Up to ConTEXt MkVI

Introduction

No, this is not a typo: MkVI is the name of upcoming functionality but with an experimental character. It is also a playground. Therefore this is not the final story.

Defining macros

When you define macros in TEX, you use the # to indicate variables. So, you code can end up with the following:

```
\def\MyTest#1#2#3#4%
  {\dontleavehmode
   \dostepwiserecurse{#1}{#2}{#3}
     {\ifnum\recurselevel>#1 \space,\fi
      \recurselevel: #4\space}%
   .\par}
```

This macro is called with 4 arguments:

```
\MyTest{3}{8}{1}{Hi}
```

However, using numbers as variable identifiers might not have your preference. It makes perfect sense if you keep in mind that TEX supports delimited arguments using arbitrary characters. But in practice, and especially in ConTEXt we use only a few well defined variants. This is why you can also imagine:

```
\def\MyTest#first#last#step#text%
  {\dontleavehmode
   \dostepwiserecurse{#first}{#last}{#step}
     {\ifnum\recurselevel>#first \space,\fi
      \recurselevel: #text}%
   .\par}
```

In order for this to work, you need to give your file the suffix `mkvi` or you need to put a directive on the first line:

```
% macros=mkvi
```

You can of course use delimited arguments as well, given that the delimiters are not letters.

```
\def\TestOne[#1]%
  {this is: #1}

\def\TestTwo#some%
  {this is: #some}

\def\TestThree[#whatever][#more]%
  {this is: #more and #whatever}

\def\TestFour[#one]#two%
  {\def\TestFive[#alpha][#one]%
     {#one, #two, #alpha}}
```

You can also use the following variant which is already present for a while but not that much advertised. This method ignores all spaces in definitions so if you need one, you have to use `\space`.

```
\starttexdefinition TestSix #oeps

  here: #oeps

\stoptexdefinition
```

These commands work as expected:

```
\startlines
  \TestOne  [one]
  \TestTwo  {one}
  \TestThree[one][two]
  \TestFour [one]{two}
  \TestFive [one][two]
  \TestSix  {one}
\stoplines
```

this is: one
this is: one
this is: two and one
two, two, one
here: one

You can use buffers to collect definitions. In that case you can force preprocessing of the buffer with `\mkvibuffer[name]`.

Implementation

This functionality is not hard coded in the LuaTEX engine as this is not needed at all. We just preprocess the file before it gets loaded and this is something that is relatively easy to implement. Already early in the development of LuaTEX we have decided that instead of hard coding solutions, opening up makes more sense. One of the first mechanisms that were opened up was

file IO. This means that when a file is opened, you can decide to intercept lines and process them before passing them to the traditional built in input parser. The user can be completely unaware of this. In fact, as LuaTEX only accepts UTF-8, preprocessing will likely happen already when other input encodings are used. The following helper functions are available:

```
local result = resolvers.macros.preprocessed(str)
```

This function returns a string with all named parameters replaced.

```
resolvers.macros.convertfile(oldname,newname)
```

This function converts a file into a new one.

```
local result =
    resolvers.macros.processmkvi(str,filename)
```

This function converts the string but only if the suffix of the filename is `mkvi` or when the first line of the string is a comment line containing `macros=mkvi`. Otherwise the original string is returned. The filename is optional.

A few details

Imagine that you want to do this:

```
\def\test#1{before#1after}
```

When we use names this could look like:

```
\def\test#inbetween{before#inbetweenafter}
```

and that is not going to work out well. We could be more liberal with spaces, like

```
\def\test #inbetween {before #inbetween after}
```

but then getting spaces in the output before or after variables would get more complex. However, there is a way out:

```
\def\test#inbetween{before#{inbetween}after}
```

As the sequence `#{` has a rather low probablility of showing up in a TEX source file, this kind of escaping is part of the game. So, all the following cases are valid:

```
\def\test#oeps{... #oeps ...}
\def\test#oeps{... #{oeps} ...}
\def\test#{main:oeps}{... #{main:oeps} ...}
\def\test#{oeps:1}{... #{oeps:1} ...}
\def\test#{oeps}{... #oeps ...}
```

When you use the braced variant, all characters except braces are acceptable as name, optherwise only lowercase and uppercase characters are permitted.
Normally LuaTEX uses a couple of special tokens like ^ and _. In a macro definition file you can avoid these by using primitives:

```
&   \aligntab
#   \alignmark
^   \Usuperscript
_   \Usubscript
$   \Ustartmath
$   \Ustopmath
$$  \Ustartdisplaymath
$$  \Ustopdisplaymath
```

Especially the `alignmark` is worth noticing: using that one directly in a macro definition can result in unwanted replacements, depending on whether a match can be found. In practice the following works out well

```
\def\test#oeps{test:#oeps
               \halign{##\cr #oeps\cr}}
```

You can use UTF-8 characters as well. For practical reasons this is only possible with the braced variant.

```
\def\blä#{blá}{blà:#{blá}}
```

There will probably be more features in future versions but each of them needs careful consideration in order to prevent interferences.

Utilities

There is currently one utility (or in fact an option to an existing utility):

```
mtxrun --script interface
       --preprocess whatever.mkvi
```

This will convert the given file(s) to new ones, with the default suffix `tex`. Existing files will not be overwritten unless `---force` is given. You can also force another suffix:

```
mtxrun --script interface
       --preprocess whatever.mkvi
       --suffix=mkiv
```

A rather plain module `luatex-preprocessor.lua` is provided for other usage. That variant provides a somewhat simplified version.
Given that you have a `luatex-plain` format you can run:

```
luatex --fmt=luatex-plain
       luatex-preprocessor-test.tex
```

Such a plain format can be made with:

```
luatex --ini luatex-plain
```

You probably need to move the format to a proper location in your TEX tree.

Hans Hagen

```
if not modules then modules = { } end modules ['luat-mac'] = {
    version   = 1.001,
    comment   = "companion to luat-lib.mkiv",
    author    = "Hans Hagen, PRAGMA-ADE, Hasselt NL",
    copyright = "PRAGMA ADE / ConTeXt Development Team",
    license   = "see context related readme files"
}

local P, V, S, R, C, Cs, Cmt = lpeg.P, lpeg.V, lpeg.S, lpeg.R, lpeg.C, lpeg.Cs, lpeg.Cmt
local lpegmatch, patterns = lpeg.match, lpeg.patterns

local insert, remove = table.insert, table.remove
local rep, sub = string.rep, string.sub
local setmetatable = setmetatable

local report_macros = logs.new("macros")

local stack, top, n, hashes = { }, nil, 0, { }

local function set(s)
    if top then
        n = n + 1
        if n > 9 then
            report_macros("number of arguments > 9, ignoring %s",s)
        else
            local ns = #stack
            local h = hashes[ns]
            if not h then
                h = rep("#",ns)
                hashes[ns] = h
            end
            m = h .. n
            top[s] = m
            return m
        end
    end
end

local function get(s)
    local m = top and top[s] or s
    return m
end

local function push()
    top = { }
    n = 0
    local s = stack[#stack]
    if s then
        setmetatable(top,{ __index = s })
    end
    insert(stack,top)
end

local function pop()
    top = remove(stack)
end

local leftbrace   = P("{")   -- will be in patterns
local rightbrace  = P("}")
```

```
local escape      = P("\\")

local space       = patterns.space
local spaces      = space^1
local newline     = patterns.newline
local nobrace     = 1 - leftbrace - rightbrace

local longleft    = leftbrace  -- P("(")
local longright   = rightbrace -- P(")")
local nolong      = 1 - longleft - longright

local name        = R("AZ","az")^1 -- @?! -- utf?
local longname    = (longleft/"") * (nolong^1) * (longright/"")
local variable    = P("#") * Cs(name + longname)
local escapedname = escape * name
local definer     = escape * (P("def") + P("egdx") * P("def"))
local startcode   = P("\\starttexdefinition")
local stopcode    = P("\\stoptexdefinition")
local anything    = patterns.anything
local always      = patterns.alwaysmatched

local pushlocal   = always   / push
local poplocal    = always   / pop
local declaration = variable / set
local identifier  = variable / get

local function matcherror(str,pos)
    report_macros("runaway definition at: %s",sub(str,pos-30,pos))
end

local grammar = { "converter",
    texcode     = pushlocal
                * startcode
                * spaces
                * name
                * spaces
                * (declaration + (1 - newline - space))^0
                * V("texbody")
                * stopcode
                * poplocal,
    texbody     = (   V("definition")
                    + identifier
                    + V("braced")
                    + (1 - stopcode)
                  )^0,
    definition  = pushlocal
                * definer
                * escapedname
                * (declaration + (1-leftbrace))^0
                * V("braced")
                * poplocal,
    braced      = leftbrace
                * (   V("definition")
                    + identifier
                    + V("texcode")
                    + V("braced")
                    + nobrace
                  )^0
```

```
            -- * rightbrace^-1, -- the -1 catches errors
               * (rightbrace + Cmt(always,matcherror)),

    pattern     = V("definition") + V("texcode") + anything,

    converter   = V("pattern")^1,
}

local parser = Cs(grammar)

local checker = P("%") * (1 - newline - P("macros"))^0
             * P("macros") * space^0 * P("=") * space^0 * C(patterns.letter^1)

-- maybe namespace

local macros = { } resolvers.macros = macros

function macros.preprocessed(str)
    return lpegmatch(parser,str)
end

function macros.convertfile(oldname,newname) -- beware, no testing on oldname == newname
    local data = resolvers.loadtexfile(oldname)
    data = interfaces.preprocessed(data) or ""
    io.savedata(newname,data)
end

function macros.version(data)
    return lpegmatch(checker,data)
end

function macros.processmkvi(str,filename)
    if (filename and file.suffix(filename) == "mkvi") or lpegmatch(checker,str) == "mkvi" then
        return lpegmatch(parser,str) or str
    else
        return str
    end
end

if resolvers.schemes then

    local function handler(protocol,name,cachename)
        local hashed = url.hashed(name)
        local path = hashed.path
        if path and path ~= "" then
            local data = resolvers.loadtexfile(path)
            data = lpegmatch(parser,data) or ""
            io.savedata(cachename,data)
        end
        return cachename
    end

    resolvers.schemes.install('mkvi',handler,1) -- this will cache !

    utilities.sequencers.appendaction(resolvers.openers.helpers.textfileactions,
                                      "system","resolvers.macros.processmkvi")

end
```