Smojo Basics

The basics of the Smojo programming language

Smojo is a new programming language for Data Science and Artificial Intelligence. It is fast, easy to learn and use.

In this module, we cover the basics of Smojo.

Words and Literals

In Smojo, everything is either a **Word** (ie, an action) or a **Literal** (eg, numbers, text, etc.). For example, this program:

```
"Hello World" .
```

Consists of the Literal text `Hello World` and the Word `.` which prints the text to screen. If you run this program, you should see:

```
Hello World ok
```

Smojo always responds with "ok" to tell you that your program has stopped.

The program:

```
1 2 + .
```

adds two literal integers, `1` and `2` then prints the result to screen.

There is a list of commonly used Math words at the end of this chapter.

Quiz 1.0: Try out this program now. Also try printing the result of `1` divided by `2`. Does the answer surprise you? There is a difference in Smojo between integer operations and those for real numbers.

Left to Right

Smojo code is executed from **left to right**. Also, all words like `+` and `.` act on preceding input. This means that data transformations are very simple. For example, if you have

an operation `clean` followed by `normalise` then `process`, of some input, in Smojo, this is just:

```
clean normalise process
```

Which is the natural way of expressing this data processing pipeline. There is no need for parentheses in Smojo because everything is from left to right.

Because of this, spaces are important in Smojo. For example, this code won't work:

```
\ WON'T WORK!!!

1 2+ .
```

because Smojo needs a space between the literal `2` and the word `+`

```
\ WORKS FINE!

1 2 + .
```

New Words

For example, in Smojo, to make a new word, `transform`, you would write:

```
: transform

    clean normalise process

;
```

The words `:` and `;` are not punctuation (remember, in Smojo, there is **no** punctuation, only Words or Literals). The word `:` begins the definition of a new word while the word `;` completes it.

Quiz 1.1: Write a word `average` that averages two numbers. For example, `3 4 average .` should display `3.5`. Hint: you need to use `+.` and `/.` which are real number addition and division respectively.

The Data Stack

All Literals in Smojo are entered into the **Data Stack**. For example, in the program:

```
1 2 + .
```

Here's what happens:

1. First, the Literal `1` is placed on the data stack.
2. Then the literal `2` is added to the top of the stack. So `1` is below `2`.
3. The word `+` takes off the top two literals on the data stack (ie, `1` and `2`) and adds them. The result (ie, `3`) is placed on the stack. So, at this point, the stack contains just one literal, `3`.
4. The word `.` takes off the literal on the top of the stack and prints it. Now, the stack is empty.

You can see what happens on the stack by using the word `BP`. This word displays the content of the stack. For example, to see steps 1 to 4 in action, we amend the code above:

```
1 BP 2 BP + BP . BP
```

If you run this program, you will see:

```
--breakpoint #8--
<1> 1

--breakpoint #9--
<2> 1 2

--breakpoint #10--
<1> 3
3
--breakpoint #11--
<0>
ok
```

The numbers in brackets (eg **<2>**) indicates the number of literals currently on the data stack.

Locals

Often, it is convenient to give the literals on the data stack temporary names. We use **locals** to do this. For example, to get the average of two numbers, we can use locals:

```
: average { a b }
     a b +. 2 /.
;
```

In this example, the word `{` begins a local definition, while `}` ends it. So, `b` gets the literal at the top of the data stack, while `a` is the item below it. So, the program:

```
3 45 average
```

results in `a = 3` and `b = 45`. Locals remove the literals from the data stack as they are being named. Also, locals only work within the definition of a new word. They are also only valid within the word containing them. For example:

```
\ Finds the average of 2 numbers.
: average ( n n -- n ) { a b }
     a b +. 2 /. ;
3 45 average .
a . b .
```

Will result in an error because `a` and `b` only have meaning within the word `average`.

As you might have guessed, the lines in purple, like `\ Finds…` are **comments**, which are ignored by Smojo and only for human readability. The word `\` starts a line comment. `(` starts a "stack comment" that ends with). So, the comment `( n n -- n )` simply means that the word expects two numbers as input and outputs a single number.

Quiz 1.2: Compare this definition of `average` with the one you did in Quiz 1.1. Which is simpler? Put in `BP`s into `average` and verify that locals do remove the literals from the data stack. Test out your new code.

Quiz 1.3: Write a word `frac` that gets the fractional part of a real number. For example, `1.2 frac .` should display `0.2`. Hint: you need to use the word `floor` that gives the integer part of a real number. You also need to use `-.` for real number subtraction. Does your word work for negative numbers?

Interpretation vs Compilation

Smojo has two **modes** of operation.

- **Interpretation Mode**: words are executed when they are encountered. This is the default mode.
- **Compilation Mode**: used in the definition of new words. Words are no longer executed, they are instead added into the new word's definition. The exception is `immediate` words which are executed even in compilation mode.

The word `:` begins compilation mode and the immediate word `;` ends it.

An example:

```
"Hi Johnny" .      — interpretation mode
: greet            — compilation mode
  "hi" . . cr ;    — compilation mode
"Johnny" greet     — back to interpretation mode
```

Conditionals

Conditionals help your program to make decisions. In Smojo, these work only in compilation mode (ie, within a word definition). We start with an example:

```
: number>text { s }
        s 2 = ->  "TWO"   |
        s 3 = ->  "THREE" |
        s 4 = ->  "FOUR"  |
        otherwise "Whatever" |.
;
2 number>text . cr
25 number>text .
```

The word **number>text** converts (some) integers into text.

The word **->** checks if the preceding value on the data stack is **true**. If so, it executes the body of the code up to the delimiter word **|**, and then skips to the code after the final delimiter **|.**. If **false**, Smojo skips to the code after the nearest delimiter. The final delimiter is the word **|.**

Quiz 1.4: Type in this code into the Autocaffe editor and run it. What does the word **cr** do? What does the word **otherwise** do? (Hint, replace **otherwise** with **true ->**).

Quiz 1.5: Here's another way to write **number>text**, but there's a bug. What is wrong with this code?

```
: number>text2 { s }
        s 2 = ->  "TWO"   |.
        s 3 = ->  "THREE" |.
        s 4 = ->  "FOUR"  |.
        “Whatever”
;
2 number>text2 . cr
25 number>text2 .
```

How would you fix it? Hint: the word `exit` causes execution to return to the caller.

Executables

An executable (or **XT** for short) is a Literal that you can execute. XTs are widely used in Smojo and is what makes Smojo a powerful programming language.

You can get the XT of any word using the word `‘` (called **tick**) in interpretation mode or `[‘]` (called **bracket-tick**) in compilation mode. For example:

`: hello “hi” . ;`	— definition of `hello`
`hello`	— runs `hello`
`‘ hello bp`	— puts the XT for `hello` on the stack.The `BP` is just to print the contents of the stack.
`execute`	— takes the XT off the stack and executes it.

So, there is **no** difference in results between:

`hello` and `‘ hello execute`

Both lines execute the word **hello**, or more correctly, its XT.

In Smojo, the XT is important; names like **hello** provide us with easy access to a word's XT. You can execute any XT using the word **execute**. In interpretation mode, Smojo retrieves a word's XT given its name and runs the XT. In compilation mode, that XT is compiled into the new word.

Quotations

Since XTs are the most important in Smojo, is it possible to create XTs with no names? Yes. We use the words **[:** and **;]** (in compilation mode) and **:>** and **;** (in interpretation mode):

`:> 1 2 + . cr ; execute` —	Creates an XT in interpretation mode.

`: mk-hello { s }`
`  [: "hi" . s . cr ;]` —
`;`
	Defines an XT in compilation mode.

`"Janna" mk-hello bp` —	Creates the defined XT. Note that the XT will be different depending on the value of the input.
`execute` —	Runs the XT. What is the output?

Quiz 1.6: Run this example and make sure you understand how it works.

The Importance of XTs

XTs are useful because they can be passed as input into other words. You will see this in action when we use Smojo for data processing.

To recap:

`'` is used to retrieve the XT given a name in interpretation mode.

`['']` is the same, but used in compilation mode.

`[:` ... `;]` is used to define a quotation in compilation mode. An XT is not produced, just defined. The XT is created when the enclosing word is run.

`:>` ... `;` is used to define a quotation in interpretation mode. The XT is produced immediately after `;`

Example: Numerical Differentiation

XTs offer a powerful way to break up your program, so that various parts can be **re-used**. For example, suppose we needed to find the differentiation of a function $f(x)$. We can use the **forward difference formula**:

$$f'(x) \approx \frac{f(x+\epsilon) - f(x)}{\epsilon}$$

where ϵ is a small number. There are much better formulae for numerical differentiation, but we will use the forward difference formula for simplicity.

Here is one way to use the equation to numerically differentiate $f(x) = 3sin(x)$. The word `diff` does the work of numerically differentiating `f`:

```
\ f(x) = 3 sin(x)
: f ( n -- n )
    sin 3 *. ;
```

```
: diff ( n -- n ) { x }
   x 0.00001 +. f
   x f
      -. 0.00001 /.
;

\ Test!
3.1 diff .
```

Quiz 1.7: Try this example in Autocaffe. What is the result? Calculate the actual differential using a calculator. Do these two match?

Quiz 1.8: What is the value of ϵ? Is there a better way of doing this? Hint: `0.1 constant epsilon` defines $\epsilon = 0.1$. Use `constant` to tidy up the code above.

Quiz 1.9: Calculate the differential of $f(x)$ analytically and write a word called `df/dx` to output the actual value of $f'(x)$. Compare the analytic value of $f'(x)$ with the one obtained by numerical differentiation using `diff`. What is the error? What is one way to improve accuracy?

Two Problems

There are 2 big problems with `diff`:

1. How can we use it to differentiate other functions like $g(x) = x^2$ or $h(x) = 31e^{2x}$?

2. How do we calculate higher order differentials like f', f'' or g^4 or h^{12} ?

The simple solution to the **first** problem to write different versions of `diff`, eg, `diff-f` to differentiate `f` and `diff-g` to differentiate `g`, etc. The example below shows how this might be done:

```
\ f = 3 sin(x)
: f ( n -- n )
    sin 3 *. ;

\ g = x^2
: g ( n -- n )
    dup *. ;
```

```
\ h = 31 e^2x
: h ( n -- n )
  2 *. exp 31 *. ;

: diff-f ( n -- n ) { x }
   x 0.00001 +. f
   x f
     -. 0.00001 /.
;

: diff-g ( n -- n ) { x }
   x 0.00001 +. g
   x g
     -. 0.00001 /.
;

: diff-h ( n -- n ) { x }
   x 0.00001 +. h
   x h
     -. 0.00001 /.
;

\ Test!
3.1 diff-f . cr
0.1 diff-g . cr
0.35 diff-h . cr
```

(**Note**: The word **DUP** duplicates the top item on the data stack. So, **23 dup bp** will display **<2> 23 23**)

It should be immediately obvious that while the functions **f**, **g** and **h** are quite different, their numerical differentiation, **diff-f**, **diff-g** and **diff-h** are remarkably similar.

In fact, I just copy-n-pasted the original code for **diff** each time, making the small change to the function used. This is to be avoided since it makes your code prone to errors.

Generalizing DIFF

The next step to amend our original **diff** to work on any function. This is possible if we can use the function's XT as input into the new **diff**:

```
: f ( n -- n )
    sin 3 *. ;

: g ( n -- n )
    dup *. ;

: h ( n -- n )
  2 *. exp 31 *. ;


: diff ( n xt -- n ) { x fn }
   x 0.00001 +. fn execute
   x fn execute
     -. 0.00001 /.
;

\ Test!
3.1 ' f diff . cr
0.1 ' g diff . cr
0.35 ' h diff . cr
```

Wow! This new program performs the same task as before but uses much less code. Also, we can use the new **diff** for any function.

This is the power of using XTs.

Quiz 1.10: Study this example of **diff** carefully. Be sure you understand how it works.

Using **fn execute** each time this way is clumsy. A better method is to use the word **~**:

```
: diff ( n xt -- n ) ~ { fn } { x }
   x 0.00001 +. fn
   x fn
     -. 0.00001 /.
;
```

~ is always used in conjunction with a **single** local **{ … }** only. If you need multiple **~** locals, you need to use **~ { … }** multiple times.

Quiz 1.11: Re-run and test this code thoroughly now. Our latest version of `diff` is able to differentiate any function.

Higher Order Differentials

We can now tackle the second problem - calculating higher order differentials.

We start by using diff to output a **function**, not a number. So, `' f diff` would give a function (ie, XT) representing the first order differential of `f`.

This change means that we can now keep adding in `diff`s to create higher order differentials. Eg,

```
' f diff diff
```

would also return a function (ie, XT) which is the second order differential of the input function `f`.
In other words, higher order differentials can be obtained simply by calling `diff` repeatedly.

Let's make the changes now.

Step 0: `diff` is `( n xt -- n )`, ie, it takes a number and an XT then returns a number:

```
: diff ( n xt -- n ) ~ { fn } { x }
   x 0.00001 +. fn
   x fn
     -. 0.00001 /.
;
```

We want our new `diff` to be `( xt -- xt )`.

Step 1: Change `diff` so that it no longer binds the input `x`:

```
\ WON'T WORK!!!
: diff ( xt -- xt ) ~ { fn }
   x 0.00001 +. fn
   x fn
     -. 0.00001 /.
;
```

This won't work of course, since **x** is undefined and **diff** does not return an XT. We must wrap **diff**'s inner code into a quotation and define **x**. Step 2:

```
: diff ( xt -- xt ) ~ { fn }
  [: { x }
      x 0.00001 +. fn
      x fn
        -. 0.00001 /.
  ;]
;
```

So, when **diff** is run, it will produce an XT **(n -- n)** which is a numerical differentiation of the input function **fn**.

Lastly, we need to amend our testing code. Since **diff** produces an XT not a number, we need to use **execute**. For example:

```
3.1 ' f diff diff execute .
```

will print out the value of $f''(3.1)$. Note that **diff** will now differentiate any function you feed into it.

Quiz 1.12: Re-write your testing code and ensure it works. Make a comparison between the analytic and numerical differentiation for f'' and g''.

***Quiz 1.13**: How would you use **diff** to differentiate $m(x, y) = x^2 + y^2 + 34$ along the line $x = 12$? Hint: you can use diff to differentiate $l(y) = m(12,y)$. You can get $l(y)$ using a quotation. In this case, $l(y)$ is known as a **curried function**. Write your code and get the answer for $l'''(12)$. Did the answer meet your expectations?

Learning Points

- When starting to work on a problem, find the simplest solution that works. In our case, we started with a solution for `diff` that could only differentiate `f` once.

- Expand your code slowly only in response to additional requirements. You expanded the code to handle differentiating of other functions, (like `g`) not just `f`. Do not anticipate complexity if its not immediately required.

- The property of **closure**, where the input and output of a word are the same is a simple but powerful idea. In our case, we were able to re-use `diff` to obtain second and higher order derivatives when we made the input and output of `diff` to be the same `( xt -- xt )`. It takes experience to spot opportunities to use closure.

- Most Smojo words are often just 2 - 3 lines of code, very rarely more than 5 lines long. This rule makes Smojo easier to debug.

- Always **thoroughly** test your code as you program. Don't be tempted to code a lot all at once. Code a bit then test. This will save you a lot of time and make you a productive Smojo programmer!

Appendix: Math Words

Word	Action

`+ - * / =`	Used to operate between integers only. If a real number is encountered, it is *truncated* -- fractional part is discarded. For example,
	`1 2 +`
	will give **3** as expected, but
	`1 2.8 +`
	will also give **3**, since **2.8** is truncated. The equal sign is used to test if two integers on the stack are equal.

`+. -. *. /. =.`	As above, but these can operate on real numbers. No truncation is performed.
`> >= < <=`	Inequality tests. These work on both integers and reals. The result is a boolean (`true` or `false`) on the stack. `1 2.2 < .` displays **true ok**
`FLOOR`	Displays the floor of a number, which is the closest integer lower than the given number.
`CEIL`	Displays the ceiling of a number, which is the closest integer higher than the given number.
`ROUND`	Rounds off to the nearest integer.
`MIN` `MAX`	Minimum and Maximum of two numbers.
`^`	Exponentiation. `2 3 ^` becomes `8`

`TAN`
`SIN`
`COS`
	Usual trigonometric functions.

Word	Action
`ATAN2`	The special arctangent, very useful for computing directions from vectors.
`#e` `#pi`	Constant values of e and π
`EXP` `LN`	Natural exponentiation and natural logarithms.
`LOG10`	Logarithms base 10.
`TO-DEG` `TO-RADIANS`	Converts angles to degrees or radians.
`MOD` `FMOD`	Integer modulus and real modulus. Use `FMOD` if either operand is a real number.
`ABS`	The absolute value.
`TRUE` `FALSE`	Puts the logical values `true` and `false` on the stack.
`OR`	Logical OR. For example, `true false OR` is `true`.
`AND`	Logical AND. For example, `true false AND` is `false`.
`NOT`	Logical NOT.