

OBSAH

Petr Sojka: Úvodník	1
Hans Hagen: CON _T E _X T–LUA Documents	3
Hans Hagen: Exporting XML and ePub from CON _T E _X T	55
Hans Hagen, Idris Samawi Hamid: Oriental T _E X: Optimizing Paragraphs	64
Taco Hoekwater: MetaPost: PNG Output	98
Aleš Kozubík: Mapy v L ^A T _E Xových dokumentoch – predstavenie balíčka getmap	101
Vít Novotný: Konference TUG@BachoT _E X 2017	110

Zpravodaj československého sdružení uživatelů T_EXu je vydáván v tištěné podobě a distribuován zdarma členům sdružení. Po uplynutí dvanácti měsíců od tištěného vydání je poskytován v elektronické podobě (PDF) ve veřejně přístupném archívu dostupném přes <http://www.cstug.cz/>.

Své příspěvky do Zpravodaje můžete zasílat v elektronické podobě, nejlépe jako jeden archivní soubor (**.zip**, **.arj**, **.tar.gz**). Postupujte podle instrukcí, které najdete na stránce <http://bulletin.cstug.cz/>. Nezapomeňte přiložit všechny soubory, které dokument načítá (s výjimkou standardních součástí T_EX Live), zejména v případě, kdy vás nelze kontaktovat e-mailem.

ISSN 1211-6661 (tištěná verze)

ISSN 1213-8185 (online verze)

Milé čtenářky a čtenáři, příznivci kvalitní typografie sázecím systémem $\text{T}_{\text{E}}\text{X}$, je mi potěšením psát úvodník k novému číslu našeho Zpravodaje, který vychází dle plánu nové redakční rady:

if máme v březnu v bufferu nemálo článků **then**
vydáme číslo a vyprázdníme buffer

end if

if máme v červnu v bufferu nemálo článků **then**
vydáme [jedno–dvoj]číslu a vyprázdníme buffer

end if

if máme v září v bufferu nemálo článků **then**
vydáme [jedno–troj]číslu a vyprázdníme buffer

end if

V listopadu vydáme [jedno–čtyř]číslu a vyprázdníme buffer.

Zafungovala druhá podmínka, v bufferu jsme měli článků skoro na trojčíslu, a některé již čekají na vydání v dalším čísle. Díky nové redakční radě za akvizici článků a jejich rychlé sesazení v době prázdnin. Nebojte se posílat další články, sdílejte svá makra, své zkušenosti s užitím $\text{T}_{\text{E}}\text{X}$ ových technologií na vašich školách, ve vašich firmách, na vašich pracovištích.

Všichni kolektivní členové a individuální členové, kteří si o ně napsali, mají k číslu přibaleno DVD $\text{T}_{\text{E}}\text{X}$ live, opět s aktualizovanou česko-slovenskou dokumentací. Přestože mnozí členové se aktivně podílí a budou podílet na aktualizacích a přípravě hlavní $\text{T}_{\text{E}}\text{X}$ ové distribuce, výbor sdružení se rozhodl již neposílat DVD paušálně všem členům, neboť většina má dostupné vysokorychlostní připojení a naopak nemá DVD mechaniku.

V čísle najdete několik příspěvků holandských kolegů Hanse Hageny a Taca Hoekwatera, vyvíjejících relativně nový makrobalík $\text{ConT}_{\text{E}}\text{Xt}$, a experimentujících s možnostmi, které přidává imperativní jazyk Lua spolu s novými rozšířeními programu MetaPost. Jde o články:

- *ConT_EXt–Lua Documents* Hanse Hageny,
- *Exporting XML and ePub from ConT_EXt* téhož autora,
- *Oriental T_EX: Optimizing Paragraphs* autorů Hanse Hageny, Idris Samawi Hamida a
- *MetaPost: PNG Output* Taco Hoekwatera.

Čtenář tak může po přečtení své MetaPost obrázky ukládat ve formátu PNG, sázet Korán v arabštině po vzoru arabských kaligrafů, exportovat své dokumenty

do formátů čteček elektronických knih nebo v jazyce Lua své dokumenty přímo programovat.

Praktické úloze, jak vložit Google mapu do dokumentu, se věnuje článek *Mapy v L^AT_EXových dokumentoch – predstavenie balíčka `getmap`* Aleše Kozubíka.

Číslo končí zprávou z konference TUG@BachoT_EX 2017. Tu kromě autora příspěvku a autora úvodníku navštívili další tři členové našeho sdružení. Ze tří zde prezentovaných příspěvků členů ζ TUGu byla prezentace dvou podpořena finančně. Dvoukonference byla dobře navštívena, jak vidíte na společné fotce účastníků v článku Víta Novotného. Vše probíhalo v obvyklé srdečné atmosféře starých i nových přátel T_EXové rodiny. Soudržnost a stálost rodiny ukazuje i to, že skoro desítka účastníků akce navštívila před čtvrt stoletím již konferenci EuroT_EX 92 v Praze, což byl velký impuls k rozšíření a rozkvětu T_EXových aktivit v Československu.

Úvodník píše na malé plachetnici u Vancouveru, kde nás s technickým redaktorem tohoto čísla pan kapitán instruoval na co slouží písmeno T na velkých lodích a tankerech. Je to místo, kam *tug boat* – malá, ale výkonná loď – tlačí či tahá velkou loď s velkou setrvačností a tonáží tak, aby se držela či zakotvila na správném místě.

Přeji T_EXovému tankeru, aby lokální skupiny uživatelů vytvářely malé, ale výkonné lodky, tugboaty či Zpravodaje, které by správným kotvením a směřováním čtyřicetiletého mohutného plavidla přinášely stále užitek.

Summary: Introduction

Editorial discusses ζ TUG's new publishing policy and comments on this issue's articles. Go forth and participate in ζ TUG to make the bright future of T_EX & Friends a reality! *You can!*

*Masarykova univerzita, Fakulta informatiky, Botanická 68a, 602 00 Brno
sojka@fi.muni.cz*

In CONT_EXt, it is now possible to prepare documents in a mixture of T_EX, XML, METAPOST, and LUA. The article gives a short introduction into the programming language of LUA and then goes on to describe how LUA can be used for programming in CONT_EXt MkIV.

Keywords: LUA, L_UA_T_EX, CONT_EXt, MkIV

Introduction

Sometimes you hear folks complain about the T_EX input language, i.e. the backslashed commands that determine your output. Of course, when alternatives are being discussed every one has a favourite programming language. In practice coding a document in each of them triggers similar sentiments with regards to coding as T_EX itself does.

So, just for fun, I added a couple of commands to CONT_EXt MkIV that permit coding a document in LUA. In retrospect it has been surprisingly easy to implement a feature like this using metatables. Of course it's a bit slower than using T_EX as input language but sometimes the LUA interface is more readable given the problem at hand.

After a while I decided to use that interface in non-critical core CONT_EXt code and in styles (modules) and solutions for projects. Using the LUA approach is sometimes more convenient, especially if the code mostly manipulates data. For instance, if you process XML files of database output you can use the interface that is available at the T_EX end, or you can use LUA code to do the work, or you can use a combination. So, from now on, in CONT_EXt you can code your style and document source in (a mixture of) T_EX, XML, METAPOST and in LUA.

In the following pages I will introduce typesetting in LUA, but as we rely on CONT_EXt it is unavoidable that some regular CONT_EXt code shows up. The fact that you can ignore backslashes does not mean that you can do without knowledge of the underlying system. I expect that the user is somewhat familiar with this macro package. Some chapters are follow ups on articles or earlier publications.

Although much of the code is still experimental it is also rather stable. Some helpers might disappear when the main functions become more clever. This manual is definitely far from complete. If you find errors, please let me know. If you think that something is missing, you can try to convince me to add it.

A bit of Lua

The language

Small is beautiful and this is definitely true for the programming language LUA (moon in Portuguese). We had good reasons for using this language in L^AT_EX: simplicity, speed, syntax and size to mention a few. Of course personal taste also played a role and after using a couple of scripting languages extensively the switch to LUA was rather pleasant.

As the LUA reference manual is an excellent book there is no reason to discuss the language in great detail: just buy ‘Programming in LUA’ by the LUA team. Nevertheless I will give a short summary of the important concepts but consult the book if you want more details.

Data types

The most basic data type is `nil`. When we define a variable, we don’t need to give it a value:

```
local v
```

Here the variable `v` can get any value but till that happens it equals `nil`. There are simple data types like `numbers`, `booleans` and `strings`. Here are some numbers:

```
local n = 1 + 2 * 3
local x = 2.3
```

Numbers are always floats¹ and you can use the normal arithmetic operators on them as well as functions defined in the math library. Inside T_EX we have only integers, although for instance dimensions can be specified in points using floats but that’s more syntactic sugar. One reason for using integers in T_EX has been that this was the only way to guarantee portability across platforms. However, we’re 30 years along the road and in LUA the floats are implemented identical across platforms, so we don’t need to worry about compatibility.

Strings in LUA can be given between quotes or can be so called long strings forced by square brackets.

```
local s = "Whatever"
local t = s .. ' you want'
local u = t .. [[ to know]] .. [--[ about Lua!]]--]
```

The two periods indicate a concatenation. Strings are hashed, so when you say:

```
local s = "Whatever"
local t = "Whatever"
local u = t
```

¹This is true for all versions upto 5.2 but following version can have a more hybrid model.

only one instance of **Whatever** is present in memory and this fact makes LUA very efficient with respect to strings. Strings are constants and therefore when you change variable **s**, variable **t** keeps its value. When you compare strings, in fact you compare pointers, a method that is really fast. This compensates the time spent on hashing pretty well.

Booleans are normally used to keep a state or the result from an expression.

```
local b = false
local c = n > 10 and s == "whatever"
```

The other value is **true**. There is something that you need to keep in mind when you do testing on variables that are yet unset.

```
local b = false
local n
```

The following applies when **b** and **n** are defined this way:

```
b == false  true
n == false  false
n == nil    true
b == nil    false
b == n      false
```

Often a test looks like:

```
if somevar then
    ...
else
    ...
end
```

In this case we enter the else branch when **somevar** is either **nil** or **false**. It also means that by looking at the code we cannot beforehand conclude that **somevar** equals **true** or something else. If you want to really distinguish between the two cases you can be more explicit:

```
if somevar == nil then
    ...
elseif somevar == false then
    ...
else
    ...
end
or
if somevar == true then
    ...
else
```

```
...
end
```

but such an explicit test is seldom needed.

There are a few more data types: tables and functions. Tables are very important and you can recognize them by the same curly braces that make T_EX famous:

```
local t = { 1, 2, 3 }
local u = { a = 4, b = 9, c = 16 }
local v = { [1] = "a", [3] = "2", [4] = false }
local w = { 1, 2, 3, a = 4, b = 9, c = 16 }
```

The `t` is an indexed table and `u` a hashed table. Because the second slot is empty, table `v` is partially indexed (slot 1) and partially hashed (the others). There is a gray area there, for instance, what happens when you nil a slot in an indexed table? In practice you will not run into problems as you will either use a hashed table, or an indexed table (with no holes), so table `w` is not uncommon.

We mentioned that strings are in fact shared (hashed) but that an assignment of a string to a variable makes that variable behave like a constant. Contrary to that, when you assign a table, and then copy that variable, both variables can be used to change the table. Take this:

```
local t = { 1, 2, 3 }
local u = t
```

We can change the content of the table as follows:

```
t[1], t[3] = t[3], t[1]
```

Here we swap two cells. This is an example of a parallel assignment. However, the following does the same:

```
t[1], t[3] = u[3], u[1]
```

After this, both `t` and `u` still share the same table. This kind of behaviour is quite natural. Keep in mind that expressions are evaluated first, so

```
t[#t+1], t[#t+1] = 23, 45
```

makes no sense, as the values end up in the same slot. There is no gain in speed so using parallel assignments is mostly a convenience feature.

There are a few specialized data types in LUA, like `coroutines` (built in), `file` (when opened), `lpeg` (only when this library is linked in or loaded). These are called ‘userdata’ objects and in L^UA_TE_X we have more userdata objects as we will see in later chapters. Of them nodes are the most noticeable: they are the core data type of the T_EX machinery. Other libraries, like `math` and `bit32` are just collections of functions operating on numbers.

Functions look like this:

```
function sum(a,b)
  print(a, b, a + b)
```

```

end
or this:
function sum(a,b)
    return a + b
end

```

There can be many arguments of all kind of types and there can be multiple return values. A function is a real type, so you can say:

```

local f = function(s) print("the value is: " .. s) end

```

In all these examples we defined variables as `local`. This is a good practice and avoids clashes. Now watch the following:

```

local n = 1

function sum(a,b)
    n = n + 1
    return a + b
end

```

```

function report()
    print("number of summations: " .. n)
end

```

Here the variable `n` is visible after its definition and accessible for the two global functions. Actually the variable is visible to all the code following, unless of course we define a new variable with the same name. We can hide `n` as follows:

```

do
    local n = 1

    sum = function(a,b)
        n = n + 1
        return a + b
    end

    report = function()
        print("number of summations: " .. n)
    end
end

```

This example also shows another way of defining the function: by assignment.

The `do ... end` creates a so called closure. There are many places where such closures are created, for instance in function bodies or branches like `if ... then ... else`. This means that in the following snippet, variable `b` is not seen after the end:

```

if a > 10 then

```

```

    local b = a + 10
    print(b*b)
end

```

When you process a blob of LUA code in T_EX (using `\directlua` or `\latelua`) it happens in a closure with an implied `do ... end`. So, `local` defined variables are really local.

T_EX's data types

We mentioned **numbers**. At the T_EX end we have counters as well as dimensions. Both are numbers but dimensions are specified differently

```

local n = tex.count[0]
local m = tex.dimen.lineheight
local o = tex.sp("10.3pt") -- scaled point is the smallest unit

```

The unit of dimension is 'scaled point' and this is a pretty small unit: 10 points equals to 655360 such units.

Another accessible data type is tokens. They are automatically converted to strings and vice versa.

```

tex.toks[0] = "message"
print(tex.toks[0])

```

Be aware of the fact that the tokens are letters so the following will come out as text and not issue a message:

```

tex.toks[0] = "\message{just text}"
print(tex.toks[0])

```

Control structures

Loops are not much different from other languages: we have `for ... do`, `while ... do` and `repeat ... until`. We start with the simplest case:

```

for index=1,10 do
    print(index)
end

```

You can specify a step and go downward as well:

```

for index=22,2,-2 do
    print(index)
end

```

Indexed tables can be traversed this way:

```

for index=1,#list do
    print(index, list[index])
end

```

Hashed tables on the other hand are dealt with as follows:

```

for key, value in next, list do
    print(key, value)
end

```

Here `next` is a built in function. There is more to say about this mechanism but the average user will use only this variant. Slightly less efficient is the following, more readable variant:

```

for key, value in pairs(list) do
    print(key, value)
end

```

and for an indexed table:

```

for index, value in ipairs(list) do
    print(index, value)
end

```

The function call to `pairs(list)` returns `next, list` so there is an (often neglectable) extra overhead of one function call.

The other two loop variants, `while` and `repeat`, are similar.

```

i = 0
while i < 10 do
    i = i + 1
    print(i)
end

```

This can also be written as:

```

i = 0
repeat
    i = i + 1
    print(i)
until i == 10

```

or:

```

i = 0
while true do
    i = i + 1
    print(i)
    if i == 10 then
        break
    end
end

```

Of course you can use more complex expressions in such constructs.

Conditions

Conditions have the following form:

```
if a == b or c > d or e then
```

```
    ...
```

```
elseif f == g then
```

```
    ...
```

```
else
```

```
    ...
```

```
end
```

Watch the double `==`. The complement of this is `~=`. Precedence is similar to other languages. In practice, as strings are hashed, tests like

```
if key == "first" then
```

```
    ...
```

```
end
```

```
and
```

```
if n == 1 then
```

```
    ...
```

```
end
```

are equally efficient. There is really no need to use numbers to identify states instead of more verbose strings.

Namespaces

Functionality can be grouped in libraries. There are a few default libraries, like `string`, `table`, `lpeg`, `math`, `io` and `os` and `LUATEX` adds some more, like `node`, `tex` and `texio`.

A library is in fact nothing more than a bunch of functionality organized using a table, where the table provides a namespace as well as place to store public variables. Of course there can be local (hidden) variables used in defining functions.

```
do
```

```
    mylib = { }
```

```
    local n = 1
```

```
    function mylib.sum(a,b)
```

```
        n = n + 1
```

```
        return a + b
```

```
    end
```

```
    function mylib.report()
```

```

    print("number of summations: " .. n)
end
end

```

The defined function can be called like:

```
mylib.report()
```

You can also create a shortcut, This speeds up the process because there are less lookups then. In the following code multiple calls take place:

```
local sum = mylib.sum
```

```

for i=1,10 do
  for j=1,10 do
    print(i, j, sum(i,j))
  end
end
end

```

```
mylib.report()
```

As LUA is pretty fast you should not overestimate the speedup, especially not when a function is called seldom. There is an important side effect here: in the case of:

```
print(i, j, sum(i,j))
```

the meaning of `sum` is frozen. But in the case of

```
print(i, j, mylib.sum(i,j))
```

the current meaning is taken, that is: each time the interpreter will access `mylib` and get the current meaning of `sum`. And there can be a good reason for this, for instance when the meaning is adapted to different situations.

In `CONTEXT` we have quite some code organized this way. Although much is exposed (if only because it is used all over the place) you should be careful in using functions (and data) that are still experimental. There are a couple of general libraries and some extend the core LUA libraries. You might want to take a look at the files in the distribution that start with `l-`, like `l-table.lua`. These files are preloaded.² For instance, if you want to inspect a table, you can say:

```

local t = { "aap", "noot", "mies" }
table.print(t)

```

You can get an overview of what is implemented by running the following command:

```
context s-tra-02 --mode=tablet
```

²In fact, if you write scripts that need their functionality, you can use `mtxrun` to process the script, as `mtxrun` has the core libraries preloaded as well.

Comment

You can add comments to your LUA code. There are basically two methods: one liners and multi line comments.

```
local option = "test" -- use this option with care
```

```
local method = "unknown" --[[comments can be very long and when
                             entered this way they can span multiple lines]]
```

The so called long comments look like long strings preceded by `--` and there can be more complex boundary sequences.

Pitfalls

Sometimes `nil` can bite you, especially in tables, as they have a dual nature: indexed as well as hashed.

```
\startluacode
```

```
local n1 = # { nil, 1, 2, nil }      -- 3
```

```
local n2 = # { nil, nil, 1, 2, nil } -- 0
```

```
context("n1 = %s and n2 = %s",n1,n2)
```

```
\stopluacode
```

results in: `n1 = 3` and `n2 = 0`. So, you cannot really depend on the length operator here. On the other hand, with:

```
\startluacode
```

```
local function check(...)
    return select("#",...)
```

```
end
```

```
local n1 = check ( nil, 1, 2, nil )      -- 4
```

```
local n2 = check ( nil, nil, 1, 2, nil ) -- 5
```

```
context("n1 = %s and n2 = %s",n1,n2)
```

```
\stopluacode
```

we get: `n1 = 4` and `n2 = 5`, so the `select` is quite useable. However, that function also has its specialities. The following example needs some close reading:

```
\startluacode
```

```
local function filter(n,...)
    return select(n,...)
```

```
end
```

```
local v1 = { filter ( 1, 1, 2, 3 ) }
```

```
local v2 = { filter ( 2, 1, 2, 3 ) }
```

```
local v3 = { filter ( 3, 1, 2, 3 ) }
```

```
context("v1 = %+t and v2 = %+t and v3 = %+t",v1,v2,v3)
\stopluacode
```

We collect the result in a table and show the concatenation: $v1 = 1+2+3$ and $v2 = 2+3$ and $v3 = 3$. So, what you effectively get is the whole list starting with the given offset.

```
\startluacode
local function filter(n,...)
    return (select(n,...))
end
```

```
local v1 = { filter ( 1, 1, 2, 3 ) }
local v2 = { filter ( 2, 1, 2, 3 ) }
local v3 = { filter ( 3, 1, 2, 3 ) }
```

```
context("v1 = %+t and v2 = %+t and v3 = %+t",v1,v2,v3)
\stopluacode
```

Now we get: $v1 = 1$ and $v2 = 2$ and $v3 = 3$. The extra $()$ around the result makes sure that we only get one return value.

Of course the same effect can be achieved as follows:

```
local function filter(n,...)
    return select(n,...)
end
```

```
local v1 = filter ( 1, 1, 2, 3 )
local v2 = filter ( 2, 1, 2, 3 )
local v3 = filter ( 3, 1, 2, 3 )
```

```
context("v1 = %s and v2 = %s and v3 = %s",v1,v2,v3)
```

A few suggestions

You can wrap all kind of functionality in functions but sometimes it makes no sense to add the overhead of a call as the same can be done with hardly any code.

If you want a slice of a table, you can copy the range needed to a new table. A simple version with no bounds checking is:

```
local new = { } for i=a,b do new[#new+1] = old[i] end
```

Another, much faster, variant is the following.

```
local new = { unpack(old,a,b) }
```

You can use this variant for slices that are not extremely large. The function `table.sub` is an equivalent:

```
local new = table.sub(old,a,b)
```

An indexed table is empty when its size equals zero:

```
if #indexed == 0 then ... else ... end
```

Sometimes this is better:

```
if indexed and #indexed == 0 then ... else ... end
```

So how do we test if a hashed table is empty? We can use the `next` function as in:

```
if hashed and next(indexed) then ... else ... end
```

Say that we have the following table:

```
local t = { a=1, b=2, c=3 }
```

The call `next(t)` returns the first key and value:

```
local k, v = next(t) -- "a", 1
```

The second argument to `next` can be a key in which case the following key and value in the hash table is returned. The result is not predictable as a hash is unordered. The generic for loop uses this to loop over a hashed table:

```
for k, v in next, t do
```

```
    ...
```

```
end
```

Anyway, when `next(t)` returns zero you can be sure that the table is empty.

This is how you can test for exactly one entry:

```
if t and not next(t,next(t)) then ... else ... end
```

Here it starts making sense to wrap it into a function.

```
function table.has_one_entry(t)
```

```
    t and not next(t,next(t))
```

```
end
```

On the other hand, this is not that usefull, unless you can spend the runtime on it:

```
function table.is_empty(t)
```

```
    return not t or not next(t)
```

```
end
```

Interfacing

We have already seen that you can embed LUA code using commands like:

```
\startluacode
```

```
    print("this works")
```

```
\stopluacode
```

This command should not be confused with:

```
\startlua
  print("this works")
\stoplua
```

The first variant has its own catcode regime which means that tokens between the start and stop command are treated as LUA tokens, with the exception of T_EX commands. The second variant operates under the regular T_EX catcode regime.

Their short variants are `\ctxluacode` and `\ctxlua` as in:

```
\ctxluacode{print("this works")}
\ctxlua{print("this works")}
```

In practice you will probably use `\startluacode` when using or defining a blob of LUA and `\ctxlua` for inline code. Keep in mind that the longer versions need more initialization and have more overhead.

There are some more commands. For instance `\ctxcommand` can be used as an efficient way to access functions in the `commands` namespace. The following two calls are equivalent:

```
\ctxlua      {commands.thisorthat("...")}
\ctxcommand  {thisorthat("...")}
```

There are a few shortcuts to the `context` namespace. Their use can best be seen from their meaning:

```
\cldprocessfile#1{\directlua{context.runfile("#1")}}
\cldloadfile    #1{\directlua{context.loadfile("#1")}}
\cldcontext     #1{\directlua{context(#1)}}
\cldcommand     #1{\directlua{context.#1}}
```

Each time a call out to LUA happens the argument eventually gets parsed, converted into tokens, then back into a string, compiled to bytecode and executed. The next example code shows a mechanism that avoids this:

```
\startctxfunction MyFunctionA
  context(" A1 ")
\stopctxfunction

\startctxfunctiondefinition MyFunctionB
  context(" B2 ")
\stopctxfunctiondefinition
```

The first command associates a name with some LUA code and that code can be executed using:

```
\ctxfunction{MyFunctionA}
```

The second definition creates a command, so there we do:

```
\MyFunctionB
```

There are some more helpers but for use in document sources they make less sense. You can always browse the source code for examples.

Getting started

Some basics

I assume that you have either the so called `CONTEXT` standalone (formerly known as `minimals`) installed or `TEXLIVE`. You only need `LUATEX` and can forget about installing `PDFTEX` or `XETEX`, which saves you some megabytes and hassle. Now, from the users perspective a `CONTEXT` run goes like:

```
context yourfile
```

and by default a file with suffix `tex`, `mkiv`, or `mkvi` will be processed. There are however a few other options:

```
context yourfile.xml
```

```
context yourfile.rlx --forcexml
```

```
context yourfile.lua
```

```
context yourfile.pqr --forcelua
```

```
context yourfile.cld
```

```
context yourfile.xyz --forcecld
```

```
context yourfile.mp
```

```
context yourfile.xyz --forcemp
```

When processing a `LUA` file the given file is loaded and just processed. This options will seldom be used as it is way more efficient to let `mtxrun` process that file. However, the last two variants are what we will discuss here. The suffix `cld` is a shortcut for `CONTEXT` `LUA` Document.

A simple `cld` file looks like this:

```
context.starttext()
```

```
context.chapter("Hello There!")
```

```
context.stoptext()
```

So yes, you need to know the `CONTEXT` commands in order to use this mechanism. In spite of what you might expect, the codebase involved in this interface is not that large. If you know `CONTEXT`, and if you know how to call commands, you basically can use this `LUA` method.

The examples that I will give are either (sort of) standalone, i.e. they are dealt with from `LUA`, or they are run within this document. Therefore you will see two patterns. If you want to make your own documentation, then you can use this variant:

```
\startbuffer
```

```
context("See this!")
```

```
\stopbuffer
```

```
\typebuffer \ctxluabuffer
```

I use anonymous buffers here but you can also use named ones. The other variant is:

```
\startluacode
context("See this!")
\stopluacode
```

This will process the code directly. Of course we could have encoded this document completely in LUA but that is not much fun for a manual.

The main command

There are a few rules that you need to be aware of. First of all no syntax checking is done. Second you need to know what the given commands expects in terms of arguments. Third, the type of your arguments matters:

nothing : just the command, no arguments
string : an argument with curly braces
array : a list between square brackets (sometimes optional)
hash : an assignment list between square brackets
boolean : when **true** a newline is inserted
 : when **false**, omit braces for the next argument

In the code above you have seen examples of this but here are some more:

```
context.chapter("Some title")
context.chapter({ "first" }, "Some title")
context.startchapter({ title = "Some title", label = "first" })
```

This blob of code is equivalent to:

```
\chapter{Some title}
\chapter[first]{Some title}
\startsection[title={Some title},label=first]
```

You can simplify the third line of the LUA code to:

```
context.startchapter { title = "Some title", label = "first" }
```

In case you wonder what the distinction is between square brackets and curly braces: the first category of arguments concerns settings or lists of options or names of instances while the second category normally concerns some text to be typeset.

Strings are interpreted as T_EX input, so:

```
context.mathematics("\sqrt{2^3}")
and if you don't want to escape:
context.mathematics([[ \sqrt{2^3} ]])
```

are both correct. As \TeX math is a language in its own and a de-facto standard way of inputting math this is quite natural, even at the \LaTeX end.

Spaces and Lines

In a \TeX file, spaces and newline characters are collapsed into one space. The same happens in \LaTeX . Compare the following examples. First we omit spaces:

```
context("left")
context("middle")
context("right")
```

resulting in: leftmiddleright. Next we add spaces:

```
context("left")
context(" middle ")
context("right")
```

resulting in: left middle right. We can also add more spaces:

```
context("left  ")
context("  middle ")
context("  right")
```

resulting in: left middle right. In principle all content becomes a stream and after that the \TeX parser will do its normal work: collapse spaces unless configured to do otherwise.

Now take the following code:

```
context("before")
context("word 1")
context("word 2")
context("word 3")
context("after")
```

resulting in: beforeword 1word 2word 3after. Here we get no spaces between the words at all, which is what we expect. So, how do we get lines (or paragraphs)?

```
context("before")
context.startlines()
context("line 1")
context("line 2")
context("line 3")
context.stoplines()
context("after")
```

results in: before

line 1line 2line 3

after.

This does not work out well, as again there are no lines seen at the `TEX` end. Newline tokens are injected by passing `true` to the `context` command:

```
context("before")
context.startlines()
context("line 1") context(true)
context("line 2") context(true)
context("line 3") context(true)
context.stoplines()
context("after")
resulting in: before
```

```
line 1
line 2
line 3
```

after. Don't confuse this with:

```
context("before") context.par()
context("line 1") context.par()
context("line 2") context.par()
context("line 3") context.par()
context("after") context.par()
which results in: before
```

```
line 1
line 2
line 3
```

after. There we use the regular `\par` command to finish the current paragraph and normally you will use that method. In that case, when set, whitespace will be added between paragraphs.

This newline issue is a somewhat unfortunate inheritance of traditional `TEX`, where `\n` and `\r` mean something different. I'm still not sure if the CLD do the right thing as dealing with these tokens also depends on the intended effect. Catcodes as well as the `LUATEX` input parser also play a role. Anyway, the following also works:

```
context.startlines()
context("line 1\n")
context("line 2\n")
context("line 3\n")
context.stoplines()
```

Direct output

The `CONTEXT` user interface is rather consistent and the use of special input syntaxes is discouraged. Therefore, the `LUA` interface using tables and strings

works quite well. However, imagine that you need to support some weird macro (or a primitive) that does not expect its argument between curly braces or brackets. The way out is to precede an argument by another one with the value `false`. We call this the direct interface. This is demonstrated in the following example.

```
\unexpanded\def\bla#1{[#1]}
\startluacode
context.bla(false, "***")
context.par()
context.bla("***")
\stopluacode
```

This results in: `[*]**`

`[***]`. Here, the first call results in three `*` being passed, and `#1` picks up the first token. The second call to `bla` gets `{***}` passed so here `#1` gets the triplet. In practice you will seldom need the direct interface.

In `CONTEXT` for historical reasons, combinations accept the following syntax:

```
\startcombination % optional specification, like [2*3]
  {\framed{content one}} {caption one}
  {\framed{content two}} {caption two}
\stopcombination
```

You can also say:

```
\startcombination
  \combination {\framed{content one}} {caption one}
  \combination {\framed{content two}} {caption two}
\stopcombination
```

When coded in `LUA`, we can feed the first variant as follows:

```
context.startcombination()
  context.direct("one", "two")
  context.direct("one", "two")
context.stopcombination()
```

To give you an idea what this looks like, we render it:

```
one  one
two  two
```

So, the `direct` function is basically a no-op and results in nothing by itself. Only arguments are passed. An equivalent but bit more ugly looking is:

```
context.startcombination()
  context(false, "one", "two")
  context(false, "one", "two")
context.stopcombination()
```

Catcodes

If you are familiar with the inner working of \TeX , you will know that characters can have special meanings. This meaning is determined by their catcodes.

```
context("$x=1$")
```

This gives: $x = 1$ because the dollar tokens trigger inline math mode. If you think that this is annoying, you can do the following:

```
context.pushcatcodes("text")
```

```
context("$x=1$")
```

```
context.popcatcodes()
```

Now we get: $\$x=1\$$. There are several catcode regimes of which only a few make sense in the perspective of the `cld` interface.

<code>ctx</code> , <code>ctxcatcodes</code> , <code>context</code>	the normal <code>CONTEXT</code> catcode regime
<code>prr</code> , <code>prrcatcodes</code> , <code>protect</code>	the <code>CONTEXT</code> protected regime, used for modules
<code>tex</code> , <code>texcatcodes</code> , <code>plain</code>	the traditional (plain) \TeX regime
<code>txt</code> , <code>txtcatcodes</code> , <code>text</code>	the <code>CONTEXT</code> regime but with less special characters
<code>vrbl</code> , <code>vrblcatcodes</code> , <code>verbatim</code>	a regime specially meant for <code>verbatim</code>
<code>xml</code> , <code>xmlcatcodes</code>	a regime specially meant for XML processing

In the second case you can still get math:

```
context.pushcatcodes("text")
```

```
context.mathematics("x=1")
```

```
context.popcatcodes()
```

When entering a lot of math you can also consider this:

```
context.startimath()
```

```
context("x")
```

```
context("=")
```

```
context("1")
```

```
context.stopimath()
```

Module writers can use `unprotect` and `protect` as they do at the \TeX end.

As we've seen, a function call to `context` acts like a print, as in:

```
context("test ")
```

```
context.bold("me")
```

```
context(" first")
```

resulting in: test **me** first. When more than one argument is given, the first argument is considered a format conforming the `string.format` function:

```
context.startimath()
```

```
context("%s = %0.5f",utf.char(0x03C0),math.pi)
```

```
context.stopimath()
```

resulting in: $\pi = 3.14159$. This means that when you say:

```
context(a,b,c,d,e,f)
```

the variables **b** till **f** are passed to the format and when the format does not use them, they will not end up in your output.

```
context("%s %s %s",1,2,3)
```

```
context(1,2,3)
```

The first line results in the three numbers being typeset, but in the second case only the number 1 is typeset.

More on functions

Why we need them

In a previous section we introduced functions as arguments. At first sight this feature looks strange but you need to keep in mind that a call to a **context** function has no direct consequences. It generates **TeX** code that is executed after the current **LUA** chunk ends and control is passed back to **TeX**. Take the following code:

```
context.framed( {  
    frame = "on",  
    offset = "5mm",  
    align = "middle"  
},  
    context.input("knuth")  
)
```

We call the function **framed** but before the function body is executed, the arguments get evaluated. This means that **input** gets processed before **framed** gets done. As a result there is no second argument to **framed** and no content gets passed: an error is reported. This is why we need the indirect call:

```
context.framed( {  
    frame = "on",  
    align = "middle"  
},  
    function() context.input("knuth") end  
)
```

This way we get what we want:

Thus, I came to the conclusion that the designer of a new system must not only be the implementer and first large-scale user; the designer should also write the first user manual.

The separation of any of these four components would have hurt T_EX significantly. If I had not participated fully in all these activities, literally hundreds of improvements would never have been made, because I would never have thought of them or perceived why they were important.

But a system cannot be successful if it is too strongly influenced by a single person. Once the initial design is complete and fairly robust, the real test begins as people with many different viewpoints undertake their own experiments.

The function is delayed till the `framed` command is executed. If your applications use such calls a lot, you can of course encapsulate this ugliness:

```
mycommands = mycommands or { }
```

```
function mycommands.framed_input(filename)
  context.framed( {
    frame = "on",
    align = "middle"
  },
  function() context.input(filename) end
end
```

```
mycommands.framed_input("knuth")
```

Of course you can nest function calls:

```
context.placefigure(
  "caption",
  function()
    context.framed( {
      frame = "on",
      align = "middle"
    },
    function() context.input("knuth") end
  )
end
)
```

Or you can use a more indirect method:

```
function text()
  context.framed( {
    frame = "on",
    align = "middle"
  }
```

```

    },
    function() context.input("knuth") end
  )
end

context.placefigure(
  "none",
  function() text() end
)

```

You can develop your own style and libraries just like you do with regular LUA code. Browsing the already written code can give you some ideas.

How we can avoid them

As many nested functions can obscure the code rather quickly, there is an alternative. In the following examples we use `test`:

```

\def\test#1{[#1]}
context.test("test 1 ",context("test 2a")," test 3")

```

This gives: test 2a[test 1] test 3. As you can see, the second argument is executed before the encapsulating call to `test`. So, we should have packed it into a function but here is an alternative:

```

context.test("test 1 ",context.delayed("test 2a")," test 3")

```

Now we get: [test 1]test 2a test 3. We can also delay functions themselves, look at this:

```

context.test("test 1 ",context.delayed.test("test 2b")," test 3")

```

The result is: [test 1][test 2b] test 3. This feature also conveniently permits the use of temporary variables, as in:

```

local f = context.delayed.test("test 2c")
context("before ",f," after")

```

Of course you can limit the amount of keystrokes even more by creating a shortcut:

```

local delayed = context.delayed

```

```

context.test("test 1 ",delayed.test("test 2")," test 3")
context.test("test 4 ",delayed.test("test 5")," test 6")

```

So, if you want you can produce rather readable code and readability of code is one of the reasons why LUA was chosen in the first place. This is a good example of why coding in \TeX makes sense as it looks more intuitive:

```

\test{test 1 \test{test 2} test 3}
\test{test 4 \test{test 5} test 6}

```

There is also another mechanism available. In the next example the second argument is actually a string.

```
local nested = context.nested
```

```
context.test("test 8",nested.test("test 9"),"test 10")
```

There is a pitfall here: a nested context command needs to be flushed explicitly, so in the case of:

```
context.nested.test("test 9")
```

a string is created but nothing ends up at the `TEX` end. Flushing is up to you. Beware: `nested` only works with the regular `CONTEXT` catcode regime.

Trial typesetting

Some typesetting mechanisms demand a preroll. For instance, when determining the most optimal way to analyse and therefore typeset a table, it is necessary to typeset the content of cells first. Inside `CONTEXT` there is a state tagged ‘trial typesetting’ which signals other mechanisms that for instance counters should not be incremented more than once.

Normally you don’t need to worry about these issues, but when writing the code that implements the `LUA` interface to `CONTEXT`, it definitely had to be taken into account as we either or not can free cached (nested) functions.

You can influence this caching to some extend. If you say

```
function()
  context("whatever")
end
```

the function will be removed from the cache when `CONTEXT` is not in the trial typesetting state. You can prevent removal of a function by returning `true`, as in:

```
function()
  context("whatever")
  return true
end
```

Whenever you run into a situation that you don’t get the outcome that you expect, you can consider returning `true`. However, keep in mind that it will take more memory, something that only matters on big runs. You can force flushing the whole cache by:

```
context.restart()
```

An example of an occasion where you need to keep the function available is in repeated content, for instance in headers and footers.

```
context.setupheadertexts {
  function()
```

```

    context.pagenumber()
    return true
end
}

```

Of course it is not needed when you use the following method:

```
context.pagenumber("pagenumber")
```

Because here CONTEX_T itself deals with the content driven by the keyword `pagenumber`.

Steppers

The `context` commands are accumulated within a `\ctxlua` call and only after the call is finished, control is back at the T_EX end. Sometimes you want (in your L_UA code) to go on and pretend that you jump out to T_EX for a moment, but come back to where you left. The stepper mechanism permits this.

A not so practical but nevertheless illustrative example is the following:

```

\startluacode
context.stepwise (function()
  context.startitemize()
    context.startitem()
      context.step("BEFORE 1")
    context.stopitem()
    context.step("\setbox0\hbox{!!!!}")
    context.startitem()
      context.step("%p",tex.getbox(0).width)
    context.stopitem()
    context.startitem()
      context.step("BEFORE 2")
    context.stopitem()
    context.step("\setbox2\hbox{????}")
    context.startitem()
      context.step("%p",tex.getbox(2).width)
    context.startitem()
      context.step("BEFORE 3")
    context.stopitem()
    context.startitem()
      context.step("\copy0\copy2")
    context.stopitem()
    context.startitem()
      context.step("BEFORE 4")
    context.startitemize()
      context.stepwise (function()

```

```

context.step("\\bgroup")
context.step("\\setbox0\\hbox{>>>}")
context.startitem()
    context.step("%p",tex.getbox(0).width)
context.stopitem()
context.step("\\setbox2\\hbox{<<<}")
context.startitem()
    context.step("%p",tex.getbox(2).width)
context.stopitem()
context.startitem()
    context.step("\\copy0\\copy2")
context.stopitem()
context.startitem()
    context.step("\\copy0\\copy2")
context.stopitem()
context.step("\\egroup")
end)
context.stopitemize()
context.stopitem()
context.startitem()
    context.step("AFTER 1\\par")
context.stopitem()
context.startitem()
    context.step("\\copy0\\copy2\\par")
context.stopitem()
context.startitem()
    context.step("\\copy0\\copy2\\par")
context.stopitem()
context.startitem()
    context.step("AFTER 2\\par")
context.stopitem()
context.startitem()
    context.step("\\copy0\\copy2\\par")
context.stopitem()
context.startitem()
    context.step("\\copy0\\copy2\\par")
context.stopitem()
context.stopitemize()
end)
\\stopluacode

```

This gives an (ugly) itemize with a nested one:

- BEFORE 1
- 12.76001pt
- BEFORE 2
- 18.88pt
- BEFORE 3
- !!!!????
- BEFORE 4
 - 31.12pt
 - 31.12pt
 - >>>><<<<
 - >>>><<<<
- AFTER 1
- !!!!????
- !!!!????
- AFTER 2
- !!!!????
- !!!!????

As you can see in the code, the `step` call accepts multiple arguments, but when more than one argument is given the first one is treated as a formatter.

A few Details

Variables

Normally it makes most sense to use the English version of `CONTEXT`. The advantage is that you can use English keywords, as in:

```
context.framed( {  
    frame = "on",
```

```

    },
    "some text"
)

```

If you use the Dutch interface it looks like this:

```

context.omlijnd( {
    kader = "aan",
    },
    "wat tekst"
)

```

A rather neutral way is:

```

context.framed( {
    frame = interfaces.variables.on,
    },
    "some text"
)

```

But as said, normally you will use the English user interface so you can forget about these matters. However, in the `CONTEXT` core code you will often see the variables being used this way because there we need to support all user interfaces.

Modes

`CONTEXT` carries a concept of modes. You can use modes to create conditional sections in your style (and/or content). You can control modes in your styles or you can set them at the command line or in job control files. When a mode test has to be done at processing time, then you need constructs like the following:

```

context.doifmodeelse( "screen",
    function()
        ... -- mode == screen
    end,
    function()
        ... -- mode ~= screen
    end
)

```

However, often a mode does not change during a run, and then we can use the following method:

```

if tex.modes["screen"] then
    ...
else
    ...
end

```

Watch how the `modes` table lives in the `tex` namespace. We also have `systemmodes`. At the `TEX` end these are mode names preceded by a `*`, so the following code is similar:

```
if tex.modes["*mymode"] then
  -- this is the same
elseif tex.systemmodes["mymode"] then
  -- test as this
else
  -- but not this
end
```

Inside `CONTEXT` we also have so called constants, and again these can be consulted at the `LUA` end:

```
if tex.constants["someconstant"] then
  ...
else
  ...
end
```

But you will hardly need these and, as they are often not public, their meaning can change, unless of course they *are* documented as public.

Token lists

There is normally no need to mess around with nodes and tokens at the `LUA` end yourself. However, if you do, then you might want to flush them as well. Say that at the `TEX` end we have said:

```
\toks0 = {Don't get \inframed{framed}!}
```

Then at the `LUA` end you can say:

```
context(tex.toks[0])
```

and get: Don't get `framed`! In fact, token registers are exposed as strings so here, register zero has type `string` and is treated as such.

```
context("< %s >", tex.toks[0])
```

This gives: `< Don't get framed! >`. But beware, if you go the reverse way, you don't get what you might expect:

```
tex.toks[0] = {[\framed{oeps}]}
```

If we now say `\the\toks0` we will get `\framed{oeps}` as all tokens are considered to be letters.

Node lists

If you're not deep into `TEX` you will never feel the need to manipulate node lists yourself, but you might want to flush boxes. As an example we put something in box zero (one of the scratch boxes).

```
\setbox0 = \hbox{Don't get \inframed{framed}!}
```

At the T_EX end you can flush this box (`\box0`) or take a copy (`\copy0`). At the L_UA end you would do:

```
context.copy()
context.direct(0)
```

or:

```
context.copy(false,0)
```

but this works as well:

```
context(node.copy_list(tex.box[0]))
```

So we get: Don't get framed! If you do:

```
context(tex.box[0])
```

you also need to make sure that the box is freed but let's not go into those details now.

Here is an example of messing around with node lists that get seen before a paragraph gets broken into lines, i.e. when hyphenation, font manipulation etc. take place. First we define some colors:

```
\definecolor[mynesting:0][r=.6]
\definecolor[mynesting:1][g=.6]
\definecolor[mynesting:2][r=.6,g=.6]
```

Next we define a function that colors nodes in such a way that we can see the different processing stages.

```
\startluacode
local enabled = false
local count   = 0
local setcolor = nodes.tracers.colors.set

function userdata.processmystuff(head)
  if enabled then
    local color = "mynesting:" .. (count % 3)
    -- for n in node.traverse(head) do
    for n in node.traverse_id(nodes.nodecodes.glyph,head) do
      setcolor(n,color)
    end
    count = count + 1
    return head, true
  end
  return head, false
end
```

```

function userdata.enablemystuff()
    enabled = true
end

function userdata.disablemystuff()
    enabled = false
end
\stopluacode

```

We hook this function into the normalizers category of the processor callbacks:

```

\startluacode
nodes.tasks.appendaction(
    "processors",
    "normalizers",
    "userdata.processmystuff"
)
\stopluacode

```

We now can enable this mechanism and show an example:

```

\startbuffer
Node lists are processed \hbox {nested from \hbox{inside} out} which
is not what you might expect. But, \hbox{coloring} does not \hbox
{happen} really nested here, more \hbox {in} \hbox {the} \hbox {order}
\hbox {of} \hbox {processing}.
\stopbuffer

```

```

\ctxlua{userdata.enablemystuff()}
\par \getbuffer \par
\ctxlua{userdata.disablemystuff()}

```

The `\par` is needed because otherwise the processing is already disabled before the paragraph gets seen by T_EX. This is the result:

Node lists are processed nested from inside out which is not what you might expect. But, coloring does not happen really nested here, more in the order of processing.

Instead of using a boolean to control the state, we can also do this:

```

\startluacode
local count = 0
local setcolor = nodes.tracers.colors.set

function userdata.processmystuff(head)
    count = count + 1
    local color = "mynesting:" .. (count % 3)

```

```

    for n in node.traverse_id(nodes.nodecodes.glyph,head) do
        setcolor(n,color)
    end
    return head, true
end

nodes.tasks.appendaction(
    "processors",
    "after",
    "userdata.processmystuff"
)
\stopluacode
Disabling now happens with:
\startluacode
nodes.tasks.disableaction("processors", "userdata.processmystuff")
\stopluacode

```

As you might want to control these things in more details, a simple helper mechanism was made: markers. The following example code shows the way:

```

\definemarker[mymarker]
Again we define some colors:
\definecolor[mymarker:1][r=.6]
\definecolor[mymarker:2][g=.6]
\definecolor[mymarker:3][r=.6,g=.6]
The LUA code looks similar to the code presented before:
\startluacode
local setcolor      = nodes.tracers.colors.setlist
local getmarker     = nodes.markers.get
local hlist_code    = nodes.codes.hlist
local traverse_id   = node.traverse_id

function userdata.processmystuff(head)
    for n in traverse_id(hlist_code,head) do
        local m = getmarker(n,"mymarker")
        if m then
            setcolor(n.list,"mymarker:" .. m)
        end
    end
    return head, true
end
end

```

```

nodes.tasks.appendaction(
  "processors",
  "after",
  "userdata.processmystuff")
nodes.tasks.disableaction("processors", "userdata.processmystuff")
\stopluacode

```

This time we disabled the processor (if only because in this document we don't want the overhead).

```

\startluacode
nodes.tasks.enableaction("processors", "userdata.processmystuff")
\stopluacode

```

Node lists are processed `\hbox \boxmarker{mymarker}{1}` {nested from `\hbox{inside}` out} which is not what you might expect. But, `\hbox {coloring}` does not `\hbox {happen}` really nested here, more `\hbox {in} \hbox \boxmarker{mymarker}{2}` {the} `\hbox {order} \hbox {of} \hbox \boxmarker{mymarker}{3}` {processing}.

```

\startluacode
nodes.tasks.disableaction("processors", "userdata.processmystuff")
\stopluacode

```

The result looks familiar:

Node lists are processed **nested from** inside **out** which is not what you might expect. But, coloring does not happen really nested here, more in **the** order of **processing**.

Some more examples

Appetizer

Before we give some more examples, we will have a look at the way the title page is made. This way you get an idea what more is coming.

```

local todimen, random = number.todimen, math.random

```

```

context.startTEXpage()

```

```

local paperwidth  = tex.dimen.paperwidth
local paperheight = tex.dimen.paperheight
local nofsteps    = 25
local firstcolor  = "darkblue"

```

```

local secondcolor = "white"

context.definelayer({ "titlepage" })

context.setuplayer(
  { "titlepage" },
  { width = todimen(paperwidth),
    height = todimen(paperheight),
  }
)
context.setlayerframed(
  { "titlepage" },
  { offset = "-5pt" },
  { width = todimen(paperwidth),
    height = todimen(paperheight),
    background = "color",
    backgroundcolor = firstcolor,
    backgroundoffset = "10pt",
    frame = "off",
  },
  ""
)
local settings = {
  frame = "off",
  background = "color",
  backgroundcolor = secondcolor,
  foregroundcolor = firstcolor,
  foregroundstyle = "type",
}
for i=1, nofsteps do
  for j=1, nofsteps do
    context.setlayerframed(
      { "titlepage" },
      { x = todimen((i-1) * paperwidth / nofsteps),
        y = todimen((j-1) * paperheight / nofsteps),
        rotation = random(360),
      },
      settings,
      "CLD"
    )
  end
end
end

```

```

context.tightlayer({ "titlepage" })
context.stopTEXpage()
return true

```

This does not look that bad, does it? Of course in pure $\text{T}_{\text{E}}\text{X}$ code it looks mostly the same but loops and calculations feel a bit more natural in LUA then in $\text{T}_{\text{E}}\text{X}$. The result is shown in figure 1. The actual cover page of the manual was derived from this.

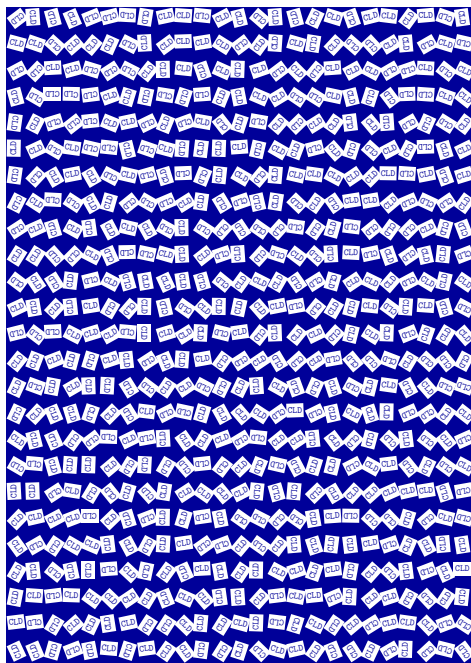


Figure 1 The simplified cover page.

A few examples

As it makes most sense to use the LUA interface for generated text, here is another example with a loop:

```

context.startitemize { "a", "packed", "two" }
  for i=1,10 do
    context.startitem()
      context("this is item %i",i)
    context.stopitem()
  end
context.stopitemize()

```

resulting in:

- a. this is item 1
- b. this is item 2
- c. this is item 3
- d. this is item 4
- e. this is item 5
- f. this is item 6
- g. this is item 7
- h. this is item 8
- i. this is item 9
- j. this is item 10

Just as you can mix \TeX with XML and METAPost, you can define bits and pieces of a document in LUA. Tables are good candidates:

```
local one = {
    align = "middle",
    style = "type",
}
local two = {
    align = "middle",
    style = "type",
    background = "color",
    backgroundcolor = "darkblue",
    foregroundcolor = "white",
}
local random = math.random
context.bTABLE { framecolor = "darkblue" }
    for i=1,10 do
        context.bTR()
        for i=1,20 do
            local r = random(99)
            context.bTD(r < 50 and one or two)
            context("%2i",r)
            context.eTD()
        end
        context.eTR()
    end
context.eTABLE()
```

Here we see a function call to `context` in the most indented line. The first argument is a format that makes sure that we get two digits and the random number is substituted into this format. The result is shown in table 1. The

28	95	39	94	37	52	38	66	98	46	30	90	39	77	62	41	49	77	80	68
22	11	17	97	67	60	7	88	6	71	48	33	67	87	28	4	40	65	70	38
12	99	29	51	77	90	91	27	67	72	94	88	82	12	86	49	72	93	38	77
65	85	11	32	73	38	36	13	4	6	50	15	6	79	66	82	69	57	9	37
30	4	26	12	16	13	61	87	6	98	65	70	83	76	2	57	14	38	69	18
43	20	33	48	98	98	31	68	56	40	5	85	43	31	97	59	43	58	46	49
57	12	19	41	87	21	97	2	58	66	19	2	85	52	50	83	50	80	51	6
21	56	91	64	87	88	23	31	47	69	79	4	80	98	44	68	19	41	70	77
8	89	78	92	41	29	76	90	9	28	96	30	84	87	93	71	76	16	2	23
84	80	26	65	79	70	34	98	12	4	75	19	92	54	12	33	82	87	24	91

Table 1 A table generated by LUA.

line correction is ignored when we use this table as a float, otherwise it assures proper vertical spacing around the table. Watch how we define the tables `one` and `two` beforehand. This saves 198 redundant table constructions.

Not all code will look as simple as this. Consider the following:

```
context.placefigure(
  "caption",
  function() context.externalfigure( { "cow.pdf" } ) end
)
```

Here we pass an argument wrapped in a function. If we would not do that, the external figure would end up wrong, as arguments to functions are evaluated before the function that gets them (we already showed some alternative approaches in previous chapters). A function argument is treated as special and in this case the external figure ends up right. Here is another example:

```
context.placefigure("Two cows!",function()
  context.bTABLE()
  context.bTR()
  context.bTD()
    context.externalfigure(
      { "cow.pdf" },
      { width = "3cm", height = "3cm" }
    )
  context.eTD()
  context.bTD { align = "{lohi,middle}" }
    context("and")
  context.eTD()
  context.bTD()
    context.externalfigure(
```

```

    { "cow.pdf" },
    { width = "4cm", height = "3cm" }
  )
  context.eTD()
  context.eTR()
  context.eTABLE()
end)

```

In this case the figure is not an argument so it gets flushed sequentially with the rest:

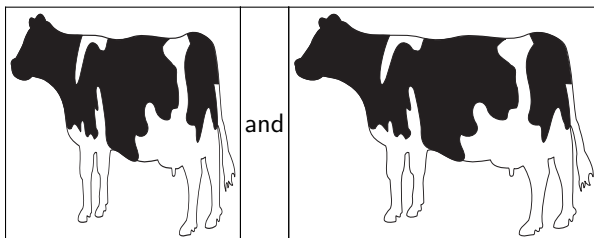


Figure 2 Two cows!

Styles

Say that you want to typeset a word in a bold font. You can do that this way:

```

context("This is ")
context.bold("important")
context("!")

```

Now imagine that you want this important word to be in red too. As we have a nested command, we end up with a nested call:

```

context("This is ")
context.bold(function() context.color( { "red" }, "important") end)
context("!")

```

or

```

context("This is ")
context.bold(context.delayed.color( { "red" }, "important"))
context("!")

```

In that case it's good to know that there is a command that combines both features:

```

context("This is ")
context.style( { style = "bold", color = "red" }, "important")
context("!")

```

But that is still not convenient when we have to do that often. So, you can wrap the style switch in a function.

```

local function mycommands.important(str)
    context.style( { style = "bold", color = "red" }, str )
end
context("This is ")
mycommands.important( "important")
context(", and ")
mycommands.important( "this")
context(" too !")

```

Or you can setup a named style:

```

context.setupstyle( { "important" },
                    { style = "bold", color = "red" } )
context("This is ")
context.style( { "important" }, "important")
context(", and ")
context.style( { "important" }, "this")
context(" too !")

```

Or even define one:

```

context.definestyle( { "important" },
                    { style = "bold", color = "red" } )
context("This is ")
context.important("important")
context(", and ")
context.important("this")
context(" too !")

```

This last solution is especially handy for more complex cases:

```

context.definestyle( { "important" },
                    { style = "bold", color = "red" } )
context("This is ")
context.startimportant()
context.inframed("important")
context.stopimportant()
context(", and ")
context.important("this")
context(" too !")

```

resulting in: This is **important**, and **this** too !

A complete example

One day my 6 year old niece Lorian was at the office and wanted to know what I was doing. As I knew she was practicing arithmetic at school I wrote a quick and dirty script to generate sheets with exercises. The most impressive part was

that the answers were included. It was a rather braindead bit of LUA, written in a few minutes, but the weeks after I ended up running it a few more times, for her and her friends, every time a bit more difficult and also using different arithmetic. It was that script that made me decide to extend the basic cld manual into this more extensive document.

We generate three columns of exercises. Each exercise is a row in a table. The last argument to the function determines if answers are shown.

```
local random = math.random

local function ForLorien(n,maxa,maxb,answers)
  context.startcolumns { n = 3 }
  context.starttabulate { "|r|c|r|c|r|" }
  for i=1,n do
    local sign = random(0,1) > 0.5
    local a, b = random(1,maxa or 99), random(1,max or maxb or 99)
    if b > a and not sign then a, b = b, a end
    context.NC()
    context(a)
    context.NC()
    context.mathematics(sign and "+" or "-")
    context.NC()
    context(b)
    context.NC()
    context("=")
    context.NC()
    context(answers and (sign and a+b or a-b))
    context.NC()
    context.NR()
  end
  context.stoptabulate()
  context.stopcolumns()
  context.page()
end

This is a typical example of where it's more convenient to write the code in
LUA than in TEX's macro language. As a consequence setting up the page also
happens in LUA:
context.setupbodyfont {
  "palatino",
  "14pt"
}
```

```

context.setuplayout {
  backspace = "2cm",
  topspace  = "2cm",
  header    = "1cm",
  footer    = "0cm",
  height    = "middle",
  width     = "middle",
}

```

This leaves us to generate the document. There is a pitfall here: we need to use the same random number for the exercises and the answers, so we freeze and defrost it. Functions in the `commands` namespace implement functionality that is used at the \TeX end but better can be done in $\text{\text{LUA}}$ than in \TeX macro code. Of course these functions can also be used at the $\text{\text{LUA}}$ end.

```

context.starttext()

  local n = 120

  commands.freezerandomseed()

  ForLorien(n,10,10)
  ForLorien(n,20,20)
  ForLorien(n,30,30)
  ForLorien(n,40,40)
  ForLorien(n,50,50)

  commands.defrostrandomseed()

  ForLorien(n,10,10,true)
  ForLorien(n,20,20,true)
  ForLorien(n,30,30,true)
  ForLorien(n,40,40,true)
  ForLorien(n,50,50,true)

context.stoptext()

```

A few pages of the result are shown in figure 3. In the `CONTEXT` distribution a more advanced version can be found in `s-edu-01.cld` as I was also asked to generate multiplication and table exercises. In the process I had to make sure that there were no duplicates on a page as she complained that was not good. There a set of sheets is generated with:

```

moduledata.educational.arithmetic.generate {
  name      = "Bram Otten",
  fontsize  = "12pt",
}

```

1			6		
6 - 4 =	6 + 8 =	4 - 3 =	6 - 4 = 2	6 + 8 = 14	4 - 3 = 1
1 + 8 =	8 + 9 =	3 - 3 =	1 + 8 = 9	8 + 9 = 17	3 - 3 = 0
4 + 1 =	1 + 10 =	8 + 2 =	4 + 1 = 5	1 + 10 = 11	8 + 2 = 10
10 + 2 =	7 - 7 =	8 + 7 =	10 + 2 = 12	7 - 7 = 0	8 + 7 = 15
5 + 3 =	5 + 5 =	10 - 5 =	5 + 3 = 8	5 + 5 = 10	10 - 5 = 5
9 - 7 =	4 - 2 =	9 - 6 =	9 - 7 = 2	4 - 2 = 2	9 - 6 = 3
6 + 10 =	9 - 3 =	5 + 1 =	6 + 10 = 16	9 - 3 = 6	5 + 1 = 6
4 + 9 =	7 - 6 =	2 - 1 =	4 + 9 = 13	7 - 6 = 1	2 - 1 = 1
9 + 10 =	8 + 3 =	1 + 2 =	9 + 10 = 19	8 + 3 = 11	1 + 2 = 3
7 - 4 =	2 - 1 =	4 + 1 =	7 - 4 = 3	2 - 1 = 1	4 + 1 = 5
10 + 2 =	6 - 5 =	5 + 4 =	10 + 2 = 12	6 - 5 = 1	5 + 4 = 9
9 - 3 =	5 + 6 =	8 - 8 =	9 - 3 = 6	5 + 6 = 11	8 - 8 = 0
7 - 1 =	7 - 2 =	4 + 10 =	7 - 1 = 6	7 - 2 = 5	4 + 10 = 14
8 - 4 =	7 + 6 =	7 + 5 =	8 - 4 = 4	7 + 6 = 13	7 + 5 = 12
4 - 3 =	4 + 10 =	10 - 5 =	4 - 3 = 1	4 + 10 = 14	10 - 5 = 5
9 - 5 =	10 - 7 =	8 - 1 =	9 - 5 = 4	10 - 7 = 3	8 - 1 = 7
5 + 6 =	4 - 3 =	5 + 9 =	5 + 6 = 11	4 - 3 = 1	5 + 9 = 14
6 + 1 =	7 - 2 =	3 - 2 =	6 + 1 = 7	7 - 2 = 5	3 - 2 = 1
5 - 4 =	4 - 3 =	5 + 1 =	5 - 4 = 1	4 - 3 = 1	5 + 1 = 6
4 - 1 =	1 + 7 =	5 + 9 =	4 - 1 = 3	1 + 7 = 8	5 + 9 = 14
1 + 3 =	4 + 3 =	9 - 9 =	1 + 3 = 4	4 + 3 = 7	9 - 9 = 0
3 + 5 =	9 - 2 =	6 - 1 =	3 + 5 = 8	9 - 2 = 7	6 - 1 = 5
5 + 5 =	8 - 6 =	9 + 2 =	5 + 5 = 10	8 - 6 = 2	9 + 2 = 11
7 - 2 =	8 + 2 =	10 - 9 =	7 - 2 = 5	8 + 2 = 10	10 - 9 = 1
10 - 9 =	5 + 8 =	10 - 1 =	10 - 9 = 1	5 + 8 = 13	10 - 1 = 9
5 - 1 =	7 - 4 =	7 - 5 =	5 - 1 = 4	7 - 4 = 3	7 - 5 = 2
8 + 8 =	5 + 3 =	7 + 10 =	8 + 8 = 16	5 + 3 = 8	7 + 10 = 17
5 + 1 =	6 + 7 =	7 - 1 =	5 + 1 = 6	6 + 7 = 13	7 - 1 = 6
10 - 9 =	9 + 9 =	3 - 2 =	10 - 9 = 1	9 + 9 = 18	3 - 2 = 1
7 - 4 =	5 - 2 =	7 + 5 =	7 - 4 = 3	5 - 2 = 3	7 + 5 = 12
6 - 3 =	6 - 5 =	8 - 8 =	6 - 3 = 3	6 - 5 = 1	8 - 8 = 0
2 + 1 =	5 - 1 =	6 + 6 =	2 + 1 = 3	5 - 1 = 4	6 + 6 = 12
8 + 7 =	2 - 1 =	9 - 6 =	8 + 7 = 15	2 - 1 = 1	9 - 6 = 3
5 + 7 =	7 + 4 =	9 - 3 =	5 + 7 = 12	7 + 4 = 11	9 - 3 = 6
9 - 4 =	9 + 4 =	6 + 4 =	9 - 4 = 5	9 + 4 = 13	6 + 4 = 10
10 - 9 =	3 - 2 =	9 - 2 =	10 - 9 = 1	3 - 2 = 1	9 - 2 = 7
6 + 2 =	1 + 9 =	8 - 7 =	6 + 2 = 8	1 + 9 = 10	8 - 7 = 1
7 - 1 =	8 - 7 =	8 - 1 =	7 - 1 = 6	8 - 7 = 1	8 - 1 = 7
6 + 5 =	5 - 1 =	7 - 5 =	6 + 5 = 11	5 - 1 = 4	7 - 5 = 2
10 + 2 =	5 - 1 =	5 + 10 =	10 + 2 = 12	5 - 1 = 4	5 + 10 = 15

exercises

answers

Figure 3 Lorien's challenge.

```

columns = 2,
run      = {
  { method = "bin_add_and_subtract", maxa = 8, maxb = 8 },
  { method = "bin_add_and_subtract", maxa = 16, maxb = 16 },
  { method = "bin_add_and_subtract", maxa = 32, maxb = 32 },
  { method = "bin_add_and_subtract", maxa = 64, maxb = 64 },
  { method = "bin_add_and_subtract", maxa = 128, maxb = 128 },
},
}

```

Interfacing

The fact that we can define functionality using LUA code does not mean that we should abandon the \TeX interface. As an example of this we use a relatively simple module for typesetting morse code.³ First we create a proper namespace:

```

moduledata.morse = moduledata.morse or { }
local morse      = moduledata.morse

```

³The real module is a bit larger and can format verbose morse.

We will use a few helpers and create shortcuts for them. The first helper loops over each UTF character in a string. The other two helpers map a character onto an uppercase (because morse only deals with uppercase) or onto an similar shaped character (because morse only has a limited character set).

```
local utfcharacters = string.utfcharacters
```

```
local ucchars, shchars = characters.ucchars, characters.shchars
```

The morse codes are stored in a table.

```
local codes = {
    ["A"] = ".-.",      ["B"] = "-...",
    ["C"] = "-.-.-",    ["D"] = "-.-.",
    ["E"] = "..",        ["F"] = ".-.-.",
    ["G"] = "-.-.-",    ["H"] = "-.-.-.",
    ["I"] = ".-.-",     ["J"] = "-.-.-",
    ["K"] = "-.-.-",    ["L"] = "-.-.-.",
    ["M"] = "---",      ["N"] = "-.-.",
    ["O"] = "---.-",    ["P"] = "-.-.-.-",
    ["Q"] = "-.-.-.-",  ["R"] = "-.-.-.",
    ["S"] = ".-.-.-",   ["T"] = "-.-",
    ["U"] = ".-.-.-",   ["V"] = "-.-.-.-",
    ["W"] = "-.-.-",    ["X"] = "-.-.-.-",
    ["Y"] = "-.-.-.-",  ["Z"] = "-.-.-.-",

    ["0"] = "-----",  ["1"] = ".-.-.-.-",
    ["2"] = ".-.-.-.-", ["3"] = "-.-.-.-",
    ["4"] = ".-.-.-.-", ["5"] = "-.-.-.-",
    ["6"] = "-.-.-.-",  ["7"] = "-.-.-.-",
    ["8"] = "-.-.-.-",  ["9"] = "-.-.-.-",
    ["."] = ".-.-.-.-", [","] = "-.-.-.-",
    [":"] = "-.-.-.-", [";"] = "-.-.-.-",
    ["?"] = "-.-.-.-", ["!"] = "-.-.-.-",
    ["-"] = "-.-.-.-", ["/"] = "-.-.-.-",
    ["("] = "-.-.-.-", [")"] = "-.-.-.-",
    ["="] = "-.-.-.-", ["@"] = "-.-.-.-",
    ["'"] = "-.-.-.-", ["'"] = "-.-.-.-",

    ["À"] = "-.-.-.-",  ["Á"] = "-.-.-.-",
    ["Ä"] = "-.-.-.-",  ["Æ"] = "-.-.-.-",
    ["Ç"] = "-.-.-.-",  ["È"] = "-.-.-.-",
    ["Ê"] = "-.-.-.-",  ["Ë"] = "-.-.-.-",
    ["Ö"] = "-.-.-.-",  ["Ø"] = "-.-.-.-",
    ["Ü"] = "-.-.-.-",  ["ß"] = "-.-.-.-",
}
```

```
morse.codes = codes
```

As you can see, there are a few non ASCII characters supported as well. There will never be full UNICODE support simply because morse is sort of obsolete. Also, in order to support UNICODE one could as well use the bits of UTF characters, although ... memorizing the whole UNICODE table is not much fun.

We associate a metatable index function with this mapping. That way we can not only conveniently deal with the casing, but also provide a fallback based on the shape. Once found, we store the representation so that only one lookup is needed per character.

```
local function resolvemorse(t,k)
  if k then
    local u = ucchars[k]
    local v = rawget(t,u) or rawget(t,shchars[u]) or false
    t[k] = v
    return v
  else
    return false
  end
end
```

```
setmetatable(codes, { __index = resolvemorse })
```

Next comes some rendering code. As we can best do rendering at the `TEX` end we just use macros.

```
local MorseBetweenWords      = context.MorseBetweenWords
local MorseBetweenCharacters = context.MorseBetweenCharacters
local MorseLong              = context.MorseLong
local MorseShort              = context.MorseShort
local MorseSpace              = context.MorseSpace
local MorseUnknown            = context.MorseUnknown
```

The main function is not that complex. We need to keep track of spaces and newlines. We have a nested loop because a fallback to shape can result in multiple characters.

```
function morse.tomorse(str)
  local inmorse = false
  for s in utfcharacters(str) do
    local m = codes[s]
    if m then
      if inmorse then
        MorseBetweenWords()
      else
        inmorse = true
      end
    end
  end
end
```

```

end
local done = false
for m in utfcharacters(m) do
    if done then
        MorseBetweenCharacters()
    else
        done = true
    end
    if m == "." then
        MorseShort()
    elseif m == "-" then
        MorseLong()
    elseif m == " " then
        MorseBetweenCharacters()
    end
end
inmorse = true
elseif s == "\n" or s == " " then
    MorseSpace()
    inmorse = false
else
    if inmorse then
        MorseBetweenWords()
    else
        inmorse = true
    end
    MorseUnknown(s)
end
end
end

```

We use this function in two additional functions. One typesets a file, the other a table of available codes.

```

function morse.filetomorse(name,verbose)
    morse.tomorse(resolvers.loadtexfile(name),verbose)
end

```

```

function morse.showtable()
    context.starttabulate { "|l|l|" }
    for k, v in table.sortedpairs(codes) do
        context.NC() context(k)
        context.NC() morse.tomorse(v,true)
    end
end

```

```

        context.NC() context.NR()
    end
    context.stoptabulate()
end

```

We're done with the LUA code that we can either put in an external file or put in the module file. The T_EX file has two parts. The typesetting macros that we use at the LUA end are defined first. These can be overloaded.

```

\def\MorseShort
{
  \dontleavehmode
  \vrule
    width \MorseWidth
    height \MorseHeight
    depth \zeropoint
  \relax}

\def\MorseLong
{
  \dontleavehmode
  \vrule
    width 3\dimexpr\MorseWidth
    height \MorseHeight
    depth \zeropoint
  \relax}

\def\MorseBetweenCharacters
{
  \kern\MorseWidth}

\def\MorseBetweenWords
{
  \hskip3\dimexpr\MorseWidth\relax}

\def\MorseSpace
{
  \hskip7\dimexpr\MorseWidth\relax}

\def\MorseUnknown#1
{
  \detokenize{#1}}

```

The dimensions are stored in macros as well. Of course we could provide a proper setup command, but it hardly makes sense.

```

\def\MorseWidth {0.4em}
\def\MorseHeight{0.2em}

```

Finally we have arrived at the macros that interface to the LUA functions.

```

\def\MorseString#1{\ctxlua{moduledata.morse.tomorse(\!bs#1\!es)}}
\def\MorseFile #1{\ctxlua{moduledata.morse.filetomorse("#1")}}

```


is constructed. It is for this reason that no color is applied: the snippets that end up in the box are already typeset.

An alternative is to delegate the task to LUA:

```
\startluacode
local function process(data)

    local words = lpeg.split(lpeg.patterns.spacer,data or "")

    for i=1,#words do
        if i == 1 then
            context.dontleavehmode()
        else
            context.space()
        end
        context.colored(words[i])
    end

end

end
```

```
process(io.loaddata(resolvers.findfile("ward.tex")))
\stopluacode
```

This results in:

The function splits the loaded data into a table with individual words. We use a splitter that splits on spacing tokens. The special case for `i = 1` makes sure that we end up in horizontal mode (read: properly start a paragraph). This time we do get color because the typesetting is done directly. Here is an alternative implementation:

```
local done = false

local function reset()
    done = false
    return true
end

local function apply(s)
    if done then
        context.space()
    else
        done = true
        context.dontleavehmode()
    end
end
```

```

    end
    context.colored(s)
end

local splitter = lpeg.P(reset)
    * lpeg.splitter(lpeg.patterns.spacer, apply)

local function process(data)
    lpeg.match(splitter, data)
end

This version is more efficient as it does not create an intermediate table. The
next one is comparable:
local function apply(s)
    context.colored("%s ", s)
end

local splitter lpeg.splitter(lpeg.patterns.spacer, apply)

local function process(data)
    context.dontleavevmode()
    lpeg.match(splitter, data)
    context.removeunwantedspaces()
end

```

Formatters

Sometimes you can save a bit of work by using formatters. By default, the `context` command, when called directly, applies a given formatter. But when called as table this feature is lost because then we want to process non-strings as well. The next example shows a way out:

```

context("the current emwidth is %p", \number\emwidth)
context.par()
context.formatted("the current emwidth is %p", \number\emwidth)
context.par()
context.bold(string.formatters["the current emwidth is %p"](\number\emwidth))
context.par()
context.formatted.bold("the current emwidth is %p",
    \number\emwidth)

```

The last one is the most interesting one here: in the subnamespace `formatted` (watch the `d`) a format specification with extra arguments is expected. This is the result: the current emwidth is 10pt

the current emwidth is 10pt
the current emwidth is 10pt
the current emwidth is 10pt.

Summary

context("...")

The string is flushed directly.

...

context("format",...)

The first string is a format specification according that is passed to the LUA function `format` in the `string` namespace. Following arguments are passed too.

`format("format",...)`

context(123,...)

The numbers (and following numbers or strings) are flushed without any formatting.

123... (concatenated)

context(true)

An explicit `newlinechar` is inserted.

^^M

context(false,...)

Strings and numbers are flushed surrounded by curly braces, an indexed table is flushed as option list, and a hashed table is flushed as parameter set.

`multiple {...}` or `[...] etc`

context(node)

The `node(list)` is injected at the spot. Keep in mind that you need to do the proper memory management yourself.

context.command(value,...)

The value (string or number) is flushed as a curly braced (regular) argument.

`\command {value}...`

context.command(value ,...)

The table is flushed as value set. This can be an identifier, a list of options, or a directive.

`\command [value]...`

context.command(key = value ,...)

The table is flushed as key/value set.

`\command [key={value}]...`

context.command(true)

An explicit `endlinechar` is inserted.

`\command ^~M`

context.command(node)

The node(list) is injected at the spot. Keep in mind that you need to do the proper memory management yourself.

`\command {node(list)}`

context.command(false,value)

The value is flushed without encapsulating tokens.

`\command value`

context.command(value , key = value , value, false, value)

The arguments are flushed accordingly their nature and the order can be any.

`\command [value][key={value}]{value}value`

context.direct(...)

The arguments are interpreted the same as if `direct` was a command, but no `\direct` is injected in front.

context.delayed(...)

The arguments are interpreted the same as in a `context` call, but instead of a direct flush, the arguments will be flushed in a next cycle.

context.delayed.command(...)

The arguments are interpreted the same as in a **command** call, but instead of a direct flush, the command and arguments will be flushed in a next cycle.

context.nested.command

This command returns the command, including given arguments as a string. No flushing takes place.

context.nested

This command returns the arguments as a string and treats them the same as a regular **context** call.

context.formatted.command

This command returns the command that will pass it's arguments to the string formatter.

context.formatted

This command passes it's arguments to the string formatter.

context.metafun.start(...)

This starts a METAFUN (or METAPOST) graphic.

context.metafun()

This finishes and flushes a METAFUN (or METAPOST) graphic.

context.metafun.stop(...)

The argument is appended to the current graphic data.

context.metafun.stop("format",...)

The argument is appended to the current graphic data but the string formatter is used on following arguments.

ConTEXt–Lua dokumenty

V rámci formátu CONTEXT lze připravovat dokumenty pomocí kombinace jazyků TEX, XML, METAPOST a LUA. Článek v krátkosti shrnuje základy jazyka LUA a následně se věnuje způsobům, jakými je možné jazyk využít při přípravě dokumentů ve formátu CONTEXT MKIV.

Klíčová slova: LUA, LUATEX, CONTEXT, MKIV

Hans Hagen, pragma@wxs.nl

The article describes the XML backend of CON_TE_XT, which can be used to produce structured XML documents out of a T_EX input. One of the many applications of the XML backend is the conversion to ePub e-book format, which the article covers in detail.

Keywords: CON_TE_XT, XML, ePub

Introduction

There is a pretty long tradition of typesetting math with T_EX and it looks like this program will dominate for many more years. Even if we move to the web, the simple fact that support for MathML in some browsers is suboptimal will drive those who want a quality document to use PDF instead.

I'm writing this in 2014, at a time when XML is widespread. The idea of XML is that you code your data in a very structured way, so that it can be manipulated and (if needed) validated. Text has always been a target for XML which is a follow-up to SGML that was in use by publishers. Because HTML is less structured (and also quite tolerant with respect to end tags) we prefer to use XHTML but unfortunately support for that is less widespread.

Interestingly, documents are probably among the more complex targets of the XML format. The reason is that unless the author restricts him/herself or gets restricted by the publisher, tag abuse can happen. At Pragma ADE we mostly deal with education-related XML and it's not always easy to come up with something that suits the specific needs of the educational concept behind a school method. Even if we start out nice and clean, eventually we end up with a polluted source, often with additional structure needed to satisfy the tools used for conversion.

We have been supporting XML from the day it showed up and most of our projects involve XML in one way or the other. That doesn't mean that we don't use T_EX for coding documents. This manual is for instance a regular T_EX document. In many ways a structured T_EX document is much more convenient to edit, especially if one wants to add a personal touch and do some local page make-up. On the other hand, diverting from standard structure commands makes the document less suitable for output other than PDF. There is simply no final solution for coding a document, it's mostly a matter of taste.

So we have a dilemma: if we want to have multiple output, frozen PDF as well as less-controlled HTML output, we can best code in XML, but when we want to code comfortably we'd like to use \TeX . There are other ways, like Markdown, that can be converted to intermediate formats like \TeX , but that is only suitable for simple documents: the more advanced documents get, the more one has to escape from the boundaries of (any) document encoding, and then often \TeX is not a bad choice. There is a good reason why \TeX survived for so long.

It is for this reason that in $\text{\texttt{CONTEXT MkIV}}$ we can export the content in a reasonable structured way to XML. Of course we assume a structured document. It started out as an experiment because it was relatively easy to implement, and it is now an integral component.

The output

The regular output is an XML file but as we have some more related data it gets organized in a tree. We also export a few variants. An example is given below:

```
./test-export
./test-export/images
./test-export/images/...
./test-export/styles
./test-export/styles/test-defaults.css
./test-export/styles/test-images.css
./test-export/styles/test-styles.css
./test-export/styles/test-templates.css
./test-export/test-raw.xml
./test-export/test-raw.lua
./test-export/test-tag.xhtml
./test-export/test-div.xhtml
```

Say that we have this input:

```
\setupbackend
  [export=yes]

\starttext
  \startsection[title=First]
    \startitemize
      \startitem one \stopitem
      \startitem two \stopitem
    \stopitemize
  \stopsection
\stoptext
```

The main export ends up in the `test-raw.xml` export file and looks like the following (we leave out the preamble and style references):

```
<document> <!-- with some attributes -->
  <section detail="section" chain="section" level="3">
    <sectionnumber>1</sectionnumber>
    <sectiontitle>First</sectiontitle>
    <sectioncontent>
      <itemgroup detail="itemize"
        chain="itemize" symbol="1" level="1">
        <item>
          <itemtag><m:math ..><m:mo>•</m:mo></m:math></itemtag>
          <itemcontent>one</itemcontent>
        </item>
        <item>
          <itemtag><m:math ..><m:mo>•</m:mo></m:math></itemtag>
          <itemcontent>two</itemcontent>
        </item>
      </itemgroup>
    </sectioncontent>
  </section>
</document>
```

This file refers to the stylesheets and therefore renders quite well in a browser like Firefox that can handle XHTML with arbitrary tags.

The `detail` attribute tells us what instance of the element is used. Normally the `chain` attribute is the same but it can have more values. For instance, if we have:

```
\definefloat[graphic][graphics][figure]
```

```
.....
```

```
\startplacefigure[title=First]
  \externalfigure[cow.pdf]
\stopplacefigure
```

```
.....
```

```
\startplacegraphic[title=Second]
  \externalfigure[cow.pdf]
\stopplacegraphic
```

we get this:

```
<float detail="figure" chain="figure">
```

```

    <floatcontent>...</floatcontent>
    <floatcaption>...</floatcaption>
</float>
<float detail="graphic" chain="figure graphic">
    <floatcontent>...</floatcontent>
    <floatcaption>...</floatcaption>
</float>

```

This makes it possible to style specific categories of floats by using a (combination of) `detail` and/or `chain` as filters.

The body of the `test-tag.xhtml` file looks similar but it is slightly more tuned for viewing. For instance, hyperlinks are converted to a way that CSS and browsers like more. Keep in mind that the raw file can be the base for conversion to other formats, so that one stays closest to the original structure.

The `test-div.xhtml` file is even more tuned for viewing in browsers as it completely does away with specific tags. We explicitly don't map onto native HTML elements because that would make everything look messy and horrible, if only because there seldom is a relation between those elements and the original. One can always transform one of the export formats to pure HTML tags if needed.

```

<body>
  <div class="document">
    <div class="section" id="aut-1">
      <div class="sectionnumber">1</div>
      <div class="sectiontitle">First</div>
      <div class="sectioncontent">
        <div class="itemgroup itemize symbol-1">
          <div class="item">
            <div class="itemtag"><m:math ...>
              <m:mo>•</m:mo></m:math></div>
            <div class="itemcontent">one</div>
          </div>
          <div class="item">
            <div class="itemtag"><m:math ...>
              <m:mo>•</m:mo></m:math></div>
            <div class="itemcontent">two</div>
          </div>
        </div>
      </div>
      <div class="float figure">
        <div class="floatcontent">...</div></div>
        <div class="floatcaption">...</div>
      </div>
      <div class="float figure graphic">

```

```

        <div class="floatcontent">...</div></div>
        <div class="floatcaption">...</div>
    </div>
</div>
</body>

```

The default CSS file can deal with tags as well as classes. The file of additional styles contains definitions of so-called highlights. In the `CONTEXT` source one is better off using explicit named highlights instead of local font and color switches because these properties are then exported to the CSS. The `images` style defines all images used. The `templates` file lists all the elements used and can be used as a starting point for additional CSS styling.

Keep in mind that the export is *not* meant as a one-to-one visual representation. It represents structure so that it can be converted to whatever you like.

In order to get an export you must start your document with:

```

\setupbackend
[export=yes]

```

So, we trigger a specific (extra) backend. In addition you can set up the export:

```

\setupexport
[svgstyle=test-basic-style.tex,
cssfile=test-extras.css,
hyphen=yes,
width=60em]

```

The `hyphen` option will also export hyphenation information so that the text can be nicely justified. The `svgstyle` option can be used to specify a file where math is set up; normally this would only contain a `bodyfont` setup, and this option is only needed if you want to create an ePub file afterwards which has math represented as SVG.

The value of `cssfile` ends up as a style reference in the exported files. You can also pass a comma-separated list of names (between curly braces). These entries come after those of the automatically generated CSS files so you need to be aware of default properties.

Images

Inclusion of images is done in an indirect way. Each image gets an entry in a special image related stylesheet and then gets referred to by `id`. Some extra information is written to a status file so that the script that creates ePub files

can deal with the right conversion, for instance from PDF to SVG. Because we can refer to specific pages in a PDF file, this subsystem deals with that too. Images are expected to be in an `images` subdirectory and because in CSS the references are relative to the path where the stylesheet resides, we use `../images` instead. If you do some postprocessing on the files or relocate them you need to keep in mind that you might have to change these paths in the image-related CSS file.

Epub files

At the end of a run with exporting enabled you will get a message to the console that tells you how to generate an ePub file. For instance:

```
mtxrun --script epub --make --purge test
```

This will create a tree with the following organization:

```
./test-epub
./test-epub/META-INF
./test-epub/META-INF/container.xml
./test-epub/OEBPS
./test-epub/OEBPS/content.opf
./test-epub/OEBPS/toc.ncx
./test-epub/OEBPS/nav.xhtml
./test-epub/OEBPS/cover.xhtml
./test-epub/OEBPS/test-div.xhtml
./test-epub/OEBPS/images
./test-epub/OEBPS/images/...
./test-epub/styles
./test-epub/styles/test-defaults.css
./test-epub/styles/test-images.css
./test-epub/styles/test-styles.css
./test-epub/mimetype
```

Images will be moved to this tree as well and if needed they will be converted, for instance into SVG. Converted PDF files can have a `page-<number>` in their name when a specific page has been used.

You can pass the option `--svgmath` in which case math will be converted to SVG. The main reason for this feature is that we found out that MathML support in browsers is not currently as widespread as might be expected. The best bet is Firefox which natively supports it. The Chrome browser had it for a while but it got dropped and math is now delegated to JavaScript and friends. In Internet Explorer MathML should work (but I need to test that again).

This conversion mechanism is kind of interesting: one enters $\text{T}_{\text{E}}\text{X}$ math, then gets MathML in the export, and that gets rendered by $\text{T}_{\text{E}}\text{X}$ again, but now as a standalone snippet that then gets converted to SVG and embedded in the result.

Styles

One can argue that we should use native HTML elements but since we don't have a nice guaranteed-consistent mapping onto that, it makes no sense to do so. Instead, we rely on either explicit tags with details and chains or divisions with classes that combine the tag, detail and chain. The tagged variant has some more attributes and those that use a fixed set of values become classes in the division variant. Also, once we start going the (for instance) H1, H2, etc. route we're lost when we have more levels than that or use a different structure. If an H3 can reflect several levels it makes no sense to use it. The same is true for other tags: if a list is not really a list than tagging it with LI is counterproductive. We're often dealing with very complex documents so basic HTML tagging becomes rather meaningless.

If you look at the division variant (this is used for ePub too) you will notice that there are no empty elements but `div` blocks with a comment as content. This is needed because otherwise they get ignored, which for instance makes table cells invisible.

The relation between `detail` and `chain` (reflected in `class`) can best be seen from the next example.

```
\definefloat[myfloata]
\definefloat[myfloatb][myfloatbs][figure]
\definefloat[myfloatc][myfloatcs][myfloatb]
```

This creates two new float instances. The first inherits from the main float settings, but can have its own properties. The second example inherits from the `figure` so in fact it is part of a chain. The third one has a longer chain.

```
<float detail="myfloata">...</float>
<float detail="myfloatb" chain="figure">...</float>
<float detail="myfloatc" chain="figure myfloatb">...</float>
```

In a CSS style you can now configure tags, details, and chains as well as classes (we show only a few possibilities). Here, the CSS element on the first line of each pair is invoked by the CSS selector on the second line.

```
div.float.myfloata { }          float[detail='myfloata'] { }
div.float.myfloatb { }          float[detail='myfloatb'] { }
div.float.figure { }            float[detail='figure'] { }
div.float.figure.myfloatb { }   float[chain~='figure']
                                [detail='myfloata'] { }
div.myfloata { }                *[detail='myfloata'] { }
div.myfloatb { }                *[detail='myfloatb'] { }
div.figure { }                  *[chain~='figure'] { }
div.figure.myfloatb { }         *[chain~='figure']
                                [detail='myfloatb'] { }
```

The default styles cover some basics but if you're serious about the export or want to use ePub then it makes sense to overload some of it and/or provide additional styling. You can find plenty about CSS and its options on the Internet.

Coding

The default output reflects the structure present in the document. If that is not enough you can add your own structure, as in:

```
\startelement[question]
Is this right?
\stopelement
```

You can also pass attributes:

```
\startelement[question] [level=difficult]
Is this right?
\stopelement
```

But these will be exported only when you also say:

```
\setupexport
[properties=yes]
```

You can create a namespace. The following will generate attributes like my-level.

```
\setupexport
[properties=my-]
```

In most cases it makes more sense to use highlights:

```
\definehighlight
[important]
[style=bold]
```

This has the advantage that the style and color are exported to a special CSS file.

Headers, footers, and other content that is part of the page builder are not exported. If your document has cover pages you might want to hide them too. The same is true when you create special chapter title rendering with a side effect that content ends up in the page stream. If something shows up that you don't want, you can wrap it in an **ignore** element:

```
\startelement[ignore]
Don't export this.
\stopelement
```

Acknowledgement

The above article is available through the `CONTEXT` distribution. It can be found in `/texmf-context/doc/context/sources/general/manuals/epub`.

We would like to thank Karl Berry for proofreading and corrections.

Export dokumentů ve formátu ConT_EXt do XML a ePub

Článek popisuje výstupní modul pro `CONTEXT`, který slouží pro generování strukturovaných XML dokumentů z `TEX`ového vstupu. Jednou z aplikací tohoto modulu, které se článek věnuje blíže, je export do formátu ePub využívaného čtečkami elektronických knih.

Klíčová slova: `CONTEXT`, XML, ePub

Hans Hagen, pragma@wxs.nl

Oriental T_EX: Optimizing Paragraphs

HANS HAGEN, IDRIS SAMAWI HAMID

The article describes the state of the art in paragraph optimization for Arabic as implemented in CON_TE_XT. The implementation is introduced using Latin script examples. The article proceeds to describe the main features of Arabic script and known approaches towards paragraph optimization. One of the described approaches is then implemented and used to typeset a passage from the Qur’ān.

Keywords: microtypography, OPEN_TY_PE, CON_TE_XT, L_UA_TE_X

Introduction

One of the objectives of the Oriental T_EX project has always been to play with paragraph optimization. The original assumption was that we needed an advanced non-standard paragraph builder to Arabic done right but in the end we found out that a more straightforward approach is to use a sophisticated OPEN_TY_PE font in combination with a paragraph postprocessor that uses the advanced font capabilities. This solution is somewhat easier to imagine than a complex paragraph builder but still involves quite some juggling.

At the June 2012 meeting of the NTG there was a talk about typesetting Devanagari and as fonts are always a nice topic (if only because there is something to show) it made sense to tell a bit more about optimizing Arabic at the same time. In fact, that presentation was already a few years too late because a couple of years back, when the oriental T_EX project was presented at TUG and Dante meetings, the optimizer was already part of the CON_TE_XT core code. The main reason for not advocating it was the simple fact that no font other than the (not yet finished) Husayni font provided the relevant feature set.

The lack of advanced fonts does not prevent us from showing what we’re dealing with. This is because the CON_TE_XT mechanisms are generic in the sense that they can also be used with regular Latin fonts, although it does not make that much sense. Anyhow, in the next section we wrap up the current state of typesetting Arabic in CON_TE_XT. We focus on the rendering, and leave general aspects of bidirectional typesetting and layouts for another time.

This article is written by Idris Samawi Hamid and Hans Hagen and is typeset by CON_TE_XT MkIV which uses L_UA_TE_X. This program is an extension of T_EX that uses L_UA to open up the core machinery. The L_UA_TE_X core team consists of Taco Hoekwater, Hartmut Henkel and Hans Hagen.

Manipulating glyphs

When discussing optical optimization of a paragraph, a few alternatives come to mind:

- One can get rid of extensive spaces by adding additional kerns between glyphs. This is often used by poor man’s typesetting programs (or routines) and can be applied to non-connecting scripts. It just looks bad. Of course, for connected scripts like Arabic, inter-glyph kerning is not an option, not even in principle.
- Glyphs can be widened a few percent and this is an option that L^AT_EX inherits from its predecessor p^dfT_EX. Normally this goes unnoticed although excessive scaling makes things worse, and yes, one can run into such examples. This strategy goes under the name hz-optimization (the hz refers to Hermann Zapf, who first came up with this solution).¹
- A real nice solution is to replace glyphs by narrower or wider variants. This is in fact the ideal hz solution – including Arabic script as well – but for it to happen one not only needs fonts with alternative shapes, but also a machinery that can deal with them.
- An already old variant is the one first used by Gutenberg, who used alternative cuts for certain combinations of characters. This is comparable with ligatures. However, to make the look and feel optimal, one needs to analyze the text and make decisions on what to replace without losing consistency.

The solution described here does a bit of everything. As it is mostly meant for a connective script, the starting point is how a scribe works when filling up a line nicely. Depending on how well one can see it coming, the writing can be adapted to widen or narrow following words. And it happens that in Arabic script there are quite some ways to squeeze more characters in a small area and/or expand some to the extreme to fill up the available space. Shapes can be wider or narrower, they can be stacked and they can get replaced by ligatures. Of course there is some interference with the optional marks on top and below but even there we have some freedom. The only condition is that the characters in a word stay connected.²

So, given enough alternative glyphs, one can imagine that excessive interword spacing can be avoided. However, it is non-trivial to check all possible combinations. Actually, it is not needed either, as carefully chosen aesthetic rules put

¹Sometimes hz-optimization also goes under the rubric of “Semitic justification”. See, e.g., Bringhurst in pre-3rd editions of his *Elements of Typographic Style*. This technique does not work well for Arabic script in general because glyphs are connected in two dimensions. On the other hand, a certain basic yet ubiquitous Semitic justification *can* be achieved by using the *taṭwīl* character, commonly called the *kashīdah* (U+0640). We will discuss this later in this article.

²Much of this is handled within the GPOS features of the OPENTYPE font itself (e.g., `mark` and `mkmk`)

some bounds on what can be done. One should more think in terms of alternative strategies or solutions and this is the terminology that we will therefore use.


Scaling glyphs horizontally is no problem if we keep the scale factor very small, say percentages. This also means that we should not overestimate the impact. For the Arabic script we can stretch more – using non-scaling methods but again there are some constraints, that we will discuss later on.

In the next example, we demonstrate some excessive stretching:



The image shows two instances of the text "We are texies!". The first instance is stretched horizontally to a width of approximately 150%. The second instance is stretched to a width of approximately 200%.

In practice, fonts can provide intercharacter kerning, which is demonstrated next:

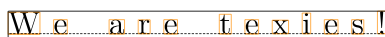


The image shows two instances of the text "We are texies!". The first instance has a kerning value of -0.984 applied between the 'e' and 'a'. The second instance has a kerning value of -0.984 applied between the 'e' and 'a'.

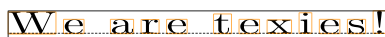
Some poor man's justification routines mess with additional inter-character kerning. Although this is, within reasonable bounds, ok for special purposed like titles, it looks bad in text. The first line expands glyphs and spaces, the second line expands spaces and add additional kerns between characters and the third line expands and add extra kerns.



The image shows the text "We are texies!" where the individual glyphs and spaces are expanded horizontally.



The image shows the text "We are texies!" where the spaces between words are expanded and additional kerns are added between characters.



The image shows the text "We are texies!" where the spaces between words are expanded and extra kerns are added between characters.

Unfortunately we see quite often examples of the last method in novels and even scientific texts. There is definitely a down side to advanced manipulation.

Applying features to Latin script

It is easiest is to start out with Latin, if only because it's more intuitive for most of us to see what happens. This is not the place to discuss all the gory details so you have to take some of the configuration options on face value. Once this mechanism is stable and used, the options can be described. For now we stick to presenting the idea.

Let's assume that you know what font features are. The idea is to work with combinations of such features and figure out what combination suits best. In order not to clutter a document style, these sets are defined in so called goodie files. Here is an excerpt of `demo.lfg`:

```

return {
  name = "demo",
  version = "1.01",
  comment = "An example of goodies.",
  author = "Hans Hagen",
  featuresets = {
    simple = {
      mode = "node",
      script = "latn"
    },
    default = {
      mode = "node",
      script = "latn",
      kern = "yes",
    },
    ligatures = {
      mode = "node",
      script = "latn",
      kern = "yes",
      liga = "yes",
    },
    smallcaps = {
      mode = "node",
      script = "latn",
      kern = "yes",
      smcp = "yes",
    },
  },
  solutions = {
    experimental = {
      less = {
        "ligatures", "simple",
      },
      more = {
        "smallcaps",
      },
    },
  },
}

```

We see four sets of features here. You can use these sets in a `CONTEXT` feature definition, like:

```
\definefontfeature
[solution-demo]
[goodies=demo,
 featureset=default]
```

You can use a set as follows:

```
\definefont
[SomeTestFont]
[tegyrepagellaregular*solution-demo at 10pt]
```

So far, there is nothing special or new, but we can go a step further.

```
\definefontsolution
[solution-a]
[goodies=demo,
 solution=experimental,
 method={normal, preroll},
 criterium=1]
```

```
\definefontsolution
[solution-b]
[goodies=demo,
 solution=experimental,
 method={normal, preroll, split},
 criterium=1]
```

Here we have defined two solutions. They refer to the **experimental** solution in the goodie file `demo.lfg`. A solution has a **less** and a **more** entry. The featuresets mentioned there reflect ways to make a word narrower or wider. There can be more than one way to do that, although it comes at a performance price. Before we see how this works out we turn on a tracing option:

```
\enabletrackers
[builders.paragraphs.solutions.splitters.colors]
```

This will color the words in the result according to what has happened. When a featureset out of the **more** category has been applied, the words turn green, when **less** is applied, the word becomes yellow. The **preroll** option in the **method** list makes sure that we do a more extensive test beforehand.

```
\SomeTestFont \startfontsolution[solution-a]
\input zapf \par
\stopfontsolution
```

In Figure 1 we see what happens. In each already split line words get wider or narrower until we're satisfied. A criterium of 1 is pretty strict.³ Keep in mind that we use some arbitrary features here. We try removing kerns to get narrower although there is nothing that guarantees that kerns are positive. On the other

³This number reflects the maximum badness and future versions might have a different measure with more granularity.

Coming back to the use of typefaces in electronic publishing: many of the new typographers receive their knowledge and information about the rules of typography from books, from computer magazines or the instruction manuals which they get with the purchase of a PC or software. There is not so much basic instruction, as of now, as there was in the old days, showing the differences between good and bad typographic design. Many people are just fascinated by their PC's tricks, and think that a widely-praised program, called up on the screen, will make everything automatic from now on.

normal

Coming back to THE USE OF TYPEFACES IN ELECTRONIC PUBLISHING: MANY OF THE new TYPOGRAPHERS RECEIVE THEIR knowledge and information about the rules of typography from books, from computer magazines OR THE INSTRUCTION MANUALS WHICH THEY GET WITH THE PURCHASE OF A PC OR SOFTWARE. THERE IS NOT SO MUCH basic instruction, as of now, as there was in the old days, showing the differences between GOOD AND bad typographic DESIGN. Many people are just fascinated by THEIR PC'S TRICKS, AND THINK THAT A widely-praised program, CALLED UP ON THE SCREEN, WILL MAKE EVERYTHING automatic from now on.

solution

Figure 1: Solution a.

hand, using ligatures might help. In order to get wider we use smallcaps. Okay, the result will look somewhat strange but so does much typesetting nowadays.

There is one pitfall here. This mechanism is made for a connective script where hyphenation is not used. As a result a word here is actually split up when it has discretionaries and of course this text fragment has. It goes unnoticed in the rendering but is of course far from optimal.

```
\SomeTestFont \startfontsolution[solution-b]
\input zapf \par
\stopfontsolution
```

In this example (Figure 2) we keep words as a whole but as a side effect we skip words that are broken across a line. This is mostly because it makes not much sense to implement it as Latin is not our target. Future versions of CONTEX_T might get more sophisticated font machinery so then things might look better.

We show two more methods:

```
\definefontsolution
[solution-c]
[goodies=demo,
solution=experimental,
method={reverse,preroll},
criterium=1]
\definefontsolution
```

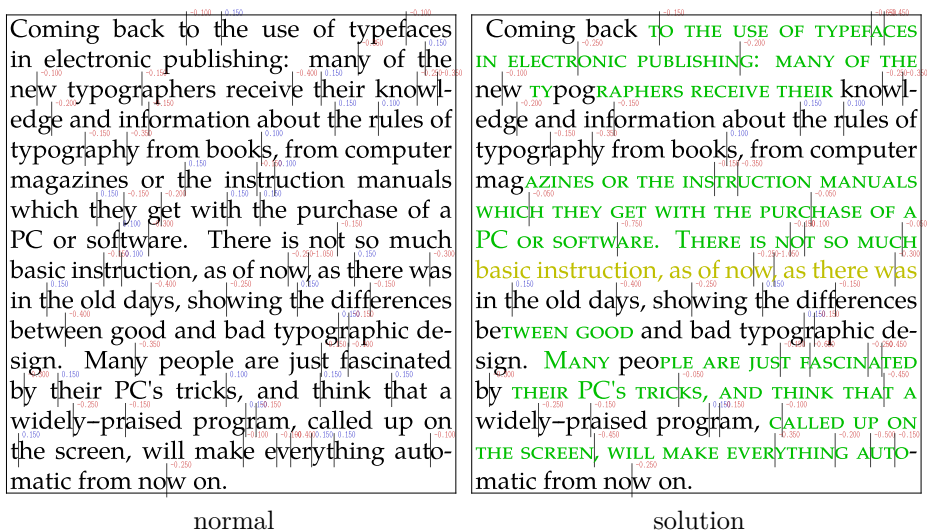


Figure 2: Solution b.

```
[solution-d]
[goodies=demo,
 solution=experimental,
 method={random,preroll,split},
 criterium=1]
```

In Figure 3 we start at the other end of a line. As we sort of mimick a scribe, we can be one who plays safe at the start of corrects at the end. In Figure 4 we add some randomness but to what extent this works well depends on how many words we need to retypeset before we get the badness of the line within the constraints.

Salient features of Arabic script

Before applying the above to Arabic script, let's discuss some salient aspects of the problem. As a cursive script, Arabic is extremely versatile and the scribal calligraphy tradition reflects that. Digital Arabic typography is only beginning to catch up with the possibilities afforded by the scribal tradition. Indeed, early lead-punch typography and typesetting of Arabic script was more advanced than most digital typography even up to this day. In any case, let us begin to organize some of that versatility into a taxonomy for typography purposes.

Coming back to the use of typefaces in electronic publishing: many of the new typographers receive their knowledge and information about the rules of typography from books, from computer magazines or the instruction manuals which they get with the purchase of a PC or software. There is not so much basic instruction, as of now, as there was in the old days, showing the differences between good and bad typographic design. Many people are just fascinated by their PC's tricks, and think that a widely-praised program, called up on the screen, will make everything automatic from now on.

normal

COMING BACK TO THE USE OF TYPEFACES IN ELECTRONIC PUBLISHING: MANY OF THE NEW TYPOGRAPHERS RECEIVE THEIR KNOWLEDGE AND INFORMATION ABOUT THE RULES OF TYPOGRAPHY FROM BOOKS, FROM COMPUTER MAGAZINES OR THE INSTRUCTION MANUALS WHICH THEY GET WITH THE PURCHASE OF A PC OR SOFTWARE. THERE IS NOT SO MUCH BASIC INSTRUCTION, AS OF NOW, AS THERE WAS IN THE OLD DAYS, SHOWING THE DIFFERENCES BETWEEN GOOD AND BAD TYPOGRAPHIC DESIGN. MANY PEOPLE ARE JUST FASCINATED BY THEIR PC'S TRICKS, AND THINK THAT A WIDELY-PRaised PROGRAM, CALLED UP ON THE GREEN SCREEN, WILL MAKE EVERYTHING automatic from now on.

solution

Figure 3: Solution c.

Coming back to the use of typefaces in electronic publishing: many of the new typographers receive their knowledge and information about the rules of typography from books, from computer magazines or the instruction manuals which they get with the purchase of a PC or software. There is not so much basic instruction, as of now, as there was in the old days, showing the differences between good and bad typographic design. Many people are just fascinated by their PC's tricks, and think that a widely-praised program, called up on the screen, will make everything automatic from now on.

normal

COMING BACK TO THE USE OF TYPEFACES IN ELECTRONIC PUBLISHING: MANY OF THE NEW TYPOGRAPHERS RECEIVE THEIR KNOWLEDGE AND INFORMATION ABOUT THE RULES OF TYPOGRAPHY FROM BOOKS, FROM COMPUTER MAGAZINES OR THE INSTRUCTION MANUALS WHICH THEY GET WITH THE PURCHASE OF A PC OR SOFTWARE. THERE IS NOT SO MUCH BASIC INSTRUCTION, AS OF NOW, AS THERE WAS IN THE OLD DAYS, SHOWING THE DIFFERENCES BETWEEN GOOD AND BAD TYPOGRAPHIC DESIGN. MANY PEOPLE ARE JUST FASCINATED BY THEIR PC'S TRICKS, AND THINK THAT A WIDELY-PRaised PROGRAM, CALLED UP ON THE GREEN SCREEN, WILL MAKE EVERYTHING automatic from now on.

solution

Figure 4: Solution d.

What's available?

We have to work within the following parameters:

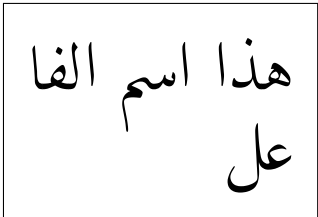
- **No hyphenation ever (well, almost never)**

It is commonly pointed out that there is no hyphenation in Arabic. This is something of a half-truth. In the manuscript tradition one actually does find something akin to hyphenation. In the ancient Kufic script, breaking a word across lines is actually quite common. But even in the more modern Naskh script, the one most normal Arabic text fonts are based on, it does occur, albeit rarely and presumably when the scribe is out of options for the line he is working on. Indeed, one could regard it as a failure on the part of the scribe once he reaches the end of the line.⁴

But there is still an important rule, regardless of whether we use Naskh, Kufic, or any other Arabic script. Consider the word below:



It is a single word composed of two cursive strings. One could actually hyphenate it, with our rule being to break it at the end of the first cursive string and before the beginning of the second cursive string:



Again, it's a rare phenomenon and hardly ever occurs in modern typesetting, lead-punch or digital, if at all. On the other hand, it could have some creative uses in future Arabic script typography.

- **Macrotypography (aesthetic features)**

In Arabic there are often numerous aesthetic ways of writing out the exact same semantic string:⁵

⁴Indeed, even Latin hyphenation, when it occurs, can be considered a “failure” of sorts.

⁵This five character string can be represented in Latin by the five character string “*al-hmd*” (not including the “-”). It is pronounced “*al-hamdu*”. Note that Arabic script is mainly consonantal: pure vowels are not part of the alphabet and are, instead, represented by diacritics.

الحمد الحمد الحمد

Normally we combine OPENTYPE features into feature sets that are each internally and aesthetically coherent. So in the above example we have used three different sets, reading from right to left. We'll call them simple, default, and dipped.

Just as Latin typography uses separate fonts to mark off different uses of text (bold, italic, etc.), an advanced Arabic font can use aesthetic feature sets to similar effect. This works best on distinguishing long streams of text from one another, since the differences between feature sets are not always noticeable on short strings. That is, two different aesthetic sets may type a given short string, such as a single word, exactly the same way. Consider the above three sets (simple, default, and dipped) once more:

علي علي علي

For the above string the default and dipped aesthetic sets (middle and left) give the exact same result, while the basic one (right) remains, well, quite basic.

Let's go back to our earlier example:

الحمد الحمد الحمد

Note that the simple version is wider than the default, and the dipped version is (slightly) thinner than the default. This relates to another point: An aesthetic feature set can serve two functions:

1. It can serve as the base aesthetic style.
2. It can serve as a resource for glyph substitution for a given string in another base aesthetic style.

This brings us back to our main topic.

- **Microtypography (paragraph optimization features)**

Here our job is to optimize the paragraph for even spacing and aesthetic viewing. It turns out that there are a number of ways to look at this issue, and we will begin exploring these in the next subsection.

Two approaches

Let us start off with a couple of samples. Qur’ānic transcription has always been the gold standard of Arabic script. Figure 5 we see a nice example of scribal optimization. The scribe here is operating under the constraint that each page ends with the end of a Qur’ānic verse (designated by the symbol U+06DD ﴿﴾). That is, no verse is broken across pages. That constraint, which is by no means mandatory or universal, gives the scribe lots of space for optimization, even more than normal.

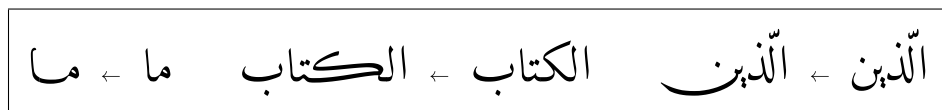
In Figure 6 we have a page of the famous *al-Husayni Muṣḥaf* of 1919–1923, which remains up to this day the only typeset copy of the Qur’ān to attain general acceptance in the Muslim world. Indeed, it remains the standard “edition” of the Qur’ān and even later scribal copies, such as the one featured in Figure 5 are based on its orthography. Unlike the scribal version, the typesetters of the *al-Husayni Muṣḥaf* did not try to constrain each page to end with the end of a Qur’ānic verse. Again, that is a nice feature to have as it makes recitation somewhat easier but it is by no means a mandatory one.

In any case, both samples share verses 172–176 in common, so there is lots to compare and contrast. We will also use these verses as our main textual sample for paragraph optimization.

Using figures 5 and 6 as benchmarks, we can begin by analyzing the approaches to paragraph optimization in Arabic script typography into two kinds:

- **Alternate glyphs**

Much of pre-digital Arabic typography uses this method. Generally, a wide variant of a letter is used to take up the space which would normally get absorbed by hyphenation in Latin. Here are examples of three of the most common substitutions, again, reading from right to left:



Each of the six strings above occurs in Figure 6. Identifying them is an exercise left to the reader. We call these kinds of alternate glyphs *alternate-shaped glyphs*.

The three substitutions above are the most common alternate-glyph sub-

وَإِذْ قِيلَ لَهُمُ اتَّبِعُوا مَا أَنْزَلَ اللَّهُ قَالُوا بَلْ نَتَّبِعُ مَا أَفْقَيْنَا
عَلَيْهِ ءَابَاءَنَا أَوْ لَوْ كَانَ ءَابَاؤُهُمْ لَا يَعْقِلُونَ شَيْئًا وَلَا
يَهْتَدُونَ ﴿١٧٠﴾ وَمَثَلُ الَّذِينَ كَفَرُوا كَمَثَلِ الَّذِي يَنْعِقُ
بِمَا لَا يَسْمَعُ إِلَّا دُعَاءً وَنِدَاءً صُمُّ بُكْمٌ عُمًى فَهُمْ لَا يَعْقِلُونَ
﴿١٧١﴾ يَا أَيُّهَا الَّذِينَ ءَامَنُوا كُلُوا مِن طَيِّبَاتِ مَا رَزَقْنَاكُمْ
وَأَشْكُرُوا لِلَّهِ إِن كُنتُمْ إِيَّاهُ تَعْبُدُونَ ﴿١٧٢﴾ إِنَّمَا حَرَّمَ
عَلَيْكُمْ الْمَيِّتَةَ وَالْدَّمَ وَلَحْمَ الْخِنْزِيرِ وَمَا أَهْلَ بِهِ لغيرِ
اللَّهِ فَمَن أَضْطَرَّ غَيْرُ بَاغٍ وَلَا عَادٍ فَلَا إِثْمَ عَلَيْهِ إِنَّ اللَّهَ
غَفُورٌ رَّحِيمٌ ﴿١٧٣﴾ إِن الَّذِينَ يَكْتُمُونَ مَا أَنْزَلَ اللَّهُ مِن
الْكِتَابِ وَيَشْتَرُونَ بِهِ ءِثْمًا قَلِيلًا أَوْ لَتِيكًا مَا يَأْكُلُونَ
فِي بُطُونِهِمْ إِلَّا النَّارَ وَلَا يُكَلِّمُهُمُ اللَّهُ يَوْمَ الْقِيَمَةِ
وَلَا يُزَكِّيهِمْ وَلَهُمْ عَذَابٌ أَلِيمٌ ﴿١٧٤﴾ أُولَٰئِكَ الَّذِينَ
أَشْتَرُوا الضَّلَالَةَ بِالْهُدَىٰ وَالْعَذَابَ بِالْمَغْفِرَةِ فَمَا
أَصْبَرَهُمْ عَلَى النَّارِ ﴿١٧٥﴾ ذَلِكَ بِأَنَّ اللَّهَ نَزَّلَ الْكِتَابَ بِالْحَقِّ
وَإِنَّ الَّذِينَ اخْتَلَفُوا فِي الْكِتَابِ لَفِي شِقَاقٍ بَعِيدٍ ﴿١٧٦﴾

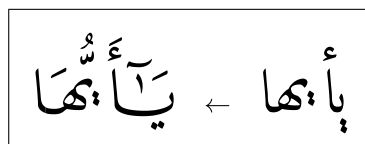
Figure 5: Scribal Optimization. Scribe: 'Uthmān Ṭāhā. Qur'ān, circa 1997.

صُمْ بُكُمْ عُمَىٰ فَهُمْ لَا يَعْقِلُونَ ﴿١٧١﴾ يٰٓأَيُّهَا الَّذِينَ ءَامَنُوا
 كُلُوا مِنْ طَيِّبَاتِ مَا رَزَقْنَاكُمْ وَاشْكُرُوا لِلّٰهِ إِنْ كُنْتُمْ إِيَّاهُ
 تَعْبُدُونَ ﴿١٧٢﴾ إِنَّمَا حَرَّمَ عَلَيْكُمُ الْمَيْتَةَ وَالدَّمَ وَلَحْمَ الْخَنَازِيرِ
 وَمَا أَهْلَ بِهِ ۚ لِغَيْرِ اللَّهِ ۖ فَمَنْ اضْطُرَّ غَيْرَ بَاغٍ وَلَا عَادٍ
 فَلَا إِثْمَ عَلَيْهِ ۚ إِنَّ اللَّهَ غَفُورٌ رَّحِيمٌ ﴿١٧٣﴾ إِنَّ الَّذِينَ
 يَكْتُمُونَ مَا أَنزَلَ اللَّهُ مِنْ الْكِتَابِ وَيَشْتَرُونَ بِهِ ۚ ثُمَّ
 قَلِيلًا ۙ أُولَٰئِكَ مَا يَأْكُلُونَ فِي بُطُونِهِمْ إِلَّا النَّارَ وَلَا يُكَلِّمُهُمُ
 اللَّهُ يَوْمَ الْقِيَمَةِ وَلَا يُزَكِّيهِمْ وَلَهُمْ عَذَابٌ أَلِيمٌ ﴿١٧٤﴾
 أُولَٰئِكَ الَّذِينَ اشْتَرُوا الضَّلَالَةَ بِالْهُدَىٰ وَالْعَذَابَ بِالْمَغْفِرَةِ
 ۚ فَمَا أَصْبَرَهُمْ عَلَى النَّارِ ﴿١٧٥﴾ ذَلِكَ بِأَنَّ اللَّهَ نَزَلَ الْكِتَابَ
 بِالْحَقِّ ۖ وَإِنَّ الَّذِينَ اخْتَلَفُوا فِي الْكِتَابِ لَفِي شِقَاقٍ
 بَعِيدٍ ﴿١٧٦﴾ * لَيْسَ الْبِرَّ أَنْ تُوَلُّوا وُجُوهَكُمْ قِبَلَ الْمَشْرِقِ

Figure 6: Scribal Optimization. Scribe: 'Uthmān Ṭahā. Qur'ān, circa 1997.

stitutions found in pre-digital Arabic script typography, including some contextual variants (initial, medial, final, and isolated) where appropriate. (The scribal tradition contains a lot more alternate-shaped glyphs. A few lead-punch fonts implement some of them, and we have implemented many of these in our Husayni font.) The results generally look quite nice and much more professional than most digital Arabic typography, which generally dispenses with these alternates.

But one also finds attempts at *extending* individual characters without changing the shape very much. One finds this already in Figure 6. We call these kinds of alternate glyphs *naturally curved widened* glyphs, or just *naturally widened* glyphs for short. Sometimes this is done for the purpose of making enough space for the vowels (which in Arabic take the form of diacritic characters). For example:



As you can see, there are two letters that have been *widened* for vowel accommodation. In Figure 6 there are some good but near-clumsy attempts at this. We say, “near-clumsy” because the typographers and typesetters mix natural, curved, *widened* variants of letters with flat, horizontal, *extended* versions. One reason for this is that a full repertoire of naturally curved glyph alternates would be much too unwieldy for even the best lead-punch typesetting machines and their operators. Even with these limitations one can find brave examples of lead-based typesetting that do a good job of sophisticated paragraph optimization via glyph alternates, both widened and alternate-shaped. Figure 7 is a representative example (in the context of columns).

Careful examination of this two-column sample will reveal the tension between naturally *widened* and horizontally *extended* glyphs in the execution of paragraph optimization. On the other hand, there is one apparent “rule” that one finds in this and other examples of lead-punch Arabic script typesetting:

Generally, there is only one naturally widened character per word or one alternate-shaped character per word.

In Figure 5 one can see that this “rule” is not always observed by scribes, see, e.g., the middle word in line 9 from the top, which uses two of the alternate-shaped characters we encountered above (can you identify that word?). But we still need some constraints for decent-looking typesetting,

وَقَرِئُ « وَرَجُلًا سَلَمًا » و (السَّلَامِيَّاتُ)	كَالسَّلِيلَةِ . وَشَيْءٌ (مَسْلَسٌ) مُتَّصِلٌ
بِفَتْحِ الْمِيمِ عِظَامُ الْأَصَابِعِ وَاحِدُهَا	بَعْضُهُ يَبْقَعُ وَمِنْهُ (سِلْسِلَةٌ) الْحَدِيدُ .
(سُلَامِيٌّ) وَهُوَ أَسْمٌ لِلوَاحِدِ وَالْجَمْعِ أَيْضًا .	* س ل م - (سَلَمٌ) أَسْمٌ رَجُلٍ
و (السَّلِيمُ) اللَّدِيغُ كَأَنَّهُمْ تَفَاءَلُوا لَهُ	و (سَلَمَانٌ) أَسْمٌ أَمْرَأَةٍ . و (سَلَمَانٌ)
بِالسَّلَامَةِ وَقِيلَ لِأَنَّهُ أُسْلِمَ لِمَا بِهِ . وَقَلْبٌ	أَسْمٌ جَبَلٍ وَأَسْمٌ رَجُلٍ . و (سَلِيمٌ) أَسْمٌ
سَلِيمٌ أَيْ سَلِيمٌ . و (سَلِيمٌ) فُلَانٌ مِنْ	رَجُلٍ . و (السَّلْمُ) بِفَتْحَتَيْنِ السَّلَفُ . وَالسَّلْمُ
الْآفَاتِ بِالْكَسْرِ (سَلَامَةٌ) و (سَلَمَهُ) اللَّهُ	أَيْضًا (الْإِسْتِسْلَامُ) . و (السَّلْمُ) أَيْضًا
مِنْهَا . و (سَلَمٌ) إِلَيْهِ الشَّيْءُ (قَسَلَمَهُ)	تَجَرَّ مِنْ الْعِضَاءِ الْوَاحِدَةِ سَلَمَةً . و (سَلَمَةٌ)
أَيْ أَخَذَهُ . و (التَّسْلِيمُ) بَذَلَ الرِّضَا	أَيْضًا أَسْمٌ رَجُلٍ . و (السَّلْمُ) بِفَتْحِ اللّامِ
بِالْحُكْمِ . وَالتَّسْلِيمُ أَيْضًا السَّلَامُ . و (أَسْلَمَ)	وَاحِدٌ (السَّلَالِيمُ) الَّتِي يُرْتَقَى عَلَيْهَا .
فِي الطَّعَامِ أَسْلَفَ فِيهِ . وَأَسْلَمَ أَمْرَهُ إِلَى اللَّهِ	و (السَّلْمُ) السَّلَامُ . وَقَرَأَ أَبُو عَمْرٍو :
أَيْ سَلَّمَ . وَأَسْلَمَ دَخَلَ فِي (السَّلْمِ) بِفَتْحَتَيْنِ	« أُدْخِلُوا فِي السَّلْمِ كَافَةً » وَذَهَبَ بِمَعْنَاهَا
وَهُوَ الْإِسْتِسْلَامُ و (أَسْلَمَ) مِنَ الْإِسْلَامِ .	إِلَى الْإِسْلَامِ . و (السَّلْمُ) الصُّلْحُ بِفَتْحِ
وَأَسْلَمَهُ خَذَلَهُ . و (التَّسْلَامُ) التَّصَالُحُ .	السَّيْنِ وَكَسَرُهَا يُذَكِّرُ وَيُؤَنِّثُ . وَالسَّلْمُ
و (المُسَالَمَةُ) الْمُصَالَحَةُ . و (أَسْتَلَمَ) الْحَجَرُ	الْمُسَالِمُ يَقُولُ أَنَا سِلْمٌ لِمَنْ سَالَنِي .
لَمَسَهُ إِمَّا بِالْقُبْلَةِ أَوْ بِالْيَدِ وَلَا يُهْمَزُ وَبَعْضُهُمْ	و (السَّلَامُ السَّلَامَةُ) . و (السَّلَامُ)
يَهْمِزُهُ . و (أَسْتَسْلَمَ) أَيْ أَتَقَادَ .	الْإِسْتِسْلَامُ . وَالسَّلَامُ الْكِسْمُ مِنَ التَّسْلِيمِ .
* س ل ا - (سَلَا) عَنْهُ مِنْ بَابِ سَمَا	وَالسَّلَامُ أَسْمٌ مِنْ أَشْيَاءِ اللَّهِ تَعَالَى .
و (سَلَى) عَنْهُ بِالْكَسْرِ (سُلْيَا) مِثْلُهُ .	وَالسَّلَامُ الْبَرَاءَةُ مِنَ الْعُيُوبِ فِي قَوْلِ أُمِّيَّةٍ .

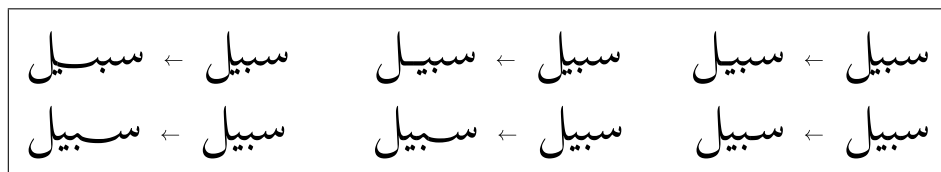
(١) ذكر في مادة - أ م ا - صفحة ٢٧ ، أنه لابد من تكرار « إتا »

Figure 7: Mixed Alternate Glyphs in Two Columns. From the classical dictionary *Mukhtār al-Şihāh*.

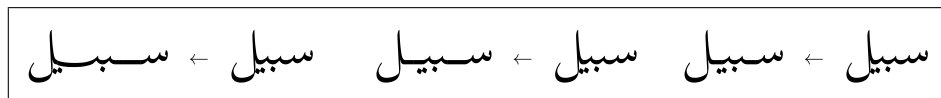
and the above tentative rule is a good place to start the analysis. For widened characters in particular we see that even the scribe (Figure 5) closely approximates this rule. So let's begin improving on our tentative rule somewhat, and expand it into a number of possibilities. Let's look at the naturally-widened-glyph case first:

Generally, there is only one naturally widened character allowed per word. However, two extended non-consecutive characters may be allowed. (*The logic of the experimental font Husayni already has constraints that prevent consecutive curved widened characters*).

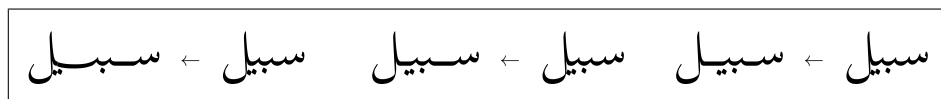
For example, we prefer to get widening like the following:



But as, e.g., a last resort or for stylistic purposes we can also do



Or even better, we mix it up a bit. That is, if there is more than one widened character, one should be longer than the other, e.g.:



One will notice that the middle substitution (where the first widened character is longer than the second) does not look as good as the two outer ones (where the second is longer than the first). These kinds of aesthetic issues can be formalized for future work. In the meantime, here is a working modified version of the rule for naturally-widened-glyphs:

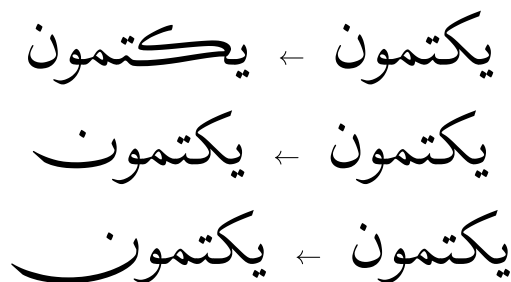
Generally, there is only one naturally widened character allowed per word. However, two non-consecutive widened characters may be allowed. In that case, the second widened character should be longer than the first.

One case where cases of two naturally widened character will be common is in poetry, which involves wide lines. We'll say more about this in the section on flat extending.

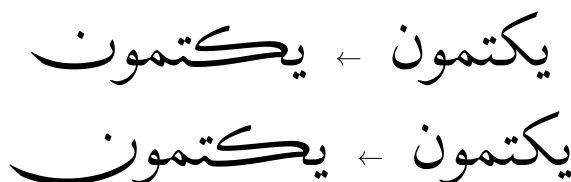
Now let's look at the alternate-shaped case:

Generally, there is only one alternate-shaped character allowed per word. However, two non-consecutive alternate-shaped characters may be allowed.

So we prefer, e.g.,



but we could have, e.g., as a last resort or as a stylistic option,



Again, in poetry this kind of multiple substitution within a single word could occur frequently. A challenge will be to develop a system of parameters where we can almost predict which kinds of substitution will happen under a given set of values of those parameters.

- **Flat extending**

In the transition from lead-punch to digital typography, alternate-glyph substitution largely vanished.⁶ The problem of spacing remained, and a

⁶Indeed, as was the case with Latin typography, Arabic script typography took a sharp turn for the worse with the advent of digital typography. On the other hand, Latin typography recovered much more quickly, in large part thanks to Knuth's development of T_EX.

فَإِنْ كَانَ الْقَوِيُّ الْوُجُودَ، إِظْمَأَّتِ النَّفْسُ وَكَانَتْ أُخْتُ الْعَقْلِ. وَرَقَّتِ الْمَاهِيَةُ وَ
شَابَهَتْ [٢٠٢] الْوُجُودَ، كَالْحَدِيدَةِ الْحَمَامَةِ فِي النَّارِ. فَلَا فَرْقَ فِي الْفِعْلِ بَيْنَهُمَا، وَإِنْ
كَانَ مَا بَيْنَهُمَا بِالْعَرَضِ، كَالْحَدِيدِ. قَالَ الشَّاعِرُ:

رَقَّ الزُّجَاجُ وَرَقَّتِ الْخُمُرُ فَتَشَابَكَا وَتَشَابَهَ الْأَمْرُ
فَكَأَنَّ خُمُرَ وَ لَا قَدَحَ وَ كَأَنَّ قَدَحَ وَ لَا خُمُرَ
وَ إِنْ كَانَ الْقَوِيُّ الْمَاهِيَةَ كَانَ الْأَمْرُ عَلَى الْعَكْسِ. وَ كُلُّ وَاحِدٍ مِنْهُمَا إِنَّمَا يَسْتَمِدُّ
وَ يَقْوِي بِمَدَدٍ مِنْ جَنْبِهِ إِذْ لَا يَسْتَمِدُّ الشَّيْءُ مِنْ نَحْوِ مَا هُوَ مِنْ ضِدِّهِ. فَلَا يَسْتَمِدُّ
النُّورُ مِنَ الظُّلْمَةِ وَ لَا الْعَكْسُ مِنْ حَيْثُ هُوَ كَذَلِكَ. وَ مِثْلُ الْآخِرِ مَعَهُ، إِنَّمَا هُوَ
لِبَقَائِهِمَا.

Figure 8: Poetry Justification in ArabTeX.

simple yet inelegant solution was adopted: flat, horizontal extending of characters. Now this solution did have *some* precedent in pre-digital Arabic typography, as you can see in Figure 6 and Figure 7. This solution had the advantage that it required only a single character: a simple horizontal bar called a *taṭwīl* or more commonly a *kashidah* (U+0640). This character could then be repeated as often as necessary to fill any extra space.

Now an examination of pre-digital books shows a (rather wise) reticence to using this method too slavishly. That reticence has now been thrown to the winds. This can be seen by looking at the standard implementation of flat extending as provided by Microsoft Word. This program provides three levels of extending that it calls “justification”. See Figure 9 for examples of all three. The minimum level is actually very close to the default (i.e., no-justification) level. Note that the sample text used in Figure 9 is the same as that used in the earlier samples from the Qur’ān.

Older implementations of Arabic script within T_EX, such as ArabT_EX and Omega/Aleph, also provided facilities for flat extending. The most common use was in poetry, which requires a fixed width for each stanza.

In Omega/Aleph, a method based on `\xleaders` was used, based on a very thin *taṭwīl* glyph (much thinner than U+0640) that could be used for very fine extending optimization based on T_EX’s badness parameter. One nice application is in marginal notes: See Figure 10, where the marginal note on the right is zoomed in. On the other hand, we see that the leaders method

يَا أَيُّهَا الَّذِينَ آمَنُوا كُلُوا مِن طَيِّبَاتِ مَا رَزَقْنَاكُمْ وَاشْكُرُوا لِلَّهِ إِن كُنتُمْ إِيَّاهُ تَعْبُدُونَ ۖ إِنَّمَا حَرَّمَ عَلَيْكُمُ الْمَيْتَةَ
وَالْدَّمَ وَخَمَ الْخَنِيزِ وَمَا أَهْلَ بِهِ لَعْنِ اللَّهِ ۖ فَمَن اضْطُرَّ غَيْرَ بَاغٍ وَلَا عَادٍ فَلَا إِثْمَ عَلَيْهِ ۚ إِنَّ اللَّهَ غَفُورٌ رَّحِيمٌ
۱۷۳ إِنَّ الَّذِينَ يَكْتُمُونَ مَا أَنزَلَ اللَّهُ مِنَ الْكِتَابِ وَيَشْتُرُونَ بِهِ ثَمَنًا قَلِيلًا ۖ أُولَٰئِكَ مَا يَأْكُلُونَ فِي بُطُونِهِمْ إِلَّا
النَّارَ وَلَا يَكْلَمُهُمُ اللَّهُ يَوْمَ الْقِيَامَةِ وَلَا يُزَكِّيهِمْ وَهُمْ عَذَابُ أَلِيمٍ ۖ ۱۷۴ أُولَٰئِكَ الَّذِينَ اشْتَرُوا الضَّلَالََةَ بِالْهَدَىٰ
وَالْعَذَابُ بِالْمَغْفِرَةِ ۚ فَمَا أَصْبَرَهُمْ عَلَى النَّارِ ۖ ۱۷۵ ذَلِكَ بِأَنَّ اللَّهَ نَزَلَ الْكِتَابَ بِالْحَقِّ ۚ وَإِنَّ الَّذِينَ اخْتَلَفُوا فِي
الْكِتَابِ لَفِي شِقَاقٍ بَعِيدٍ ۖ ۱۷۶

يَا أَيُّهَا الَّذِينَ آمَنُوا كُلُوا مِن طَيِّبَاتِ مَا رَزَقْنَاكُمْ وَاشْكُرُوا لِلَّهِ إِن كُنتُمْ إِيَّاهُ تَعْبُدُونَ ۖ إِنَّمَا حَرَّمَ
عَلَيْكُمُ الْمَيْتَةَ وَالْدَّمَ وَخَمَ الْخَنِيزِ وَمَا أَهْلَ بِهِ لَعْنِ اللَّهِ ۖ فَمَن اضْطُرَّ غَيْرَ بَاغٍ وَلَا عَادٍ فَلَا إِثْمَ
عَلَيْهِ ۚ إِنَّ اللَّهَ غَفُورٌ رَّحِيمٌ ۖ ۱۷۳ إِنَّ الَّذِينَ يَكْتُمُونَ مَا أَنزَلَ اللَّهُ مِنَ الْكِتَابِ وَيَشْتُرُونَ بِهِ ثَمَنًا
قَلِيلًا ۖ أُولَٰئِكَ مَا يَأْكُلُونَ فِي بُطُونِهِمْ إِلَّا النَّارَ وَلَا يَكْلَمُهُمُ اللَّهُ يَوْمَ الْقِيَامَةِ وَلَا يُزَكِّيهِمْ وَهُمْ عَذَابُ
أَلِيمٍ ۖ ۱۷۴ أُولَٰئِكَ الَّذِينَ اشْتَرُوا الضَّلَالََةَ بِالْهَدَىٰ وَالْعَذَابُ بِالْمَغْفِرَةِ ۚ فَمَا أَصْبَرَهُمْ عَلَى النَّارِ ۖ ۱۷۵
ذَلِكَ بِأَنَّ اللَّهَ نَزَلَ الْكِتَابَ بِالْحَقِّ ۚ وَإِنَّ الَّذِينَ اخْتَلَفُوا فِي الْكِتَابِ لَفِي شِقَاقٍ بَعِيدٍ ۖ ۱۷۶

يَا أَيُّهَا الَّذِينَ آمَنُوا كُلُوا مِن طَيِّبَاتِ مَا رَزَقْنَاكُمْ وَاشْكُرُوا لِلَّهِ إِن كُنتُمْ إِيَّاهُ تَعْبُدُونَ ۖ ۱۷۲
إِنَّمَا حَرَّمَ عَلَيْكُمُ الْمَيْتَةَ وَالْدَّمَ وَخَمَ الْخَنِيزِ وَمَا أَهْلَ بِهِ لَعْنِ اللَّهِ ۖ فَمَن اضْطُرَّ غَيْرَ بَاغٍ
وَلَا عَادٍ فَلَا إِثْمَ عَلَيْهِ ۚ إِنَّ اللَّهَ غَفُورٌ رَّحِيمٌ ۖ ۱۷۳ إِنَّ الَّذِينَ يَكْتُمُونَ مَا أَنزَلَ اللَّهُ مِنَ
الْكِتَابِ وَيَشْتُرُونَ بِهِ ثَمَنًا قَلِيلًا ۖ أُولَٰئِكَ مَا يَأْكُلُونَ فِي بُطُونِهِمْ إِلَّا النَّارَ وَلَا يَكْلَمُهُمُ اللَّهُ
يَوْمَ الْقِيَامَةِ وَلَا يُزَكِّيهِمْ وَهُمْ عَذَابُ أَلِيمٍ ۖ ۱۷۴ أُولَٰئِكَ الَّذِينَ اشْتَرُوا الضَّلَالََةَ بِالْهَدَىٰ
وَالْعَذَابُ بِالْمَغْفِرَةِ ۚ فَمَا أَصْبَرَهُمْ عَلَى النَّارِ ۖ ۱۷۵ ذَلِكَ بِأَنَّ اللَّهَ نَزَلَ الْكِتَابَ بِالْحَقِّ ۚ
وَإِنَّ الَّذِينَ اخْتَلَفُوا فِي الْكِتَابِ لَفِي شِقَاقٍ بَعِيدٍ ۖ ۱۷۶

Figure 9: Flat justification from Microsoft Word 2010.

<p>«الْمُقَدَّمَةُ» الْأُولَى :</p> <p>«هِيَ» أَنَّهُ لَا إِشْكَالَ فِي</p> <p>فِطْرَتِهِ - كَأَلِ الدِّينِ بِالْوَلَا</p> <p>تَحَقُّقًا وَوُجُودًا . وَ ذَلِكَ</p> <p>بِالْهَيْلَةِ الْبَسِيطَةِ . وَ كَأَلِ</p>	<p>فِي بَيَانِ مَرَاتِبِ مَعْرِفَةِ</p> <p>اللَّهِ تَعَالَى لِلسَّالِكِ إِلَيْهِ</p> <p>يُسْلُوكُهُ إِلَى مَقَامِ</p> <p>الْوَلَايَةِ ، عَلَى حَسَبِ</p> <p>الْأَطْوَارِ الْيَقِينِيَّةِ</p> <p>الْثَلَاثَةِ : عِلْمِ الْيَقِينِ ،</p> <p>عَيْنِ الْيَقِينِ ، وَ حَقِّ</p> <p>الْيَقِينِ .</p>
---	--

Figure 10: Marginal-note justification in Omega/Aleph.

creates extending that may be considered too perfectly even: Do we want to impose the rule that only one character should be extended per word (or at most two non-consecutive characters)? I have seen a lot of older digital Arabic typography that does even extending, including the poetry in the ArabTeX sample in figure 8. Compare this with the Microsoft Word method (Figure 9). The method used in Microsoft Word, with only one extension per word, seems to be the current standard for flat-extending justification.

On the other hand, the justification used in Microsoft Word is not particularly aesthetically pleasing. The answer will lie, again, in parameterization of some sort to be determined. As TeXies, we want to be able to have fine control over this kind of behavior in any case. In the meantime, we mirror the same rule we arrived at for naturally-widened-glyphs:

Generally, there is only one flat extended character allowed per word. However, two non-consecutive extended characters may be allowed. In that case, the second extended character should be longer than the first.

For example:

سبيل ← سبيل

سبيل ← سبيل

In accordance with our working rule, the top substitution uses only one flat extended character. The bottom uses two, but the second is longer than the first.

In our own estimation, the smaller the type, as in, e.g., footnotes and marginal notes, the less aesthetic variants that are needed. And the less aesthetic variants needed, the better that flat extending will work as a solution. Consider another example of the same word processed in three different variants:

The image shows three variants of the Arabic word 'الحمد' (Al-Hamd) written in a cursive script. The first variant on the left is a sophisticated aesthetic style. The middle variant is a basic, standard style. The third variant on the right is a basic style where the flat extension of the final letter is longer than the first, as per the working rule.

In this case our default is on the left. The variant on the right is about as basic as one can get; the default on the left is a sophisticated aesthetic variant. The middle one is, well, in between. Let's try them with flat extending, using only one extended character per word:

The image shows three variants of the Arabic word 'الحمد' (Al-Hamd) written in a cursive script. The first variant on the left is a sophisticated aesthetic style. The middle variant is a basic, standard style. The third variant on the right is a basic style where the flat extension of the final letter is longer than the first, as per the working rule.

On the left, we have an aesthetic combination of letters followed by a flat *tatwīl*. This is what Microsoft Word would give us, and the result is aesthetically distasteful. In the word on the right, however, the flat extension fits well with the basic nature of the feature set. As for the middle one, it could go either way and we leave it to the reader to decide what one thinks.

Now let's repeat with more naturally curved widening:

The image shows three variants of the Arabic word 'الحمد' (Al-Hamd) written in a cursive script. The first variant on the left is a sophisticated aesthetic style. The middle variant is a basic, standard style. The third variant on the right is a basic style where the flat extension of the final letter is longer than the first, as per the working rule.

Here, the variant on the left comes out much nicer. The one on the right looks okay with curved widening, although one could arguably do better with flat extending, at least in some contexts. The middle one, again, could go either way, though we think it does somewhat better with curved widening compared to the one on the right. The variant on the left only works well with curved widening.

Towards a ConT_EXt solution

In what follows, we will focus on a solution to the problem of paragraph optimization via alternate glyphs (including alternately-shaped and naturally-widened variants). It turns out that the `\xleaders` method used by Omega/Aleph does not work in L^AT_EX, so flat extending could not be naively implemented that way. At the moment flat extending is yet to be implemented in ConT_EXt.

Since flat extending is so ubiquitous in current Arabic script typography, and since it does have important applications (poetry and small font sizes where one prefers simpler aesthetic variants), one could ask why this was not implemented first. In part, this is because the immediate priority of the Oriental T_EX project has been top-notch, unparalleled aesthetic sophistication of the script. As we noted above, flat extending does not work so well with sophisticated aesthetic variation. So although the flat-extending problem is apparently simpler, it is understandable that we have focused on the more difficult problem first. A clear understanding of the issues and challenges involved with the more general alternate glyph method will help us implement a solution to the the flat-extended problem as a special case. We will come back to this issue towards the end.

Let us now consider the current experimental ConT_EXt setup for paragraph optimization for Arabic script.

Applying Features to Arabic script

We're now ready for the real thing: Arabic script. The initial setup is not that different from the Latin script case.

Applying Features to Arabic script

We're now ready for the real thing: Arabic script. The initial setup is not that different from the Latin script case.

```
\definefontfeature
  [husayni-whatever]
  [goodies=husayni,
   featureset=default]
\definefontsolution
  [FancyHusayni]
  [goodies=husayni,
   solution=experimental]
\definefont
  [FancyHusayni]
  [file:husayni*husayni-whatever at 24pt]
```

But here the definitions in the goodies file look way more complex. Here we have only one shrink set but multiple expansion sets.

```
local yes = "yes"
local basics = {
    analyze = yes,
    mode     = "node",
    language = "dflt",
    script   = "arab",
}
local analysis = {
    ccmp = yes,
    init = yes, medi = yes, fina = yes,
}
local regular = {
    rlig = yes, calt = yes, salt = yes, anum = yes,
    ss01 = yes, ss03 = yes, ss07 = yes, ss10 = yes, ss12 = yes,
    ss15 = yes, ss16 = yes, ss19 = yes, ss24 = yes, ss25 = yes,
    ss26 = yes, ss27 = yes, ss31 = yes, ss34 = yes, ss35 = yes,
    ss36 = yes, ss37 = yes, ss38 = yes, ss41 = yes, ss42 = yes,
    ss43 = yes, js16 = yes,
}
local positioning = {
    kern = yes, curs = yes, mark = yes, mkmk = yes,
}
local minimal_stretching = {
    js11 = yes, js03 = yes,
}
local medium_stretching = {
    js12=yes, js05=yes,
}
local maximal_stretching= {
    js13 = yes, js05 = yes, js09 = yes,
}
local wide_all = {
    js11 = yes, js12 = yes, js13 = yes, js05 = yes, js09 = yes,
}
local shrink = {
    flts = yes, js17 = yes, ss05 = yes, ss11 = yes, ss06 = yes,
    ss09 = yes,
}
local default = {
```

```

    basics, analysis, regular, positioning,
}
return {
    name = "husayni",
    version = "1.00",
    comment = "Goodies that complement the" ..
        "Husayni font by prof.Hamid.",
    author = "Idris Samawi Hamid and Hans Hagen",
    featuresets = {
        default = {
            default,
        },
        minimal_stretching = {
            default,
            js11 = yes, js03 = yes,
        },
        medium_stretching = {
            default,
            js12=yes, js05=yes,
        },
        maximal_stretching= {
            default,
            js13 = yes, js05 = yes, js09 = yes,
        },
        wide_all = {
            default,
            js11 = yes, js12 = yes, js13 = yes, js05 = yes,
            js09 = yes,
        },
        shrink = {
            default, flts = yes,
            js17 = yes,
            ss05 = yes, ss11 = yes, ss06 = yes, ss09 = yes,
        },
    },
    solutions = {
        experimental = {
            less = {
                "shrink",
            },
            more = {
                "minimal_stretching", "medium_stretching",

```

```

    "maximal_stretching", "wide_all"
  },
},
},
...
}

```

There are some 55 stylistic and 21 justification features. Not all make sense when optimizing. We predefine some LUA tables to make the sets and solutions easier to understand. The default rendering looks as follows:

```

\FancyHusayni
\righttoleft
\definefontfeature[rasm][script=arab,ss05=yes,js06=no,ss55=yes]
\addff{rasm}
\getbuffer[sample] \par

```

يَا أَيُّهَا الَّذِينَ ءَامَنُوا كُلُوا مِن طَيِّبَاتِ مَا رَزَقْنَاكُمْ وَاشْكُرُوا لِلّٰهِ إِن كُنتُمْ
إِيَّاهُ تَعْبُدُونَ ﴿١٧٢﴾ إِنَّمَا حَرَّمَ عَلَيْكُمُ الْمَيْتَةَ وَالدَّمَ وَلَحْمَ الْخِنْزِيرِ وَمَا أُهْلَ
بِهِ لِغَيْرِ اللَّهِ ۖ فَمَنِ اضْطُرَّ غَيْرَ بَاغٍ وَلَا عَادٍ فَلَا إِثْمَ عَلَيْهِ ۚ إِنَّ اللَّهَ غَفُورٌ
رَّحِيمٌ ﴿١٧٣﴾ إِنَّ الَّذِينَ يَكْتُمُونَ مَا أَنزَلَ اللَّهُ مِنْ الْكِتَابِ وَيَشْرُونَ بِهِ
ثَمَنًا قَلِيلًا ۖ أُولَٰئِكَ مَا يَأْكُلُونَ فِي بُطُونِهِمْ إِلَّا النَّارَ وَلَا يُكَلِّمُهُمُ اللَّهُ يَوْمَ
الْقِيَمَةِ وَلَا يَزْكِيهِمْ ۖ وَلَهُمْ عَذَابٌ أَلِيمٌ ﴿١٧٤﴾ أُولَٰئِكَ الَّذِينَ اشْتَرُوا الضَّلَالَهٗ
بِالْهُدَىٰ وَالْعَذَابِ بِالْمَغْفِرَةِ ۚ فَمَا أَصْبَرَهُمْ عَلَى النَّارِ ﴿١٧٥﴾ ذَٰلِكَ بِأَنَّ
اللَّهَ نَزَّلَ الْكِتَابَ بِالْحَقِّ ۖ وَإِنَّ الَّذِينَ اخْتَلَفُوا فِي الْكِتَابِ لَفِي شِقَاقٍ
بَعِيدٍ ﴿١٧٦﴾

Note that we already have a degree of widened substitution in this example. This is all for the accommodation of vowels, and is defined entirely in the OPEN-TYPE tables of the font. We also added some special orthography (the `rasm` font feature to get the Qur'anic features just right). You can also do this by adding the feature to the `lfg` file (`local regular = .`). There is no paragraph optimization as yet, although the default L^AT_EX engine does a good job to start with.

Next we show a more optimized result:

```
\setupfontsolution
[FancyHusayni]
[method={preroll,normal},
criterium=1]

\startfontsolution[FancyHusayni]
\FancyHusayni
\righttoleft
\definefontfeature[rasm] [script=arab,ss05=yes,js06=no,ss55=yes]
\addff{rasm}
\getbuffer[sample] \par
\stopfontsolution
```

يَا أَيُّهَا الَّذِينَ ءَامَنُوا كُلُوا مِن طَيِّبَاتِ مَا رَزَقْنَاكُمْ وَاشْكُرُوا لِلّٰهِ إِن كُنتُمْ
إِيَّاهُ تَعْبُدُونَ ﴿١٧٢﴾ إِنَّمَا حَرَّمَ عَلَيْكُمُ الْمَيْتَةَ وَالدَّمَ وَلَحْمَ الْخِنْزِيرِ وَمَا أُهْلَ
بِهِ لِغَيْرِ اللَّهِ ۖ فَمَنِ اضْطُرَّ غَيْرَ بَاغٍ وَلَا عَادٍ فَلَا إِثْمَ عَلَيْهِ ۚ إِنَّ اللَّهَ غَفُورٌ
رَّحِيمٌ ﴿١٧٣﴾ إِنَّ الَّذِينَ يَكْتُمُونَ مَا أَنزَلَ اللَّهُ مِنَ الْكِتَابِ وَيَشْرُونَ بِهِ
ثَمَنًا قَلِيلًا ۖ أُولَٰئِكَ مَا يَأْكُلُونَ فِي بُطُونِهِمْ إِلَّا النَّارَ وَلَا يُكَلِّمُهُمُ اللَّهُ يَوْمَ
الْقِيَامَةِ وَلَا يُزَكِّيهِمْ وَلَهُمْ عَذَابٌ أَلِيمٌ ﴿١٧٤﴾ أُولَٰئِكَ الَّذِينَ اشْتَرُوا الضَّلَالَةَ
بِالْهُدَىٰ وَالْعَذَابَ بِالْمَغْفِرَةِ ۚ فَمَا أَصْبَرَهُمْ عَلَى النَّارِ ﴿١٧٥﴾ ذَٰلِكَ بِأَنَّ
اللَّهَ تَزَّلَ الْكِتَابَ بِالْحَقِّ ۖ وَإِنَّ الَّذِينَ اخْتَلَفُوا فِي الْكِتَابِ لَفِي شِقَاقٍ
بَعِيدٍ ﴿١٧٦﴾

Now let's see what happens when `\parfillskip = 0pt`, i.e., the last line has no extra space after the end of the paragraph. This is important for getting, e.g., the last line of the page to end with the end of a verse as we discussed earlier:

```
\setupfontsolution
[FancyHusayni]
[method={preroll,normal},
criterium=1]
```

```

\startfontsolution[FancyHusayni]
  \FancyHusayni
  \righttoleft
\definefontfeature[rasm] [script=arab,ss05=yes,js06=no,ss55=yes]
\addff{rasm}
  \parfillskip=0pt
  \getbuffer[sample] \par
\stopfontsolution

```

يَا أَيُّهَا الَّذِينَ ءَامَنُوا كُلُوا مِن طَيِّبَاتِ مَا رَزَقْنَاكُمْ وَاشْكُرُوا لِلَّهِ إِن
كُنْتُمْ إِيَّاهُ تَعْبُدُونَ ﴿١٧١﴾ إِنَّمَا حَرَّمَ عَلَيْكُمُ الْمَيْتَةَ وَالْدَّمَ وَلَحْمَ الْخِنْزِيرِ
وَمَا أَهْلَ بِهِ لغيرِ اللَّهِ ۖ فَمَن أَضْطَرَّ غَيْرُ بَاغٍ وَلَا عَادٍ فَلَا إِثْمَ
عَلَيْهِ ۚ إِنَّ اللَّهَ غَفُورٌ رَّحِيمٌ ﴿١٧٢﴾ إِنَّ الَّذِينَ يَكْتُمُونَ مَا أَنزَلَ
اللَّهُ مِنَ الْكِتَابِ وَيُسْتَرُونَ بِهِ ۖ ثَمَنًا قَلِيلًا لَّا أُؤْتِيكَ مَا يَأْكُلُونَ فِي
بُطُونِهِمْ إِلَّا آلَ النَّارِ وَلَا يُكَلِّمُهُمُ اللَّهُ يَوْمَ الْقِيَمَةِ وَلَا يُزَكِّيهِمْ وَلَهُمْ
عَذَابٌ أَلِيمٌ ﴿١٧٣﴾ أُولَٰئِكَ الَّذِينَ أَشْرَوْا الضَّلَالَةَ بِأَهْدَىٰ وَالْعَذَابُ
بِالْمَغْفِرَةِ ۚ فَمَا أَصْبَرَهُمْ عَلَى النَّارِ ﴿١٧٤﴾ ذَٰلِكَ بِأَنَّ اللَّهَ نَزَلَ الْكِتَابَ
بِالْحَقِّ ۖ وَإِنَّ الَّذِينَ أَخْتَلَفُوا فِي الْكِتَابِ لَفِي شِقَاقٍ بَعِيدٍ ۝ ﴿١٧٥﴾

Just as the effects are more visible in the `\parfillskip = 0pt` case, the impact is much larger when the available width is less. In figures 11, 12, 13, 14, and 15 we can see the optimizer in action when that happens.

In our estimation, the current experimental solution works best for alternate-shaped glyphs, although there is some success with naturally widened characters. Clearly, some widened substitutions work better than others. A lot of fine tuning is needed, both within the OPENTYPE features as well as the optimization algorithm.

Without going into a detailed analysis at the moment, we restrict ourselves to two critical observations.

يَا أَيُّهَا الَّذِينَ ءَامَنُوا كُلُوا مِن
طَيِّبَاتِ مَا رَزَقْنَاكُمْ وَاشْكُرُوا لِلَّهِ إِن
كُنْتُمْ إِيَّاهُ تَعْبُدُونَ ﴿١٧٢﴾ إِنَّمَا حَرَّمَ
عَلَيْكُمُ الْمَيْتَةَ وَالْدَّمَ وَلَحْمَ الْخِنْزِيرِ وَمَا
أُهْلَ بِهِ لِغَيْرِ اللَّهِ ۖ فَمَنِ اضْطُرَّ غَيْرَ
بَاطِلٍ وَلَا عَادٍ فَلَا إِثْمَ عَلَيْهِ ۚ إِنَّ اللَّهَ
غَفُورٌ رَّحِيمٌ ﴿١٧٣﴾ إِنَّ الَّذِينَ يَكْتُمُونَ
مَا أَنزَلَ اللَّهُ مِنَ الْكِتَابِ وَيُسْرُونَ
بِهِ ۖ ثَمَنًا قَلِيلًا ۙ أُولَٰئِكَ مَا يَأْكُلُونَ
فِي بُطُونِهِمْ إِلَّا النَّارَ وَلَا يُكَلِّمُهُمُ اللَّهُ
يَوْمَ الْقِيَامَةِ وَلَا يُزَكِّيهِمْ وَلَهُمْ عَذَابٌ
أَلِيمٌ ﴿١٧٤﴾ أُولَٰئِكَ الَّذِينَ اشْتَرُوا
الضَّلَالََةَ بِالْهُدَىٰ وَالْعَذَابَ بِالْمَغْفِرَةِ
ۚ فَمَا أَصْبَرَهُمْ عَلَى النَّارِ ﴿١٧٥﴾ ذَٰلِكَ
بِأَنَّ اللَّهَ نَزَلَ الْكِتَابَ بِالْحَقِّ ۖ
وَإِنَّ الَّذِينَ اخْتَلَفُوا فِي الْكِتَابِ لَفِي
شِقَاقٍ بَعِيدٍ ﴿١٧٦﴾

narrow

يَا أَيُّهَا الَّذِينَ ءَامَنُوا كُلُوا مِن
طَيِّبَاتِ مَا رَزَقْنَاكُمْ وَاشْكُرُوا لِلَّهِ إِن
كُنْتُمْ إِيَّاهُ تَعْبُدُونَ ﴿١٧٢﴾ إِنَّمَا حَرَّمَ
عَلَيْكُمُ الْمَيْتَةَ وَالْدَّمَ وَلَحْمَ الْخِنْزِيرِ وَمَا
أُهْلَ بِهِ لِغَيْرِ اللَّهِ ۖ فَمَنِ اضْطُرَّ غَيْرَ
بَاطِلٍ وَلَا عَادٍ فَلَا إِثْمَ عَلَيْهِ ۚ إِنَّ اللَّهَ
غَفُورٌ رَّحِيمٌ ﴿١٧٣﴾ إِنَّ الَّذِينَ يَكْتُمُونَ
مَا أَنزَلَ اللَّهُ مِنَ الْكِتَابِ وَيُسْرُونَ
بِهِ ۖ ثَمَنًا قَلِيلًا ۙ أُولَٰئِكَ مَا يَأْكُلُونَ
فِي بُطُونِهِمْ إِلَّا النَّارَ وَلَا يُكَلِّمُهُمُ اللَّهُ
يَوْمَ الْقِيَامَةِ وَلَا يُزَكِّيهِمْ وَلَهُمْ عَذَابٌ
أَلِيمٌ ﴿١٧٤﴾ أُولَٰئِكَ الَّذِينَ اشْتَرُوا
الضَّلَالََةَ بِالْهُدَىٰ وَالْعَذَابَ بِالْمَغْفِرَةِ
ۚ فَمَا أَصْبَرَهُمْ عَلَى النَّارِ ﴿١٧٥﴾ ذَٰلِكَ
بِأَنَّ اللَّهَ نَزَلَ الْكِتَابَ بِالْحَقِّ ۖ
وَإِنَّ الَّذِينَ اخْتَلَفُوا فِي الْكِتَابِ لَفِي
شِقَاقٍ بَعِيدٍ ﴿١٧٦﴾

normal

Figure 11: A narrower sample (a).

يَا أَيُّهَا الَّذِينَ ءَامَنُوا كُلُوا مِن
طَيِّبَاتِ مَا رَزَقْنَاكُمْ وَاشْكُرُوا لِلَّهِ إِن
كُنْتُمْ إِيَّاهُ تَعْبُدُونَ ﴿١٧٢﴾ إِنَّمَا حَرَّمَ
عَلَيْكُمُ الْمَيْتَةَ وَالْدَّمَ وَلَحْمَ الْخِنْزِيرِ وَمَا
أُهِلَّ بِهِ لِغَيْرِ اللَّهِ ^ط فَمَنِ اضْطُرَّ غَيْرَ
بَاغٍ وَلَا عَادٍ فَلَا إِثْمَ عَلَيْهِ ^ج إِنَّ اللَّهَ
غَفُورٌ رَّحِيمٌ ﴿١٧٣﴾ إِنَّ الَّذِينَ يَكْتُمُونَ
مَا أَنزَلَ اللَّهُ مِنَ الْكِتَابِ وَيُسْرُونَ
بِهِ ^ب ثَمَنًا قَلِيلًا ^ل أُولَٰئِكَ مَا يَأْكُلُونَ
فِي بُطُونِهِمْ إِلَّا النَّارَ وَلَا يُكَلِّمُهُمُ اللَّهُ
يَوْمَ الْقِيَمَةِ وَلَا يُزَكِّيهِمْ وَلَهُمْ عَذَابٌ
أَلِيمٌ ﴿١٧٤﴾ أُولَٰئِكَ الَّذِينَ أَسْرَوْا
الْضَّلَالَةَ بِالْهُدَىٰ وَالْعَذَابُ بِالْمُغْفِرَةِ
^ج فَمَا أَصْبَرَهُمْ عَلَى النَّارِ ﴿١٧٥﴾
ذَٰلِكَ بِأَنَّ اللَّهَ نَزَلَ الْكِتَابَ
بِالْحَقِّ ^ط وَإِنَّ الَّذِينَ اخْتَلَفُوا فِي
الْكِتَابِ لَفِي شِقَاقٍ بَعِيدٍ ^م ﴿١٧٦﴾

narrow

يَا أَيُّهَا الَّذِينَ ءَامَنُوا كُلُوا مِن
طَيِّبَاتِ مَا رَزَقْنَاكُمْ وَاشْكُرُوا لِلَّهِ إِن
كُنْتُمْ إِيَّاهُ تَعْبُدُونَ ﴿١٧٢﴾ إِنَّمَا حَرَّمَ
عَلَيْكُمُ الْمَيْتَةَ وَالْدَّمَ وَلَحْمَ الْخِنْزِيرِ وَمَا
أُهِلَّ بِهِ لِغَيْرِ اللَّهِ ^ط فَمَنِ اضْطُرَّ غَيْرَ
بَاغٍ وَلَا عَادٍ فَلَا إِثْمَ عَلَيْهِ ^ج إِنَّ اللَّهَ
غَفُورٌ رَّحِيمٌ ﴿١٧٣﴾ إِنَّ الَّذِينَ يَكْتُمُونَ
مَا أَنزَلَ اللَّهُ مِنَ الْكِتَابِ وَيُسْرُونَ
بِهِ ^ب ثَمَنًا قَلِيلًا ^ل أُولَٰئِكَ مَا يَأْكُلُونَ
فِي بُطُونِهِمْ إِلَّا النَّارَ وَلَا يُكَلِّمُهُمُ اللَّهُ
يَوْمَ الْقِيَمَةِ وَلَا يُزَكِّيهِمْ وَلَهُمْ عَذَابٌ
أَلِيمٌ ﴿١٧٤﴾ أُولَٰئِكَ الَّذِينَ أَسْرَوْا
الْضَّلَالَةَ بِالْهُدَىٰ وَالْعَذَابُ بِالْمُغْفِرَةِ
^ج فَمَا أَصْبَرَهُمْ عَلَى النَّارِ ﴿١٧٥﴾
ذَٰلِكَ بِأَنَّ اللَّهَ نَزَلَ الْكِتَابَ بِالْحَقِّ ^ط
وَإِنَّ الَّذِينَ اخْتَلَفُوا فِي الْكِتَابِ لَفِي
شِقَاقٍ بَعِيدٍ ^م ﴿١٧٦﴾

normal

Figure 12: A narrower sample with no parfillskip (b).

يَا أَيُّهَا الَّذِينَ ءَامَنُوا كُلُوا مِن
طَيِّبَاتِ مَا رَزَقْنَاكُمْ وَاشْكُرُوا لِلَّهِ
إِن كُنْتُمْ إِبَّاهُ تَعْبُدُونَ ﴿١٧٦﴾ إِنَّمَا
حَرَّمَ عَلَيْكُمُ الْمَيْتَةَ وَالْدَّمَ وَلَحْمَ
الْخِنْزِيرِ وَمَا أُهْلَ بِهِ لِغَيْرِ اللَّهِ
فَمَنِ اضْطُرَّ غَيْرَ بَاغٍ وَلَا عَادٍ
فَلَا إِثْمَ عَلَيْهِ ۚ إِنَّ اللَّهَ غَفُورٌ
رَّحِيمٌ ﴿١٧٧﴾ إِنَّا الَّذِينَ يَكْتُمُونَ مَا
أَنزَلَ اللَّهُ مِنَ الْكِتَابِ وَيُسْرُونَ
بِهِ ثَمَنًا قَلِيلًا ۖ أُولَٰئِكَ مَا
يَأْكُلُونَ فِي بُطُونِهِمْ إِلَّا النَّارَ وَلَا
يُكَلِّمُهُمُ اللَّهُ يَوْمَ الْقِيَمَةِ وَلَا
يُزَكِّيهِمْ وَلَهُمْ عَذَابٌ أَلِيمٌ ﴿١٧٨﴾
أُولَٰئِكَ الَّذِينَ أَشْرَوْا الضَّلَالَةَ
بِأَهْدَىٰ وَالْعَذَابَ بِالْمَغْفِرَةِ ۚ
فَمَا أَصْبَرَهُمْ عَلَى النَّارِ ﴿١٧٩﴾
ذَٰلِكَ بِأَنَّ اللَّهَ نَزَلَ الْكِتَابَ
بِالْحَقِّ ۖ وَإِنَّ الَّذِينَ اخْتَلَفُوا فِي
الْكِتَابِ لَفِي شِقَاقٍ بَعِيدٍ ﴿١٨٠﴾

normal

يَا أَيُّهَا الَّذِينَ ءَامَنُوا كُلُوا مِن
طَيِّبَاتِ مَا رَزَقْنَاكُمْ وَاشْكُرُوا لِلَّهِ
إِن كُنْتُمْ إِبَّاهُ تَعْبُدُونَ ﴿١٧٦﴾ إِنَّمَا
حَرَّمَ عَلَيْكُمُ الْمَيْتَةَ وَالْدَّمَ وَلَحْمَ
الْخِنْزِيرِ وَمَا أُهْلَ بِهِ لِغَيْرِ اللَّهِ
فَمَنِ اضْطُرَّ غَيْرَ بَاغٍ وَلَا عَادٍ
فَلَا إِثْمَ عَلَيْهِ ۚ إِنَّ اللَّهَ غَفُورٌ
رَّحِيمٌ ﴿١٧٧﴾ إِنَّا الَّذِينَ يَكْتُمُونَ مَا
أَنزَلَ اللَّهُ مِنَ الْكِتَابِ وَيُسْرُونَ
بِهِ ثَمَنًا قَلِيلًا ۖ أُولَٰئِكَ مَا
يَأْكُلُونَ فِي بُطُونِهِمْ إِلَّا النَّارَ وَلَا
يُكَلِّمُهُمُ اللَّهُ يَوْمَ الْقِيَمَةِ وَلَا
يُزَكِّيهِمْ وَلَهُمْ عَذَابٌ أَلِيمٌ ﴿١٧٨﴾
أُولَٰئِكَ الَّذِينَ أَشْرَوْا الضَّلَالَةَ
بِأَهْدَىٰ وَالْعَذَابَ بِالْمَغْفِرَةِ ۚ
فَمَا أَصْبَرَهُمْ عَلَى النَّارِ ﴿١٧٩﴾
ذَٰلِكَ بِأَنَّ اللَّهَ نَزَلَ الْكِتَابَ
بِالْحَقِّ ۖ وَإِنَّ الَّذِينَ اخْتَلَفُوا فِي
الْكِتَابِ لَفِي شِقَاقٍ بَعِيدٍ ﴿١٨٠﴾

narrow

Figure 13: An even narrower sample (c).

يَا أَيُّهَا الَّذِينَ ءَامَنُوا
كُلُوا مِنْ طَيِّبَاتِ مَا رَزَقْنَاكُمْ
وَأَشْكُرُوا لِلَّهِ إِنْ كُنْتُمْ
تَعْبُدُونَ ﴿١٧٢﴾ إِنَّمَا حَرَّمَ
عَلَيْكُمْ الْمَيْتَةَ وَالْدَّمَ وَلَحْمَ
الْخِنْزِيرِ وَمَا أَهْلَ بِهِ لغيرِ
ٱللَّهِ ۖ فَمَنْ أَضْطَرَّ غَيْرُ
بَآغٍ وَلَا عَادٍ فَلَا إِثْمَ عَلَيْهِ ۚ
إِنَّ ٱللَّهَ غَفُورٌ رَّحِيمٌ ﴿١٧٣﴾
إِنَّ الَّذِينَ يَكْتُمُونَ مَا أَنزَلَ
ٱللَّهُ مِنَ الْكِتَابِ وَيُسْرَتُونَ
بِهِ ۖ ثَمَنًا قَلِيلًا ۖ أُولَٰئِكَ مَا
يَأْكُلُونَ فِي بُطُونِهِمْ إِلَّا النَّارَ
وَلَا يَكْتُمُهُمُ ٱللَّهُ يَوْمَ الْقِيَمَةِ
وَلَا يُزَكِّيهِمْ وَلَهُمْ عَذَابٌ
أَلِيمٌ ﴿١٧٤﴾ أُولَٰئِكَ الَّذِينَ
أَشْرَوْا الصَّلَاةَ بِأَهْدَىٰ ۖ وَالْعَذَابُ
بِالْغَفْرِ ۚ فَمَا أَصْبَرَهُمْ عَلَى
النَّارِ ﴿١٧٥﴾ ذَٰلِكَ بِأَنَّ ٱللَّهَ
نَزَلَ ٱلْكِتَابَ بِٱلْحَقِّ ۖ وَإِنَّ
ٱلَّذِينَ أَحْتَلَفُوا فِي ٱلْكِتَابِ
لَنفَىٰ شِقَاقِ بَعِيدٍ ﴿١٧٦﴾

normal

يَا أَيُّهَا الَّذِينَ ءَامَنُوا كُلُوا
مِنْ طَيِّبَاتِ مَا رَزَقْنَاكُمْ
وَأَشْكُرُوا لِلَّهِ إِنْ كُنْتُمْ
تَعْبُدُونَ ﴿١٧٢﴾ إِنَّمَا حَرَّمَ
عَلَيْكُمْ الْمَيْتَةَ وَالْدَّمَ وَلَحْمَ
الْخِنْزِيرِ وَمَا أَهْلَ بِهِ لغيرِ
ٱللَّهِ ۖ فَمَنْ أَضْطَرَّ غَيْرُ
بَآغٍ وَلَا عَادٍ فَلَا إِثْمَ عَلَيْهِ ۚ
إِنَّ ٱللَّهَ غَفُورٌ رَّحِيمٌ ﴿١٧٣﴾
إِنَّ الَّذِينَ يَكْتُمُونَ مَا أَنزَلَ
ٱللَّهُ مِنَ الْكِتَابِ وَيُسْرَتُونَ
بِهِ ۖ ثَمَنًا قَلِيلًا ۖ أُولَٰئِكَ مَا
يَأْكُلُونَ فِي بُطُونِهِمْ إِلَّا النَّارَ
وَلَا يَكْتُمُهُمُ ٱللَّهُ يَوْمَ الْقِيَمَةِ
وَلَا يُزَكِّيهِمْ وَلَهُمْ عَذَابٌ
أَلِيمٌ ﴿١٧٤﴾ أُولَٰئِكَ الَّذِينَ
أَشْرَوْا الصَّلَاةَ بِأَهْدَىٰ ۖ وَالْعَذَابُ
بِالْغَفْرِ ۚ فَمَا أَصْبَرَهُمْ عَلَى
النَّارِ ﴿١٧٥﴾ ذَٰلِكَ بِأَنَّ ٱللَّهَ
نَزَلَ ٱلْكِتَابَ بِٱلْحَقِّ ۖ وَإِنَّ
ٱلَّذِينَ أَحْتَلَفُوا فِي ٱلْكِتَابِ
لَنفَىٰ شِقَاقِ بَعِيدٍ ﴿١٧٦﴾

narrow

Figure 14: An even narrower sample (d).

يَا أَيُّهَا الَّذِينَ ءَامَنُوا
كُلُوا مِنْ طَيِّبَاتِ مَا
رَزَقْنَاكُمْ وَاشْكُرُوا لِلَّهِ إِنْ
كُنْتُمْ تَعْبُدُونَ ﴿١٧١﴾
إِنَّمَا حَرَّمَ عَلَيْكُمُ الْمَيْتَةَ
وَالْدَّمَ وَحُمَ الْجَنَازُ وَمَا
أَهْلٌ بِهِ يَعْتَرِ اللَّهُ
فَمَنْ أَضْطَرَّ غَيْرَ بِأَعْدٍ
فَلَا إِثْمَ عَلَيْهِ إِنْ
أَلَّهِ غَفُورٌ رَحِيمٌ ﴿١٧٢﴾
إِنَّ الَّذِينَ يَكْتُمُونَ مَا
أَنْزَلَ اللَّهُ مِنْ الْكِتَابِ
وَيُسْتَرُونَ بِهِ ثَمَنًا قَلِيلًا
أُولَئِكَ مَا يَأْكُلُونَ فِي
بُطُونِهِمْ إِلَّا النَّارَ وَلَا
يَكَلِّهِمْ اللَّهُ يَوْمَ الْقِيَمَةِ
وَلَا يُزَكِّيهِمْ وَهُمْ عَذَابُ
الْإِمْ ﴿١٧٣﴾ أُولَئِكَ الَّذِينَ
أَشْرَوْا الصَّلَاةَ بِالْهَدْيِ
وَالْعَذَابُ بِالْغَفْرِ فَآ
أَصْبَرَهُمْ عَلَى النَّارِ ﴿١٧٤﴾
ذَلِكَ بِأَنَّ اللَّهَ نَزَلَ الْكِتَابَ
بِالْحَقِّ وَإِنَّ الَّذِينَ
أَخْتَلَفُوا فِي الْكِتَابِ لَفِي
شِقَاقٍ بَعِيدٍ ﴿١٧٥﴾

normal

يَا أَيُّهَا الَّذِينَ ءَامَنُوا كُلُوا
مِنْ طَيِّبَاتِ مَا رَزَقْنَاكُمْ
وَاشْكُرُوا لِلَّهِ إِنْ كُنْتُمْ
إِلَّاهُ تَعْبُدُونَ ﴿١٧١﴾
إِنَّمَا حَرَّمَ عَلَيْكُمُ الْمَيْتَةَ وَالْدَّمَ
وَحُمَ الْجَنَازُ وَمَا أَهْلٌ
بِهِ يَعْتَرِ اللَّهُ فَمَنْ
أَضْطَرَّ غَيْرَ بِأَعْدٍ وَلَا عَادٍ
فَلَا إِثْمَ عَلَيْهِ إِنْ
أَلَّهِ غَفُورٌ رَحِيمٌ ﴿١٧٢﴾
إِنَّ الَّذِينَ يَكْتُمُونَ مَا
أَنْزَلَ اللَّهُ مِنْ الْكِتَابِ
وَيُسْتَرُونَ بِهِ ثَمَنًا قَلِيلًا
أُولَئِكَ مَا يَأْكُلُونَ فِي
بُطُونِهِمْ إِلَّا النَّارَ وَلَا
يَكَلِّهِمْ اللَّهُ يَوْمَ الْقِيَمَةِ
وَلَا يُزَكِّيهِمْ وَهُمْ عَذَابُ
الْإِمْ ﴿١٧٣﴾ أُولَئِكَ الَّذِينَ
أَشْرَوْا الصَّلَاةَ بِالْهَدْيِ
وَالْعَذَابُ بِالْغَفْرِ فَآ
أَصْبَرَهُمْ عَلَى النَّارِ ﴿١٧٤﴾
ذَلِكَ بِأَنَّ اللَّهَ نَزَلَ الْكِتَابَ
بِالْحَقِّ وَإِنَّ الَّذِينَ
أَخْتَلَفُوا فِي الْكِتَابِ لَفِي
شِقَاقٍ بَعِيدٍ ﴿١٧٥﴾

narrow

Figure 15: An even narrower sample (e).

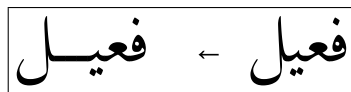
First, in our tests one will notice that the glyph substitutions tend to take place on the right side of the line. They should be more evenly distributed throughout each line.

Second, we can say that the current method works better for alternate-shaped glyph substitution than it does for naturally-widened glyph substitution. This leads us to the next step in this research project:

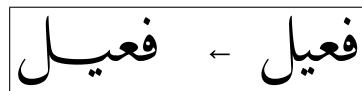
Within the Husayni font there is now a mapping between flat extending via *taṭwīl* and curved widening via alternate glyphs. Consider the following manually typed UTF text using the *taṭwīl* character (U+0640):

فعيل \ARROW\ فعيل

In flat-extended typography that comes out like this:



Husayni, through the optional Stylistic Alternates feature (`salt`) will map the flat *taṭwīl*-extended characters to curved widened characters. So with `salt=yes` selected in `CONTEX`T we get



This opens up a way to connect a forthcoming solution to the flat *taṭwīl*-extended character method with the curved widened-glyph method. A future version of the optimizer may be able to optimize the paragraph in terms of the *taṭwīl* character and a set of rules along the lines we discussed earlier. Then we can simply convert the result to curves using the *taṭwīl* character. At least this is one possibility.

The second author is currently working on an updated version of the Husayni font. The updates include improved character shapes and possibly the use of the `OPENTYPE` variable font mechanism to provide a bold typeface.

Conclusion

In this article, we introduced the `CONTEX`T paragraph optimizer and showed how it can be used in conjunction with a sophisticated `OPENTYPE` font for Arabic script typesetting. We argue that the current paragraph optimizer, even in its experimental status at the moment, represents one of the greatest and most important steps in the evolution of digital Arabic script typography. Its potential

impact on for Arabic script typesetting is immense, and we excitedly look forward to its completion.

Orientálský T_EX: Optimalizace odstavcového zlomu

Článek popisuje systém pro optimalizaci odstavcového zlomu při sazbě arabštiny ve formátu CON_TE_XT. Implementace je představena na úryvcích v latině. Článek následně popisuje základní vlastnosti arabského písma a známé postupy při optimalizaci odstavcového zlomu. V závěru článku je jeden z popsanych postupů využit při sazbě pasáže z Koránu.

Keywords: mikrotypografie, OPENTYPE, CON_TE_XT, LUAT_EX

Since the version 1.80x, MetaPost has a third output backend: it is now possible to generate PNG bitmaps directly from within MetaPost.

Keywords: MetaPost, PNG.

Introduction

For one of my presentations at EuroT_EX2012 in Breskens, I wanted to create an animation in order to demonstrate a MetaPost macro that uses timer variables to progress through a scene.

While working on that presentation, it quickly became obvious that the ‘traditional’ method of creating an animation with MetaPost by using ImageMagick’s `convert` to turn EPS images into PNG images was very time consuming. So much so, that I actually managed to write a new backend for MetaPost while waiting for ImageMagick to complete the conversion.

Simple usage

MetaPost will create a PNG image (instead of Encapsulated PostScript or Scalable Vector Graphics) by setting `outputformat` to the string `png`:

```
outputformat := "png";
outputtemplate := "%j-%c.%o";
beginfig(1);
fill fullcircle scaled 100 withcolor red;
endfig;
end.
```

This input generates a bitmap file with dimensions 100×100 pixels, with 8-bit/color RGBA. It shows a red dot on a transparent background.

Adjusting the bitmap size

In the simple example given above, MetaPost has used the default conversion ratio where one point equals one pixel. This is not always desired, and it is tedious to have to scale the picture whenever a different output size is required.

To allow easy modification of the bitmap size independent of the actual graphic, two new internal parameters have been added: `hppp` and `vppp` (the names come from Metafont, but the actual meaning is specific to MetaPost).

In MetaPost, ‘`hppp`’ stands for ‘horizontal points per pixel’, and similarly for ‘`vppp`’. Adding

```
hppp := 2.0;
```

to the example above changes the bitmap into 50×100 pixels. Specifying values less than 1.0 (but above zero!) makes the bitmap larger.

Adjusting the output options

MetaPost creates a 32-bit RGBA bitmap image, unless the user alters the value of another new internal parameter: `outputformatoptions`.

The syntax for `outputformatoptions` is a space-separated list of settings. Individual settings are *keyword* `++` *value*. The only currently allowed ones are:

```
format=[rgba|rgb|graya|gray]
```

```
antialias=[none|fast|good|best]
```

No spaces are allowed on the left, nor on the right, of the equals sign inside a setting.

The assignment that would match the compiled-in default setup is:

```
outputformatoptions := "format=rgba antialias=fast";
```

however, `outputformatoptions` is initially the empty string, because that makes it easier to test whether a user-driven change has already been made.

Some notes on the different PNG output formats:

- The `rgb` and `gray` subformats have a white background. The `rgba` and `graya` subformats have a transparent background.
- The bitdepth is always 8 bits per pixel component.
- In all cases, the current picture is initially created in 8-bit RGB mode. For the `gray` and `graya` subformats, the RGB colors are reduced just before the actual PNG file is written, using a standard rule:

$$g = 0.2126 * r + 0.7152 * g + 0.0722 * b$$

- CMYK colors are always converted to RGB during generation of the output image using:

$$r = 1 - (c + k > 1 ? 1 : c + k);$$

$$g = 1 - (m + k > 1 ? 1 : m + k);$$

$$b = 1 - (y + k > 1 ? 1 : y + k);$$

If you care about color conversions, you should be doing a `within <pic>` loop inside `extra_endfig`. The built-in conversions are more of a fallback.

What you should also know

MetaPost uses cairo (<http://cairographics.org>) to do the bitmap creation, and then uses libpng (<http://www.libpng.org>) to create the actual file.

Any **prologues** setting is always ignored: the internal equivalent of the **glyph of** operator is used to draw characters onto the bitmap directly.

If there are points in the current picture with negative coordinates, then the whole picture is shifted upwards to prevent things from falling outside the generated bitmap.

Summary: MetaPost: PNG Output

Od verze 1.80x má MetaPost třetí možný výstupní formát obrázků. Nyní je možné generovat obrázek ve formátu PNG přímo v MetaPostu.

Klíčová slova: MetaPost, PNG.

Mapy v L^AT_EXových dokumentoch – predstavenie balíčka `getmap`

ALEŠ KOZUBÍK

Cieľom príspevku je predstavenie balíčka `getmap`. Tento balíček umožňuje do L^AT_EXových dokumentov zaradiť mapové materiály získané z externých zdrojov, ako sú OpenStreetMap alebo Google Maps a to aj s podporou Google Street View. V najjednoduchšom prípade pritom postačí aj špecifikácia požadovanej adresy. Balíček pre sťahovanie máp používa externý Lua skript, ktorý si vyžaduje aktiváciu funkcie `\write18`. Tento skript môže byť použitý aj samostatne z príkazového riadku.

Kľúčové slová: Mapy, L^AT_EX, `getmap`, Lua

Úvod

Poznáme to všetci – občas treba do dokumentu zaradiť obrázku, ktorých obsahom je mapa určitej oblasti alebo lokality, prípadne aj s vyznačením trasy. Niektorí sa s týmto problémom stretávajú častejšie (ak sa napríklad venuje geografii alebo pripravuje materiály pre cestovateľov), iní zriedkavejšie, napríklad pri príprave pozvánok a pokynov pre konferencie. Ale určite sa s touto úlohou stretol každý. My si v tomto príspevku predstavíme užitočný balíček `getmap`, ktorý umožňuje priamo do L^AT_EXového dokumentu zakomponovať obrázky z takých zdrojov, ako sú OpenStreetMap alebo Google Maps včítane vkladania obrázkov z Google Street View.

Podstata činnosti balíčka `getmap`, ktorého autorom je Josef Kleber, je pomerne jednoduchá. Balíček vlastne obsahuje jeden jediný príkaz `\getmap`, ktorý prostredníctvom jednoduchého Lua skriptu zabezpečí stiahnutie požadovaného obrázku, teda mapy alebo fotografie z Google Street View, do špecifikovaného grafického súboru. Takto získaný súbor potom vložíme do L^AT_EXového dokumentu pomocou `\includegraphics`. Zložitejšie ako samotný príkaz sú teda jeho voliteľné argumenty, ktoré špecifikujú požadovaný obrázok. V článku sa teda budeme venovať týmto voliteľným položkám.

Na správnu činnosť príkazu `\getmap` je potrebné aktivovať funkciu `\write18`, čo pri použití T_EXLive znamená kompilovanie s prepínačom `--shell-escape` resp. pri použití MiKTeXu `--enable-write18`. Nakoľko mapy vkladáme do textu ako obrázky, je taktiež potrebné načítať balíček `graphicx`.



Obrázok 1: Ukážka výstupu získaného pomocou `\getmap`. Zobrazená je náhodne vybraná adresa v centre mesta Žilina

Príkaz `\getmap`

Ako sme už spomenuli, balíček `getmap` obsahuje de facto jediný príkaz, ktorého plná syntax má tvar:

`\getmap[voľby]{adresa}`

V najjednoduchšom prípade stačí poznať len adresu, POI alebo geografické GPS súradnice zvoleného miesta. Položka **adresa** musí byť plne rozvinutá a nesmie obsahovať žiadne makrá. Získaný obrázok sa implicitne ukladá do súboru s názvom `getmap.png`, ktorý sa je uložený v aktuálnom pracovnom adresári. V prípade, že do nášho dokumentu vkladáme len jednu mapu je tento mechanizmus plne postačujúci.

Príslušný zdrojový kód by teda mohol vyzeráť napríklad takto:

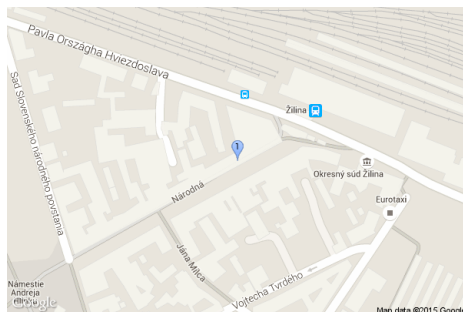
```
\getmap{Národná 25, 01001 Žilina, Slovakia}
\includegraphics[width=.5\linewidth]{getmap}
```

s výsledkom zobrazeným na obrázku 1.

Pri opätovnej kompilácii súboru je potrebné si uvedomiť, že nedochádza ku prepisovaniu získaného obrázku `getmap.png`. To síce urýchľuje kompiláciu, na druhú stranu, ak dôjde ku zmene v zobrazovanej adrese, to má za následok, že obrázok v dokumente sa vlastne nezmení. To je možné riešiť buď odstránením súboru `getmap.png` z pracovného adresára pred novou kompiláciou, alebo vhodnou voľbou pre prepisovanie súborov, o ktorej budeme hovoriť neskôr.

Voľby príkazu `\getmap`

Činnosť príkazu `\getmap` podstatným spôsobom ovplyvňujú voliteľné argumenty. Dokonca je možné povedať, že sú dôležitejšie než samotný príkaz. My si predsta-



Obrázok 2: Ukážka výstupu na obrázku 1 získaného z Google Maps pomocou voľby `mode=gm`. Zobrazená je rovnaká adresa v centre mesta Žilina

víme iba tie najdôležitejšie, resp. najpoužívannejšie z nich, úplný zoznam je možné nájsť v manuáli [3].

Voľba `mode`

Voliteľný argument `mode` môže nadobúdať jednu z troch hodnôt `osm|gm|gsv` a určuje, ktorý mapový zdroj bude použitý. Z hodnoty argumentu je intuitívne zrejmé, ktorý zodpovedá OpenStreetMap, Google Maps alebo Google Street View. Ako implicitný zdroj je preddefinovaný OpenStreetMap. Ak by sme teda chceli výsledok na obrázku 1 získať pomocou Google Maps, je potrebné príkaz upraviť takto:

```
\getmap[mode=gm]{Národná 25, 01001 Žilina, Slovakia}
```

Výsledok si môžeme pozrieť na obrázku 2 .

Voľby `file` a `imagetype`

Voľba `file=subor` umožňuje pomenovať výstupný súbor, ktorý bude vytvorený príkazom `\getmap`. Názov súboru sa uvádza bez prípony. Na stanovenie grafického formátu získaného výstupu sa potom používa voľba `imagetype`, ktorá môže nadobúdať niektorú z hodnôt `png|jpeg|jpg|gif`, pričom ako implicitný typ je súbor vo formáte `png`. Táto voľba už je ale viazaná na použitý mód, pričom uvedené hodnoty sú dostupné pre `mode=osm`. Pri použití módu `gm` sú dostupné ešte ďalšie grafické formáty, ako napríklad `png8|png32`.

Ak si teda uvedomíme, že pri spracovaní obrázku 1 bol použitý implicitný názov súboru `getmap`, je zrejmé, že súbor vložený do obrázku 2 je potrebné aj premenovať. Úplný zdrojový kód pre výstup na obrázku 2 teda vyzerá takto:

```
\getmap[mode=gm,file=myobr]{Národná 25, 01001 Žilina, Slovakia}
```



Obrázok 3: Ukážka výstupu rovnakej oblasti ako na obrázku 1, získaného z Google Maps pomocou voľby `type=satellite`. Zobrazená je rovnaká adresa v centre mesta Žilina

Voľba `type`

Voľba `type` môže v móde `osm` nadobúdať niektorú z troch hodnôt `map|sat|hyb` pre zobrazenie mapy, satelitnej snímky alebo hybridné zobrazenie mapy a satelitnej snímky. Pri nastavení módu na hodnotu `gm` sa možné hodnoty voliteľného argumentu `type` menia na `roadmap|satellite|hybrid|terrain`.

Na obrázku 3 ilustrujeme satelitný záber rovnakého výrezu centra mesta ako na mapách na obrázkoch 1 a 2.

Voľba `overwrite`

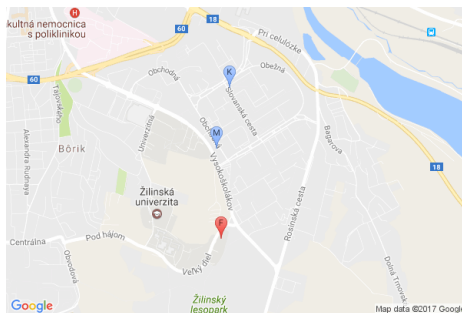
Ide o dôležitý voliteľný argument, ktorý určuje, či sa pri jednotlivých kompiláciách budú získané obrázky prepisovať alebo nie. Nastavuje sa na logické hodnoty `true|false`. Tento argument je užitočný, ak meníme definovaný výrez z mapy, aby sme nemuseli manuálne odstraňovať obrázky z predchádzajúcich kompilácií. Naopak, jeho nastavenie na hodnotu `false` urýchľuje preklad, ak máme obrázkov veľa a sú už z predchádzajúcich behov `LATEX`u vytvorené.

Voľba `xsize` a `ysize`

Tieto voliteľné hodnoty určujú rozmery získaného obrázku v pixeloch, pričom hodnota `xsize` určuje jeho šírku a hodnota `ysize` určuje jeho výšku. Implicitné rozmery sú nastavené na hodnoty 600×400 , pričom hodnotu je možné meniť. V režime `osm` sú pre oba rozmery horné hranice 3 840, kým slobodná verzia Google maps je obmedzená na rozmery 640×640 .

Vyznačovanie objektov na mapách

Na vyznačenie významných objektov či orientačných bodov na mape slúžia tzv. markery. Tieto sa definujú pomocou voliteľného argumentu `markers` a ako hodnota



Obrázok 4: Ukážka mapy okolia FRI ŽU (marker F) s vyznačením stravovacích zariadení „Miláno“ (marker M) a „Kazačok“ (marker K)

je mu priradený zoznam všetkých značiek, ktoré majú byť vyznačené na mape. Zoznam markerov sa uzatvára do zložených zátvoriek, pričom každý z nich je definovaný sekvenciou:

```
&markers=size:mid|color:blue|label:S|loc1|loc2|...|locn
```

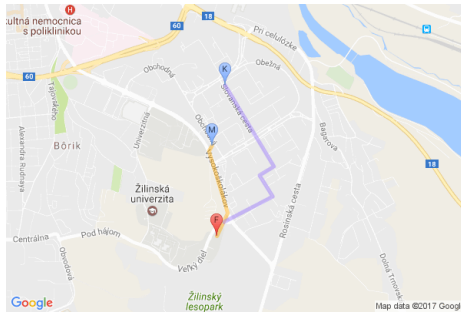
Jednotlivé položky potom definujú vlastnosti markerov. Hodnota parametra **size** môže nadobúdať hodnoty **tiny**, **mid**, **small**, pričom ako implicitná hodnota sa berie **mid**. Parameter **color** definuje farbu markera, ktorá môže byť zadaná buď pomenovaním alebo hexadecimálnym kódom. Parameter **label** určuje značku vo vnútri markera (len pri veľkosti **mid**). Prípustné sú numerické znaky alebo kapitálky. Hodnoty **loc** potom definujú lokalizáciu markera na mape.

Ukážku mapy s vyznačenými objektmi vidíme na obrázku 4, kde je zobrazené okolie Fakulty riadenia a informatiky a v jej blízkosti dve reštaurácie. Príslušný kód príkazu `\getmap` vyzerá takto:

```
\getmap[file=fricka,overwrite=true,mode=gm,
  markers={&markers=size:mid|label:F|color:red|
    Fakulta riadenia a informatiky, 01008 Žilina, Slovakia%
    &markers=size:mid|label:M|color:blue|
    Obchodná 3269, 01008 Žilina, Slovakia,%
    &markers=size:mid|label:K|color:blue|
    Slovanská 3278, 01008 Žilina, Slovakia},%
  visible={{49.16, 18.65}|{49.21, 18.45}}
]{}

```

V zdrojovom kóde si môžeme všimnúť tiež argument **visible**. Jeho úlohou je zabezpečenie zobrazenia takého výrezu mapy, aby všetky lokality uvedené v zozname a oddelené zvislou čiarou boli na obrázku viditeľné. Je ich možné zadať ako objekty alebo pomocou GPS súradníc.



Obrázok 5: Ukážka mapy okolia FRI ŽU (marker F) s vyznačením stravovacích zariadení „Miláno“ (marker M) a „Kazačok“ (marker K) a trasy ku nim

Vyznačenie trasy

Často je potrebné v mape vyznačiť aj cestu, ako je možné prejsť od jedného objektu ku druhému, prípadne viacero trás. Na tieto účely má príkaz `\getmap` voliteľný parameter `path`, ktorý podobne ako `markers` očakáva zoznam URL parametrov oddelených zvislou čiarou v tvare:

`&path=weight:5|color:orange|loc1|loc2|...|locn`

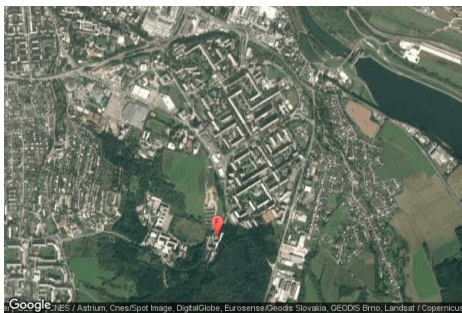
Hodnota `weight` udáva hrúbku čiary vyznačujúcej trasu, `color` farbu tejto čiary a ostatné hodnoty predstavujú lokalizačné údaje. Lokalizačné údaje je pri tom možné zadávať buď ako adresy alebo dvojicu GPS súradníc. Vyznačenie dvoch trás je ilustrované na obrázku 5.

Príslušný zdrojový kód upravíme takto:

```
\getmap[file=fricka3,overwrite=true,mode=gm,
  markers={&markers=size:mid|label:F|color:red|
    Fakulta riadenia a informatiky, 01008 Žilina, Slovakia%
    &markers=size:mid|label:M|color:blue|
    Obchodná 3269, 01008 Žilina, Slovakia,%
    &markers=size:mid|label:K|color:blue|
    Slovanská 3278, 01008 Žilina, Slovakia},%
  path={&path=weight:5|color:orange|Fakulta riadenia a informatiky,
    01008 Žilina, Slovakia|49.203912,18.763293|
    49.208426,18.761190|Obchodná 3269, 01008 Žilina, Slovakia%
    &path=weight:5|color:purple|Slovanská 3278, 01008 Žilina,
    Slovakia|49.207304,18.768442|49.206660,18.766833|
    49.205117,18.768314|49.203267,18.762885}}{}
```



Obrázok 6: Snímka FRI ŽU zo služby Google Street View



Obrázok 7: Satelitný záber okolia FRI ŽU

Obrázky Google Street View

Balíček `getmap` umožňuje vkladať do dokumentu aj fotografie, získané prostredníctvom služby Google Street View. Na tieto účely je potrebné pomocou voliteľných argumentov nastaviť službu `mode=gsv`. Pre špecifikáciu záberu sú dôležité voľby `heading`, ktorá udáva smer záberu zo stanoveného miesta v rozpätí 0–360 (pre sever použijeme hodnotu 0, pre východ 90 atď.), `pitch`, ktorá definuje uhol sklonu kamery od vodorovnej roviny a udáva sa v rozpätí od –90 po 90, a voľba `fov` udávajúca veľkosť rozsahu horizontálneho záberu v stupňoch, a to v rozsahu 0–120. Všetky uvedené parametre sú merané v stupňoch.

Ilustrujeme si to na obrázku fakulty. Príslušnú snímku získame napríklad takýmto príkazom:

```
\getmap[file=pfricka,mode=gsv,heading=120,fov=120,pitch=20,
  xsize=600,ysize=450,scale=2,zoom=20,overwrite=true]
{49.202217,18.761407}
```

Výsledok potom vidíme na obrázku 6. Na susednom obrázku 7 vidíme satelitný záber okolia fakulty s vyznačeným bodom snímania.

Script `getmapdl`

Ako sme sa už zmienili v úvode tohto príspevku, balíček `getmap` vlastne využíva jeden Lua skript, ktorý môže byť použitý aj z príkazového riadku. Úplnú nápovedu ku skriptu získame z príkazového riadku obvyklým postupom, teda:

```
$ getmapdl -h
```

Z dostupných prepínačov si uvedieme len niekoľko najdôležitejších:

- m určuje mód, obvykle `osm|gm|gsv`,
- l určuje lokalitu,
- i určuje formát grafického súboru,
- o určuje pomenovanie výstupného súboru (bez prípony).

Ak to zhrnieme, potom pomocou príkazu

```
getmapdl -m gm -l 'Národná 25,01001 Žilina, Slovakia' -i jpg -o zilina
```

získame rovnaký obrázok, ako je na obrázku 2 a výsledok bude uložený v súbore `zilina.jpg`.

Záver

V príspevku sme predstavili balíček `getmap`. Na ukážkach bolo demonštrované, ako je do textu možné zaradiť nielen mapy získané zo služieb Open Street Map alebo Google Maps, ale aj fotografie objektov získané pomocou Google Street View.

Poďakovanie

Tento príspevok vznikol s láskavým príspevom grantu KEGA-011ŽU-4/2014 „Experimentálna matematika – zviditeľnenie neviditeľného“ podporeného Slovenskou kultúrno-edukačnou grantovou agentúrou.

Reference

- [1] BLAŠKO, R.: *L^AT_EX nie je farba na maľovanie*, Otvorený softvér vo vzdelávaní, výskume a IT riešeníach, zborník medzinárodnej konferencie OSSConf 2010, Žilina, 1.–4. júla 2010, str. 43–52, ISBN 978-80-970457-0-8, <http://ossconf.soit.sk/images/zborniky/zbornik2010.pdf>.
- [2] BLAŠKO, R.: *L^AT_EX nie je farba na maľovanie, ale na písanie*, Otvorený softvér vo vzdelávaní, výskume a IT riešeníach, zborník medzinárodnej konferencie OSSConf 2011, Žilina, 1.–4. júla 2011, str. 249–258, ISBN 978-80-970457-1-5, <http://ossconf.soit.sk/images/zborniky/zbornik2012.pdf>.
- [3] KLEBER, J.: *Downloading maps from OpenStreetMap, Google Maps or Google Street View*.
- [4] KOPKA, H. – DALY, P. W.: *L^AT_EX – Podrobný príručiek*, Brno, Computer Press, 2004, ISBN 80-722-6973-9.

- [5] KOZUBÍK, A.: *Prezentačné materiály v triede Beamer*, Zborník príspevkov z medzinárodnej konferencie OSSConf 2011, Žilina, 1.–4. júla 2011, str. 223–235, ISBN 978-80-970457-1-5.
- [6] RYBIČKA, J.: *L^AT_EX pro začátečníky*, Brno, KONVOJ 2003, ISBN 80-7302-049-1.
- [7] RYBIČKA, J., ČAČKOVÁ, P., PŘICHYSTAL J.: *Průvodce tvorbou dokumentů*, Bučovice, Nakladatelství Martin Stříž 2011, ISBN 978-80-87106-43-3.
- [8] STŘÍŽ, P.: *Sazba v T_EXu a kresba v METAPOSTu*, Bučovice, Nakladatelství Martin Stříž 2011, ISBN 978-80-87106-51-8.

Summary: Maps in L^AT_EX Documents – an Introduction of the getmap Package

The aim of this article is to introduce the `getmap` package. This package allows to include into the L^AT_EX documents the map materials obtained from the external resources such as OpenStreetMap and Google Maps and even with the support of Google Street View. In the simplest case, the specification of an address is sufficient. The package loads the map using the `\write18` feature, which must be activated to use this package. The image will be downloaded by an external Lua script that can be used also from the command line.

Keywords: Maps, L^AT_EX, `getmap`, Lua

*Katedra matematických metód a operačnej analýzy,
Fakulta riadenia a informatiky, Žilinská univerzita,
Univerzitná 8215/1, 010 26 Žilina, Slovenská republika*

Od soboty 29. 4. do středy 3. 5. 2017 se u polského jezera Bachotek konala společná konference sdružení GUST a TUG.

Sobota 29. 4.

První konferenční den otevřel Hans Hagen příspěvkem *Children of TeX*, ve kterém se v souladu s konferenčním tématem „Premises, predilections, predictions“ zabíral minulostí, současností a budoucností TeXu. Následující příspěvek *Revealing semantics using subtle typography and punctuation* od Kumarana Sathasivama se věnoval způsobům, jakými je možné obohatit interpunkci a přispět tak ke zjednoznačení psaného projevu.

Po přestávce na kávu navázal Frank Mittelbach příspěvkem *Through The Looking Glass – and what Alice found there . . .*, ve kterém představil algoritmus pro optimalizaci stránkového zlomu a umísťování plovoucího obsahu; s tímto příspěvkem obdržel cenu ACM Best Paper Award na konferenci DocEng 2016. V následujícím příspěvku *To justify or not to justify?* Leila Akhmadeeva a Boris Veytsman provedli experimentální srovnání rychlosti čtení textu sázeného do bloku bez dělení slov a textu sázeného na praporek s dělením slov. Dopolední blok uzavřel Przemysław Scherwentke s příspěvkem *LaTeX restaurant*, ve kterém zrecenzoval knihu LaTeX, książka kucharska.

Po sobotním obědě následovalo LaTeXové okénko, ve kterém se nejprve Barbara Beeton v příspěvku *Debugging LaTeX files – Illegitimi non carborundum* podělila o rady, jak systematicky řešit chyby při překladu dokumentů, a následně Boris Veytsman v příspěvku *Making ltxsparklines package: A journey of a CTAN contributor into the world of CRAN* představil knihovnu v jazyce R, která umožňuje generovat vstup pro LaTeXový balíček `sparklines` na tvorbu Tufteho sparkline grafů.

Po přestávce na kávu následoval příspěvek Grzegorza Murzynowského *The GM-Scenarios two years later. A complete madness. But – Turing-complete or not? Or: how the spirit of l3expan made me conceive and bear a monster* o změnách v LaTeXovém balíčku `GM-Scenarios`, který nad jazykem expl3 buduje framework pro řízenou expanzi argumentů maker. Na závěr odpoledního bloku referovali Jean-Michel Hufflein v příspěvku *MLBibTeX Now Deals with Unicode* o vývoji bibliografického preprocesoru `MLBibTeX` a autor tohoto článku v příspěvku *Typesetting Bibliographies Compliant with the International Standard*



Obrázek 1: Skupinová fotografie účastníků konference

ISO-690 in L^AT_EX o balíčku `biblatex-iso690` od Michala Hofticha a Dávida Luptáka pro sazbu citací a referencí podle normy ISO 690:2010. Na závěr prvního konferenčního dne se konal táborový oheň.

Neděle 30. 4.

Druhý konferenční den byl věnován workshopům. Po snídani se konal úvodní kurz jazyka ConT_EXt pod názvem *ConT_EXt: tutorial/workshop (for ConT_EXt beginners)*, jenž vedli Willi Egger a Mojca Miklavec. Po obědě byla pořízena skupinová fotografie. Následoval pracovní seminář *Bookbinding workshop: portfolio* od Williho Eggera na výrobu papírových desek s klopami a pracovní seminář *Hackaton: documenting L^AT_EX packages* od Damiena Thirieta, během kterého byla doplněna dokumentace oblíbených L^AT_EXových balíčků.

Po přestávce na kávu přednesli Maciej Rychły a Piotr Bolek příspěvek *Released sounds* o hudbě zachycené formou notového zápisu v historických malbách; na příspěvek navázali Maciej a Mateusz Rychły hudebním představením. Po večeri následovalo výroční zasedání sdružení TUG a GUST. Novým prezidentem sdružení TUG byl zvolen jediný kandidát, Boris Veytsman. Členy správní rady TUGu byli do roku 2021 zvoleni Karl Berry, Johannes Braams, Kaja Christiansen, Taco Hoekwater, Klaus Höppner, Frank Mittelbach, Ross Moore, Arthur Reutenauer, Will Robertson a Herbert Voß. Po zasedání se omezený počet zájemců se mohl zúčastnit degustace piv z malých pivovarů vedené Michałem Gasewiczem v rámci semináře *Off topic (completely): Many faces (and types) of beer*.



Obrázek 2: Úvodní kurz formátu ConT_EXt od Williho Eggera



Obrázek 3: Pracovní seminář na výrobu papírových desek od Williho Eggera



Obrázek 4: Demostrace kombinovatelnosti znaků emoji na rodinných příslušnících Hanse Hagara

Pondělí 1. 5.

Třetí konferenční den se točil především kolem písma. Po snídani referoval Ulrik Vieth o posledních deseti letech vývoje matematických OpenType fontů a Jerzy Ludwowski o aktuálních projektech písmolijny GUST's e-foundry; na základě výsledků poslední výborové schůze ζ TUGu bude sdružení ζ TUG písmolijně v letech 2017–2019 přispívat roční částkou 1000 EUR. V následujícím příspěvku *Xdvipsk: dvips ready for OpenType fonts and more image types* představil Sigitas Tolušis *xdvipsk* – verzi programu *dvips* s podporou OpenType fontů a bitmapových obrázků.

Po přestávce na kávu představil v příspěvku *Variable and color OpenType fonts: chances and challenges* Adam Twardoch mechanismy formátu OpenType, kterými je možné vytvářet vícebarevné a multiple-master fonty (tzv. variable fonts); CSS rozhraní variable fonts demonstruje webová stránka axis-praxis.org. V následujících dvou příspěvcích předělených obědem pod názvem *Colorful fonts, an update and peek into the future* a *Variable fonts* se Hans Hagen věnoval znakům emoji, mechanismu variable fonts a jejich implementaci v systému ConTeXt.



Obrázek 5: Doprovod ke konferenční večeři zajistila skupina Katarzyny Jackowské

Na začátku odpoledního bloku prezentoval Bogusław Jackowski v příspěvku *Parametric math symbol font* postup, kterým lze z textových fontů odvozovat fonty matematické. Po přestávce na kávu následoval příspěvek Adama Twardoch *STIX, Fira, Noto and friends: beautiful new opensource fonts* o existujících svobodných fontech. V závěrečných dvou příspěvcích nazvaných *One rule to break them all* a *Automating binary building for T_EX Live* referovala Mojca Miklavec o možném budoucím vývoji vzorů dělení slov a o automatické kompilaci programů pro T_EX Live. Den byl uzavřen konferenční večeří, v rámci které se slavilo 25. výročí konferencí sdružení GUST. Hudební doprovod k jídlu i k tanci zajistila skupina Katarzyny Jackowské.

Úterý 2. 5.

Čtvrtý konferenční den otevřel Siep Kroonenberg příspěvkem *The T_EX Live Launcher*, ve kterém představil spouštěč komponent síťové instalace T_EX Live pro operační systémy MS Windows. V následujícím příspěvku *Fmtutil and updmap – past and future changes (or: cleaning up the mess)* popsal Norbert Preining změny v rozhraní nástrojů *fmtutil* a *updmap* v distribuci T_EX Live 2017. Ranní

blok uzavřel Luigi Scarso příspěvkem *MFLua 0.8*, ve kterém referoval o vývoji programu MFLua; ten nyní umožňuje spouštěním kódu v jazyce Lua z MetaFontu přímo generovat OpenType fonty.

Po přestávce na kávu následoval příspěvek Takuto Asakury *Implementing bioinformatics algorithms in T_EX-Gotoh package, a case study* o L^AT_EXovém balíčku Gotoh pro výpočet podobnosti DNA sekvencí pomocí Gotohova algoritmu a pro vizualizaci tohoto výpočtu. V následujícím příspěvku *T_EX users habits versus publishers requirements* Lolita Tolené analyzovala trendy použití L^AT_EXových balíčků v dokumentech zpracovávaných společností VTeX a představila problémy pojící se s převodem L^AT_EXových dokumentů do formátu XML. Dopolední blok uzavřeli Petr Sojka příspěvkem *T_EX in Schools? Just Say Yes: the Use Case of T_EX Usage at the Faculty of Informatics, Masaryk University* shrnujícím historii využití T_EXu na Fakultě informatiky Masarykovy univerzity v Brně a autor tohoto článku příspěvkem *Using Markdown Inside T_EX Documents* představujícím balíček `markdown.tex` pro přípravu dokumentů pomocí značkovacího jazyka Markdown.

Po obědě referoval Jean-Michael Hufflen v příspěvku *History of accidentals in music* o historii posuvek používaných v notovém zápisu a Andrzej Tomaszewski popsal v příspěvku *An example of a humanist scholarly book* proces grafického návrhu knihy s překladem Ovidiovy básně *Halieutica* o rybách Černého moře.

Po přestávce na kávu proběhla diskuze o způsobech konverze dokumentů v jazycích Markdown a CSS do tisknutelného výstupu ve formátu PDF v rámci příspěvku *CORDIDA! Collaborative Opensource Rapid Digital Internet Documentation Authoring* Adama Twardocha. Následovala diskuze v rámci příspěvku *T_EX Annoyances – what is on the way to a full production environment* Paula de Ney Souza o současných problémech T_EXu, jako je pomalý běh kompilátoru LuaT_EX, nedostatečná podpora mikrotypografických rozšíření v kompilátoru X_YT_EX a neuspokojivé výsledky L^AT_EXového balíčku `hyperref` při sazbě hypertextových odkazů v místě stránkového zlomu. Na závěr konferenčního dne Paulo de Ney Souza referoval v příspěvku *T_EX Production – ePub the new target* o výstupním formátu ePub a Zunbeltz Izaola představil v příspěvku *DocVar: manage document variables* L^AT_EXový balíček DocVar pro správu metadat spojených s dokumentem.

Středa 3. 5.

Pátý konferenční den otevřel Jerzy Ludwikowski příspěvkem *T_EX at secondary schools – an idea to be taken up by GUST*. V něm představil návrh Anny Kwiatkowské, podle kterého by sdružení GUST mělo připravit studijní materiály pro výuku T_EXu na střední škole Liceum i Gimnazium Akademickie v Toruni. Následně představil Marcin Woliński L^AT_EXový balíček `bredzenie.sty` pro sazbu polského výplňového textu, který je generován pomocí jazykových modelů. Konferenci uzavřel Marcin Borkowski příspěvkem *What can a T_EXnician learn from ten*

years' editorial work o využití textového editoru Emacs při redigování časopisu Wiadomości Matematyczne.

Fotografie do článku laskavě poskytl Frans Goddijn.

Summary: Conference TUG@BachTeX 2017

The article is a summary report of the TUG@BachTeX 2017 conference, which was held from April 29 to May 3 jointly by GUST and TUG at Bachotek near Brodnica, Poland.

Vít Novotný, witiko@mail.muni.cz

Zpravodaj Československého sdružení uživatelů T_EXu
ISSN 1211-6661 (tištěná verze), ISSN 1213-8185 (online verze)

Vydalo: Československé sdružení uživatelů T_EXu vlastním
nákladem jako interní publikaci
Obálka: Antonín Strejc
Ilustrace na obálce: Hans Hagen
Počet výtisků: 310
Uzávěrka: 31. 7. 2017
Odpovědný redaktor: Jan Šustek
Redakční rada: Pavel Haluza, Lukáš Novotný, Vít Novotný,
Michal Růžička a Jan Šustek (šéfredaktor)
Technická redakce: Vít Novotný
Evidenční číslo MK: E 7629
Tisk: ASMETI, Klášterní 1187, 735 11 Orlová
Adresa: ČS_{TUG}, Nejedlého 373/1, 638 00 Brno
Email: cstug@cstug.cz

Zřízené poštovní aliasy sdružení ČS_{TUG}:

bulletin@cstug.cz, zpravodaj@cstug.cz
korespondence ohledně Zpravodaje sdružení

board@cstug.cz
korespondence členům výboru

cstug@cstug.cz, president@cstug.cz
korespondence předsedovi sdružení

gacstug@cstug.cz
grantová agentura ČS_{TUGu}

secretary@cstug.cz, orders@cstug.cz
korespondence administrativní síle sdružení, objednávky CD a DVD

cstug-members@cstug.cz
korespondence členům sdružení

cstug-faq@cstug.cz
řešení otázky s odpověďmi navrhované k zařazení do dokumentu ČS_{FAQ}

bookorders@cstug.cz
objednávky tištěné T_EXové literatury na dobírku

ftp server sdružení:
ftp://ftp.cstug.cz

www server sdružení:
http://www.cstug.cz

CONTENTS

Petr Sojka: Introduction	1
Hans Hagen: CONTEX-T-LUA Documents	3
Hans Hagen: Exporting XML and ePub from CONTEX-T	55
Hans Hagen, Idris Samawi Hamid: Oriental TEX: Optimizing Paragraphs	64
Taco Hoekwater: MetaPost: PNG Output	98
Aleš Kozubík: Maps in L ^A T _E X Documents – an Introduction of the getmap Package	101
Vít Novotný: Conference TUG@BachoTEX 2017	110