Data Transformations

The Data Transformation Approach to Programming

Every programming language has its own “approach” or “style” of programming. In C, it’s loops and pointers; in Java you can’t do anything without objects. Lisp is about manipulating list-like structures. And so on.

In Smojo, your programs are best thought of as a **data transformation pipeline**.

The idea is you have a something (usually a data structure like a hash or sequence) and your program transforms this step by step from one form into the final form, which can then be displayed or somehow consumed.

This way of looking at programming is fundamentally different from the mindset needed for languages like C, which emphasise looping.

To illustrate this difference, let’s suppose you were given this programming challenge:

```
Find the sum of the first 10 squares
not divisible by 5
```

- A Smojo programmer would immediately think of this as a **series of data transformations**.
- A C or Python or Java programmer might instead think of this as **writing a loop**.

Both approaches are valid of course. But the data transformation approach has many advantages if it is suitable to be used.

To see this, let's write a solution in C:

```c
#include <stdio.h>

// Sum first 10 square not divisible by 5

int main(){

    int sum = 0; // accumulator
    int counter = 0; // keeps track of the no. of squares
    int i = 1; // generates the squares
    int sq; // temp variable for square.

    while(1){

        sq = i*i;
        if(sq % 5 > 0){
            sum += sq;
            ++counter;
            if(10 == counter){
                printf("Sum is:%d",sum);
                break;
            }
        }
        ++i;
    }

    return 0;
}
```

Listing 1: Solution using the looping approach.

This solution illustrates many common features and problems of loop-style thinking:

- There are a number of **state variables**: `sum`, `counter`, `i` and `sq` that are hold important information.
- The loop needs some **halting criterion**, in this case a check that `counter` has reached `10`.
- **Problem #1**: The variables must be **updated in the right order**. For example, if the `counter` increment were moved past the halting criterion, that would not result in the correct answer - see the figure on the right. This isn't easy to check and needs to be simulated.
- **Problem #2**: You can't easily deduce the problem statement just by glancing at the code. This makes code correctness harder to check. For example, the "buggy" code on the right might be interpreted as "Find the sum of the first 11 squares not divisible by 5".

```c
sq = i*i;
if(sq % 5 > 0){
    sum += sq;
    // ++counter;
    if(10 == counter){
        printf("Sum is:%d",sum);
        break;
    }
    ++counter;
}
++i;
```

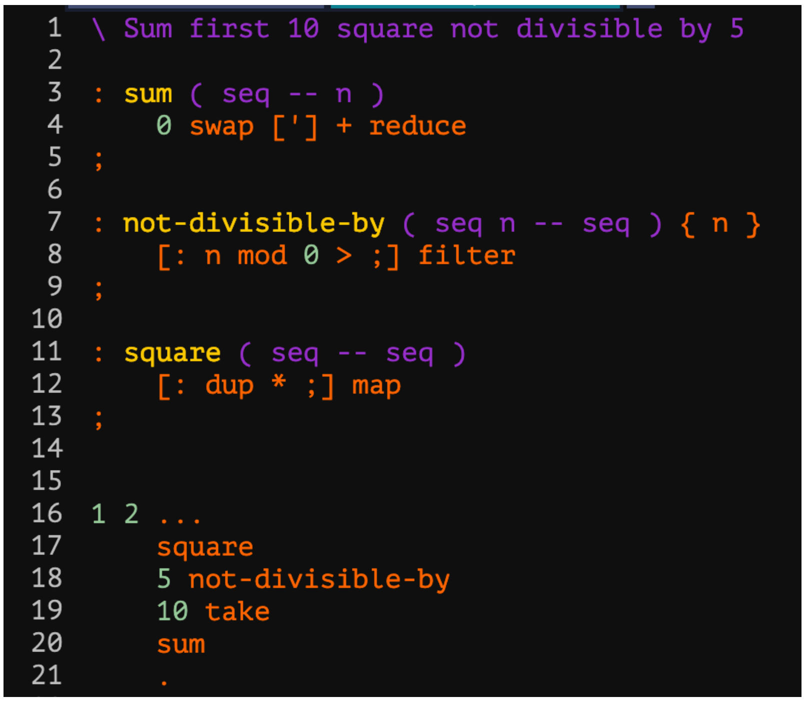
Moving the counter update results in an erroneous calculation.

Let's see how the data transformation approach compares with this.

A recipe for the solution might be:

1. Begin with all integers 1,2 ...
2. Find their squares
3. Remove those divisible by 5
4. Get the first 10 elements
5. Sum them
6. Print the result

Each step here in this recipe is a data transformation over an infinite sequence of integers. Here's the code in Smojo:



```
\ Sum first 10 square not divisible by 5

: sum ( seq -- n )
    0 swap ['] + reduce
;

: not-divisible-by ( seq n -- seq ) { n }
    [: n mod 0 > ;] filter
;

: square ( seq -- seq )
    [: dup * ;] map
;


1 2 ...
    square
    5 not-divisible-by
    10 take
    sum
    .
```

Listing 2: Solution using the data transformation approach

There are several features of this program:

- There are **no** state variables. This has many positive implications for code reuse, optimisation and parallelisation.
- There is **no** halting criterion.

- There are **no** variable updates, so the question of the correct order is moot.
- You **can** easily deduce the problem statement just by glancing at the code. This makes code correctness easy to check.

Each of the words in this program define a transformation of some kind:

- `square` converts a sequence of integers into their squares using `MAP`
- `not-divisible-by` converts a sequence of integers into a new sequence with the desired property using `FILTER`
- `take` truncates a sequence, making it finite
- `sum` transforms a sequence of integers into a single integer using `REDUCE`
- `.` transforms an integer into nothing.

The key ingredients behind this style of programming are:

1. **Sequences** both finite and of indeterminate length,
2. **XTs** or **lambdas** as they might be known in other programming languages. In our solution, quotations `[:` … `;]` are used to create XTs.
3. **Higher-order functions** and **lazy evaluation** of sequences. In this example, `MAP` and `FILTER` are both lazy. They don't perform their task immediately, but rather only when their elements are read.
4. A way to convert a sequence into a single value. This conversion is called a **reduction** and we use the word `REDUCE` to do this.

Benefits of Using Data Transformations

The data transformation approach is greatly beneficial:

- **Debug easier** since the transformation is done in stages, not mixed up as it is in a loop. You can print the results at any stage individually. This is very helpful especially in larger programs.
- **Amend the program** easier since this means just changing a single stage in the process. Useful for example, if your program requirements change. Or if you want to add functionality.
- **Reuse code better**, since words like sum are now a transformation rather than being mixed up in a loop.

The big takeaway is that this approach lets you tame complexity, especially for transformation tasks.

Automatic code generation, which is one of the two Big Ideas I describe in this book is about converting Smojo into another language (in our case, C).

Counting Characters

To illustrate the point better, let's solve a tutorial question (Q4, under "Text and Strings" section of the Smojo Tutorial):

```
Write a word COUNT that takes a
string and a character as arguments
and returns the number of occurrences
of the character in the string.
```

The official answer provided uses a `BEGIN`…`UNTIL` loop, and the programming is hard to understand, especially since it does not use locals, but just the stack and spare

```
: COUNT ( "s" char -- d )
        0 0 >R >R
        begin
                over over R> indexofc2
                R> 1 + >R
        dup 1 + >R -1 = until
        R> drop R> 1 -
;
```

Listing 3: The official solution for count. Uses a loop. Hard to understand!

stack to store temporary variables:

Here is the same solution in idiomatic C, using pointer arithmetic:

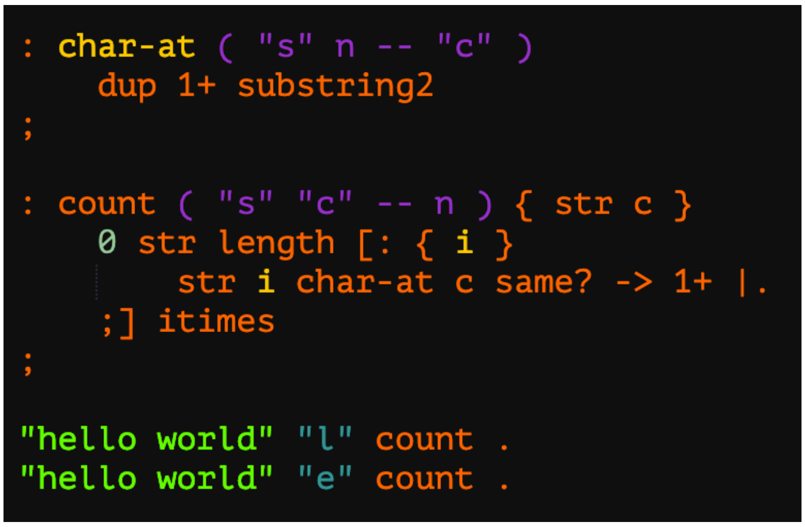
```
#include <stdio.h>

int count(char* str, char c){

   int n = 0;
   while(*str != 0){
      if(*str == c) ++n;
      ++str;
   }

   return n;

}

int main(){

   char* str = "Hello World";

   printf("Count=%d\n",count(str,'l'));
   printf("Count=%d\n",count(str,'e'));

   return 0;

}
```

Listing 4: Solution in C using a loop.

This is better than the “official” Smojo version, because it reads the string character by character. Here is a simpler Smojo version that reads the string character-by-character, using a `ITIMES` loop:

Recall that `ITIMES ( n XT — )` executes the `XT` `n` times, each time feeding the XT the iteration number, starting from zero. For example,

```
5 ' . itimes
```



```
: char-at ( "s" n -- "c" )
    dup 1+ substring2
;

: count ( "s" "c" -- n ) { str c }
    0 str length [: { i }
        str i char-at c same? -> 1+ |.
    ;] itimes
;

"hello world" "l" count .
"hello world" "e" count .
```

Listing 5: A “better” solution but still using a loop.

will print the numbers 0 1 2 3 4. There is also `TIMES ( n XT — )` that simply executes the `XT n` times, without feeding it the iteration number.

The C version will have to be completely revised if a Unicode string is used. The Smojo version will work for unicode.

What the C and improved Smojo versions have in common is that they need to maintain a variable — the accumulator — in the case of C it is named **n**, and in the case of Smojo it is anonymous and just an item on the stack.

How might we apply the data transformation approach?

Start with a recipe for counting occurrences:

1. Split the string into a sequence of characters
2. Remove all characters not equal to the target
3. Find the length of the resulting sequence
4. Print the result

Here’s a solution using this recipe:

```
\ Finds length of a sequence
\ (not needed actually, since this is built-in)
: seqlen ( seq -- n )
    0 swap [: drop 1 + ;] reduce
;

\ Splits a string into a seq of chars
: string>seq ( "s" -- seq )
    "" tokenize array>seq
;

\ finds number of occurrances of CHAR in STRING.
: occurrance ( "string" "char" -- n ) { c }
    string>seq  \ sequence of chars
    [: c same? ;] filter \ sequence of char = CH
    seqlen
;

\ Test code
"hello world" "l" occurrance .
```

Listing 6: Using the data transformation approach

The word **TOKENIZE ("string" "pattern" – tuple)** splits a string into an array of strings. The **pattern** acts as a delimiter and can be a literal string or a

```
: get-month ( "s" -- n ) ucase
    "JANFEBMARAPRMAYJUNJULAUGSEPOCTNOVDEC" swap indexof 3 / 1 +
;

"23-Aug-2023  12:35" "[-\s+\:]" tokenize => xs

xs 2 @@ int
xs 1 @@ get-month
xs 0 @@ int
xs 3 @@ int
    ymdh .
```

Listing 7: Using TOKENIZE. You should research Java Regular Expressions

Java **regular expression**. The output of **TOKENIZE** is a tuple. Listing 7 shows a typical example using **TOKENIZE**.

ARRAY>SEQ (tuple – seq) converts a tuple (ie array) into a sequence. We will use **TOKENIZE** and **ARRAY>SEQ** a lot in the data transformation approach.

The pattern of transformations should be familiar by now:

- Begin with converting your data structure into a sequence. In the previous example we did not have to, but here I've used `string>seq`.
- You use a series of `MAP`s and `FILTER`s to shape your initial sequence,
- It always ends with a reduction using `REDUCE` to get a final answer.

Also, note that the transformations themselves might be composed of transformations. For example `string>seq` first converts a string into a tuple then from a tuple into a sequence.

Exercises

1. Write a word `CADDDR ( seq -- * )` to find the fourth item in a sequence. Eg, `{{ 1 2 3 4 5 6 7 }} CADDDR .` should display `4 ok`
2. How would you access the fourth element in a tuple called `xs` ? Write the code.
3. Write a word `SEQLEN ( seq -- n )` to determine the length of a sequence. Eg, `{{ "hello" 1 2 4 "world" }} SEQLEN .` should display `5 ok`. Hint: You need to use quotations `[: ... ;]` and `REDUCE`
4. Write a word `SUM ( seq -- n )` that sums the numbers in a list. Eg: `{{ 1 2 3 4 }} SUM .` should display `10 ok`. *Hint #1* : follow the same pattern as (Q3). *Hint #2*: `['] +` is the same as `[: + ;]` but it is faster.
5. Write a word `PRODUCT ( seq -- n )` that finds the product of numbers in a list. Eg: `{{ 1 2 3 4 }} PRODUCT .` displays `24 ok`. *Hint*: This is a small modification of your code from Q4.
6. What is the `PRODUCT` of this list: `{{ 493921 512345 657839 712465 2134567 54123476 }}` Do the results make sense? How can you fix this?

7. Do you see any commonalities between `PRODUCT` and `SUM`? Can you write a new word that captures these commonalities and use it to re-write both `PRODUCT` and `SUM`?

8. Write a word `.SEQ` that prints elements of a list, in a single column. *Hint*: this follows the same pattern as Q3 - Q5.

9. Write a word `EVEN ( seq -- seq )` that extracts only even integers from a list of integers. Eg, `{{ 1 2 3 4 5 6 9 10 12 13 }} EVEN` should result in the sequence `{{ 2 4 6 10 12 }}` *Hint*: you need to use `[: ... ;] FILTER` and modulus `MOD`.

10. Write a word `SQUARE ( seq -- seq )` that squares the integers in a list. Eg, `{{ 1 2 3 }} SQUARE` should give `{{ 1 4 9 }}`. Hint: Use `[: ... ;] MAP`. Does this work with `1 2 ... SQUARE` ? How would you test?

11. Re-do Q10 but using `REDUCE` instead of `MAP`. *Hint*: You need to use `NIL` and `CONS`. What is the big difference between using `MAP` and `REDUCE`?

12. Write a word `PP ( # -- )` that neatly prints out the key/value pairs in a hash. *Hint*: You need to use quotations, `REDUCE`, `#KEYS`, and `#@`.

13. Given a sequence $s = (s_0, s_1, s_2, \dots)$, write a word `sum3 ( seq -- seq )` which produces a new sequence: $sum3(s) = (s_0 + s_1 + s_2,\ s_1 + s_2 + s_3, \dots)$

14. 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.

15. In the original example, we found "the sum of the first 10 squares not divisible by 5" - can you write a program that instead find the sum of the **next** 10 such squares? Amend the solution already given in Listing 2. Also find a solution in C by amending Listing 1.