Data Processing

Smojo for Data Processing

So far, our programs have been stateless -- they can't remember what happened before. In this lab, you will learn a few new words to help your programs remember things.

Constants

A **constant** is any value that doesn't change in the lifetime of your program. For example,

```
3.14159 constant PI

: circumference
    2 *. PI *.
;

3 circumference .
```

Note: `constant` can only be used outside a word, ie in interpretation mode.

The Data Stack

As you've seen in Chapter 1, Smojo has a special internal data storage area called the **data stack**. It is a temporary storage area to hold the input and output of words.

The data stack is capable of storing up to 512 items. You may "push" (ie, put into the top of the Data Stack) or "pop" (ie, remove from the top of the Data Stack) items.

You can examine the contents of the data stack at any time using `.S` or `BP`.

Smojo has a few words to manipulate items on the data stack:

`drop ( a -- )` removes the top item on the stack and discards it.

`dup ( a -- a a )` duplicates the top item, so the there are two of them.

`swap ( a b -- b a )` swaps the positions of the top two items.

The **stack comments** eg `( a -- a a )` give you an indication of what's happening on the data stack. They are not used by Smojo and only meant to help you, the programmer.

Variables

Variables are temporary storage areas for your program. You can change their value using the words `@` (called fetch) and `!` (called store):

```
0 variable N

: count ( -- )
  N @          ———————— Fetch the value stored in N
  dup . cr     ———————— Make a copy and Print the value,
  1+ N !       ———————— Increment the value and save to N
;

\ Run count 3 times
count count count
```

In the example above the variable `N` is initialised to zero. The word `count` displays the existing value of `N` then increments it. The word `@` fetches data from the variable and the word `!` stores data into it.

Quiz 2.0: Run this example and make sure you understand how it works. Re-write `count` without using `dup`.

An alternative form of a variable is made using the word `=>`. Here is the previous example using `=>`

```
0 => N
: count
  N . cr
  N 1+ ['] N !
;
count count count
```

`=>` makes reading variables easier, since the fetch is not needed. Writing to variable is slightly harder using `=>` because you need to access the variable as an XT then call `!`. In most Smojo code, `=>` is used more often than `variable`, since you're more likely to read a variable than to change its value.

Note: `variable` and `=>` can only be used outside a word.

Use variables sparingly. If you code has lots of variables, it's a good sign you've probably coded the problem the wrong way!

Hashes

A hash is a collection of **key-value pairs**. For example, a person's details - his age, height and name can be stored in a hash this way:

```
# => joe

"name" "Joey" joe #!
"age" 41 joe #!
"height" 170 joe #!

joe .
```

The word `=>` assigns the empty hash `#` to a name (`joe`)
`#!` **stores** key-value pairs into the hash named `joe`.

The word **#@ reads** a single value from a hash given the key. If they key is not present on the hash, the special value **null** is returned. You can check for this using **null?**

```
# => joe

"name" "Joey" joe #!
"age" 41 joe #!

"name" joe #@ . cr
"gender" joe #@ null? . cr
```

Quiz 2.1: Run this example now.

Setters and Getters

You should use **setter** words instead of writing the hash directly:

```
: name! ( # v -- # ) { h v }
  "name" v h #!
  h ;

: age! ( # v -- # ) { h v }
  "age" v h #!
  h ;

: height! ( # v -- # ) { h v }
  "height" v h #!
  h ;

# "Joey" name! 41 age! 170 height! => joe

joe .
```

Quiz 2.2: Write additional setter words to set the person's nationality and gender.

As with setters, you can write **getter** words like so:

```
: name@ ( # -- v ) { h }
  "name" h #@ ;
```

Quiz 2.3: Write getter words to match each setter word.

Tuples

Tuples (also known as Arrays) are a sequential collection of items, addressable with an index, using the words `!!` and `@@`. You can create a tuple using `tuple` or `empty-tuple`:

```
"a" 1 # 3
    4 tuple => tp1
tp1 .tuple

2 empty-tuple .tuple
```

Quiz 2.4: Run this example now. Notice that the empty tuple is filled with `null`s. You can store data into a tuple using `!!` and fetch it using `@@`.

```
"a" 1 # 3
     4 tuple => ts
ts 2 @@ .
ts "hello" 2 !!
ts .tuple
```

Notice that `@@` and `!!` are zero-indexed, (ie, the first item is at index `0`.). You can determine the tuple's length using `tuple-len`:

```
"a" 1 # 3
    4 tuple => ts
ts tuple-len .
```

Sorting Tuples

You can sort a tuple using the word sort `( tuple xt -- tuple )`. The XT needs to be of type `( a b -- f| null )`. In other words, it needs 2 items to compare and should output:

- `null` if the inputs a and b are “equivalent”.
- `true` if a is "larger" than b
- `false` if a is "smaller" than b.

For example, we can sort an array this way:

```
10 3 41 2 6 19 100

     7 tuple   \ create the tuple

' <            \ use < to compare

  sort .tuple  \ sort & print.
```

Quiz 2.5: Run this example now.
Quiz 2.6: Sort the array above by ascending order.

*Example: Argmin and Argmax

Let's define a word `argmax` to calculate the index of the maximum value in an array. If the value occurs more than once, the last index is returned. To do this, we obviously need `argmax ( tup -- n )`, and for it to iterate over all the values of the tuple:

```
\ Draft 0 -- basic ingredients of argmax.

: argmax ( tup -- n ) { xs }

  xs tuple-len [: ;] itimes ;
```

What we need next is to store the current max value (`mv`) and the current max index (`mi`). We also need to output `mi` towards the end.

```
\ Draft 1 -- current max and max index
: argmax ( tup -- n ) { xs }
  0 \ mi
```

```
  0 \ mv
  xs tuple-len [: { mi mv i }
    xs i @@ { v }
    v mv >= -> i v exit |. mi mv
  ;] itimes
  drop \ remove mv from output.
;
```

Finally, we need to correctly initialize **mv** at the start:

```
\ Draft 2 -- initialize mv.
: argmax ( tup -- n ) { xs }
  0       \ mi
  xs 0 @@ \ mv
  xs tuple-len [: { mi mv i }
    xs i @@ { v }
    v mv >= -> i v exit |. mi mv
  ;] itimes
  drop \ remove mv from output.
;
```

Quiz A.1: Suppose we wanted to retrieve the first occurrence of the maximum item instead of the last. How would you amend **argmax**? Make the changes and test out your new **argmax**.

Quiz A.2: Suppose we wanted to write **argmin** to get the index of the last minimum item. Can you amend **argmax** to do this? Try it out and test your code thoroughly.

We can obviously generalize **argmax** and **argmin**, using the word **arg (tup xt -- n)**. The XT in arg's input is the sign used for comparison (**<** or **<=**, **>** or **>=**) each of which will give the two versions of **argmax** and **argmin**:

```
\ Final version
: arg ( tup xt -- n ) ~ { cmp } { xs }
  0
  xs 0 @@
  xs tuple-len [: { mi mv i }
    xs i @@ { v }
    v mx cmp -> i v exit |. mi mv
```

```
  ;] itimes drop ;

: argmax ['] >= arg ;
: argmin ['] <= arg ;
```

This way of programming is common in Smojo. Start out by making your words specific. Generalise when necessary, by using XTs as inputs.

Quiz A.3: Solve Quiz A:1 using `arg`.

Sequences

Sequences are the workhorse of many Smojo applications. They are useful because many programming problems -- especially in data science -- can be cast into the manipulation of sequences.

A sequence is:

- Something with a `head` followed by a `tail`.
- The `tail` is always a sequence.
- The empty sequence is called `nil`, whose `head` is `null` and whose `tail` is also `nil`.
- A sequence may be infinitely long, or of indefinite length.

An example of a finite sequence is a **list**

```
1 2 3 "hey" joe 5 list  => myList
myList . cr
myList .list cr
myList head . cr
myList tail .list cr
```

Quiz 2.7: Run this example and ensure you understand it thoroughly. The word `.list` prints out the entirety of a finite sequence.

You may:

- add to the head of a sequence using `cons`
- join two sequences using `++`

```
1 2 3 4 5
   5 list => myList

myList "hey" cons => myList2

myList .list
myList2 .list
myList2 head . cr
myList myList2 ++ .list
```

Quiz 2.8: Run this example now and be sure you understand how the words `cons` and `++` work.

Quiz 2.9: Add the two items "hello" "world" to `myList2` and check your answer using `.list`.

Quiz 2.10 You can `reverse` a list. Try reversing `myList2` and check your answer.

Quiz 2.11: There is a shortcut to creating small lists, using the words `{{` and `}}` eg:

```
{{ 1 2 3 4 }} .list
```

Try this out now.

Infinite Sequences

Sequences may be of infinite length. To create one, we often start with a simple infinite sequence 1,2,3,...

```
1 2 ... => ns   \ don't .list this!

ns head  .
```

Quiz 2.12: Print the second element of `ns`. Hint: you need to use `head` and `tail`.

Quiz 2.13: Can you reverse an infinite sequence?

Quiz 2.14: Define `zs` as `0` followed by the list `ns`. How would you create `zs`? check your code using `head`. Hint: you need to use `cons`.

You can truncate an infinite sequence using `take` and `take-while`:

```
1 2 ... 12 take .list
```

```
4 13 ... :> 41 < ; take-while .list
```

Quiz 2.15: Run this example now. Do you understand how `take` and `take-while` behave?

Quiz 2.16: Write code that prints out the first 100 elements of the sequence 2,4,6,8...

Quiz 2.17: Write code that prints out the first even numbers less than or equal to 100. Hint: `<=` is less than or equal to, `>=` is greater than or equal to.

*__Quiz 2.18__: Define a word `take-until ( seq xt -- seq )` that "takes" a sequence until the XT returns true. For example,

```
2 4 ... :> 5 > ; take-until .list
```

should print `2 4`. Hint: You need to use `~` , quotations and `take-while`. Also `not` is logical negation, for example, `true not` is `false`.

Higher Order Functions (HOFs)

Higher Order Functions — `map`, `reduce`, `filter` and `zipwith` are words that transform sequences. These words are key building blocks for Smojo programs that process data.

Map

Recall that a sequence is an ordered collection of items. A sequence may be of finite or infinite length.

The word `map ( seq xt -- seq )` transforms any sequence into a new one using an XT. For example,

```
{{ 6 5 4 3 2 1 }}
   dup .list cr
   :> 3 * ; map .list cr
```

Quiz 2.19: Try this example now.

Quiz 2.20: Transform the list {6,5,4,3,2,1} so that it computes the powers of 2 using the elements of the list as indices.

The important thing to note about map is that it is **lazy**. This means that it doesn't output the actual answer, but a promise to do the answer. Let me give you an example:

```
1 2 ...      \ Make an infinite list.
  ' 1+ map   \ adds 1 to each element.
             \ do not .list, since the
             \ result is infinitely long.
10 take .list \ truncate and print.
```

In the example, `map` transforms the infinite sequence 1,2... into the infinite sequence 2,3... Since the transformation is lazy, the `1+` additions are never done until `.list` is called.

Without laziness, `map` would never complete because the sequence is **infinite**.

This property of laziness turns out to be very important in data processing, as it simplifies your code dramatically.

Quiz 2.21: In the example above, instead of 10 take to make the sequence finite, how would you get the first items which are less than 10?

Zipwith

Sometimes you want to combine two sequences using a function to get a third sequence. For example,

```
2 5 ...          \ sequence 1
0 1 ...          \ sequence 2
  ' - zipwith    \ sequence 1 - sequence 2.
10 take .list
```

Like `map`, `zipwith ( seq seq xt -- seq )` is a lazy word. It doesn't do anything until a **reduction** (in thie example, `.list`) is called.

Quiz 2.22: Try out this example. Why was `take` needed?

If the input sequences are of different lengths, the length of the output sequence is the same as the shorter input:

```
{{ 2 3 4 5 4 }} \ sequence 1
0 1 ...         \ sequence 2
  ' - zipwith   \ sequence 1 - sequence 2
.list
```

Quiz 2.23: Do we need `take` in this example?

Quiz 2.24: You are asked to build a word `double ( seq -- seq )` that doubles the values in a sequence. Write and test this word using `map` then using `zipwith`. Which approach is best?

Filter

Use `filter ( seq xt -- seq )` to select items from a sequence satisfying a given criterion. The XT should output a flag (ie, `true` or `false`):

```
2 4 ... \ even numbers
1 3 ... \ odd numbers
  ' * zipwith \ pairwise multiply them
:> 256 mod 0= ; filter \ selects items
                      \ divisible by 256.
10 take .list
```

Like `map` and `zipwith`, `filter` is also lazy. This means that nothing is done until the results are requested for by `.list`.

Quiz 2.25: Write a short program to get the first 300 cubes divisible by 13. Hint: use `map`, `filter` and `take`.

Thinking in HOFs

HOFs are so powerful because their output can be re-used. For example, to expand on our previous code:

```
2 4 ...
1 3 …
' * zipwith :> 256 mod 0= ; filter
          => myData
\ You can do a lot with myData:
\ Print the first 10 items.
myData 10 take .list
\ first items less than 700,000.
myData :> 700000 < ; take-while .list
```

```
\ first 10 items divisible by 3.
myData :> 3 mod 0= ; filter 10 take .list
```

Quiz 2.26: Run this example and see the results.

Important: So far, our examples all use "simple" sequences like 1 2 .... In real applications, you'd instead be working with sequences of data drawn from a file (eg, CSV or binary) or other data sources (eg, a network or databases). For now, try to get a good grasp of using HOFs for simple sequences.

Quiz 2.27: Write a word that outputs a sequence of cube differences. Ie, the kth term of $\frac{1}{(1+k)^3} - \frac{1}{k^3}$. Hint: you need to use part of your answer from Quiz 2.25 and use `tail` and `zipwith`.

Quiz 2.28: Override the usual arithmetic operations `+ - *` and `^` so that they work on sequences. For example:

```
: + ( seq seq -- seq ) ['] + zipwith ;
\ -- TEST!
1 2 ...
3 4 ...
    + 10 take .list
```

Quiz 2.29: Extend your operators in Quiz 2.28 to handle mixed operations between sequences and scalars. Eg,

```
3 {{ 1 2 3 }} *
```

should output the sequence `{{ 3 6 9 }}`. Hint: you need the word `seq? ( x -- f )` that outputs `true` if x is a sequence. Be sure your words work which ever position the scalar is in, and also when only scalars are involved. For now, use only integers for scalars.

Reduce

A **reduction** is a way to get actual results from a lazy calculation. We use the word `reduce ( seq xt -- )`. Here are some examples

```
\ Prints all elements in a sequence
: .list ( seq -- )
  ['] . reduce
  cr
;

{{ 1 2 3 4 5 6 7 8 }} .list
```

Quiz 2.30: Try out this example.

Exercises

Complete these exercises before proceeding. Answers are at the end of this Lab:

Q1: Write a word `minimum ( seq -- n )` to find the minimum value of a sequence. Does your word make any assumptions about the maximum value?

Q2: Write a word `reverse` that reverses any finite sequence. Can this word be Lazy? Is it possible to reverse infinite sequences?

Q3: Given a sequence $s = (s_0, s_1, s_2, \dots)$, write a word `sum3 ( seq -- seq )` which produces a new sequence:

$sum3(s) = (s_0 + s_1 + s_2,\ s_1 + s_2 + s_3, \dots)$

Q4: Generalize `sum3` to `sumN ( seq n -- seq )`, which is able to sum any length window `n`. `sumN` should take the initial sequence and `n` as input.

Q5: Write a word `zip ( seq seq -- seq )` that produces 2-tuples from each pair of elements of the input sequences.

Q6: Write a word `take ( seq n -- seq )` that produces a sequence of the first `n` elements of an infinite sequence. Hint: You may have to use `zipwith` and `take-while ( seq xt -- seq )`

Q7: Is:

```
1 2 ...  255 take
    :> 31 mod 0 = ; filter
```

the same as:

```
1 2 ... :> 31 mod 0 = ; filter
     255 take
```

explain why.

Q8: Write a word that performs this sum over N terms:

$$\frac{6}{1*3}+\frac{7}{3*7}+\frac{8}{9*11}+\frac{9}{25*15}+\ldots$$

Q9: Write a word `consa ( seq seq -- seq )` that `cons`es the second (finite) list to the first.

Example: Polynomial Arithmetic

A polynomial in x is an expression of the form:

$$P(x)=\sum c_k x^k$$

So, all we need to determine a polynomial are the sequence of numbers c_k . This means that sequences are perfect for representing polynomials. Infinite sequences are a better choice since they can easily represent polynomials of arbitrary length. For example:

- The trivial polynomial $P(x) \equiv 0$ can be represented by the sequence: `0 0 ...`
- The polynomial $P(x) \equiv x$ can be represented by the sequence: `0 0 ... 1 cons 0 cons`
- You can **add** two polynomials by simply adding their elements in piecewise fashion. You've done this in Quiz 2.28. This takes advantage of the fact that the `zipwith` HOF is lazy.
- You can **multiply** a polynomial with a scalar k by multiplying every element with k. You have done this in Quiz 2.29. Again this takes advantage of the laziness of `map`.

Quiz 2.31: Write a word that multiplies a polynomial $P(x)$ (ie, an infinite sequence) by x. Hint: How would the coefficient of x^k change after multiplication by x?

Quiz 2.32: Can we multiplying two arbitrary polynomials using HOFs?

Application: Characteristic Functions

A characteristic function is a polynomial of the form :

$$P(x) = \prod a_k - x$$

For example, $P(x) = (1 - x)(3 - x)(45 - x)$ is a characteristic function. How can we find the coefficients of x^k?

Here's a solution:

1. We create the polynomial $P_0(x) \equiv 1$. Recall, this is represented by the sequence { 1 , 0 , 0 ... }.
2. We multiply each factor $(a_k - x)$ to P_0. We call this word `mul-factor`.
3. We can represent the numbers a_k as a list, then run a reduction on it using `mul-factor`.

Here is a partial solution, with all of the code except for `mul-factor`:

```
\ multiplies seq by ( a - x )
: mul-factor ( seq a -- seq )
  \ .... You need to write this code!
;
: char ( seq -- seq )
  0 0 ... 1 cons  \ P0 = 1
  swap ['] mul-factor reduce
;

\ Test!
{{ 1 3 45 }} char 4 take .list
```

Quiz 2.33: Complete the code for `mul-factor`. Hint: Use your answer from Quiz 2.31 and Quiz 2.28.

Quiz 2.34: In the code listing we had to `4 take`. Can you get rid of this unsightly hack? Ensure that your answers work for arbitrary, finite sized inputs. What is the coefficient of x^5 if the input is {{ 1 2 3 4 5 6 }} ?

I hope this lab gives you some insight into what's possible with lazy HOFs and infinite sequences. Things get more interesting of course when we work with real datasets instead of purely mathematical examples like the ones we've encountered thus far.

Learning Points

Let's review what you learned in this chapter:

- You've learnt about constants, variables, hashes, tuples and sequences.
- Sequences have a special place in Smojo since they can be lazy. Paired with lazy HOFs, this allows us to express and solve tricky programming problems quite simply.

Answers to Exercises

Q1: Write a word `minimum ( seq -- n )` to find the minimum value of a sequence. Does your word make any assumptions about the maximum value?

Answer:

```
: minimum  ( seq -- n ) { xs }
    xs empty? -> "Empty List!" abort |.
    xs head xs tail ['] min reduce
;
\ -- Test
1 2 ... 10 take minimum .
10 9 ... 10 take minimum .
1 2 ... :> 1 swap /. ; map
     10 take minimum .
```

Q2: Write a word `reverse` that reverses any finite sequence. Can this word be Lazy? Is it possible to reverse infinite sequences?

Answer:

```
: reverse ( seq -- seq )
    nil swap ['] cons reduce
;
1 2 ... 10 take reverse .list
```

It isn't possible for `reverse` to be lazy. Since the whole sequence has to be known prior to reversal, it would be impossible to `reverse` an infinite sequence.

Q3: Given a sequence $s = (s_0, s_1, s_2, \dots)$, write a word `sum3 ( seq -- seq )` which produces a new sequence:

$sum3(s) = (s_0 + s_1 + s_2,\ s_1 + s_2 + s_3, \dots)$

Answer:

```
: +s ( seq seq -- seq ) ['] + zipwith ;
: sum3 ( seq -- )
    dup tail dup tail
    +s +s
;
1 2 ... sum3 10 take .list \ 6 9 12 ...
1 3 ... sum3 10 take .list \ 9 15 21 ...
```

Q4: Generalize `sum3` to `sumN ( seq n -- seq )`, which is able to sum any length window `n`. `sumN` should take the initial sequence and `n` as input.

Answer:

```
: +s ( seq seq -- seq ) ['] + zipwith ;
: SUMN ( seq n -- seq ) 1- { n }
     n [: dup tail ;] times
     n ['] +s times
;
1 2 ... 2 sumN 5 take .list \ 3 5 7 9 11
1 2 ... 3 sumN 5 take .list \ 6 9 12 15
1 2 ... 4 sumN 5 take .list \ 10 14 18 22
```

I've used `times ( n xt -- )` that executes the given `xt` , `n` times. If `n`=0, the XT is not executed at all.

Q5: Write a word `zip ( seq seq -- seq )` that produces 2-tuples from each pair of elements of the input sequences.

Answer:

```
: pair ( a b -- a,b )
    2 tuple ;

: zip ( seq seq -- seq )
    ['] pair zipWith
;

1 2 ... 3 take
3 5 ... 3 take zip

dup .list   \ 3 tuple objects
dup head .tuple  \ (1,3)
dup tail head .tuple  \ (2,5)
dup tail tail head .tuple  \ (3,7)
```

Q6: Write a word `take ( seq n -- seq )` that produces a sequence of the first `n` elements of an infinite sequence. Hint: You may have to use `zipwith` and `take-while ( seq xt -- seq )`

Answer:

```
: take ( seq n -- seq ) { n }
    0 1 ... [: n < ;] take-while
    ['] drop zipwith
;

1 2 ... 5 take .list  \ 1 2 3 4 5
2 9 ... 1 take .list  \ 2
```

Q7: Is:

```
1 2 ...  255 take
    :> 31 mod 0 = ; filter
```

the same as:

```
1 2 ... :> 31 mod 0 = ; filter
     255 take
```

explain why.

Answer:

They are different. The first gives integers <= 255 that are divisible by 31. The second gives the first 255 integers that are divisible by 31.

Q8: Write a word that performs this sum over N terms:

$$\frac{6}{1*3}+\frac{7}{3*7}+\frac{8}{9*11}+\frac{9}{25*15}+\ldots$$

```
: sum ( seq -- n ) 0 swap ['] +. reduce ;
: series ( n -- n ) { n }
  n 6 +.
     3 n ^
     n 4 *. 3 +.
          *.
     /.
;
: series ( n -- seq )
  0 1 ... ['] series map swap take
;

0 series sum . cr  \ 0
1 series sum . cr  \ 2.0
2 series sum . cr  \ 2.33333333
3 series sum . cr  \ 2.41414141
4 series sum . cr  \ 2.43636363
5 series sum . cr  \ 2.44286136
```

Q9: Write a word `consa ( seq seq -- seq )` that `cons`es the second (finite) list to the first.

Answer:

```
: consa ( seq seq -- seq )
     reverse ['] cons reduce
;

1 2 ... 3 take
10 9 ... 5 take
    consa .list   \ 10 9 8 7 6 1 2 3
```