` prints the values on the right-hand side of the arrow ← to the terminal, represented by ⎕. If one wishes to print more than one value, they must separate the sequence of values with a comma.

1.3 Functions with arguments

Arguments of a function must be placed between brackets and separated by a semicolon. In Figure 1.3 on page 6, the arguments of `distance` are `[vx; vy]`.

The caller of a function with arguments must put their values between brackets and separated by a semicolon, following the same order as the arguments appear in the function definition. See how the function `distance` is called in the example below.

```
~/apl/apldocs$ rlwrap sbcl --noinform
CL-USER(1): (ql:quickload :april :silent t)

(:APRIL)
CL-USER(2): (april:april-load #P"src/args.apl")
Test vector V=  4.5 8 3.6 12
Average=  7.025
Sdev=  3.309361721
#\
CL-USER(3): (april:april-f "distance [V; V+5.0]")
10.0
10.0d0
```

1.4 Read macro

To streamline the process of entering lengthy Lisp expressions each time you wish to execute a basic calculation using APL, it is recommended to create a read macro.

In Figure 1.4 on page 7, I present a straightforward read macro. This macro scans a line to retrieve an APL expression and subsequently applies the `april:april-f` function to it.

The read macro is triggered by the `@` character. For instance, typing

```
@{+/ω ÷ ρ ω} 3.0 4 5
```

and then hitting `Enter` creates the sexpr `(april::april-f "{+/ω ÷ ρ ω} 3.0 4 5")` and invoke APL to evaluate it.

Listing 1.3: File src/args.apl – Function with args

```
⍝⌘` gives the concatenating operator ⋄
avg ← {+/ω ÷ ρ ω }

sdev ← {[vx]
  m← avg vx
  ⋄ n ← ρ vx ⍝ commands are concatenated with ⋄
  ⋄ ((+/(vx − m)*2) ÷ n)*0.5} ⍝ ⌘p *

distance ← {[vx;vy] ⍝ args are separated by semicolon
    (+/(vx − vy)*2)*0.5}

⍝ Test functions avg and sdev
⎕← 'Test vector V= ' , V← 4.5 8 3.6 12
⎕← 'Average= ' , avg V
⎕← 'Sdev= ' , sdev V
' '
```

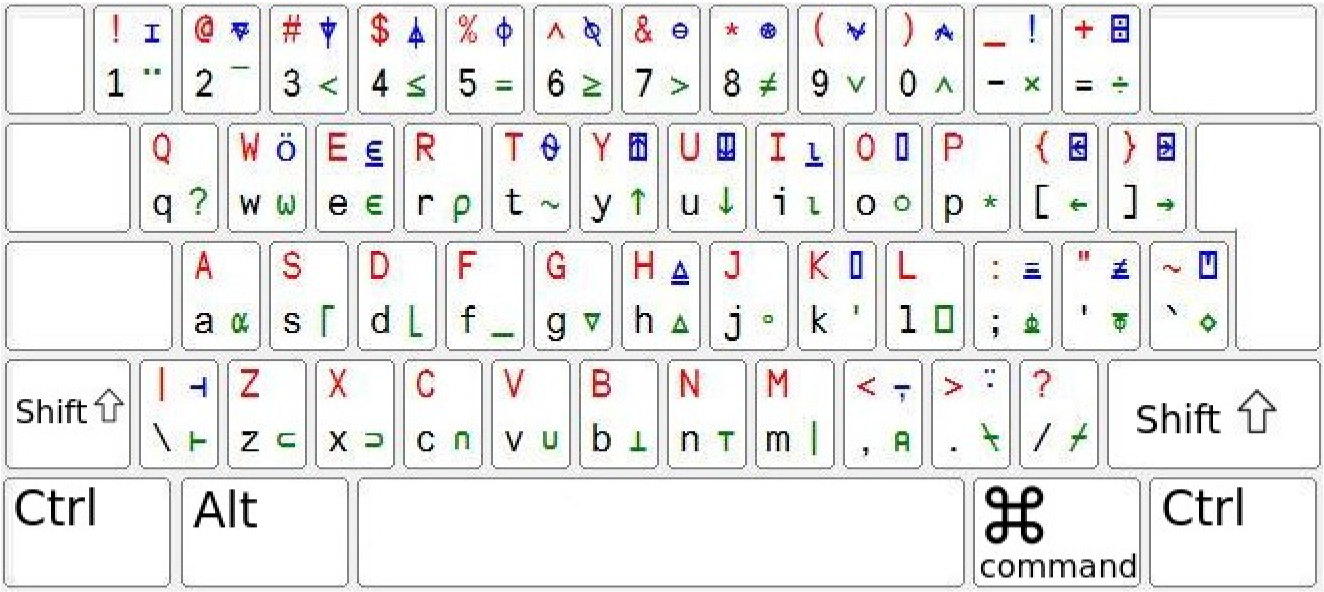
You can enhance the read macro featured in Figure 1.4 and integrate it into the `.sbclrc` initialization file. This ensures that the read macro is readily available every time you access the SBCL REPL.

In case you're unfamiliar with read macros, it's advisable to delve into their understanding and implementation. This knowledge will prove beneficial for maximizing the utility of such macros in your programming endeavors.

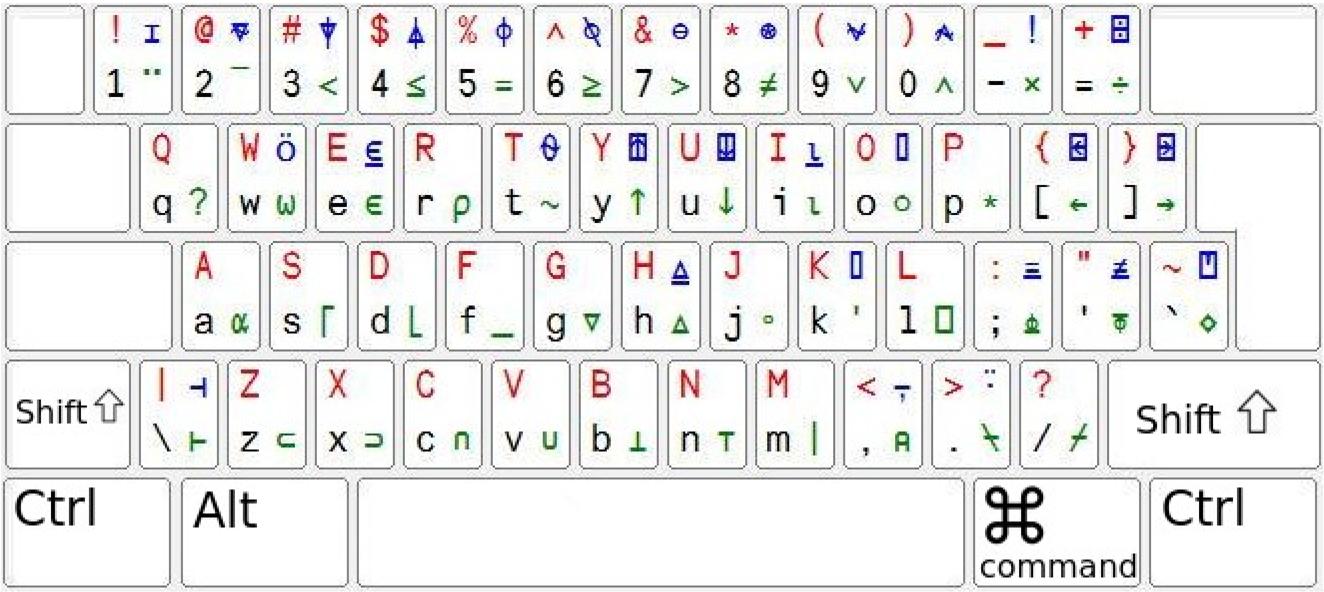
```
~/apl/apldocs$ rlwrap sbcl --noinform
CL-USER(1): (ql:quickload :april :silent t)

(:APRIL)
CL-USER(2): (load "src/rdapl.lisp")

T
CL-USER(3): @{+/ω ÷ ρ ω} 3.0 4 5 8
5.0
5.0d0
```

Listing 1.4: File src/rdapl.lisp – Read Macro for APL

```
(defun read-as-string-until-newline (stream)
  (with-output-to-string (out-stream)
    (loop for next = (peek-char nil stream t nil t)
          until (member next '(#\newline #\tab))
          do (write-char (read-char stream t nil t)
                 out-stream))))

(defun string-reader (stream char)
  (declare (ignore char))
  `(april::april-f
    ,(read-as-string-until-newline stream)))

(set-macro-character #\@ #'string-reader )
```




```
~/apl/apldocs$ rlwrap sbcl --noinform
CL-USER(1): (ql:quickload :april :silent t)
(:APRIL)
CL-USER(2): (load "src/rdapl.lisp")

T
CL-USER(6): @42 {α} 64 ⍝ left argument
42
42
CL-USER(7): @42 {ω} 64 ⍝ right argument
64
64
```




```
CL-USER(8): @42 ⌈ 64 ⍝ max
64
64
CL-USER(9): @42 ⌊ 64 ⍝  min
42
42
CL-USER(22): @×/ 1 2 3 4 5 ⍝ insert × between elements
120
120
CL-USER(23): @⍳12 ⍝  iota
1 2 3 4 5 6 7 8 9 10 11 12
#(1 2 3 4 5 6 7 8 9 10 11 12)
CL-USER(24): @3 4  ρ  ⍳12 ⍝  reshape
1   2   3   4
5   6   7   8
9  10  11  12
#2A((1 2 3 4) (5 6 7 8) (9 10 11 12))
CL-USER(25): @+⌿  (3 4  ρ  ⍳12) ⍝  reduce colums
15 18 21 24
#(15 18 21 24)
CL-USER(26): @+/ (3 4  ρ  ⍳12) ⍝  reduce lines
10 26 42
#(10 26 42)
```

Chapter 2

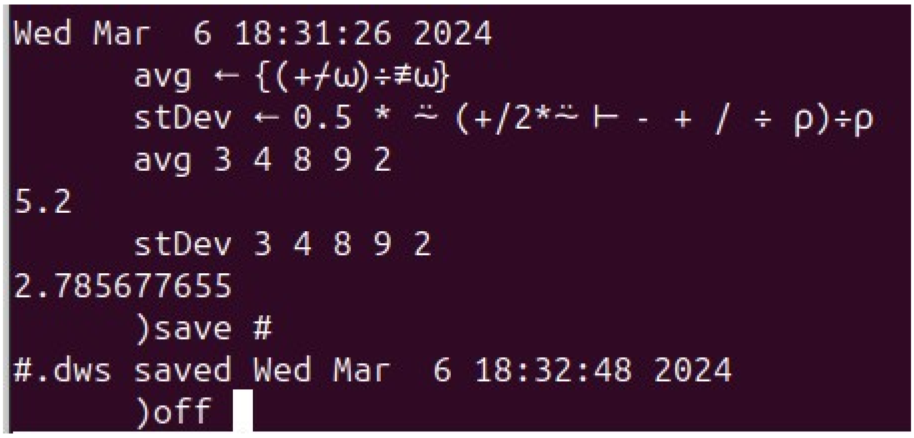
Dyalog APL

You may want to download a fully-featured commercial APL to learn the language. In this case, I recommend a free subscription to Dyalog APL.

In this chapter, I will focus on handling the Dyalog system rather than teaching APL. After all, you can learn APL through one of the many tutorials available on the web.

After requesting a free subscription, download and install the package. Subsequently, initiate Dyalog APL by calling its executable file from the terminal.

```
~/apl$ dyalog
```



```
Wed Mar  6 18:31:26 2024
      avg ← {(+⌿⍵)÷≢⍵}
      stDev ← 0.5 * ⍨ (+/2*⍨ ⊢ - + / ÷ ⍴)÷⍴
      avg 3 4 8 9 2
5.2
      stDev 3 4 8 9 2
2.785677655
      )save #
#.dws saved Wed Mar  6 18:32:48 2024
      )off
```

Figure 2.1: Dyalog APL terminal

In Figure 2.1, I defined `avg`, which calculates the average of a vector, and `stDev`, which produces the standard deviation. After testing these functions, I saved the root `#` workspace. Following that, I entered `)off` to exit APL.

Below is an APL keyboard. The four new APL keys that haven't been covered are circled in red.

The convention of using the sharp (#) character to represent the key sequence that produces the APL symbols is reiterated in the following figure for easy reference.

(#/ ⌿) (#" ≢) (#T ⍨) (#\ ⊢)

(#r ⍴)(#w ⍵)(#p *)

The next time I use the Dyalog ecosystem, I can download the saved workspace and utilize the `avg` and `stDev` functions.

```
~/apl$ dyalog
```

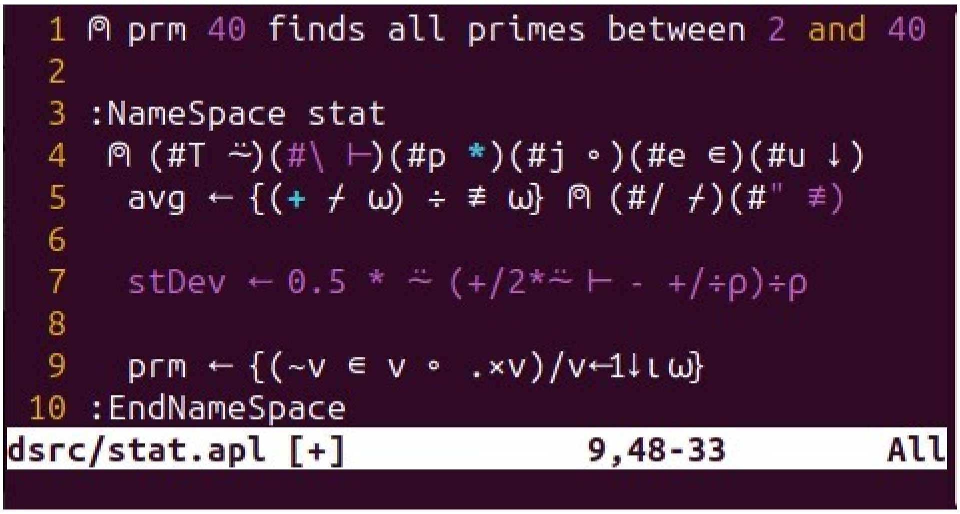
```
      )load #
./#.dws saved Wed Mar  6 18:32:48 2024
      avg 3 4 5 7
4.75
      stDev 3 4 5 7
1.479019946
```

2.1 No source code in plain text

APL was invented by the Canadian programmer Kenneth Iverson well before the widespread availability of text editors. Therefore, it provided its own tools to edit functions and store them in the so-called workspaces.

However, APL vendors have kept their products *"up to date"* by offering facilities to deal with source code files in text format, in a scheme not incompatible with workspaces. In particular, the Link library distributed with Dyalog APL allows linking directories populated with text files to a workspace.

Let's place the text file `stat.apl` into the `dsrc/` directory, as illustrated in the figure 2.2, page 11.



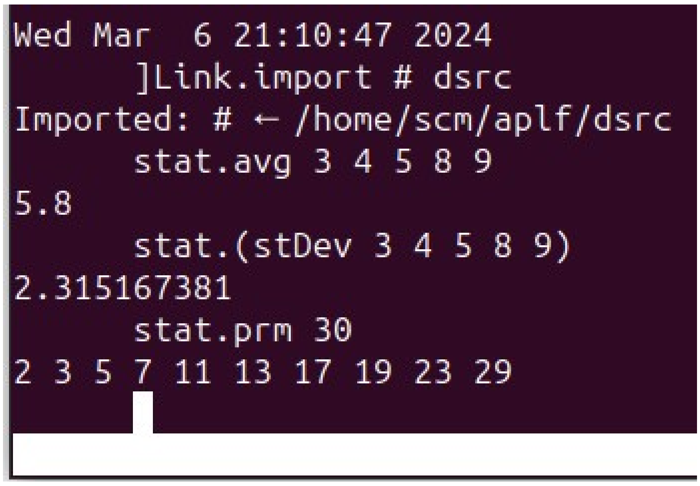
```
⍝ prm 40 finds all primes between 2 and 40

:NameSpace stat
 ⍝ (#T ⍨)(#\ ⊢)(#p *)(#j ∘)(#e ∊)(#u ↓)
  avg ← {(+ ⌿ ω) ÷ ≢ ω} ⍝ (#/ ⌿)(#" ≢)

  stDev ← 0.5 * ⍨ (+/2*⍨ ⊢ - +/÷⍴)÷⍴

  prm ← {(~v ∊ v ∘ .×v)/v←1↓⍳ ω}
:EndNameSpace
dsrc/stat.apl [+]          9,48-33        All
```

Figure 2.2: Source code of the stat workspace

I executed `dyalog` from the `aplf` directory, which contains the subdirectory `dsrc`.

```
~/aplf$ ls
dsrc
~/aplf$ dyalog
```

As seen in Figure 2.3, the command `]Link.import # dsrc` imported the workspace defined in the file `stat.apl`.



```
Wed Mar  6 21:10:47 2024
      ]Link.import # dsrc
Imported: # ← /home/scm/aplf/dsrc
      stat.avg 3 4 5 8 9
5.8
      stat.(stDev 3 4 5 8 9)
2.315167381
      stat.prm 30
2 3 5 7 11 13 17 19 23 29
```

Figure 2.3:]Link.import # dsrc

```
      3 4 ⍴ ⍳12
1  2  3  4
5  6  7  8
9 10 11 12
        ≢ (3 4 ⍴ ⍳12)
3
```

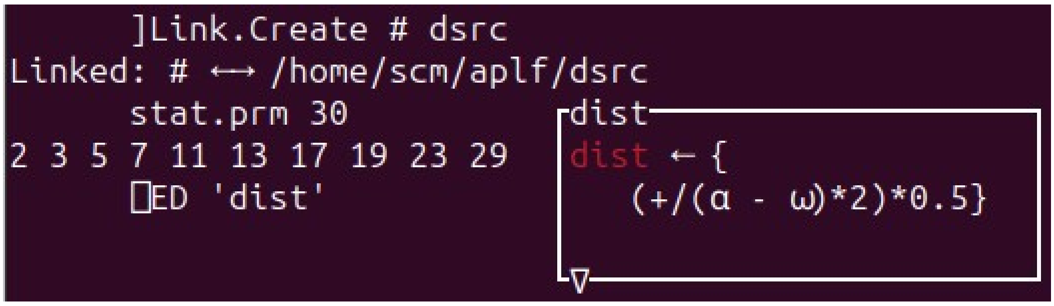


Figure 2.4: ⎕ED 'dist' – Press Esc to quit

2.2]Link.Create

When you enter a Dyalog session, you can create a link between the active workspace and a source directory.

```
      ]Link.Create # dsrc
Linked: # ←→ /home/scm/aplf/dsrc
      stat.prm 30
2 3 5 7 11 13 17 19 23 29
      ⎕ED 'dist' ⍝ Press Esc to quit editor
      3 4 5 6 dist 8 7 4 2
7.141428429
```

After linking `dsrc` to the root workspace `#`, I tested the functions and workspaces defined in `dsrc`, namely, the source code of the `stat` workspace listed in Figure 2.2 on page 11. Finally, I used `⎕ED 'dist'` to define the function `'dist'`, as shown in Figure 2.4 on page 12.

Then, I need to resync in order to store the definition of figure 2.4 into `dsrc`. After that, I can leave Dyalog APL with `)Off`.

```
      ]Link.Create # dsrc
Linked: # ↔ /home/scm/aplf/dsrc
      stat.prm 30
2 3 5 7 11 13 17 19 23 29
      ⎕ED 'dist'  ⍝ Press Esc to quit editor
      3 4 5 7 dist 9 8 3 4
8.062257748
      ]Link.Resync dsrc -proceed
No action required
      )Off
```

When I open a new session, I just need to execute `]Link.Create # dsrc` again to recover all definitions stored in `dsrc`. By the way, when using an external editor to populate a linked directory, you can place only one function or workspace in each file.

```
      ]Link.Create # dsrc
Linked: # ↔ /home/scm/aplf/dsrc
      3 4 5 7 dist 9 8 3 4
8.062257748
      stat.prm 40
2 3 5 7 11 13 17 19 23 29 31 37
```

2.3 ⎕ED

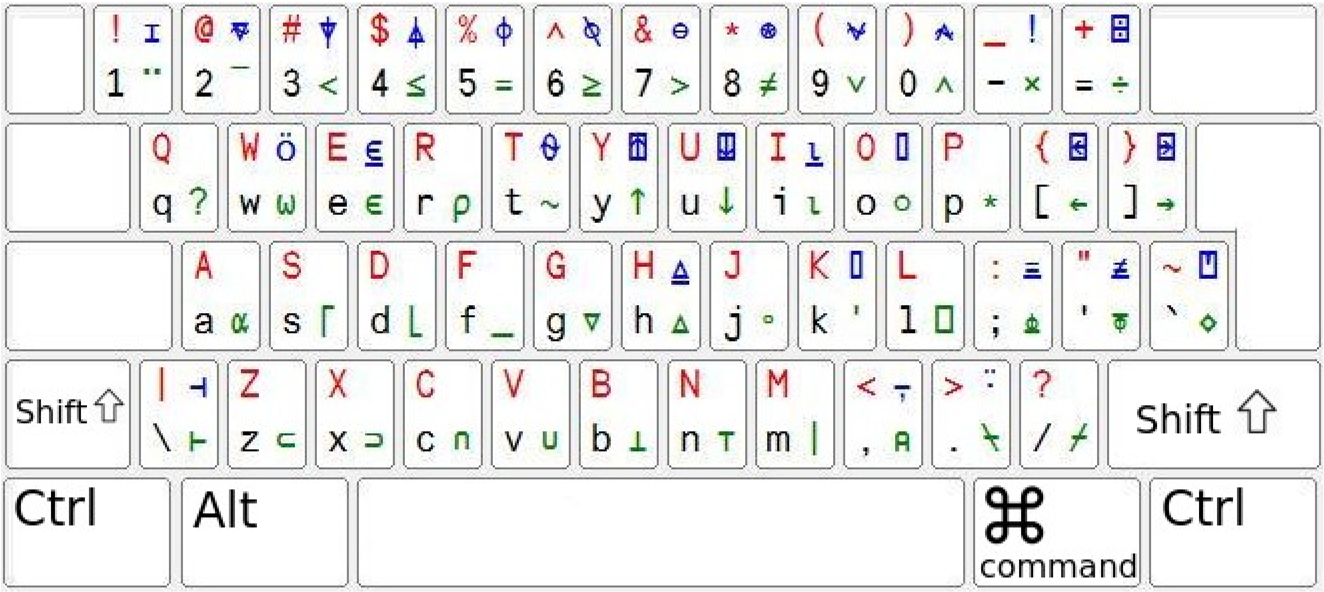
Dyalog provides a structure editor for defining functions. Here's how to request the editing of the `dist` function:

```
⎕ED 'dist'
```

When I start the editor, I get a window like the one shown in Figure 2.4 on page 12. The `Esc` key closes the current editor window after saving any changes to the function being edited. The main commands of the editor are:

- `Ctrl-x j` – toggles between the editor and the session.
- `Esc` closes the current window, having saved any changes.
- `Ctrl-x q` quits the editor.
- `Ctrl-x Ctrl x m` enters the resize loop. Arrows move the window around. `PageUp`(or `Ctrl-f`) and `PageDown` (or `Ctrl-h`) resize the window. Type `Esc` to exit the resize loop.
- `Ctrl-x i` – toggles insert mode.
- `Ctrl-x o` – opens a new line.
- `Ctrl-x z` – toggles the window to full size and back.
- `Ctrl-x t` plus arrows – selects a region.
- `Ctrl-x c` – copies the selected region or cursor line.
- `Ctrl-x l` – toggles line numbering.
- `Ctrl-x ,` – comments the cursor line.
- `Ctrl-x s` – search.
- `Ctrl-x r` – replace.

Keyboard practice




```
      fat ← { ω≤1: 1 ⋄ ω ×∇ ω-1}
      fat 5
120
      42 + ¯12 ⍝  Prefix negative numbers with ⌘2
30
      Δx ← 0.001 ⍝ A nice way of writing small values
      Δx
0.001
     6 + ∘ ≢ 3 4 5 6 ⍝ X f∘g Y → X f (g Y)
10
     6 + (≢ 3 4 5 6)
10
     herName ← 'Tati'
     howMany ← 8
     'Bugs Bunny ', (⍕howMany),' carrots.'
Bugs Bunny 8 carrots.
     'Hello, ', (⍕herName), '!'
Hello, Tati!
     'The factorial of 10 is ', (⍕ !10)
The factorial of 10 is 3628800
```

Chapter 3

Tacit programming

There are two ways of programming in APL

1. Direct function, or dfn
   `{(+/⍵) ÷ ⍴⍵} 3 4 5` → 4 ⍝ Average

2. Tacit programming
   `(+/ ÷ ⍴) 3 4 5` → 4 ⍝ Tacit average

3.1 Direct function

A direct function has the form `{expr}` where `expr` is an APL expression containing the parameters ⍺ and ⍵. Here, the parameter ⍺ represents the left argument, and the parameter ⍵ represents the right argument.

```
      {+/⍵ ÷ ⍴⍵} 3 4 5 8 9 ⍝ Average
5.8
      distance ← {(+/(⍺ - ⍵)*2)*0.5}
      3 4 5 distance 8 10 2
8.366600265
      42 {⍺} 64
42
      42 {⍵} 64
64
```

3.2 Guards

A guard is a boolean expression followed by a colon and an APL expression. The APL expression is calculated when its associated guard is the first one to evaluate to true.

A dfn may contain any number of guarded expressions, each on a separate line. When the guarded expressions are on the same line, they must be separated by ◊ diamond symbols, obtained by typing the ⌘ ` key sequence.

Direct functions use recursion in place of iteration. Normally, a reference to the current function is performed through the function's name, as in Lisp. However, anonymous functions represent self-reference with the ∇ symbol, typed with ⌘g.

```
      {ω < 0: -ω ◊ ω ≠ 0: ω * 0.5} ¯42
42
      {ω < 0: -ω ◊ ω ≠ 0: ω * 0.5} 0
      {ω < 0: -ω ◊ ω ≠ 0: ω * 0.5} 64
8
      fib ← { 1 ≥ ω : ω ◊ (fib ω - 1) + (fib ω - 2)}
      fib 5
5
      fib 6
8
      { ω < 1 : 1 ◊ ω × ∇ ω - 1} 6
720
```

The last example calculates the factorial of 6 recursively. As it defines an anonymous function, recursion is represented by the ∇ triangle symbol.

3.3 Tacit programming

In tacit programming, also called point-free style, functions are defined in terms of implicit arguments. This is in contrast to the explicit use of arguments in dfns (α ω).

```
       (+ ⌿ ÷ ≢ ) ⍳10 ⍝   Mean of the first ten integers
5.5
      (1 ∧ ⊢, ÷ )2.625
21 8
       21  ÷  8 ⍝ Fraction
2.625
      (1 ∧ 0 1∘⊤, ÷ )2.625
2 5 8
      ○ 2
6.283185307
```

The following seven recursive rules transform a dfn (definition with explicit references to the left parameter `⍺` and to the right parameter `⍵`) into tacit form (definition without explicit use of parameters):

fgh-fork `{X f Y} → ({X} f {Y})`

gh-atop `{  f Y} → (    f {Y})`

Agh-fork `{A f Y} → ( A  f {Y})`

commute `{X f A} → (A(⊢f⊣) {X})`

left `{⍺} → ⊣`

right `{⍵} → ⊢`

constant `{A} → ((A)⊣⊣)`

where:

- `X` and `Y` are value expressions with free occurrences of `⍺` or `⍵`.
- `A` is an expression without free occurrences of `⍺` or `⍵`.
- `f` is a functional expression with no occurrences of free `⍺` or `⍵`.
- `⍺` and `⍵` are said to occur *free* in an expression if they are not enclosed in curly braces. In the expression `(⍺+2{⍺+⍵}3)`, `⍺` is free, but `⍵` is not.

I am reproducing below an example of conversion from a dfn to tacit form that I found in a tutorial on the Dyalog portal. My version includes some small corrections. The subexpression that will be converted at each step is enclosed within French quotes.

```
  «{(2+⍺)×⍵÷3}»            ⍝fgh-fork: {X f Y} to ({X} f {Y})
→ («{2+⍺}» × {⍵÷3})        ⍝Agh-fork: {A f Y} to ( A  f {Y})
→ ((2+«{⍺}»)×{⍵÷3})        ⍝left: {⍺} to ⊣
→ ((2+⊣) × «{⍵÷3}»)        ⍝commute: {X f A} to (A(⊢f⊣) X)
→ ((2+⊣) × «{3(⊢÷⊣)⍵}»)    ⍝Agh-fork: {A f Y} to ( A  f {Y})
→ ((2+⊣) × (3(⊢÷⊣) «{⍵}»)) ⍝right: {⍵} to ⊢
→ ((2+⊣)×(3(⊢÷⊣)⊢))        ⍝the tacit form
```

Let's consider another example. The function `{(+⌿⍵)÷≢⍵}` calculates the average of a vector, as illustrated below.

```
      {(+⌿⍵)÷≢⍵}  3 4 5 8 9
5.8
```

```
(f⊢)g(h⊢) to (f g h)⊢  ⍝ ⊢ factoring
(f⊢)g ⊢   to (f g ⊢)⊢  ⍝ ⊢ factoring
⊣ g ⊢   to g           ⍝ dyadic function application
(f(g h)) to ((f g)h)   ⍝ association
```

Figure 3.1: Simplification rules for tacit functions

Let us apply the inference rules to derive the tacit version.

```
≪{+/⍵ ÷ ≢⍵}≫                  ⍝ {X f Y} to ({X} f {Y})
(≪{+/⍵}≫ ÷ ≪{≢⍵}≫)            ⍝ {f Y} to (f {Y})
((+/ ≪{⍵}≫) ÷ (≢ ≪{⍵}≫))      ⍝ {⍵} to ⊢
((+/⊢) ÷ (≢ ⊢))
```

The above program works; however, its expression could be much simpler by applying the rules in Figure 3.1.

```
          ((+/⊢) ÷ (≢ ⊢)) 3 4 5 8 9
    5.8
```

By applying the rule `(f⊢)g(h⊢) to (f g h)⊢`, the expression `((+/⊢) ÷ (≢ ⊢))` becomes `((+/)÷ ≢)⊢`. If we remove the superfluous parentheses, we get the final result `((+/÷≢)⊢)` in tacit form. The only function of the presence of `⊢` in the expression is to ignore an unwanted left argument.

Below, you will find a test of the average function, both in its dfn version and also in the tacit version. Note that all steps of the conversion to the tacit version work as expected. The option `-b` suppresses the banner in a session.

```
~/aplf$ dyalog -b
     {(+/⍵)÷≢⍵} 3 4 5 8 9 ⍝ dfn
5.8
      ({+/⍵} ÷ {≢ ⍵}) 3 4 5 8 9
5.8
      ((+/⊢) ÷ (≢ ⊢)) 3 4 5 8 9
5.8
      42 ((+/÷≢)⊢) 3 4 5 8 9
5.8
      (+/÷≢) 3 4 5 8 9
5.8
```

If you provide a left argument to the `(+/÷≢)` function, APL will generate an error. However, in the `((+/÷≢)⊢)` version, the `⊢` operator will ignore the unwanted argument.

3.4]box

If you want a boxing display for your tacit functions, you can go to your session and type the commands below.

```
     ]Box ON -trains=box
Was OFF -trains=box
      (+⌿÷≢)
      ]box -trains=tree
Was -trains=tree
      (+⌿÷≢)
```

The result is shown in Figure 3.2 below. As you can see, the option `-trains=tree` draws a tree instead of a box.

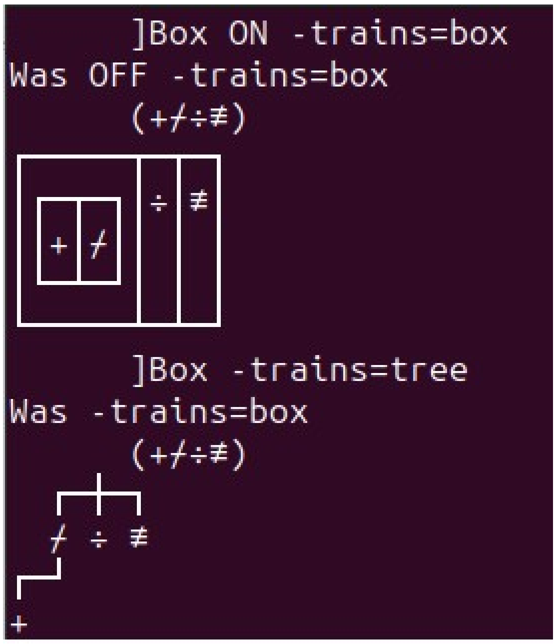


Figure 3.2:]box ON

You can get help for `]Box` by typing `]Box -??` during a session. In particular, the help shows that there are four options for `-trains`, namely, `-trains=box`, `-trains=tree`, `-trans=def` and `-trains=parens`.

```
      ]Box on -trains=def
Was OFF -trains=box
      (32∘+)∘(×∘1.8) ⍝ Fahrenheit from Celsius
32∘+∘(×∘1.8)
      32 ∘ + ∘ (×∘1.8) 100
212
```

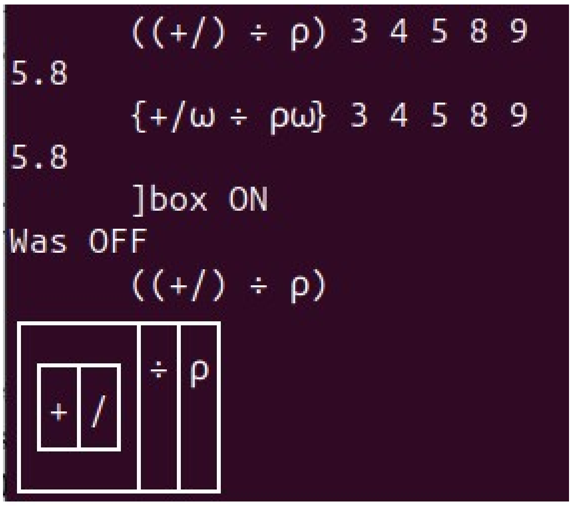
3.5 Tacit average

Let us use the rules on page 17 to transform the anonymous function `{+/⍵ ÷ ⍴⍵}` into its tacit form.

```
≪{+/⍵ ÷ ⍴⍵}≫                  ⍝ {X f Y} to ({X} f {Y})
(≪{+/⍵}≫ ÷ ≪{⍴⍵}≫)            ⍝ {f Y} to (f {Y})
(((+/)≪{⍵}≫) ÷ (⍴ ≪{⍵}≫))     ⍝ {⍵} to ⊢
(((+/)⊢) ÷ (⍴ ⊢))             ⍝ (f ⊢) g (h⊢) to (f g h) ⊢
((+/) ÷ ⍴) ⊢                  ⍝ ⊢ ignores left argument
```

Let us test both the explicit (defn) expression and the tacit function. You may also want to see what the tacit function looks like in the `]box ON` format.

```
      ((+/) ÷ ⍴) 3 4 5 8 9
5.8
      {+/⍵ ÷ ⍴⍵} 3 4 5 8 9
5.8
```

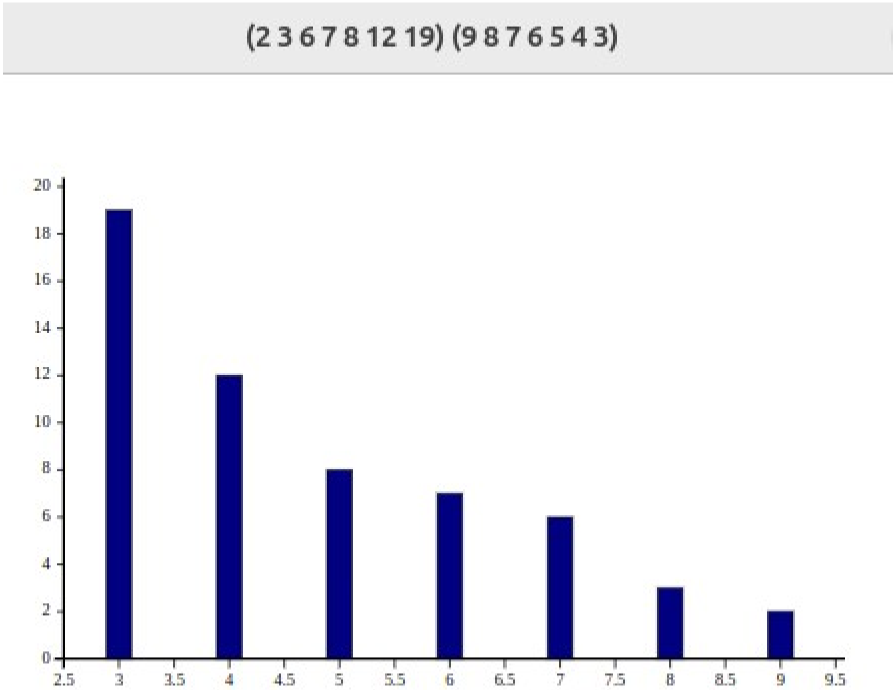


```
      ((+/) ÷ ⍴) 3 4 5 8 9
5.8
      {+/⍵ ÷ ⍴⍵} 3 4 5 8 9
5.8
      ]box ON
Was OFF
      ((+/) ÷ ⍴)
┌─────┬─┬─┐
│┌─┬─┐│÷│⍴│
││+│/││ │ │
│└─┴─┘│ │ │
└─────┴─┴─┘
```

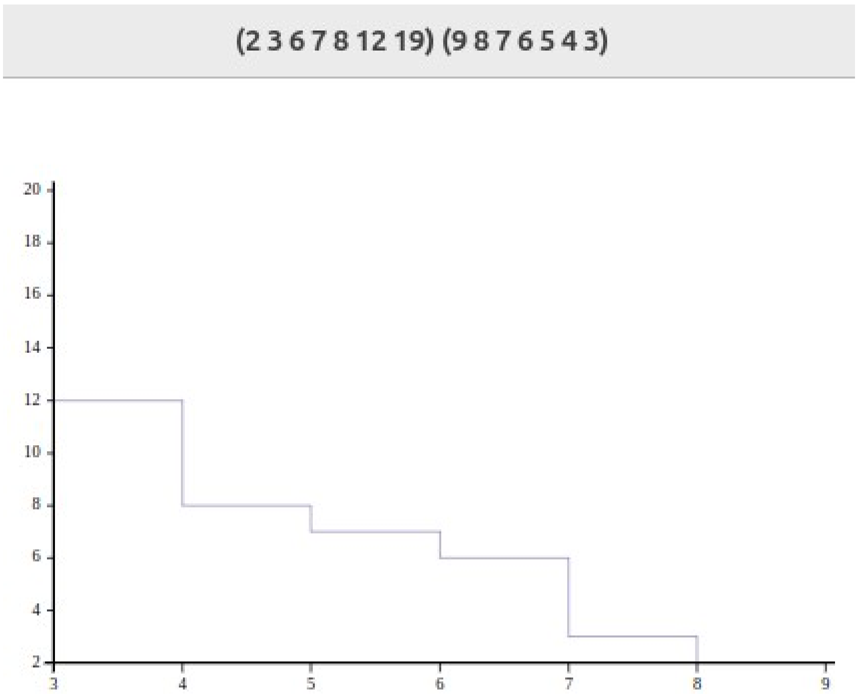
3.6 Plots

To access the Plot options, type]Plot -?? during a session. Below, you will find a few examples of plots.

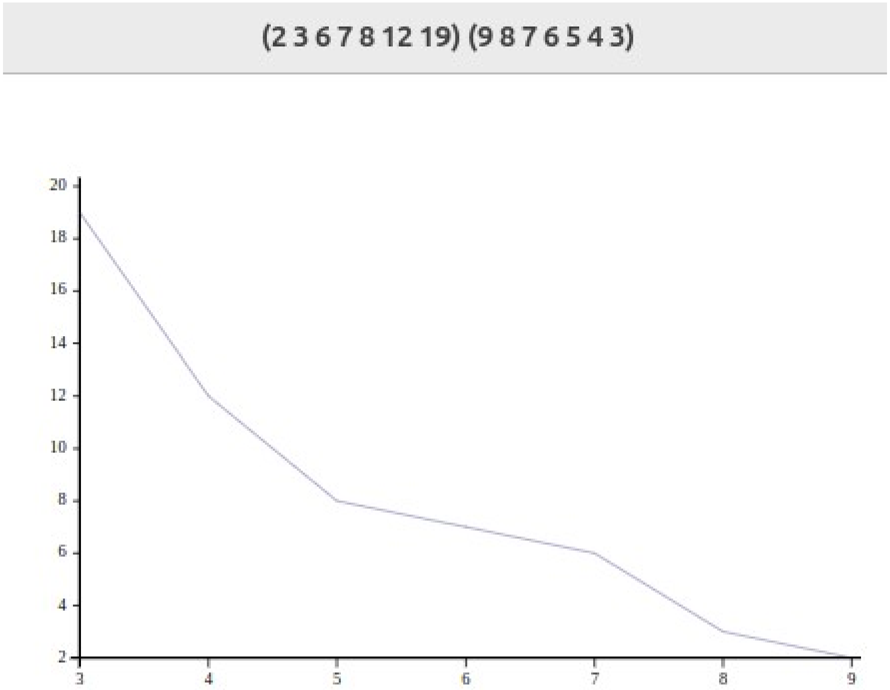
```
]Plot (2 3 6 7 8 12 19)  (9 8 7 6 5 4 3) -type=XBar
```



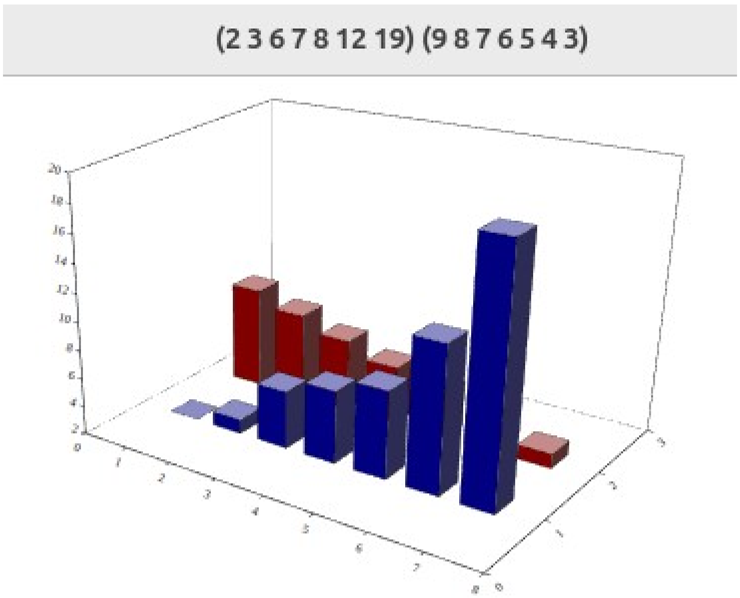

```
]Plot (2 3 6 7 8 12 19)  (9 8 7 6 5 4 3) -type=Step
```



```
]Plot (2 3 6 7 8 12 19)  (9 8 7 6 5 4 3) -type=Line
```




```
]Plot (2 3 6 7 8 12 19)  (9 8 7 6 5 4 3) -type=Tower
```



Chapter 4

Sockets

Let us explore how to receive calculations in Python from Dyalog APL. Start the Python interpreter and create a socket assigned to the variable `srv`.

```
~/apl/apldocs$ python -q
>>>> import socket
>>>> srv= socket.socket(socket.AF_INET, socket.SOCK_STREAM)
>>>> srv.bind(("localhost", 7342))
>>>> srv.listen()
>>>> client, address= srv.accept()
```

APL Side. Send messages.

```
      )copy conga DRC
/opt/mdyalog/19.0/64/unicode/ws/conga.dws
       DRC.Init '' ⍝ Empty string
0  Conga loaded from: conga35_64.so
       DRC.Clt 'my socket' 'localhost' 7342 'Text'
0  my socket
       DRC.Send 'my socket' ('7!= ', ⍕(!7),⎕UCS 10)
0  my socket.Auto00000000
       DRC.Send 'my socket' (⍕ , (2 4 ⍴ ⍳ 8), ⎕UCS 10)
0  my socket.Auto00000001
```

Python Side. Receive and decode message.

```
>>>> aplInput= client.recv(1024)
>>>> print(aplInput.decode("utf-8"))
7!= 5040
1 2 3 4
 5 6 7 8
```

4.1 Messages from Python

Now, let's explore how to send messages from Python to APL.

```
~/apl/apldocs$ python -q
>>>> import socket
>>>> srv= socket.socket(socket.AF_INET, socket.SOCK_STREAM)
>>>> srv.bind(('localhost', 7342)) #Achtung: nested ((oJo))
>>>> srv.listen()
>>>> client, address= srv.accept() # Go to APL to Init conga
```

APL side. Create socket.

```
     )copy conga DRC
/opt/mdyalog/19.0/64/unicode/ws/conga.dws
      DRC.Init ''
0  Conga loaded from: conga35_64.so
      DRC.Clt 'my socket' 'localhost' 7342 'Text'
0  my socket
⍝        At this point, python is free.
⍝        Type the command below, and go to
⍝        the Python side and send a message.
      DRC.Wait 'my socket' 999999
```

Python side. Send the message.

```
>>>> client.send(b"Hello, Dyalog.")
37
```

APL side. Receive the message.

```
      DRC.Wait 'my socket' 999999
      0  my socket  Block  Hello, Dyalog.
```

4.2 Second message from Python

```
⍝         Request a second message from Python.
⍝         Go to the Python side and send the
⍝         second message. Dyalog is waiting.
       msg ← DRC.Wait 'my socket' 999999
```

Python side sends a second message.

```
>>>> client.send(b"Second message from Python.")
36
```

APL side. The message is in a vector, and the text is the fourth element.

```
       msg ← DRC.Wait 'my socket' 999999
⍝         The message is stored in the variable msg
       msg
0  my socket  Block  Second message from Python.
       msg[4] ⍝  It is the 4th element of array msg
 Second message from Python.
```

4.3 Keyboard Practice

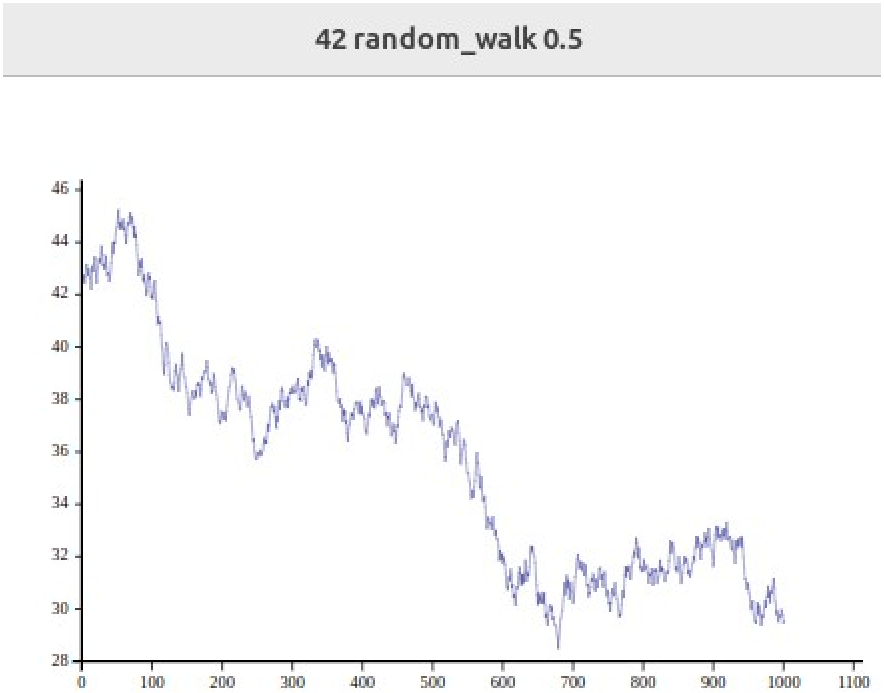
Remember the convention: #\ means pressing ⌘\ and produces the ⊢ symbol. The same rule holds for other keys.

```
      split ← {(~ω ∊ α) ⊆ , ω}  ⍝ #Z ⊆, #t ~, #w ω , #a α
      ' ' split 'She walks in beauty'
 She  walks  in  beauty
      '/' split 'apl/apldocs/apltutorial.pdf'
 apl  apldocs  apltutorial.pdf
      θ    ⍝ #} θ (Empty numeric vector)

      ⊢ 42   ⍝ #\ ⊢ (monadic: I-combinator)
42
      4 ⊢ 5  ⍝ (diadic: K-combinator)
5
      3 ⊣ 4  ⍝ #| ⊣ (KI-combinator)
3
```

4.4 Random walk

In the simulation below, there are 1000 data points. The right argument ⍵ gives the minimum stock value, and the left argument ⍺ represents the initial investment.

```
random_walk ← {ω + | + \α, 0.5-?1000 ρ 0}
      ]plot 42 random_walk 0.5
```



In financial economics, the random walk is used to model share prices. However, beware, as empirical studies have found deviations from this theoretical model in the correlations between short-term and long-term prices.

Chapter 5

Reading from files

In this chapter, we will explore the fundamentals of text input. To delve deeper into this topic, I recommend the chapter titled **Dealing With The Real World** in the tutorial **Learning APL** by Stephen Kruger.

Let's assume you typed the array shown below into the `mat.txt` file. However, beware that there is no line break after the last line of the array.

```
~/apl/apldocs$ cat mat.txt
34 82 56.4
12.5 8 7
```

In the following session, you can observe how the `⎕NGET` procedure reads the entire array into the variable `m`.

```
      ⍎⍤1⊢ ↑(⊃⎕NGET'mat.txt' 1) ⍝ ⌘;⍎ ⌘J⍤
34    82 56.4
12.5   8  7
      m ← ⍎⍤1⊢ ↑(⊃⎕NGET'mat.txt' 1)
      m
34    82 56.4
12.5   8  7
      m[;2]
82 8
      +/m[;2]
90
```

However, the above code has a few intriguing features. It can be inferred that the function `⍎⍤1⊢↑` converts text to an array of numbers. Yet, one might wonder why `⎕NGET` is prefixed by `⊃` and has 1 as the second argument.

The session below demonstrates that using ⎕NGET without the ⊃ prefix and with 1 as the second argument returns a vector with three elements: the text representing the array, the encoding of the file, and the line separator.

```
      ]box on
Was OFF
      ]rows on
Was OFF
      ⎕NGET 'mat.txt'
┌────────────┬───────────┬──┐
│34 82 56.4  │UTF-8-NOBOM│10│
│12.5 8 7    │           │  │
│            │           │  │
└────────────┴───────────┴──┘
      ⎕NGET 'mat.txt' 1
┌─────────────────────────┬───────────┬──┐
│┌──────────┬──────────┐  │UTF-8-NOBOM│10│
││34 82 56.4│12.5 8 7  │  │           │  │
│└──────────┴──────────┘  │           │  │
└─────────────────────────┴───────────┴──┘
      ⊃⎕NGET 'mat.txt' 1
┌──────────┬──────────┐
│34 82 56.4│12.5 8 7  │
└──────────┴──────────┘
      ⍴ ⊃⎕NGET 'mat.txt' 1
2
```

The hydrant ⍎ function evaluates a string, similar to the eval function in sbcl. The ↑V function transforms the nested vector V into an array. The ⊃V function retrieves the first element of the vector V. When provided with a left argument, as in 3 ⊃V, the dyadic function n⊃V returns the nth element of V.

I believe that the information provided in this chapter is sufficient to set you on the right path for reading arrays from files. However, for a more in-depth understanding of some challenging concepts, I highly recommend reading the excellent tutorial **Learning APL** by Stephen Kruger.

Chapter 6

Comma Separator Value

Type the file `emerg.csv` that contains the number of applicants to emergency medicine since 2019. Remember not to introduce a line break after the last line.

```
MD,  DO, IMG
493, 284, 152
490, 282, 171
500, 310, 190
418, 260, 188
270, 200, 189
```

In the United States, there are three categories of candidates for medical residency: Medical Doctors (MD), Doctors of Osteopathic Medicine (DO), and International Medical Graduates (IMG). Each column represents one of these classes of doctors.

In a CSV file, the elements of a line are separated by commas. There is no newline after the last line. Therefore, if you hit Enter after the last line, you will get an error when the file is processed.

```
      ⎕CSV 'emerg.csv' ⍬ 4 1
 493 284 152   MD  DO  IMG
 490 282 171
 500 310 190
 418 260 188
 270 200 189
```

The monadic ⎕CSV function has four arguments. The first argument is the source file name. The second argument is a description of the CSV data, which was left empty in the example. Next, the third argument specifies the column types, and the index 4 indicates that the field is to be interpreted as numeric, but invalid data is tolerated. Finally, the last argument indicates whether there is a head line or not. The value 1 indicates that there is a head line.

```
      ⊃⎕CSV 'emerg.csv' ⍬ 4 1 ⍝ #} ⍬
493 284 152
490 282 171
500 310 190
418 260 188
270 200 189
      +⌿⊃⎕CSV 'emerg.csv' ⍬ 4 1
2171 1336 890
```