Parsers

Creating Custom Parsers

Smojo has an important facility to let you amend it's syntax. As a simple example, let's consider this piece of code:

```
\ WON'T WORK!
1/2 3/4 +. .
```

In this program, we want Smojo to somehow recognise fractions like `1/2` or `3/4`. **Parsers** are the way to do this.

Note: Don't confuse this concept with the `PARSE` word!

First, you need to understand how Smojo processes tokens.

Processing the Token Stream

As you already know Smojo reads input code text as a token stream, token by token. This causes the right-to-left behaviour of Smojo.

But what happens once a token is read?

Here is some pseudocode to outline the process:

Step 1: Read in the next token, `T`. Note that `T` is a string.

Step 2.a: Check if `T` is an **integer**, **real** or **string**. This check is done by directly examining the string `T`. For example:

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

If `T` is an integer/real/string, then it is converted to an actual object of the right type. Note that strings need to be converted since the token read will have a surrounding double-quote "..." , which has to be removed.

This resulting object is bound to a variable **V**, and we move to Step 3.

Step 2.b: If the token **T** is not an integer/real/string, it is assumed to represent a word. In this case the dictionary is searched for the XT of **T**. This is done using **xt-from-name**:

```
: xt-from-name ( "T" -- xt | null )

    lcase get-dictionary #@

;
```

The resulting XT (or null value if it is not found) is then bound to the variable **V**. In the case of a null value, the system will warn the user with an error message saying "No such word"

Step 3: What happens next depends on compilation mode Smojo is in.

Step 3.a: In Compilation Mode:

(i) If **V** is an integer/real/string, it is added into a Literal and this literal is compiled into the current word being defined.

(ii) Otherwise **V** is an XT, and if it is immediate, it is executed*. If not immediate, it is compiled directly into the current word.

* **Note**: The mode switching words **[** and **]** are handled slightly different if they are embedded in Quotations. This has been discussed in Session 3. This exception is needed for correctly initialising the data sections of Quotations.

Step 3.b In Interpretation Mode:

(i) If **V** is an integer/real/string, it is placed on the data stack.

(ii) Otherwise (**V** is an XT), it is executed.

We then loop back to Step 1, and process the next token.

Parsers

Take the time to thoroughly understand Steps 1 - 3.

Parsers allow you to tap into this process by adding your own custom processing right after Step 1 (the token being read from the token stream) and before Step 2 (the token being processed).

A parser is any XT which takes in a token (ie, string) and returns a specific set of values:

```
( "token" -- 0 | x 1 | -1 )
```

- A return value of `0` means "ignore this token", ie, it will **not** be processed further. None of the following steps 2 and 3 will be performed.
- A double value return of `x 1` means use `x` as transformed token, and continue downstream processing, **starting at Step 3**.
- A value of `-1` means use the token "as is", ie, continue processing it as usual. Ie, processing of the token resumes at Step 2 (or passed to the next parser to handle, if there is more than one).

Finally, to register a parser to Smojo, we use

```
+PARSER ( xt -- )
```

where `xt` is the parser you want to register. Similarly, you can remove a parser using

```
-PARSER ( xt -- )
```

Note that you can add as many parsers as you wish, but they will run in the order they are added.

Example #1: Parsing Fractions

Let's go back to the program for adding fractions and get it to work using parsers.

```
\ WE WANT TO MAKE THIS WORK!

1/2 3/4 +. .
```

We want this program to display either `1.25` (easy) or `5/4` (harder).

Let's tackle the easier option first. Essentially we need to add a parser that checks if the token is indeed a fraction:

- If it is, it converts it into a real and uses the x 1 return.
- If it is **not** a fraction, we just return -1 to signal to Smojo that the token needs to be processed by someone else.

The code:

```
: my-parser ( "s" -- 0 | x 1 | -1 ) { s }
    s fraction? -> s to-real 1 exit |.
    -1
;
```

We need the words `fraction? ( "s" -- f )` to detect a fraction. This is easy enough, but you have to be careful to handle negative numbers in the numerator:

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

The word `to-real ( "s" -- n )` converts a fraction into a real number:

```
: to-real ( "s" -- n )
    "/" tokenize { xs }
    xs 0 @@ int
    xs 1 @@ int /.
;
```

Lastly, we need a word to register my-parser into Smojo:

```
: use-fractions ( -- )
     ['] my-parser +parser
;
```

That's it! Now, we can use fractions in interpretation mode:

```
use-fractions \ invoke the parser.
\ WORKS!!!
1/2 3/4 +. .
```

Will respond with:

```
1.25 ok
```

You can also test this out in compilation mode:

```
use-fractions
: xyz ( -- )
     7/2 -3/4 +. .
;
see xyz
```

Will respond with:

```
[0]  Literal(3.5)
[1]  Literal(-0.75)
[2]  +.
[3}  .
ok
```

Quiz 4.0

Quiz 4.0.0: Test out this code and ensure it works for you.

Quiz 4.0.1: How would you implement the "harder" option of displaying fractions?

Hint:

(i) You need to store fractions in a 2-tuple, to store the numerator `n` and denominator `d`, `(n,d)`

(ii) Alter `my-parser` to convert integers into fractions. Eg, `23` becomes `(23,1)`

(iii) Alter all integer arithmetic operations `+`, `-`, `*` and `/` to handle these operations between tuples of the form `(n,d)`.

(iv) Add an explicit casting operator `(real)` `( tuple -- n )` to convert a fraction into a real.

Implement these suggestions and ensure your code works.

Quiz 4.0.2: What are some drawbacks to the solution of Quiz 4.0.1? Can these be mitigated?