IMPLEMENTERS

an old feature still evolving

context **2020** meeting

Interfacing with Lua

- Quite some activity is delegated to Lua.
- Normally the initiative is at the TEX end.
- We can set variables or call functions etc.
- We can parameters to function calls.
- From the Lua end we can use scanners to pick up data.
- We provide some consistent interfaces for doing all that.
- From TEX to Lua we use `\ctxlua{...}` and friends.
- From Lua to TEX we use `context(...)` and alike.
- For adding functionality we use so called implementers.

Calling Lua

```
\ctxlua{context("ok")}
```

ok

```
\ctxlua{context(2 * tokens.scanners.integer())} 10
```

20

```
\startluacode
function document.MyThing() context(2 * tokens.scanners.integer()) end
\stopluacode
```

```
\ctxlua{document.MyThing()} 20 \quad
\ctxlua{document.MyThing()} 30 \quad
\ctxlua{document.MyThing()} 40
```

40 60 80

Streamlining Lua

```
\startluacode
interfaces.implement {
    name      = "MyThing",
    public    = true,
    arguments = "integer",
    actions   =   function(i) context(i * 2) end,
 -- actions   = { function(i)  return i * 2  end, context },
}
\stopluacode
```

```
\MyThing 20 \quad \MyThing 30 \quad \MyThing 40
```

40 60 80

Making commands

```
\startluacode
interfaces.implement {
    name      = "MyRoot",
    public    = true,
    actions   = function()
        local a = tokens.scanners.integer()
        if not tokens.scanners.keyword("of") then
         -- tex.error("the keyword 'of' expected")
        end
        local b = tokens.scanners.integer()
        context("%0.6N",math.sqrt(b,a))
    end,
}
\stopluacode
```

```
\MyRoot 2 of 40 \quad \MyRoot 3 60
```

```
6.324555  7.745967
```

Scanners

There are lots of scanners:

`argument argumentasis array boolean box bracketed bracketedasis cardinal char cmdchr cmdchrexpanded code conditional count csname delimited dimen dimenargument dimension float glue gluespec gluevalues hash hbox integer integerargument ischar key keyword keywordcs letters list luacardinal luainteger luanumber lxmlid next nextchar nextexpanded number optional peek peekchar peekexpanded real scanclose scanopen skip skipexpanded string table token tokencode tokenlist tokens toks value vbox verbatim vtop whd word`

A more complex example

Let's implement a matcher:

```
\doloopovermatch {(.)} {luametatex} { [#1] }
```

[l] [u] [a] [m] [e] [t] [a] [t] [e] [x]

```
\doloopovermatch {([\letterpercent w]+)} {\cldloadfile{tufte.tex}} { [#1] }
```

[We] [thrive] [in] [information] [thick] [worlds] [because] [of] [our] [marvelous] [and] [everyday] [capacity] [to] [select] [edit] [single] [out] [structure] [highlight] [group] [pair] [merge] [harmonize] [synthesize] [focus] [organize] [condense] [reduce] [boil] [down] [choose] [categorize] [catalog] [classify] [list] [abstract] [scan] [look] [into] [idealize] [isolate] [discriminate] [distinguish] [screen] [pigeonhole] [pick] [over] [sort] [integrate] [blend] [inspect] [filter] [lump] [skip] [smooth] [chunk] [average] [approximate] [cluster] [aggregate] [outline] [summarize] [itemize] [review] [dip] [into] [flip] [through] [browse] [glance] [into] [leaf] [through] [skim] [refine] [enumerate] [glean] [synopsize] [winnow] [the] [wheat] [from] [the] [chaff] [and] [separate] [the] [sheep] [from] [the] [goats]

A more complex example (TEX)

Here is the macro definition of this loop:

```
\protected\def\doloopovermatch#1#2#3%
  {\pushmacro\matchloopcommand
   \def\matchloopcommand##1##2##3##4##5##6##7##8##9{#3}%
   \ctxluamatch\matchloopcommand{#1}{#2}%
   \popmacro\matchloopcommand}
```

- The pushing and popping makes it possible to nest this macro.
- The definition of the internal match macro permits argument references.

A more complex example (Lua)

At the Lua end we use an implementer:

```
local escape = function(s) return "\\" .. string.byte(s) end

interfaces.implement {
    name    = "ctxluamatch",
    public  = true,
    usage   = "value",
    actions = function()
        local command = context[tokens.scanners.csname()]
        local pattern = string.gsub(tokens.scanners.string(),"\\.",escape)
        local input   = string.gsub(tokens.scanners.string(),"\\.",escape)
        for a, b, c, d, e, f, g, h, i in string.gmatch(input,pattern) do
            command(a, b or "", c or "", d or "", e or "", f or "", g or "",
                h or "", i or "")
        end
        return tokens.values.none
    end,
}
```

So what does the usage key tells the implementer?

Value functions

Normally we pipe back verbose strings that are interpreted as if they were files. Value functions are different;

- The return value indicates what gets fed back in the input.
- This can be: none, integer, cardinal, dimension, skip, boolean, float, string, node, direct.
- When possible an efficient token is injected.
- Value function can check if they are supposed to feed back a value.
- So, they can be used as setters and getters.
- A variant is a function that is seen as conditional.
- In (simple) tracing they are presented as primitives.
- They are protected against user overload (aka: frozen).
- All this is experimental and might evolve.

So

Say that we want an expandable command:

```
\edef\foo{\doloopovermatched{.}{123}{(#1)}} \meaning\foo
```

macro:(1)(2)(3)

Or nested:

```
\edef\foo {%
    \doloopovermatched {(..)} {123456} {%
        \doloopovermatched {(.)(.)} {#1} {%
           [##1][##2]%
        }%
    }%
} \meaning\foo
```

macro:[1][2][3][4][5][6]

So

Compare:

```
\protected\def\doloopovermatch#1#2#3%
  {\pushmacro\matchloopcommand
   \def\matchloopcommand##1##2##3##4##5##6##7##8##9{#3}%
   \ctxluamatch\matchloopcommand{#1}{#2}%
   \popmacro\matchloopcommand}
```

With:

```
\def\doloopovermatched#1#2#3%
  {\beginlocalcontrol
     \pushmacro\matchloopcommand
     \def\matchloopcommand##1##2##3##4##5##6##7##8##9{#3}%
   \endlocalcontrol
   \the\ctxluamatch\matchloopcommand{#1}{#2}%
   \beginlocalcontrol
     \popmacro\matchloopcommand
   \endlocalcontrol}
```

Local control hides the assignments (it basically nests the mail loop).

A few teasers (TeX)

```
\doloopovermatch {(\letterpercent d+)} {this 1 is 22 a 333 test} { [#1] }

\doloopovermatch {(\letterpercent w+) *(\letterpercent w*)} {aa bb cc dd} {
    [
        \doloopovermatch{(\letterpercent w)(\letterpercent w)} {#1} {(##1 ##2)}
        \doloopovermatch{(\letterpercent w)(\letterpercent w)} {#2} {(##1 ##2)}
    ]
}

\doloopovermatch
    {(.-)\letterpercent{\bf (.-)\letterpercent}(.*)}
    {this is {\bf a} test}
    {#1{\it not #2}#3}
```

A few teasers (Lua)

```lua
interfaces.implement {
    name = "bitwisexor", public = true, usage = "value", actions =
    function(what)
        local a = tokens.scanners.cardinal()
        scankeyword("with")
        local b = tokens.scanners.cardinal()
        if what == "value" then
            return tokens.values.cardinal, a ~ b
        else
            logs.texerrormessage("you can't use \\bitwiseor this way")
        end
    end
}
interfaces.implement {
    name = "ifbitwiseand", public = true, usage = "condition", actions =
    function(what)
        local a = tokens.scanners.cardinal()
        local b = tokens.scanners.cardinal()
        return tokens.values.boolean, (a & b) ~= 0
    end
}
```

Questions and more examples

More examples will be given in the editor.