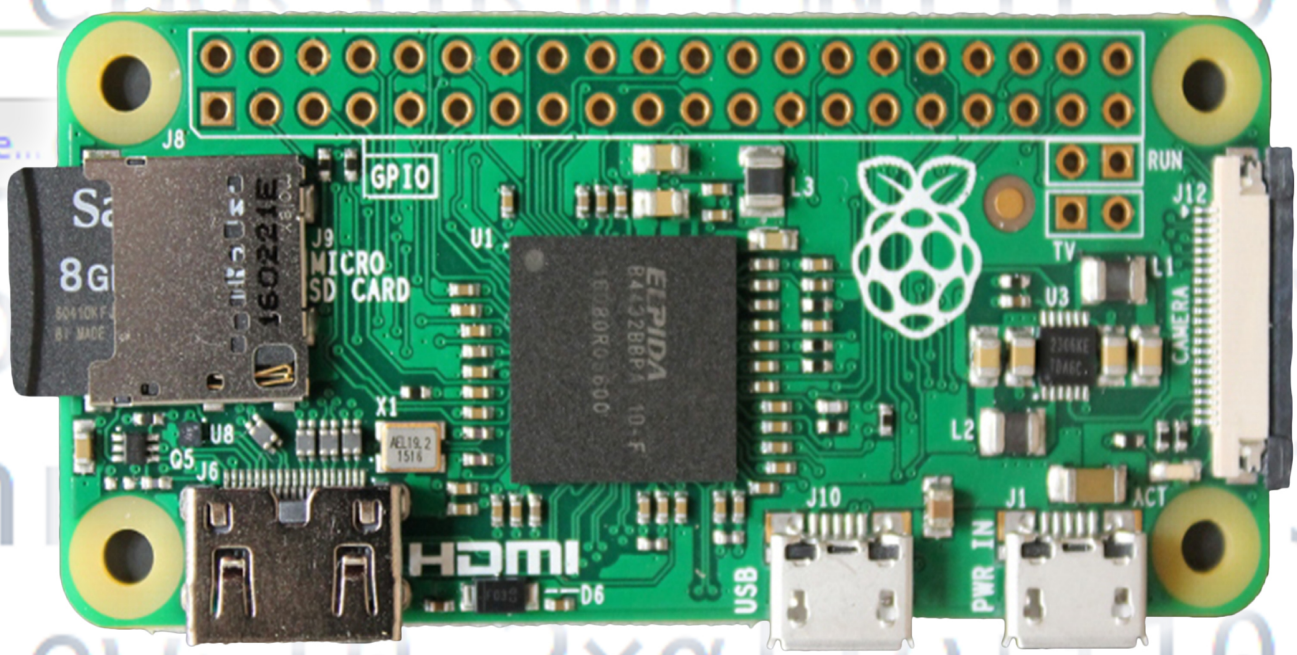
granfire←{⍵<0⌈+⌿neig
granuleThresholds←{s
neighbours←{⍵,-(1⊖⍵),

Learn APL on the $5 Raspberry Pi

Your fast track from ideas to code



ROMILLY COCKING

Learn APL on the $5 Raspberry Pi

Your fast track from ideas to code

Romilly Cocking

This book is for sale at http://leanpub.com/learnapl

This version was published on 2017-02-13

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Romilly Cocking by spreading the word about this book on Twitter!

The suggested hashtag for this book is #learnaplonthe$5pi.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#learnaplonthe$5pi

This book is dedicated to Kenneth Iverson, who gave us APL, and Eben Upton, who gave us the Raspberry Pi.

Contents

Learn APL - book extract

This is an extract from a new Introductory book on APL. It features Dyalog's free implementation on the Raspberry Pi, but you can also use it if you are learning Dyalog APL on Microsoft Windows, Mac OS or other Linux systems.

I hope it will motivate you to take a look at this powerful language and help you to get started.

The book is ideal if you are entering the Dyalog annual problem-solving competition[1].

An early access version of the book is available on Leanpub[2].

It's about 40% complete at the moment and if you buy it now you will get free access to future updates.

The book should be complete by the end of April 2017.

All Leanpub purchases offer an unconditional 45-day money-back guarantee.

There's a slower-paced, more detailed, and much longer textbook called *Mastering Dyalog APL* by Bernard Legrand. It's written for Dyalog for Microsoft Windows, but the language elements (which form most of the content) apply to all platforms.

You can download a free PDF[3] of Bernard's book or buy a print-on-demand version on Amazon[4]

The next chapter will give you a first taste of the language and its development environment.

Have fun!

---

[1] http://www.dyalog.com/student-competition.htm

[2] https://leanpub.com/learnapl

[3] http://www.dyalog.com/uploads/documents/MasteringDyalogAPL.pdf

[4] http://www.amazon.co.uk/Mastering-Dyalog-APL-Complete-Introduction/dp/0956463800/ref=sr_1_1?ie=UTF8&qid=1387290291&sr=8-1&keywords=mastering+dyalog

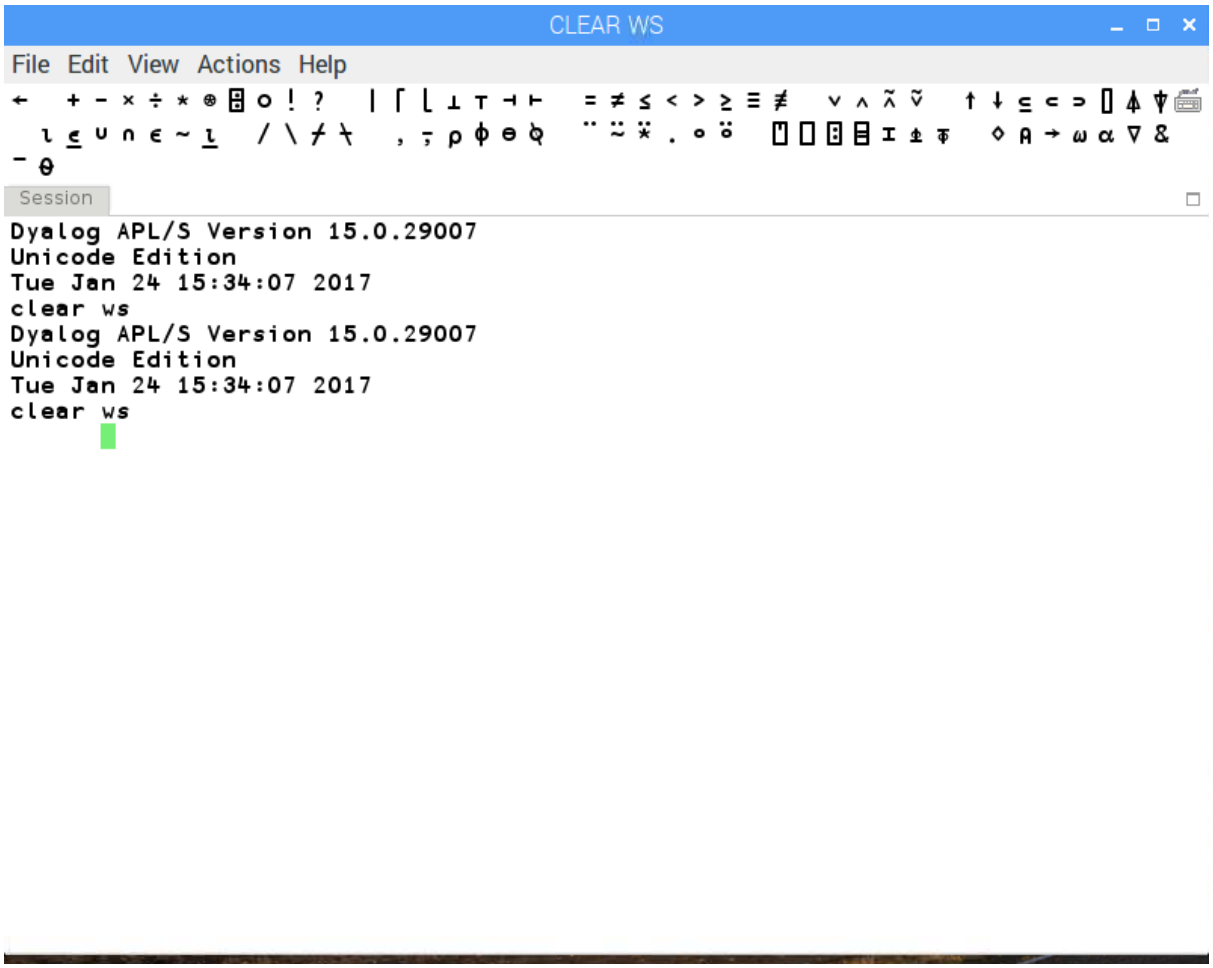
Chapter 1

Getting started

Time to start APL - and start learning this fun, expressive language!

To begin your first APL session on a Raspberry Pi, run *Dyalog* from the *Programming* menu.

You should see a screen like this. That's the *RIDE* (APL's development environment).



Dyalog menu

The RIDE includes a REPL[5] - a Read-Evaluate-Print-Loop.

That means that you can write code and try it out right away. That's a great way to learn a language, and it's also a great way to develop software.

[5] https://en.wikipedia.org/wiki/Read%E2%80%93eval%E2%80%93print_loop

For now, don't worry about the keyboard-like display at the top of the RIDE. You'll find out about that a little later.

In the examples that follow, the code that **you** type is indented by six spaces. Once you start your session you'll see that APL inserts those spaces for you when it's your turn to type.

APL's output is **not** indented, so you can see what you should type and what you should expect APL to output.

So - here goes. Type:

```
      2 + 2
4
```

Try some more:

```
      10 - 4
6
      4 - 5
¯1
```

Note how APL represents negative numbers using a ¯ symbol. The ¯ (called *high minus*) is part of the way you write the value *negative one*.

That's different from the - symbol (called *minus*) which tells APL to do a subtraction.

Plus and minus are *primitive functions* in APL. What about multiply and divide? Of course APL does those too.

APL uses the same symbols that I was taught at school: × for multiply, and ÷ for divide.

Special APL characters

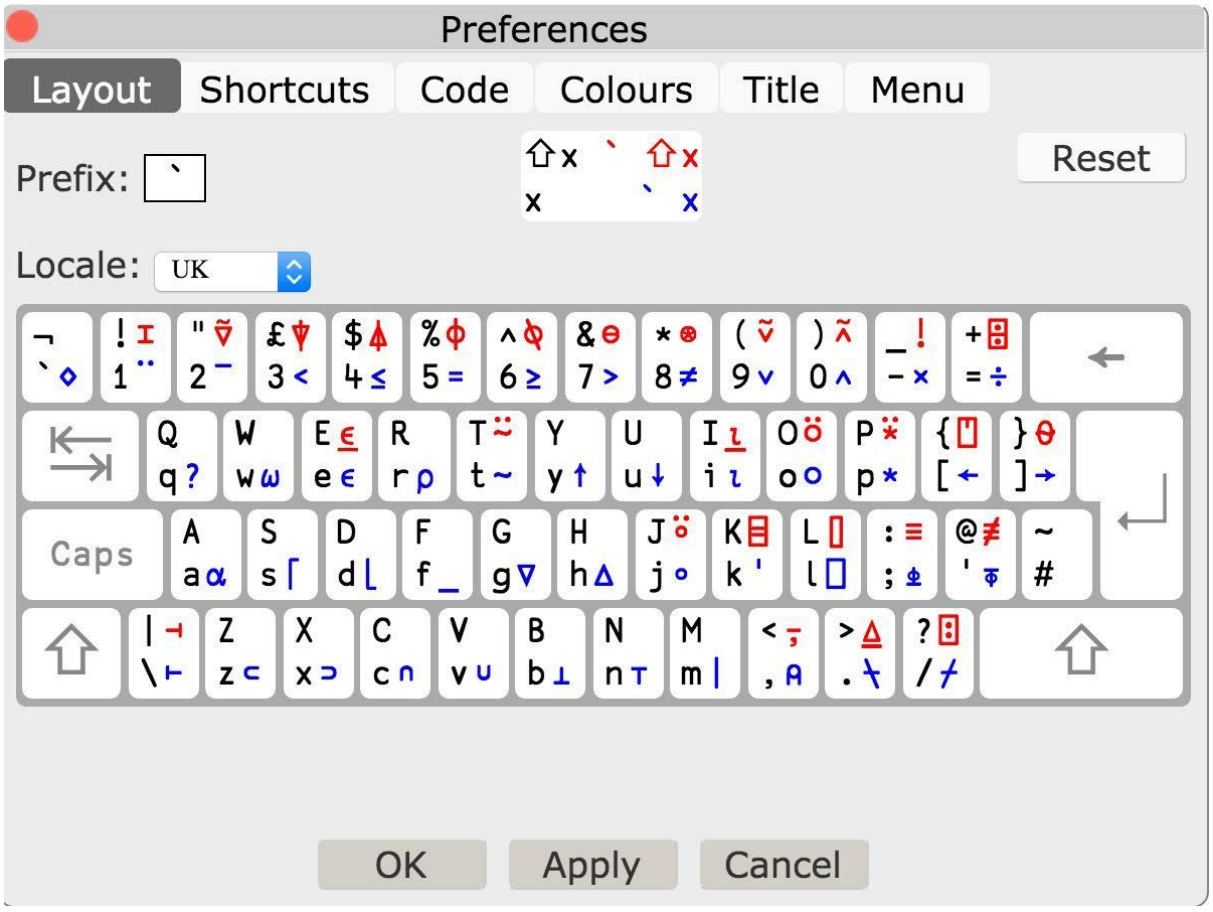
There are *two ways* that you can type the special APL characters when you are writing code.

Remember that keyboard that's displayed at the top of the RIDE window? If you hover your mouse over an APL symbol, the RIDE will tell you its name and what it does.

If you mouse-click on the symbol, the RIDE will type that symbol for you in your APL session.

Alternatively you can type APL symbols directly from the keyboard.

Here's the layout of the UK APL keyboard with APL symbols added.



Dyalog APL keyboard

Use the Windows key to help you type APL symbols.

- Type `Windows-` and you should see a × symbol
- Type `Windows-equals` to get the ÷ sign.
- Type `Windows-shift-equals` to get the ⌹ sign.

Multiplication and division

Time to try out multiplication and division.

```
      2 × 3
6
      5÷2
2.5
      12÷4
3
      4÷3
1.333333333
```

Now for something rather different. Try the experiment below:

```
      1 2 3 + 4 5 6
5 7 9
```

What's going on?

Array programming without explicit loops

APL treats the two lists of numbers as *vectors* and it adds the corresponding elements together.

A lot of calculations need to be done on vectors, and APL's built-in looping makes this really easy.

Try some more examples:

```
      2 3 4 5 × 1 2 1 2
2 6 4 10
      3 4 2 5 - 4 0 ¯1 3
¯1 4 3 2
      120 ÷ 2 3 4 5 6
60 40 30 24 20
      0.1 0.1 0.1 0.1 × 3 5 4 2
0.3 0.5 0.4 0.2
```

That last example works, but it's a bit tedious to type. Fortunately there is an easy shortcut.

```
      0.1 + 3 5 4 2
3.1 5.1 4.1 2.1
```

If you ask APL to multiply (or add, or subtract, or divide) a number on its own and a vector of numbers, APL will use the single number repeatedly. A single number on its own is called a *scalar*.

The repeated use of a scalar when you're adding it to a vector is called *singleton extension*.

What happens if you try to add two vectors of different lengths?

```
      1 2 3 + 4 5 6 7
LENGTH ERROR
      1 2 3+4 5 6 7
     ^
```

APL doesn't know what you want to do, so it treats the expression as an error.

Don't worry about causing APL errors. APL will try to tell you what went wrong, and you won't break anything :)

More about the RIDE

The RIDE has a lot of useful features. We won't cover them all here but one is so valuable when you're learning that we have to mention it.

Try moving your cursor up a few lines to an APL expression that you entered earlier in your session. Now make a change to the line and press enter.

The RIDE will restore the version you typed earlier and enter your changed version at the bottom of your session, so you can make a small change to something without having to retype it all.

Really useful!

Let's go back to APL.

Assigning values to variables

It would be rather tedious if you had to type values in to APL every time you wanted to use them. Fortunately, you can tell APL to remember values you want to use repeatedly.

Suppose you are currently 23. Type:

```
      age ← 23
      age + 10
33
```

The first line you typed told APL to assign the value 23 to a new variable *age*.

In the second line you asked APL to add 10 to your current age, and APL displayed the result.

Notice that APL will display a result if you don't tell it what to do with it.

APL variables can contain vectors as well as scalars.

```
      ages ← 12 23 19
      ages
12 23 19
      ages×2
24 46 38
```

A shortcut to counting

In one of the earlier examples you added the vector 1 2 3 to the vector 4 5 6.

Mathematicians call vectors like that *arithmetic progressions*, and you may well need to use them in your software.

APL has a particularly easy way to create arithmetic progressions using the ⍳ function.

Here are some examples:

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

By default APL starts counting at one. In Chapter 6 you will see a way to get APL to start counting at zero. Some programs are simpler when written that way.

Illuminate your code - use comments

As you get more experienced in APL programming the code you write will get more complex.

Most code is read more often than it is written, so you should consider documenting it using *comments*.

The APL symbol for a comment is ⍝ - often called *lamp* because it's intended to *illuminate* your code.

Whenever the APL interpreter encounters a comment it ignores the rest of that line. You'll find two styles of comment widely used in APL code.

1. A stand-alone comment starts with a lamp symbol. That means that nothing on that line will get executed.
2. An in-line comment follows some executable code on the same line.

The comments should explain what the code does or why it is written that way.

When should you comment? The Three AM rule

I first heard this tip at a conference many yeas ago. It's called *The three AM rule*, and it applies to programming in any language. Here's how I once heard the presenter explain the rule:

> Imagine that you're asleep at home at 3 o'clock in the morning.
>
> The phone rings. And rings. And rings.
>
> You answer it.
>
> 'Hi there. The production system has just fallen over. Can you fix it?'
>
> When you take a look at the application, what style of code do you hope you'll see? That's the way *you* should code.

That's the three AM rule: write code that you, or other developers, would be relieved to see if they are trying to fix a problem at three o'clock in the morning.

If comments would help you or others to read your code at 3 AM, **add those comments**!

Catenate

So far you've seen ways of combining vectors based on arithmetic functions.

There's another common way to create new arrays from old: by joining them together.

In APL, a `,` (comma) is the *catenate* function.

Try it out:

```
      1 2 3, 6 5 4
1 2 3 6 5 4
      1, 4 7 11
1 4 7 11
      5 3 7, 0
5 3 7 0
      2 3, 5 6 8
2 3 5 6 8
```

You can catenate any two vectors, or a vector and a scalar, or a scalar with a vector. Later in the book you will see that there are even more possibilities.

System commands

If you've been working on an APL session for a while it can be useful to check what variables you have created.

APl has a *system command* to do that. System commands in APL don't create values, but they do other useful things. One such command will tell you the names of all the variables you have defined. Try it out:

```
      )vars
age ages
```

In APL, system commands start with an open right parenthesis `)`. The *vars* command tells you the name of the variables that are currently defined.

When you work in an APL session, the variables you create are held in what APL calls the *current workspace*.

A workspace can also contain functions and other things. We'll cover these later in this book.

When you have finished an APL session, you can save the contents of your workspace, and return to it when next you use APL. Try the following commands:

```
      )wsid course
was CLEAR WS
      )save
course saved Sun May 22 17:09:17 2016
```

What did that do?

The first command gave a name to your workspace. Previously it had no name, so APL called it CLEAR WS (a *clear workspace*).

Then you asked APL to save your workspace. It stored it on disk. If you look in your home directory, you should see a file called *course.dws*.

Windows may hide the **dws** extension.

It's a binary file, so don't try to edit it!

Names are useful. If you are working on more than one project, you can have several workspaces, one for each project. Each has a name which will help you find the workspace you want to use for any given session.

You can find out all your local workspaces using the)lib command.

Here's what happened when I ran it:

```
      )lib
.
        startapl.dws
/opt/mdyalog/15.0/32/unicode/ws
        apl2in.dws      apl2pcin.dws    buildse.dws     conga.dws       ddb.d\
ws
        dfns.dws        display.dws     eval.dws        fonts.dws       ftp.d\
ws
        groups.dws      isolate.dws     loaddata.dws    max.dws min.dws ops.d\
ws
        postscri.dws    quadna.dws      rconnect.dws    salt.dws
        sharpplot.dws   smdemo.dws      smdesign.dws    smtutor.dws
        sqapl.dws       tube.dws        tutor.dws       util.dws
        xfrcode.dws     xlate.dws
/opt/mdyalog/15.0/32/unicode/samples/fun
        intro.dws       life.dws        sudoku.dws
```

Wow! Lots of workspaces.

The first two lines show that there is a workspace called startapl.dws in the current directory.

That's a workspace that I saved earlier. It contains the functions and variables used in this course.

The next line shows that there is a directory called `/opt/mdyalog/15.0/32/unicode/ws` which contains 31 workspaces. That directory and the workspaces in it are created by Dyalog during the installation process.

There's also a Dyalog directory called `/opt/mdyalog/15.0/32/unicode/samples/fun` which contains some fun workspaces including John Scholes' implementation of Conway's Game of Life[6] and the game of sudoku[7].

Finishing your session

Once you've finished a session you can close APL down by typing a system command:

```
      )off
```

APL will close down.

If you want to get your work back, restart APL from the *Programming* menu.

You will start a new session with a new clear workspace. To resume your work you must load your saved workspace.

Type:

```
      )load course
./course saved Sun May 22 17:09:17 2016
```

Now you can check that the workspace still contains your variable:

```
      )vars
age ages
      age
23
```

Well done! You've taken the first step to mastering APL. Now try the following simple exercises to consolidate what you've learned.

[6] https://www.youtube.com/watch?v=a9xAKttWgP4
[7] https://www.youtube.com/watch?v=DmT80OseAGs

Exercises

1.1

Create a variable called *income* containing the vector 10000 11570 11000 12550. (Imagine this contain someone's income for the last four quarters of the year.)

Create another variable containing the vector 7250 8345 9547 12650. This might show how much that person spent in each quarter.

Now calculate and display what they saved each quarter. Of course, if they spent more than they earned the savings will be negative.

1.2

Create a variable *weights* containing the numbers 10.2 8,3 7.5 and convert from pounds to kilogrammes.

A pound is roughly 0.45 kilogrammes.

Having fun?

That's the end of the sample content. If you want more, you can buy the early access version of the book at Leanpub[8].

[8] https://leanpub.com/learnapl

Appendix 1

Installing APL on the Raspberry Pi

You can download and install a copy of APL for the Raspberry Pi from Dyalog. It's free for personal use. If you want to use it for work, or create a product that you or others sell, you will need to get a commercial license.

You can install and run Dyalog APL on the following Raspberry Pi versions:

```
Pi 1 Model A
Pi 1 Model A+
Pi 1 Model B
Pi 1 Model B+
Pi 2 Model B
Pi 3 Model B
Pi zero (all versions)
```

The current version (15.0) of Dyalog APL runs on the *jessie* version of Raspbian.

You can find installation instructions here[9].

[9] http://packages.dyalog.com/