Metaprogramming I

Beginning Metaprogramming with Smojo

There are several words can concepts that you need to thoroughly understand in this Session because they form the basis of our approach to metaprogramming:

- What an XT is and how it differs from a Word. Using **`EXECUTE`** to run an XT,
- **`PARSE`** and the token stream,
- **`TOKENIZE`** to split up a string and **`ARRAY>SEQ`** to convert an array into a sequence,
- **`LITERAL`** to programmatically add any literal into a Word,
- **`IMMEDIATE`** and the concept of immediacy,
- **`POSTPONE`** and how it affects immediate and ordinary words.

These concepts may seem unrelated at first, but they form the base upon which we will build all our subsequent metaprogramming work.

NOTE: This topic can be difficult. To help your understanding, I've added "Quizzes" to this Session. Completing them will greatly aid your understanding.

XTs and Words

An **XT** — which are short for e**X**ecu**T**able — is essentially a potential action. An XT can be treated like any ordinary literal (ie, a String, Number or Hash). It can be printed, put on the stack, etc.

A **Word** is simply a **`(name, XT)`** pair. These pairs are stored in Smojo's **dictionary**. All words in Smojo are stored this way. There are no "special" words in Smojo, all are treated alike. You can access this dictionary as a hash using `GET-DICTIONARY ( -- # )`. There is a corresponding word `SET-DICTIONARY ( # - )` that replaces the dictionary with the given hash.

Quiz 2.0

Quiz 2.0.0: Write a word called `WORDS ( - )` that prints out all the words currently defined in the dictionary. Note that there is already a built-in word called `WORDS`. How does your version differ from the built-in one? **Hint**: you need to use `GET-DICTIONARY` and an amended version of `PP` from Session 1.

Quiz 2.0.1: Write a pair of words `m{` and `}m` that **temporarily** replace the current dictionary with a dictionary that only has the words `+`, `-`, `*`, and `.`. In other words, `m{` should save the current dictionary and replaces it with your own one, and `}m` restores the previously saved dictionary. Hint: you need to use `SET-DICTIONARY` for both these words.

What does `m{ 1 2 3 + * . }m` show? What does `m{ 24 dup . }m` show? How might words like `m{` and `}m` be useful? **Give this some thought**. We will discuss this topic extensively in later sessions.

Getting an XT with Tick

We can get an XT either by extracting it from an existing word using tick `'` or bracket-tick `[']` words. For example the code below will print out the XT for the word `DUP`:

```
' DUP .
```

The output will vary because the internal XT used for `DUP` will change every time you run a Smojo program.

The word bracket-tick `[']` is used to get the XT from an existing word. Here's an example using it to reverse a list:

```
: reverse ( list — list )
     nil swap ['] cons reduce
;
```

In this example, `['] cons` finds the XT of `CONS` and compiles it as a literal into the body of `REVERSE`. A decompilation of `REVERSE` shows this clearly:

```
' reverse decompile
```

which outputs:

```
[0]  NIL
[1]  SWAP
[2]  Literal XT(59a8bc15)
[3]  REDUCE
ok
```

The literal XT for `CONS` is **`XT(58a8bc15)`** on line 2, but this will change every time you run the decompilation.

Quiz 2.1

Quiz 2.1.0: What is the difference between tick `'` and bracket-tick `[']` ? Why do we need two words to extract the XT from an existing Word?

Quiz 2.1.1: In the word for `REVERSE`, replace `[']` with `'` and decompile it. What do you see? What does that tell you?

Quiz 2.1.2: In the program `' reverse decompile`, replace tick with bracket-tick and run. What happens?

Quiz 2.1.3: Use tick to get the XT of tick or bracket-tick and print it out.

Quiz 2.1.4: Decompile tick or bracket-tick.

Quiz 2.1.5: What happens if you tick (or bracket-tick) a word that does not exist?

The Token Stream and PARSE

Smojo is supposed to be strictly left-to-right, so how does a word like tick `'` work? It seems to jump ahead in the input! Let's break it down step by step:

1. Smojo reads any program (which is just text) as a sequence of **tokens**. For example, the program: `"Hello World" . cr` consists of 3 tokens: `"Hello World"`, `.` and `cr`. The program text is broken up into tokens using the space character as a delimiter, but special care is taken when handling strings. This is called the **token stream**. It is important to note that this splitting of the program into tokens is not done all at once, but gradually, step by step. This is what gives Smojo its left-to-right behaviour.

2. Tick `'` first looks one step ahead into the token stream and places the token (ie, text) on the stack.

3. It then looks up the dictionary for the corresponding XT, which is placed on the stack.

The word `PARSE ( char — "s" )` lets you access the token stream **before** it is read by Smojo. The input `char` into `PARSE` tells it when to stop reading the token stream. `char` is a single character, not a string and you need to use the word `[CHAR]` to create a character. For example, the program

```
[char] / parse Hello World/ . cr
```

Will print out:

```
Hello World
ok
```

In this example, the character `/` is used as a delimiter, so `PARSE` will read the token stream until the character / is first met. The character `/` itself is not included as an output of `PARSE`. The effect it to put the text **`Hello World`** on the stack. The normal Smojo flow then resumes, and it executes `.` and `cr`.

IMPORTANT: `PARSE` works on the current line of text only. It won't read the token stream past the current line.

Quiz 2.2

Quiz 2.2.0: What happens if we use PARSE on an empty line? Eg:

```
[char] \ parse
```

or where the character is never encountered, eg:

```
[char] x parse Hello World . cr
```

or where the character is the only thing available, eg:

```
[char] h parse h . cr
```

Quiz 2.2.1: Write a word `XT-FROM-NAME` `( "name" — XT|null )` that looks up the dictionary for the given name and places the corresponding XT on the stack. **Hint**: You need to use `GET-DICTIONARY`.

Be sure your `XT-FROM-NAME` works for all cases, eg: `“Dup” XT-FROM-NAME .` must work correctly.

Quiz 2.2.2: Write a word `NEXT-TOKEN` that reads the next token in the token stream. **Hint**: Use `BL` `( — char )` which outputs the space character. You need to use `PARSE` also.

Quiz 2.2.3: Does your word work if the user typed in more than one space or a tab? Eg:

```
NEXT-TOKEN     hello .
```

Should also output **`hello`**. Hint: You need to use `BEGIN` … `AGAIN`

Quiz 2.2.4: Write your own version of tick ' using NEXT-TOKEN and XT-FROM-NAME. Make your version print out a warning if the word does not exist. Eg, ' yabbadabbadoo should print out a suitable warning.

Quiz 2.2.5: Write your own version of the line comment word \ and the stack comment word (**Hint**: You need to use [char] \n to represent the end-of-line character.

Quotations

Another way to get an XT is to create it using a Quotation, with the words :> ... ; in interpretation mode and the more common [: ... ;] in compilation mode. For example:

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

or:

```
: square ( seq — seq )
     [: dup * ;] map
;
{{ 1 2 3 4 5 }} square .list
```

In this example, a brand-new XT is created every time you run SQUARE. Quotations are powerful because they allow you to create XTs without naming a word, but also because (and this is important) they **bind locals into the resulting XT** itself. For example:

```
: greet ( "message" — XT ) { msg }
     [: msg . ;]
;
```

```
"hello" greet decompile cr
"world" greet decompile cr
```

will show:

```
[0]  Literal(hello)
[1}  .

[0]  Literal(world)
[1}  .

ok
```

indicating that each message (**hello** and **world**) is part of the two XTs created by **greet**. Each invocation of **greet** creates a new XT through the quotation.

To perform the action represented by an XT, we use the word **EXECUTE** (xt -- *) that takes an XT as input and runs it. For example, continuing from the code above,

```
"hey joe" greet execute
```

will output:

```
hey joe ok
```

Quiz 2.3

Quiz 2.3.0: decompile **greet** and try to make sense of every line in the decompilation.

Quiz 2.3.1: In the code for **reverse**, should we use

```
['] cons
```

or

```
[: cons ;]
```

Is there any difference?

Quiz 2.3.2: Why are Quotations useful? Why not just use Words? **Hint**: Try to re-write the word `PP` which prints a hash:

```
: pp ( # — ) { h }
   h #keys [:
        dup . "=" . h #@ . cr
   ;] reduce
;
```

without using Quotations. You can use `[']`.

Why are XTs Useful?

XTs are useful because they allow you to treat actions like literals. For example, you can:

- **save them** to disk, send then over the internet, etc. This is called **serialisation**. Smojo's chatbots (you can see some here at https://smojo.ai) are serialised XTs. Another example is cloud words (like `smojo/doc`)
- **amend them dynamically**, eg, create a new word from an existing XT to perform additional tasks like error detection or performance monitoring.
- **easily implement advanced programming techniques** like late binding or temporary changes to the dictionary (eg, the `m{` ... `}m` of Quiz 2.0.1) which are useful for metaprogramming. We will see these in later sessions.
- **use them as input into words**. Examples are higher order functions like `MAP` or `REDUCE`.
- **they can be used to store data**. XTs are not just functions, they can also be used to store data in their **data section**, a bit like an object in OOP. They differ from traditional OOP because they are much simpler and do not need cumbersome class definitions. We will see an example of this when we re-create Smojo's `smojo/doc` documentation system.

Most of the power of Smojo derives from having XTs and being able to create them easily.

Lastly, XTs can also be **defined** at runtime (that is, not just created at runtime like objects in OOP). This is an advanced use case and also an example of metaprogramming, which we will not cover in this book.

These ideas may seem a bit abstract, so let's deep dive into a an actual example.

Creating Getters and Setters

Suppose we have a key-value database, stored in a hash. Eg:

```
# => h

"name" "arnold" h #!

"country" "Singapore" h #!

"height" 180 h #!

h .
```

This code is "problematic" because we (a) need access to the hash (b) need to remember the keys we used to store the values. Ie, `"Name"` , `"NAME"` and `"name"` are all different keys.

A better way is to create getters & setters to get and set properties in the hash. For example:

```
\ Stores a name into our database.
: name! ( "s" -- )
     "name" swap h #!
;
\ Fetches the name from the database.
: name@ ( -- "s" )
     "name" h #@
;
```

Now we can use `name!` to store like so:

```
"arnold" name!
name@ . cr
```

Note, by convention, we use `xyz!` to denote a setter and `xyz@` for a getter.

However the problem now is that we have to also define similar words for `country` and `height`. This is not too bad, but what if we also needed getters/setters for more properties like `age gender address` and `email` ? The code for getters and setters would be error-prone to write (since we are essentially writing the same thing) and clutter our program.

We want to use Smojo's flexibility to our advantage. First, our goal is to simply declare properties, eg:

```
prop: name height country age gender

prop: address email
```

and let Smojo automate the creation of all getters and setters. We'll break down this process in several steps:

Step 0: We need a word to bind a `(name, XT)` pair into the dictionary:

```
: bind ( "name" xt -- )
     get-dictionary #!
;
```

There is actually no need to define `bind`, since it is already built-in. I mention it here for completeness and to help in your understanding. A more cautious version of `bind` would check if the name exists and warn the programmer:

```
: bind ( "name" xt -- ) { n xt }
     n xt-from-name null? not if
       "Overwriting:" . n ucase . cr
     then
     n xt bind \ previous version of bind
;
```

Notice that this version of `bind` overwrites the previous one, but uses it. This is **considered good style** in Smojo, where new functionality (like error checking) is layered onto a previous version of the word.

Step 1: We need to define words to create a getter and setter given a property name like "name" or "country". I'll call these `mk-!` and `mk-@.` For example, `mk-!` is:

```
: mk-! ( "property" -- ) lcase { s }
     s "!" concat
     [: ( v -- ) s swap MY-DBASE #! ;]
          bind
;
```

There are several points to note:

- We use `LCASE ( "s" -- "s" )` to lower case the property name, since by default, all words names are expected to be lowercased.
- The quotation creates an XT that expects an input, which is the value to save. We indicate this with the stack comment on the quotation. Always do this as it eliminates bugs.
- `MY-DBASE` is a `#` defined as a `constant`.

The code for `mk-@` is similar and I've left this as an exercise for you.

Step 2: We need to write a word `args ( |s -- seq )` that (i) reads in the token stream to the end of the line, (ii) splits it using whitespace and (iii) converts it into a sequence:

```
: args ( |s -- seq )
     \ reads the token stream to EOL.
     [char] \n parse
     \ splits by whitespace into a tuple
     "\s+" tokenize
     \ converts into a sequence.
      array>seq
;
```

The use of `PARSE` should be familiar to you.

`TOKENIZE ( "s" "delim" -- tuple )` splits a string using the delimiter into a tuple. It takes in two input, the string to split and a delimiter to split it with. For example, `","  TOKENIZE` would split a string using commas.

The delimiter we used in this example is `"\s+"` which is a **regular expression** meaning "one or more whitespace characters". You can find out more by searching for "Java regular expressions" online.

Finally the word `ARRAY>SEQ ( tuple -- seq )` converts the tuple into a sequence.

We frequently use `TOKENIZE` and `ARRAY>SEQ`, so you need to be familiar with these words.

Step 3: We need to write a word `prop:` that reads in the properties and calls `mk-@` and `mk-!` for each property. We'll assume that the properties are always on one line:

```
: prop: ( |s -- )
     args [:
          dup mk-! mk-@
     ;] reduce
;
```

This last step completes the solution.

Quiz 2.4

Quiz 2.4.0: Write out all three steps and test out your code. Be sure to complete the implementation for `mk-@`.

Quiz 2.4.1: Suppose we wanted to use a comma `,` to separate the property names. Eg:

```
prop: name , title , country
```

How would you do this? Test out your solution.

Immediate Words

Smojo has 2 separate modes of operation:

- **Interpretation Mode**: in which words are run when they are encountered in the token stream,
- **Compilation Mode**: where words are compiled as they are encountered in the token stream.

These modes are mutually exclusive. You are either in one or the other.

Smojo always starts in interpretation mode, but there are a few words that can switch Smojo from interpretation to compilation mode:

- When a new Word is being defined using colon `:`
- When a new Quotation is defined using `:>`
- Using the mode switching operator `]`

In the first two cases, once the system is in compilation mode, there is no way to switch back to interpretation mode because all subsequent words are compiled. To get around this, we have two classes of words:

1. **Ordinary Words** are compiled during compilation mode. This is the normal scenario. Most words are like this.
2. **Immediate Words** are executed during compilation mode. They are not compiled. A good example of an immediate words is `;` which completes a word definition and switches Smojo from compilation back to interpretation mode. `;` has to be immediate for it to work.

You can check an XT for immediacy by using `IMMEDIATE? ( xt -- f )`, eg:

```
' ; immediate? . cr
```

Should display:

```
true
ok
```

You can also create your own immediate words using `(IMMEDIATE) ( xt -- )`. For example:

```
:> "hello" . cr ; constant fn
fn immediate? . cr
fn (immediate)
fn immediate? . cr
```

Will display:

```
false
true
ok
```

The word `IMMEDIATE` is used more often, to mark a word as immediate, like so:

```
: greet ( -- )

    "hello" . cr

; immediate

' greet immediate? .
```

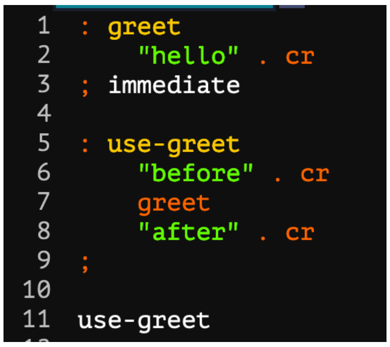
Will display:

```
true ok
```

Because `greet` is an immediate word, it is executed, not compiled in compilation mode. For example, if we use `greet` in another word as in Listing 1, we get the unexpected response:

```
hello
before
after
ok
```

The initial **`hello`** is caused by `greet` executing when `use-greet` is being compiled. Running:



```
: greet
    "hello" . cr
; immediate

: use-greet
    "before" . cr
    greet
    "after" . cr
;

use-greet
```

Listing 1: Using GREET

```
' use-greet decompile
```

we see:

```
[0]  Literal(before)
[1}  .
[2}  CR
[3]  Literal(after)
[4}  .
[5}  CR
```

In other words, `greet` is nowhere to be found in the body of `use-greet`.

Immediacy is very important for programming in Smojo, but it only makes sense in conjunction with other words, especially `POSTPONE`, `LITERAL` and others.

Quiz 2.5

Quiz 2.5.0: Is bracket-tick `[']` an immediate word? How about tick `'` ? Think about it first, then check your answers using `IMMEDIATE?`

Quiz 2.5.1: Is the word `LITERAL` an immediate word? What does it do?

Quiz 2.5.2: Would the `prop:` word work in compilation mode (ie, within a word)? How might this be solved? We will revisit this point later in this session.

Quiz 2.5.3: What do the words `[` and `]` do? **Hint**: Read the "switching modes" section in the Smojo Tutorial.

Postponing Immediacy

The word `POSTPONE ( |s -- )` is a word that **prevents** an immediate word from executing during compilation mode. `POSTPONE` is used very frequently in Smojo metaprogramming. So much that there is a common alias for `POSTPONE`, which is the back-tick `` ` ``.

Don't confuse the back-tick ` (which is just **POSTPONE**) with the tick **'**.

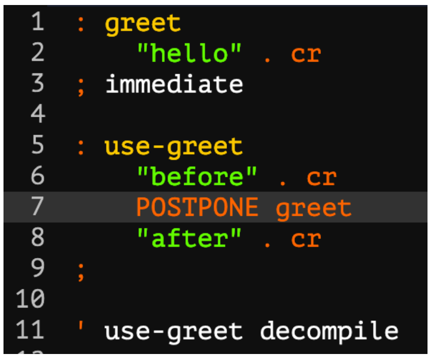
Listing 2 shows the **greet** & **use-greet** example using **POSTPONE**. This time, if we decompile use-greet, we get a very different result:

```
[0]  Literal(before)
[1}  .
[2}  CR
[3}  GREET
[4]  Literal(after)
[5}  .
[6}  CR
ok
```

You can see on Line 3 that **GREET** has been compiled into the body of **use-greet**. In other words,

POSTPONE greet

suppresses the immediacy of **greet**.



```
1  : greet
2      "hello" . cr
3  ; immediate
4
5  : use-greet
6      "before" . cr
7      POSTPONE greet
8      "after" . cr
9  ;
10
11 ' use-greet decompile
```

Listing 2: Using POSTPONE

Quiz 2.6

Quiz 2.6.0: Run Listing 2 on the Smojo editor. What happens if you run use-greet?

Quiz 2.6.1: Try changing line 7 of Listing 2 from **POSTPONE** greet to **POSTPONE DUP**, then just decompile **use-greet**. Does the result surprise you? Try changing **DUP** to **;** and decompile again. Does **POSTPONE** treat ordinary and immediate words the same way?

I will not discuss postponing ordinary words in this session as that is an advanced topic.

Example #1: Bracket-Tick

Bracket-tick **[']** might be defined this way using **IMMEDIATE**, **POSTPONE** and **LITERAL**:

```
: ['] ( |s -- )
    bl parse lcase
    xt-from-name ` literal
; immediate
```

- The first line `bl parse lcase` extracts the name of the word as a lowercase string. Recall that `BL` is just the empty space character.
- The second line gets the XT from the name and makes a literal from it. `XT-FROM-NAME ( "name" -- XT| null )` looks up the XT from the current dictionary using the given name. `LITERAL` is an immediate word, so its execution needs to be suppressed using `POSTPONE`. I've used alias back-tick ` for `POSTPONE`.
- We also need to mark `[']` as an `immediate` word, since it needs to be executed during compile mode.
- I've used the notation `|s` in the stack comment to indicate that `[']` reads from the token stream.

Although this is a short example, it is not an easy one to understand: You must become familiar with the idea of interpretation/compilation modes and immediacy.

Quiz 2.7

Study the code for bracket-tick above carefully.

Quiz 2.7.0: What would happen if it were not made immediate? What would happen?

Quiz 2.7.1: What is the purpose of `POSTPONE LITERAL`? What would happen if this was omitted, or if just `POSTPONE` was omitted?

Quiz 2.7.2: Does `PARSE` read the next token ie, `lcase`? Explain your answer clearly. Hint: Is `PARSE` an immediate word or not? When will it be executed?

Quiz 2.7.3: Amend this code to suitably warn users that the XT does not exist if `XT-FROM-NAME` returns a `null` value. Ie,

```
: xyz
   ['] yabadabadoo
;
```

Should give a warning saying that "yabadabadoo is an unknown word"

Hint: `null? ( x -- f )` can check for a null value.

Example #2: Constants

Constants in Smojo are used to emulate a fixed value. For example, consider the use of a `constant` called `NUMBER`:

```
23 constant NUMBER
: xyz ( -- )
    NUMBER . cr
;
```

If `xyz` is decompiled (using `see xyz` or `' xyz decompile`), we obtain:

```
[0]  Literal(23)
[1}  .
[2}  CR
ok
```

You can see clearly that the literal value of the constant is present in the XT of `xyz`.

How can we create such a word `constant` by ourselves? Let's think through the steps:

1. First, the word created by `constant` is named `NUMBER`, and it needs to be executed in compilation mode. This means `NUMBER` must be immediate.
2. The action of `NUMBER` when Smojo is in compilation mode is to compile the literal it represents into the

body of the XT currently being defined. In the example above, it means `NUMBER` needs to compile in the literal `23` into the body of `xyz`.

3. If `NUMBER` is executed in interpretation mode, it needs to put the value it represents on the stack.

Let's turn this recipe into code:

In step 1, the syntax of constant is that it expects a value on the stack and a name on the token stream ahead. Eg,

```
23 constant NUMBER
```

means that `23` is already on the stack while `NUMBER` is ahead of `constant` in the token stream. So, at the very least, constant needs to save the value 23 and read the next token:

```
: next-token ( |s -- "s" ) bl parse ;
: bind ( "name" XT -- )
    get-dictionary #!
;

\ INCOMPLETE!
: constant ( v |s -- ) { v }
    next-token lcase \ name of const.
    [:
        \ TODO!
    ;] bind
;
```

This code is of course incomplete (we need steps 2 and 3), but you should understand the basic structure. The above code completes Step 1. Note that `next-token` and `bind` are actually built-in words -- so they do **not** need to be defined -- but I have included them here just for completeness. We use `lcase` to convert names like `NUMBER` into `number`, as this is the standard form used in Smojo's dictionary.

For Steps 2 and 3, we need to use the word

```
COMPILATION? ( -- f )
```

that tells us if Smojo is in compilation mode (`true`) or not. With this, we can easily complete the code for `constant`:

```
\ STILL INCOMPLETE!
: constant ( v |s -- ) { v }
     next-token lcase \ name of const.
     [:
          v compilation? -> ` literal |.
     ;] bind
;
```

This version is still incomplete because the XT created by `constant` is not immediate. We use the word:

```
(IMMEDIATE) ( xt -- )
```

to make any XT into an immediate one. So finally we have:

```
\ FINAL VERSION
: constant ( v |s -- ) { v }
     next-token lcase \ name of const.
     [:
          v compilation? -> ` literal |.
     ;] dup (immediate) bind
;
```

Quiz 2.8

Quiz 2.8.0: Create your own constant using the code above and test it throughly. Make sure it works in interpretation mode, and compilation mode for both ordinary words and also within Quotations.

Quiz 2.8.1: Make a new form of constant that will warn users if the name used by the constant already exists. Eg:

```
123 constant HELLO
```

```
"Hello" constant HELLO
```

Will generate a warning "WARNING: Word HELLO exists."

Should we abort in such circumstances?

Quiz 2.8.2: Why are the names of words in Smojo's dictionary always stored in lowercase? What do you think is the rationale?

Example #3: Special Contexts

The ability of Smojo's dictionary to be amended by Smojo programs has many practical applications.

Especially for metaprogramming, we will need names like `+` or `-` or `dup` to take on new meanings **temporarily**.

As a practical example, suppose we want to temporarily use the symbols `+`, `-`, `*` and `/` to mean arithmetic between two sequences, instead of two numbers. But we don't want to replace the ordinary meaning of these arithmetic symbols permanently, only temporarily.

To do this, we create a pair of **context switching** words, for example `v{` and `}v` to temporarily save the old meanings of our words, then put in the new ones.

In other words:

- `v{` would save the current dictionary to a variable, and replace it with one containing our new words,
- `}v` would restore the saved dictionary.
- So, **outside** `v{` ... `}v`, the arithmetic symbols would have their original meaning,
- **inside** `v{` ... `}v` they would mean arithmetic over sequences.

Here's how it could be done:

```
null variable OLD-DICTIONARY
```

```
: }v ( -- )
    OLD-DICTIONARY @ set-dictionary
;

\ INCOMPLETE:
: export-words ( -- # ) .... ;

: v{ ( -- )
     get-dictionary OLD-DICTIONARY !
     export-words set-dictionary
;
```

This code is complete except for **EXPORT-WORDS** that we used to load our custom words into a new hash. There are many ways to do this:

- A simple way is to assume that the words we want to export into the special context `v{` ... `}v` have a standard prefix, eg `_+` or `_*` Here, the underscore is used to prefix and to separate `_+` from `+` in the original dictionary.
- Another way is to explicitly say what the out-context and in-context names are. Eg, we could explicitly specify that `_+` in the out-context is now called `+` in-context. This way, we don't need standard prefixes.
- Another way is to use Smojo's module system and specify which of these we want to export.

All three methods are valid. However, I will illustrate using the second method.

Let's say we already have these words:

```
\ adds 2 sequences.
: s+ ( seq seq -- seq ) ['] +. zipwith ;
\ subtracts 2 sequences.
: s- ( seq seq -- seq ) ['] -. zipwith ;
```

```
\ Multiplies 2 sequences pairwise
: s* ( seq seq -- seq ) ['] *. zipwith ;

\ multiplies seq with num
: c* ( seq n -- seq ) { n }
     [: n *. ;]  map
;

\ divides seq with num
: c/ ( seq n -- seq ) { n }
     [: n /. ;] map
;

\ Raises to a power
: c^ ( seq n -- seq ) { n }
     [: n ^ ;] map
;

\ Finds the sum of a sequence
: sumseq ( seq -- n )
     0 swap ['] +. reduce
;
```

These 7 words encompass all common operations in linear algebra, if vectors were represented by sequences.

For example, we can express the dot product between two vectors as:

```
: dot ( seq seq -- n ) s* sumseq ;
```

Our goal is to create a special context `v{` ... `}v` where we can more naturally write:

```
: dot ( seq seq -- n ) ** sum ;
```

This may seem a lot of work for very little payoff! But consider the following:

1. For realistic metaprogramming work we need to redefine many basic Smojo words. To use `_:` , `_;` etc directly in our code is confusing and causes errors. Much better to use `:` , `;` etc. Easier to write and read.

2. If we redefine basic words in the dictionary, that would alter subsequent words. Eg, if we redefine `:` directly without giving it a new name, subsequent definitions would use this redefined version of `:`. This can often cause subtle errors or hard-to-debug crashes. Best to cleanly separate new words from old ones, like I've done above for the vector arithmetic example. None of the new words exist in the original dictionary.

The technique I outline here sidesteps both these series issues by (a) giving our new words a distinct name in the original dictionary but (b) temporarily renaming them within the special context.

Like most programming languages, Smojo has a module system (we'll take a closer look at this in Example #4), but using modules won't solve problem #2.

In short, it's true the vector arithmetic example above isn't super practical. I've chosen it because it is easy to understand. Going forward, here's how we might define `EXPORT-WORDS`:

```
: copy-dictionary ( -- # )
     # { h }
     h get-dictionary >#
     h
;

: export-words ( -- # )
     copy-dictionary { h }
     "+" ['] s+ h #!
     "-" ['] s- h #!
```

```
    "**" ['] s* h #!
    "*" ['] c* h #!
    "/" ['] c/ h #!
    "^" ['] c^ h #!
    "sum" ['] sumseq h #!
    h
;
```

With this completed, our new `v{` ... `}v` should work as expected. Note that all the original Smojo words are imported by `copy-dictionary`. Just a few words (5) are overwritten and two new words are defined (`**` and `sum`).

Quiz 2.9

Quiz 2.9.0: Copy paste this code into your Smojo editor and get it to work. Test out:

```
v{
: dot ( seq seq -- n ) ** sum ;
{{ 1 2 3 4 5 }} {{ 3 4 5 6 7 }} dot .
}v
```

Quiz 2.9.1: Can we access words like `*` and `-` with their original meaning within `v{` ... `}v` ? Is this a problem? If so, how would you overcome it?

Quiz 2.9.2: It quickly becomes tedious, impractical or error prone to manually bind the new names. Write a word `% ( # | new old -- # )` that uses our old friends `PARSE`, `TOKENIZE` , `XT-FROM-NAME` and `IMMEDIATE` to do this binding automatically. Eg:

```
: % ( # | new old -- # ) .... ;
: export-words ( -- # )
   #
   % + s+
    % - s-
    % * c*
    ...
;
```

Use this in your new **EXPORT-WORDS** and test.

Quiz 2.9.3: Your friend Joe wants to extend % so that it can read multiple words, separated by a comma, eg:

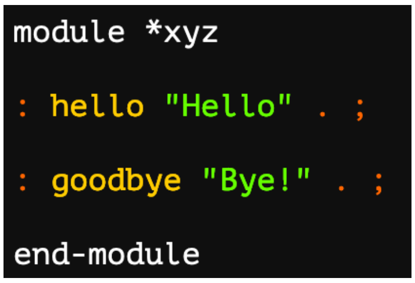
```
    % + s+ , - s- , * c*
```

Is this a good idea?

Example #4: Modules

Smojo's module system is built entirely using the basic metaprogramming words and the words `get-dictionary` and `set-dictionary`, along with the usual data structures.

Listing 3 shows a simple module called `*XYZ` containing 2 words, `hello` and `goodbye`.



```
module *xyz
: hello "Hello" . ;
: goodbye "Bye!" . ;
end-module
```

Listing 3: A simple module

Modules help to manage **namespaces**. In this example, the words `hello` and `goodbye` only exist within the module, not outside it. Also, if there is another `hello` defined outside the module, `*xyz`'s `hello` will not overwrite it. You can activate `*xyz`'s words by prefixing with `*xyz`. Eg,

```
: hello ( -- )
    "How are you?" . cr
;
hello      \ Prints How are you?
*xyz hello \ Prints Hello
```

How is this done? Essentially:

1. `module` saves the name of the module as the next token in the token stream (using `bl parse lcase`)

2. It then saves a copy of the current dictionary (using `get-dictionary`) and creates a copy of the current dictionary (using `clone`) and replaces the dictionary with the clone (using `set-dictionary`).

3. From this point on, all newly defined words are saved to the cloned dictionary. The original saved version is untouched.

4. The word `end-module` completes the module. It first creates an empty hash to store the newly defined ("name",XT) pairs. Let's call this hash `h`.

5. It then compares the XTs in the cloned version with that of the original dictionary. If an XT is in the cloned version but not in the original, it is added to `h`.

6. Once this is done, it reverts the dictionary to the original version, using `set-dictionary`.

7. It creates an XT using a Quotation. This XT looks ahead in the token stream to determine which module word is called, then retrieves the XT from `h`.

8. Lastly, the end-module word binds the newly-created XT with the name of the module, using `BIND` (which we've met earlier, and is based on `get-dictionary`).

This is a lot of steps! Don't worry if you don't understand it all right now. Let's start with the code:

```
null variable OLD-DICTIONARY
```

```
null variable MODULE-NAME
```

These are our global variables, storing the original dictionary and name of the dictionary. Continuing:

```
: module ( |s -- )
   bl parse lcase MODULE-NAME !
   get-dictionary OLD-DICTIONARY !
   copy-dictionary set-dictionary
```

```
;
```

I've re-used `copy-dictionary` from Example #3 above. That's it for the `module` word. We can now work on the helpers required for `end-module`:

We first need a word to get all the new words defined in the cloned dictionary:

```
: lookup ( # -- # )
     # { h }
     #values [: dup h #! ;] reduce
     h
;


: get-new-words ( -- # )
     # { h }
     OLD-DICTIONARY @ lookup { p }
     GET-DICTIONARY { d }
     d #keys [: { n } \ name of the word
          n d #@ { xt } \ its XT
          \ Ignore words that are
          \ in the old dictionary
          xt p #contains? -> exit |.
          \ Save the new word into H
          n xt h #!
     ;] reduce
     h
;
```

This code simply carries out Step 4 & 5. The `lookup` word creates a lookup table based on the input dictionary.

With these helpers we can finally build the `end-module` word. We'll do this in a few steps:

```
\ DRAFT #1 -- INCOMPLETE!

: end-module ( -- )
   MODULE-NAME @       \ Step 1
```

```
    get-new-words { h } \ Step 2
                        \ Step 3
    OLD-DICTIONARY @ set-dictionary
    [: ( |s -- )
      \ ... TODO ...
    ;] dup (immediate)  \ Step 4
       bind             \ Step 5
;
```

In this first draft, you can clearly see that `end-module`

1. Reads the module name. This is just stored on the stack.

2. Calculates the (name,XT) pairs of the newly created words. This is saved into the local `h`.

3. Restores the old dictionary. This has to be done **after** Step 2. Why?

4. We make the XT immediate so that it will execute even during compilation mode.

5. Lastly, we bind the name of the module (already on the stack in Step 1) to the incomplete XT. This XT is `( |s -- )` meaning it expects the name of the word to lookup on the token stream. For example, `*xyz hello`, the `*xyz` is the name of the module and it needs to find the word named `hello` within the hash `h`.

A simple way to implement Step 5 is to directly check the hash `h`:

```
: execute-or-compile ( XT -- )
     compilation? -> compile, exit |.
     execute
;
```

```
\ DRAFT #1 -- COMPLETE, BUT CAN BE
\ IMPROVED.
: end-module ( -- )
   MODULE-NAME @
   get-new-words { h }
   OLD-DICTIONARY @ set-dictionary
   [: ( |s -- )
     bl parse lcase
     h #@ execute-or-compile
   ;] dup (immediate) bind
;
```

`h #@` retrieves the XT if it exists and executes or compiles it.

The helper `execute-or-compile ( xt -- )` behaves differently depending on whether it is called in compilation mode (in which case the XT needs to be compiled) or executed in interpretation mode. The word `COMPILE, ( xt -- )` compiles the given XT into the current word.

This solution isn't great. That's because the module's collection of words are inaccessible to other words, since `h` is "hardcoded" into the XT itself. You can't easily access it.

What could we do if the hash `h` were accessible to other words? Well, here are some ideas:

- We could create a word to list the functions on any module,
- We could iterate through the module's words and sandbox them for security,
- We might add to the hash new words, for example for documentation.
- We could rename the module words dynamically if there is a name clash,

- We could introduce checksums to prevent tampering with the words.

None of this is possible if `h` is hidden within the XT's body as a literal.

So, to expose `h`, we make use of the XT's **data section**. This gives us 2 problems we need to solve:

1. How do we add `h` to the XT's data section? The answer is by using the comma `,` word. This has to be done just after the XT is being created. So we have to use the mode switch words `[` ... `]` to temporarily drop into interpretation mode.

2. The second problem is how to retrieve `h` from the XT's data section when it is being executed? The answer is to use the built-in word `self ( -- XT )` that puts itself on the stack. Once the XT is on the stack, we can use `@` to fetch the hash `h`.

Here's a solution for the XT part alone:

```
[: ( |s -- )
   [ h , ] \ add h to the data section.
   bl parse lcase \ name of word to find.
   self 0 @ #@ \ get XT from stored h.
   execute-or-compile
;]
```

In this code,

- The phrase `[ h , ]` simply means switch to interpretation mode, then adds `h` to the data section of the XT that is being defined. And then restores compilation mode.

- `self 0 @ #@` means first put the XT on the stack, then read the zeroth-index element from the data section (this will be the hash `h`) then retrieve the XT of the word from this hash. Remember the name of the word is already on the data stack from the earlier `bl parse lcase`.

This brings us to the final version for end-module:

```
\ FINAL VERSION
: end-module ( -- )
   MODULE-NAME @
   get-new-words { h }
   OLD-DICTIONARY @ set-dictionary
   [: ( |s -- )
     [ h , ]
     bl parse lcase
     self 0 @ #@ execute-or-compile
   ;] dup (immediate) bind
;
```

It helps to think what we have just done: We have built a usable module system from scratch using just a few metaprogramming words: `parse`, `literal`, `immediate`, `postpone`, `latestXT`, `compilation?`, `compile,` , `[` , `]` and `,`. With these words and a couple more, we can essentially change Smojo to our liking.

Quiz 2.10

Quiz 2.10.0: Copy paste the code for modules into your Smojo editor and get it to work. Be sure to test it out thoroughly.

Quiz 2.10.1: The built-in module system does not warn you if the word does not exist in the hash h. Amend the module code so that it does.

Quiz 2.10.2: Write `immediate` in terms of `(immediate)` and some other metaprogramming words.

Quiz 2.10.3: I mentioned one of the benefits of putting the hash h into the module's data section is that it would allow you to do many things, including listing the words in a module. Write a word `.MODULE ( |s -- )` that does this. Eg,

```
.module *xyz
```

Should list out all the words in `*xyz`.

Quiz 2.10.4: How would you add documentation capabilities to a module? Discuss this.

Notes on SELF

I will end by saying that we can also define `self ( -- XT )` using our friends `postpone` , `literal`, `latestXT` and the switch mode words `[` and `]`:

```
: (self) latestxt ` literal ;

: self
    ` [
    ` (self)
    ` ]
; immediate
```

This code is an advanced usage of metaprogramming words. You don't have to understand how it works for now. It relies on the behaviour of `postpone` for ordinary words, since `(self)` is an ordinary word. I haven't explained this yet. We will cover this in later sessions.

You only need to know what `self` does:

- At compilation time, it compiles the XT of the currently defined XT into itself.
- At execution time, the result is to place the XT on the stack.

`self` is often useful when we are dealing with data from the XT's data section.

Summary of Words

`PARSE ( char -- "s" )` reads the token stream until the delimiting character `char` is met. It places the text it reads onto the stack. The token stream is resumed just past `char`.

`[CHAR] ( |s -- char )` converts the next token into a character. `\t` means the tab character, `\n` the end of line character.

`TOKENIZE ( "s" "delim" -- tuple )` splits s into a tuple using the regular expression `delim`. Simple regular expressions are `""` which splits the strings into single characters, `"\s+"` which splits it using one or more whitespaces. More examples can be found online by searching "Java Regular Expressions" which is what Smojo uses.

`ARRAY>SEQ ( tuple -- seq )` converts an array into a sequence.

`LITERAL ( x -- )` is an immediate word that takes whatever is on the stack and converts it into a literal. This is needed when you want to create a word that can compile literals into *another* word.

- We've seen it used in `[']`, where it compiles the XT of the ticked word into the currently defined word as a literal.
- We've also seen it used in `constant`, where it compiles the value of the constant into the word where the constant is used.
- Lastly, we've seen it in `self`, where it is used to compile the literal value of the XT into itself.

It will take you some time to really get a good feeling on how to use `literal`.

`(IMMEDIATE) ( xt -- )` makes the given XT immediate. This is a one-way ticket: there is no easy way to remove immediacy except to define a new non-immediate word.

`IMMEDIATE ( -- )` uses the `latestXT` and makes it immediate.

`POSTPONE ( |s -- )` **temporarily** prevents an immediate word from executing during compilation mode. It compiles the word into the currently defined word.

- `POSTPONE` must only be used in compilation mode. It cannot be used in interpretation mode.
- `POSTPONE` of an ordinary word (eg, like `DUP`) is different from postponing an immediate word. We will discuss this in later sessions, but we have seen an example of this usage in `self`.
- A common alias for `POSTPONE` is back-tick `` ` ``.

`GET-DICTIONARY ( -- # )` and its companion `SET-DICTIONARY ( # -- )` get and set the current dictionary. All words in Smojo are stored on the dictionary as ("name",XT) pairs. These are a very useful pair of words, as we have seen.

`LATESTXT ( -- xt )` puts the last defined XT on the stack. This word is often useful in metaprogramming.

`SELF ( -- xt )` is a word with two different actions: In compilation mode, it compiles the current XT into itself as a literal. When it is executed in interpretation mode, the XT is put on the stack.

`[` and `]` are mode switching words. Both are immediate words that switch compilation/interpretation modes. A common example is their use in adding to the data section of a currently defined word (we saw this in Example #4: Modules), but a more advanced usage is in switching modes in another word, as we saw in the definition of `self`.

`COMPILE, ( xt -- )` compiles the given XT into the currently defined word.

Here is a last fun example using the words above and **COMPILE,** to create an XT at runtime:

```
\ This is a simple program supplied as a
\ sequence.

{{
    "Hello"
    ' .
    ' cr
    "World"
    ' .
    1
    2
    ' +
    ' .
}} => MY-PROGRAM

\ This is how we can check if something
\ is an XT.
: xt? ( x -- f )
   "com.terraweather.mini.XT" instanceof?
;

\ Compiles XTs but makes everything else
\ a literal
: (program) ( seq -- )
    [:
        dup xt? -> compile, exit |.
        ` literal
    ;] reduce
;

\ Runs (program), much like how
\ self works.
: program, ( -- )
```

```
    ` [
    ` (program)
    ` ]
; immediate

\ Main word that creates a custom XT
\ using data from a sequence.
: prog ( seq -- xt )
    [: program, ;]
;

\ Test decompilation and execution:
MY-PROGRAM prog decompile cr
MY-PROGRAM prog execute
```

Quiz 2.11

Quiz 2.11.0: Copy paste this code into your Smojo editor and get it to work. Try creating different programs.

Quiz 2.11.1: How might we remove the use of tick `'` in specifying the program? Hint: use `xt-from-name` and use strings instead. Eg, use `"cr"` not `' cr`

Quiz 2.11.2: From your answer in 2.11.1, write a word `EVAL ( "program" -- )` that evaluates a string as a Smojo program. **Hints**:

- Write a word `convert ( "s" -- seq )` that transforms the input program into a sequence of XTs , integers and Strings, like we had in `MY-PROGRAM`.
- You need to use `tokenize` and `array>seq`.

- You also need to use `[regex]` to test if something is an integer. Eg:

```
: integer? ( "s" -- f ) [regex] \d+ ;
```

- You need a way to tell if something is a string. For now, you could use single-quotes '

Test out your work with the following program:

```
" 'Hello' . cr . cr 1 2 + ." eval
```

Quiz 2.11.3: If you have successfully completed 2.11.2, then think about how you might represent strings with spaces. **Hint**: You might want to be lazy and replace spaces with underscores. That is an easy solution. Handling actual spaces is much harder. You would have to make `convert` do more work to keep track of strings.

Quiz 2.11.4: If you completed the above, you are very close to defining Smojo in Smojo. Give some thought how this might be done!