# Metaprogramming II

## The Data Section, Using Literal, Flow Control and Parsers

In this session, we will cover a few seemingly disjoint topics which complete our metaprogramming toolbox:

- **The data section**, especially for Quotations. We will look at how the data section for Quotations can be set using the COMMA `,` and `[` ... `]` words, and how you can access the data section using `SELF`. We will also look at the definition of `SELF`. We will also see how the mode switching words `[` and `]` are different from other immediate words.
- `LITERAL` and how it is used in greater depth. There is a very common design pattern using `LITERAL`, called the **Injector/Payload/Host** pattern. Once you see this, you can easily spot it in previous examples.
- **Flow Control** is how your program decides to perform jumps and conditionals. We will look at the big 3 words for creating flow control words: `IF`, `AHEAD` and `THEN`.
- Lastly, we will show you how you can add **parsers** (not to be confused with the `PARSE` word) so you can directly change Smojo's syntax. For example, we will see how you can add rational numbers (like 1/2 or 3/4) to extend Smojo's pre-built syntax for numbers.

## The Data Section

Let's start with a review of the Data Section. As you now know, Smojo words are just ("name", XT) pairs in Smojo's dictionary. You can access these pairs using `GET-DICTIONARY ( -- # )`.

Each XT has:

- a **code section** to store code that governs its behaviour. This code section can be executed using the word `EXECUTE ( xt -- )`
- a **data section** to store data for this XT. The data section must be defined when the XT is being created. Once created, the XT may no longer be amended. To create the data section we use the COMMA `, ( v -- )` operator. COMMA implicitly inserts the item on the top of stack (`v`) into the word that is being created.

The data section is important since many metaprogramming techniques depend on its use: In Session 2's Example 4, when we built the module system, usage of the data section was crucial.

Here's a simple example storing two numbers into a word `XYZ`:

```
: xyz ( -- )
     [ 43 , 21 , ]
     "Hello World" . cr
;
\ Read the data section of XYZ:
' xyz 0 @ . cr \ 43 ok
' xyz 1 @ . cr \ 21 ok
```

Note that the phrase `[ 43 , 21 , ]` will run during the creation of `XYZ`'s XT, and this places `43` then `21` into the data section of `XYZ`. This works because `[` ... `]` are both immediate words, and they switch modes temporarily so that the , word can build the data section.

The FETCH word `@ ( xt n -- v )` retrieves the value `v` from the given XT. `n` is just the index needed. You can omit `n` if it is zero. Eg,

```
' xyz @ . cr \ 43 ok -- no need for n
```

You can use the STORE `! ( v xt n -- )` word to store a new value into the slot n of an XT's data section:

```
123 ' xyz 0 ! \ Replace 43 with 123
```

```
' xyz @ . \ 123 ok
```

Note that you can only use `!` to store data only if the slot exists on the XT's data section. Otherwise you will get an error. For example,

```
"abc" ' xyz 23 ! \ !ERROR!
```

will raise an error because slot **23** does not exist.

## Quiz 3.0

**Quiz 3.0.0**: Run this example. Store a third value in `XYZ's` data section.

**Quiz 3.0.1**: The number of items you can store in the data section is fixed at creation time. Is this a severe limitation? How might you change the number of items stored in the data section even after the XT is created? Hint: What data structures would you use?

The pattern `[ value , ]` can also be used in Quotations. For example, suppose our XYX now makes an XT:

```
: xyz ( -- xt )
     [:
          [ 43 , 21 , ]
          "Hello World" . cr
     ;]
;

\ Run XYZ to create the XT:
xyz => h
\ Read the data section of the XT:
h 0 @ . cr \ 43 ok
h 1 @ . cr \ 21 ok
```

## Quiz 3.1

**Quiz 3.1.0**: Run this example. Replace the first slot with a hash.

**Quiz 3.1.1**: If we ran `XYZ` twice, and saved the XTs each time (say `h1` and `h2`), would changing the first slot value in `h1` affect `h2`? Test this idea out with some code to check your answer.

**Quiz 3.1.2**: Do you notice something strange about our new `XYZ`? Hint: are `[` and `]` immediate words? Run a decompilation of `XYZ`. What do you notice?

By right, immediate words should execute during compilation mode. But in the example `XYZ` above, `[` and `]` although they are immediate, **do not** execute during the definition of `XYZ`! Instead, they are compiled.

This detail is very important: `[` and `]` in quotations are only executed during the creation of the quotation, not during its definition. In our example, the Quotation is defined when `XYZ` is defined, but it is only created with `XYZ` is being run.

This exception only applies to `[` and `]`. They are special. All other immediate words are indeed executed when the enclosing word is defined, as we would expect.

This exception to the behaviour of `[` and `]` allows us to design the data section of quotation. Without it, we cannot build data sections of quotations easily.

# References

As you might have noted by now any locals used by word are "frozen" when they are used in a quotation. For example:

```
: mk-greet ( "s" -- xt ) { s }
     [:
          "Hello" . s .
     ;]
;

\ Run XYZ to create the XT:
"arnold" mk-greet => h
\ Run the XT:
h execute \ Hello arnold ok
\ Decompile the XT:
h decompile
```

Will result in the decompilation below:

```
[0]  Literal(Hello)
[1}  .
[2]  Literal(arnold)
[3}  .
ok
```

In this decompilation, you can see that the input string **`"arnold"`** which was bound to the local **`s`** is now hard-coded as a literal in the body of the resulting XT (line 2).

This behaviour, known as **lexical binding** is an important feature because helps us avoid difficult bugs and has many other benefits. But in many cases, we want to initialise a local (eg, a counter), then change it in within the quotation.

For example, suppose we want to create a quotation that keeps track of how many times it is executed:

```
\ !WON'T WORK!
: xyz ( -- xt )
     0 { n }
     [:
          "Hello" . n . cr
          n 1 + { n }
     ;]
;

xyz => h
h execute \ Hello 0
h execute \ Hello 0
h execute \ Hello 0
```

We would expect the counter to increase (eg, **`Hello 0`**, **`Hello 1`** and **`Hello 2`**), but this doesn't happen because of lexical binding. A decompilation of **`h`** shows why:

```
[0]  Literal(Hello)
[1}  .
[2]  Literal(0)
[3}  .
[4}  CR
[5]  Literal(0)
[6]  Literal(1)
[7]  +
[8]  write local: >n<
ok
```

Line 2 which we expect to print the local n will instead show the literal **`0`** every time. There is a local n created on the XT (line 8), but it is bound to the same value: **`0 1 + = 1`**, and this local is never used within the XT. All of this is because of lexical binding.

To get around this, we can build something called a reference. There are many ways to do this, but a simple one is to utilise the data section of an XT:

```
: ref ( v -- ref )
     [: [ , ] ;]
;
\ WORKS!
: xyz ( -- xt )
     0 ref { n }
     [:
          "Hello" . n @ . cr
          n @ 1 + n !
     ;]
;

xyz => h
h execute \ Hello 0
h execute \ Hello 1
h execute \ Hello 2
```

**REF ( v -- XT )** creates an XT with one item in its data section, which is the input **v** of the reference. Subsequently, even with lexical binding, we can still use **@** and **!** to fetch and store values into this XT.

Note: **REF** is a built-in word.

## Quiz 3.2

**Quiz 3.2.0**: Run this example. How would you start the counting at **1** instead of **0**?

**Quiz 3.2.1**: Carefully study the definition of **REF**. Do you understand how it works? If you do, propose how you might find a different way to write **REF** without using Quotations. What other changes would you have to make? Write some code to test it out.

**Quiz 3.2.2**: If we ran **XYZ** twice, and saved the XTs each time (say **a** and **b**), would running **a** affect the values printed by **b**? Test this idea out with some code to check your answer.

**Quiz 3.2.3**: How might we re-write **XYZ** so that running **a** **would** affect **b** and vice-versa? Ie,

```
a execute \ Hello 0

a execute \ Hello 1

b execute \ Hello 2 (not Hello 0)

a execute \ Hello 3 (not Hello 2).
```

Write your code and test to ensure it works. Hint: You need to use the data section of **XYZ**.

**Quiz 3.2.4**: Is there another way to accomplish Quiz 3.2.3 **without** using the data section of **XYZ**? Hint: You need to define a global variable.

# Using LITERAL

One of the things that confuses beginners learning Smojo metaprogramming is when and how to use **LITERAL**.

In almost every case I can think of, usage centres on a simple design pattern called **Injector/Payload/Host**:

- The **Payload** is the literal that needs to be inserted.
- The **Host** is the word into which the literal Payload is inserted into.
- The **Injector** is a word that actually injects the Payload into the Host word. The Injector is always immediate.

A typical use case is outlined below. In this example, the payload is a string, **`"Hello World"`**. In more realistic scenarios, the payload will be far more useful.

```
\ Injector
: inject ( -- )
     "Hello World" \ Payload
     postpone literal
; immediate

\ Host
: host ( -- )
     "Hey" . cr
     inject . cr \ Injection done
;
```

In the code above, the three important things are:

1. The Injector is always immediate.
2. It always contains (or calls) a **POSTPONE LITERAL**
3. It somehow needs to have access to the Payload when it is executed.

I'll give you two common examples of this pattern which we've seen already in Session 2:

```
\ Injector
: ['] ( |s -- )
     bl parse xt-from-name \ Payload
     postpone literal
; immediate

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

BRACKET-TICK **[']** is the injector which compiles a literal of the XT following it. Note how the XT (ie, the Payload) needs to be accessible when **[']** executes.

Another example is the word **SELF**, which is used within a quotation for self-reference. We saw it being used in building Smojo modules:

```
\ Injector (main)
: (self) ( -- )
     latestXT \ Payload
     postpone literal
;

\ Injector (final)
: self ( -- )
     ` [ ` (self) ` ]
; immediate

\ Host
: xyz ( -- XT )
     [:
          [ 0 , ]
          self @
               dup . cr   \ Print out
               1 + self ! \ increment
     ;]
;

\ TEST
xyz => h
h execute \ 0
h execute \ 1
h execute \ 2
```

In this example, **SELF** is used to refer to the XT's data section. The Injector comes in two parts: The first part is just like an ordinary injector except that it is not immediate. The final part constructs the necessary environment for the injector to run correctly in

Quotations, and takes advantage of the special behaviour of `[` and `]` within quotations. The final injector part is always immediate.

### Quiz 3.3

**Quiz 3.3.0**: Run this example, then try to explain the following:

- What would happen if the injector `SELF` were not immediate?
- Why does `(SELF)` need to be ordinary? Hint: We've taken advantage of the behaviour of `` ` `` when it encounters ordinary words.
- Why does `SELF` need to be split into two parts? Why not just put the contents of `(SELF)` into `SELF`? Try and see if that works.

**Quiz 3.3.1**: Will `SELF` work within an ordinary word (ie, not a Quotation?) Test out your answer. If your answer is "no" then come up with a version of `SELF` (let's call it `THIS`) that works for ordinary words.

## Flow Control

An XT's code section is a sequence of operations. These operations are each numbered with an **address**, starting from `0`. Executing an XT causes Smojo to perform each operation beginning at address `0`.

**Flow control** words allow us to alter the order in which these operations are performed, causing Smojo to **jump** to a particular address.

There are 2 kinds of jumps:

1. **IF** creates a **conditional jump**, causing Smojo to jump to a given address if the top of stack evaluates to **false**.
2. **AHEAD** creates an **unconditional jump**, which causes Smojo to jump to a given address ahead with no pre-conditions.

In the code below, the word HOT displays Hot ok if the input temperature is above 29°*C*:

```
: hot ( n -- )
    29 > if
        "Hot!"
    else
        "Cold"
    then
    . cr
;

see hot
```

Which results in the following decompilation:

```
[0]  Literal(29)
[1]  >
[2]  CJump<5>
[3]  Literal(Hot!)
[4]  Jump<6>
[5]  Literal(Cold)
[6}  .
[7}  CR
ok
```

The addresses are numbers on the left in brackets. You can see the **IF** has been transformed into a **CJump<5>** on address 2, which means a conditional jump to address 5. This means that the jump is made if the top of the data stack is false. Similarly, **ELSE** has been transformed into the operation on address 4, **Jump<6>** which is an unconditional jump to address 6.

Both **IF** and **AHEAD** are **immediate words**, which compile the respective Jump operation into the current word. At this point, this jump operation is **unresolved** (ie, the jump address is unknown). **IF** and **AHEAD** also place this jump operation on the data stack. In other words:

**IF** is **( -- jump )** and **AHEAD ( -- jump )**

All jump operations need to be resolved by

**THEN ( jump -- )**

which sets the jump address, which is the location of **THEN**. **THEN** is also an immediate word.

As an example, suppose we are in the process of compiling the following fragment (note, we are in compilation mode):

```
IF "Hello" . THEN
```

For simplicity, say that **IF** is at address **0**. When **IF** is compiled, the result is:

Code (underlined = compiled):

```
IF "Hello" . THEN
```

XT:

```
[0] CJump<-1>
```

Data Stack:

```
<1> CJump<-1>
```

Note the **CJump<-1>** is the jump operator as an object on the stack and it has also been compiled into the XT. The jump address is -1 because this jump has not been resolved. That is the work of **THEN**.

Then the phrase **"Hello" .** is compiled:

Code (underlined = compiled):

```
IF "Hello" . THEN
```

XT:

```
[0] CJump<-1>
[1] Literal(Hello)
[2} .
```

Data Stack:

```
<1> CJump<-1>
```

Note that the conditional jump is still unresolved at this point. Finally, the **THEN** is compiled:

Code (underlined = compiled):

```
IF "Hello" . THEN
```

XT:

```
[0] CJump<3>
[1] Literal(Hello)
[2} .
```

Data Stack:

```
<0>
```

The conditional jump to address 3 might mean exiting the XT if 3 exceeds the length of the XT itself.

## Quiz 3.4

**Quiz 3.4.0**: Run this example yourself by creating a new word and examine the decompilation.

**Quiz 3.4.1**: How would you check the claim that **IF** and **AHEAD** both add a jump object on the data stack?

# Compiling ELSE

Let's take another example **ELSE**, which is:

```
: else ( cjump -- jump )
     ` ahead swap ` then
; immediate
```

and see how it is used to compile the following fragment:

```
IF "A" . ELSE "B" . THEN
```

Again, I'll assume we start from address **0**:

Code (underlined = compiled):

```
IF "A" . ELSE "B" . THEN
```

XT:

```
[0] CJump<-1>
[1] Literal(A)
[2} .
```

Data Stack:

```
<1> CJump<-1>
```

Unrolling the definition for **ELSE** and remembering it is also an immediate word:

Code (underlined = compiled):

```
IF "A" . ahead swap then "B" . THEN
```

XT:

```
[0] CJump<-1>
[1] Literal(A)
[2} .
[3] Jump<-1>
```

Data Stack:

```
<1> Jump<-1> CJump<-1>
```

Compiling the **THEN** from **ELSE**, note that this resolves the **CJump** first, because the **SWAP** placed it on the top of the stack.

Code (underlined = compiled):

```
IF "A" . ahead swap then "B" . THEN
```

XT:

```
[0] CJump<4>
[1] Literal(A)
[2} .
[3] Jump<-1>
```

Data Stack:

```
<1> Jump<-1>
```

Lastly, the rest of the code is compiled and the Jump can be resolved by the final **THEN**:

Code (underlined = compiled):

```
IF "A" . ahead swap then "B" . THEN
```

XT:

```
[0] CJump<4>
[1] Literal(A)
[2} .
[3] Jump<6>
[4] Literal(B)
[5} .
```

Data Stack:

```
<0>
```

# The Big Four

It may surprise you to learn that Smojo has just **four** basic flow control words: `IF`, `AHEAD`, `BEGIN` and `(THEN)`.

`BEGIN ( -- n )` is an immediate word that puts its address on the stack.

`(THEN) ( jump n -- )` resolves a `jump` on the stack with the address `n`.

**All** other flow words that you might have used like `ELSE`, `->`, `|.`, `AGAIN`, `UNTIL`, `EXIT` etc. are actually built from the four basic words.

For example, `EXIT` can be written this way:

```
: EXIT ( -- )
    ` ahead 99999 ` (then)
; immediate
```

## Quiz 3.5

**Quiz 3.5.0**: Test out this version of `EXIT`. Why is the number `99999` used? Will a different number work?

**Quiz 3.5.1**: Write `THEN ( jump -- )` in terms of the Big Four.

**Quiz 3.5.2**: Write `AGAIN ( n -- )` in terms of the Big Four. **Hint**: `AGAIN` uses an unconditional jump.

**Quiz 3.5.3**: Write `UNTIL ( n -- )` in terms of the Big Four. **Hint**: `UNTIL` uses a conditional jump.

**Quiz 3.5.4**: Write `DO` ... `LOOP` in terms of the Big Four. **Hint**: You could save the counter in the spare

stack. That is how it is done now. A better solution might be to use a variable.